统一建模语言参考手册
—基本概念

翻译：*Adams Wang*

The Unified Modeling Language Reference Manual

James Rumbaugh
Ivar Jacobson
Grady Booch

译者序

统一建模语言（UML）是一种直观化、明确化、构建和文档化软件系统产物的通用可视化建模语言。UML 由面向对象领域的三位大师：Grady Booch、Jim Rumbaugh 和 Ivar Jacobson 于 1996 年发布，并提交给 OMG。UML 于 11 月为 OMG 所采用，现已成为业界标准。

本文讲述了 UML 基本概念，为 UML 的深入理解提供一个起点。全文共分为十四个章节。

*UML 概述*对 UML 语言、目标以及概念作了简略的介绍。

第二章至第十章展示了 UML 的各种视图，同时显示了各种结构如何配合工作。*UML 一览*中，以一个简单的例子开始，对 UML 的视图、概念作了简单的介绍。然后，在后续的章节中（*静态视图*、*用例视图*、*状态机视图*、*活动视图*、*交互视图*、*物理视图*、*模型管理视图*、*扩展机制*）对各个视图进行详细的讨论。它们分别从静态建模机制、动态建模机制、模型管理机制以及扩展机制对 UML 进行探讨。

*UML 标准元素*讨论了一些与核心概念的区别较小或者重要性不足以被包括至 UML 核心概念的元素。原文中仅有标准元素表。为了便于大家的理解，它们以 UML 的相关核心元素组织起来，并提供了对应核心元素的解释。

*OMG 建模词汇表*引自 UML 规范，原文中未有相关内容。英语原文可参见 www.rational.com的 UML Specification 1.3。

*中英文词汇对照*提供了所有文中术语的词汇对照，详细、精确的解释可以参见 UML 规范相应的内容。

最后，*UML 标记一览*展现了 UML 各种概念的标记符号。

为了忠实原文，仅在 *UML 标准元素*中添加了相应的核心元素描述，以及增加了 *OMG 建模词汇表*，以方便对 UML 概念的理解。

文中释译不当之处，肯请各位批评指正。

译者：Adams Wang
2000 年 11 月

目录

UML 概述（*UML OVERVIEW*）..... 1

UML 简述..... 1
UML 目标..... 1
UML 概念范围..... 2

UML 一览（*UML WALKTHROUGH*）..... 5

UML 视图..... 5
静态视图..... 6
用例视图..... 7
交互视图..... 8
顺序图..... 8
协作图..... 9
状态机视图..... 10
活动视图..... 11
物理视图..... 12
模型管理视图..... 15
扩展结构..... 16
视图间的联系..... 17

静态视图（*STATIC VIEW*）..... 19

概述..... 19
分类..... 19
关系..... 22
关联..... 23
概括..... 25
继承..... 26
多重继承..... 27
分类和多重分类..... 27
静态和动态分类..... 27
实现..... 28
依赖..... 29
约束..... 30
实例..... 31
对象图..... 32

用例视图（*USE CASE VIEW*）..... 33

概述..... 33
活动者..... 33
用例..... 34

状态机视图（*STATE MACHINE VIEW*）..... 37

概述 37
状态机 37
事件 37
状态 39
迁移 39
复合状态 42

活动视图（*ACTIVITY VIEW*） 47

概述 47
活动图 47
活动和其它视图 48

交互视图（*INTERACTION VIEW*） 51

概述 51
协作 51
交互 51
顺序图 52
激活 53
协作图 53
模式 55

物理视图（*PHYSICAL VIEW*） 57

概述 57
构件 57
结点 58

模型管理视图（*MODEL MANAGEMENT VIEW*） 59

概述 59
包 59
包的依赖 59
访问和引入依赖 60
模型和子系统 61

扩展机制（*EXTENSION MECHANISMS*） 63

概述 63
约束 63
标签值 64
版型 65
剪裁 UML 65

UML 标准元素（*UML STANDARD ELMENTS*） 67

概述 67
核心概念 67
元素 67
可概括元素 67
行为特征 67

属性 *67*
分类 *68*
关系 *69*
操作 *70*
注释 *70*
约束 *70*
版型 *70*
包 *70*
分类角色 *71*
关联角色 *71*
关联端点 *71*
调用事件 *71*
标准元素 72

OMG 建模词汇表（*OMG MODELING GLOSSARY*） 83

介绍 83
表示法的约定 83
建模词汇 84

中英文词汇对照 （*CHINESE-ENGLISH GLOSSARY COMPARSION*） 99

中文顺序对照表 99
英文顺序对照表 101

UML 标记一览（*UML NOTATION SUMMARY*） 105

UML 概述（*UML Overview*）

UML 简述

统一建模语言（UML）是一种直观化、明确化、构建和文档化软件系统产物的通用可视化建模语言。它捕捉了被构建系统的有关决策和理解，用来理解、设计、浏览、配置、维护以及控制系统的信息。UML 可以与所有的开发方法、生命阶段、应用领域和媒介一同使用。它意图统一过去建模技术的经验，将当前软件最佳实践合并至标准的方法。UML 包括语义概念、标记符号和指南，具有静态、动态、环境上的和组织性的部分。它可以被具有代码产生和报表生成的交互式可视建模工具所支持。UML 规范没有定义标准过程，但可用于迭代的开发过程，并支持现有的大多数面向对象的开发过程。

UML 捕捉系统静态结构和动态行为的信息。系统建模成独立对象的集合，它们互相交互以实现功能，从而最终使外部使用者获益。静态结构定义了对系统具有重要意义的各种对象和实现，以及它们之间的关系。动态行为定义了对象时间上的历史和为达成目标对象间的通讯。从不同但是相关的视角来对系统建模，允许了多种角度对系统的理解。

UML 还包括用包来分解模型的组织性结构，它允许软件团队将系统分解为可工作的单元，对包之间的依赖进行理解和在复杂的开发环境中管理模型单元的版本。它包含了表达实现上的决策和用构件来组织运行时元素的结构。

UML 不是编程语言。工具可以提供 UML 至各种编程语言的代码生成，以及可以从现有的程序逆向构筑模型。UML 不是用于定理证明的高度正式的语言。实际上有很多正式的语言，但它们不易理解或不适用于多种用途。UML 是通用性的建模语言。对于特定的领域，如 GUI 设计、VLSI 电路设计或基于规则的人工智能，更特定的语言和工具可能更加合适。UML 是离散的建模语言，它不打算对如工程和物理的连续系统建模。UML 是对诸如软件、硬件或数字逻辑的离散系统建模的通用语言。

UML 目标

在 UML 开发的背后，有许多目标。第一个且最重要的目标：UML 是所有建模人员可以使用的通用建模语言。它基于许多计算机团体的共识，是一种非私有的语言。UML 特意包含了主流建模方法的概念，从而可以作为它们的建模语言。它至少可以替代 OMT、Booch、Objectory，以及那些规范参与者的模型。在 UML 中，OMT、Booch、Objectory 和其它方法的标记尽可能的被使用，从而尽可能的为人们所熟悉。这同时意味着对良好设计实践如封装、问题划分、目标捕获的支持，它特意针对了许多当前软件开发的问题，如大规模、分布、并发、模式和团队开发等等。

UML 不是完整的开发方法。它不包括逐步的开发流程。我们相信对于软件开发而言，好的开发过程是非常关键的。认识到 UML 和使用 UML 的过程的不同是很重要的。现代迭代过程是基于建造强壮结构来解决用例驱动的需求，UML 包括了所有的必要概念。

最后一个 UML 目标是在能对众多系统建模的同时，尽可能的简洁。UML 具有足够的表

达能力来处理现代系统中的所有概念，如封装和构件。它必须是一种普遍性的语言，如同许多通用编程语言。不幸的是，这意味着如果我们想处理不仅仅是实验系统的事物，它的规模会较大。例如，现代语言和现代操作系统较 40 年前复杂了许多，因为我们对它们的要求也越来越高。UML 具有多种模型；它不是一朝一夕就可以被掌握的。因为它设计为更加全面的语言，所以较一些先行者更为复杂。然而，UML 也不必一次性的被掌握，如同我们在使用编程语言、操作系统或复杂的应用系统时情形一样。

UML 概念范围

UML 概念和模型可以被划分为以下的范围。

静态结构。精确的模型必须首先定义讨论的各种事物，即应用中的关键概念、它们的内部特征和相互之间的关系。该一系列构造是静态视图。应用概念建模成类，类描述了一系列拥有信息和相互通讯以实现行为的离散对象。对象所拥有的信息建模成属性；它们执行的行为建模成操作。多个类可以使用概括共享通用的结构。孩子类将新添的结构和行为增加至通过继承得到的结构和行为。对象还可以拥有与其它对象的运行时连接，上述对象—对象关系建模成类之间的关联。元素之间的一些关系用依赖来分组，包括抽象层次的转移、模板参数的绑定、许可的授予和元素对其它元素的使用。其它关系包括用例和流的合并。静态视图显示为类图。静态视图可以用于产生大多数程序中的数据结构声明。UML 类图中有许多种元素，如接口、数据类型、用例和信号。它们合在一起被称为分类，它们的举止如同具有某种限制的类。

动态行为。有两种方式来建模行为。一种是通过与外界交互的对象的生命史；另一种是使用一系列对象的通信模式，这些相互连接的对象交互实现行为。状态机是被隔离的对象视图——视图中，对象依照当前状态对事件响应，执行动作，迁移至新状态。状态机在状态图中显示。

相互交互对象的系统视图是一种协作，即依赖上下文的对象和互相之间链的视图，连同对象间数据链上的消息流。该视点在单张视图中统一了数据结构、控制流和数据流。协作和交互在顺序图和协作图中表达。指导所有行为视图的是一系列用例，用例展示了活动者——系统的外部使用者所见的一部分系统功能。

实现构造。UML 模型对逻辑分析和物理实现均可以表达。一定的构造代表了实现单元。构件是与一系列接口一致和为其提供实现的物理、可替换的系统组成部分。它可以作为满足相同说明的其它构件替代品。结点是定义了位置的运行时段的运算资源。它可以容纳构件和对象。配置视图描述了运行系统中结点的配置，构件和对象在结点中的分布，及包括结点内容的可能迁移。

模型组织。计算机可以处理大型的模型，但人不可以。大型系统中，建模信息必须划分成条理分明的单元，以使开发团队可以并发的工作在不同的部分。即使在小型系统中，人类的理解能力需要模型内容被组织到适度大小的包中。包是 UML 模型中通用的层次组织结构。它们可用于储存、访问控制、配置管理和构造包含复用模型块的库。包上的依赖总结了包内容的依赖关系。包之间的依赖可以被整体系统的体系结构来强制。从而包内容必须同包依赖和系统体系结构强制相一致。

扩展机制。无论语言的设施多么完备，人们总是需要对其进行扩展。我们对 UML 提供了有限的扩展能力，无需对基本语言进行修改。我们相信它可以容纳日常的大多数扩展需要。UML 扩展机制包括版型、约束和标签值。版型是与现有元素结构相同的新元素，它具有附

加的约束、不同的解释和图标，并被代码生成和后端工具不同的对待。标签值是可以附加在任何模型元素，容纳任意信息的任意标签—值的文字串对，如项目管理信息、代码产生指导和版型所需的值。标签和值用文字串来表达。约束是使用某些约束语言——如编程语言、特殊的约束语言和自然语言，用文字串表达的条件。UML 包括称为 OCL 的约束语言。同任何扩展机制一样，它们必须小心的使用，因为它对于其它人可能是本地化、难以理解的。然而，它们可以避免某些根本的更改。

UML 一览（*UML Walkthrough*）

本章用一个简单的例子来浏览 UML 概念和图。本章的目的是用较少的图形化表达概念的视图和图，来组织高层次的 UML 概念。它展示了各种概念是如何描述系统以及视图是如何配合在一起的。本章总结不是全面的；一些概念被省略。更详细的内容，可以参见勾画 UML 语义视图的后续章节，以及 UML 规范。

本文中的例子是一个计算机化操作的剧院票房。它是一个设计的例子，目的是在较短的篇幅中突出各种 UML 结构。它被特意的简化，没有详细的表述。已实现系统的完整模型既不适合简短的篇幅，也不能显著表达足够范围的 UML 结构。

UML 视图

UML 的各种概念和结构并不存在明显的界线，但为了方便，我们将它们划分至多个视图。视图是表达系统单个方面的 UML 建模结构的简单子集。不同视图的分隔有时有些模糊，但我们希望它是直观的。一种或两种图为各种视图中的概念提供了可视化标记。

视图在最高层次可以划分为三个领域：结构性分类、动态行为和模型管理。

结构性分类描述了系统中的事物和事物间的关系。分类包括类、用例、构件和结点。分类提供了动态行为构建的基础。分类视图包括静态视图、用例视图和实现视图。

动态行为描述了系统时间上的行为。行为可以用静态视图中系统快照的一系列变更来描述。行为视图包括状态机图、活动图和交互图。

模型管理描述了用层次式的单元对模型自身的组织。包是模型的通用组织单元。特殊的包包括模型和子系统。模型管理视图与其它视图相交迭，为团队工作和配置控制把它们组织起来。

UML 还包括欲提供有限但实用扩展能力的若干结构。这些结构包括约束、版型和标签值。它们适用于所有视图的元素。

表 3-1 显示了 UML 视图和显示它们的图，以及与各视图有关的主要概念。视图混合使用时，该表不应作为硬性的规则，而仅仅是日常使用的指南。

表 3-1：UML 视图和图

主要领域	视图	图	主要概念
结构性	静态视图	类图	类、关联、概括、依赖、实现、接口
	用例视图	用例图	用例、活动者、关联、扩展、包含、用例概括
	实现视图	构件图	构件、接口、依赖、实现
	配置视图	配置图	结点、构件、依赖、位置
动态	状态机视图	状态图	状态、事件、迁移、动作
	活动视图	活动图	状态、活动、结束迁移、分叉、连接
	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
扩展	所有	所有	约束、版型、标签值

静态视图

静态视图对应用领域的概念建模，以及将内建的概念作为应用实现的一部分。该视图不描述时间相关的行为，因而是静态的。时间相关的行为由其它视图描述。静态视图的主要组成部分是类和关系：关联、继承和各种依赖，如实现和使用。类是对应用领域或应用方案概念的描述。类视图围绕着类组织；其它元素属于或附加于类。静态视图显示为类图，名称的由来是因为它们主要的重点是类的描述。

类绘制为长方形，属性和操作类表放置在不同的分隔中。当不需要完整的细节时，分隔可以被隐藏。类可以在多个图形中显示。它的属性和操作经常在其它的图中被隐藏。

类间的关系绘成连接类的路径。不同种类的关系由线上的结构和路径或端点上的修饰来区分。

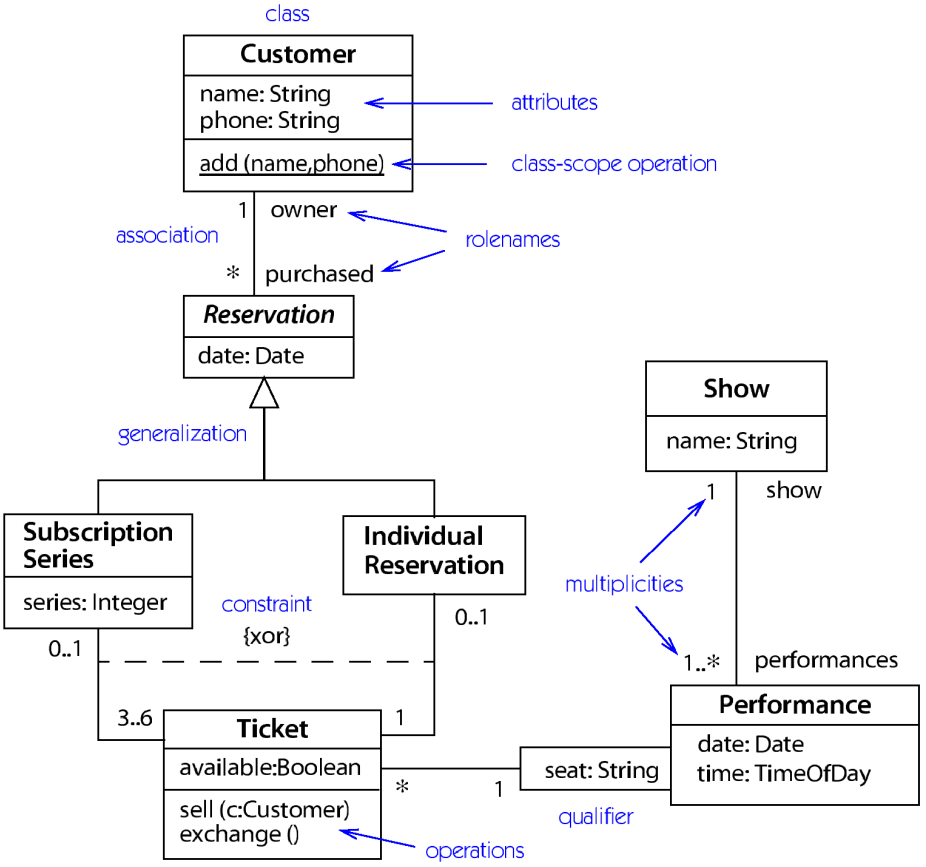


Figure 3-1. *Class diagram*

图 3-1 显示了票房应用的类图。该图包含了部分售票领域的模型。它显示了许多重要的类，如**顾客（customer）**、**预定（reservation）**、**票（ticket）**和**演出（performance）**。顾客可以有多次预定；而每个预定只能归属一个顾客。预定有两种形式：订购连票和单张订购。它们订票的数量不同。每张票是订购连票或单张订购的一个部分，但不能同时属于两中情况。每场演出有多张票，每张票有唯一的座位号码。演出可以通过节目、日期和时间来区别。

类可以用不同的精度和粒度来描述。在早期设计阶段，模型捕获问题逻辑方面的内容；在后期设计阶段中，模型还捕获设计决策和实现细节。大多数视图具有演化的性质。

用例视图

用例视图对外部用户——称为活动者——所感知的系统功能进行建模。用例是用活动者和系统之间的交互来表达、条理分明的功能单元。用例视图的目的是列举活动者和用例，显示活动者在每个用例中的参与情况。

图 3-2 显示了票房例子中的用例图。活动者包括职员、主管人和售票亭。售票亭是接收顾客订单的另一个系统。顾客在该例中不是活动者，因为它们没有直接与系统关联。用例包括了通过售票亭或职员买票，订单购买（仅通过职员），和监督整体售票情况（为主管人的请求）。购票和订单购买包括了通用的部分——即通过信用卡服务收费（完整的订票系统描述包括了许多其它用例，如换票和检查有效性等）。

用例可以在各种详细程度上描述。它们可以用其它较简单形式的用例来代理和描述。用例作为交互视图中的协作来实现。

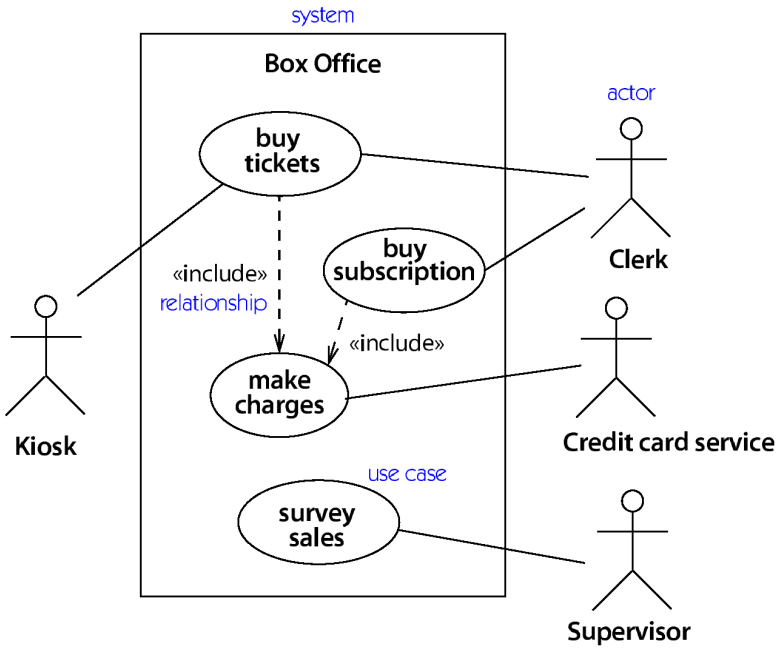


Figure 3-2. *Use case diagram*

交互视图

交互视图描述了实现系统行为角色之间的消息交换序列。分类角色是对交互中充当特殊角色的对象的描述，从而使该对象区别于相同类的对象。视图提供了系统中行为全局的描述——它显示了多个对象间的控制流程。交互视图用侧重点不同的两种图来显示：顺序图和协作图。

顺序图

顺序图表示了随时间安排的一系列消息。每个分类角色显示为一条生命线——代表整个交互期间上的角色。消息则显示为生命线之间的箭头。顺序图可以表达场景——即一项事务的特定历史。

顺序图的一个用途是显示用例的行为序列。当行为被实现时，每个顺序图中的消息同类的操作或状态机中迁移上的事件触发相一致。

图 3-3 显示了**买票**用例的顺序图。用例由售票处的顾客与票房的交互产生。**收费**用例的步骤被包括在序列中，它包括与售票处和信用卡服务的通讯。该顺序图处于早期开发阶段，没有显示用户界面的细节。例如，座位列表的准确形式和指定座位的机制仍需要决定，但用例指明了交互的必要通讯。

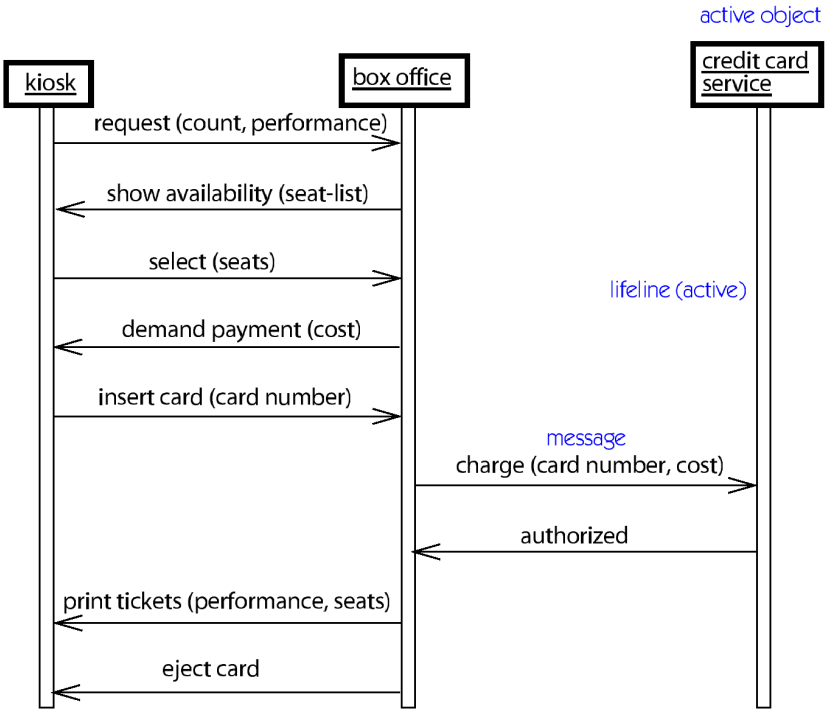


Figure 3-3. *Sequence diagram*

协作图

协作对交互中存在意义的对象和链建模。对象和链仅在提供的上下文中存在意义。分类角色描述了对象，关联角色描述了协作中的链。协作图通过图形的几何排布显示交互中的角色（图 3-4）。消息显示为附属在连接分类角色的关系直线上的箭头。消息的顺序由消息描述前的顺序号来表示。

协作图的一个用途是表现操作的实现。协作显示了操作的参数和局部变量，以及更永久性的关联。当行为被实现时，消息的顺序与程序的嵌套调用结构和信号传递一致。

图 3-4 显示了开发的后期阶段预订票交互的协作图。协作显示了对于订票，应用程序内部对象的交互。来自售票亭的请求被用于在数据库中从所有的演出中选择特定的一场。返回给**售票员**（**ticketSeller**）的指针 **db** 表达了至演出节目数据库的暂时性链，该链在交互中维持以及完成后被丢弃。售票员需要节目的若干座位；不同价格范围座位选择项被获得，并暂时被锁定，接着返回给售票亭以供顾客选择。当顾客在座位列表中作出了选择，所选的座位被索取，剩余的座位被解锁。

顺序图和协作图均显示了交互，但它们强调了不同的方面。顺序图显示了时间顺序，但角色间的关系是隐式的。协作图表现了角色之间的关系，并将消息关联至关系，但时间顺序由于用顺序号表达，并不十分明显。每一种图应根据主要的关注焦点而使用。

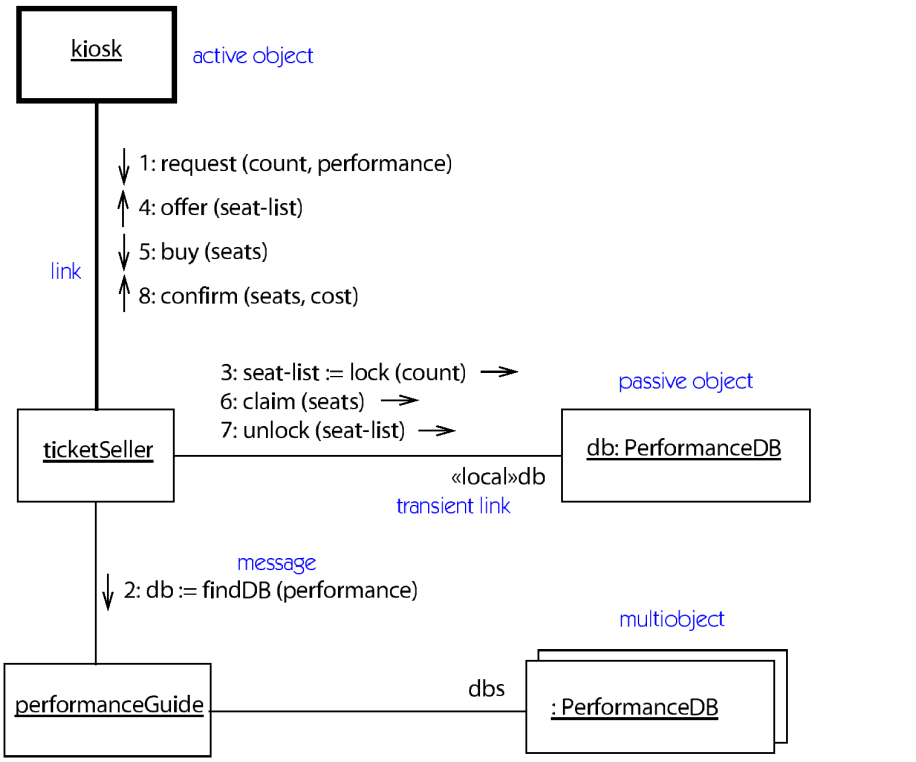


Figure 3-4. *Collaboration diagram*

状态机视图

状态机对类的对象的可能生命历史建模。状态机包含由迁移连接的状态。每个状态对对象生命期中的一段时间建模，该时间内对象满足一定的条件。当事件发生时，它可能导致迁移的激发，使对象改变至新状态。当迁移激发时，附属于迁移的动作可能被执行。状态机显示为状态图。

图 3-5 显示了某场演出戏票历史的状态图。票的初始状态（示为黑色圆点）是**有效**（**Available**）的状态。在季度开始之前，季票订购者的座位被分配。当顾客选票时，交互式购买的单张票被锁定。它或者被售出或在没有被选择的情况下被解锁。如果顾客选票花费太长时间，交易超时，座位被释放。季票订购者的场次可能会与其它演出相交换，则该场次的座位会再度有效。

