2002－10（第18期）

Tom DeMarco

X-Programmer 非程序员
软 件 以 用 为 本

主编：davidqql
审稿：davidqql，think

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

http://www.umlchina.com/

驳 UML 三大“硬伤”论

张恂，think 著

查看评论

摘要

本文对 2002 年 5 月《程序员》杂志刊登的《UML 三大“硬伤”》（指 UML“上不着天、下不着地、一盘散沙”）一文所存在的 18 个错误逐一进行了批驳，指出原文在论据、论证、论点和文风上存在的严重问题，并结合 RUP（瑞理统一过程）给出 UML 建模实例，演示了如何对原文所提及的商业公司进销存系统进行正确的业务分析。本文适合的读者包括 UML 和面向对象技术的爱好者、初学者，软件开发人员、软件企业与行业企业的技术主管等。

关键词

OOAD 方法论、RUP、UML、面向对象业务建模、业务过程

1 引言

世界上最大的软件工业组织对象管理集团（Object Management Group）颁布的统一建模语言（UML, Unified Modeling Language）是迄今为止最好的面向对象通用可视化建模语言。它取代了以往各种面向对象表示法，可以全面、细致地同时描述业务和软件系统，实现软件开发全生命周期建模的无缝统一。UML 不仅能从宏观和微观上精确、完整、一致地描述企业的组织结构、业务过程和业务信息，更好地消除信息不对称，促进软件开发人员与用户、领域专家的沟通以捕获真正的业务和系统需求，而且还可以有效地帮助、指导开发人员编写程序和文档，从而有利于提高软件整体质量和生产率。

2002 年第 5 期《程序员》杂志刊登了一篇题为《UML 三大“硬伤”》的专栏文章[高 102]，在互联网上曾经一度引起激烈的反响。我们发现该文严重失实，具有很大的迷惑性，势必给不少初学者、不了解内情的软件企业技术主管以及软件客户带来针对 UML 和 OOAD（面向对象分析与设计）方法的困惑与误解，造成他们对当前软件技术发展主流和走势的判断失误。因此，作为 UML 和面向对象技术的实践者，我们感到有责任撰写文章来予以澄清。

原文的主要论点为 UML 有三大硬伤——“**上不着天**（无法与用户/领域专家沟通获得真正的需求），**下不着地**（无法提供直接到位的素材指导程序员编程），**一盘散沙**（没有在细微之处建立建模图形之间的联系……建模图形之间的关系凌乱不堪）……说明了 UML 在建模内容中并未实现 Unified 的原旨”

这些结论把矛头直接指向了 UML 被公认的几大优点，如完善的统一性、良好的沟通性和强大的表达能力，对于我们这些稍懂些 UML 的人来说，乍看之下，的确有些耸人听闻，但其实它们都是经不起推敲，缺乏依据，根本不成立的姑妄之辞，原文在论据的真实性、可靠性和论证的逻辑严密性、正确性上也都存在着严重的缺陷。

我们先从技术角度对原文批驳如下。

2 基本错误

首先，原文通篇混淆了这样几个概念：UML 能力、UML 标准（规范）、UML 方法、UML 工具和 UML 实践。图 2-1 描述了它们之间的区别和联系。

UML 是一种语义丰富、通用、可视化的面向对象建模语言和事实上的国际工业标准，最新版本为 1.4，UML 2.0 正在研制过程中。UML 可用于对软件和业务系统进行可视化、描述、构造以及文档化。它统一了以往各种面向对象表示法，是面向对象技术的集大成者。1997 年以来 OMG 陆续制定和完善的 UML 规范对 UML 的语法、语义、表示法等作了明确的规定，该标准文本是研究、探讨 UML 现状与发展的一个重要基础材料[UMLS01]。

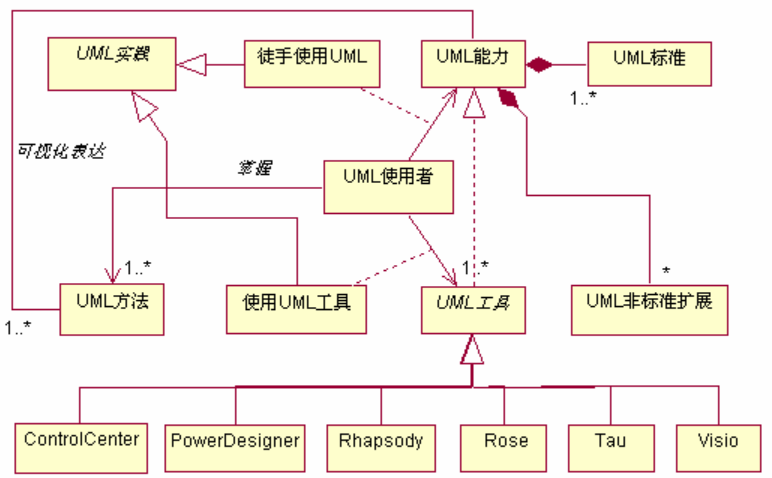


图 2-1　UML 能力与 UML 标准、工具和实践的区别与联系

UML 语言适用于各种软件开发方法、软件生命周期的各个阶段、各种应用领域和各种开发工具[U 参 01]。UML 规范实质上仅仅阐明了建模的基本元素、表示法、相关语义和扩展机制，好比只提供了构筑大厦的砖头和水泥，因此在 UML 的应用过程中选用正确的开发方法论（methodology）是成功建模的关键。

只有在对基本概念正确理解的基础上采用了正确的方法和步骤才能建立正确的 UML 模型。这里我们把采用 UML 的面向对象开发方法，通常包括 OOAD 方法论和开发过程模型等，统称为“UML 方法”。例如，RUP（Rational Unified Process，瑞理统一过程）就是一种基于 UML 方法的开发过程向导和框架。

目前国内外流行的 UML 建模工具有很多种，例如 iLogix Rhapsody、Microsoft Visio、Rational Rose/Rose RT/XDE、Sybase PowerDesigner、TogetherSoft ControlCenter、 Telelogic Tau UML Suite 以及一些免费工具等等。

各个工具开发商根据自己的定位、能力和市场的需要使得每一种 UML 工具都不同程度地分别实现了 UML 标准的不同子集，有些厂商、咨询公司和专家还发明、提供了非标准的 UML 扩展集（profile）。因此，市面上的 UML 工具尽管基本上都能提供 UML 规范所定义的主要功能，但不同产品甚至同一产品的不同版本在具体的功能实现上总存在一些差异，表现出各自的优缺点。然而，某些工具的不足显然不能与 UML 本身的缺点划上等号，**评价 UML 与评价 UML 工具俨然是两码事**。

由此看来，UML 能力——用 UML 究竟能够表达、描述哪些内容——应该是 UML 基本规范及其各种标准、非标准扩展集（如数据建模、业务建模、实时建模、分布式对象建模、Web 建模、XML 建模等等）的语义能力的总和。

UML 实践是指 UML 的使用者（建模者）在掌握 OOAD 方法论的基础上运用 UML 工具（或徒手）建立模型的活动。它的成功或失败与使用者对 UML 的能力、OOAD 技术和工具的熟悉、理解程度，UML 工具的差别乃至团队的项目和过程管理水平等等都有直接关系。影响 UML 实践结果好坏的因素是多方面的（见图 2-1），不能把 UML 建模的失败简单地一概归结为 UML 本身的问题和缺陷，必须认真地排查找出真正的原因。

其次，原文没有明确指出全程建模方法与 UML 在概念上的区别，拿两者直接作比较并不恰当。如果不作特别说明，提到 UML，一般认为就是指统一建模语言本身而不是 UML 方法。所以，既然文章题目叫 UML 的硬伤，那么就应该针对 UML 语言的真实能力来进行论证，不能把建模语言、建模方法和建模过程混为一谈。全程建模方法采用了组成结构树、顺序图、PAD 图、数据汇总图等表示法，作者也许忘了为自己的建模语言取名，在此我们姑且称之为“全程建模语言”。正确的比较关系应该是——UML 对全程建模语言，RUP（UML 方法）对全程建模方法。

任何模型都只是建模者认识世界的一种思维反映。原文作者通篇不加区分地把属于 UML 实践和工具的问题，个人在使用 UML 建模过程中的某些经历、体验以及错误的做法，乃至 UML 范畴之外（比如需求和项目管理）的问题，都统统指责、夸大为 UML 的硬伤，犯了偷换概念、逻辑颠倒的毛病，客观上很容易造成误导。（参见错误 1、2、4、6、7、8、9、10、11、13、14、15、16、17）

本文第 3 到第 5 节运用了 RUP 的术语和方法对原文中给出的一个商业公司进销存业务案例进行建模，在指出原文错误的同时给出正确的 UML 画法以证明 UML 语言的有效性。

3 UML“上不着天”？

原文认为 UML“上不着天”的根源在于：（1）“难以完整全面地描述企业的分工结构”（2）“难以从宏观把控业务流程的完整与准确”（3）“无法从微观把控业务信息的操作过程”（4）“无法彻底全面描述用户的需求”（5）UML“是造成信息不对称的‘功臣’”，因此 UML“与用户/领域专家无法沟通获得真正的需求”。果真如此吗？

业务建模（business modeling）的目标是通过建立组织的业务模型，促进客户、最终用户和开发人员对目标组织的内部结构、动态行为（业务过程）以及存在的问题取得一致理解，找到有效的解决措施，并生成为支持目标组织业务的实现和改进而服务的（软件）系统需求。

统一了面向对象软件分析、设计和实现建模的 UML 到底能否应用于业务建模，能否在最大程度上保证业务模型的准确性、一致性和完整性？回答是肯定的。这主要归功于 UML 规范在严格定义、充分抽象、灵活使用的基础上，通过其扩展机制——构造型（stereotype）、标记值（tagged value）、约束（constraint）以及对象约束语言（OCL，Object Constraint Language）具备了强大的一般建模语言所不具备的语法和语义扩展能力。

当前 UML 业务建模领域有两个重要成就。早在 1994 年，OO 大师 Ivar Jacobson 博士（现为 Rational 负责过程战略的副总裁）就在其名著《The Object Advantage: Business Process Reengineering with Object Technology》开创性地提出了运用面向对象技术（如业务用例和业务对象）进行业务过程建模的概念和方法。后来基于这一思想的延伸与发展逐渐反映到如今的 UML 规范和商用过程产品 RUP 之中，目前业务建模扩展集已经成为 RUP 的一个主要模块[UMLS01][RUP02]（见图 3-1）。

在 RUP 中，企业的业务过程用业务用例（business use-case）表示，软件系统的功能需求用一般的 UML 用例表示，在此基础上 RUP 很好地解决了从业务建模到系统建模的过渡和衔接问题（见图 3-2），充分体现了 UML 统一性的突出优点。

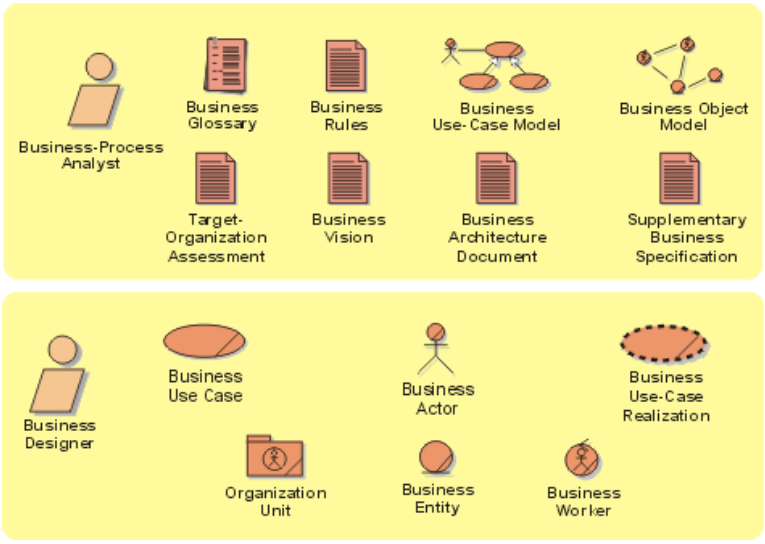


图 3-1、RUP 业务建模工件集

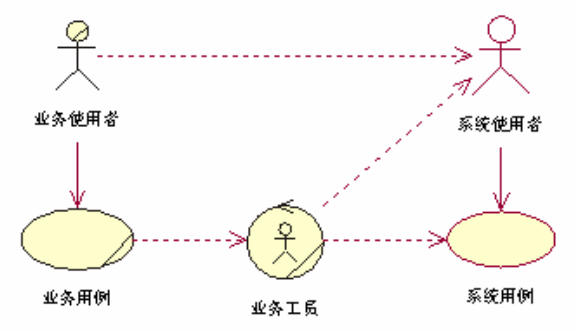


图 3-2、从业务到系统

UML 业务建模的另一个分支是由 Hans-Erik Eriksson 与 Magnus Penker 共同在 1999 年发明的 UML-EPBE（Ericksson-Penker Business Extensions for UML）。他们在《Business Modeling with UML: Business Patterns at work》这本书中创造性地提出了用于 UML 业务建模的一个丰富的基本框架，介绍了如何用 UML 结合 4 种业务视图与 26 个业务模式（business pattern）来描述商业组织业务模型的各个方面，包括业务架构（business architecture）、业务过程、业务资源、业务目标（goal）和业务规则（rule）等等。[EPBE99]

EPBE 从业务构想视图（Business Vision View）、业务过程视图（Business Process View）、业务结构视图（Business Structural View）和业务行为视图（Business Behavioral View）4 个方面来描述业务，用 UML 的 OCL 描述业务规则，从而可以建立一个完整的业务模型。

EPBE 对 UML 的活动（Activity）类元进行了扩展以表示业务过程（如图 3-3），在此基础上还可以表示业务的装配线（assembly line）结构以反映信息系统与业务过程的互动关系（如图 3-4）。图 3-5 用 EPBE 描述了一个简单的 Web 广告开发的业务过程，图 3-6 则用 UML 的类图和对象图描述了一个软件公司的组织结构。

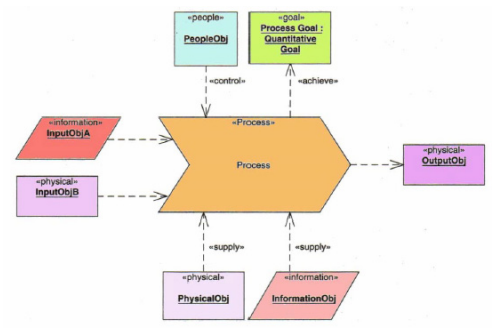


图 3-3、UML-EPBE 业务过程

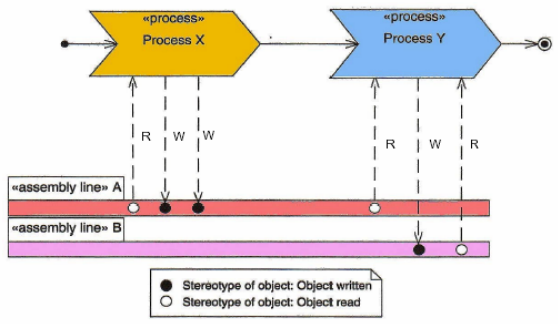


图 3-4、用 UML-EPBE 表示装配线

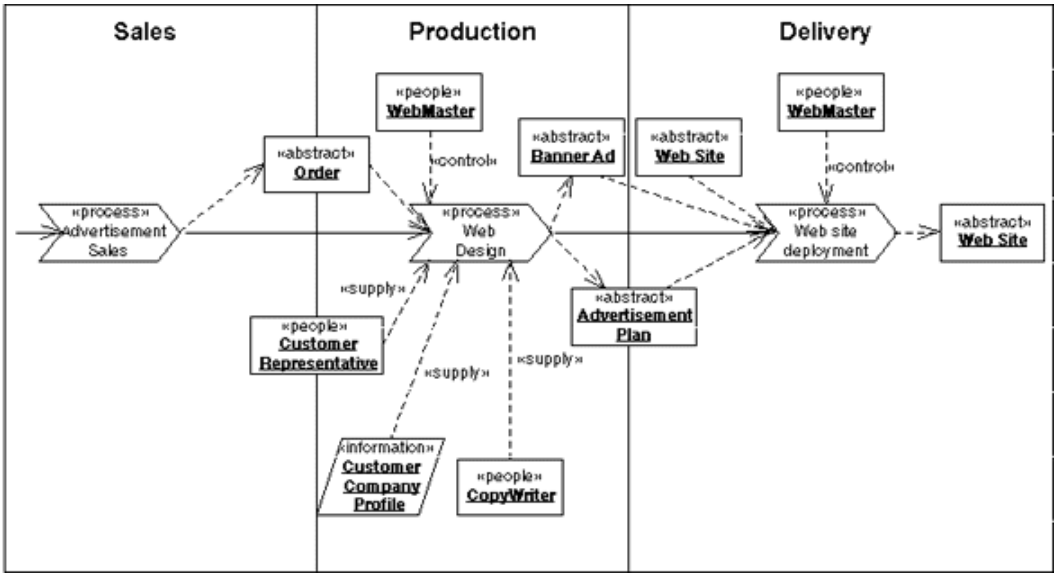


图 3-5、用 UML 活动图和 EPBE 描述商业过程

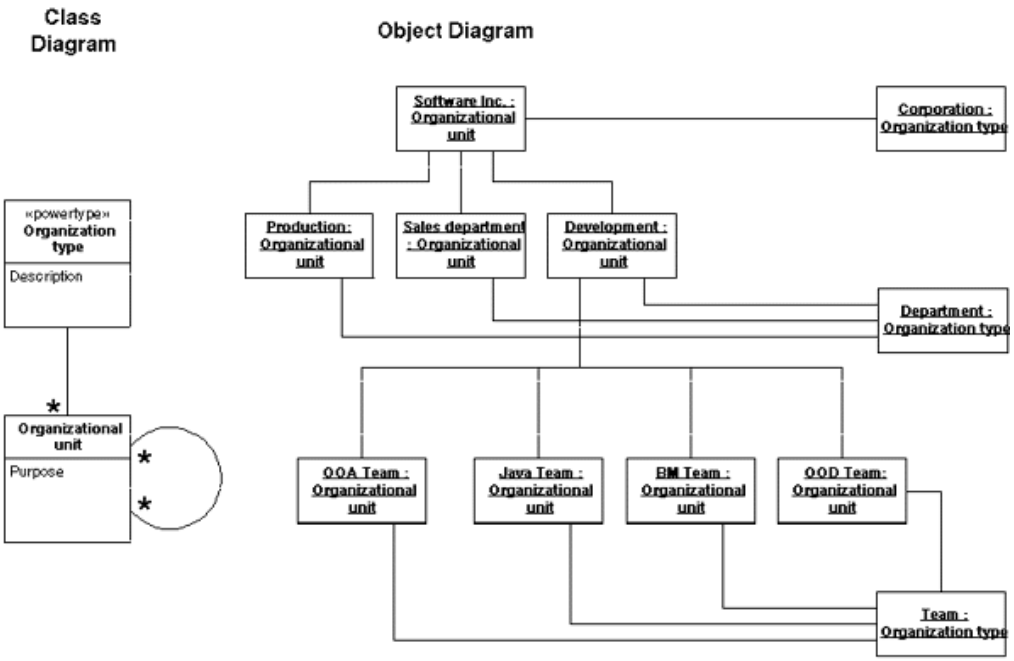


图 3-6、用 UML-EPBE 表示公司组织结构

UML 适用于各种应用领域的建模，包括大型、复杂、实时、分布式、集中式数据或计算以及嵌入式系统等等[U 参 01]。迄今为止 UML 已经成功地应用在诸如电信、金融、政府、电子、国防、航天航空、制造与工业自动化、医疗、交通、电子商务等广泛的领域中，用 UML 方法来描述原文中商业公司进销存系统的业务更是不在话下了（见下文实例）。

然而，UML 图形的表达能力也不是无限的。UML 是一种离散的建模语言，并不适合对诸如工程和物理学领域中的连续系统建模。在 UML 业务建模过程中也少不了用文字或其他符号来说明需求（参见图 3-1、图 3-7）。例如，UML 用例图能描述绝大部分的功能需求，但对于一些非功能性需求（比如性能、时间、规则等），则需要用 UML 约束、标记值、附加的文字注释（或独立文档）以及其他的传统表示方法来补充说明。**设计 UML 的真实目的并非企图完全取代文字说明**，而是希望用半形式化、更加抽象的面向对象可视化手段来促进人们对问题取得更为深入和一致的理解，从而提高软件设计的质量和开发过程的自动化水平。设想，如果没有用例的详细规格说明文档作参照，仅凭口头表述何以确认一个用例的活动图画得是否正确？如果一个需求连文字都无法描述清楚，又怎能指望用 UML 来正确描述它呢？所以，**像原文这样不分语境笼统地评论 UML 能否彻底地、完整地描述业务需求，本身就是一个假命题**。

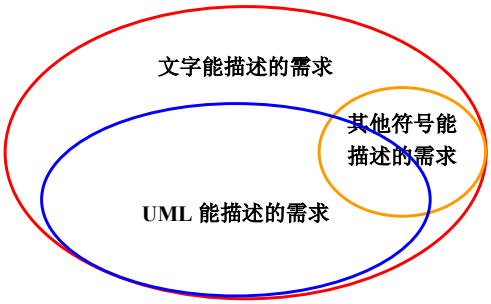


图 3-7、UML、文字、其他符号需求描述能力的关系

相比传统的结构化业务建模方法（如 SA/SD、IDEF、SADT），UML 业务建模给企业用户和软件开发商带来的好处是明显的。这主要表现在 UML 基于面向对象方法既可以描述软件又可以描述业务，获得了主流的面向对象编程语言和先进的面向对象 CASE 工具的有利支撑，从业务分析、系统分析设计、模块编程直到软件测试，项目生命周期的每一个阶段都可以统一采用**一种**模型描述语言，从而保证软件、文档以及人际沟通的连贯性、一致性和有效性，实现**真正意义上的“全程建模”**。

以上，我们从理论和概念上初步分析、阐述了 UML 业务建模的方法和特点，说明了 UML 业务建模方法能够最大程度地从宏观和微观角度完整、准确地描述企业的静态结构和动态行为，促进用户、领域专家与软件开发人员的有效沟通。下面，我们结合实例一一指正原文中有关 UML 业务建模的许多错误观点和用法。

错误 1：UML 很难与企业用户沟通

原文认为，所谓的“上不着天”是指“使用 UML 建模后很难与……企业用户沟通，因为 UML 的表达方式与上游用户的行业知识相差甚远，用户一看见满篇的软件工程术语与符号就发怵，根本无法理解使用 UML 所描述的业务流程，也难以真正理解 UML 所陈述的需求”。

UML 业务建模的目标之一就是促进用户和开发人员对业务模型的一致理解和沟通，因此无论内容和形式都始终强调业务建模应该面向问题域，从用户视点而非开发者视点，尽可能采用业务术语、行业知识来描述信息，少用或几乎不用属于解决域的软件工程术语和符号。UML 的业务建模元素（用例图、活动图、序列图、类图等）也大多采用了一般用户易于理解的直观、形象的表达方式。（见下文实例）

诞生仅 5 年，UML 就已经成为事实上的国际工业标准，被大量软件公司和行业企业所成功采用，这无可辩驳地验证了它的实用性和强大生命力。大量实践证明，基于用例分析和对象观点的面向对象描述方法比单纯基于功能分解的传统结构化方法更加贴近现实世界，更有利于从宏观与微观的角度准确、完整地描述企业的组织结构、业务过程和业务信息。用可视化的 UML 抽象表达大部分过去必须用繁琐、晦涩的文字描述的复杂业务需求，带给用户和开发人员的好处是显而易见的，它代表了时代的进步。

当然，采用任何一种新技术（包括 UML 和全程建模方法在内），都离不开对用户和开发人员进行正确的培训，否则应用效果必然受到影响。作者所说的什么“相差甚远”、“发怵”、“根本无法理解”完全是夸大不实之词，难道因培训不够导致一些人 UML 业务建模的做法不当能等同于 UML“上不着天”吗？（参见下文实例和错误分析）

错误 2：UML 用例图无法描述工作步骤

在“1 UML 难以完整全面地描述企业的分工结构”小节中，原文认为“采用 UML 的 Use Case 图来描述组织结构，它只能描述到岗位职责，对岗位职责中的工作步骤无法描述。”这完全是误解。

首先，作者说用“Use Case 图来描述组织结构”本身就是错误的。UML 用例图是用来描述系统和业务功能的，组织结构应该用类、对象和包（package）图来描述（正确画法如图 3-8 所示，另一种类似组成结构树的画法参见图 3-6）。

其次，作者在原图 3（“采用 UML 描述的分工组成结构至多只能描述到职责级”）中没有采用业务用例和业务使用者（business actor）符号也是不符合规范的，说明作者并不了解 UML 规范（或 RUP）已经定义了业务建模表示法。

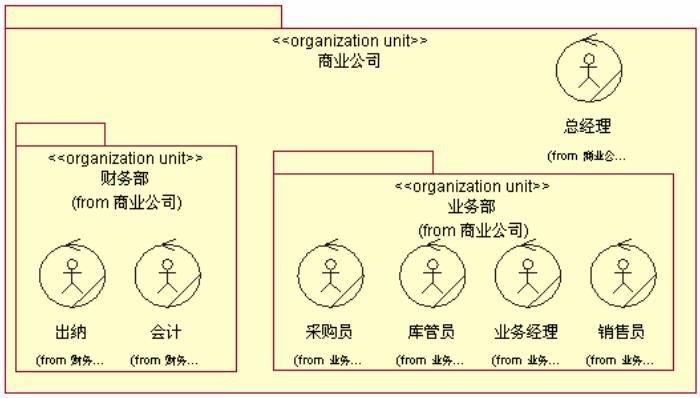


图 3-8、用 UML 类和包图描述企业组织结构

第三，作者为什么非要在一张组织结构图中把岗位职责中的工作步骤乃至原子步骤都一起描述出来呢？工作步骤是业务过程的一部分，属于动态行为，不易在一张主要描述静态特征的组织结构图中表示清楚和全面。通常正确的做法是先用 UML 业务用例图描述组织的业务过程全貌（如图 3-9），再用活动图可视化地分别详细说明每个业务过程的具体工作步骤（包括原子步骤）和细节（如图 3-11）。

如果非要像原图 2（“采用全程建模方法描述的分工组成结构可以细化到原子级工作步骤”）那样把所有东西都放在一起以达到所谓的“直观、彻底、一目了然”的效果，我们也完全可以在 UML 图中加入注释（或在 UML 工具的文档框中表示），照样可以表示出原子步骤（如图 3-9）。但是，如果企业的结构和业务要远比原文所描述的情况复杂得多，有大量岗位、职责分工、工作步骤及其他细节需要描述，全程建模的这种做法显然是不可取的。绘制模型的各部分图形时应保持**关注重点**，避免眉毛胡子一把抓，正是 UML 和面向对象建模的要领之一。

难道非要把分工组成和原子步骤都画在一张图里才叫“直观、彻底、一目了然”，而 UML 为了分层次、多角度，清晰、准确地描述大型复杂需求，用几张图分别来说明企业的组成结构、职责分工和工作步骤，就“不直观、不彻底、不一目了然”了？理论上，如果显示屏或图纸足够大，我们甚至可以把所有的 UML 图形都连起来做成一张图，那么这样做是否就一定“直观、彻底、一目了然”了呢？也未必。

模型从某一个建模观点出发，抓住事物最重要的方面而简化或忽略其他方面。在不损失细节的情况下，模型可以抽象到一定的层次以使人们能够理解[U 参 01]。看来作者还不懂得这个**建模的基本道理**。显然，原文的指责与 UML 的表达能力（硬伤）毫无关系，这只不过与使用者如何在实践中对 UML 模型进行更好地布局和组织有关罢了。

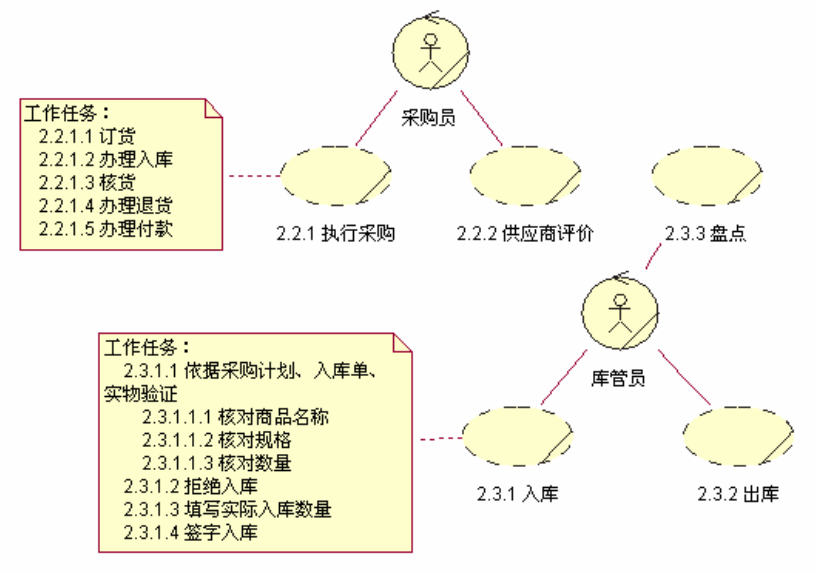


图 3-9、用 UML 业务用例图描述采购员、库管员的岗位职责和任务

此外，原图 2 在岗位职责后面所列出的工作步骤并不能真正反映出一个完整的业务流程。全程建模的组成结构树是以岗位为基础来划分功能的，一个业务，比如执行采购，往往需要多个岗位的联合参与，其中还可能涉及子流程（如图 3-10 中的“入库”）。所以，确切地说，它们并不表示“步骤”，而只是一些细化的职责或任务，不如干脆用文字来表达（如图 3-9）。

我们用 UML（图 3-8、图 3-9）分层次、完整、全面地表达出了原图 2 所反映的所有信息（包括企业的组成结构、岗位职责分工和工作步骤），甚至还描述出了原文无法表达的重要信息——如图 3-10 中，业务使用者与业务过程、业务工员（business worker）的联系，业务过程与业务过程实现（用 business use-case realization 表示）的区别，采购流程需要哪些业务工员共同参与，以及采购与入库流程的关系等等。通过用业务用例来表示业务过程，使得业务过程及其功能和价值可以作为一个整体被显式地表达和量化处理，这是 UML 需求建模优于其它方法的一大特色，远非全程建模（组成结构树）所能比及。

全程建模为什么要在组成结构树中把组织结构、岗位分工和工作步骤都全部画出来呢？我们分析，无非是为了将来可以在同一张图上直接进行一一对应的功能分解和模块定义（见原图 8“采用全程建模方法组成结构树进行功能定义”）。然而，难道软件结构直接照搬业务功能结构是唯一和最佳的设计吗？在讲求更好的软件架构和软件质量的今天，这种设计方法也未免过于草率和简单化了，它如何与主流的面向对象编程方法结合，究竟有多少实用性？这充分反映出机械、片面地套用传统结构化方法与灵活运用健全的面向对象方法相比存在着严重不足。（参见错误 11、图 4-1）

原文仅仅因为画法上的不同（事实上 UML 的处理方式要显著优于全程建模方法），以及错误地认为 UML 在用例图中无法描述工作步骤就得出“UML 难以完整全面地描述企业的分工结构”这一结论，显得十分荒谬。

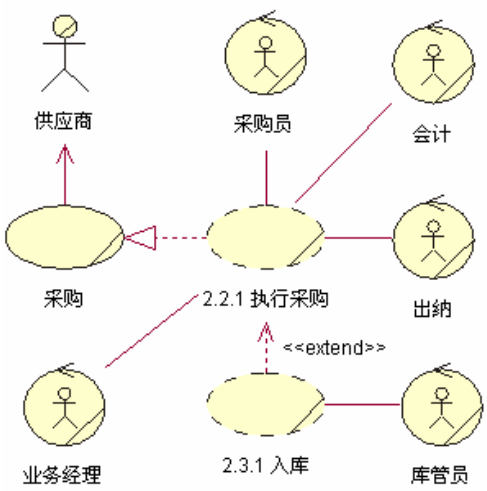


图 3-10、用 UML 业务用例图描述采购业务流程

错误 3：UML 分别用序列图和活动图来描述业务流程是多余的

原文在“2 UML 难以从宏观把控业务流程的完整与准确”小节中认为 UML 用活动图（activity diagram）和序列图（sequence diagram，原文称顺序图）两种图来描述业务流程，“费时费力”，“难免地会出现遗漏、不一致”。这完全是牵强附会，难道把消息交换和活动流程信息全部画在一张图里就能省时省力，不会出现需求遗漏、不一致的问题了吗（参见错误 18）？这反映出作者并不了解 UML 序列图和活动图的真正作用和区别，也不理解 UML 用不同视图分层次、多角度来描述业务的真正目的和好处。

序列图和活动图是两类形式和分工不同的 UML 表示法，语义上存在互补关系。

活动图用于对计算和工作流程的建模，类似于加强型的流程图，可以描述任何**对象内部**（大到一个商业公司，小到一个 C++对象）的工作步骤（业务过程）和执行流程（算法）。在业务建模中，活动图表示了活动进行的流程，但没表示出活动执行的对象，为了完成设计，每个活动必须扩展细分成一个或多个操作，每个操作被指定到具体的类[U 参 01]。序列图被用来描述**对象之间**的消息交换顺序和职责划分，对象之间的消息发送导致服务对象操作的执行。因此，活动图中的活动应该映射到序列图中具体协作对象的操作上，可见两者是完全不同的概念，不可混为一谈。

原文说全程建模可以“将业务事件序列与业务活动有机地集成在一个图形中”是误导，在这里作者不过是挑选了一个**过于简单**的特例——原图 4（“采用全程建模方法的顺序图描述业务协作流程”）中库管员的工作步骤仅有一个条件分支，而且入库活动也是顺序执行。一旦业务很复杂，流程中出现多个分支、多处并发或者呈现网络状的状态变迁，用全程建模的表示方法将无法在一张图中清楚、直观地同时表示出关于业务流程的**整体和局部、高层和低层**的所有信息。（参见错误 2）

仔细比对后我们发现，既然原文的全程建模顺序图（原图 4）与正确的 UML 序列图（如图 3-12）画法之间并无本质上的差别，而后者的表达更为清晰，语义更强，更加适用于描述**一般和特殊、简单和复杂**的情况，那么作者“发明”一种无法普遍适用的顺序图又有何实际意义？

不知是有意还是无意，作者所画的 UML 活动图和序列图（原图 5、图 6）很不规范（见错误 6、7）。以下我们给出正确的画法：用 UML 活动图描述采购业务流程（图 3-11），并在此基础上用 UML 序列图对采购业务流程的工作职责进行了指派和实现（图 3-12）。

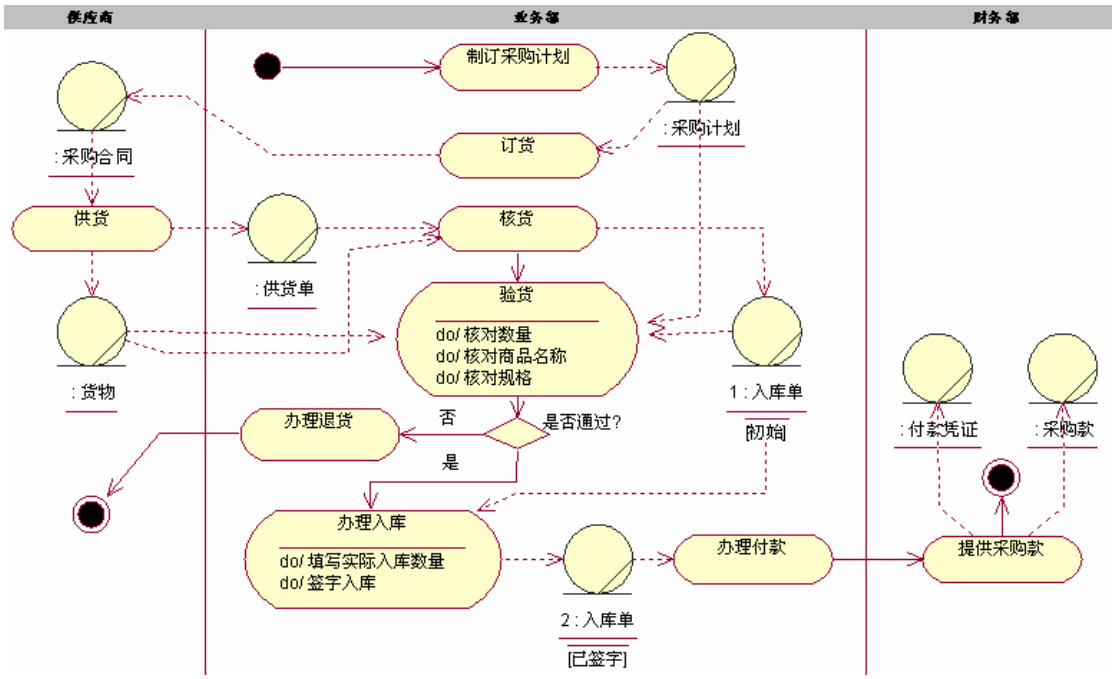


图 3-11、用 UML 活动图描述采购业务流程

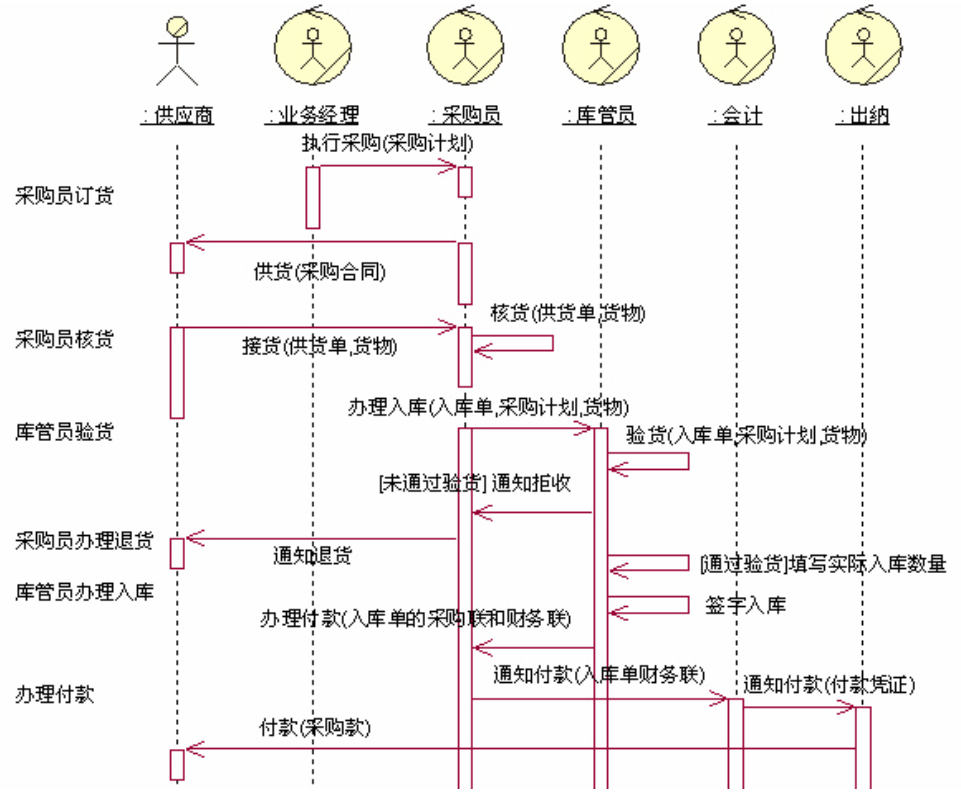


图 3-12、用 UML 序列图描述采购业务流程的实现

错误 4：UML 序列图无法表示条件分支

低级错误！

原文认为“UML 顺序图缺少条件分支的表达方法，表达内容不完整”，简直令人诧异！（正确画法见图 3-12）

2001 年 1 月出版的《UML 参考手册》第 334 页（图 13-162“具有过程控制流的顺序图”）也明白地给出了分叉画法。UML 序列图中的消息标签（message tag）不但能表示分支，而且还可以表示调用嵌套、迭代、并行和同步等复杂信息。[U 参 01]

错误 5：UML 序列图和活动图应该等价

原文认为“UML 顺序图和活动图从形式上到内容上不存在等价关系”。这其实是作者对 UML 基本概念的误解（如错误 3）。

形式上，序列图和活动图原本就是两类不同的图，当然不可能等价。实际上，在 UML 中序列图与协作图（collaboration diagram）近似等价，它们之间可以相互转换，被统称为交互图（interaction diagram）；而活动图是一种特殊的状态图（statechart diagram）。

内容上，序列图与活动图也不是等价关系而是互补关系（见错误 3）。

错误 6：原图 6 活动图错误

对照原图 4（“采用全程建模方法的顺序图描述业务协作流程”）我们发现，不知是有意还是无意，原图 6（“采用 UML 的活动图描述业务协作流程”）中 UML 活动图的画法存在下列问题：

