第5章 统一建模语言

在20世纪80年代到20世纪90年代，面向对象的分析与设计方法获得了长足的发展，而且相关的研究也十分活跃，涌现出了一大批新的方法学。其中，最著名的是Booch的Booch 1993、Jacobson的OOSE（Object-Oriented Software Engineering，面向对象软件工程）和Rumbaugh的OMT（Object Modeling Technique，对象建模技术）方法。UML正是在融合了Booch、Rumbaugh和Jacobson方法论的基础上形成的标准建模语言。

UML并不是本书的直接研究对象，而只是一种建模的工具。为了保持知识的连续性，本章首先介绍UML的一些基础知识，然后重点研究如何直接使用UML对软件体系结构建模，以及如何使用UML的扩展机制对软件体系结构建模。

5.1 UML概述

UML是用于系统的可视化建模语言，尽管它常与建模OO软件系统相关联，但由于其内建了大量扩展机制，还可以应用于更多的领域中，如工作流程、业务领域等。

（1）UML是一种语言：UML在软件领域中的地位与价值就像“1、2、3、+、-、…”等符号在数学领域中的地位一样。它为软件开发人员之间提供了一种用于交流的词汇表，一种用于软件蓝图的标准语言。

（2）UML是一种可视化语言：UML只是一组图形符号，它的每个符号都有明确语义，是一种直观、可视化的语言。

（3）UML是一种可用于详细描述的语言：UML所建的模型是精确的、无歧义和完整的，因此，适合于对所有重要的分析、设计和实现决策进行详细描述。

（4）UML是一种构造语言：UML虽然不是一种可视化的编程语言，但其与各种编程语言直接相连，而且有较好的映射关系，这种映射允许进行正向工程、逆向工程。

（5）UML是一种文档化语言：它适合于建立系统体系结构及其所有的细节文档。

5.1.1 UML的发展历史

面向对象的建模语言最早出现在20世纪70年代中期，在20世纪80年代末进入快速发展阶段，截至1994年，从不到10种发展到50多种。1994年之后，各种方法论逐渐拉开了差距，以Booch提出的Booch方法和Rumbaugh提出的OMT成为可视化建模语言的主导，而Jacobson的Objectory方法则成为最强有力的方法。

Booch是面向对象方法最早的倡导者之一。他在1984年出版的*Software Engineering with Ada*一书中就描述了面向对象软件开发的基本问题。1991年，他在*Object-Oriented*

Design 一书中,将以前针对 Ada 的工作扩展到整个面向对象设计领域,他对继承和类的阐述特别值得借鉴。Booch1993 比较适合于系统的设计和构造。

Runbaugh 等提出了 OMT,采用了面向对象的概念并引入各种独立于程序设计语言的表示符号。这种方法用对象模型、动态模型、功能模型和用例模型共同完成对整个系统的建模,所定义的概念和符号可用于软件开发的分析、设计和实现的全过程,软件开发人员不必在开发过程的不同阶段进行概念和符号的转换。OMT-2 特别适用于分析和描述以数据为中心的信息系统。

Jacobson 于 1994 年提出了 OOSE 方法,其最大特点是面向用例,在用例的描述中引入了外部角色的概念。用例的概念贯穿于整个开发过程(包括对系统的测试和验证),是精确描述需求的重要武器。目前,在学术界和工业界已普遍接受用例的概念,并认为是面向对象技术走向第二代的标志。OOSE 比较合适支持业务工程和需求分析。

各种建模语言各有长短,面对众多的建模语言,用户无力区分不同建模语言之间的差别和使用范围。由于不同的用户使用不同的建模语言和不同建模语言表达方式上的差异,使得用户之间的沟通方面出现了困难。要解决以上问题,就必须在综合分析各种不同建模语言的优缺点、适用情况的基础上统一各种不同的建模语言。

1994 年 10 月,Booch 和 Rumbaugh 开始了这项工作。首先,他们将 Booch93 和 OMT-2 统一起来,并于 1995 年 10 月发布了第一个公开版本,称之为 UM 0.8(Unified Method,标准方法)。同年,Jacobson 加盟到这项工作中,经过 Booch、Rumbaugh 和 Jacobson 的共同努力,于 1996 年 6 月和 10 月分别发布了两个新的版本(UML 0.9 和 UML 0.91),并将 UM 重新命名为 UML。

1997 年,OMG 采纳了 UML,一个开放的 OO 可视化建模语言工业标准诞生了。现在 UML 已经经历了 1.1、1.2 和 1.4 共 3 个版权的演变,2003 年发布了 2.0 版标准。

5.1.2 UML 的应用领域

UML 统一了 Booch、OMT、OOSE 和其他面向对象方法的基本概念和符号,同时汇集了面向对象领域中很多人的思想,是优秀的面向对象方法和丰富的计算机科学实践中总结而成的。

目前,UML 是最先进、最实用的标准建模语言,而且还在不断发展演化之中。UML 是一种建模语言而不是一种方法,其中并不包括过程的概念,其本身是独立于过程的,可以在任何过程中使用它。不过与 UML 结合最好的是用例驱动的、以体系结构为中心的、迭代的、增量的开发过程。

作为一种标准建模语言,UML 的核心是以面向对象的思想描述客观世界,具有广阔的应用领域。目前,主要应用领域是在软件系统建模,但是它同样可以应用于其他领域,如机械系统、企业机构或业务过程,以及处理复杂数据的信息系统、具有实时要求的工业系统或工业过程等。总之,UML 是一个通用的标准建模语言,可以对任何系统的动态行为和静态行为进行建模。

UML 可以为信息系统从需求分析到系统维护的整个生命周期提供有效的支持。在需求分析阶段,可以通过用例模型来捕获和组织用户的需求,分析系统对于用户的价值。通过用例建模,描述对系统感兴趣的外部角色及其对系统(用例)的功能要求。分析阶段主要关

心问题域中的主要概念(如抽象、类和对象等)和机制,以及这些概念之间的相互协作,并用UML类图来描述。至于类之间的协作关系则可以用交互图和顺序图来描述。在分析阶段,只对问题域的对象(现实世界的概念)建模,而不考虑定义软件系统中技术细节的类(如处理用户接口、数据库、通信、并行性等问题的类)。这些技术细节将在设计阶段引入,因此设计阶段为构造阶段提供更详细的规格说明。

编码阶段的主要任务是将设计阶段的设计结果转换成为实际的代码。在设计阶段需要注意的是,不要过早地考虑设计结果要用哪种编程语言实现。如果过早地考虑这个问题,会使设计工作陷入细节的泥潭,不利于对模型进行全面理解。

UML还可以对测试阶段提供有效的支持。不同的测试阶段可以使用不同的UML图作为测试的依据。例如,在单元测试阶段,可以使用类图和类的规格说明来指导测试;在集成测试阶段,可以使用通信图、活动图和部署图;系统测试和验收测试阶段则可以使用顺序图和用例图来验证系统的外部行为。

总之,UML能够用面向对象的方法描述任何类型的系统,并对系统开发从需求调研到测试和维护的各个阶段进行有效的支持。

5.2 UML的结构

UML的结构包括UML的基本构造块、支配这些构造块如何放在一起的规则(体系结构)和一些运用于整个UML的机制。

5.2.1 结构概述

本节将详细介绍UML结构的各组成元素。

1. 构造块

UML有3种基本的构造块,分别是事物(thing)、关系(relationship)和图(diagram)。事物是UML中重要的组成部分,关系把事物紧密联系在一起,图是很多有相互关系的事物的组。

2. 公共机制

公共机制是指达到特定目标的公共UML方法,主要包括规格说明(详细说明)、修饰、公共分类(通用划分)和扩展机制4种。

(1) 规格说明:规格说明是元素语义的文本描述,它是模型真正的核心。

(2) 修饰:UML为每一个事物设置了一个简单的记号,还可以通过修饰表达更多的信息。

(3) 公共分类:包括类元与实体(类元表示概念,实体表示具体的实体)、接口和实现(接口用来定义契约,而实现就是具体的内容)两组公共分类。

(4) 扩展机制:包括约束(添加新规则来扩展事物的语义)、构造型(用于定义新的事物)、标记值(添加新的特殊信息来扩展事物的规格说明)。

3. 规则

UML用于描述事物的语义规则分别是为事物、关系和图命名。给一个名字以特定含义的语境,即范围;怎样使用或看见名字,即可见性;事物如何正确、一致地相互联系,即完

整性；运行或模拟动态模型的含义是什么，即执行。

UML对系统体系结构的定义是系统的组织结构，包括系统分解的组成部分、它们的关联性、交互、机制和指导原则，这些提供系统设计的信息。而具体来说，就是指5个系统视图，分别是逻辑视图、进程视图、实现视图、部署视图和用例视图。

(1) 逻辑视图：以问题域的词汇组成的类和对象集合。

(2) 进程视图：可执行线程和进程作为活动类的建模，它是逻辑视图的一次执行实例，描述了所设计的并发与同步结构。

(3) 实现视图：对组成基于系统的物理代码的文件和构件进行建模。

(4) 部署视图：把构件部署到一组物理的、可计算的结点上，表示软件到硬件的映射及分布结构。

(5) 用例视图：最基本的需求分析模型。

提示：UML还允许在一定的阶段隐藏模型的某些元素、遗漏某些元素以及不保证模型的完整性，但模型逐步地要达到完整和一致。

5.2.2 事物

UML中的事物也称建模元素，包括结构事物(structural things)、行为事物(behavioral things，动作事物)、分组事物(grouping things)和注释事物(annotational things，注解事物)。这些事物是UML模型中最基本的面向对象的构造块。

