面向对象设计 UML 实践

2013 年

目录

第 1 章　UML 导论

The Unified Modeling Language is a visual language for specifying, constructing, and documenting the artifacts of systems. It is a general-purpose modeling language that can be used with all major object and component methods, and that can be applied to all application domains.

— OMG

统一建模语言（*Unified Modeling Language*），简称 UML，按照其设计者所言，是一种通用的可视建模语言，用于规约、可视化、构造并文档化软件系统的体系结构。本章阐述软件开发过程中如何使用模型以及建模语言（如 UML）的作用。文中描述了 UML 的高级结构及其语义的非形式说明，以及设计表示法和代码之间的关系。

1.1 模型与建模

模型在软件开发中的使用非常普遍。本节先介绍模型的两种典型用法，一是在描述现实世界的应用中的用法，二是在实现应用的软件系统中的用法，随后讨论这两种模型之间的关系。

1.1.1 软件模型

软件开发通常按以下的方式进行：在决定了要建立一个新系统之后，就要编写一个非正式的描述说明软件应该做些什么，该描述被称作**需求说明书**（*requirements specification*），通常要经过与系统未来的用户磋商而制定，并且可以作为用户和软件供应商之间的正式合同的基础。

完成的需求说明书被移交给负责编写软件的程序员或者项目组，他们根据说明书编写程序。幸运的话，能够在预算成本内按时完成程序，而且能够满足最初方案的目标用户的需要。但不幸的是，在许多情况下，事情并非如此。

大量软件项目的失败引发了人们对软件开发方法的研究，试图了解项目为何会失败，结果得到了许多对如何改进软件开发过程的建议。这些建议通常以过程模型的形式呈现，描述了开发所涉及的多个活动及其应该执行的次序。

过程模型可以图解说明。例如，图 1.1 表示一个非常简单的过程：直接从系统需求开始编写代码，没有中间步骤。图中除了圆角矩形表示的过程之外，还显示了过程中每个阶段的产物。如果过程中的两个阶段顺次进行，一个阶段的输出通常就作为下一个阶段的输入，如虚线箭头所示。

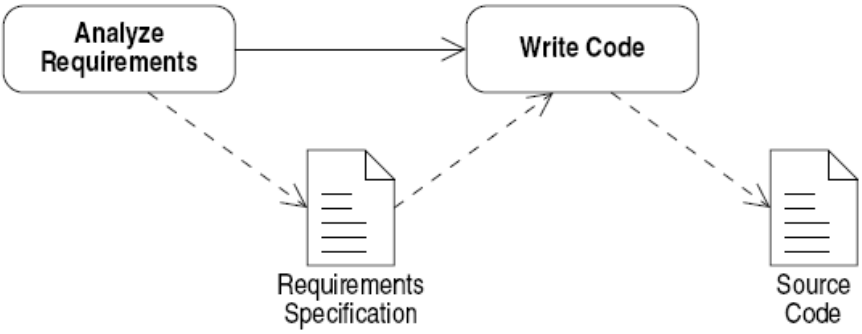


图 1.1 软件开发的原始模型

开发初期产生的需求说明书可以采取多种形式。书面的说明书可以是所需系统的一个非常不正规的概要轮廓，也可以是非常详细、井井有条的功能描述。在小规模的开发中，甚至可能不会以书面形式写下最初的系统描述，而只有程序员对需要些什么的一个非正式的理解。在有些情况下，可能会和未来的用户合作开发一个原型系统，作为后续开发工作的基础。上面所述的所有可能性都包括在“需求说明书”这个一般性术语中，但并不意味着只有书面的文档才能够作为后继开发工作的起点。

还要注意的是，图 1.1 没有描述整个软件生命周期。在本书中，术语“软件开发”是在比较狭隘的意义上使用的，它只包括软件系统的设计和实现，而忽略了生命周期的其他一些重要组成部分。一个完整的项目计划还应该提供诸如项目管理、需求分析、质量保证和维护等关键活动。

单独一个程序员在编写简单的小程序时几乎不需要比图 1.1 更多地组织开发过程。有经验的程序员在写程序时心中会很清楚程序的数据和子程序结构，如果程序的行为不是预期的那样，他们能够直接对代码进行必要的修改。在某些情况下，这是完全适宜的工作方式。

然而，对比较大的程序，尤其是如果不止一个人参与开发时，在过程中引入更多的结构是有必要的。软件开发不再被看作是单独的自由的活动，而是被分割为多个子任务，每个子任务都涉及一些中间文档资料的产生。

图 1.2 描述的是比图 1.1 稍微复杂一些的软件开发过程。在这种情况下，程序员不再只是根据需求说明书编写代码，而是先创建一个结构图，用以表示程序的总体功能如何划分为一些模块或子程序，并说明这些子程序之间的调用关系。

图 1.2 的过程模型表明，结构图以需求说明书中包含的信息为基础，说明书和结构图在编写最终代码时都要使用。程序员可以用结构图让程序的总体结构更清晰，并在编写各个子过程的代码时参考说明书核对所需功能的详细说明。

在软件开发期间所产生的中间描述或文档称为**模型**。图 1.2 中给出的结构图在此意义上即是一个模型的例子。模型展现系统的抽象视图，突出了系统设计的某些重要方面（比如子程序和它们的关系），而忽略了大量的低层细节（比如各个子程序代码的编写）。因此，模型比系统的完整代码更容易理解，通常用模型来阐明系统的整体结构或体系结构。上面的结构图中包含的子程序调用结构就是这里所说的结构的一个例子。

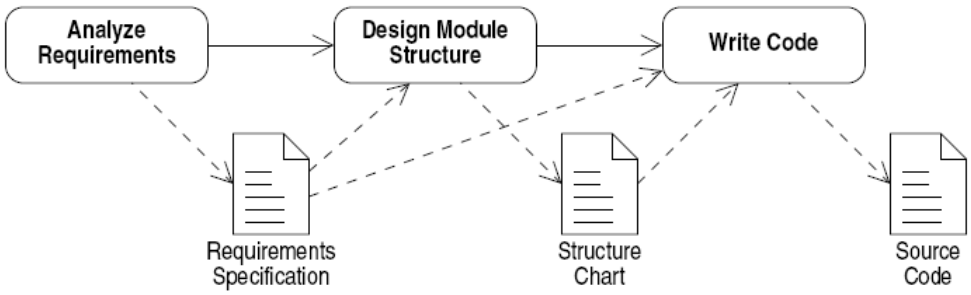


图 1.2 较复杂的软件开发过程

随着开发的系统的规模增大、复杂性增加以及开发组人数的增多，需要在过程中引入更多的正规形式。这种复杂性增加的外部表现之一就是在开发期间使用了更广泛的模型。实际上，软件设计有时就被定义为构造一系列的模型，这些模型越来越详细地描述系统的重要方面，直到充分理解需求，能够开始编程为止。

因此，模型的使用是软件设计的中心，它具有两个优点，有助于处理重大软件开发中的复杂性。第一，系统要作为整体来理解可能过于复杂，模型则提供了对系统重要方面的简明描述。第二，模型为开发组的不同成员之间以及开发组和外界（如客户）之间提供了一种颇有价值的沟通手段。本书描述面向对象设计中所用的模型，并举出了几个使用模型的实例。

1.1.2 应用模型

在软件开发进入系统设计和编码阶段之前，也用模型来帮助开发人员理解系统所针对的应用领域。这些模型通常称为**分析模型**，与此相对的是**设计模型**，如上文所讨论的结构图。这两类模型的区别在于：分析模型不同于设计模型，它不涉及待开发的系统的任何特性，而是力求捕捉“现实世界”中的业务的某些方面和特性。

总之，分析模型和设计模型满足相同的需要并带来同样的益处。它们所支持的或与之相互作用的软件系统和现实世界系统往往都非常复杂，千头万绪。为了管理这种复杂性，系统的描述需要着重于结构而非细节，并要提供系统的一个抽象视图。抽象视图确切的特性将依赖它产生的目的，而且通常需要多个视图或模型，以便足够提供系统的一个全景。

典型地，分析模型描述应用中处理的数据和处理数据的各种过程。在传统的分析方法中，这些模型用图表示，如逻辑数据模型和数据流图。值得注意的是：使用分析模型描述业务过程比业务过程的计算机化要更早，并且与之独立。例如，组织结构图和说明特定生产过程的示意图在商业和工业中已经使用了相当长的时间。

1.1.3 分析模型和设计模型的关系

在开发软件系统期间，上面定义的分析模型和设计模型很可能都要产生。这就引出了一个问题：它们之间的关系是怎样的？

软件开发过程在传统上被划分为若干阶段。分析阶段最终以产生一组分析模型而结束，随后是设计阶段，它产生一组设计模型。在这种情况下，分析模型用来形成设计阶段的输入，设计阶段的任务是创建支持分析模型中规定的特性和要求的结构。

这样划分工作有一个问题：在多数情况下，分析和设计模型产物中使用的是完全不同的语言和表示法，这就导致从一个阶段转移到下一个阶段时需要一个翻译过程，分析模型中包含的信息必须用设计模型要求的表示法重新阐述。

显然，这里存在一个危险，就是这个过程容易出错，而且很浪费。问题是，如果在开发过程剩下的各个阶段中要用设计模型取代分析模型，那为什么还要特意创建一个分析模型呢？而且，如果两种模型之间存在表示法上的差异，就很难肯定分析模型中包含的全部信息都被准确地提取并用设计表示法描绘了出来。

面向对象技术的承诺之一就是，通过对分析和设计使用同样的模型和建模概念，来消除这些问题。按照这种设想，分析和设计模型之间任何明显的差别将会消除。显然，设计模型包含分析模型中未表现出来的低层细节，但分析模型的基本结构在设计模型中能够保持并且可以直接识别。可以期望，这样至少能够消除分析和设计表示法之间的转换所产生的相关问题。

分析和设计使用相同建模概念的后果是这两个阶段之间的区别变模糊了。这种转变最初的动机是希望软件开发能够被视为一个“无缝”的过程：分析将标识现实世界系统中的有关对象，并在软件中直接表示这些对象。从这个观点看，设计基本上就是向基础的分析模型中加入详尽的实现细节，分析模型在整个开发过程中将保持不变。在第 2 章我们详细讨论面向对象的主要概念之后，将更深入地讨论这个观点的真实性。

本书的目的是解释面向对象方法使用的建模概念，说明如何用 UML 中定义的表示法表示模型。本书的中心是设计以及在软件开发中使用设计模型，但是相同的建模概念同样适用于分析模型。分析是不同于设计的技术，要有效地实现分析还要学习许多技巧，但是作为结果的分析模型可以使用书中介绍的表示法完美地表示。

1.2 方法学（Methodology）

软件开发并不是简单地坐在终端前敲入程序代码，在大多数情况下，需要采取中间开发步骤，并且产生程序结构的一些抽象模型，以解决面对的问题的复杂性。这对于只有一个程序员的开发工作和常规的团队开发工作一样适用。

多年以来，已经有许多不同的开发软件策略经过了试验，一些特别成功的或者广泛适用的策略已经形成，并作为**方法学**发表。在软件工程界，术语“方法学”通常用于简单地指一种建议的开发软件系统的策略或方法。如图 1.1 和 1.2 所示的过程模型，它们说明了若干开发活动和各个阶段产生的制品，解释了一种方法学的本质方面。然而，用于“工业化生产”的方法学要比这两个图复杂得多。

方法学至少在两个重要方面指导软件开发。第一，方法学定义了若干模型，这些模型有助于开发一个系统。如上节所说明的，模型在一个抽象层次上描述系统的特定方面，因此使得关于系统的讨论可以在一个适宜的高度层次上进行，而不会过早地涉及低层细节。方法学还定义了一组规范表示法来描绘模型，形成文档。通常，这些规范表示法是图形化的，因而

导致了图形在软件开发中的广泛使用。模型以及描述模型的表示法一般称为方法学定义的**语言**。

在定义语言的同时，方法学还定义了软件开发中包含的各种不同的活动，并指定了执行这些活动应当有的次序。这些活动和次序共同定义了一个开发**过程模型**，过程模型可以用像图 1.1 和 1.2 那样的图形描述。由方法学定义的过程在规范化程度上远不如语言的定义，为了适用于不同需求的应用，甚至是不愿意在写程序之前进行设计的程序员的应用，通常要预想到充分的灵活性，使方法能够在广泛多样的情形下使用。

方法学定义的过程可以被看作是定义了一个项目的概要进度表或计划。这个过程中的每个阶段定义了特定的“可交付的工作成果”（如图 1.1 和 1.2 所示），这些可交付物的形式一般由方法学的语言指定。方法学的使用，除了在软件开发技术方面的帮助外，对项目管理也会有很大帮助。

1.2.1 方法学的分类

已经公开发表的许多方法学之间存在着大量的相似之处，在此基础上可以将方法学分为几大类。这些相似性的出现，是因为不同的方法学对如何描述软件系统的底层结构，有着共同理解的基础。因此，相关的方法学经常会建议在开发中使用非常类似的模型。当然，有时两个模型之间的关系可能因为使用的表示法不通而不明显，这种表面上的差异可能隐藏但是不会消除底层结构的相似性。

众所周知的是被称为**结构化方法**（*structured methods*）的一类方法学，包括结构化分析、结构化设计以及它们的许多变体。这些方法使用的典型模型是数据流图，描述系统中数据如何在不同的处理之间传递。结构化方法描绘的软件系统由一组数据组成，这些数据能够被数据之外的一些函数处理。这些方法特别适于设计数据密集型的系统，通常用于为了特定目的预定要在关系数据库上实现的系统。