1）很不完整，有 4 个空白活动框。可能是因为作者搞不清 UML 的消息与活动有何区别，所以不知该填些什么。

2）作者搞不清 UML 活动图与序列图的区别，因此干脆对照原图 5 序列图直接画了 6 个泳道，每个参与对象 1 个泳道，不但不规范而且还遗漏了泳道的名称。其实在业务建模中，每个泳道通常代表一个真实世界的组织单元[U 参 01]（参见图 3-11）。

3）“办理报销”这个活动不知从何而来，与原图 4 不一致。

4）在验货通过后，遗漏了库管员入库（填写实际入库数量、签字入库）等活动。

错误 7：原图 5 顺序图差错

对照原图 4（“采用全程建模方法的顺序图描述业务协作流程”）我们发现，不知是有意还是无意，原图 5（“采用 UML 的顺序图描述业务协作流程”）中 UML 序列图的画法存在下列问题:

1）“通知拒收”消息应该是从库管员到采购员，却被画成了从会计到库管员。

2）“通知付款”消息应该是从采购员到会计，却被画成了从库管员到供应商。

3）遗漏了所有的消息参数。

4）遗漏了采购员核货这一步骤。

5）作者不会表示条件分支。

错误 8：UML 无法从微观上描述业务信息的操作

原文在“3 UML 无法从微观把控业务信息的操作过程”小节中认为“UML **根本无法**从微观上描述业务信息的操作过程，只能等到编程时再由有经验、责任心强的程序员去了解”。此种说法毫无依据。

在业务建模阶段，UML 通过活动图、序列图不仅可以描述岗位职责执行的具体逻辑步骤，还可以同时描述业务工员如何对业务实体（business entity）进行操作等细节，很好地从微观上描述业务信息的操作过程。

根据原图 7（“采用全程建模方法的 PAD 图描述的职责细化流程”），我们用 UML 序列图从微观上描述了职责细化流程——库管员如何验货和入库（图 3-13、图 3-14）。

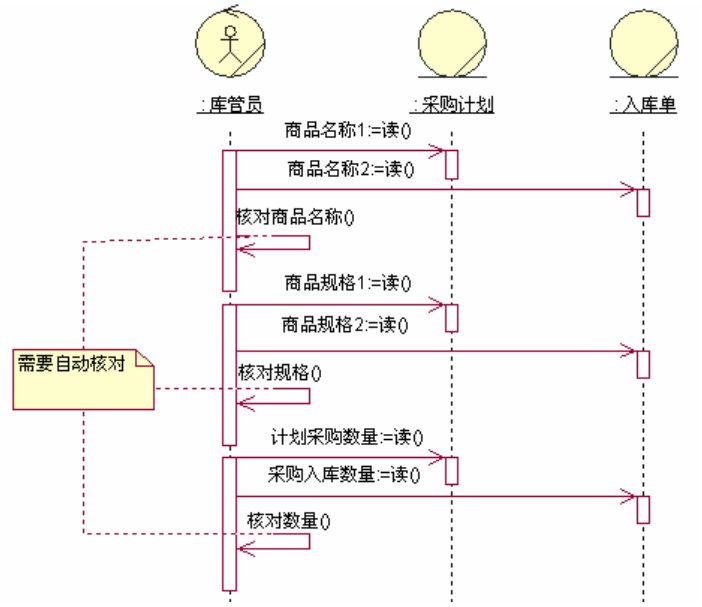


图 3-13、用 UML 序列图描述职责细化流程（验货）

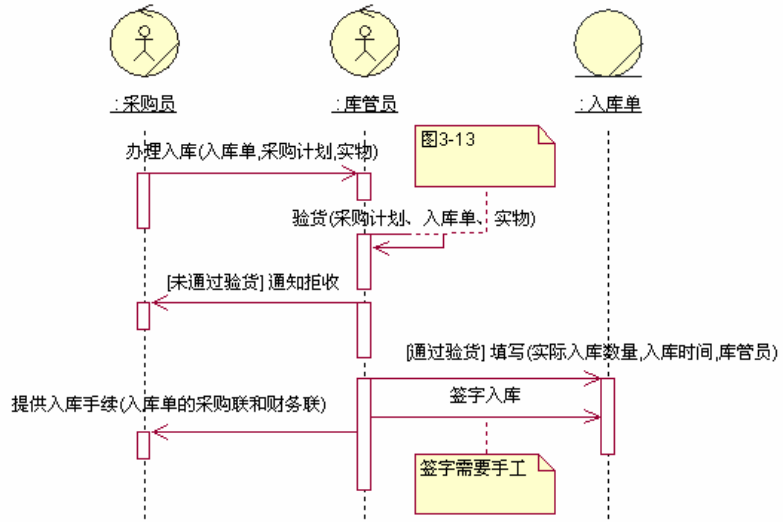


图 3-14、在业务分析阶段用 UML 序列图描述职责细化流程（入库）

错误 9：UML 无法直观地定位细致的功能需求

原文认为全程建模方法的功能定义可以细致到原子工作步骤级，而“采用 UML 无法对这种功能需求（指签字入库需要手工进行）直观地定位”。这又是明显错误的。

用 UML 在原子工作步骤级别上直观地表示诸如“签字需要手工”、“需要自动核对”等需求约束信息是再简单不过的事情（见图 3-13 和图 3-14）。作者怎么能凭一个如此低级的错误判断，就得出“UML 无法彻底全面描述用户的需求”的荒唐结论呢？

后一段作者借题发挥，列举了因需求描述不细致，遗漏了大量用户需要的功能，导致返工和成本增加的例子，而这是与 UML 完全无关的需求管理问题。（参见错误 10）

错误 10：UML 是造成信息不对称的功臣

原文在第 5 小节发出了“UML 是造成信息不对称的‘功臣’”的奇谈怪论，真可谓牵强附会、离题万里，反映了作者对需求工程与建模语言之间的关系还没有真正搞清楚，而且在对软件工程和 IT 项目管理成功关键的理解上也存在着明显缺陷。

UML 模型不过是人的思维的一种反映。在信息化建设中，合同的甲乙双方（用户和开发商）之间存在信息不对称是很正常的现象，正因如此我们需要通过良好的项目管理手段（如需求管理），采用合适的工具（如 UML 建模语言）、方法（如 OOAD 方法论）和过程（如 RUP、PSP/TSP），来增加开发过程的透明度、增进合同双方对彼此的了解、减少项目的盲点和误区。但是，项目的成败与作为抽象表达工具的业务需求描述语言（UML）的好坏之间并没有直接、必然的因果联系。在需求管理中人的主动性起决定作用，UML 作为一种语言工具只是被动地反映。需求管理强调定期地跟踪、核查、确认和调整需求，以适应需求的不断变化。假如项目管理出了问题，用户和开发商之间沟通不畅，对概念、问题的理解不一致，又没有被及时发现、纠正，即使建模语言再好，也可能导致需求分析失败。

另一方面，世界上过去、现在和将来都不存在 100%没有差错和缺陷的、功能 100%完备的业务建模语言。作者敢担保全程建模方法能丝毫不遗漏、无二义地反映所有业务需求吗（见错误 18）？UML 实现了 OO 表示方法的历史性大统一，作为人类目前在面向对象建模领域所共同取得的最高成就，我们相信 UML 所代表的成熟的 OO 方法至少比所谓的全程建模方法要好得多。当然 UML 还有很多不足，有待进一步完善，但这并不等于业务建模乃至整个 IT 项目的失败就可直接归罪于 UML 存在的问题了。前面本文已经对 UML 业务建模的有效性、优点和实质作了理论与实践上的说明。退一步讲，即便在用 UML 描述客户的某些业务需求时可能存在不准确、不全面或不一致的现象，用户和开发商也理应用文字、符号等其他书面形式结合可视化的 UML 来表达需求（参见图 3-7），而且不应该忽视能及时发现问题的需求评审，这正是良好的需求工程和项目管理实践的基本要求。

为了验证自己的观点，作者描述了 3 种现象，可以概括为：1）甲乙双方由于对业务术语理解不一致而导致项目扯皮；2）某些公司利用各种手段欺骗用户使其蒙受巨大损失；3）某些公司雇佣新手做项目，出了问题则推卸责任。这些现象显然都属于工程需求和项目管理方面的问题。如果用户

和开发商没有共同对业务模型进行正确的校验和评审，不管用不用 UML（或其他任何方法比如全程建模），都一样可能发生。

本来 UML 的可视化、丰富的语义和统一性等特点明显要优于文字和已有的许多建模语言，是用户和开发商从事业务建模的良好工具和帮手，能最大程度地减少信息不对称，实现开发全过程信息交流媒介的统一，现在却被作者反打一耙说成是妨碍沟通，实在无理！作者的这些论述东拉西扯，毫无条理和逻辑可言，也许是怕字数不够吧。

4 UML“下不着地”？

错误 11：UML 不支持详细设计

原文认为，所谓的 UML“下不着地”是指“费尽心力地使用 UML 建模后，很难让.....程序员直接接受去编程，因为 UML 的表达方式不直接支持详细设计，程序员还得费尽周折地琢磨如何把建模结果转换成程序代码”。这种说法是完全错误的。

面向对象详细设计（OOD）的主要任务是对类的属性、操作、相互联系和内部状态进行明确、一致的定义。在这个阶段，UML 可以通过类图（代码的逻辑静态结构）、状态图（对象内部的状态变迁）、活动图（操作执行流程）、交互图（对象协作）等各种描述手段直接指导程序员轻松地编程。流行的 UML 工具还可以把 OOD 模型（类图、状态图等）自动地转换成程序代码框架，甚至包括与序列图调用消息对应的操作内部实现代码（如 TCC）。（参见错误 12）

针对原图 9（“采用全程建模方法 PAD 图描述的模块级流程与伪代码”），我们用 UML 序列图完整地描述了类似的模块级信息。图 4-1 通过引入接口类（StoreEntryForm）、控制类（StoreEntryManager）、实体类（StoreEntrySheet）和子系统（采购管理子系统），实现了从需求分析（参见图 3-14）到详细设计的过渡，图中还指明了具体的对象调用操作和参数，程序员完全可以在此 MVC（Model, Controller, View）框架的基础上进一步编程实现有关功能。由于查验实物、签字入库需要手工介入（图 3-14），我们加入了系统使用者“库管员”验货和 StoreEntryManager 打印入库单的步骤。该序列图还采用了异步消息和返回消息等表示方法。相比之下，原图 9 的 PAD 图和伪代码的内容要粗糙得多，除了简单重复原图 7 已明确的业务流程之外，对指导程序员设计具体软件结构以实现业务功能并无实际用处。

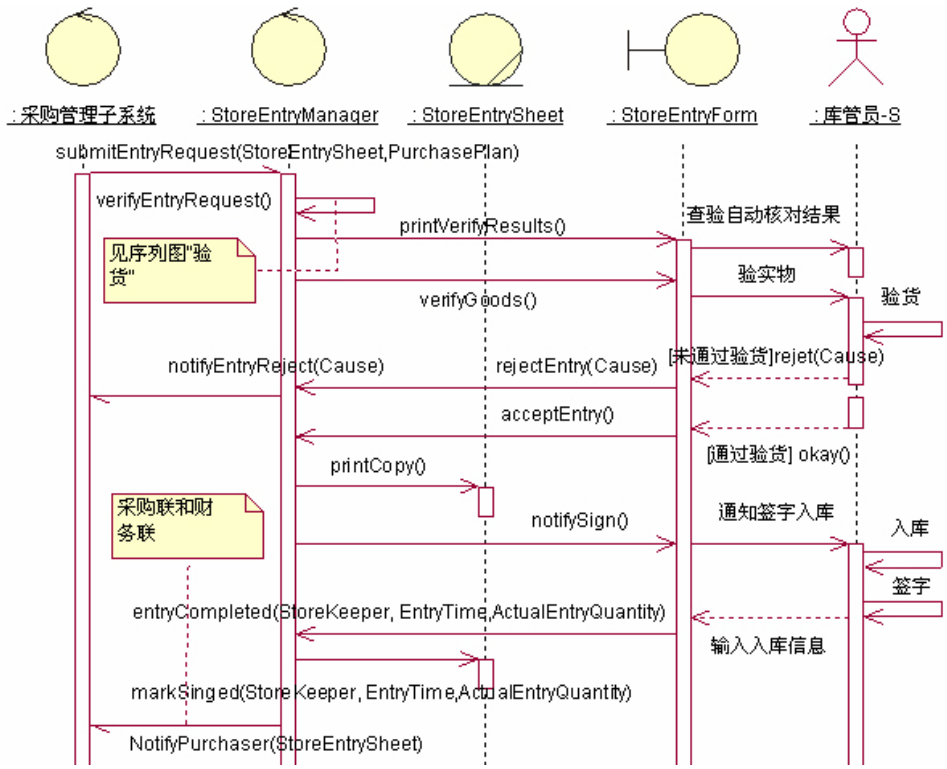


图 4-1、在系统分析阶段用 UML 序列图表示模块级流程（入库）

错误 12：现代编译器支持的是结构化程序设计

原文认为“与现代编译器对接的是结构化程序设计，如伪代码、PAD（问题分析图）”。作者尽然凭这样一句话，就轻而易举地否定了主流的 C++、Delphi、Java、VB 等现代语言编译器（解释器）及其所代表的面向对象程序设计方法在业界早已被广泛采用的现实！

作者说，“伪代码的优点是从语法结构上与通常的高级语言非常接近”（原图 9），可是“接近”并不等于就是高级语言，其实这也算不上什么优点。UML 工具支持从 UML 模型直接、方便、自动地生成各种平台的主流编程语言代码（“正向工程”），可以根据许多已知的软件模式生成代码框架或模板（如 Rose、TCC），实时 UML 建模工具（如 Rhapsody、Rose RT、Tau）甚至还可以从 UML 状态机直接生成可执行的代码和程序，并自动执行、调试模型和代码。另外，这些工具可以从已有的高级语言代码中抽取出具有相同语义的 UML 模型（“反向工程”），实现了真正的可同步“循环工程”，极大地方便、简化了程序员复杂、繁重的设计与编程工作。

原文还写道：“PAD 由于可视化的特点十分便于理解”。不错，可视化的 UML 不但比全程建模语言的 PAD 图更加胜任 OOD 的工作，而且可以完全取代伪代码（如图 4-1）。实际情况是，如今几乎没有听说还有哪些企业和个人在用伪代码或 PAD 图指导程序员编程。

错误 13：UML 没有考虑系统级和模块级的接口

原文认为“UML 没有对系统级、模块级接口的考虑，这在大型复杂系统开发中是不可想象的”。又一个低级错误！这种无知简直令人“不可想象”！

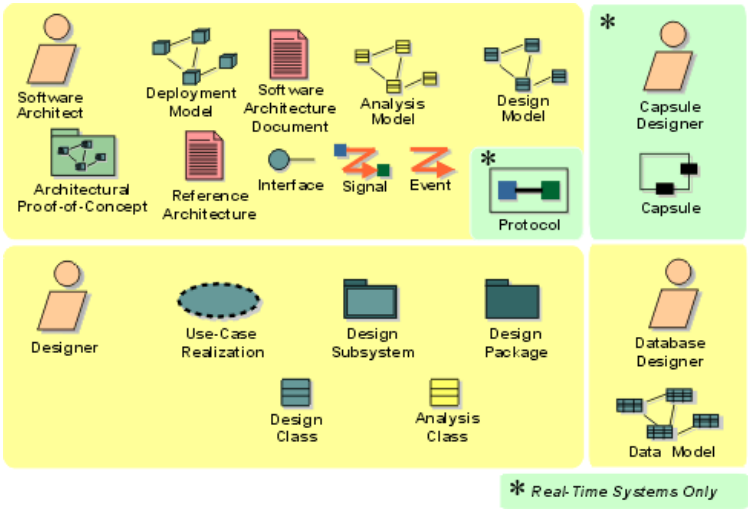


图 4-2、RUP 分析设计工件集

通过接口(interface)实现对象行为的多态性以及对数据和功能的一致封装是面向对象的核心思想。UML 方法（如 RUP）从系统架构（architecture）到子系统（subsystem）再到构件（component），都极其强调接口的设计（参见图 4-2）。UML 在这方面的表示方法和手段也是丰富多样的，比如可以用类图描述系统级、模块级接口的组成和静态关系，用交互图描述子系统之间如何通过接口进行协作（参见图 4-1、4-3），以及用构件图来表示接口的使用关系等等。

针对原图 10（“采用全程建模方法中数据汇总图描述的系统之间的接口”），我们用 UML 协作图完整地描述了类似的信息。

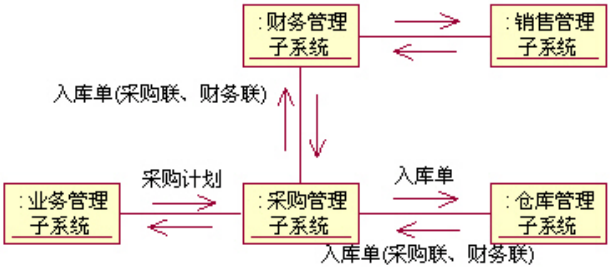


图 4-3、用 UML 协作图描绘系统之间的接口

5 UML“一盘散沙”？

错误 14：UML 图形的内部联系十分松散

原文认为“UML 建模图形之间的内部联系十分松散，这种隔阂造出了 UML 的第三大硬伤”，并列举了一些“散沙”来证明。这些观点其实都是错误的。

OMG 在正式发布的 UML 1.4 规范中用了多达 566 页的篇幅对 UML 的基本语法和语义进行了细致、精确、严格和完整的定义与说明，《UML 参考手册》、《UML 用户指南》、RUP 等大量传统和网络文献也非常详细地对用 UML 进行面向对象分析设计的过程和方法进行了阐述。我们不难发现 UML 规范定义的建模元素之间无论语法还是语义上都存在着紧密的内在联系。

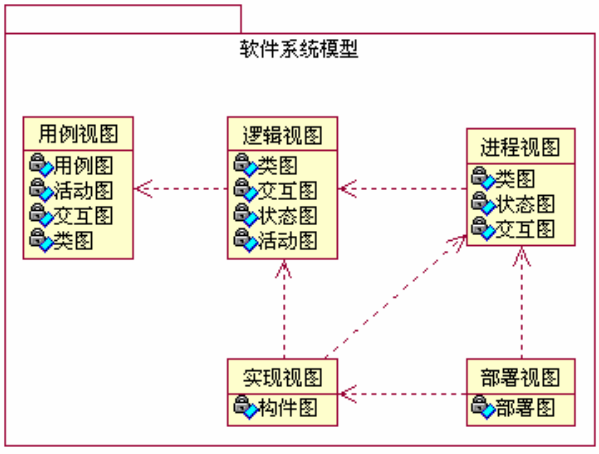


图 5-1、软件系统“4+1”视图

从语义上看，我们通常从用例视图、逻辑视图、进程视图、实现视图和部署视图（“4+1”Views）5 个不同的角度来完整、清晰地描述大型复杂软件系统。UML 的各种图示和元素在这 5 种对应建模不同阶段的视图中分别起到了不同的作用。例如，用例视图中的用例代表了系统的功能需求，用例的具体内容可用活动图、交互图来描述，它们体现了系统的动态行为，实现系统的用例则是整个设计活动的起点与核心；逻辑视图中的类图和对象图反映了实现用例的参与对象的静态结构，包括内部信息和外部联系，以及软件架构、子系统的分层结构；进程视图中的交互图和状态图说明了对象所在进程、线程的动态执行关系；实现视图中的构件图说明了软件的静态物理组成，构件的封装和使用关系等等……所有这些图形之间都可以建立明确的语义追溯（trace）关系。（图 5-1）

从语法上看，UML 规范 1.4 引入了 4 层体系结构，自下而上分别为：UML 元元模型、UML 元模型、UML 模型和用户数据，下层模型为上层模型提供定义基础，上层模型则是下层模型的实例化（图 5-2）。其中，我们熟悉的 UML 元素（如类、属性、操作和构件等）组成了 UML 元模型，UML 元模型的内部结构由基本元素、行为元素和模型管理 3 个包组成（如图 5-3 所示）。

Layer	Description	Example
meta-metamodel	The infrastructure for a metamodeling architecture. Defines the language for specifying metamodels.	*MetaClass, MetaAttribute, MetaOperation*
metamodel	An instance of a meta-metamodel. Defines the language for specifying a model.	*Class, Attribute, Operation, Component*
model	An instance of a metamodel. Defines a language to describe an information domain.	*StockShare, askPrice, sellLimitOrder, StockQuoteServer*
user objects (user data)	An instance of a model. Defines a specific information domain.	*<Acme_SW_Share_98789>, 654.56, sell_limit_order, <Stock_Quote_Svr_32123>*

图 5-2、UML 体系结构[UMLS01]

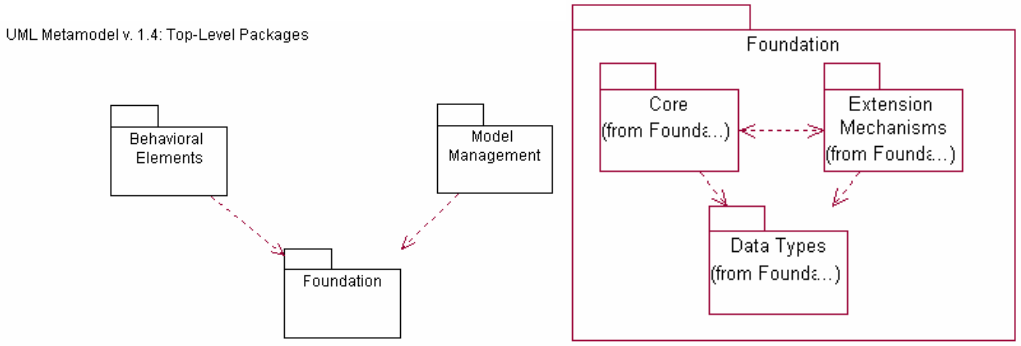


图 5-3、UML 元模型的内部结构

以下我们依次指出原文所列举的“散沙”错在何处，所谓的“UML 建模图形之间的内部联系十分松散”、“没有在细微之处建立建模图形之间的联系”、“一盘散沙”纯粹是无端指责。

错误 15：UML 状态图中的事件与使用者、类和包无关

原文认为“1)状态转移图中，事件与外部 Actor、Class、Package 等无关”。

常识性错误！

UML 状态图中的事件有几种类型：信号事件、调用事件、改变事件和时间事件等[U 参 01]。信号事件和调用事件的发生涉及到对象之间的通信，事件的来源可以是外部的系统使用者（system actor）或内部对象（如图 4-1），与这两者都有关。另外，包通常只是用来管理、组织 UML 模型内容的，不是类、对象或子系统，怎么可能与事件有关呢？

错误 16：无法从语法上建立 UML 活动（状态）图与序列图的联系

原文认为“2)无法从语法上建立状态转移图与顺序图的联系；3)无法从语法上建立活动图应与顺序图在流程描述中的关系”。

在 UML 语法上，状态（活动）图、序列图原本就是两类完全不同的图，如果一定要说什么联系的话，那么我们可以发现它们共用了一些公共的行为元素（图 5-4）。例如，状态（活动）图和序列图都用到了 Common Behavior 包中的动作（Action）元素，动作既可以附属于状态机中的变迁（在两个状态之间或在一个状态之中）也可以附属于对象之间交互消息的发生[U 参 01]。

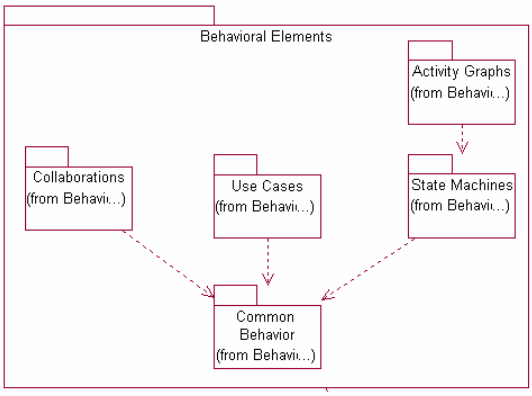


图 5-4、UML 协作图与状态（活动）图的语法联系

实际上，从语法上建立这两类图的联系，并无多大的使用意义，作者大概错把“语义”（semantics）说成了“语法”（syntax）。前面我们已经介绍了如何从语义上建立 UML 状态（活动）图与序列图的联系（参见错误 3、5）。

错误 17：UML 交互图中的消息参数与类图无关

原文认为“4) 协作图和顺序图中与 Message 相伴的参数与类图无关”。

错得莫名其妙！

UML 语义规定了 UML 模型中任何调用消息的参数最终都应与目标类的操作参数对应，理论上它们毫无疑问与类图是有关的，否则模型在语义上就是不一致、不完整的。在完成一个阶段的设计后，我们可以利用某些工具的一致性自动检查功能来保证 UML 模型的正确性。

然而，由于 UML 方法通常采用迭代开发方式建模，所以在业务和系统分析阶段，交互图中的消息也可用短语表示，暂时不与类图对应（这时的类还不是真正的设计类），当然这只是中间结果（如图 4-1）。

作者不是混淆了 UML 在分析和设计阶段使用上的差别，就是把某些 UML 工具的功能特点——比如即可以用短语表示消息，也可以在快捷菜单中指定需要调用的类操作和参数，或者没有提供模型一致性检查功能——硬说成了 UML 本身的缺陷。

错误 18：用全程建模方法所建模型的错误

除了作者在 UML 认识和使用上的错误外，我们利用 UML 业务建模还发现一直攻击 UML 的作者在

自己用全程建模方法所画的图中“尽然”也存在需求遗漏、不完整、不一致、不准确的问题，例如:

1）原图 2 中采购员的工作步骤“接货”到了图 8 就不见了。

2）原图 2 中库管员的工作步骤“拒绝入库”在原图 4 全程建模的顺序图中消失，而顺序图中的消息“通知拒收”和步骤“办理退货”却没有在组成结构树中有所反映。

3）原图 4 顺序图中库管员的第 1 个步骤写为“依据采购计划、入库单、实物验货”，可是采购员到库管员的消息“办理入库”输入参数中只有入库单，遗漏了采购计划和实物。

4）业务流程设计不准确。其一，“签字入库”似乎不应被视为“一个”原子动作，究竟应该是先签字后入库，还是先入库后签字？其二，填写实际入库数量和签字入库两个步骤应该哪个在先？如果按照原文库管员在验货通过后立即填写“实际”入库数量，由于入库发生在后，那么一旦实际的入库数量有变，与入库单上原先填写的入库数量不一致怎么办？所以，填写实际入库数量似乎应该在入库完毕之后进行。本文图 4-1 对此作了改进。

6 经验教训

6.1 原文错误总结

以上我们从理论和实践两个方面对《UML 三大“硬伤”》的大量错误观点一一进行了批驳，总结如下:

在提倡学术民主的同时我们不应违背客观、公正、诚信的基本科学准则，抛弃严谨求实的学术态度和作风。通过以上分析，我们发现作者在指责 UML 之前，根本就没有认真地阅读过 UML 规范、参考手册，对 UML、面向对象技术的作用和本质的理解也是非常片面、肤浅和错误的。难道对于自己不了解、不熟悉的新生事物，不做全面细致的调查研究，不拿出确凿的事实数据，不经过缜密的论证，就可以信口胡言、妄下结论了吗？对面向对象和 UML 的原理一知半解，连基本的序列图、活动图都画不好，搞不清两者的区别，不理解 UML 的统一性和内聚性，甚至不知道 UML 完全可以支持详细设计，序列图完全可以表示分支（《UML 参考手册》中好几处白纸黑字地写着），这样的人怎么有资格、有能力、有胆量来对 UML 进行所谓的“攻击”（原文用语）呢？作者竟然把 UML 的优点说成缺点，而且明明自己概念不清、使用不当，却硬是把自己的错误转嫁到 UML 身上，说成是 UML 的“硬伤”，岂有此理！这无疑是一种以己之短攻彼之长，夜郎自大、颠倒是非的浮躁行为。

编号	描述	程度	类别	可能原因分析
1	UML 很难与企业用户沟通	严重	使用错误	作者不懂 UML 业务建模的正确做法
2	UML 用例图无法描述工作步骤	严重	概念和使用错误	作者不懂静态结构和动态行为表示的区分
3	UML 分别用序列图和活动图来描述业务流程是多余的	严重	概念错误	作者对序列图与活动图的关系理解有误
4	UML 序列图无法表示条件分支	严重	概念和使用错误	作者不懂如何表示分支
5	UML 序列图和活动图应该等价	严重	概念错误	作者对序列图与活动图的关系理解有误
6	原图 6 活动图错误	严重	使用错误	作者不会画活动图
7	原图 5 顺序图差错	严重	使用错误	作者不会画序列图
8	UML 无法从微观上描述业务信息的操作	严重	概念和使用错误	作者不了解 UML 的表达方法和手段
9	UML 无法直观地定位细致的功能需求	严重	使用错误	作者不了解 UML 的表达方法和手段
10	UML 是造成信息不对称的功臣	严重	分析错误	作者搞不清 UML 与需求管理的关系
11	UML 不支持详细设计	严重	概念和使用错误	作者不知道 UML 交互图、活动图和状态图在 OOD 中的作用
12	现代编译器支持的是结构化程序设计	严重	分析错误	作者无视大量采用 OO 语言和工具的成功应用的存在
13	UML 没有考虑系统级和模块级的接口	严重	概念和使用错误	作者对 OO 和 UML 接口本质的理解缺陷
14	UML 图形的内部联系十分松散	严重	概念错误	作者对 UML 统一性、内聚性理解缺陷
15	UML 状态图中的事件与使用者、类和包无关	严重	概念错误	作者不懂 UML 事件的真实含义
16	无法从语法上建立 UML 活动（状态）图与序列图的联系	严重	概念错误	作者不理解序列图与活动（状态）图的关系
17	UML 交互图中的消息参数与类图无关	严重	概念和使用错误	作者混淆了 UML 语义与 UML 工具能力的差别
18	用全程建模方法所建模型的错误	普通	使用错误	作者建模失误

表 6-1、《UML 三大“硬伤”》错误总结

6.2 UML 建模与全程建模

对于全程建模方法，由于我们不太了解，不便多说，但是从原文的介绍中我们还是可以看出些许端倪。

首先，所谓的全程建模，尽管借用了 UML 的个别符号（如使用者）和术语（如顺序图、接口），但本质上还是一种基于功能分解的传统结构化方法。这种方法的最大缺点是，软件内部结构与外部业务结构单纯对应（如原图 2、8），业务数据和软件功能分离，没有真正的接口和多态概念，导致设计出来的软件重用性、扩展性和稳定性较差，质量远不如用面向对象方法设计出来的软件，因此很难适用于大中型复杂商业应用的开发。UML 和 OOAD 方法是软件重用、架构、构件、设计模式等当代先进的软件构造技术的基石，而基于过时的、机械的结构化思想的全程建模方法并不符合软件构件化的发展趋势。

其次，在业务建模方面结构化分析和基于 UML 的面向对象分析**都是可行的**，但是由于面向对象编程语言已经成为商用软件开发的主流，用全程建模方法设计出来的结构化模型如何与主流编程语言对接将成为一大“隔阂”。如果办不到，只能应用于结构化程序设计的“全程”建模方法在如今的商业软件开发中又有多大实际用处呢？即使全程建模方法能够实现从结构化分析到面向对象分析设计的转换（例如国际上一些著名的结构化建模工具纷纷采用了 UML 用例来做过渡），我们也不禁要问，既然已经有了作为通用工业标准的 UML，既能完成业务建模又能实现面向对象软件建模（尽管仍有不足尚需改进），那么我们还有什么必要再去闭门造车地“发明”新的“轮子”呢？可见，全程建模方法的基本设计思路是与 8 年以来统一软件设计方法的国际努力和大潮流背道而驰的。

根据以上分析，UML 建模和全程建模方法孰优孰劣，不辩自明。

6.3 正确对待 UML

之所以不少读者懵懵懂懂地认可原文的论调，是因为在国内 UML 的实践中确实存在一些问题和现象（包括正确和错误的体验），但是我们要问，从这些现象中能得出 UML 硬伤的结论吗？在 UML 实践中遇到问题，我们应该首先分清主客观原因，到底是自己思想认识和使用方法的“高不成、低不就、凌乱不堪”，还是 UML 语言和 OOAD 方法本身的“上不着天、下不接地、一般散沙”？一个科学的实践者，在类似的原则问题上一定要严谨求实，不然就会犯常识性的根本错误。

有效分析 UML 的优缺点应该建立在对 OOAD 方法论和 UML 规范**虚心学习、正确理解**的基础上。任何人要想正确地证明 UML 的能力缺陷，都应当首先查找各种已公开的文献、资料和工具，对由 UML 元模型、UML 标准和非标准的扩展集所反映的综合能力进行研究。即使所有已知的显式手段都不能解决，也不能轻易得出 UML 存在硬伤的结论，还应该通过严密的推导（可能用到形式化方法）来验证 UML 隐式的扩展机制（构造型、标记值、约束和 OCL）是否存在本质上的缺陷，并且一定要用确凿的实例加以准确的逻辑分析来验证自己的判断（参见图 2-1）。

UML 并非空想家理论上新的臆造，而是 30 年来面向对象理论探索、技术创新和工程实践发展水到渠成的结果，UML 于 1997 年成为 OMG 标准如同 1994 年《设计模式》一书的发表都是软件史上的里程碑。何谓“硬伤”？按字面理解，硬伤不是一般容易治愈的小伤，作者又在另一篇文章中对该诊断做了危言耸听的注脚——“先天缺陷”[高 202]。然而，UML 不过是比普通文字或其它面向对象描述语言更好用的对 OOAD 方法的可视化表达，它的问题充其量也不过是能在多大程度上正确地反映面向对象的概念，因此如果说 UML 存在什么重大问题，那一定首先是 OOAD 方法本身出了严重问题，然而近 10 年来全球软件技术进步所取得的成就和未来发展趋势的真相又如何呢？如果没有厂商愿意推出新的更强大的 UML 工具，愿意使用 UML 的人越来越少，UML 标准也不再更新，那么 UML 才真正遇到了大问题！证明 UML 存在硬伤、OOAD 存在先天缺陷肯定是国际上一项了不起的学术成就，可惜原文只顾从一个狭隘保守的结构化方法推崇者的视点坐井观天式地无谓攻击，并没有为我们提供任何真实可信的证据。

实践证明，“**有用的即是好的**”。本文的写作目的只是为了就事论事地指出原文所存在的认识和使用错误，并无意给出对 UML 能力完备性的完整证明，这超出了作者的能力，对于解决当前国内软件业技术和管理实践中的迫切问题也无实际意义。

世界上原本就不存在完美无缺的技术，UML 自然不能例外，其规范文本也一直在不断完善、修订以消除差错，然而我们认为，**UML 出现的问题是发展中的问题**，远非国内某些专家所谓的“重大问题”。其实，UML 面临的主要问题不是不能业务建模、不能详细设计、一盘散沙，恰恰相反，它的统一框架、丰富语义、灵活性和强大扩展性，导致短短几年内吸引了国际上各个领域的软件专家为各自应用的目的纷纷为 UML 添加新的功能，增加自定义的构造型和扩展集，因此，如不及时对此加以规范和整理，很可能造成 UML 架构膨胀，内核出现特性冗余、“根部肥大”（Root Bound）[张 02]的现象，因此 OMG 在制定 UML 2.0 中的主要任务就是对 UML 内核进行一次重构（refactoring），保

持 UML 体系结构高内聚、低耦合的优良特点。同时由于 UML 具有良好的分层结构和屏蔽机制(图 5-2), UML 内核的改进、优化对于建模者一如既往地使用 UML 来说并无多大影响。

当前国内与国际面向对象技术领先水平的差距在 5-8 年左右。提高国内 UML 和 OOAD 的应用水平, 只能靠正规的培训、有效的实践和充分的经验交流。关于如何看待 UML 的价值与发展中的问题，还有哪些 UML 的应用和认识误区，UML 及其工具有哪些丰富的功能，以及结构化方法与面向对象方法的比较等相关问题，由于篇幅限制本文不再展开讨论，请对此感兴趣的朋友关注后续文章并参加我们在 IT 之源[ITS02]和 UMLChina 上组织的有关讨论。因水平局限，本文难免存在差错和不足，敬请您提出宝贵意见。

参考资源

[EPBE99] Eriksson, Hans-Erik and Penker, Magnus: "Business Modeling with UML: Business Patterns at work", Wiley & Sons, Fall 1999.