1. 结构事物

结构事物在模型中属于最静态的部分，代表概念上或物理上的元素。总共有7种结构事物，具体如下。

(1) 类。类是描述具有相同属性、方法、关系和语义的对象的集合。一个类实现一个或多个接口。

(2) 接口。接口是指类或构件提供特定服务的一组操作的集合。因此，一个接口描述了类或构件的对外的可见的动作。一个接口可以实现类或构件的全部动作，也可以只实现一部分。

(3) 协作。协作定义了交互的操作，是一些角色和其他元素一起工作，提供一些合作的动作，这些动作比元素的总和要大。因此，协作具有结构化、动作化、维的特性。一个给定的类可能是几个协作的组成部分。这些协作代表构成系统的模式的实现。

(4) 用例。用例是描述一系列的动作，这些动作是系统对一个特定角色执行，产生值得注意的结果的值。在模型中用例通常用来组织行为事物。用例是通过协作实现的。

(5) 活动类。活动类的对象有一个或多个进程或线程。活动类和类很相似，只是它的对象代表的元素的行为和其他的元素是同时存在的。

(6) 构件。构件是物理上或可替换的系统部分，它实现了一个接口集合。在一个系统中，可能会遇到不同种类的构件。

(7) 结点。结点是一个物理元素，它在运行时存在，代表一个可计算的资源，通常占用一些内存和具有处理能力。一个构件集合一般来说位于一个结点，但有可能从一个结点转到另一个结点。

2. 行为事物

行为事物是 UML 模型中的动态部分。它们是模型的动词，代表时间和空间上的动作。总共有两种主要的行为事物，具体如下所述。

(1) 交互(内部活动)。交互是由一组对象之间在特定上下文中，为达到特定的目的而进行的一系列消息交换而组成的动作。交互中组成动作的对象的每个操作都要详细列出，包括消息、动作次序(消息产生的动作)、连接(对象之间的连接)等。

(2) 状态机。状态机由一系列对象的状态组成。

交互和状态机是 UML 模型中最基本的两个动态事物，它们通常和其他的结构事物、主要的类、对象连接在一起。

3. 分组事物

分组事物是 UML 模型中组织的部分，可以把它们看成一个盒子，模型可以在其中被分解。UML 只有一种分组事物，称为包。包是一种将有组织的元素分组的机制。结构事物、行为事物甚至其他的分组事物都有可能放在一个包中。与构件(存在于运行时)不同的是包纯粹是一种概念上的东西，只存在于开发阶段。

4. 注释事物

注释事物是 UML 模型的解释部分。

5.2.3 关 系

UML 用关系把事物结合在一起，UML 中的关系主要有以下 4 种。

(1) 依赖(dependencies)：两个事物之间的语义关系，其中一个事物发生变化会影响另一个事物的语义。

(2) 关联(association)：描述一组对象之间连接的结构关系，例如，聚合关系描述了整体和部分间的结构关系。

(3) 泛化(generalization)：一般化和特殊化的关系，描述特殊元素的对象可替换一般元素的对象。

(4) 实现(realization)：类之间的语义关系，其中的一个类指定了由另一个类保证执行的契约。

1. 用例之间的关系

两个用例之间的关系可以概括为两种情况。一种是用于重用的包含关系，用构造型<< include >>表示；另一种是用于分离出不同行为的扩展关系，用构造型<< extend >>表示。

(1) 包含关系：当可以从两个或两个以上的原始用例中提取公共行为，或者发现能够使用一个构件来实现某一个用例很重要的部分功能时，应该使用包含关系来表示它们。提取出来的公共用例称为抽象用例。

例如，希赛图书订单处理系统中，“创建新订单”和“更新订单”两个用例都需要检查客户的账号是否正确，为此定义一个抽象用例“核查客户账户”。用例“创建新订单”和“更新订单”与用例“核查客户账户”之间的关系就是包含关系。

(2) 扩展关系：如果一个用例明显地混合了两种或两种以上的不同场景，即根据情况可能发生多种事情，则可以断定将这个用例分为一个主用例和一个或多个辅用例进行描述可能更加清晰。

例如,希赛图书管理系统中,读者归还图书时,需要判断当前日期是否已经超过了图书借阅的周期。如果超过了借阅周期,则必须罚款。“归还图书”和“罚款”用例之间的关系就是扩展关系。

用例之间还存在一种泛化关系。用例可以被特别列举为一个或多个子用例,这被称作用例泛化。当父用例能够被使用时,任何子用例也可以被使用。例如,购买飞机票时,既可以通过电话订票,也可以通过网上订票,则订票用例就是电话订票和网上订票的泛化。

2. 类之间的关系

在建立抽象模型时,很少有类会单独存在,大多数都将会以某种方式彼此通信。因此,还需要描述这些类之间的关系。

(1) 关联关系。描述了给定类的单独对象之间语义上的连接。关联提供了不同类之间的对象可以相互作用的连接。其余的关系涉及类元自身的描述,而不是它们的实例。关联关系用———表示。

(2) 依赖关系。有两个元素 X、Y,如果修改元素 X 的定义可能会引起对另一个元素 Y 的定义的修改,则称元素 Y 依赖于元素 X。在 UML 中,使用带箭头的虚线------▶表示依赖关系。

在类中,依赖由各种原因引起。例如,一个类向另一个类发送消息;一个类是另一个类的数据成员;一个类是另一个类的某个操作参数。如果一个类的接口改变,则它发出的任何消息都可能不再合法。

(3) 泛化关系。泛化关系描述了一般事物与该事物中的特殊种类之间的关系,即父类与子类之间的关系。继承关系是泛化关系的反关系,即子类是从父类继承的,而父类则是子类的泛化。在 UML 中,使用带空心箭头的实线———▷表示泛化关系,箭头指向父类。

(4) 聚合关系。聚合是一种特殊形式的关联,它是传递和反对称的。聚合表示类之间的关系是整体与部分的关系。例如,一辆轿车包含 4 个车轮、一个方向盘、一个发动机和一个底盘,就是聚合的一个例子。在 UML 中,使用一个带空心菱形的实线———◇表示聚合关系,空心菱形指向的是代表“整体”的类。

(5) 组合关系。如果聚合关系中的表示“部分”的类的存在与否,与表示“整体”的类有着紧密的关系,如“公司”与“部门”之间的关系,那么就应该使用“组合”关系来表示这种关系。在 UML 中,使用带有实心菱形的实线———◆表示组合关系。

(6) 实现关系将说明和实现联系起来。接口是对行为而非实现的说明,而类则包含实现的结构。一个或多个类可以实现一个接口,而每个类分别实现接口中的操作。在 UML 中,实现关系用-----▷表示。

(7) 流关系将一个对象的两个版本以连续的方式连接起来。它表示一个对象的值、状态和位置的转换。流关系可以将类元角色在一次相互作用中连接起来。流的种类包括变成(同一个对象的不同版本)和复制(从现有对象创造出一个新的对象)两种。在 UML 中,用----->表示流关系。

说明:对于聚合关系和组合关系,各种文献的说法有些区别。在这些文献中,首先定义聚集关系(整体与部分的关系),然后再把聚集关系分为两种,分别是组合聚集(相当于上述的“组合关系”)和共享聚集(相当于上述的“聚合关系”)。

5.2.4 图形

UML 2.0 包括 14 种图，分别列举如下。

(1) 类图(class diagram)：描述一组类、接口、协作和它们之间的关系。在面向对象系统的建模中，最常见的图就是类图。类图给出了系统的静态设计视图，活动类的类图给出了系统的静态进程视图。

(2) 对象图(object diagram)：描述一组对象及它们之间的关系。对象图描述了在类图中所建立的事物实例的静态快照。和类图一样，这些图给出系统的静态设计视图或静态进程视图，但它们是从真实案例或原型案例的角度建立的。

(3) 构件图(component diagram)：描述一个封装的类和它的接口、端口，以及由内嵌的构件和连接件构成的内部结构。构件图用于表示系统的静态设计实现视图。对于由小的部件构建大的系统来说，构件图是很重要的。构件图是类图的变体。

(4) 组合结构图(composite structure diagram)：描述结构化类(如构件或类)的内部结构，包括结构化类与系统其余部分的交互点。它显示联合执行包含结构化类的行为的构件配置。组合结构图用于画出结构化类的内部内容。

(5) 用例图(use case diagram)：描述一组用例、参与者(一种特殊的类)及它们之间的关系。用例图给出系统的静态用例视图。这些图在对系统的行为进行组织和建模时非常重要。

(6) 顺序图(sequence diagram，序列图)：是一种交互图(interaction diagram)。交互图展现了一种交互，它由一组对象或角色以及它们之间可能发送的消息构成。交互图专注于系统的动态视图。顺序图是强调消息的时间次序的交互图。

(7) 通信图(communication diagram)：也是一种交互图，它强调收发消息的对象或角色的结构组织。顺序图和通信图表达了类似的基本概念，但每种图所强调的概念不同，顺序图强调的是时序，通信图则强调消息流经的数据结构。在 UML 1.X 版本中，通信图被称为协作图(cooperation diagram)。

