摘 要

软件构件技术是支持软件复用的核心技术之一，近年来，依靠中间件平台提供的基础设施，通过标准化的运行级构件的规约，为使用标准软件构件构造系统提供了一种自底向上的有效途径，得到了广泛的成功应用。但是，目前的构件技术还仍然面临着一些困难：就开发方法而言，主要还是着眼于构件实现模型和运行时互操作，缺乏一套系统的方法指导构件开发的整个过程；就基础设施而言，构件运行平台的异构性和易变性致使构件组装、集成及互操作困难重重。

模型驱动体系结构（MDA）是 OMG 在 UML、MOF、XMI、CWM 等成功技术标准的基础上提出的一种新的系统框架，它提倡使用形式化的系统模型作为解决企业应用系统集成问题的核心，通过使用软件工程方法和工具去理解、设计、操作、发展企业系统的所有方面，从而为企业应用在不同生命周期阶段的集成提供了完备解决方案。

本文在深入研究 MDA 架构的基础上，通过使用 MDA 的优点来弥补当前构件技术的不足，提出了一种系统的、全面的基于MDA的构件开发方法——MDAC 方法，其主要思想是融合模型驱动和构件的思想到软件开发生命周期的各个阶段，利用模型来开发构件，化构件开发为模型开发，化构件组装为模型组装，化构件实现为模型映射，化构件复用为高层设计的复用、模型的复用。论文的主要工作概括为如下几点：

（1）对 MDA 进行了剖析，分析了 MDA 涉及的基本概念及其相关的核心技术，论述了 MDA 对软件开发产生的重要意义。

（2）提出一种基于 MDA 的构件开发方法——MDAC 方法，对其关键技术进行了详细阐述。

（3）提出了一种基于 UML 的构件建模框架 CMF，对其 MOF 元模型及到 UML Profiles 的映射规则进行了详细阐述。

（4）给出一个示例，论证和解释了 MDAC 方法的构件建模、模型转换实现等关键内容。

（5）研究分析了相关的技术和工作，并进行了比较。

本文的研究工作得到河海大学计算机学院承担的国家“863”项目“水资源调度系统软件构件柔性组装技术研究”（编号：2001AA113170）的支持。

关键字：模型；模型驱动体系结构；MDAC 方法；构件建模框架

ABSTRACT

Software component is one of the key technologies of software reuse. In recent years, based on the infrastructure provided by middleware, by standardizing the runtime component specification, the bottom-up way of building software systems from components has been very effective, popular, and successful. However, the component technology still confronts some problems as following: First, current component development methods focus on the component implementation models and runtime interoperability, lacking systematic approach to guiding the whole development process. Second, the heterogeneousness and changeability of component runtime environment build many blocks in component assembly, integration, and interoperation.

Model-Driven Architecture (MDA) is a new framework given by OMG based on UML, MOF, XMI, CWM, and etc, concentrating on using formal model to understand, design, operate, and evolve all aspects of enterprise systems, using software engineering methods and tools. MDA provides an all-around approach to the full lifecycle integration of enterprise application.

Combing the advantages of MDA, this thesis put forward a component development method, that is MDAC (MDA-based Component development). The main idea of MDAC is that developing, implementing, compositing, and reusing components by modeling, model transformation, model integration, and model reuse. The main work of this thesis as follows:

(1) Study the framework, related concepts, key technologies, and the importance of MDA.
(2) Put forward the MDAC method and analyze its core elements.
(3) Provide a Component Modeling Framework (CWF) based on UML and analyze its MOF meta-model and mapping rules to UML profiles.
(4) Verify the effectiveness of the MDAC method by a common example.
(5) Research some related technologies and work and compare MDAC with them.

Key words: Model; MDA; MDAC; CWF

第一章 绪 论

1.1 软件构件技术

软件工程作为一门独立的学科，其发展已逾 30 年。上世纪 60 年代，由于高级语言的流行，使得计算机的应用范围得到较大扩展，对软件系统的需求急剧上升，从而产生了所谓“软件危机”，软件开发从质量、效率等方面均远远不能满足需求。60 年代末，如何克服“软件危机”，为软件开发提供高质、高效的技术支持，受到人们的高度关注。1968 年，在北大西洋公约组织（NATO）软件工程会议上首次提出了“软件工程”这一概念，从而使软件开发开始了从“艺术”、“技巧”和“个体行为”向“工程”和“群体协同工作”转化的历程。30 多年来，软件工程的研究和实践取得了长足的发展，软件工程界已经提出了一系列的理论、方法、语言和工具，解决了软件开发过程中的若干问题，虽然距离彻底解决“软件危机”尚有较大差距，但对软件开发的工程化以及软件产业的发展起到了积极的推动作用，提供了良好的技术支持。近年来，面对日益复杂的软件系统，人们开始认识到，软件复用是实现软件的工业化生产、获得软件产业发展所需的软件生产率和质量的一条现实可行的途径。

软件构件技术（Component）是支持软件复用的核心技术之一[1]。软件复用可以分为产品复用和过程复用，其中产品复用是指复用已有的软件构件，通过对已有构件的集成或组装获得新系统，这也是目前现实的、主流的途径。基于构件的软件复用作为一种提高软件生产率和软件质量的有效途径，是近几年软件工程界研究的重点之一，被认为是继面向对象方法之后，一个新的技术热潮。

简而言之，软件构件是指一个可以独立交付的软件单元，它封装了设计和实现，向外提供接口，并可以通过接口与其它构件组装成更大的软件单元。软件构件的思想及概念从提出到在业界形成一定的共识，经历了几十年的演化。同样在 1968 年的 NATO 会议上，McIlroy 在论文[2]中提出了软件构件、构件工厂等概念。在 1970、1980 年代，软件构件一般被称为代码件，主要指可复用的程序代码片段，例如子程序、程序包、类、模板等，其目标是如何充分利用已有的源程序代码、子程序库和类库来提高软件开发的效率。到了 90 年代，软件构件的概念已经得到延伸，其包括分析件、设计件、代码件、测试件等多种类型,并随之产生了许多新的概念，如设计模式，框架以及软件体系结构等。1995 年，Will Tracz 提出，构件应具有以下属性：有用性，构件必须提供有用的功能；可用性，构件必须易于理解和使用；高质量，构件及其变形必须能正确工

作；适应性，构件应该易于通过参数化等方式在不同语境中进行配置；可移植性，构件应能在不同硬件运行平台和软件环境中进行工作。后来，文献[3][4][5][6]中都给出了自己的构件的定义。这些定义虽然在表述上各自不同，但存在着共同要素，即软件构件是单独开发并具有特定功能的软件单位，用于与其它构件及支撑环境组装成应用系统。这一共同要素反映了构件的三个基本特征：单元特征，构件不是完整的应用程序，需要组装；复用特征，构件的价值在于实现软件复用，需要规范；商品特征，构件是预制的知识服务，需要封装。回顾构件技术的发展历史，说明业界对构件技术的认知正逐渐趋于一致。

随着软件构件技术研究和实践的不断深入,当前软件系统开发的趋势是大量使用商业构件（COTS，Commercial Off-The-Shelf)。COM/DCOM、CORBA、J2EE 等构件标准的成熟为 COTS 市场的形成奠定了基础。（Component-Based System Development, CBSD）是指利用已开发完成的 COTS（或可复用的构件）按应用需求组装形成软件应用系统的软件开发方法。目前，CBSD 已成为软件工程进步中的又一里程碑。

1.2 模型驱动软件开发方法

任何一个软件项目，都可以看作是由若干子系统组成，这些子系统以复杂的方式彼此相互联系，建立模型正是处理这些复杂性的一种手段。模型是对现实的简化，它提供了系统的蓝图。通过建模，可以达到以下四个目的：模型允许人们按照实际情况或按照人们所需要的样式对系统进行可视化；模型允许人们详细说明系统的结构或行为；模型给出了一个指导人们构造系统的模板；模型对人们做出的决策进行文档化。

回顾软件技术的发展历史，其核心技术之一是软件的基本模型，而驱动软件技术不断向前发展的核心动因之一是复杂性控制。高级语言的发展是为了控制计算机硬件平台的复杂性，结构程序设计的发展是为了控制程序开发过程和执行过程的复杂性，面向对象方法则是为了控制系统需求易变所导致的复杂性。近 20 年来，面向对象程序设计语言的诞生并逐步流行，为人们提供了一种以对象为基本计算单元，以消息传递为基本交互手段的软件模型，该模型以拟人化的观点来看待客观世界（客观世界由一系列对象构成，这些对象间的交互就形成了客观世界的活动），符合人们的思维模式和现实世界的结构，随后而兴起的面向对象方法学也就逐步成为软件开发的流行方法[7]。

由于面向对象的分析与设计(OOA/OOD)方法的重要性日益突出，人们对它的研究、开发和应用的热情也在不断升高。在 1989 年,以专著、论文或技术报告等形式提出的 OOA/OOD 方法或 OO 建模语言有近 10 种，到 1994 年，其数

量增加到 50 种以上。各种方法的出现都对 OOA/OOD 技术的研究与发展作出了或多或少的新贡献[7]。这种“百花齐放”的繁荣局面表明面向对象的方法与技术已得到广泛的认可并成为当前的主流。然而多种方法的同时流行也带来一些问题：各种 OOA 和 OOD 方法所采用的概念既有许多共同部分也有一定的差异(例如许多方法在 OO 基本概念基础上各自提出了一些扩充概念，字面上相同的概念其语义解释也不尽相同)；在表示符号、OOA 模型及文档组织等方面差别则更为明显。这种情况往往使一些新用户在进行建模方法及工具的选择时感到难以决策，也不利于彼此之间的技术交流。在这种背景下，UML 于 1996 年树起了统一的旗帜，使不同厂商开发的系统模型能够基于共同的概念，使用相同的表示法，呈现彼此一致的模型风格。而且它从多种方法中吸收了大量有用（或者对一部分用户可能有用）的建模概念，使它的概念和表示法在规模上超过了以往任何一种方法，并且提供了允许用户对语言做进一步扩展的机制。

UML[8]的出现为面向对象建模语言的历史翻开了新的一页，并受到工业界、学术界以及用户的广泛支持，成为面向对象技术领域占主导地位的建模语言，在许多领域的软件开发中得到应用。不过 UML 在取得巨大成功的同时，也不断地受到批评。来自工业界的批评主要是，它过于庞大和复杂，用户很难全面、熟练地掌握它，大多数用户实际上只使用它一少部分的概念；它的许多概念含义不清，使用户感到困惑。来自学术界的批评则主要针对它在理论上的缺陷和错误，包括语言体系结构、语法、语义等方面的问题。为了解决这些问题，通过广泛收集各方面的意见后，OMG 于 2003 年 6 月正式给出了 UML2.0 规范。升级后的 UML 标准有以下特点[9]：第一类的扩展机制允许建模人员增加自己的元类，从而可以更加容易地定义新的 UML Profile，将建模扩展到新的应用领域；对基于构件开发的内置支持简化了基于 EJB、CORBA、COM 的应用建模；对运行时架构的支持允许在系统的不同部分进行对象和数据流建模；对可执行模型的支持也得到了普遍加强；对关系更加精确的表示改进了继承、组合和聚合以及状态机的建模。行为建模方面，改进了对封装和伸缩性的支持，去掉了从活动图到状态图的映射，并改进了顺序图的结构；对语言的句法和语义的简化，以及整体结构上更好的组织。

与此同时，业界对 UML 的广泛认可和接受大大促进了以模型为中心的软件开发， OMG 在 2002 年提出了支持这种开发的一系列标准框架，即模型驱动体系结构（MDA，Model-Driven Architecture）[10]。MDA[11]的关键特点就是软件开发的重点和输出不再是程序，而是各种模型，开发人员的工作是不断拓展模型，只有到了最后阶段才会考虑将其实现。MDA 把建模语言用作一种编程语言而不仅仅是设计语言，它能够创建出机器可读和高度抽象的模型，这些模型以独立于实现的技术开发，以标准化的方式储存，因此，这些模型可以被重复访

问，并被自动转化为 Schema、代码框架、集成化代码以及各种平台的部署描述。同时，MDA 以一种全新的方式将 IT 技术的一系列新的趋势性技术整合到一起，这些技术包括 CBSD、设计模式、中间件、说明性规约、抽象、多层系统、企业应用集成、以及契约式设计等内容。总之， MDA 为企业应用在不同生命周期阶段的集成提供了完备解决方案，它提倡使用形式化的系统模型作为解决企业应用系统集成问题的核心，通过使用软件工程方法和工具去理解、设计、操作、发展企业系统的所有方面，为提高软件开发效率，增强软件的可移植性、协同工作能力和可维护性，以及文档编制的便利性提供了统一途径。

MDA 已经被认为是促进软件开发的另一个黄金时代的开创。那么，是什么使得 MDA 同其它无数软件社区的标准相比显得如此与众不同呢？首先，MDA 是由 OMG 推动的，OMG 是软件产业界最大的联盟，而 OMG 已经发布并维护了业界一些最成功的标准，比如 CORBA 和 UML。其次，在 OMG 内部，MDA 从系统和软件供应商群体获得了异乎寻常的强有力支持。通常，这种程度的一致同意和支持需要好些年才能获得，但是，即便像 IBM、Sun 和微软这样相互竞争激烈的对手们也都在支持 MDA 这一点上达成了一致，并且积极地支持 MDA 所包含的主要标准——UML、XMI、MOF、CWM 以及 JMI 等。毫无疑问，他们会就细节问题争论不休，但是它们都坚定地支持这一方法。这意味着，MDA 成长和繁荣所必需的主流工具和平台的支持已经指日可待了。再者，MDA 并没有声称要大规模取代以前的计算方法、语言或者工具。相反，它试图融合它们，使得每个人都可以按照他们自己的节奏，根据他们自己的需要，平稳地过渡到 MDA 的世界。MDA 同时也被特意设计得足够灵活，可以适应不可避免会快速浮现的软件新技术。因此，MDA 实际上很有希望给软件构架实践带来新生，并促进软件开发的另一个黄金时代的开创。尽管完整的 MDA 还没有成为现实，但是，模型驱动开发现在已成为可能，并且模型驱动开发确实正在起作用，并必将改变我们开发系统的方式。

1.3 本文工作

1.3.1 选题依据

从国内外关于软件构件技术研究的最新进展分析，虽然构件技术已经得到广泛运用，但是，本文认为，目前的软件构件技术还面临两大主要困难：（1）就开发方法而言，主要还是着眼于构件实现模型和运行时互操作，缺乏一套系统的方法指导构件开发的整个过程。（2）构件运行平台的异构性和易变性致使构件组装、集成及互操作困难重重。

首先，当前 CBSD 关注的重点都局限在二进制构件的规范上，例如 CORBA、EJB 和 DCOM，仅仅提供了在实现层次上支持构件交互的基础机制，缺少指导开发过程的系统化的方法学，对高抽象层次的构件组装无能为力。实际上，构件复用应该涵盖软件生命周期的各个阶段，而不是仅仅局限在运行层次上来看待构件，也不应该只是对代码进行复用。虽然，自上世纪 90 年代初期开始，软件体系结构（Software Architecture）的研究受到了广泛的关注和重视，并被认为将会在软件开发中发挥十分重要的作用。SA 将大型软件系统的总体结构作为研究的对象，认为系统中的计算元素和它们之间交互的高层组织是系统设计的一个关键方面。作为其最重要的一个贡献，SA 的研究将构件之间的交互显式地表现为连接子（Connector），并将连接子视为系统中与构件同等重要的第一类实体，这样，SA 提供了一种在较高抽象层次观察、设计系统并推理系统行为和性质的方式，也提供了设计和实现可复用性更好的构件、甚至复用连接子的途径。但是，经过十多年的研究，虽然 SA 在理论上已经较为成熟，近年来也有一些将 SA 实用化的尝试，可是这些尝试都不是很成功，其原因首先在于大多数 SA 的研究都还集中在对体系结构的描述和高层性质验证上，对体系结构的求精和实现的支持能力明显不足，由于目前主流的设计和实现语言都是面向对象的，如何从高层抽象的 SA 模型转换到具体的底层实现一直都没有一个比较好的解决方法。

其次，随着分布式计算变得越来越重要，特别是随着 Internet 的发展，各种技术标准层出不穷，比如 COM/DCOM、COM＋、J2EE、CORBA、CCM、XML、.net、Web Services 等。业界人士已经得出了一个这样的结论：对平台的未来所能做出的唯一预测是“无法预测到的事情将会发生”。哪怕开发经理对平台下了正确的赌注，平台也不太可能保持原样，而是会随着时间流逝而演变成几乎完全不同的新平台。即使经理的猜测是正确的，依赖于该平台的代码也可能会很快过时，唯一的解决方法是进行代价高昂的手工编码升级。同时，正是由于这种平台多样性和易变性，导致了构件互操作难与集成难以实现，主要表现为两个方面的异构性。一方面是构件模型的异构性，这样限制系统开发人员使用来自同一个构件模型的构件。比如说，如果一个构件提供了所需求的功能并且是可用的，但是为了符合其他构件模型的要求，不得不重新开发这个构件。这给系统开发人员带来了极大的困难，增加了成本。另一方面是运行平台的异构性，用来源于同一个构件模型的构件经过组装得到的应用系统，其系统运行环境必须支持此构件模型，否则可能出现问题。比如说，用 COM 构件组装得到的应用系统必须在支持 Windows 的运行平台下运行。所以，异构构件的互操作问题阻碍了基于构件的软件工程的快速发展。

如前所述，MDA 使用形式化的系统模型作为解决企业应用系统集成问题的

核心，为企业应用在不同生命周期阶段的集成提供了完备解决方案。综观 MDA 方法的优势和当前构件技术研究面临的主要问题（缺乏完整的方法学、跨平台组装及互操作能力差），显而易见，使用 MDA 架构，可能全面解决当前构件技术研究面临的困境，提升 CBSD 开发的效率和质量。这也是本文的主要选题依据。

1.3.2 本文的内容及结构

本文的指导思想是：基于 MDA，融合模型驱动和构件的思想到软件开发生命周期的各个阶段，利用模型来开发构件，化构件开发为模型开发，化构件组装为模型组装，化构件实现为模型映射，化构件复用为高层设计的复用、模型的复用。

本文的主要工作是提出了一种基于 MDA 的构件开发方法并对其体系结构进行了详细探讨。论文首先对目前构件技术研究的现状进行了总结，在此基础上对目前国内外的相关研究工作进行了阐述和分析；论文接着详细讨论了 MDA 方法产生的背景，以及其涉及的核心技术的功能和作用，论述了 MDA 对软件开发产生的深远意义；论文随之提出了一种基于 MDA 的构件开发方法 MDAC，并从不同的层次分析了 MDAC 方法的体系结构，主要包括模型分类、构件建模框架、模型转换实现、构件开发过程、建模工具等内容，并利用一个典型示例对 MDAC 方法的可行性进行了论证；论文最后是总结和展望。

论文的其余部分安排如下：

第二章，相关工作。介绍和讨论了几种与本文研究工作密切相关的几种构件开发方法，包括三种主流的构件实现模型（COM/DCOM、EJB、CORBA），及其它几种典型的系统的构件开发方法（CoSMIC 、COMBINE 、AOP 、Catalysis、Cosmic、ABC），论文比较了它们之间的异同点，分析了缺点与不足。

第三章，模型驱动体系结构。是对 OMG 提出了模型驱动体系结构 MDA 的一个概述，介绍了 MDA 的基本概念，包括模型、模型驱动、平台无关模型（PIM）、平台相关模型（PSM）等术语的定义，分析了 MDA 中模型之间的映射关系以及 MDA 的核心技术，并论述了 MDA 对软件开发产生的重要意义。

第四章，MDAC：一种基于 MDA 的构件开发方法。提出一种基于 MDA 的构件开发方法，从体系结构、模型类型、开发人员的角色、构件建模框架、模型映射规则、模型映射实现、中间件在 MDAC 方法中的作用、MDAC 构件开发全过程等方面对 MDAC 方法进行了详细阐述。

第五章，实例分析。利用一个典型的应用示例进一步论证和解释了 MDAC 构件开发方法的构件建模、模型转换实现等关键内容。

第六章，总结与展望。对论文的工作进行总结，比较了 MDAC 方法和传统构件开发方法，同时分析了 MDAC 方法同第二章中提及的几种方法的各自的优缺点；展望了今后需要进一步完善和开展的工作。

由于本文的研究而需要致谢的集体与个人也在正文之后列出。

本文的研究工作得到河海大学承担的国家“863”项目“水资源调度系统软件构件柔性组装技术研究”（编号：2001AA113170）的支持。

第二章 相关工作

如何有效地开发构件，一直是业界研究的热点。近年来，随着计算机技术的发展，各种构件开发方法和技术层出不穷，不断有新的技术融入到构件开发方法学中去。为了给构件开发提供一套完整的、系统的解决方案，许多国际组织和研究人员从不同的角度和技术出发，已经做出了许多努力，并已取得了较大的成功，比如，OMG 制定了一系列标准来规范基于 CORBA 的构件开发，SUN 为基于 Java 的构件开发提出了许多流行的技术标准，学术界掀起了软件体系结构研究的热潮，而面向 Aspect 的编程思想也异军突起，同时，集成模型的分布式计算（MIC）也在嵌入和实时系统中得到了成功应用。总结目前构件研究的主要活动，可以发现，研究的重点正逐渐向设计层转移，而软件体系结构和建模已经成为主要手段，本章将详细讨论几种与本文研究相关的重要工作。

2.1 COM/DCOM、EJB 和 CCM

COM[12]是由微软提出的基于 Windows 平台的构件标准，其本质是一种二进制代码级的集成策略。COM 构件主要以 DLL 或 EXE 的形式发布。DCOM 是对 COM 的扩展，它是一个高层网络协议，支持 COM 构件在位于不同机器上的两个进程间进行协作。DCOM 使得程序员可以不必编写网络代码去处理分布式构件跨网络交互所需要的通信，所以 DCOM 具有以下特性：可伸缩性、可配置性、安全性、协议无关性。

EJB[13]是 SUN 公司提出的，用于开发和部署多层结构的、分布式的、面向对象的 Java 应用系统的、跨平台的构件体系结构。EJB 给出了服务器端的分布构件规范,定义了构件、构件容器接口规范、构件打包、构件配置等内容。从企业应用多层结构的角度,EJB 是业务逻辑层的中间件技术,它提供了事务处理的能力,是处理事务的核心;从分布式计算的角度,EJB 像 CORBA 一样,提供了分布式技术的基础，提供了对象之间的通讯手段。EJB 是 J2EE 的一部分,自从 J2EE 推出之后,得到了广泛的发展,已经成为应用服务器端建立企业应用的主流标准。

CORBA[14][15]是由 OMG 制订的一种标准的面向对象应用程序体系规范。CORBA 采用标准的对象模型，它通过发布的接口使远程对象调用变得简单可用，CORBA 独立于任何语言和平台，并且 CORBA 的 ORB 核心和服务为对象互操作和系统功能提供了完整而规范的支持。CCM（CORBA Component Model，CORBA 构件模型）是 CORBA3.0 规范的主要贡献，它通过定义一些特征和服务以允许应用程序编程人员实现、管理、配置和使用由在标准环境下的 CORBA 服务集成的构件来扩展 CORBA 对象模型，这些 CORBA 服务包括持久服务、

事务服务、事件服务、安全服务等。CCM 标准不只使得服务方更多的软件可重用，而且为动态配置 CORBA 应用程序提供了更大的灵活性。

以 COM/DCOM、EJB、CORBA 为代表的基于分布式对象技术的构件实现模型正在向实用化快速发展，它们对构件的基本构成及演化产生着十分重要的影响，已成为实现级的主流构件模型，但正因为它们是实现级的，所以在企业快速多变复杂的应用需求下，其复用性、演化性、互操作性等问题正逐渐成为困扰企业应用升级的巨大难题。

2.2 CosMIC

CoSMIC[16]（Component Synthesis using Model Integrated Computing）是由美国华盛顿大学提出的一种基于 MDA 的 CORBA 构件开发方法，其主要目的是提供一套系统的方法用于规约、实现、组装、集成、验证实时和嵌入式系统的构件，并且提供严格的 QoS 控制。CoSMIC 建模工具是实现 CoSMIC 方法的基础。CoSMIC 工具建立在 ACE 中间件框架和 QuO 质量服务控制框架的基础上，主要具有以下两种功能：建模和分析实时系统的功能和 QoS 需求；组装 CCM 构件，并利用反射机制灵活实现 QoS 的静态和动态控制。图中解释了 CoSMIC 工具实现 CCM 构件组装的七个集成点：（1）配置和部署应用服务，生成服务和资源的配置策略；（2）组装并部署构件到构件服务器，并检查构件之间连接的语义匹配；（3）配置构件容器，生成 QoS 相关的配置策略，比如安全、持久、多线程等。（4）生成构件实现，并加以裁剪以满足特殊需求；（5）基于反射机制合成动态的 QoS 策略；（6）生成具体的中间件平台相关的配置策略；（7）生成基于 CCM 构件的中间件实现。CoSMIC 方法已经在一些领域得到了成功应用，比如航空、电信、制造业等，但由于 CoSMIC 主要用于基于 CORBA 的分布式应用，所以缺乏通用性。

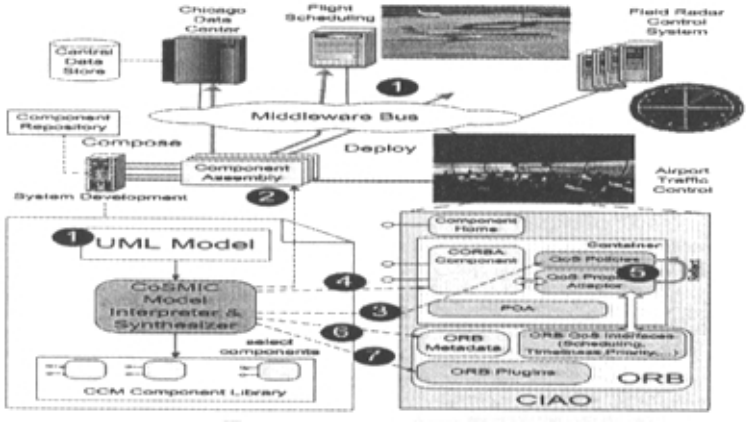

图 2.1 CoSMIC 方法的集成点

2.3 COMBINE

COMBINE (COMponent-Based INteroperable Enterprise system development)[17]是由欧洲几家著名的公司和组织（Open Group、Adaptive、INESC等）发起的研究项目，声称最早把MDA的思想运用到CBSD软件开发的全过程。通过融合CBSE、工作流、MDA、软件体系结构、建模等多种技术，基于构件中心的思想，COMBINE为企业应用的整合提供了完整的框架。COMBINE构件中心的结构如图所示，它认为一个项目组由系统架构（Architecture Organization）、产品开发（Product Development Organization）、生产设施（Production Facility Organization）三个相互独立的组织构成，特别地，系统架构由生产设施来实现，而产品开发部分负责指导系统的架构和实现。采用这样的方式，应用系统支撑平台相关的细节（例如事务服务、可扩展性支持等）均由生产设施来完成，而产品开发部分则只需要关心业务构件的创建，构件是否具有复用性则由系统架构部分来约定。整个基于构件中心的构件开发过程由一系列集成开发环境和工具（建模工具、模型转换工具、编程工具、部署工具等）来实现，值得一提的是，COMBINE主要使用一种称为Model Link的技术来合成业务模型和系统模型。总之，基于工作流的思想，通过明确而清晰的分工，采用MDA开发方法，COMBINE大大提高了构件开发和软件生产的效率。COMBINE方法的缺点在于，由于重点放在了工作流的建模处理上，而对构件建模的支持相对薄弱。

图2.2 COMBINE构件中心

2.4 AOP

关注点的分离[18]（separation of concerns）一直是软件工程领域研究的重要方法之一，其主要思想就是把复杂的软件系统分解成不同的子问题来各个击破。所谓关注点是指一个特殊的目标、概念或兴趣域，一个复杂的软件系统是可以看作由若干功能关注点和非功能关注点组成，功能关注点就是系统要完成的业务功能，而非功能关注点，即横切关注点，它是完成功能关注点所必须的配套