[RUP02] Rational Unified Process, Rational Software Corporation, (http://www.rational.com)

[UMLS01] UML Specification 1.4, OMG, 2001/9 , (http://cgi.omg.org/cgi-bin/doc?formal/01-09-67)

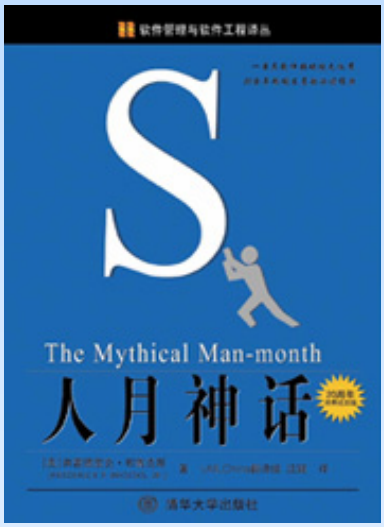
[高 102] 高展，“UML 三大‘硬伤’”，《程序员》，2002/5, (http://www.csdn.net/subject/79)

[高 202] 高展，“质疑第一代面向对象建模技术”，《程序员》，2002/9, [ITS02] UML 与 OOAD 专题讨论，IT 之源, http://www.iturls.com

[U 参 01] James Rumbaugh、Ivar Jacobson、Grady Booch 著，姚淑珍、唐发根等译，《UML 参考手册》，机械工业出版社，2001/1

[张 02] David Dikel、David Kane、James Wilson 著，张恂译，《软件架构——组织原则与模式》，机械工业出版社，2002/9

《人月神话》发行了！



《人月神话》20 周年纪念版

Fred Brooks

翻译：UMLChina 翻译组 汪 颖

二十五年后，我们仍然在读这本书

——Ed Yourdon

Fred Brooks（右，《人月神话》作者）接受图灵奖

Fred Brooks 在作虚拟现实研究报告

电梯系统的 UML 文档

Lu Luo 著，王君 译

查看评论

1 简介

这是一份 Carnegie Mellon 大学博士课程（分布式嵌入系统）项目报告。整个课程完成了一个分布式实时系统的设计、搭建和模拟。设计时用到了 OOA 和 OOD，特别是 UML。

系统的大多数类省略了很多细节。现在看到的这份电梯系统的 UML 文档和真实的电梯系统有很大的不同。因此不是很清楚 UML 是否能真正地完成电梯系统的设计。这份报告基于当前的系统设计给出了一个教学项目严谨 UML 文档包。

分布式实时系统中如何使用 UML，报告从不同的视图给出了三组 UML 图。这些图分别从对象结构、软件结构和系统结构三个角度着眼，不同主要集中在各自的类图上。

接下来的第二节和第三节分别是 UML 和分布式系统的简介。第四节是从静态结构的角度来描述我们的电梯系统的设计，例如从用例图和类图来描述和分析。第五节的用例图和状态图主要描述系统的动态方面，而第六节是结论。

2 UML 简介

统一建模语言（UML）是描述、形象化、构建及文档化软件系统和其他非软件系统构件的工业建模语言。它简化了软件设计的复杂过程，为构造建立了“蓝图”。并且，现在是软件构造的标准符号语言。

UML 提供了系统的结构图和行为图。一组包含不同图形元素的图表是 UML 最具表现力的核心部分。UML 包括 9 种图。为了把握电梯系统设计最典型的部分，在本文中只用 UML 图表来分析：

用例图描述一组用例和角色（一种特殊的类）以及它们的关系。用例图用在一个系统的静态用例视图中，这些图在一个系统行为的组织和建模中是重要的。

类图描述了一组类、接口、协作以及它们的关系。它在面向对象系统建模中是最常用的图。类图用于系统的静态设计视图。

顺序图是一种交互图。交互图用于系统的动态视图。在 UML 中，除了顺序图，**协作图**也是交互图。顺序图着重于系统中对象间消息传递的时间顺序，而协作图着重于发送和接受消息的对象的结构组织。它们是异形同构的，可以从一个转换为另一个。虽然它们都是对我们系统同样的扩展理解，但顺序图给出了时间，这是实时系统的要素，所以本报告中只给出顺序图。

状态图由状态、转换、事件和活动组成，它表明了状态机制。状态图用于一个系统的动态视图。状态图在接口、类或协作的建模时特别重要。它突出了一个对象的事件顺序行为，这在交互式系统建模中非常有用。

其他的四种 UML 图是：**对象图**-描述一组对象和它们的关系；**活动图**--一种特殊的状态图，描述一个系统中从一个活动到另一个的流程；**组件图**-描述一组组件的组织和依赖关系；而**实施图**描述运行时处理点的结构和依附于它们的组件。

3 实时分布嵌入式系统总览

在讨论用 UML 设计我们的电梯系统的细节问题之前，必须先对实时分布式嵌入系统的定义做一个界定。简述用 UML 进行完全面向对象的设计和分析时，它和一般的软件不同。接下来，本文将对在实时分布式嵌入系统的设计中使用 UML 的优缺点作大量的讨论。

Kopetz 说过**实时**计算机系统行为的正确性不仅和计算的逻辑结果而且和结果产生的物理及时性有关 [1]。谚云：“过时的正确结果是无效的”。在实时系统中性能要求和功能要求同样重要，我们不仅要完成正确的功能，还要有必须完成这些功能的清晰系统边界。**嵌入式计算机系统**用一台计算机作为组件，但是它的主要功能不是一台计算机。

作为一种面向对象技术，**UML** 用于实时系统的开发基本上是适合的。UML 中的方法自然地适合于描述和设计实时系统。用例图描述的是人及外部设备和系统的相互作用。对象的顺序图描述的是**包含时间**的用例、引起相互作用的事件和系统回应的细节。

类图帮助我们将系统的组件相互分开并定义它们之间的接口。这些技术足够用于捕获处理场景和识别可靠的时间问题。

现在我们来回答用 UML 设计电梯系统的实践中遇到的问题：“UML 是一种适合于实时系统的建模语言吗?”我们发现基于上段提到的特征，UML 是适合的但有不足。用 UML 设计实时系统有以下问题：

•特定硬件及它们特征的定义。

•在对象、任务和硬件层次描述时间约束。

•网络建模。

在接下来的章节，我们将指出如何较好地用 UML 描述电梯系统。下面的实践建议可能是标准 UML 有益的补充。

4 系统静态建模

4.1 电梯控制系统快照

作为我们的教学项目，电梯系统的设计与“真实”的系统相比省去了很多技术上的细节。我们的电梯系统有所有的电梯系统都有的基本功能，如上升和下降、开门和关门当然还有载客。电梯假设被用在一幢大楼的第一层到第 MaxFloor 层，第一层是大厅。电梯里有每一层对应的呼叫按钮。除了第一层和顶层，每一层都有两个按钮，乘客可以呼叫上楼或下楼。顶楼只有一个下楼按钮，而大厅只有一个上楼按钮。当电梯停在某一层，电梯开门，电梯指示灯亮标明当前运行的方向，这样乘客就知道了当前电梯运行的方向。电梯在两个楼层之间快速移动，但它应该能提前减速停在目的层。为了保证电梯系统的安全，在任何不安全的情况下，紧急制动就会被促发，电梯被强制停止。

4.2 用例图

所有系统和人或为了某种目的使用系统的自动化角色交互。人和角色都希望系统的行为是可预知的。在 UML 中，一个用例对一个系统行为或系统的一部分建模，是对一组行为序列的描述，系统对一个角色产生的可见的结果[2]。

用例图对系统的**动态设计视图**建模。它是一个系统行为、一个子系统或一个类建模的中心。用例图描述一组用例、角色和它们的关系。用例图的主要内容是：

• 用例

• 角色

• 依赖、泛化和关联关系

按照我们课程的需求文档，电梯系统的用例图如图 1 所示：

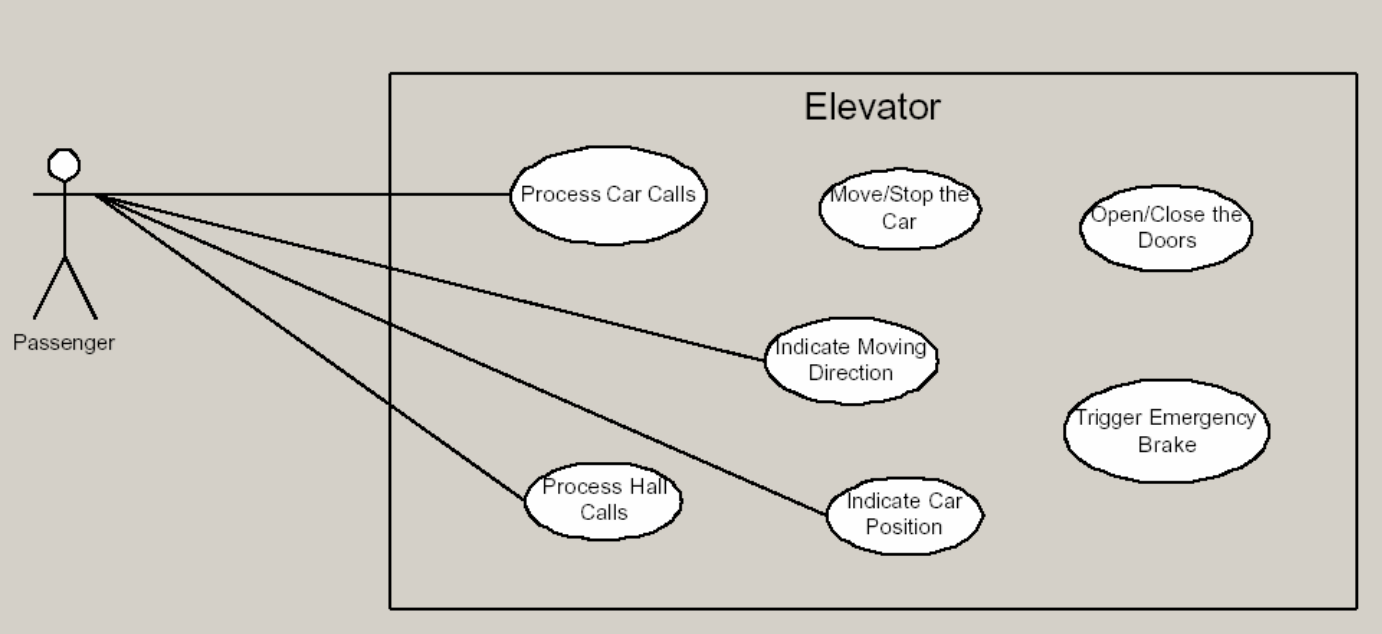


图 1：电梯系统的用例图

共有七个用例基于我们课程的需求文档，如图 1 所示：

• 处理电梯呼叫：这个用例包括几个场景，本文接下来的部分兼作详细描述。这些场景有电梯接受乘客的呼叫、电梯呼叫按钮的亮灭、系统控制部分电梯呼叫按钮信息的更新等等。

• 处理楼层呼叫：和处理电梯呼叫类似，这个用例包括电梯接受乘客的楼层选择、楼层按钮的亮灭和系统控制部分楼层按钮信息的更新等等。

• 电梯的动/停：这是一台电梯的主要功能，详细的动作包括驱动速度的改变，停止的判定，电梯的运动方向驱动。

• 标识运行方向：电梯应该有这种机制，即让乘客知道电梯目前的运动方向，决定是否进电梯。

• 标识电梯位置：类似的，电梯应该让乘客知道他/她的目的层是否到达，决定是否离开电梯。

• 开/关门：乘客进出电梯，电梯应该开关门。这个用例应该包括当电梯正关闭时乘客想进入，乘客可以使电梯门反转。

• 触发紧急制动器：电梯有安全机制确定一个不安全的状态不是瞬时产生的。

电梯系统的唯一角色就是乘客，乘客和系统交互完成任务。乘客通过呼叫电梯和楼层与电梯系统交互。

乘客通过观察电梯移动方向和电梯位置指示器决定是否进/出电梯。因此用例图表明角色和处理电梯呼叫、处理楼层呼叫、标识运行方向和标识电梯位置四个用例有关。

4.3 类图

类图是面向对象系统中应用最广的图，它对系统进行静态建模。静态图主要描述系统的功能需求-系统给最终用户提供的服务。从我们过去的实践经验来看，当用类图完成系统建模后会得到许多乐趣。系统类图的不同视图将在本文后面的章节重点讨论。

类图描述一组类、接口和协作，及它们的关系。类图包括整个系统的描述，如系统的结构和细节，还有类的属性和操作等细节。一个类图的基本内容有：

• 类

• 接口

• 协作

• 依赖、泛化和关联关系

• 节点和约束

接下来的小节，将详细描述和分析三组类图。

在 4.4 节中介绍每一张类图对应的顺序图。

4.3.1 类图-对象构造视图

从 4.2 节的电梯系统用例和系统需求，我们直观地得到了如图 2 所示的类图。

从图 2 中类的描述我们可得出系统组成的一些概念。我们不深入细化类成分，如每个类属性和操作，这超出了我们当前视图的范围。基于此，我们从系统对象组成的角度，建立了类图。

• **电梯控制器**：电梯系统的核心控制对象。和系统中所有其他对象通信并控制它们。

• **门**：系统中有两扇门，“上帝”对象-电梯控制器-命令门打开和关闭，这和用例中的描述相对应。

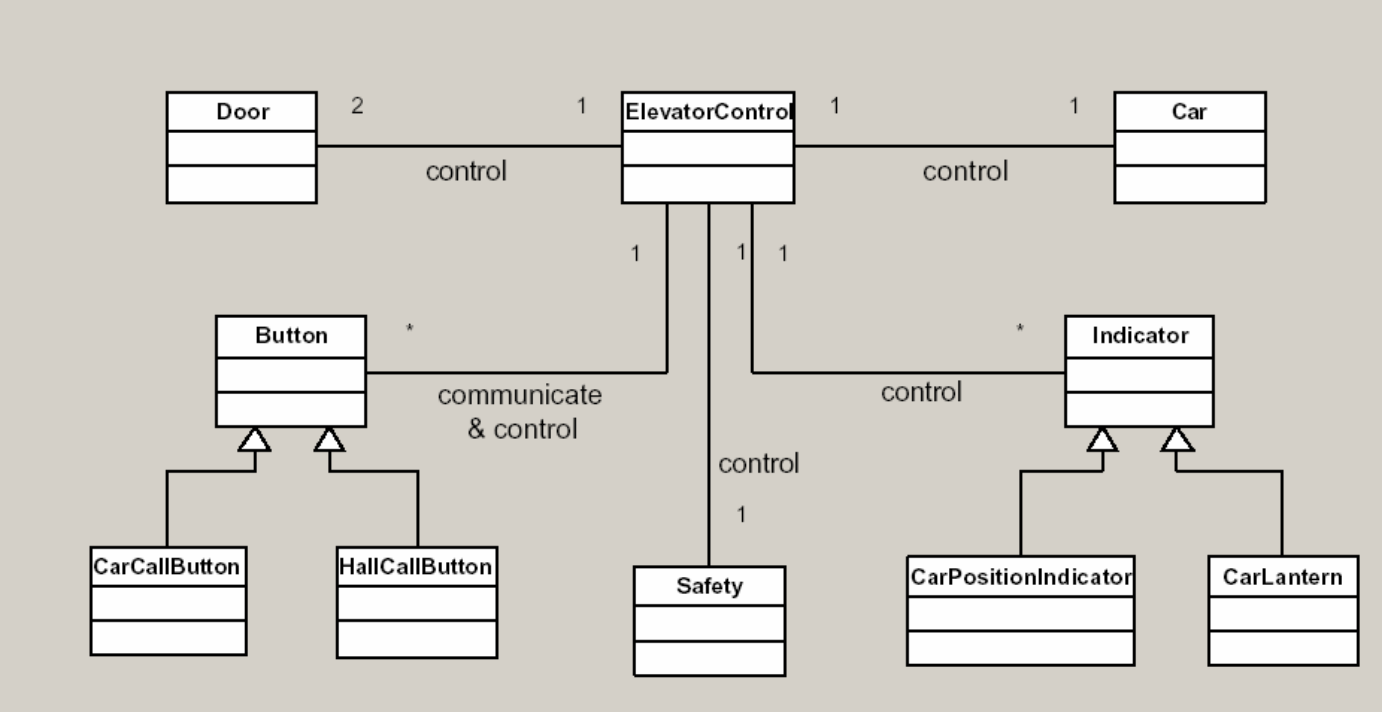


图 2：类图-对象构造视图

• 电梯：电梯在控制下上升和下降（用不同的速度），需要时可以停下。

• 按钮：电梯控制器类（ElevatorControl）也控制按钮类，按钮类生成两个子类电梯呼叫按钮类和楼层呼叫按钮类（CarCallButton and HallCallButton）。控制对象和按钮对象通信，得到按钮是否被按下，反过来控制按钮灯的发光。

• 指示器：系统有两类指示器，电梯位置指示器（CarPositionIndicator）和电梯方向指示器（CarDirectionIndicator）（例如：电梯灯）。指示器提供电梯的当前位置和移动方向。

• 安全装置：依照需求文档中紧急制动器的定义，任何紧急情况时，电梯控制器触发安全装置。

这个版本的类图是直接从4.2节中用例图的描述得来的，这个视图中的类覆盖了系统所有的功能。我们用电梯类和电梯控制器类（ElevatorControl）移动或停止电梯；用门类开门或关门；用指示器类让乘客知道电梯的位置和方向；乘客用按钮类来完成呼叫电梯或选择楼层；我们用安全装置类来满足系统紧急制动的要求。所有的类和中心控制器类都有接口，而中心控制器类的任务是控制所有类的动作。这个类图帮助我们从对象划分和系统功能的

的角度理解系统的基本设计。

当我们试图深入我们电梯控制系统的设计，找到我们自己的详细设计方法，从这个类图开始得到我们系统的好的实现时，问题就出现了。在本文中，用已有的架构设计系统和完成 UML 文档的顺序是颠倒的。我们从老师那里“继承”一个设计好的电梯系统，不是首先用 UML 进行系统设计，而且在用 UML 之前，手中已经有了软件的部分设计。正是由于上述的原因，在遇到真正的难题之前，我们就知道这个类图不是一个完美的最终设计。

但在其他情况下，设计者迟早会在以后发现这个设计对开发阶段不适合，这几乎是肯定的。基于前面的讨论，每一个组件（软件/硬件）是由一个处理器来控制的，假如我们的系统是一个普通的中央控制的系统，则我们现在类图的方案可能不会导致将来的设计缺陷。但是分布嵌入系统的特征决定了电梯系统的类图仅仅从对象的角度来设计是不够的。

分析手中已有的类图，我们未来软件的潜在缺陷如下。如果不能找到更好的方案，软件的设计会失败。

• 控制对象负担过重：从前面的分析我们可以发现作为控制中心，电梯控制器对象（ElevatorControl）必须和其他所有的对象交互。所有的计算和控制任务必须由这个对象完成。

• 其它一些对象的空闲：电梯控制器（ElevatorControl）不停的工作，其它的一些对象，如按钮和指示器象系统的界面一样，更糟的是象门和电梯等对象竟然是系统的一部分-如“硬件“。从软件控制的角度来看，他们在系统的范围之外。

• 计算资源的争夺：当超过一个对象想同时得到中央控制对象的控制时，这些对象竞争控制器有限的计算资源是不可避免的，一些对象不能及时得到维持正常运行的控制消息，而这在实时系统中会导致致命的缺陷。

• 整个系统的低效率：即使控制器的计算资源足够快/多，能处理每一个控制请求并及时做出反应，中央控制对实时系统（如电梯）仍然不是一个有效的方案。

4.3.2 类图-软件架构视图

前面的分析和教学项目的软件架构类图被模拟证明非常适合电梯控制系统，并从从这个角度得到类图。

类图提供怎样设计和实现控制系统的方法。真实电梯控制系统的软件架构精确的反映在这个图表中。除了 Dispatcher，所有其它的控制对象都是从超类电梯控制器继承而来。这些控制对象共享电梯控制器的一些属性，而且有用于其控制对象的自己属性和方法。被控制对象所控制的对象被定以为环境对象。虽然这些环境对象存在于电梯系统，但不属于软件控制系统。下一节我们将从系统架构的角度详细讨论这些不能控制的对象。

• **门控制器**（DoorControl）控制**门马达**的动作，一个电梯的两个门马达都是由一个门控制对象控制的。门马达能够发出打开门，关门或门反向运动的命令。

• **驱动控制器**（DriveControl）控制电梯**驱动**，它将电梯上下移动在需要时停下，是主要的马达。

• **指示灯控制器**（LanternControl）有两个，每一个控制一个电梯灯，标示电梯的当前运动方向。

• **楼道按键控制器**（HallButtonControl）每一层有两个，一个控制上升一个控制下降。楼道按钮控制器处理楼道按键的按下及给楼道呼叫灯反馈。

• **电梯按钮控制器**（CarButtonControl）用于每一层都在电梯里。电梯按钮控制器接受电梯呼叫按钮（CarCallButton）的呼叫，而且控制相应的电梯呼叫灯的开关。

• **电梯位置指示器**（CarPositionIndicator）赋值给电梯位置指示器，乘客可以知道电梯当前的位置。

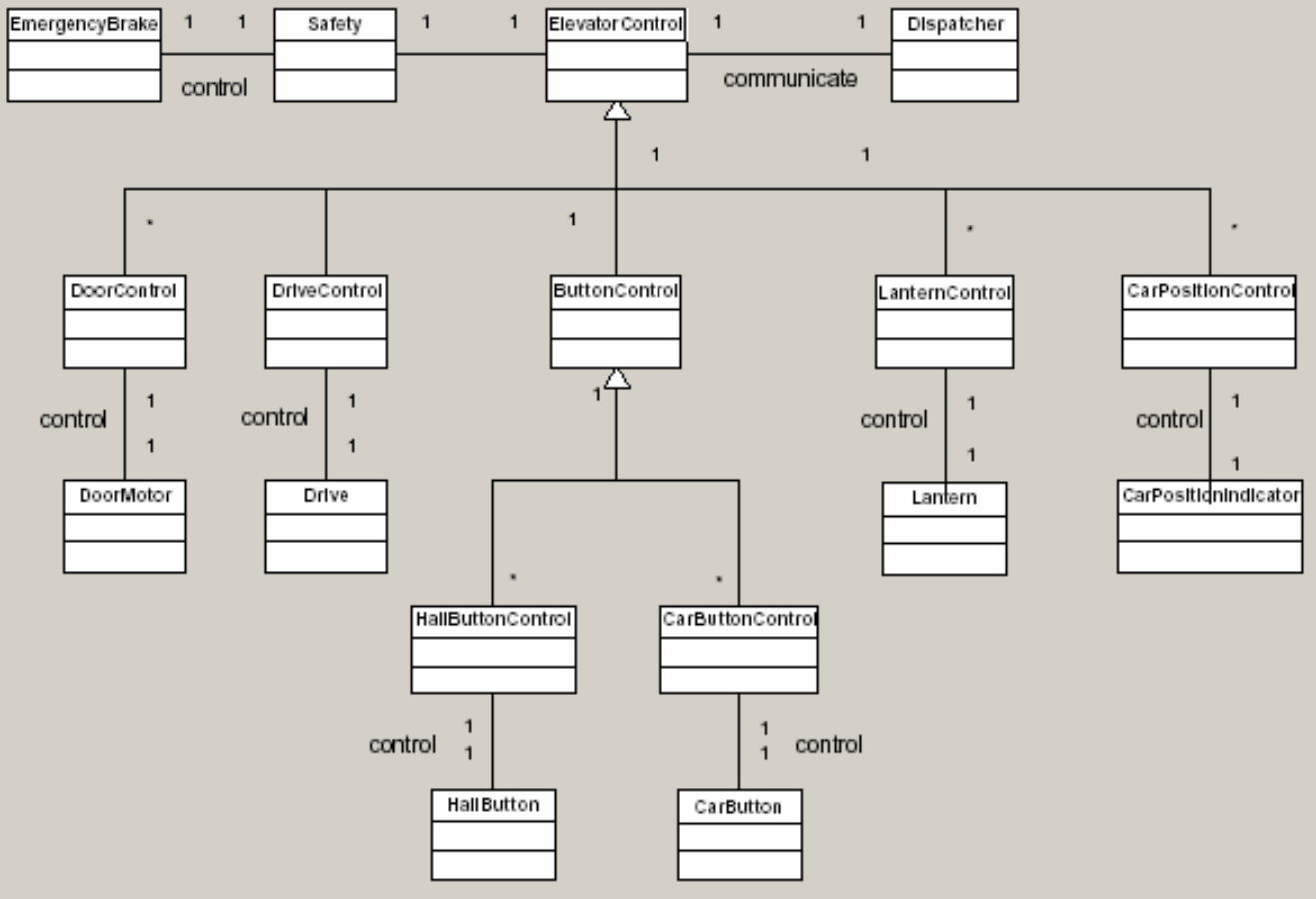


图 3：类图——软件架构图

在系统中有两个非控制对象：

• Dispatcher 不控制实际的电梯组件，但它在软件系统中是重要的。每一个电梯有一个 Dispatcher，主要功能是计算电梯的移动方向、移动目的地以及保持门的打开时间。它和系统中除灯控制器以外的几乎所有控制对象交互。

• **安全装置**也是一个环境对象，它不属于控制软件，但是系统的重要部分。在真实世界中，如果一台电梯的紧急制动被触发，则安全装置动作变化。但在我们的模拟系统中，只显示一些信息。

在我们的系统中，乘客也作为一个环境对象来建模。乘客和楼层呼叫按钮、电梯呼叫按钮交互，使门反转，观察电梯的方向和位置等。为了简单，乘客对象没有在图 3 中列出（不像其他的环境对象）。

软件的类图解决了前一节提出的大多数问题。控制任务被分配到几个控制对象中，每个控制一个或两个环境对象，都没有负担过重或空闲。不需要竞争中心控制器的计算资源，因为由控制器控制其受控对象。

但是从这个类图引发的，关于我们系统实现细节问题如下：

• 控制对象如何控制环境对象？

• 一个对象如何从其他的对象得到必需的信息？

• 如何对网络建模

从系统架构的角度，这些问题必须回答。

4.3.3 类图——系统架构图

为了回答上一节提出的问题，类图要加入网络、传感器/传动装置进行细化，以对真实系统的架构进行建模。从这一点来看，系统的类图和普通 UML 图的类图不完全一样。但类图是描述系统静态结构的一种有效的途径，为什么不用它来帮助更好的表达系统架构？

类图中的各个部分如图 4，被分成如下 8 类：

控制类

• 前一节我们对系统中的控制对象作了大量的陈述。从系统架构来看控制对象包括电梯位置控制、电梯按钮控制、灯控制、门控制、驱动控制、楼层按钮控制和 Dispatcher（CarPositionControl，CarButtonControl，LanternControl，DoorControl，DriveControl，HallButtonControl and Dispatcher.）。

• 所有的控制对象连接到网络，从网络得到输入并发送输出消息到网络给其他的对象。

• 控制对象控制一个和传感器及传动装置相连的系统实体（如门和按钮），从传感器得到信息，并发送反馈到传动装置执行控制功能。

网络

• 所有的控制对象和通信网络连接，在图的中间用网络类来建模。网络是（编注：此处缺 1 页，抱歉）

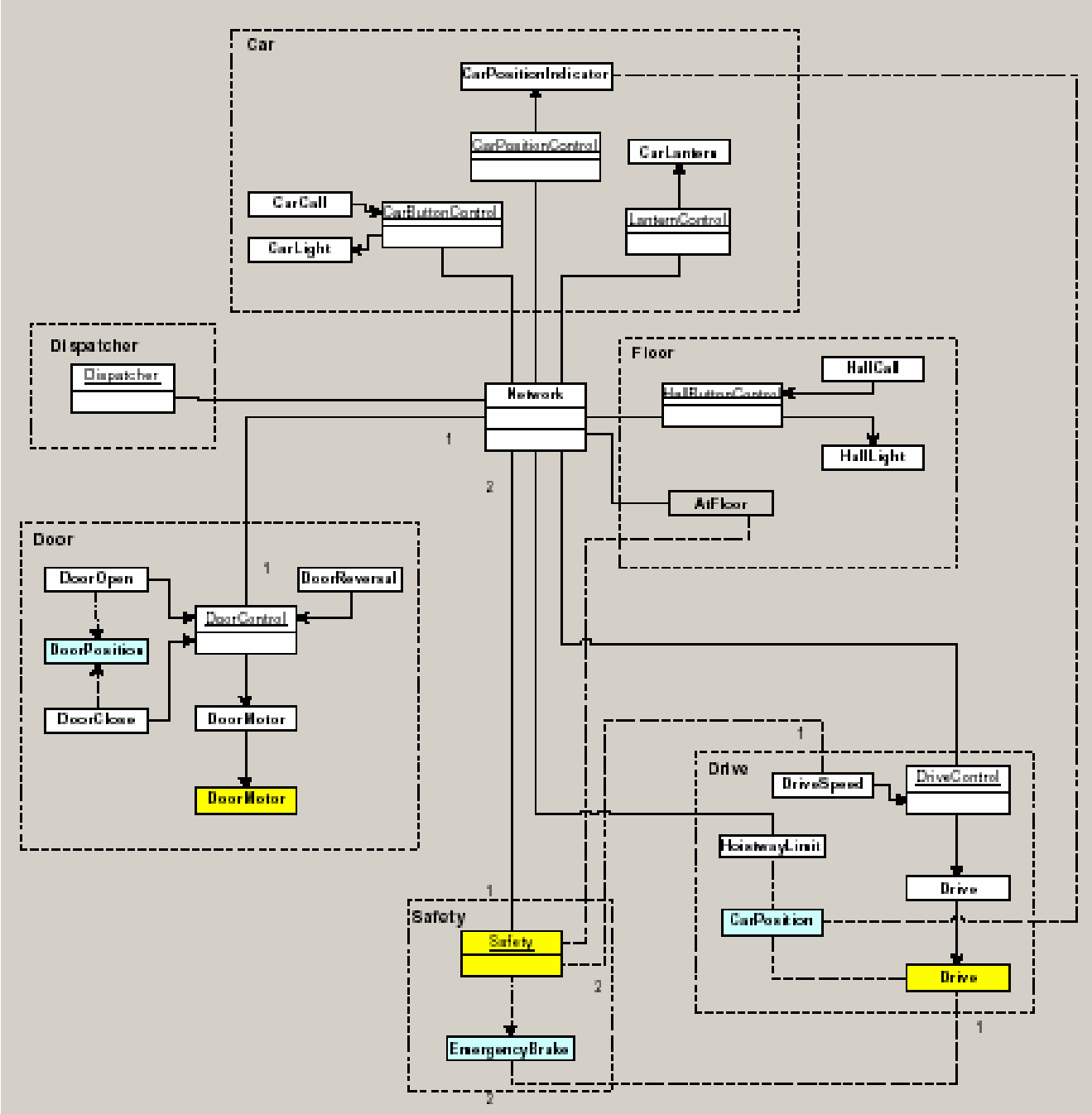


图 4：类图——软件架构图

系统传感器

• 系统值是用于控制系统的。在类图中系统传感器用一个箭头和系统控制对象连接。

•类图中的系统传感器包括 AtFloor、电梯呼叫器、关门、开门、门反转、楼层呼叫器和驱动(AtFloor，CarCall，DoorClosed，DoorOpen，DoorReversal，HallCall，and DriveSpeed.）。

• 除了 AtFloor，所有系统传感器通过物理网络接口和他们的控制对象相连，控制对象从传感器得到消息，通过网络发出正确的控制消息。

仅环境传感器

• 系统中有两个仅环境传感器：，门的位置和电梯的位置（DoorPosition and CarPosition）。他们用虚线和系统控制对象相连。

• 仅环境传感器是“伪随机传感器”，不能被控制系统访问，但可用于模拟。

系统制动器

• 在类图中，系统制动器用从控制对象出发的箭头和控制对象相连。

• 类图中的系统制动器包括门马达、电梯灯、电梯位置指示器、楼层灯和驱动（DoorMotor，CarLantern，CarLight，CarPositionIndicator，HallLight，and Drive）。

仅环境制动器

• 类图中紧急制动是仅环境制动器，使用虚线和安全装置对象相连。

环境对象

• 在类图中用阴影列出的安全装置、驱动和门马达是环境对象。

• 环境对象通过操作激励者间接访问控制系统。

对象组

• 对象组包括电梯、门、Dispatcher、驱动、层，和安全装置，每一个由一个虚方框包围。

• 系统架构中对象组的关系如图 5。

• 看一看前面段落，可以对图 5 和图 2 做出比较。

我们发现图 5 中的对象结构向更加分布作了改进。与图 2 中的实现不同（，图 2 中，以一个中心控制对象处理系统中的所有控制任务），每一个(组)对象有自己的功能范围，和系统中的其他对象协作。我们加入环境类“乘客”得到的图 5 中的类图的进化版。

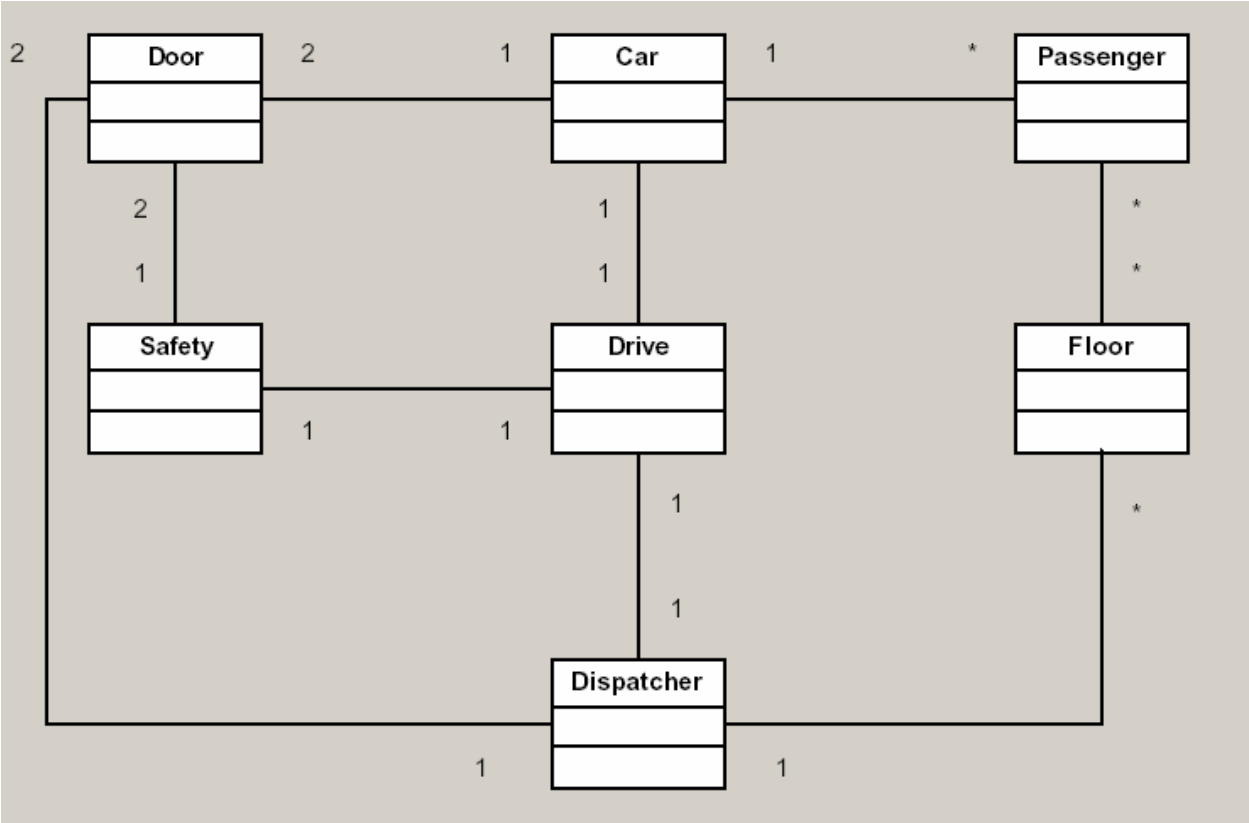


图 5：类图-修正的对象构造图

4.4 静态结构小结

4.3 节中，三个不同的类图以进化的方式给出了电梯系统的不同视图。每个视图描述系统的一个方面，给出了系统集成时系统设计的全局理解。

从对象构件的角度，类图描述了对象结构问题的解决方案。通过描述一组对象的通信和协作实现一个功能。对象通过发送消息和别的对象通信。相同功能的对象归为一个类。

类图通过捕获系统的主要功能而得到，给出了系统的框架。从软件架构的角度，捕获了更多设计和实现的细节。

从这个类中得到的类图，构划出了软件的大部分设计。

系统结构视图提供软件和整个系统结构最复杂的也是最优雅的描述。和通常的软件系统相比，在分布式嵌入系统中了解系统组件如何协同工作是非常重要的。毕竟，每个类图仅仅是一个系统的静态设计视图的一个图型表示。

单个类图不能捕获一个系统设计视图的全部内容。

一个系统的类图结合起来描述系统的全部静态设计，分开只是描述一个方面。

5 系统动态建模

UML 提供顺序图和协作图对系统动态建模。在本文中，只给出了电梯系统的顺序图，协作图能由顺序图很容易得到。

基于这学期课程项目的设计，电梯系统的状态图也在这部分给出。从我们的设计练习，给出了一些关于如何从需求到设计的经验方法。

5.1 顺序图

顺序图是一种交互图，描述一组对象的交互和它们的关系。(另一种交互图是协作图)。顺序图的目的是在一个有时间关系的视图中描述对象之间的消息序列。对于单个(部分)用例，典型的顺序图范围包括所有的消息相互作用。每个用例可以有多个顺序图，而每个用例场景有一个顺序图。

状态图通常包括:

- 对象
- 联结
- 消息
- 响应时间 （ 在实时系统中尤其有用）

垂直的 "生命线" 表明对象间的联结。消息在对象生命线之间流动。UML 支持在顺序图中标明响应时间，描述一个实时系统的性能需求。时间流从上到下。

在以下的章节的顺序图中，对象从软件结构角度的类图得来。那样做的原因是我们不想停在对象构造视图上，因为对象构造视图中对象的功能是模糊和不充分的； 也不想停在系统结构视图中，因为许多技术上的细节阻碍了对对象的相互作用的快速理解。

因为乘客发出一些消息，在一些顺序图中乘客似乎是系统的一个对象。

5.1.1 用例 1 - 处理门厅呼叫

在这个用例中有二个场景:

当乘客按门厅呼叫按钮请求一个门厅呼叫服务时，一个场景是电梯向与乘客的目的地相同的方向移动，另一个是反向移动。

这二个场景共享一个顺序图，唯一的不同是乘客进入之前的推进的时间，也就是(x sec) 在图中反映电梯的运行时间。

场景 1.1 门厅呼叫服务 - 电梯的移动方向和乘客的目的地相同。

场景 1.2 门厅呼叫服务 - 电梯向乘客的目的地的反方向移动。

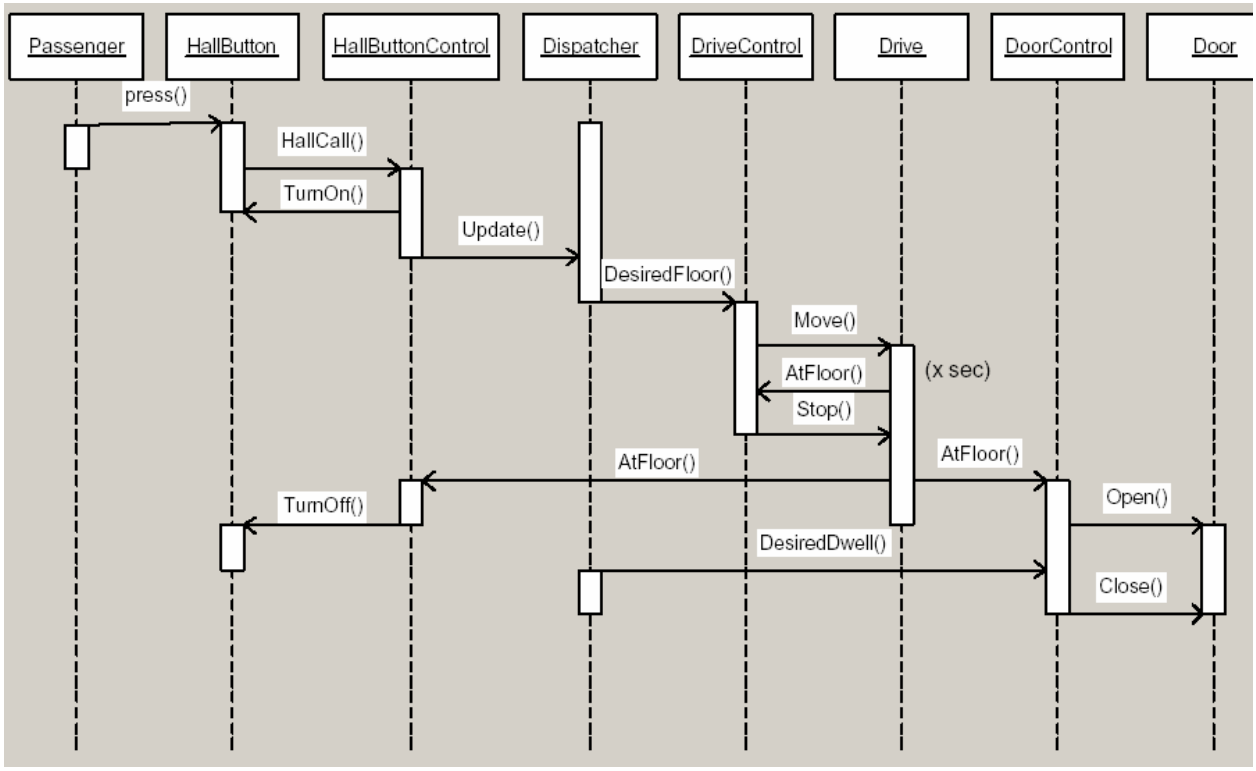


图 6: 场景 1.1&1.2- 门厅呼叫服务

5.1.2 用例 2 – 处理电梯呼叫

这个用例有二个场景:乘客进入电梯和按电梯呼叫按钮。乘客可能想要去楼上或楼下，与电梯当前的移动方向有关。当电梯经过乘客或当在附近的楼层徘徊时，乘客可以到达目的层。这二个场景可以共享同一个用例。

场景 2.1 电梯呼叫服务 – 电梯向乘客的目的相同的方向移动。

场景 2.2 电梯呼叫服务 – 电梯向乘客的目的地的反方向移动。

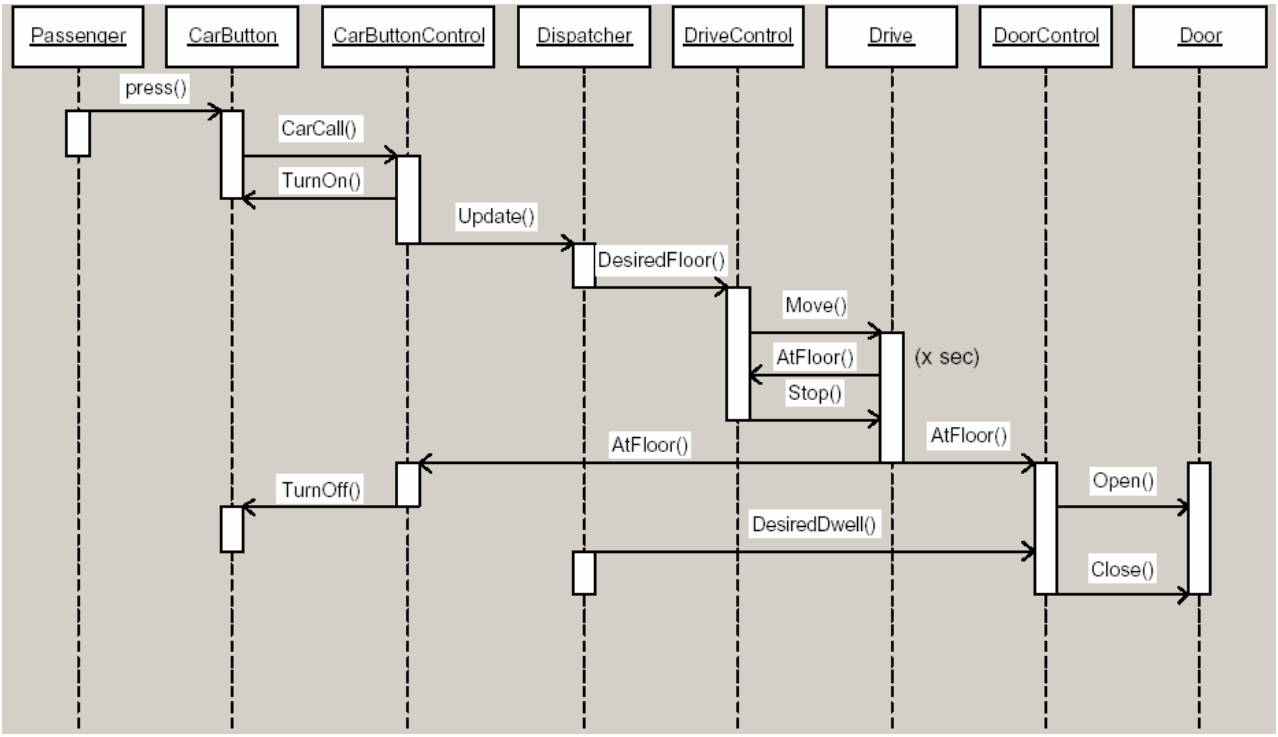


图 7: 场景 2.1&2.2- 电梯呼叫服务

5.1.3 用例 3 – 移动/ 停止电梯

这个用例有二个场景:

场景 3.1&3.2 移动电梯 – 命令停止状态的电梯开始移动。移动方向和电梯的目的层由调度器给出。电梯的移动从慢速到快速。场景 3.1 向上移动而 场景 3.2 向下移动。