(8) 定时图(timing diagram，计时图)：也是一种交互图，它强调消息跨越不同对象或角色的实际时间，而不仅仅只是关心消息的相对顺序。

(9) 状态图(state diagram)：描述一个状态机，它由状态、转移、事件和活动组成。状态图给出了对象的动态视图。它对于接口、类或协作的行为建模尤为重要，而且它强调事件导致的对象行为，这非常有助于对反应式系统建模。

(10) 活动图(activity diagram)：将进程或其他计算的结构展示为计算内部一步步的控制流和数据流。活动图专注于系统的动态视图。它对系统的功能建模特别重要，并强调对象间的控制流程。

(11) 部署图(deployment diagram)：描述对运行时的处理结点及在其中生存的构件的配置。部署图给出了体系结构的静态部署视图，通常一个结点包含一个或多个部署图。

(12) 制品图(artifact diagram)：描述计算机中一个系统的物理结构。制品包括文件、数据库和类似的物理比特集合。制品图通常与部署图一起使用。制品也给出了它们实现的类和构件。

(13) 包图(package diagram)：描述由模型本身分解而成的组织单元，以及它们的依赖关系。

(14) 交互概览图(interaction overview diagram)：是活动图和顺序图的混合物。

5.3 用 例 图

用例实例是在系统中执行的一系列动作,这些动作将生成特定参与者可见的价值结果。一个用例定义一组用例实例。它确定了一个和系统参与者进行交互、并可由系统执行的动作序列。用例模型描述的是外部参与者(actor)所理解的系统功能。用例模型用于需求分析阶段,它的建立是系统开发者和用户反复讨论的结果,表明了开发者和用户对需求规格达成的共识。

在UML中,用例表示为一个椭圆。图5-1显示了希赛教育图书管理系统的用例图。其中,"新增书籍信息""查询书籍信息""修改书籍信息""登记外借信息""查询外借信息""统计金额与册数"等都是用例的实例。

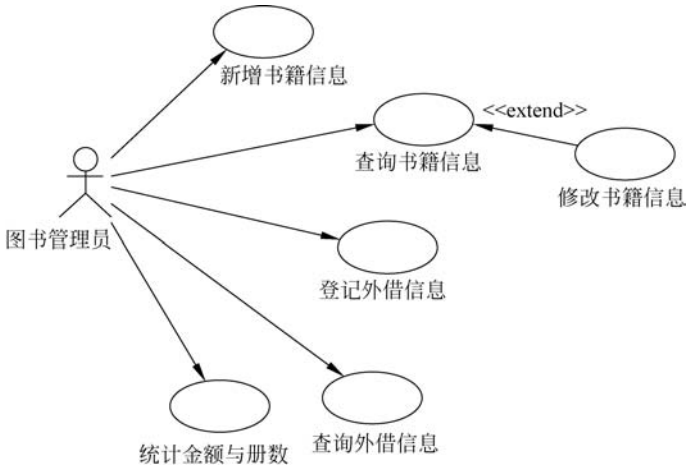

图5-1 用例图示例

1. 参与者

参与者代表与系统接口的任何事物(包括其他系统),它是指代表某一种特定功能的角色,因此参与者是虚拟的概念。在UML中,用一个小人表示参与者。

图5-1中的"图书管理员"就是参与者。对于该系统来说,可能可以充当图书管理员角色的有多个人,因为他们对于系统而言均起着相同的作用,扮演相同的角色,因此,只用一个参与者表示。切记不要为每一个可能与系统交互的人画出一个参与者。

可以通过以下问题找到系统的相关参与者。

- 谁是系统的主要用户?
- 谁从系统获得信息?
- 谁向系统提供信息?
- 谁从系统删除信息?
- 谁支付、维护系统?
- 谁管理系统?
- 系统需要与其他哪些系统交互?

- 系统需要操纵哪些硬件？
- 在预设的时间内，有事情自动发生吗？
- 系统从哪里获得信息？
- 谁对系统的特定需求感兴趣？
- 几个人在扮演同样的角色吗？
- 一个人扮演几个不同的角色吗？
- 系统使用外部资源吗？
- 系统用在什么地方？

2. 用例

用例是对系统行为的动态描述，它可以促进设计人员、开发人员与用户的沟通，理解正确的需求，还可以划分系统与外部实体的界限，是系统设计的起点。在识别出参与者之后，可以使用下列问题帮助识别用例。

- 每个参与者的任务是什么？
- 有参与者将要创建、存储、修改、删除或读取系统中的信息吗？
- 什么用例会创建、存储、修改、删除或读取这个信息？
- 参与者需要通知系统外部的突然变化吗？
- 需要把系统中正在发生的事情通知参与者吗？
- 什么用例将支持和维护系统？
- 所有的功能需求都对应到用例中了吗？
- 系统需要何种输入输出？输入从何处来？输出到何处？
- 当前运行系统的主要问题是什么？

5.4 类图和对象图

在面向对象建模技术中，对象指现实世界中有意义的事物，具有封装性和自治性的特点；而类指具有相同属性和行为的一组对象。类、对象和它们之间的关联是面向对象技术中最基本的元素。对于一个想要描述的系统，其类模型和对象模型揭示了系统的结构。在UML中，类和对象模型分别由类图和对象图表示。类图技术是OO方法的核心。图5-2显示了希赛教育图书管理系统的类图。

对象与人们对客观世界的理解相关，它通常用来描述客观世界中某个具体的事物。而类是对一组具有相同属性，表现相同行为的对象的抽象。因此，对象是类的实例。在UML中，类的可视化表示为一个划分成3个格子的长方形(下面两个格子可省略)。在图5-2中，“书籍”“借阅记录”等都是一个类。

(1) 类的命名：最顶部的格子包含类的名字。类的命名应尽量用应用领域中的术语，应明确、无歧义，以利于开发人员与用户之间的相互理解和交流。

(2) 类的属性：中间的格子包含类的属性，用以描述该类对象的共同特点。该项可省略。在图5-2中，“书籍”类有“书名”“书号”等属性。UML规定类的属性的语法为：“可见性 属性名：类型 = 默认值{约束特性}”。

① 可见性：包括Public、Private和Protected，分别用＋、－、＃表示。

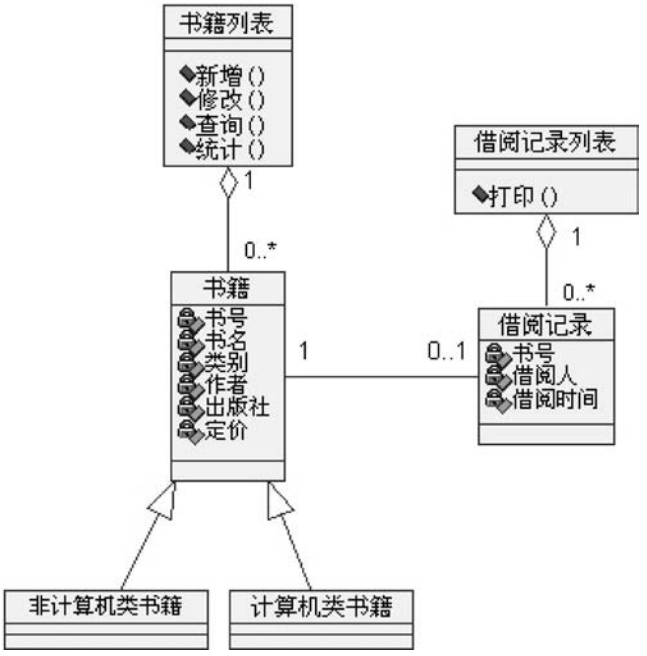

图 5-2　希赛教育图书管理系统的类图

② 类型：表示该属性的种类，它可以是基本数据类型(如整数、实数、布尔型等)，也可以是用户自定义的类型。一般它由所涉及的程序设计语言确定。

③ 约束特性：用户对该属性性质一个约束的说明。例如，“{只读}”说明该属性是只读属性。

(3) 类的操作(Operation)：该项可省略。操作用于修改、检索类的属性或执行某些动作。操作通常也被称为功能，但是它们被约束在类的内部，只能作用到该类的对象上。操作名、返回类型和参数表组成操作界面。UML 规定操作的语法为：“可见性：操作名(参数表)：返回类型 {约束特性}”。

类图描述了类和类之间的静态关系。定义了类之后，就可以定义类之间的各种关系了，请参考 5.2.3 节的介绍。与数据模型不同，类图不仅显示了信息的结构，同时还描述了系统的行为。类图是面向对象建模中最重要的模型。

UML 中对象图与类图具有相同的表示形式，对象图可以看作是类图的一个实例。对象之间的链(Link)是类之间的关联的实例。

5.5　交　互　图

交互图是表示各组对象如何依某种行为进行协作的模型。通常可以使用一个交互图表示和说明一个用例的行为。在 UML 中，包括 4 种不同形式的交互图，分别是顺序图、通信图、定时图和交互概览图。其中，顺序图强调的是时序，通信图则强调消息流经的数据结构，定时图强调消息跨越不同对象或角色的实际时间，交互概览图是活动图和顺序图的混合物。

顺序图和通信图是两种基本的交互图，它们之间没有什么本质不同，只是排版不尽相同而已；定时图和交互概览图是两种特殊的变体。

本节简单介绍顺序图、通信图和定时图，有关交互概览图的知识，请阅读 5.7.3 节。

5.5.1 顺序图

顺序图用来描述对象之间动态的交互关系，着重体现对象间消息传递的时间顺序。顺序图允许直观地表示出对象的生存期，在生存期内，对象可以对输入消息做出响应，并且可以发送信息。

