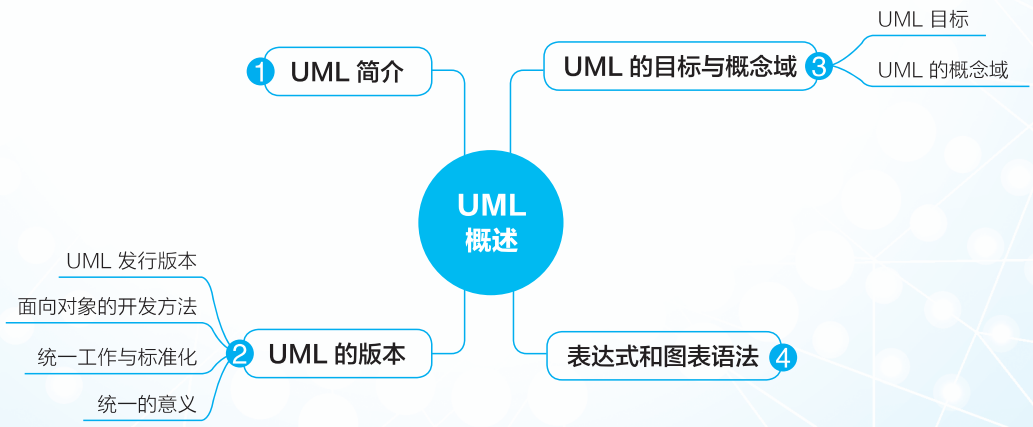
目录

第 1 章 UML 概述

学习目标

1. 了解 UML 的发行版本和标准化。
2. 熟悉表达式和图表的语法。

知识导图



笔记

1.1 UML 简介

统一建模语言（Unified Modeling Language，UML）是一种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，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。它可由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 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。静态结构和动态行为从不同但相互联系的角度对系统建立的模型用于不同的目的。

UML 还可将模型分解成包的结构组件，以便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组件。

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 不是一种可用定理证明的高度形式化的语言，这样的语言有很多种，但它的通用性较差，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。

1.2 UML 的版本

1.2.1 UML 发行版本

UML 是为了实现简化和强化现有的大量面向对象开发方法这一目的而开发的。当人类发明了抽象的编程语言后，随着软件开发规模的不断扩大，有一个问题一直困扰着软件开发人员，那就是如何在开发人员之间进行交流，因为编程语言比较抽象，所以交流也就成了一个难题。

为了解决这个难题，从 20 世纪 70 年代开始，就不断有面向对象的建模语言面世，但新的问题也随之而来了，正如前面讲的，有人把白粉笔定义为白色，有人把白粉笔定义为黑色，双方定义的语义基础不同，不同的建模语言交流起来就很困难。

面向对象软件工程的概念最早是由 Booch 提出的，他是面向对象方法最早的倡导者之

一。后来，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。在 1994 年，Jacobson 提出了 OOSE 方法，其最大特点是面向用例，并在用例的描述中引入了外部角色的概念。

笔记

1994 年 10 月，Grady Booch 和 Jim Rumbaugh 首先将 Booch 93 和 OMT-2 统一起来，并于 1995 年 10 月发布了第一个公开版本 UM 0.8，后来 Jacobson 也加盟到这一工作中，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。

1996 年，一些公司和组织，比如 DEC、HP、IBM、Microsoft、Oracle、RationalSoftware 等成立了 UML 成员协会，以完善、加强和促进 UML 的定义工作。

1997 年 1 月 UML 1.0 发布，1998 年发布了 UML 1.2 版本，一年后发布了 UML 1.3 版本，2003 年 3 月发布了 UML 1.5 版本。

2017 年 12 月 UML 2.5.1 版本发布。

1.2.2 面向对象的开发方法

利用传统程序设计语言（如 Cobol 和 FORTRAN 语言）的软件开发方法出现于 20 世纪 70 年代，在 80 年代被广泛采用，其中最重要的是结构化分析和结构化设计方法 [Yourdon-79] 及其变体，如实时结构化设计方法 [Ward-85] 等。这些方法最初由 Constantine、Demarco、Mellor、Ward、Yourdon 和其他人发明并推广，在一些大型系统，特别是在与政府签约的航空和国防领域的系统中取得了一定突破。在这些系统中，主持项目的人员强调开发过程的有组织性和开发设计文档的完备和充分。结果不总是像预料的那么好，即许多计算机辅助软件工程（CASE）系统只是摘录一些已实现的系统设计的报表生成器。尽管如此，这些方法中仍包含了一些好的思想，有时在一些大系统中是很有效的。商业应用软件更不愿采用大型的 CASE 系统和开发方法，大部分商业企业都独立开发内部使用的软件，客户和缔约人之间没有对立关系，而这种关系正是大型政府工程的特征。一般认为商用系统比较简单，不论这种看法是否正确，反正它不需要经过外界组织的检查。

普遍认为，诞生于 1967 年的 Simula-67 是第一个面向对象的语言。尽管这个语言对后来的许多面向对象语言的设计产生了很大的影响，但是它没有后继版本。20 世纪 80 年代初，Smalltalk 语言的广泛使用掀起了一场“面向对象运动”，随之诞生了面向对象的 C、C++、Eiffel 和 CLOS 等语言。尽管面向对象的编程语言在实际使用中有一定的局限性，但它仍然吸引了很多人的注意力。在 Smalltalk 语言成名约 5 年后，第一批介绍面向对象软件开发方法的书籍出现了，包括 *Shlaer/Mellor*[*Shlaer-88*] 和 *Coad/Yourdon*[*Coad-91*]，紧接着又有 Booch 的 [*Booch-91*]、Rumbaugh/Blaha/Premerlani/Eddy/Lorensen 的 [*Rumbaugh-91*] 和 Wirfs-Brock/Wilkerson/Wiener[*Wirfs-Brock-90*]（注意：图书版权年往往包括上一年度 7 月份以后出版的书）。这些著作再加上 Goldberg/Robson[*Goldberg-83*]*Cox*[*Cox-86*] 和 Meyer[*Meyer-88*] 等有关程序语言设计的著作，开创了面向对象方法的先河。第一阶段在 1990 年末完成。稍后出版了 [*Jacobson-92*]，它建立在以前成果的基础上，介绍了一种稍

笔记

微不同的方法，即以用例和开发过程为中心。

在以后的 5 年中，大批关于面向对象方法的书籍问世，它们各有自己的一套概念、定义、表示法、术语和适用的开发过程。有些书提出了一些新概念，但总的来说，各个作者使用的概念大同小异。许多后继出版的书都照搬前人，自己再做一些小的扩充或修改。最早的作者们也没闲着，大部分人都更新了自己前期的著作，采纳了其他人一些好的思想。总之，出现了一些被广泛使用的核心概念，另外还有一大批被个别人采纳的概念。即使在被广泛接受的核心概念里，面向对象方法也有一些小的差异。这些面向对象方法之间的细微差异常使人不知依据哪个概念为好，特别是对非专业的读者来说。

1.2.3 统一工作与标准化

在 UML 之前，已经有一些试图将各种方法中使用的概念进行统一的初期尝试，比较有名的一次是 Coleman 和他的同事 [Coleman-94] 对 OMT[Rumbaugh-91]、Booch[Booch-91]、CRC[Wirfs-Brock-90] 方法使用的概念进行融合。由于这项工作没有这些方法的原作者参与，实际上仅形成了一种新方法，而不能替换现存的各种方法。第一次成功合并和替换现存的各种方法的尝试始于 1994 年在 Rational 软件公司的 Rumbaugh 与 Booch 合作。他们开始合并 OMT 和 Booch 方法中使用的概念，于 1995 年提出了第一个建议。此时，Jacobson 也加入了 Rational 公司开始与 Rumbaugh 和 Booch 一同工作。他们共同致力于设计统一的建模语言。三位最优秀的面向对象方法学的创始人共同合作，为这项工作注入了强大的动力，打破了面向对象软件开发领域内原有的平衡。而在此之前，各种方法的拥护者觉得没有必要放弃自己已经采用的概念而接受这种统一的思想。