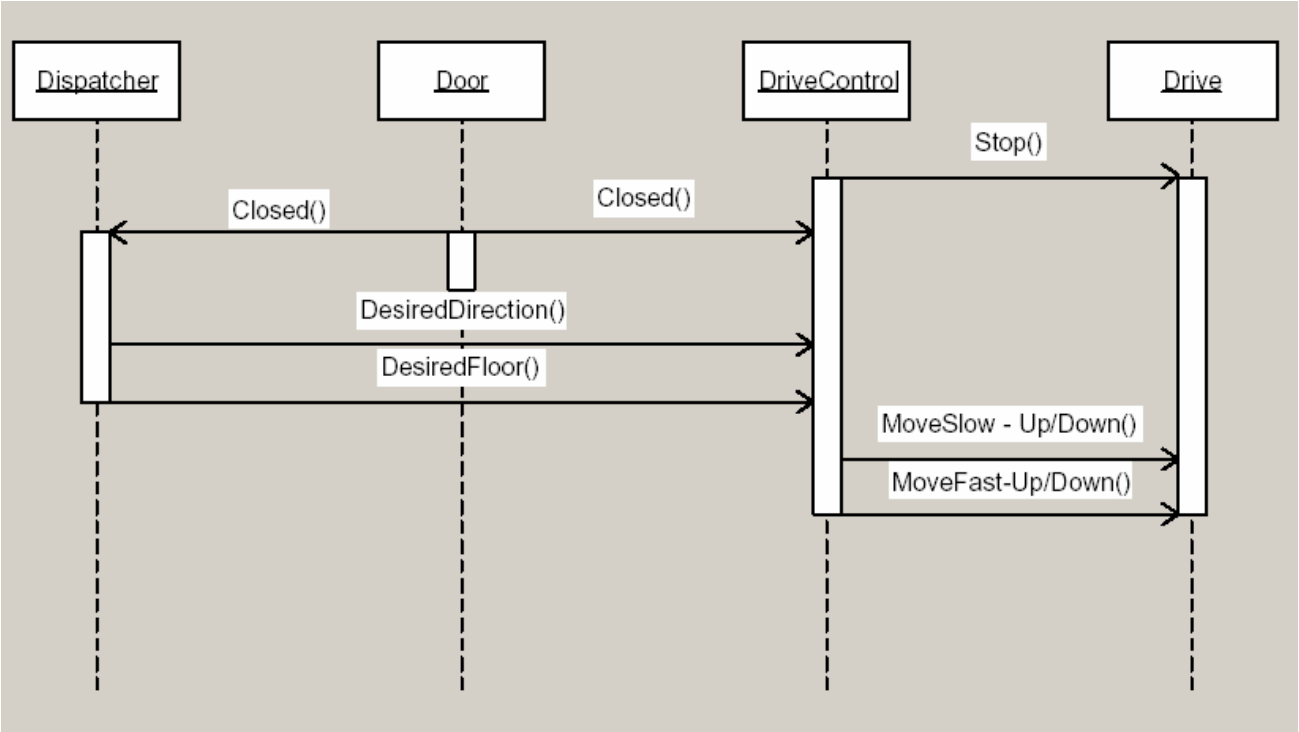


图 8:场景 3.1&3.2-电梯由停止到慢速移动到快速移动

场景 3.3&3.4 停止电梯 - 当电梯快要到达目的的层时，它应该被命令减速，最后停在目的层。

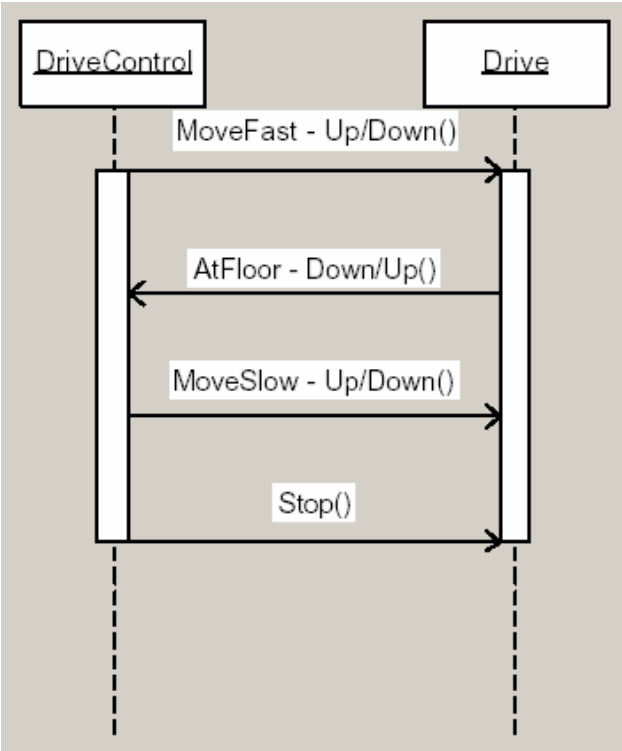


图 9: 场景 3.3&3.4-电梯由停止到慢速移动到快速移动

5.1.4 用例 4 – 指示电梯位置

这个用例有二个场景，分享一个顺序图:

场景 4.1 指示电梯位置 – 每当电梯的门打开，命令 CarPositionIndicator 亮标识当前的电梯位置。

场景 4.2 完成指示电梯位置 – 当门关闭的时候，命令 CarPositionIndicator 亮标识目的层。

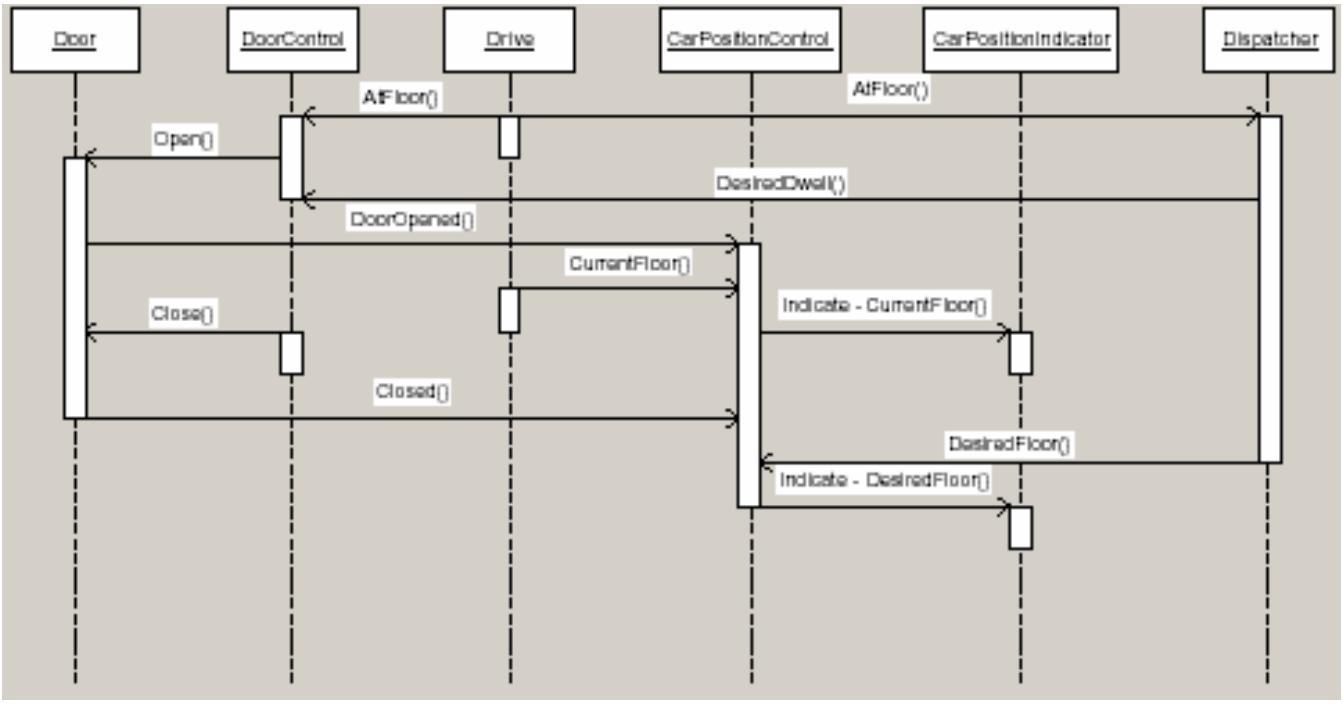


图 10:场景 4.1&4.2-标识电梯位置

5.1.5 用例 5 – 标示移动方向

这个用例有二个场景，分享一个顺序图：场景 5.1 指示移动方向（向上的） – 电梯的门打开及电梯的目的方向是向上的，向上的 CarLantern 亮。门关闭时，CarLantern 熄灭。

场景 5.2 指示移动方向（下） – 当电梯的门打开及电梯的目的方向向下，下 CarLantern 亮。当门关闭，CarLantern 熄灭。

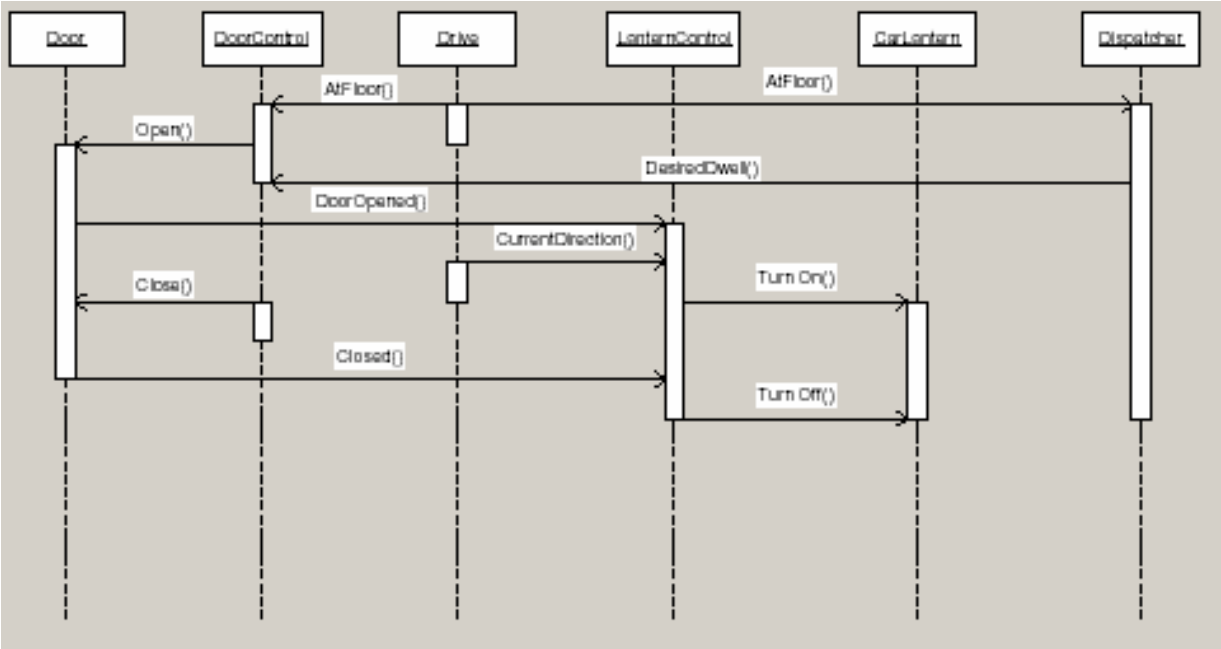


图 11：场景 5.1&5.2- 指示移动方向

5.1.6 用例 6 - 开/关 门

这个用例有三个场景：

场景 6.1 开门 - 当电梯停在楼层，门应该开启一段时间(DesiredDwell)，乘客才能进入电梯。

场景 6.2 关门 - 在一特定的时段 (Desiredperiod) - 门应该关闭，以便电梯能移动到下个目的地。

征稿

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

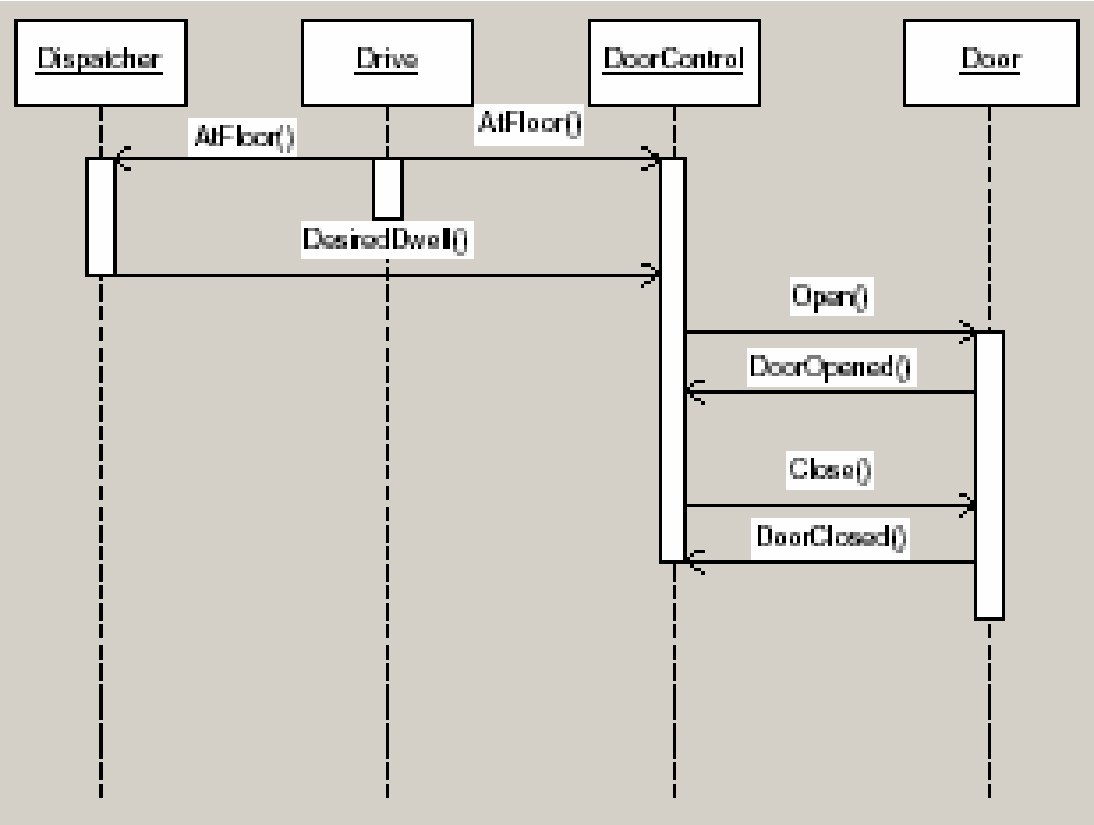


图 12:场景 6.1&6.2-开关门

场景 6.3 门逆向 - 当门正在关但不完全关闭，如果有乘客

想要进入电梯，门应该再打开一段时间，然后再关闭。

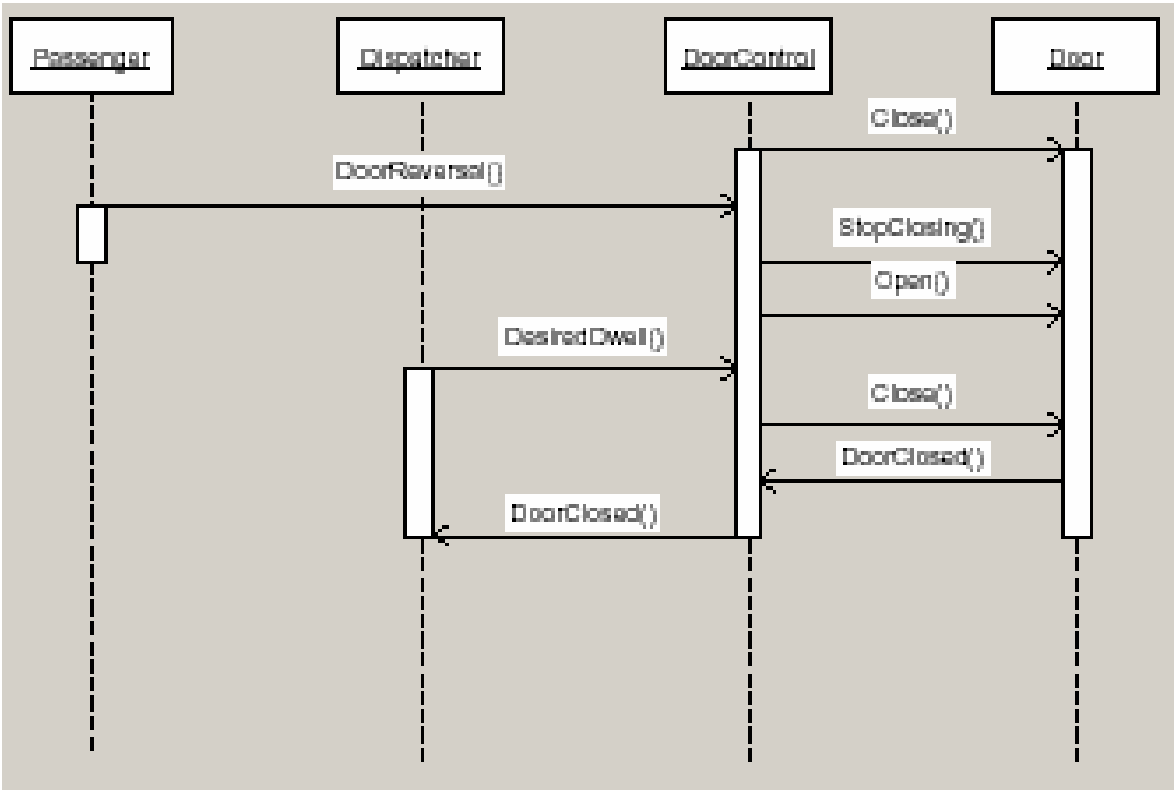


图 13：场景 6.3- 门逆向

5.1.7 用例 7 - 触发紧急制动

这个用例有五个场景：场景 7.1 紧急制动 1 - 如果电梯停止但是它没有停止在目的层，紧急制动被触发。

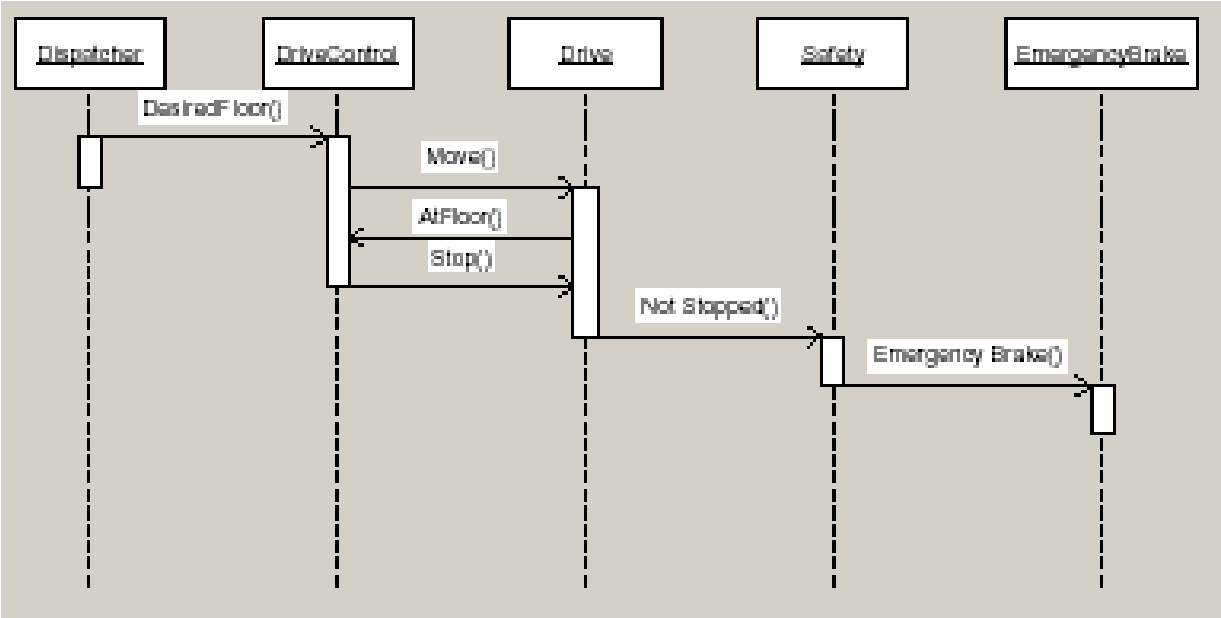


图 14：场景 7.1- 紧急制动 - 电梯在目的层不停止

场景 7.2 紧急制动 2 －如果电梯被命令移动但是它没有动，则紧急制动被触发。

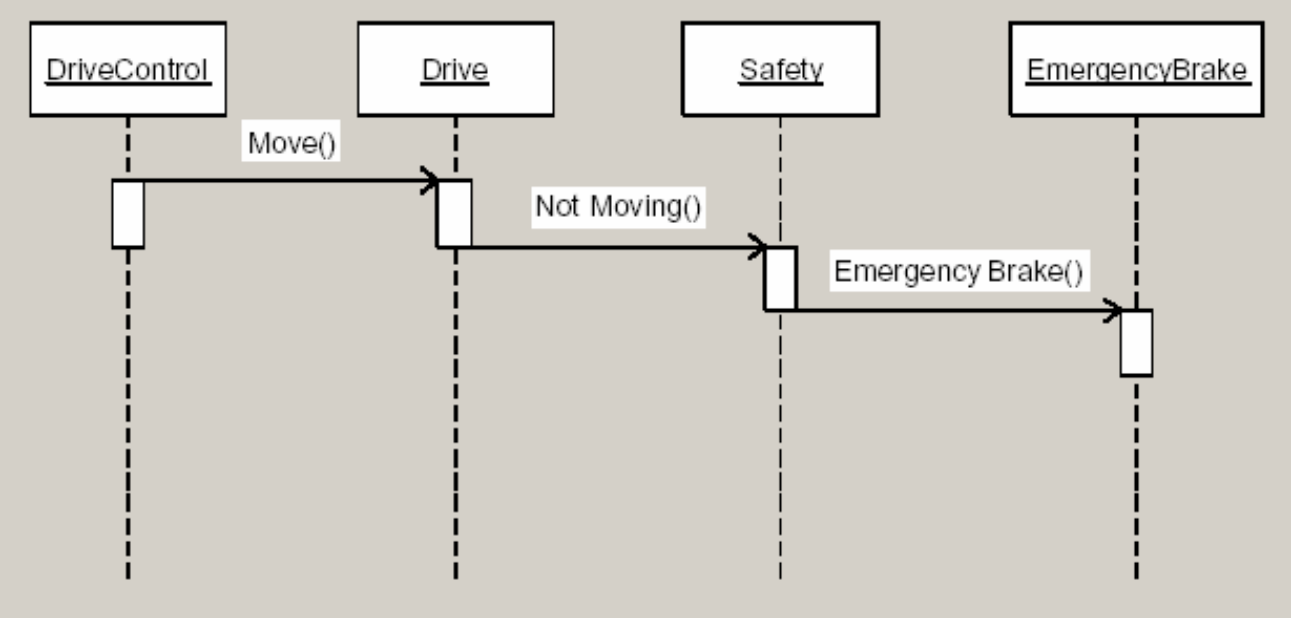


图 15：场景 7.2- 紧急制动 － 电梯不动

场景 7.3 紧急制动 3 －当电梯停止在某一楼层时，如果命令门打开，但是门不开，则紧急制动将触发。

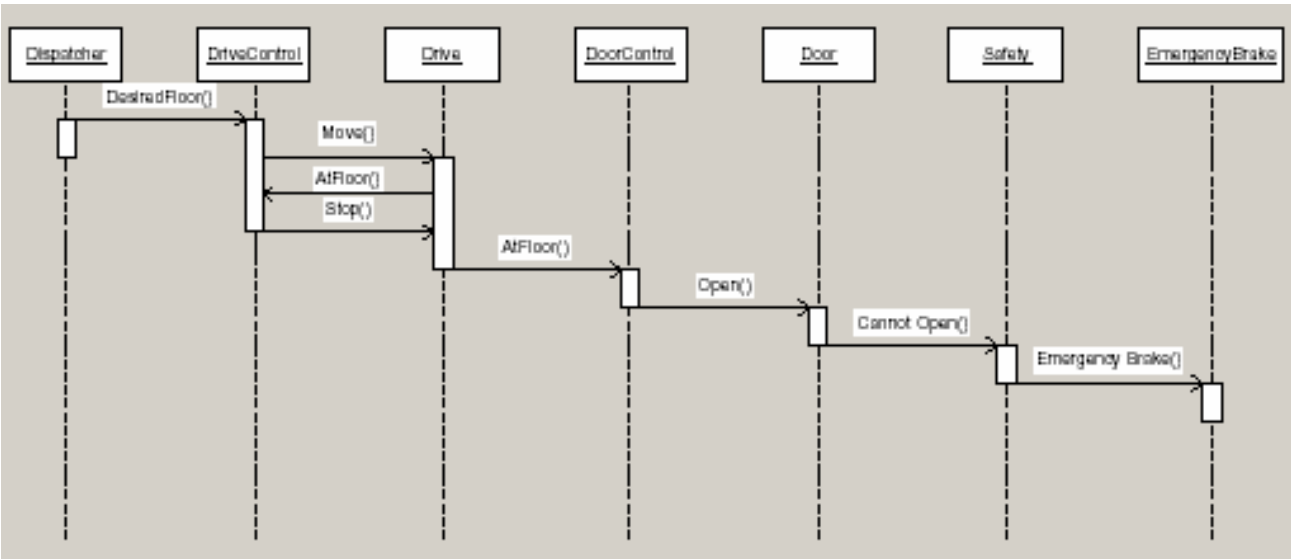


图 16：场景 7.3- 紧急制动 － 门将不开启当电梯停止在目的层

场景 7.4 紧急制动 4 －如果电梯移动时门打开，紧急制动将触发。

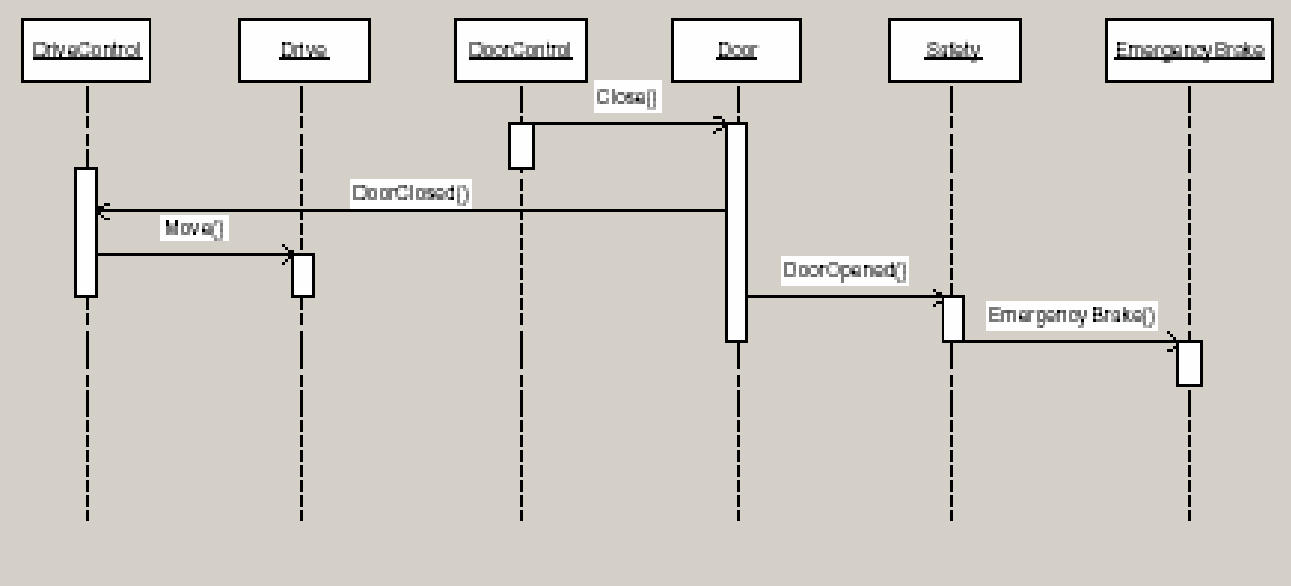


图 17：场景 7.4- 紧急制动 -当电梯门是打开的时候移动

场景 7.5 紧急制动 5 - 如果到达升高界限时电梯继续上升，紧急制动将触发。

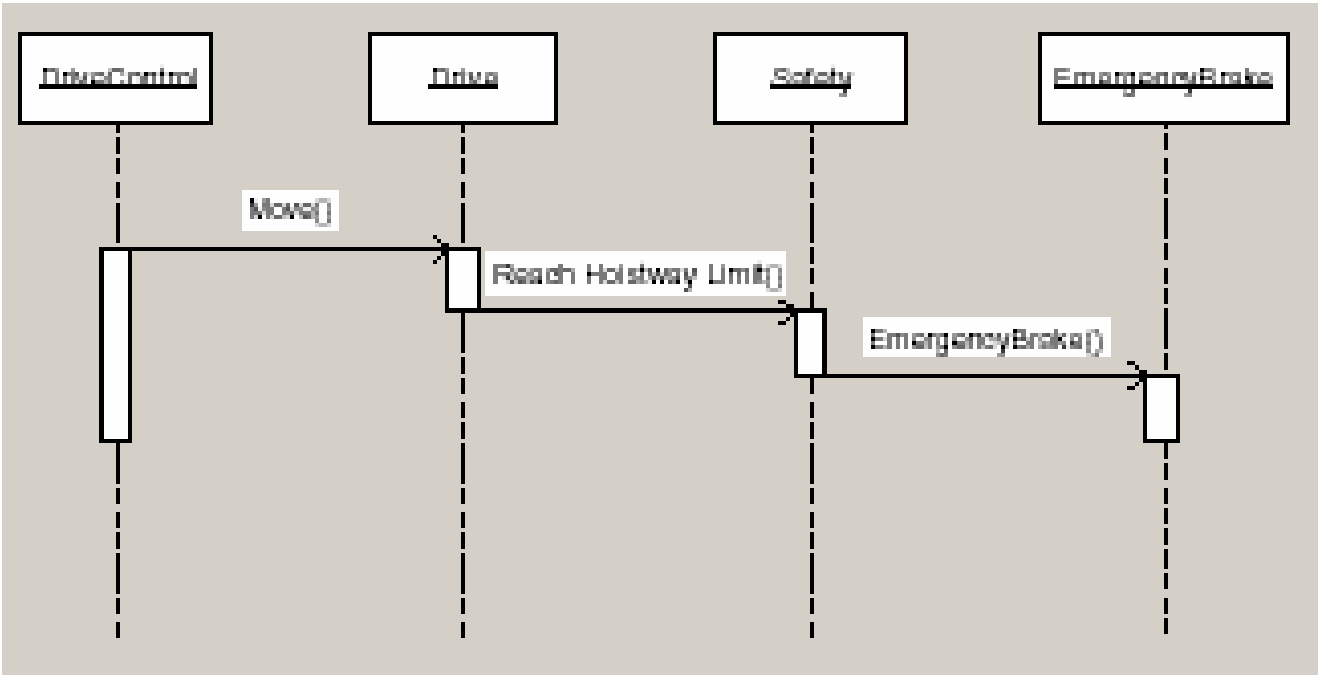


图 18:场景 7.5- 紧急制动 -到达升高界限时电梯继续上升

5.2 状态图

一张状态图表明一种状态机。

通常状态图中状态机针对对象行为建模。对象的行为通过响应它的上下文之外分发来的事件最好地表征。对象有一个清楚的生命期，它的过去影响了当前的行为。状态图对通过正向工程和逆向工程构造可运行的系统很重要。这是承认在从需求到状态图的设计过程存在一个鸿沟，在从需求画状态图没有足够的方法可用。在本节中，介绍我们设计电梯系统状态图一些实际的方法。这些方法可能不是如何从需求文档画出状态图严格的规则或指令，但是在练习中他们是有帮助的。

5.2.1 DoorControl 的状态图

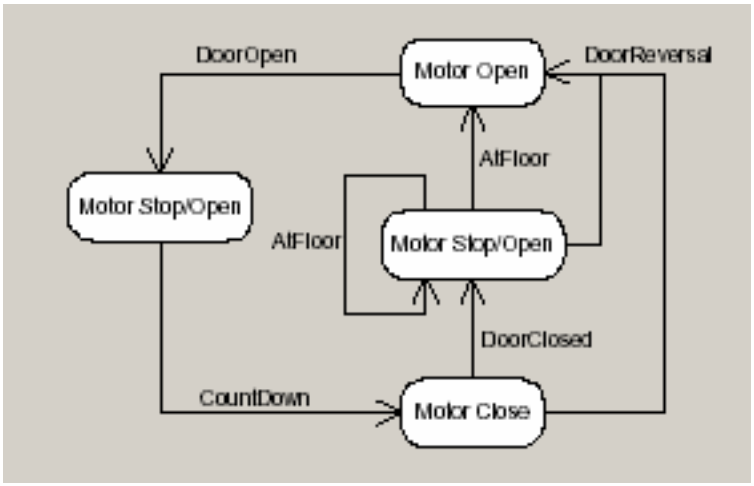


图 19：DoorControl 的状态图

5.2.2 DriveControl 的状态图

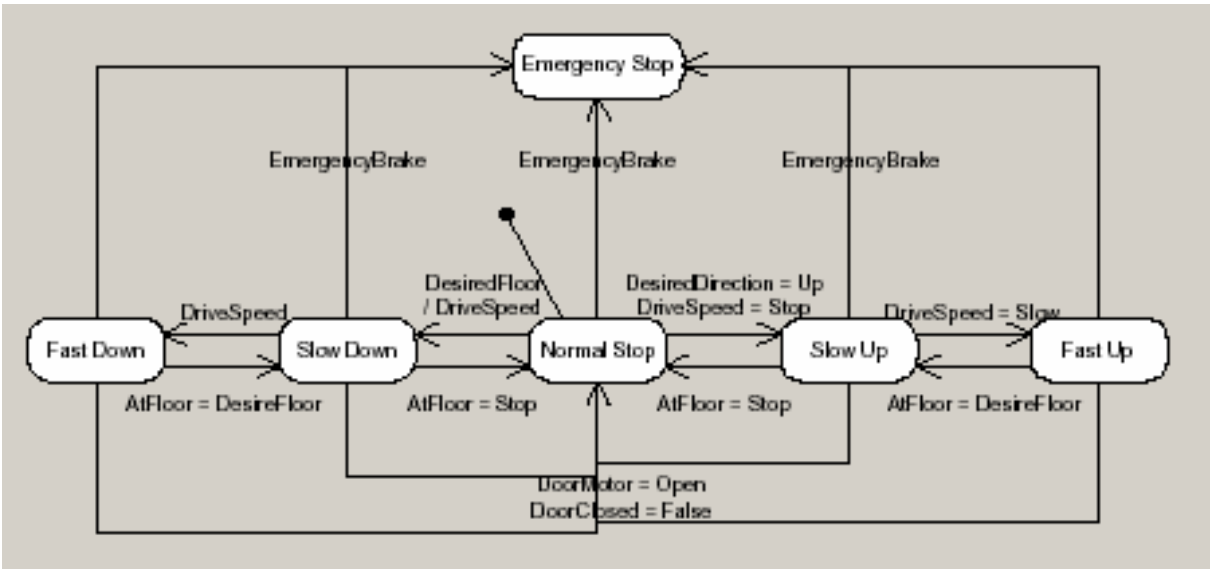


图 20：DriveControl 的状态图

5.2.3 LanternControl 的状态图

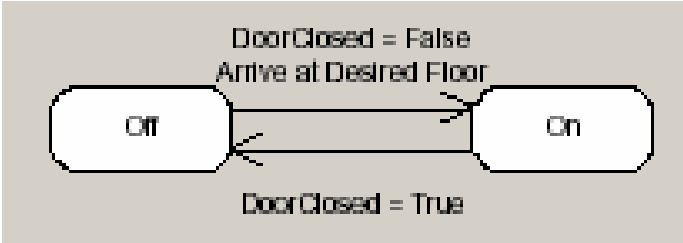


图 21: LanternControl 的状态图

5.2.4 HallButtonControl 的状态图

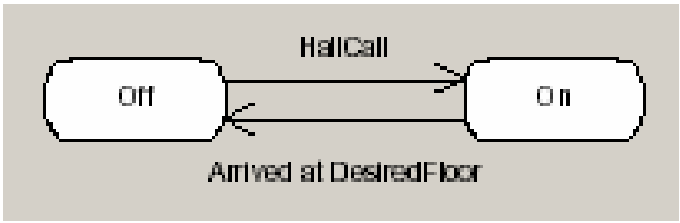


图 22: HallButtonControl 的状态图

5.2.5 CarButtonControl 的状态图

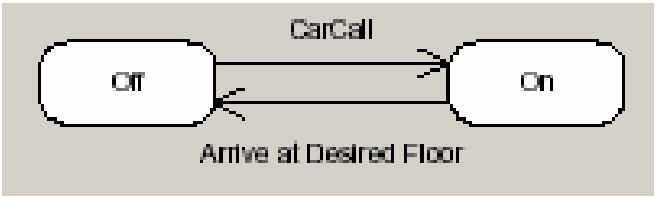


图 23: CarButtonControl 的状态图

5.2.6 CarPositionControl 的状态图

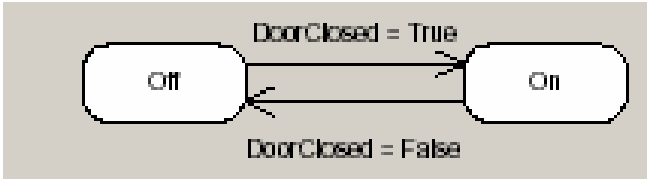


图 24: CarPositionControl 的状态图

5.2.7 Dispatcher 的状态图

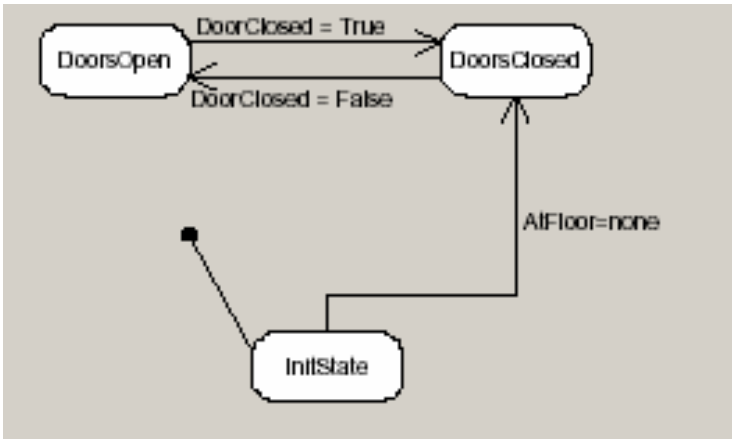


图 25: Dispatcher 的状态图

5.3 填补从需求到状态图鸿沟的实用方法

状态图能对类的行为，一个用例，或系统整体建模。在本文中，状态图被用于每个对象的行为建模。在系统后面的阶段，如实施阶段和测试阶段，每个对象的状态机都将用到。

UML 中没有提供详细的方法，即该如何从需求文档或 UML 图表如类图画出状态图。我尝试在这里从课程上的项目经验总结一些从需求文档画出状态图实用的方法，如下:

第 1 步:

如果你想画出每个对象的状态图，应该完成对象结构和系统架构的分析。在我们的课程项目中，这部份工作在我们开始自己的项目之前由老师完成了。

第 2 步:

仔细读系统中每一个类或对象的需求。

状态图的大部分需要的信息都可以通过这个方法找到，提供好的充足的需求文档就可以了。我不确定需求文档有没有任何格式标准。

举一个我们课堂中的例子来说用，我们的电梯系统需求文档中有一些方面应给于不同的关注。

• **回答**：这个部份简述主要功能和特殊对象的存在条件。画状态图的时候这个部份用处不大，但是当状态图完成时可以用来检查功能实现了没有。

•**初始化**:初始状态由给出的信息决定。以 HallButtonControl 为例，初始状态是 ″ 所有的 HallCalls 初值是 false ″ 和 ″ 所有的 HallLights 初值是关″。从这些描述中我们至少能得出结论：初始状态是″门厅呼叫是 False ″ 或 ″ 门厅 Light is off 灯是 False ″，但这还不完全，让我们继续。

• **输入介面**：这个对象将会从其他的对象中取得的输入信息。

输入变量将触发状态变化，例如状态图中的过渡。

在 HallButtonControl 例子中，DesiredFloor 和 HallCall 的数值变化，建立一组事件，触发我们未来状态图中所有的状态变化。

• **输出介面**:这些信息帮助建立状态图中的状态。在我们的例子中，

HallLight 是这个对象的处理的单一输出。我们可以想象一下 HallLight 可以处于多少个状态。凭直觉门厅灯能打开和关闭，因此它在状态图中可能有两个状态 ：

开和关。

让我们看看还有没有别的东西需要加入。

• **状态**：这里的状态和状态图无关，在这些项中一些记号用来速记描述行为。

• **约束和行为**:都将用来检查这个对象的状态机是否实现了系统的功能需求而不打破约束。我们从行为描述知道状态机的变化如何被触发，这很重要。

第 3 步:

现在我们有来自需求的一些想法。现在可以产生最初的状态机。

对于 HallButtonControl，我们有二个状态:

″门厅灯开 ″ 和 ″ 门厅灯关″。

从给出的初始信息，初始的状态应该是″门厅灯关″。行为定义： ″ 当 HallCall[f，d]是真，则指令 HallLight[f，d] 为 On ″，这是第一个状态变化从 on 到 off；同样地，″如果 DesiredFloor.d 是 Stop，则命令

两个 HallLight 切换到 off″，改变状态从 on 到 off。在此，状态机停下来等一个新的门厅呼叫。

第 4 步:

我们决定增加每个状态机的前置条件、 后置条件、行动、入口码和退出码，这些状态机是从约束和行为相关的需求文档得到。

第 5 步:

检查事件的组合是否覆盖所有状态。

第 6 步:

检查是否有死状态，没有（ 组合）事件可以使状态机从该状态变换到其他状态。

第 7 步:

一项项地按照行为运行状态机，确定所有的需求条件被覆盖，而且状态机改变状态，采取行动，正确地修改变量。确定没有遗漏和冗余。

第 8 步:

正确地画出每一个对象的状态图、标示状态、守卫条件、进出码和过渡，记录用于跟踪的相应需求。

6 结论

在这份报告中，给出了一个模拟电梯控制系统详细的 UML 文档。这个文档中用到的 UML 图包括用例图、类图表、顺序图和状态图。在课程项目设计过程中，实时系统中如何使用 UML 图得到了大量的关注，我们项目的成功对这个问题给出了一个很好的答案。由于当前 UML 版本的流行和广泛的符号化，OO 技术可以在实时系统开发中得到适度的发展。

目前面向对象分析和设计方法重心只是在系统的软件。对于实时系统不是完全合适，实时系统需要对系统开发作出整体苛刻的要求而不仅仅是软件。

实时系统的一些方面:

• 硬件元件的定义和他们的特性

- 任务的定义和任务的通信
- 时间限制
- 网络的建模。

如果适当地注意系统的实时特征和不同点的组合，对实时系统的设计和分析有很大的帮助。

为了描述硬件元素和对网络建模，我们用三种不同的视图对系统结构建模。对象构造和软件结构都将重点放在系统的软件结构上，而从系统结构角度给出了一个系统硬件的略图和系统组件间的通信方法。为了描述时间约束给出了顺序图和协作图，通过消息和对象的名称标识时间约束标识系统的实时特征。每个图表仅仅是系统的一些方面的一个图形表示。没有单个图表可以覆盖一个系统设计的所有东西。图表结合起来表达实时系统的完全描述。系统类图的三个不同的视图有助于了解系统的结构。

本文给出的一些我的项目经验实用方法，可能有助于填补需求和设计之间的间隙。当建立系统的图表的时，已经存在一些组件，如系统结构和状态图。不清楚上面总结的方法在一般系统的分析和设计过程中是否仍会有效。举例来说，系统架构 -类图是以Phil Koopman的电梯架构为基础的（这个报告的附件），它使用非标准的UML语言。这里的问题是:UML语言有没有好到，在没有架构图时仍然可以设计系统架构 ？

本文中电梯系统的功能描述仍然限制在课程项目。而在真实世界中更可能需要一些其他特征，例如一个火警按钮、或一个风扇锁。然而，给出了系统的框架，这些附加的功能可以被毫不费力的增加到系统的静态和动态的描述中。

7 参考文献

[1] Hermann Kopetz. *Real-Time Systems，Design Principles for Distributed Embedded Applications*.