如图 5-3 所示，顺序图存在两个轴：水平轴表示不同的对象；垂直轴表示时间，表示对象及类的生命周期。

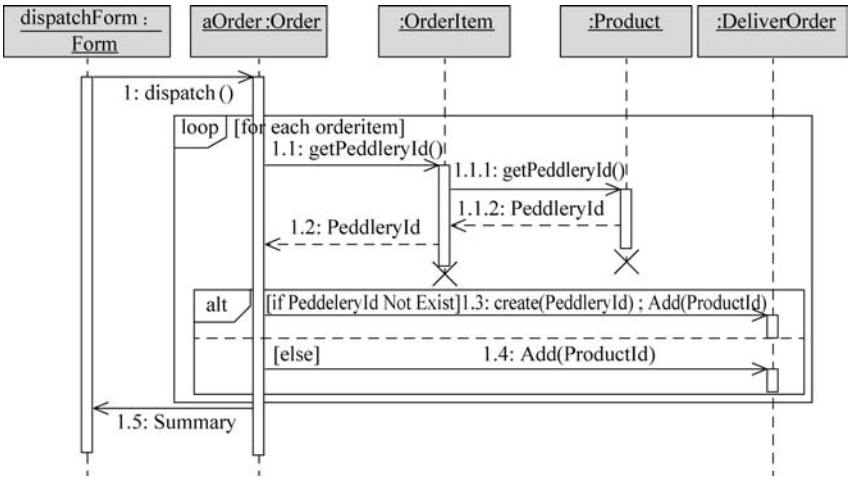

图 5-3 顺序图示例

对象间的通信通过在对象的生命线间画消息表示。消息的箭头指明消息的类型。顺序图中的消息可以是信号、操作调用或类似于 C++中的 RPC 和 Java 中的 RMI。收到消息时，接收对象立即开始执行活动，即对象被激活。通过在对象生命线上显示一个细长矩形框表示激活。

消息可以用消息名及参数标识，消息也可带有顺序号。消息还可带有条件表达式，表示分支或决定是否发送消息。如果用于表示分支，则每个分支相互排斥，即在某一时刻仅可发送分支中的一个消息。

5.5.2 通信图

通信图强调收发消息的对象或角色的结构组织。顺序图和通信图表达了类似的基本概念，但每种图强调概念的不同视图，顺序图强调时序，通信图强调消息流经的数据结构。图 5-4就是与图 5-3 相对应的通信图，从图 5-4 中可以很明显地发现通信图与顺序图之间的异同点。

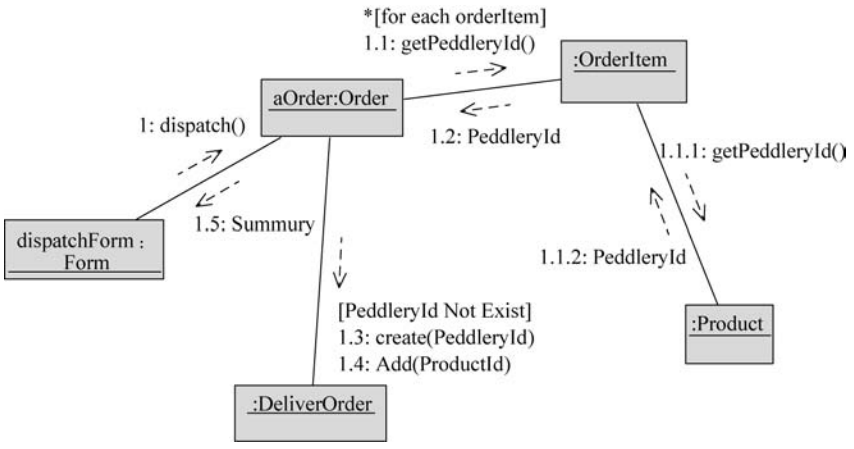

图 5-4　通信图示例

5.5.3　定时图

如果要表示的交互具有很强的时间特性(例如,现实生活中的电子工程、实时控制等系统中),在 UML 1.X 中是无法有效地表示出来的。而在 UML 2.0 中引入了一种新的交互图来解决这类问题,这就是着重表示定时约束的定时图。

根据 UML 的定义,定时图实际上是一种特殊形式的顺序图(即一种变化),它与顺序图的区别主要体现在以下几个方面。

(1) 坐标轴交换了位置,改为从左到右表示时间的推移。

(2) 用生命线的"凹下凸起"表示状态的变化,每个水平位置代表一种不同的状态,状态的顺序可以有意义,也可以没有意义。

(3) 生命线可以跟在一根线后面,在这根线上显示一些不同的状态值。

(4) 可以显示一个度量时间值的标尺,用刻度表示时间间隔。

图 5-5 是一个定时图的实例,其中小黑点加曲线表示的是注释。它用来表示一个电子门禁系统的控制逻辑,该门禁系统包括门(物理的门)、智能读卡器(读取用户的智能卡信息)、处理器(用来处理是否开门的判断)。

在这个例子中,它所表示的意思是一开始读卡器是启用的(等用户刷卡)、处理器是空闲的(没有验证的请求)、门是关的;接着,当用户刷卡时,读卡器就进入了"等待校验"的状态,并发一个消息给处理器,处理器就进入了校验状态。如果校验通过,就发送一个"禁用"消息给读卡器(门开的时候,读卡器就可以不工作了),使读卡器进入禁用状态;并且处理器转入启用状态,这时门的状态变成了"开"。而门"开"了 30s(根据时间刻度得知)之后,处理器将会把它再次"关"上,并且发送一个"启用"消息给读卡器(门关了,读卡器又要重新工作了),这时读卡器再次进入启用状态,而处理器已经又回到空闲状态。

从上面的例子中,不难看出定时图所包含的图元并不多,主要包括生命线、状态、状态变迁、消息、时间刻度,可以根据需要来使用它。

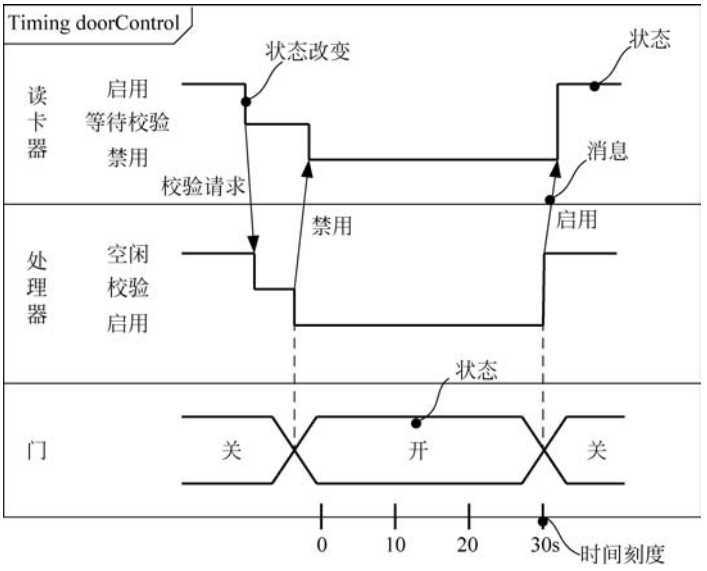

图 5-5　定时图示例

5.6　状　态　图

状态图用来描述对象状态和事件之间的关系，通常用状态图来描述单个对象的行为。它确定了由事件序列引出的状态序列，但并不是所有的类都需要使用状态图来描述它的行为，只有具有重要交互行为的类，才会使用状态图来描述。

图 5-6 是一个数码冲印店的订单状态图实例，如图 5-6 所示，状态图包括以下部分。

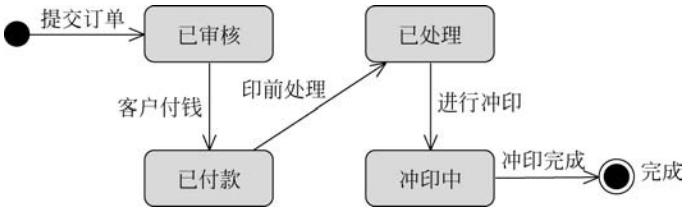

图 5-6　状态图示例

(1) 状态：又称中间状态，用圆角矩形框表示。

(2) 初始状态：又称初态，用一个黑色的实心圆圈表示。一张状态图中只能有一个初始状态。

(3) 结束状态：又称终态，在黑色的实心圆圈外面套上一个空心圆。在一张状态图中可能有多个结束状态。

(4) 状态转移：用箭头说明状态的转移情况，并用文字说明引发这个状态变化的相应事件是什么。

一个状态也可能被细分为多个子状态,那么如果将这些子状态都描绘出来,则这个状态就是复合状态。

状态图适合用于表述在不同用例之间的对象行为,但并不适合于表述包括若干协作的对象行为。通常不会需要对系统中的每一个类绘制相应的状态图,而通常会在业务流程、控制对象、用户界面的设计方面使用状态图。

5.7 活 动 图

活动图用来表示系统中各种活动的次序,它的应用非常广泛,既可用来描述用例的工作流程,也可以用来描述类中某个方法的操作的行为。活动图是由状态图变化而来的,它们各自用于不同的目的。活动图依据对象状态的变化来捕获动作(将要执行的工作或活动)与动作的结果。活动图中一个活动结束后将立即进入下一个活动(在状态图中,状态的变迁可能需要事件的触发)。