另一类方法学由被称为**面向对象**（*object-oriented*）的方法组成。尽管在一些面向对象方法和结构化方法中使用的表示法之间存在相似之处，但是面向对象方法是建立在对软件系统基本结构的完全不同的理解上的，具体的理解将在第 2 章的对象模型中描述。

虽然表示法的细节有很大差异，但是不同的面向对象方法学在关于进行面向对象开发时有效应用的模型种类上却高度一致。因此，以不受某一方法定义约束的一般方式讨论面向对象设计是完全可能的。但是，使用一致的、清晰定义的表示法很重要，本书将使用统一建模语言中的表示法。

1.3 统一建模语言

顾名思义，统一建模语言（UML）是一些早期面向对象建模语言的统一。UML 的三位主要设计师 Grady Booch、Ivar Jacobson 和 James Rumbaugh 在 UML 之前都曾经发表过他们各自的方法，UML 原来是为了通过将这三种方法的深入的理解结合为一体，并发展可用的普

遍公认的统一表示法来促进面向对象技术的传播而准备的。原有这些方法共享的公共框架，以及这三位加入了同一家公司（Rational），也促进了这种结合。

由于 UML 令人印象深刻的起源，以及对象管理组织（Object Management Group，OMG）将其作为标准采纳所带来的推动作用，在 UML 以权威性的形式公布之前，就在软件行业引发了极大的兴趣。Rational 公司于 1997 年发布了 UML1.0，随后 OMG 采纳 UML1.1 作为官方标准。经历了 UML1.2 到 1.5 版本（现统称为 UML1）的演变，OMG 于 2003 年发布了 UML2.0，即 UML2。可以预料，在可预知的将来 UML 将会作为主要的面向对象建模表示法继续下去。

UML 在清楚、明确地区分用于软件设计文档的语言和用于产生文档的过程方面是对早期方法学进行的重大变革。UML 只是定义了一种语言，并没有提供与开发过程相关的描述或建议。这是因为，在软件行业中关于过程很少有一致的意见，并且，大家也逐渐认可，过程的选择关键是受产品的性质和开发的环境影响。UML 的目的是成为一种良好定义的语言，能够和各种各样的不同过程一起有效使用。

UML 在软件开发中的角色的核心是人们使用它的方式。Martin Fowler 描述了使用 UML 的 3 种模式：草图（sketch）、蓝图（blueprint）和编程语言。草图模式是指开发人员通过 UML 交流和沟通系统的某些重要方面，只是选择性地使用，并不作为完整的规范。蓝图模式则是用 UML 图作为详尽完整的系统规约，需要专业工具的支持，通常涉及正向工程和逆向工程。将 UML 作为编程语言则是用 UML 规约系统的所有方面，并编译 UML 图生成可执行代码，即用 UML 作为源代码而不止是模型。当然，这至少需要考虑两个问题：支持工具和 UML 表示法的精确语义。

本节剩余的部分将讨论一些 UML 的基本的高层概念，并描述 UML 对模型进行分类和描述模型的方式。

1.3.1 视图（View）

可以通过软件系统结构的视图来了解 UML，即 **4+1 视图模型**。该模型的 UML 版本如图 1.3 所示。从图中可以清楚地看到，该模型得名于系统的结构通过五个视图描述的事实，其中用例视图具有将其他四个视图的内容结合到一起的特殊作用。

图 1.3 中的五个视图并不对应于 UML 中描述的特定的结构或图，更恰当的是每个视图对应于研究系统的一个特定角度。不同的视图突出特定的参与群体所关心的系统的不同方面，通过合并五个视图中得到的信息就可以形成系统的完整描述，而为了特殊的目的，只考虑这些视图的子集中所包含的信息可能就足够了。

用例视图（*use case view*）定义了系统的外部行为，是最终用户、分析人员和测试人员所关心的。该视图定义了系统的需求，因此约束了描述系统设计和构造的某些方面的其他 4 个视图。这正是用例视图具有中心作用的原因，也是通常所说的用例驱动的开发过程。

设计视图（*design view*）描述的是支持用例视图中规定的功能需求的逻辑结构。它由程序构件（主要是类）的定义以及它们所持有的数据，它们的行为和交互的说明组成。包含在这个视图中的信息是程序员特别关心的，因为如何实现系统功能的细节都在这个视图中描述。

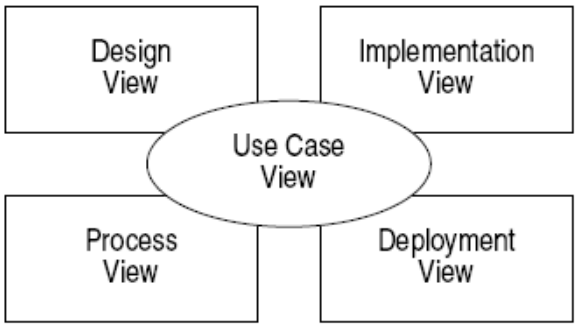


图 1.3　4+1 模型视图

实现视图（*implementation view*）描述构造系统的物理构件。这些构件不同于设计视图中描述的逻辑构件，这些构件包括诸如可执行文件、代码库和数据库等内容。这个视图中包含的信息与配置管理和系统集成这类活动有关。

进程视图（*process view* ）涉及系统中并发性的问题，**部署视图（*deployment view* ）**描述物理构件如何在系统运行的实际环境（如计算机网络）中分布。这两个视图涉及系统的非功能性需求，例如容错性和性能等问题。进程视图和部署视图在 UML 中相对未获得充分的开发，尤其是与设计视图相比。

1.3.2 模型

模型是对现实的简化，模型从特定角度捕捉所建模的事物的一些重要方面，而简化或忽略其余方面。建模是为了更好地理解我们正在开发的系统。对复杂的系统，我们往往要建立多个模型，因为我们很难整个地理解一个系统。

不同的视图对应于研究系统的不同角度。与各个视图相关的信息记录在 UML 定义的各种模型中。例如，用例模型是以用例视图面向的读者容易理解的方式表示用例视图中的信息。

模型也可以在不同的抽象层次产生，或者在开发过程的不同阶段产生。例如，像在 1.1 节中说明的，在开发过程的不同阶段定义系统的分析和设计模型是很常见的。因此，定义这些模型之间的关系并确保它们相互之间的一致性就十分重要。

1.3.3 模型元素

“模型”一词在 UML 的写作中使用得很不严谨。有时它是指正在开发的系统的全部信息，包含所有五个视图，有时也称为*系统模型*。更多的时候是指单个视图中包含的信息的子集。

所有这些用法共有的模型定义的特征是：一个模型由一组**模型元素**组成。这些元素是建模的原子成分，UML 定义了各种各样的不同类型的模型元素，包括我们熟悉的概念，例如类、操作和函数调用。模型是由若干相关的模型元素建立起来的结构。

如果使用 CASE 工具支持开发，工具会用一个数据库来存储所有已声明的系统所知的模型元素的信息。这些信息的总和组成系统模型。

1.3.4 图（Diagrams）

模型通常以一组**图**的形式呈现给设计人员。图是一组模型元素的图形化表示。不同类型的图表示不同的信息，一般是它们描述的模型元素的结构或行为。各种图都有一些规则，规定哪些模型元素能够出现在这种图中以及如何表示这些模型元素。

UML2 定义了 13 种不同类型的图，分为结构图和行为图两大类，如表 1.1 所示。

表 1.1　UML 的图的类型

	图（diagram）	分类	用途
1	类图（Class）	结构图	系统逻辑结构建模
2	对象图（Object）	结构图	系统逻辑结构建模
3	构件图（Component）	结构图	系统的物理构件建模
4	组合结构图（Composite structure）	结构图	类的内部结构建模
5	部署图（Deployment）	结构图	部署的系统建模
6	包图（Package）	结构图	模型元素分组
7	用例图（Use case）	行为图	系统需求建模
8	状态机图（State machine）	行为图	对象的状态建模
9	活动图（Activity）	行为图	系统工作流建模
10	顺序图（Sequence）	行为图/交互图	交互序列建模
11	通信图（Communication）	行为图/交互图	交互链接建模
12	时序图（Timing）	行为图/交互图	交互时序建模
13	交互概况图（Interaction overview）	行为图/交互图	多个交互的高层视图

容易混淆的是，有时候也把图称为模型，因为二者都包含一组模型元素的信息。这两个概念的根本区别是：模型描述的是信息的逻辑结构，而图是它的特殊物理表示。如果使用 CASE 工具，模型对应于工具在数据库中存储的信息，而图对应于这些信息的整体或部分的特定图形表示。

UML 图在形式上主要是图形，因为大多数人发现用图形表示复杂结构比用纯文本表示更容易使用。但是，图形符号能够容易地表达的东西相当有限，所以一般使用一种规范的文本语言作为 UML 的补充，例如对象约束语言（Object Constraint Language）。

1.3.5 理解 UML

学习 UML 要理解两件相关的事情。第一，必须理解多种模型元素以及它们如何在 UML 模型中使用。第二，需要学习各种不同类型的图的细节，以及这些图的图形形式与它们表示的模型元素之间的关系。

模型的结构在 UML 规范中通过**元模型**（*metamodel*）形式地定义，元模型的意思是“模型的模型”。元模型有点像程序设计语言的定义，它定义了不同类型的模型元素、它们的属性和它们可能相关的方式。

明确地描述元模型是可能的，但是开始学习 UML 时更普遍的是将注意力集中在不同类型的图的**语法**或合法结构上。在本书中，用于表示不同模型元素的符号，将在讨论各种图的时候非形式地予以介绍。

除了学习 UML 图的语法，理解它们的**语义**或意义也很重要。设计语言从设计产品的特性方面最好理解。在 UML 中，这些产品通常是面向对象的程序，在本书中，将设计和代码间的关系着重作为理解 UML 的手段。

1.4 设计模型和代码

设计和代码之间的关系比起初看起来的更微妙一些，本节将更详细地说明它们的确切关系。这种关系的重要性在于许多 UML 表示法的含义可以从面向对象程序运行时的特性来理解。

系统设计中使用的模型呈现的是系统的一个抽象视图，而实现增加了足够的细节使这些模型可以执行。如果设计文档和源代码一致，就意味着这些模型是表示了代码结构和特性的一个抽象视图。这种关系如图 1.4 的左部所示。

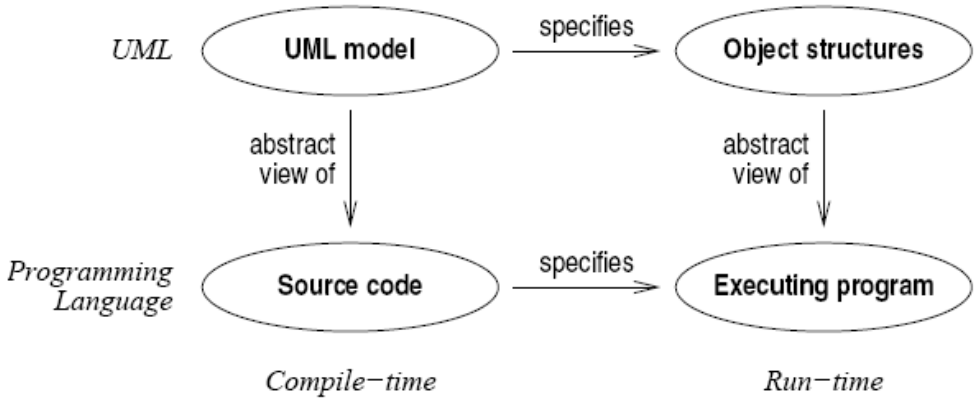


图 1.4 模型和代码间的关系

设计文档和源代码都可以称作编译时的制品。用如 Java 或 C++这种语言编写的源程序是定义我们想让程序展现的行为的文档，必须被编译和执行后才能有效果。UML 设计图是说明系统的一般结构和行为的文档，虽然不可能将它们直接翻译为可执行的代码。

程序一旦被编译和执行，就具有多种运行时的特性，如图 1.4 的右下部所示。在最低层，这些可以说是程序在运行时该程序对计算机的处理器和内存作用的结果。当编写程序时，程

序员通常试图把代码的行为和结果形象化，依据机器操纵的原始地址和二进制数据来考虑是很罕见的。代之的是他们常常会使用一个抽象模型来考虑程序运行时会发生什么。

特别是，使用面向对象语言的程序员常常使用程序执行的抽象模型，从程序运行时对象的创建和销毁，而且对象知道其他对象并能够和它们通信的观点，看待程序执行时发生的事情。这些抽象模型在图 1.4 中称为*对象结构*。图中右边的箭头表明，对象结构是程序运行时真正发生的事情的抽象，正像设计图是源程序中包含的信息的抽象一样。

所有这一切的重要意义是，我们用来理解程序的抽象模型——对象结构，也能够用来解释 UML 设计模型的含义，如图 1.4 中上端的箭头所示。这意味着面向对象模型和程序本质上描述的是同样的东西，而且解释了对象模型如何能够被翻译为代码，同时保留设计中指定的特性。它还意味着 UML 在很大程度上能够按照在编程中熟悉的概念来理解。

从上面的讨论可以得到两个结论。第一，使用像 UML 这样的语言定义的图不只是形象化，而是在规定系统的运行时期的特性方面具有确切的含义。在这方面，图就像程序一样，只不过比较抽象。