1996 年，OMG 向外界发布了征集关于面向对象建模标准方法的消息。UML 的三位创始人开始与来自其他公司的软件工程方法专家和开发人员一道采用一套使 OMG 感兴趣的方法，并设计一种能被软件开发工具提供者、软件开发方法学家和开发人员这些最终用户所接受的建模语言。与此同时，其他人也在做这项富有竞争性的工作。1997 年 9 月，所有建议终于被合并成一套 UML 方法提交到 OMG，因此最后的成果是许多人共同努力的结果。

1997 年 11 月，UML 由 OMG 全体成员一致通过，并被采纳为标准。OMG 承担了进一步完善 UML 标准的工作。许多软件开发工具供应商声称他们的产品支持或计划支持 UML，若干软件工程方法学家宣布他们将使用 UML 的表示法进行以后的研究工作。UML 的出现似乎深受计算机界欢迎，因为它是由官方出面集中了许多专家的经验而形成的，减少了各种软件开发工具之间无谓的分歧。希望建模语言的标准化既能促进软件开发人员广泛使用面向对象建模技术，同时也能带来 UML 支持工具和培训市场的繁荣，因为不论是用户，还是供应商，都不用再考虑到底应该采用哪种开发方法。

知识拓展

核心组员

提出 UML 建议或进行 UML 标准修订工作的核心组员有下列人员：

数据存取公司：TomDigre

DHR 技术公司：EdSeidewitz

HP 公司：MartinGriss

IBM 公司：SteveBrodsky，SteveCook，JosWarmer

I-Lgix 公司：EranGery，DavidHarel

ICONComputing 公司：DesmondD'Souza

IntelliCorpandJamesMartin 公司：ConradBock，JamesOdell

MCI 系统企业：CrisKobryn，JoaquinMiller

ObjecTime 公司：JohnHogg，BranSelic

Oracle 公司：GuusRamackers

铂技术公司：DilharDesilva

Rational 软件公司：GradyBooch，EdEykholt，IvarJacobson，GunnarOvergaard，Karin Palmkvist，JamesRumbaugh

SAP 公司：OliverWiegert

SOFTEAM：PhilippeDesfray

Sterling 软件公司：JohnCheesman，KeithShort

Taskon 公司：TrygveReenskaug

笔记

1.2.4 统一的意义

“统一”这个词在 UML 中有下列相互关联的含义。

1. 在以往出现的方法和表示法方面

UML 合并了许多面向对象方法中被普遍接受的概念对每一种概念，UML 都给出了清晰的定义、表示法和有关术语。使用 UML 可以对已有的用各种方法建立的模型进行描述，并比原来的方法描述得更好。

2. 在软件开发的生命周期方面

UML 对开发的要求具有无缝性。开发过程的不同阶段可以采用相同的一套概念和表示法，在同一个模型中它们可以混合使用。在开发的不同阶段不必转换概念和表示法。这种无缝性对迭代式的、增量式的软件开发是至关重要的。

3. 在应用领域方面

UML 适用于各种应用领域的建模，包括大型的、复杂的、实时的、分布式的、集中式数据或计算的、嵌入式的系统。也许用某种专用语言来描述一些专门领域更有用，但在大部分应用领域中，UML 比其他的通用语言更好。

4. 在实现的编程语言和开发平台方面

UML 可应用于运行各种不同的编程实现语言和开发平台的系统。其中包括程序设计语言、数据库、组织文档及固件等。在各种情况下，前部分工作应当相同或相似，后部分

笔记

工作因各种开发媒介的不同而有某种程度上的不同。

5. 在开发全过程方面

UML 是一种建模语言，不是对开发过程的细节进行描述的工具。就像通用程序设计语言可以用于许多风格的程序设计一样，UML 适用于大部分现有的或新出现的开发过程，尤其适用于迭代式增量开发过程。

6. 在内部概念方面

在构建 UML 元模型的过程中，特别注意揭示和表达各种概念之间的内在联系并试图用多种适用于已知和未知情况的办法去把握建模中的概念。这个过程会增强对概念及其适用性的理解。这不是统一各种标准的初衷，但却是统一各种标准最重要的结果之一。

1.3 UML 的目标与概念域

1.3.1 UML 目标

UML 的开发有多个目标。首先，最重要的目标是使 UML 作为一个通用的建模语言，可供所有建模者使用。它并非某人专有，且建立在计算机界普遍认同的基础上，即它包括了各种主要的方法并可作为它的建模语言。至少，希望它能够替代 OMT、Booch、Objectory 方法以及参与 UML 建议制订的其他人使用的方法建立的模型。其次，希望 UML 采用源自 OMT、Booch、Objectory 及其他主要方法的表示法，即尽可能地支持设计工作，如封装、分块、记录模型构造思路。此外，希望 UML 准确表达当前软件开发中的热点问题，如大规模、分布、并发、方式和团体开发等。

UML 并不试图成为一种完整的开发方法。它不包括一步一步的开发过程。一个好的软件开发过程对成功地开发软件至关重要，在这里向读者推荐一本书 *Jacobson-99*。UML 和使用 UML 的软件开发过程是两回事，这一点很重要。希望 UML 可以支持所有的，至少是目前现有的大部分软件开发过程。UML 包含了所有的概念，这些概念对于支持基于一个健壮的构架解决用例驱动的需求的迭代式开发过程是必要的。

UML 的最终目标是在尽可能简单的同时能够对实际需要建立的系统的各个方面建模。UML 需要有足够的表达能力以便可以处理现代软件系统中出现的所有概念，例如并发和分布以及软件工程中使用的技巧，如封装和组件。它必须是一个通用语言，像任何一种通用程序设计语言一样。然而，这样就意味着 UML 必将十分庞大，不可能像描述软件系统那样简单。现代语言和操作系统比起 40 年前要复杂得多，因为对它的要求越来越多。UML 提供了多种模型，这不是在一天之内就能够掌握的。它比先前的建模语言更复杂，因为它更全面。但是不必一下就完全学会它，就像学习任何一种程序设计语言、操作系统或是复杂的应用软件一样。

1.3.2 UML 的概念域

UML 的概念和模型可以分成以下几个概念域。

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 的静态组件称为静态视图。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。对象包含的信息作为属性，它执行的行为作为操作。多个类通过泛化处理可以具有一些共同的结构。子类在继承它们共同的父类的结构和行为的基础上增加了新的结构和行为。对象之间也具有运行时间连接，这种对象之间的关系称为类间的关联。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。另一类关系包括用例和数据流的合并。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，它的行为很像在每种类元上具有一定限制的类。

笔记

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。视图点将数据结构、控制流和数据流在一个视图中统一起来。相互操作和协作用顺序图和协作图描述。对所有行为视图起指导作用的是一组用例，每个用例描述了一个用例参与者或系统外部用户可见的一个功能。

3. 实现构造

UML 模型既可用于逻辑分析，又可用于物理实现。某些构件代表了实现。构件是系统中物理上的可替换部分，它按照一组接口设计并实现，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，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。

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，建模信息必须划分成连贯的部分，以便工作小组能够同时在不同部分上工作。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，它可以用于存储、访问控制、配置管理以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架构要求。

5. 扩展机制

无论一种语言能够提供多么完善的机制，人们总是还想扩展它的功能。UML 已经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

笔记

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。

1.4 表达式和图表语法

本书列举了许多演示实际模型的表达式和图表，以及表达式的语法和图表的注释。为了尽量避免混淆解释说明和实例，本书采用了一些约定的格式。

在图表和文本表达式中实际的表示法部分用 ComicSans 字体印刷。例如，模型中出现的 Helvetica 字体的类名是一个合法的名称。语法表达式中的括弧是一个可能出现在实际表达式中的括弧，它不是实际语法结构的一部分，例如 Order.create（customer，amount）在连续的文中，关键词和模型元素名都用 ComicSans 字体印刷，如 Order 或 Customer。