5.7.1 基本活动图

图 5-7 给出了一个基本活动图的例子。

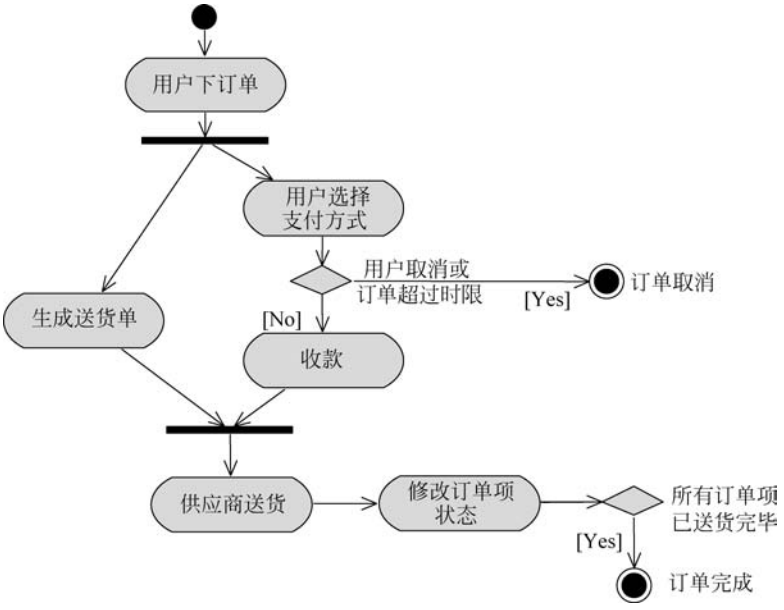

图 5-7 基本活动图示例

如图 5-7 所示,活动图与状态图类似,包括初始状态、终止状态,以及中间的活动状态,每个活动之间,也就是一种状态的变迁。在活动图中,还引入了以下几个概念。

(1) 判定:说明基于某些表达式的选择性路径,在 UML 中使用菱形表示。

(2) 分叉与结合:由于活动图建模时经常会遇到并发流,因此,在 UML 中引入了如图 5-7所示的粗实线表示分叉和结合。

5.7.2 带泳道的活动图

在5.7.1节说明的基本活动图中，虽然能够描述系统发生了什么，但无法说明完成这个活动的对象。针对OOP(Object-Oriented Programming，面向对象的程序设计)而言，这就意味着活动图没有描述出各个活动由哪个类来完成。要想解决这个问题，可以通过泳道。泳道将活动图的逻辑描述与顺序图、通信图的责任描述结合起来，如图5-8所示。

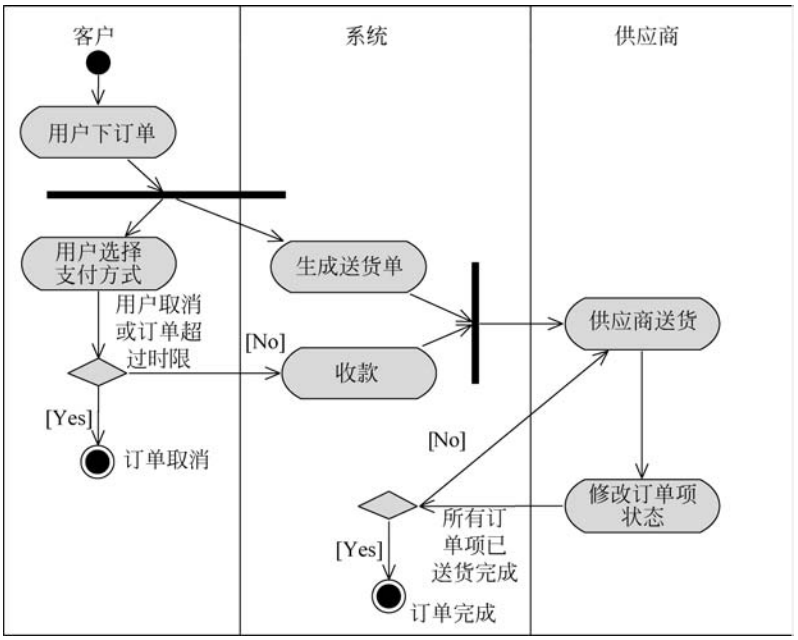

图5-8 带泳道活动图示例

在活动图中，对象可以作为活动的输入或输出，对象与活动间的输入/输出关系由虚线箭头表示。如果仅表示对象受到某一活动的影响，则可用不带箭头的虚线连接对象与活动。

在活动图中，可以通过信号的发送和接收标记表示信号的发送和接收；发送和接收标记也可与对象相连，用于表示消息的发送者和接收者。

5.7.3 交互概览图

交互概览图并没有引入新的事物，其所有的图示法都已经在顺序图和活动图中阐述过。例如，在图5-7中关于“用户订单处理”的活动图中，如果想表达出“用户下订单”和“生成送货单”两个活动结点内的对象控制流，就可以结合5.5.1节中的顺序图的知识，绘制出一张交互概览图，其结果如图5-9所示。

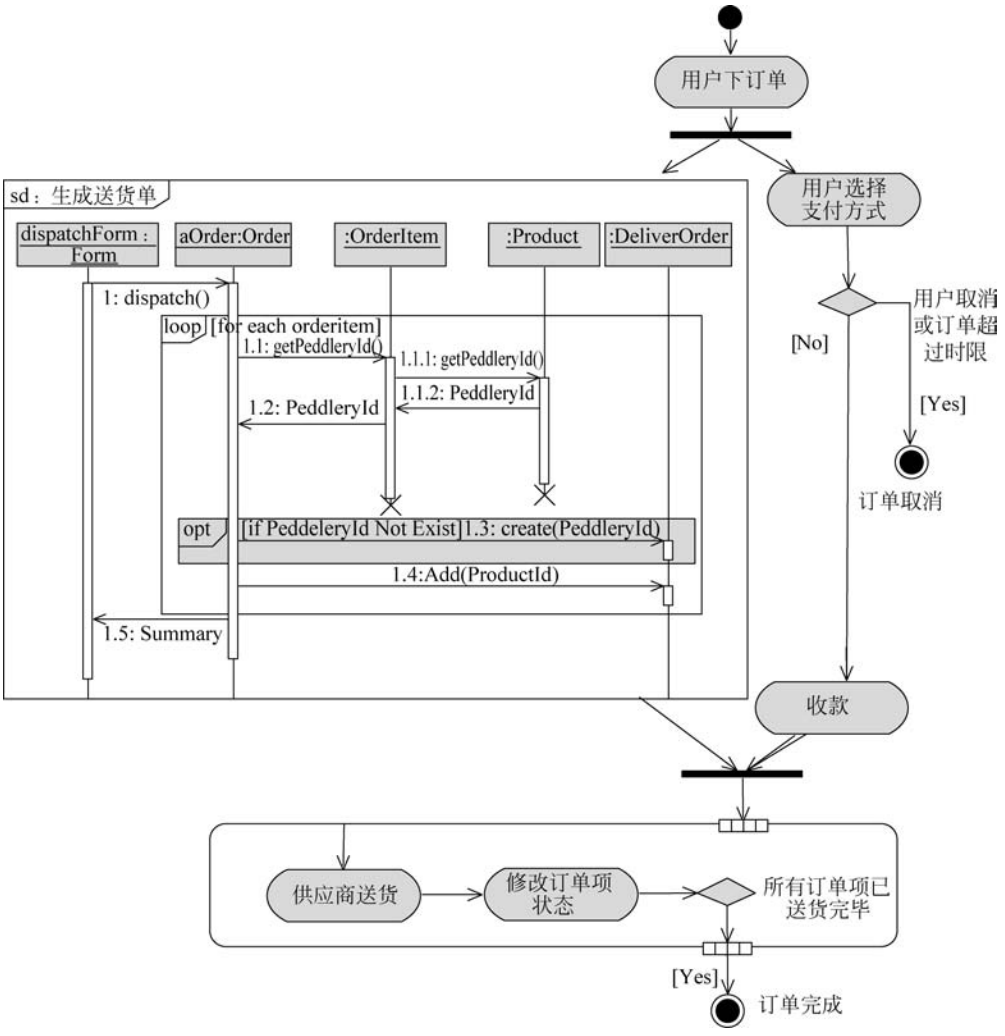

图 5-9　交互概览图示例

5.8　构　件　图

对面向对象系统物理方面进行建模，需要使用两种图，分别是构件图和部署图。构件图可以有效地显示一组构件，以及它们之间的关系。构件图中通常包括构件、接口以及各种关系。图 5-10 就是一个构件图的例子。

构件指源代码文件、二进制代码文件、可执行文件等，而构件图用来显示编译、链接或执行时构件之间的依赖关系。例如，在图 5-10 中，就是说明 QueryClient. exe 将通过调用 QueryServer. exe 来完成相应的功能，而 QueryServer. exe 则需要 Find. exe 的支持，Find. exe 在实现时调用了 Query. dll。