第二，用于解释 UML 表示法含义的模型，非常类似于面向对象程序员用来帮助他们形象化自己所编写的程序行为的模型。因此，UML 和面向对象语言具有相同的语义基础，这意味着实现一个设计，或者反过来把一个已有程序文档化会相对容易，因为可以确信系统的两种表示是一致的。

第 2 章将详细阐明这些观点，描述对象模型和 UML 定义对象结构的方法，并说明如何使用图建立程序结构和特性的文档。

1.5 软件开发过程

UML 是一种建模语言，不是一个过程，使用 UML 的项目也必须决定使用什么样的开发过程。软件开发者对什么开发过程是适当的，远不如对建模表示法的看法一致。 甚至提出建议，每个项目都应该从定义一个合适的过程开始，要考虑所开发系统的性质、开发队伍的规模和经验，以及其他因素。

统一软件开发过程是由开发 UML 的同一批方法学家与 UML 同时开发的一个过程模型。统一过程吸收了管理软件项目的大量经验，同时试图保留过程中看来必须的灵活性。

第 3 章简要叙述构成统一过程基础的原则的发展，并一般性地描述了 UML 如何与基于统一过程的开发一起使用。

1.6 小结

- 几乎所有重要软件的开发都使用某种*方法学*，即使只是一个非常不正规的方法学。通常包括建立*模型*以帮助理解系统的结构和设计。

- 方法学定义了语言和过程。语言定义了使用的模型和表示这些模型的表示法。过程定义了如何以及何时产生各种不同的模型。
- 定义了结构化的和面向对象的两种方法学。UML 是一种表达面向对象设计模型的语言，不是一个完整的方法学。
- UML 从若干个视图来描述一个系统，这些视图从多个不同角度表示系统的特性并与不同的用途相关。
- 视图用模型表示，模型定义了若干模型元素、它们的特性和相互之间的关系。
- 模型中包含的信息用各种图以图形的形式来交流。
- 设计模型和源代码共享一个公共的语义基础——对象模型，这保证了可能在系统的设计和代码之间维持密切的关系。

1.7 习题

（1）对用餐方法的一种描述可能包括以下步骤：设计菜单，购买材料，做饭，洗餐具。为这个过程中的每个步骤定义一个合适的“交付物”。每个阶段产生的交付物在某种意义上是下一步的输入吗？如果不是，那么应该是什么？画一个类似于图 1.1 和 1.2 的图说明这个过程。

（2）用类似于图 1.1 和 1.2 的图详细说明你所熟悉的软件开发方法。

（3）你是否同意将设计语言的学习和使用该语言的过程的学习分开是理想的，甚至是可能的？对程序设计语言的意见是否相同？

（4）针对下列任务考虑分别用图形和用文本进行使用说明和文档化的优缺点：

(*a*) 编写录像机程序；

(*b*) 更换汽车轮胎；

(*c*) 到一个你从未拜访过的朋友家；

(*d*) 烹调一道复杂的新菜；

(*e*) 描述你工作的组织的结构；

你是否能够指出你认为最适合图形支持的任务的共同特征？软件设计的活动是否具有这些特征？

第 2 章 对象建模

对于软件是什么以及程序如何工作，面向对象编程语言和设计语言有一个共同的理解。**对象模型**是 UML 和面向对象编程语言共享的公共计算模型。尽管编程语言和设计语言是在不同的抽象级别来表示程序的，但是我们理解这两种语言的基础都是对象模型所提供的对运行程序的抽象描述。

本章在一个简单应用的背景下，引出并描述对象模型的本质特征。通过这个例子介绍 UML 提供的这些概念的表示法，说明如何实现这些概念，解释设计语言和编程语言之间的密切联系。

2.1 对象模型

对象模型不是某个具体的 UML 模型，而是一种考虑程序结构的一般方式。它由构成面向对象设计和编程活动的基础的概念框架组成。对象模型的基本性质是：*计算是发生在对象之内和对象之间的。*

每个对象负责维护系统数据的一部分，并负责实现系统整体功能的某些方面。当程序运行时，对象一般由内存区域表示，这块内存区域包含着该对象存储的数据和其他信息。对象还支持方法或函数，以访问和更新对象包含的数据。因此，对象结合了计算机程序的两个根本方面——数据和处理，在其他软件设计方法中这二者是分离的。

然而，程序不只是一组孤立的对象的集合。必须要记录各个对象中存储的数据之间的关系，而程序的整体行为只有从多个不同对象的交互中才能显现出来。允许将对象连接到一起可以支持这些需求。典型地，这是通过使一个对象能够拥有对另一个对象的引用，或者更具体地讲，是知道其他对象的位置来实现的。

因而，对象模型将一个运行的程序视作是一个对象网络或图（graph）。对象构成该图中的结点，连接对象的弧称为**链接（*link*）**。每个对象包含程序数据的一个小子集，对象网络的结构则表示这些数据之间的关系。对象可以在运行时创建和销毁，对象之间的链接也可以改变。因此，对象网络的结构（或拓扑结构）是高度动态的，会随着程序的运行而改变。

对象之间的链接还可以作为对象交互的通信路径，使得对象能够通过互相发送**消息（*messages*）**进行交互。消息与函数调用类似：消息一般是请求接收对象执行其方法之一，而且可以附有用参数表示的消息的数据。通常，对象对一个消息的响应可能是向其他对象发送消息，这样，计算就通过网络而展开，网络中包含响应初始消息所涉及到的多个对象。

描述一个运行程序的对象的图结构并跟踪各个消息的结果是有可能的：适合做这件事的工具是调试程序。但是，通过定义各个对象来编写程序通常是不可行的，而是要给出相似的对象的**类（*class*）**的结构描述来定义对象能够持有的数据和方法的执行效果。因此，面向对象程序的源代码不是直接描述对象的图，而是描述组成这个图的对象的特性。

2.1.1 对象模型在设计中的作用

在设计中，对象模型的重要性在于它为 UML 的设计表示法提供了语义基础。UML 中许多特征的含义可以通过将它们解释为对相互连接的、互通消息的对象的集合的说明来理解。

可以绘制 UML 图(diagrams)来表示对象特定运行时的配置。然而，更常见的是绘制和源代码作用相同的图，从一般结构上来定义运行时会发生什么。这些图分成两大类。静态图描述对象之间可能存在的关系的种类，以及结果对象网络可以具有的可能的拓扑结构。动态图描述可以在对象之间传递的消息以及该消息对接收消息的对象的影响。

对象模型的双重作用使得将 UML 设计表示法与实际的程序相关起来非常容易，这也解释了为什么 UML 是适合设计和文档化面向对象程序的语言。本章剩下的部分将通过用一些基本的 UML 表示法文档化一个简单程序的例子对此进行说明。

2.1.2 一个库存控制的例子

在制造业中，某些类型的复杂产品是由零件装配而成的，常见的需求是记录所拥有的零件的库存以及这些零件的使用方式。本章我们将开发一个简单的程序来模拟不同种类的零件和它们的特性，以及用这些零件构造复杂组件的方式，通过这个例子来阐明对象模型。

这个程序必须管理描述系统中不同零件的信息。除了维护所使用的零件的不同类型信息，我们还设想对系统来说记录各个实际零件的信息也很重要，比如用作质量保证和跟踪。

对这个例子来说，假定对每个零件我们感兴趣的是下列三项信息：

（1） 零件的目录查找号（整数）；

（2） 零件的名字（字符串）；

（3） 单个零件的成本（浮点数值）。

零件可以被装配成更复杂的结构，称为组件。一个组件可以包含多个零件，而且可以具有层次结构，也就是说，一个组件可以由许多子组件构成，每个子组件又由零件或它自己的更深一层的子组件构成。

维护零件、组件及它们的结构信息的程序应该能用于多种用途，例如维护目录和库存信息，记录制造的组件的结构，支持对组件的各种操作，例如计算组件中零件的总成本，或者打印组件的所有零件的清单。本章我们将考虑一个简单的查询应用：通过累加组件中包含的所有零件的成本，查出一个组件中的材料成本。

2.2 类和对象

面向对象系统中的数据和功能分布于系统运行时存在的对象之中。每个单独的对象维护部分系统数据并提供一组允许系统中的其他对象对这些数据进行某些操作的方法。面向对象设计的难题之一就是如何将系统的数据划分到一组对象中，而这些对象将成功地交互以支持所要求的总体功能。

识别对象经常应用的一个经验准则是：用模型中的对象表示来自应用领域的现实世界中的对象。库存控制系统的主要任务之一是记住厂商库存中的所有物理零件。因此，很自然的起点就是考虑将这些零件中的每一个都表示为系统中的一个对象。

一般会有许多零件对象，每个对象描述一个不同的零件，因而保存了不同的数据，但是每个对象都具有相同的结构。表示同一种实体的一组对象的共有结构由**类**描述，该类型的每个对象被称为是该类的一个**实例**（*instance*）。那么，作为库存管理系统设计的第一步，我们可以考虑定义一个“零件（part）”类。

一旦确定了候选类，我们可以考虑该类的实例中应该放些什么数据。在 Part 类的情况中，一个自然的想法是每个对象应该保存系统必须保存的关于该零件的信息：它的名字、编号以及成本。这反映在下面的代码中。

```
//---------------------------------------------------------------------------
public class Part
{
  private String name ;
  private long number ;
  private double cost ;
  public Part(String nm, long num, double cst) {
    name = nm ;
    number = num ;
    cost = cst ;
  }
  public String getName() { return name ; }
  public long getNumber() { return number ; }
  public double getCost() { return cost ; }
}
//---------------------------------------------------------------------------
```

UML 类的概念与诸如 C++和 Java 这样的程序设计语言中的概念非常类似。一个 UML 类定义了许多特征（*feature*）：细分为属性（*attribute*）和操作（*operation*），属性定义类实例存储的数据，操作定义类实例的行为。一般地说，属性相当于 Java 类中的域，操作则相当于方法。

在 UML 中，类由一个分为三栏的矩形图标表示，分别包含该类的名字、属性和操作。‘Part’类的 UML 表示如图 2.1 所示。

类图标最上面的一栏包含类的名字，第二栏包含类的属性，第三栏是类的操作。在操作的特征标记中可以使用程序语言中的类型，用冒号把属性名、参数名或操作名与类型隔开。UML 也表示了类的各种特征的访问级别，用减号表示‘private（私有）’，加号表示‘public（公有）’。构造函数下面有下划线，以便和类的一般的实例方法相区分。

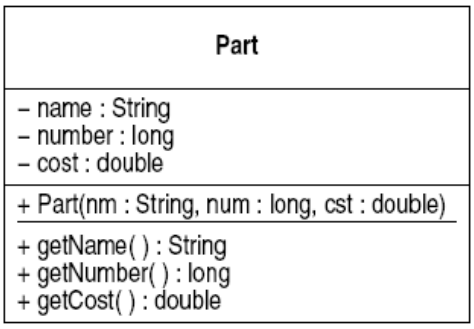


图 2.1 “零件”类的 UML 表示

第 8 章将给出 UML 语法的详尽细节，在这里值得注意的是，图 2.1 所示的许多细节都是可选的。其中包含类名字的一栏是必需的：在特定的图中，如果没有要求，那么可以省略其他信息。

2.2.1 对象创建

类是在编译时定义的，而对象是在运行时作为类的实例创建的。执行下列语句的结果是创建一个新对象。它包括两个步骤：首先为对象分配一块内存区域，然后适当地初始化。一旦创建了新对象，将在变量 `myScrew` 中保存它的一个引用。

```
Part myScrew = new Part("screw", 28834, 0.02) ;
```

UML 定义了描述单个对象及其所保存的数据的图形化表示法。上面一行代码创建的对象可以用 UML 描述，如图 2.2 所示。

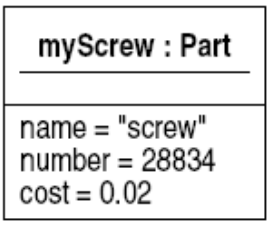


图 2.2 一个 Part 对象

对象由分为两栏的矩形表示。上面一栏包含对象的名字及其类的名字，都加有下划线。对象不一定要命名，但如果有和对象相关的变量，用变量的名字为对象命名有时会有用。当要知道对象的类时，对象的类名总要说明。通常的风格习惯是类的名字以大写字母开头，而对象的名字以小写字母开头。

数据作为属性的值保存在对象中。对象图标中下面的一栏包含有对象属性的名字和当前值。这一栏是可选的，如果在图中不必要显示对象的值时可以省略。

2.3 对象的特性

对象通常的特性描述表明对象是具有状态、行为和本体的某个事物。下面将更详细地解释这个概念以及相关的封装的概念，另外还讲述了实现这些特征的类定义的各种术语。

2.3.1 状态

对象的第一个重要特征是它们充当数据的容器。在图 2.2 中，对象的这个特性通过在表示对象的图标中包含数据来描绘。在纯面向对象系统中，系统维护的所有数据都保存在对象中：不存在其他模型中的全局数据或中央数据存储库的概念。

包含在对象属性中的数据值通常称为对象的**状态**（*state*）。例如，图 2.2 中所示的三个属性值构成了对象“`myScrew`”的状态。由于这些数据值会随着系统的变化而改变，结果当然是对象的状态也可以改变。在面向对象的程序设计语言中，对象的状态由对象的类中所定义的域指定，而在 UML 中由类的属性指定，例如图 2.2 所示的三个属性值。

2.3.2 行为

每个对象除了保存数据之外还提供了一个由若干操作组成的接口。通常其中的一些操作将提供访问和更新对象中所保存的数据的功能，但其他操作将更通用，并实现系统全局功能的某些方面。

