第三章标准建模语言 UML

3.1 简述

面向对象的分析与设计(OOA&D)方法的发展在 80 年代末至 90 年代中出现了一个高潮,UML 是这个高潮的产物。它不仅统一了 Booch、Rumbaugh 和 Jacobson 的表示方法,而且对其作了进一步的发展,并最终统一为大众所接受的标准建模语言。

公认的面向对象建模语言出现于 70 年代中期。从 1989 年到 1994 年,其数量从不到十种增加到了五十多种。在众多的建模语言中,语言的创造者努力推崇自己的产品,并在实践中不断完善。但是,OO 方法的用户并不了解不同建模语言的优缺点及相互之间的差异,因而很难根据应用特点选择合适的建模语言,于是爆发了一场"方法大战"。90 年代中,一批新方法出现了,其中最引人注目的是 Booch 1993、OOSE 和 OMT-2 等。

Booch 是面向对象方法最早的倡导者之一,他提出了面向对象软件工程的概念。1991 年,他将以前面向 Ada 的工作扩展到整个面向对象设计领域。Booch 1993 比较适合于系统的设计和构造。Rumbaugh 等人提出了面向对象的建模技术(OMT)方法,采用了面向对象的概念,并引入各种独立于语言的表示符。这种方法用对象模型、动态模型、功能模型,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(Use-Case),并在用例的描述中引入了外部角色的概念。用例的概念是精确描述需求的重要武器,用例贯穿于整个开发过程,包括对系统的测试和验证。OOSE 比较适合支持商业工程和需求分析。此外,还有 Coad/Yourdon 方法,即著名的 OOA/OOD,它是最早的面向对象的分析和设计方法之一。该方法简单、易学,适合于面向对象技术的初学者使用,但由于该方法在处理能力方面的局限,目前已很少使用。

概括起来,首先,面对众多的建模语言,用户由于没有能力区别不同语言之间的差别,因此很难找到一种比较适合其应用特点的语言;其次,众多的建模语言实际上各有千秋;第三,虽然不同的建模语言大多类同,但仍存在某些细微的差别,极大地妨碍了用户之间的交流。因此在客观上,极有必要在精心比较不同的建模语言优缺点及总结面向对象技术应用实践的基础

上,组织联合设计小组,根据应用需求,取其精华,去其糟粕,求同存异,统一建模语言。

1994 年 10 月,Grady Booch 和 Jim Rumbaugh 开始致力于这一工作。他们首先将 Booch93 和 OMT-2 统一起来,并于 1995 年 10 月发布了第一个公开版本,称之为统一方法 UM 0.8(Unitied Method)。1995 年秋,OOSE 的创始人 Ivar Jacobson 加盟到这一工作。经过 Booch、Rumbaugh 和 Jacobson 三人的共同努力,于 1996 年 6 月和 10 月分别发布了两个新的版本,即 UML 0.9 和 UML 0.91,并将 UM 重新命名为 UML(Unified Modeling Language)。1996 年,一些机构将 UML 作为其商业策略已日趋明显。UML 的开发者得到了来自公众的正面反应,并倡议成立了 UML 成员协会,以完善、加强和促进 UML 的定义工作。当时的成员有 DEC、HP、I-Logix、 Itellicorp、 IBM、ICON Computing、MCI Systemhouse、Micr osoft、Oracle、Rational Software、TI 以及 Unisys。这一机构对 UML 1.0(1997 年 1 月)及 UML 1.1(1997 年 11 月 17 日)的定义和发布起了重要的促进作用。

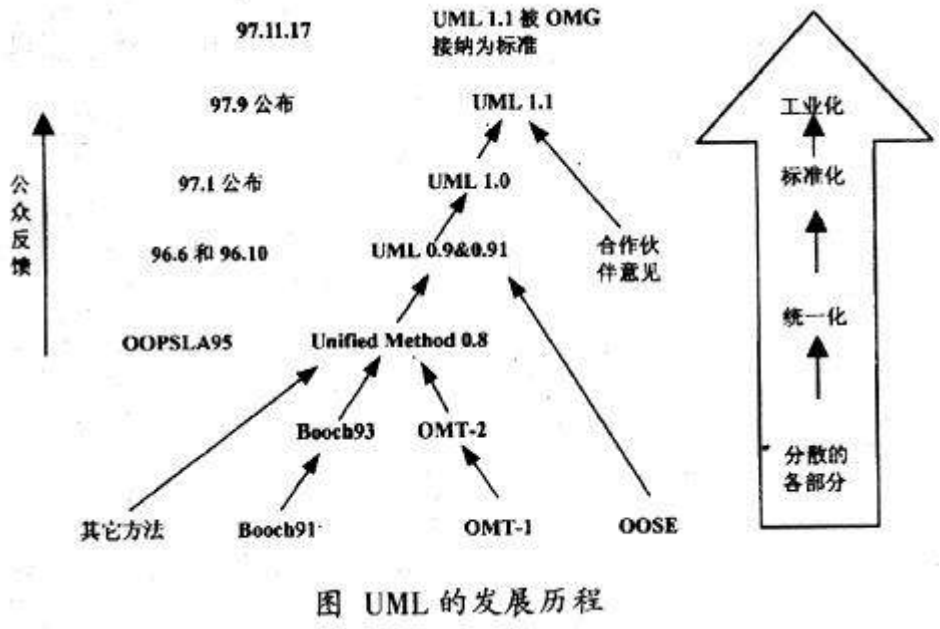


图 UML 的发展历程

图 1 UML 的发展历程

面向对象技术和 UML 的发展过程可用上图来表示,标准建模语言的出现是其重要成果。在美国,截止 1996 年 10 月,UML 获得了工业界、科技界和应用界的广泛支持,已有 700 多个公司表示支持采用 UML 作为建模语言。1996 年底,UML 已稳占面向对象技术市场的

85%,成为可视化建模语言事实上的工业标准。1997 年 11 月 17 日,OMG 采纳 UML 1.1 作为基于面向对象技术的标准建模语言。UML 代表了面向对象方法的软件开发技术的发展方向,具有巨大的市场前景,也具有重大的经济价值和国防价值。

首先,UML 融合了 Booch、OMT 和 OOSE 方法中的基本概念,而且这些基本概念与其他面向对象技术中的基本概念大多相同,因而,UML 必然成为这些方法以及其他方法的使用者乐于采用的一种简单一致的建模语言;其次,UML 不仅仅是上述方法的简单汇合,而是在这些方法的基础上广泛征求意见,集众家之长,几经修改而完成的,UML 扩展了现有方法的应用范围;第三,UML 是标准的建模语言,而不是标准的开发过程。尽管 UML 的应用必然以系统的开发过程为背景,但由于不同的组织和不同的应用领域,需要采取不同的开发过程。

作为一种建模语言,UML 的定义包括 UML 语义和 UML 表示法两个部分。

统一建模语言（UML）是一个通用的可视化建模语言，用于对软件进行描述、可视化处理、构造和建立软件系统制品的文档。它记录了对必须构造的系统的决定和理解，可用于对系统的理解、设计、浏览、配置、维护和信息控制。UML 适用于各种软件开发方法、软件生命周期的各个阶段、各种应用领域以及各种开发工具，UML 是一种总结了以往建模技术的经验并吸收当今优秀成果的标准建模方法。UML 包括概念的语义，表示法和说明，提供了静态、动态、系统环境及组织结构的模型。它可被交互的可视化建模工具所支持，这些工具提供了代码生成器和报表生成器。UML 标准并没有定义一种标准的开发过程，但它适用于迭代式的开发过程。它是为支持大部分现存的面向对象开发过程而设计的。

UML 描述了一个系统的静态结构和动态行为。UML 将系统描述为一些离散的相互作用的对象并最终为外部用户提供一定的功能的模型结构。静态结构定义了系统中的重要对象的属性和操作以及这些对象之间的相互关系。动态行为定义了对象的时间特性和对象为完成目标而相互进行通信的机制。从不同但相互联系的角度对系统建立的模型可用于不同的目的。UML 还包括可将模型分解成包的结构成分，以便于软件小组将大的系统分解成易于处理的块结构，并理解和控制各个包之间的依赖关系，在复杂的开发环境中管理模型单元。它还包括用于显示系统实现和组织运行的成分。UML 不是一门程序设计语言。但可以使用代码生成器工具将 UML 模型转换为多种程序设计语言代码，或使用反向生成器工具将程序源代码转换为 UML。UML 不是一种可用于定理证明的高度形式化的语言，这样的语言有很多种，但它们通用性较差，不易理解和使用。UML 是一种通用建模语言。对于一些专门领域，例如用户图形界面（GUI）设计、超大规模集成电路（VLSI）设计、基于规则的人工智能领域，

使用专门的语言和工具可能会更适合些。UML 是一种离散的建模语言，不适合对诸如工程和物理学领域中的连续系统建模。它是一个综合的通用建模语言，适合对诸如由计算机软件、固件或数字逻辑构成的离散系统建模。

3.2 UML 概念域

UML 的概念和模型可以分成以下几个概念域。

3.2.1 静态结构

任何一个精确的模型必须首先定义所涉及的范围，即确定有关应用、内部特性及其相互关系的关键概念。UML 的静态成分称为静态视图。静态视图用类构造模型来表达应用，每个类由一组包含信息和实现行为的离散对象组成。对象包含的信息被作为属性，它们执行的行为被作为操作。多个类通过概括化处理可以具有一些共同的结构。子类在继承它们共同的超类的结构和行为的基础上增加了新的结构和行为。对象与其他对象之间也具有运行时间连接，这种对象与对象之间的关系被称为类间的关联。一些元素通过依赖关系组织在一起，这些依赖关系包括在抽象级上进行模型转换、模板参数的捆绑、授予许可以及通过一种元素使用另一种元素等。静态视图主要使用类图。静态视图可用于生成程序中用到的大多数数据结构声明。在 UML 视图中还要用到其他类型的元素，比如接口、数据类型、用例和信号等，这些元素统称为类元，它们的行为很像在每种类元上具有一定限制的类。

3.2.2 动态行为

有两种方式对行为建模。一种是根据一个对象与外界发生关系的生命历史；另一种是一系列相关对象之间当它们相互作用实现行为时的通信方式。孤立对象的视图是状态机—当对象基于当前状态对事件产生反应，执行作为反应的一部分的动作，并从一种状态转换到另一种状态时的视图。状态机模型用状态图来描述。相互作用对象的系统视图是一种协作，一种与语境有关的对象视图和相互之间的链，通过数据链对象间存在着消息流。视图将数据结构、控制流和数据流在一个视图中统一起来。协作和互操作用顺序图和协作图来描述。对所有行为视图起指导作用的是一组用例，每一个用例描述了一个用例执行者或系统外部用户可见的一个功能。

3.2.3 实现构造

UML 模型既可用于逻辑分析又可用于物理实现。某些成分代表了实现。构件是系统中物理上的可替换的部分，它按照一组接口来设计并实现。它可以方便地被一个具有同样规格说明的构件替换。节点是运行时间计算资源，资源定义了一个位置。它包括构件和对象。部署图描述了在一个实际运行的系统中节点上的资源配置和构件的排列以及构件包括的对象，并包括节点间内容的可能迁移。

3.2.4 模型组织

计算机能够处理大型的单调的模型，但人力不行。对于一个大型系统，建模信息必须被划分成连贯的部分，以便工作小组能够同时工作在不同部分上。即使是一个小系统，人的理解能力也要求将整个模型的内容组织成一个个适当大小的包。包是 UML 模型通用的层次组织单元，它们可以用于存储、访问控制、置以管理配及构造包含可重用的模型单元库。包之间的依赖关系是对包的组成部分之间的依赖关系的归纳。系统整个构架可以在包之间施加依赖关系。因此，包的内容必须符合包的依赖关系和有关的构架要求。