通常来说，可以使用构件图完成以下工作。

(1) 对源代码进行建模。构件图可以清晰地表示出各个不同源程序文件之间的关系。

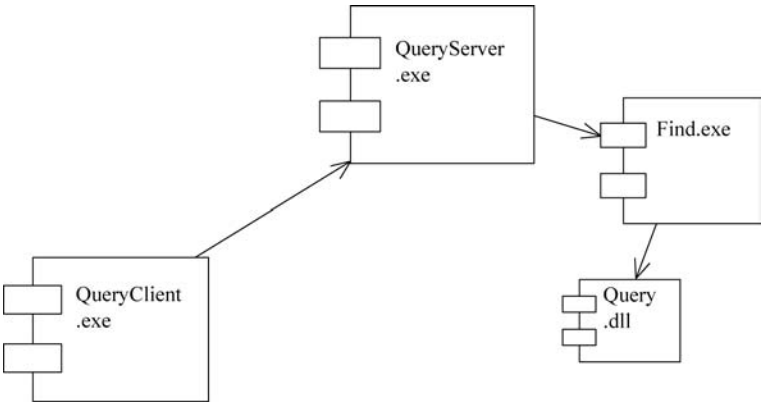

图 5-10　构件图示例

(2) 对可执行体的发布建模：如图 5-10 所示，将清晰地表示出各个可执行文件、DLL (Dynamic Link Library，动态链接库)文件之间的关系。

(3) 对物理数据库建模。用来表示各种类型的数据库、表之间的关系。

(4) 对可调整的系统建模。例如，对于应用负载均衡、故障恢复等系统的建模。

在绘制构件图时，应该注意侧重于描述系统的静态实现视图的一个方面，图形不要过于简化，应该为构件图取一个直观的名称，在绘制时避免产生线的交叉。

5.9　部　署　图

部署图也称实施图，构件图说明构件之间的逻辑关系，而部署图则是在此基础上更进一步，描述系统硬件的物理拓扑结构，以及在此结构上执行的软件。部署图可以显示计算结点的拓扑结构和通信路径、结点上运行的软件构件，常常用于帮助理解分布式系统。

图 5-11 就是与图 5-10 对应的部署图，这样的图示可以使系统的安装、部署更为简单。在部署图中，通常包括以下一些关键的组成部分。

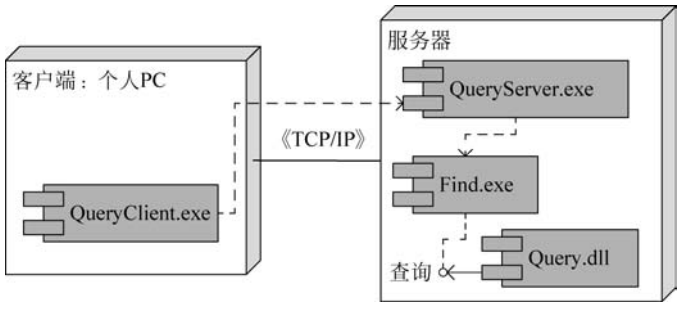

图 5-11　部署图示例

(1) 结点(Node)和连接。结点代表一个物理设备以及其上运行的软件系统，如一台 Windows Server 主机、一个 PC 终端、一台打印机、一个传感器等。如图 5-11 所示，“客户端：PC”和“服务器”就是两个结点。在 UML 中，使用一个立方体表示一个结点，结点名放在左上角。结点之间的连线表示系统之间进行交互的通信路径，在 UML 中被称为连接。

通信类型则放在连接旁边的《》之间,表示所用的通信协议或网络类型。

(2) 构件和接口。在部署图中,构件代表可执行的物理代码模块,如一个可执行程序等。逻辑上它可以与类图中的包或类对应。例如,在图 5-11 中,"服务器"结点中包含 QueryServer. exe、Find. exe 和 Query. dll 3 个构件。在面向对象方法中,类和构件等元素并不是所有的属性和操作都对外可见。它们对外提供了可见操作和属性,称为类和构件的接口。界面可以表示为一头是小圆圈的直线。在图 5-11 中,Query. dll 构件提供了一个"查询"接口。

5.10 使用 UML 建模

UML 基于主流的软件开发方法及开发经验,是明确定义了语法和语义的可视化建模语言。其中,图形表示的语法由实例和图形元素到语义模型中元素的映射表示,语义模型的语法和语义由元模型、自然语言和约束来说明。

UML 的 4 层元模型的体系结构如图 5-12 所示。其中元-元模型(meta-meta model)层定义了元模型(meta model)层的规格说明语言,元模型层为给定的建模语言定义规格说明,模型层用来定义特定软件系统的模型,用户对象(user object)用来构建给定模型的特定实例。在 4 层元模型中,UML 的结构主要体现在元模型中,分为 3 个逻辑包,分别是基础包(foundation)、行为元素包(behavioral elements)和一般机制包(general mechanisms),这些包依次又分为若干子包。例如,基础包分为核心元素、辅助元素、扩充机制、数据类型几个子包。

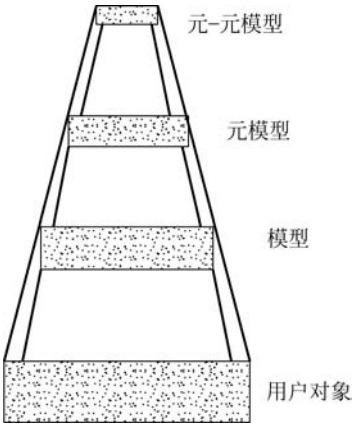

图 5-12 UML 的 4 层元模型体系结构

语义约束由对象约束语言 OCL 表示,OCL 基于一阶谓词逻辑,每一个 OCL 表达式都处于一些 UML 模型元素的背景下(由 self 引用),可使用该元素的属性和关系作为其项(term),同时 OCL 定义了在集合(sets)、袋(bags)等上的公共操作集和遍历建模元素间关系的构造,因此,其他建模元素的属性也可以作为它的项。

UML 模型从以下几个方面说明软件系统:类及其属性、操作和类之间的静态关系,类的包(package)和它的依赖关系;类的状态及其行为;对象之间的交互行为,使用事例;源码的结构以及执行的实施结构。UML 中的通用表示如下。

(1) 字符串:表示有关模型的信息。

(2) 名字:表示模型元素。

(3) 标号:不同于编程语言中的标号,是用于表示或说明图形符号的字符串。

(4) 特殊字符串:表示某一模型元素的特性。

(5) 类型表达式:声明属性、变量及参数,含义同编程语言中的类型表达式。

(6) 实体类型(stereotype):它是 UML 的扩充机制,运用实体类型可定义新类型的模型元素。

(7) UML 语义部分是对 UML 的准确表示,主要由以下 3 部分组成。

① 通用元素(common element):主要描述 UML 中各元素的语义。通用元素是 UML

中的基本构造单位，包括模型元素和视图元素，模型元素用来构造系统，视图元素用来构成系统的表示成分。

② 通用机制(common mechanism)：主要描述使 UML 保持简单和概念上一致的机制的语义，包括定制、标记值、注记、约束、依赖关系、类型-实例、类型-类的对应关系等机制。

③ 通用类型(common type)：主要描述 UML 中各种类型的语义。这些类型包括布尔类型、表达式类型、列表类型、多重性类型(multiplicity)、名字类型、坐标类型、字符串类型、时间类型、用户自定义类型等。

UML 中语义的 3 部分不是相互独立的，而是相互交叉重叠、紧密相连，共同构成了 UML 的完整语义。

下面以 4.4.3 节中会议安排系统为例，来说明如何使用 UML 描述软件体系结构及其元素。

图 5-13 是用 UML 的类图表示会议安排系统，该图描述了会议安排系统的领域模型，包括领域中的类及其继承和关联的关系。

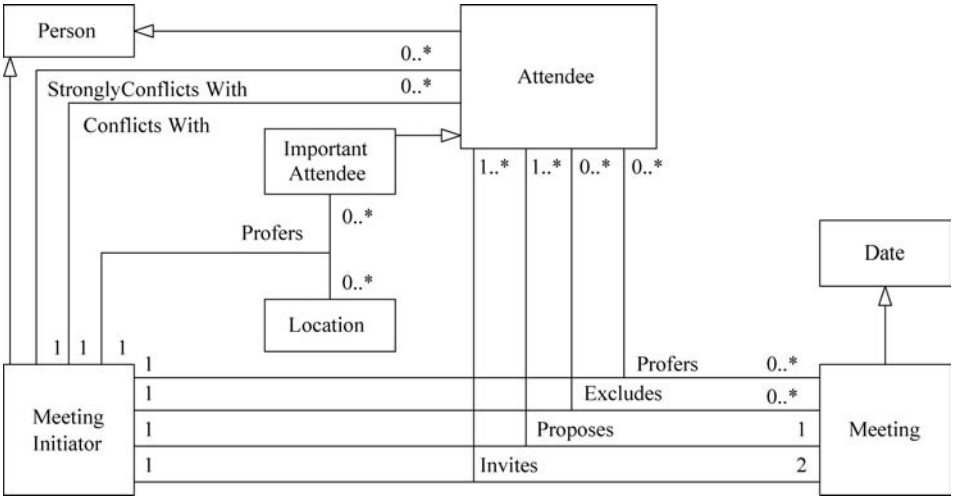

图 5-13　会议安排系统的类图

在图 5-13 中，除了限制 MeetingInitiator 为单一实例外，省略了许多体系结构细节。例如，把领域中的类映射到可实现的构件，不同类之间的交互顺序等。图中还省略了类交互的语义。例如，Invites 关联把两个 Meeting 与一个或多个 Attendee 或一个 MeetingInitiator 关联，然而这种关联只标明了两个会议，并没有指明这两个 Meeting 打算提交会议可能举行的日期范围。