设施。使用目前的一些主流方法，比如 OOA/OOD、SA，当设计层的关注点根据系统规则映射到实现层时，横切关注点的分离往往难以实现，因为需求空间是一个多维空间，而实现空间是一维空间，这样的不匹配造成了从需求到实现的模糊映射，从而导致代码交织和代码分散，使导致程序难于开发、理解、维护、升级。近年来，一种有效解决横切关注点分离问题的新的程序设计方法浮出水面，并已进入实用阶段，这就是面向 Aspect 的程序设计方法（Aspect-Oriented Programming ，AOP）[19]。在 AOP 中，程序设计语言中定义了新类 Aspect，允许程序员单独编写、检查和编辑模块间的横切关系，支持使用日志技术追踪模块，当需要改变或更新某个模块的功能时，只需要将修改后的 Aspect 模块直接织入（weaving）需要改变的地方，可见，AOP 巧妙、直观、有条有理、有效的解决了当前程序设计方法的弊端。AOP 正成为软件方法学研究的新热点。

许多研究人员已经开始把 AOP 的思想运用到构件开发过程中。面向 Aspect 的构件工程 AOCE 是一种开发可复用、可扩展、动态适配的构件的新方法。AOCE 使用 Aspect 来描述构件的功能和非功能属性，构件通过提供或请求接口和一些具体的 Aspect 相关联。Aspect 的使用为构件开发人员提供了系统的全局特征，使构件的配置、管理和组装更加灵活。JAC 是一种基于 Java 的面向 Aspect 的构件模型，它建立在 J2EE 的基础上，但是，由于目前的 J2EE 应用服务器并不能总是满足从代码中分离关注点的需求，JAC 使用 POJO 对象来替代 EJB，而与横切关注点密切的 EJB 容器则使用一种更加松散的、即插即用的 Aspect 构件来实现，Aspect 构件具有以下特征：实现了持久性的无缝集成；提供弹性的集群机制（负载平衡、数据一致性等均可定制）；可以随时定义用户管理、安全等特征；提供 IDE 支持应用和 Aspect 构件的快速开发；提供基于 JOTM 的事务支持。JasCo 是一种面向 Aspect 的实现语言，它主要用于基于构件的软件开发，特别是 Java Bean 的开发。JasCo 在 Java 语言的基础上增加了两个实体：Aspect Bean 和 Connector. Aspect Bean 使用一种内部类（称为 Hook）来描述构件的横切行为，Hook 的定义独立于任何构件类型和 API，因此 Aspect Bean 具有很好的可复用性；而 Connector 用来部署一个或多个 Aspect Bean 到具体的运行环境，同时，还可以通过 Connector 预定义管理策略来处理 Aspect 之间的集成和互操作。

虽然 AOP 方法有许多优点，但是作为一门新的方法学，但是，其在许多方面仍然需要不断完善和发展，尤其是在建模支持、动态特征、集成开发环境和工具等方面的研究。

2.5 Catalysis

Catalysis[20]是 ICON 公司开发的一种基于对象框架的构件开发方法。ICON 公司曾在 1996 年 OMG 制定 UML 标准建模语言的过程中起到了重要作用。Catalysis 定义了一个清晰的构件开发过程，包括分析、设计、实现、测试各个阶段（如图所示），采用软件工程的螺旋模型指导开发过程。Catalysis 精确定义了模型、设计和说明的模式，使用 UML 连续视图来描述复杂系统，每一视图定义了模型的合作方式，Catalysis 还提供了模型间的一致性原则，以及将视图组合起来描述复杂系统的强大机制。Catalysis 的建模原则是使词汇表与建立在简单类型模型基础上的丰富建模工具相符合，而不规定具体实现，从而让用户和开发人员使用共同的词汇表，使得两者讨论问题更加方便。Catalysis 使用框架来进行基于构件的软件开发，可以从可重用构件和框架来快速建立业务模型、需求说明、设计和代码。Catalysis 适用于许多环境，并可以跟踪从业务模型到代码的提取过程，支持连续性和体系结构的转换，定义了完善的语义一致性检查。Catalysis 方法在分离关注点上取得了一定的成果，它明确划分了做什么，谁来做，如何做这些概念。“做什么”描述参与事务的对象的行为；“谁来做”就是分配参与事务的责任，标识它们之间的依赖关系。从而产生需要的效果；“如何做”就是标识细粒度对象和提供组件服务的交互的模式，提取交互粒度。这些特性贯穿整个开发过程，在任何阶段都应遵循，这样就可以明白地将注意力放在要解决的问题上。Catalysis 方法的缺点在于使用十分复杂，对基于 UML 的构件建模给出了很多规定，但是缺乏具体而清晰的过程指导。

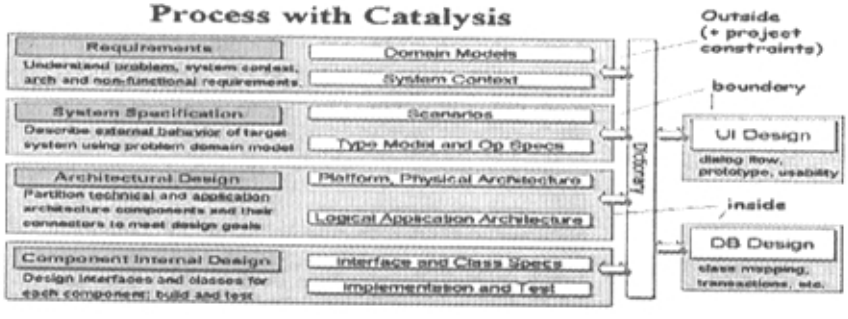

图 2.3 Catalysis 方法的体系结构

2.6 ABC 方法

ABC[21]是由北京大学软件工程研究所提出的一种基于软件体系结构的面向构件的软件开发方法。其主要思路是：将软件体系结构描述作为构件开发的框架和组装系统的蓝图；将中间件技术作为构件组装所得系统的运行时支撑；

使用一系列的映射规则和工具来缩短设计和实现间的距离，自动进行从设计到实现的转换；软件体系结构将用作贯穿整个软件生命周期的重要制品，软件系统的构造将围绕软件体系结构来进行。图 2.4 给出了基于 ABC 的软件开发过程：首先是基于 SA 的需求获取和分析，将 SA 的概念引入需求空间，从而为分析阶段到设计阶段的过渡提供了更好的支持。在得到需求分析结果的基础上，进行体系结构的设计，考虑系统的总体结构以及系统的构成成分，根据构成成分的语法和语义要求在构件库中寻找匹配的构件。当不存在符合要求的构件时，则需根据具体情况或者根据 CBSD 的原则和方法开发新的构件，或者将某些已有构件进行组装而得到满足需求的构件。在组装阶段，每一个系统构件都具有了实现体，在经过语法和语义检查后，这些构件将会通过胶合代码组装到一起，最后，被部署到相应的中间件平台上。在实践中，整个开发过程将呈现多次迭代性。目前，ABC 方法关注的主要还是从 SA 建模到系统的组装、部署阶段的工作。ABC 方法已经基本具备模型驱动开发的思想，但由于软件体系结构本身的研究仍然存在许多未解决的难题，例如 ADL 语言的统一、SA 的建模等，所以 ABC 方法目前并不适用。

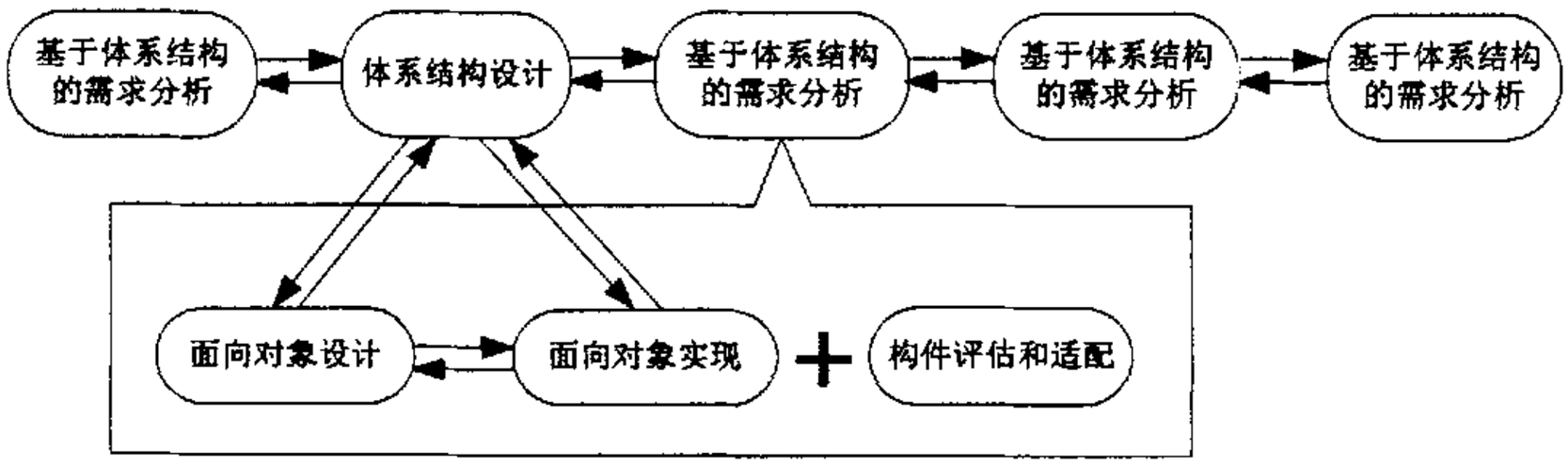

图 2.4　ABC 的开发过程

2.7 小结

本章中主要分析了现今在国内外比较流行的几种基于构件的软件开发方法，指出了虽然在构件开发方法上出现了不少的新技术，但是不同的构件开发方法仍都存在自身的不足之处，也为论文后面分析基于模型驱动的开发方法的在这方面的优势，找到了比较点。

第三章 模型驱动体系结构

制定一个开放的对象互操作工业规范，实现异构环境下企业应用的集成，一直是国际对象管理组织（OMG ,Object Management Group）的核心任务。OMG成立于1989年，最初有3Com、AmericanAirlines、Cannon Inc、DataGeneral、HP、Philips Telecommunication N.M、SUN、Unisys八个成员，目前已经发展到超过700个成员的国际组织。在1990年底，OMG推出了对象管理体系结构（OMA ，Object Management Architecture，），对象请求代理（ORB，Object Request Broker）是OMA的核心；1991年，OMG在OMA的基础上提出了CORBA1.1规范，从1991到2002年，CORBA规范已经发展到CORBA3.0版本，CORBA为解决企业异构应用的集成和互操作的实现提供了好的途径；虽然OMG最初因为CORBA的成功而闻名，但是，从1997年以来，OMG关注的范围显著拓宽，先后制定了统一建模语言（UML，Unified Modeling Language）、元对象设施（MOF，Meta Object Facility）、XML元数据交换（XMI，XML-based Metadata Interchange）、公共仓库元模型（CWM ，Common Warehouse Metamodel）等一系列重要规范，逐渐从软件开发的整个生命周期为企业应用的集成提供全面的解决方案。但是，随着多种技术标准的繁衍，各种标准之间集成的问题变得日益突出，为了解决这个问题，OMG于2001年提出了模型驱动体系结构（MDA，Model-Driven Architecture），它提倡使用形式化的系统模型作为解决企业应用系统集成问题的核心，为企业应用在不同生命周期阶段的集成提供了完备的解决方案：MDA提供了一个全面的系统框架，通过使用软件工程方法和工具去理解、设计、操作、进化企业系统的所有方面，并把系统抽象为不同层次对之建立相应的模型，从而致力于处理这些不同抽象层次模型之间的相互关系，强调系统概念设计的复用。

3.1 MDA 的基本概念

3.1.1 相关定义

模型（Model）是用某种工具对同类或其它事物的表达方式，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[22]，工程、建筑和其它许多领域中都用到模型概念。软件系统的模型用建模语言来表达，如UML，是对系统的功能、结构或行为的形式化规范。软件模型包含两个主要方面：语义方面的信息（语义）和可视化的表达方法（表示法）。语义模型用一套逻辑组件表达应用系统的含义，如类图、关联图、状态图、用例图、消息图等。一个语义

模型具有一个词法结构、一套高度形式化的规则和动态执行结构，以及这些内部结构之间分析、推理的规则。语义模型元素常用于代码生成、有效性验证、复杂性度量等。表示法携带了模型的可视化表达方式，即语义是用一种可被人直接理解的方式来表达的。它们并未增添新的语义，但用一种有用的方式对表达式加以组织，因此它们对模型的理解起指导作用。

模型驱动（Model-driven）是指利用模型来指导系统开发[22]，包括系统理解、设计、架构、开发、部署、维护、集成等系统生命周期相关的全过程。

软件平台（Platform）是指用来构建与支撑应用软件的独立软件系统。它是开发与运行应用软件的基础，是任何一个应用软件得以实现与应用的必要条件。软件平台有两个基本要素，即支撑环境和开发体系，其中支撑环境是指应用软件系统开发与运行的基本条件，开发体系是指开发与维护管理应用软件的工具与方法。

MDA[22][23]是模型驱动体系结构（Model Driven Architecture）的缩写，它是由 OMG 制定的一套软件开发框架。其关键之处是，模型在软件开发过程中扮演了非常重要的角色，软件开发过程是由对软件系统的建模行为驱动的。MDA 把建模语言当作编程语言来使用，而不只是当作设计语言。MDA 开发生命周期和传统的软件开发生命周期并没有很大的不同，只是 MDA 的工作对象是形式化的模型，也就是可以被计算机理解的模型。

MDA 主要包括两类核心模型：平台无关模型（Platform Independent Model ，PIM）和平台相关模型（Platform Specific Model ，PSM）[22] [23]。PIM 是一个系统功能和结构的形式化规范，它从与实现技术无关的细节中抽象而来；PSM 是合成了系统实现平台技术细节的平台模型，它是平台无关模型到具体平台的映射。例如，对一个执行银行账户转账的操作，它具有加（目标账户）、减（源账户）基本功能，有一个约束（目标账户和源账户的顾客特征必须一致），不管它是由 CORBA、EJB 还是 SOAP 对象来实现的，这个操作的功能是不会发生变化的，如果操作规范是由 CORBA 定义的，那么这个操作就是平台具体的了，一个依赖 CORBA 提供的如 ORB、服务或 GIOP/IIOP 接口的系统规范就是一个平台相关模型了。

采用传统的方法，从模型到模型的映射，或者从模型到代码的变换，主要是手工完成的，与此相反，MDA 中的模型变换总是由工具执行的，许多工具可以把 PSM 变换成代码。这并不令人惊奇，MDA 的创新之处就在于把 PIM 到 PSM 的映射也自动化了。

3.1.2 模型映射关系

MDA处理不同抽象层次模型之间相互联系的过程主要通过两个抽象模型来完成，即PIM和PSM。图3.1显示了MDA中模型之间的映射关系，PIM被分为两个子模型：业务逻辑（计算无关）和构件视图（计算相关）。MDA中一共包括四种类型的映射：PIM到PIM，PIM到PSM，PSM到PSM，PSM到PIM[24]。

PIM到PIM的映射主要用于平台无关模型内部的精炼与抽象，它也包括业务逻辑和构件视图的转换，优化了系统的需求分析和设计；PIM到PSM的映射是将平台无关模型转换到平台相关模型，这一步映射通常在PIM根据具体平台信息经过精炼和分析后执行，目的是把构件视图映射到具体的中间件平台(如CORBA、EJB)；PSM到PSM的映射用来精炼构件实现和部署过程中相关模型之间的关系，比如具体服务的选择和属性配置；PSM到PIM的映射是逆向工程操作，它从基于具体中间件平台的实现抽象出平台无关模型，这个过程与“数据挖掘”的概念类似，很难完全自动实现。

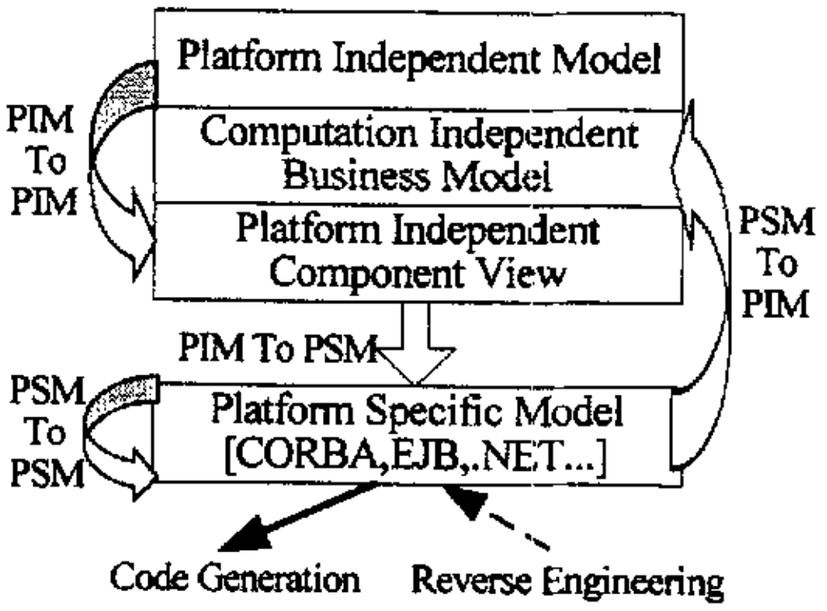

图 3.1 PIM、PSM 映射关系

在MDA中，代码能够使用有效的UML工具根据选择的平台和具体的技术自动生成，但是，对于逆向工程，MDA保留了开放性，虽然有许多逆向工程的工具可以使用，但是其有效性还有待进一步证实。

将平台无关模型和平台相关模型显式地分开有如下优点：

（1） 平台无关模型可以被多种实现技术复用，当技术平台发生变迁的时候平台无关模型不必做改动；

（2） 由于和具体实现技术无关，平台无关模型可以更加精确地体现系统的本质特征，因此可以跨越具体的技术圈子进行交流和共享；

（3） 由于使用了简单的、通用的模型，可以更加容易地对平台无关模型进行验证；

（4） 在平台无关模型这个层次上对跨平台互操作问题进行建模非常容易，因为使用与平台无关的通用术语，所以语义表达会更加清晰。

3.2 MDA 的基础技术

MDA 由许多重要的 OMG 标准共同构成（如图 3.2 所示），包括统一建模语言（UML，Unified Modeling Language）、元对象设施（MOF，Meta Object

Facility）、XML 元数据交换（XMI，XML-based Metadata Interchange）、公共仓库元模型（CWM，Common Warehouse Metamodel）。

UML[8]负责体系结构、对象及对象之间关系、数据、构件结构及组装关系的建模。从 MDA 的观点出发，UML 的主要优点是：把抽象语法和具体语法分离；UML 不是一个一成不变的语言，可以通过 Profile 进行扩展；UML 使得提升软件开发的抽象层次成为可能；它由开放的标准团体管理。而一些值得注意的 UML 的缺点是：元模型元素之间有太多的交叉依赖性；与 MOF 存在一定的冲突；没有标准的方法让工具之间交互。OMG 已经在 UML2.0 中对 UML 的许多缺陷进行了修正，但等待其商业实现仍需要一段时间。

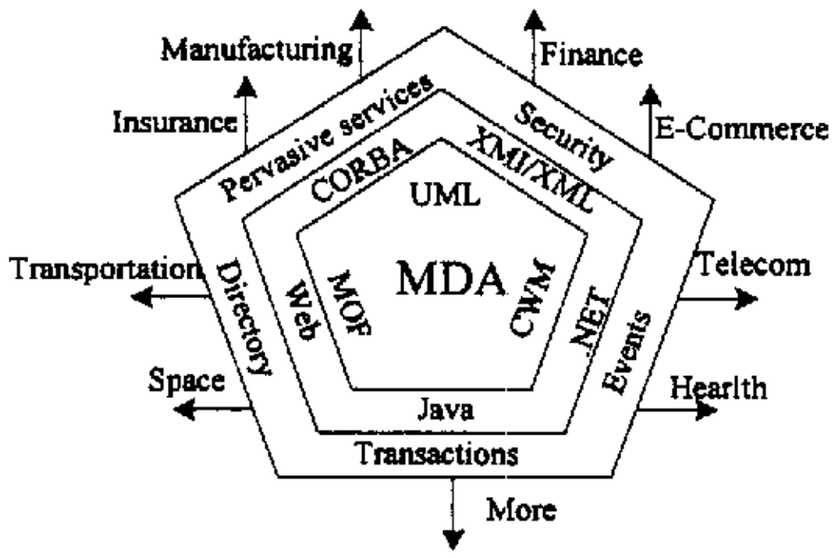

图 3.2　MDA 的基本结构

MOF[25][26]是一个标准的基础设施，用于管理信息库中的对象模型的生命周期。MOF 是 MDA 的关键基础，它允许系统架构人员定义不同的建模语言，但仍然以整合的方式来管理不同的元数据。MOF 元数据包括了对建模语言语法的定义和对语言语义的一些非形式化的文字阐述。MOF 技术映射规定如何把任何遵从 MOF 的抽象语法翻译成可以用来表示遵循抽象语法的模型的具体形式。目前，已有三种标准化的映射：MOF-CORBA，MOF-XML（XMI），MOF-Java（JMI）。到 WSDL 的映射也正在标准化过程中。

XMI[27]是一种标准的数据交换机制，用于不同工具、仓库、中间件之间的元数据交换，XMI 也能用于自动地生成 XML DTD 和 XML Schema。在 MDA 中，XMI 主要用于对 UML 元模型进行管理，它将 XML 应用到 UML 的抽象系统，捕捉和表达 UML 模型的关系，而忽略特定 UML 图的大多数可视细节。这种将事物划分成必不可少的内容与可有可无的形式的做法增强了 UML 的可管理性。

CWM[28]是建立在 UML、MOF、XMI 三种规范基础上的一组元模型，目的是为分布异构环境下的数据仓库工具、数据仓库平台和数据仓库存储建立一个商务智能元数据的交换机制。CWM 覆盖了数据仓库应用的整个生命周期，包括数据源表达、分析、仓库管理以及典型的数据仓库环境基础组件。数据源元模型支持对多种数据源进行建模；分析元模型则用于对数据转化、OLAP、信息可视化、商业术语以及数据挖掘的建模；仓库管理元模型则负责仓库处理、动态跟踪以及时间规划方面的建模；基础组件元模型则支持多种通用的数据仓库元素和服务。

应用系统的存在必然依赖一系列的服务，具体的服务类型和不同的应用系统有关，但一些服务通常是共有的，典型地如目录服务、事件服务、持久服务、事务服务、安全服务等；另外，一些实时系统还需要硬件或软件方面的特殊属性，如必须是可扩展的、具有容错功能或被设计为满足特殊环境（嵌入式系统）。上面这些服务如果定义在特定的平台上，那么它必然受到该平台的限制，或者它们只在该平台上具有良好的性能。可见，只有集成了服务的特征和结构以后，MDA关于平台无关的定义才能适合所有的中间件平台。为了使服务也具有平台无关的特征，MDA定义了平台无关模型层次的普遍性服务（Pervasive Services）[29]，这样，系统所需的服务也可以使用UML来描述，然后映射到平台相关服务。在平台无关模型层，普遍性服务仅仅在较高层次可见（类似于CCM或EJB中构件使用人员对构件的视图），当这些服务被映射到具体的平台时，能够自动产生调用具体服务的代码；在平台相关模型层，这时普遍性服务仅仅对底层应用可见（如直接写服务的应用程序）。目前，OMG正在为相关应用领域制定标准化的MDA服务规范，以最大限度的发挥MDA的优势。

表1　MDA的基本组成元素及作用

Problem Domain	Source Meta Model (PIM)	Mapping Specification	Target Middleware, Language (PSM)	Interchange Format/API	Status
Application Development OO Analysis & Design	UML	UML Profile for CORBA	CORBA	XMI	Adopted 1997
Software Process Engineering	UML	UML Profile for SPEM, SPEM Metamodel	Agnostic	XMI	Adopted 2001
Metadata Management	MOF	MOF to IDL	CORBA	IDL interfaces	Adopted 1997
	MOF	MOF to XML DTD (XMI) MOF to XML Docs (XMI)	XML DTD, XML Documents	XMI	Adopted 1999
	MOF	MOF to XML Schema (XMI),XML to MOF (Reverse)	XML Schema , XML Documents	XMI, Native XSD	Adopted 2002
	MOF	MOF to Java (JMI)	Java	JMI, XMI	Adopted 2002
Enterprise Components/ Middleware	CORBA	CCM	CORBA3.0	IDL, XMI	Adopted 2001
	UML	UML Profile for EJB	J2EE	XMI, JMI	Public Final Draft 2002
	UML	UML Profile for EDOC; JCA, EJB and Java Metamodels	J2EE,Java,EJB	XMI, IDL (JMI)	Adopted 2002
Data Warehousing	CWM	MOF to IDL, XMI, JMI*	Various Database Middleware	XMI, JMI①	Adopted 2001
	CWM	Java for OLAP	OLAP - Multi Dimensional Database	JOLAP;② XMI	Public Draft 2002
	CWM	Java for Data Mining	Data Mining	JDM,③XMI	Public Draft 2002
	CWM	CWM for Web Services	Database Middleware	SOAP, UDDI, WSDL	Initial Submission
Web Services	MOF	MOF to WSDL	WSDL/SOAP	WSDL	Initial Submission
	MOF	MOF to UDDI	UDDI/SOAP	UDDI	Initial Submission

注：①Java Metadata Interface（Java元数据接口）[30],②Java OLAP Interface（Java联机数据分析处理接口）[31],

③Java Data Mining（Java数据挖掘接口）[32].