在编程语言中，对象的操作在它的类中定义为一组方法。对象定义的一组方法定义了该对象的**接口**（*interface*）。例如，2.2 节中定义的 Part 类的接口包括一个构造函数和一些访问函数，以返回对象的域中所保存的数据。

在 UML 中，操作不同于属性，操作没有出现在对象图标中。这是因为对象提供的完全是它的类所定义的操作。由于一个类可以有许多实例，每个都提供同样的操作，因此显示每个对象的操作会很多余。在这方面，对象的行为不同于它的属性，因为，通常同一个类的不同实例将保存不同的数据，因而具有不同的状态。

2.3.3 本体

对象定义的第三个方面是每个对象和其他所有对象都是可区别的，即使两个对象保存完全相同的数据，并在接口中提供完全相同的操作集合时也是如此。例如，下面几行代码创建两个状态相同的对象，但它们还是不同的对象。

```
Part screw1 = new Part("screw", 28834, 0.02) ;
Part screw2 = new Part("screw", 28834, 0.02) ;
```

对象模型假定为每个对象提供了一个唯一的**本体**（*identity*），作为区别于其他对象的标志。对象的本体是对象模型固有的一部分，不同于对象中存储的任何其他数据项。

设计人员不需要定义一个特殊的数据来区分一个类的各个实例。但是，有时应用领域会包含对每个对象都不相同的真实的数据项，例如各种识别号码，这些数据项通常作为属性建

模。然而，在没有这样的数据项的情况下，也没有必要只是为了区分对象而引入一个这样的数据项。

在面向对象的程序设计语言中，对象的本体一般由它在内存中的地址表示。由于不可能在同一个位置保存两个对象，所有对象都保证具有唯一的地址，因而任意两个对象的本体都是不同的。

2.3.4 对象名字

UML 允许为对象命名，对象名字不同于其所属类的名字。这些名字是模型内部的，允许在一个模型中的其他地方引用这个对象。这些名字不对应对象中存储的任何数据项，不过，可以将名字看作是为对象的本体提供了一个方便的别名。

对象的名字不同于刚好保存该对象的引用的变量名。在举例说明对象时，如在图2.2 中，使用保存对象引用的变量名字作为对象的名字通常比较方便。但是，可以有多个变量保存对同一个对象的引用，并且一个变量在不同的时候可以引用不同的对象，所以，这种惯例如果随意应用，可能很容易引起混淆。

2.3.5 封装

对象一般理解为**封装**（*encapsulate*）了它们的数据。这意味着对象内保存的数据只能够通过属于该对象的操作来操纵，因而一个对象的操作不能直接访问在不同的对象中存储的数据。

在许多面向对象语言中，通过语言的访问控制机制来提供一种封装形式。例如，在 2.2 节中的 Part 类的数据成员被声明为“private”，意思是它们只能被同类对象的操作访问。注意，这种基于类的封装形式比基于对象形式的封装要弱，后者不允许对象访问任何其他对象的数据，即使是属于同一个类的对象的数据。

2.4 避免数据重复

尽管 2.2 节中采用的对零件建模的方法简单直接，很有吸引力，但是在真正的系统中不可能令人满意。它的主要缺点是描述给定类型零件的数据是重复的：数据保存在零件对象中，而且如果有两个或多个同类型的零件，数据会在每个相关对象中重复。这样，至少存在着三个重大问题。

首先，它含有高度冗余。系统可能记录一个特定类型的数千个零件，它们共有相同的查找号、描述和成本。如果在每个零件中都保存这些数据，将不必要地耗尽大量的存储。

其次，数据的重复可能会导致维护问题，尤其是成本。如果一个零件的成本变了，每个受到影响的对象中的成本属性都需要更新。这样不仅低效，而且也难以保证在这种情况下每一个相关的对象都会被更新，并且不会错误地修改了表示不同种类零件的对象。

第三，需要永久保存零件的目录信息。但是，在某些情况下，一个特定类型的零件对象可能不存在，例如，在零件还没有制造出来的情况下。倘若这样，就没有地方保存目录信息。然而，不可能容许只有在相关零件存在时才能保存目录信息。

设计这个应用更好的一种方法应该是将描述给定类型零件的共享信息保存在另外的对象中。这些“描述符”对象并不表示单独的零件，而是表示与描述一类零件的目录条目（*catalogue entry*）相关的信息。图 2.3 非正式地举例说明了这种情况。

这个新设计要求，对于系统所知的每种不同类型的零件，都应该存在单独一个目录条目对象，用来保存该类型零件的名字（name）、查找号（number）和成本（cost）。Part 对象不再保存任何数据。要查找一个零件，必需访问描述该零件的目录条目对象。

这种方法解决了上列问题。数据只存储在一处，因而没有冗余。修改给定类型零件的数据很直接：如果一种零件的成本改变了，只需要更新一个属性，即相应的目录条目对象中的成本属性。最后，一个目录条目对象，即使在没有 Part 对象与之相关联时，也可以存在，因而解决了在创建任何零件之前如何能够保存零件信息的问题。

2.5 链接

现在，库存控制程序的设计包括了两个不同的类的对象。目录条目对象保存适用于给定类型的所有零件的信息，而每个零件对象则表示一个单个的实际零件。

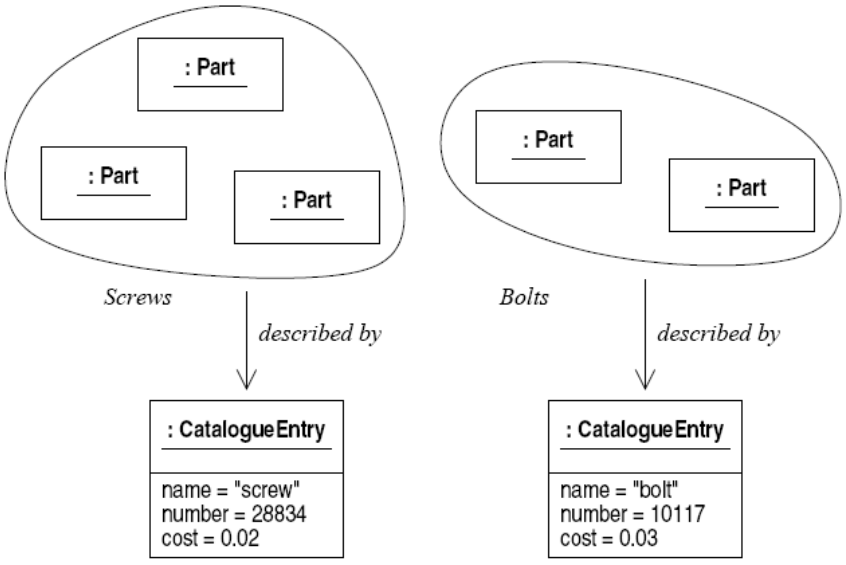


图 2.3　目录条目描述的零件

然而，这些类之间存在一个重要的关系：为了获得对一个零件的完整描述，必需查看的不仅是零件对象，还要查看相关的描述零件的目录条目对象。

实际上，这意味着对每个零件对象，系统必须记录哪个目录条目对象在描述它。实现这种关系一般方法是每个对象包含一个到相关目录条目的引用，如下列代码所示。目录条目类

现在包含着描述零件的数据，零件是用一个目录条目进行初始化并且保存对该目录条目的一个引用。

```
//-------------------------------------------------------------------------
public class CatalogueEntry
{
  private String name ;
  private long number ;
  private double cost ;
  public CatalogueEntry(String nm, long num, double cst) {
    name = nm ;
    number = num ;
    cost = cst ;
  }
  public String getName() { return name ; }
  public long getNumber() { return number ; }
  public double getCost() { return cost ; }
}
public class Part
{
  private CatalogueEntry entry ;
  public Part(CatalogueEntry e) {
    entry = e ;
  }
}
//-------------------------------------------------------------------------
```

在创建一个零件时，必须提供一个适当的目录条目对象的引用。这样做的根本原因是：创建一个未知的或未指定类型的零件是没有意义的。下面的代码显示了如何用这些类创建零件对象。首先，必须创建一个目录条目对象，而后可以用它初始化所需的任意多个零件对象。

```
//-------------------------------------------------------------------------
CatalogueEntry screw = new CatalogueEntry("screw", 28834, 0.02) ;
Part s1 = new Part(screw) ;
Part s2 = new Part(screw) ;
//-------------------------------------------------------------------------
```

如同在 2.2 节中解释的那样，一个类的域在 UML 中通常是作为类的属性来建模。然而，如果一个域包含着对另一个对象的引用，例如上面的 Part 类中的 entry 域，这种方法就不合适。属性定义的是保存在对象内部的数据，但目录条目对象并不是保存在零件内部，而是独立完整的对象，能够独立于任何零件对象存在，并能够同时被多个零件对象引用。

在 UML 中，一个对象保存另一个对象的引用的事实通过在这两个对象之间画一个**链接**（*link*）来表示。链接表示为一个箭头，从保存引用的对象指向被引用的对象，而且在链接的箭头上可以标示保存引用的域的名字。因此，上面的代码所创建的对象可以用 UML 建模，如图 2.4 所示。

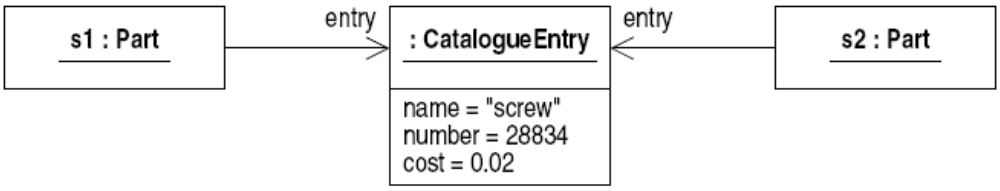


图 2.4　对象之间的链接

链接上的箭头表示只能在一个方向上遍历，或导航。这意味着 Part 对象知道它所链接的目录条目对象的位置，并有权访问目录条目对象的公有接口。这并不意味着目录条目对象对引用它的 Part 对象具有任何访问权限，它甚至不知道该 Part 对象。在另一个方向的访问只能通过在目录条目中保存对零件的引用来提供，从而使链接在两个方向都是可导航的。

2.5.1 对象图

对象图（*object diagram*）是展现对象和对象之间的链接的图形表示。图 2.4 是很简单的一个对象图的例子。对象图是以可视形式表示 2.1 节所讨论的对象的图结构的一种方式：它们给出了系统中的数据在给定时刻的一个“快照”。

在相互链接的对象的结构中，信息是用两种不同的方法记录的。一些数据作为属性保存在对象中，而另一些信息纯粹是在结构上依靠链接保存的。例如，零件属于给定类型的事实是通过零件对象和相应的目录条目之间的链接表示的：在系统中没有显式记录零件的类型的数据项。

2.6 关联

如同用类定义一组相似对象的共同结构一样，这些对象之间的链接的共同特性也可以通过相应的类之间的关系定义。在 UML 中，类之间的数据关系称为**关联**（*association*）。因而，链接是关联的实例，就如同对象是类的实例一样。

在库存控制的例子中，图 2.4 中的链接必须是 Part 类和目录条目类之间的一个关联的实例。在 UML 中用一条连接相关的类的线来表示关联；与上面的链接相对应的关联如图 2.5 所示。注意，为了清晰起见，图中没有显示目录条目类的所有已知信息。

这个关联用 UML 建立了一个模型，它定义了 Part 类中的 entry 域。因为 entry 域保存了一个引用，所以关联表示为只在一个方向上可导航。类似于相应的链接，在关联的一端标记着域的名字。以这种方式放置在关联端点的标记称为**角色名**（*role name*）：它的位置反映了“entry”这个名字在 Part 类中用来引用所链接的目录条目对象。

在关联的端点还标明了一个**重数约束**（*multiplicity constraint*），这个例子中是数字“1”。重数约束表明一个给定的对象在任一时间能够和多少个实例链接。在这个例子中，约束是每个零件对象必须链接到一个且只有一个目录条目对象。不过，图中没有规定多少个零件对象可以和一个目录条目对象链接。

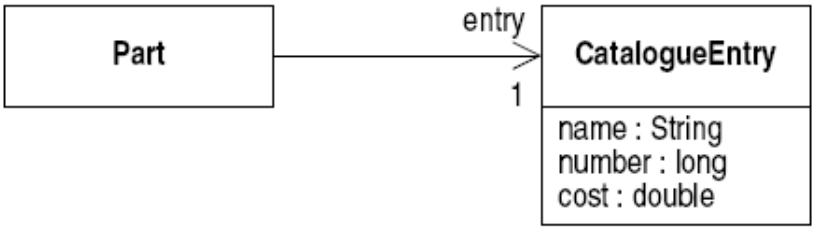


图 2.5　两个类之间的关联

重数约束给出了关于模型的很有价值的信息，但是，它没有表达出来的东西同样重要。例如，对库存控制系统的一个合理的特性需求是：一个零件应该总是链接到同一个目录条目对象，因为建模的实际零件不能从一种类型变成另一类型。然而，这里显示的重数约束并没有强制这点：在任何给定时间应该只有一个链接的目录条目这个约束，并未含有在任何时候都是相同一个目录条目的意思。

值得注意的是，上面给出的 Java 的 Part 类实际上没有实现图 2.5 所示的重数。因为 null 在 Java 中是引用域的合法值，所以，如果将 null 作为 Part 类的构造函数的参数，那么有未链接到任何目录条目的 Part 对象就是可能的。这与一个零件应该总是链接到恰好一个目录条目对象的重数约束相矛盾。Part 类的一种更健壮的实现可以是在运行时对此进行检查，在试图用一个 null 引用初始化时抛出一个异常。