状态机可以用于描述用户界面、设备控制和其它交互式子系统。它们还可用于在生命期中经历了若干特定阶段，每个阶段拥有特殊的行为的对象。

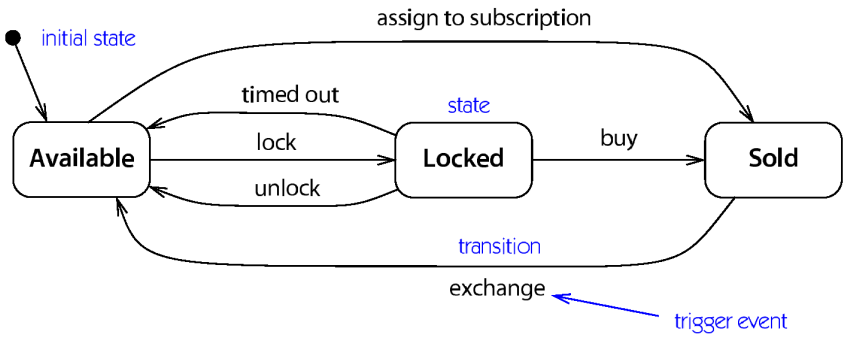


Figure 3-5. *Statechart diagram*

活动视图

活动视图是用于显示执行某个计算过程中的运算活动的状态机的一种变形。活动状态表现了一项活动：工作流的步骤或操作的执行。活动图描述了顺序和并发活动分组。活动视图表达为活动图。

图 3-6 显示了票房例子中的活动图。该图展示了放映一场演出所包含的活动。箭头表示时间上的依赖——例如，指定演出时间表前，必须选定节目。横条表示了控制的分叉和连接。例如，在演出安排完成后，剧院可以并发的进行宣传、剧本购买、艺术家雇佣、舞台搭建、灯光设计和服装定制工作。在排练开始之前，剧本和艺术家必须到位。

该例子显示了现实世界中人员机构工作流的建模。商业建模是活动视图的主要目标，但它还可用于对软件活动的建模。活动图能帮助理解系统的高层次的执行行为，无需顾虑活动图中的消息传递细节。

动作的输入和输出参数可以显示成连接动作的流关系和对象流状态。

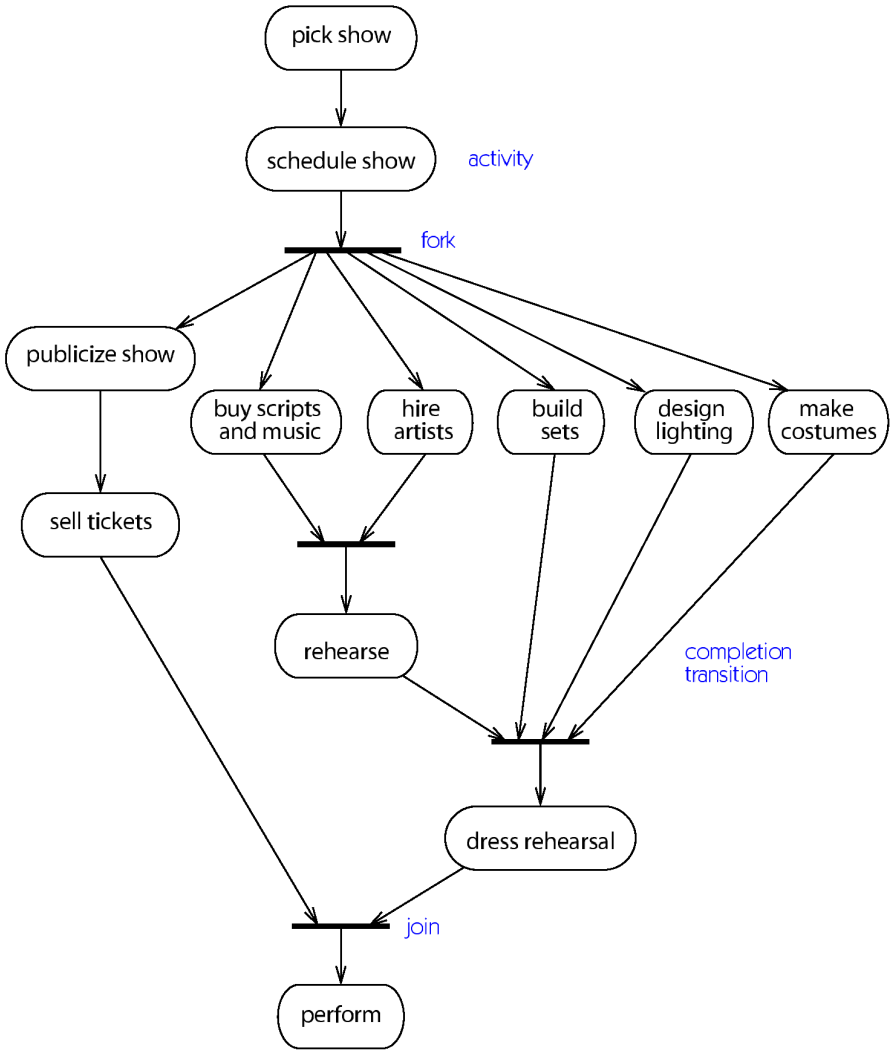


Figure 3-6. *Activity diagram*

物理视图

前面的视图从逻辑角度对应用中的概念建模。物理视图对应用本身的实现结构建模，如将其组织为构件和在运行结点上进行配置。这些视图提供了将类映射至构件和结点的机会。有两种物理视图：实现视图和配置视图。

实现视图对模型中的构件建模，即应用程序搭建的软件单元；以及构件之间的依赖，从而可以估计所预计到的更改的影响。它还对类及其它元素至构件的分配建模。

实现视图显示成构件图。图 3-7 显示了票房例子中的构件图。其中包括：三个用户界面，分别用于——与售票亭交互的顾客、使用在线预订系统的职员、查询销售情况的主管；依次处理来自于售票亭和职员需求的售票构件；处理信用卡收费的构件；包含票信息的数据库。构件图显示了系统中构件的类型；特殊配置的系统可能具有多个构件拷贝。

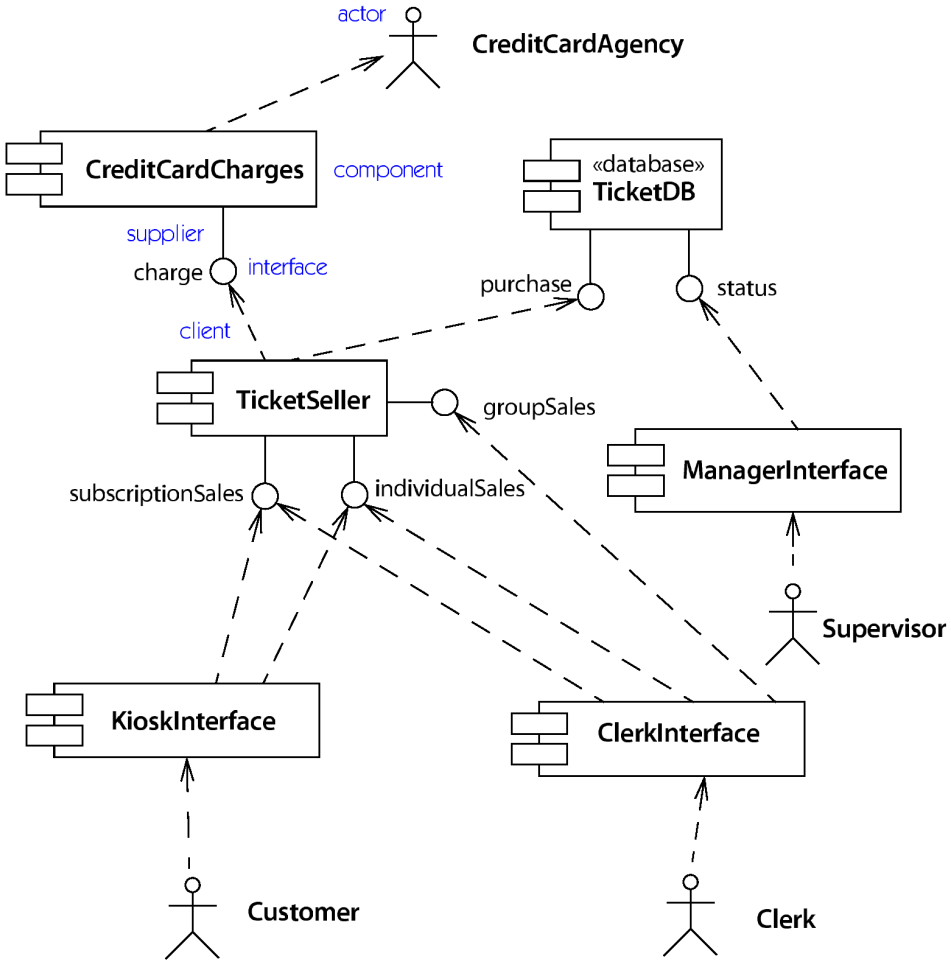


Figure 3-7. *Component diagram*

接口显示为具有名称的圆，即相关的服务集。连接构件和接口的实线表示构件提供接口所列举的服务。从构件至接口的虚线表明构件需要接口所提供的服务。例如，售票构件提供预订售票和集体售票；售票亭和职员均可访问预订售票接口，而集体售票接口只能供职员使用。

配置视图表达了运行时段构件实例在结点实例中的分布。结点是运行资源，如计算机、设备或内存。该视图允许分布式的结果和资源分配被评估。

配置视图显示为配置图。图 3-8 显示了票房例子中的描述级别的配置图。该图展示了系统中结点的种类和结点所拥有构件的种类。结点显示为方块。

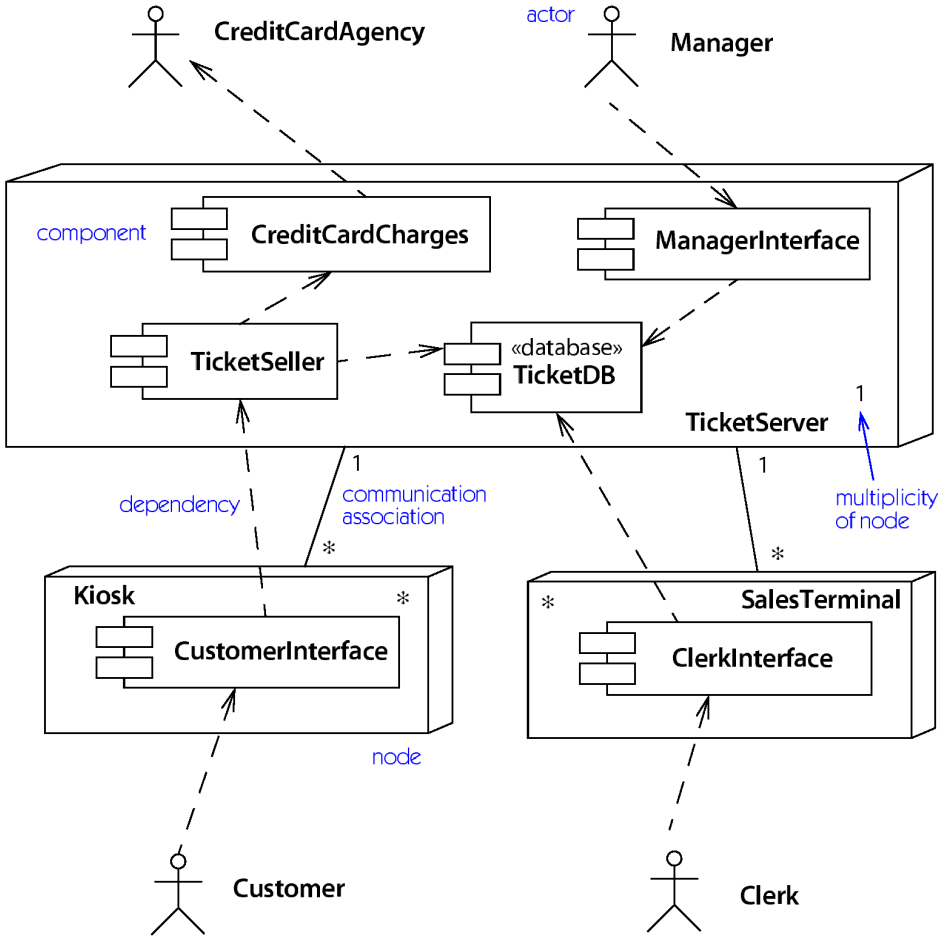


Figure 3-8. *Deployment diagram (descriptor level)*

图 3-9 展现了上例中实例级别的配置图。该图表现了特定版本系统的单个结点和它们之间的链。该模型中的信息一致于图 3-8 中描述级别的信息。

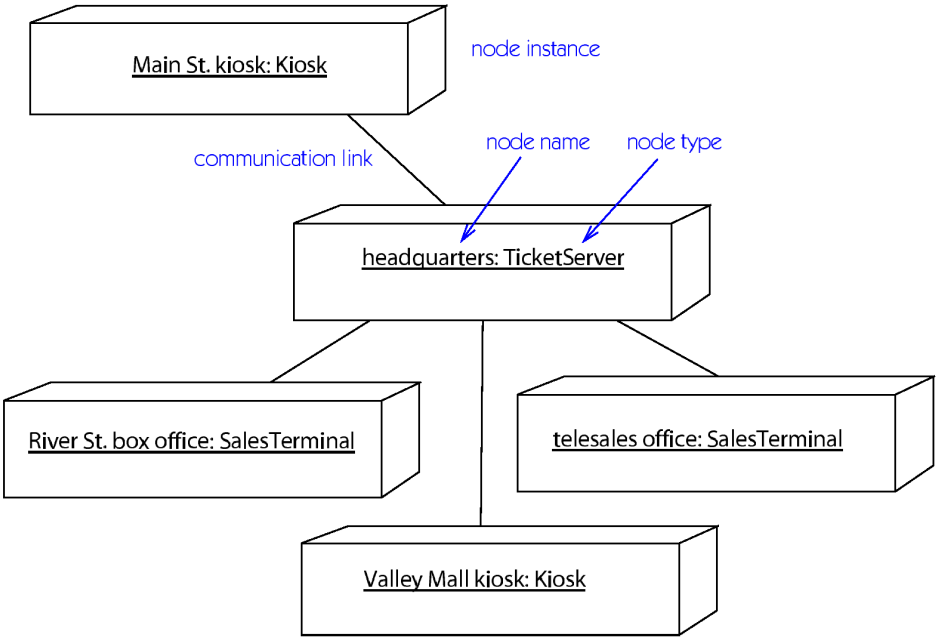


Figure 3-9. *Deployment diagram (instance level)*

模型管理视图

模型管理视图对模型本身的组织建模。模型由一系列包含模型元素（如类、状态机、用例）的包构成。包可以包含其它包；因此，模型指派了一个根包，间接包含了模型的所有内容。包是操纵包内容，以及访问控制和配置控制的单元。每个模型元素被包或其它元素所拥有。

模型是某个视角、给定精度的对系统的完整描述。因此，可能存在不同视角下系统的多个模型——例如，分析模型和设计模型。模型显示为特殊的包。

子系统是另外一种特殊的包。它代表了系统的一部分，并具有明晰的接口，接口可以实现为独特的构件。

模型管理信息经常在类图中出现。

图 3-10 展示了将整个剧院系统分解为包和它们之间的依赖关系。票房子系统包括了本章中的例子；整个系统还包含剧院运营和计划子系统。每个子系统包含若干个包。

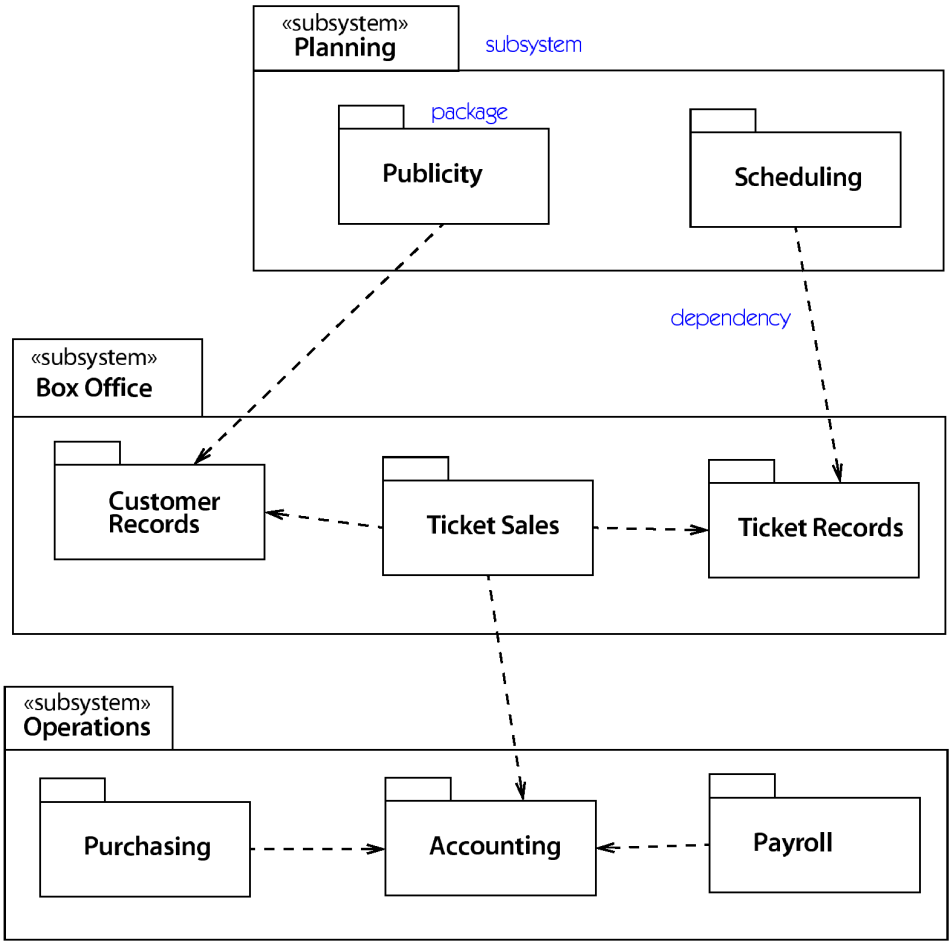


Figure 3-10. *Packages*

扩展结构

UML 包含了三种扩展结构：约束、版型、标签值。约束是用某种正式语言或自然语言表达的语义关系的文字陈述。版型是基于已有的模型元素，由建模人员修订的新模型元素。标签值是一条可以附加给任何模型元素的命名信息。

这些结构在不更改基本 UML 元模型的前提下，对 UML 进行各种扩展。它们可以用于特定领域 UML 的剪裁。

图 3-11 显示了约束、版型和标签值的例子。类**演出（Show）**上的约束确保了它的唯一性。图 3-1 展示了关联之间的 **xor** 约束；对象仅能拥有其中之一。约束适用于用文字表达 UML 结构不直接支持的陈述。

构件 **TickDB** 上的版型指明了该构件是数据库，它允许省略构件所支持的接口，因为它们是被所有数据库支持的接口。建模人员可以增加新的版型来表达特殊的元素。一系列约束、标签值或代码特性可以附加至版型。建模人员可以为给定的版型名称定义图标，以作为辅助。当然，文字形式仍可使用。

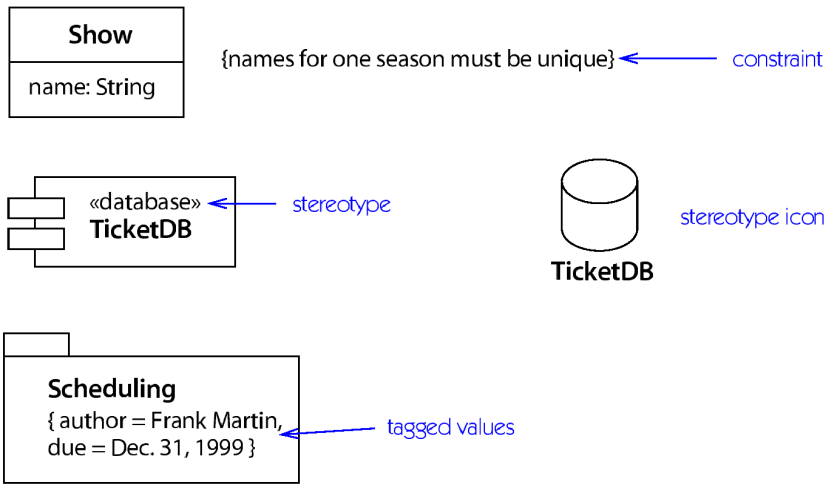


Figure 3-11. *Extensibility constructs*

包 **Scheduling** 上的标签值显示了在千年之前必须由 Frank Martin 完成。任意信息可以作为标签值附在建模人员所选的模型元素名称的下方。文字特别适用于项目管理信息和代码生成参数。大多数标签值应作为编辑工具的弹出信息，并通常不应在打印图形上显示。

视图间的联系

不同的视图在单个模型中并存，它们的元素之间存在许多关连，表 3-2 显示了其中的一部分。该表并不意图提供完整的描述，它显示了不同视图元素间主要的关系。

表 3-2：不同视图元素之间的部分关系

元素	*元素*	*关系*
类	状态机	拥有
操作	交互	实现
用例	协作	实现
用例	交互实例	场景示例
构件实例	结点实例	位置
动作	操作	调用
动作	信号	发送
活动	操作	调用
消息	动作	启用
包	类	拥有
角色	类	分类

静态视图（*Static View*）

概述

静态视图是 UML 的基础。模型静态视图的元素是应用中具有意义的概念，包括现实世界概念、抽象概念、实现概念、运算概念——系统中发现的所有概念。例如，戏院订票系统包括如下的概念：票、预定、订购计划、座位安排算法、订购的 WEB 交互以及档案数据等。

静态视图捕获对象结构。面向对象的系统将数据结构和行为特性统一成单个的对象结构。静态视图包括所有的传统数据结构内容，以及数据上操作的组织。数据和操作量化成类。从面向对象的角度来说，数据和行为紧密的联系在一起。如：**票**对象携带了数据，如它的价格、演出日期、座位号码，和数据上的操作，如预定和计算某个折扣下的价格。

静态视图将行为实体描述为离散的模型元素，但它不具有动态行为的细节。它将实体认为是被命名的，为类所拥有的或调用的事物。它们的动态执行被其它描述动态特性内部细节的视图所描述。这些视图包括交互视图和状态机视图。动态视图要求静态视图描述动态交互的事物——不可能在阐明交互的事物之前，描述事物如何的交互。静态视图是其它视图构建的基础。

静态视图中的关键元素是分类和它们之间的关系。分类是描述事物的模型元素。有许多种分类，包括类、接口和数据类型。行为性的事物由其它分类来细化，包括用例和信号。另一些分类，如子系统、构件和结点则体现了实现方面的内容。

为了便于理解和重用性，大型模型必须划分成较小的单元。包是控制和管理模型内容通用的组织单元。每个元素均属于某个包。模型是描述系统整体视图的包，它或多或少可以独立于其它模型使用；它是包含了更加详细描述系统包的根。

对象是建模人员理解和构建系统的分离的单元。它是类的实例——即结构和行为由类来描述的具有标识的个体。对象是具有可调用的良好定义行为的可标识的一个状态。

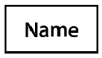
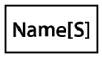
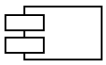
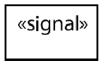
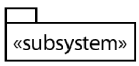
分类之间的关系是关联、概括，以及各种依赖，包括实现和使用。

分类

分类是模型中的离散概念，它具有标识、状态、行为和关系。分类的种类包括类、接口和数据类型。其它类型为行为概念、环境事物或实现结构的具体化。这些分类包括用例、活动者、构件、结点和子系统。表 4-1 列举了各种分类和它们的功能。元模型术语分类包括了所有的概念，但正如类是我们最熟悉的概念一样，首先讨论类，再定义其它的概念。

类。类表征了正在建模应用中离散的概念——物理事物（如飞机）、商业事物（如订单）、逻辑事物（如广播日程表）、运算事物（如哈希表）、或行为事物（如某个任务）。类是具有相似结构、行为、关系的一系列对象的描述。所有属性和操作附加于类或其它分类。面向对象系统围绕着类进行组织。

表 4-1：分类的种类

分类	功能	标记
活动者	系统的外部用户	
类	建模系统中的概念	
状态类	受限正处于某给定状态的类	
分类角色	协作中受限于特定使用的分类	
构件	系统的物理块	
数据类型	对不具有标识一系列基本数值的描述	Name
接口	描述行为特性的一系列命名操作	
结点	运算资源	
信号	对象间的异步通信	
子系统	作为单元的包，具有规格说明、实现和标识	
用例	与外界交互实体行为的规格说明	

对象是具有标识、状态和可调用行为的分离实体。对象构造了系统；类是理解和描述众多对象个体的概念。

类定义了一系列具有状态和行为的对象。状态由属性和关联来描述。属性通常用于无标识的单纯数值，如数字或字符串，关联用于用于具有标识的对象间的连接。可调用的行为表达为操作，方法是操作的实现。对象的生命期由类所附带的状态机来表述。类使用具有类名称、属性、操作分隔的长方形来表示。如图 4-1。

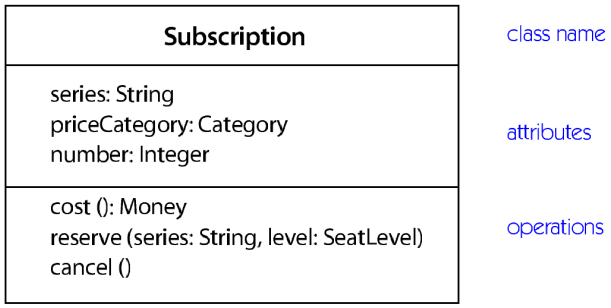


Figure 4-1. *Class notation*

一系列的类可以使用概括关系，和构建其上的继承机制来共享状态和行为描述。概括将更特定的类（子类）和包含若干类共有属性的更一般化的类（超类）联系起来。类可能有 0 个或多个的双亲（超类）和 0 个或多个的孩子（子类）。类从它的双亲或其它祖先继承状态和行为描述，以及定义被孩子和其它后代继承的状态和行为。

类在它的容器中具有唯一的名称，容器通常是包，但有时是其它的类。类具有相对于容器的可见性；可见性指明了类被容器外的其它类如何使用。类具有重数，说明了该类存在多少个实例。通常重数为多（0 或多，无明显的限制），但单子类在执行过程中仅具有一个实例。

接口。接口是未给出实现或状态的对象行为的描述；接口包含操作，但没有属性，它不具有可视的外出关联。构件、一个或多个类可以实现接口，每个类实现接口中的操作。

数据类型。数据类型是对无标识的基本数据的描述（独立的存在和副作用的可能性）。数据类型包括数字、字符串和枚举值。数据类型采用值传递，且是不变的实体。数据类型不具有属性，但可能有操作。操作不会改变值，但可能将值作为结果返回。

意义的层次。类可以在模型的不同意义的层次上存在，包括在分析、设计和实现的层次。在表达现实世界的概念时，捕获现实世界状态、关系和行为是很重要的。而实现的概念，如信息隐藏、效率、可见性和方法，与现实世界的概念无关（它们属于设计的概念）。许多类的潜在特性在该层次是无关的。分析层次的类表达了应用领域或应用本身的逻辑概念。分析模型可能是对被建模系统的较小规模表达，足够捕获基本的系统逻辑，而无需涉及运行或构建方面的问题。

在表现高层次的设计时，与类相关的概念有：状态转化至特定的类，对象漫游的效率，外部行为和内部实现的划分，精确操作的说明等。设计层次的类表现了将状态信息和操作打包至离散单元的决策。它捕获了关键的设计决定，对象对信息和操作的封装。设计层次的类包含了现实世界和计算机系统的内容。

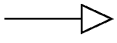
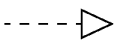
最后，在表达编程语言代码时，类的形式与所选择的编程语言紧密相联。如果没有直接的语言实现，一些类的能力可能被抛弃。实现级别的类可以直接映射成代码。

同个系统可以包含多个级别的类；面向实现的类可能实现模型中更逻辑化的类。实现类表现了特定语言中类的声明。它捕获了语言所需要类的精确形式。然而，许多情况下，分析、设计和实现信息可以嵌入在单个的类中。

关系

分类中的关系指关联、概括、流和各种依赖，包括实现和使用（见表 4-2）。

表 4-2：关系的种类

分类	功能	标记
关联	类实例之间连接的描述	
依赖	两模型元素之间的关系	
流	连续时间上同个对象的两个版本之间的关系	
概括	一般化描述和更具体化事物之间的关系，用于继承	
实现	说明和实现之间的关系	
使用	某个元素需要其它元素来完成功能	

关联关系描述了给定类的对象个体之间的语义连接。关联提供了不同交互类对象间的连接。剩余的关系则相关于分类本身的描述，而非它们的实例。

概括关系将双亲分类（超类）的一般化描述和更具体的孩子分类（子类）联系起来。概括通过增量的声明方便了对分类的表述，增量的声明添加至从祖先继承而来的描述。继承机制从使用概括关系的增量描述中构造分类的完整描述。概括和继承允许不同的分类共享通用的属性、操作和关系，而无需重复。