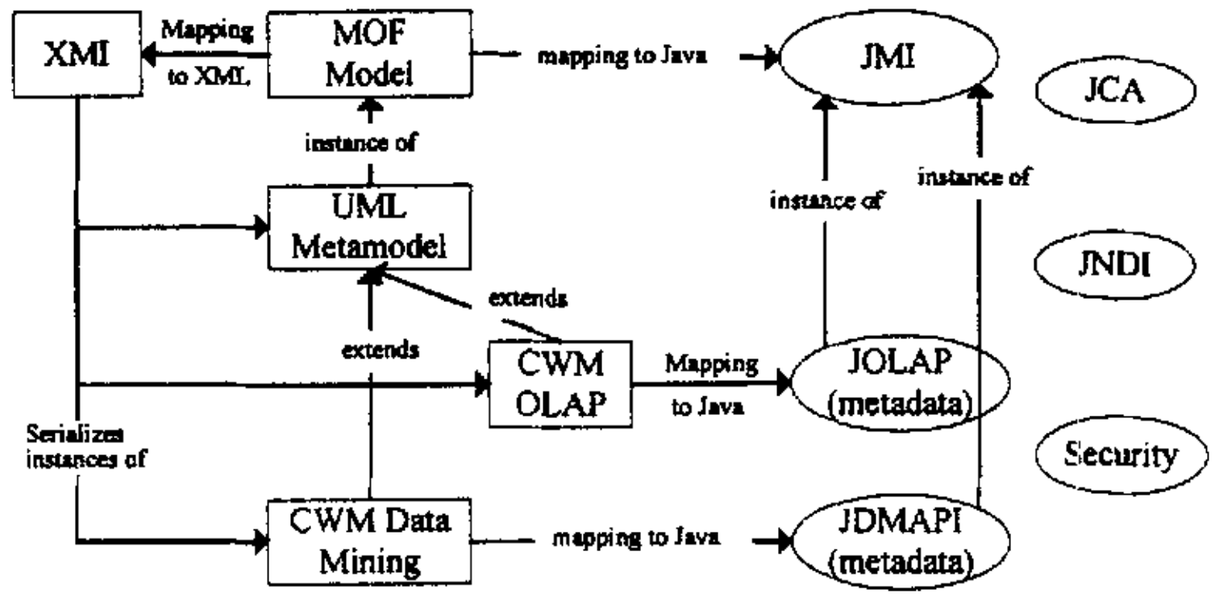

图3.3 MDA各标准之间的关系及作用

现在，应用系统之间的集成完全由MDA的关键元素来完成：具体的模型和XML DTDs划分了整个软件开发的生命周期，UML Profiles提供了不同生命周期阶段模型之间的映射，XMI通过XML完成了模型、元数据 （MOF ，XML）和中间件 （UML Profiles for Java, EJB, IDL, EDOC等）的无缝集成，表3.1给出了MDA的核心元素在模型转换过程中的作用，概括为MDA的“4M”特征（Model，Metadata，Mapping，Middleware）。图3.3给出了以EJB为PSM示例的MDA的各个标准在软件开发过程中的关系及作用，其中，JMI（Java元数据接口）主要用于访问J2EE服务，JOLAP（Java联机数据分析处理接口）负责数据处理，JDM（Java数据挖掘接口）负责知识的发现和处理。

3.3 MDA 和 CBSD

3.3.1 MDA 的优点

为了满足应用不断增加的需求，软件技术总是跟着繁衍与发展，应用系统标准一旦和具体的软件技术联系在一起，那么，系统中最有复用价值的概念设计将随着技术的变迁而消失，MDA 使得系统概念设计的保存和复用成为可能，MDA 从真正意义上应用各种最佳模式去实施各种标准和准则，必将提升应用对商业变化的响应速度和开发人员的工作效率，真正实现“少投入，多产出，高质量”，其主要优点如下：

（1）实现了对应用标准的复用。商业现实导致了现存的技术标准越来越多，各个标准之间的集成及复用问题已经成为急待解决的主要问题。MDA 完全建立在商业现实的基础上的，真正意义上将商业应用中的体系结构和底层的软件环境分离开来，通过分离出领域标准中最重要、最具复用性的部分（概念设计），MDA 使存在的标准能够在新技术上快速而廉价地被认识和复用，这样，企业既可以同步享有基础软件更新所带来的优势，同时又可以确保商业模型的相对稳定。

（2）统一了软件工程规则。MDA 消除了开发过程中各参与方之间的隔阂，使需求工程师、系统分析员、软件开发人员和测试者都可以使用同一种语言[8]，并把系统和软件开发更多地纳入到软件工程规则中。

（3）实现了系统设计的复用。即使最有经验的开发者参与标准的制定，能够立即得到一个正确的概念设计将是十分困难的。MDA实现了使用软件技术进行概念设计，让用户更容易利用这些概念模型到现在或将来的应用环境。当技术革新发生时，传统的升级费用被减少，因为新旧系统将能同时分享共有的、标准的概念设计；更少精力被用来集成软件系统，因为代替或复制全部的概念设计，集成人员仅仅需要完成相同概念的不同表示之间的转换工作；再者，由于系统的基本功能将遵循相同的概念设计，所以大大降低了职员再培训的花费。

（4）在概念层实现了系统的集成。MDA提出使用系统模型作为解决应用互用性问题的核心，使系统规格说明从具体的实现技术或平台的独立出来，允许将不兼容的技术映射到同一技术，以及一个标准的PIM的两种实现分享一个共同的概念设计。

（5）开放性。其它的标准组织也可以从MDA中受益，由于强调系统概念设计的复用，MDA不仅能够满足OMG原有标准的需要，而且为应用在其它技术标准（如WebServices）的实现提供了广泛的开放性，开发人员可以根据具体的需要选择一个或多个PSM，而系统中最难、花费最高的概念设计却不需要重复处理。

总之，与OMG以前的所有技术标准不同的是，MDA在项目管理、需求工程、软件开发过程改进、软件方法学、系统集成、增加软件的复用程度和质量、降低系统花费等多方面都迈出了具有革新意义的一步。当然，作为一种全新的思想，MDA仍然面临着许多挑战，比如对模型的表示和处理功能的完善，相关辅助标准的制定，提供更多的动态功能等。

3.3.2 MDA 和构件开发

构件开发是软件复用的技术基础，基于 MDA 的思想，通过对 UML 的扩展，可以实现一个基于 MDA 构件开发方法，从而集成了 MDA 的所有优点，实现了化构件开发为模型开发，化构件组装为模型组装，化构件实现为模型映射，化构件复用为高层设计的复用、模型的复用。

针对与论文上一节所提到的优势，首先，MDA 开发标准能够实现对应用标准的复用，以及实现系统系统设计的复用，因而基于 MDA 的构件开发能够提高构件复用的层次，从构件实现层的构件复用提升到构件设计层的构件设计复用。对于构件复用层次的提升也使得构件集成的层次相应提升，可以实现在概念层的构件集成。MDA 标准的的开发性也使得基于 MDA 的构件开发不仅限于

满足 OMG 原有标准的需要，而且为其它技术标准，如 WebServices，提供了广泛的开放性，适应构件运行平头的异构性和易变性。

其次，对于开发人员来说，基于 MDA 标准进行的构件开发，能够使得构件开发的可视化，以及提高构件开发效率。模型驱动开发的基础是模型和表达模型的语言[33]。模型提供了这样一种能力，能够一致性地显示这个系统的不同视图。利用模型可以实现构件的可视化开发，并消除开发过程中各参与方之间的隔阂。MDA 将开发人员的注意力转移到开发 PIM 上，PIM 开发人员由于不需要设计和撰写平台相关的细节，减少了工作量，而在 PSM 和 Code 层面，开发人员仅需要编写少量的代码，因为大部分的构件框架代码都已经从 PIM 中自动生成了，大大提高了软件生产率，也能在较高的抽象层实现构件的互操作和集成。同时可以提高构件模型的可维护性和文档管理，因为模型本身是对代码的最佳诠释，PIM 完全可以当做设计文档来对待。

但是基于构件的软件开发不同与一般的软件开发，基于构件的软件开发是要利用已开发完成的商业构件或可复用的构件按照应用需求组装，形成软件应用系统的软件开发方法。CBSD 是以结构化及面向对象程序设计为基础，强调系统的模块化，力求通过定义良好的接口对系统的各个组成部分进行组装，基于构件的开发过程由构件检索、选择、评估、连接、配置和组装构成。因此，使用 MDA 标准进行构件开发必须建立一个既融合 MDA 开发优势又复合基于构件的软件开发过程的开发框架，以及对建模技术和转换规则的扩展使用，这些都将是论文下一章所讨论的重点。

3.4 小结

论文在本章中详细阐述了 MDA 相关的基本概念和它的四种映射关系，并在此基础上深入分析了包括统一建模语言 UML、元对象设施 MOF、XML 元数据交换 XMI 等 MDA 核心技术的实现原理，以及它们在模型驱动体系结构中的作用。而后在本章的最后，结合 MDA 的优点并联系软件构件开发的特点，总结了使用 MDA 思想进行构件开发的可行性及其优点。

第四章　基于 MDA 的构件开发方法

通过前面几章的分析可以看到，虽然以分布式对象为基础的构件实现技术日趋成熟，但是，复杂的应用需求导致目前的实现级构件模型很难适应平台的易变性、异构性等不可预测的变化，构件技术研究的重点正在向设计层转移，其中，对基于体系结构和模型的构件开发方法更是热点，本文的主题也是沿着这条思路展开的。

MDA 是一个开放的、中立于软件供应商的架构，它广阔地支持不同的应用领域和技术平台。在 MDA 中，PIM 代表对需求的建模，PSM 代表应用具体技术后的模型，这使得 MDA 成为需求和技术之间的杠杆；它们各自的改变都可以是相互独立的，不会造成商业逻辑和实现技术的紧密藕合，同时，MDA 又可以通过转换来弥补它们之间的鸿沟，从而保护系统的投资。MDA 开发途径使得系统能够灵活地被实现、集成、维护和测试，系统的轻便性、互操作性和可重用性都是可以长期保持的，能够应对未来的变化。

结合 MDA 的优点，论文在本章提出一种基于 MDA 的构件开发方法（MDAC, MDA-based Component development），给出 MDAC 方法的体系结构，并分析其层次、开发人员的角色；结合 UML 建模技术，提出了一种构件建模框架（CMF，Component Modeling Framework），讨论了 MDAC 方法中模型映射的实现细节，主要包括基于 CMF 框架的构件建模技术、元模型及其向 UML Profile 的映射规则等内容；给出了基于 MDAC 的构件开发的详细过程，并对构件运行平台和构件建模工具进行了探讨；本章最后对 MDAC 方法的优缺点进行了总结和分析，并与第二章提及的当前的一些主要研究工作做了比较。

4.1 MDAC 的体系结构

MDAC 的体系结构如图 4.1 所示。从应用和工具实现两种视角，整个 MDAC 框架分为四个部分，它们分别为领域建模层、构件建模及实现层、应用实现层和构件运行平台层。应用视角显示了构件开发中的划分，而工具实现视角着重于相关技术和工具的支持。

对于领域建模层来说，是由领域专家根据需求进行建模，该模型是与实现技术完全无关的，它所对应的实现工具是领域建模技术与工具。

构件建模及实现层是整个框架的核心。首先，由软件设计人员根据领域需求模型建立平台无关的构件模型 PIM，PIM 以元数据的形式存入模型库；然后，由构件开发工程师根据特定平台和实现语言，选取 PIM 构件模型，将 PIM 构件模型映射成具体平台相关的 PSM 构件模型；在获得 PSM 构件模型以后，再利

用映射工具生成构件实现和相应文档（描述构件功能及构件之间的关系），并存入构件库和文档库。其间可以对 PIM 和 PSM 构件模型进行精练。构件建模及实现层所对应的工具是建模工具、模型转换工具。

下一层是应用开发层。应用开发人员根据上层所建立的构件库和文档库选取构件，构件生成应用系统，并部署到应用服务器。与传统方法不同的是，构件组装不仅可以从构件库中复用构件，还可以提高抽象的层次，进入模型库中直接复用与实现技术细节无关的 PIM 构件模型，通过工具自动生成系统所需构件。应用实现层所对应的工具是构件组装工具和部署工具。

最低层是构件运行平台，主要是指中间件层或应用服务器。构件运行平台提供构件运行时的支撑机制，它是构件互操作的标准和通信平台，如分布通讯机制、安全问题、事务处理等。它所对应的实现主要是一些流行的中间件应用服务器，比如 Weblogic、WebSphere、JBoss 等。

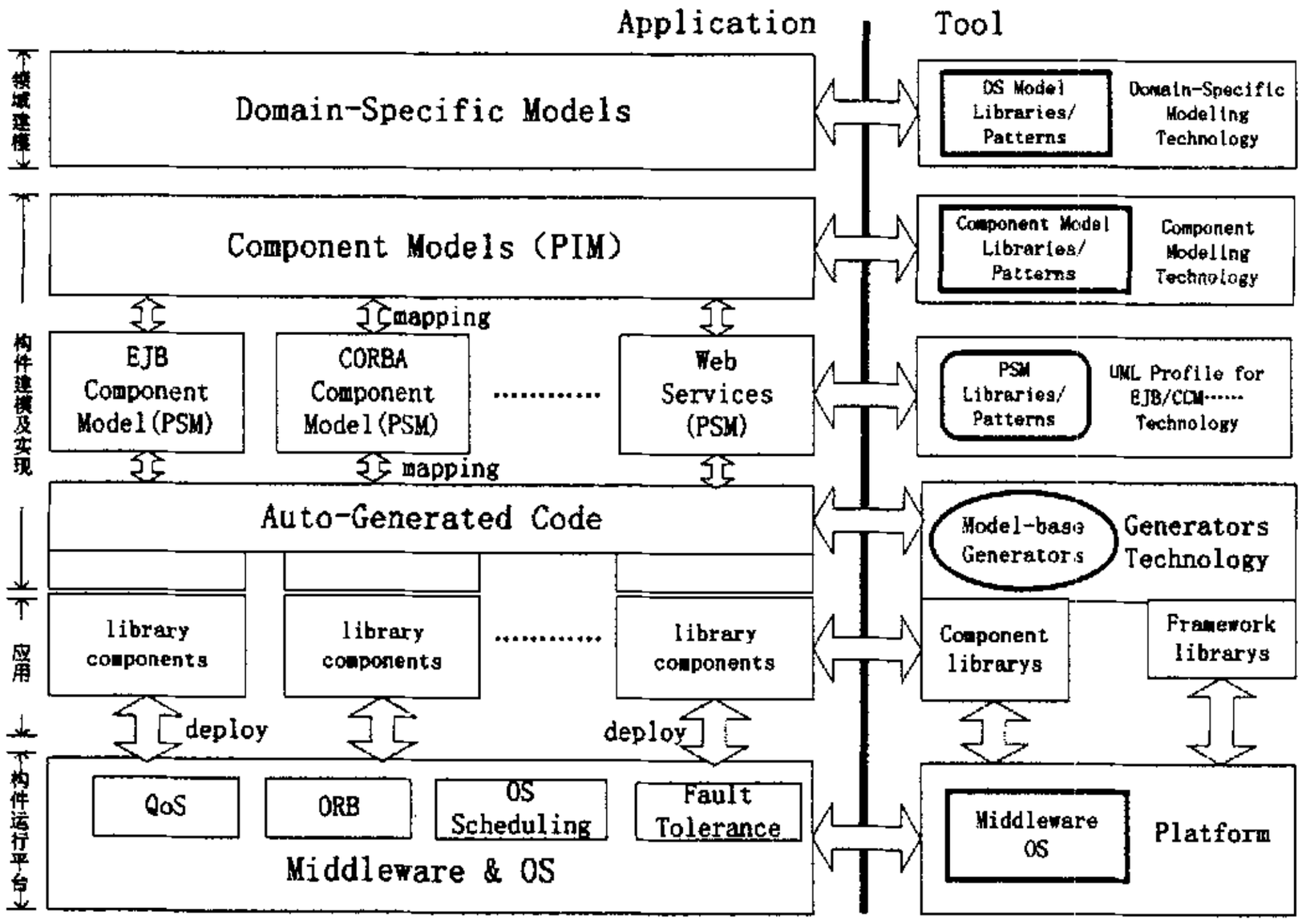

图 4.1 MDAC 方法的体系结构

4.2 MDAC 方法中模型

MDAC 软件开发过程是以系统建模作为驱动力的，模型是软件开发过程中的关键，包括对模型的理解、设计、创建、维护和修改。在详细讨论基于 MDA 的构件开发实现细节之前，首先必须对 MDAC 中模型的类型有一个全面的了解。下图描述了不同种类的模型的基本分类法，这种模型分类法并不是确定不变的。不同的开发过程对模型的分类可能会有差异，或者可能使用不同的术语。

MDAC 中的模型主要分为如下几种类型，如图 4.2 所示：

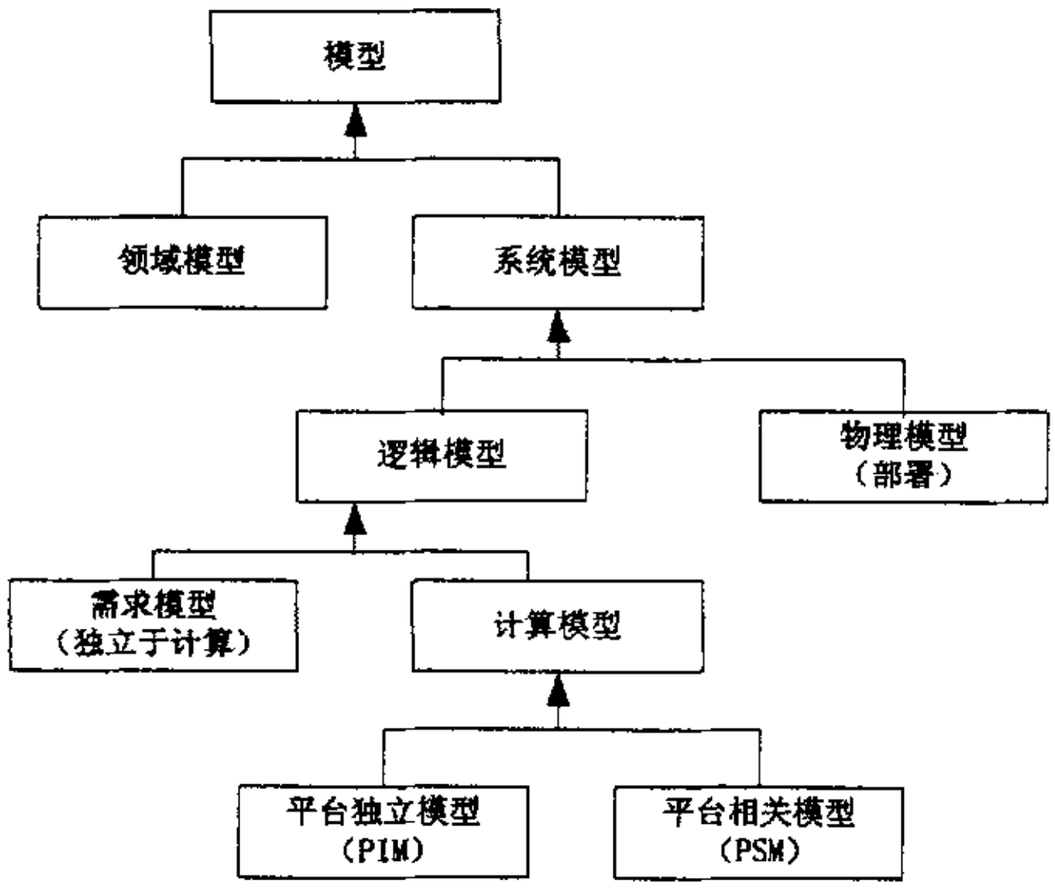

图 4.2 MDAC 的模型类型

（1）业务模型和系统模型。业务模型描述了业务方面，不管这些方面是否要被自动化。系统模型描述了计算机系统把业务元素自动化的方面。因此，系统模型的作用域要比相应的业务模型的作用域小。对于非企业系统而言，更通用的术语领域模型可能要比业务模型更适合。

（2）逻辑模型和物理模型。逻辑模型通过类和行为模型描述了系统的逻辑。物理模型描述了在开发时和运行时用到的物理资源，包含模型文件、源代码文件、可执行文件、档案文件、处理器等等。

（3）需求模型和计算模型。需求模型描述了逻辑系统，而不是业务，但是它是用独立于计算的方式来描述的，这就意味着，需求模型尽可能地不考虑技术因素。计算模型也描述了逻辑系统，但是它考虑了技术因素。计算模型为生成器提供了更完整地信息。

（4）平台独立模型和平台相关模型。平台独立模型（PIM）和平台相关模型（PSM）之间的差异取决于对平台技术参考集合的规定，PIM 独立于这些平台。PIM 和 PSM 都是计算模型。PIM 与技术因素无关，从 PIM 派生的 PSM 则考虑了更多的同特定平台相关的技术因素。

图 4.2 也对各种模型之间的配合关系进行了说明。图中所示阴影方框的模型类型表示分类法中具体的“叶”类型，它们表示实际创建的模型：即业务模型、需求模型、平台独立模型（PIM）、平台相关模型（PSM）和物理模型。上述模型中，除了业务模型之外都其余的都是系统模型，每种模型都反映了系统的不同视点。业务模型的视点离计算环境最远。业务模型可以包含业务中未被自动化的部分，系统架构人员可能对这些部分也感兴趣。因此，他们的视角最宽。需求模型从最抽象的视点描述系统，PIM 提供了稍具体一点的系统逻辑的

视点，PSM 提供了更具体的系统视点，于是，可以认为，PIM 是需求模型的细化，而 PSM 是 PIM 的细化。物理模型同系统逻辑无关，因此它提供了对系统的一个另外的视点。理想情况下，工具可以让你从这些视点中的任意一个来观察系统。每个视点都涉及了开发过程中的特定人员。本文接下来主要讨论 PIM 和 PSM 两类模型。

4.3 MDAC 角色

MDAC 方法是一套系统的构件开发方法，因此，在 MDAC 的整个开发流程中，不同的阶段具有不同的角色，从软件工程的角度看，角色划分对团队开发非常有益。结合参考文献[34]的思想，MDAC 中开发人员的角色大致可以分为以下几种类型，如图 4.3 所示。注意，这种分类方法不固定。

（1）业务分析员

业务分析员（Business Analyst）负责理解公司的业务过程和过程使用的信息，并使用流程图和表来定义业务过程的经验，创建业务模型。业务分析员不需要有任何编程经验，经培训后他也可以使用业务过程建模工具，只是这些工具不是针对计算机专业人员的。例如，学习如何创建 UML 模型。

（2）需求分析员

需求分析员（Requirements Analyst）要求具有技术背景，编过程序，主要负责为业务构件和应用程序创建需求模型，同业务分析员合作来确定业务模型的哪些方面需要被自动化。同业务分析员的合作通常需要贯穿于系统开发和部署生命周期的始终，这样才能确保需求模型同业务模型的同步。

（3）应用工程师

应用工程师（Application Engineer）负责开发业务构件和应用程序，创建并维护 PIM，利用模型转换工具从 PIM 生成 PSM 和代码。根据开发组织策略，应用工程师可以手工增强生成的 PSM 和代码，并和需求分析员一起工作，以确保 PIM 和需求模型保持同步。

（4）中间件工程师

中间件工程师（Middleware Engineer）是掌握 EJB、CORBA 或面向消息的中间件的程序员，并且要熟悉 UML。中间件工程师同应用工程师一起工作以确保应用程序在特定中间件环境下工作良好。

（5）质量保证工程师

质量保证工程师（Quality Assurance Engineer）是经验丰富的程序员，并且熟悉 UML，主要负责测试应用工程师和中间件工程师的成果，当发现问题或是值得怀疑之处时会同他们一起工作。质量保证工程师可以使用从 PIM 生成的测

试框架来测试系统，也可能会手工来增强它们。PSM也有助于质量保证工程师的管理测试过程。

（6）部署工程师

部署工程师（Deployment Engineer）负责系统部署，他可以不必须具备很强的编程背景，但必须具有非常丰富的网络知识。他主要使用自动化部署工具进行工作，工具可以生成指令来获得系统运行时的性能和资源数据，并经常同质量保证工程师和中间件工程师一起分析获得的数据。他有时候也同应用工程师一起工作。

（7）构架师

构架师（Architect）负责定义并维护驱动企业构架的模型，规定不同的建模框架和其它基础设施元素如何配合。同时，构架师还审核各分析员和工程师的工作以确保他们遵从构架，并且收集反馈信息以改善构架。

（8）基础设施工程师

基础设施工程师（Infrastructure Engineer）负责开发和维护基础设施软件。在开发业务构件和应用程序时会用到基础设施软件。基础设施包括系统服务，这些系统服务可能位于第三方中间件之上，还可能包括建模框架。其中有一些是基础设施工程师开发的，还有一些是从第三方购买的。基础设施工程师有时同中间件工程师一起工作，中间件工程师可以帮助基础设施工程师优化生成器。基础设施工程师还定期同应用工程师会晤，以确保基础设施满足应用工程师的需要。

论文前面已经说过，不是所有的开发组织都会用到所有不同类型的模型，此外，一个人可能会扮演多个角色。因此，真实的开发部门可能看上去同这个理想化的角色分类有些不同。

4.4 构件建模框架（CMF）

构件建模是MDAC方法的核心。如前所述，当前COM、EJB、CORBA等技术提供的代码级的构件复用已经不能满足复杂多变的应用需求，构件复用必然向更高层次复用方向发展，如分析规格说明构件复用、设计构件复用、测试构件复用、文档构件复用等，对构件进行建模则是实现构件高层复用的有效途径。同时，构件组装技术是基于构件的软件开发的核心技术，构件必须经过组装才能形成应用系统，才能实现构件的价值。所以，构件建模要解决的核心问题是：构件本身的建模和构件组装的建模，解决了这两个问题，才能真正实现化构件复用为模型复用，化构件组装为模型组装，化构件实现为模型映射。UML自然是建模语言的首选，UML 目前虽然对构件建模的支持较弱，但是，UML

提供了极为灵活的扩展机制允许用户根据实际需要扩展其元模型，而且 UML 本身的一些视图，如类图、状态图、协作图等，已经可以直接用来对构件进行建模，于是，基于 UML，论文提出了一个构件建模框架（CMF，Component Modeling Framework）为 MDAC 方法提供支持，其体系结构如图所示，包括以下几个部分：

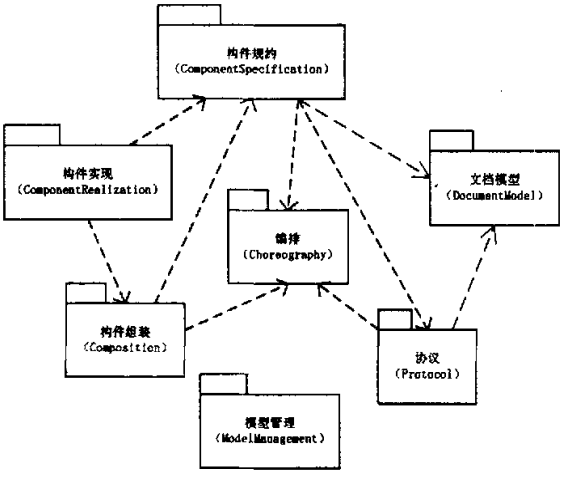

图 4.3　CMF 构件建模框架

（1）　构件规约（Component Specification）

构件是指应用系统中可以明确辨识的构成，同对象一样，具有封装性，所以构件可分为外部契约和内部实现，构件的外部契约就是构件规约。构件可以只有规约，没有内部实现，这类构件称为抽象构件（Abstract Component），抽象构件让构件具有了对象的多态特征。构件分为原子构件（Primitive Component）和复合构件（Composed Component）两类。原子构件是在系统开发中无须再分的最小基本单元，原子构件有其对应的实现体，复合构件在规约层次上表达了成员构件之间的复合，本身并不对应于任何实现体。

CMF 通过扩展 UML 子系统图（Subsystem）来进行构件建模。UML 子系统是具有独立的说明和实现部分的包（UML 的一种组合机制，把各种各样的模型元素通过内在的语义连在一起成为一个整体就叫做包），它代表了与系统其它部分具有整洁接口的清晰单元，通常表现了系统在功能和实现上的划分，可见，UML 子系统基本上满足了构件建模的需求。子系统图示为类似书签卡片的形状，由二个长方形组成，小长方形标签位于大长方形的左上角。

图 4.4 给出了基于 UML 子系统的构件规约视图，构件表示为一个子系统及其扩展端口（Ports）、属性（Property）、类型（t）。构件定义了一组端

口与其它构件互操作，并有一组参数用来在过程构件被使用时进行配置。图中的“Receives”、“Sends”、“Responder”、“Initiator”都是端口，分为流端口（Flow Ports）和协议端口（Protocol Ports）两种类型。所谓流端口是简单端口，它用来描述构件使用外来对象提供的对象引用的能力，这样，构件就可以关联其它构件，并激活这些构件上的操作，“Receives”、“Sends”为流端口；而协议端口是复杂端口，它定义构件消息发送接收端口，这样，构件除了可以通过激活操作相互作用外，构件之间还可以通过检测其它构件的状态来相互作用，“Responder”、“Initiator”为协议端口。CWF 还支持构件属性定义，可以定义属性名（Property）、类型（Type）和值（Value）。

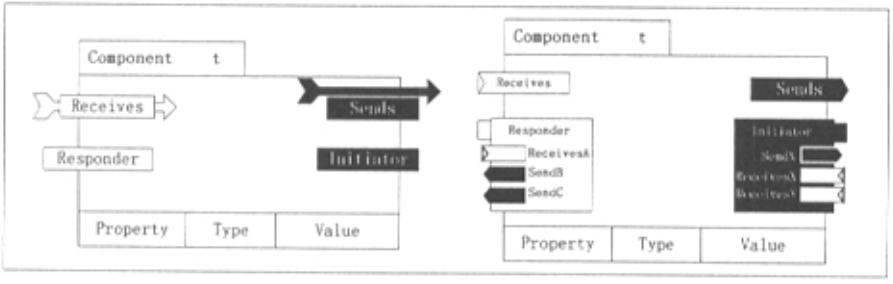

图 4.4　CMF 构件模型图

（2）　协议和编排（Protocol and Choreography）

协议（Protocol）规定了构件在与其它构件进行交互时所传输和接受的消息类型，以及这些消息的编排规则。构件所支持的每个协议是通过端口对外提供。

编排（Choreography）规定了在端口使用者间消息的传输过程，它包括外部编排和内部编排两种类型。外部编排显示构件与构件之间的消息传输，而内部编排则是指在构件内部的消息流传输过程。