3.2.5 扩展机制

无论一种语言能够提供多么完善的机制，人们总是想扩展它的功能。我们已使 UML 具有一定的扩展能力，相信能够满足大多数对 UML 扩充的需求而不改变语言的基础部分。构造型是一种新的模型元素，与现有的模型元素具有相同的结构，但是加上了一些附加限制，具有新的解释和图标。代码生成器和其他的工具对它的处理过程也发生了变化。标记值是一对任意的标记值字符串，能够被连接到任何一种模型元素上并代表任何信息，如项目管理信息、代码生成指示信息和构造型所需要的值。标记和值用字符串代表。约束是用某种特定语言（如程序设计语言）的文本字符串表达的条件专用语言或自然语言。UML 提供了一个表达约束的语言，名为 OCL。与所有其他扩展机制一样，必须小心使用这些扩展机制，因为有可能形成一些别人无法理解的方言。但这些机制可以避免语言基础发生根本性变化。

3.3 UML 视图

UML 中的各种成分和概念之间没有明显的划分界限，但为方便起见，我们用视图来划

分这些概念和组件。视图只是表达系统某一方面特征的 UML 建模组件的子集。视图的划分带有一定的随意性，但我们希望这种看法仅仅是直觉上的。在每一类视图中使用一种或两种特定的图来可视化地表示视图中的各种概念。

在最上一层，视图被划分成三个视图域：结构分类、动态行为和模型管理。

结构分类描述了系统中的结构成员及其相互关系。类元包括类、用例、构件和节点。类元为研究系统动态行为奠定了基础。类元视图包括静态视图、用例视图和实现视图。

动态行为描述了系统随时间变化的行为。行为用从静态视图中抽取的瞬间值的变化来描述。动态行为视图包括状态机视图、活动视图和交互视图。

模型管理说明了模型的分层组织结构。包是模型的基本组织单元。特殊的包还包括模型和子系统。模型管理视图跨越了其他视图并根据系统开发和配置组织这些视图。

UML 还包括多种具有扩展能力的成分，这些扩展能力有限但很有用。这些成分包括约束、构造型和标记值，它们适用于所有的视图元素。

表 3-1 列出了 UML 的视图和视图所包括的图以及与每种图有关的主要概念。不能把这张表看成是一套死板的规则，应将其视为对 UML 常规使用方法的指导，因为 UML 允许使用混合视图。

表 3-1 UML 视图和图

主要的域	视 图	图	主要概念
结构	静态视图	类图	类、关联、泛化、依赖关系、实现、接口
	用例视图	用例图	用例、执行者、关联、扩展、包括、用例泛化
	实现视图	构件图	构件、接口、依赖关系、实现
	部署视图	部署图	节点、构件、依赖关系、实现
动态	状态机视图	状态机图	状态、事件、转换、动作
	活动视图	活动图	状态、活动、完成转换、分叉、结合
	交互视图	顺序图	交互、对象、消息、激活
		协作图	协作、交互、协作角色、消息
模型管理	模型管理视图	类图	包、子系统、模型
可扩展性	所有	所有	约束、构造图、标记值

3.3.1 静态视图

静态视图对应用领域中的概念以及与系统实现有关的内部概念建模。这种视图之所以被称之为是静态的是因为它不描述与时间有关的系统行为，此种行为在其他视图中进行描述。静态视图主要是由类及类间相互关系构成，这些相互关系包括：关联、概括和各种依赖关系。

一个类是应用领域或应用解决方案中概念的描述。类图是以类为中心来组织的，类图中的其他元素或属于某个类或与类相关联。静态视图用类图来实现，正因为它以类为中心，所以称其为类图。

在类图中类用矩形框来表示，它的属性和操作分别列在分格中。如不需要表达详细信息时，分格可以省略。一个类可能出现在好几个图中。同一个类的属性和操作只在一种图中列出，在其他图中可省略。

关系用类框之间的连线来表示，不同的关系用连线上和连线端头处的修饰符来区别。

图 3-1 是售票系统的类图，它只是售票系统领域模型的一部分。图中表示了几个重要的类，如 Customer、Reservation、Ticket 和 Performance。顾客可多次订票，但每一次订票只能由一个顾客来执行。有两种订票方式：个人票或套票；前者只是一张票，后者包括多张票。每一张票不是个人票就是套票中的一张，但是不能又是个人票又是套票中的一张。每场演出都有多张票可供预定，每张票对应一个唯一的座位号。每次演出用剧目名、日期和时间来标识。

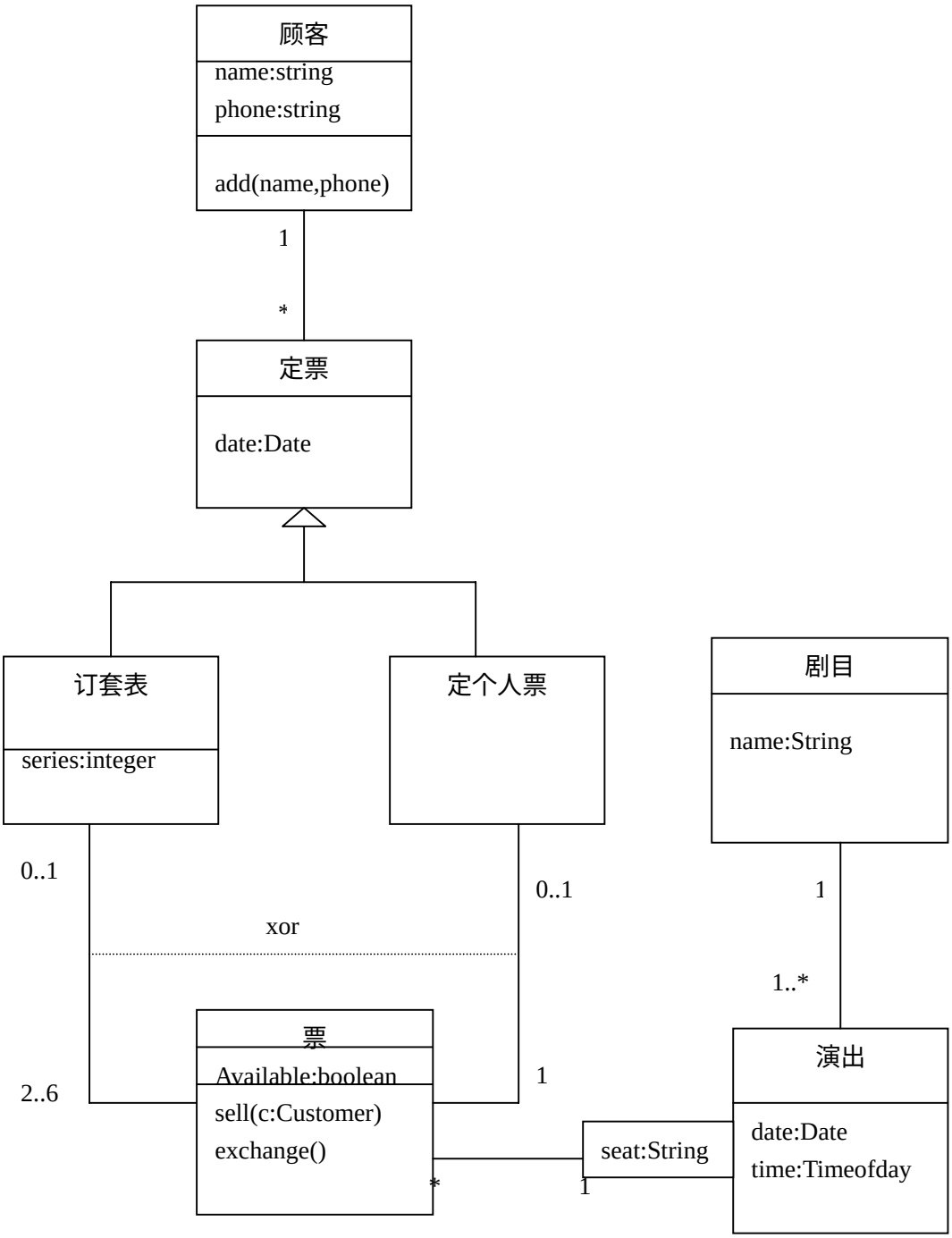


图 3-1 售票系统类图

3.3.2 用例视图

用例视图是被称为执行者的外部用户所能观察到的系统功能的模型图。用例是系统中的一个功能单元，可以被描述为执行者与系统之间的一次交互作用。用例模型的用途是列出系

统中的用例和执行者，并显示哪个执行者参与了哪个用例的执行。

图 3-2 是售票系统的用例图。执行者包括售票员、监督员和公用电话亭。公用电话亭是另一个系统，它接受顾客的订票请求。在售票处的应用模型中，顾客不是执行者者，因为顾客不直接与售票处打交道。用例包括通过公用电话亭或售票员购票，购预约票（只能通过售票员），售票监督（应监督员的要求）。购票和预约票包括一个共同的部分—即通过信用卡来付钱（对售票系统的完整描述还要包括其他一些用例，例如换票和验票等）。

用例也可以有不同的层次。用例可以用其他更简单的用例进行说明。在交互视图中，用例做为交互图中的一次协作来实现。

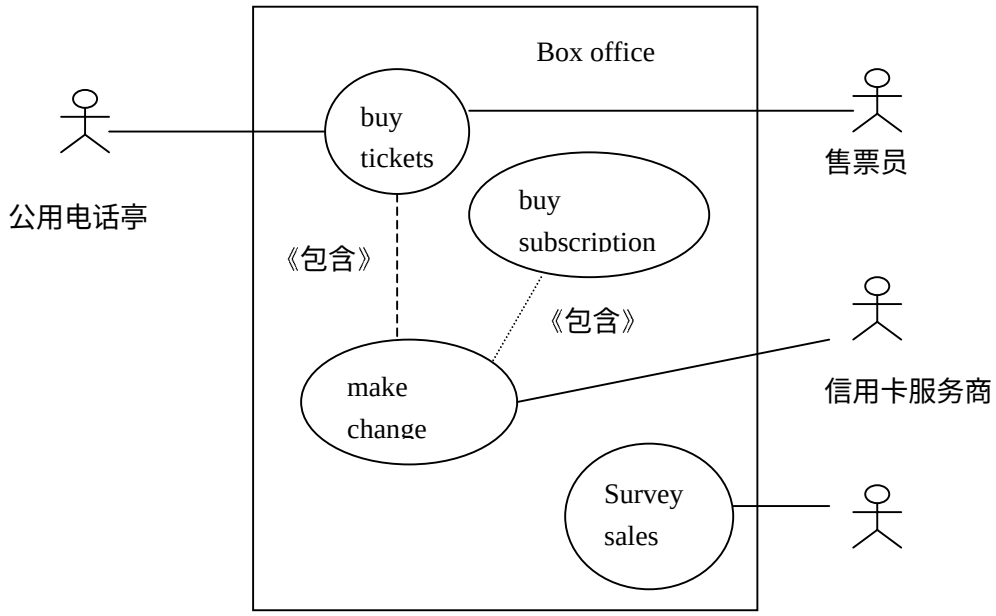


图 3-2 用例图

3.3.3 交互视图

交互视图描述了执行系统功能的各个角色之间相互传递消息的顺序关系。类元是对在系统内交互关系中起特定作用的一个对象的描述，这使它区别于同类的其他对象。交互视图显示了跨越多个对象的系统控制流程。交互视图可用两种图来表示：顺序图和协作图，它们各有不同的侧重点。

1 顺序图

顺序图表示了对象之间传送消息的时间顺序。每一个类元角色用一条生命线来表示

—即用垂直线代表整个交互过程中对象的生命期。生命线之间的箭头连线代表消息。顺序图可以用来进行一个场景说明—即一个事务的历史过程。

顺序图的一个用途是用来表示用例中的行为顺序。当执行一个用例行为时，顺序图中的每条消息对应了一个类操作或状态机中引起转换的触发事件。

图 3-3 是描述购票这个用例的顺序图。顾客在公共电话亭与售票处通话触发了这个用例的执行。顺序图中付款这个用例包括售票处与公用电话亭和信用卡服务处的两个通信过程。这个顺序图用于系统开发初期，未包括完整的与用户之间的接口信息。例如，座位是怎样排列的；对各类座位的详细说明都还没有确定。尽管如此，交互过程中最基本的通信已经在这个用例的顺序图中表达出来了。