[2] Grady Booch，James Rumbaugh and Ivar Jacobson. *The Unified Modeling Language User Guide.*

[3] Perdita Stevens and Rob Pooley. *Using UML，Software Engineering with Objects and Components.*

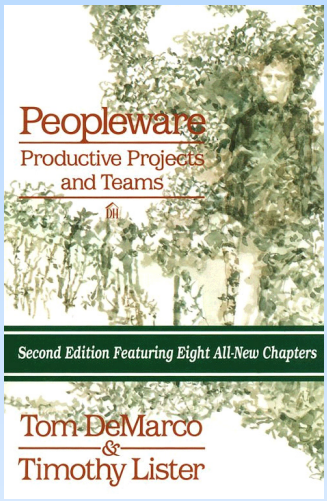
[4] Martin Fowler and Kendall Scott. *UML Distilled，A Brief Guide to the Standard Object Modeling Language.*

[5] Bruce Powel Douglass. *Doing Hard Time: Developing Real-time Ssystems with UML，Objects，Frameworks，and Patterns.*

[6] Desmond F. D'Souza and Ala n Cameron Wills. *Objects，Components，and Frameworks with UML.*

[7] Alan Moore and Niall Cooling. *Developing Real-Time Systems using Object Technology，A white paper from Artisan Software Tools.*

《人件》



《人件》第 2 版

Tom Demarco 和 Tim Lister

翻译：UMLChina 翻译组方春旭、叶向群

微软的经理们很可能都读过—amazon.com

那些早在 20 世纪 80 年代就带给我们大 M 方法论的杰出的人们并没有闲着。他们最新的贡献在于倡导过程改进运动，过程改进的方法更新，更大，更华丽，更雄心勃勃……但这一方法仍然是旧瓶装新酒。你自己的过程改进方案仍然是再生的大 M 方法论。这次它把“一个尺寸适合所有”带动到了极端：不仅仅是在你公司，而且在全世界范围都要一个尺寸适合所有。你公司的能力通过一个固定的模型来衡量。你越精确地适合模型，你的得分越高。分数越高越好，分数最高最好。

假设你在为一个中等规模的商务软件公司工作。公司官员认为 CMM 不是来自匹兹堡而是来自上帝的旨意。他们已经请 SEI 来评估公司的成熟度等级。评估是完整的，而且你们被招集在公司礼堂，SEI 的评估领导会登上礼堂的讲台宣布新的发现。礼堂大厅内一片寂静……

“女士们，先生们，我们有一些不平常的消息要宣布。我们已经完成了我们的评估工作，并且确定你们的 CMM 等级在第五级上。结果是这个公司已经证明它自己是……第六级！对的，是第六级。在我们来这里之前我们甚至都不知道有第六级。对我们所有人而言，这是不平常的一天。我们感觉似乎我们在等级阶段表上已经发现了一个新的等级。你们是现在人类所知道的最好的软件开发公司。谢谢。祝你们愉快。”

中文译本即将发行！

《人件》作者 Tom Demarco（右）和 Tim Lister

Tom Demarco

构建 EJB 应用—模式集合（上）

Eberhard Wolff 著，杨德仁 译

查看评论

摘要

虽然 EJB 为相对复杂的任务提供了简单的 API，然而设计和实现一个基于 EJB 的可扩展的、可维护的和合理的快速应用系统并不容易。随着时间的推移，一套公认的模式已经形成，本文将讨论其中的一些。这些模式将是本文作者写作中的有关 EJB 体系结构和应用模式一书的部分篇章。我们不准备讨论 J2EE 的其它部分如 Servlets 或 JMS，只讨论由 EJB 组成的中间层。

适用对象

这些模式适用于 EJB 应用的体系架构师，设计者和实施者。我们希望读者去理解 EJB 体系结构和 API，也希望读者有一些构建 EJB 应用的经验。

模式形式

本文是 EJB 领域内的模式集，它们并非形成模式语言去解释如何从头构建组件，而为用 EJB 开发某些组件提供细致帮助。

这些模式利用稍做修改的 Alexandrian（亚历山大式的）模式形式。为了简明，省略了对该模式的描述。

我们采用了三种不同形式对于模式的结果语境部分进行表达。关于哪种最好，非常感谢能够得到反馈。

引用的模式和原理

本文是我们正在编写的一本书的部分章节。这本书概括地解释基于组件的系统，并且介绍一些模式和原理来完成这个工作。因为本文不陈述这些模式，而只提供其简短概述。完整的模式能从 www.voelter.de/cpl 下载。它们将在 EuroPloP 2001 峰会中进行了专题讨论，这意味着它们也能从会议论文中找到。

问题的分离

这个原理的目的是把软件体系结构的功能部分与技术分离。这样，系统某部分的开发者不必考虑系统的其它部分。例如，在 EJB 中，体现为具有不同责任的不同角色（部署者、组件开发者、容器提供者等）。

容器

要达到问题的分离，大多数组件技术要在容器中实现。术语容器也用在 EJB 中，指组件在其中运行的软件。它为组件提供了某些服务，也考虑到组件的某些方面，即它调用生命周期方法。

组件

为了实现重用性和减少系统各部分之间的依赖性，引入了组件。它们划分了系统的功能，只依赖于其它组件的接口，是可独立重用的。在 EJB 中有三种组件：实体豆(Bean)、有状态的会话豆(Bean)和无状态的会话豆(Bean)。

组件接口

要减小组件之间的耦合、实现组件的可交换性，只有靠其接口去访问。在 EJB 中，这是远程接口，它定义了由一个客户应用或另一个组件访问的所有方法。

组件实施

要定义组件接口如何实现，就要用组件实现。在 EJB 中，这是一种实现类，即分别扩展了会话豆(Bean)或实体豆(Bean)的类。

主键

每个持久组件必须有一个唯一的标识符，以便检索该组件。这种唯一标识符称之为主键。在 EJB 中，这直接实现为实体豆(Bean)的主键。

受限实现（Restricted Implementation）

因为组件只实现软件的功能部分，组件的实现通常是受限的。因此组件不能干涉由容器处理的技术问题。例如，在 EJB 中，组件不能做线程管理、文件访问或服务器 Socket 连接。

拦截

在组件被调用时，容器通常负责行使某些策略，例如涉及安全或事务方面。这样必须拦截每个调用以完成这

些策略，然后才转发给组件实现。在 EJB 中，安全和事务特征就用这种方法实现。

配置参数

要使组件适应不同的使用场合，就要设置参数。这些参数不能在代码中定义，因为组件在不改变代码的情形下就能适配。在 EJB 中，参数配置在部署描述符中定义。

注释

访问控制清单和方法的事务属性是技术性问题，不应包括在代码中。它们也趋于频繁变化，因此它们必须定义成注释单独存放。在 EJB 中，注释是部署描述符的组成部分。

体系结构模式

速查表

假如……	那么使用……
…实体豆(Bean)对于实体太笨重，即你不需要同步。	类型管理器（Type Manager）：无状态的会话豆(Bean)，直接访问数据库。
…你必须实施一个单件（Singleton）作为系统中的唯一点。	分布式单件（Distributed Singleton）：同步实体豆(Bean)或组件，通过数据库或外部服务进行同步。
…一个业务过程需要合作和/或持久，因此不能实现为会话豆(Bean)。	处理成实体豆（Process as Entity Bean）：用那些具有所需特征的实体豆(Bean)替换会话豆(Bean)。
…因 EJB 限制，不能实现一个服务，例如不能访问文件或线程。	外部服务（External Service）：为受限制的部分提供外部应用系统。
…必须定义实体豆(Bean)之间的协作或者跨实体豆(Bean)的事务。	会话豆(Bean)外观（Session Bean Facade）：额外的会话豆(Bean)调用实体豆(Bean)，这样就定义了协作和事务。
…在应用中出现长事务。	长事务（Long Transaction）：一个有状态的会话豆(Bean)收集数据，并在一个短数据库事务中完成工作。

外部服务（EXTERNAL SERVICE）

由于受限实现（RESTRICTED IMPLEMENTATION），实现 EJB 应用时，你意识到需要一个在 EJB 编程模型中不能实现的服务。容器约定[1]不允许实现你需要的服务。

设想一个应用系统需要某种基于时间的通知机制如 cronjob[1]。在 Java 中实现的通常方法是启动一个线程处理时间表，然后在需要的时间点通知客户端。这种方法在 EJB 中是非法的，因为容器约定禁止在容器中启动你自己的线程。

另一个例子是日志组件，它把日志信息输出到一个文件。这不能用 EJB 实现，因为不允许直接访问文件。还有其它几个限制，详见*受限实现*部分。因此，这种服务不能在容器中实现。

因此

实现一个单独的应用来提供你需要的服务，称之为外部服务。外部服务可以激活然后调用 EJB 应用（例如基于时间的通知）。在其它情况下，它是被动的，由你的 EJB 应用（例如日志组件）调用。

优点

由于外部服务不受容器约定限制，你可以在外部服务中自由地使用任何需要的特性，甚至也有可能不在 Java 中而使用不同于 Java 语言来实现外部服务，如利用 CORBA 实现。

缺点

外部服务并非 EJB，因此失去了 EJB 的优点如高可用性和负载平衡。在大多数情形下，必须在外部服务中手工实现这些特性，以免系统中出现既不可扩展又不能自动排除故障的单点。否则，整个系统的性能和可靠性就会降低。

你也要考虑它要在安全性和扩展性理念下集成。这相对复杂，因为事务和安全语境衍生到外部服务是困难的。

额外的管理工作也是必需的，因为不仅是应用服务器、而且还必须部署和管理服务。特别当容器管理和监视工具极有可能不配合外部服务时，这就很困难且昂贵。

变体

有些容器支持在容器的虚拟机内运行外部服务，这改进了调用外部服务的性能，因为容器调用外部服务时不需要内部进程调用。但是这种特征目前仍未标准化，当然可移植性就差。

长事务（Long Transaction）

在应用中，有些事务持续几分钟甚至更长时间。数据库系统缺乏对这种长时间运行事务的支持。好多数据库入口由于长事务而经常被锁定。这导致用户长时间等待，从而影响了应用的可用性。

通常，许多应用级的工作仅仅是数据库级的一个事务，特别是交互性应用。然而，大多数据库设计只用于处理短事务。长时间锁住大量数据行可能使数据库对其它客户端不可用，因为他们不能访问被锁定的数据。

设想一个电子商务网站中的购物车。在购物过程中，订购了几个商品项。因此在库存中商品项的数目必须逐项改变。若在购物的开始，便启动一个事务，这意味着数据库中的每个商品项都被锁定。那么在事务活动期间其他人就不能同时订购这些商品。这当然是不可接受的。

因此

提供一个有状态的会话豆(Bean)，不打开数据库事务就收集数据。因此，用户能长时间，即 web 站点的一个购物会话期输入数据。所有数据收集完毕时、数据库作业在单个操作中完成。因此对客户端而言，这像一个长时间运行的事务。然而，真正的事务在最后的操作完成后才开始。那时，采集到的数据在一个短事务中写入数据库。

变体

即使更长的事务也能用实体豆(Bean)代替带状态的会话豆(Bean)实现（详见**处理成实体豆(Bean) (Process as Entity Bean)**）。性能优化对在客户端收集数据并使用无状态会话豆(Bean)敏感。注意，在这种情形下事务逻辑处理是分布式的：真正的事务在服务器端执行，而数据收集在客户端进行。

缺点

要保证写入数据真正一致，可能应在最后操作期间检查数据库。若在这个会话豆(Bean)活动期间，别人改变了数据，那么收集的数据是无效的。例如，有人改变了数据库数据而收集到的仍然是旧数据。若写入数据库，其它事务所做的修改就会丢失。在这种情形下，应当发送一个错误到客户端，数据不会写入数据库。

另一个问题是，假若使用这种模式，会破坏事务的独立性。设想在事务的更新动作发生之前，必须从数据库中读出某些数据，更新操作在最后方法调用中完成。因数据在数据库中没有锁定，以前读出的值或许那时已经改变。这样有必要检查在事务完成之前读取的数据是否已改变，否则指示失败。注意，假若没有读数据或因其它规则能排除冲突，这种检查就可以省略。

假若包括了这种检查，模式会模仿乐观的锁定，这种技术能用于数据库实现长时间运行的事务。通常，数据库利用悲观的锁机制在事务开始就锁定数据库行直到事务结束。在这种情况下，事务肯定成功并把结果写入数据库中。在乐观锁的情形下，数据不会被锁定。在事务提交时，检测与其它事务的冲突。也就是，如果其它事务修改了当前事务要改写的数据，就回退事务。

另一个潜在问题是，这种模式依赖于数据库表中的时间戳。假若这个时间戳并不存在，因为表的大小关系，不能简单地把它加入到现有数据库中。在这种情形下，时间戳能存储在附表中，这个附表只含有主键和时间戳。

在确信冲突很少发生或事务很容易重做时，才能利用这种模式，因为每个冲突都将使事务失效。假若你能处理不完全一致的数据库，那么也可省略这种检查。在这种情形中，每个事务都成功，但有些修改或许会丢失。这减少了失败次数，但与悲观事务相对比，频繁的失败乐观事务的最大缺点。

处理成实体豆(Bean) (Process as Entity Bean)

复杂的业务过程有复杂状态、运行时间很长和/或需要多个用户合作。状态化的会话豆(Bean) 不能提供必要的数据安全性和来自多客户的并发访问。这时候你的业务过程不能用会话豆(Bean)建模。

带状态的会话豆(Bean)偶尔会因容器破坏或系统异常而遭到破坏。如果状态会话豆(Bean)并非引用，在超时之后它将被破坏。这意味着存诸在状态会话豆(Bean)中的数据是不安全的。利用会话豆(Bean)的业务处理仅包含了几个操作，并且假设它们由一个客户端调用。因此并发访问是不可能的，从一个客户向另外一个客户传递会话豆(Bean)的引用并不直接。来自一个客户端的访问并往往不如协作工作流系统及其它工具所显示得那样充分。

因此

把长时间运行或多用户业务处理建模为实体豆(Bean)而非会话豆(Bean)。这允许你的业务处理持久并可由多个客户端并发访问。

概要

即使发明实体豆(Bean)的初衷是用于建模业务实体，实体豆(Bean)的一些特征对业务处理也非常有用。例如实体豆(Bean)可并发访问，这对于支持协同工作非常有效。对于实体状态的长久存储的额外支持也有助于实施长时间运行的业务过程。注意，有些事情显示实体豆(Bean)并非真正用于业务过程：一个实体豆(Bean)有一个标识符并永久地存储在数据库中。一个过程只需要把那个标识传递给参与这个过程的其它客户端。数据库只用于保证过程的状态不丢失。在这个过程完成后，这个状态也通常被删除。

通常，实体豆(Bean)利用豆(Bean)管理持久性，因为它们表示的数据在实体豆(Bean)被开发之前被存储在数据库中，它们需要额外的灵活性将数据库数据映射给实体豆(Bean)。因为业务过程很少表示在数据库中，用容器管理持久性实现这个模式是有意义的：这个过程只需是长久的，而不需要数据库层的显式控制。

相关模式

这种模式能用于增强长事务，去处理甚至更长的事务。

会话豆外观（Session Bean Façade）

访问业务实体通常需要按某些确定的顺序或在同一事务内调用某些方法。有些实体豆(Bean)的方法只在与其它实体豆(Bean)联合时才有意义。然而，你不能定义一个事务跨越多个方法或不同实体豆(Bean)。

事务属性只能为每种方法设置。这意味着没有方法使得多个操作或多个实体豆(Bean)的操作必须在相同的事务内被调用。这有待客户端去解决。对于跨越多个实体豆(Bean)的已定义的过程也一样：这种事情难于按一般实体豆(Bean)定义。

因此

提供一个会话豆(Bean)外观即一个无状态的会话豆(Bean)去访问实体豆(Bean)。因两个组件都位于服务器端，只有服务器内部的通信发生，而没有产生网络开销。另外，无状态的会话豆(Bean)中的一个操作可能调用不同实体豆(Bean)的几个方法。一个事务能跨越实体豆(Bean)调用的所有方法。要尽可能避免直接访问实体豆(Bean)。

概要

无状态会话豆(Bean)是为短期业务过程设计的，而实体豆(Bean)意为实现业务对象。因业务过程要用业务对象工作，利用无状态会话豆(Bean)访问实体豆(Bean)就很自然了。然而，通过缺省的无状态会话豆(Bean)间接访问实体豆(Bean)，这种模式甚至还可扩展。这是因为客户端通常对实体豆(Bean)不感兴趣，而是由无状态会话豆(Bean)实现完整的业务过程。这样一个过程必须执行为一个事务，有关事务问题也就得到解决。

利用这种模式的另一个好处是需要的远程调用较少。假若所有的实体豆(Bean)都直接从客户端访问，那么每种方法调用都将是远程调用。在应用这种模式后，至少一部分这种远程调用将由在服务器上的局部调用代替，即从会话豆(Bean)外观向实体豆(Bean)的调用。

相关模式

与***长事务***模式相比，这种模式暗示在事务期间用到的参数要提前知道，而用长事务收集它们，此后事务被执行。值对象（Value Objects）能用作外观的参数，不需修改就能转发给实体豆(Bean)。这种模式似乎是把[GHJV94]中的外观模式转换成 EJB，但有一个不同的目标：当原来的外观模式试图隐藏一个对象后面的子系统的复杂性时，这种模式只提供一种表达实体豆(Bean)之间的协作。***处理成实体豆***(Bean)也是一种旨在拥有多种方法调用的模式，但强调协作实现任务。长事务强调事务的长度。

分布式单件（Distributed Singleton）

企业 EJB 应用要实施[GHJV94]单件模式。通过受限实现，你不能用静态成员和同步状态实现单件模式。你要找出实现遵循 EJB 规范的单件模式的方法。

在实现一个复杂应用时，你要实现一个服务，这个服务能以一个受控制的方法从应用的各个部分访问。例如，当你要实现类似一个标识号产生器时，你就需要提供这种标识号的单一实例。单件模式描述了一种方法实现能满足这种需求的一个服务。很不幸，它不容易在 EJB 应用中实现。这是因为通过受限实现，Java 的一些特性不允许使用。

例如，要实施负载平衡，多个容器会成为你的豆(Bean)的宿主。这些容器运行在不同计算机的不同的虚拟机上，静态成员不能为它们共享。因此每个容器都有自己的静态标识成员，这种使得不可能产生唯一标识号。

因此

在你的只运行一个实例或至少表现如此的体系结构中找到一个点，把单件同步机制转移到该点。

概要

要在体系结构中找到行为类似单点的点，有几种不同的选择可发生同步：

你可用数据库同步，有几种方法：

- 利用实体豆(Bean)同步方法调用。因为容器负责同步访问实体豆(Bean)，它表现如同应用中的同步点。
- 利用访问数据库的无状态会话豆(Bean)，以及通过数据库锁定机制同步。在此也可以用存储过程。这样，外部系统也能用 EJB 应用同步。例如，用这种方法可以同步历史系统。存储过程也能用于实现这个目标。

可选地，你还可使用外部服务保持“通常的”***单件***实现。在外部服务内，这个“通常的”***单件***实现是合法的，因***受限实现***并没有覆盖外部服务。

因为不同方法特性不一，需要检查系统的具体实现和体系结构以确定用哪种方法实现单件最符合需求。

缺点

当然，用单件的问题是你锁住了那些要并行访问单件的部分系统功能。这有碍于系统的扩展性。因此要尽力避免单件（在上述的标识号发生器例子中，你可以使用数字环）。

当你必须发现/引进应用的单点去同步调用器时，注意不要引入一个失败的单点，否则，系统的高可用性就不再有保证。

类型管理器（Type Manager）

实体豆(Bean)提供了并发同步机制、缓冲池机制和扩展生命周期管理。这些特性施加了性能惩罚（性能会下降）。对于许多客户端并发访问一套实体集合并且使用附加特征的情况，这种性能降低是可接受的。然而，在一些不需要这些特征的系统中，实体豆(Bean)并没有优化性能。特别是对一些高度并发数据读取访问更是如此。

例如在典型的股票交易系统中，股票数据由系统管理。大多数操作只实现对股票数据的某种监控，因此在只读情况下，不会发生冲突，这时就不需要锁定机制或其它并发同步机制，那么使用实体豆(Bean)将意味着许多无谓的花费。

因此

利用会话豆(Bean)直接处理数据库中的实体。使用值对象描述每个调用中的数据，并用主键去识别操作所应用的实体。

优点

获得的性能要比访问实体豆(Bean)高，因容器并不需要实例化和支持单个实体豆(Bean)。系统性能直接与数据库系统的性能相关联。

缺点

因为你没有按 EJB 规范所提倡那样使用实体豆(Bean)，你实现的体系结构不能得益于容器提供的一些特性。例如，利用容器管理持久性的实体豆(Bean)被容器缓存，因此在一些情形下实现持久性的性能优于类型管理器或豆(Bean)管理持久性（BMP）。

变体

假若组件是用于为那些遗留应用提供接口，遗留系统提供面向过程的 API，基于类型管理器的体系结构也有用。甚至遗留应用具有不同的接口，例如一个面向消息的中间件，它把类型管理器作为接口使得对所有系统的访问都统一具有一定的意义。

类型管理器通过会话豆(Bean)代替实体豆(Bean)。因此你要用会话豆(Bean)的生命周期去处理实体操作。

相关模式

若性能问题并非来源于实体的并发使用或工作集的分布，而是来自于对大量实体豆(Bean)的写操作，你应该用***实体批量修改***(ENTITY BULK MODIFICATION)代替。

依赖管理模式 (Dependency Management Patterns)

速查表

如果……	那么使用……
…因实体豆(Bean)相互引用，而不能重用	主键作为引用（Primary Key as Reference）：而不直接引用存储主键。
…实体豆(Bean)间关系复杂而且频繁改变	关系服务（RELATIONSHIP Service）：在外部服务中、而不在实体豆(Bean)中存储引用。
…循环依赖使得难于重用和维护	事件侦听器（Event Listener）：方法调用使用在一个方向中，在其它事件中被传递。

主键作为引用(Primary Key as Reference)

因为实体豆(Bean)直接相互引用，不能孤立地使用其中之一。在编译时，至少必须访问其它实体豆(Bean)的组件接口。不能用一个具有不同接口的等价豆(Bean)代替另一个实体豆(Bean)。

实体豆(Bean)是组件，这意味着它们不应该依赖于其它组件。一旦你使用另外组件的组件接口，这些组件在技术上就相互依赖，也就是说，必须至少具有接口使用或编译引用的组件。这足以使组件本身或者在其它组件不可访问的语境下不可用。这是对重用的最大限制，而重用正是使用组件的主要目的之一。使用句柄并不能解决这个问题：你仍然必须访问引用豆(Bean)的接口，句柄是应用服务器特定的，这是另外一个缺点。

因此

利用主键而不是引用或句柄去建立关系模型。因为每个实体豆都必须有主键，这对每类实体豆(Bean)都是可能的。这使得实体豆(Bean)松散耦合：它们仍然在逻辑上而不在软件工件上依赖其它组件。

概要

去耦是组件最重要的目标。当然组件必须能相互引用。然而，这应该采用避免不必要技术依赖的方法来完成。

存储主键代替对象引用的实现思想违反面向对象原则，因面向对象设计试图建立包含对象及相互引用的模型。不过，对于基于组件的系统，这是一个良好的折衷。要注意的有趣的一点是，实现实体豆(Bean)的持久性必须使用主键或句柄存储对实体豆的引用。因此，不管怎样，在数据库层只存在主键和句柄。

缺点

注意，实体豆(Bean)减弱了直接访问它们所引用的实体豆(Bean)的能力。然而，工作于多个实体豆上的操作应当使用会话豆(Bean)实现。这对于使用主键取回引用的实体豆(Bean)是有责任的。

这导致问题：实体豆(Bean)本身不能取回这些引用。例如，实体豆(Bean)不能删除其它依赖的实体豆(Bean)。然而，这种依赖的对象应该真正实现成依赖对象（DEPENDENT OBJECTS）。注意实体豆(Bean)应该是粗糙的，这样才能自我包容。不应该揭示访问相关对象的需要。然而，利用这种模式拒绝实体豆(Bean)调用它所引用的其它实体豆(Bean)的任何方法。这也是该模式很难用于会话豆(Bean)的原因所在：会话豆(Bean)必须实际利用引用的豆(Bean)，这是意味着它们依赖被引用的豆(Bean)的方法。

相关模式

若要在运行期间增加引用，你可以创建一个会话豆(Bean)管理系统中的所有引用（一个关系服务）。这将会具有增加和取回实体豆(Bean)间关系的操作。

关系服务(RS)

当实体与另一实体有关系时，它必须存储主键（或键，如 1：n 关系），并且必须提供访问相关实体的操作。在应用中，你有许多不同的关系，这些关系变化很频繁，在你需要增加一种新的关系时，就要改变实体实现。这需要组件接口和实现的修改、重新编译，也需要表结构改编和重新部署。

以一个产品管理系统为例，产品项目被包装在不同实体中。诸如调色板，有 10、20、50 个的包装形式等等。设想你在产品实体中存储了对某个调色板的引用，去跟踪它属于哪个递送过程。假若产品改变了包装容器或运输工具之类，你必须调整实体以反映这些变化：对容器或运输工具的引用必须加到这个实体中。创建了新的依赖性，修改这种关系的操作也要加到这个实体中。

当一个产品按一个调色板或按 20 个包装或按一卡车（或许甚至按卡车上的 20 个包装按单个调色板）。只要这种关系变化，你必须改变实体，你需要频繁重新部署应用，造成巨大花销。

因此

在关系类型频繁变化的系统中，把关系归于外部的一个关系服务。这种服务能存储两个或多个实体之间的关系。它存储实体、一种关系、甚至属于那个关系的附加属性、任何非实体的内容。

优点

因为实体之间的关系不再是部分实体本身，改变关系结构根本不影响实体。(顺便指出：若从概念观点看问题，询问关于哪个调色板/卡车递送等产品项目是否有意义值得怀疑。因此把这些关系外部化甚至比保存在实体内部更自然。)

利用关系服务把这种实体降解成真正的实体，而不存储那些并非自然属于该实体的实体间关系，因关系总是用例特定的，这种模式极大地扩展了重用的机会。

缺点和忠告

实体不再完全自容，如果关系服务存在，这些实体才能成功使用。程序员必须显式地访问这种关系服务，这在某种程度上把编程模型复杂化了。

变体

关系服务有额外的益处：你能用关系存储名/值对清单，你能赋予每个关系一种类型，你甚至能预定义哪种关系类型允许用于哪种实体。

这种关系通用的外部组件在数据库中使用分离的表进行维护。为了方便编程人员，组件能提供如 *getRelationships（type）* 的操作，这种操作内部使用关系服务组件，从而隐藏额外组件的出现。

关系服务能实现成一个组件，与使用它的客户端运行于相同的地址空间。若因组件模型的限制，不可能使用关系服务，那么可以使用外部服务。

相关模型

减小对相关组件的实际组件接口的形式依赖性的另一种方法是*主键作为引用*。

请注意，OMG 已经为 CORBA 对象定义了一个关系服务[OMGREL]。它能用于好的角色模型，尽管它的许多特性是许多体系所没有的。

事件侦听器（Event Listener）

你不需要的事情是相互或循环的依赖性，因为其后果是特别不利于维护和部署，也降低了组件的重用性。依赖性只应该是单方向的，产生一个“层状体系”[POSA]。在层状体系中，依赖性只存在于一个方向，即从高层到低层。较低的层级不能直接访问较高的层级。那么，怎样从低层向高层通信呢？

考虑客户*(Customer)*实体豆(Bean)和订单*(Order)*实体的 1：n 关系集。客户*(Customer)*有一个标志指示其可靠性，这取决于他是否及时承付所有订单。只要当一个订单*(Order)*的支付状态变为到期未付(*overdue*,)，就会通知相关的客户*(Customer)*，其可靠性会得到调整。

订单是将用于各种系统的实体，即它应该是可重用的。一个客户通常直接依赖于订单组件，因为客户应提供类似 *ListOrders()*或 *ListOpenOrders()*的操作。

如果订单额外地直接依赖于客户*(Customer)*接口，你只能用具有这个客户组件的订单，反之亦然。重用是折衷的。如果后来注意到系统中的其它组件对订单是否过期未付得到通知感兴趣——或许财务部门要向客户*(Customer)*发送催款单，那么它就变得更加糟糕。

因此

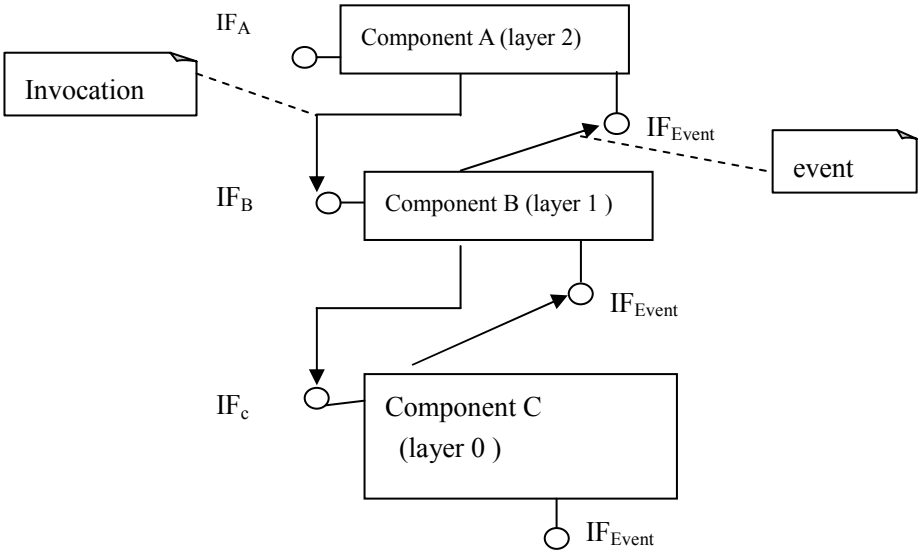
要保证直接依赖性只存在于一个方向。在此方向中，利用直接方法调用。在相反方向中，利用基于通用事件接收器接口的类事件通信机制。事件的接收器用事件的生产器注册。如果事件发布，接收器就得到通知。

优点

事件发布者只依赖于一个通用事件倾听器接口，而不依赖接收器的具体组件接口。减少了对单一方向的依赖。这种技术详见[BCF]。

缺点和忠告

要注意，滥用这种模式会致使大量性能衰减，调用组件的一个操作的会导致事件被层层激活，不用原来的方法调用者直接检查。当然，若不用这种模式，用另一种方法实现直接调用业务方法也是可行的。



还要注意，这种模式要求事件的产生器要保持事件接收器的清单，可能要按照事件类型分组。这个组件必须提供属性和操作管理这些关系。因这些关系对发布事件的任何组件都一样，关系服务组件能用于保存和管理这些关系。

变体

事件能同步或异步传递。对于小系统，同步回调是可行的。对于具有许多依赖和潜在事件的较大系统，正如异步事件组件中描述，应使用外部的、异步事件分配器（通常是外部服务）。在 EJB 中，你要特别小心，因为在缺省情况下不允许直接向操作的调用者回调。要解决这个问题，某些容器组件可以标记为“重入”，这样允许回调当前活动方法的调用者。如果不可能这样，时用外部服务把事件传递给调用者。

附加信息

你要注意，这种模式根本并非特指实体豆(Bean)。会话豆(Bean)也能作为事件的生产者和消费者。

注意 EJB2.0 中的消息驱动豆(Bean)和 JMS 在此并非直接有用。这是因为消息驱动豆(Bean)*仅仅*是被驱动的消息。其结果是，Bean 不可能有正常的业务接口，*也*不能利用来自于队列或主题的事件。当然，去创建适配器 MPB 利用事件并向注册的接收器转发调用是可行的。

这种模式基本上是[GHJV94]中描述的*观察者(Observer)*模式的 EJB 改编版本。

组件内部结构模式

速查表

假若……	那么使用……
…实体豆(Bean)包含持久性代码和业务逻辑，因此难以测试，	数据访问对象（Data Access Object）：持久性代码在单独的类中实现。
…因为在测试之前必须部署新版本，测试实体豆(Bean)的业务逻辑很困难	封装的业务对象（Wrapped Business Object）：业务逻辑在单独类中实现。
…因远程接口中的方法没有实现，出现部署错误	业务逻辑接口（Business Logic Interface）：创建一个接口，定义所有业务逻辑方法。
…必须定义比实体豆(Bean)更细粒度的持久性对象。	依赖对象（Dependent Object）：细粒度对象实现为实体豆(Bean)的一部分。

数据访问对象（DAO）：

在使用豆(Bean)管理持久性时，实体豆(Bean)是业务逻辑代码和系统级逻辑代码的混合体。因此实体豆(Bean)非常复杂，因为在其实现中，混合了不同的关注点。不能从业务逻辑分开单独开发和测试持久性代码。

考虑一个典型利用关系型数据库存储持久性的实体豆(Bean)。在 *ejbstore()*、*ejbload()*等操作中，你将反复使用 JDBC 去访问这个数据库。这包括连接和语句管理，更重要的是，它包括 SQL 语句。这种 SQL 语句甚至要动态创建，从而导致许多复杂化的字符串处理代码。

如果在不得已情况下更换数据库供应商，还将出现一些问题。虽然 SQL 语句是标准化的，在各种数据库产品之间仍然有些不兼容，因此你要全面修改实体豆(Bean)中的 SQL 语句。

你面临的另一个问题是，若要测试或调试实体豆(Bean)中数据库特定的代码，你要先在应用服务器部署豆(Bean)。这使得测试复杂化，你还需要有支持在应用服务器的调试工具来调试。

因此

增加一个类，让它包括用于处理某个实体豆(Bean)持久性的所有逻辑。这便是所谓的数据访问对象（DAO）。实体豆(Bean)在其生命周期内调用数据访问对象，回调方法去处理持久性。

概要

数据访问对象（DAO）必须获得和设置实体豆(Bean)的成员。有几种实现方法，最简单的方法是让数据访问对象直接访问实体豆(Bean)的成员，这是两者之间的很紧密的耦合，至少使得对数据访问对象的单独测试困难。较好的方法是用值对象交换实体豆(Bean)和数据访问对象之间的数据。这种变体的问题是，由于这些***值对象***的频繁生成和销毁，导致应用服务器的负载较重。当你使用***值对象***交换数据频繁访问***数据访问对象***时，这会导致很差的性能。

优点

利用***数据访问对象***，比直接从实体豆(Bean)内处理 Bean 管理持久性有几个优点。首先，实体豆(Bean)变得简单，因为它们只包含业务逻辑和委托持久性操作。

当你遵循*关注分离*原则时，便能单独开发和测试持久性代码，而不考虑与实体豆(Bean)的其它部分。

另外，若用数据访问对象，能获得更加灵活的体系结构。随着应用需求的变化，你要从容器管理持久性转到 Bean 管理持久性转到类型管理器或其它相关方法。若用数据访问对象，切换将变得更容易，因数据访问对象能被实体豆(Bean)（借助于 Bean 管理持久性）和类型管理器使用。而且，因为明确分离了数据库代码与 Bean 的其余部分，在 Bean 管理持久性和容器管理持久性之间的切换也比较容易。

缺点

因为引入一个新类处理持久性，具有更复杂的组件实现。这导致开发与维护实体豆(Bean)的工作加重。再有，引入了新的迂回层次，略微引起性能下降。不过，与客户机与服务器之间的进程间通信、容器侦听工作例如检查安全性、事务设置以及实际数据库访问相比，这种迂回是微小的，因此它通常不会导致明显的性能降低。

变体

如果使用*工厂*（模式）[GHJV94]去获取正确的数据访问对象，便获得了实体豆(Bean)和数据访问对象之间的松散耦合。假若你要改变持久性技术或数据库设计，可以很容易改变持久性代码。

也要考虑基于框架（framework）的办法产生 SQL 语句，以便独立访问数据库。

相关模式

对于业务逻辑性，***封装的业务对象***（WRAPPED BUSINESS OBJECT）模式与数据访问对象对持久性代码所做的工作一样。应用这两个模式都把实体豆(Bean)降低成只用于业务逻辑和持久性代码的“基础设施适配器。”

依赖对象（Dependent Object）

表示业务实体（即实体豆(Bean)或类型管理器）的组件是相对粗粒度的工件。通常，这些实体有更细粒度的

内部结构，其中有些属性形成了一个语义组，有时这些语义组甚至能被多次请求。然而，这些组本身并非独立实体。

以*人（Person）*组件为例，*人*通常包含了如姓名和生日等标准属性。然而你通常还要存储其它信息如地址、电话号码、电子邮箱或 web 地址。例如，地址本身又有几个属性，如街道、邮编、城市和国家。从 *Person* 组件的观点，这些属性形成了一个语义组并合成一体。对于区号、电话号码也是一样。

另外的考虑是，一个人*(Person)*或许不止一个地址或一个电话号码。在大多数实际应用中，*人*需要几个地址，可能是家庭地址、工作地址等等。通常你不能提前知道需要哪个以及需要多少个地址。

要使诸如地址和电话号码之类的实体有自我权力是不可行的，有两个原因：第一，它们被紧密耦合到其“宿主”实体——实体在没有隶属于真正的实体之前，地址和电话号码没有用；第二个原因是性能，为每个属性组创建单独的实体会导致使服务器开销过大。