协议、端口、编排共同构成了构件的外部行为契约。值得注意的是，协议定义必须要定义两个互补的角色，即“Receives”和“Sends”对应，“Responder”和“Initiator”对应。

CWF 使用 UML 中的状态图（Statechart）及活动图（Activity Diagram）来表示协议和编排。在 UML 中，状态图主要用来描述对象、子系统、系统的生命周期，通过状态图可以了解到一个对象所能到达的所有状态以及对象收到的事件对对象状态的影响等。状态图指定对象的行为以及根据不同的当前状态行为之间的差别，同时它还能说明事件是如何改变一个类的对象的状态。活动图是状态图的一个变种，主要目的是描述动作（执行的工作和活动）及对象状态改变的结果，即描述采取何种动作做什么（对象状态改变），动作何时发生（动作序列），以及在何处发生，活动图和状态图的一个区别是活动图中的动作可以放在泳道中，泳道用来聚合一组活动并指定负责人和所属组织。可见，UML 状态图和活动图基本上满足了构件状态建模的需求。状态图中的状态用一个圆角

四边形表示，状态图可以有一个起点和多个终点起点（初始态），用一个黑圆点表示终点（终态），用黑圆点外加一个圆表示，状态之间为状态转换用一条带箭头的线表示，引起状态转换的事件可以用状态转换线旁边的标签来表示，活动图的表示方法与状态图一样，只是泳道用纵向矩形来表示。

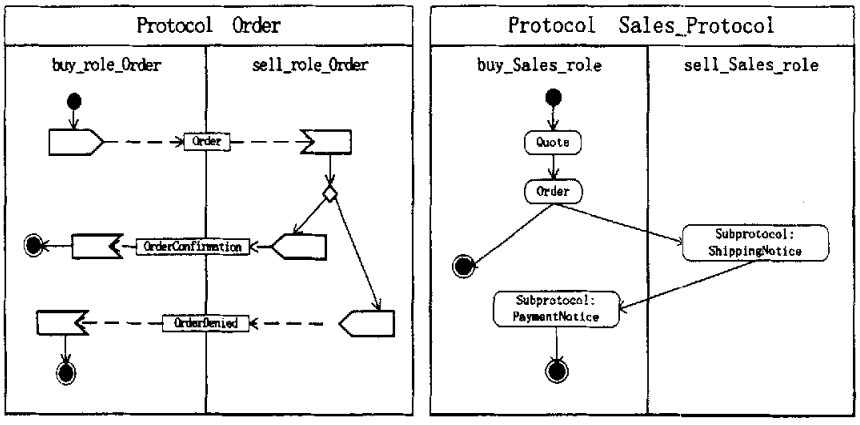

图 4.5　CMF 协议模型图

图 4.5 给出了 CMF 的协议和编排的建模视图的示例。协议角色（Protocol Roles），包括 buy_role_Order、sell_role_Order、buy_Sales_role、sell_Sales_role，由泳道来表示，Initiator（buy_role_Order、buy_Sales_role）位于最左边，协议角色的名字标在泳道的上方。消息使用信号来表示，分为发送信号和接收信号，发送信号由凸出五边形表示，接收信号由凹入五边形表示，其间用虚线箭头连接，如果是发送信号，则箭头指向对象，如果是接收信号，则箭头指向接收符号。子协议（subprotocol）表示为中间状态。

（3） 组装（Composition）

组装定义了构件如何被使用，通过组装，构件被使用、配置和连接，组装的结果是生成一个更大的复合构件。可以对符合需求的任何类型的构件进行组装，即可以在原子构件、复合构件、抽象构件之间灵活组装。对构件组装建模主要需要解决以下问题：（1）构件组装流程的表示，即如何编排构件以让它们之间如何协同工作。（2）构件之间连接的表示；（3）复合构件属性值的配置；（4）如何绑定一个具体构件供复合构件使用，并且正确控制其生命周期（Component Usage）。

CMF 主要扩展了 UML 协作图（Collaboration Diagram）对构件组装进行建模。在 UML 中，序列图（Sequence Diagram）和协作图都可以用来描述协作对象间的交互和链接，但是，序列图强调的是时间，而协作图强调的是空间链接，显示的是真正的对象以及对象间是如何联系在一起的，也可以只显示对象的内部结构（构成对象的对象显示在对象的内部），可见，协作图更能满足构件组装建模的需要，因此，本文选择使用协作图进行构件组装建模。在 UML 中，协

作图使用同类一样的符号来表示对象，但是对象的名字下面有下划线（对象符号），链接用线条来表示，在一条链接上，可以给消息加一个消息标签用来定义消息的序列号。

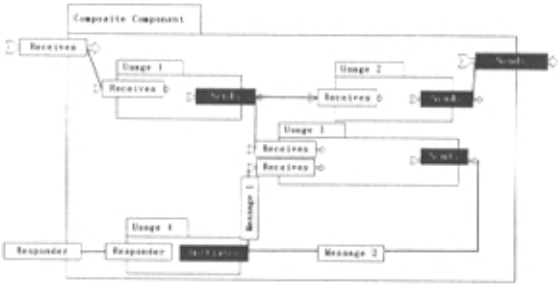

图 4.6 CMF 构件组装建模图

图 4.6 给出了 CMF 中的构件组装建模的示例。在图中，复合构件由若干具体构件（Usage1、Usage2、Usage3、Usage4）连接而成，端口的表示方式与图 4.4 中构件端口表示方式相同，只是复合构件的端口称作其内部构件端口的代理端口（Port Proxy）。图中构件之间所有的连线被称为连接器（Connectors），连接器上的消息注释（Message1、Message2）表明了该消息同属于协议端口（Initiator），并且 Usage3 可以把它们当作数据流来处理。构件属性的表示按需要而定，并不强制规定。

（4） 文档模型（Document Model）

文档模型用来描述构件及复合构件的结构、数据处理流程等相关的信息，以用于建模文档的生成。文档模型是 CMF 框架提供的一个扩展功能，本文不详细讨论其如何实现。

（5） 模型管理（Model Management）

模型管理是 CMF 提供的一种帮助建模人员管理模型的机制。与 UML 类似，CMF 使用包（Package）来分层组织构件模型。建模人员可以将模型分配至一系列包，但为了可行，分配必须遵守一些基本原则，如通用功能、紧密耦合、相同视角。如果包经过良好的选择，它们可反映系统的高层次结构、子系统分解和依赖关系等内容，所以，将系统合理的分解为包可以极大的提高系统模型可维护性。

4.5 构件模型转换

4.5.1 实现原理

论文在第三章已经阐述了 MDA 的模型转换映射过程，MDAC 方法完全按照 MDA 的模型转换规则进行构件模型的映射，因此，一个 CMF 构件模型从建

模到实现也要经历如下两个基本过程：PIM 转换到 PSM，PSM 映射到实现。其实，如果完全按照标准的 UML 元模型对构件进行建模，整个映射过程目前已经可以实现，OMG 也为这一过程制定了许多标准，比如 UML Profiles for EJB、UML Profiles for CORBA。但是，正是由于标准的 UML 元模型不能满足构件建模的需要，论文才提出了 CMF 构件建模框架对其进行扩展，如果仅仅是为了让建模人员之间能够顺利交流思想，CMF 已经能够满足要求，但这并不是 MDAC 方法的最终目的，MDAC 方法的目标是能够根据模型自动生成大部分构件代码，因此，除了让建模人员能够明白模型的语义之外（CWF 中构件、协议、编排等模型的含义），CWF 构件模型必须要能够被机器可识别，只有这样，才能通过模型转换工具对 CWF 构件模型进行转换。因此，构件模型转换要解决的主要问题就是要把 CWF 模型转换为标准的 UML 模型。而标准的 UML 模型已经能够完成从 PIM 到 PSM，PSM 到实现映射，从而最终实现了基于模型的构件开发这一根本目的。

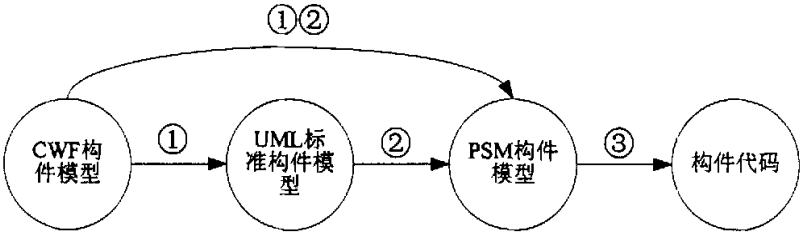

图 4.7　MDAC 的模型转换实现原理图

综上所述，MDAC 中的模型转换过程可以概括为如下几个阶段（如图 4.7 所示）：（1）CWF 构件模型到 UML 标准模型的转换（属于 PIM 层的转换）；（2）UML 标准 PIM 模型根据 UML Profiles 标准映射规则转换为具体平台的 PSM 模型；（3）PSM 到代码的映射。其中第（1）（2）步可以合而为一，可以直接利用建模工具实现 CWF 构件模型（PIM）到 PSM 的映射。

现在，继续回到基本问题上来讨论，即如何把 CWF 模型转换为标准的 UML 模型。由于 CMF 是论文提出的一种直观的构件建模框架，因此并没有相应的 OMG 标准转换规则能够直接利用，这就必须要求建立一套 CWF 元模型和标准 UML 元模型之间的映射规则。MOF 的强大功能、XMI 的通用性和 UML 的扩展机制为此提供了依据和可能。

MOF 是 MDA 的关键基础，它是一种面向对象的元元模型（meta-meta-model）。元元模型就是对元模型进行定义的一种模型。对模型进行定义、验证以及提供模型级交互性的需求催生了元元模型。如图所示，这种模型架构是建立在四层元数据体系结构基础上的，可见，这种元模型可以是通用的，如 UML，也可以是针对特殊应用领域的，如 CWM。MOF 允许在特定的实现技术下将 MOF 定义的元模型实例化或映射到其它 M2 层元模型和 M1 层

产物，目前，MOF 支持三种类型的映射：抽象映射、IDL 映射、XML 映射。抽象映射是一组规则，它确定了 meta-model 里的各种语义是如何对依据该 meta-model 定义的 model 进行限制和规范的；IDL 映射是一组标准模版，可以将一个用 MOF 定义的 meta-model 映射成一组 CORBA IDL 接口，这组接口的用途是访问一组 CORBA Object，而这组 CORBA Object 则是表现用这个 meta-model 定义的 meta-data 的。这些 IDL 接口的典型用途是用在存放元数据的存储上；XML 映射产生该 meta-model 对应的 XML DTD，以及提供一组规则将 meta-model 定义的 model 映射成遵守该 DTD 的 XML 文件。这些标准的 XML 文件非常适合异构的环境下传递[35][36]。

元层次	描述	元素
M3	MOF，即定义元模型的构造集合	MOF 类、MOF 属性、MOF 关联等
M2	元模型，由 MOF 构造的实例组成	UML 类、UML 属性、UML 状态、UML 活动等；CWM 表、CWM 列等
M1	模型，由 M2 元模型构造的实例组成	“Customer”类，“Account”类；“Employee”表、“Vendor”表等
M0	对象和数据，也即 M1 模型构造的实例	客户“张三”、账户“88787434322”、员工“李四”等

表 4.1　MOF 的层次结构

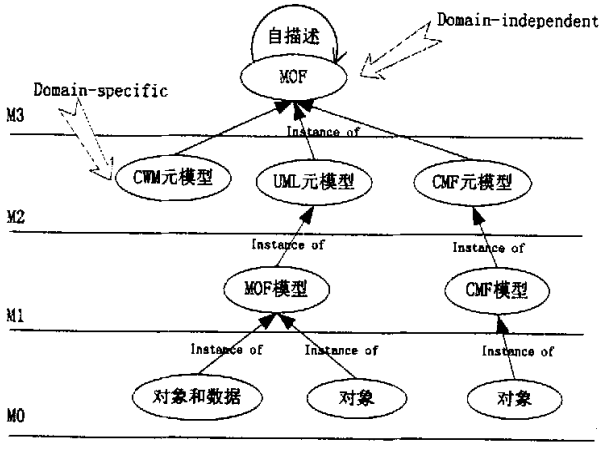

图 4.8　模型转换原理

MOF 的强大在于它使得不同的元模型可以互操作，能够理解 MOF 的应用程序可以在不了解特定领域模型实例接口的情况下，仍然能够使用反射接口提供的通用操作来读取和更新模型。MOF 语义还定义了元数据仓库服务(metadata repository services)，它支持模型构建(construction)、发现(discovery)、移动(traversal)、更新(update)。前提条件是此模型必须是可被理解的某种元模型的实例。MOF 支持模型生命周期语义，意味着 MOF 的实现提供了一种有效的元数

据授权和分发工具。例如，新开发的元模型可以保存在 MOF 仓库中，根据 MOF 生命周期语义和复合语义(继承 inheritance、集群 clustering、嵌套 nesting 等)，它能够和现存的元模型组合。随后，模型接口和默认的实现可以被生成，并对环境可用。模式实现可以通过手工或者工具来附加其它编程逻辑被增强(例如使用 OCL 约束)。一个完全兼容 MOF 的仓库可以提供众多的元数据服务，比如持久性(persistence)、版本(versioning)和目录服务(directory services)等。

XMI 将 MOF 映射到 W3C 的 XML 语言上。XMI 定义了 XML 标记(tags)如何表示序列化的兼容 MOF 的模型。基于 MOF 的元模型被转换为 DTDs 或者 XMLSchema，模型根据其对应的 DTD 或者 XML Schema 被转换为 XML 文档。以前基于标记的语言在表示对象和对象的关联上存在许多的困难，而这在 XMI 中得到了解决。此外，XMI 基于 XML 的特性，意味着元数据(用 tags 表示)和其实例(元素内容)可以共存于同一个文档中，这使得应用程序可以容易地通过其元数据来理解实例。XMI 同时具有自描述和天生的同步特性，这就是为什么基于 XMI 的交换在分布式的、异构环境中是如此重要的原因。

下面来看 UML 的扩展机制，称为 UML Profile，主要包括约束（Constraint）、标记值（Tagged Value）和构造型（Stereotype）三种方式，如图所示。约束是用文字表达式表示的语义限制。每个表达式有一种隐含的解释语言，这种语言可以是正式的数学符号，如集合论表示法；或是一种基于计算机的约束语言，如 OCL；或是一种编程语言，如 C++；或是伪代码或非正式的自然语言。约束用大括弧内的字符串表达式表示。约束可以附加在表元素、依赖关系或注释上。标记值是一对字符串——一个标记字符串和一个值字符串—存储着有关元素的一些信息。标记值可以与任何独立元素相关，包括模型元素和表达元素。标记是建模者想要记录的一些特性的名字，而值是给定元素的特性的值。标记值用字符串表示，字符串有标记名、等号和值。构造型是在一个已定义的模型元素的基础上构造的一种新的模型元素。构造型的信息内容和形式与已存在的基本模型元素相同，但是含义和使用不同。构造型可以用标记值来存储不被基本模型元素所支持的附加特性。

通过对 MOF、XMI 和 UML 扩展机制的分析可见，MOF 提供了一种定义元模型的能力，而且具有灵活的映射机制，XMI 提供了一种数据转换的统一机制，而 UML 扩展机制允许使用标准的 UML 符号构造新的 UML 注释符号，结合二者的特点，论文提出一种解决 CWF 模型转换问题的方法：即建立 CMF 元模型，然后把 CMF 元模型通过 XMI 技术映射到标准的 UML 扩展 Profile，而 Profile 提供了一种重用 UML 元模型的机制，这样就可以用标准的 UML 元模型来表示 CMF 构件模型，从而实现了从 CMF 模型到 UML 标准模型的转换。其优点在于使用 MOF 建立的元模型与具体的建模语言无关，可以根据需要随时

进行扩展，而 XMI 是模型的转换过程更加标准化和灵活，同时，通过对 UML 增加 CMF Profile，可以直接利用现有的一些 UML 建模工具。

4.5.2 CMF 元模型及映射规则

CMF 框架的元模型的主要元素如图所示，根据 CMF 框架，该元模型可以从以下三个方面来讨论：Component 元模型，Choreography 元模型，Composition 元模型。其中，由于 CMF 构件建模扩展了 UML 子系统的图，而编排和组装则直接利用了 UML 的状态图和协作图，所以 Component 元模型相对复杂，而后两种则比较简单，与 UML 对应的元类类似。

（1）Component 元模型

Component 元模型如图所示，由图可知，Component 描述了一个构件的契约，定义了构件的行为，有 granularity、isPersistent、primitiveKind、 primitiveSpec 四种类型，主要通过 Port 和 Choreography 定义来实现。Port 是 Component 之间的连接点，由 Flow Port 和 Protocol Port 两种类型，一组行为紧密结合的 Port 称为 Multi-port。 Component 可以有 Property Definition 定义。Component 也可以通过 Composition 成为复合构件，复合构件的内部子构件之间的执行序列由 Choreography 来编排。

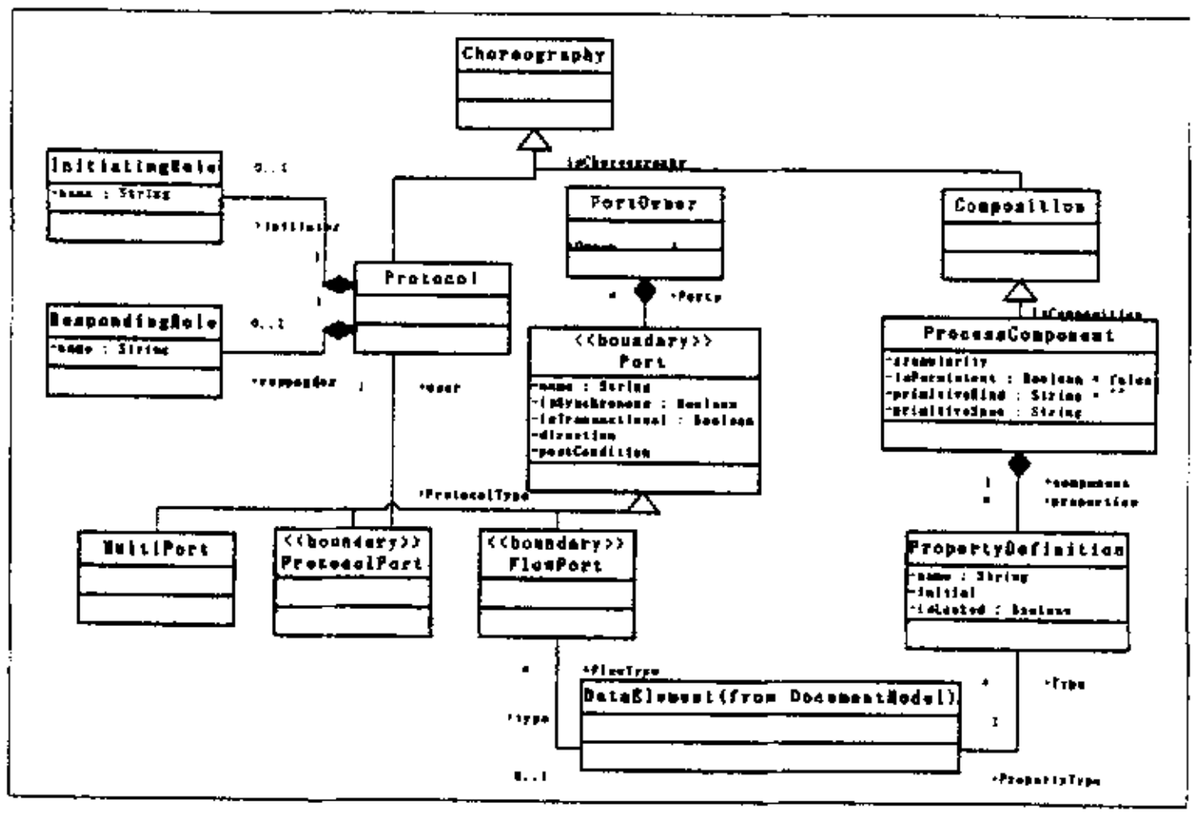

图 4.10　Component 元模型

根据 UML 的扩展机制和 CMF 构造规则，论文在图中给出了 Component 元模型到 UML 标准元模型的映射规则。由图可知，Component 元模型元素 Component 映射为 UML 的构造型（Stereotype）Component，具有标签（Tags）granularity、isPersistent、primitiveKind、 primitiveSpec 及相应的值，继承自 UML 核心类 Classifier；Port 映射为 UML 的构造型 Port，具有标签 isSynchronous、isTransactional、direction、postCondition 及相应的值，继承自 UML 核心类 Class，FlowPort、ProtocolPort、MutiPort 映射为 UML 的构造型 Port 的子类；Protocol

映射为 UML 的构造型 Protocol，具有标签 initiatingRoleName、respondingRoleName 及相应的值，继承自 UML 核心类 Class；PropertyDefinition 映射为 UML 的构造型 PropertyDefinition，具有标签 isLocked 及相应的值，继承自 UML 核心类 Attribute。

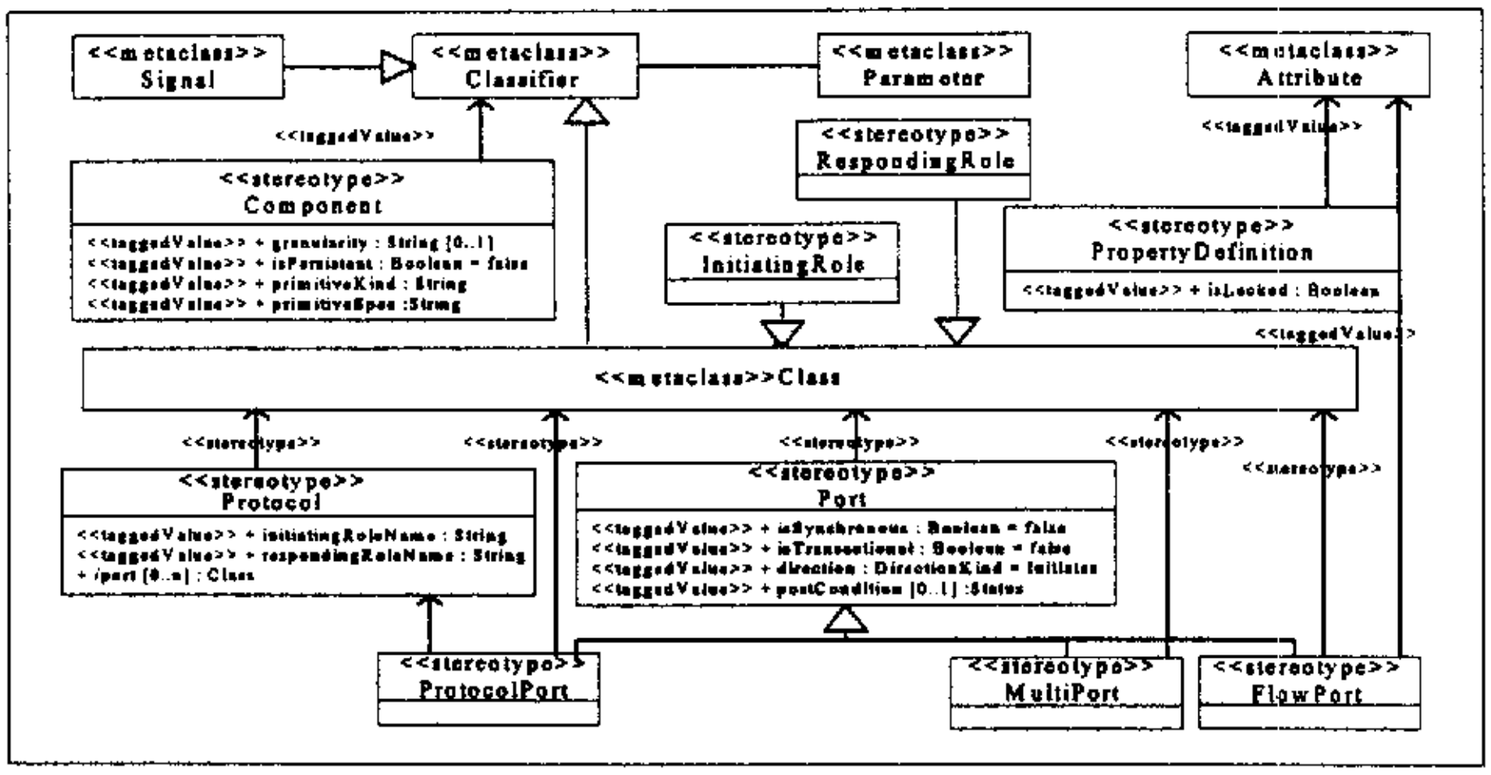

图 4.11　Component 元模型映射规则

（2）Choreography 元模型

Choreography 元模型如图所示，有图可知，整个编排过程 Choreography 中的对象由不同的 Node 构成，一个 Node 代表一个 PortUsage，一个 PortUsage 表示一个使用的 Port 实例，与 UML 类似，每个 Node 有相应的伪状态 PseudoState，即 success、failure 等状态。在编排过程中，Node 也必须有相应的执行状态，Choreography 使用 Connection 和 Transition 来定义 PortUsage 的执行状态，Connection 定义了 PortUsage 的执行顺序，Transition 则规定了 PortUsage 在满足某种条件（Precondition）时应该执行的动作和具有的状态。Choreography 元模型与 UML 状态机元模型类似。

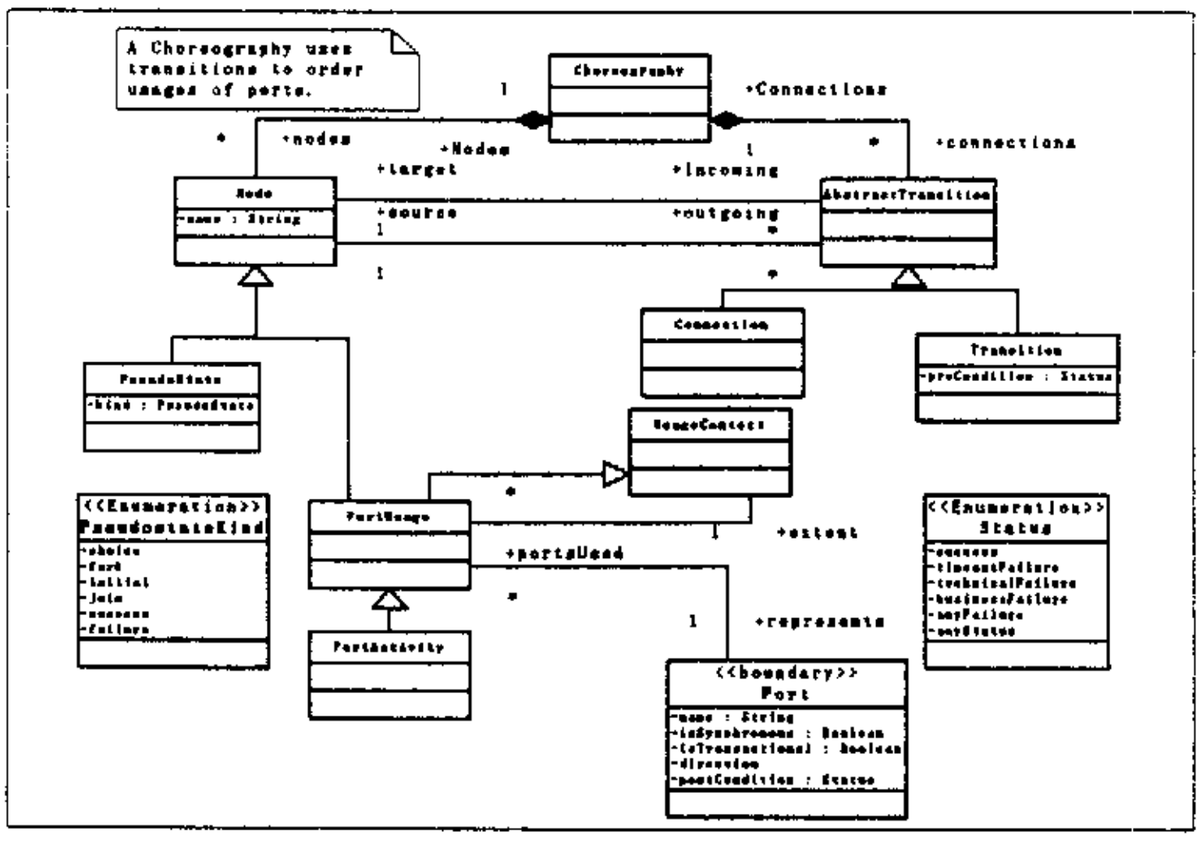

图 4.12　Choreography 元模型

根据 UML 的扩展机制和 CMF 构造规则，论文在图中给出了 Choreography

元模型到 UML 标准元模型的映射规则。由图可知，Choreography 元模型元素 Choreography 映射为 UML 的构造型（Stereotype）Choreography，继承自 UML 元类 StateMachine。由于 Choreography 建模时直接基于 UML 状态机，所以 Choreography 元模型的映射比较简单。