实现关系将说明与实现联系起来。接口是无实现的行为说明；类包括了实现结构。一个或多个类可以实现接口。每个类实现接口中的操作。

流关系将连续时间上同个对象的两个版本联系在一起。它表现了对象值、状态和位置的改变。流关系可以连接交互中的分类角色。流的变形包括 become（同个对象的两个版本）copy（从已有的对象创建新的对象）。

依赖关系联系了行为和实现上相互影响的类。除了实现，还有多种依赖，包括跟踪（不同模型中元素的松散连接），细化（不同理解层次的映射），使用（要求单个模型中其它元素的出现），绑定（模板参数的赋值）。使用依赖频繁的用于表征实现层的关系，如代码级别的关系。依赖在总结模型组织单元时特别有用，如使用包显示系统的结构。另外，编译的约束也可以用依赖来表达。

关联

关联描述了系统中对象和其它实例之间离散的连接。关联联系了两个或多个分类的有序表（元组），且允许重复。最普遍的关联是一对分类的二元关联。关联的实例是链。链包含了一个元组（有序表）的对象，每个来自于相应的类。一个二元链包含了一对对象。

关联携带了系统中对象间的信息。当系统运行时，对象间的链可以被创建和销毁。关联可以认为是连接系统的“粘合剂”。如果没有关联，所存在的不过是些无法工作的孤立对象。

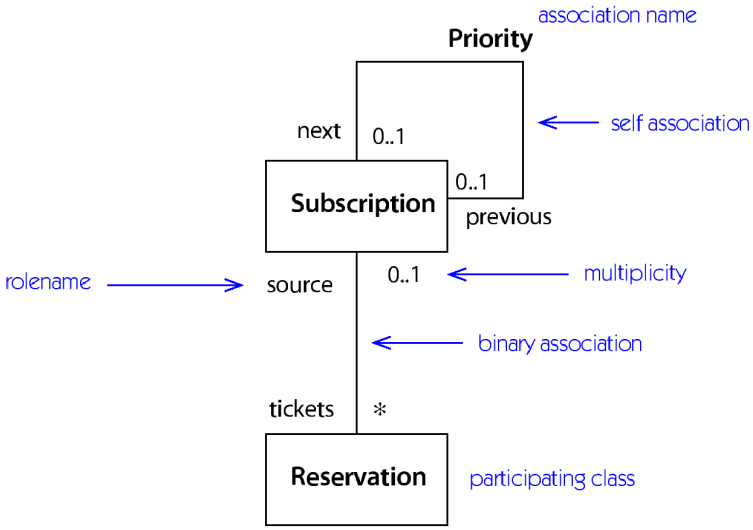


Figure 4-2. *Association notation*

如果在同个关联中相同的类出现不止一次，那么对象个体可以与自身关联。如果在关联中同个类出现了两次，则两个实例不必是，往往也不是同一个对象。

关联至对象的连接点被称为关联端点。大部分有关信息被附在关联的某个端点。关联端点可以拥有名称（角色名）和可见性。而它最重要的特性是重数——多少个类的实例可以关联于另一个类的实例。重数对于二元关联是非常有用的，而对于 n 元关联，它的定义非常复杂。

二元关联的标记是连接参与类的直线或路径。线段附近显示关联的名称，端点则附带角色名和重数，如图 4-2 所示。

关联还可以拥有自己的属性，这种情况下它既是关联，又是类——关联类（见图 4-3）。如果关联属性在一系列相关对象中是唯一的，则为限定（见图 4-4）。限定是从关联中若干相关对象中选取唯一对象的值。查询表和查询数组可以被建模成限定关联。限定对于名称和识别码的建模是比较有意义的。限定也可用于设计模型中对索引的建模。

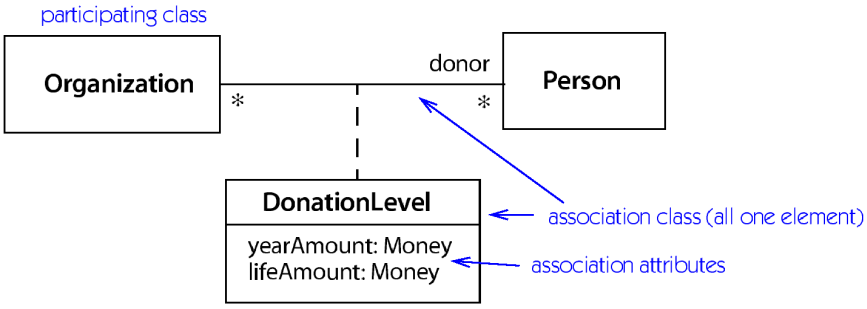


Figure 4-3. *Association class*

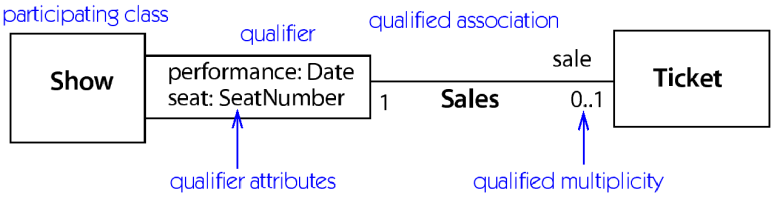


Figure 4-4. *Qualified association*

在分析阶段，关联表表现了对象间逻辑关系。在如何实现关联上，无需强加方向或特别的考虑。冗余的关联应该避免，因为它没有添加逻辑信息。在设计阶段，关联捕捉有关数据结构，以及类间责任划分等设计决策。此时，关联的方向性是重要的，而出于对象访问的效率和将信息封装至某个特定类等方面的考虑，可能添加冗余的关联。然而，在该建模的阶段，关联不等同于 C++的指针。设计阶段可漫游关联表示了类可用的状态信息，它可以采用多种方法映射至编程语言。它的实现可以是指针，嵌于类中的容器类，甚至完全独立的表对象。其它设计属性包括链的可见性和可更改性。图 4-5 显示了关联的设计属性。

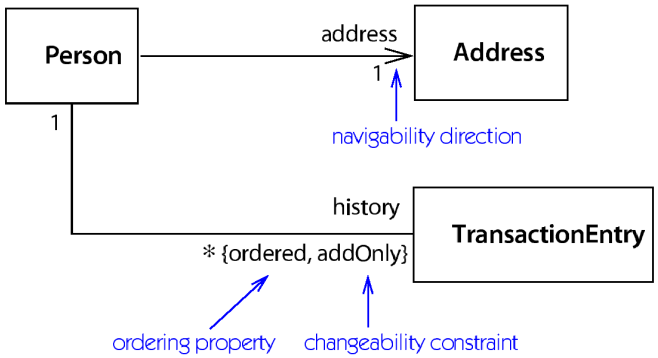


Figure 4-5. *Design properties of association*

聚集和组合。聚集是表达主体—部分关系的关联。它用在聚集端的菱形符号来表示。组合是关联的更强的形式。该关系中组合具有管理组成部分的特有责任，如它们的分配和释放。它用组合端的实心菱形来表达。在每个体现整体和部分的类之间具有各自的关联，但出于方便，整体端的路径被统一在一起，从而整个关联可以绘制成树结构。图 4-6 显示了聚集和组合。

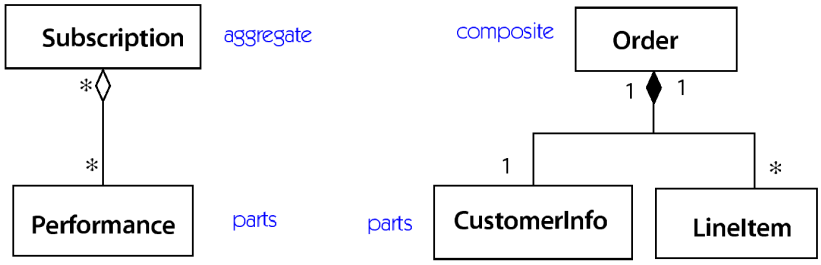


Figure 4-6. *Aggregation and composition*

链。链是关联的一个实例。链是对象引用的有序列表，每个对象引用必须是关联中相应类的一个实例或者类后代的一个实例。系统中的链是系统状态的组成部分。链不会独立于对象而存在；它们从相关联的对象获取标识（在数据库术语中，对象列表是链的主键）。概念上，链独立于相关联的类。而在实践中，关联常常用参与类中的指针来实现。它们也可用于独立于所连接类的容器来实现。

双向性。关联的不同端点是可以区分的，即使它们相关于相同的类。这意味着相同类的不同对象可以被关联。因为端点是可被区分的，关联是不对称的（特殊情况除外）。端点是不可交换的，这在平常的语言中是很容易理解的，如动词的主语和宾语是不可互换的。关联有时被称为双向的，意味着逻辑关系在两个方向均成立。该陈述经常被误解，甚至为许多方法学家所错误理解。它并不意味着类“知道”对方，或实现上可以相互访问。它仅仅指逻辑关系是可以反转的，而无关于该关系是否容易被实现。声明关联单方向的遍历能力，可以使用漫游性来标注关联。

为何基本模型是关系型的，而编程语言中的流行的指针模型不是？其原因是模型用来捕获实现背后的目的。如果两个类之间的关系被建模成一对指针，则它们是相互关联的。关联方法承认关系在两个方向均是有意义的，而无关于它们的实现。将关联映射成实现中的一对指针是非常容易的，但识别两个指针是相互的逆转却很困难，除非它们是关联模型的一个部分。

概括

概括是一般化何具体化描述之间的分类关系。具体化的描述构建在一般化描述之上，并对它进行扩展。后者与前者完全一致（具有所有属性、成员和关系），并可能包含新增的信息（如，抵押贷款是贷款中的更细化一种。抵押贷款包含了贷款的所有基本特性，并且增加了许多新的描述，如使用房屋来作为贷款的抵押）。更一般化的描述被称为双亲，多个层次中的该类元素称为祖先；更加具体化的描述被称为孩子，多个层次中的该类元素称为后代。在上例中，**贷款**是双亲类，**抵押贷款**是孩子类。概括用于分类（类、接口、数据类型、用例、活动者、信号等）、包、状态机和其它元素。对于类，术语超类和子类分别用于双亲和孩子。

概括绘制为从孩子指向双亲的空三角形箭头（图 4-7）。多个概括关系可以画成树状，即一个指向双亲的箭头和若干至孩子的分支。

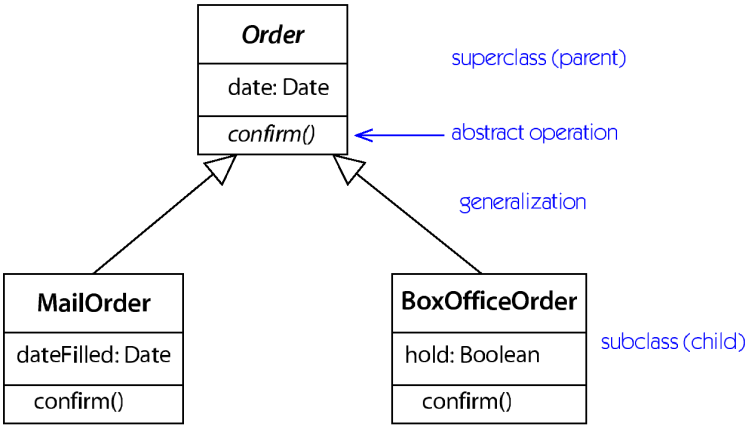


Figure 4-7. *Generalization notation*

概括的目的。概括具有两个目的。第一是定义当声明存放给定类值的一个变量时（如参数或过程变量），类实例（或其它元素）可以被使用的条件。这被称为替代原理（来自于 Barabara Liskov）。该规则指出后代的实例可以用于任何祖先被声明使用的地方。例如，如果一个变量被声明成存放贷款实例，则抵押贷款对象是合法的取值。

概括使多态操作成为可能——即操作的实现（方法）由实际对象的类来决定，而非调用者显式指定的类。之所以如此，是由于双亲类可能有多个孩子，对于定义于整个类层次上操作，每个孩子具有自己的实现。例如，对于抵押贷款和汽车贷款，利息的计算可能是不一致的，但它们均是双亲贷款类的操作——计算利息的一个版本。如果一个变量被声明成可以存放双亲的类对象，则任何子类的对象均可以被使用。每个均具有自己的特殊操作。这一点非常有意义，因为新的类可以在日后加入，而无需修改已有的多态调用。例如，新的贷款类型被加入后，原有的使用 compute interest 操作的代码无需修改。双亲类中的多态操作可无实现，而由孩子类提供具体实现。这种不完全的操称为抽象操作。

概括的另一个目的是共享对祖先的描述，允许对元素进行增量描述。这被称之为继承。继承是一种机制。通过该机制，类对象的描述由该类和它祖先中的声明所组成。继承允许描述的被共享部分只被声明一次，且为多个类共享；而不是在每个使用它的类中重复。这种共享减少了模型的尺寸。更重要的是，它减轻了模型更新时的修改量和减少了意外不一致性的机会。对于其它类型元素，如状态、信号和用例，继承以相同的方式工作。

继承

每种可概括的元素都具有一系列可继承的属性。对于任何模型元素，它们包括约束。对于分类，还包括特征（属性、操作和信号接收）和关联中的参与。孩子继承所有祖先的可继承特征。它的完整特征集是所继承特征集和直接定义特征集的总和。

对于分类，拥有相同签名的属性只能被声明一次（直接或继承）。否则，会存在冲突，模型塑造不良。换句话说，在祖先中声明的属性不能在子孙中重复声明。操作可以在若干类中声明，只要它们的形式一致（相同参数、约束和含义）。重复的声明是冗余的。方法可以被层次中的多个类声明。后代中的方法替代（覆盖）祖先中相同签名的方法。如果同个方法中的两个或两个以上的不同拷贝被继承（通过多重继承），则它们相互冲突，模型塑造不良（某些语言允许方法被显式的选择，但我们发现在孩子类中重新定义更为简单和安全）。

元素上的约束是元素本身和它所有祖先约束的总和；如果它们中间存在任何的不一致，则该模型塑造不良。

在具体类中，每个被继承和声明的操作必须直接的或通过祖先继承被定义。

多重继承

如果分类具有一个以上的双亲，则每个双亲均被继承（图 4-8）。它的特征（属性、操作和信号）是所有双亲特征的联合。如果相同的类在一个以上的路径中作为祖先出现，则它仅为每个成员提供一个拷贝。如果具有相同签名的特征被两个类声明，而非继承同个祖先（独立声明），则该声明存在冲突，且模型塑造不良。对于该情况，UML 不提供解决冲突的规则，因为经验显示设计人员应显式的解决冲突。一些如 Eiffel 的语言，允许程序员显式的解决冲突。这相对于隐含的解决规则更简单和安全，也是常常引起设计者惊讶的地方。

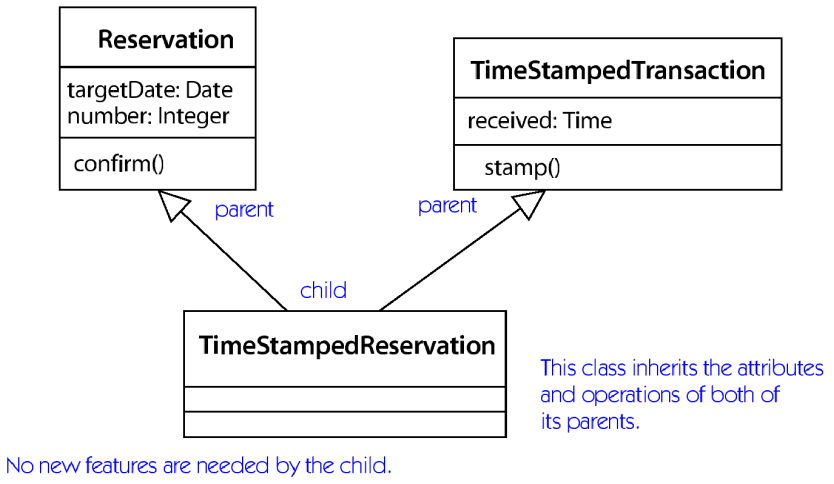


Figure 4-8. *Multiple inheritance*

分类和多重分类

在最简单的公式化中，一个对象只具有一个直接类。许多面向对象的语言具有上述限制。对于一个对象只能有一个类的限制，并没有逻辑的必然性——典型的，我们从不同的角度观察现实世界。在 UML 的更通用公式化中，对象可以具有一个或多个直接类。对象的行为如同是属于多个直接类的隐含子类——实际上，类似于无需声明新类的多重继承。

静态和动态分类

在最简单的公式化中，对象在创建后可能无法改变自己的类。同样，对此限制并不存在逻辑的必然性。它主要是为了使面向对象的语言实现更为简单。在更通用的形式中，对象可以动态的改变直接类。动态改变时，它可能丢失或得到属性和关联。在丢失的情况下，属性和关联中的信息被遗失，即使它改回为原先的类也不能再恢复。如果得到属性或关联，它们在改变时必须被初始化，类似于新对象的初始化。

当多重分类和动态分类结合在一起时，对象可以在生命期中得到或失去类。动态类有时称为角色和类型。一个普遍的建模模式要求每个对象具有一个静态的内在类（在对象的生

命期中不可以更改的类），加上 0 个或多个在生命期中可以增删的角色类。内在类描述了它的基本特征，角色类描述了暂时性特征。尽管许多编程语言不支持类生命层次中的多重动态分类，但它仍然是可以映射成关联的有价值的建模概念。

实现

实现关系将一个模型元素，连接至另一个提供了行为说明而无结构或实现的模型元素，如接口。用户必须至少支持（通过继承或直接声明）后者所提供的所有操作。尽管实现用于如接口等说明性元素，但它还可用于具体实现元素，以指出它的说明（而非实现）必须被支持。例如，它可以现实类的经优化版本同较简单但不充分版本之间的关系。

概括和实现均将一般化描述和更详细的版本关联在一起。概括关联了相同语义级别的两元素（例如，同级别的抽象），且通常在同一个模型中；实现连接了不同语义级别的元素（如分析类和设计类，或接口与类），并常常在不同的模型中。不同开发阶段可能存在两个或多个完整的类层次，它们的元素通过实现来关联。两个层次不需要有相同的形式，因为实现类可能有与说明类不相关的实现依赖。

实现显示成闭合的空心虚线箭头（如图 4-9）。它与虚线的概括符号相似，意味着与继承相类似。

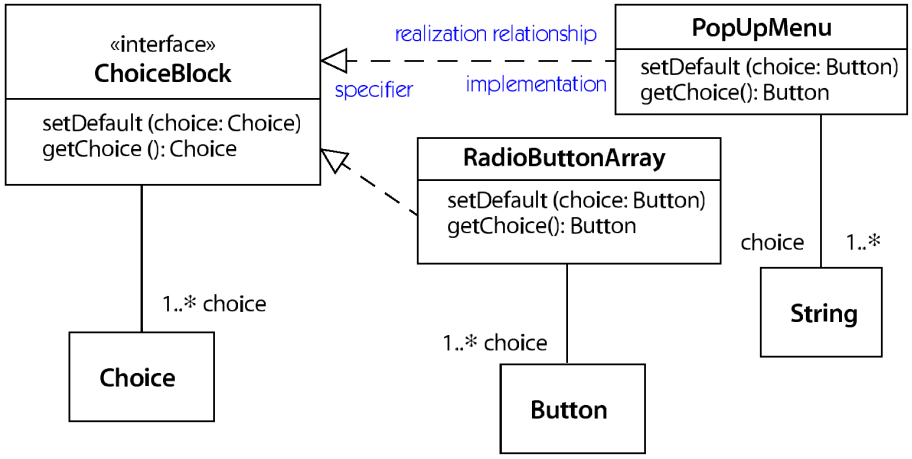


Figure 4-9. *Realization relationship*

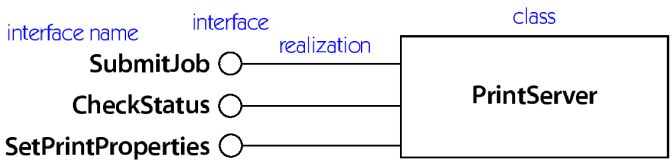


Figure 4-10. *Interface and realization icons*

对于显示接口（无内容）合实现它们的类或构件专门有一种缩略的标记方法。接口显示成通过实线与表示分类的方形相连接的圆。

依赖

依赖指明了两个或两个以上模型元素之间语义的关系。它直接关联了模型元素本身而无需一系列实例来表达意义。它表明了供应商元素的修改需要或指明客户元素的更改。

根据该定义，关联和概括关系也属于依赖，但它们具有重要意义的特殊语义。因此，它们具有自己的名称和详细语义。我们通常将依赖用于不适合更细致范畴的所有其它关系。表 4-3 列举了 UML 基本模型中依赖的类型。

跟踪是不同模型中元素间的概念，常常是不同开发阶段的模型。它缺乏详细的语义，典型用于跟踪模型间的系统需求和了解自身模型的更改对其它模型的可能影响。

细化是不同开发阶段或者不同抽象层次概念的不同版本之间的关系。这两种概念在最后的模型中不会共存。其中之一往往是另一个的未完成版本。理论上，存在未完成版本至完成版本的映射，但这并不意味着变换是自动的。通常，更详细的概念包含设计人员多种途径的设计决策。理论上，模型的更改能被另一个模型所校验；事实上，现有的工具无法完成该功能，尽管一些简单的映射可以被强制实施。因此，细化对于建模人员是一个提醒，即多个模型以可以预计的方式相互关联。

表 4-3：依赖的种类

依赖	*功能*	*关键字*
实现（realization）	说明和实现之间的映射	**realize**
跟踪（trace）	存在于不同模型元素间的声明，但不如映射精确	**trace**
细化（refinement）	不同语义层次映射的声明	**refine**
派生（derivation）	一个实例可以由其它实例运算得到的声明	**derive**
使用（usage）	一个元素为正确行使责任（包括调用、实例化、参数、发送）而要求其它元素存在的声明	**usage**
调用（call）	一个类中方法调用另一个类操作的声明	**call**
实例化（instantiation）	类方法创建其它类实例的声明	**instantiate**
访问（access）	允许一个包访问另一个包的内容	**access**
引入（import）	允许一个包访问另一个包的内容，并将被引入者的别名加至引入者的名字空间	**import**
友元（friend）	允许一个元素访问另一个元素的内容，而无可见性的限制	**friend**
绑定（binding）	模板参数的赋值，产生新的模型元素	**bind**
参数（parameter）	操作与参数之间的关系	**parameter**
发送（send）	信号发送者与接收者之间的关系	**send**

派生依赖指一个元素可以由其它元素运算而来（但派生元素可以被显式的添加至系统，避免代价高昂的重复计算）。实现、跟踪、细化和派生属于抽象依赖——它们连接了同个事物的不同版本。

使用依赖是某个元素的行为或实现会影响其它元素或实现的声明。经常它来自于实现阶段的问题，如编译器在编译某个类时，要求其它类的定义。大多数使用依赖可以从代码中引出，而无需显式的声明，除非它们是限制了系统组织的由上至下设计风格的一部分（例如，使用预定义的构件或库）。可以指定使用依赖的特定种类，但它常常被忽略，因为关系的目的是强调依赖。精确的细节经常可以从实现代码中获得。使用的版型包括调用和实例化。调用依赖表明了一个类的方法调用其它类中的操作。实例化指出了一个类中的方法会创建其它类的实例。

若干使用依赖的变形说明了元素间的访问许可。访问依赖允许一个包可以看见另一个包的内容。引入依赖则更进一步，它将目的包内容加至引入包的名字空间。友元依赖是允许客户元素看见供应商元素私有成员的访问依赖。

绑定是对模板参数的赋值。它通过替代模板拷贝中的参数获得精确语义，是一种高度结构化关系。

使用和绑定依赖包含了相同的语义层次元素间的很强的语义。它们必须连接相同层次的模型（均为分析或设计，或者相同层次的抽象）。跟踪和细化依赖更模糊一些，可以连接不同模型或抽象层次的元素。

实例（instance of）关系（元关系，非严格的依赖）指明了一个元素（如对象）是另一个元素（如类）的实例。

依赖绘制成从客户元素至供应商元素的虚线箭头，使用版型来区分类型，如图 4-11 所示。

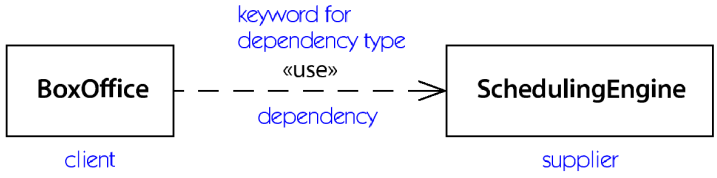


Figure 4-11. *Dependencies*

约束

UML 为图形建模提供了一系列概念和关系。然而，某些情况使用文字表达更加可行。约束是表现为给定语言中字符串的布尔表达式。自然语言、集合理论化语言、约束语言或各种编程语言均可以用于表达约束。UML 包括了约束语言的定义，称为 OCL。它用于表达 UML 的约束并具有广泛的支持。OCL 的更详细信息可参见 OCL 的词条和资料[Warmer-99]。

约束可以用于陈述各种非本地的关系，如关联路径上的限制。特别的，约束可以用于陈述特征的存在（*there exists an X such that condition C is true*）和普遍特征（*for all y in Y, condition D must be true*）。

许多标准约束被预定义为 UML 的标准元素，包括关联中的异或关系和概括中的子类关系上的各种限制。

详细信息参见标准元素。

约束显示为括号中的文字表达式。它可以是正式语言和自然语言。文字串可被放置在标注中或附加在依赖符号上。图-12 显示了一些约束。

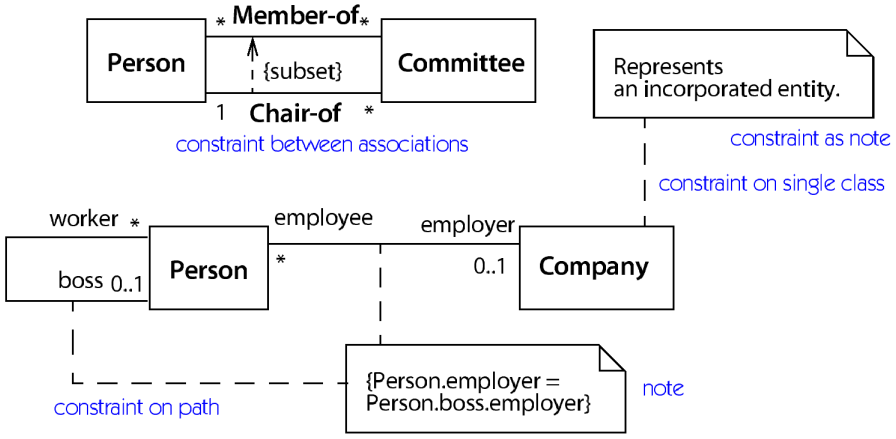


Figure 4-12. *Constraints*

实例

实例是具有标识的运行实体，它可以与其它运行实体区分。它在任何时刻具有值，值会随时间根据操作发生变化。

模型的目的是描述系统可能的状态和行为。模型是对潜在可能性、可能存在的对象集合、对象可能经历的行为历史的陈述。静态视图定义和限制了运行系统可预料值的配置。动态视图定义了运行系统从一个配置迁移至另一个配置的可能途径。基于模型，静态视图和动态视图共同定义了系统的结构和行为。

在某个瞬间系统的静态配置被称为快照。快照由对象和其它实例和链组成。对象是类的实例。每个对象是对它进行完整定义类的直接实例和该类祖先的间接实例（如果允许多重分类，则对象可以是多于一个类的实例）。类似的，每个链是关联的实例。每个值是数据类型的实例。

对于类中的每个属性，对象具有值。每个属性值必须与属性的数据类型相一致。如果属性是可选的或具有多个重数，则属性可以拥有 0 个或多个值）。链包括了一组值。每个值是给定类（或它的后代）对象的一个引用。对象和链必须遵从类或关联上的约束（包括显式的或内建约束，如重数）。

如果系统中每个实例是塑造良好的模型中某个元素的实例，且所有模型的约束被实例满足，则系统的状态是*有效系统实例*。

静态视图定义了单个快照中可以存在的一系列对象、值和链。理论上，任何于静态视图一致的对象和链的组合是模型的可能配置，但这不意味每个可能的快照可能或将会发生。一些快照在静态上是合法的，但在动态特性上可能是不可到达的。