在一个语法表达式中，句法单元名可以被实际的一段文字用蓝色 ComicSans 字体替代，如 name。表达式中的黑色正文表示出现在目标图示上字面上的值。斜体或下画线说明替换文本具有给定的性质。例如：

name.operation（argument，...）

object-name：class

在语法表达式中，下标和上画线用于指示某种语法性质。例如：

expressionopt 是任选的。

expressionlist，用逗号分隔一系列表达式。如果出现零个或者一个重复符号，则不需要分隔符。每个重复符号都要用一个单独的替换符号。如果一个除逗号之外的标点符号出现在下标中，则它是分隔符。

expressionlist 用上画线连接两个或多个属于同一单元的可选的或重复出现的项目。在这个例子中，等号和表达式构成一个可以使用或省略的单元。如果只有一个项目，就可以不用上画线。

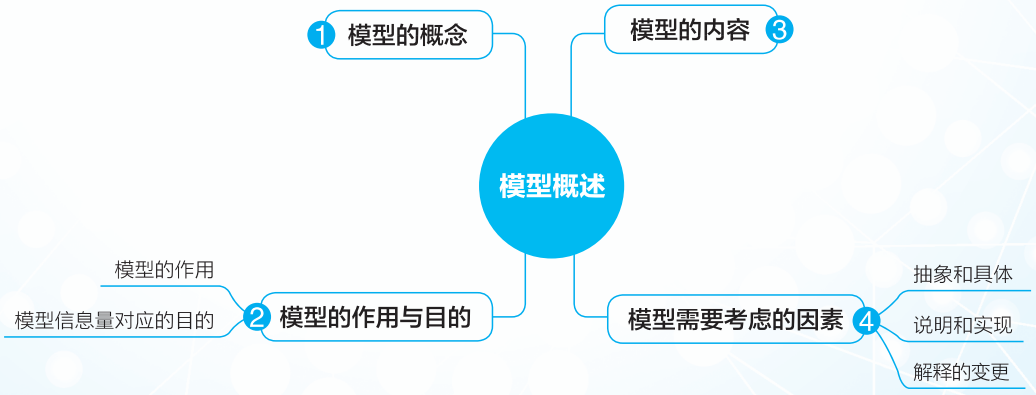
在图表中，中文楷体、蓝色的文字与箭头是注释，它是解释性说明，而不是实际表示法的一部分。其他文字和符号是实际表示法的一部分。

第 2 章 模型概述

学习目标

1. 了解模型的概念和作用。
2. 掌握模型的内容。

知识导图



笔记

2.1 模型的概念

模型是用某种工具对同类或其他工具的表达方式。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。工程、建筑和其他许多需要具有创造性的领域中都使用模型。

表达模型的工具要求便于使用。建筑模型可以是图纸上所绘的建筑图，也可以是用厚纸板制作的三维模型，还可以用存于计算机中的有限元方程来表示。一个建筑物的结构模型不仅能够展示这个建筑物的外观，还可以用它来进行工程设计和成本核算。

软件系统的模型用建模语言来表达，如 UML。模型包含语义信息和表示法，可以采取图形和文字等多种不同形式。建立模型的目的是因为在某些用途中模型使用起来比操纵实物更容易和方便。

2.2 模型的作用与目的

2.2.1 模型的作用

1. 模型有多种用途

精确表达项目的需求和捕捉应用领域的知识，以使各方面的利益相关者能够理解并达成一致。

建筑物的各种模型能够准确表达出这个建筑物的外观，周边交通和服务设施，抗风和抗震性能，消费及其他需求。各方面的利益相关者则包括建筑设计师、建筑工程师、合同缔约人、各个子项目的缔约人、业主、出租者和市政当局。

软件系统的不同模型可以捕获关于这个软件的应用领域、使用方法、试题手段和构造模式等方面的需求信息。各方面的利益相关者包括软件结构设计师、系统分析员、程序员、项目经理、顾客、投资者、最终用户和使用软件的操作员。在 UML 中要使用各种各样的模型。

2. 进行系统设计

建筑设计师可以用画在图纸上的模型图、存于计算机中的模型或实际的三维模型使自己的设计结果可视化，并用这些模型来做设计方面的试验。建造、修改一个小型模型比较简单，设计人员不需花费什么代价就可以进行创造和革新。

在编写程序代码前，软件系统的模型可以帮助软件开发人员方便地研究软件的多种构架和设计方案。在进行详细设计前，一种好的建模语言可以让设计者对软件的构架有全面的认识。

3. 使具体的设计细节与需求分开

建筑物的某种模型可以展示出符合顾客要求的外观，另一类模型可以说明建筑物内部的电气线路、管线和通风管道的设置情况。实现这些设置有多种方案。最后确定的建筑模型一定是建筑设计师认为最好的一个设计方案。顾客可以对此方案进行检查验证，但通常顾客对具体的设计细节并不关心，只要能满足他的需要即可。

软件系统的一类模型可以说明这个系统的外部行为和系统中对应于真实世界的有关信息，另一类模型可以展示系统中的类以及实现系统外部行为特性所需要的内部操作。实现

这些行为有多种方法。最后的设计结果对应的模型一定是设计者认为最好的一种。

笔记

4. 生成有用的实际产品

建筑模型可以有多种相关产品，包括建筑材料清单、在各种风速下建筑物的偏斜度、建筑结构中各点的应力水平等。

利用软件系统的模型，可以获得类的声明、过程体、用户界面、数据库、合法使用的说明、配置草案以及与其他单位技术竞争情况的对比说明。

5. 组织、查找、过滤、重获、检查以及编辑大型系统的有关信息

建筑模型用服务设施来组织信息：建筑结构、电器、管道、通风设施、装潢等。除非利用计算机存储，否则对这些信息的查找和修改没那么容易。相反，如果整个模型和相关信息均存储在计算机中，则这些工作很容易进行，并且可方便地研究多种设计方案，这些设计方案共享一些公共信息。

6. 软件系统用视图组织信息

软件系统用视图组织信息：静态结构视图、状态机视图、交互视图、反映需求的视图等。每一种视图均是针对某一从模型中挑选的一部分信息的映射，没有模型管理工具的支持不可能使模型做得任意精确。一个交互视图编辑工具可以用不同的格式表示信息，可以针对特定的目的隐藏暂时不需要的信息并在以后再展示出来，可以对操作进行分组、修改模型元素以及只用一个命令修改一组模型元素等。

7. 研究多种设计过程中的解决方案

对同一建筑的不同设计方案的利弊在一开始可能不是很清楚。例如，建筑物可以采用的不同的子结构彼此之间可能有复杂的相互影响，建筑工程师可能无法对这些影响做出正确的评价。在实际建造建筑物以前，利用模型可以同时研究多种设计方案并进行相应的成本和风险估算。

通过研究一个大型软件系统的模型可以提出多个实际方案并对它进行相互比较。当然模型不可能做得非常精细，但即使是一个粗糙的模型也能够说明在最终设计中所要解决的许多问题。利用模型可以研究多种设计方案，所花费的成本只是实现其中一种方案所花费的成本。

利用模型可以全面把握复杂的系统。一个关于龙卷风袭击建筑物的工程模型中的龙卷风不可能是真实世界里的龙卷风，仅仅是模型而已。真正的龙卷风不可能呼之即来，并且它会摧毁测量工具。许多快速，激烈的物理过程现在都可以运用这种物理模型来研究和理解。

一个大型软件系统由于其复杂程度可能无法直接研究，但模型使之成为可能。在不损失细节的情况下，模型可以抽象到一定的层次以使人能够理解。可以利用计算机对模型进行复杂的分析以找出可能的“问题点”，如时间错误和资源竞争等。在对实物做出改动前，通过模型研究系统内各组成部分之间的依赖关系可以得出这种改动可能会带来哪些影响。