2.6.1 类图

对象图显示对象和链接的集合，类图（*class diagrams*）则包含了类和关联。图 2.5 是一个简单的类图的例子。

对象图显示系统的对象的图结构的许多特定状态，类图则以一种更一般的方式指定了系统的任何合法状态都必须满足的特性。例如，如果图 2.5 表示在任何给定时间，库存控制系统的对象图能够包含“Part”和“CatalogueEntry”类的实例，并且每个 Part 对象必须链接到恰好一个目录条目，那么，假如程序进入了一种状态，譬如说存在连接两个目录条目的链接，或者零件没有链接到目录条目，就会出现一个错误，而程序则处于一种非法状态。按照术语固有的外延，如果对象图满足类图中定义的各种约束，那么称对象图是类图的实例。

2.7 消息传递

上面的例子已经阐明了面向对象程序中的数据是如何分布在系统中的对象之中的。一些数据作为属性值显式地保存，对象之间的链接也含有信息，它描述了对象之间持有的关系。

信息的分布意味着，一般而言，为了完成任何有意义的功能，需要多个对象进行交互。例如，假设我们想要为 Part 类增加一个方法，支持检索单独一个零件的成本。但是，表示零件成本的数据值并没有保存在零件对象中，而是保存在零件所引用的目录条目对象中。这意味着为了检索该数据，这个新方法必须调用目录条目类中的 getCost()方法，如下面的实现所示。

```
public class Part
{
  public double cost() {
    return entry.getCost() ;
  }
  private CatalogueEntry entry ;
}
```

现在，如果客户持有一个 Part 的引用并要查询它的成本，可以如下调用 cost 方法。

```
Part s1 = new Part(screw) ;
double s1cost = s1.cost() ;
```

UML 将方法调用表示为从一个对象发送到另一对象的**消息**。当一个对象调用另一对象的方法时，可以看作是请求被调用的对象执行某些处理，这个请求作为一个消息建模。图 2.6 显示了对应于上面的代码中调用 `s1.cost()` 的消息。

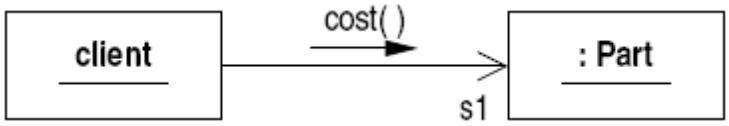


图 2.6 发送一个消息

图 2.6 中的 client 对象只有对象名而没有类名。“cost”可以由多个不同类的对象发送给一个零件，而消息发送者的类与理解消息以及零件对象对消息的响应无关。因此，在这种说明特定交互的对象图中省略客户类比较方便。

客户代码持有到 Part 对象的一个引用，保存在变量 s1 中，如前所述，这在图 2.6 中表示为一个链接。这个引用还使得客户能够调用链接的对象的方法，然而，这意味着在 UML 中，对象之间的链接也表示了消息的通信信道。在对象图中，消息用链接旁边带标记的箭头表示。在图 2.6 中，显示了一个客户对象向 Part 对象发送消息请求得到零件的成本。消息本身则用常见的“函数调用”符号书写。

对象在接收到一个消息时，通常会以某种方式响应。在图 2.6 中，预期的响应是 Part 对象向客户对象返回自己的成本。但是，为了查找成本，Part 对象必须调用它所链接的目录条目对象中的 `getCost` 方法。这可以用第二个消息表示，如图 2.7 所示。

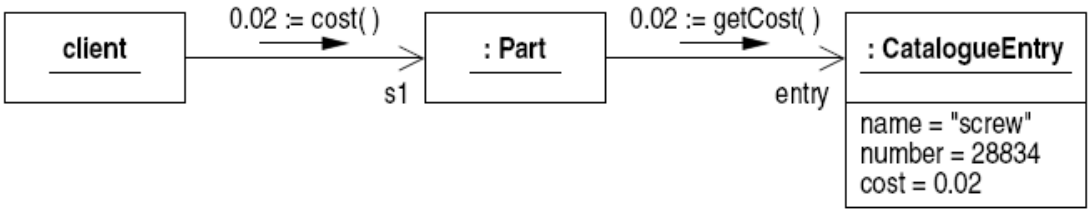


图 2.7 查找零件的成本

图 2.7 的例子还说明了消息返回值的 UML 表示法。返回值写在消息名字的前面，并用赋值符号“:=”分隔开。在不显示返回值时，或者没有返回值时，如图 2.6 所示，可以地省略这个符号。

以上所示消息的语义是普通程序函数调用的语义。当一个对象给另一个对象发送消息时，程序中的控制流从发送者传递给接收消息的对象，发送消息的对象一直等到控制返回时才继续自己的处理。

2.8 多态性

除了维护单个零件的详细资料之外，库存控制程序还必须能够记录这些零件如何装配成组件。图 2.8 所示的是一个包含一个 strut（支杆）和两个 screw（螺旋）的简单组件。注意，在这个图中省略了目录条目类中的无关属性。

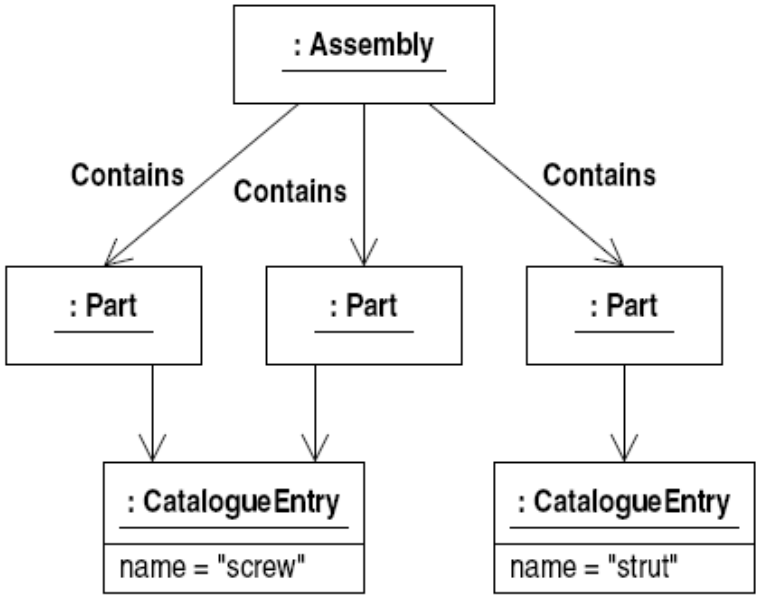


图 2.8 一个简单组件

在图 2.8 中，组件（Assembly）中包含哪些零件（Part）的信息是通过连接组件对象和零件对象的链接表示的。这些链接不是用角色名标示，而是用**关联名**标示。关联名描述链接的对象之间具有的关系。关联名通常是经过选择的，像这里一样，使得可以从关联名以及所

链接的类名构造出描述这种关系的语句。在这个例子中，合适的语句可以是“一个组件包含（contains）零件”。

组件类的实现必须提供一种方法，能保存对不定个数零件的引用。一种简单的支持方法是在类中包含一个数据结构，该数据结构能够保存该组件对所有零件的引用，如下所示。

```
public class Assembly
{
  private ArrayList parts = new ArrayList() ;
  public void add(Part p) {
    parts.add(p) ;
  }
}
```

图 2.8 中的链接实例所属的关联在图 2.9 中表示了出来。如链接那样，关联上标明了关联名，写在关联的中间。关联端点的“*”符号是一个重数标记，意思是“0 个或多个”。在这个图中，它指定一个组件可以链接或包含零或多个零件。

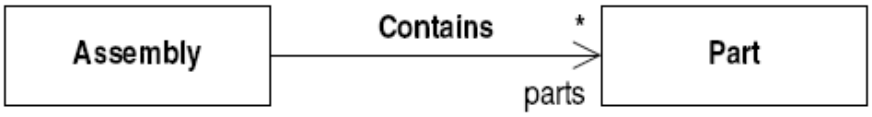


图 2.9 组件和零件之间的关联

图 2.9 中的图在关联端点标示的角色名‘parts’，对应于 Assembly 类中用于保存引用的域，从而文档化了上面的代码中关联的实现。通常，为了让图的含意更清晰，可以使用所需要的任何名字和角色名的结合来标明关联和链接。

然而，将一个组件简单地作为一组零件建模是不够的。组件可以有层次结构，零件能够装配为子组件，子组件可以和其他子组件与零件装配在一起，形成更高层次的组件，可以达到任何需要的复杂度。图 2.10 显示了一个简单的例子，它在图 2.8 所示的结构中引入了一个子组件。

为了实现层次结构，组件必须能包含零件和其他组件。这意味着和图 2.8 不同，2.8 中标示有‘Contains’的链接全都是把一个组件对象连接到一个零件对象，而在图 2.10 中，标示着‘Contains’的链接还可以把一个组件对象连接到另一个组件。

如同许多程序设计语言一样，UML 也是强类型的语言。链接是关联的实例，因而由链接连接的对象必须是相应的关联端点的类的实例。在图 2.8 中，这个要求是满足的：如图 2.9 规定的那样，每个标示有‘Contains’的链接连接该 Assembly 类的一个实例到该 Part 类的一个实例。

但是，图 2.10 中违反了这个条件，因为‘Contains’链接将顶层 Assembly 实例不是连接到一个 Part 对象，而是连接到了一个 Assembly 类的对象。如果我们想要建立层次组件的模型，在这些链接“Contains”被包含的一端，必须不能像图 2.9 所指定的那样约束为只是

‘Part’类，而是‘Part’类或‘Assembly’类。这是一个**多态性**（*polymorphism*）的例子：多态性的意思是“许多形态”，暗示在某些情况下，通过同一类型的链接需要连接多个类的对象。

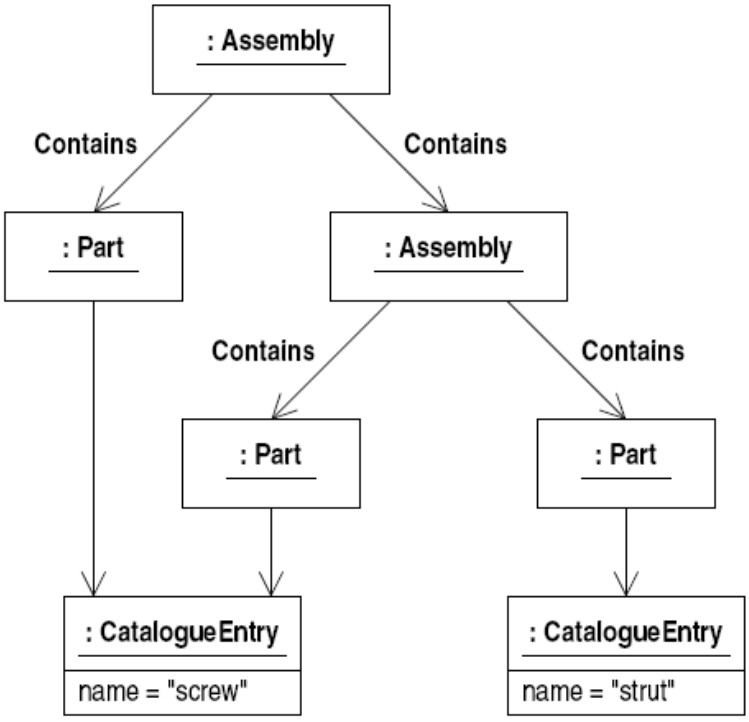


图 2.10 层次组件

2.8.1 多态性的实现

UML 是强类型的，所以不允许链接连接任意类的对象。对于如图 2.10 所示的多态链接，必需指定能够参与链接的类的范围，通常的实现方法是定义一个更一般的类，并说明我们希望链接的特殊类是这个一般类的特化。

在库存控制程序中，我们想要建立的模型是：组件能由构件（*component*）组成，而每个构件可以是一个子组件或是一个零件。这样就能够规定‘Contains’链接把组件对象连接到构件对象，而按照定义，构件对象要么是 Part 对象，要么是表示子组件的其他 Assembly 对象。

在面向对象语言中，是使用继承（*inheritance*）机制来实现多态性。可以定义一个构件类，而将 Part 类和 Assembly 类定义为构件类的子类，如下所示。

```
public abstract class Component { ... }
public class Part extends Component { ... }
public class Assembly extends Component
{
  private ArrayList components = new ArrayList() ;
  public void add(Component c) {
    components.add (c) ;
```

```
    }
}
```

Assembly 类之前的实现中定义了将一个零件加入到组件中的方法，这里相应的方法是将一个构件加入到组件中。然而，在运行时，实际创建并加入到组件的对象将是 Part 类和 Assembly 类的实例，而不是构件类自身的实例。Java 中继承语义的含义是：可以在任何指定超类引用的地方使用子类的引用。在这里，这意味着零件和组件的引用都可以作为 add 函数的参数来传递，因为这些类都是构件类的子类，而函数的参数指定为构件类。

Assembly 类的实现可能甚至比这更具多态性，因为其中使用了 Java API 的 java.util.ArrayList 类来存储对构件的引用，而 ArrayList 类可以保存任何类型对象的引用。对构件类的限制是由“add”方法的参数类型强加的，它提供了客户向组件中加入构件的唯一方法。