在 C2 风格的体系结构中，消息结构是一个重要的元素。因此，在 UML 设计中，要对类的接口进行建模，如图 5-14 所示。

在图 5-14 中，接口 ImportantMtgInit 和 ImportantMtgAttend 分别继承了接口 MtgInit 和 MtgAttend，唯一的区别是增加了会议地点的请求和通知。要注意的是，图 5-14 中每一个方法签名(Method Signature，即 UML 的操作)都对应了一个 C2 消息(见 4.4.3 节)。UML 模型中的所有操作将作为异步消息传递来实现。正因如此，图 5-14 中的方法签名没有写返回类型。

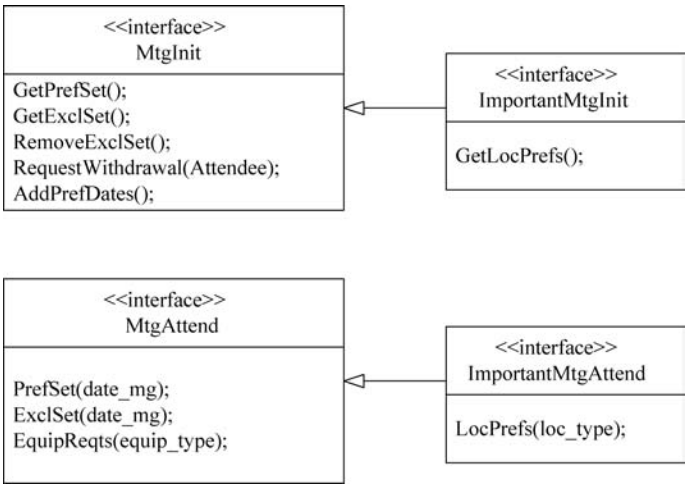

图 5-14　会议安排系统类接口

为了在 UML 中为体系结构建模,还必须定义连接件。虽然连接件充当的角色与构件不同,但是它们也可以使用 UML 的类图来建模。然而,一个 C2 连接件可以为任何数量和类型的 C2 构件提供连接,其实是一个与其有关联的构件接口的集合。但是 UML 不支持这种属性,相反,在 UML 中指定的连接件必须是与应用特定的,并且必须有固定的接口。为了体现 C2 连接件的特性,会议安排系统的连接件类实现与其关联的构件的同样接口。每个连接件被当作一个简单的类,这个类能够把自己接收到的消息传递给适当的构件,因此,在图 5-14 给出的构件接口的基础上,连接件模型如图 5-15 所示。

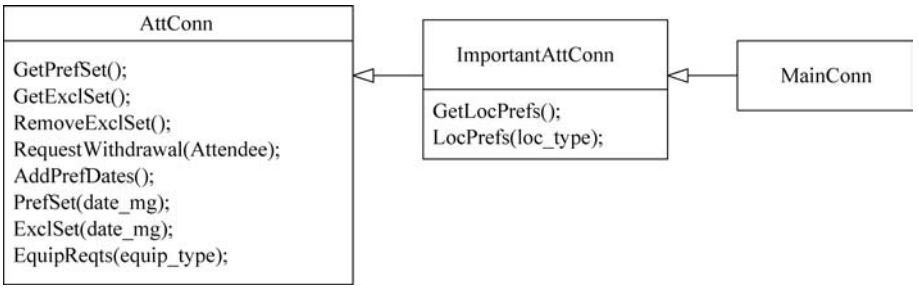

图 5-15　C2 连接件模型

在领域模型层次上,连接件不能增加任何功能,因此在图 5-13 中没有被标记出来。图 5-16 是一个细化了的会议安排系统的类图,该图描述了类之间的接口关系。

在图 5-16 中,每个实线圆弧旁边的文字是接口的名称,描述了接口的实现。每个虚线箭头表示类对接口的依赖性。要特别说明的是,在图 5-14 中,类 Attendee 和 ImportantAttendee 是由接口继承进行关联的,但在图 5-16 中没有明显体现出来。图 5-16 省略了图 5-13 中的 Location、Meeting 和 Date 类,因为这些类只是代表系统中构件之间交换的数据;同时,也省略了构件超类 Person 和连接件超类 Conn。

图 5-17 描述了 MeetingInitiator 类实例和 Attendee 类以及 ImportantAttendee 类实例之间的协作。

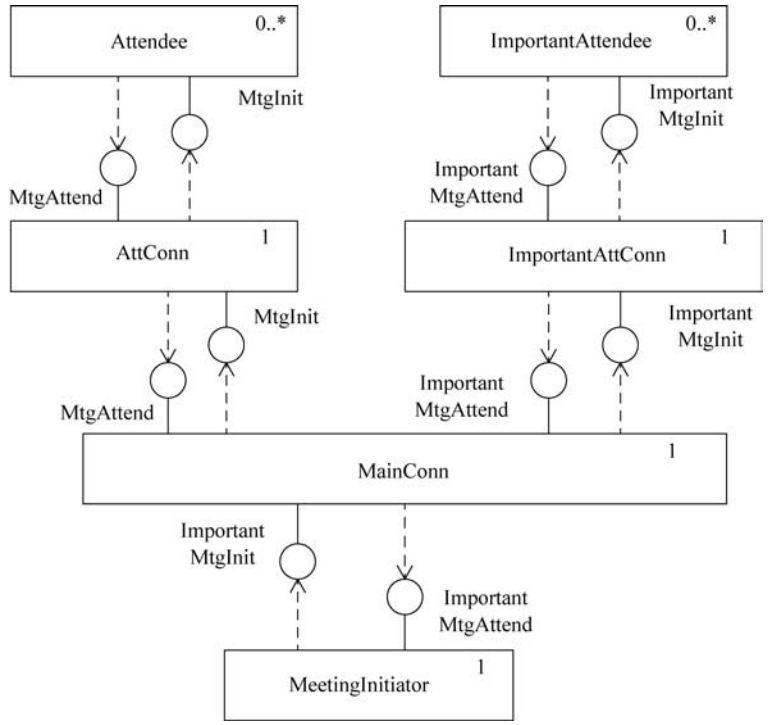

图 5-16　细化的类图

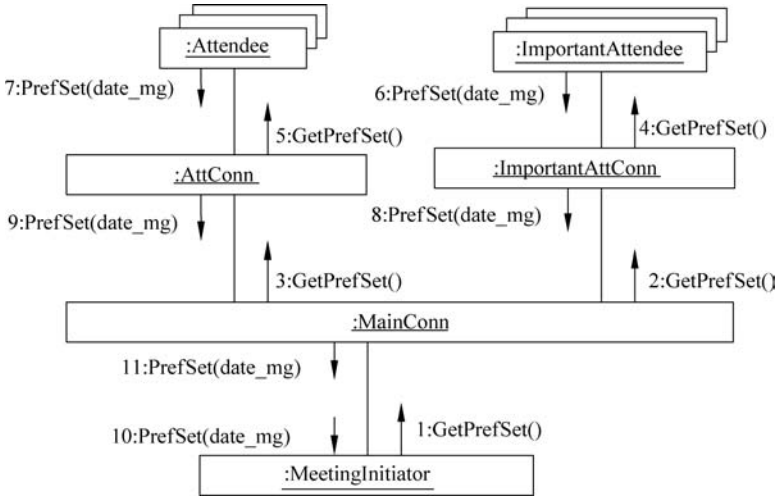

图 5-17　会议安排系统的协作图

在图 5-17 中，MeetingInitiator 发出会议安排日期请求，连接件 MainConn 的实例把这个请求转交给其上方的连接件 AttConn 和 ImportantAttConn 的实例，这两个实例再把这个请求转交给其上方的构件，每个与会人员构件选择自己喜欢的日期，并把该选择通知其下方的构件，这个通知信息最终由连接件转交给 MeetingInitiator。但是，如果 MeetingInitiator 发出会议地点安排请求，则 MainConn 只把该请求转交给 ImportantAttConn，则任何 Attendee 类的实例都无法接收该请求。

5.11 使用 UML 的扩展机制

UML 提供了丰富的建模概念和表示符号以满足通常的软件开发。但是,用户有时候需要另外的概念或符号表示其特定领域的需求,因此,需要 UML 具有一定的扩充能力。UML 提供了 3 种嵌入的扩充机制:Stereotypes、Constraints 和 Tag values。

(1) Stereotypes 是 UML 中最重要的扩充机制,提供了一种在模型层中加入新的建模元素的方式,可以在 Stereotypes 定义相关的 Constraints 和 Tagged values 以说明特定的语义和特征。定义 Stereotypes 必须满足以下规则。

① Stereotype 名不能与其基类重名,

```
Stereotype.oclAllInstances→forAll(st|st.baseClass <> self.name)
```

② Stereotype 名与它所继承的 Stereotype 名不能重名,

```
Self.allSupertypes→forAll(st:Stereotype|st.name <> self.name)
```

③ Stereotype 名不能与元类命名空间冲突。

④ Stereotype 所定义的 Tag 名不能与其基类元素的元属性命名空间冲突,也不能与它所继承的 Stereotype 的 Tag 名冲突。

(2) 对于模型中的建模元素,Tag Values 允许加入新的属性。Tag 表明了建模元素可扩展的特性的名称,value 可以是任意的值,值的范围取决于用户或工具对 Tag 的解释。对每一个 Tag 名,一个建模元素至多有一个给定的 Tag Values。

```
Self.Taggedvalue→forAll(t1,t2:Tagvalues|t1.tag = t2.tag implies t1 = t2)
```