2.2.2 模型信息量对应的目的

针对不同目的，模型可以采取各种形式及不同的抽象层次。模型中所包含的信息量必须对应于以下几种目的。

1. 指导设计思路

在项目早期所建立的高层模型用于集中利益相关者的思路和强调一些重要的选择方

笔记

案。这些模型描述了系统的需求并代表了整个系统设计工作的起点。早期的模型帮助项目发起者在把精力放在系统的细节问题之前研究项目可能的选择方案。随着设计工作的进展，早期模型由更为精确的模型所替代。没有必要详细保存早期研究过程中的种种选择方案和返工情况。了解早期模型的目的是帮助获得思路。但最后得到的“思路模型”要在进行详细设计前记录下来。早期模型不需要达到实现阶段的模型的精确程度，无须涉及有关系统实现的一套概念。建立这种模型只使用了 UML 定义的组件的一个子集，比后期的设计阶段的模型使用的组件要少得多。

当早期模型发展到具有一定精度的完整的视图模型时，例如，分析系统需求的模型，那么要在开发过程进入下一阶段时将其保存下来。不断向模型中添加信息的增量式开发（在这种情况下开发的推理过程也要保存和记录）与针对“死端点”进行研究直到得出正确的解决方案的随意漫步式开发之间有重要的区别。后一种情况通常使人不知怎样，并且根本没有必要对整个开发过程进行记录保存，除非遇到特殊情况需要对开发过程进行回溯。

2. 系统基本结构的抽象说明

在分析阶段和初步设计阶段建立的模型以关键概念和最终系统的各种机制为中心。这些模型以某种方式与最终系统匹配。但是，模型中丧失了细节信息，在设计过程中必须显式地补充这些信息。使用抽象模型的目的是在处理局部细节问题前纠正系统高层次的普遍问题。通过一个仔细的开发过程，这些模型可以发展成最终模型，该过程保证最终获得的模型能够正确实现初期模型的设计意图。必须具备跟踪能力来跟踪从基本模型到完备模型的过程，否则无法保证最终系统正确包含了基本模型所要表达的关键特性。基本模型强调语义，它不需要涉及系统实现方面的细节。有时确实会出现这种情况：在低层实现方面的差别会使逻辑语义模糊不清。从基本模型到最后的实现模型的途径必须清晰、简明，不论这个过程是由代码生成器自动实现，还是由设计者人工实现。

3. 最终系统的详细规格说明

系统实现模型包含能够建造这个系统的足够信息，它不仅包括系统的逻辑语义、语法、算法、数据结构和确保能正确完成系统功能的机制，还包括系统制品的组织决定，这些制品对个人之间的相互协作和使用辅助工具来说十分必要。这种模型必须包括对模型元素进行打包的组件，以便理解和用计算机自动处理。这不是目标应用系统的特性，而是系统构造过程应具有的特性。

4. 典型或可能的系统范例

精心挑选的实例可以提高人的观察能力，并使系统的说明和实现有实际效果。然而，即使有非常多的例子，也达不到一段详细定义所起的效果。最终希望的是要让模型能够说明一般的情形，这也是程序代码要做的事情。不过，典型的数据结构、交互顺序或对象生命历程的例子对于理解复杂系统很有益处，必须小心使用。从逻辑上来说，从一大堆特例中归纳出一般规律是不可能的，但是大部分人思考某一问题时总是首先考虑一些精心挑选出来的有关该问题的例子。范例模型仅是模型的示例而不带有一般性的描述，因此，人们会觉得这两种模型之间有差异。范例模型一般只使用 UML 定义的组件的子集。说明型模型和范例模型在建模中都很有用。

5. 对系统全面的或部分的描述

模型可以完全描述一个独立系统，并且不需要参考外部信息。更通常的情况是，模型

是用相互区别的、不连续的描述单元组织起来的，每个单元作为整体描述的一部分可以被单独进行存储和操纵。这种模型带有必须与系统其他模型联系在一起的散件，因为这些散件具有相关性和含义，因此它能够与其他散件通过各种方式结合来构造不同的系统。获得重用是一个好的建模方法的重要目标。

模型要随时间发展变化。深度细化的模型源于较为抽象的模型，具体模型源于逻辑模型。例如，开始阶段建立的模型是整个系统的一个高层视图。随着时间的推进，模型中增加了一些细节内容并引入了一些变化。之后，模型的核心焦点从一个以用户为中心的前端逻辑视图转变成了一个以实现为中心的后端物理视图。随着开发过程的进行和对系统的深入理解，必须在各种层次上反复说明模型以获得更好的理解，用一个单一视角或线性过程理解一个大型系统是不可能的。对模型的形式无所谓“正确”和“错误”之分。

笔记

2.3 模型的内容

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）。

语义方面用一套逻辑组件表达应用系统的含义，如类、关联、状态、用例和消息。语义模型元素携带了模型的含义，它表达了语义。语义建模元素用于代码生成、有效性验证、复杂度的度量等，其可视化的外观与大多数处理模型的工具无关。语义信息通常称作模型。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。这些方面通常分别加以描述（定义 UML 的文献即如此），但它们紧密相关，并且是同一模型的一部分。

可视化的表达方式以可使人观察、浏览和编辑的形式展示语义信息。表示方式元素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。它并未增添新的语义，而是用一种有用的方式对表达式加以组织，以强调模型的列。因此它对模型的理解起指导作用。表达式元素的语义来自语义模型元素。但是，由于是由人来绘制模型图，所以表达式元素并不是完全来自模型的逻辑元素。表达式元素的排列可能会表达出语义关系的另外含义，这些语义关系很不明显或模棱两可，以至于在模型中不能形式化地表达，但可给人启迪。

上下文（语境）。模型自身是一个计算机系统的制品，可应用在一个给出了模型含义的大型语境中。该语境包括模型的内部组织、整个开发过程中对每个模型的注释说明、一个缺省值集合、创建和操纵模型的假定条件以及模型与其所处环境之间的关系。

模型需要有内部组织，允许多个工作小组同时使用某个模型而不发生过多的相互干涉。这种对模型的分解并不是语义方面所要求的——与一个被分解成意义前后连贯的多个包的模型相比，一个大的单块结构的模型所表达的信息可能会同样精确，因为组织单元的边界确定会使准确定义语义的工作复杂化，故这种单块模型表达的信息可能比包结构的模型表达得更精确。但是，要想使工作在一个大的单块模型上的多个工作组不彼此相互妨害是不可能的。其次，单块模型没有适用于其他情况的可重用的单元。最后，对大模型的某些修改往往会引起意想不到的后果。如果将模型适当分解成具有良好接口的小的子系统，那么对其中一个小的、独立的单元进行的修改造成的后果可以跟踪确定。不管怎样，将大系统分解成由精心挑选的单元构成的层次组织结构，是人类千百年来发明的设计大系统的方法中最可靠的方法。

笔记

模型捕获一个应用系统的语义信息，还需记录模型自身开发过程中的各种信息，如某个类的设计者、过程的调试状态和对各类人员的使用权限的规定。这些信息至多是系统语义的外围信息，但对开发过程非常重要。因此，建立一个系统的模型必须综合考虑两方面。最简便的实现方法是将项目管理信息作为注释加入语义模型中，即可以对模型元素用非建模语言进行任意描述。在 UML 中用文本字符串来表示注释。

文本编辑器或浏览器所使用的命令不是程序设计语言的一部分，同样，用于创建和修改模型的命令也不是建模语言语义的一部分。模型元素的属性没有缺省值，在一个特定的模型中，它均具有值。然而，对于实际的开发过程，人们要求建立与修改模型时无须详细说明有关的所有细节。缺省值存在于建模语言和支持这种语言的建模工具的边界处。在建模工具所使用的创建模型的命令中，它是真正的缺省值，尽管它可能会超过单个的建模工具并如用户所期望的那样成为建模工具所使用的通用语言。