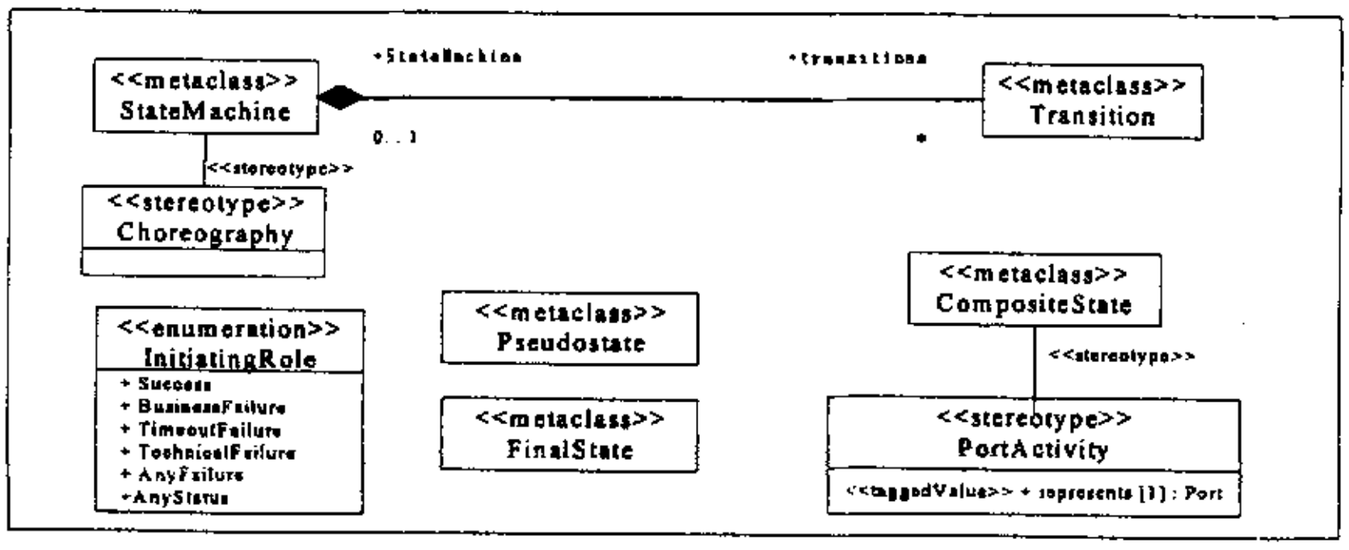

图 4.13　Choreography 元模型映射规则

（3）Composition 元模型

Composition 的元模型如图所示，由图可知，Composition 通过 ComponentUsage 来表示构件之间如何进行组装。由于同一构件可以用作不同的用途，为了加以区分，每一个被使用的 Port 都有通过一个 PortConnector 与其它 Port 相连。如果把组装的结果当作一个复合构件来看待的话，Composition 可以是 Choreography 的一个 Node，所以 Composition 也可以具有一系列的 Connection 和 Transition。

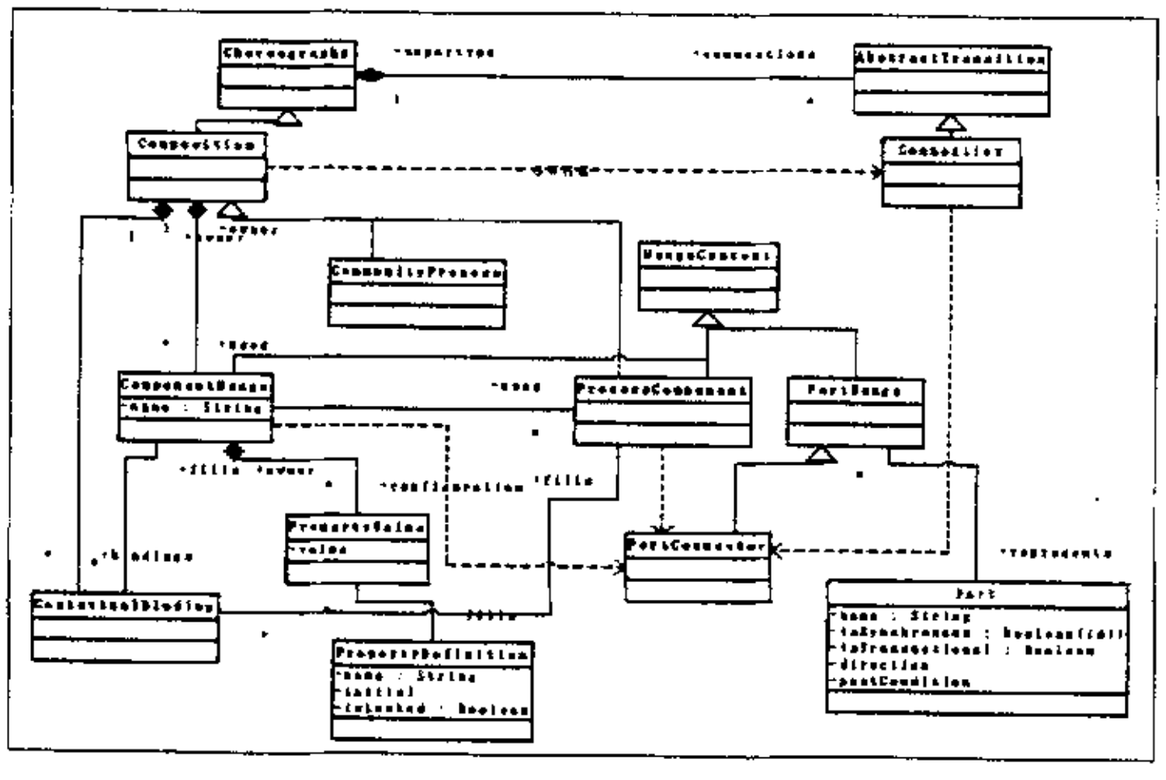

图 4.14　Composition 元模型

同样，根据 UML 的扩展机制和 CMF 构造规则，论文在图中给出了 Composition 元模型到 UML 标准元模型的映射规则。由图可知，Composition 元模型元素 Composition 映射为 UML 的构造型（Stereotype）Composition，继承自 UML 元类 Collaboration；ComponentUsage 映射为 UML 的构造型 ComponentUsage，继承自 UML 元类 ClassifierRole；PortConnector 映射为 UML 的构造型 PortConnector，继承自 UML 元类 ClassifierRole；Connection 映射为 UML 的构造型 Connection，继承自 UML 元类 AssociationRole；PropertyValue

映射为 UML 的构造型 PropertyValue，继承自 UML 元类 Constraint。由于 Choreography 建模时直接基于 UML 协作图，所以 Composition 元模型的映射也比较简单。

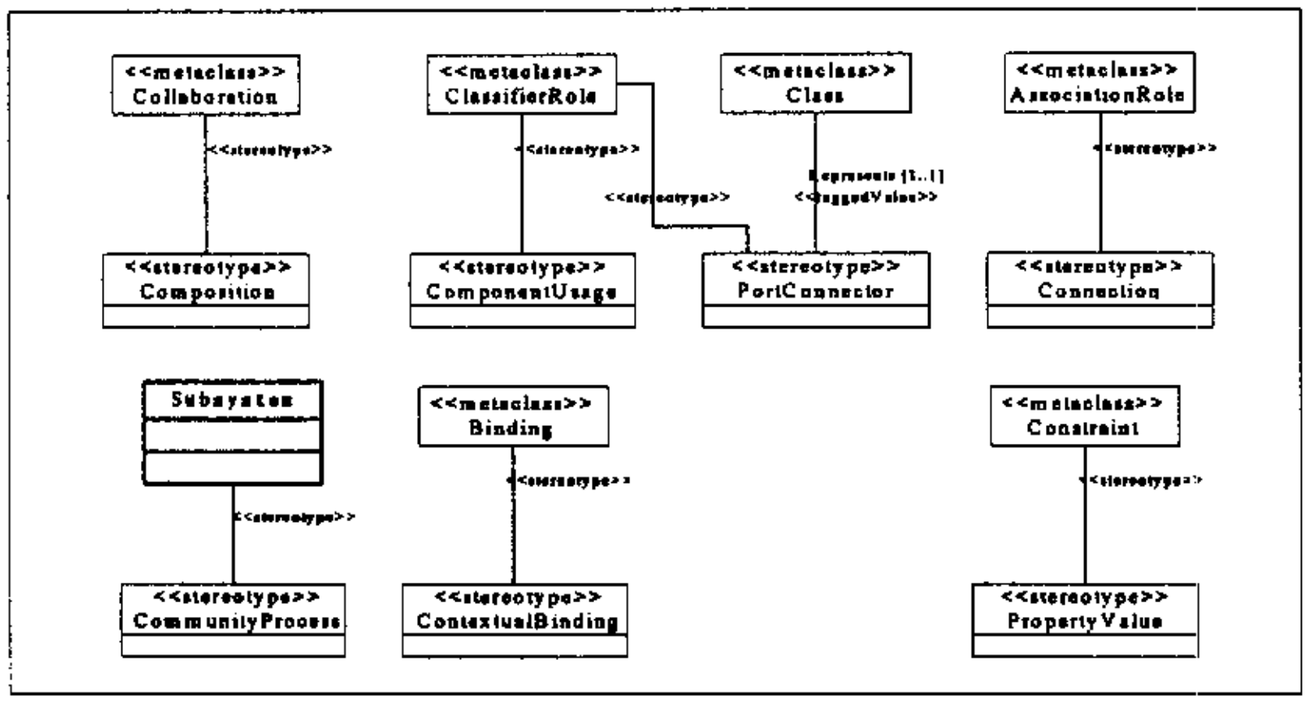

图 4.15　Composition 元模型映射规则

4.6 构件运行平台

构件运行平台是 MDAC 方法得以实现的另一关键因素。MDAC 方法中的构件运行平台主要指中间件或应用服务器，负责对构件进行管理，通过提供连接会话、负载均衡、线程池、持久、安全等服务，为构件提供运行时环境。中间件，从本质上是对分布式应用的抽象，因而抛开了与应用相关的业务逻辑的细节，保留了典型的分布交互模式的关键特征。经过抽象，将纷繁复杂的分布式系统经过提炼和必要的隔离后，以统一的层面形式呈现给应用。应用在中间件提供的环境中可以更好地集中于业务逻辑上，并以构件化的形式存在，最终自然而然地在异构环境中实现良好的协同工作。

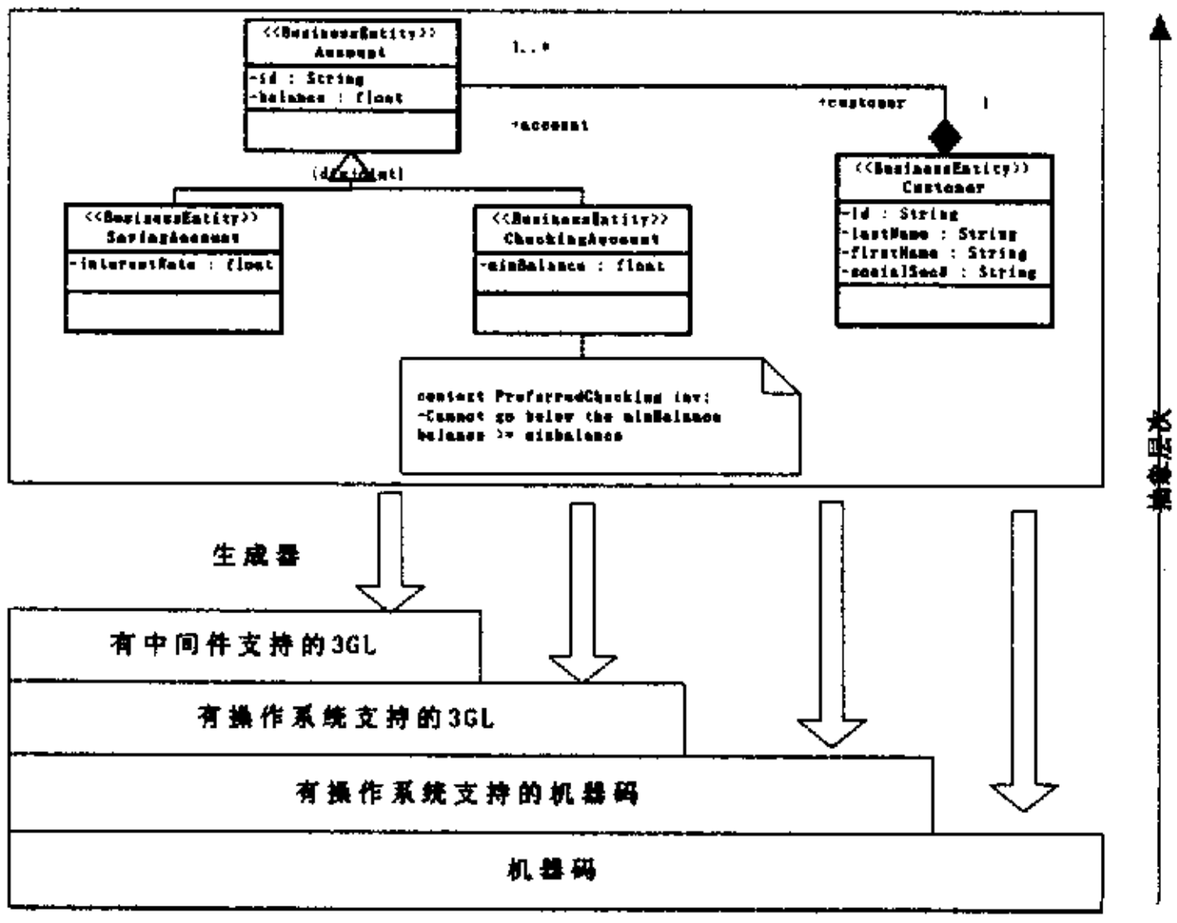

图 4.16　中间件让抽象沟壑变窄

从建模的观点来看，在 MDAC 方法中，构件运行平台提升了计算平台的抽象层次，从而简化了建模和模型转换的工作。如图所示，要实现一个处理模型的代码生成器，一方面如果直接产生机器代码，则必须要实现交易、命名、安全等基本服务生成 3GL 代码的工具，另一方面，如果有了中间件，则只需要生成可依赖中间件提供的服务的 3GL 代码的生成器，相比之下，前者的实现要困难得多。可见，中间件为应用提供了更高的抽象层次，让系统设计和开发人员可以集中精力处理业务逻辑，而不需要关心多用户访问、交易管理、安全等一系列系统级底层技术的实现。

MDAC 构件运行平台主要涉及以下几个方面的内容：

（1）构件运行基准平台。提供构件运行的基础框架，支持 Internet 环境异构、开放、动态和多变等特性。在服务方面，提供命名、安全、事务、日志、数据库连接池、消息等基本服务；在互操作协议方面，要能够支持 IIOP 、JRMP、SOAP、等标准协议；在工具方面，要求提供可视化的部署和管理工具。

（2）构件互操作技术。中间件的基本功能是支持软件之间的互操作，不同中间件技术提供不同的互操作技术，因此，需要支持构件接收和发送基于不同互操作协议的消息。可定制、易扩展的互操作框架是一个主要途径。

（3）构件演化技术。在保持系统运行的前提下，实现构件的增加、删除、替换等操作，以满足当今用户需求持续变化的现状，进而为高质量的大中型软件系统提高基础支持。主要涉及构件运行状态管理、构件动态加载 / 卸载、路由请求策略、请求缓冲等技术。

（4）中间件的动态特征，即反射中间件。在动态、开放、多变的 Internet 环境下，中间件的“黑盒思想”严重限制了中间件平台及其应用的适应能力，因此，在中间件平台的构造中引入反射原理，以允许动态监测与调整中间件平台及其应用。

（5）面向服务体系结构（SOA）。Web Services 的兴起标识了未来的 Internet 软件之间的集成将是一种面向服务（大粒度、松耦合、动态绑定）的模式。Web Services 是独立的、模块化的应用，它使用系列的 WWW 标准来描述（WSDL）、发布（UDDI）、定位以及调用（SOAP），具有松散耦合的特点，更易于应用程序集成。Web Services 可以看作是基于 Internet 的构件。

4.7 工具支持

MDAC 构件开发方法需要有一系列的工具给以支持。根据论文前面对 MDAC 框架的分析可以看出基于 MDAC 构件开发的过程中包括构件建模工具、模型转换工具、构件实现工具和构件运行平台等各种工具的支持。在本节中主

要讨论的是构件建模工具以及相关的转换工具的，而对于构件运行平台等工具，是和其他的构件开发方法的一样，对构件进行相应平台的部署，在此就不详细讨论了。

（一）建模工具

一般来说 UML 作为标准建模语言，可以使用合适的 UML 建模工具直接图示化地编辑模型，这些建模工具是由 UML Profile 支持。这些模型，包括它们的软件构件或文本形式，可以被企业知识库复用。

作为插件集合在 Eclipse 集成环境中 Objecteering UML 建模工具支持 CMF 模型系统结构中不同层次的模型开发。它实现一组规则关联到 UML 模型元素：表示规则（Presentation rules），确认规则（validation rules）和转换规则（transformation rules）。模型规则管理提供机制允许基于规则定义的高级别的自动化。

每个建模工具是相同模式，提供对不同模型开发和使用的支持。一个 UML Profile 定义了相应 UML 元模型的规则和约束，就如同图示的构造型。确认规则确保模型符合相应的 UML Profile；表示规则确保正确的图示显示，转换规则保证不同输出格式之间的映射。

（二）模型转换工具

模型转换工具现在主要分为三种类型：PIM 模型到 PSM 模型的转换工具，PSM 到代码（Code）的转换工具和直接由 PIM 到代码转换工具。

PIM 到 PSM 的转换工具实现从一个高层的 PIM 转换为一个到多个的 PSMs。这一类型的工具虽然是近来才出现的，但随 UML Profile for Java 和 UML Profile for CORBA 等一系列规范的出现已经成为 MDA 方法研究的热点，也出现了很多不错的转换工具，如 Rational Rose 等支持模型转换的工具。支持 MDA 方法的工具提供的从 PSM 到代码的转换一般是指黑盒转换。它们有一个内置的转换定义，并使用一种预定义类型的模型作为源模型以及生产另一种预定义类型的模型作为目标。源模型既是 PSM，目标既是代码。事实上，传统的 CASE 工具的代码生成就是这种模式。这一类的工具发展的较早，也是现在 MDA 方法中最为成熟的一类工具。另一类的工具同时支持 PIM 到 PSM 的转换和 PSM 到代码的转换，这时使用者将只看到直接从 PIM 到代码的转换，而 PSM 被隐含了。这一类的工具，源语言、目标语言和转换定义都建立在工具内部。

UMT 是一个支持模型转换和代码生成的工具，它基于 XMI 格式的 UML 模型。XMI 模型被工具输入，并转换为一种简单的中间形态格式，它是不同目标平台确认和生成的基本。这个中间格式是 XML/HUTN 格式的，在 UMT 中被称为 XMI-Light。工具的最终使用者不需要关心这个文件的格式，然而代码生成器的开发者需要了解它的细节。使用过程就是输入 UML XMI 模型而为目标

平台生成代码。

（三）转换定义工具

转换定义工具支持创建和修改转换定义。当人们不能仅使用现有的转换而需要创建自己的转换定义时，这一类工具将被使用。我们所能遇见的唯一一类的转换定义工具是特定工具的脚本语言，已在前面的章节中说过。MDA 对于复杂转换定义的大量需要驱动了对转换定义语言（QVT）和适合这种任务的工具的需求。大多数柔性工具需要允许一种新的语言定义实现插件和使用在转换中。这种工具现在市场上还没有成品出现，但有些工具已宣称支持该功能。

4.8 MDAC 构件开发过程

基于前面几节的分析，论文将在本节对 MDAC 构件开发的过程做一个详细的总结。整个开发流程将从 MDAC 的最早的生命周期开始，即当业务分析者和结构设计者定义一个最早的高级别的构件的体系结构，到实际上建立构件所要考虑的各种技术细节，以及到最后的构件组装和部署到运行平台。其步骤如图 4-17 所示，整个过程主要以模型设计和映射为基础，可以分为如下几个步骤：

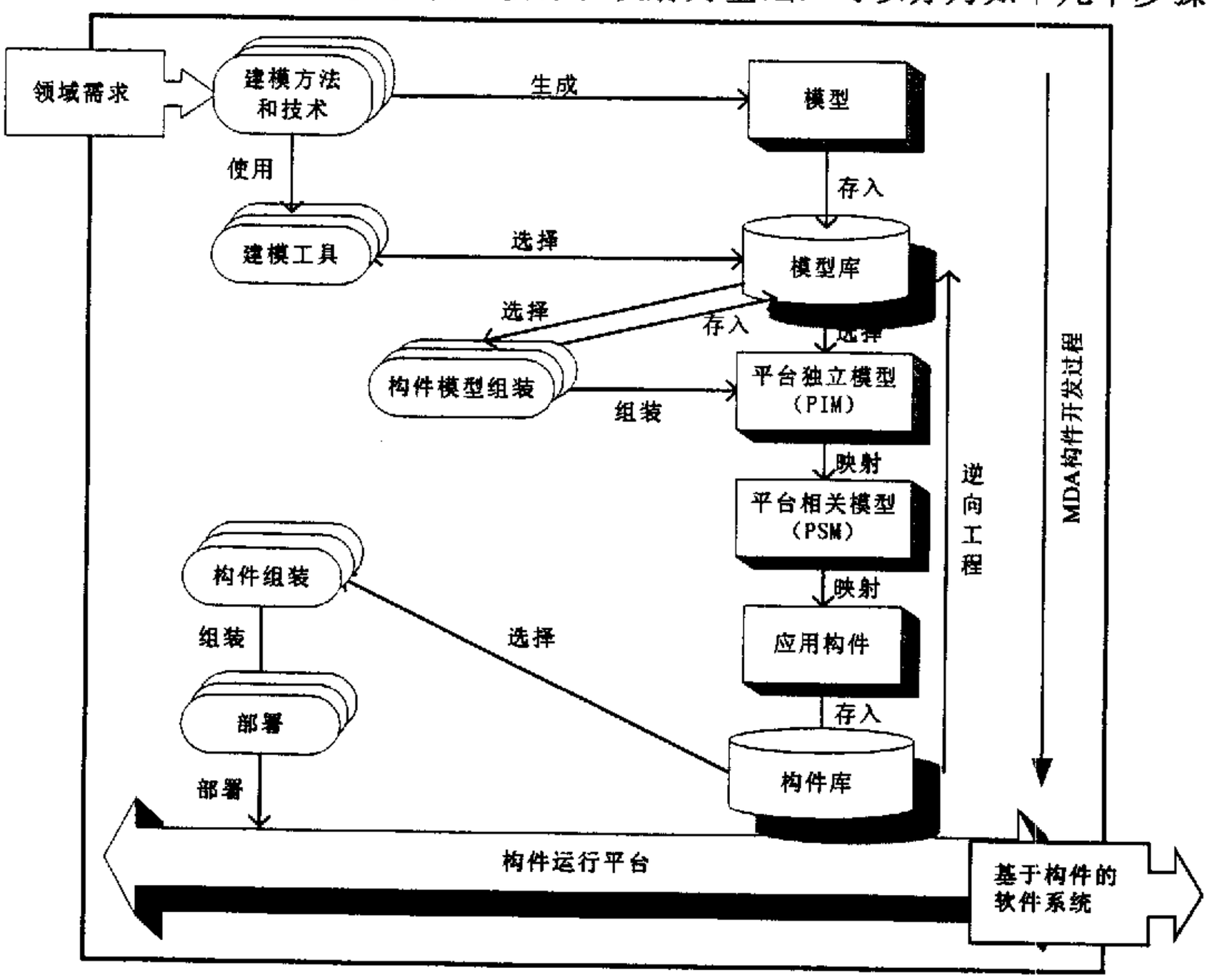

图 4.17　MDAC 构件开发过程

（1）领域分析阶段，建立业务和需求构件模型。

领域分析在对领域中若干典型成员系统的需求进行分析的基础上，考虑预

期的需求变化、技术演化、限制条件等因素，确定恰当的领域范围，识别领域的共性特征和变化特征，获取一组具有足够可复用性的领域需求，并对其抽象形成领域模型，即业务模型，然后，以业务模型为基础，建立需求模型。在一般情况下，业务建模和需求建模往往是结合在一起的，特别在构件开发的过程中，为了能够创建出合适的需求模型，业务分析员应该熟悉相关构件实现模型（如 EJB、CORBA、CCM 等），这样，有利于减少从业务模型到需求建模转换中出现的理解不足。

在领域分析阶段，业务分析员和需求分析员能够在一个高层次上对业务过程和需求进行建模，采用的是针对领域而与实现环境无关的方式，他们可以集中精力于确定业务构件流程以及这些构件之间的关联。其中，构件粒度的选取是一个关键问题，构件粒度的大小对于构件的组装有直接的影响。如果构件的粒度太小，就无法减少软件的复杂度，更不具备复用的价值；构件粒度太大，则构件本身的复杂度太高，同时又降低了可复用程度。合适的构件粒度与构件所属领域的应用特征密切相关。

该阶段的详细过程如下：业务分析员分析业务需求后建立粗粒度的模型交与需求分析员，需求分析员在详细分析模型后，首先查询模型库，利用已有的模型增加功能；或是依赖建模工具直接建模，同时所建立的模型也存入模型库中。

（2）建立和细化 PIM 构件模型。

一个 MDAC 构件模型可以有多个 PIMs，但是只有一个基本 PIM 包含了平台无关的系统行为，即业务需求功能，其它的 PIMs 则包括一些与具体平台无关的技术细节（如构件类型、服务、配置信息等）。通过这种方式，可以使 PIM 到 PSM 的映射更加准确。

为了PIM模型的语言定义更加精确，可以使用UML中的对象约束语言OCL（Object Constraint Language）来协助创建转换规则。转换规则将源模型中的模型元素映射到目标模型中的模型元素，OCL query 描述了源模型中的模型元素，OCL expression 则描述了目标模型中的模型元素。许多转换规则必须在一定的条件下才能执行，这些条件可以使用OCL 来描述。转换规则中使用的所有OCL expressions 被应用在源和目标建模语言的元模型上。

另外，本阶段还涉及PIM构件模型的精炼。

要清楚的是，基本PIM构件模型必然出自于模型库，应用工程师首先查询模型库查找所需模型，尽量使用已有的构件模型；如没有相匹配的构件模型，经过业务分析员和需求分析员创建后而需存入构件模型库，应用工程师仍从模型库中取得。这个基本PIM可由应用工程师和业务分析员共同来完成，以保证模型的完整与正确性。