(3) Constraints 是对建模元素的语义上的限制。

下面,用 UML 的扩充机制描述软件体系结构,同样,还是以 C2 风格为例。

1. C2 的消息

UML 的元类"操作"与 C2 风格的消息概念相对应,UML 的操作包括名字、参数列表和返回值。操作可以是公有的(public)、私有的(private)或受保护的(protected)。为了描述 C2 风格的消息,在请求中增加一个 tag,并且约束操作使之没有返回值,如以下程序所示。

```
Stereotype C2Operation for instances of meta class Operation
 -- 1 --  C2Operations are tagged as either notifications or requests.
c2MsgType : enum { notification, requests }
 -- 2 --  C2Operations are tagged as either incoming or outgoing.
c2MsgDir : enum { in, out }
 -- 3 --  C2messages do not have return values.
Self.parameter -> forAll(p | p.kind <> return)
```

以上程序中的实体类型既包含 tagged values(c2MsgType 和 c2MsgDir),又包含一个通用的对实体类型化的操作的参数约束。

2. C2 的构件

UML 的元类"类"和 C2 风格的构件概念接近。类可以提供多重带有操作(operation)

的接口，可以参与与其他类的交互。但是，也有不太一致的地方，例如，类可以有方法(method)和属性(attribute)，在 UML 中，操作是过程抽象的规格说明，而方法是过程体。C2 中的构件只提供操作，而不提供方法，且这些操作必须是构件提供的接口的一部分，而不是构件本身的一部分。

```
Stereotype C2Interface for instances of meta class Interface
 -- 1 -- A C2Interface has a tagged value identifying its position.
c2pos : enum { top, bottom }
 -- 2 -- All C2Interface operations must have stereotype C2Operation
self.operation -> forAll(o.stereotype = C2Operation)

Stereotype C2Component for instances of meta class Class
 -- 1 -- C2Components must implement exactly two interfaces, which must be
   -- C2Interfaces, one top, and the other bottom.
self.interface -> size = 2 and
self.interface -> forAll(i | i.stereotype = C2Interface) and
self.interface -> exists(i | i.c2pos = top) and
self.interface -> exists(i | i.c2pos = bottom)
 -- 2 -- Requests travel "upward" only, i.e., they are sent through top interface
   -- and received through bottom interfaces.
let topInt = self.interface -> select(i | i.c2pos = top) in
let botInt = self.interface -> select(i | i.c2pos = bottom) in
topInt.operation -> forAll(o | o.c2MsgType = request) implies (o.c2MsgDir =
out)) and
botInt.operation -> forAll(o | o.c2MsgType = request) implies (o.c2MsgDir = in))
 -- 3 -- Notifications travel "downward" only. Similar to the constraint above.
 -- 4 -- Each C2Component has at least one instance in the running system.
self.allInstances -> size >= 1
```

以上程序中的实体类型使用了很多 OCL 特征，C2 Component 的第二个约束定义了附件的属性(topInt 和 botInt)，这些属性用来辅助约束的定义。属性 allInstances 返回所关联的模型元素的所有实例，操作 select 选择关联集合的一个子集，条件是其指定的表达式为真。

3. C2 的连接件

C2 的连接件与构件共享了很多约束，但是，在下面，把构件和连接件当作不同的体系结构组成成分来处理。连接件可以不定义它自己的接口，其接口是由它连接的构件决定的。

与描述 C2 的构件一样，也可以使用实体类型来描述 C2 的连接件，下面重用一些约束，再增加两个新约束。为了把构件的约束引入连接件中，首先定义 3 个构件的附加属性，这些附加属性是决定构件接口所必需的。

```
Stereotype C2AttachOverComp for instances of meta class Association
 -- 1 -- C2attachments are binary associations.
self.associationEnd -> size = 2
 -- 2 -- One end of the attachment must be a single C2Component.
let ends = self.associationEnd in
ends[1].multiplicity.min = 1 and ends[1].multiplicity.max = 1 and
ends[1].class.sterrotype = C2Component
```

```
-- 3 -- The other end of the attachment must be a single C2Connector.
let ends = self.associationEnd in
ends[2].multiplicity.min = 1 and ends[2].multiplicity.max = 1 and
ends[2].class.sterrotype = C2Connector

Stereotype C2AttachUnderComp for instance of meta class Association. Same as
C2AttachOverComp, but with the order reversed.

Stereotype C2AttachConnConn for intstance of meta class Association
-- 1 -- C2 attachments are binaty associations.
self.associationEnd -> size = 2
-- 2 -- Each end of the association must be on a C2 connector.
self.associationEnd -> forAll(ae | ae.multiplicity.min = 1 and
    ae. multiplicity.max = 1 and ae.class.stereotype = C2Connector)
-- 3 -- The two ends are not the same C2Connector.
self.associationEnd[1].class <> self.associationEnd[2].class

Stereotype C2Connector for instance of meta class Class
-- 1 through 3 -- Same as constraints 1 - 3 on C2Component.
-- 4 -- Each C2 connector has exactly one instance in the running system.
self.allInstances -> size = 1
-- 5 -- The top interface of a connector is determined by the components and
  connectors attached to its bottom.
let topInt = self.interface -> select(i | i.c2pos = top) in
let downAttach = self.associationEnd.associatin - > select(a |
    a.associationEnd[2] = self) in
let topsIntsBelow = downAttach.associationEnd[1].interface -> select(i | i.c2pos = top) in
topsInsBelow.operation -> asset = topInt.operation -> asset
-- 6 -- The bottom interface of a connector is determined by the components and
connectors attached to its top. This is similar to the constraint above.
```

以上程序中的实体类型使用了关联端的 multiplicity 属性,因为在 Association 和 AssociationEnd 之间的元级关联是有序的(ordered),所以 AssociationEnd 是一个序列而不是一个集合。但对 UML 来说,这种有序没有语义表示。

4. C2 体系结构

C2 体系结构由构件和连接件组成,构件在 top 端可以与一个连接件关联,在 bottom 端与一个连接件关联,而一个连接件的 top 端或 bottom 端可以与任意数量的其他连接件的 top 端或 bottom 端关联。

```
Stereotype C2Architecture for instances of meta class Model
-- 1 -- The classes in a C2Architecture must all be C2 model elements.
Self.modelElement -> select(me | me.oclIsKingdof(Class)) -> forAll(c |
    c.stereotype = C2Compnent or c.stereotype = c2Connector)
-- 2 -- The associations in a C2Architecture must all be C2 model elements.
Self.modelElement -> select(me | me.oclIskindof(Association)) -> forAll(a |
    a.stereotype = C2AttachOverComp or
    a.stereotype = C2AttachUnderComp or
    a.stereotype = C2AttachConnConn)
```

```
-- 3 -- Each C2Component has at most one C2AttachOverComp.
let comps = self.modelElement -> select(me | me.stereotype = C2Component) in
comps -> forAll(c | c.associationEnd.association -> select(a | a.stereotype =
    C2AttachOverComp) -> size <= 1)
-- 4 -- Each C2Component has at most one C2AttachUnderComp. Similar to the
  constraint above.
-- 5—C2Connectors do not participater in any non-C2 associations.
let cons = self.modelElement -> select (me | me.stereotype = C2Connector) in
cons.associationEnd.association -> forAll(a | a.stereotype = C2AttachOverComp
or
a.stereotype = C2AttachUnderComp or
a.stereotype = C2AttachConnConn)
-- 6 -- C2Components do not participate in any non-C2 associations. Similar to
    the constraint above, but without the third disjunct.
-- 7 -- Each C2Connector must be attached to some connector or component.
let cons = self.modelElement -> select(e | e.stereotype = C2Connector) in
cons -> forAll(c | c.associationEnd -> size > 0)
-- 8 -- Each C2Component must be attached to some connector. Similar to the
constraint above.
```

以上程序的实体类型中 oclIsKingof 操作是所关联的模型实例中的元模型类的断言，当这个实例属于所指定的类或其子类时，断言为真。

思 考 题

1. 在某银行业务的用例模型中，“取款”用例需要等到“存款”用例执行之后才能执行，两个用例之间的关系什么？“取款”和“存款”两个用例中都需要执行查询余额的功能，将查询余额提取成独立的用例，那么“取款”和“存款”用例与“查询余额”用例之间的关系是什么？

2. 希赛图书订单处理系统中，“创建新订单”和“更新订单”两个用例都需要检查客户的账号是否正确，为此定义一个通用的用例“核查客户账户”。用例“创建新订单”和“更新订单”与用例“核查客户账户”之间是什么关系？

3. 如果一个并发的状态由 n 个并发的子状态图组成，那么，该并发状态在某时刻的状态由多少个子状态图中各取一个状态组合而成？

4. 用例从用户角度描述系统的行为。用例之间可以存在一定的关系。在希赛教育图书馆管理系统用例模型中，所有用户使用系统之前必须通过“身份验证”，“身份验证”可以有“密码验证”和“智能卡验证”两种方式，则“身份验证”与“密码验证”和“智能卡验证”之间是什么关系？

5. 软件体系结构可以通过 UML 直接进行描述，请说明 UML 包括哪些图，以及各自的作用是什么。

6. 在 UML 的众多图形中，最常用的图形是哪一个？在为同一个软件系统建模的过程中，是否需要将这些图形都要画出来？为什么？

7. 如何使用 4 层元模型描述体系结构？

8. 请选择一个小型的软件系统，并用 UML 为该系统建模。

参考文献