模型不是被孤立地建造和使用的，它是大环境的一部分，这个大环境包括建模工具、建模语言和语言编译器、操作系统、计算机网络环境、系统具体实现方面的限制条件等。系统信息应该包括环境所有方面的信息。其中的一部分信息应保存在模型中，即使这些信息不是语义信息，例如项目管理注释（在上面已经讨论过）、代码生成提示、模型的打包、编辑工具缺省命令的设置。其他方面的信息应分别保存，如程序源代码和操作系统配置命令。即使是模型中的信息，对这些信息的解释也可以位于多个不同方面，包括建模语言、建模工具、代码生成器、编译器或命令语言等。本书用 UML 自身对模型进行解释。但是当进行系统的具体物理实现时，要用到其他用于解释的资源，这些资源对 UML 来说是不可见的。

2.4 模型需要考虑的因素

模型是一个系统潜在配置的发生器，系统是它的范围或值。按照模型进行系统配置是一种理想化的情况。然而，有时模型要求的种种条件在现实中无法实现。模型还是对系统含义和结构的一般性说明。这种描述是模型的范围或含义。模型具有一定的抽象层次，它包含了系统中最基本的成分而忽略了其他内容。对模型来说，有以下三方面需要考虑。

2.4.1 抽象和具体

模型包含了系统的基本成分而忽略了其他内容。哪些是基本内容需要根据建模的目的判定。这不是说要把模型所含信息的精度一分为二，即只有精确和不精确两种情况。可能会存在一系列表达精度不同，但逐渐提高的模型。建模语言不是程序设计语言。建模语言所表达的内容可能很不精确，因为多余的详细内容与建模目的无关。具有不同精度级别的模型可应用于同一个项目的各个阶段。用于程序设计编码的模型要求必须与程序设计语言有关。典型的情况是，在早期分析阶段使用高层次的，表达精度低的模型。随着开发过程的深入，所用的模型越来越细化，最终使用的模型包含了大量的细节内容，具有很高的精度。

2.4.2 说明和实现

笔记

一个模型可以告诉我们“做什么”（说明），也可以告诉我们“一个功能是如何实现的，即怎么做”（实现）。建模时要注意区分这两方面。在花大量时间研究怎么做之前很重要的一点是，要正确分析出系统要做什么。建模的一个重要侧面是对具体实现细节进行抽象。在说明和实现之间可能有一系列关系，其中在某层次中的说明就是对它上一层次的实现。

阐述和举例。模型主要是描述性的。模型所描述的是一个个实例，这些实例仅是作为例子才出现在模型中。大部分实例仅在运行的一部分时间中才存在。有时，运行实例自身是对其他事物的描述，这些混杂对象称作元模型（Meta Model）。从更深层次地看，认为任何事物不是描述性的，就是实例性的，这种观点是不符合实际情况的。某一事物是实例，还是描述不是孤立区分的，与观察角度有关，大部分事物都可以从多种角度考察。

2.4.3 解释的变更

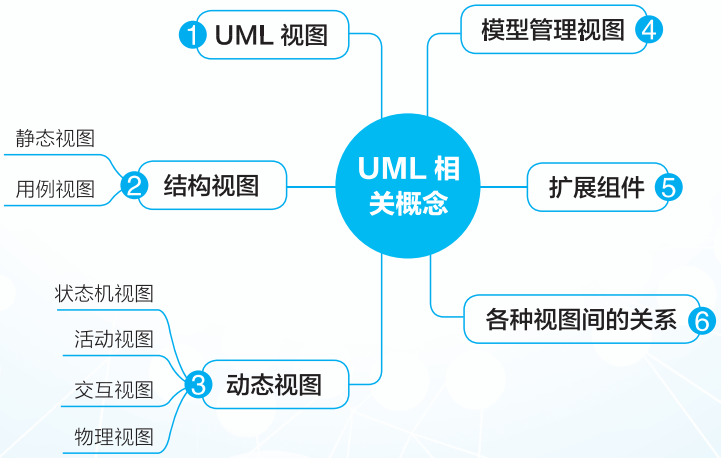
用一种建模语言对模型可能有多种解释。可以定义语义变更点（semantic variation point）——可能会出现多种解释的地方——给每个解释一个语义变更名，以便可以标识究竟使用了哪种解释。例如，Smalltalk 语言的使用者要避免在系统实现模型中出现多重继承关系，因为 Smalltalk 语言不支持多重继承。而其他程序设计语言使用者可能会在模型中使用多重继承。

第 3 章 UML 相关概念

学习目标

1. 了解 UML 的扩展组件。
2. 掌握 UML 的各种视图及其关系。

知识导图



笔记

3.1 UML 视图

UML 中的各种组件和概念之间没有明显的界限，但为方便起见，可以用视图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。用视图划分带有一定的随意性，但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可视化地表示视图中的各种概念。

在最上层，视图可划分成三个视图域：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组件，这些扩展能力有限但很有用。这些组件包括约束、构造型和标记值，适用于所有的视图元素。

UML 视图和视图所包括的图以及与每种图有关的主要概念，如表 3-1 所示。不能把这张表看成一套死板的规则，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
结构分类	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、参与者、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、位置
动态行为	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	报、子系统、模型
可扩展性	所有	所有	约束、构造型、标记值

3.2 结构视图

3.2.1 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。这种视图之所以称之为是静态的是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、泛化和各种依赖关系，如使用和实现关系。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。

笔记

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。

售票系统的类图是售票系统领域模型的一部分，如图 3-1 所示，图中表示了几个重要的类，如 Customer、Reservation、Ticket 和 Performance。顾客可多次订票，但每次订票只能由一个顾客执行。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、日期和时间标识。

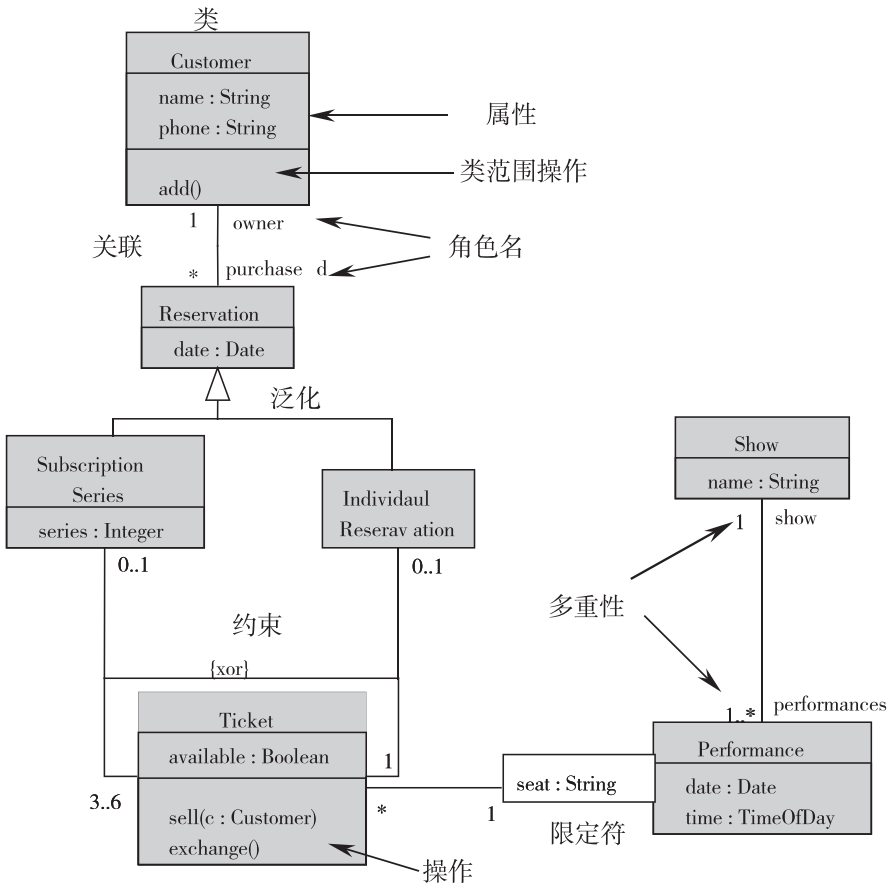


图 3-1　售票系统的类图

3.2.2 用例视图

用例视图是被称为参与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，可以被描述为参与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参与者，并显示哪个参与者参与了哪个用例的执行。