2.8.2 UML 中的多态性

这个例子中，多态性的实现由两种不同机制相互作用而产生。第一，定义 Assembly 类使得它能够保存对多个构件对象的引用，第二，用继承定义子类，表示现有的不同类型的构件。于是程序设计语言规则就导致一个组件能够保存对不同类型构件的混合引用。

Java 的“extends”关系在 UML 中用类之间的**特化**（*specialization*）关系表示：如果类 E 是通过扩展类 C 而定义的，那么就说 E 是 C 的特化。如果从超类比子类有更大的范围的关系的角度看，这种关系也被称为**泛化**（*generalization*）：等价的描述可以说 C 是 E 的泛化。泛化，或者特化，在 UML 类图中用一个将关系中的子类连接到超类的箭头描绘。这些关系与关联在直观上的区别是箭头的形状。图 2.11 显示了库存控制例子中的类之间的特化关系。

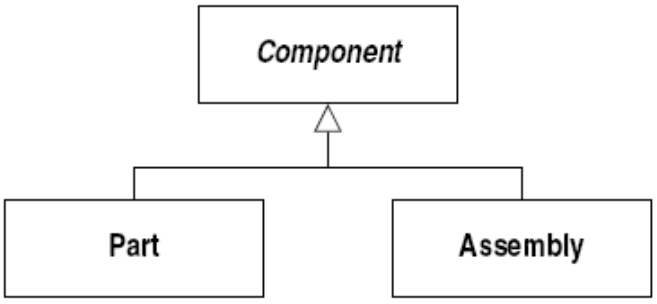


图 2.11 构件之间的泛化关系

和关联不同，泛化没有在对象图中出现的“实例”。关联描述的是对象能够链接到一起的方式，泛化描述的则是一个类的对象能被另一类的对象替换的情形。正因为这样，重数的概念不适用于泛化，并且一般也不标注泛化关系。

最后，考虑到图 2.11 中的泛化，我们可以重新定义图 2.9 中的关联。结果如图 2.12 所示，该图也文档化了上面给出的 Component、Part 和 Assembly 类的实现。

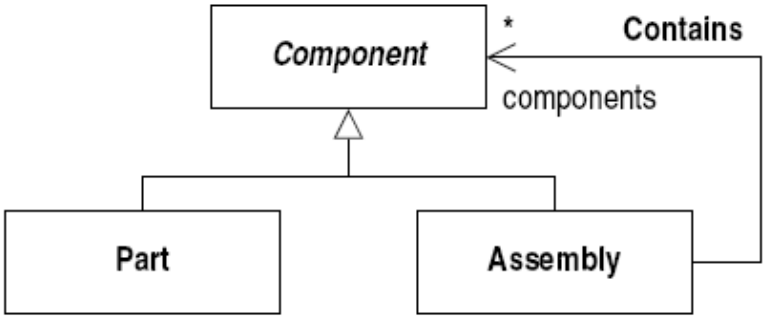


图 2.12　允许层次组件的模型

图 2.12 表明组件能够包含零个或多个构件（关联），每个构件可以是一个 Part 或是一个组件（特化）。在后一种情况下，我们有了一个组件包含在另一个组件之中的情形，因此这个类图允许构造如图 2.10 所示的层次对象结构。

2.8.3 抽象类

与 Part 类和 Assembly 类不同，我们决不会期望创建构件类的实例。Part 和 Assembly 对应于应用领域中的真实对象或结构，但是，Component 类是一个概念的表示，即 Part 和 Assembly 可以认为是更一般的构件概念的特例。在模型中引入构件类的原因不是为了能够创建构件对象，而是为了指定 Part 和 Assembly 在某些情况下是可互换的。

像‘Component’这样，引入它主要是为了指定模型中其他类之间的关系，而不是为了支持新类型对象的创建，这样的类称为**抽象类**（*abstract classes*）。如上面例子说明的，Java 允许将类声明为抽象的，而在 UML 中，可以通过将类名字写成斜体表示，如图 2.11 和 2.12 所示。

2.9 动态绑定

如果传递给 Assembly 对象一个消息请求得到它的成本，那么 Assembly 对象响应这个请求的方法是向自己的构件请求得到它们的成本，然后返回所得到的成本的总和。自身也是组件的构件对象将向自己的构件发送类似的‘cost’消息；如果构件是简单零件，就向它所链接的目录条目对象发送 ‘getCost’消息，如图 2.7 所示。

如果组件对象处于图 2.10 中的层次的顶端时，向它发送一个‘cost’消息后将会产生的所有消息在图 2.13 中给出。注意：在面向对象程序中，单个请求非常容易引起系统中对象之间的一个复杂的交互网。

在这个交互中，组件对象通过向自己的所有构件发送同样的消息来计算出成本，即‘cost’。发送消息的对象不知道特定的构件是一个 Part 还是一个 Assembly，实际上也不需要知道。它只是简单地发送消息，并依赖接收对象以一种恰当的方式解释这个消息。

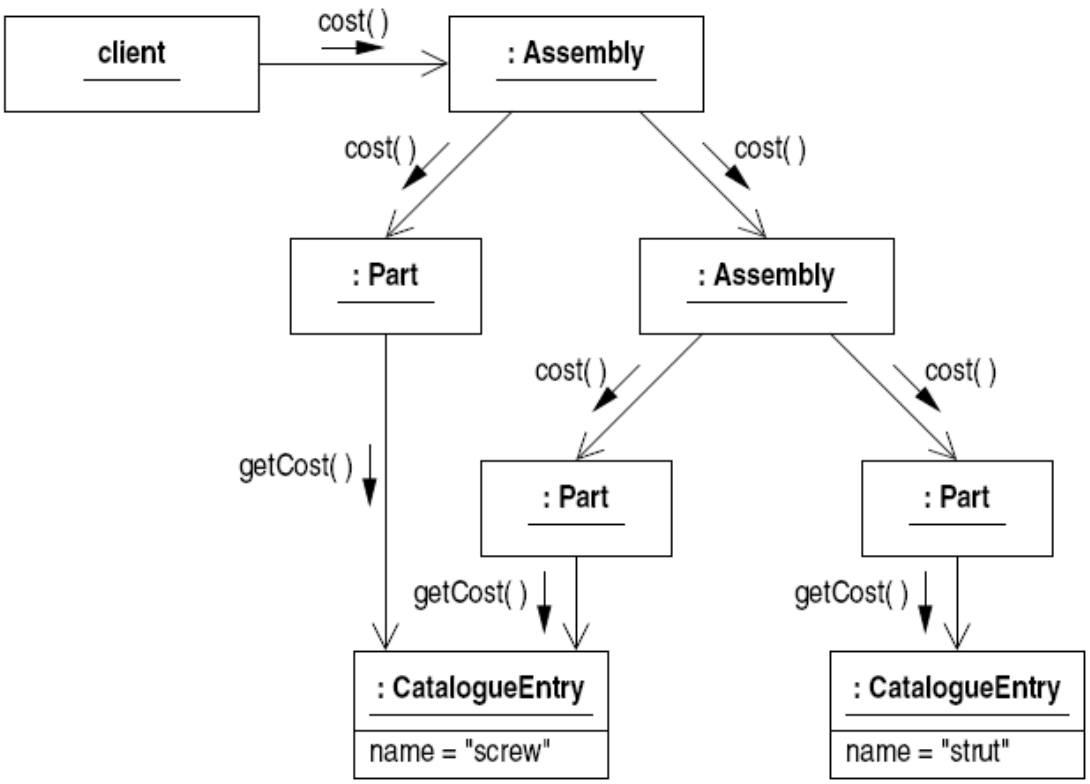


图 2.13 层次中的消息传递

当接收到一个‘cost’消息时，实际进行的处理将依赖于消息是发送给了 Assembly 对象还是 Part 对象。这种行为称为**动态绑定**（*dynamic binding*），或**晚绑定**（*late binding*）：实质上，是消息的接收者，而不是发送者，来决定执行什么代码响应所发送的消息。在多态的情况下，消息接收者的类型可能直到运行时才可以知道，因而，响应消息所执行的代码只能在运行时选择。

在 Java 中，获得这种行为很简单：在 Component 类中声明 cost 函数，然后在 Part 类和 Assembly 类中重定义 cost 函数，为各个类提供需要的功能，如下面所示的来自相关类的摘录。其他语言提供晚绑定有不同的方法：例如在 C++中，就必须使用虚函数机制。

```
//-------------------------------------------------------------------------
  public abstract class Component
  {
    public abstract double cost () ;
  }
  public class Part extends Component
  {
    private CatalogueEntry entry ;
    public double cost() {
      return entry.getCost() ;
    }
  }
```

```
  public class Assembly extends Component
  {
    private ArrayList components = new ArrayList() ;
    public double cost() {
      double total = 0.0 ;
      for(int i = 0; i < components.size(); i++)
        total += ((Component) components.get(i)).cost() ;
      }
      return total ;
    }
  }
//---------------------------------------------------------------------------
```

2.10 对象模型的适用性

本章讨论了对象模型的基本特征，并着重讨论了对象模型的概念与面向对象编程语言的概念之间的联系。对象模型在软件开发生命周期中从需求分析开始的所有阶段都要使用，因而研究对象模型对这些活动的适用性也很重要。

通常说，面向对象的观点是受到了我们平常看待世界的方式的启示。事实上我们的确是把世界作为是由对象组成来感知和认识的，这些对象具有各种特性、互相交互、以各种独特的方式行动，而对象模型则被看作是这种一般意义上观点的反映。因此，有时候会认为对象建模非常容易，软件系统中需要的对象可以简单地通过研究所建模的现实世界领域而发现。

在某些系统中，情况的确是如此，其中一些被建模的现实世界对象和一些软件对象之间存在着相当直接的对应，但是这不能类推而作为建立面向对象系统的一种非常有用的指导原则。设计软件的首要目标是产生满足用户需求的、易于维护的、易于修改和复用并且能够有效利用资源的系统。

一般而言，认为简单地复制所感知的现实世界中的对象就能产生具有这些良好特性的软件是不合理的。例如，将零件直接表示为 2.2 节的 Part 对象，2.4 节表明，这样会导致重大的效率问题和可维护性问题，甚至它能否满足系统的功能需求也存在疑问。由于过分强调现实世界对象的特性而导致的拙劣设计的另一个典型例子将在 14.5 节中给出。

此外，在对象模型中对象之间通信的消息传递机制，似乎也没有准确地表现现实世界中许多事件发生的方式。例如，考虑这样一种情形，两个人没有注意走到某个地方，偶然相遇。将这种情形想象为是其中一人向另一人发送了一个“相遇”消息，是与直观相违背的。两个人是平等的而且无意识地参与了一个事件的发生，这样似乎更恰当一些。由于这样的原因，一些面向对象的分析方法建议，根据对象和事件建立现实世界的模型，而只在设计过程的后期才引入相关的消息。

无疑地，存在一些案例，其中现实世界的对象，特别是代理，可以认为是给其他对象发送消息。然而，对象模型最有意义的优点不是它对现实世界建模的适宜性，而是在于具有面向对象结构的程序和软件系统更可能拥有许多人们想要的特性，例如易于理解和维护。

将面向对象作为一种特别的方法，它解决如何将软件系统中的数据和处理关联在一起，这样的方式更有助于理解面向对象。所有的软件系统都必须处理一组给定的数据，并提供操纵和处理这些数据的能力。在传统的过程式系统中，数据和处理数据的函数是分离的。系统的数据存储在一个地方（中央存储库），而应用需要的功能则通过一些操作提供，这些操作能自由访问任何部分的数据，同时在本质上保持与数据的分离。每个操作都有从中央存储库中选取自己感兴趣的一部分数据的责任。

对这种结构，可以看到：大多数操作将仅仅使用系统整个数据的一小部分，大多的数据块将只是由少数操作访问。面向对象方法试图做的是将数据存储库划分为许多独立的数据块，并将数据块和直接操纵该数据块的操作集成在一起。

与更传统的结构相比，这种方法能够提供许多显著的技术优势。然而，从理解的角度来说，面向对象设计的益处似乎不是来自对象模型特别忠实于现实世界的结构，而是来自这些操作与它们所影响的数据都一起放在一个局部，而不是作为庞大而复杂的全局结构的一部分。

2.11 小结

- 面向对象建模语言是建立在抽象的对象模型的基础之上的，对象模型将运行的系统看作是交互的对象构成的一个动态网络。该模型还提供了对面向对象程序运行时特性的抽象解释。
- 对象包含数据和一组操纵该数据的操作。每个对象和其他对象都是可区分的，不论其保存的数据或提供的操作相同与否。对象的这些特征称为对象的*状态*、*行为*和*本体*。
- 类描述了一组共享相同结构和特征的对象，这些对象是这个类的*实例*。
- 对象一般阻止外部对象访问自己的数据，称为*封装*。
- 对象图显示运行时的一组对象以及对象之间的链接。对象可以被命名，并且可以显示它们的属性值。
- 对象通过发送消息和其他对象合作。当对象收到一个消息时，它执行自己的一个操作。向不同的对象发送相同的消息可以引起执行不同的操作。
- 对象之间以消息形式进行的交互（包括参数和返回值）可以在对象图中表示。
- 类图提供了对一组对象图中所显示的信息的抽象总结。它们显示的信息与一般在系统源代码中找到的信息相同。
- 在面向对象建模中经常使用的一条经验规则是以现实世界中发现的对象作为设计的基础。但是，以这种方式得到的设计的适宜性需要谨慎地评价。