本阶段实现了构件组装的高层次抽象。

（3）PIM构件模型到PSM构件模型的映射以及PSM的精炼。

PIM建立以后就存入模型库，作为模型映射的输入。为了产生PSM，必须选择一个目标平台，然后通过MDA规定的映射规则把PIM所包含的信息映射为目标平台所需要的形式。在完成了PIM到PSM的映射以后，就可以根据具体的平台选择相应的编程语言了。比如，如果选择.net架构，那支持.net的MDA映射工具将根据选择的编程语言自动产生源代码，以及配置信息、WSDL、UDDI、XML DTD等内容；如果目标平台为CORBA，那么支持CORBA的MDA映射工具将产生CIDL、PSDL等内容。

如果有需要或发现映射中存在误差，可以对PSM构件模型的进行修正和求精。

本阶段实现了构件组装的较高层次抽象。

（4）PSM构件模型到构件代码。

使用PSM模型映射工具之间生成构件代码和相应文档，并把相关产品存入构件，这一步比较简单，目前多数UML建模工具都能实现这个功能。

（5）构件的组装和部署。

即实现级的组装和部署，主要通过构件组装和部署工具来完成。构件组装涉及构件选取、构件的自适应、构件的属性配置等几个方面。需要强调的是，构件组装构件的选择可以按照映射工具生成的文档信息来发现合适的构件，这一过程也可以由组装工具来自动完成，这样，大大增加了构件选取的方便性，使构件组装更加容易。

（6）逆向工程。

它从基于具体中间件平台的实现抽象出平台无关 PIM 构件模型，这个过程与“数据挖掘”的概念类似，很难完全自动实现。

最后要说明的是，模型库、文档库、构件库在是整个过程的基础，在建立一个新的模型的时候，必须查阅文档库的信息，以确定哪些需要新建一个模型，哪些可以直接复用现有的模型。对构件的复用也有同样的要求。灵活有效的工具为此提供了可能。

4.9 比较分析

MDAC 的构件开发途径看起来和传统的开发方法很类似，但是这里有一个至关重要的不同，就是模型在MDA 中是可执行的，是能够产生产出的。为什么说MDA 中的模型是可执行的？因为MDA 中的模型的级别处于精确模型级，精确模型级的模型具有足够的精确度，可与实际代码直接连接（语义距离更小），

但模型仍保持高级别的抽象性，而不是编程语言概念的直接表示。该级别的特征是[37]：（1）编码者不再作出任何商业决定。（2） 模型与代码的互更新功能成为关注焦点（相对于工具开发商而言），且更容易使用。（3） 模型到代码的直接转换（自动化和可定制）方便了迭代和增量的开发。从这里我们可以看出，事实上建模语言在MDA 中已经成为了一种编程语言而不仅仅是设计语言，模型所发挥的作用更大了，能够从模型中生成代码，也使得我们对建模的投资获得了更大的回报。

同时，传统意义上，从模型到模型，从模型到代码的转换主要是手工完成的。虽然有些工具可以从模型中产生部分代码，但是它们一般都是通过指定的模板来生成，有很大的局限性，大多数的工作仍然需要手工来完成。与之形成鲜明对照的是 MDA 中的转换通常都是通过工具来完成的。许多工具能够将 PSM 转换成代码。事实上，PSM 已经和代码非常接近，这种转换其实没什么稀奇的。对于 MDAC 来说，其重要的改变就是它将从 PIM 到 PSM 的转换也自动化了，它是 MDA 给开发人员带来的最大的惊喜，想想以前，从详细设计中创建数据库模型要花费开发人员多少工作？从相同的设计中创建 COM 或者 EJB 构件要花开发人员多少时间？现在 MDAC 使这一过程完全自动化了。

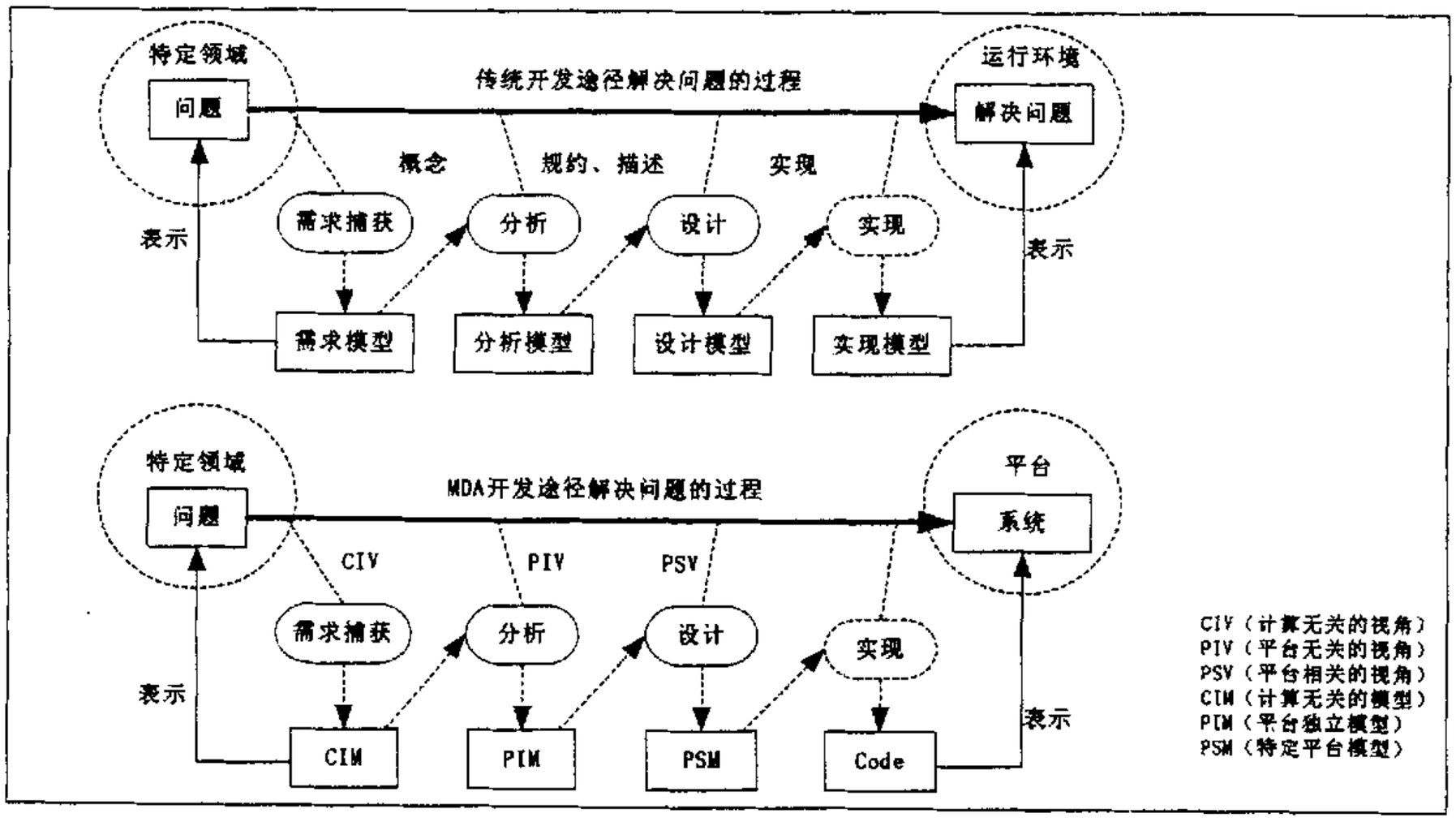

图 MDAC 方法与传统方法的比较

简而言之，MDAC 与传统构件开发方法的最大区别就在于它实现了精确构件模型的自动转换。另外，论文在表中就开发过程、构件建模、软件体系结构、工作流、通用性、易用性、成熟度、支持度等方面，把 MDAC 方法同第二章提到的一些主要研究工作进行了比较。

系统支持 CBSD 的程度：COM/DCOM 只定义了实现模型，系统的全面的 CBCD 支持较差；EJB 和 CCM 规范定义了比较详细的开发过程和开发人员的角

色，而 AOP 方法注重关注点的分离，所以它们支持一般；CoSMIC、COMBINE、Catalysis、ABC、MDAC 方法则都对 CBSD 提供了系统的、全面的支持。

构件建模支持：COM/DCOM、EJB、CCM、AOP 均未定义构件建模框架；CoSMIC、COMBINE、ABC 的对构件建模技术只作了一般性的描述；Catalysis、MDAC 方法定义了详细的构件建模框架，所以支持度最好。

软件体系结构（SA）支持：COM/DCOM、EJB、CCM、CoSMIC、COMBINE、AOP 都不支持；Catalysis、ABC、MDAC 支持一般。基本原因在于 SA 的研究目前主要在学术界，离商业应用还有一定的距离。

工作流支持：这些构件模型、方法中，只有 COMBINE 对工作流提供了较好的定义。

通用性：：COM/DCOM、EJB、CCM、CoSMIC 这几种方法都局限于特点的构件模型；COMBINE、AOP、Catalysis、ABC、MDAC 则是系统开发的指导性方法，通用性较强。

易用性：COM/DCOM、EJB、CCM、MDAC 都是以一些主流技术为基础，易于接受和使用；而 CoSMIC、AOP、ABC 方法都提供各自相应的工具支持，程度一般；COMBINE、Catalysis 这两种方法最为负责，比较难以理解和使用。

成熟度：毫无疑问，COM/DCOM、EJB、CCM 都已经是较为成熟的构件标准；而其它几种方法都基于各自的出发点，任处于探索阶段。

支持度：其中 EJB 的最为流行，而 MDA 是 OMG 的未来战略目标，所以 MDAC 方法的思想必将得到广泛支持；COM/DCOM 则是微软的标准，AOP 正获得一定范围的关注，支持度一般；其它几种方法则相对较差。

方法 / 标准	COM/ DCOM	EJB/ CCM	CosMIC	COMBINE	AoP	Catalysis	ABC	MDAC
CBSD	差	一般	好	好	一般	好	好	好
构件建模	／	／	一般	一般	／	好	一般	好
SA	／	／	／	／	／	一般	一般	一般
工作流	／	／	／	好	／	／	／	／
通用性	差	差	差	好	好	好	好	好
易用性	好	好	一般	差	一般	差	一般	好
支持度	一般	好	差	差	一般	差	差	好
成熟度	好	好	好	一般	一般	一般	一般	一般

表 4.2　　MDAC 与相关构件开发方法性能对比表

通过上述比较分析，可见，总的来说，MDAC 方法具有较大优势，但对软件体系结构和工作流的支持需要进一步加强。

4.10 小结

基于模型驱动体系结构的思想，论文在本章中给出了一个完整的基于 MDA 的构件开发框架 MDAC，详细说明了该构件开发框架的四层体系结构，同时系统分析了框架中模型以及开发角色的定义。MDAC 方法通过对 UML 进行扩展，给出了一个完整的建模框架实现了构件的建模，并从软件工程的角度，系统地给出了构件开发的整个生命周期，实现了化构件开发为模型开发，化构件组装为模型组装，化构件实现为模型映射，化构件复用为高层设计的复用、模型的复用。

第五章 MDAC方法应用示例

本章将利用MDAC方法来开发一个实例系统，着重分析了如何利用CMF构件建模框架定义的一系列规则来设计构件模型，以及根据转换规则进行构件模型转换的实现过程。

5.1 示例简介

电子商务是通过电子信息技术、网络互联技术和现代通讯技术,使得交易涉及的各方当事人借助电子方式联系,而无需依靠纸面文件、单据的传输,从而实现整个交易过程的电子化。电子商务产生后,立刻形成了一个以信息技术服务为支撑的全球活动的动态发展过程,由此引发了一场以现代信息技术,特别是网络互联技术作为推动的、跨越时空界限的商业领域的革命。

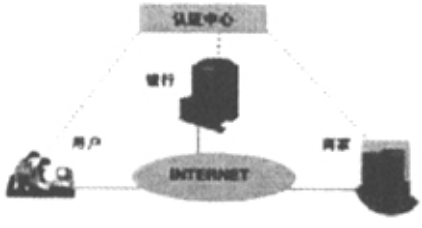

图5.1 电子商务基本场景图

电子商务最基本的应用之一就是进行网络销售，其基本过程为：商家通过在网络上发布商品相关的详细信息及图片，客户则通过网上查询商品信息和报价后，如果发现满意的商品，就可以直接在网上购买，商家在接收到客户订货信息后对配货部门下达送货指示，配货部门就可按照客户信息把货物发送给客户，最后客户付款结束整个网络销售服务过程。

上述过程描述起来十分简单，却很具代表性，因此论文将使用基本的电子商务场景来解释MDAC方法的构件开发过程。当然真正的电子商务应用还必须包括很多相应的服务，如客户身份认证、网上银行结算、商家信用认证等，由于这些因素与本文的主题无关，所以在例子中就不作讨论。

5.2 业务建模

整个应用场景由买方（Buyer）、卖方（Seller）、送货方（Logistics）三方一起协作完成。基本的业务流程是，买方向卖方提出订货请求，卖方响应请求并传送命令到送货方请求托运服务，而后送货方运送货物到客户。根据CMF建模

框架，论文在图 5.1 中给出了整个应用的业务模型。在图 5.1 中，Buyer、Seller、Logistics 三个构件功能相互独立，通过端口和协议相互协作，Buyer 通过 Buy 协议端口和 Seller 的 Sell 协议端口直接交互，而 Seller 则通过 Logistics 的 Ship、Delivery 协议端口与 Buyer 通信。

整个交互过程中的活动由 Sales 协议进行规范，其模型如图 5.2，首先是 Buyer 根据 Sales 协议通过 Buy 端口发起活动，然后 Seller 根据 Sales 协议通过 Sell 端口连接到 Buyer 的 Buy 端口，并且根据 Sales 协议发出相应，初始化 Logistics 的 Ship 端口，Logistics 随之相应激活 Buyer 上的 Delivery 端口，同样，Buyer 再根据 Sales 协议来终止整个协作过程。

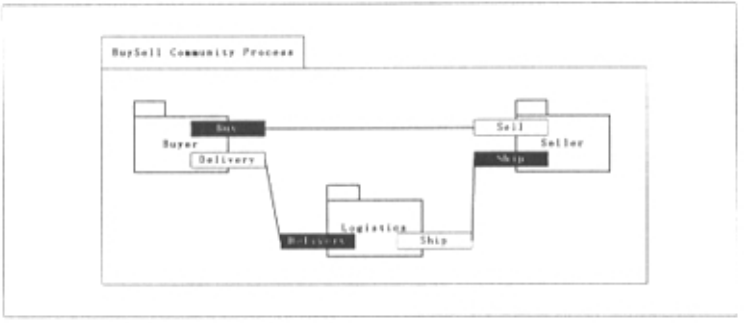

图 5.1 电子商务基本业务模型

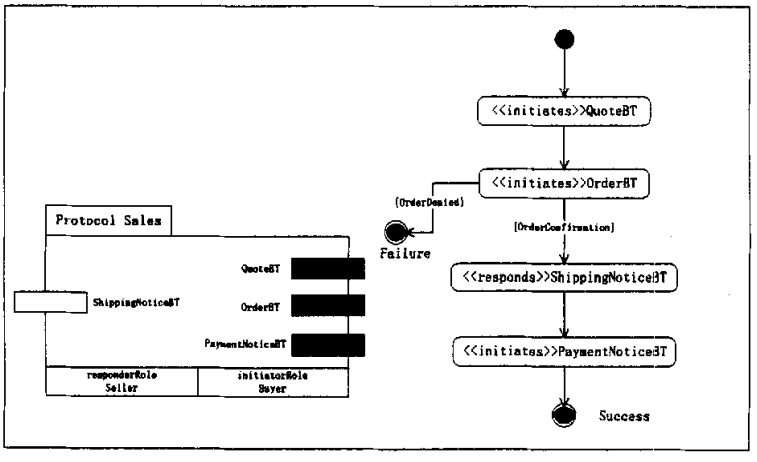

图 5.2 Sales 协议模型

5.3 协议

5.3.1 Sales 协议

Sales 协议（Sales_Protocol）由四个简单的子协议组成：QuoteBT、OrderBT、

ShippingNoticeBT 和 PaymentNoticeBT。Sales_Protocol 使用协议端口使用这些简单协议。图 5.3 给出了 Sales 协议模型的详细结构，关于四个子协议的编排图将在接着的几节中给出。由图可知，Sales_Protocol 的信息交互由其 initiatorRole 触发，首先完全执行报价子协议 QuoteBT，而后 initiatorRole 继续激活并完全执行订货子协议 OrderBT，如果在执行 OrderBT 协议期间接收到一个撤销信息 OrderDenied，那么整个 Sales 协议将终止，状态为 Failure，如果接收到一个订货确定信息 OrderConfirmation，那么 responderRole 将响应信息，完全执行托运子协议 ShippingNoticeBT，最后由 initiatorRole 激活支付协议 PaymentNoticeBT。

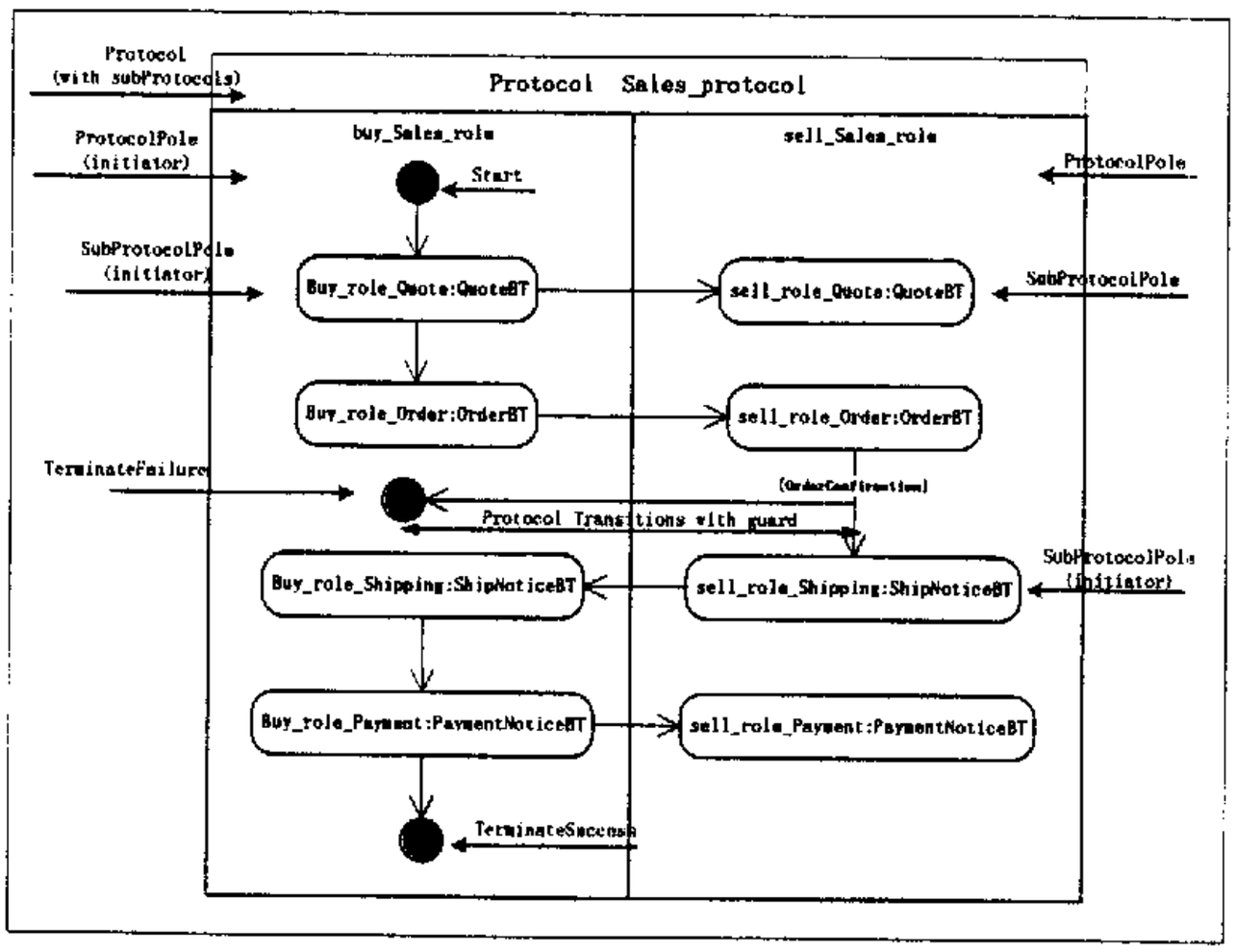

图 5.3　Sales 协议及编排

5.3.2 QuoteBT 协议

报价协议（QuoteBT Protocol）是一种请求/应答（Request/Reply）模式的协议，在 QuoteBT 协议端口，initiatorRole 将发送 QuoteRequest 请求，接收 Quote 应答。QuoteRequest 和 Quote 都是 QuoteBT 协议的流端口（FlowPort）。

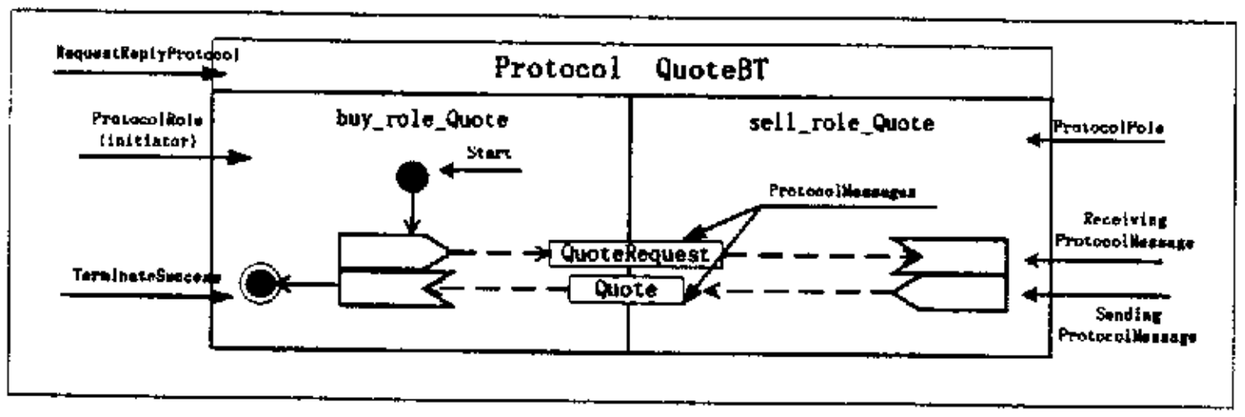

图 5.4　QuoteBT 协议及编排

5.3.3 OrderBT 协议

定购协议（OrderBT Protocol）是一种具有特定转换条件的协议形式，在 OrderBT 协议端口，initiatorRole 将发送 Order 请求，接收 OrderConfirmation 或 OrderDenied 两种类型的响应。Order、OrderConfirmation、OrderDenied 都是 OrderBT 协议的流端口（FlowPort）。

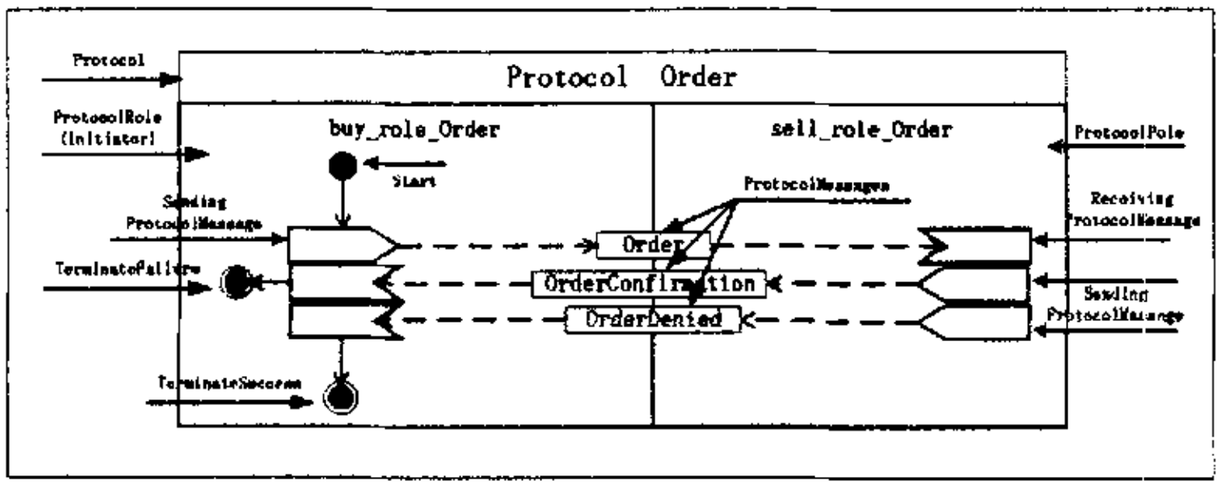

图 5.5　OrderBT 协议及编排

5.3.4 ShippingNoticeBT 协议

托运通知协议协议（ShippingNoticeBT Protocol）是一个具有单独流端口的协议，它负责发送 ShippingNotice 消息。其实，定义一个仅带有单一信息流的协议是完全没有必要的，因为这个唯一地流端口可以包括到任何一个与之相连的协议中去。在例子中定义了 ShippingNoticeBT 协议是为了更清晰地划分业务模块，说明建模方法。

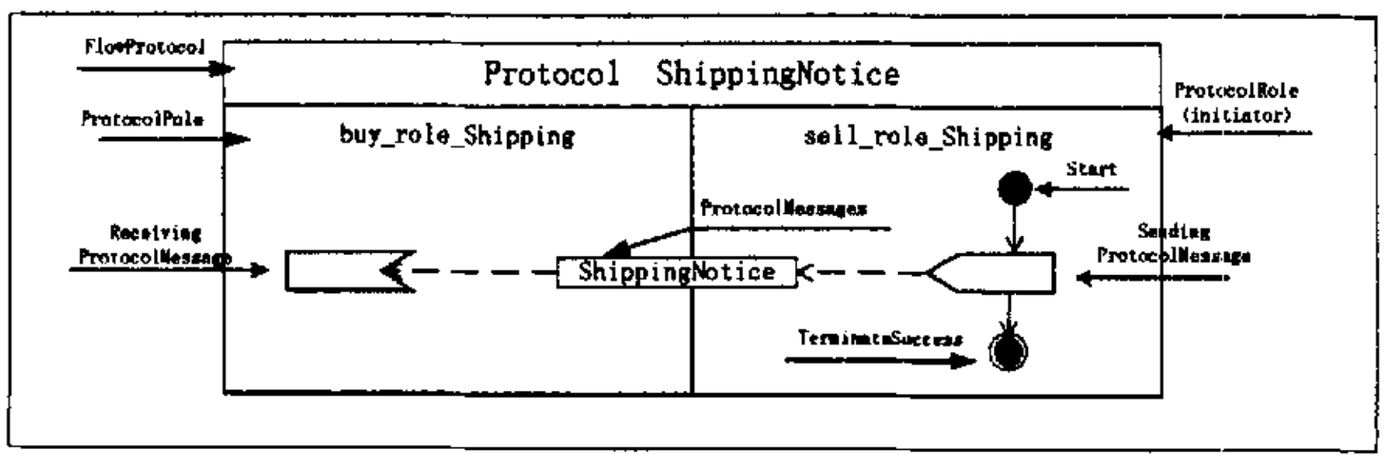

图 5.6　ShippingNoticeBT 协议及编排

5.3.5 PaymentNoticeBT 协议

付款通知协议（PaymentNoticeBT Protocol）和 ShippingNoticeBT 协议一样，也是一个单流端口协议，它负责发送 PaymentNotice 消息。

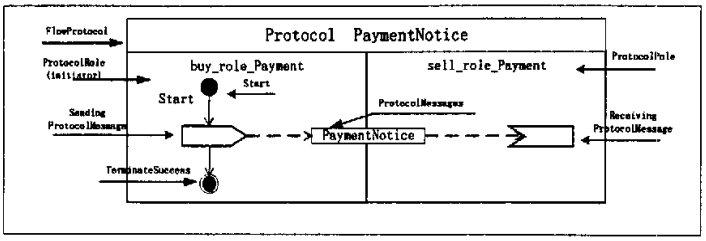

图 5.7　PaymentNoticeBT 协议及编排

5.3.6 ShipBT 协议

托运协议（ShipBT Protocol）是一种请求/应答（Request/Reply）模式的协议，在 ShipBT 协议端口，initiatorRole 将发送 ShippingRequest 请求，接收 PickupReceipt 应答。ShippingRequest 和 PickupReceipt 都是 ShipBT 协议的流端口（FlowPort）。