如图 3-2 所示是售票系统的用例图。参与者包括售票员、监督员、信用卡服务商和公用电话亭。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，顾客不是参与者，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、购票和预约票（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

笔记

一个共同的部分，即通过信用卡付钱（对售票系统的完整描述还要包括其他一些用例，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，用例作为交互图中的一次协作实现。

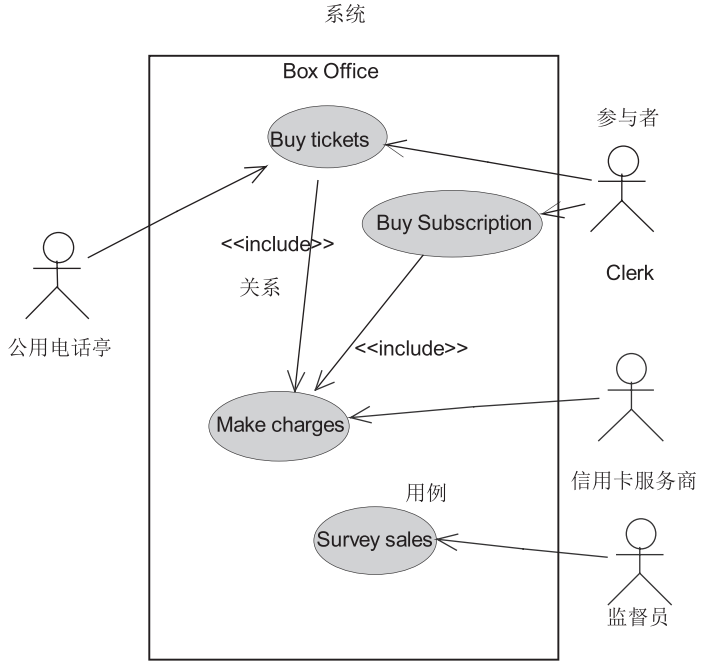


图 3-2　售票系统的用例图

实现视图又称组件视图，主要用来显示代码组件的组织方式，主要描述了实现模块和它们之间的相互关系。

部署视图是用来显示系统的物理结构，也就是系统的物理展开。例如，计算机和设备以及它们之间的连接方式，其中计算机和设备都称为节点。

3.3 动态视图

3.3.1 状态机视图

状态机视图是一个类对象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种状态。与转换相关的活动执行时，转换也同时发生。状态机用状态图表达。

图 3-3 所示是票的状态图。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。当顾客预定票时，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。如果超过指定的期限顾客仍未做出选择，此票就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，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周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。

笔记

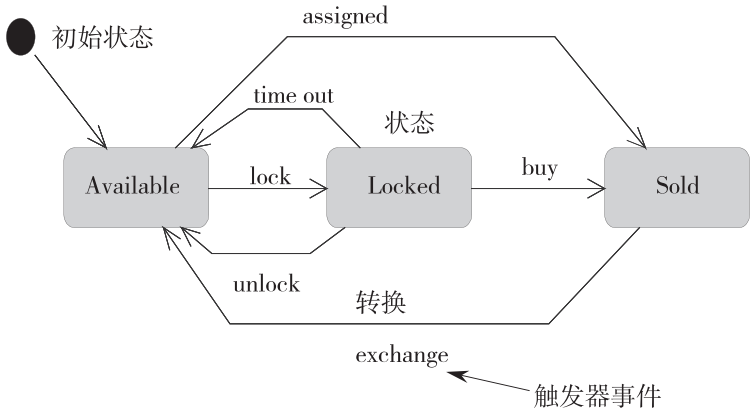


图 3-3　票的状态图

3.3.2 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。活动视图用活动图体现。

图 3-4 所示为售票处的活动图。它表示了上演一个剧目所要进行的活动。箭头说明活动间的顺序依赖关系。例如，在规划进度前，首先选择演出的剧目。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。

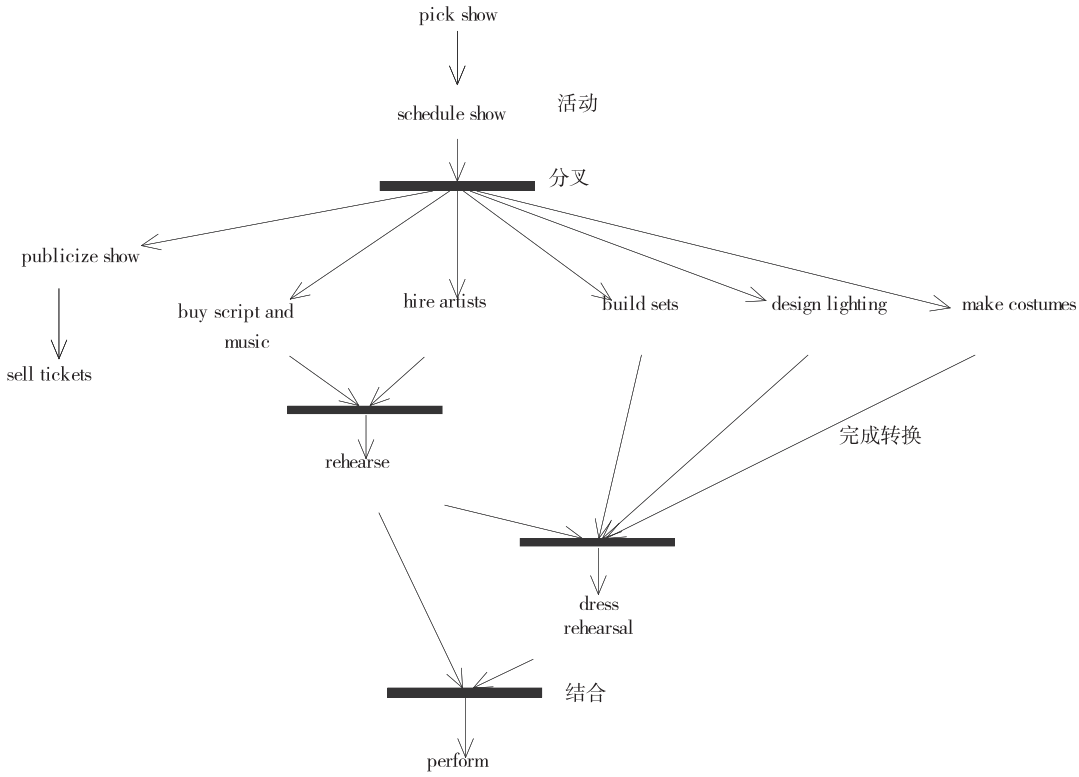


图 3-4　售票处的活动图

笔记

这个例子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必需的消息传送细节。

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。

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。每个类元角色用一条生命线表示，即用垂直线代表整个交互过程中对象的生命周期。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，即一个事物的历史过程。