2.12 习题

（1） 画出一个完整的类图，描述本章讲述的库存控制程序的最终情形，要包括在代码段中定义的所有属性和操作。

（2）假设执行了下面一段代码：

```
CatalogueEntry frame = new CatalogueEntry("Frame", 10056, 49.95) ;
CatalogueEntry screw = new CatalogueEntry("Screw", 28834, 0.02) ;
CatalogueEntry spoke = new CatalogueEntry("Spoke", 47737, 0.95) ;
Part screw1 = new Part(screw) ;
Part screw2 = new Part(screw) ;
Part theSpoke = new Part(spoke) ;
```

(*a*) 画图说明已经创建的对象、它们的数据成员以及它们之间的链接。

(*b*) 下面的代码创建一个 Assembly 对象并向其中加入了上面创建的一些对象：

```
Assembly a = new Assembly() ;
a.add(screw1) ;
a.add(screw2) ;
a.add(theSpoke) ;
```

画图说明这段代码执行之后组件 a 中包含的对象以及它们之间的链接。

(*c*) 下面代码的执行可以说明向组件 a 发送了一个 cost()消息。

```
a.cost() ;
```

将执行这个函数期间会在对象之间发送的消息加入到图(*b*)中。

（3）图 Ex2.3 中的对象图描绘了库存控制系统的不可能状态，根据图 2.5 和 2.12 中的类图，解释这是为什么。

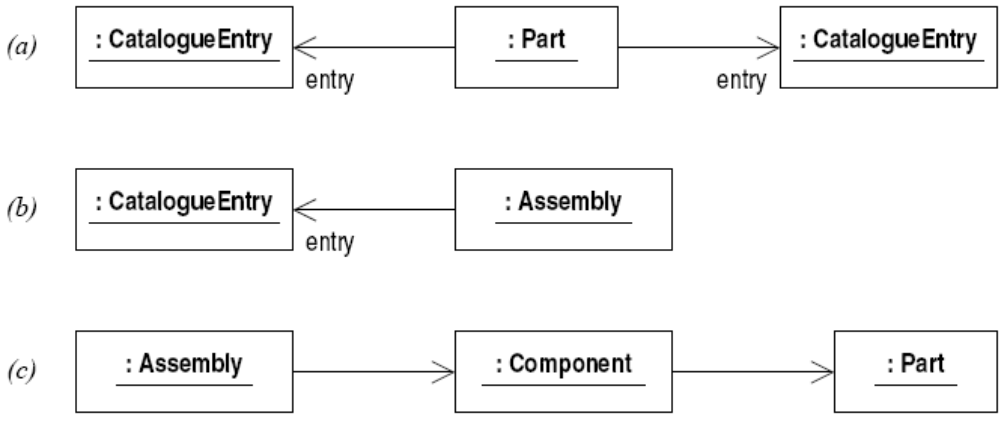


图 Ex2.3 库存控制程序的非法状态

（4）用一个图举例说明下列 UML 的对象、链接和消息表示法的使用。

(*a*) 类 Window 的一个对象，不显示属性。

(*b*) 类 `Rectangle` 的一个对象，以及属性 *length* 和 *width*。假设矩形类支持一个返回矩形对象的面积的操作。

(*c*) Window 对象和 Rectangle 对象之间的一个链接，模拟矩形定义了窗口的屏幕坐标的事实。

(*d*) Window 对象向 Rectangle 对象发送一个消息，请求得到它的面积。

画出一个类图，描绘具有题中提到的特性的 `Window` 和 `Rectangle` 类。

（5）假定一个环境监测台（monitoring station）装有三个传感器（sensors），分别是温度计（thermometer）、雨量计（rain gauge）和湿度计（humidity reader）。另外，还有一台打印机作为输出设备，显示这三个传感器的读数。每5分钟取一次读数，并转录到打印机。这个过程称为“获取检查点（checkpoint）”。

(*a*) 绘制一个对象图说明这些对象可能的配置，图中要包含在每次获取检查点时系统中可能产生的消息。假设检查点由一个从定时器（timer）对象发送到监测台的消息启动。

(*b*) 图(*a*)中是否清楚地显示了消息发送的次序？如果没有，应该如何说明？

(*c*) 画出概括监测台结构的类图。

（6）一个工作站（workstation）当前有三个用户（user）登录，帐户（account）分别是 A、B 和 C。这些用户运行了 4 个进程（process），进程的 ID 分别是 1001、1002、1003 和 1004。用户 A 正在运行进程 1001 和 1002，B 在运行 1003，C 在运行 1004。

(*a*) 绘制对象图，描述表示工作站、用户和进程的对象，并显示出工作站上运行的进程和拥有进程的用户的关系的链接。

(*b*) 考虑有一个操作，列出当前在工作站上运行的进程的信息。该操作可以报告所有当前进程的信息，或者在用适当的参数调用时，报告某个指定用户的进程的信息。讨论为了实现这个操作，需要在(*a*)部分所显示的对象之间传递什么消息。

（7）对本章讨论的程序的另一种设计可以是取消零件类和目录条目类，而用不同的类来表示每种不同类型的零件。模型中可能包括例如“螺旋（screw）”、“支杆（strut）”和“螺栓（bolt）”等类。每种零件的查找号、描述和成本将作为静态数据成员保存在相关的类中，单个零件只是这些类的实例。

(*a*) 这种改变对程序的存储需求会产生什么影响？

(*b*) 为这种新设计设计一个组件类。为了保证组件能够包含不同类型的零件，你需要作什么假设？

(*c*) 画出这个新设计的类图。

(*d*) 随着系统的进化，可能必须加入新类型的零件。解释在两种情形下这将如何完成：一是在本章提出的原始设计中，二是在本习题考虑的另一种设计中。

(*e*) 根据这些考虑，你认为这两种设计中哪个更可取，是在什么情况下更可取？

（8）假设为库存控制系统定义了一个新需求：维护库存中有的并且当前没有在组件中使用的每种零件的数量。确定这些数据应该保存在哪里，并在适当的对象图上画出消息，显示每当将零件加入到组件时数量是如何递减的。

（9）组件的 “零件分解”是指一份报表，它以某种适当的格式完整地列出了组件中的全部零件。扩充库存管理程序，要求支持打印一个组件的零件分解。如同图 2.8 中那样，通过一个表示典型组件的包括有消息的对象图来阐明你的设计。

（10）在 2.5 节中提到，创建 Part 对象时没有将其链接到适当的目录条目对象是错误的。但是，2.5 节中给出的 Part 类的构造函数并没有强制这个约束，因为它没有检查传给它的目录条目引用不是 null，如果是 null，那么将创建一个没有链接到任何目录条目的 Part 对象。扩充 2.5 节定义的构造函数，以切合实际的方式处理这个问题。

第 3 章 软件开发过程

如在第 1 章所讨论的，每种软件开发方法学都包括两个主要构成部分：软件开发活动实施过程的描述和该过程的结果文档的表示法。还强调了 UML 是一种语言而不是方法学，也不是为支持任何特殊过程而专门设计的。UML 定义的图和表示法可以成功地用于各种不同的过程。

本章描述有关软件开发过程的一些思想的发展，从第一个广为人知的过程模型开始，以对统一过程的简要描述结束。尽管几乎可以肯定地说，统一过程不是过程建模的最终结论，但是将统一过程和 UML 结合起来考虑很有意义，因为二者是由同一组研究人员开发的。本章最后将讨论统一过程所强调的不同 UML 图之间的逻辑关系。

3.1 瀑布模型

定义软件开发过程的早期尝试，主要是以从其他工程学科获得的思想为基础，把这些思想应用于这个新的领域。关键的思想是标识在生产软件中涉及的不同的活动，例如设计、编码和测试，并定义应当以怎样的次序实施这些活动。这导致了过程模型的产生，例如经典的“瀑布”模型，如图 3.1。

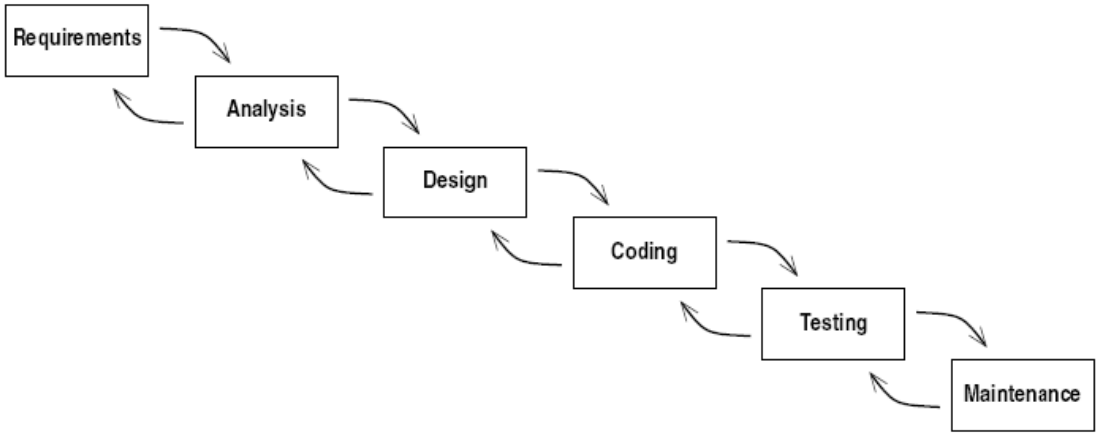


图 3.1 瀑布模型（Royce,1970）

虽然这些阶段的确切数目和命名在瀑布模型的不同版本中不尽相同，但是在所有版本中，这个过程总是系统地从高层开始，说明性地描述系统应当做什么，经由详细描述，最终是用代码详细描述系统将如何做，最后，以系统部署和后续运行所涉及的活动的一些阶段的描述而结束。

因此，大多数瀑布模型提供的是软件系统的整个生命周期的一个模型，而不仅仅是软件系统的开发模型。但是，系统的开发构成了模型的很大一部分，无疑地，这也是在软件工程文献中被关注最多的部分。

瀑布模型中的每个阶段都定义一个活动。瀑布中连续的阶段通过箭头连接，向下的箭头表示阶段的时间次序。因此，模型的基本结构暗示了不同的阶段是依顺序进行的。所设想的过程并不是像建造一所房子那样，电工、水工、屋顶工等等可能同时工作，而是更像一条生产线，比如，分析人员完成他们的工作，接着将产品移交给设计人员进行下一个阶段的工作。这个模型中隐含的是：期望每个阶段完成的工作都以某种方式形成文档，而且该文档将被传递下去，并成为下一阶段要做的工作的基础。

因此，瀑布模型提出了软件开发的一种理想化的构想：项目将从文档化需求开始，经历分析和设计活动，接着是编码和测试系统。在部署之后，只要它还在使用，系统将被维护。这种经过这些阶段的工作流是将这类模型命名为“瀑布”的缘由。

当然，实际上，以这样一种没有疑问的方式开发的项目很少。更典型地是，一个阶段中的工作将暴露前面阶段所做的工作中的问题、错误和遗漏。显然，在继续下去之前先纠正问题是明智的，而不是在有错误的工作的基础上前进。图中向上的箭头表示了这种反馈过程，由此在一个阶段做的工作可能引起前面阶段的修改和返工。

在图 3.1 中没有明确这种反馈是否可以往回延伸越过多个阶段。瀑布模型的一些最初的版本限制只能反馈到过程中的前一个阶段。尽管从管理的角度看来这显得很有吸引力，但是这似乎是一个相当武断的限制。例如，如果编码暴露了系统设计中的重大问题，完全可以想象的是这些问题本身可能严重到了需要修改分析阶段所作的工作。如果说事实上问题是在编码阶段碰巧发现的，而不是在设计审查阶段，这似乎也不能作为拒绝考虑对分析进行必要修改的有力理由。

总之，瀑布模型代表了一种直观、实用、通用的工程方法在软件工程中的应用。它强调在执行活动之前先进行设计的重要性，并因此提供了一种有价值的平衡，平衡软件开发中对总体架构或程序结构不作任何考虑就开始编码的普遍倾向。与此相关，它建议一种自顶向下的方法：最初在较高的抽象层次上描述系统，随着开发的进行而增加更多的细节。最后，至少在理论上，它提供了控制软件过程的一种有力的管理工具：使用该模型，能够给予项目一个清晰的结构，在每个阶段的最后有一个里程碑，标志着该阶段的完整文档的产生并为下一阶段做好了准备。

然而，实际上，瀑布模型的应用并没有真正做到它所承诺的那样，提供一种组织软件项目的方法，使得项目将会是可控制的，按时产生在预算内交付功能正确的系统。造成这种失败的两个主要原因是，该模型对项目中涉及的风险的管理方式和模型中对系统需求的处理。

3.1.1 瀑布模型中的风险管理

与瀑布模型建议的相反，在软件开发中，尤其是新系统或者复杂系统的开发中，存在着高度不可预知的活动。有许多软件项目之所以失败或取消，是因为含有需要花很大代价去更正的错误或者由于性能问题而使系统不能使用，或者只是因为它们不能可靠地工作。开发过程的一个主要目标就是：寻找可以让导致项目最终彻底失败的风险最小的方法。

虽然已经有很多相关工作是关于开发可证明正确的软件的技术，但在软件开发过程中，测试仍然是极其重要的部分。测试虽然远不够完美，但是，要在合理的时间内保证程序所做的是预料要做的事情，测试是最有效的方法，并且没有意外的副作用。