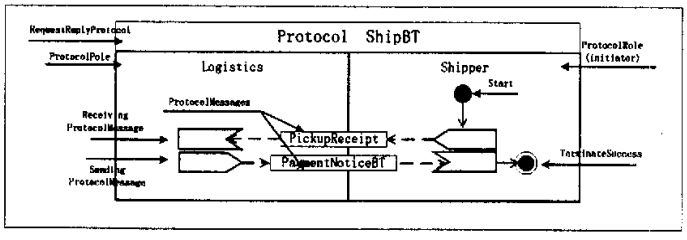

图 5.8　ShipBT 协议及编排

5.4 PIM 构件模型（CMF）

5.4.1 Buyer 构件

Buyer 构件有 Buy 和 Delivery 两个协议端口，Buy 端口根据 Sales 协议负责启动交互过程，Delivery 端口则负责响应。Buyer 构件是整个电子商务应用的起始点，在通过 Buy 端口发出请求后，如果中途发生 OrderDenied，那么编排过程到达将终止状态 Failure，如果接收到确定信息 OrderConfirmation，那么 Buyer 构件就会通过 Delivery 端口进行相应，成功完成交互活动。

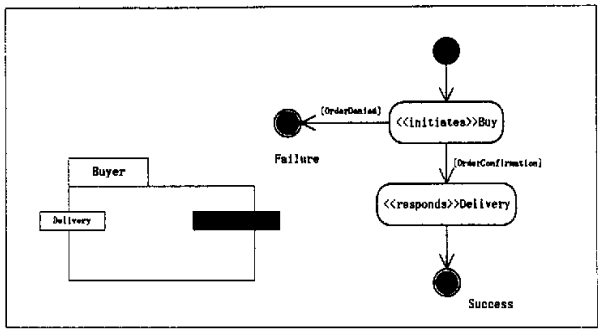

图 5.9　Buyer 构件及编排

5.4.2 Seller 构件

Seller 构件有 Sales 和 Ship 两个协议端口，Sales 端口是一个端口集，负责和 Buyer 构件的交互，包括 Quote、Order、ShippingNotice、PaymentNotice 四个子端口；Ship 端口则用于与 Logistics 构件进行交互。Seller 构件活动由 Quote 端口对 Buyer 构件的响应而激活，Quote 端口接收 Buyer 构件发出的查询商品报价信息，返回买方所选的报价信息，接着 Order 端口对 Buyer 构件进行相应，如果 OrderDenied，则同样到达状态 Failure，如果 OrderConfirmation 则激活 Ship 端口，并向 Logistics 构件发送 ShippingNotice，最后对 PaymentNotice 进行响应，到达 Success 状态。

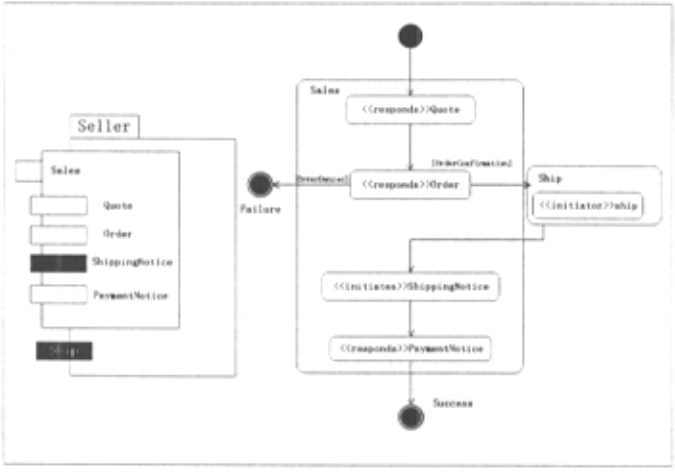

图 5.10　Seller 构件外部结构和编排

5.4.3 Seller 构件的内部结构

在图 5.10 给出了 Seller 构件的外部结构和协议编排过程，但是并没有展示 Seller 构件的内部细节。当设计一个系统，要实现 Seller 角色，Seller 构件必须要进一步规约，被分解为复杂性度更小的单元，直到这些单元能够直接映射，即必须 Seller 构件是一个复合构件，设计时还必须定义构成它的原子构件。

对 Seller 构件的内部分解必须以外部可见的编排为基准。只有符合外部端口结构，Seller 构件才能在 Sales 业务流程中很好的执行，同时，对于 Seller 构件的使用者而言，Seller 构件的内部定义是完全独立的，体现了信息隐蔽的特点。在本例中，Seller 构件的内部实现可以被分解为四个原子构件：QuoteCalculator、Seller_Orders，Warehouse 和 AccountsReceivable，如图 5.11 所示。

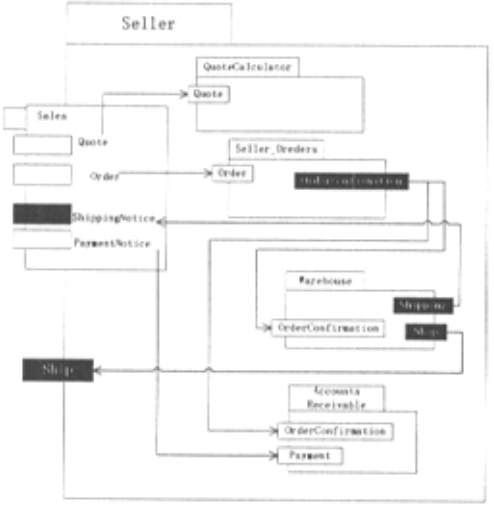

图 5.11　Seller 构件的内部结构

5.4.4 QuoteCalculator 构件

QuoteCalcultor 构件定义比较简单，如图 5.11，它定义了一个单独的协议端口 Quote，是 Seller 构件的子端口 Quote 的直接代理，所以 QuoteCalcultor 构件的主要功能就是对 Seller 构件的 Quote 子端口的消息进行响应和处理。

5.4.5 Seller_Orders 构件

Seller_Orders 构件负责响应和处理 Seller 构件的 Order 子端口的消息，Seller_Orders 构件定义了 OrderConfirmation 流端口连接到 Warehouse 和

AccountsReceivable 构件。当 Seller_Orders 对一个 OrderConfirmation 信息进行响应时，相同的 OrderConfirmation 信息将同时发送到 Warehouse 和 AccountsReceivable 构件。

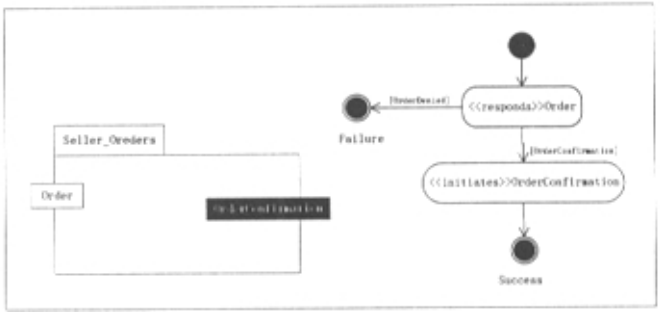

图 5.12 Seller_Orders 构件及编排

5.4.6 Warehouse 构件

Warehouse 构件负责响应从 Seller_Orders 构件端口发出的 OrderConfirmation 信息，并通过 Ship 协议端口发起与 ShipBT 协议的交互活动，随后，通过 Seller 构件的 ShippingNotice 子协议端口，Warehouse 构件进一步向前激活 ShippingNoticeBT 协议定义的交互活动。

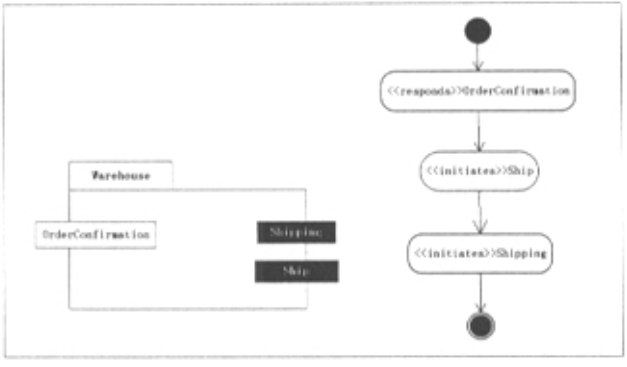

图 5.13 Warehouse 构件及编排

5.4.7 AccountsReceivable 构件

与 Warehouse 构件类似，AccountsReceivable 构件也接收从 Seller_Orders 构件发来的 OrderConfirmation 信息，然后响应和处理 Seller 构件的 PaymentNotice 子端口。

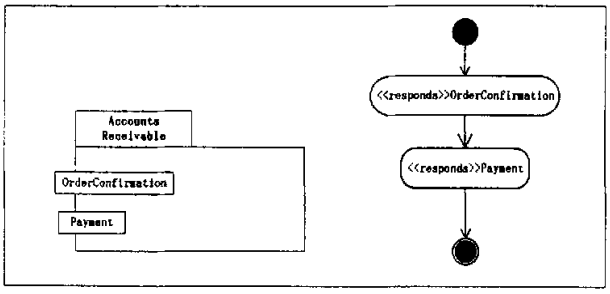

图 5.14　AccountsReceivable 构件及编排

5.4.8 Logistics 构件

如图 5.15 所示，Logistics 构件定义了 Ship 和 Delivery 两个协议端口，Ship 端口主要负责响应 ShipBT 协议定义的活动，而 Delivery 端口则负责激活 DeliveryBT 协议定义的交互活动。由图可见，Logistics 构件在一个序列里集成了 ShipBT 和 DeliveryBT 两个协议，只有在 ShipBT 协议完全执行和完成以后，DeliveryBT 协议才能开始。

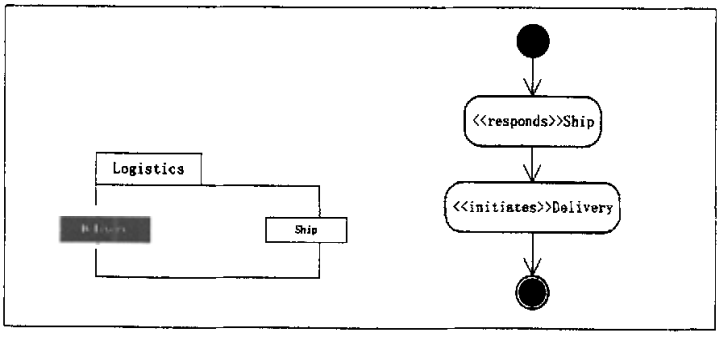

图 5.15　Logistics 构件及编排

5.5 PIM 模型（UML）

基于 CMF 构件建模框架和构件思想，论文在前面几节描述了电子商务应用中在高层抽象层上如何互操作，以及它们如何在抽象层被进一步分解，同时，通过分析构件端口传送信息的协议（Protocol）和编排（Choreography），实现了对构件接口的定义。所以，前面建模的构件模型已经是标准的 CMF 构件模型，下一步就是实现 CMF 构件模型到标准 UML 模型的转换。根据第四章定义的模型转换规则和 UML 扩展机制（Constraint、Tagged Value、Stereotype），很容易就能实现 CMF 构件模型到标准 UML 模型的转换。由于转换过程基本类似，下面论文就以协议、构件、构件组装等关键元素为例来讨论转换是如何实现，注

意，在实际开发过程中，下面描述的转换细节均由模型转换工具来实现，对开发人员并不可见，具体的映射规则参见 4.5 节，本章就不再重复给出，只给出映射后的 UML 图。

5.5.1 场景模型（UML）

图 5.16 给出了按 CMF 映射规则所得到的 UML 场景图，它是对应用的高层抽象，利用标准 UML 元模型对场景角色（Buyer、Seller、Logistics）的协作关系进行了描述。其中，Buyer、Seller、Logistics 对应于 CMF 元模型中的 ComponentUsage，Buy、Sell、Ship、Delivery 对应于 CMF 元模型中的 PortConnectors，它们之间的连线则对应于 CMF 元模型中的 Connection。

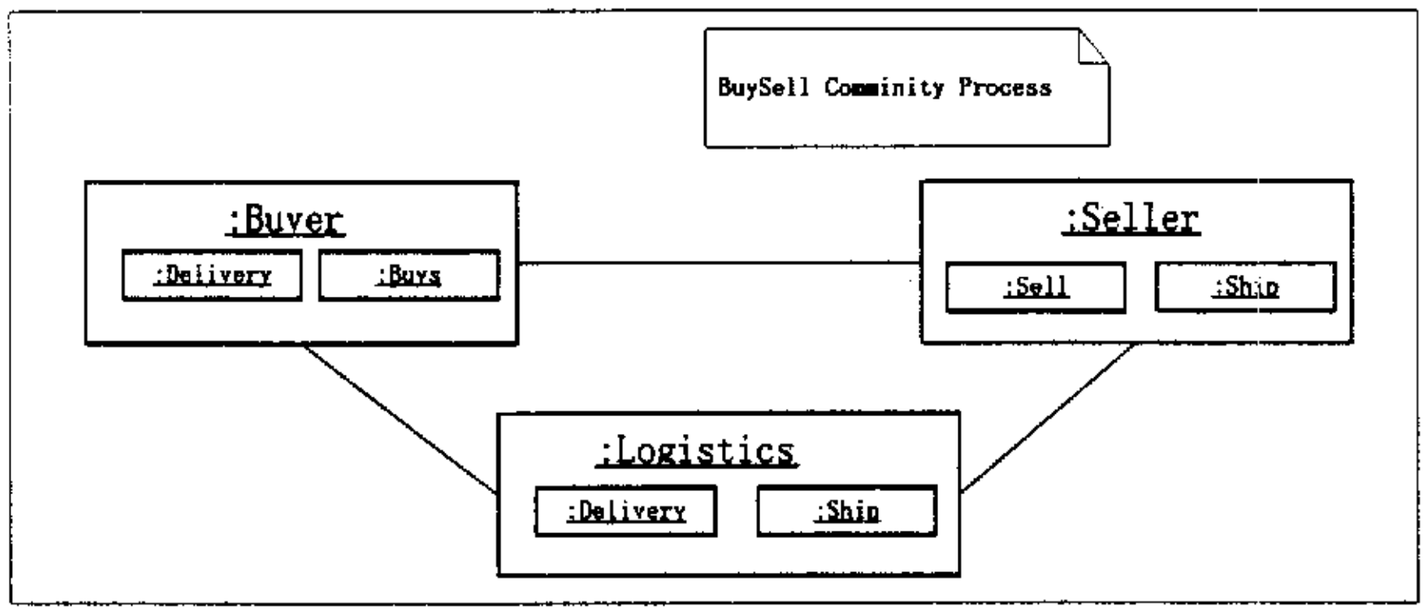

图 5.16　UML 场景模型

5.5.2 协议（UML）

图 5.17、图 5.18、图 5.19、图 5.20 分别给出了按 CMF 映射规则所得到的 Sales 协议的各个子协议（QuoteBT、OrderBT、ShippingNoticeBT、PaymentNoticeBT）的 UML 图，图 5.21 则给出了上述各协议编排过程的映射图，图 5.22 和图 5.23 则分别给出了 CMF Sales 协议转换后的 UML 图及相应的编排图。