UML 的行为部分描述了作为外部和内部行为结果而产生的合法快照序列。动态视图定义

了系统如何从一个快照移至另一个快照。

对象图

快照图是某个时间点系统的映象。由于它包含了对象的映象，它被称为对象图。作为系统的例子它很有用，例如，阐述复杂的数据结构或显示时间上的行为序列（图 4-13）。需要记住的是所有快照是系统的例子，而非系统的定义。系统结构和行为的定义在定义性的视图中，构建定义性的视图是建模和设计的目标。

静态视图描述了可能发生的实例。除了在例子中，真正的实例并不常常直接出现在模型中。

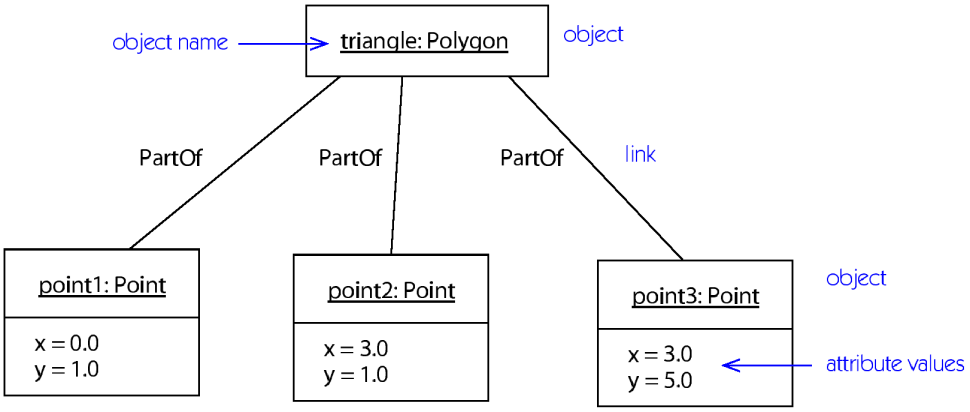


Figure 4-13. *Object diagram*

概述

用例视图从外部用户的角度捕获系统、子系统或类的行为。它将系统功能划分为对活动者（系统的理想用户）具有意义的事务。这些功能片被称为用例。用例通过系统与一个或多个活动者之间的一系列消息描述了与活动者的交互。术语活动者包括人员、其它的计算机系统和进程。图 5-1 显示了出售电话本的用例图（use case diagram），该模型作为例子已被简化。

活动者

活动者是与系统、子系统或类交互的外部人员、进程或事务的理想化。在运行时，具体人员会充当系统的多个活动者。不同的用户可能会成为同一个活动者，从而代表同一个活动者定义中的不同实例。

每个活动者参与一个或多个用例。活动者通过交换消息与用例交互（因而同拥有用例系统或类交互）。活动者的内部实现与用例无关；活动者可以被一系列定义状态的属性来充分的描绘。

活动者可以定义成概括层次结构，其中抽象活动者的描述被共享，被一个或多个特定的活动者描述扩充。

活动者用一个小人及名称来表示。

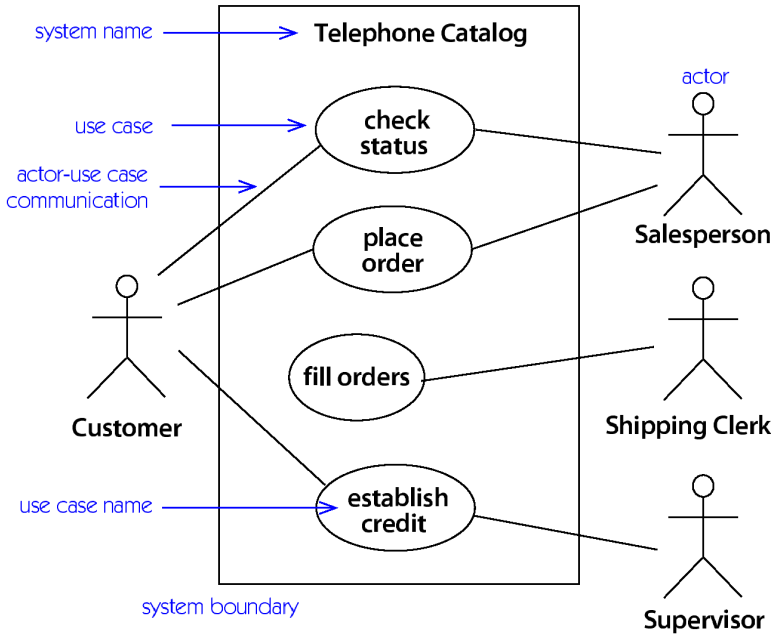


Figure 5-1. *Use case diagram*

用例

用例是系统单元提供的外部可感知的功能单元，表达成系统单元和与之相交互的一个或多个活动者的消息序列。用例的目的是定义清晰的行为块而不解释系统的内部结构。用例的定义包括了它所承担的所有行为——主线、常规行为的不同变形、行为的全体异常条件、以及所需的响应。从用户的角度，这些可能是非正常的情况；而从系统的角度，它们是必须描述和处理的额外变形。

在模型中，尽管用例的实现可能由于对象的共享造成潜在的依赖关系，每个用例的执行是独立于其它的用例。每个用例表示了正交的功能块，功能的执行可以与其它用例的执行混合在一起。

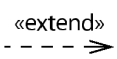
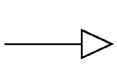
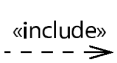
用户的动态部分可以用 UML 的交互来指定，显示为状态图、顺序图、协作图或非正式的文本描述。当用例被实现时，它们由系统中类的协作来实现。同个类可以参与多个协作，因而可以用于多个用例。

在系统级别，用例表达了外部用户所见的整个系统的外部行为。用例与系统操作相似，即由外部用户调用的操作。与操作不同的是，用例能在执行过程中连续的接受输入。用例还可用于系统的较小单元，如子系统和单个类。内部的用例表达了系统的某部分展示给系统其它部分的行为。例如，类的用例表达了由该类提供给在系统中充当角色的其它类的功能块。类可以拥有多个用例。

用例是一部分系统功能的逻辑描述，它不是系统实现的显式结构。相反的，每个用例必须映射至实现系统的类，用例的行为映射至类的迁移和操作。如类可以在系统的实现中充当不同的角色一样，它可以用来实现多个用例。实际上，部分设计工作是寻找能整洁的连接正确角色，以实现所有用例的类，而无需引入不必要的复杂性。用例的实现可以被建模成一个或多个协作，一个协作是一个用例的实现。

除了与活动者关联，用例可以参与多种关系，如表 5-1。

表 5-1：用例关系的种类

关系	*功能*	*标记*
关联	活动者和用例之间的通信路径	
扩展	额外行为的插入，基用例对插入不知情	
用例概括	一般用例与更具体化用例之间的关系，更加特定用例继承一般用例并添加特性	
包含	附加行为的至基用例的显式插入	

用例用椭圆来表示，用例名标在椭圆中或下方，它用实线与同自身通信的活动者相连接。

尽管每个用例实例是独立的。用例的描述可以分解成其它更简单的用例。这与一个类的描述可以通过对超类描述的增量定义来阐述是相类似的。用例可以简单的合并其它的用例，

将其作为自身行为的片段，称之为包含关系，该情况下，新用例不是原用例的特例，不能被原用例所替代。

用例还可以被定义为基用例的增量扩展，被称为扩展关系。可以对同个基用例进行多个扩展，它们可以一同被应用。基用例的扩展被增加至语义；被实例化的是基用例，而非扩展用例。

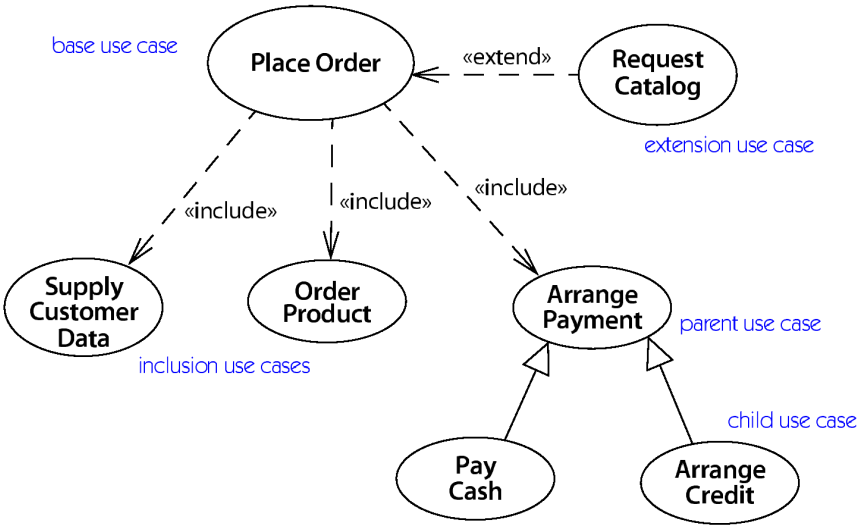


Figure 5-2. *Use case relationships*

包含和扩展关系以带关键字《**include**》和《**extend**》的虚线来绘制。包含关系指向被包含的用例；扩展关系指向被扩展的用例。

用例可以被特化成一个或多个子用例，即用例概括。任何子用例可以出现在父用例要求被使用的地方。

用例概括的绘制与其它任何概括相同，即用从子用例至父用例的带三角的线来表达。图 5-1 显示了目录出售中的用例关系。

状态机视图（*State Machine View*）

概述

状态机视图通过对每个类的对象的生命期进行建模，描述了对象时间上的动态行为。每个对象被认为是通过检测事件并对之响应来与外界进行通讯的孤立实体。事件表达了对象可以检测的变动——对象间的调用或显式信号、某个值的改变或时间的推移。任何影响对象的事物可以被描述成事件。真实世界发生的事情被建模成外部世界至系统的信号。

状态指就某个特定类而言，对于发生的事件具有相同性质响应的一系列对象值。换言之，同一状态的所有对象以相同的方式响应某个事件，即对于给定的所有对象在接收到同一事件时执行相同的动作。而不同状态的对象可能对相同事件具有不同的响应，执行不同的动作。例如，自动取款机在处理某项事物时和空闲时对取消键的响应是不一样的。

状态机不但可以描述类的行为，而且可以描述用例、协作和方法的动态行为。对于这些对象，状态代表了执行的一个步骤。我们使用类和对象来描述状态机，但它们可以直接的用于其它元素。

状态机

状态机是由状态和迁移组成的图。通常状态机附属于类，描述了类实例对接收事件的响应。状态机还可以附加于操作、用例、协作，以描述它们的执行。

状态机是某个类的对象所有可能生命历史的模型。对象被孤立的进行检查。所有外部世界的影响被总结为事件。当对象检测到事件，它以相关于当前状态的方式来响应。响应可能包括动作的执行和改变到新的状态。状态机可以被结构化成迁移的层次，并且可以对并发建模。

状态机是对象的局部化视图，该视图将对象与周围世界分开，独立的检查它的行为。它是系统的最小缩影，是一种很好的精确指明行为的方法，但常常不是一种好的理解系统整体操作的方法。描述系统行为结果的更好的方法是使用交互视图。状态机对理解控制机制较实用，如用户界面和设备控制器。

事件

事件是具有时间和空间位置的显著发生的某件事。它发生在时间点上；不具有持续时间。如果某件事存在结果，则将它建模成事件。当单独使用事件时，我们常常指的是事件描述符，即对所有具有相同一般形式事件个体的描述，正如类指的是具有相同结构的全体对象。一个特定事件的发生被称为事件实例。事件可能有参数来辨识单个事件实例，如同类具有属性来辨识每一个对象。同类一样，信号可以被分配至概括层次来共享相同的结构。事件可以划分为各种显式和隐式的种类：信号事件、调用事件、变更事件和时间事件。表 6-1 是事件类型和它们的描述。

表 6-1：事件的种类

事件类型	描述	语法
调用事件	对象间需要响应的，显式同步请求的接收	op（a:T）
变更事件	布尔表达式值的更改	**when**（exp）
信号事件	对象间显式、被命名的、异步通讯的接收	sname（a:T）
时间事件	绝对时间的到达或渡过了相对时间	**after**（time）

信号事件。信号是显式作为两对象间通讯媒介的被命名的实体；信号的接收是接收对象的一个事件。发送对象显式的创建和初始化信号实例，并发送给一个或若干个对象。信号包含了异步的单向通信——最基本的通信形式。发送者并不等待接收者对信号的处理，而是继续自己的工作。建模双向通信，使用多个信号，至少一个方向一个。发送者和接收者可以是相同对象。

信号可以在类图中使用关键字《**signal**》作为分类来声明；信号的参数声明为属性。作为分类，信号可以使用概括关系。信号可以是其它信号的孩子；它们继承父亲的参数，并且触发父参数上的迁移。

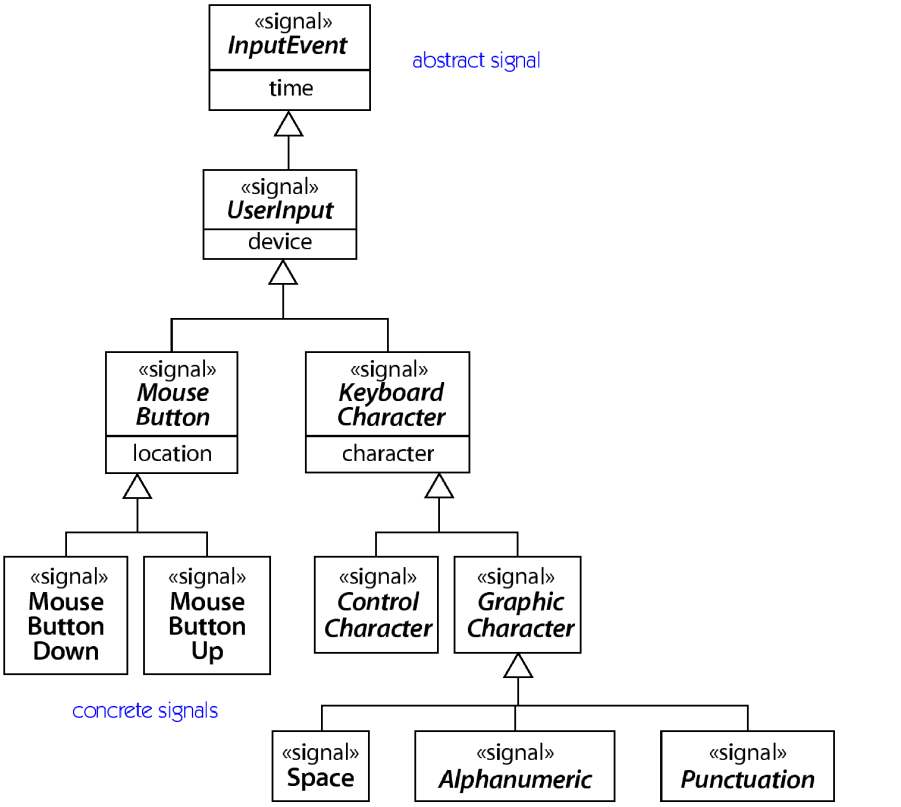


Figure 6-1. *Signal hierarchy*

调用事件。调用事件是对象对调用的接收，该对象选择操作而非固定过程来实现状态机迁移。对于调用者，一般的调用（实现为方法）无法与调用事件区分。接收者选择使用方法来或状态机中的事件触发来实现操作。操作的参数是事件的参数。一旦接收对象实现事件上迁移的触发对调用事件进行处理，或者迁移失败，控制返回给调用对象。与一般的调用不同，调用事件的接收者可能与调用者平行的继续执行。

变更事件。变更事件是依赖某些属性值布尔表达式的满足。这对“等待直到条件满足”是

一种易于表达的方式，但必须小心使用，因为它表达了持续的、潜在非本地的的计算（远程活动，因为值测试可能是远程的）。它既有优点，也有缺点。优点是它集中在真实依赖关系的模型——而非测试条件的机制。缺点是它模糊了值变更活动和最终结果之间的原因-结果关系。测试变更事件的开销可能是很大的，因为理论上它是持续活动。而在实践中，往往有许多方法来避免不必要的计算。变更事件应仅当更显式通讯形式不合适时使用。

注意变更事件和迁移条件之间的区别。迁移条件在迁移上的触发事件发生和接收者处理事件时被求值。如果为假，迁移不会被激发，该条件不会被重新求值。变更事件则持续的被求值直到为真，此时迁移被激发。

时间事件。时间事件代表了事件的流逝。时间事件可以采用绝对时间（某天某时刻）或相对时间（给定事件后的一段事件）。在高层次的模型中，时间事件可以被认为宇宙中的事件；在实现模型中，它们由某些特定对象的信号产生，如操作系统或应用中的对象。

状态

状态描述了对象生命期中的一段时间。它可以通过三个互补的方面来指定：某些性质上具有相似性的一系列对象值；对象等待某个或某些事件发生的一段时间；对象执行某些正在进行活动的一段时间。状态可以具有名称，尽管它常常是匿名的及用它的动作来描述。

状态机中，一系列状态由迁移来连接。尽管迁移连接两个状态（或更多，如果存在控制分叉或连接），迁移由离开的状态来处理。当对象处于一个状态，它对于离开状态迁移的触发事件敏感。

状态由带圆角的长方形来表示。（图 6-2）

Figure 6-2. *State*

迁移

离开状态的迁移定义了该状态对象对某事件发生的响应。通常，迁移具有事件触发、迁移条件、动作和目标状态。表 6-2 显示了迁移的种类和迁移调用的隐式动作。

表 6-2：迁移和隐式动作的种类

迁移类型	*描述*	*语法*
进入动作	当状态进入时执行的动作	entry/action
退出动作	当状态离开时执行的动作	exit/action
外部迁移	对产生状态改变或自迁移事件的响应，连同特定的动作。它常常还导致退出和/或进入动作。	e（a:T）[exp]/action
内部迁移	对产生动作执行事件的响应，但不产生状态的变化，或退出或进入动作的执行。	e（a:T）[exp]/action

外部迁移。外部迁移是改变活动状态的迁移，也是最普遍的一种迁移。由从源状态至目的

状态的箭头连同属性的文字表达来显示。（图 6-3）

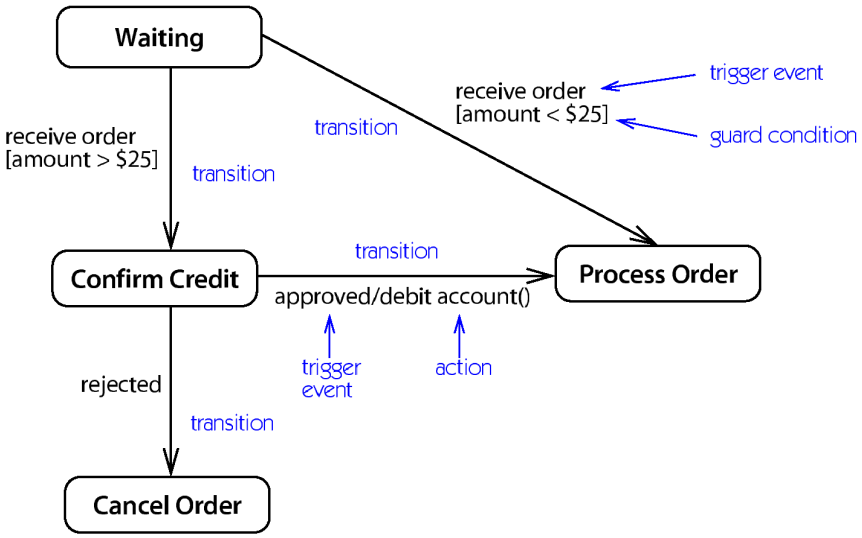


Figure 6-3. *External transitions*

触发事件。触发是会使能迁移的事件的发生。事件可能带有参数，参数可用于迁移中的动作。如果信号具有后代，则任何信号的后代使能迁移。例如，如果迁移将 **MouseButton** 作为触发（参见图 6-1），则 **MouseButtonDown** 仍可以触发该迁移。

事件是不连续的；它发生在时间点。当对象接收事件时，如果不能及时处理，该事件被保存。对象某一时刻处理一个事件。迁移必须在对象处理事件时被激发；事件不会继续被“记忆”（除了特殊的，它们被保存直到触发迁移或直到对象到达它们不被延迟的状态）。如果两个事件同时发生，它们只能依次的被处理。不触发迁移的事件被简单的忽略或遗失。这不是错误。忽略不需要的事件较指明所有的事件要容易一些。

迁移条件。迁移可能伴随布尔表达式的迁移条件。它可能引用拥有该状态机对象的属性，以及触发事件的参数。迁移条件在触发事件发生时被求值。如果表达式为真，则迁移被激发——即结果发生。如果表达式为假，迁移不被激发。迁移条件仅在触发事件发生时被求值一次。如果条件为假，而后为真，迁移是不会被激发的。

相同事件可以作为离开某个状态多个迁移的触发。相同事件的迁移必须具有不同的迁移条件。如果事件发生，且条件为真，事件触发的迁移可能被激发。常常所有的可能性被一系列迁移条件所覆盖，从而事件的发生保证了某个迁移的激发。如果所有的可能性没有被覆盖，且没有迁移被使能，则事件被简单的忽略。对应一个事件的发生仅有一个迁移可以发生（在单个控制线索中）。如果事件使能了多个迁移，则只能有一个会被激发。嵌套状态的迁移在外层的迁移具有优先权。如果两个相互冲突的迁移在同一时间被使能，则激发是不确定的，选择是随机的或依赖于实现细节。建模人员不应该依赖于预计的结果。

结束迁移。缺乏显式触发事件的迁移在状态中的活动结束时被触发（即结束迁移）。结束迁移可能包含迁移条件，它在状态中的活动结束时被求值（而后不会再求值）。

动作。迁移被激发时，它的动作（如果存在）被执行。动作是原子操作和简短的运算，常常是赋值语句或单个的计算。其它动作包括向别的对象发送型号、调用操作、设置返回值、创建和销毁对象以及接触有外部语言指定的的控制动作。动作可以分为动作序列，即一系

列更简单的动作。动作和动作序列是不可以被同时间的动作影响和终止。概念上，它的持续时间同外部事件的计时是可以忽略的。第二个事件在动作的执行期间不会发生。在实际情况中，动作会花一些时间，到达的事件必须被放置在队列中。

系统可以同时执行多个动作。当我们称动作为原子时，并不暗示整个系统是原子的。系统可以在多个动作中进行硬件中断和分时。动作在自己的处理线索中是原子的，一旦开始，它必须结束且不能为其它同时间的动作所影响。动作不能用于长时间的事务机制，它们的时间与外部响应时间相比应是较短的。否则，系统不能用时间相关的方式来响应。

动作可能使用触发事件的参数和属主对象的属性作为它表达式的一部分。

表 6-3 列举了动作的种类和对应的描述。

表 6-3：动作的种类

动作种类	*描述*	*语法*
赋值（assignment）	设置变量的值	target := expression
调用（call）	调用目标对象的操作；等待操作的结束；可能有返回值	opname（arg, arg）
创建（create）	创建新对象	new Cname（arg, arg）
销毁（destory）	销毁对象	object.destory（）
返回（return）	为调用者指定返回值	return value
发送（send）	创建信号实例，并发送个一个或若干对象	sname（arg, arg）
终止（terminate）	自行销毁属主对象	terminate
未解释（uninterpreted）	特定于语言的动作，如条件或迭代	[language specific]

状态的更改。状态可能嵌套于其它复合状态中。离开外层状态的迁移适用所有内嵌的所有状态。无论何时内嵌状态活动时，迁移均是可以被激发的。如果迁移被激发，迁移的目标状态变为活动态。合成状态适用于表达异常和错误条件，因为迁移适用于所有内嵌状态，而无需每个内嵌状态显式的处理异常。

进入和退出动作。分布在一个或多个嵌套层次的迁移可以进入或退出状态。状态可能具有无论何时进入和退出时均要执行的动作。进入目标状态需要执行附属于状态的进入动作。如果迁移离开原先状态，则退出动作在迁移上动作和新状态的进入动作之前执行。

进入动作经常用于执行状态所需的设置工作。因为进入动作不能被跳过。任何状态内部的动作可以认为设置工作已经完成，而无需考虑如何进入状态。类似的，退出动作时无论何时退出状态时均要执行的动作，提供了执行清除工作的机会。这对表达内嵌状态出错条件的高层次迁移非常有用。在类似情况下，退出动作可以执行清除工作以保持对象状态的一致性。概念上，进入和退出动作附带在到达和离开的迁移上，但将它们作为状态的特殊动作允许状态独立于迁移的定义而被封装。

内部迁移。内部迁移具有源状态，但无目的状态。内部迁移的激发规则与改变状态的迁移的激发规则一致。内部迁移无目标状态，所以作为激发的结果，活动状态不会更改。如果内部迁移具有动作，动作会被执行；但状态的变更不会发生，从而进入和退出动作不会被执行。内部状态对于不改变状态的中断动作建模很有用。（如对事件的计数或帮助屏幕的弹出）。

进入和退出动作使用与内部迁移相同的标记方法，除了它们在事件触发名称的位置使用保留字 **entry** 和 **exit**，尽管它们进入或离开状态是被外界迁移所激发。

自迁移在它的状态上调用自身上的进入和退出动作(概念上,它会离开和重新进入状态);同时,它不等同与内部迁移。图 6-4 显示了进入、退出动作和内部迁移。

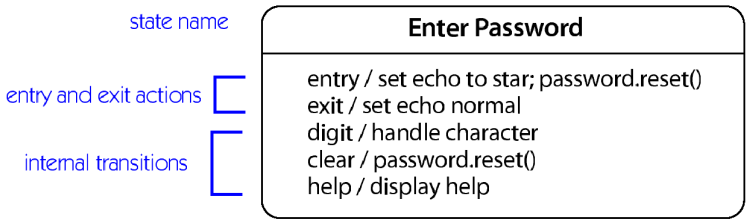


Figure 6-4. *Internal transitions, and entry and exit actions*

复合状态

简单状态无子结构,只有若干迁移和可能的进入和退出动作。复合状态可以分解为连续的或并发的子状态。表 6-4 列举了各种状态的类型。

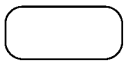
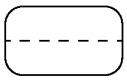
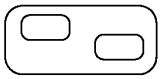
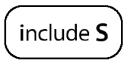
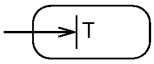
将状态分解为互斥子状态是状态的一种细化。外层的状态细化成多个内部状态。每个子状态继承父状态的迁移,互斥子状态在某一时刻只有一个是活动的。外层状态表现了进入任何一个子状态的条件。

进入或离开复合状态的迁移调用状态的进入动作或退出动作。如果有多个复合状态,则遍布多个级别的迁移可能调用多个进入动作(外层优先)或多个退出动作(内层优先)。如果迁移本身带有动作,它在退出状态之后、进入状态之前执行。

复合状态中可能有初始状态。至复合状态边界的迁移即隐式为至初始状态的迁移。对象从最外层的初始状态开始。类似的,复合状态可包含结束状态。至结束状态的迁移触发复合状态上的结束迁移(无触发迁移)。如果对象到达了最外层的结束状态,它会被销毁。初始状态、结束状态、进入状态、退出状态允许对状态定义独立于迁移进行封装。

图 6-5 显示了状态的顺序分解,包括初始状态。它表示的是提款机的控制机。

表 6-4：状态的种类

状态类型	描述	标记
简单状态	无子结构的状态	
并发复合状态	状态分解为两个或多个的并发子状态，当复合状态活动时，所有并发子状态处于并发活动。	
顺序复合状态	状态包含一个或多个不相交的子状态，当复合状态活动时，在任何一个时刻，只有一个子状态是活动的。	
初始状态	当外围状态被调用时，指明起始状态的伪状态	
结束状态	指明外围状态结束活动的特殊状态。	
汇合状态	链接多个迁移至单个结束迁移的伪状态	
历史状态	保存复合状态中的先前活动状态的伪状态	
引用子状态机状态	引用子状态机的状态，子状态机在被引用的地点插入	
占位状态	子状态机引用状态中的伪状态，用来标识子状态机中的某个状态	