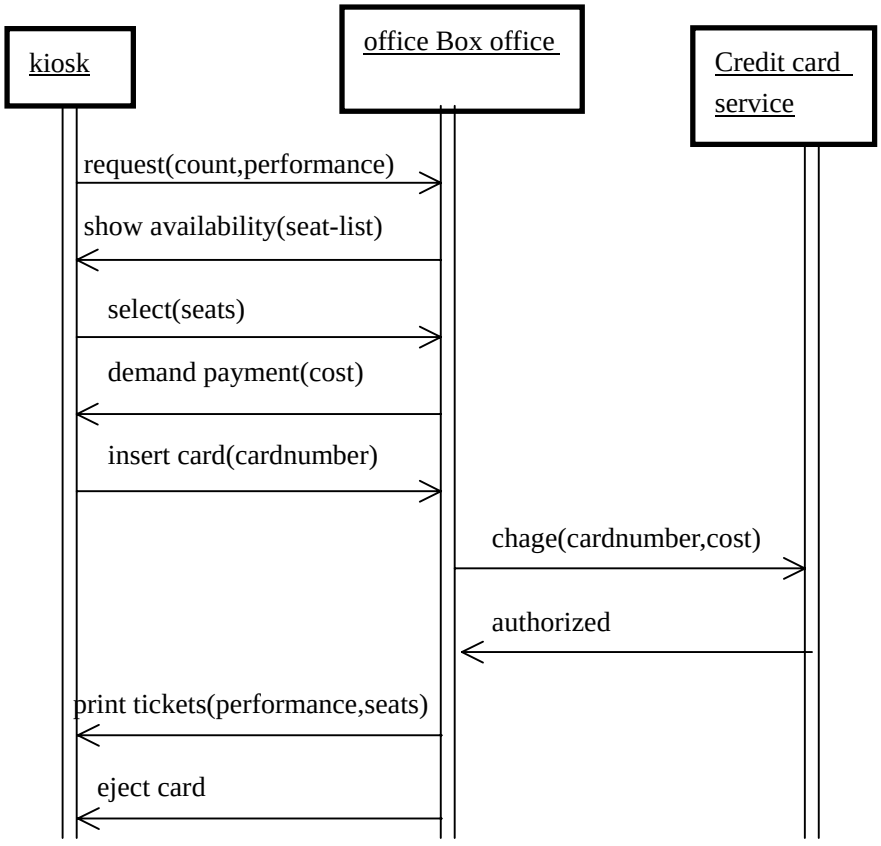


图 3-3 顺序图

2 协作图

协作图对在一次交互中有意义的对象和对象间的链建模。对象和关系只有在交互的才有意义。类元角色描述了一个对象，关联角色描述了协作关系中的一个链。协作图用几何排列来表示交互作用中的各角色（如图 3-4）。附在类元角色上的箭头代表消息。消息的发生顺序用消息箭头处的编号来说明。

协作图的一个用途是表示一个类操作的实现。协作图可以说明类操作中用到的参数和局部变量以及操作中的永久链。当实现一个行为时，消息编号对应了程序中嵌套调用结构和信号传递过程。

图 3-4 是开发过程后期订票交互的协作图。这个图表示了订票涉及的各个对象间的交互关系。请求从公用电话亭发出，要求从所有的演出中查找某次演出的资料。返回给 ticketseller 对象的指针 db 代表了与某次演出资料的局部暂时链接，这个链接在交互过程中保持，交互结束时丢弃。售票方准备了许多演出的票；顾客在各种价位做一次选择，锁定所选座位，售票员将顾客的选择返回给公用电话亭。当顾客在座位表中做出选择后，所选座位被声明，其余座位解锁。

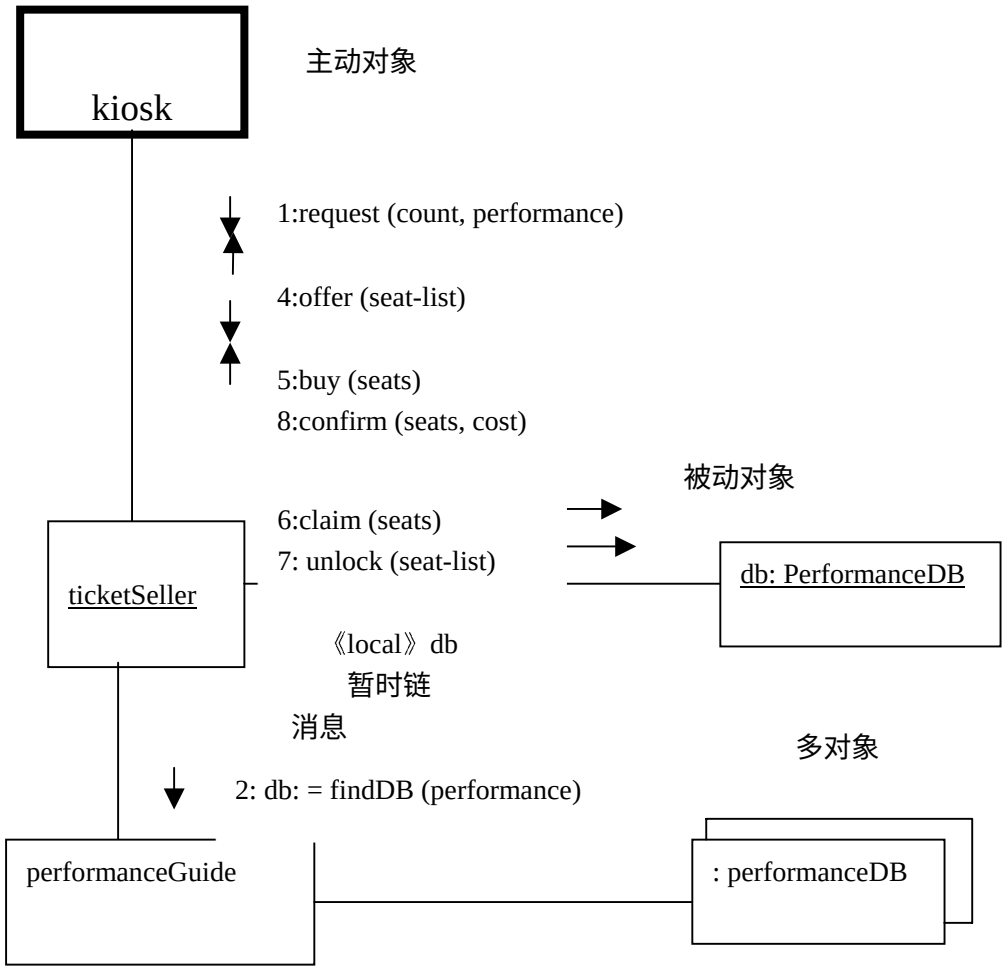


图 3-4 协作图

顺序图和协作图都可以表示各对象间的交互关系，但它们的侧重点不同。顺序图用消息的几何排列关系来表达消息的时间顺序，各角色之间的相关关系是隐含的。协作图用各个角色的几何排列图形来表示角色之间的关系，并用消息来说明这些关系。在实际中可以根据需要选用这两种图。

3.3.4 状态机视图

状态机视图是一个类对象所可能经历的所有历程的模型图。状态机由对象的各个状态和连接这些状态的转换组成。每个状态对一个对象在其生命期中满足某种条件的一个时间段建模。当一个事件发生时，它会触发状态间的转换，导致对象从一种状态转化到另一新的状态。与转换相关的活动执行时，转换也同时发生。状态机用状态图来表达。

图 3-5 是票这一对象的状态图。初始状态是 Available 状态。在票开始对外出售前，一部分票是给预约者预留的。当顾客预定票，被预定的票首先处于锁定状态，此时顾客仍有是否确实要买这张票的选择权，故这张要票可能出售给顾客也可能因为顾客不要这张票而解除锁定状态。如果超过了指定的期限顾客仍未做出选择，此票被自动解除锁定状态。预约者也可以换其他演出的票，如果这样的话，最初预约票也可以对外出售。

状态图可用于描述用户接口、设备控制器和其他具有反馈的子系统。它还可用于描述在生命期中跨越多个不同性质阶段的被动对象的行为，在每一阶段该对象都有自己特殊的行为。

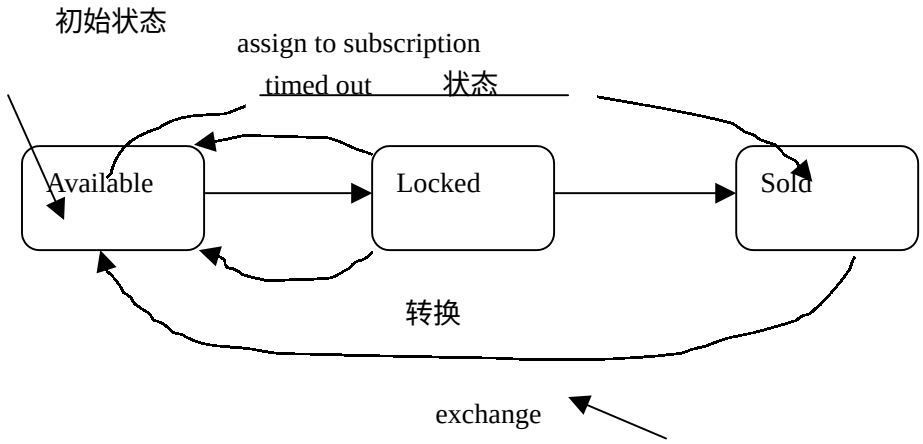


图 3-5 状态图

3.3.5 活动视图

活动图是状态机的一个变体，用来描述执行算法的工作流程中涉及的活动。活动状态代表了一个活动：一个工作流步骤或一个操作的执行。活动图描述了一组顺序的或并

发的活动。活动视图用活动图来体现。

图 3-6 是售票处的活动图。它表示了上演一个剧目所要进行的活动（这个例子仅供参考，不必太认真地凭着看戏的经验而把问题复杂化）。箭头说明活动间的顺序依赖关系—例如，在规划进度前，首先要选择演出的剧目。加粗的横线段表示分叉和结合控制。例如，安排好整个剧目的进度后，可以进行宣传报道、购买剧本、雇用演员、准备道具、设计照明、加工戏服等，所有这些活动都可同时进行。在进行彩排之前，剧本和演员必须已经具备。

这个例说明了活动图的用途是对人类组织的现实世界中的工作流程建模。对事物建模是活动图的主要用途，但活动图也可对软件系统中的活动建模。活动图有助于理解系统高层活动的执行行为，而不涉及建立协作图所必须的消息传送细节。