因此

把一个实体的状态分成几个依赖对象。每个*依赖对象*代表着属于那个实体值的语义组。依赖对象的生命周期与其“宿主”实体的生命周期一样——它们不能自主存在，也没有自己的标识，通常也不变化。依赖的对象也可有 0、1 或多个实例，这取决于应用的需求。

优点

利用***依赖对象***具有组件结构组织得更好的优点。若要多次存储每个实体的某些属性语义组，依赖对象是唯一可行的方法。

缺点和忠告

请注意，依赖对象没有逻辑标识，因为它们只是“宿主”组件的部分状态。这有一些微妙的后果：向组件的地址清单增加一个新地址（简单地传入一个新地址对象）容易，而移去一个地址就困难多了，因你不容易识别出这个地址。要解决这个问题，有几种方法：

- 依赖的对象的“标识”由它的一套属性值定义。这样，若想移去地址“*High Streetw 10，89520，Heidenheim，Germany*”，你给删除操作传入具有这些参数的依赖对象，如果相同，就从组件的状态中删除这个对象。因依赖对象没有标识，并因为由此导致它们通常由一套属性值识别，你要确信一旦依赖对象创建就不能修改——它们应该永恒不变。如果存在修改的可能，则没有办法识别特定的依赖对象，因为其标识随着属性的变化而变化。因此，若要修改诸如一个地址的邮编，你必须用修改后的邮编创建一个新的依赖对象，并删除旧的、加入新的对象。
- 要促进平等比较，可使用客户端不可见的 ID（标识），在创建新实例时，它们由服务器创建。

- 可选地，也可以为整个依赖对象清单提供设置者和获取者操作。客户端然后将检索地址清单，按其意愿修改并回写整个清单。

哪种方法最好取决于用例——如果通常需要所有实例而修改很少，那么可用列表变量。列表变量有些缺点：下载整个列表的客户端“锁”住了所有数字——会产生竞争情形，因为实体不支持长事务。在 Java 中，因为没有通用编程，这个列表接口采用弱类型，这样，实体必须检查新设置的列表，只提供正确的元素类型。

变体

依赖对象能、但不需要在组件接口中可见。若它们可见，其行为类似值对象，客户端能直接创建这种依赖/值对象，并把它们作为参数传入或作为返回值接收。要简化客户端编程，提供工厂操作去创建依赖对象的“空”实例。其缺点是你“发布”了组件的内部数据结构，这使得更难改变这种内部结构——你还要调整客户端。若选择依赖对象不可见，系统能用单独的值对象类或***批量设置器***增加或删除那些包含在***依赖对象***中的属性。

附加信息

如果组件对于那些由依赖对象组成的实体支持自动持久性，依赖对象的使用就极大地简化了。否则，应当创建自己的“框架”去管理依赖对象的持久性。

也要注意到，依赖对象并非等同于实体之间的关系。实体关系通常暗示相关的实体是实体——这意味着它们有自己的标识。实体关系通常比依赖对象招致更多的性能开销。当然，在有些情形下，难于决定如何设计系统，例如，若对顾客、职员和咨询实体使用相同的地址，那么，把地址作为单独实体去管理才有意义，关系服务能提供这种帮助。

相关模式

这种模式与值对象相似。依赖对象和值对象之间的区别是，值对象是因观察一组参数的频繁使用而由客户端引入的，并且主要是依靠减少所需的网络跳动次数而优化性能的一种方法。依赖对象用于结构化组件的内部状态。

因为依赖对象非常大，且不用于实体访问的每个情形，它通常是惰性状态（LAZY STATE）或延期存储（DEFERRED STORE）的良好的侯选者。

值得注意的是，目前的 EJB2.0 规范的草案包含了这种模式的实现。然而，这个规范仍在标准化过程中，特别是这部分仍未定稿。

封装的业务对象（WRAPPED BUSINESS OBJECT）

在豆(Bean)中实现复杂的业务逻辑。由于业务逻辑非常复杂，豆(Bean)也就非常复杂。因为豆(Bean)必须部署并在*容器*中执行，开发、测试或调试业务逻辑都很困难。

设想客户*(Customer)*的实现。除属性外，还有一个方法测试客户*(Customer)*是否具有某些额度资金的信用度。

这种方法的实现非常复杂，因为顾客的信用度并非只取决于他的属性，也要检查旧系统的一些工作——然后是复杂的计算。因为这非常易于出错，因此要进行广泛的测试，为了方便起见，应在应用服务器之外完成。

另一个实例是处理矩阵乘法的服务，这项服务的实现相对复杂。如果要在一个无状态的会话豆(Bean)实现，在要测试和调试实现时，你就会碰到问题。如果你希望调试被部署的 Bean，就需要一个支持在容器内调试的调试工具。虽然存在这种远程调试工具，但是一般的调试工具更容易使用。

因此

增加一个额外的类来保存业务逻辑，即所谓的业务对象。利用 bean 将对业务操作的调用委托给业务对象。因此 bean 只是一个包装器。对于测试和调试，可以直接使用业务对象类。

优点

利用封装的业务对象把业务逻辑从 EJB 特定的需求中分离出来。因此，能独立实现和测试业务代码。由于你把业务代码从处理容器的代码中分离出来，也就能在应用的其它部分使用业务逻辑，例如，在胖客户端使用。

缺点

有一个缺点是，你必须处理每个组件的附加类和/或接口，因此在系统中有更多的类。所以只有在业务逻辑复杂，需要独立于容器进行测试和/或调试的时候使用***封装的业务对象***。

变体

容器管理持久性有一个特殊问题：你通常只能把 bean 的成员用作持久属性。因此你必须在业务类中实现逻辑，并使得其成员放在豆(Bean)中。这就导致业务类和豆(Bean)之间的紧密耦合，这抵消了***封装业务对象***的优点。因此，解决这个问题的一个方案是，成员既在豆(Bean)中又在***封装业务对象***中，在使用***封装业务对象***之前，豆(Bean)必须用自己的数据填充***封装业务对象***的成员。

另一个实现变体是从业务对象类中导出豆(Bean)的实现。这也为大多数容器解决了容器管理持久性所具有的问题。而且，你不必实现任何代码从 Bean 到业务逻辑类委派调用。

相关模式

要注意与***策略***（STRATEGY）模式的区别：***策略***的目标是使整体业务逻辑的某些方面可互换。利用***封装业务对象***，完整的业务逻辑将移到一个单独类中，有助于测试和调试。***封装业务对象***的动机并非让业务逻辑可以互换，而是清理豆(Bean)的结构并使得开发容易进行。***策略***模式也能用于***封装业务对象***之内。

若要实现***封装业务对象***，考虑使用业务逻辑接口组件保证 bean 和***封装业务逻辑****(WRAPPED BUSINESS LOGIC)*实现相同的接口。

对于从组件中分离数据库访问代码，数据访问对象组件是一个相似的模式，若使用***封装业务逻辑***和***数据访问对象***，你的***组件***只要处理体系特定的功能并作为对业务逻辑的一个***外观（facade）***模式。这种***关联分离***模式赋予了改变技术的高度灵活性，因为类的结构和目的比较简单，开发更加容易，因此更易于理解。

业务逻辑接口（Business Logic Interface）

组件接口并非技术上与***组件***的实现相关。这意味着直到组件部署或验证，发现不了***组件接口***实现中的错误。

在***组件接口***实现中的简单错误，例如方法名称拼写错误能够导致部署失败。通常这种错误由编译器发现。而对 EJB 实现不能直接实现***组件接口***。原因是组件接口包含了只能由容器实现的方法，例如把句柄传递给组件或删除组件的方法。定义在***组件接口***中的业务方法必须由组件实现。因这并非受到编译器的强制和检查，这个域中的错误在相当后期才能检测到。

因此

提供一个只包含业务方法的接口，这种业务方法必须在组件接口中实现和声明。

概要

这样，确保业务方法真正如同声明的那样实现。也会有些小问题，例如在组件接口中，必须声明方法抛出 *RemtoteException*（异常）。然而，这个实现并不能抛出或声明这些异常。因此，尽管要实现方法成功编译应用，仍然要能够声明或抛出 RemoteExceptions（异常），在应用检验或部署之前，而这种错误将检测不到。注意，这些错误与规范冲突，但大多数服务器仍然接受带有这些错误的豆(Bean)。

缺点

某些 EJB 部署使用这种模式的 Bean 存在问题。

参考资料

这种模式也在[MH00]中也有解释。有关这种模式的细节，特别是涉及 Java 类型系统时，可参照[WO00]。

构建大型系统的模式

速查表

假若……	那么使用……
……组件不应该知道其它哪些组件负责某项任务以进一步去耦	请求集线器（中心）（Request Hub）：一个中央组件向负责的组件转发请求。
……组件要有一些设置环境的安装代码	可管理组件（Administratable Component）：增加额外接口，允许调用安装代码。
……系统的一些组件形成一组，但没有正式分组	业务组件（Business Component）：提供额外组件作为 GOF 外观（FASCADE）。
……容器没有涵盖必须对所有组件实现的功能	卷入自我侦听(Roll You Own Interception)：提供代码，在每个方法调用前和调用后执行。
……一组组件需要相同的配置参数	配置服务（Configuration Service）：提供一个额外组件，以便其它组件能用于访问配置参数。

可管理组件（Administratable Component）

除了为客户端提供业务功能外，*组件*通常需要完成一些管理、设置或诊断工作，往往正好在*组件安装*之后。若要把（*业务*）*组件*重用为完整的、可重用的包，组件必须能独立完成这项任务——然而，客户端不能访问这种功能。

用于管理公司合同（人、客户、供应商等）的大型***业务组件***由一套协同工作的组件组成。当安装时，它们依赖于一些外部资源，诸如数据库表。你不希望组件的用户手工创建表，组件应该自动创建。

这对于某些“安装测试”也一样，它们检查***组件***是否正确安装、所有资源是否都可用、在通常情况下是没有意外错误发生。

在系统运行时，你要访问一些有关***组件***的诊断信息，诸如发生的错误数目（清单）等。

因此

把管理、设置和测试功能实现为（*业务*）*组件*的一部分。只对特殊管理员提供这些功能的访问权限，不对一般用户开放。要保证必要的操作在业务接口中不可见。

变体

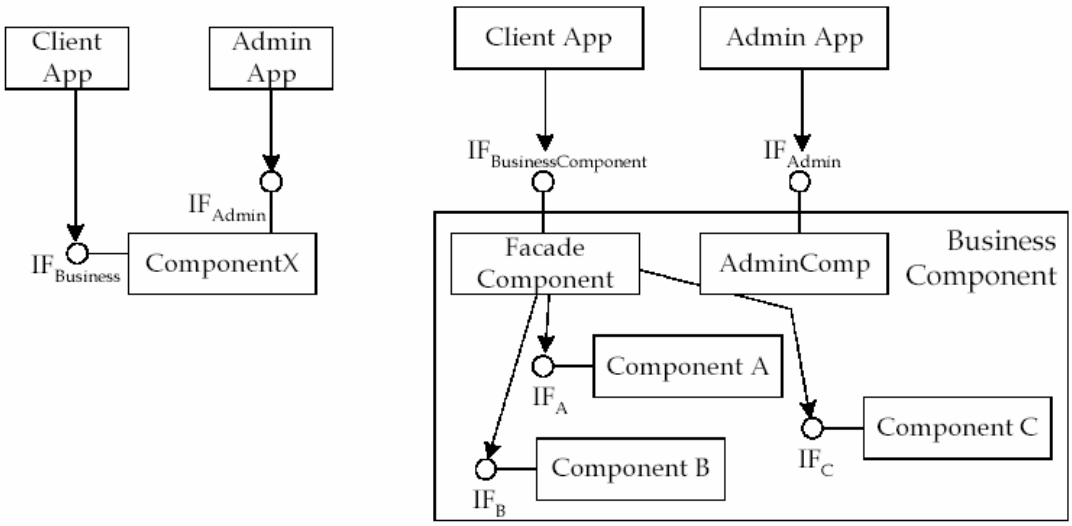
有两种实现这种模式的方法，因为对标准的 EJB 和用 EJB 构建的***业务组件***并不相同。

对于标准的 EJB，你要保证***组件***的两个***组件接口***：一个提供业务功能；另一个提供管理操作。分别定义接口，创建一个公共接口（扩展两个其它接口）作为针对豆(Bean)的远程接口。你的客户端应该只知道业务接口。而且，用*注释（Annotation）*定义安全属性，只允许一些管理员角色访问管理操作。实现这种模式的另外一种方法是使用***多接口***（MULTIPLE INTERFACES）模式：这个组件有一个用于管理的接口，还有一个“真正的”业务接口。

在包含了一组 EJB 的***业务组件***中，提供一个或多个额外的 EJB 实现管理功能。再三强调的是，在*注释（Annotation）*中利用安全设置防止一般用户访问管理豆(Bean)。当对***业务组件***使用***服务组件外观***（*SERVICE COMPONENT FAÇADE*）时，要保证这些操作并非属于这个接口——管理性 EJB 必须单独查询。

在***可管理组件***包含一个小客户端应用很有用，它为访问管理功能提供图形用户界面（GUI）。

在 CORBA 组件模型中，直接定义在***组件***（而非小面 facet）上的操作完全保留用于管理目的。



业务组件（Business Component）

要把满足需求的完整功能“聚集”到一个 EJB 中是很难的（有时在概念上也不可能）。因此，你得有一套 EJB，它们总是用作一组。然而，并没有对这些*组件*的正式分组，因为客户端必须对多个而非一个组件操作，所以其负担很重。

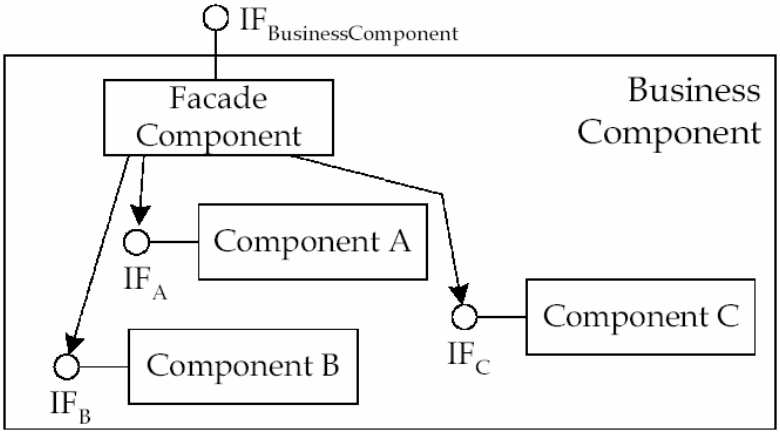
医疗信息系统具有用于管理手术的“子系统”。它要跟踪在手术期间所做的诊断和测量（诸如心跳和呼吸频率），要记录在手术期间用过的材料，也要存储负责治疗患者的医生和助手、并发症等。

除此外，它要提供与某人相关的所有手术的概况（清单），或在某些情况出现时通告其他助手。

很明显，把这些封装进一个 EJB 并没有什么意义，虽然组件的结果集归于一体，且在通常情况下，不可单独使用。一个客户应用要知道哪个组件做了什么工作，也要根据它的用例分别处理它们。其中一个组件的变化通常要求改变所有客户端。

因此

利用内部由几个 EJB 组成，称之为*业务组件*的抽象。把这些*组件*分配和发行成“子系统”。提供一个外观（*Facade*）组件[GHJV94]，作为对整个业务组件的单一访问点，简化客户端访问。这种外观（Façade）组件或利用*弱类型接口*（WEAKLY TYPED INTERFACES）简化重用和集成。



优点

这种模式为组件提供了一个较高级别的粒度，这简化了维护、版本控制和软件发布工作。

缺点和忠告

然而，某些对 EJB 可用机制不能直接应用于***业务组件***，因容器并不了解这种较高级别抽象的任何事情。

变体

外观组件可以非常严格，为客户端隐藏所有其它组件（因不能暴露实体豆(Bean)，这要求由*值对象*做数据交换）。这个外观然后把所有调用委托给相应的组件。可选的办法是利用一个惰性***外观***，即那个外观只对其它组件承担*工厂*角色，允许客户端直接访问业务组件中的所有组件。利用***外观***组件具备在 GOF 的书[GHJV94]中所描述的外观模式的所有结果。

通常，一个业务组件支持复杂过程，该过程包括将对内部几个组件施加的操作当作过程的一部分。利用***外观组件***实现这些过程有两个优点：一是性能得到改进，因为需要的网络跳动数降低；二是外观起到***会话豆(Bean)外观***（SESSION BEAN FACADE）的作用，把客户端从处理工作流和事务完整性中解放出来。

如果使用严格的外观组件，通过对外观组件使用***弱类型接口***（WEAKLY TYPED INTERFACES），进一步分离通信和（正式）依赖性。然后业务组件才互换“请求对象”。用***请求集线器***（REQUEST HUB）能取得进一步的去耦。要简化内部 EJB 集的一致配置，***配置服务（CONFIGUATION SERVICE）***很有用处。

相关模式

这个模式与***会话豆(Bean)外观***不同，虽然两者都利用会话豆(Bean)去分离与一套组件的通信量。***会话豆(Bean)外观***用于增强事务完整性和改善性能，反过来，业务组件是一个较高层的设计工件。

配置服务（Configuration Service）

若有几个相关组件，例如在*业务组件*中，它们通常在配置方面有共性。这些共性是技术的或功能的。在所有*注释（ANNOTATIONS）*中指定全部的*配置参数（CONFIGURATION PARAMETERS）*低效且易于出错。

一个地址业务组件由许多部分组成。因你不想限制重用组件的机会，你做了验证邮编配置的正则表达式。邮编的这种验证由业务组件中的许多不同组件完成。然而，当业务组件被部署为应用的一部分时，在使用中通常只有一个邮编检验表达式——并且业务组件内的所有***组件***都使用这个表达式。

要保证应用正常工作，你不希望为每个组件单独配置这个正则表达式。

另一个例子是国际化，提示有关异常的文本消息要根据所选择的语言而变化，但所有***组件***都应该使用相同文本（和相同语言！）。

因此

提供中央配置服务，参与的组件访问它（通常在初始化阶段）去检索自己的配置。如果必要的话，也能让客户端访问这种服务。

优点

这种模式把你从反复做相同配置的工作中解脱出来。在部分配置可从其它配置中导出时特别有用，更能减轻配置工作。

变体

有两种实现这种模式的基本方法：

- 配置服务可以是真正的组件，这导致一些性能消耗，但这种花销并非严重，因为对配置服务的访问通常只在客户端组件真正物理实例化（通常是创建缓冲池）时发生。好的方面，也允许客户端访问配置信息，提供一种中央化的反映或配置库，也就是说，部分 GUI 也能调整用于配置。

- 另一种可能是使用一种简单的 Java 类。每个组件都创建实例，并访问中央配置文件（通过管理资源的工厂）读取它的配置。这是一个更轻量级的方案，但从客户端不容易访问配置。

你甚至能够让这个配置依赖于逻辑实体或者运行时状态的另一部分。例如，这允许由应用中的一个组件管理美国或德国的地址。当然，因*特征区别*（*DISTINCTION OF IDENTITIES*），在逻辑标识改变时必须重新配置，而在物理实例创建时，不足以重新配置。

请求集线器（Request Hub）

当系统由一套协作的业务组件构建时，通过使用*弱类型接口*可以降低耦合度。然而，组件仍然要知道其他组件负责完成哪个特定任务。而且，这些组件相互依赖对方的接口，包含操作参数、*值对象*类型等等。这损害了系统和业务组件的独立进化。

设想电子商务应用。它用到几个业务组件：个性化和交叉销售引擎、一个目录引擎、订单管理和几个仓库子系统，每个都对应一个业务分支。它们必须协作以实现应用目的。因为所有子系统都由不同的第三方供应商开发，它们都使用基于 XML 的*弱类型接口*。

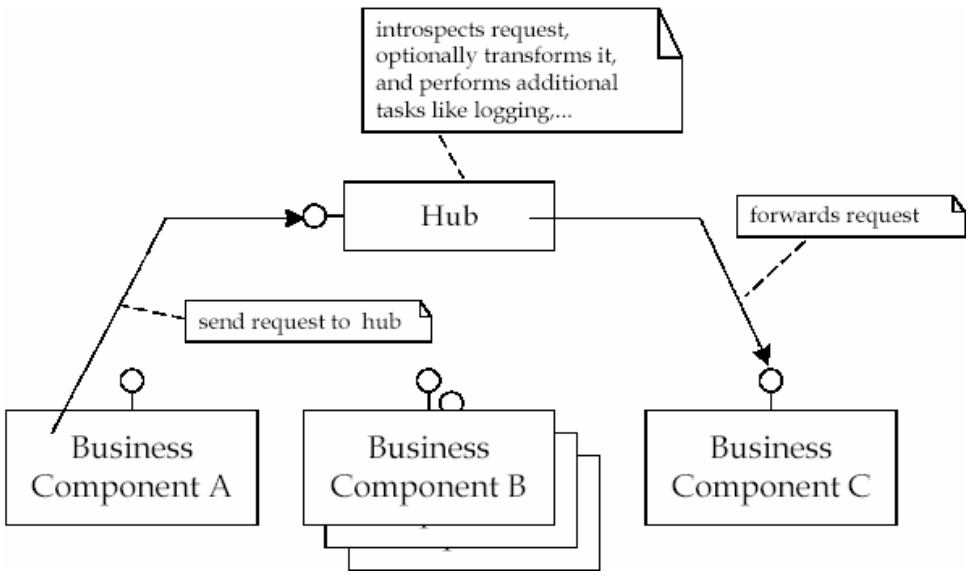
控制组件（也可称作工作流引擎）必须与所有这些组件协同工作。虽然都有基于 XML 的接口，但是他们用的数据结构（DTD）都不同。它们包含相同的内容，但其结构并未标准化。因此，你需要一种方法去调整在业务组件间通信的数据格式。

再有，当这些软件包的新版本引进到系统时，你需要一个方法，怎样才能修改一个接口而不需要改编所有客户端，因它们来自于不同的供应商。

第三方面，如果几个业务组件提供重叠的功能，你也许希望使用另外的组件完成特定的任务，而不用改变所有的客户端。

因此

在系统中提供一个中央*组件*，它接收请求并转发到正确的接收者。除此外，这种*请求集线器*能调整参数、改变数据类型，为已修改的接口提供缺省参数等等。它也能用于屏蔽不同的通信协议。



优点

正如在引言提到的例子，请求集线器特别适于把一套第三方组件“粘合起来”，因为它能用作所有组件的一个中央适配器。如果有些组件使用非 EJB 通信策略，如 SOAP 或低层的 Sockets，这特别有用。

缺点和忠告

集线器是一个中央系统组件。要完成其工作，它要知道系统的哪一部分负责哪个任务——当系统中的责任变化时，集线器必须要更新。

附加信息

理解集线器能、但不应该用作容器的替代品很重要，容器负责技术事宜如负载平衡、事务、错误恢复和安全。实现这些并非琐事（对请求集线器也一样），这便是容器引入组件系统的原因所在。然而，在请求集线器中实现技术关系是可行的：例如集中化的日志或更复杂的安全策略能在此实现，如果容器不支持它们的话。简单形式的应用级负载平衡也能通过向不同组件转发请求来实现。

集线器并非一定是系统崩溃的单点。因为它是无状态的，所以也能够容易复制到不同的机器上——只要保证不同集线器是按同样方式配置即可。

变体

若不需要太复杂的请求转移，集线器可以是通用组件，因为***弱类型接口***通常是反射的，因此路由策略能够配置而不需要编程。若这样不可行，考虑使用***策略***（模式）使之足够灵活。

除了上述提到的责任外，假如组件不可用，集线器也能负责伪技术事宜，如集中化的日志，异步请求转发，消息存储等。它提供一个中央位置，在此能配置所有事宜，如使用策略等。

请求的转发可以仅靠调用操作同步发生，或基于异步通信模式，例如使用 JMS。

相关模式

因为每个请求都要穿过请求集线器，这是钩住***侦听者***的一个好地方，这样，请求集线器可以是***卷入自我侦听***的一个替代品。

这本质上是*中间人（Mediator）*或*代理（Broker）*模式的实现。

卷入自我侦听

在创建大系统时，通常遇到一些要在项目中实施而不能在容器中实现的需求。容器通常不支持这些特性，因为在容器意义上它们并非技术性的，从开发者看来，它们也非功能性的。

在组件中固化编码这些需求是不可接受的——特别地，假如你要能在项目中实施某些策略，或你想在组件被实施后能改造某些需求。

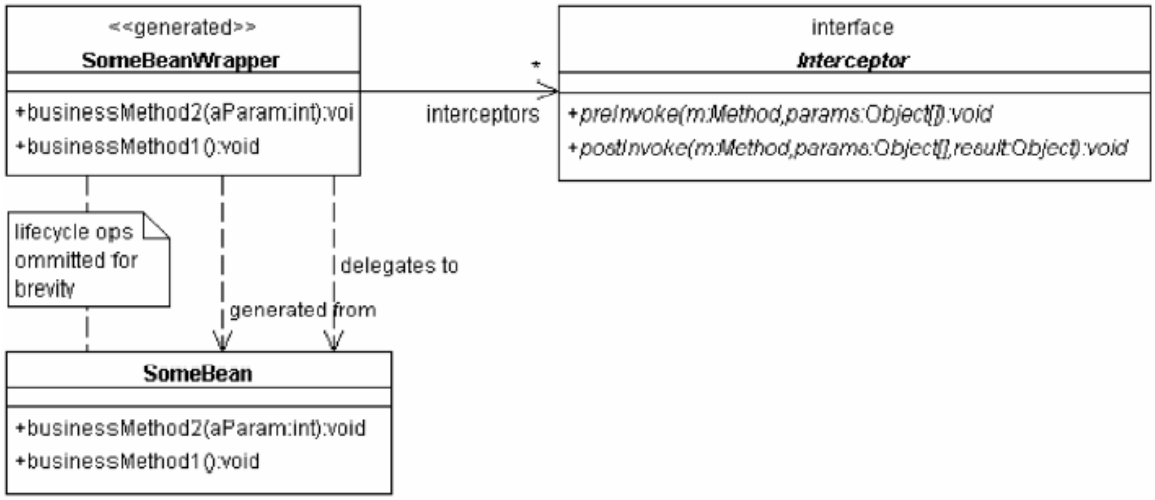
典型例子是资源访问决策。EJB 标准只允许你“固化编码”用户的哪个角色能访问某个操作。但你不能编码诸如“只有负责人与创建那个实体的人在同一个团队，实体才能被访问”这类情形。对于组件实现者而言，将相应的检查合并到每个操作中并在必要时抛出一个异常是令人灰心、易于出错的任务。

如果你希望在组件实现改造这些检查，你必须轮流改变每个组件，这通常不可接受。

这种问题的另一个典型案例是性能探测。在一个网站上，你或许要记录哪个操作在什么时间被谁调用，例如用于计费或仅仅是统计；你或许对一个操作执行了多长时间感兴趣，以便测量站点性能。

因此

创建自己的*侦听*接口，利用代码产生创建一个封装器豆(Bean)实现，它将每个业务操作的前后处理（操作）委托给一个或多个侦听器。利用 Java 的反射去排列被调用操作的信息。封装器然后将调用转发给真正的实现类。



优点

这种模式允许你采用由“平常的”组件程序员创建的任何组件实现，并插入由基础架构人员创建的素材。

缺点和忠告

因为有时候笨拙地集成在开发环境中，代码产生总是有些问题。然而，在此描述的情形并不是很成问题，因为*组件安装*总是需要代码产生，而且产生的代码简单、结构好。

附加信息

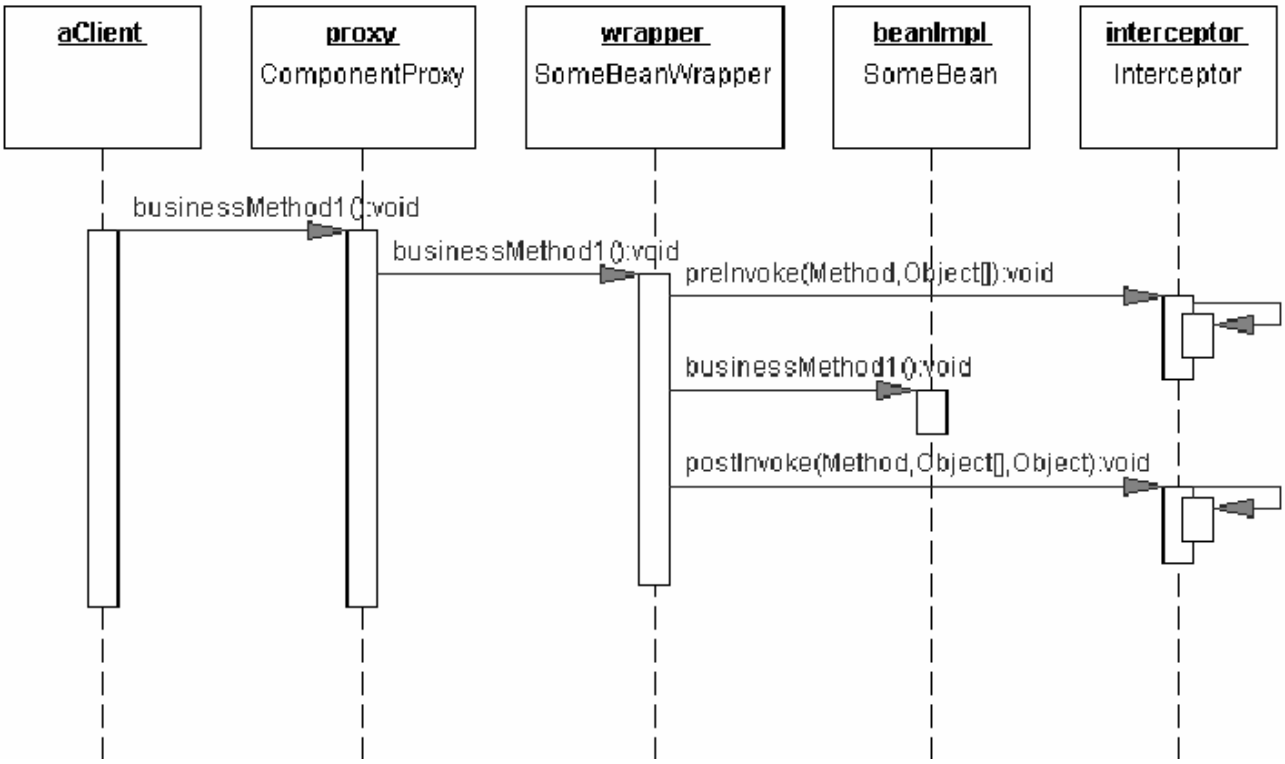
一个特殊的代码产生器检测 *SomeBean* 的实现类，并创建一个封装器类，按下述步骤实现每种操作：

2、 调用每个可用侦听器的 *preInvoke()*

3、 调用原来的豆(Bean)实现上的原始业务操作

4、 调用每个可用侦听器的 *postInvoke()*。

这种过程在下述序列图中予以说明：



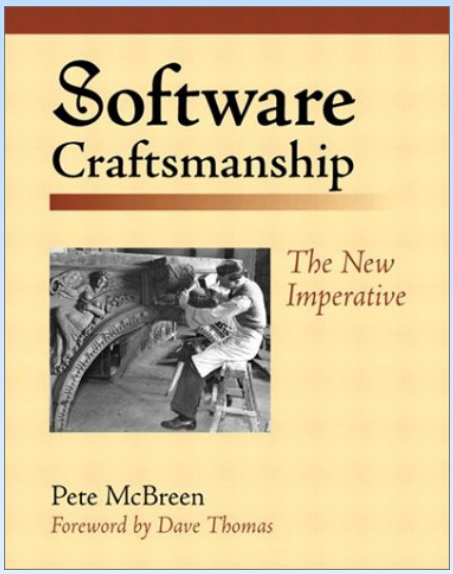
对于部署，要把产生的封装器类指定为在*注释（Annotations）*中的实现类、而非原始豆(Bean)实现，因方法调用必须达到封装器以允许它调用侦听器。因为它有与正常实现一样的操作，这总是可能的。

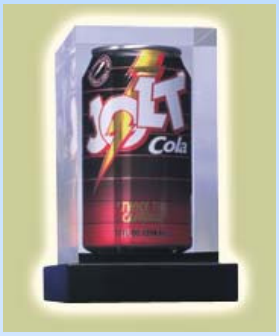
要允许部署者灵活地增加或删除侦听器实例，需要的侦听器必须在***配置参数***组件中对部署指明。封装器在实例化时要检查这些参数。

相关模式

因为通常情况下，在*业务组件*中将对许多*组件*使用相同的侦听器，*配置服务*（模式）对此有很大的帮助。

《软件工艺》





2002 Jolt Awards 获奖书籍

Pete McBreen

翻译：UMLChina 翻译组

工艺不止意味着精湛的作品，同时也是一个高度自治的系统－－师傅（Master）负责培养他们自己的接班人；每个人的地位纯粹取决于他们作品的好坏。学徒（Apprentice）、技工（Journeyman）和工匠（Craftsman）作为一个团队在一起工作，互相学习。顾客根据他们的声誉来进行选择，他们也只接受那些有助于提升他们声誉的工作。

中文译本即将发行！

一种在线拍卖管理的模式语言（上）

Paulo C. Masiero 著，vicky wei 译

查看评论

摘要

在线拍卖提供了一种销售货物的绝妙手段，因而日益普及。在线拍卖的好处包括：减低交易成本、销售通过常规渠道难以销售的货物、扩大潜在客户圈。本文为 web 方式拍卖系统提供一种模式语言。该模式语言以 10 种分析模式支持本应用领域所需的所有基本功能。本模式语言的开发基于几个现有的 Internet 拍卖系统，其应用可指导开发者分析这类系统。

关键词：模式语言，软件模式，在线拍卖

1 介绍

电子商务近来迅速扩张，赢得了越来越多的商家和顾客。基于 web 的信息系统使电子商务发生了巨大的变化。作为电子商务的一种，在线拍卖日益普及。在线拍卖系统可从不同的角度（如投标定义、拍卖规则说明、交易定义等）进行分类。有几个原因促进了其应用，其中包括扩大潜在客户圈的能力、低廉的交易成本、较短的交易时间等。

商业资源管理的模式语言主要处理商业资源的交易、位置和维护。在线拍卖管理的模式语言正是在这种环境中开发出来的，用以帮助开发那些通过 web 方式拍卖来进行交易管理的系统。在线拍卖管理的模式语言的开发是基于三个现有 Internet 拍卖系统——DBay，iBazar，Arremate.com。由于缺乏对这些系统的文档或源代码的访问权限，我们基于用户界面对这些系统作了逆向工程。然后基于我们自己对拍卖系统的理解，并参考现有的关于拍卖系统的手册，我们开发了一些模型以描述系统的功能。通过发掘三个系统的共有功能，我们从这些模型抽象出这些模式。他们展示了在线拍卖系统中的功能特点，而非设计或实现问题。本文中提供的范例也是从这些模型中抽象出来的。

结构和模式语言的范例都采用 UML 描述。这种语言建议了可能在实例化时为必需的几种基本属性和方法。轮询方法由于可能增加复杂度却不能带来类模型的有效性，因此被省略了。我们使用了由 Larman 提出的“操作”概念。系统的输出操作用“!”前缀标注，而输入操作则用“?”前缀标注。当调用消息被送往一个对象集时，我们使用“#”前缀，而不是用一个独立的实例。

2 模式语言概述

图 1 展示了语言模式间的关系。10 种模式可被划分为两大类。第一类，必需模式，覆盖一种资源可以成功拍卖所必需满足的要求。这些必需模式在所有拍卖系统中都必须实现。第二类，可选模式，覆盖拍卖系统的一些有需求但并非必须满足的特性。图 1 显示了各种模式的一个应用顺序。表 对模式语言作了总结，可用作快速参考。

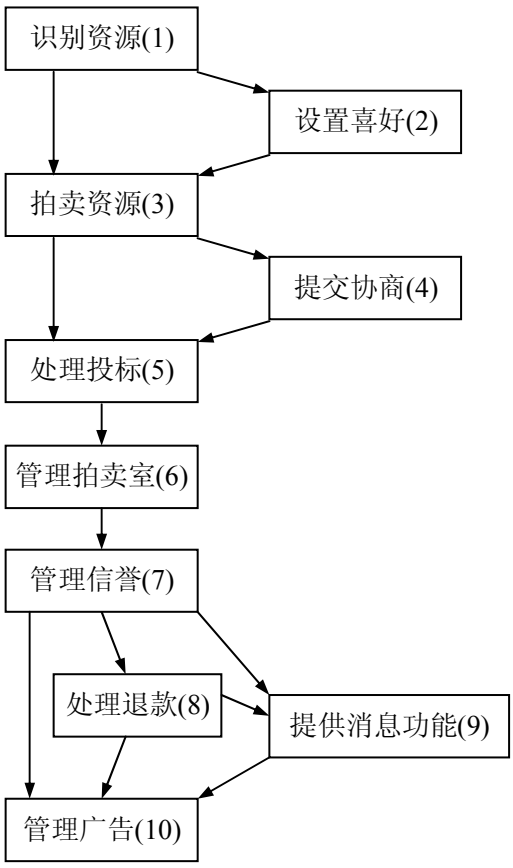


图 1 组成模式语言的模式间的关系

模式	问题	方案
识别资源(1)	你如何表现通过系统拍卖的商业资源？	3.1
设置喜好(2)	你的系统如何为顾客建立感兴趣的资源类？	3.2
拍卖资源(3)	你如何处理通过系统执行的不同资源拍卖？	3.3
提交协商(4)	你的系统怎样支持拍卖交易各方之间的协商和沟通？	3.4
处理投标(5)	你的系统如何处理各类拍卖的各种投标？	3.5
管理拍卖室(6)	你的系统如何管理参与拍卖过程的拍卖室所需遵循的规则？	3.6
管理信誉(7)	在相关方的可信度方面，你的系统能提供怎样的评估/证据？	3.7
处理退款(8)	对于不应当的收费，你的系统提供怎样的退款方式？	3.8
提供消息功能(9)	你的系统怎样管理传递给顾客的消息？	3.9
管理广告(10)	你的系统怎样管理拍卖广告？	3.10

表 1 模式语言总结

3 在线拍卖管理的模式语言

3.1 识别资源

环境

你的商业系统处理一种特定的商务交易：拍卖。这些交易包括产品或服务交易，如机票、旅馆、旅行包、CD、书、艺术品、收藏品等都可称作资源。这些商业资源的管理通过特定的系统实现，这些系统可实现个人/企业顾客之间的交易。拍卖系统支持在不同类型的消费者间进行的各种资源的交易。在某些情况下，拍卖是销售一种资源的最好办法，例如，卖方无法把握该资源的真实价值，如易腐烂品、艺术品、收藏品、库存过剩的产品等。

问题

你如何表现通过系统拍卖的商业资源？

约束

- 商业资源通常拥有相同的属性或质量。对于管理这些资源的组织而言，维护每一特定资源的信息非常重要。
- 商业资源必须被划分为几大类。例如，在一个计算机产品的拍卖室，资源可以首先按其本质属性进行划分（硬件或软件），其次可按类别进行划分（输入设备、CPU、笔记本），再次可按品牌或生产商划分。这种分类有助于迅速找到所需资源。这证明了划分为两个甚至更多的类是正确的。为每一资源保存这些属性所需的空间以及所得到的冗余让人不快。然而，这种划分可能增加处理时间——因为一个类将需要索引到另一个类，这种索引需要系统仔细处理。这些在优化系统性能时必须认真考虑。

结构

图 2 显示了“识别资源”模式的类图。

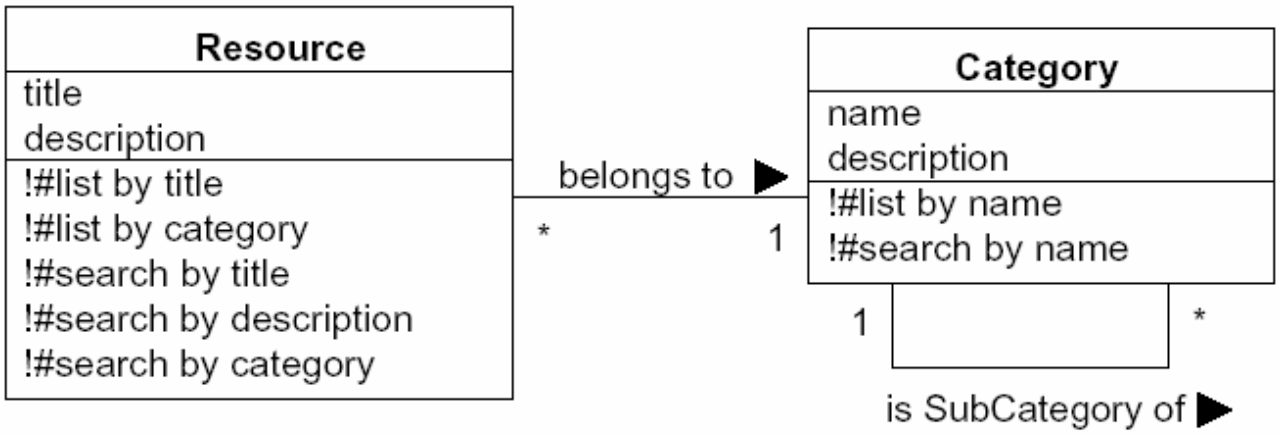


图 2 “识别资源”模式

参与者

- 资源：表示由系统管理的商业资源，定义其所有重要的特性。其他属性可根据特定的资源实例，通过如“属性模式”的方法添加。
- 分类：表示资源分组的类别，是资源通过分类进行陈列和查找的基础。通常，至多允许定义三层分组。