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一个类操作或状态机中引起转换的触发事件。

图 3-5 所示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。

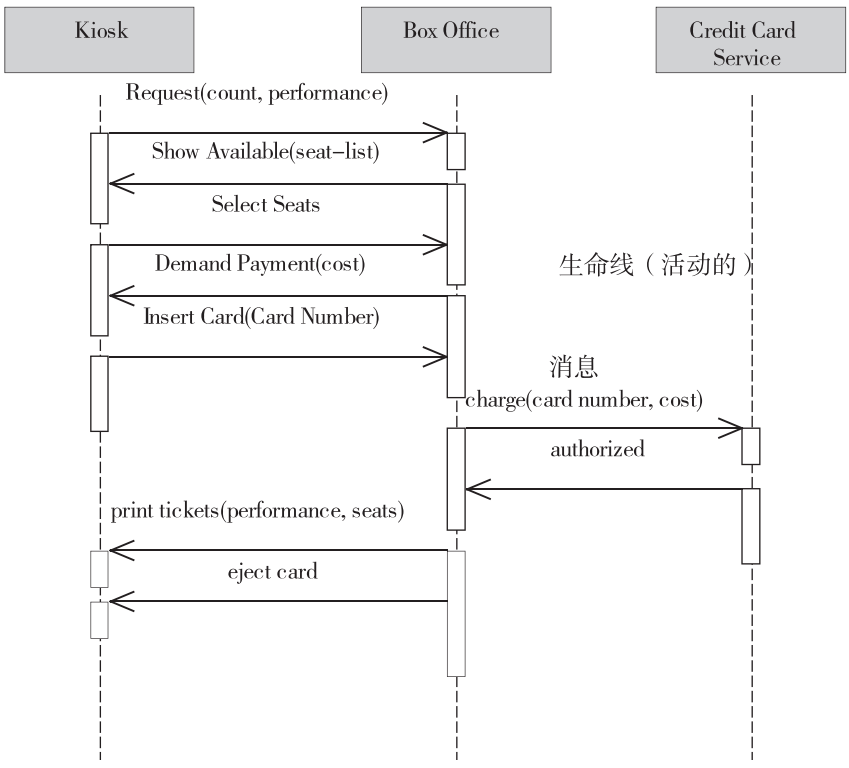


图 3-5　购票的顺序图

笔记

2. 协作图

协作图是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时才有意义。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。协作图用几何排列表示交互作用中的各角色，如图 3-6 所示。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，消息编号对应程序中的嵌套调用结构和信号传递过程。

图 3-6 所示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，在交互结束时丢弃。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。

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表示角色之间的关系，并用消息说明这些关系。在实际中可以根据需要选用这两种图。

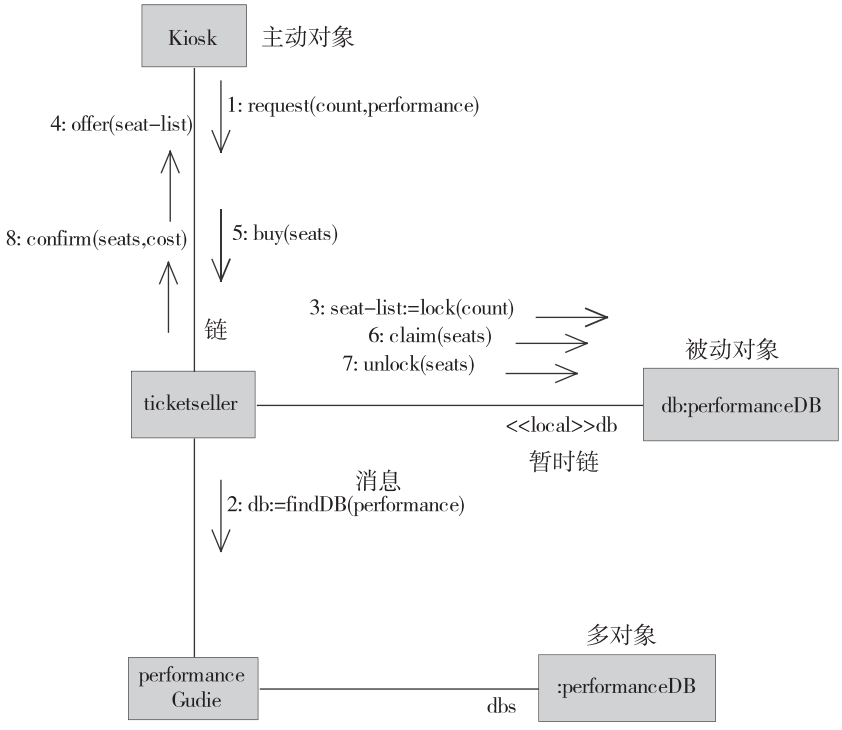


图 3-6　订票交互的协作图

笔记

3.3.4 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。物理视图有实现视图和部署视图两种。

实现视图为系统的构件进行建模，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。

实现视图用构件图表现。图 3-7 所示是售票系统的构件图。图中有三个用户接口：顾客与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以及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。

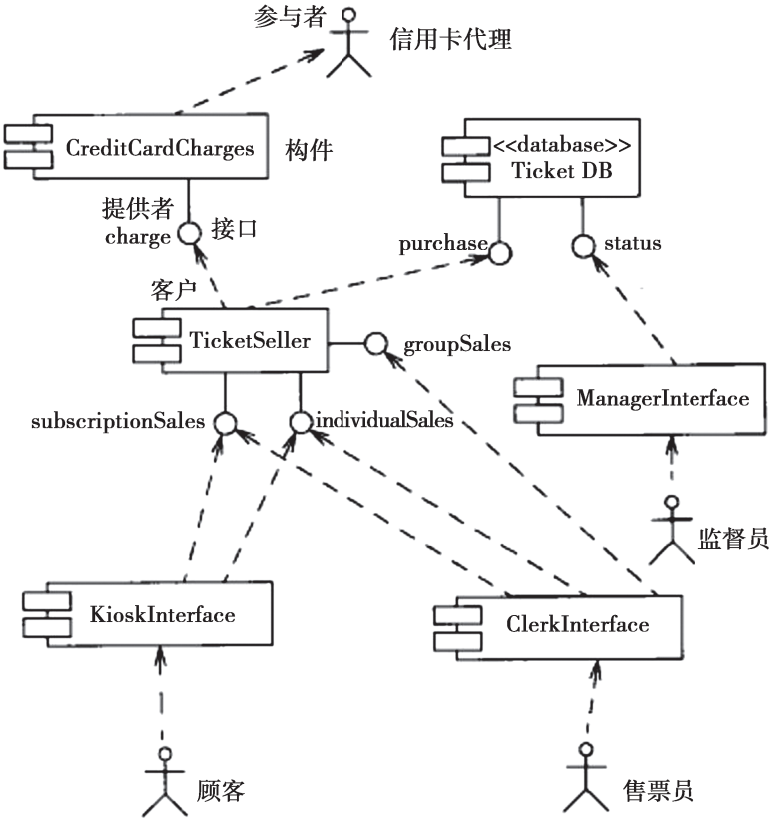


图 3-7　售票系统的构件图

图 3-7 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服务（列在接口旁）。从构件到接口的虚线箭头表示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。