用连接活动和对象流状态的关系流表示活动所需的输入输出参数。

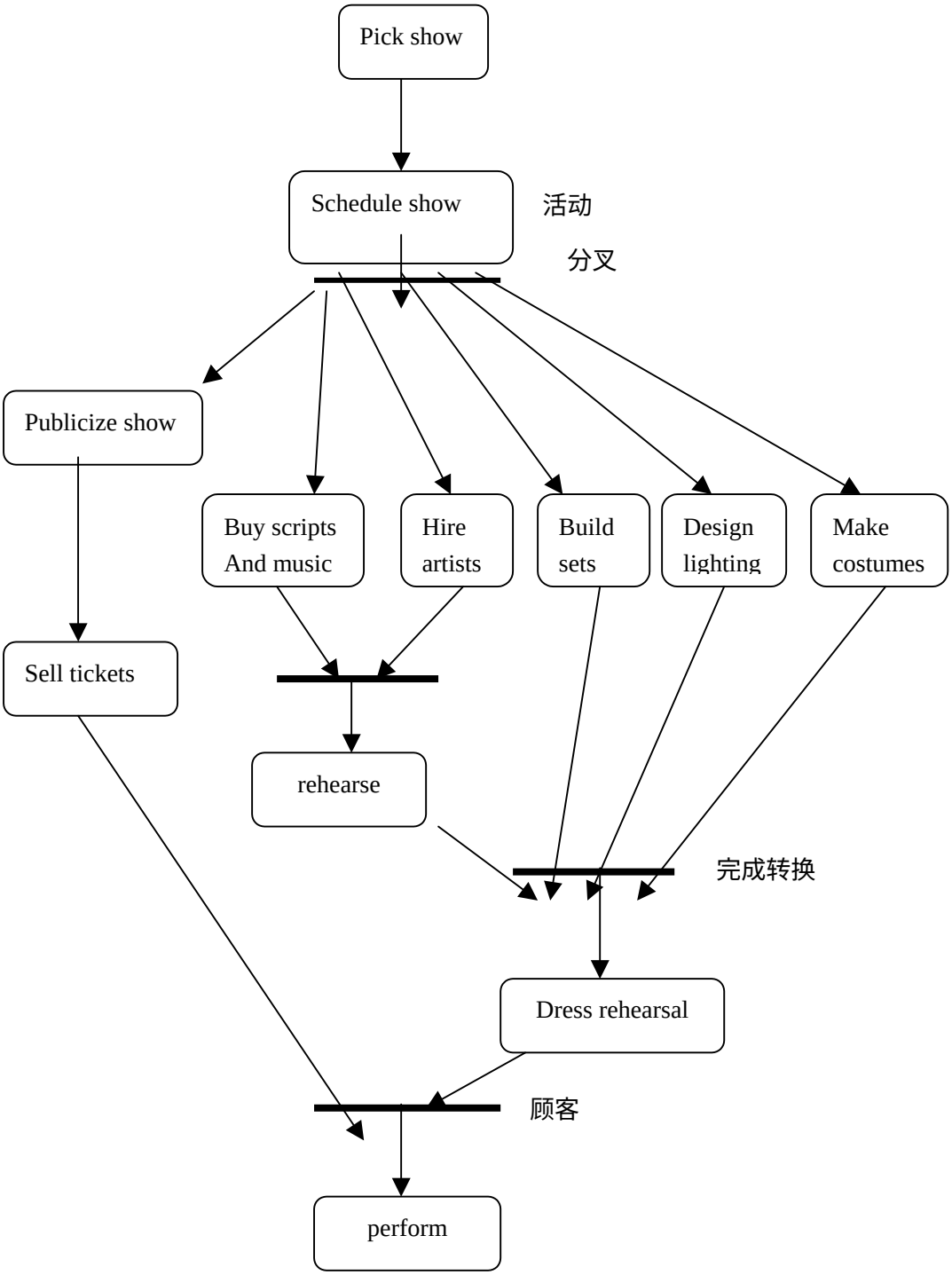


图 3-6 活动图

3.3.6 物理视图

前面介绍的视图模型按照逻辑观点对应用领域中的概念建模。物理视图对应用自身的实现结构建模，例如系统的构件组织和建立在运行节点上的配置。这类视图提供了将

系统中的类映射成物理构件和节点的机制。物理视图有两种：实现视图和部署视图。

实现视图为系统的构件建模型—构件即构造应用的软件单元—还包括各构件之间的依赖关系，以便通过这些依赖关系来估计对系统构件的修改给系统可能带来的影响。

实现视图用构件图来表现。图 3-7 是售票系统的构件图。图中有三个用户接口：顾客和公用电话亭之间的接口、售票员与在线订票系统之间的接口和监督员查询售票情况的接口。售票方构件顺序接受来自售票员和公用电话亭的请求；信用卡主管构件之间处理信用卡付款；还有一个存储票信息的数据库构件。构件图表示了系统中的各种构件。在个别系统的实际物理配置中，可能有某个构件的多个备份。

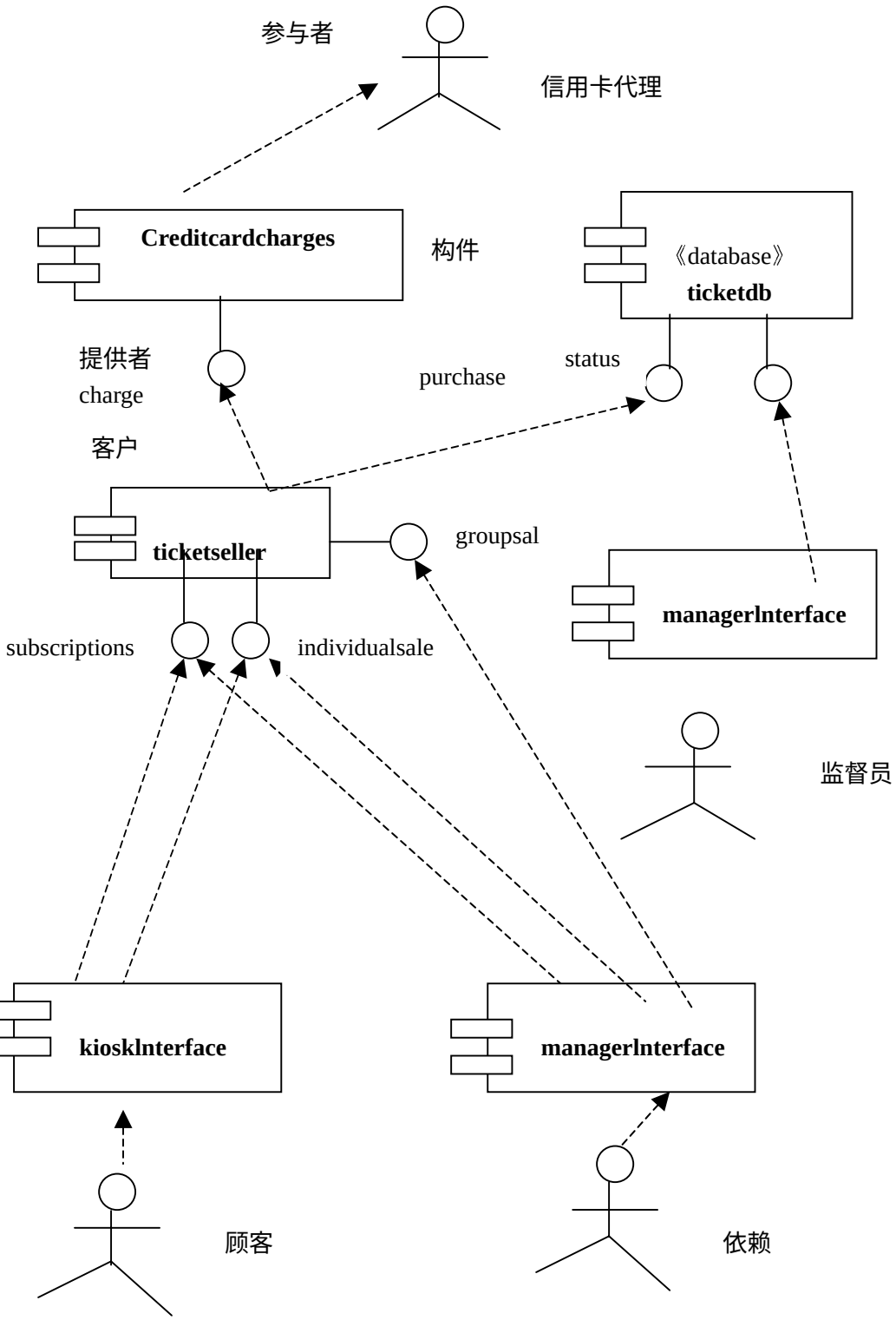


图 3-7 构件图

图中的小圆圈代表接口，即服务的连贯集。从构件到接口的实线表明该构件提供的列在接口旁的服务。从构件到接口的虚线箭头说明这个构件要求接口提供的服务。例如，购买个人票可以通过公用电话亭订购也可直接向售票员购买，但购买团体票只能通过售票员。