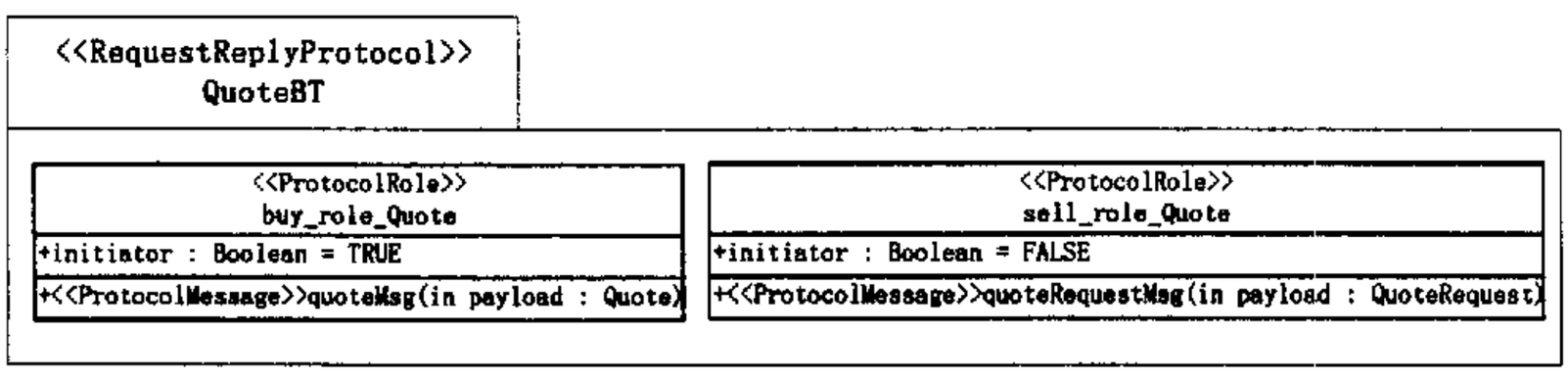

图 5.17　QuoteBT 协议 UML 图

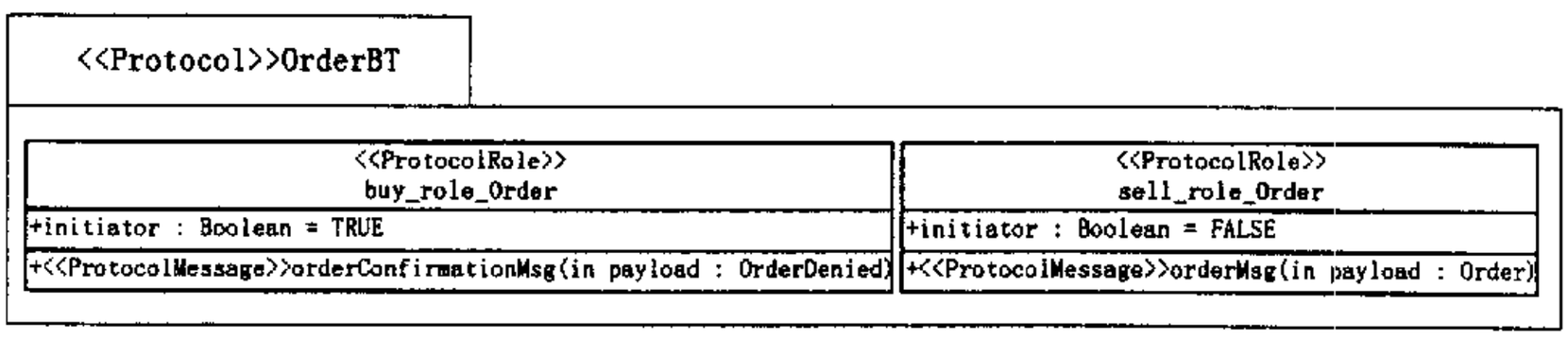

图 5.18　OrderBT 协议 UML 图

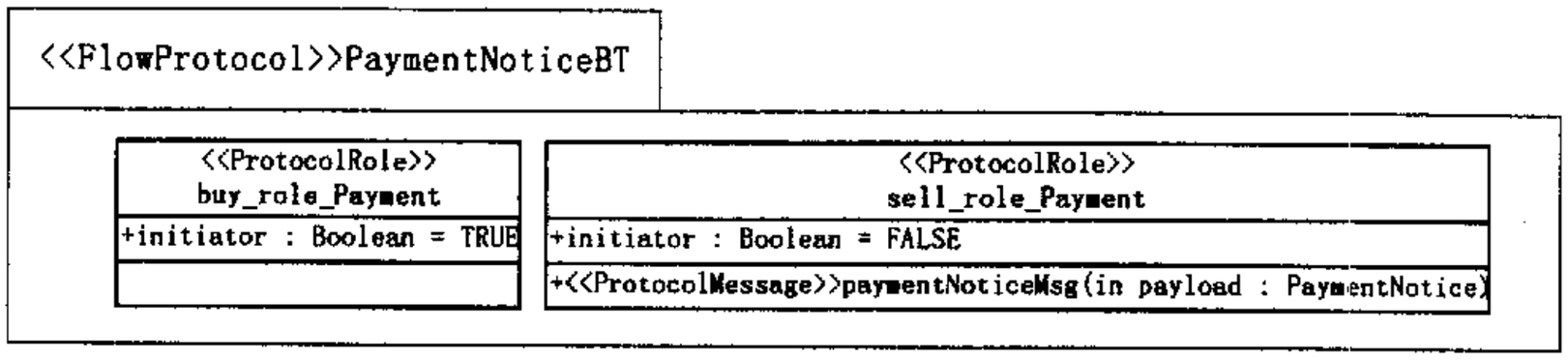

图 5.19　PaymentNoticeBT 协议 UML 图

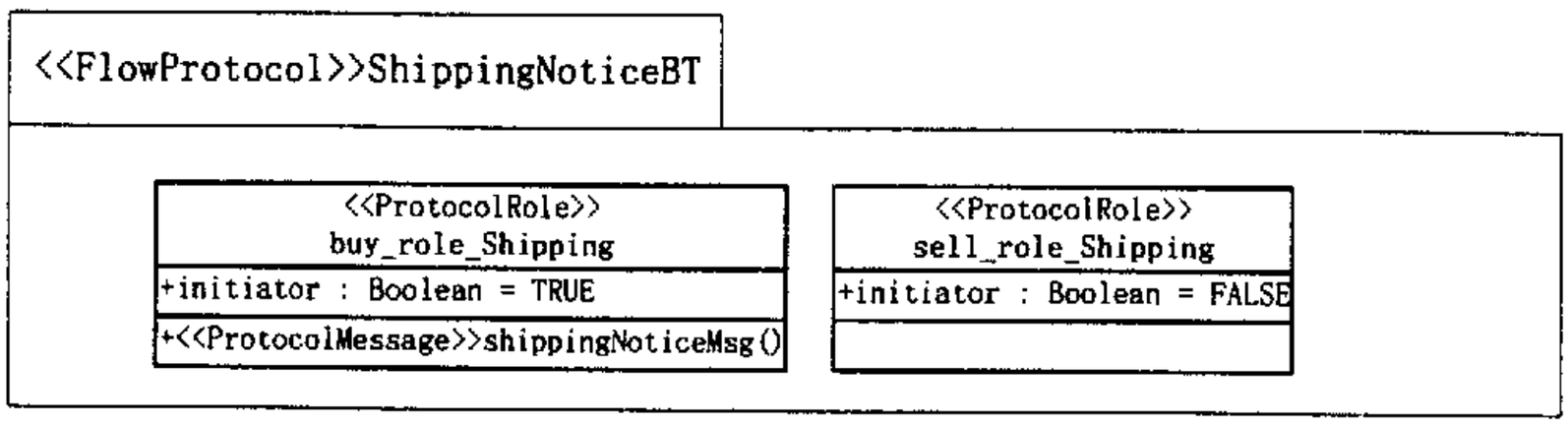

图 5.20　ShippingNoticeBT 协议 UML 图

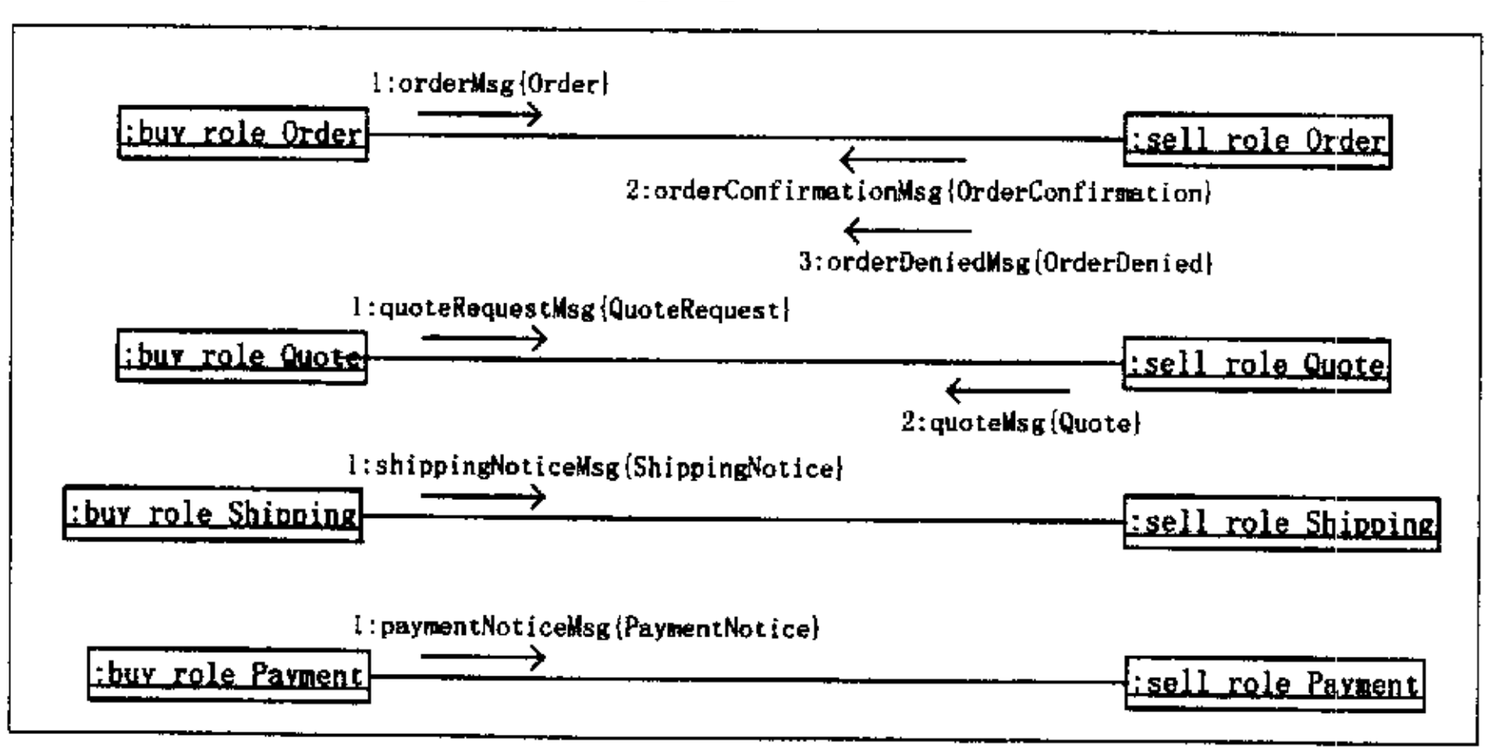

图 5.21　Sales 各子协议的 UML 编排图

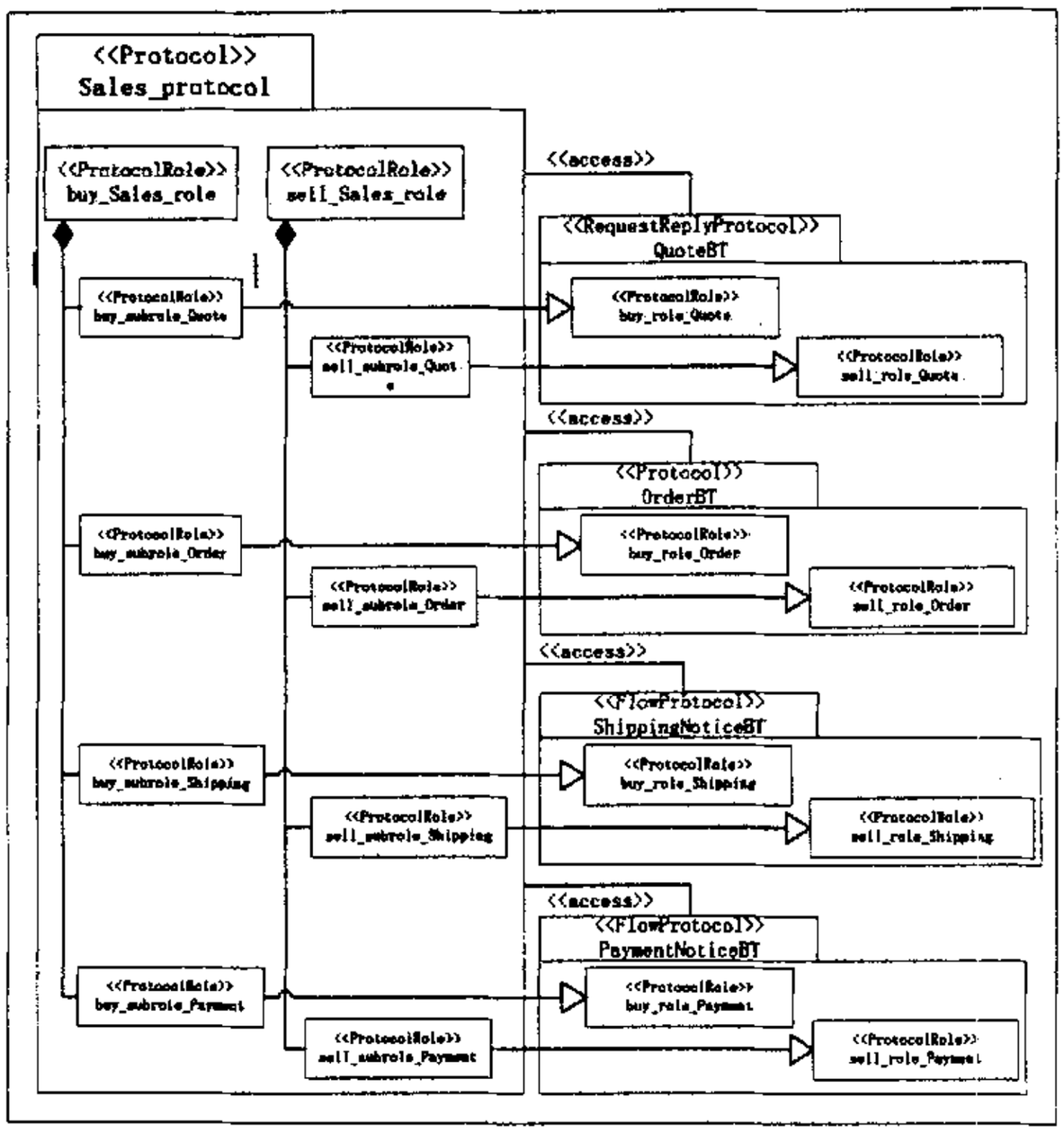

图 5.22　Sales 协议 UML 图

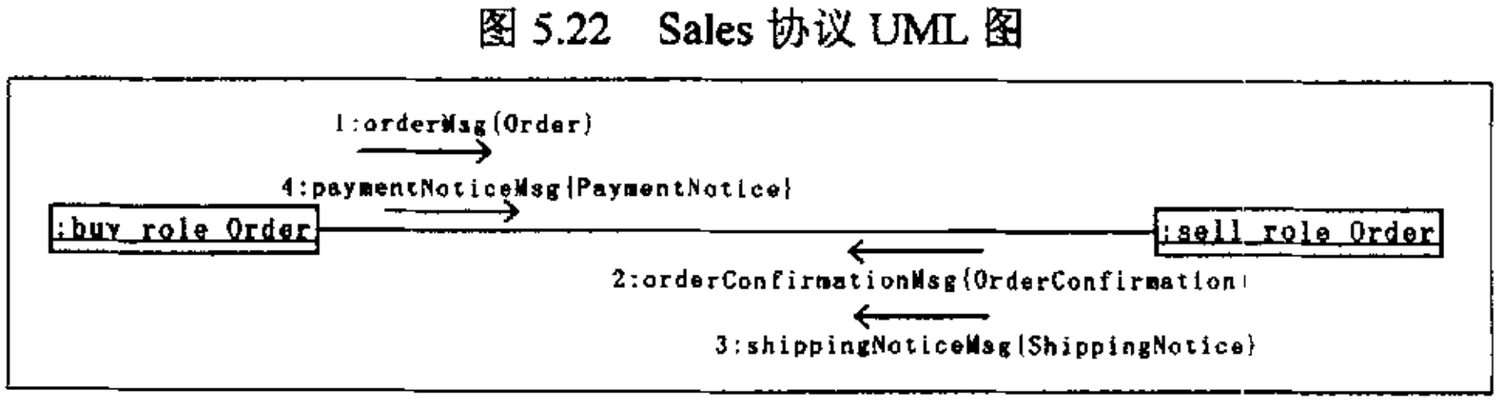

图 5.23　Sales 协议编排 UML 图

5.5.3 构件规约（UML）

CMF构件模型的映射相对简单，根据映射规则，图5.24给出了Buyer构件和Seller构件的部分映射图，图5.25则给出了QuoteCalcultor、Seller_Orders、Warehouse、AccountsReceivable四个构件的UML图。

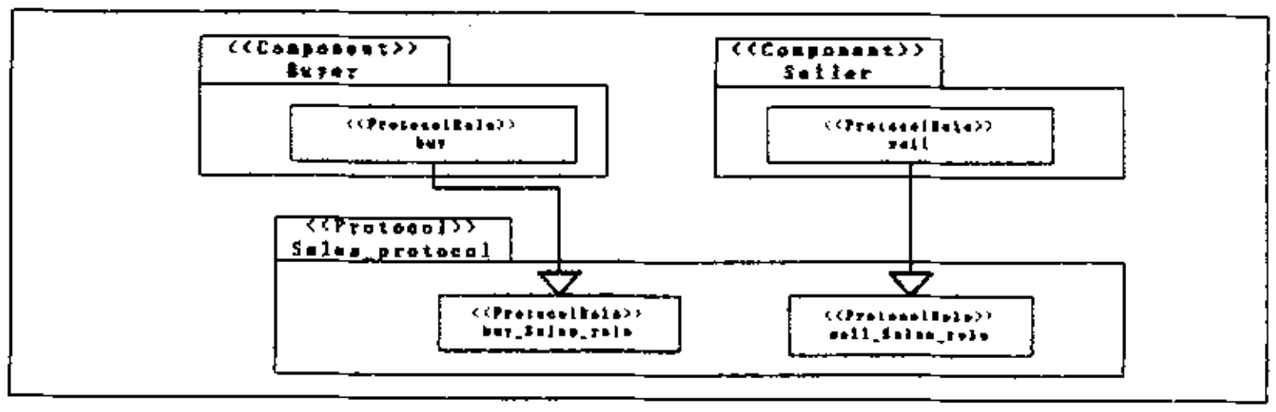

图 5.2 Buyer、Seller 构件 UML 图

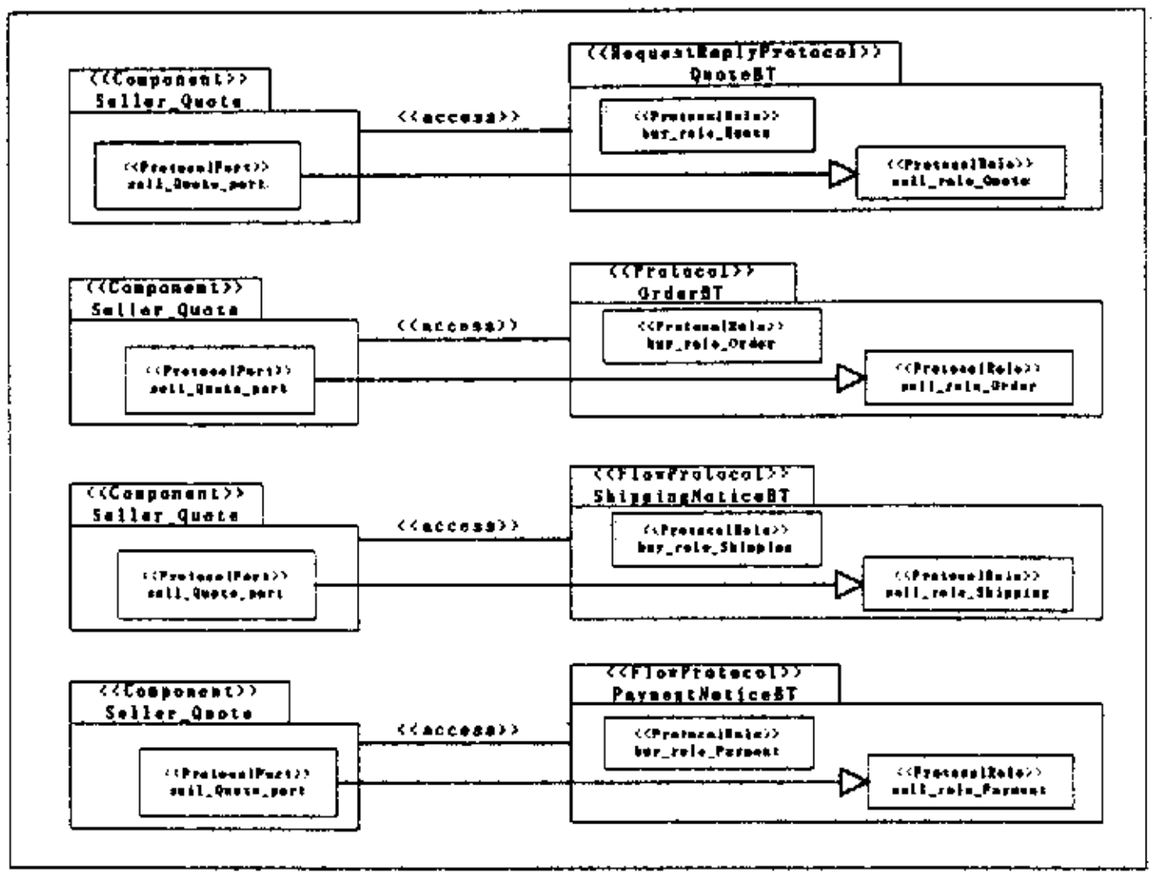

图 5.25　Seller 子构件 UML 图

5.5.4 构件组装（UML）

图 5.26 给出了 Seller 构件内部各子构件之间的组装关系图，如图所示，在实际的组装过程中，Component 以 ComponentUsage 的形式存在，Port 则以 PortUsage 的形式存在，Seller 构件端口和协议之间存在连接 Connection，图 5.27 则是相应的编排关系的映射。

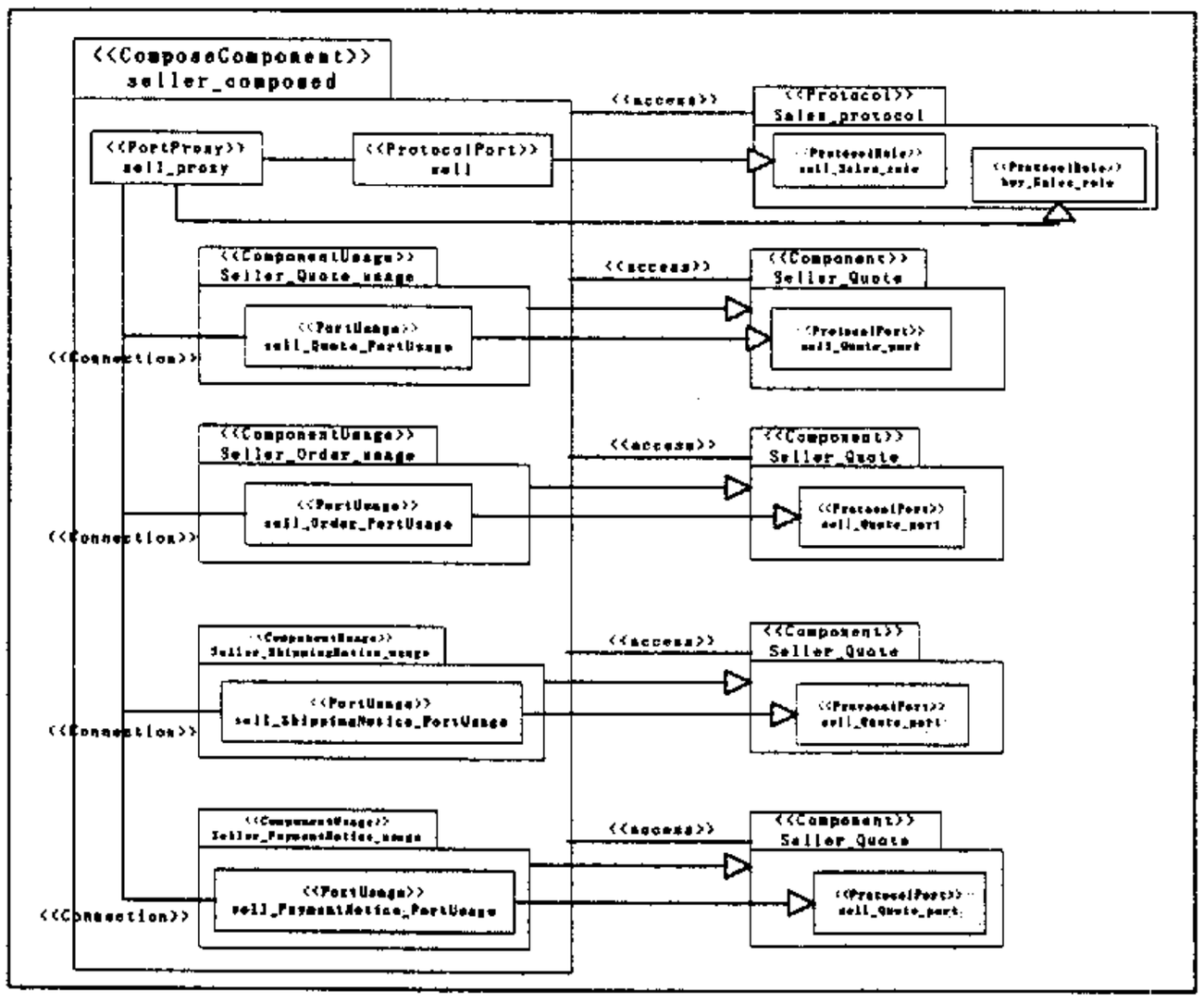

图 5.26　构件组装 UML 图

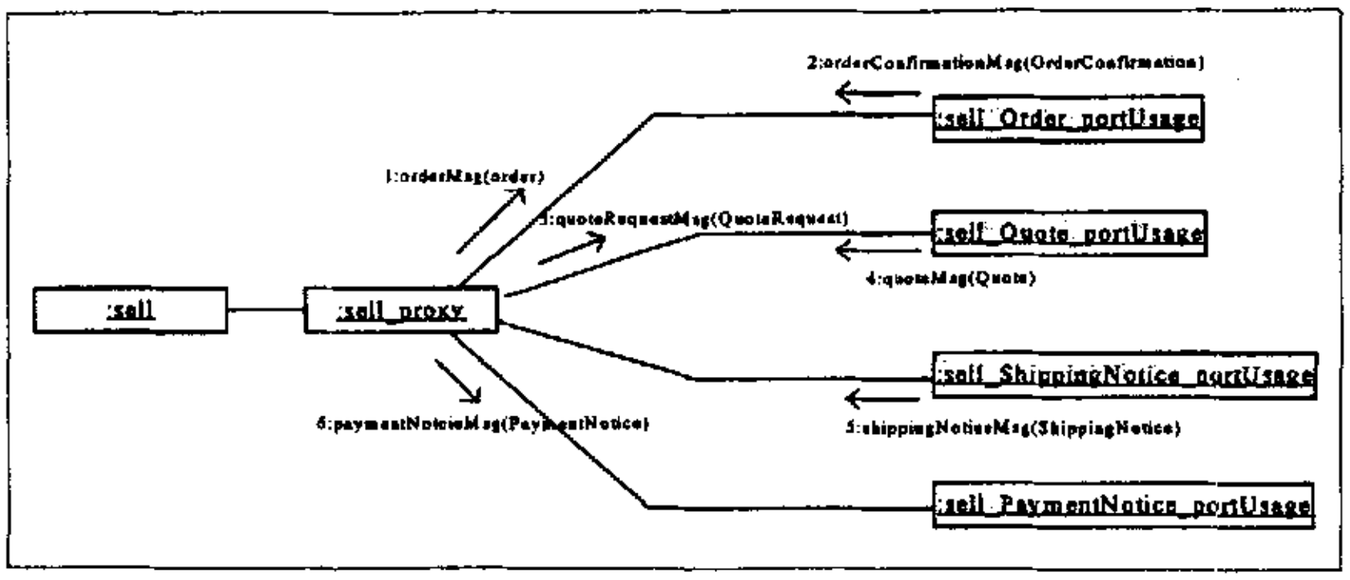

图 5.27 构件组装协作图

5.6 PSM 模型

在获得了标准的 UML PIM 模型之后，由于标准的 UML 模型已经能够完成从 PIM 到 PSM，PSM 到实现的映射，例如，Rose、UMT 等工具均支持 UML Profile for EJB、CORBA，所以本章在这里就不列出其转换结果了。

5.7 小结

论文在本章中利用一个电子商务实例，该实例详细显示了 MDAC 构件开发框架下利用构件建模技术实现构件的开发及构件转换。该实例在建立基本业务模型的基础上，利用上一章所提出的构件建模框架（CMF）建立协议模型和构件模型，并在此基础上根据模型转换规则建立标准 UML 构件模型。

第六章 总结和展望

论文研究的主要内容是，基于MDA提出了一种系统的构件开发方法MDAC，该方法强调概念设计和模型的复用，把模型驱动和构件的思想融合到软件开发生命周期的各个阶段，利用模型来开发构件，化构件开发为模型开发，化构件组装为模型组装，化构件实现为模型映射，化构件复用为高层设计的复用、模型的复用，为解决构件技术面临的困境提供了一条较完备的、有效的、切实可行的途径。

但是，由于MDAC方法主要基于MDA架构，在集成了MDA的所有优点的同时，也同时继承了MDA的不足。虽然MDA是OMG树立的新的一面大旗，但是，作为一种全新的思想，MDA仍然面临着许多挑战，比如对模型的表示和处理功能的完善、相关辅助标准的的制定、提供更多的动态功能等。这些也是MDAC构件开发方法要进一步解决的问题，总结为如下几点：

（1）工具实现。无容置疑，模型转换工具是MDA得以实现的核心。整个基于MDA的开发需要高度自动化的工具支持，目前，已经有许多软件供应商（IBM、Borland等）表示了对MDA的支持，在自由软件组织，MDA也十分活跃，例如Eclipse。但由于MDA的映射标准还在进一步补充和完善过程中，所以期待好的MDA工具的出现还会有一段时间。

（2） 模型难以测试。在基于MDA的系统中，系统的兼容性主要依赖于从PIM到PSM映射的完整性和一致性，所以基于MDA的系统必然会包括一个PSM，如果使用目前的建模语言来描述PSM的测试，模型的测试将十分困难，因为这种情况下它必然要求建立的PSM模型越详细越好，这必然降低PSM模型对具体编程人员的指导能力，往往会增加编程人员对模型细节的理解，这显然违背了MDA的模型清晰的原则。一个可能的解决方法是将UML中最基本的部分抽取出来，用于常规问题的建模，而对于面向不同中间件平台的建模，则按照平台进行划分定义形成预定义文件，这样领域应用建模人员只需要了解与自身领域相关的UML建模知识，而无需了解UML的全部细节，从而简化了模型的建立过程，增加了模型的准确性。

（3） 模型的兼容与裁剪存在问题。从PIM到PSM的转换是基于MDA开发的核心，对应于不同的中间件平台生成不同的PSM，生成的PSM模型是否能够相互兼容将会是一个很大的问题。解决这个问题需要制定相关的标准来统一模型的裁剪过程。

（4）完全依赖于UML。MDA主要依赖UML建模语言或者至少是建模范例，虽然UML目前已经成为主流的建模语言，但也不能排除出现一种新的更加流行的建模语言的可能，在那时，如果新的建模语言不能和UML集成，那它就不能

一的途径就只能是MDA和UML的不断完善和发展。

（5）元数据标准有待完善。任何基于MDA的系统必须具有存储、管理、发布应用或系统层的元数据(包括对环境的描述)。为了确保共享的元数据能够被所有的构件容易地理解，基于MDA的系统需要进一步制定如下标准：一种正式的语言(包括语法和语义)用于表示元数据、元数据的格式、元数据编程模型、扩展机制等。

（6）提供决策支持功能。企业系统功能必将进一步依赖于知识，并有能力自动地发现不同领域的公共属性，而且能基于这些发现做出智能的决定，并能够对结果进行推理。因此，有必要在MDA中引入Ontology等概念。

（7)动态体系结构的支持。模型驱动的本质在于模型的时效性和可演化性，基于动态体系结构，系统能够直接根据领域知识进行智能决策，接纳无法预料的环境变化，并且做出适当地反应而不需要程序员的干预。

（8）MDA 还面临许多其它企业应用的挑战，如动态联盟、工作流等方面的支持。

2004 年 OMG 将会对上面所描述的一些缺点进行进一步的完善和澄清。2004 年会是 MDA 大发展的一年，MDA 架构将带领软件工业朝着可互操作、可用、轻便的软件构件、基于标准的数据模型等方向迅速发展。

参考文献

[1] 杨芙清，《软件复用及其相关技术》，计算机世界 C 版，1999 年 3 月.

[2] McIlroy M D.， Mass-Produced Software Components, Software Engineering Concepts and Techniques. In: 1968 NATO Conference on Software Engineering, Van Nostrand Reinhold, 1976,pp.88-98.

[3] 6th International Workshop On Component-Oriented Programming, http://ecoop2001.inf.elte.hu/workshop/wcop-ws.html, 2001.

[4] Szyperski C., Component Software. Addison-Wesley, 1998.

[5] Felix Bachman, et al., Technical Concepts of Component-Based Software Engineering, Technical Report CMU/SEI-2000-TR-008.

[6] Guijun Wang, Liz Ungar, Dan Klawitter, Component Assembly for OO Distributed Systems, IEEE Compouter, 1999, Vol.32 (7), 71-78.

[7] Meilir Page-Jones，UML 面向对象设计基础，人民邮电出版社，2001.4.

[8] 邵维忠,梅宏，统一建模语言 UML 述评，计算机研究与发展, Vol. 3 6, No. 4 , 1999.4.

[9] Object Managemant Group, Unified Modeling Language Specification, Version 2.0, http://www.omg.org/.

[10] Carma McClure, Model-Driven Software Reuse, http://www.reusability.com/paper2.html.

[11] OMG Model-Driven Architecture Home Page: http://www.omg.org/mda/index.htm.

[12] Microsoft, http://www.microsoft.com/isapi.

[13] SUN, EJB Specification, http://www.sun.com.

[14] CORBA Specification Version 3.0, formal/02-06-33, July 2002, http://www.omg.org/cgi-bin/doc?formal/02-06-33.

[15]艾萍，博士论文，构件柔性组装描述的形式化方法研究及其在水利领域的应用，2002.12.

[16] Douglas C. Schmidt, CoSMIC: An MDA Generative Tool for Distributed Real-time and Embedded Component Middleware and Applications, www.cs.wustl.edu/~schmidt/PDF/mda_wkshp.pdf.

[17] Arne-Jorgen Berre, Support for Model-centric and Architecture-centric Development - with workflow-based composition http://www.cs.iastate.edu/~lumpe/WCL2003/Camera/BerreS.pdf.

[18] Ramnivas Laddad, I want my AOP! , http://www.javaworld.com/javaworld/ jw-01-2002/jw-0118-aspect.html.

[19] Gregor Kiczales, Aspect-Oriented Programming, www.cs.ubc.ca/~gregor/kiczales-ECOOP1997-AOP.pdf.

[20] Alan Cameron Wills, Designing Component Kits and Architectures with Catalysis, http://www.trireme.u-net.com/catalysis/ckitsArch.pdf.

[21] 梅宏,陈锋，冯耀东，杨杰， ABC:基于体系结构、面向构件的软件开发方法，软件学报，Vol.14, No.4, 2003.

[22] OMG Architecture Board MDA Drafting Team, "Model-Driven Architecture: A Technical Perspective", ftp://ftp.omg.org/pub/docs/ab/01-02-01.pdf.

[23] Object Managemant Group, MDA_Guide_Version1.0, http://www.omg.org/mda/presentations.htm.

[24] Jon Siegel and the OMG Staff Strategy Group, Developing in OMG's Model-Driven Architecture:White Paper, http://www.bitpipe.com/detail/RES/1025719188_355.html.

[25] Object Management Group, Meta Object Facility (MOF) Specification Version 1.3.New Edition, 2000, http://www.omg.org.

[26] Object Managemant Group, OMG MOF 2.0 query, views, transformations request for proposals, http://www.omg.org/techprocess/ meetings/ schedule/MOF 2.0 Query View Transf. RFP.html.

[27] Object Managemant Group, XML Metadata Interchange Specification, Version 1.2, http://www.omg.org/.

[28] Object Managemant Group, The Common Warehouse Metamodel (specifications, papers, presentations, OMG press kit, etc.), http://www.cwmforum.org/, http://www.omg.org/.

[29] Object Managemant Group, Pervasive Services Specifications, http://www.omg.org/mda/specs.htm#PervasServics.

[30] Java Metadata Interface, JSR-40 Home Page, http://java.sun.com/aboutJava/communityprocess/jsr/jsr_040_jolap.html

[31] Java OLAP Interface, JSR-69 Home Page, http://java.sun.com/aboutJava/communityprocess/jsr/jsr_069_jolap.html.

[32] Java Data Mining API, JSR-73 Home Page, http://java.sun.com/aboutJava/communityprocess/jsr/jsr_073_jolap.html.

[33] D.Souza, Model-Driven Architecture and Integration: Opportunities and Challenges, Version 1.1. http://www.catalysis.org/publications/papers/2001-mda-reqs-desmond-6.pdf.

[34] David S.Frankel 著、鲍志云译，应用 MDA，人民邮电出版社，2003.11.

[35] Jean Bézivin, From Object Composition to Model Transformation with the MDA, http://www.sciences.univ-nantes.fr/info/ lrsg/Recherche/mda/TOOLS.USA.pdf.

[36] Guy CAPLAT, Model Mapping in MDA, www.metamodel.com/wisme-2002/papers/caplat.pdf.

[37] David Flater, Impact of Model-Driven Standards, http://www.omg.org/mda/presentations.htm.

致 谢

值此学位论文完稿之际，衷心感谢我的导师——王志坚教授。在我整个研究生阶段，王老师给予了悉心关心和耐心指导，并提供了很好的学习、研究的环境和机会，我的每一点进步都与王老师的言传身教分不开。王老师求实的治学态度、缜密的思维以及广博的学识令我钦佩，并使我终生受益。

感谢计算机及信息工程学院的所有老师们，特别是艾萍老师、周晓峰老师、费玉奎博士、娄渊胜老师、尹燕敏老师、许峰老师对本文研究工作的指导。

感谢 2000 级硕士研究生：陈智强、宋键华、梁奕、毛莺池、赵瑜、刘英、程莉、阎映松等同学，感谢他们在我研究生学习期间的悉心帮助和热心指导。

感谢 2001 级硕士研究生：张雪洁、李婷婷、周惠、陈国新、蒋学锋、徐小峰、吕小燕、徐泽丰、吕行、李大科、郑晓东、任勇军、纪波林等同学。他们长久以来在项目和学习中提供了合作和帮助，文中的许多思想来源于平时与他们的交流和讨论。

感谢 2002 级和 2003 级的师弟师妹们和我们共同营造了一个活泼、团结、向上的学术氛围。

在此还要衷心感谢陈勇在三年的研究生学习生活中对我的帮助和照顾。

感谢同在一片屋檐下的郭梅、吕小燕两位室友在生活和学习上对我的帮助，一舍 414 寝室的点点滴滴永远是我珍藏的回忆。希望她们在以后的生活中充实而快乐！

感谢计算机学院的许多老师和同学在我研究生阶段给予我的关心和帮助。

最后，我要感谢我的父母。感谢他们多年来在生活上给予的无微不至的关怀和对我学业的鼓励与支持，激励我不断进取，我的成长凝聚着他们的心血，千言万语汇聚成为“谢谢”二字！