范例

图展示了 iBazar 所采用的一个“识别资源”模式范例。这是这种模式的直接应用，仅仅添加了一个属性（照片）。[0..1]后缀意味着拍卖物可提供也可不提供照片。Arremate.com 和 eBay 采用了一种不同的分类方法，如**变体**部分所示。

图 3 识别资源模式范例

变体

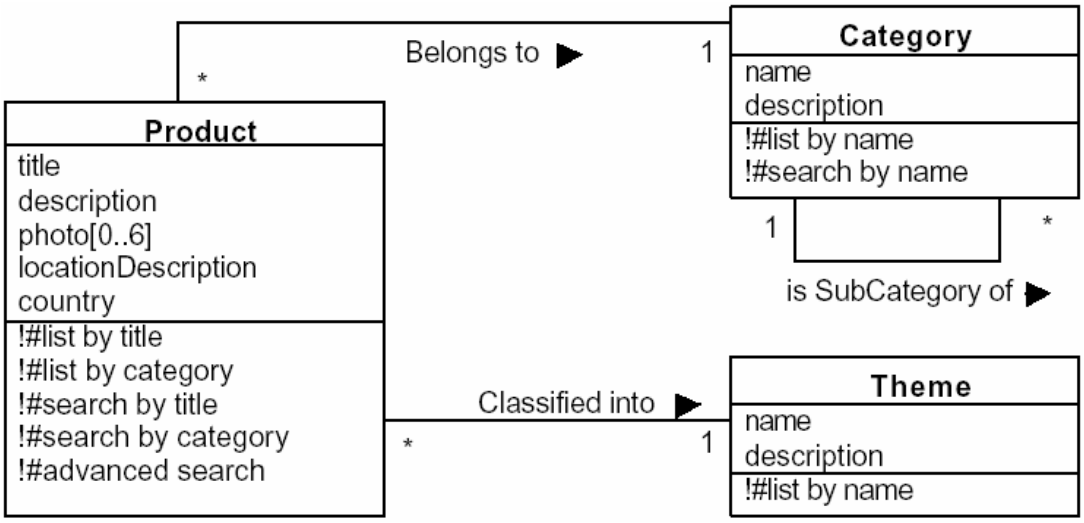


图 4 识别资源模式的变体范例

本模式是“识别资源”的一个变体[2]。最初提出的模式拥有更少的属性和一些其它的操作。但如果存在多层次的分组，我们就会遇到关系交错的类别对象[12]。

为改进分类的有效性，资源可从多个角度进行分类，以增强其灵活性。Johnson 和 Woolf 称这种变体为“多类型对象”，类似于我们在主模式中所提到的“交错的类别对象”。为必要的类别创建一个新的类，

并关联到“资源”类。这种模式在 eBay 找到一个范例。“产品”有两个分类类型类，“分类”和“主题”，如所示。“主题”类将具备某些共性的资源组织起来，例如收藏品、足球、烹饪，但这种划分独立于“分类”。属于不同“分类”的“产品”可能会属于同一种“主题”，例如，厨房搅拌器可以被包含在“电气设备”“分类”中，同时也可以被包含在烹饪“主题”中。所以，这种情况我们的标准模式不可解决，因为标准模式只允许有从属型的子类。

相关模式

本模式也是“管理声誉”模式的一种应用[6]。

后续模式

在使用“识别资源”模式后，可尝试使用“设置个人喜好”。若不适用，使用“拍卖资源”模式。

3.2 设置个人喜好

环境

你的系统需要处理已经被标识并分类的资源。在有些系统中允许顾客对某些资源类进行组合——因为他们可能对这些资源最感兴趣。这种选项可帮助定义消费者购物卷宗。因此，通过观察消费者购物卷宗，我们可以为他们提供一些建议，以便推进拍卖的繁荣。

问题

你的系统如何为顾客建立感兴趣的资源类？

约束

- 需要为顾客提供一种方便的方法以了解他们感兴趣的资源；
- 收集顾客卷宗信息是为特定资源类型分发广告的一个好思路。这种策略不仅对拥有这套系统的组织而言很重要，同时对顾客而言也是方便的——使他们的注意力可聚集在更可能有兴趣的资源上。
- 向顾客提供建议有好处也有坏处，在好坏之间获得平衡非常重要，因为顾客也可能被这些持续不断的提议惹恼。广告应当吸引顾客经常访问拍卖系统网站。

图 5 展示了“设置个人喜好”模式的类图。

结构

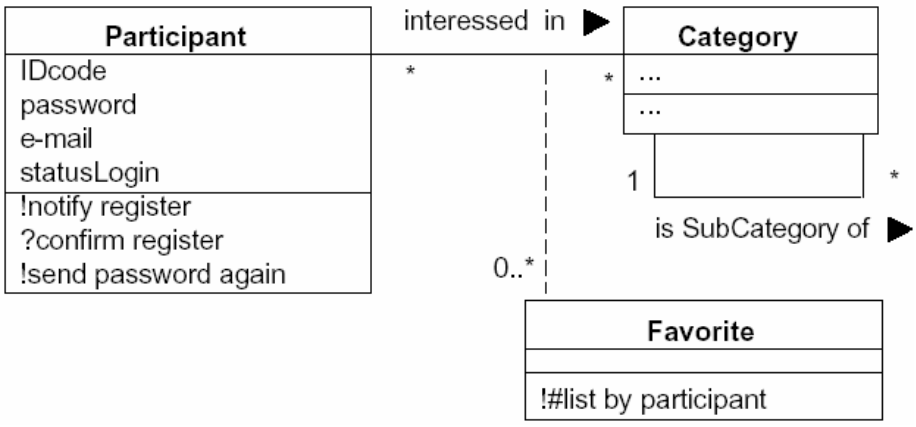


图 1 “设置个人喜好”模式

参与者

- 参加方：表示参与拍卖或获取某项资源的相关方：个人或组织。
- 分类：如“识别资源”模式中所述。
- 喜好：表示参加方根据兴趣所选择的资源分类。

范例

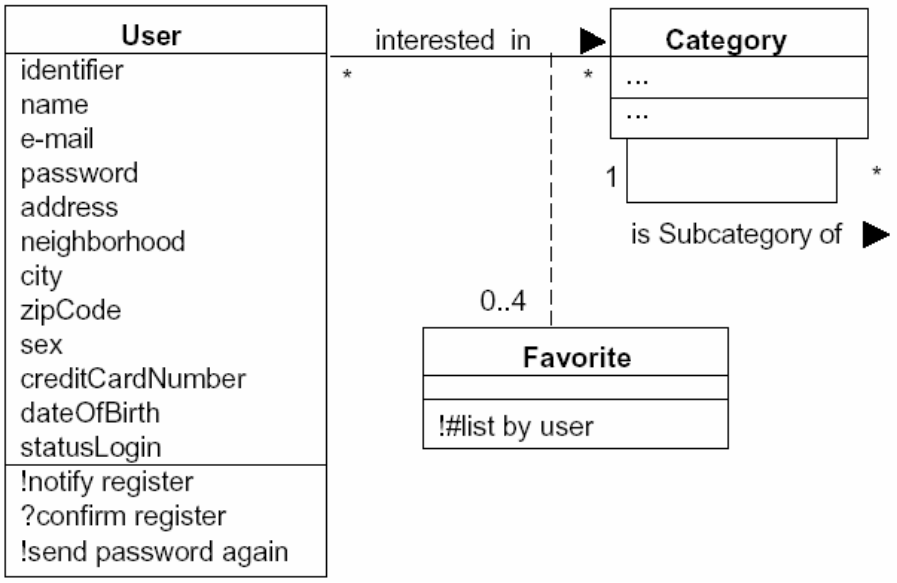


图 6 “设置个人喜好”范例

图 6 展示了在 eBay 在线拍卖网站应用的一个“设置个人喜好”模式范例。“喜好”类保存顾客最感兴趣的资源分类。eBay 只存储最多 4 种分类。而在 Arremate.com 和 iBazar 则并不存在首选顾客分类的概念。

后续模式

在使用“设置个人喜好”模式后，使用“拍卖资源”模式。

3.3 拍卖资源

环境

你的系统用于处理已标识并分类的资源。资源拍卖可视为一种财富交换，通过这种交换，由一方拥有的财富变为由其他方拥有。当通过拍卖来完成交易时，拍卖方将资源提供出来销售，其他几个竞拍方则期望以最低的价格购得。有几种不同类型的拍卖可以提供给拍卖方选择，每一种都有自己的规则决定哪个竞拍方将成为赢家。

问题

你如何处理通过系统执行的不同类型的资源拍卖？

约束

- 参加方的信息必须存储起来，以便为交易过程和系统功能提供信息。
- 重要的是必须提供多种拍卖方式，观察那些对特定类型资源最适当的，通过某种拍卖类型所拍卖的资源数量、效率、约束，考虑到交易发生的环境：Internet。
- 在某些情况下，拍卖方可能希望改变关于拍卖甚至拍卖类型的信息。有必要为信息变更建立规则以免相关方的利益受损。
- 有必要建立拍卖取消的规则以免相关方的利益受损。

结构

图 7 显示了“拍卖资源”模式的类图。

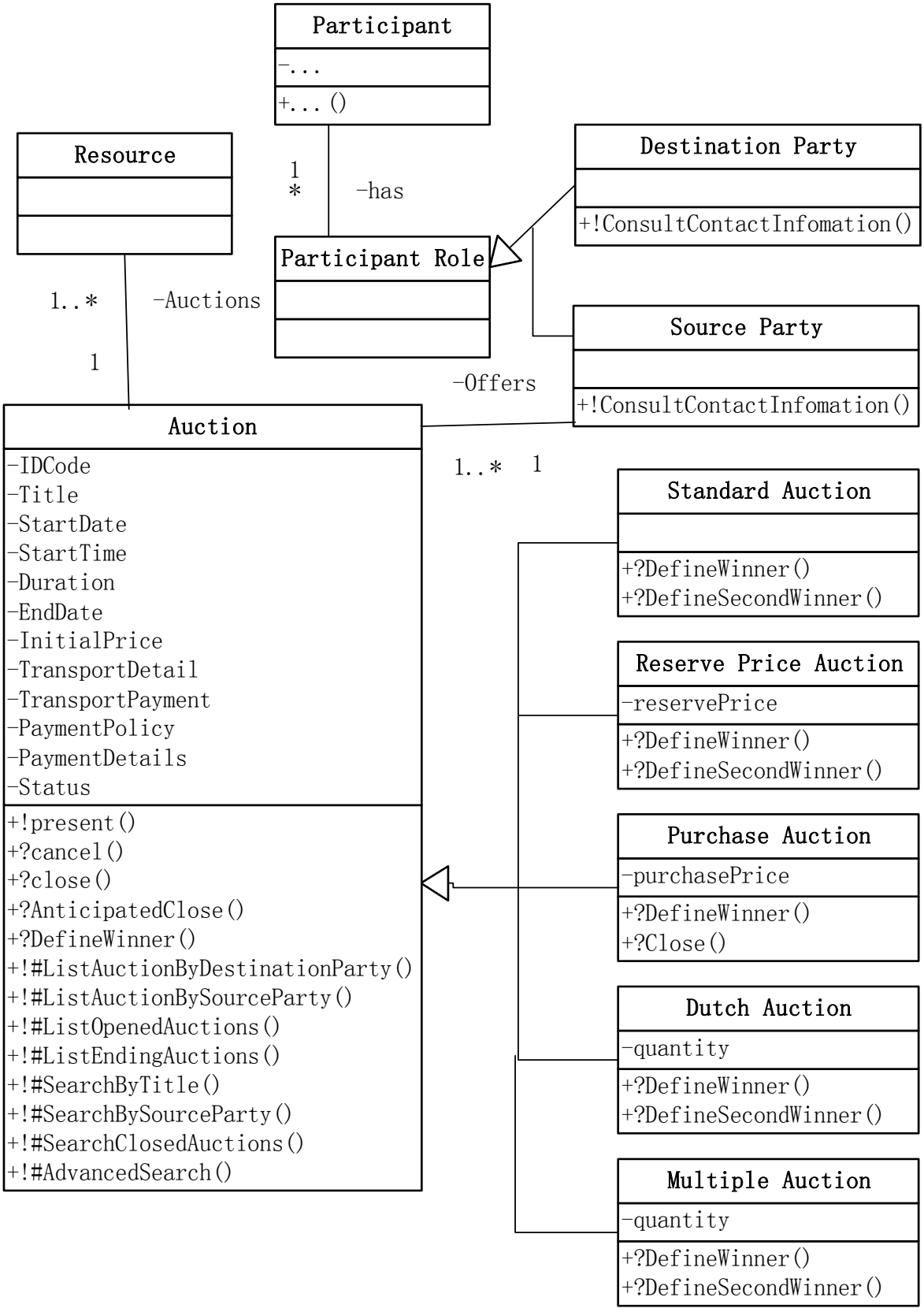

Participant
-...
+...()
Resource
1
*
-has
Destination Party
+!ConsultContactInfomation()
Participant Role
1..*
-Auctions
1
-Offers
Source Party
+!ConsultContactInfomation()
Auction
-IDCode
-Title
-StartDate
-StartTime
-Duration
-EndDate
-InitialPrice
-TransportDetail
-TransportPayment
-PaymentPolicy
-PaymentDetails
-Status
+!present()
+?cancel()
+?close()
+?AnticipatedClose()
+?DefineWinner()
+!#ListAuctionByDestinationParty()
+!#ListAuctionBySourceParty()
+!#ListOpenedAuctions()
+!#ListEndingAuctions()
+!#SearchByTitle()
+!#SearchBySourceParty()
+!#SearchClosedAuctions()
+!#AdvancedSearch()
1..*
1
Standard Auction
+?DefineWinner()
+?DefineSecondWinner()
Reserve Price Auction
-reservePrice
+?DefineWinner()
+?DefineSecondWinner()
Purchase Auction
-purchasePrice
+?DefineWinner()
+?Close()
Dutch Auction
-quantity
+?DefineWinner()
+?DefineSecondWinner()
Multiple Auction
-quantity
+?DefineWinner()
+?DefineSecondWinner()

图 7 “拍卖资源”模式

参与者

- 参与者：表示打算拍卖或获取某项资源的相关方（组织或个人）。有两个特别的类：资源方或目标方（见后文）。注意同一个参与者在不同的拍卖中可分别扮演这两种角色，这通过使用角色模式[3，7]实现。StatusLogin 属性说明参与者是否已提供其 IDCode 和 password，因此可有效地参与拍卖活动。重要的是注意到设计时需按照某种安全策略对密码作特殊的处理[19]。但在此模式语言中我们不考虑这种问题。这个类有一些基本属性，但也可添加其他的属性——这取决于参与者的特定实例。
- 参与者角色：表示在某次特定的拍卖中由参与者扮演的角色，可为资源方或目标方（见后文）。
- 资源方：表示在交易完成前拥有资源的相关方。
- 目标方：表示投标并可能在拍卖完成后成为资源所有者的相关方。
- 资源：如“识别资源”模式中描述。
- 拍卖：表示资源拍卖过程详细细节的抽象类。此类包含执行一次拍卖所需的所有基本细节，与拍卖类型无关。此类控制了变更规则的实现，例如何种信息可以变更。还有其他的一些属性来保证这些规则的实现，例如拍卖相关信息可以变更几次，拍卖可以发生变更的期限，当没有投标时拍卖是否可发生变更等。这些属性作为拍卖运转的参数，存储在“拍卖室”类（在“管理拍卖室”中描述）中。根据特定拍卖实例所要求的细节和功能，还可为将其他属性添加到此类以及“拍卖室”类。这种行为可通过“策略”模式实现。
- 标准拍卖
- 预约价格拍卖
- 采购拍卖
- 荷式拍卖
- 多种拍卖

范例

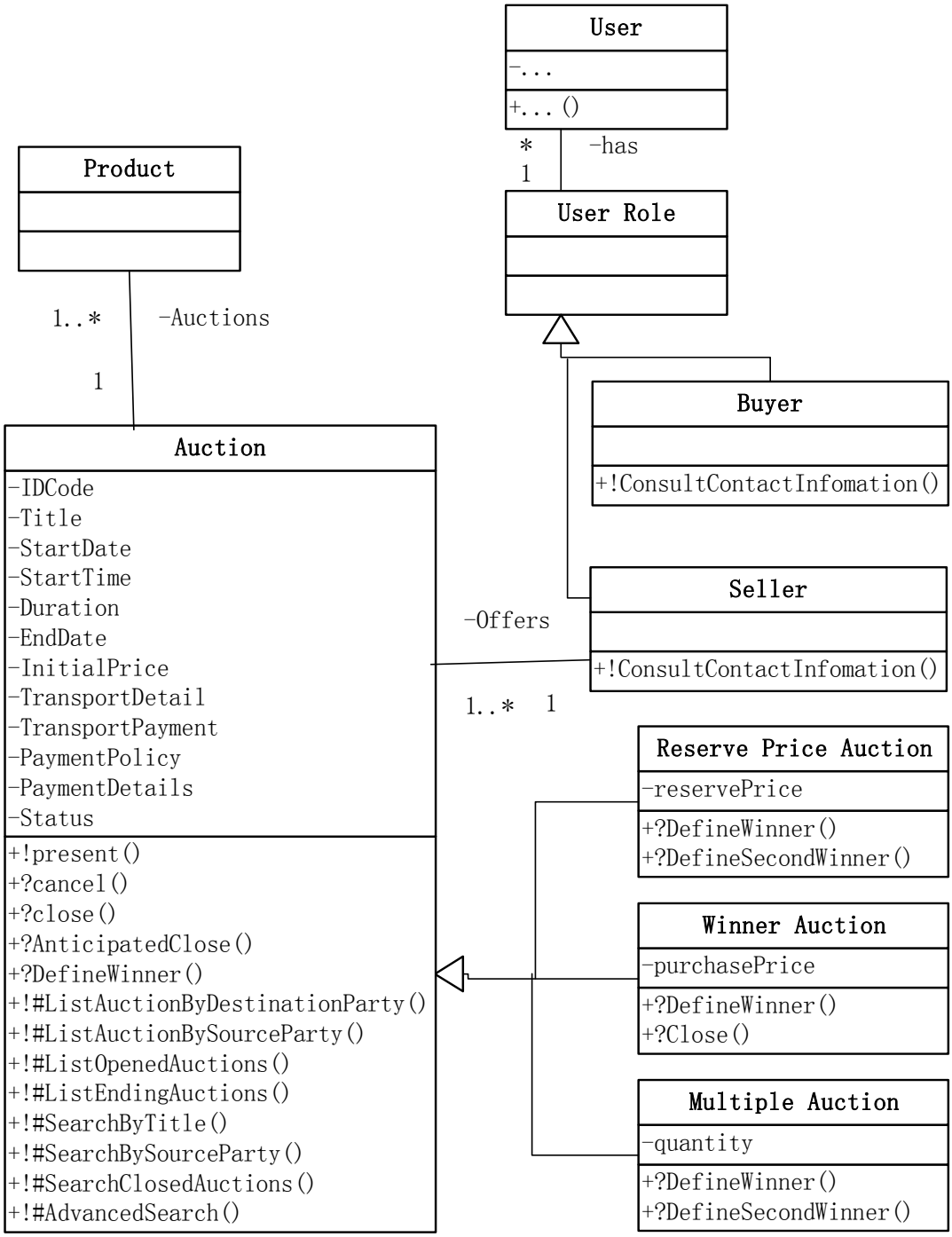
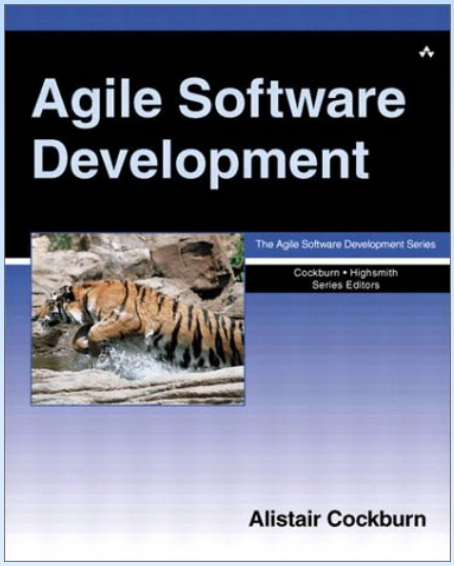
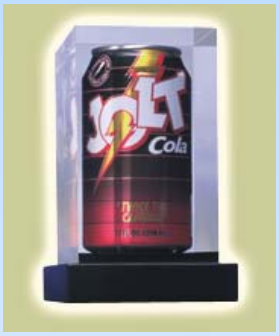


图 8 “拍卖资源”模式范例

《敏捷软件开发》





2002 Jolt Awards 获奖书籍

Alistair Cockburn

翻译：UMLChina 翻译组 Jill

我是一个好客人。到达以后，我把我的那瓶酒递给女主人，然后奇怪地看着她把酒放进了冰箱。

晚餐时她把酒拿了出来，说道，“吃鱼时喝它，好极了。”

“但那是一瓶红酒啊。”我提醒她。

“是白酒。”她说。

“是红酒。”我坚持道，并把标签指给她看。

“当然不是红酒。这里说得很明白...”她开始把标签大声读出来。“噢！是红酒！我为什么会把它放进冰箱？”

我们大笑，然后回顾我们为了验证各自视角的“真相”所作的努力。究竟为什么，她问道，她已经看过这瓶酒很多遍却没有发现这是一瓶红酒？

中文译本即将发行！

CRC 建模方法

——跨跃开发者与用户之间的交流障碍

Scott W. Ambler 著，huang_shen 译

查看评论

本文描述了用 CRC（class responsibility collaborator——类、职责、协作者）建模方法界定用户需求的过程。CRC 建模方法是一套能够使开发者与用户密切合作，从而界定和理解业务需求的方法，这种方法高效、易用。

类、职责、协作者卡片（以后简称 CRC 卡片）

CRC 卡片（Beck & Cunningham，1989；Ambler，1995）是一种标准的索引卡片。如图 1 所示，这种卡片被分成三个部分。

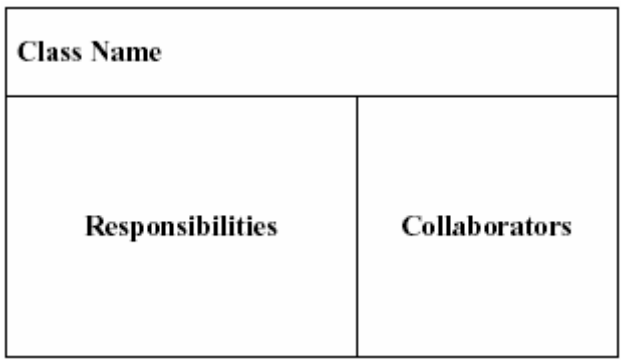


图 1 CRC 卡片的布局

类是一组相似对象的集合。而对象则是与要开发的系统相关的人、地点、事物、事件、概念或报告等。让我们来看一个例子，图 2 展示了一套物流管理系统的类。其中包括：**库存明细（Inventory Item）**、**订单（Order）**、**订单明细（Order Item）**、**邮政地址（Surface Address）**。类的名字标注在卡片的顶部。

职责是类所知的和所做的。换句话说，就是类的属性和行为。以客户类为例，客户有名称、客户编号和电话号码，这些都是客户所知的。而客户可以订购产品、取消订单和付款，这些是客户所做的。类所知的和所做的组成了类的职责。类的职责被放置在卡片左侧的列中。

有时，类有一项职责要履行，但它没有足够的信息去完成。这时，这个类必须与其他类合作来完成这项工作。例如，某**订单**对象要履行一个职责——计算本订单对象的合计。虽然订单对象知道**订单明细**对象是它的一部分。

但是它不知道订单明细对象的具体订购数量（这是**订单明细**所知的），也不知道这些订单明细对象的价格（这是**库存明细**所知的）。为计算订单的合计，**订单**对象必须和**订单明细**对象协作。先计算出每一**订单明细**对象的合计，再通过累加，得出整个订单的合计。而计算每一**订单明细**对象的合计，就需**订单明细**对象与**库存明细**对象协作，从而检索出本对象的价格，再乘以订购数量得出合计值。类的协作者被放置在卡片右侧的列中

CRC 模型

CRC 模型是反映整个或部分应用或问题域的 CRC 卡片的集合。正如这篇文章所介绍的，CRC 模型最普遍的作用是收集和界定面向对象应用的用户需求。图 2 是为一套物流管理系统设计的 CRC 模型。假想这些 CRC 卡片是摆放在桌子上的。注意这些卡片的位置：有协作关系的卡片被放置在很近的位置，而没有协作关系的卡片则相距较远。

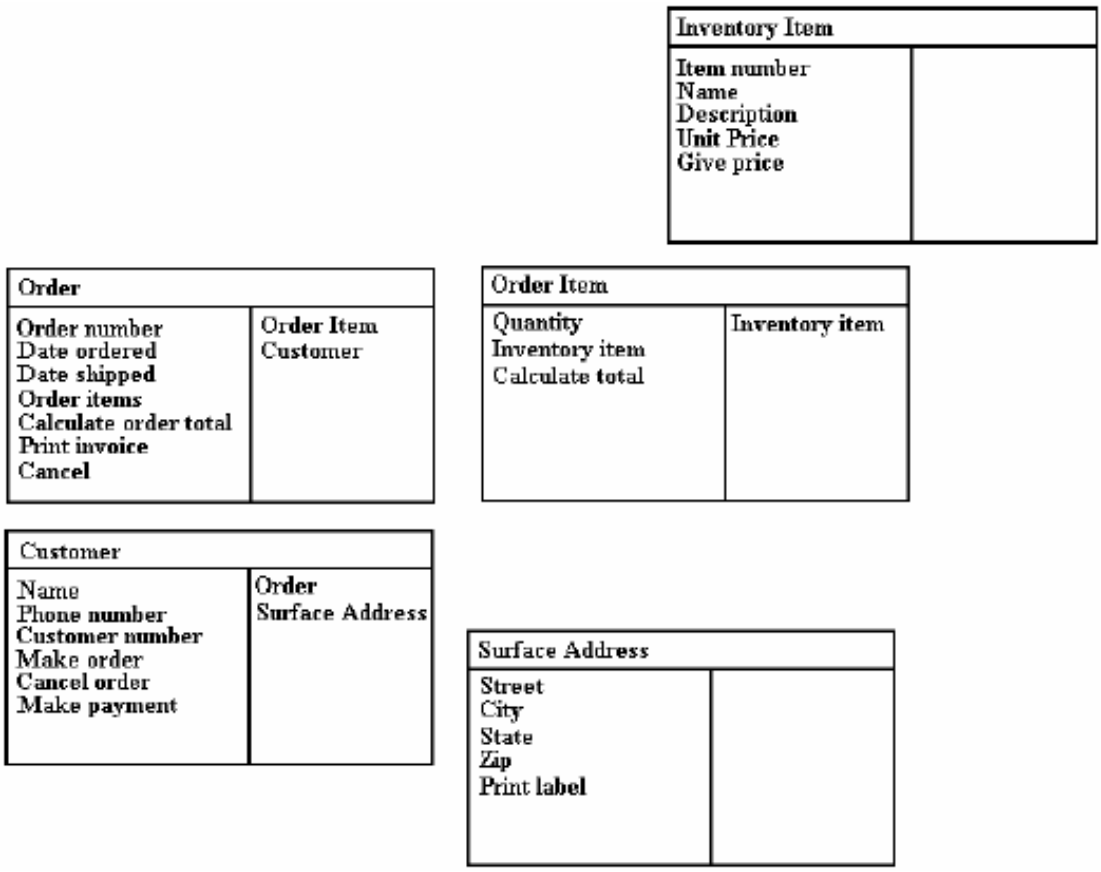


图 2 一套简单的物流管理系统的 CRC 模型

CRC 模型是由业务领域专家（Business Domain Expert）创建的，在 CRC 辅导员（Facilitator）的指导下和一两个记录员（Scribe）的帮助下完成的。CRC 建模研讨会的筹划和进行由 CRC 辅导员负责。下一节，我们将详细地讨论 CRC 建模过程。

CRC 建模简介

那么，如何创建 CRC 模型呢？CRC 建模共包括六个步骤，它们是：

1. 组建 CRC 建模小组；
2. 准备房间；
3. 头脑风暴（brainstorming）；
4. 解释 CRC 建模的技巧；
5. 用迭代方式执行 CRC 建模的几个步骤；
6. 用例场景测试。

组建 CRC 建模小组

在 CRC 建模过程中有四种角色：

1. **业务领域专家（BDE）**。他们是系统真正的用户，有时还包括对于本问题域有多年经验的开发者。无论如何都不要忘了，这些用户成天与这些业务打交道——他们具备丰富的业务知识。最好有四到五位 BDE 参与到 CRC 建模的工作里来。如果参与工作的 BDE 少于四位，你将缺少足够的业务知识和经验。反之，如果有六个以上的 BDE 参与进来，你又会觉得太多了，他们各司其职。好的 BDE 熟悉业务，具有逻辑性的思维，善于沟通。而且在开发过程中，他们会有效地利用时间。

2. **辅导员**。研讨会的组织者。辅导员的主要职责是：联系一切与建模相关的事宜；确认卡片是否填写正确；在建模过程中提出一些相关的问题；决定在什么时候需要创建原型以及保证成功地创建原型。辅导员需要有良好的沟通能力和技术实力。辅导员一般是有经验的会议组织者。

3. **记录员**。在整个过程中应该有一两个记录员参加进来。记录员记录下那些 CRC 卡片不能反映的详细业务逻辑。例如，在**订单**卡片上我们可以指出，订货可以通过某种途径申请到折扣，但我们不能将具体的业务逻辑记录在卡片上。而这类信息就需要记录员来记录。在研讨过程中，记录员不能过于积极，但可以问一些问题来确认自己记录的业务规则或处理逻辑是否准确。记录员必须能够听得很清楚，需要有良好的口头和书面沟通能力，需要能很快理解业务逻辑。

4. **旁听者**。出于培训的目的，你可能希望一些人以旁听者的身份参加研讨会。而这些人只是坐在一边，并不真正地参加讨论。

准备房间

为准备 CRC 建模用的房间，你需要做以下事情：

1. **预定会议室。**会议室里应该有一个活动挂图或白板，以便在做头脑风暴或创建原型时，可以在上面写些东西。活动挂图的好处是你可以在上面记录下你的草图。白板的好处是能很快画出草图并且可以在上面修改。

2. **准备必要的用具。**几包索引卡、几支白板笔。如果要做用例场景测试，还需要一个海绵小球。

3. **一张建模用的桌子。**你需要一张足够大的桌子，从而使你的人能够有地方工作。

4. **为记录员准备写字台和椅子。**将它们放在房间的后方或一边。使记录员处在旁观的位置，但又必须让他们能很容易看到工作的进展情况。

5. **为 BDE 准备足够的椅子。**谁都希望能坐下来工作，所以你要确保椅子够用。

6. **为旁听者准备位子。**如果有旁听者，就把他们安置到房间的后面。这是因为旁听者不参加研讨，放在后面就可以了。

头脑风暴

头脑风暴是产生想法的技巧，而不是评估想法的技巧。你的团队将用头脑风暴的方法来确定和理解所要开发应用的需求。对于别人的想法，应给予鼓励。在头脑风暴过程中，一些可能遇到的问题可以促使想法的产生，例如：

谁是系统的用户？

他们将用本系统做什么事？

为什么我们做这些？

为什么我们用这种方法去做？

本系统支持哪些业务需求？

客户想要什么以及将要向我们提出什么要求？

业务是如何变化的？

我们的竞争对手正在做什么？为什么做这些？我们能不能做得更好？

我们是不是需要做这些？

如果我们开始着手打草稿，那我们该如何去做？

如果说我们过去在这方面很成功，那么我们将来还会这么成功吗？

我们能不能将几项工作合并成一个？我们是真的想要这么做吗？

人们的工作将会受到什么样的影响？哪些影响是我们希望做到的，哪些是要避免的？

人们在做他们的工作时，需要哪些信息？

工作是否进行到最敏感的地步？

有些琐碎的任务，我们是不是可以让系统自动完成？

是不是有人正在完成哪些系统无法处理的任务？

系统内部能否很好的衔接？

系统是否支持联合作业？这对系统的运行有没有妨碍？

我们的用户有没有足够的技术能力来使用本系统？需要对他们进行什么样的培训？

我们公司的战略目标和工作重点是什么？本系统是否与之相符？

我们如何做得更快？

我们如何降低成本？

我们如何做得更好？

解释 CRC 建模技术

当头脑风暴一结束，辅导员应该开始描述 CRC 建模过程。这通常需要十到十五分钟。整个描述过程一般应包括做几个示范性质的 CRC 卡片。因为，实践是最好的学习方式，所以由辅导员带着 BDE 做几张 CRC 卡片，能帮助他们尽快地熟悉 CRC 建模过程。

用迭代方式执行 CRC 建模的几个步骤

这些 BDE 或站或坐在大桌子周围填写 CRC 卡片。他们花大部分时间是坐着来讨论系统的，但 BDE 偶尔也会站起来，以便更全面地观察他们创建的模块，使他们能更好地把握应用的全局。CRC 建模的步骤包括：

寻找类

寻找职责

定义协作者

定义用例

在桌子上排列卡片

寻找类

寻找那些属于系统一部分或与系统有交互的事务

问自己：“是否有客户存在？”

跟着钱走

找出那些系统产生的报表

找出那些在系统中用到的屏幕界面

迅速地创建接口和报告类的原型

提取出三到五个主类

迅速地为一个类填写新卡片

用一两个词描述类

类名应是唯一的

寻找职责

自问什么是类所知的

自问什么是类所做的

当你定义了一项职责，问自己它属于什么类？

有时候，我们找到的职责可能不在本系统里实现，这没关系

有许多职责需要类之间相互协作完成

定义协作者

协作发生在类需要那些它不具备的信息的时候

协作发生在类要去修改那些它不具备的信息的时候

任何给定的协作都有一个以上的发起者

有时协作者会做大量的工作

不要推诿责任

在协作的过程中会产生新的职责

定义用例

BDE 将定义用例来作为相关类的职责

做一些头脑风暴

将场景转录到卡片上

移动卡片的位置

有协作关系的卡片应该放置的相互近一些

协作关系越强的卡片，越应该放置得紧密

尽量在开始的时候挪动卡片

将协作关系多，经常需要挪动的卡片放在桌子的中间

真正地移动卡片

在移动过程中，应该确定类之间的关联

用例场景测试

用例场景测试（Ambler，1995；Ambler，1998a；Ambler，1998b）是一个任务过程模式。如图 3 所示，在这个过程中用户主动地参与进来，从而确保用户需求的准确性。基本的思路是，在辅导员的帮助下，业务领域专家（BDE)通过一系列用例来确认创建的 CRC 模型是不是准确地反映了他们的需求。辅导员将卡片分发给每个 BDE，尽量使有协作关系的卡片不在一个人手里（这有时不像说的那么简单）。然后辅导员带领大家将每个场景一次性演示出来。其中，包括六个步骤：

1. **发起新场景。**辅导员提出对场景和所要做的活动的描述。然后，小组判定这个场景是否合理（记住，有些场景是不能被系统处理的）。还要判断第一个场景由哪张卡片负责处理，辅导员将小球传给拿着这张卡片的人。当最后整个场景完成时，小球还应回到辅导员手里。

2. **决定职责应该由哪张卡片处理。**当描述了一个场景以后，或者是一个协作被确定之后，小组应该决定哪张 CRC 卡片要处理这项职责。一般情况下，现有卡片已有这项职责的记录。如果没有，就需要修改卡片。之后，将小球传给拿着这张卡片的人。

3. **如有需要，随时修改卡片。**在两种情况下需要修改卡片：将职责添加到现有卡片上；为职责创建新卡片。如果是前者，你要看看哪张卡片应该在逻辑上包含这项职责，让拿着这张卡片的 BDE 添加上去。如果是后者，将一张空白 CRC 卡片交给其中一个 BDE 让他填写。同时，你应该修改你的原型图（当接口或报告类改变的时候，原型可能也需要改变）。

4. 描述过程逻辑。当小球传到某人的手里，他应该将这个职责的业务逻辑一步一步地描述出来。应该看作：BDE 正在为职责叙述伪代码（高层程序代码）。这往往是用例场景测试中最难的部分，许多 BDE 也许不太习惯一步一步地描述业务过程。遇到这种情况时，辅导员应该帮助他们理顺逻辑。你会发现，通过头几个场景的测试，BDE 能很快地掌握描述过程逻辑的技巧。BDE 描述过程逻辑时，记录员应该将其记录下来（记录员的工作就是为系统记录业务逻辑和规则，这正是 BDE 所描述的）。

5. 协作。当 BDE 描述职责的业务逻辑时，常常会在描述到某一步骤时需要与其它卡片协作才能完成。这很好——这正是用例场景测试所关心的。这时需要退回到第二步。

6. 完成时传回小球。最终，BDE 完成职责描述后，将小球传回给当初传给他的人。这个人可能是另一个 BDE（记得吗？每次当你需要协作的时候，你要将小球传给拿着协作卡片的人），也可能是辅导员（记得吗？是辅导员第一个发出小球，传给拿着最初处理职责的卡片的人）。

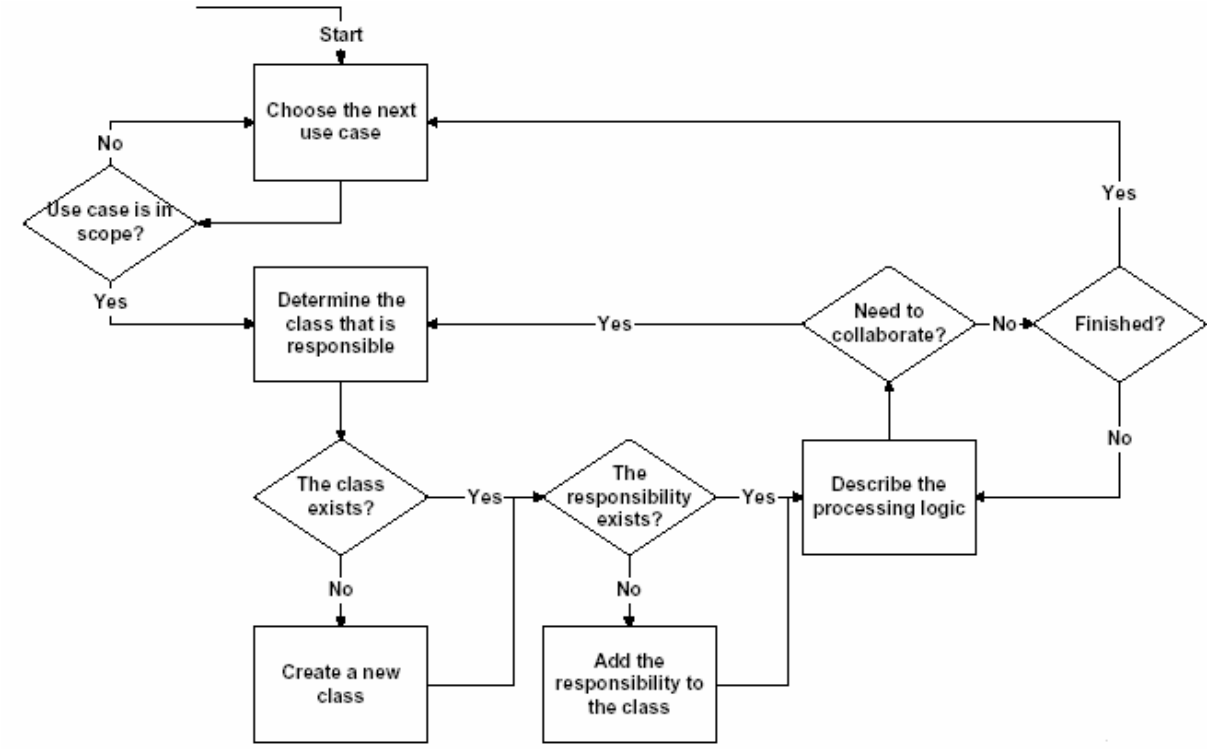


Figure 3. The Use-Case Scenario Testing process pattern (Ambler, 1998b).