部署视图描述位于节点实例上的运行构件实例的安排。节点是一组运行资源，如计算机、设备或存储器。这个视图允许评估分配结果和资源分配。

部署视图用部署图来表达。图 3-8 是售票系统的描述层部署图。图中表示了系统中的各构件和每个节点包含的构件。节点用立方体图形表示。

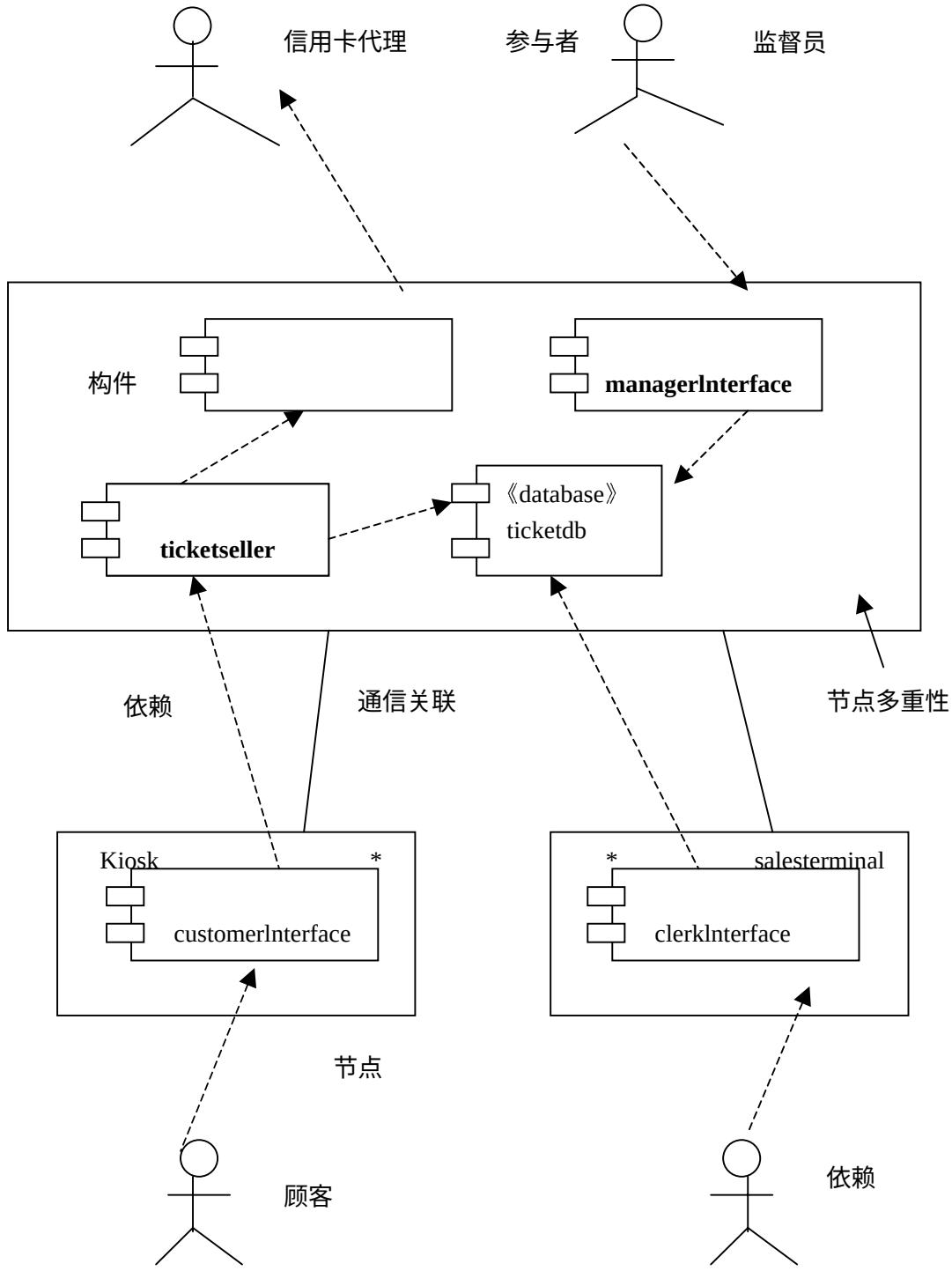


图 3-8 部署图（描述层）

图 3-9 是售票系统的实例层部署图。图中显示了各节点和它们之间的连接。这个模型

中的信息是与图 3-8 的描述层中的内容相互对应的。

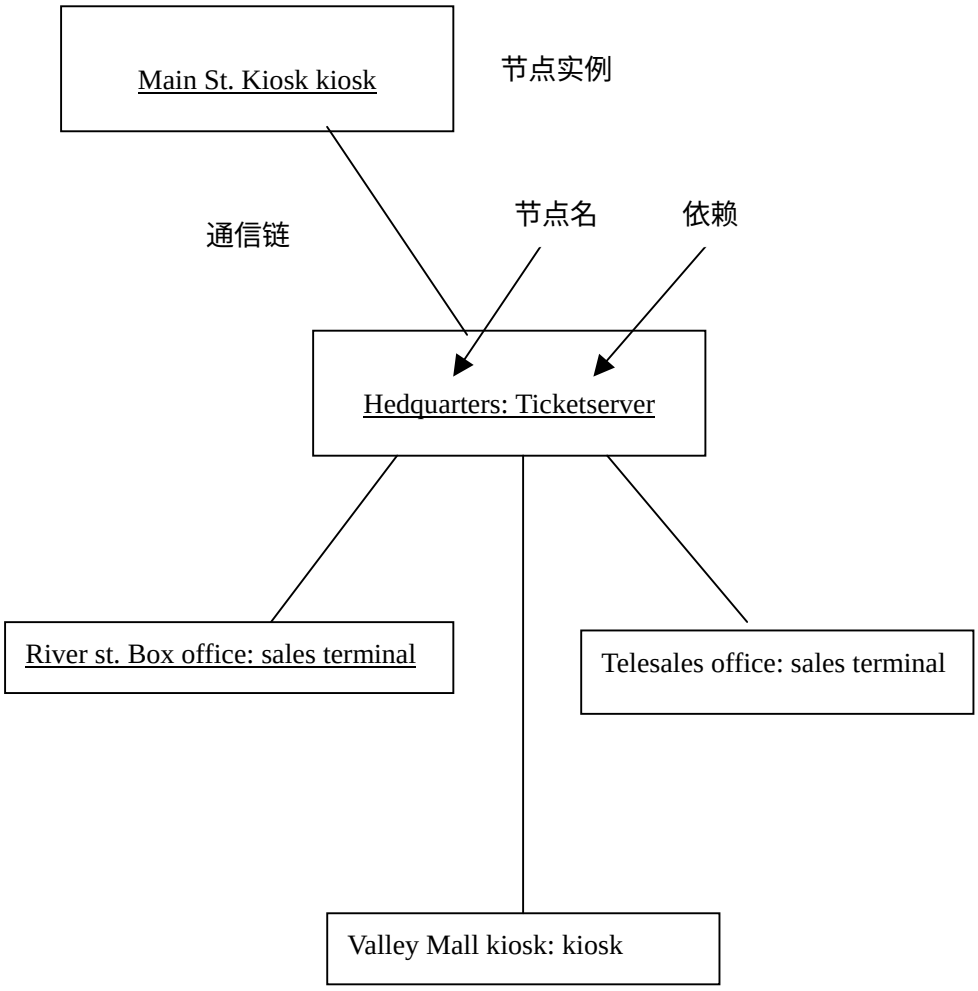


图 3-9 部署图（实例层）

3.3.7 模型管理视图

模型管理视图对模型自身组织建模。一系列由模型元素（如类、状态机和用例）构成的包组成了模型。一个包(package)可能包含其他的包，因此，整个模型实际上可看成一个根包，它间接包含了模型中的所有内容。包是操作模型内容、存取控制和配置控制的基本单元。每一个模型元素包含于包中或包含于其他模型元素中。

模型是从某一观点以一定的精确程度对系统所进行的完整描述。从不同的视角出发，对同一系统可能会建立多个模型，例如有系统分析模型和系统设计模型之分。模型是一种特殊的包。

子系统是另一种特殊的包。它代表了系统的一个部分，它有清晰的接口，这个接口

可作为一个单独的构件来实现。

模型管理信息通常在类图中表达。

图 3-10 显示了将整个剧院系统分解所得到的包和它们之间的依赖关系。售票处子系统在前面的例子中已经讨论过了，完整的系统还包括剧院管理和计划子系统。每个子系统还包含了多个包。

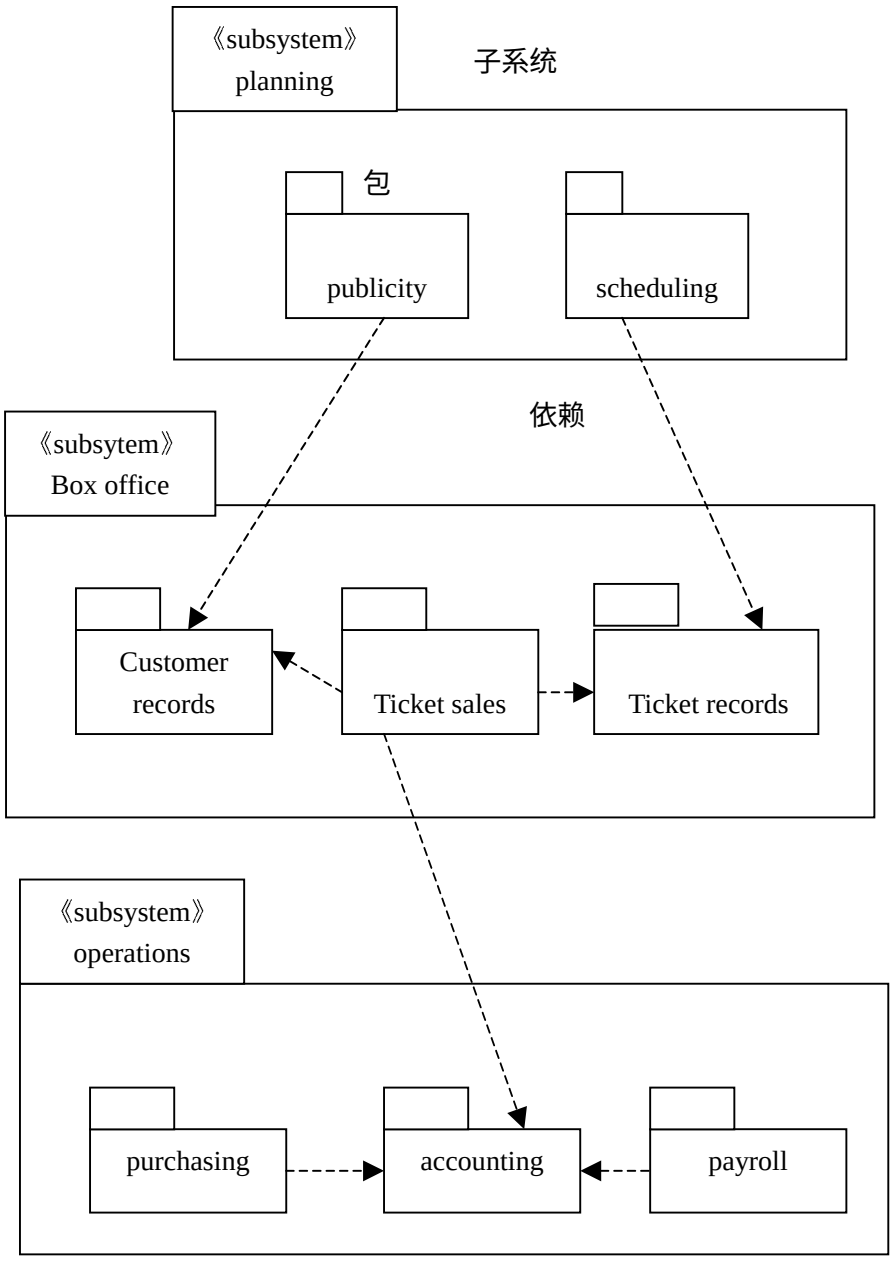


图 3-10 包

3.3.8 扩展组件

UML 包含三种主要的扩展组件：约束、构造型和标记值。约束是用某种形式化语言或自然语言表达的语义关系的文字说明。构造型是由建模者设计的新的模型元素，但是这个模型元素的设计要建立在 UML 已定义的模型元素基础上。标记值是附加到任何模型元素上的命名的信息块。

这些组件提供了扩展 UML 模型元素语义的方法，同时不改变 UML 定义的元模型自身的

语义。使用这些扩展组件可以组建适用于某一具体应用领域的 UML 用户定制版本。

图 3-1 举例说明了约束、构造型，和标记值的使用。对剧目类的约束保证了剧目具有唯一的名称。图 3-1 说明了两个关联的异或约束，一个对象某一时刻只能具有两个关联中的一个。用文字表达约束效果较好，但 UML 的概念不直接支持文字描述。

TicketdDB 构件构造型表明这个是一个数据库构件，允许省略该构件的接口说明，因为这个接口是所有数据库都支持的通用接口。建模者可以增加新的构造型来表示专门的模型元素。一个构造型可以带有多个约束、标记值或者代码生成特性。如图所示，建模者可以为命名的构造型定义一个图标，作为可视化的辅助工具。尽管如此，可以使用文字形式说明。

Scheduling 包中的标记值说明 Frank Martin 要在年底世纪前完成计划的制定。可以将任意信息作为标记值写于一个模型元素中建模者选定的名字之下。使用文字有益于描述项目管理和代码生成参数。大部分标记值保存为编辑工具中的弹出信息，在正式打印出的图表中通常没有标记值。

图 3-11 扩展组件

3.3.9 各种视图间的关系

多个视图共存于一个模型中，它们的元素之间有很多关系，其中一些关系列在表 3-2 中。表中没有将各种关系列全，但它列出了从不同视角观察得到的元素间的部分主要关系。

表 3-2 不同视图元素间的部分关系

3.4 用例视图

长期以来,在面向对象开发和传统的软件开发中,人们根据典型的使用情景来了解需求。但是,这些使用情景是非正式的,虽然经常使用,却难以建立正式文挡。用例模型由 Ivar Jacobson 在开发 AXE 系统中首先使用,并加入由他所倡导的 OOSE 和 Objectory 方法中。用例方法引起了面向对象领域的极大关注。自 1994 年 Ivar Jacobson 的著作出版后,面向对象领域已广泛接纳了用例这一概念,并认为它是第二代面向对象技术的标志。

用例视图也称用例模型，用例模型描述的是外部执行者(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。首先,它描述了待开发系统的功能需求;其次,它将系统看作黑盒,从外部执行者的角度来理解系统;第三,它驱动了需求分析之后各阶段的开发工作,不仅在开发过程中保证了系统所有功能的实现,而且被用于验证和检测所开发的系统,从而影响到开发工作的各个阶段和 UML 的各个模型。在 UML 中,一个用例模型由若干个用例图描述，用例图中显示执行者、用例和用例之间的关系。在 UML 语言中，用例模型是用用例图描述的。用例模型可以由若干各用例图组成。用例图包含系统、执行者和用例三种模型元素。

3.4.1 系统

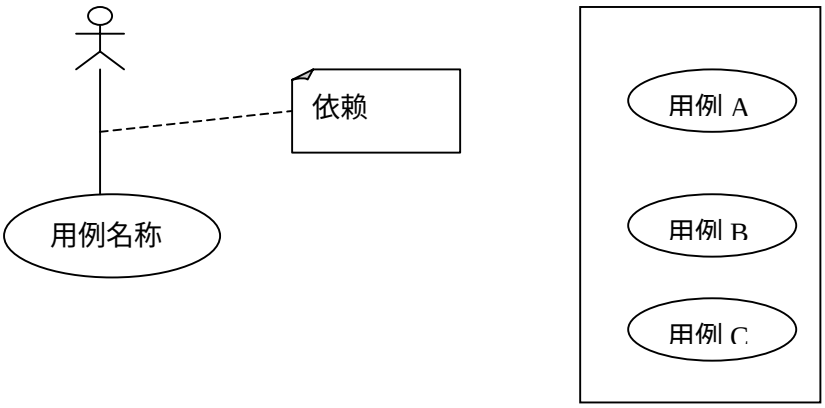
系统是用例模型的一个组成部分，代表的是一部机器或一个业务活动，而不是真正实现的软件系统。系统的边界用来说明构建的用例模型的应用范围。例如一台自助式售货机（被看作系统）应提供售货、供货、提取销售款等功能。这些功能在自动售货机之内的区域起作用，自动售货机之外的情况不考虑。准确定义系统的边界并不总是容易的事，因为严格地划分那种任务最好由系统自动实现，哪些任务由其他系统或人工实现是很困难的。另外系统最初的规模应有多大也应该考虑。一般的做法是，先识别出系统的基本功能，然后以此为基础定义一个稳定的、精确定义的系统架构，以后再不断地扩充系统功能，逐步完善。

在建模初期，定义一些术语和定义是很有必要的。因为在描述系统、用例或进行作用域分析时采用统一的术语和定义能够规范表述系统的含义，不致于出现误解。

3.4.2 用例(use case)