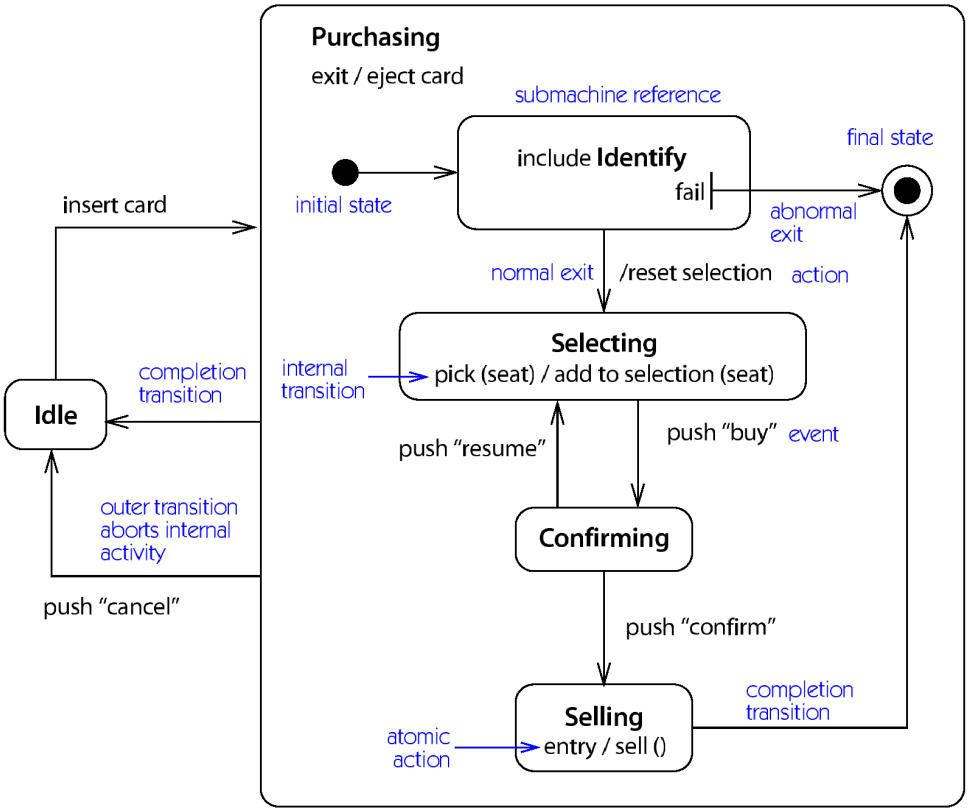


Figure 6-5. *State machine*

并发子状态的分解表达了独立的运算。当并发的超状态被进入时，控制线索的数量会递增。而当离开时，控制线索的数量减 1。每个子状态常常对应一个专门的对象来实现并发，单并发子状态也可表达单个对象中的逻辑并发性。图 6-6 演示了大学选课的并发分解。

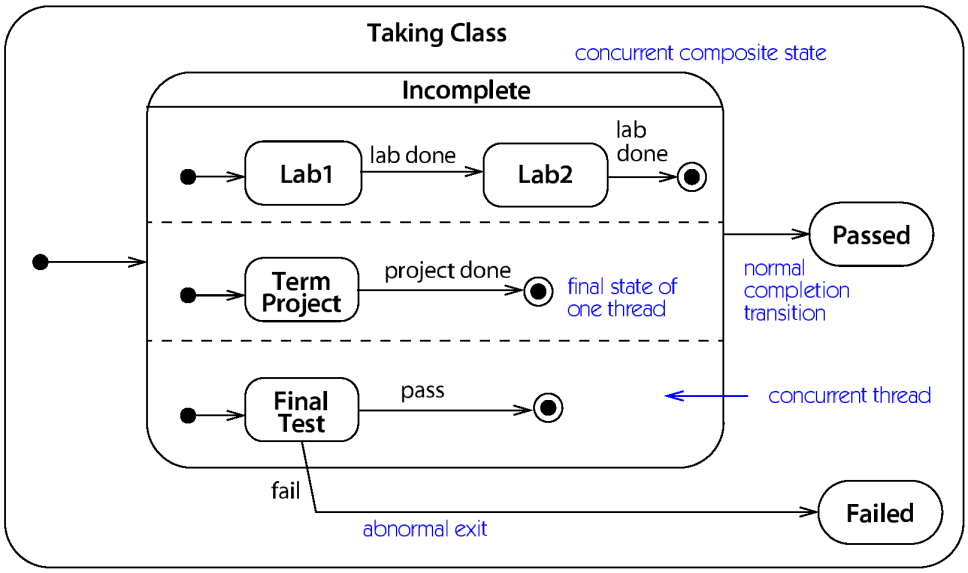


Figure 6-6. *State machine with concurrent composite state*

在状态机中重用其他状态机的片段常常是很方便的。状态机可以被命名并被一个或多个状态机引用。目标状态机被称为子状态机，引用的状态被称为引用子状态机状态。它暗示（概念上）在引用地点使用被引用的子状态机的拷贝来替代。作为子状态机的替代，状态可以使用活动——即需要一段事件来完成的计算或连续事件，它可以被事件打断。图 6-7 显示了子状态机的引用。

至引用子状态机状态的迁移导致目标状态机初始状态的激活。从其他状态进入子状态机，在引用子状态机中放置一个或多个占位状态。占位状态标识了子状态机中的状态。

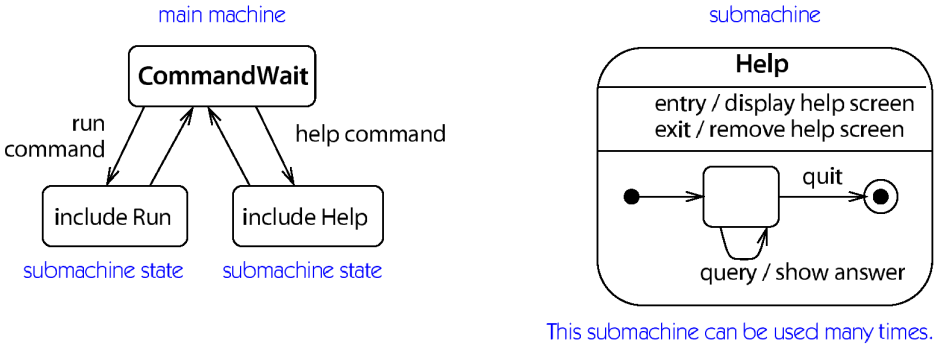


Figure 6-7. *Submachine state*

活动视图（*Activity View*）

概述

活动视图（activity graph）是状态机的一种对计算和工作流建模的特殊形式。活动图的状态代表了运算执行的状态，而非一般对象的状态。通常活动图假设无外界的基于事件的中断来影响计算过程（否则，一般的状态机更适合）。

活动图包含活动状态。活动状态表现了过程中语句的执行或工作流中活动的运行。与一般等待状态等待事件不同，活动状态等待的是运算的结束。当活动结束时，执行处理到图中的下一个活动。前一个活动结束时，活动图中的结束迁移被激发。活动状态通常没有外部事件的迁移，但它们可以由外围状态的事件而被取消。

活动图可能包含动作状态，动作状态与活动状态很相似，但动作状态是原子的且当它活动时不允许迁移。动作状态通常用于短时间的记录操作。

活动图可能包含分支（branch），以及并发线索中控制的分叉（fork）。并发线索代表了可以被组织中不同对象或人员并发执行的活动。并发常常从聚集中衍生，聚集中每个对象具有自己的控制线索。并发活动可以同时或以任何次序运行。活动图与传统的流程图很相似，除了它不但允许顺序控制，而且允许并发控制——非常大的区别。

活动图

活动图（activity diagram）是活动视图（activity graph）的标记形式（图 7-1）。它包含了一些方便使用的速记符号。事实上，这些符号可以用于任何的状态图（statechart diagram）中，尽管混合的标记有时可能会很难看。

活动状态显示为带有活动描述的圆侧边的方框（而一般的状态是带四个圆角的方框）。简单的结束迁移显示为箭头。分支显示为迁移上的迁移条件或者带有多个标签箭头的菱形。控制的分叉（fork）和连接（join）与状态图中的表达方式一致，即进入和离开同步条的多个箭头。图 7-1 显示了订票处理顺序的状态图表。

对于外部事件必须被包括的情况，事件的接收可以显示为迁移上的触发或表示等待信号的特殊内嵌符号。类似的符号表达了发送信号。如果存在多个事件驱动的情况，则状态图可能更加合适。

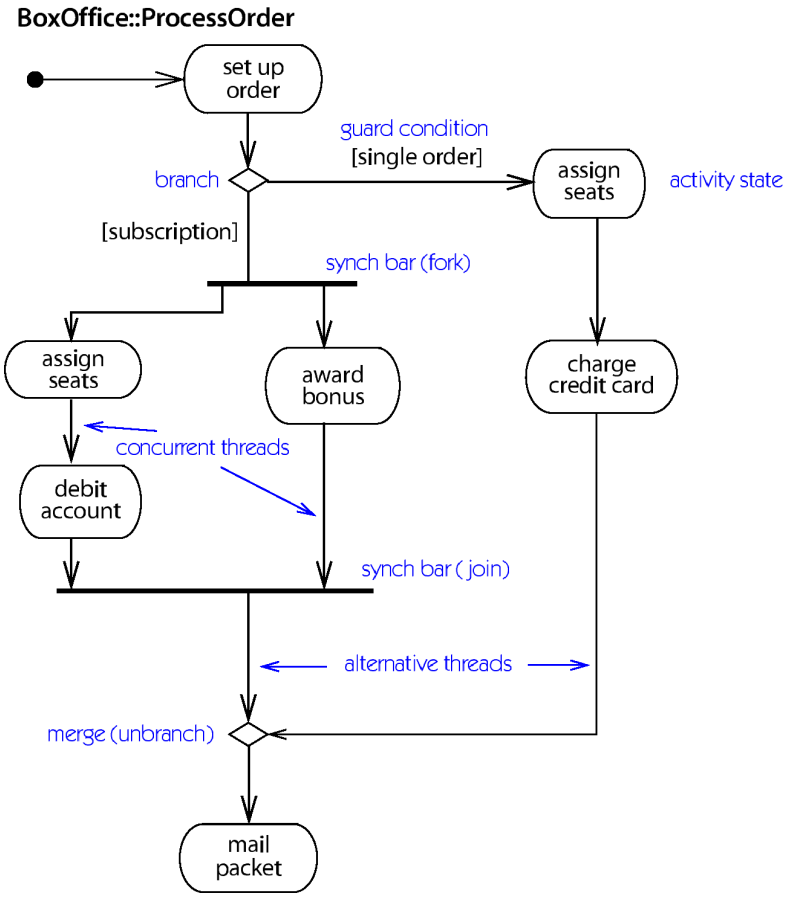


Figure 7-1. *Activity diagram*

泳道。根据责任来在模型中组织活动常常是非常有用的——例如：将由某个商业组织控制的活动划分在一起。这类划分可以通过图中分隔的区域来表达。由于它们的外观，每个区域被称为泳道。图 7-2 显示了泳道。

对象流。活动图不但可以显示控制流，还可以显示对象取值的流。对象流的状态表达了对象是活动的输入还是输出。对于输入值，由对象流状态至活动的虚线来表示。如果活动具有一个以上的输出值或后续的控制流，则箭头从分叉（fork）的符号出发。类似的，多个输入值则汇集至连接（join）符号。

图 7-2 显示了活动和对象流状态均被分派到泳道的活动图。

活动和其它视图

活动图没有显示所有运算的细节。它们显示了活动的流，但是没有显示执行活动的对象。活动图是设计的一个起点。为了完成设计，每个活动必须被扩展成一个或多个的操作，每个操作被指派给特定的对象来实现。上述的指派导致了实现活动图的协作设计。

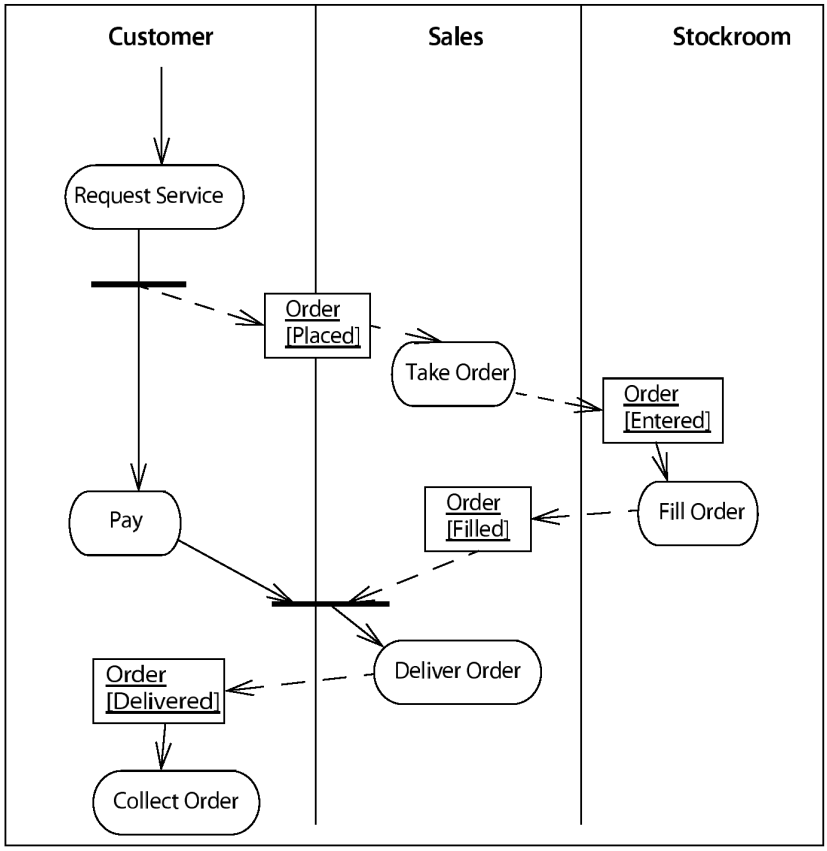


Figure 7-2. *Swimlanes and object flows*

交互视图（*Interaction View*）

概述

对象通过交互来实现行为。交互可以从两个互补的方面来描述，其中一个重点在单个对象，另一个重点在进行合作的对象群体。

状态机是一种窄而深的行为视图，一种观察单个对象的紧缩视图。状态机的说明是精确的，可以直接导出代码。然而，通过它较难理解系统的整体功能，因为状态机集中在单个的对象，而整体系统的行为必须由多个状态机的结果决定。交互视图提供了描述一系列对象行为更全局的视图。该视图用协作来建模。

协作

协作是对上下文中交互实现某种行为对象群体的描述。它描述了许多相互合作的的对象集中起来实现某种目标。协作包括了由对象和连接多填充的空槽。协作槽被称为角色，因为它描述了协作中对象和链的用途。分类角色代表了对参与协作执行的对象的描述；关联角色代表了对参与协作执行的链的描述。分类角色是受到协作中它所充当角色约束的分类；关联角色是受到协作中它所充当角色约束的关联。协作中分类角色与关联角色之间的关系仅在该协作的上下文中有意义。一般而言，离开了协作，相同的关系不会适用与角色对应的分类和关联。

静态视图记录了类的固有属性，如**车辆**具有车主。协作描述了类实例由于在协作中充当特殊的角色而具有的特性，如**租赁车辆**在**租车协作**中有**租车司机**，一般情况它与车辆并不相关，但却是协作中的基本部分。

系统中对象可以参与多个协作。尽管协作的执行通过共享对象连接，但上述情况中的对象是不需要是直接关联的。例如，作为**渡假**模型的一部分，同个人可以成为**租车司机**和**酒店客人**。同个对象可以在相同协作中充当不同的角色，但这不是很常见。

协作具有结构和行为两个方面。结构方面类似与静态视图——它包含了为行为方面定义上下文的一系列角色和关系。行为方面是由绑定于角色的对象间的一系列交换的消息。这些消息在协作中称为交互。协作可以包含一个或多个交互。每个交互描述了协作中用来完成某个目的对象间一系列消息的交换。

状态机是窄而深，协作则是广而浅。协作从对象网络内的消息交换中捕获更全局的行为视图。协作显示了运算中三个主要结构的一致：数据结构、控制流和数据流。

交互

交互是在协作中由分类角色通过关联角色进行交换的一系列消息。当协作在运行期间存在时，绑定于分类角色的对象通过绑定于关联角色的链来交换消息实例。交互对操作、用例或其它行为实体的执行建模。

消息是对象间的单向通信，从发送者至接收者的携带信息的控制流。消息可能带有对象间传递值的参数。消息可能是信号（显式的、命名的、异步对象通信）或调用（具有返回机制的同步操作调用）。

新对象的创建建模成事件，该事件由创建对象产生，由类接收。创建事件对于新的实例作为来自顶层初始状态的迁移上的当前事件仍然有效。

消息可以被安排在顺序的控制线索中。不同的线索表达了并发的消息。线索中的同步由不同线索中消息间的约束来表达。一个同步的构造可以对控制的分叉、连接和分支来建模。

消息序列可以用两种图来表达。顺序图（重点在消息的时间顺序）和协作图（重点在交换消息的对象间的关系上）。

顺序图

顺序图以二维图表来显示交互。纵向是时间轴；时间自上而下。横向显示了代表协作中单个对象的分类角色。每个分类角色表现为垂直列——生命线。在对象存在的时间内，角色显示为虚线；在对象的过程激活时间内，生命线显示为双线。

消息显示为从一个对象生命线出发至另一个生命线的箭头。箭头用从上而下来的时间顺序来安排。

图 8-1 显示了典型的异步消息顺序图。

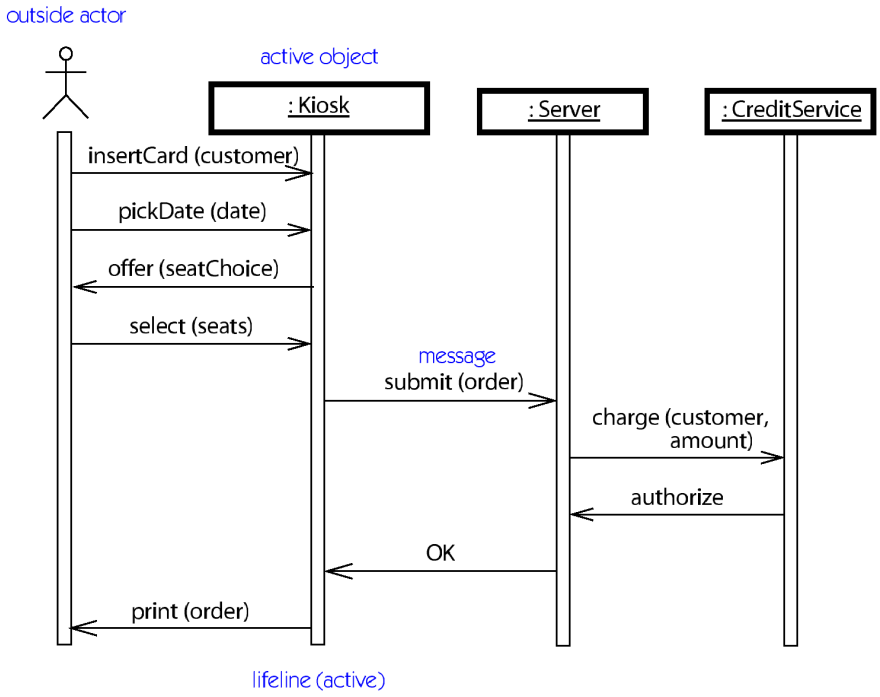


Figure 8-1. *Sequence diagram*

激活

激活是过程的执行，包括它等待内嵌过程执行的时间。它由顺序图中替代部分生命线的双线来显示。调用被指向它所初始的激活的顶端的箭头来表示。递归调用发生在控制进入一个对象的操作时，第二次调用是不同于前次的激活。递归或在同个对象上另一个操作的调用以对激活线的进栈来表达。图 8-2 显示了具有控制程序流的顺序图，包括递归调用和运行中对象的创建。

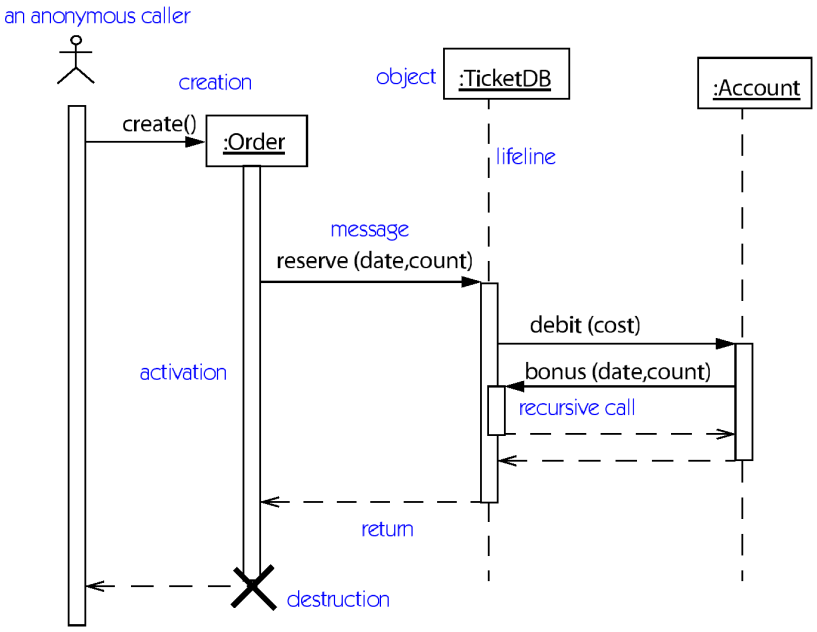


Figure 8-2. *Sequence diagram with activations*

主动对象是保持激活堆栈根的对象。每个主动对象具有平行于其它主动对象的事件驱动控制线索。被主动对象调用的对象称为被动对象；它们仅在被调用是接收控制，在返回时释放。

如果多个并发的线索具有使用嵌套调用的控制程序流。当两个线索在同一个对象上相遇时（例如：合并），不同的线索必须使用线索名称、颜色或其它方法来避免混淆。通常，最好不在单张图上混用过程调用和信号。

协作图

协作图是包含分类角色和关联角色，而非仅包含分类和关联的类图。分类角色和关联角色描述了当协作实例被执行时可能产生的对象和链的配置。当协作被实例化时，对象被绑定至分类角色，而链被绑定至关联角色。关联角色还可能被各种暂时性链来充当，如过程参数或局部过程变量。链符号可以携带指明暂时性的版型（《**parameter**》或《**local**》）或同个对象的调用（《**self**》）。尽管在整个系统中还有其它许多对象，只有与协作相关的对象会被表达。换言之，协作图对实现协作的对象和链进行建模，而忽略其它对象。图 8-3 显示了协作图。

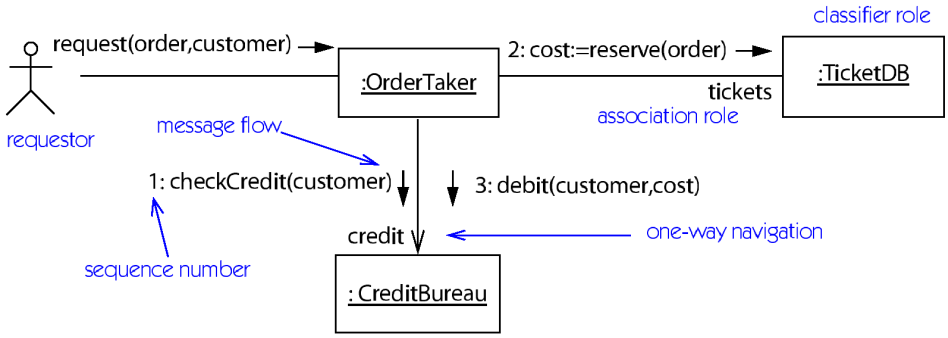


Figure 8-3. *Collaboration diagram*

将对象进行分成四组是很有用的：在整个协作中存在的对象；在交互中创建的对象（约束{**new**}）；在交互中销毁的对象（约束{**destroyed**}）；以及在交互中被创建和销毁的对象（约束{**transient**}）。在设计时，可以由显示在操作起始时可用的对象和链开始，然后再决定如何将控制流向图中正确的对象来实现操作。

尽管协作直接显示了操作的实现，它们还可以显示整个类的实现。该用法中，它们显示实现类所有操作所需的上下文。这允许建模者观察各种操作中对象的多种角色。该视图可以通过联合描述对象全体操作的所有协作来建立。

消息。消息显示为附加在链上的带标签的箭头。每个消息带有顺序号、可选的前驱消息列表、可选的迁移条件、名称和参数表、和可选的返回值名称。顺序号包括了（可选）线索名称。在同一线索中所有消息按次序排序。不同线索中的消息时并发的，除非存在明显的顺序上的依赖。不同的实现细节可能被添加，如同步消息和异步消息的区别。

流。通常一个协作图在整个操作中为一个对象分配一个符号。然而，不同状态的对象有时需要显式的指出。例如，对象可能改变位置，或它的关联在不同时期有很大的改变。对象可以由类和它的状态来表示——状态类的对象。同个对象可以多次显示，每次不同的位置或状态。

表示相同对象的不同对象符号可以用 become 流来连接。become 流是对象状态至另一个状态的迁移。它以带《**become**》版型的虚线来绘制，且可能用顺序号来标记发生的时间（图 8-4）。become 流还可用于表达对象位置的改变。

版型《**copy**》显示了通过拷贝其它对象值产生的对象值，但这并不常用。

表 8-1 显示了对象流关系的种类。

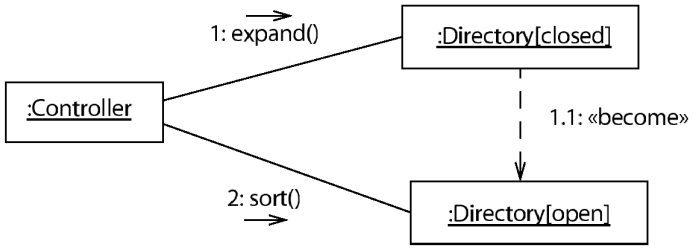


Figure 8-4. *Become flow*

表 8-1：流关系的种类

流	功能	标记
become	从对象的一个值至另一个值的变换	«become»
copy	拷贝对象，拷贝后该对象独立	«copy»

协作图和顺序图。协作图和顺序图均显示了交互，但它们强调了不同的方面。顺序图清晰的显示了时间次序，但没有显式的指明对象间关系。协作图清晰的显示了对象间关系，但时间次序必须从顺序号来获得。顺序图最常用于场景显示；协作图在显示过程设计细节更适用。

模式

模式是连同何时使用指南的参数化协作。参数可以被不同的值替代以产生不同的协作。参数常常为类指定了空槽。当模式被实例化时，它的参数被绑定至类图中实际的类或更大协作中的角色。

模式的使用显示为通过带角色名称标签的虚线连接的虚线椭圆。例如，图 8-5 显示了[Gamma-95]中的观察者模式。在模式的使用中，**CallQueue** 替代了 **Subject** 角色，**SlidingBarIcon** 替代了 **Handler** 的角色。

模式可能出现在分析、体系结构、详细设计和实现层次中。它们是为了重用而捕获最频繁出现结构的一种方式。图 8-5 显示了观察者模式的使用。

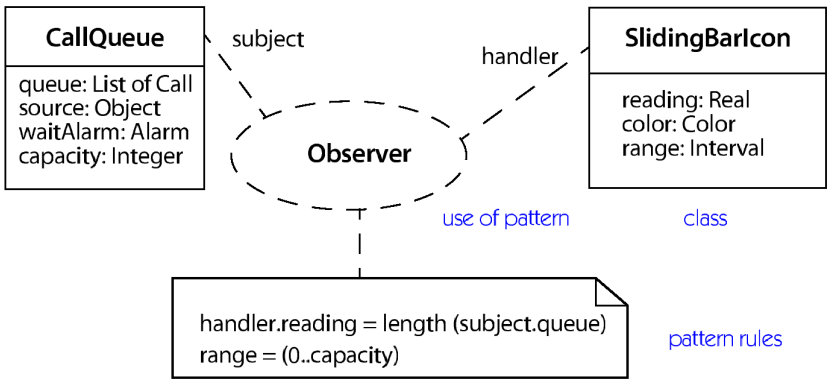


Figure 8-5. *Pattern usage*

概述

许多系统模型是为了独立于最终的实现，显示系统的逻辑和设计。系统实现方面在重用性和性能考虑上是非常重要的。UML 包括了两种视图来表现实现单元：实现视图和配置视图。

实现视图显示了将可重用的系统片段物理打包成可替代的单元，称为构件。实现视图显示采用构件、接口以及构件间依赖，对设计元素（如类）的实现。构件是用于构造系统的高层次的可重用片段。

配置视图显示了运行时段运算资源的物理分布，如计算机和它们之间的互连。它们被称为结点。在运行时，结点可以容纳构件和对象。构件和对象在结点上的分布可以是静态的，也可在结点中转移。如果构件实例及依赖被放置在不同的结点，配置视图可以显示性能上的瓶颈。