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

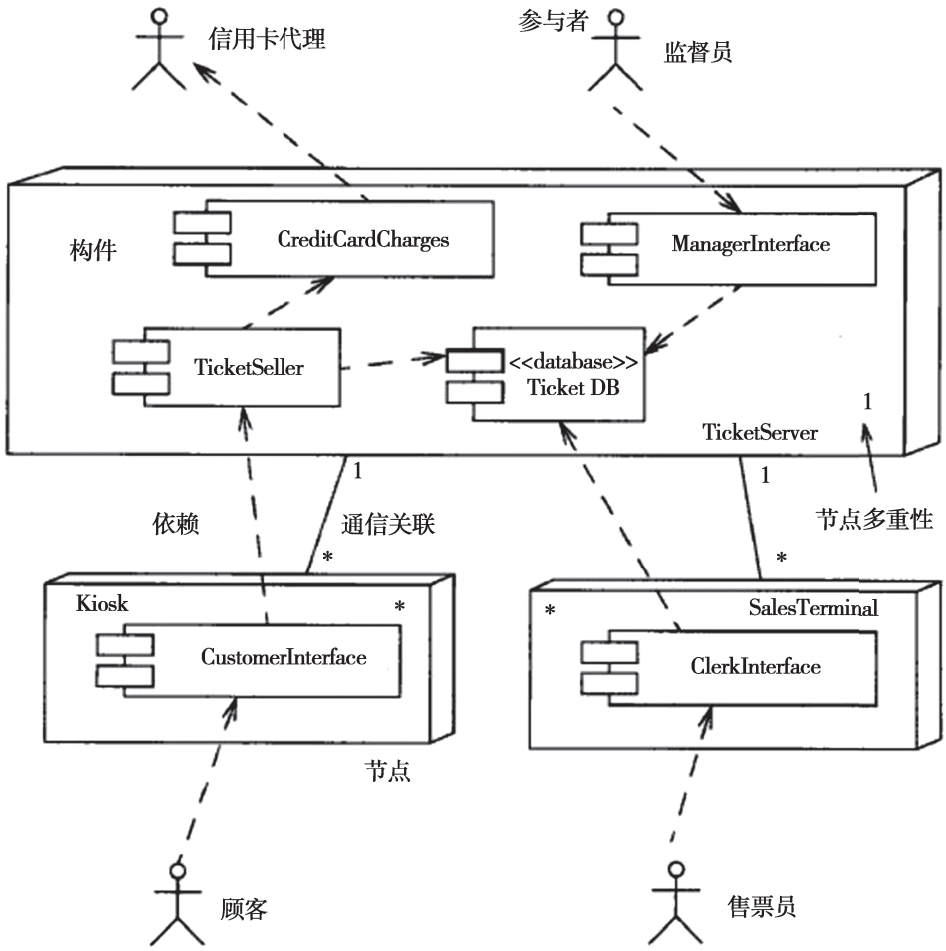


图 3-8　售票系统的（描述层）部署图

图 3-9 所示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 中描述层中的内容相互对应的。

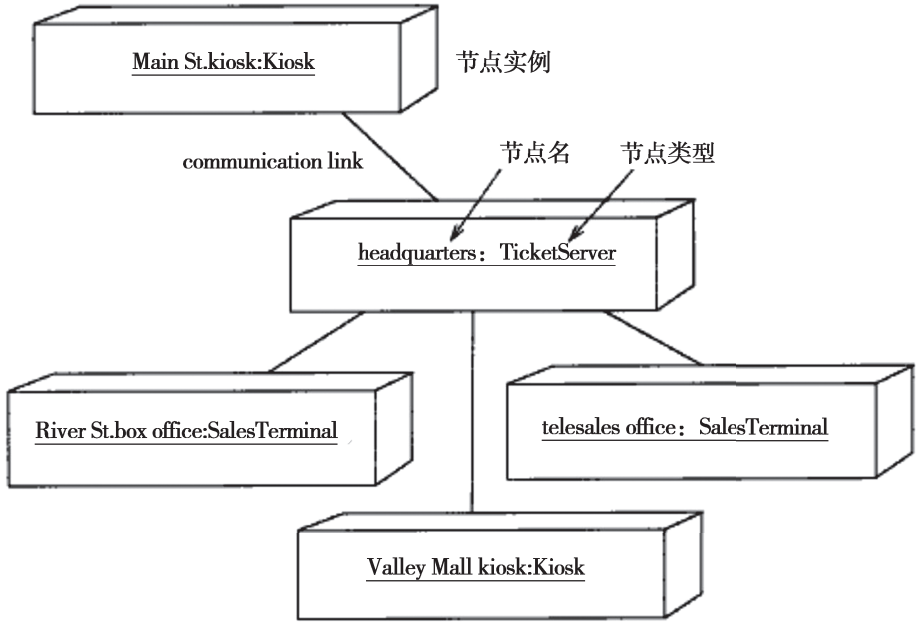


图 3-9　售票系统的（实例层）部署图

笔记

3.4 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。它代表系统的一个部分，有清晰的接口，这个接口可作为一个单独的构件实现。

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。每个子系统还包含多个包。

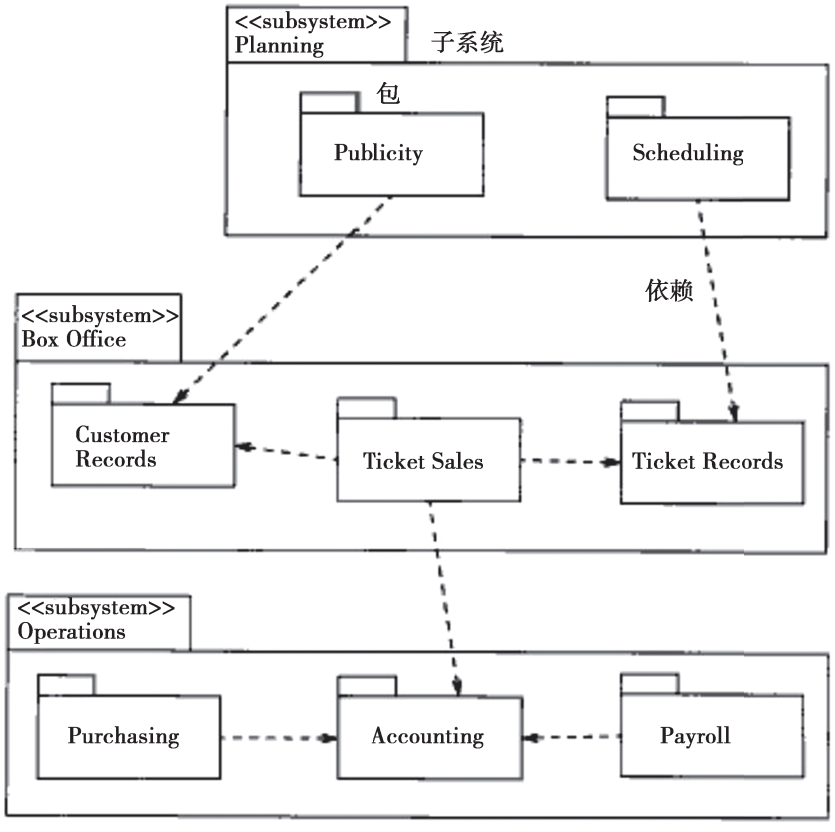


图 3-10　包和它们之间的依赖关系

3.5 扩展组件

UML 主要包含三种扩展组件：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。

如图 3-11 所示举例说明了约束、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 3-11 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 的概念不支持直接文字描述。

TicketDB 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 3-11 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 要在年底前完成计划的制订。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。

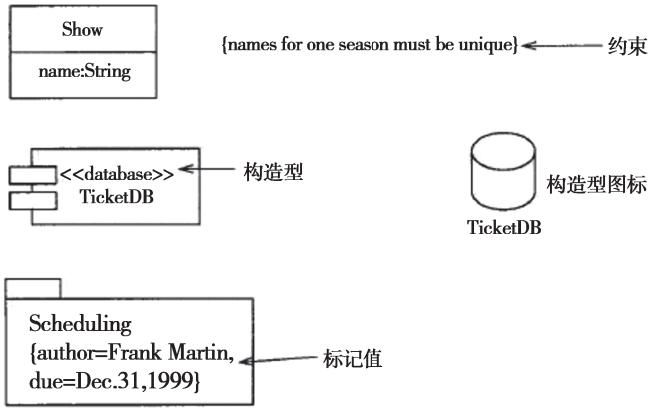


图 3-11　扩展组件

3.6 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。表中没有列出所有关系，但它们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　不同视图元素间的关系

元素 1	元素 2	关系
类	拥有	状态机
操作	交互	实现
用例	合作	实现
用例	交互实例	样本场景
构件实例	节点实例	位置
动作	操作	调用
动作	信号	发送
活动	操作	调用
消息	动作	激发
包	类	拥有
角色	类	分类