用例代表的是一个完整的功能，用例是动作步骤的集合。动作是系统的一次执行。从

本质上讲,一个用例是用户与计算机之间的一次典型交互作用。在 UML 中,用例被定义成系统执行的一系列动作,动作执行的结果能被指定执行者察觉到。例如在自动售货系统中，当顾客付款之后，系统自动送出顾客想要的饮料，这是一种动作；付款后若需要的饮料无货，则提示可否买其他货物？或退款等等。系统中的每种可执行情况就是一个动作，每种动作由许多步骤组成。



用例图示例

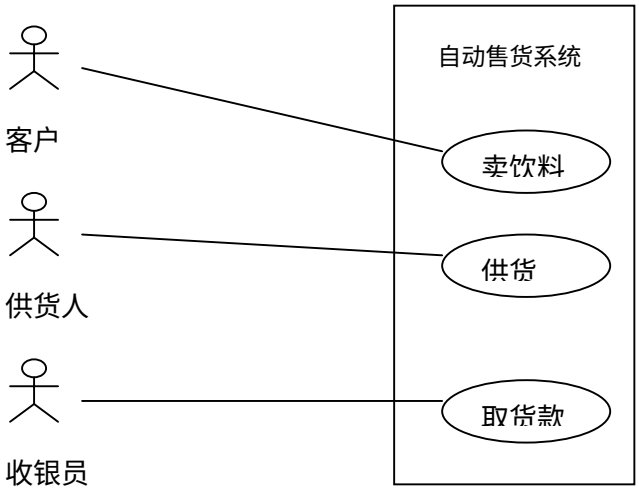


图 3-12 自动售货机系统用例图

在 UML 中,用例表示为一个椭圆。方框表示系统边界，上图左边的小人状图案表示执行者。

用例和执行者之间也有连接关系，它们直接的关系属于关联（association），又称通信关联，这种关联表示那种执行者能与该用例通信。关联关系是双向的多对多关系，一个执行者可以与多个用例通信，一个用例也可以与多个执行者通信。

用例表示的也是一个类，而不是某个具体的实例。用例描述了他代表的功能的各个方面。也就是包含了用例执行期间可能发生的种种情况。用例的实例（也是一种动作）代表系统的一种实际使用方法，这个实例通常叫作情景（scenario）。情景是系统的一次具体执行路线。例如在自动售货系统中，张三投币买希望矿泉水，系统收到消息后，将矿泉水送出的过程就是一种情景。

概括地说,用例有以下特点:

（1）用例总由执行者初始化。用例所代表的功能必须由执行者激活，而后才能执行。一般情况下执行者可能并没有意识到初始化了一个用例。换句话说，执行者需要系统完成的功能都是通过用例完成的，执行者一定会直接或间接地命令系统执行用例。

（2）用例为执行者提供值。用例必须为执行者提供实在的值，虽然这个值并不总是重要的，但是能被执行者识别。

（3）用例具有完全性。用例是一个完整的描述。用例可大可小,但它必须是对一个具体的用户目标实现的完整描述。

3.4.3 执行者(Actor)

执行者是指用户在系统中所扮演的角色，是与系统交互的人或事。其图形化的表示是一个小人状图案。图中有四个执行者:客户、供货人和收银员。

执行者是一个群体概念，代表的是一类能使用某个功能的人或事，执行者不是指某个个体。比如，在自动售货系统中，系统有售货、供货和提取售货款等功能，启动售货功能的是人，那么人是执行者，如果再把人具体化，该人可以是张三，也可以是李四，但是张三和李四这些个体对象不能称作执行者，它们是执行者的实例。事实上，一个具体的人可以在系统中充当不同的执行者，例如张三既可以为售货机添加新物品（执行供货），也可已将售货

机中的钱取走（执行提取执行款）。通常系统会会对执行者的行为有所约束，使其不能执行某些功能。

需要注意的是执行者在用例图中是用类似人的图形来表示,尽管执行的,但执行者未必是人。例如,执行者也可以是一个外界系统,该外界系统可能需要从当前系统中获取信息,与当前系统有进行交互。

通过实践,我们发现执行者对提供用例是非常有用的。面对一个大系统,要列出用例清单常常是十分困难。这时可先列出执行者清单,再对每个执行者列出它的用例,问题就会变得容易很多。

执行者可以分为主要执行者和次要执行者。主要执行者（primary actor）指的是执行系统主要功能的执行者，次要执行者指的是使用系统次要功能的执行者，次要功能是指一般完成维护系统的功能，例如管理数据库、通信备份等。将执行者分级的目的是，保证把系统的所有功能表示出来。

执行者是一个类，它拥有与类相同的关联描述，在用例图中，只用概括化（generalization）关联描述若干执行者之间的关系。既将某些执行者的共同行为抽取出来表示成概括的行为，且将它们描述为超类。

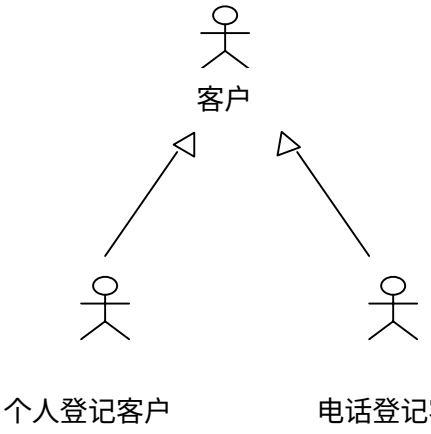


图 3-13　执行者之间的概括化联

上图中，客户是超类，它描述了客户的基本行为，由于客户申请业务的方式可以不同，故可将客户具体分为两类：一类是电话登记客户，一类是亲自登门登记客户。但这两类客户的基本行为是一致的，差别只是在于申请的方式不同，于是在定义这两个类行为时，基本行为可以从客户类中继承得到，与客户类不同的行为则定义在各自的执行者类中。

3.4.4 使用和扩展(Use and Extend)

中除了包含执行者与用例之间的关联外,还有另外两种类型的关联,用以表示用例之间的使用和扩展关联。。

（1） 扩展关联

一个用例中加入一些新的动作后则构成另一个用例，这两个用例之间的关联是概括化关系，称作扩展关联。后者通过继承前者的一些行为得来，前者通常称为概括化用例，后者常称作扩展用例。扩展用例可以根据需要有选择地继承概括化的用例部分行为。

引入扩展用例的好处在于便于处理概括化用例中不易描述的某些具体情况，便于扩展系统，提供系统性能，减少不必要的重复工作。

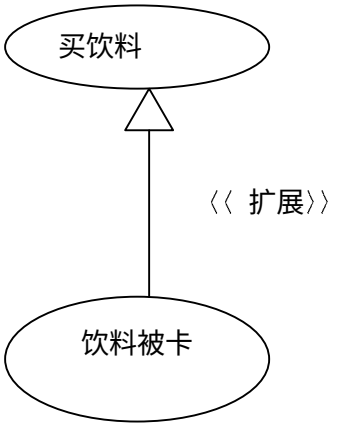


图 3-14 扩展关系图例

例如在卖饮料的用例中，如果出现饮料被卡的情形，只要对普通的买饮料用例加上一系列的故障处理动作，就扩展成为一个新的用例。

（2） 使用关联

一个用例使用另一个用例时，这两个用例之间就构成了使用关系。一般情况下，如果若干个用例的某些行为是相同的，则可以把这些相同的行为提取出来单独作为一个用例，这个用例称作抽象用例。这样当某个用例使用该抽象用例时，就好像这个用例包含了抽象用例的所有行为。

例如自动售货系统中，“供货”和“提取销售款”这两个用例的开始动作都是去掉机器保险并打开它，最后的动作一定是关闭机器并加保险，于是可以从这两个用例中把开始的动作抽象成“打开机器”用例，被最后的动作抽象成“关闭机器”用例，那么“供货”和“提取销售款”用例，把最后的动作抽象成“关闭机器”用例，那么“供货”和“提取销售款”用例在执行时一定要使用上述的两个抽象用例，它们之间构成使用关联。

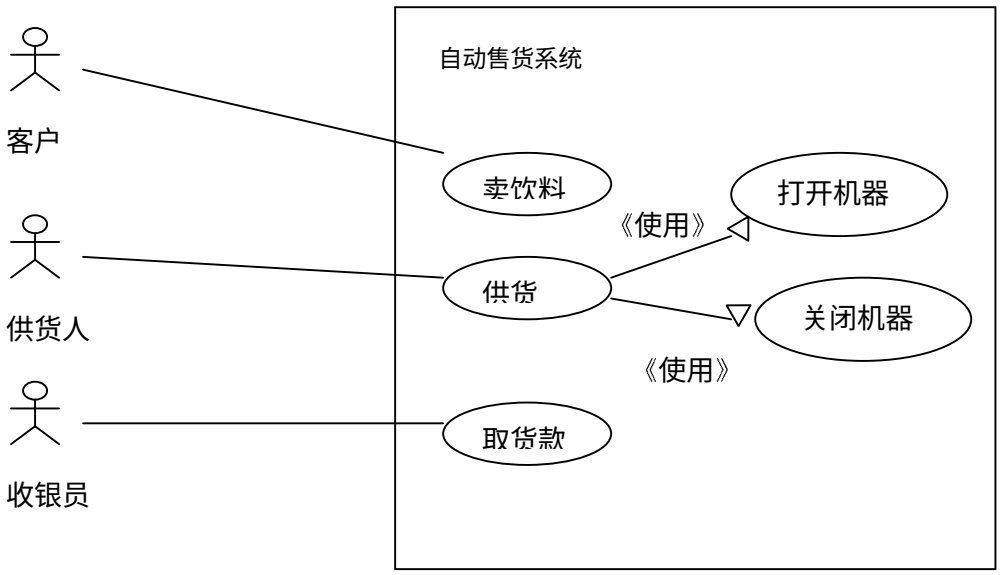


图 3-15 自动售货机系统用例图

请注意扩展与使用之间的相似点和不同点。它们两个都意味着从几个用例中抽取那些公共的行为并放入一个单独用例中,而这个用例被其他几个用例使用或扩展。但使用和扩展的目的是不同的。当描述一般行为的变化时，采用关联。当在两个或更多的用例中出现重复描述而又想避免这种重复时采用扩展

3.4.5 用例模型的获取

几乎在任何情况下都会使用用例。用例用来获取需求,规划和控制项目。用例的获取是

需求分析阶段的主要任务之一,而且是首先要做的工作。大部分用例将在项目的需求分析阶段产生,并且随着工作的深入会发现更多的用例,这些都应及时增添到已有的用例集中。用例集中的每个用例都是一个潜在的需求。

3.4.5.1 获取执行者

获取用例首先要找出系统的执行者。可以通过用户回答一些问题的答案来识别执行者。以下问题可供参考:

- 谁使用系统的主要功能(主要使用者)。
- 谁需要系统支持他们的日常工作。
- 谁来维护、管理使系统正常工作(辅助使用者)。
- 系统需要操纵哪些硬件。
- 系统需要与哪些其它系统交互,包含其它计算机系统和其它应用程序。
- 对系统产生的结果感兴趣的人或事物。

3.4.5.2 获取用例

一旦获取了执行者,就可以对每个执行者提出问题以获取用例。

以下问题可供参考:

- 执行者要求系统提供哪些功能(执行者需要做什么)?
- 执行者需要读、产生、删除、修改或存储的信息有哪些类型。
- 必须提醒执行者的系统事件有哪些?或者执行者必须提醒系统的事件有哪些?怎样把这些事件表示成用例中的功能?
- 为了完整地描述用例,还需要知道执行者的某些典型功能能否被系统自动实现?

还有一些不针对具体执行者问题(即针对整个系统的问题):

- 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(也许是一些手工操作而不是计算机系统)的主要问题?