构件

构件是作为系统可替换部分，具有良好定义的接口的物理实现单元。每个构件包含了系统设计中某些类的实现。设计良好的构件不依赖其它构件而是依赖构件所支持的接口。该情况下，系统中的构件可以被支持相同接口的构件所替代。

构件具有它们所支持的接口和所需要其它构件的接口。接口是由硬件和软件块支持的操作列表。命名接口的使用允许构件之间的直接依赖被避免，使新构件的替代更加容易。构件视图显示了构件之间依赖的网络。它具有两种形式：一种显示了一系列有效构件（构件库）以及它们的依赖性，系统从它们中被建造。它还可显示为已配置的系统，连同用于建造该系统的构件（从构件库中选取）。该形式下，每个构件与服务被使用的其它构件相连，这些连接必须与构件的接口相一致。

构件用一侧带有两个小长方形的大长方形来表示。它可能与代表接口的圆连接在一起(图 9-1)。

构件图显示了构件之间的依赖（图 9-2）。每个构件实现（支持）一些接口，并使用一些别的接口。如果构件之间的依赖由接口来仲裁，则构件可以用实现相同接口的构件来代替。

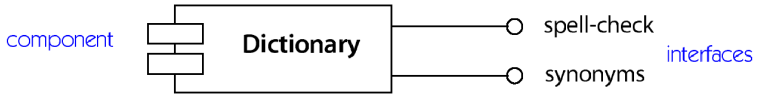


Figure 9-1. *Component with interfaces*

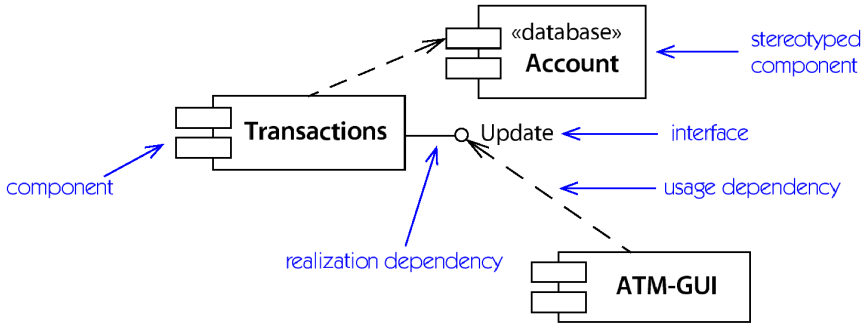


Figure 9-2. *Component diagram*

结点

结点是代表运行资源的运行时的物理对象，它们至少拥有内存且常常具有运算能力。结点可能带有区别不同的资源的版型，如 CPU、设备和内存。结点可以容纳对象和构件实例。

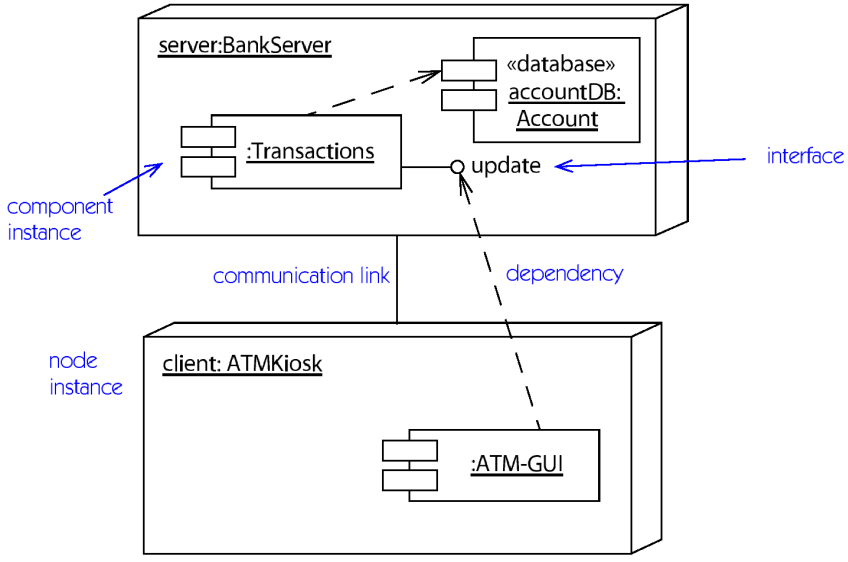


Figure 9-3. *Deployment diagram*

结点表示为具有名称、分类（可选）的风格化的方块（图 9-3）。结点之间的关联代表了通讯路径。关联可带有区分不同种类路径的版型。结点可以使用概括关系将结点的概括描述和更特化的变形联系在一起。

结点中的对象用结点符号中内嵌的对象符号来表达。如果不方便，对象符号可以包括标签 **location**，它的值为对象所驻留结点的名称（它的位置）。对象和构件实例在结点上的迁移也可以被表示。参见 become。

模型管理视图（*Model Management View*）

概述

任何大系统必须被划分为较小的单元，以使人们可以在某一时刻与有限的信息工作，使团队中的工作不相互影响。模型管理包括包（包含了特殊种类的包）和包之间的依赖关系。

包

包是一部分模型。模型的每个部分必须属于某个包。建模人员可以将模型可内容分配至一系列包。但为了可行，分配必须遵守一些基本原则，如通用功能、耦合紧密的实现、相同的视角。UML 不为组建包强加某项原则，但将系统合理的分解为包可以极大的提高系统可维护性。

包包含了顶层模型元素，如类、类之间的关系、状态机、用例图、交互、协作——任何没有被包括在其它元素中的事物。如属性、操作、状态、生命线、消息等包含与其它元素中的元素不直接作为包的内容。每个顶层元素在包中声明，该包是元素的“home”包。它可以被其它包引用，但它的元素内容仅由它所有。在以配置控制的系统中，建模人员必须访问 home 包来修改元素内容。这为大型模型提供了访问控制机制。包也是任何版本控制机制的单元。

包可以包含其它的包。存在一个根包间接的包含了整个系统的模型。有许多方式来组织系统中包。它们可以根据视图、功能或建模人员选择的其它基本特征来划分。包是 UML 模型中具有通用用途的层次组织单元。它们可以被用于存储、访问控制、配置管理以及构建包含重用模型块的库。

如果包经过良好的选择，它们可反映系统的高层次结构——子系统分解以及之间的依赖。包之间的依赖总结了包内容之间的依赖。

包的依赖

依赖由单独的元素所引发，但对于任何规模的系统，它们必须在较高的层次观察。包之间的依赖总结了包元素之间的依赖——即包的依赖从元素个体之间的依赖派生而来。

包之间依赖的出现暗示着在自底向上的路径（存在的声明）中，或者允许在自顶向下的路径（限制了其它关系的约束）中稍后存在着，在相应包内的元素个体间至少存在一个给定依赖类型的关系元素。它是“存在的声明”，并不意味着包中所有元素具有依赖。对于建模人员，它是进一步信息存在的标志，但包一级的依赖本身不包含进一步的信息；它仅仅是个总结。

自顶向下的路径反映了整个系统的体系结构。自下而上的路径可以自动的从元素个体中产生。两种路径在建模中均占有一席之地，即使在同一个系统中。

元素个体之间的同种类的多个依赖被聚集成单个的包级别的依赖。如果元素个体间的依

赖包括版型（如不同的用法），该版型在包级别的依赖中被忽略以产生单一的高级别依赖。

包显示为带方型突起的长方形（桌面的“文件”图标）。依赖显示为虚线箭头。

图 10-1 显示了订票子系统的包结构。它依赖于外界的包以及两个 **Seat selection** 包的变形。任何子系统的实现会包含一个变形。

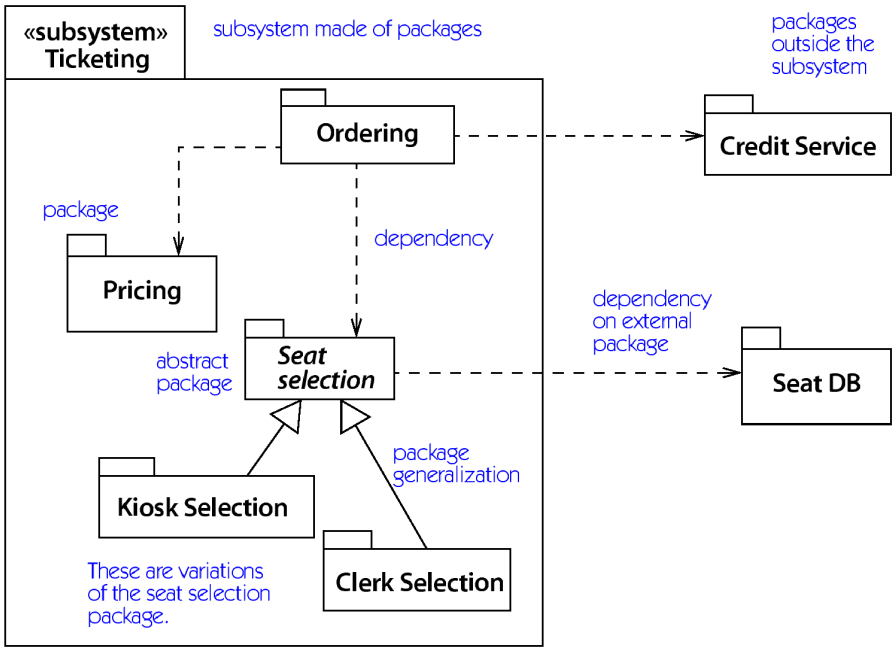


Figure 10-1. *Packages and their relationships*

访问和引入依赖

通常包不能访问其它包的内容。包是不透明的，除非被访问和引入依赖所打开。访问依赖直接应用于包和其它容器。在包级别，访问依赖指供应商包可能被客户包或客户所包含包的元素所引用。供应商中的元素在包中必须具有足够的可见性以使客户可以看见它。通常，包中的被赋予公共可见性的元素在其它包中才可见。可见性为保护的元素仅在该包的后代中可见。可见性为私有的元素在包含它的包中以及该包的内嵌包中可见。可见性同样适用于类的内容（属性和操作）。类的后代可以看见祖先公共的和保护可见性的成员；其它类只能看见公共可见性的成员。引用元素需要访问许可和正确的可见性。所以包中的元素欲访问不相关包中的元素，必须访问或引入第二个包，并且目标元素在第二个包中必须具有公共的可见性。

内嵌在其它包中的包是该容器的一部分，具有对它的完全访问权限。而容器在没有对嵌入的包访问时，可能无法进行存取。内容被封装。

注意访问依赖不会修改客户的名字空间或以任何其它的方式自动的创建引用。它仅仅是对创建引用的许可。引入依赖用于在客户包的名字空间增加名字，作为全路径的别名。

模型和子系统

模型是包含了系统特殊视图完整描述的包。它从某个视点为系统提供很接近的描述。它对于其它的包不具有很强的依赖性，如实现依赖或继承依赖。跟踪关系是不同模型的元素间的弱化依赖形式，它表明了无特定语义暗示的某些连接的出现。

模型通常是树状结构。根包包含了组成给定视点下所有系统细节的内嵌包。

子系统是具有独立的说明和实现部分的包。它代表了与系统其它部分具有整洁接口的清晰单元。它通常代表了系统在功能或实现范围上的划分。模型和子系统均以带版型关键字的包来表达（图 10-1）。

扩展机制（*Extension Mechanisms*）

概述

UML 提供了允许建模人员进行一些扩展，而无需对基本建模语言修改的扩展机制。扩展机制需要设计成使工具在无需理解它们的全部语义情况下能进行存储和操纵。处于该原因，扩展设计为字符串来存储和操纵。对于不了解它们的工具，它们仅仅是文字，但可以作为模型的一部分输入和储存，及传递给其它工具。并期望一些后端和附加工具可以处理各种扩展。这些工具为它们所需的扩展定义特殊的语义和标记。

该扩展方法可能不能满足所有的需要，但我们认为它能用易于实现的方式容纳大部分剪裁的要求。

UML 扩展机制包括约束、标签值和版型。

需要注意的是：根据定义，扩展偏离了 UML 的标准形式，并可能导致互用性问题。建模人员在使用之前必须仔细的衡量它的代价和所带来的益处，特别是当已有的机制能合理的工作时。典型的，扩展的意图是针对特殊的应用领域或编程环境，但它产生了 UML 方言，以及方言的所有优缺点。

约束

约束是用文字表达式表达的语义限制。每种表达均有一种解释语言，它可以是正式的数学标记，如集合理论语言；也可是基于计算机的约束语言，如 OCL；编程语言，如 C++；或者是伪语言或非正式的语言。当然，如果语言是非正式的，其解释也是非正式的，必须由人来完成。即使约束表达为正式语言，也不意味着它能自动的被施加。大多数情况下的完全真实的维护超越了当前计算机技术的状态，但至少语义是精确的。

约束可以表达 UML 标记无法表现的限制和关系。它对声明全局的或影响大量元素的条件特别适用。

约束表示为括号中的表达式字符串。它可以附加于列表元素、依赖或者出现在注释中。图 11-1 展示了若干种约束。

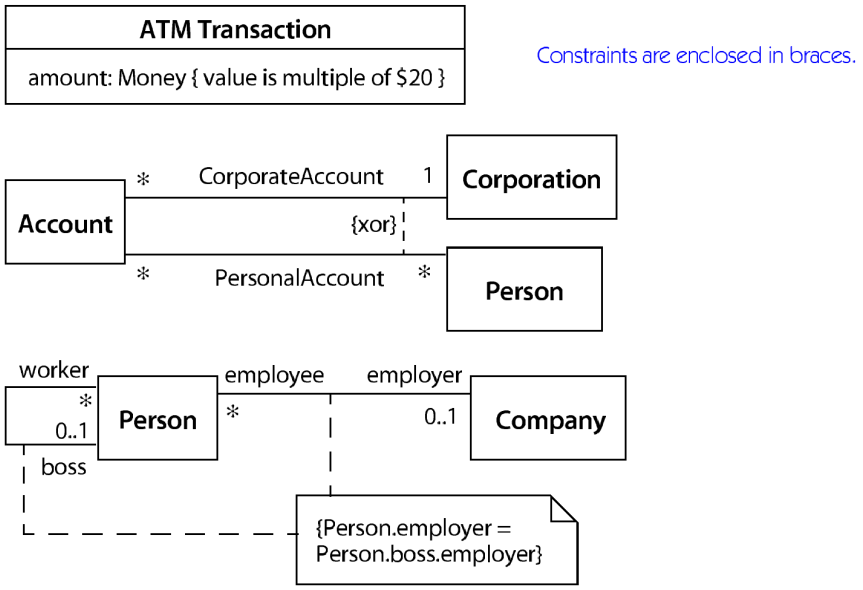


Figure 11-1. *Constraints*

标签值

标签值是存储元素相关信息的文字串对——标签字串和值字串。标签值可以附加在任何独立的元素，包括模型元素和视图元素。标签是建模人员需要记录某些特性的名称，值是给定元素特性的值。例如，标签可以是**作者**（**author**），而值是该元素负责人的姓名，如**Charles Babbage**。

标签值可以用来存储有关元素的任意信息。它们对项目管理特别有用，如元素创建日期、开发状态、完成日期和测试状态。任何文字串可以用于标签，除了内建元模型属性的名称应避免使用（因为标签和属性会一同被认为是元素的属性，被工具统一的存取）。UML 预定义了一些标签名称（见标准元素）。

标签值提供了向元素添加特定实现附加信息的一种方法。例如，代码生成器需要有关代码种类的信息从模型中产生代码。常常有许多方式产生正确的代码；建模人员必须提供相应的选择。某些标签可以作为向代码产生器提供信息的标志，另一些标签可用于其它附加工具，如项目计划和报表书写器。

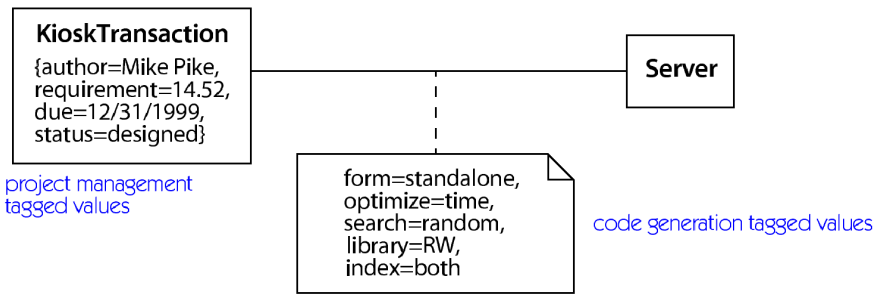


Figure 11-2. *Tagged values*

标签值可以用于存储经版型修饰模型元素的信息。

标签值显示为标签名称、等号和值的文字串。它们通常放置在列表中的括号中（图 11-2）。它们常常在图中被省略，在弹出列表或窗体中显示。

版型

许多建模人员希望能就特定的应用领域剪裁建模语言。这带来了一些危险，因为被剪裁的语言不会被普遍的理解，然而人们常常仍试图这样做。

版型是一种在模型本身中定义的一种模型元素。版型的信息内容和形式与那些现有的基本模型元素一致，但它们的意义和用法不同。例如，商业建模领域的建模者常常希望将商业对象和商业过程区别成特定开发过程中的用途独特的建模元素。它们被认为是一种特殊的类——同样拥有属性和操作，但在与其它元素的关系和使用上具有特殊的约束。

版型基于现有的模型元素。版型化元素的信息内容与现有的模型元素相同。这允许工具用对待现有元素相同的方法来存储和操作新元素。版型化元素可以有自己的图标——这很容易得到工具的支持。例如，“商业组织”可能有看上去象一组人员的图标。版型还可在使用上有若干约束。例如，“商业组织”可能只能与其它“商业组织”相关联。并不是所有的约束可以被通用的工具自动的校验，但可以被理解它们的附加工具来手动的施加和验证。

版型可以使用标签值来存储不能被基本元素支持的附加特征。

版型用放置在基本模型元素符号中或附近的被《》包围的文字串来显示。建模人员还可为特殊的版型创建图标，该图标会替换基本元素的符号（图 11-3）。

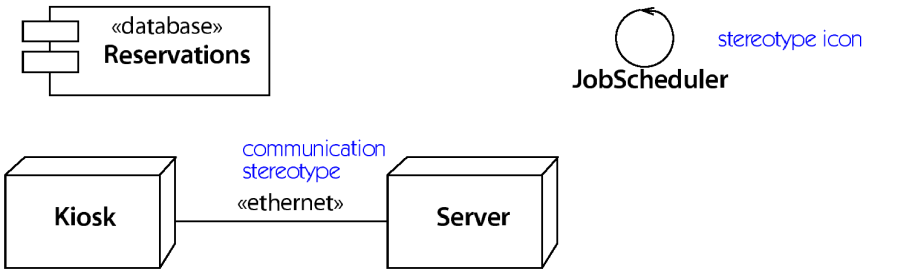


Figure 11-3. *Stereotypes*

剪裁 UML

扩展机制约束、标签值和版型使得可以为特定的领域来剪裁 UML 的配置。许多配置已经被完成和描述。另外一些也由用户所建议。建模语言剪裁的能力意味着应用领域在共享通用和一致的优势概念的同时，根据自身需要来使用建模语言。

UML 标准元素（*UML Standard Elments*）

概述

标准元素是为约束、版型和标签预定义的关键字。它们代表了通用的一些概念，这些概念的与核心概念的区别较小或者重要性不足以被包括至 UML 核心概念。它们与 UML 核心的概念的关系与内建库同编程语言的关系一样。它们不是核心语言本身的一部分，而是在使用核心语言时，用户可以依赖的环境的一部分。

列表还包括了标注关键字——标记符号上的关键字，用来表示其它模型元素，而非版型。

核心概念

元素

元素	定义	元素是模型的原子组成部分。
	标签值	documentation

可概括元素

可概括元素	定义	可概括元素是可参与概括关系的元素。 关键字：《leaf》

行为特征

行为特征	定义	行为特征指的是模型元素的动态特征，如操作或方法。 关键字：《leaf》
	版型	《create》
		《destroy》

属性

属性	定义	属性是在分类中的命名条目，描述了分类实例可能拥有的某个范围的值。
	标签值	persistence

分类

分类	定义	描述行为性和结构性特征的元素。分类的种类包括: 类、接口、数据类型、构件、结点（*Core Package*） 信号（*Common Behavior Package*） 活动者、用例（*Use Case Package*） 子系统（*Model Management Package*）
	版型	《metaclass》
		《powertype》
		《process》
		《thread》
		《utility》
	标签值	persistence
		sematics
		location
类	定义	类是对共享相同属性、操作、方法、关系和语义的一系列对象的描述。
	版型	《implementationClass》
		《type》
接口	定义	接口是刻画一个元素行为的具有名称的操作集合。
数据类型	定义	数据类型是不具有标识的值（即纯粹的值）。 分类符号上的关键字:《enumeration》
构件	定义	构件是打包了实现和提供了一系列接口实现的，物理的、可替换的系统部分。 构件实例符号上的关键字：location
	版型	《document》
		《executable》
		《file》
		《library》
		《table》
结点	定义	结点是代表运算资源的运行时物理对象，通常它至少拥有内存及常常具有处理能力，构件可以被配置在结点上。
信号	定义	信号是实例间异步激励的说明。
活动者	定义	活动者定义了实体的用户在与实体交互的过程中，可以充当的一系列相关的角色。
用例	定义	用例用于在不暴露实体内部结构前提下，定义系统或其它语义实体的行为。
子系统	定义	子系统是代表了物理系统的行为单元的模型元素分组。另外，子系统的模型元素可以划分为说明和实现元素，前者以及子系统的操作被后者实现。

关系

关系	定义	关系是模型元素之间的连接。 在元模型中，关系是不具有具体语义，方便使用的术语。它是抽象的。 关系派生出概括、关联、流、依赖和元关系。 元关系表示为依赖符号上的关键字：《instanceOf》。
概括	定义	概括是一般化元素和具体化元素之间的分类关系。具体化元素完全相容于一般化元素（具有一般化元素的所有属性、成员和关系），并可能包含额外的信息。
	版型	《implementation》
	约束	disjoint
		overlapping
		complete
		incomplete
关联	定义	关联定义了分类之间的语义关系。关联的实例是关联分类实例的一系列元组。每个元组值最多出现一次。
	版型	《implicit》
	约束	xor
	标签值	persistence
流	定义	流是对象的两个版本或对象与它的拷贝之间的关系。
	版型	《become》
		《copy》
依赖	定义	用于方便描述关联、概括、流或元关系（如分类与它的实例之间的关系）以外的关系的术语。 依赖的种类包括抽象、绑定、许可和使用。 依赖符号上的关键字：《include》、《extend》*（活动者和用例之间的关系）*、《powertype》
抽象	定义	抽象是联系两元素或元素集的依赖关系，代表了不同抽象层次或来自不同视点的相同概念。
	版型	《derive》
		《realize》 *注：realize 具有单独的标记符号。*
		《refine》
		《trace》
绑定	定义	绑定是模板和模板生成的模型元素之间的关系。 依赖符号上的关键字：《bind》
许可	定义	许可是依赖的一种。它准许模型元素访问其它名字空间的元素。
	版型	《access》
		《import》
		《friend》
使用	定义	使用是一种关系，其中一个元素的实现或操作需要其它元素（或元素集）。 关键字：《bind》
	版型	《call》
		《create》
		《instantiate》
		《send》

操作

操作	定义	操作是服务，对象请求该服务以实现行为。操作具有签名，签名描述了可能的实在参数（包括可能的返回值）。
	标签值	semantics

注释

注释	定义	注释是附加在一个或一系列模型元素上的注解。它不具有语义，但包含了对建模人员有用的信息。
	版型	《requirement》
		《responsibility》

约束

约束	定义	约束是用文字表达的语义条件或限制.
	版型	《invariant》
		《precondition》
		《postcondition》

版型

版型	定义	版型概念提供了区分（标识）元素的一种方法，它们的行为在某些方面好象它们是新的“虚拟”元构造的实例。实例具有与非版型化实例相同结构（属性、关联、操作）。版型可以指明额外的约束和要求应用于实例的标签值。另外，版型可以用于区分相同结构两元素意义上的或用途上的不同。 分类符号上的关键字：《stereotype》

包

包	定义	包是模型元素的分组。
	版型	《facade》
		《framework》
		《stub》
		《topLevel》*（参见 UML 语义 1.3）*
		《system》

分类角色

分类角色	定义	分类角色是协作中的参与者充当的特定角色。它指明了分类的一个具有限制的视图，协作中的要求定义了该视图。实例是分类角色的一个实例。
	约束	new
		transient
		destroyed
实例	定义	实例定义了一系列操作可以应用的实体，它具有储存操作结构的状态。
	约束	new
		transient
		destroyed

关联角色

关联角色	定义	关联角色是一个协作中关联的特定使用。链是关联角色的一个实例。
	约束	new
		transient
		destroyed
链	定义	链是实例间的连接。
	约束	new
		transient
		destroyed

关联端点

关联端点	定义	关联端点是关联的一个端点，它将关联连接至分类。每个关联端点是关联的一个部分。每个关联的关联端点是有序的。
	版型	《association》
		《global》
		《local》
		《parameter》
		《self》

调用事件

调用事件	定义	调用事件代表了请求的接收，同步的调用特定的操作。
	版型	《create》
		《destroy》

标准元素

access

（stereotype of Permission dependency）

（许可依赖的版型）

两个包之间的版型化依赖，指出目的包的公共部分在源包的名字空间内可以访问。

参见： 访问

association

（stereotype of AssociationEnd）

（关联端点的版型）

应用于关联端点（包括链端点或关联角色端点）的约束，指明相应的实例通过真实的关联可见，而非通过瞬时链，如参数或局部变量。

参见： 关联、关联端点、关联角色

become

（stereotype of Flow relationship）

（流关系的版型）

版型化依赖，它的源和目的元素表达了不同时间点的相同实例，但是源和目的潜在的具有不同值、状态实例和角色。从 **A** 至 **B** 的 become 依赖意味着 **A** 成为 **B**，**B** 可能具有不同空间/时间上的新值、状态实例和角色。become 的标记是从源至目的带有关键字《**become**》的虚线箭头。

参见： become

bind

（keyword on Dependency symbol）

（依赖符号上的关键字）

依赖符号上的关键字，指明了绑定关系。它后面紧跟括号括起的以逗号分隔的实参列表。

参见： 绑定、绑定元素、模板

call

（stereotype of Usage dependency）

（使用依赖的版型）

版型化依赖，它的源和目的均是一个操作。call 依赖指明了源操作调用目的操作。call 依赖可以将源操作连接至任何作用域中的目的操作，包括——但不限制于，外围分类的操作及其它可见分类的操作。

参见： 调用、使用

complete

（constraint on Generalization）

（概括上的约束）

应用于一系列概括的约束，指明所有的孩子已经被指定（尽管有些可能被省略），并且额外的孩子不会在以后被声明。

参见： 概括

copy

（stereotype of Flow relationship）

（流关系的版型）

版型化依赖，它的源和目的元素是不同的实例，但是每一个具有相同的值、状态实例和角色（但具有不同的标识）。从 **A** 至 **B** 的 copy 依赖意味着 **B** 是 **A** 的精确拷贝。**A** 的修改不会影响 **B**。copy 的标记是从源至目的带有关键字《**copy**》的虚线箭头。