之所以这么做，就是要确认你创建的模型是否正确地反映了问题域。如果没有，那么你的项目就陷入了困境。不过，我宁愿在项目的早期发现问题，至少这时还能做些什么。总比到了无法挽回的地步才发现要好。

如何安排 CRC 建模

正如我在第二本书里 Building Object Applications That Work（Ambler，1998a）介绍过的，图 4 中四项技术是完全相互关联的。整体来说，用例和原型的创建是在 CRC 建模之前进行的，CRC 建模之后是类图的生成。原因很简单，用例和用户接口原型没有 CRC 模型详细，而 CRC 模型又没有类图详细。当然，在你进行 CRC 建模的时候，你会画草图或用户界面原型，同时，你也会定义新用例或修改已有用例。总之，你最好从较“简单”的用例和原型开始，到稍微复杂的 CRC 卡片，然后到更复杂的类图。

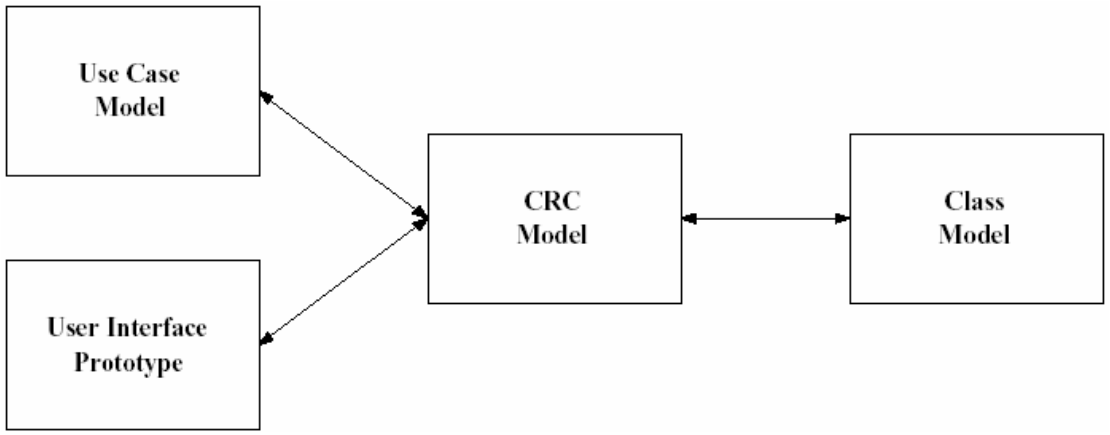


图 4 如何安排 CRC 建模

正如你所知道的，OO 建模可远远不止图 4 所示的这些。图 5 描述了详细的建模过程模式（Detailed Modeling process pattern Ambler,1998b）。其中，框图表示 OO 建模中主要的技术或图；箭头表示它们之间的关系；箭头的指向代表了一种输入关系。例如，我们可以看到活动图是类图的输入。在每个框图的右下角都有字母来表示谁将参与这项技术或图的工作。它的含义很明了：U＝User 用户，A＝Analyst 分析员，D＝Designer 设计师，P＝Programmer 程序员。带下划线的字母表示谁是这项工作的主角。例如，用户是 CRC 建模的主角，而设计师是创建状态图的主角。

有意思的是，从图 5 可以看出，面向对象建模过程同时具备了宏观上的连续性和微观上的反复性。OO 建模的连续性体现在，当你从左上角看到右下角，是一个很清晰的从需求收集到分析，再到设计的过程。而反复性体现在，每项技术在驱动其它某项技术的同时，又被那项技术所驱动。换句话说，就是模型之间是相互推动的。

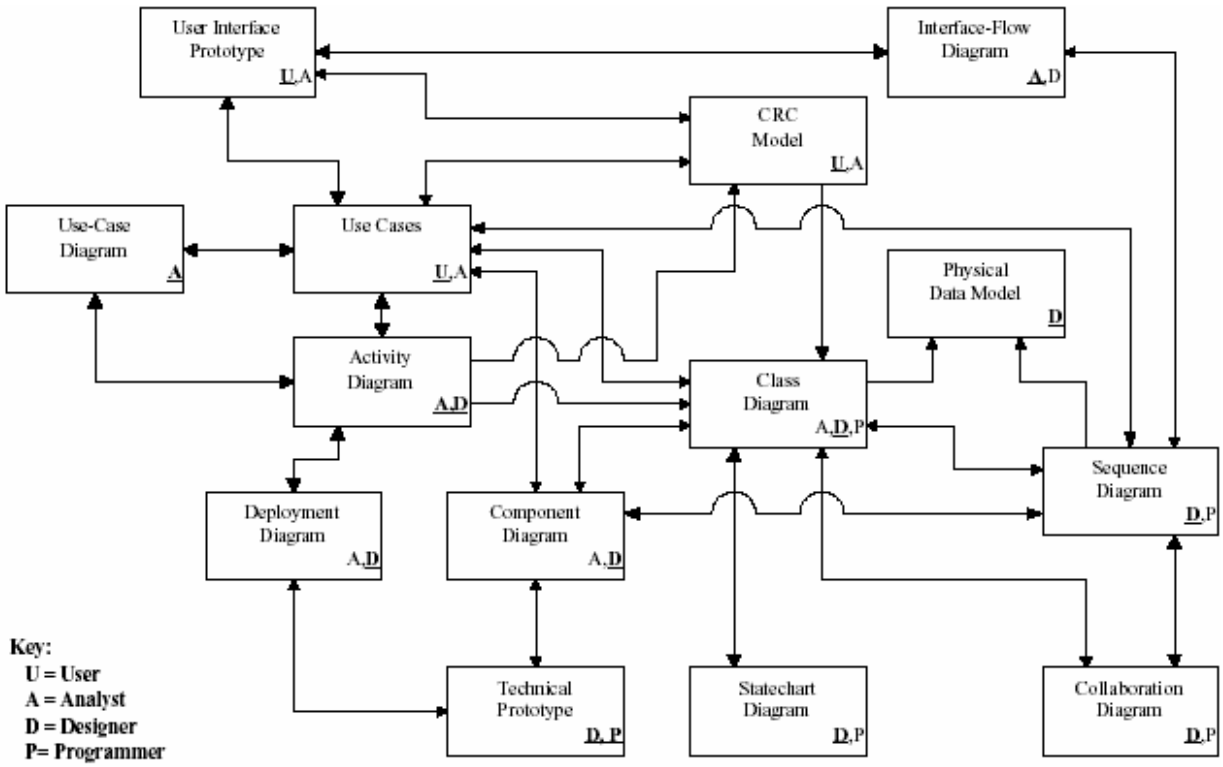


Figure 5. The Detailed Modeling process pattern.

从连续性的角度看，图 6 展现了“可交付成果驱动可交付成果”方法的过程模式（Deliverables Drive Deliverables approach process pattern Ambler，1998b），指明了在构建阶段，你处理可交付成果的整体顺序。这说明了图 5 和图 6 是相互补充的，而不是相互矛盾的。从图 5 可以看到，整个建模过程是从用例和 CRC 建模这些关心用户需求的技术开始，到顺序图、组件图这些面向分析的技术，然后是设计技术和最终编码。图 6 中的箭头代表了一种印证关系。例如，通过用例来印证用例图，而用例又是通过顺序图来印证。比较有意思的是组件图，它经常是通过其他组件图、类图或用例图来印证的。

作为题外话，我要说的是，在 UML（Rational，1997）里，记录下联系各个分离模块之间的关联信息的轨迹，能使你的工作具有可追踪性。可追踪性在测试时是非常关键的。

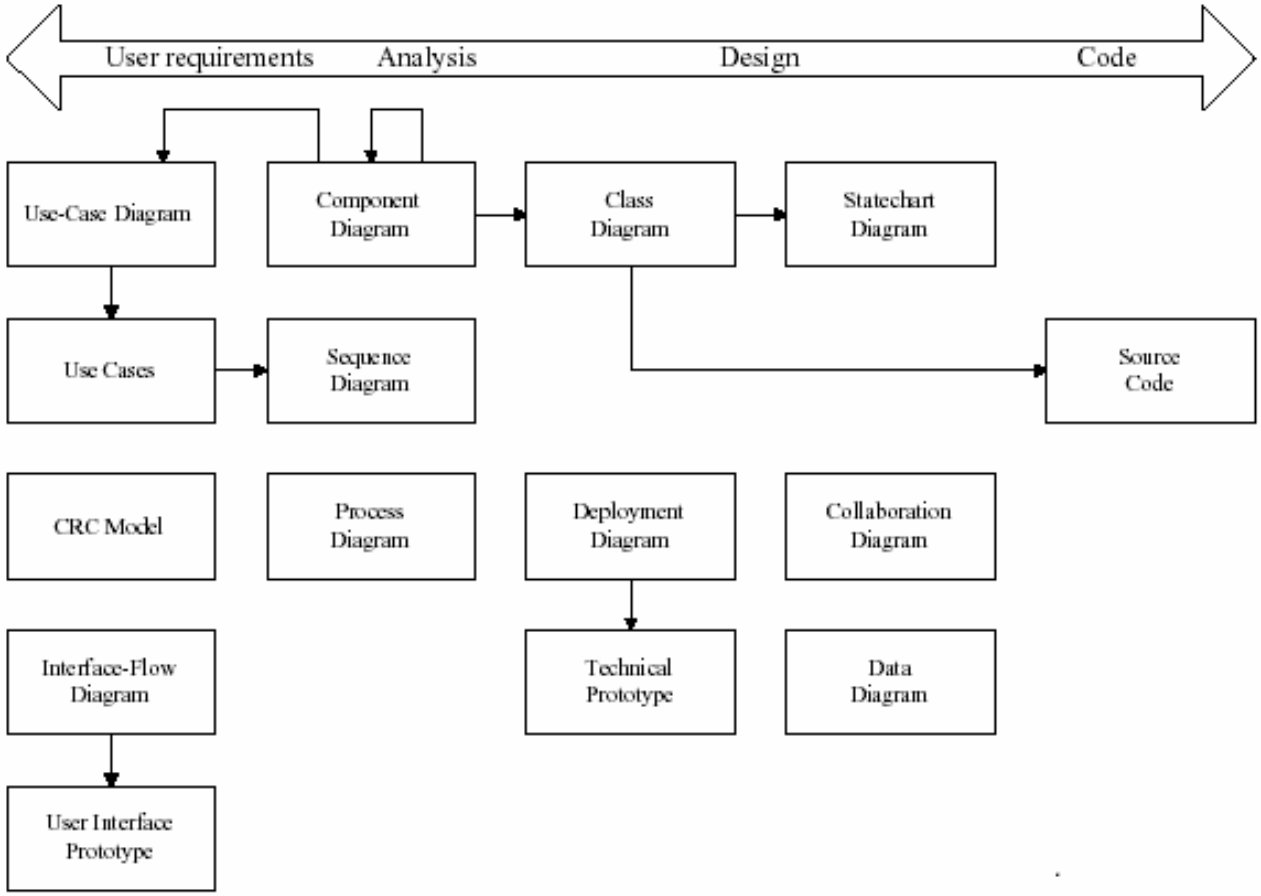


Figure 6. The Deliverables Document Deliverables process pattern.

总结

CRC 建模的技术和技巧

以下的技术和技巧（Ambler，1995）能使你的 CRC 建模工作更有效：

1. **在建模研讨会的几天前，发出会议议程。**议程包括：研讨会在哪儿，什么时候举行；谁参加，角色是什么；会议的目的，以及如何做准备。议程能帮助人们做好会议的准备工作，从而提高研讨会的效率。

2. **将 CRC 的定义在明显的地方展示出来。**我喜欢做一张非常大的 CRC 卡片，上面写着对类、职责和协作者的描述。将其放在房间的前方，以便所有的人都能看到。这样做的目的是，在 BDE 建模的时候有一个参照物。同时，提醒他们如何填写 CRC 卡片。

3. **使用术语。**你应该尽一切可能地使用问题域的术语，避免使用哪些新奇的或自造的术语。这样可以使 BDE 很容易理解他们在做什么。

4. **简单的事情简单做。**90 年代早期，出现了一种促使 CRC 建模过程自动化的趋势。不过，我没看到这有多大意义。就方法而言，CRC 卡片已经是非常高效的。用 CASE（计算机辅助系统工程 Computer Aided System Engineering）工具创建类图，你需要更多的建模技能，而 CRC 卡片不需要。因此，要用恰当的工具工作。

5. **创建原型。**在 CRC 建模过程中，总是会需要你画出一些用户界面和报表的草图。这能帮助人们很直观地理解他们在说什么，从而提取出有价值的东西，投入到开发过程中。

6. **可能要用上几天。**对于较大的系统，你可能要多开几次研讨会。没关系，这很正常。越大的系统，越需要花更多的精力在理解和界定需求上。

7. **得到管理层的支持。**你公司的管理者必须认识到，在写代码之前，了解问题域和为解决问题域建模是非常有好处的。许多管理者只是在口头上支持，但真正要做的时候，往往会忽略需求收集和建模，而是热衷于写代码。

8. **将 CRC 建模包含到你的系统开发周期中。**CRC 建模是一种高效的、用来界定和整理用户需求的技术。为什么不用呢？

9. **只与一线人员一起创建 CRC 模型。**我的经验是要与一线人员一起创建 CRC 模型，因为这样你可以很容易的深入到非常细节的地方。而对于那些关心宏观部署的行政管理者，用例更适合他们。还是那句话，要用恰当的工具工作。

CRC 建模的优点

CRC 建模有很多的优点（Ambler，1995），包括：

1. **由专家来做分析。**业务领域专家（BDE）都非常了解问题域，由他们创建模型。也许这是确保你得到正确信息的最好方法。

2. **使用户更多地参与进来。**因为用户积极地参与建模，定义的模型更让他们满意，从而使开发工作更有效。这样用户也愿意加大对项目的投入。

3. **打破沟通障碍。**用户和开发者肩并肩地共同创建 CRC 模型。

4. **简单、易用。**找个房间，召集一组人，填几张卡片。就因为它简单，所以你可以在 10 到 15 分钟的时间里解释什么是 CRC 建模。

5. **不会让用户产生顾虑。**用户往往会担心，自动化操作会取代他们，使他们失业。而谁会担心几张卡片会让他们丢掉工作。

6. **成本低、方便。**100 张索引卡片，花不了多少钱。而且，又不占地方，你可以放在你的公文包里。

7. **与创建原型相结合。**CRC 建模与创建原型都是循环往复的过程，而且都需要用户积极地配合。在 CRC 建模过程中，绘制一些用户界面和报表的草图是很常见的。

8. **可直接推导出类图。**CRC 模型和类图有许多相似的方面。在许多情况下类图可以简单的看作 CRC 模型的超集。

CRC 建模的缺点

当然 CRC 建模也有不足之处，包括：

1. **一些开发者会产生疑虑。**许多开发者认为与用户密切合作是没有必要的，他们觉得自己精通技术，所以也清楚业务领域。这种短见是不可取的，用户成天与这些业务打交道，他们必然比开发者更了解业务。

2. **很难召集用户。**召集人员开会一直是很困难的事。保证人员出席，尤其是几位关键人物的到场，你至少要提前一周做准备。

3. **CRC 卡片有一定的局限性。**CRC 建模只是为 OO 应用系统界定需求的一部分。你也要考虑用例、原型和常规需求文档的使用。此外，大多数企业不会接受用一堆卡片作为分析结果的做法。

后记

CRC 建模是一种高效的、用来界定和整理用户需求的技术。与用例和原型相结合，可以直接推导出类图。应用程序开发的目的是解决业务问题，而不是满足那些一心只想玩新玩意儿的开发者的好奇心。要与用户合作，而不是与他们对着干。

我非常欢迎就我的白皮书提出意见和问题。请随时与我联系。我的 e-mail：scott@ambysoft.com。

让我们共同进步。

原文链接：http://www.ambysoft.com/crcModeling.PDF

征稿

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

通俗而且深刻的软件开发思想杰作—UMLChina 推荐

《面向对象项目求生法则》

《面向对象项目求生法则》

Alistair Cockburn

翻译：UMLChina 翻译组乐林峰

Cockburn 一向通俗，本书包括十几个项目的案例

面向对象技术在给我们带来好处的同时，也会增加成本，其中很大一部分是培训费用。经验表明，一个不熟悉 OO 编程的新手需要 3 个月的培训才能胜任开发工作，也就是说他拿一年的薪水，却只能工作 9 个月。这对一个拥有成百上千个这样的程序员的公司来说，费用是相当可观的。一些公司的主管们可能一看到这么高的成本立刻就会说"不能接受。"由于只看到成本而没有看到收益，他们会一直等待下去，直到面向对象技术过时。这本书不是为他们写的，即使他们读了这本书也会说（其实也有道理）"我早就告诉过你，采用 OO 技术需要付出昂贵的代价以及面临很多的危险。"另外一些人可能会决定启动一个采用 OO 技术示范项目，并观察最终结果。还有人仍然会继续在原有的程序上修修改改。当然，也会有人愿意在这项技术上赌一赌。

中文译本即将发行！

轰然巨响

Martin Fowler 著，Jady 译

查看评论

我曾经跟客户谈到他们要我完成的一个对象模型复审。“我们能先给你一些文档，有用吗？”他们问。希望我没有说谎，我给了肯定的答复。两天后，随着一声闷响，UPS 在我的门外卸下了包裹。那是足有 1.5 英寸厚的文档。

我打开包裹，发现那些由 CASE 工具产生的印刷品。显示了一些图、给出了每个类的详尽描述、包括类的所有的属性和操作。这些都有定义。Contract 类定义成“一个很多团体之间的合同”，其 dateSigned 属性定义为“签订合同的日期”。我读遍了这 1.5 英寸厚的文档，最后我却糊涂了。那有很多关于那些对象是什么的描述，但是没有它们真正意味着什么的解释。这种情况已经不是第一次出现了，我想也不会是最后一次。

为什么我们对这些模型或者文档感到反感呢？它们不能执行，我们的客户付钱给我们是为了可以工作的代码，而不是一些漂亮的图画。我们讨厌用模型来交流。想法本来是这样的，图形化的对象模型用来表明对象之间的关系比看代码更清楚，交互图(interaction diagram)表明协作关系比画出很多类之间的调用路径更好。但是，设计文档往往都做得很失败，从而总让我埋在沙发里迷惑不解。

部分问题是人们用来完成这种工作的 CASE 工具。(CASE 工具有两个用途：文档和代码生成，在此只讨论前者。) CASE 工具提倡词典精神。为每个类都做一个条目，你在图上列出每个类和每个属性，为每个用例(Use Case)画一个交互图。它们通过帮你回答“我为所有东西都做了文档吗？”来提倡完全主义。

但是，那个问题错误的。如果所有东西都做了文档，那么就给所有东西都给予了相同的重要性。如果在一个复杂的系统中这样做的话，就会埋没在所有的细节中。在任何一个系统中，都会有些方面更加重要，只要理解了系统的关键方面，就会帮助人们了解更多。文档的艺术就是如何尽可能清晰地对这些关键方面进行文档化。从而你强调这些重点之处，细节由代码负责。

最重要的是，文档要简洁。只有简洁，人们才会阅读并理解它。只有简洁，你才有耐心去更新它。你不能也不应该讲述所有的东西。我的一个朋友跟我谈到他们的一个项目，他们不愿修改类的名字，不是因为需要花太长时间修改代码，而是因为要花太长时间更新文档。当文档成为要对付的一个问题的时候，至少要扔掉一半。

你应该说些什么？

你应该怎样选择要表现的东西呢？恐怕这要取决于你的职业判断。没有什么规则指导你，只有你具备类似一个设计者和交流者的本领。也许这就是为什么总想写下所有东西的原因，因为他们无法决定抛弃些什么。因此，在此陈述的观点只代表目前我个人的方法。

如果你的系统是适当大小，(按 UML 或者 JAVA 的方式)把你的系统分成包(package)。每个包由为某个特定目的而一起工作的类(class)组成。用一个图画出包和它们之间的依赖关系说明系统的总体结构。(在 UML 中，这是类图(class diagram)的一个特殊用法。我经常用这种方式，所以我称之为包图(package diagram)。参看我的书 UML Distilled《UML 精华》。) 考虑你的设计，把这些依赖关系减少到最少的程度，这是将系统耦合降到最小的关键。(没有多少相关的书籍讨论到如何做这件事，我所知最好的就是 Robert Martin 的《用 Booch 方法设计面向对象 C++应用程序》（Designing Object-Oriented C++ Applications Using the Booch Method） 。

对每个包，写一个简略的文档。文档主要是用一些叙述性的文字描述包完成的关键的事情及其实现方法。UML 图可用以辅助说明。画一个类图展示那些重要的类，但不必画出所有的类。对于每个类，只展示关键的属性和操作，决不要全部显示。集中于接口(interface)而不是实现(implementation)。对于包中每个重要的协作关系(collaboration)，给出交互图(interaction diagram)。如果一个类有值得注意的生命周期行为，用一张状态图(state diagram)描述。文档应该足够小到你觉得保持更新它不是什么问题。我通常把文档保持在不超过 12 页。

与每个包都写一些文档一样，展示包之间的协作关系也很有用。因为这可以识别系统的关键用例(use case)，用交互图和一些叙述性的文字说明它们。用一个类图来突出那些关键的类也是很有用的。许多人提倡对系统的每个用例都画交互图，我觉得这样会导致太多的文档，但是，如果你觉得这样有用，而且你觉得保持更新不是什么问题，那么就这样做吧。即使这样，也不应该像突出那些每个人都要理解用例一样，强调的关键用例超过一打。

交流是关键

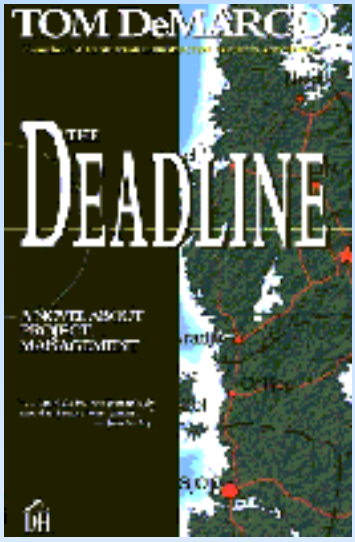
这整篇文章中，我都在强调交流。我已经对 CASE 工具写过一些批判，主要是想说使用一个工具本身并不意味着你就在交流。任何一个工具可以帮助也可以妨碍交流，如何使用决定了结果。

我知道有个项目，他们买了一个多用户的 CASE 工具，任何开发者都可以在他们的工作站上使用。所有的设计都必须放到 CASE 工具上。但正是因为所有的开发者都可以用，并不意味着每个开发者都确实在使用。事实上，很少有开发者去看 CASE 工具里的模型，更少人理解它们。认识到这一点，项目的构架师就用办公室里的一面墙来取代它，在墙上贴了一连串的图来展示系统中半打关键的协作关系。他用不同颜色的对象图 (object diagram) 来突出进展情况。虽然这并不意味着所有的开发者都理解了所有的设计，但现在他们起码能看到重要的要素是什么。

当我开始写这篇文章的时候，我也被我所能谈到的很多东西淹没着。我脑子里充满着大量的轶事和提示。但我知道，要让你去阅读而且记住这篇文章，我只能写其中的一些。我只能挑选一些关键的东西来表达。交流就是这样。良好交流的关键是突出重点。什么东西都讲不是交流。只把那些经过挑选的重要东西传达给你的读者，厚重的文档只会让他们气馁。对信息的挑选是交流最重要的部分之一，这是每一个设计者的责任。

http://martinfowler.com/articles/thud.html

《最后期限》



《最后期限》

Tom Demarco

翻译：UMLChina 翻译组 透明

这是一本软件开发小说

汤普金斯在飞机的座位上翻了一个身，把她的毛衣抓到脸上，贪婪地呼吸着它散发出的淡淡芬芳。文案，他对自己说。他试图回忆当他这样说时卡布福斯的表情。当时他惊讶得下巴都快掉下来了。是的，的确如此。文案……吃惊的卡布福斯……房间里的叹息声……汤普金斯大步走出教室……莱克莎重复那个词……汤普金斯重复那个词……两人微张的嘴唇触到了一起。再次重播。"文案。"他说道，转身，看着莱克莎，她微张的嘴唇，他……倒带，再次重播……

....

"我不想兜圈子，"汤普金斯看着面前的简报说，"实际上你们有一千五百名资格相当老的软件工程师。"

莱克莎点点头："这是最近的数字。他们都会在你的手下工作。"

"而且据你所说，他们都很优秀。"

"他们都通过了摩罗维亚软件工程学院的 CMM 2 级以上的认证。"

中文译本即将发行！

烧毁这本书，别让员工看到—《人件》评论集

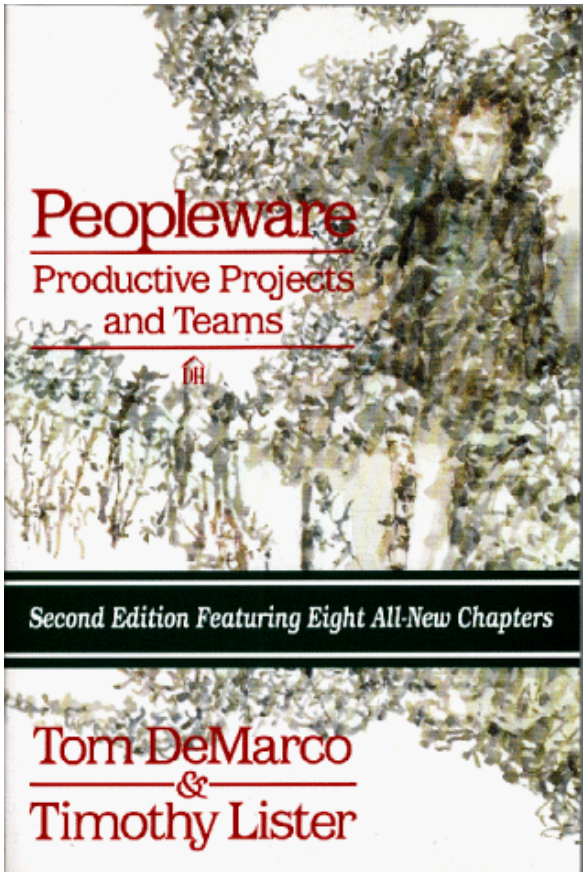
柳林 编译

查看评论

Tom Demarco 和 Tim Lister 的“Peopleware：Productive Projects and Teams”（人件：高效的项目和团队）第一版于 1987 年出版，专门讨论了软件开发和维护的团队管理问题，向传统的管理方法提出了挑战，推崇人本管理思想，给予软件工人自由和信任。和《人月神话》一样，该书现在已经成为软件团队管理的经典之作。**《人月神话》关注“软件开发”本身，《人件》关注软件开发中的“人”。**1999 年 2 月，《人件》第二版出版，增补了 8 章新内容。这些增补的内容视角更加宽广，对比较大型的组织中的团队如何运作进行了探索。《人件》的中文译本将于 2002 年 12 月发行，译者为 UMLChina 翻译组的方春旭、叶向群。

（老板，）在其他人看到这本书之前烧毁它。

——《人件》第二部分



Frederick P. Brooks, Jr

近年来，软件工程领域的一个重大贡献是 DeMarco 和 Lister 在 1987 年出版的《人件》，我衷心地向我的读者推荐这本书。

--《人月神话》20 周年纪念版第 19 章

Edward Yourdon

《人件》在 1987 年出版后，立即成为最畅销的作品，特别是一些组织，开始认识到软件开发问题与程序语言、软件工程方法、软件过程成熟度无关。正如 Bill Clinton 所言，“是人，笨蛋！”你怎样去招聘、面试、挑选最好的软件开发人才？你怎样去奖励与激励他们？你怎样将他们组织为有效的团队？在这些事情中管理扮演什么样的角色？每一个对此都有不同的观点，但很少有人去评价自己的组织，或者问一问自己是否愿意为 Scott Adams 经常在漫画中进行讽刺的组织工作。当人件第一版出版时，我写了一份评论，“我强烈推荐你买一份人件给你或你的老板，如果你是一个老板，那么为你部门的每一个人买一份，并给自己买一份”。这建议在 12 年后依然有效，并且更加热烈——新的版本增加了 8 章，覆盖了关于竞争、过程改进程序、“辞职员工再访”、组织学习、“首要的人”的要领、关于“终极”管理的讨论和一些怎样最好地创建软件开发“群落”的极好建议。

Jason Bennett

这本书说什么？

人件是两位资深项目管理者丰富知识的结晶。他们对管理者在对待下属与工作环境方面的错误尤其深有见地。这本书是从软件项目管理方面入手的，但它对任何 “智力工人”都有价值。也就是说，这些人需要集中精力进行工作而不是间断性的工作。

人件分为很多相互关连的章节。我主要从以下几个主题进行说明：

I. 管理人力资源

II. 工作环境

III. 适当的人

IV. 建造有生产能力的团队

V. 使每一个人工作愉快

“管理人力资源”论述的是如何对待智力工人。这个主题在书的第四页概括：“我们工作中主要的问题并不是现实的技术问题”。最后项目失败并不是他们不能解决技术问题，而是他们自己解散了。员工必须作为一个人对待，而不是一个机器部件。很多管理理论采取“装配线”(如汽车自动生产线)的方式来对待非智力工人。用这种对抗性，无理性的方法对待这些请来进行脑力劳动的人，会破坏工作环境，因为每一个人都是唯一的，不可替代的。你不能从地下挖掘出人力资源。

“工作环境”这章是我最集中，影响最大的部分。D&L 推翻了便宜，开放空间的理论，推翻了应该在工作空间上节约钱，环境不会影响人们工作的观念。想想一个人工的办公费用仅为他呆在公司工资的 1/20。如果由于糟糕的环境影响的员工的生产率，则在办公环境上节约的金钱与员工加班完成任务的贡献相形见拙。事实上，在最好的组织与最差的组织间生产率是 11:1。难道你能工作 10 倍的时间？

“适当的人”的说法后面是说，尽早雇佣最好的员工，因为管理者不可能自己去塑造员工。因此管理者应雇佣最适合于工作的人，而不是满足于一些标准的人。通过允许团队成员相互帮助，保持长期的稳定有助于这种情况。

“建造有生产能力的团队”着力于“整体团队”的思想。这些团队的每一个成员都关注目标，比单个的成员更有生产力。这种团队很难建造，但很容易破坏。如果管理者给予成员足够的自由，就更容易产生优秀团队。专注于质量，精英和保护团队的优秀组织容易产生整体团队。

“使每一个人工作愉快”的目的是：工作很愉快。那不是说理想的工作令人愉快，而是说好的管理者尽他的责任使工作更加愉快。做不同的事件，如战争游戏和飞行项目，并给予员工足够的自由，让他们用自己的方式去完成工作，这是一个使工作愉快的好方法。现在正是这种情形。

什么是好的？

当我读这本书的时候，我感到每一个章节都内容充实。D&L 不停地抛出以数据与自己的经验为支撑的重要观点。每一个与“智力工人”打交道的人都能从本书中获益，书中的观点可以提升工作环境。拿起这本书，读它，并找到改进的内容。你的合伙人将会因此而感谢你。

什么是差的？

我只能说这本书都很好，只是读者群比其它我评论的书更局限于独立承包人，或者其它喜欢按自己的方式工作的人。那些不在桌边工作的人可能与本书关联不多。

那些是写给我们的？

Redhat 更适合这本书，或者… 不，我只是在开玩笑。严肃的说，应该用“开放源代码”更中肯，我相信，你愿意将你的虚拟办公室设定在任何地方。无论如何，一个整体的团队比一群随机组成的团队更有生产率，以团队为导向加强了管理。

Compendiumdevelopments

扩展后的第二版在第一版出版 12 年后才出版，此时第一版已经成为经典著作。

令人不安的前景是，由于本书太过于真实，它的内容要么还没有被读到，要么被忽视了。

作者轻轻地引领读者，偶尔来一些幽默， 完成 34 章的内容。指出了偏离计划的动态原因。

这种书主要是针对于管理，读者主要针对管理者，但它对于被管理者也有用。团队成员可以通过设身处地的对比来帮助管理者和组织。

人是重要的，他们都是独立的个体，不断努力提高自己，他们有坚强的一面，也有虚弱的一面。如果没有被挑衅很少怀有恶意。他们享受教育，建立联系，他们愿意工作，管理应该是帮助他们而不是阻碍他们。

你应该尽可能地去读这本书。

如果你发现你的组织与书中的情形类似，就应该开始改变。

Sue Petersen

对于新出版的《人件》这本书，我除了“买下这本书”这句话外，真不知道还有更好的话。如果你还没有看过这本关于管理实践的杰作，那你应该在以后的学习中得到很多。如果你已经看过，DeMarco and Lister 通过新增的八章内容，从变更过程——人们怎样变更和为什么他们经常不，到“人力资本”和组织学习。在第一版出版后的十一年内他们又形成了两个关于团队自杀的方式。（Dilbert 的老板将感到骄傲。剩下的我们也密切注意）。他们在“大 M 方法论”中的思想可以使 SEI 中的人有所动作，正如帮助我们在过程改进努力中得到真价值。但我喜欢书的这些部分，完全重构了我对团队工作的图象，就是第 28 章。“竞争”。在这章中，他们以运动作比喻来说明我们的行为，然后指出为什么一个合唱团俱乐部更接近于一个“团结的”工作队伍。最后“如你的合唱队不能完美的作为一个整体，你绝对得不到人们的喝彩”。如果你仅仅是只做了你工作的部分，这对于你的公司和顾客没有好处。

Pmnetwork

这是一本关于项目管理的杰作的 1999 新版。基于人们－－你的下属是独特的、不可互换的这个事实。

这本书，最初在 1987 出版，引入了很多管理技术团队的的杰出思想。两个最杰出的是——

“团队自杀”，大公司事实上的政策与官僚习惯阻碍了团队的接合与连续性。还有

“家具警察”表示结构办公空间式的管理方式好象监狱（损害生产率和质量）

我在培训中特别引用一该书中的一句话：“在今天某个地方，一个项目失败了”

这本书的力量是它的预见性和易读性。每章都是独立的短文，尽管在通用观念与认识上有一些交叉，你都可以轻松地在几分钟内读完一部分。

对这个新版本，作者新引入了 8 章内容，而对原版本没有进行大的改变。新引入的每一章比原来的章节有更多的检查倾向，如减少规模，过程改进程序和管理变更。在这些主题中我发现了与原来一样的预见性价值。

正如作者就：“大多数管理者愿意承认他们对人的忧虑大于对技术的忧虑。但他们很少管理人的方面”这本书对一个试图从技术转向到团队管理的专业人士来说，是可理解和有价值的工具，我推荐它。

Mark A. Herschberg

这是我一直喜爱的软件工程书籍。《人件》正确指出软件工程是对人，而不是针对技术。它看到在软件开发过程中人的许多方面，并指出人并不是软件开发机器中简单的小齿轮。

这本书花费了许多的时间讨论团队，使你认识到团队的价值。但它没有一般的管理书籍列出“好团队”的标准，而是论述如何创造一个好团队，并指出它有多难。对于一个尽力去建造一个团队的管理者，这本书帮助他认识到成功的技术与技巧。它并不教你关于开发过程的知识，而是通过教你认识到在软件开发中人的价值去管理开发过程（但它并不仅仅论述管理者，我强烈推荐这本书给每一个人，从低级工程师到 CEO）。

这本书也包含涉及办公环境的章节，并提供证据说明为什么通常的观点并不适用于软件开发。我只学了这些章节，就促使我辞掉了原来的工作！

哟，这本书有一点不同于其它的书，它提供了更多基于多年研究的证据。

这本书改变了人对于软件工程的观点。

Shorewalker

脑力劳动需要办公室。

DeMacro 和 Lister 拼凑了一种理论：管理者应该帮助程序员、设计师、作家和其它的脑力劳动达到一种心理学上称为“灵感”的状态——人们可以静思以达到解决复杂问题的重要飞跃。你开始工作，抬起头来时发现三个小时已经过去。但这需要时间——平均十五分钟——进入这种状态。DeMacro 和 Lister 说现在的嘈杂，狭小的办公室很难使人有十五分钟内不受干扰。也就是说，在世界众多的地方，那些高薪并专注的程序员和创造性的艺术家花费一天时间而不能完成任何真正核心的工作。

Alan Cooper（交互设计之父）

这本美妙的书是关于管理编程项目的，无可否认，它有些偏离我们的主题。但是你要理解它就得将管理项目看成是一种人类的活动过程。作者用与以设计为中心的其它其它书籍同样的隐喻，所以他们的思考过程对我来说是一样的。通过这本书，作者展示通常对于办公室与办公人员的至理名言不一定正确，尤其当工作人员是程序员时，为什么？当我们更深地进入信息经济时代，所有的办公人员将逐渐都象程序员：原始的知识员工。因此，程序员的有意义的工作方法大多数都与设计二十一世纪的商业软件相关。

ROI

投资回报

“15-25%的项目被中途取消或流产…而失败的原因大部分归咎于人的问题。

(人件, DeMarco and Lister 第二版. 1999)

投资回报在现今社团社会中是第一位的问题。通过减少并使规模适度，公司对各部门进行考察以确保他们能有回报。团队中冗余的部分被排除出组织，管理要求行销项目能提供可以看到的投资回报。

激励程序的一个重要价值之一是，浪费是可知的，并且回报可以追踪。即使利益是无形的，例如员工满意度，也有方法去证明激励程序的效果。

PNC 银行集团和 PAF(公众事务论坛)的一次调查显示：

- 全美国发挥了全部潜力的员工少于 25%.
- 50%的人只按吩咐去做
- 75%的人认为他们的工作可以提高效率

- 80%消费满意度来自于知识渊博和专心的员工

无论如何，Demarco 和 Lister 补充，大多数的人喜爱他们的工作，有时非金钱的奖励更能激励他们。这就是激励程序的出发点。

- 一项美国补偿研究表明对每个员工花费$2000/人的激励程序可以使销售增加 20%。
- 总的 88%到 95%的激励程序达到或超过预定目标。
- 平均的销售激励 POI 是 134%。
- 非销售员工的激励 POI 一般为 200%。

还有，Loyalty Effect，Harvard Business，Forum Corp 在全球开展的关于顾客/职员关系的研究报告表明：

- 10-30%的客户流量/每年
- 50%的客户流量/5 年
- 50%的雇员为每 4 年一换
- 50%的投资者为每年一换
- 70%的消费者流失是因为服务不好
- 5%的消费者保持增长能够使从一个消费者的回报提高 75%。

Raghavendra Gururaj

我们终于有了一本关注论述软件工业中人的因素的著作，这可是一个好消息。当我第一次读完这本书时，我兴奋异常，即使现在再去读它也会激动。我们工作的主要问题并不是技术问题而是社会问题。我们在前面错误认为这如同给我们带来损失的政治。但是作者解释软件比政治更接近自然。我不禁嫉妒作者有如此丰富的学识与经验。

我们不常去阅读这本书的一个重要因素是由于软件工业所需要的管理风格。作者指出（非常直接），相对于生产环境，开发环境需要不同的管理风格。一些常见的错误是因为人们没有理解这种不同所造成的。设定不切实际或不可行的期限，使人们加班加点工作，追求华丽的，不现实的外表，追求高生产率，取悦高层管理者，等等。这些问题的叙述都令人难以忘怀。作者一定有卷入到事务的泥潭中不得不进行妥协的切身经历。这部分内容读起来非常有趣，作者还列出了项目管理者在管理中的一些错误期望。

作者在文中非常直接地指出管理者的功能不是使人们工作，而是使人们有工作的可能。强烈强调管理者建立一个健康，有益于工作的环境，管理者认为雇佣人才更重要。以上的三个论点令人愉快，读者可以从中受益。众所周知，整体比它的部分的和作用更大，毫无疑问，一个成功运作的团队能克服巨大困难。带领这样的团队达到

目标真是太好了。作者也指出团队的目的不是目标达成，而是目标安排。只有管理者能接受作者关于团队的观点，并开始按此去做，我才会非常高兴。通过强制要求每一个项目管理课程/学习/培训都阅读本书，我们才能公正地对待这项伟大的贡献。

简而言之，这本书是关于软件工业的最好的著作之一。它给软件工业的管理带来了显著的价值，因为该书论述的是人，人，更多的人是这个软件工业最大的资产。买下这本书，更重要的是尽可能地多次去读它。

http://flyingchihuahuas.editthispage.com/faq

我不是一个伟大的程序员。尽管我可以用特定的专长完成一些工作，但在最好的程序员与平均的程序员生产率为 10:1 的差距上，我只是达到平均水平的人，我处于人件金字塔的底端。

在《人件》的第二版出版的前夕，有人问 Tom DeMarco 和 Timothy Lister，软件开发的导师，软件经理应该从书中获取些什么？

DeMARCO: 好的管理者应该具有足够的才能熟练地建立一个社区。

LISTER: 我补充一点，人们都渴望把工作做好。如果你用别的方式管理他们，你会感到混乱。对我来说，渴望的是成为独立的贡献者，而不是，为什么我要从事我现在的工作。

MFESD

这本书最早大约出现在十几年以前，但它的内容在现在也还是适用的。

如果你相信雇用优秀的员工，为他们提供良好工具、设施、环境是软件开发成功的途径，则这本书是真正你所需要的。如果你认为只有自己才能完成成功的软件，则这本书将给你一些启发（可能有些争议）。

作者在文中涉及时间估计、空间与工作环境（使其安静）、电话（切断电话）、开放思想、雇佣、娱乐等内容。

这本书是一本经典——如果你重视软件的开发并希望你的团队成员作出最好的表现，而且你还没有读《人件》，那么请立即读这本书——许多智慧与实践的方法等着你去获取。