需要注意,最后两个问题并不是指没有执行者也可以有用例,只是获取用例时尚不知道执行者是什么。一个用例必须至少与一个执行者关联。还需要注意:不同的设计者对用例的利用程度也不同。例如,Ivar Jacobson 说,对一个十人年的项目,他需要二十个用例。而在一个相同规模的项目中,Martin Fowler 则用了一百多个用例。我们认为:任何合适的用例都可使用,确定用例的过程是对获取的用例进行提炼和归纳的过程,对一个十人年的项目来说,二十个用例似乎太少,一百多个用例则嫌太多,需要保持二者间的相对均衡。

3.5 类图、和对象图

数千年以前,人类就已经开始采用分类的方法有效地简化复杂问题,帮助人们了解客观世界。在面向对象建模技术中,我们使用同样的方法将客观世界的实体映射为对象,并归纳成一个个类。类(Class)、对象(Object)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。

3.5.1 类图

类图(Class Diagram)描述类和类之间的静态关系。与数据模型不同,它不仅显示了信息的结构,同时还描述了系统的行为。类图是定义其它图的基础。在类图的基础上,状态图、交互图等进一步描述了系统其他方面的特性。

类描述一类对象的属性(Attribute)和行为(Behavior)。在 UML 中,类的可视化表示为一个划分成三个格子的长方形(下面两个格子可省略)。

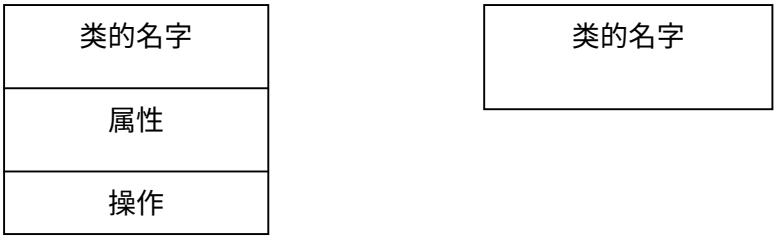


图 3-16 类的完整和简化的符号

最顶部栏表示类的名字，中间层栏表示类的属性，底层栏表示类的操作。在图的右边显示了一个简化的符号，它省略了属性和操作栏。

3.5.1.1 类的获取和命名

最顶部栏包含类的名字。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。类的获取是一个依赖于人的创造力的过程,必须与领域专家合作,对研究领域仔细地分析,抽象出领域中的概念,定义其含义及相互关系,分析出系统类,并用领域中的术语为类命名。一般而言,类的名字是名词。

3.5.1.2 类的属性

中间层栏包含类的属性,用以描述该类对象的共同特点。该项可省略。图 1 中"客户"类有"客户名"、"地址"等特性。为清楚起见，这里只考虑实例属性。了解属性时应该考虑以下因素:

- 属性表示关于对象的信息，原则上来说,类的属性应能描述并区分每个特定的对象;
- 只有系统感兴趣的特征才包含在类的属性中;
- 系统建模的目的也会影响到属性的选取。
- 通常对于外部对象来说，属性是可获取(gettable)和可设置的(settable)。
- 另一方面，一些属性是只读的，即只能从对象外部获取。这对应于 CORBA 接口定义语言（IDL）中 Readonly 的表示法。
- 术语属性和变量的含义完全不同。属性表示一个抽象定义的特性，独立于特性的内部实现；而变量是一个内部的实现机制。然而大多数情况下，属性最终是由一个简单的变量实现的。

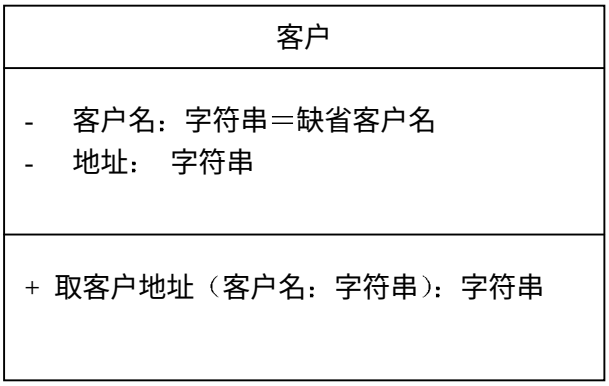


图 3-17 属性及操作的可见性实例

根据图的详细程度,每条属性可以包括属性的可见性、属性名称、类型、缺省值和约束特性。UML 规定类的属性的语法为：

可见性 属性名 ： 类型 = 缺省值 {约束特性}

图 3- "客户"类中,"客户名"属性描述为"- 客户名 ： 字符串 = 缺省客户名"。可见性"-"表示它是私有数据成员,其属性名为"客户名",类型为"字符串"类型,缺省值为"缺省客户名",此处没有约束特性。

不同属性具有不同可见性。常用的可见性有 Public、Private 和 Protected 三种,在 UML 中分别表示为"+"、"-"和"#"。

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

约束特性则是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。

3.5.1.3 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}

在图 3- 中,"客户"类中有"取客户地址"操作,其中" +"表示该操作是公有操作,调用时需要参数"客户名",参数类型为字符串,返回类型也为字符串。

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。

3.5.2 关联关系

3.5.2.1 关联

关联(Association)表示两个类之间存在某种语义上的联系。例如,一个人为一家公司工作,一家公司有许多办公室。我们就认为人和公司、公司和办公室之间存在某种语义上的联系。在分析设计的类图模型中,则在对应人类和公司类、公司类和办公室类之间建立关联关系。

关联可以有方向,表示该关联单方向被使用。关联上加上箭头表示方向,在 UML 中称为导航(Navigability)。我们将只在一个方向上存在导航表示的关联,称作单向关联(Uni-directional Association),在两个方向上都有导航表示的关联,称作双向关联(Bi-directional Association)。UML 规定,不带箭头的关联可以意味着未知、未确定或者该关联是双向关联三种选择,因此,在图中应明确使用其中的一种选择。关联关系一般都是双向的,即关联的对象双方彼此都能与对方通信。根据不同的含义,关联可以分为普通关联、递归关联、限定关联、或关联、有序关联、三元关联和聚合等七种。我们首先讨论普通关联。

图 3-18 单向导航关联

既然关联可以是双向的,可以在关联的一个方向上为关联起一个名字,而在另一各方向上起另一个名字(也可不起名字),名字通常仅挨着直线书写。为了避免混淆,在名字的前面或后面带一个表示关联方向的黑三角,黑三角的尖角指明这个关联只能用在尖角所指的类上。为关联命名有几种方法,其原则是该命名是否有助于理解该模型。

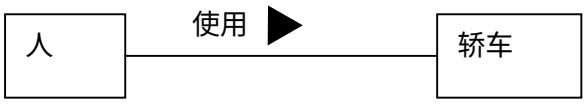


图 3-19 普通关联

3.5.2.2 关联的角色

关联两头的类以某种角色参与关联。例如图 2 中,"公司"以"雇主"的角色,"人"以"雇员"的角色参与的"雇用"关联。"雇主"和"雇员"称为角色名。如果在关联上没有标出角色名,则隐含地用类的名称作为角色名。角色还具有多重性(Multiplicity),表示可以有多少个对象参与该关联。在图 2 中,雇主(公司)可以雇佣(签工作合同)多个雇员,表示为"*"; 雇员只能被一家雇主所雇佣,表示为"1"。多重性表示参与对象的数目的上下界限制。例如

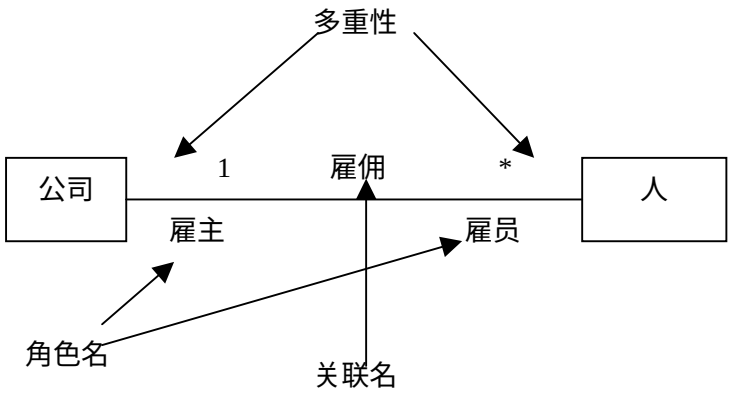


图 3-20 关联的角色

0..1	表示 0 到 1 个对象
0..* 或*	表示 0 到多个对象
5..7	表示 5 到 17 个对象
2	表示 2 个对象

如果图中没有明确表示关联的重数，那就意味着是 1。

3.5.2.3 关联类

在有些问题中，关联关系不仅需要一个名称、需要定义相关对象的角色及其参与这些角色的对象数量，而且还需要设置一些属性、操作以及其它特征。从下图可以看出来，人可以为一个公司工作。我们需要保存一些信息，用于记录雇员为公司工作的年限等。但存在下列几个问题。

首先从概念上讲，这种信息应当从属于人与公司之间的“雇佣关系”，而不是人或者

公司自身的属性，既然工作年限是关联的一个属性，就应当把它记录在关联关系中。

其次从实现上讲，这种信息记录的实现方式受到关联关系中各个角色的多重性的影响。例如图中规定一个人只能被一家公司所雇佣，所以共工作年限可以作为人的一个属性，在实现中直接在人的的类中增加一个属性即可，当然此时还应当增加另一个属性记录他所受雇的公司。

但是如果这种关系发生变化，例如国家允许个人谋求第二职业，甚至多个职业时，这种实现必须改变。

从上述可知，关联可能要记录一些信息,因此引入一个关联类来记录，也就是说，与一个关联关系相连的类称作关联类。图 3 是在图 2 的基础上引入了关联类。关联类并不位于表示关联关系的直线两端，而是对应一个实际的关联，用关联类表示该实际关联的一些附加信息。关联类通过一根虚线与关联连接。图 4 是实现上述目标的另外一种方法,就是使雇用关系成为一个正式的类。在这种方法中，原来有关联的每个类（人和公司）相对于雇佣类各增加了一个单值的角色，这是“雇主”角色就变成了派生的角色，甚至不必画出来。

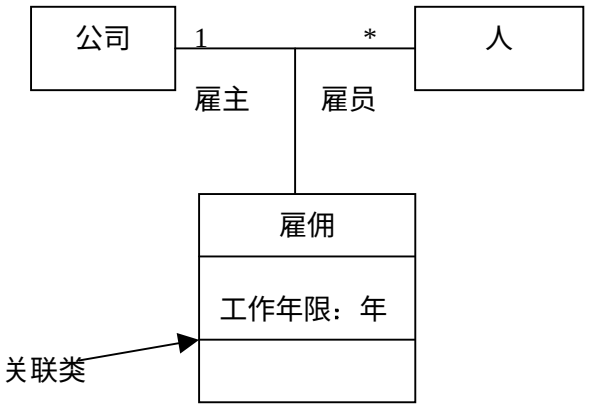


图 3-21 关联类

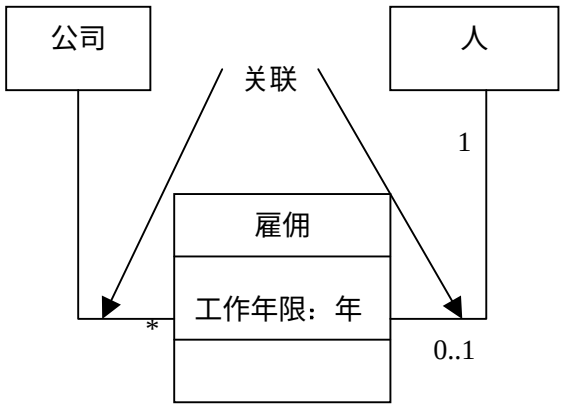


图 3-22 另一种实现方法

3.5.2.4 整体/部分关联

UML 对于整体/部分关联有特殊的表示法：组成和聚集。

组成（composition）关联表示整体拥有各部分,部分与整体共存,如整体不存在了,部分也会随之消失。“整体”称作组成对象，“部分”称作成分对象。例如电子邮件包括报头和一些文本段落组称；报头是发送者姓名、接受者姓名、消息标题和其它类容按顺序组成。组成具有三个重要特征。

（1）如果没有成分对象，组成对象也不存在。例如从一个牙刷上去除刷毛、手柄和一层橡胶则牙刷不复存在。实际上只要把牙刷的刷毛去掉牙刷就已经不存在了。既然这样，一个组成对象的生存期不能超过它的成分对象生存周期。它说明了成分对象和组成对象共存亡。删除组成对象可以看作删除组成对象以及其所有的成分对象。