参见： 访问、become

create

（stereotype of BehaviroalFeature）

（行为特征的版型）

版型化的行为特征，表明指定的特性创建该特性所属的分类。

（stereotype of Event）

（事件的版型）

版型化事件，指明状态机所属的实例（该事件所适用的）被创建。create 只可以用于状态机的顶层初始迁移。实际上，它是可以用于初始迁移的唯一的一种触发。

（stereotype of Usage dependency）

（使用依赖的版型）

create 是版型化依赖，指明客户分类创建供应商分类。

参见： 创建、使用

derive

（stereotype of Abstraction dependency）

（抽象依赖的版型）

版型化依赖，其中源和目的通常是但不必须是同类型的元素。派生依赖指明了源可以从目的元素计算而来。源可能出于如效率等的设计原因，而被实现，尽管它们在逻辑上是冗余的。

参见： 派生、派生元素

destroy

（stereotypoe of BehaviroalFeature）

（行为特征的版型）

版型化行为特征，表明了指定的特征销毁了所属分类的实例。

（stereotypoe of Event）

（事件的版型）

版型化事件，表明了状态机所属的实例（该事件所适用的）被销毁。

参见： 销毁

destroyed

（constraint on ClassifierRole and AssociationRole）

（分类角色和关联角色上的约束）

标明了角色的实例在交互执行的过程之前已存在，但实例在执行结束之前被销毁。

参见： 关联角色、分类角色、协作、创建、销毁

disjoint

（constraint on Generalization）

（概括上的约束）

应用于一系列概括的约束，指明在概括集合中的一个对象不能是一个以上孩子的实例。该状况仅随多重继承出现。

参见： 概括

document

（stereotype of Component）

（构件的版型）

代表文档的版型化构件

参见： 构件

documentation

（tag on Element）

（元素的标签）

对附属的元素的注释、描述或解释。

参见： 注释、文字

enumeration

（keyword on Classifier symbol）

（分类符号的关键字）

枚举数据类型的关键字，它的细节指明了包含一系列标识的域，这些标识是数据类型实例的可难　　这

术语	*中文*	*解释*
multiplicity	**重数**	对一个集合假设的尺寸——允许的基数范围——的说明。重数说明可以用于关联中的角色、组合中的成员、重复和其它目的。基本上，重数是非负整数的（可能为无穷）子集。对比：*基数（cardinality）*。
n-ary association	**多元关联**	三个或三个以上类之间的关联。从类的角度，每个关联的实例是一个 n-元组值。参见：*二元关联（binary association）*。
name	**名字**	用来标识模型元素的字符串。
namespace	**名字空间**	名字可以被定义和使用的模型一部分。名字空间中，每个名字是唯一的。参见：*名字（name）*。
node	**结点**	结点是代表运行时运算资源的分类，它至少具有内存，且通常有处理能力。运行时对象和构件可能驻留在结点上。
object	**对象**	具有良好定义范围和标识的实体，它封装了状态和行为。状态用属性和关系来表达，行为表达成操作、方法和状态机。对象时类的实例。参见：*类（class）*、*实例（instance）*。
object diagram	**对象图**	包含了某个时间点上对象和它们之间关系的图。对象图可以被认为是类图或协作图的特殊情况。参见：*类图（class diagram）*、*协作图（collaboration diagram）*。
object flow state	**对象流状态**	活动图中的一个状态，代表对象的传递——从一个状态的动作的输出至另一个状态的动作的输入。
object lifeline	**对象生命线**	顺序图中代表一个时间段上对象存在的线段。参见：*顺序图（sequence diagram）*。
operation	**操作**	对象请求的服务，用以实现行为。操作具有签名，它可能限制可能实际参数。
package	**包**	对元素进行分组的通用机制。包可以嵌套在其它的包中。
parameter	**参数**	对可以被改变、传送或者返回的变量的说明。参数可以包含名字、类型和方向。参数用于操作、消息和事件。同义词：*形式参数（formal parameter）*。对比：*实参（argument）*。
parameterized element	**参数化元素**	具有一个或多个未绑定参数的类的描述符。同义词：*模板（template）*。

术语	*中文*	*解释*
parent	**双亲**	在概括关系中，对其它元素（孩子）的概括。参见：*子类（subclass）*、*子类型（subtype）*。对比：*孩子（child）*。
participate	**参与**	模型元素至关系或者细化关系的连接。例如，类参与关联，活动者参与用例。
partition	**分区**	1. 活动图：为动作组织责任的活动图的部分。参见：*泳道（swimlane）*。2. 体系结构：层次结构中，在同一抽象层次或穿越了若干层次一系列相关分类或包。分区代表了体系结构的垂直切片，而层次代表了水平切片。对比：*层次（layer）*。
pattern	**模式**	模板协作。
persistent object	**持久对象**	在创建此对象的进程或线索结束后继续存在的对象。
postcondition	**后置条件**	在一个操作结束时必须为真的限制条件。
precondition	**前置条件**	调用一个操作时必须为真的限制条件。
primitive type	**基本类型**	无子结构的预定义基本类型，如整型，或字符串。
process	**进程、过程**	1. 操作系统中并发和执行的重量级单元。对比：*线索（thread）*，它包括了重量级和轻量级进程。如果必要，可以使用版型来表达实现上的区别。2. 软件开发过程——开发系统的步骤和指南。3. 算法的执行或动态的控制某事物。
projection	**投影**	从一个集合到它的一个子集的映射。
property	**特性**	标志元素特定的具有名称的值。特性具有语义内涵。一些特性在 UML 中预定义；其余可由用户定义。参见：*标签值（tagged value）*。
pseudo-state	**伪状态**	状态机中具有状态的形式的一个顶点，但行为不象状态。伪状态包括初始化及历史顶点。
physical system	**物理系统**	1. 模型的主题。2. 相连的物理单元集合，它可以包括被组织在一起的实现特定目标的软件、硬件和人员。物理系统可以用一个或多个模型来描述，从不同的视角。对比：*系统（system）*。
qualifier	**限定**	一个关联属性或属性元组，这些属性值可以划分出通过关联相关于某个对象的对象的集合。

术语	*中文*	*解释*
recieve [a message]	**接收[消息]**	对发送者实例传送的激励进行的处理。参见：*发送者（sender），接收者（receiver）。*
receiver [object]	**接收者[对象]**	对发送者对象传递的激励处理的对象。参见：*发送者（sender）。*
reception	**接收**	分类准备对接收信号作出反应的声明。
reference	**引用**	1. 模型元素的表示。2. 方便了至其它分类漫游的命名空位。同义词：*指针（pointer）。*
refinement	**细化**	表示对在某个细化层次上已明确的事物的更完整说明的一种关系。例如，设计类是分析类的细化。
relationship	**关系**	模型元素之间的语义连接。关系的例子包括关联和概括。
repository	**仓库**	存储对象模型、接口和实现的设施。
requirement	**需求**	所要求的系统的特征、特性或行为。
responsibility	**责任**	分类的合约或义务。
reuse	**复用**	一个已存在产物的使用。
role	**角色**	参与特殊上下文的一个实体的已命名的特定行为。角色可以是静态的（例如，关联角色）或者是动态的（例如，协作角色）。
run time	**运行时段**	计算机程序执行的一段时间。对比：*建模时段（modeling time）。*
scenario	**场景**	描述行为的特定动作序列。场景可以用于表明交互或用例实例的执行。参见：*交互（interaction）。*
schema [MOF]	**纲要[MOF]**	在 MOF 的上下文中，纲要类似于模型元素容器的包。纲要一致于 MOF 包。对比：*元模型（metamodel）、包（package）。*
semantic variation point	**语义变化点**	元模型语义的变化点。它为元模型语义的解释提供了特意程度的自由。
send [a message]	**发送[消息]**	激励从发送者实例到接收者实例的传送。参见：*发送者（sender），接收者（receiver）。*
sender [object]	**发送者[对象]**	传递激励给接收者对象的对象。对比：*接收者（receiver）。*

术语	*中文*	*解释*
sequence diagram	**顺序图**	显示以时间顺序安排的对象交互的图。特别的，它显示了参与交互的对象和交换的消息序列。与协同图不同的是，顺序图包括时间序列，但不包括对象关系。顺序图可以一般形式（描述所有可能的场景）和实例形式（描述一个实际场景）存在。顺序图和协作图表达了相似的信息，但以不同的方式显示。参见：*协作图（collaboration diagram）*。
signal	**信号**	实例间通信的异步激励的说明。信号可能有参数。
signature	**签名**	行为特征的名字和参数。签名可能包括可选的返回参数。
single inheritance	**单继承**	概括的语义变形，其中一个只有一个超类型。同义词：*多继承[OMA]（multiple inheritance [OMA]）*。*对比：多继承（multiple inheritance）*
specification	**说明**	对某事物是什么或做什么的声明性描述。对比：*实现。（implementation）*
state	**状态**	对象生命期中的一个条件或状况，其中对象满足某些条件、执行某些活动、或者等待某些事件。*对比：状态[OMA]*。
statechart diagram	**状态图**	表示状态机的图。参见：*状态机（state machine）*。
state machine	**状态机**	规定对象或交互在其生命期内对事件的响应所经历的状态序列，以及它的响应和动作。
static classification	**静态分类**	概括的语义变形，其中对象不能改变分类。对比：*动态分类（dynamic classification）*。
stereotype	**版型**	用于扩展元模型语义的建模元素的新类型。版型必须基于元模型中已存在的类型或类。构造型可以扩展语义，但不能扩展预先存在的分类或类的结构。一些版型在 UML 中已被预定义，其它版型可由用户定义。版型是 UML 中三种可扩展机制中的一种。参见：约束条件（constraint），标签值（tagged value）。
stimulus	**激励**	从一个实例至另一个实例的信息的传送，如产生一个信号或调用一个操作。信号的接收通常被认为是一个事件。参见：*消息（message）*
string	**字符串**	一个正文字符的序列。字符串表达细节依赖于实现，可能包含支持国际字符或图形的字符集。

术语	*中文*	*解释*
structural feature	**结构性特征**	模型元素的静态特征，如属性。
structural model aspect	**模型结构性方面**	用于强调系统中对象结构的模型方面，包括它们的类型、类、关系、属性以及操作。
subactivity state	**活动状态**	活动图中代表非原子步骤序列的执行，它持续一定的时间。
subclass	**子类**	概括关系中，对其它类（超类）的具体化。参见：*概括（generalization）*。*对比：超类（superclass）*。
submachine state	**子状态机状态**	状态机中等同于组合状态的状态，但它的内容由其它状态机描述。
substate	**子状态**	作为组合状态的组成部分的状态。参见：*并发子状态（concurrent substate）*、*互斥子状态（disjoint substate）*。
subpackage	**子包**	包含于其它包的包。
subsystem	**子系统**	代表物理系统中行为单元的模型单元分组。子系统提供了接口和操作。另外，子系统的模型元素可以划分成说明和实现元素。参见：*物理系统（physical system）*。
subtype	**子类型**	概括关系中，对其它类型（超类型）的具体化。参见：*概括（generalization）*。*对比：超类型（supertype）*。
superclass	**超类**	概括关系中，对其它类（子类）的概括。参见：*概括（generalization）*。对比：*子类（subclass）*。
supertype	**超类型**	概括关系中，对其它类型（子类型）的概括。参见：*概括（generalization）*。对比：*子类型（subtype）*。
supplier	**供应商**	提供服务的分类，服务可被其它分类调用。对比：*客户（client）*。
swimlane	**泳道**	活动图上为活动组织责任的分区。它们典型的对应于商业模型中的组织单元。参见：*分区（partition）*。
synch state	**同步状态**	状态机中的一个顶点，用来同步状态机中的并发区域。
system	**系统**	模型中的最高层次子系统。对比：*物理系统（physical system）*。

术语	*中文*	*解释*
tagged value	**标签值**	作为名字-值对的对特性的显式定义。在标签值中，名字成为标签。UML 中预定义了一些标签；其它的可以由用户预定义。标签值式 UML 中三个扩展机制之一。参见：*约束（constraint）*、*版型（stereotype）*。
template	**模板**	同义词：*参数化元素（parameterized element）*。
thread [of control]	**线索[控制的]**	通过程序、动态模型或其它控制流表现方式的单一路径。以及，用轻量级进程实现主动对象的版型。参见：*进程（process）*。
time event	**时间事件**	表示自从当前状态进入开始所经历时间的事件。参见：*事件（event）*。
time expression	**时间表达式**	决定绝对或相对时间的表达式。
timing mark	**计时标志**	事件或消息发生时刻的标志。计时标志可用于约束。
top level	**顶层**	表示包容层次中最顶层包的包版型。最高层次版型为查找名字定义了外层限制，如同名字空间向外面“看”。例如，顶层子系统代表了子系统包容层次中的最高级别。
trace	**跟踪**	指出了无特定派生规则的、代表相同概念的两个元素之间的历史关系或处理关系的依赖。
transient object	**瞬时对象**	仅在创建某对象的进程或线程的执行过程中存在的对象。
transition	**迁移**	两个状态之间的关系。指明第一个对象将执行某些特定动作，及当指定事件发生和条件满足时会进入第二个状态。在该状态的改变中，迁移称为激发。
type	**类型**	类的版型，用于指明某个领域的实例（对象）以及适用于对象的操作。类型可能不包含任何方法。参见：*类（class）*、*实例（instance）*。对比：*接口（interface）*。
type expression	**类型表达式**	求值为一个或多个类型引用的表达式。
uninterpreted	**未解释的**	实现未被 UML 指明的一个或多个类型的占位符。每个未解释的值有一个相应的字符串表达。参见：*any* [CORBA]
usage	**使用**	为了正确的功能或实现，一个元素（客户）需要另一个元素（供应商）出现的依赖。

术语	*中文*	*解释*
use case [class]	**用例[类]**	与系统活动者交互的动作序列的说明，包括系统（或其它实体）可能产生各种变形。参见：*用例实例（use case instance）*。
use case diagram	**用例图**	显示系统中活动者和用例之间关系的图。
use case instance	**用例实例**	在用例中被指明的动作序列的执行。用例的一个实例。参见：*用例类（use case class）*。
use case model	**用例模型**	使用用例来描述系统功能需求的模型。
utility	**公用**	类型的版型，以类声明的形式组合全局变量和过程。公用的属性和操作分别成为全局变量和全局过程。公用不是基本的模型构造，只是为编程方便而设置的。
value	**值**	类型域中的一个元素。
vertex	**顶点**	状态机中迁移的源或目的。顶点可以是状态或者伪状态。参见：*状态（state）*、*伪状态（pseudo-state）*。
view	**视图**	模型的投影，从一个给定的视角或合适的视点观察，并省略与此视角无关的实体。
view element	**视图元素**	视图元素是一个模型元素集合的文本和／或图形投影。
view projection	**视图投影**	模型元素到视图元素的投影。视图投影为每个视图元素提供位置和风格。
visibility	**可见性**	枚举，其值（公有，保护或私有）显示了它所涉及的模型元素在包含该元素的名字空间外的可见程度。

中英文词汇对照
（*Chinese-English Glossary Comparsion*）

中文顺序对照表

中文	英文
版型	stereotype
绑定	binding
绑定元素	bound element
包	package
包含	include
保护	protected
变更事件	change event
标签	tag
标签值	tagged value
标识	identity
表达式	expression
并发子状态	concurrent substate
不变式	invariant
布尔表达式	boolean expression
参数	parameter
操作	operation
场景	scenario
超类	supperclass
冲突	conflict
重数	multiplicity
抽象	abstract
初始化	initialization
初始状态	initial state
触发	trigger
创建	creation
单子	singleton
当前事件	current event
调用	call
调用事件	call event
动态分类	dynamic classification
动作	action
动作序列	action sequence
对象	object
对象流状态	object flow state
多	more
多态	polymorphic
多重分类	multiple classification
多重继承	multiple inheritence

中文	英文
发送	send
发送者	sender
访问	access
分叉	fork
分隔	compartment
分类	classifier
分类角色	classifier role
分支	branch
复合状态	composite state
概括	generalization
跟踪	trace
公共	public
公用	utility
供应商	supplier
构件	component
关联	association
关联端点	association end
关联角色	association role
关联类	association class
关系	relationship
过程	process
孩子	child
合并	combination
后代	descendant
后置条件	postconditon
互斥子状态	disjoint substate
汇合状态	junction state
活动	activity
活动视图	activity graph
活动图	activity diagram
活动者	actor
活动状态	activity state
激发	fire
激活	activiation
继承	inheritence
间接实例	indirect instance
简单状态	simple state
交互	interaction
交互视图	interaction view

中文	英文
交互图	interactive diagram
角色	role
角色名	rolename
接口	interface
接收者	receiver
结点	node
结束迁移	completion transition
结束状态	final state
进程	process
进入动作	entry action
静态视图	static view
具体	concrete
具体化	reification
聚集	aggregation
开发过程	development process
可更改性	changeability
可见性	visibility
可漫游	navigable
客户	client
快照	snapshot
扩展	extend
类	class
类图	class diagram
类型	type
历史状态	history state
连接	join
链	link
列表	list
流	flow
路径	path
漫游性	navigability
描述符	descriptor
名字空间	namespace
模式	pattern
模型管理视图	model management view
模型元素	model element
目标状态	target state
目标状态	target state
目的状态	destination state
内部迁移	internal transition
派生	derivation
派生元素	derived element
配置视图	deployment view
配置图	deployment diagram
迁移	transition
迁移条件	guard condition
签名	signature

中文	英文
前置条件	preconditon
强类型	powertype
生命线	lifeline
时间事件	time event
实例	instance
实例	instance of
实例化	instantiation
实现	implementation
实现	realization
实现类	implementation class
实现视图	implementation view
使用	usage
事件	event
视图	view
视图元素	presentation element
数据类型	data type
双亲	parent
顺序号	sequence number
顺序图	sequence diagram
瞬时链	transient link
私有	private
私有继承	private inheritance
塑造不良	ill formed
特征	feature
体系结构	architecture
替代原理	substitutability principle
退出动作	exit action
外部迁移	external transition
伪状态	pseudostate
位置	location
文字	
无触发迁移	triggerless transition
系统	system
细化	refinement
线程	thread
线索	thread
限定	qualifier
消息	message
协作	collaboration
协作角色	collaboration role
协作图	collaboration diagram
信号	signal
信号事件	signal event
需求	requirement
许可	permission
依赖	dependency
引入	import

中文	英文
引用	reference
引用子状态机状态	submachine reference state
泳道	swimlane
用例	use case
用例概括	use case generalization
用例视图	use case view
友元	friend
语义	sematics
元关系	metarelationship
源状态	source state
约束	constraint
责任	responsibility
占位状态	stub state
直接类	direct class

中文	英文
直接实例	direct instance
值	value
属性	attribute
注释	note
状态	state
状态机	state machine
状态机视图	state machine view
状态图	statechart diagram
子类	subclass
子系统	subsystem
子状态机	submachine
自迁移	self transition
组、元组	tuple
组合	composition
祖先	ancestor

英文顺序对照表

英文	中文
abstract	抽象
access	访问
action	动作
action sequence	动作序列
activiation	激活
activity	活动
activity diagram	活动图
activity graph	活动视图
activity state	活动状态
actor	活动者
aggregation	聚集
ancestor	祖先
architecture	体系结构
association	关联
association class	关联类
association end	关联端点
association role	关联角色
attribute	属性
binding	绑定
boolean expression	布尔表达式
branch	分支
bound element	绑定元素
call	调用
call event	调用事件
changeability	可更改性
change event	变更事件
child	孩子

英文	中文
class	类
class diagram	类图
classifier	分类
classifier role	分类角色
client	客户
collaboration	协作
collaboration role	协作角色
collaboration diagram	协作图
combination	合并
completion transition	结束迁移
compartment	分隔
component	构件
composite state	复合状态
composition	组合
concrete	具体
conflict	冲突
concurrent substate	并发子状态
constraint	约束
creation	创建
current event	当前事件
data type	数据类型
derivation	派生
derived element	派生元素
development process	开发过程
dynamic classification	动态分类
dependency	依赖

英文	中文
deployment diagram	配置图
deployment view	配置视图
derivation	派生
descriptor	描述符
descendant	后代
destination state	目的状态
direct class	直接类
direct instance	直接实例
disjoint substate	互斥子状态
entry action	进入动作
event	事件
exit action	退出动作
expression	表达式
extend	扩展
external transition	外部迁移
feature	特征
final state	结束状态
fire	激发
flow	流
fork	分叉
friend	友元
generalization	概括
guard condition	迁移条件
history state	历史状态
identity	标识
ill formed	塑造不良
initialization	初始化
implementation	实现
implementation class	实现类
implementation view	实现视图
import	引入
include	包含
indirect instance	间接实例
inheritence	继承
instance	实例
instance of	实例
instantiation	实例化
initial state	初始状态
interaction	交互
interaction view	交互视图
interactive diagram	交互图
interface	接口
internal transition	内部迁移
invariant	不变式
join	连接
junction state	汇合状态
lifeline	生命线
link	链

英文	中文
list	列表
location	位置
message	消息
metarelationship	元关系
model element	模型元素
model management view	模型管理视图
more	多
multiple classification	多重分类
multiple inheritence	多重继承
multiplicity	重数
namespace	名字空间
navigable	可漫游
navigability	漫游性
node	结点
note	注释
object	对象
object flow state	对象流状态
operation	操作
package	包
parameter	参数
parent	双亲
path	路径
pattern	模式
permission	许可
polymorphic	多态
postcondition	后置条件
powertype	强类型
preconditon	前置条件
presentation element	视图元素
private	私有
private inheritance	私有继承
process	进程、过程
protected	保护
pseudostate	伪状态
public	公共
qualifier	限定
realization	实现
receiver	接收者
reference	引用
refinement	细化
reification	具体化
relationship	关系
requirement	需求
responsibility	责任
role	角色
rolename	角色名
scenario	场景

英文	中文
self transition	自迁移
sematics	语义
send	发送
sender	发送者
sequence diagram	顺序图
sequence number	顺序号
singleton	单子
signal	信号
signal event	信号事件
signature	签名
simple state	简单状态
snapshot	快照
source state	源状态
state	状态
state machine	状态机
state machine view	状态机视图
statechart diagram	状态图
static view	静态视图
stereotype	版型
string	文字
stub state	占位状态
subclass	子类
submachine	子状态机
submachine reference state	引用子状态机状态
substitutability principle	替代原理
subsystem	子系统

英文	中文
supperclass	超类
supplier	供应商
swimlane	泳道
system	系统
target state	目标状态
time event	时间事件
tag	标签
tagged value	标签值
target state	目标状态
thread	（控制）线索、线程
trace	跟踪
transient link	瞬时链
transition	迁移
trigger	触发
triggerless transition	无触发迁移
tuple	组、元组
type	类型
usage	使用
use case	用例
use case generalization	用例概括
use case view	用例视图
utility	公用
value	值
view	视图
visibility	可见性