（2）在任何时候，每个给定的成分对象只能是组成对象的组成部分。说明一个成分对象仅属于一个整体。

（3） 组成是典型的异构。成分对象很可能由多个类型混合组成：几个轮子、几个车轴和一些木头…，这些构成四轮马车。

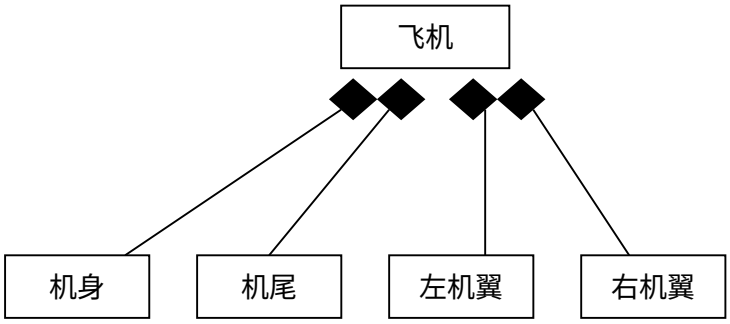


图 3-23　一个组成对象和其成分对象

在图中组成对象和每个成分对象之间的关联表示为一端有黑色菱形块的关联线，菱形块放在组成对象一侧。

聚集(Aggregation)是一种特殊形式的关联。聚集也表示类之间的整体/部分关联，但主要强调组/成员的关联。整体被称作聚集对象，部分称作构成对象。例如森林是树木的集合。聚集具有三个特征：

（1）构成对象不存在，聚集对象还可以存在。例如把一个部门的所有职工都解雇，那么该部门仍然存在。这个特性用处不太大。

（2）在任何时候，每个对象都可以是多个聚集的构成。一个人可能属于多个俱乐部。

（3）聚集往往是同构的，也就是，典型的聚集的构成对象将属于同一个类。例如树木都是由树木组成。

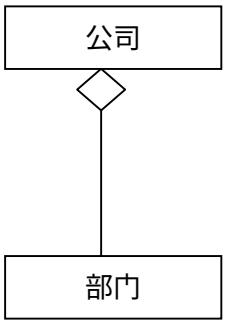


图 3-24 聚集对象和它的构成对象

在图中聚集对象和每个构成对象之间的关联表示为一端有菱形块的关联线，菱形块放在聚集对象一侧。

3.5.2.5 继承关系

一个类（一般元素）的所有特征（属性或或操作）能被另一个类（特殊元素）继承。也称概括（generalization）。概括定义了一般元素和特殊元素之间的关系。在 UML 中,继承表示为一头为空心三角形的连线。
在 UML 定义中对继承有三个要求：

- 特殊元素应与一般元素完全一致,一般元素所具有的关联、属性和操作,特殊元素也都隐含性地具有；
- 特殊元素还应包含额外信息；
- 允许使用一般元素实例的地方,也应能使用特殊元素。

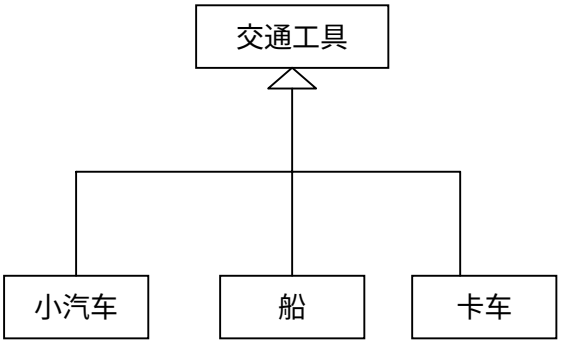


图 3-25 类的继承

没有实例的类称作抽象类。在类图中表示抽象类时在类的名字下附加一个标记值{abstract}。

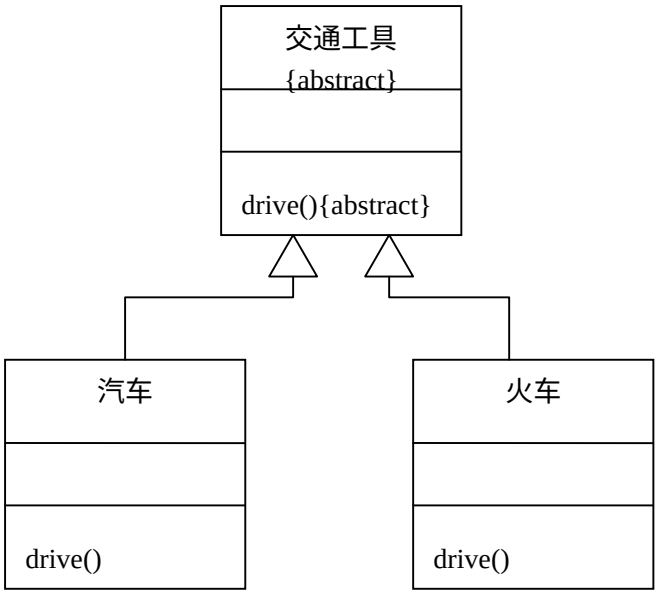


图 3-26 抽象类实例

抽象类中一般都带有抽象的操作。抽象操作仅仅用来描述该抽象类的所有子类应有什么样的行为，抽象操作只标记返回值、操作的名称和参数表，关于操作的实现细节并不详细书写出来，抽象操作的具体实现细节由继承抽象类的子类实现。抽象操作的图示方法与抽象类相似，是在操作后面跟随一个性质串{abstract}。

在继承中往往给继承关联一个约束条件。预定义的约束有三种分类，一种分类是互斥分类，包括重叠(overlap)和互斥(disjoint)，第二种分类是不完全分类，包括完全(complete)和不完全(incomplete)；第三种分类是动态分类，包括动态(dynamic)和静态(static)。

互斥适合于具有两个以上分组，能够清晰区分事物，在同一时间一个事物只能属于一组。重叠刚好与互斥相反。

不完全适用于一个组的子组。在一个组中，不是所有可能的子组都包括在模型中。这个组可能存在一些成员，他们不属于任何建模的子组。完全分类刚好相反，表示该模型包含了所有的子组。

动态表示事物随时间的变化而从属于不同的子组。一个具有动态分类的事物可能在开始时属于一个子组，但随后又成为另一个子组的成员。而静态表示自始至终属于一个子组的成员。

上述三种分类在项目初期非常有用，也就是说当项目小组正分析商业需求时，并且他们在设计阶段时也非常有用。

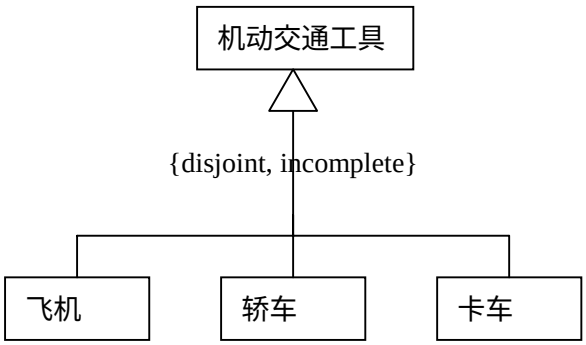


图 3-27 互斥的、不完全的继承

从上例可以看出飞机、轿车和卡车时互不重叠的，即互斥的，分类也是不完全的，因为至少还有火车、轮船也是机动交通工具。

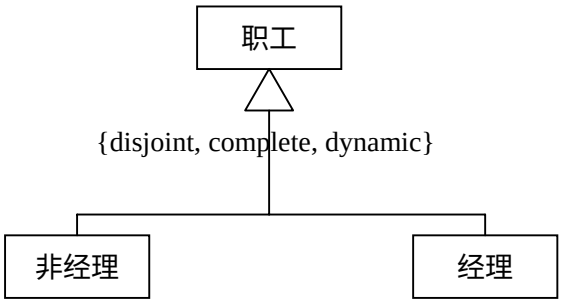


图 3-28 一个包括动态继承分类

在上例中，对于某职工来说，随着时间的变化，他可能由非经理变成经理，也由可能经理变为非经理。

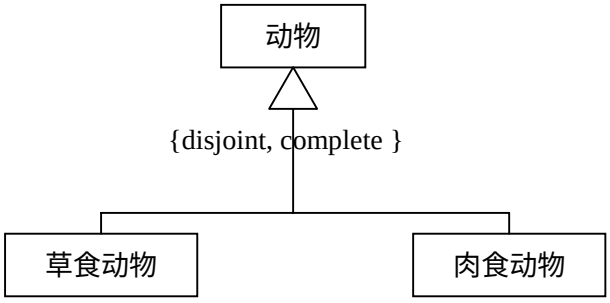


图 3-29 一个重叠、完全的分类

3.5.3 参数化类

参数化类，在第二章中叫做类属类，也叫做模版类，它是一个尚未完全说明的类。该类中提供参数表，利用参数表向参数化类传递参数，最终形成用户需要的具体类。参数可以是类，也可以是基本数据类型。在 UML 中，用一个位于标准类符号的右上角的虚线框表示一个参数表。

下图表示 Set，一个典型的容器类。它把形式上命名为 T 的一个类作为参数。当赋予 T 一个实际的类，例如 car，那么，参数化类的每个对象都将表示为汽车的集合，这个以赋予参数的类叫做 Set<Car>。

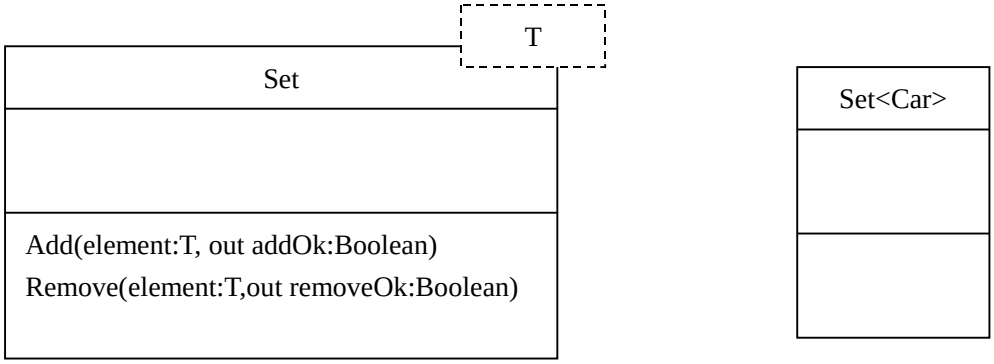


图 3-30 参数化的类

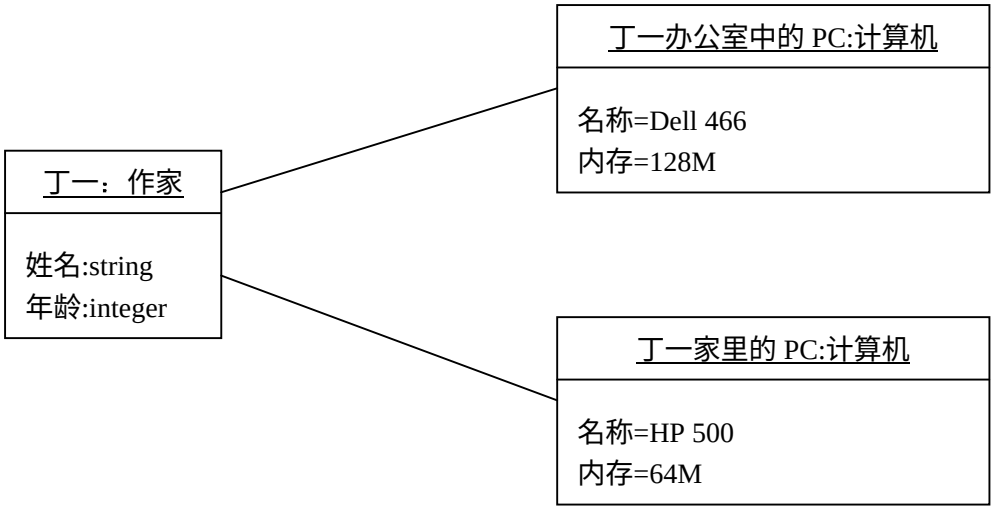
3.5.4 对象图

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。对象是类的实例;对象之间的链(Link)是类之间的关联的实例。对象的图示方法与类的图示方法几乎一样。主要差别在于对象的名字下面要加下划线。链的图形表示与关联相似。对象图常用于表示复杂的类图的一个实例。

对象名有下列三种表示格式:（1）对象名：类名（2） ：类名 （3）对象名。

下图给出一个类图和一个对象图。其中对象图是类图的示例。

(a) 类图



(b) 对象图

图 3-31 对象图示例