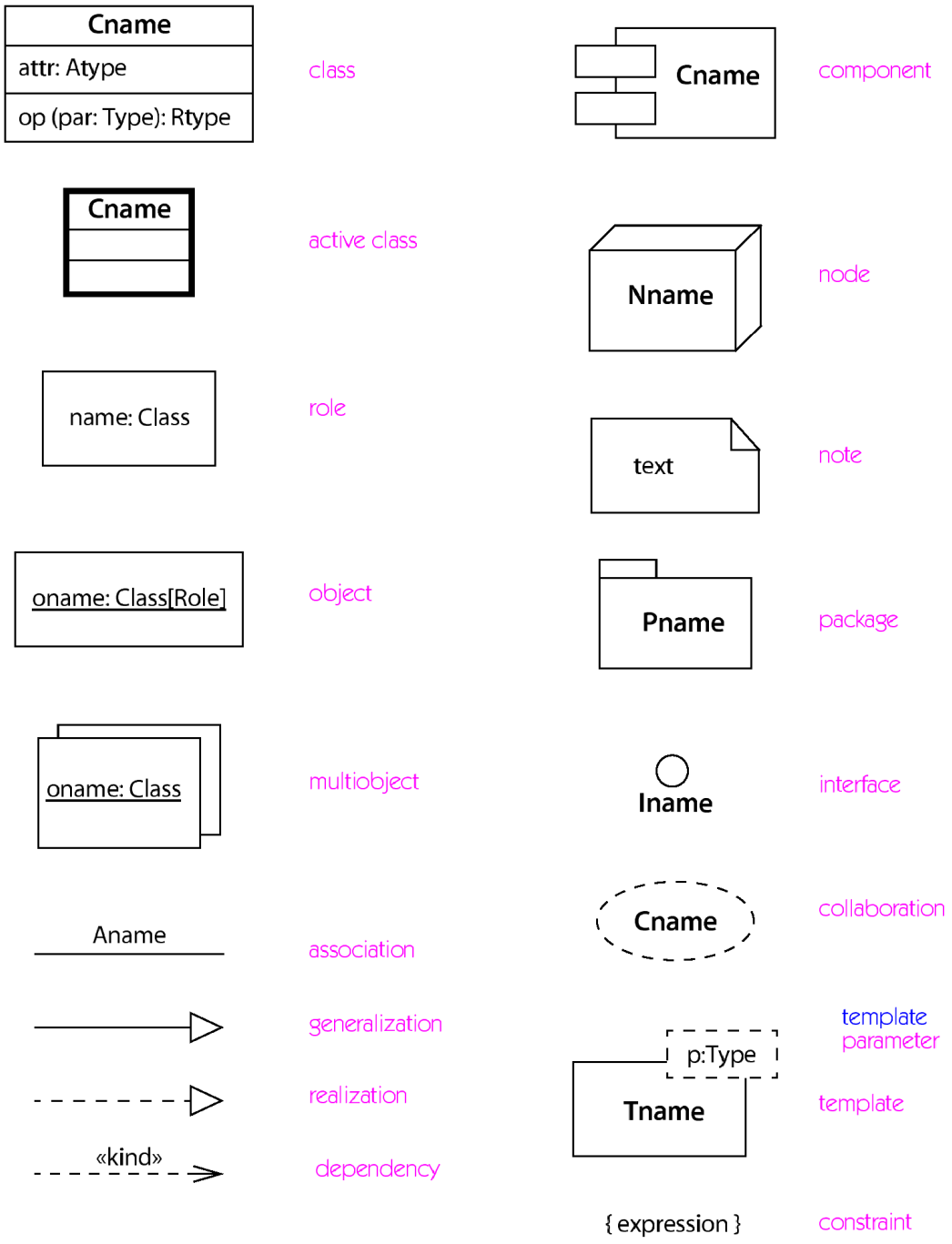


Figure B-1. *Icons on class, component, deployment, and collaboration diagrams*

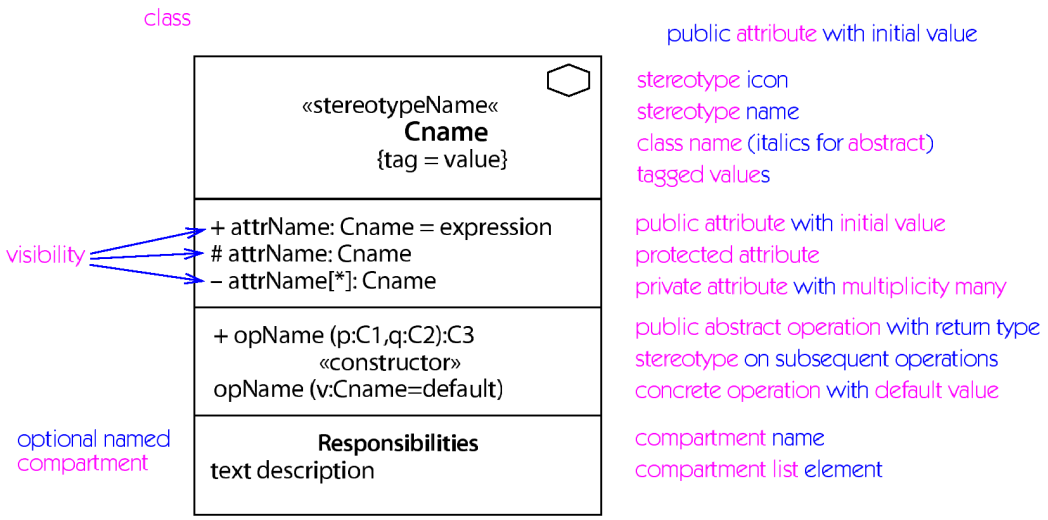


Figure B-2. *Class contents*

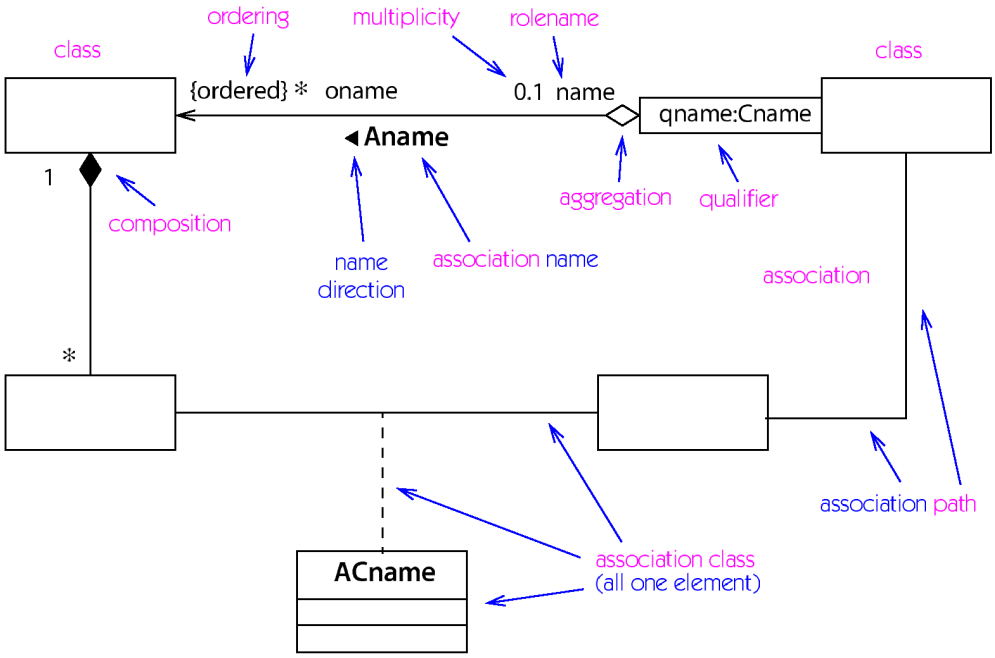


Figure B-3. *Association adornments within a class diagram*

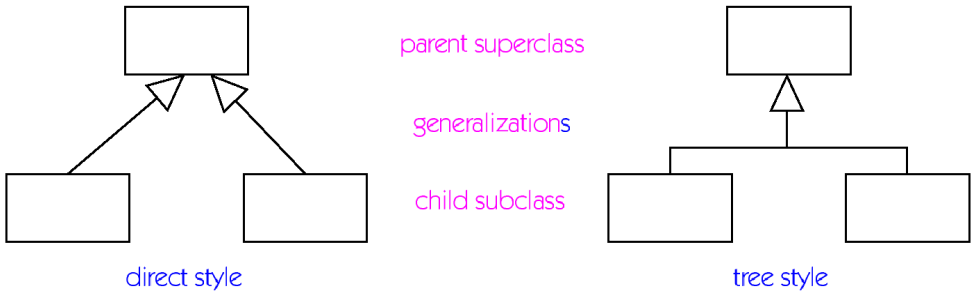


Figure B-4. *Generalization*

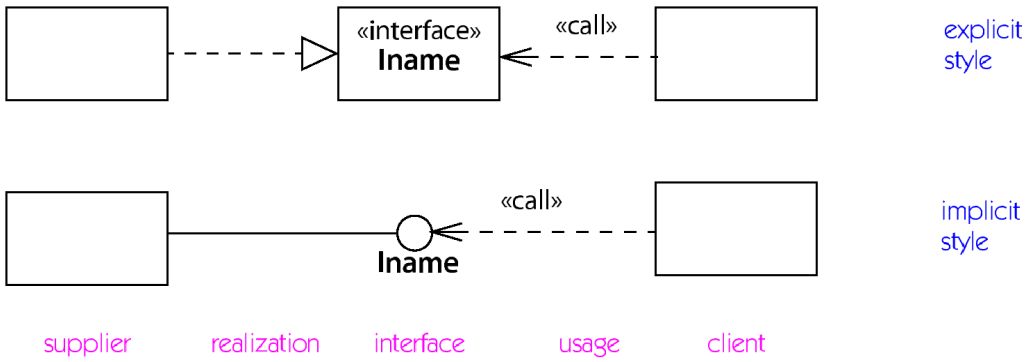


Figure B-5. *Realization of an interface*

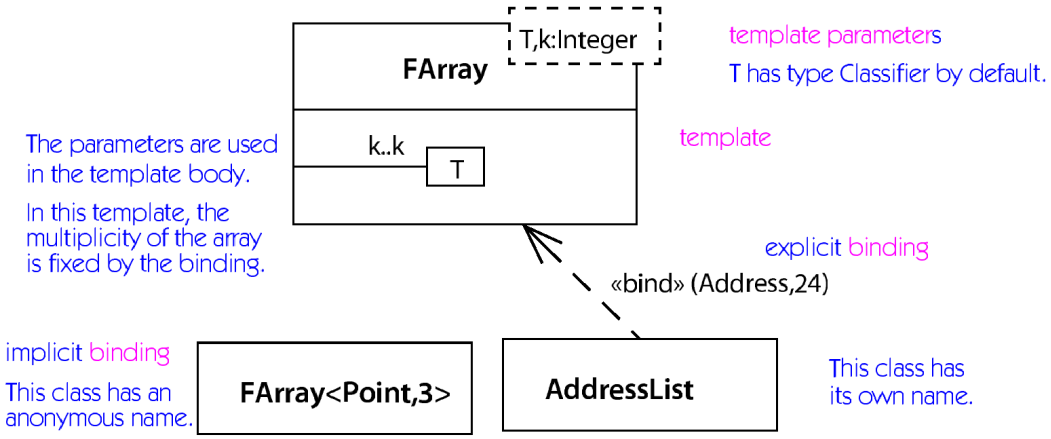


Figure B-6. *Template*

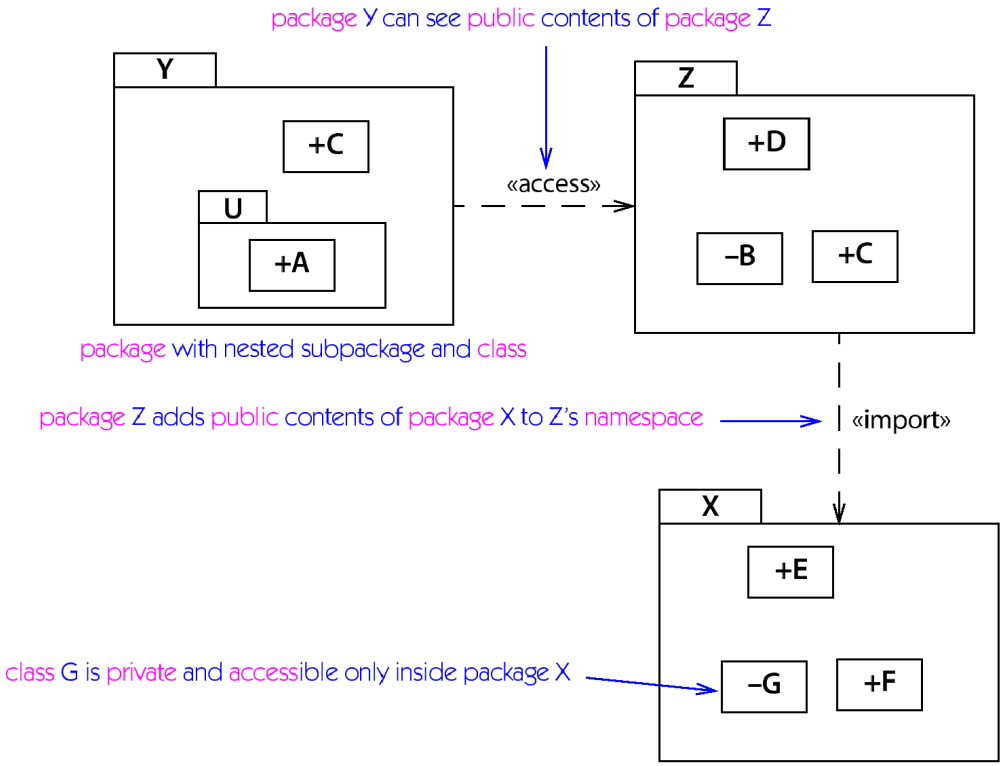


Figure B-7. *Package notation*

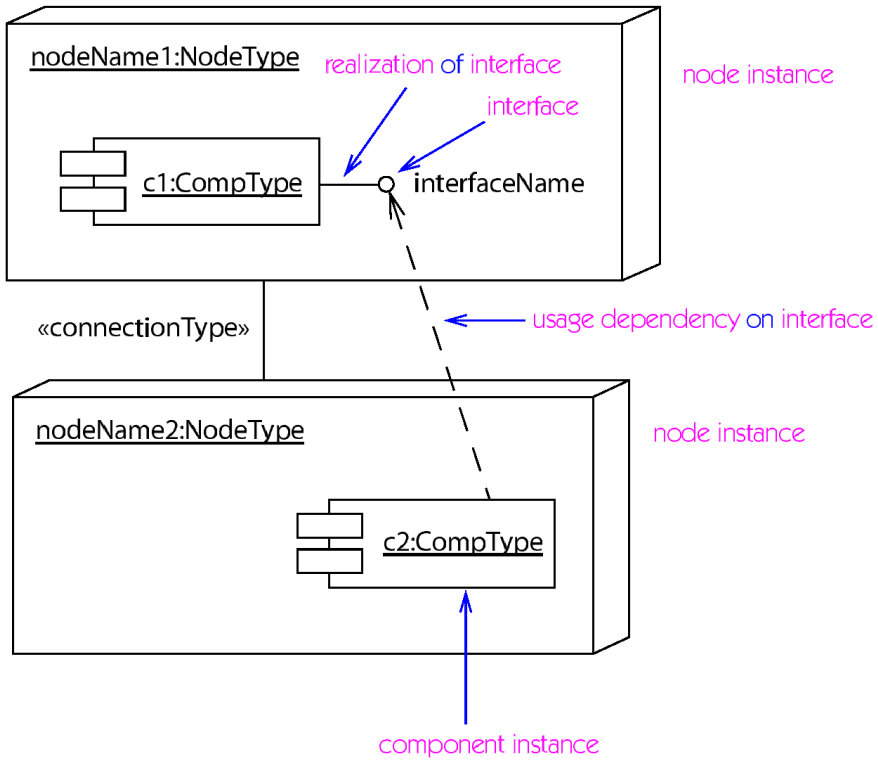


Figure B-8. *Component and node notation*

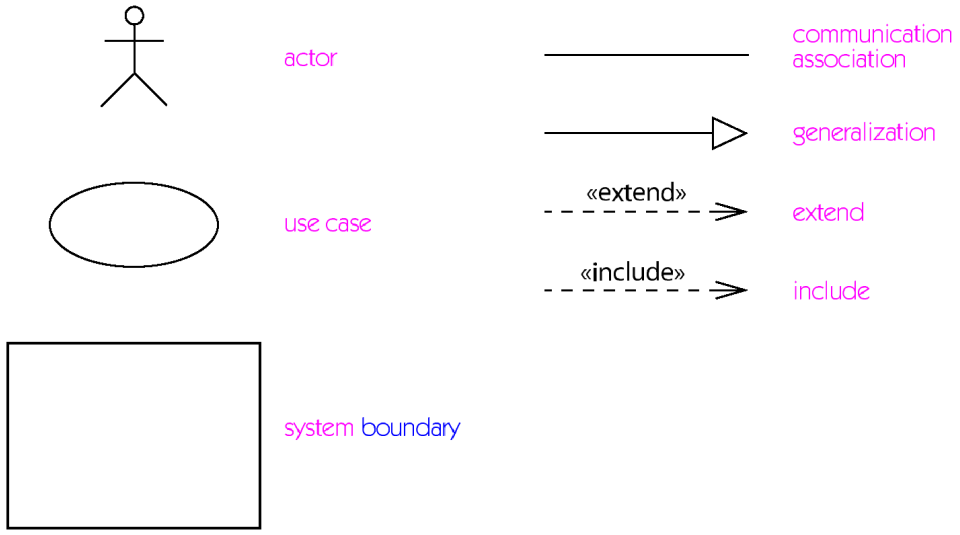


Figure B-9. *Icons on use case diagrams*

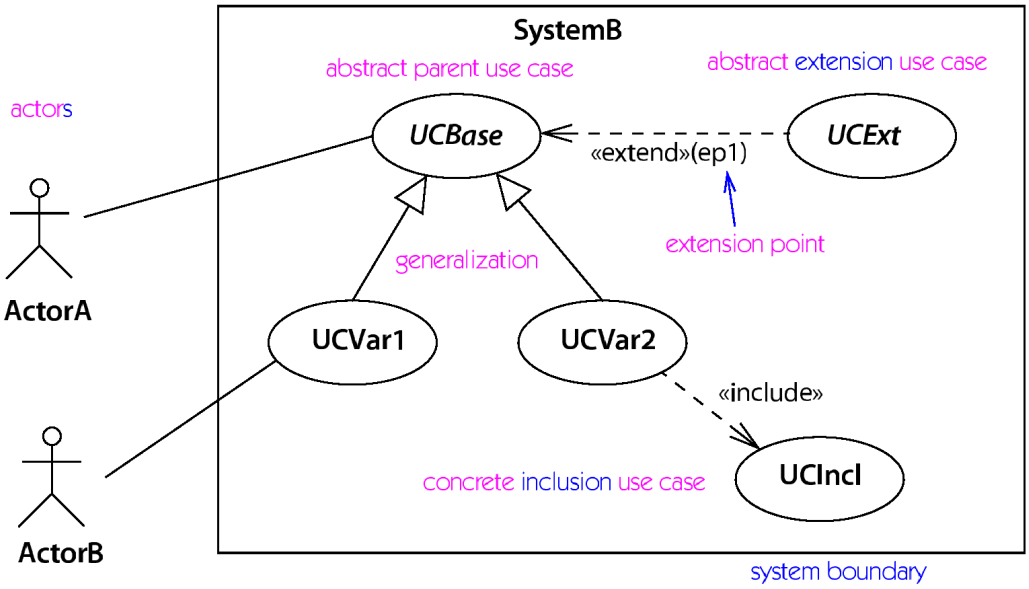


Figure B-10. *Use case diagram notation*

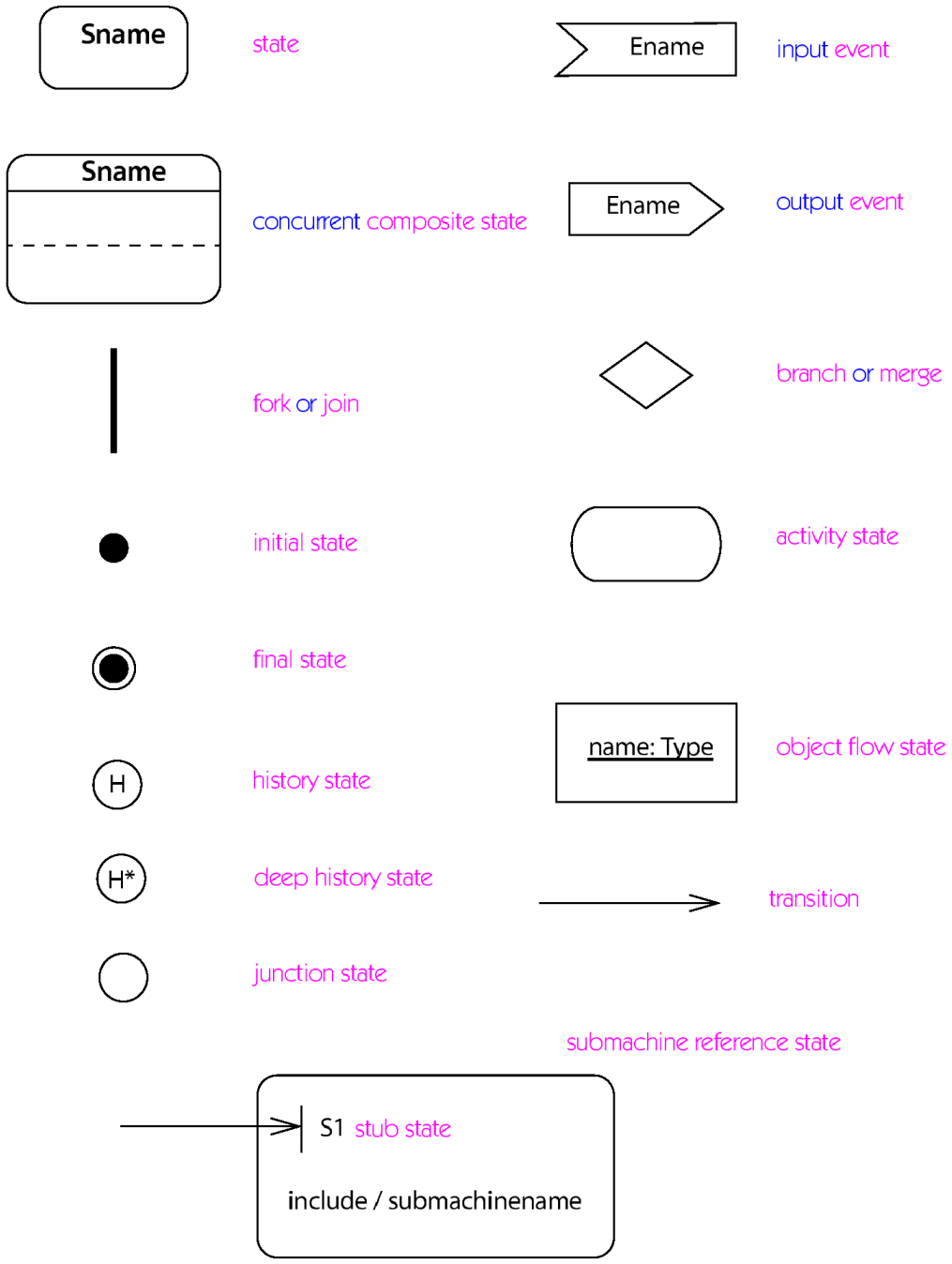


Figure B-11. *Icons on statechart and activity diagrams*

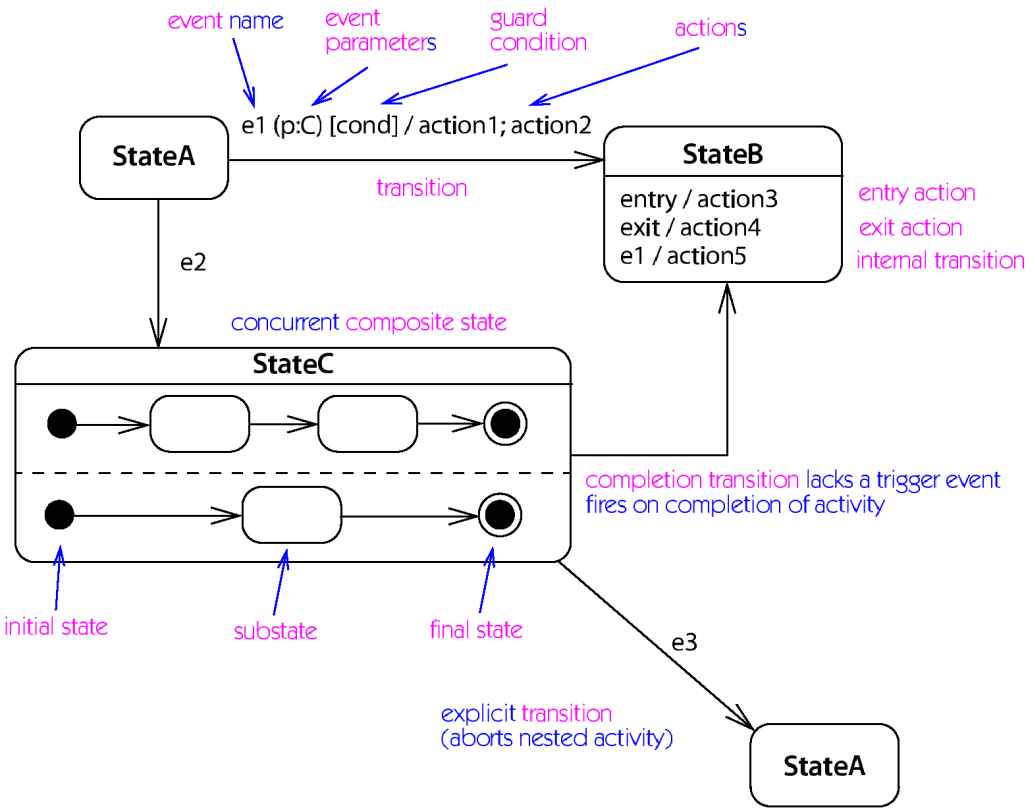


Figure B-12. *Statechart notation*

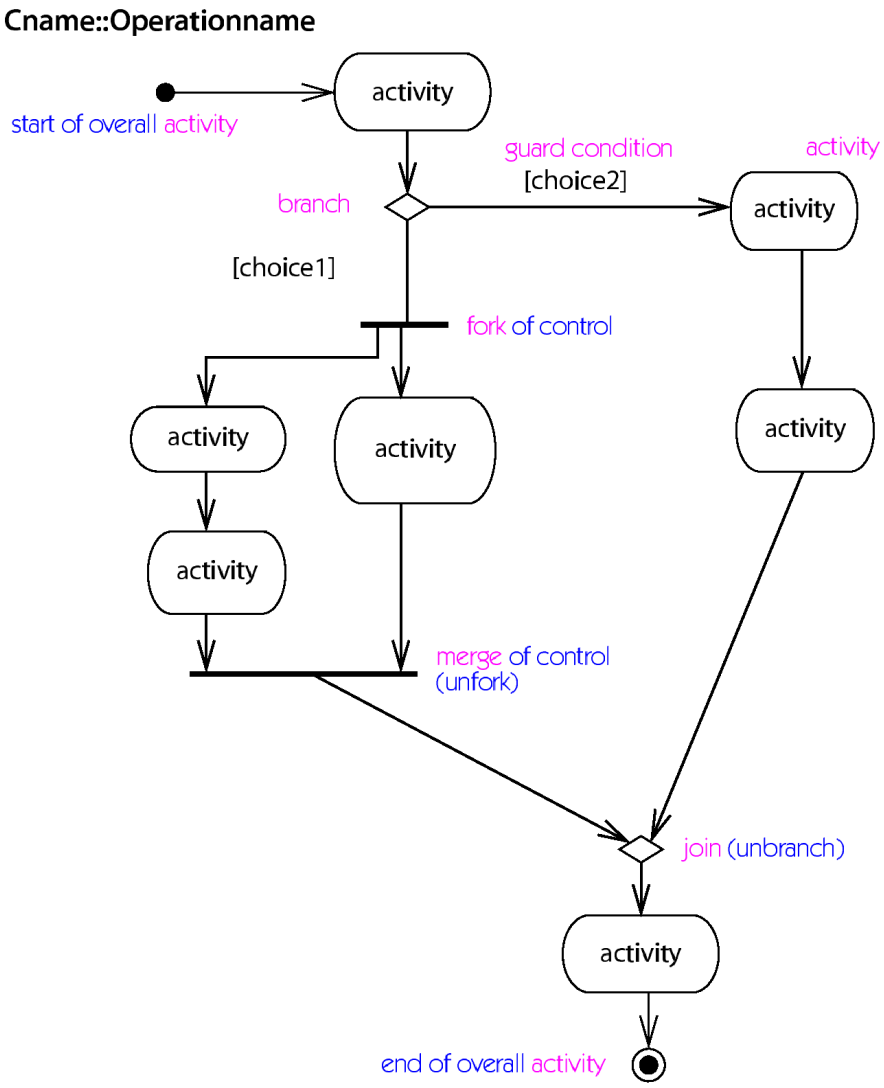


Figure B-13. *Activity diagram notation*

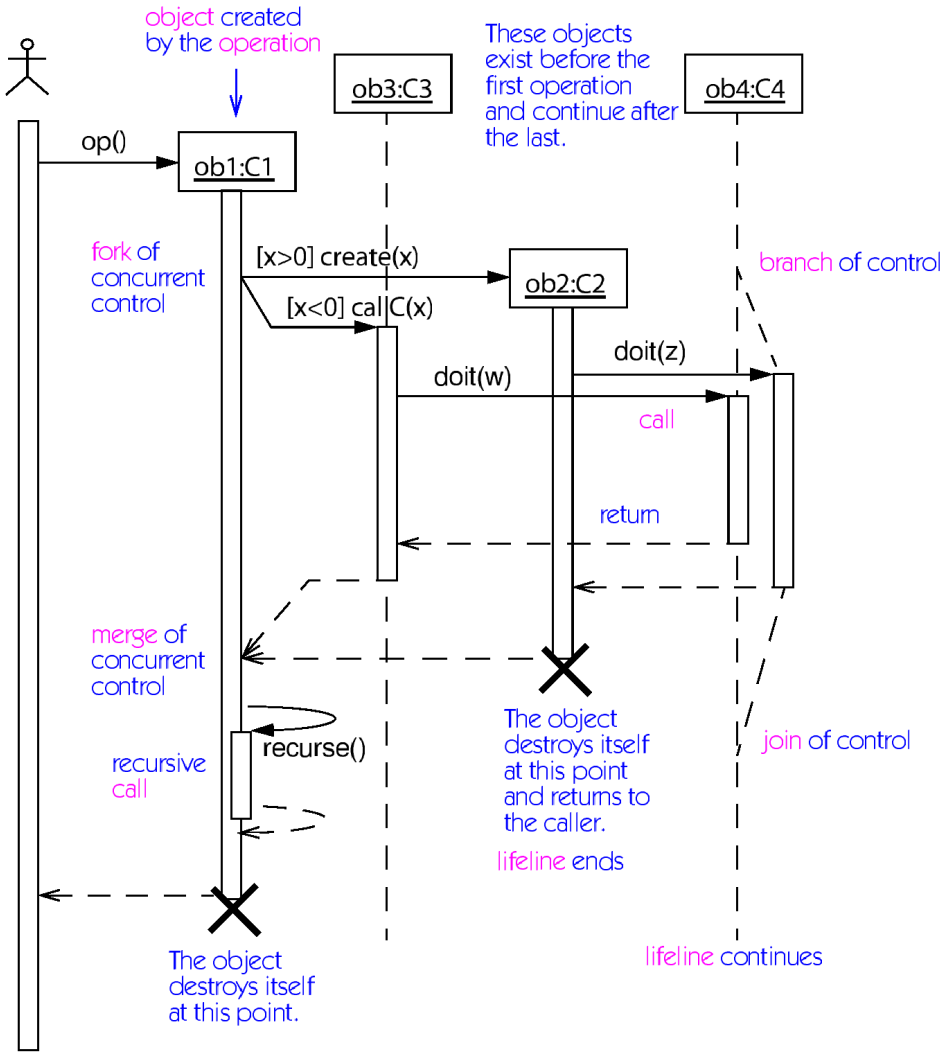


Figure B-14. *Sequence diagram notation*

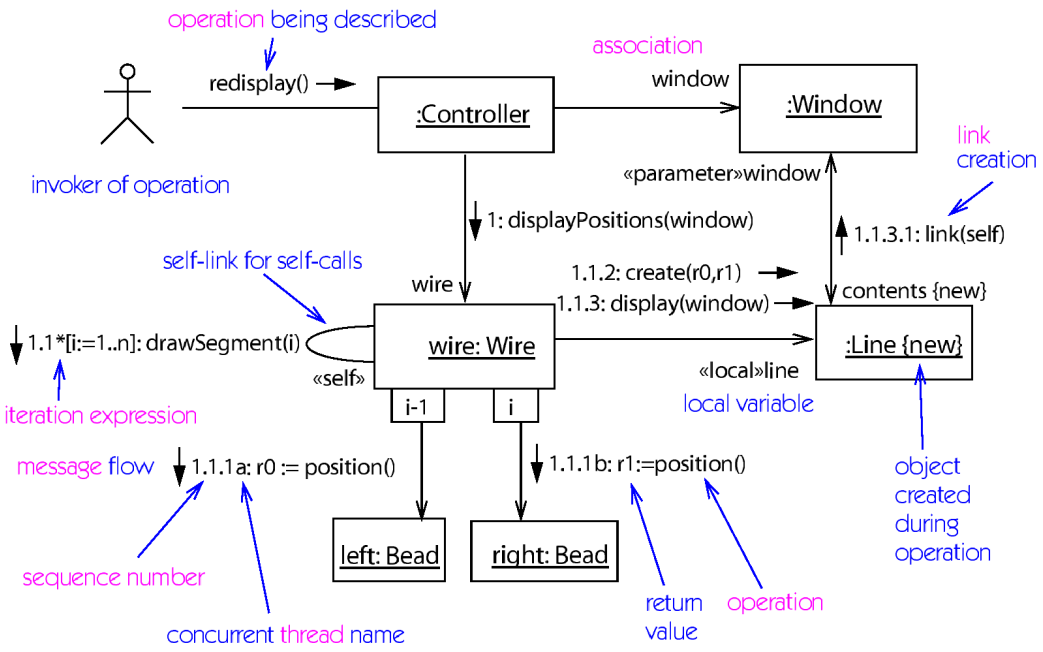


Figure B-15. *Collaboration diagram notation*

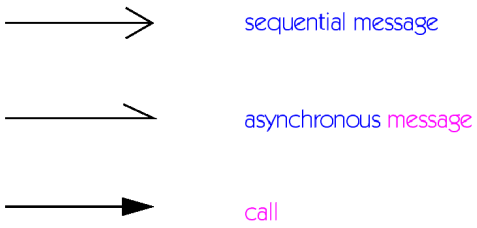


Figure B-16. *Message notation*