在找不到其他技术的情况下，只有测试系统，开发者才能获得关于系统实际行为的信息，而不是计划的信息。如果系统的功能或性能存在严重问题，也只有到测试时才会被发现。

在瀑布模型中，测试活动一直推迟到开发的最后一个阶段。某些测试能够早于模型建议的时间进行，比如在编码阶段，程序员可以测试他们开发的模块，但是瀑布模型建议，只有在过程中的早期阶段完成之后，才将系统作为一个整体测试。

那么，就其本质而言，瀑布模型提出的过程就把项目中的多数风险推迟到开发周期将近结束时才能发现。如果测试是已知的揭露问题最有效的方法，而测试在很大程度上推迟到其他活动已经完成才进行，那么跟随而来的是，在遵循瀑布原理管理的项目中，总是存在着只有在开发末期才能发现严重问题的重大风险。

测试中揭露的问题决不只是由编码错误引起的。测试暴露出分析阶段是建立在错误假设的基础上，或者指导整个后续开发的选择被证明是不可行的，也并不少见。

因而，潜在地，测试能够暴露一直追溯到项目一开始的问题。在最坏的情况下，这可能意味着所有后续工作都是无效的，不得不重做。在成本方面，这意味着没有实际设置修复测试中所暴露问题的成本，这当然会给软件项目管理和控制带来严重的后果。

3.1.2 瀑布模型中的系统需求

瀑布模型假设的是，需求获取是过程中一个独立的步骤，就像测试一样，只不过是出现在项目一开始。并且假设结果能够产生关于系统需求的明确陈述，并作为后续开发的基础。经验表明这是与实际不符的假设，会导致系统开发中的很多问题和失败。

为什么在软件项目开始制定一个确定的需求陈述比在其他领域更困难，这似乎有很多原因。首先，许多软件系统的需求非常复杂，要明确地预见每种必须考虑的情况并定义合适的响应是很困难的。这意味着期望在项目一开始就能达成稳定的需求陈述往往很不现实。

其次，许多软件项目都被要求适应动态的和快速变化的环境。例如，考虑正在由财务公司编写的一个复杂的贸易系统。对这样一个系统的需求将会受到许多外部因素的影响，包括公司的商业惯例以及贸易所处的法律体制。在系统开发期间，作为正常经营活动的一部分，很有可能这两个因素都发生重大改变，而这两类改变都将意味着系统的需求陈述需要重大修改。

最后，用户参与软件开发过程可能会改变他们对系统需要的是什么的感知。即使系统是按照最初的需求说明书开发的，用户也会频频抱怨结果不是他们预期的或要求的。这暗含着许多原因。例如，可能是与用户商议获取原始需求的过程有问题：用户所做的假设没有明显地说明，或者可能被写需求文档的人曲解了。此外，目睹一个具体系统的经历也会暗示用户，他们可能以另外的方式使用系统，这样，当系统交付时，对系统的业务需求实际上已经改变。

3.2 非瀑布模型

瀑布模型的两个主要问题是项目中的风险管理和能够在项目一开始产生确定的需求陈述的假设。这两个问题，至少在理论上，都来自于瀑布模型所强调的，开发在很大程度上是一个以或多或少确定的次序执行固定的一组活动的过程。

作为对这些问题的回应，产生了许多其他软件开发过程的描述，本节介绍其中的两个。这些新模型提出的新观念已经被当前的过程模型所采纳，其中也包括统一过程。

3.2.1 演化模型

作为对瀑布模型过于严格的结构的认识的回应，有许多建议被提出，认为软件应该以一种更加“演化”的方式开发。这意味着开发应该从产生一个实现，也许只是模拟完整系统的一小部分核心功能的原型系统开始。

然后，可以和用户讨论这个原型，用户也能够获得使用系统的第一手经验。于是，来自用户的反馈将指导后续的开发，随着在每个给定阶段小规模的功能性增量的增加以及与用户反复商讨，原型将演化为一个更完整的系统。

演化方法希望通过用户参与开发过程，避免在项目结束时才发现开发的系统不是用户所需要的问题，从而克服了瀑布模型的一个缺点。同样地，强调开发一系列可用的原型意味着演化中的系统从项目的早期阶段就已经被测试，使得在项目早期就确定问题成为可能，并将风险分散到了生命周期的各个部分。

演化模型的缺点在很大程度上与进行中的项目的管理及其可预测性有关。由于模型的特性，很难看出项目经理如何能够合理地规划一个项目，或者进行项目的工作分配。此外，该模型也没有保证演化过程实际上总是会向着一个稳定的系统汇聚，而且没有如果事实上未能如此汇聚时的应对策略。最后，演化过程也不能保证，所形成的系统的整个体系结构将会使维护活动能够有效进行，比如修复错误和实现系统的变更。

3.2.2 螺旋模型

螺旋模型的开发是为了尝试开发一个明确的软件过程，克服瀑布模型的缺点，但与演化模型相比又保留了瀑布模型传统的面向管理的优点。图 3.2 展现了螺旋模型的结构。

螺旋模型脱离了瀑布模型提出的线性结构，而将软件开发描绘为以一系列迭代不断进展的过程，重复地修正相同的阶段和活动。项目的进展被形象化为沿着图 3.2 中的螺旋顺时针移动，从图的中心开始，在外围的一个点结束，项目完成。

图 3.2 中虚线箭头表示了该模型的两个主要方面。距中心的距离指出项目到一个确定的点所需要的成本。图 3.2 是解释性的图，其中将成本表示为按恒定的速度增长，但描绘实际项目的螺旋并不是这么规则的结构。

从开始点渐增的角位移显示了自项目一开始所经过的时间，同样，这不是一个线性的度量，而是每完成旋转一圈表示项目的一次单独迭代，但不同的迭代将包含不同的活动，并且不会消耗同样的时间。

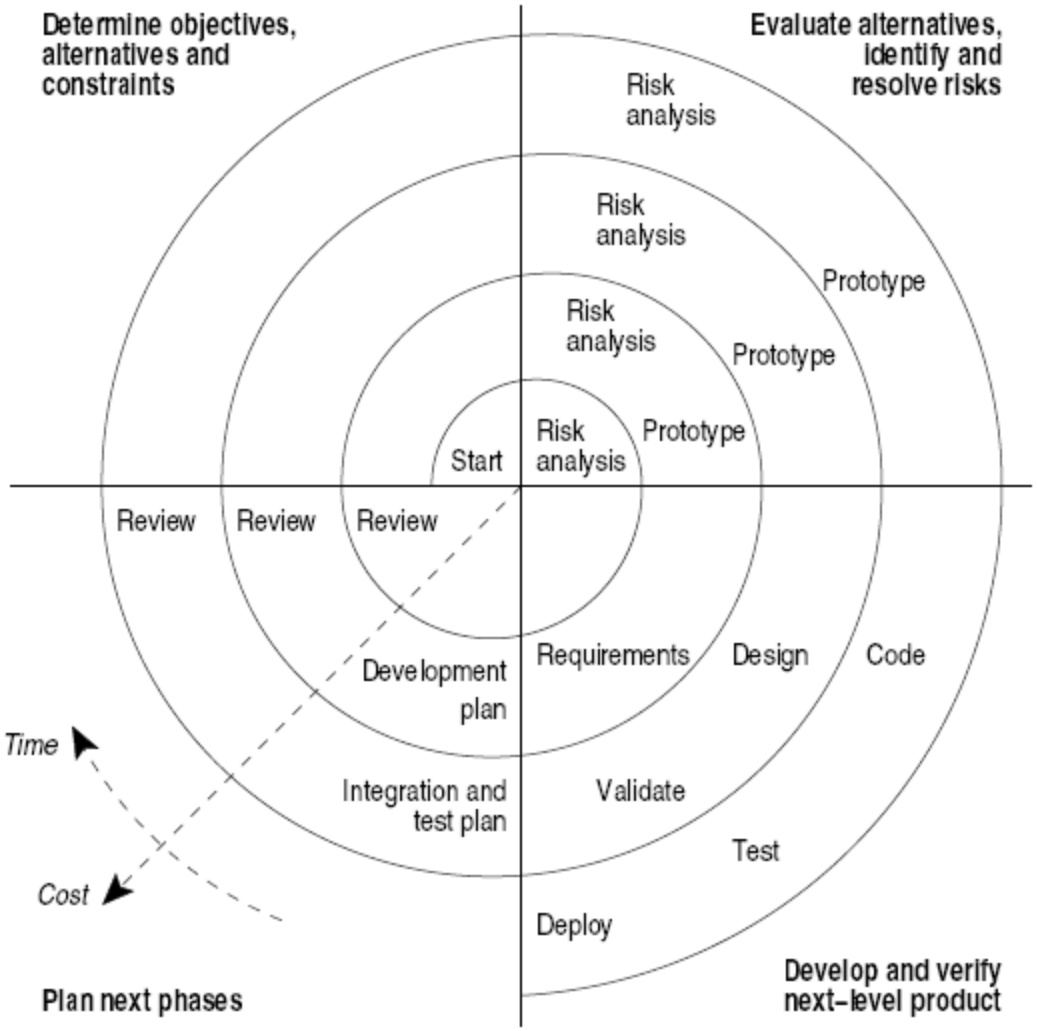


图 3.2 螺旋模型（after Boehm，1988）

图中的四个象限表示在每次迭代中应该保证模型提出的四个主要活动。每次迭代从左上象限开始，考虑本次迭代的目标，各种应该考虑的候选解决方案，以及开发解决方案必须遵守的约束。一旦确定了这些，就进行风险分析，如第二个象限所示。这样做的目的是保证在项目的每次迭代时，重点关注构成威胁的最高风险。风险分析之后，在提交到具体开发之前，应该先构造原型，以帮助评估解决风险的可能方法。

一旦确认和解决了当前的风险，接下来每次迭代是产品的开发和验证阶段。这个阶段可能由不同的活动组成，这取决于项目到达的时期。图 3.2 表明，随后的迭代遵循一个传统的轨迹，从需求到分析到编码，但是这种顺序并不是螺旋模型的必备特征。每个开发阶段都包括一个验证步骤，例如测试该阶段所写的代码。

最后，每次迭代都以对本次迭代中所做工作的评审结束，同时制定随后的迭代计划。螺旋模型主要的创新在于，它提出了开发应当被作为正规的一系列迭代来组织，每次迭代包含对项目风险的明确考虑，并且该次迭代中所做的工作的意图就是要解决所认识到的最大的

风险。根据项目的种类，风险可能各种各样，并且也不局限于潜在的技术问题，类似进度安排和预算控制等因素也是风险的重要来源。

由此可以得出结论，螺旋模型没有像瀑布模型那样以简单的方式提出一个单程的确定的过程模型。而是提出项目计划编制的进行应该贯穿于项目的生命期，而且项目经理必须准备好按照进度和发现的风险随时修改计划。

根据影响给定项目的风险的种类，在某些情况下螺旋模型可能看来与其他过程模型类似。例如，如果对一个特殊的项目，与需求获取相关的风险较低，但项目管理被认为是较大的风险，那么应用于该项目的螺旋模型可能和瀑布模型非常接近。然而，如果最大的风险在于开发复杂的新用户界面，那么螺旋模型最终将类似于演化的方法，因为需要尽快开发能够尽量由用户测试的可操作的代码。

最后，值得注意的是瀑布模型和螺旋模型之间的另一层关系。如图 3.2 所示，螺旋模型中单个迭代自身可能遵循的一系列阶段会让人联想到瀑布模型。

3.2.3 迭代和增量开发

演化模型和螺旋模型确定了一个改进的软件开发过程应该具有的两个重要特性。第一，软件开发应该增量地进行：与其把完成整个系统开发的全部工作集中在一次冲刺中进行，不如首先开发核心功能，得到一个虽然有限但是能够运行的系统，然后再以一系列增量的方式逐部地增加剩余的功能。

这种方法的优点是在项目生命周期的早期就产生可以运行的代码，除了通过示范论证项目的可行性降低了风险之外，还解决了需求说明的问题。用户能够获得对运行系统的体验，并用这样的体验对随后的开发反馈修改和建议。

第二，螺旋模型提出软件过程应该被作为一系列迭代管理，而不是一次性地经历瀑布模型确定的各种活动。尽管原本的螺旋模型明确地作为管理项目中风险的一种方法被提出，但它也能够解决管理需求变化的问题：如果需求变化被确定为是一个重大风险，迭代将包含构造一个合适的原型，以引出来自用户的反馈。

当前的过程模型试图把增量和迭代开发的优点与瀑布模型提供的传统的过程结构结合起来，这类模型中如今最著名的是统一软件开发过程（*Unified Software Development Process*），将在下一节讨论。

3.3 统一过程

为了利用从软件开发的历史中学到的东西，软件过程不得不寻求一种方法来集成乍看相当矛盾的两种理解。尽管对瀑布模型有一些批评，但是它对软件开发中包含的不同活动以及各个活动产生的典型制品的确认，仍被广泛采用。例如在编码开始之前进行的需求分析和设计活动似乎具有永恒的价值。

但是，这些活动应该如何适合迭代和增量的过程并不清楚。在这些方法强调及早产生代码的情况下，它们似乎没有为独立的分析和设计活动留下多少空间。

统一过程试图用图3.3所示的开发过程的整体模型来调和这两方面的影响。在图3.3中，时间流从左到右，项目通过一些阶段和迭代而进展。图中的垂直轴表示了一个有些像传统的开发活动列表，或以统一过程的术语称为工作流（*workflows*）。这个图中最值得注意的思想，对应于每个工作流的阴影区指示的，是每个活动可能在任何迭代中发生，但是在一次迭代中这些活动的比例的协调将会随着项目从开始到完成的进展而变更。

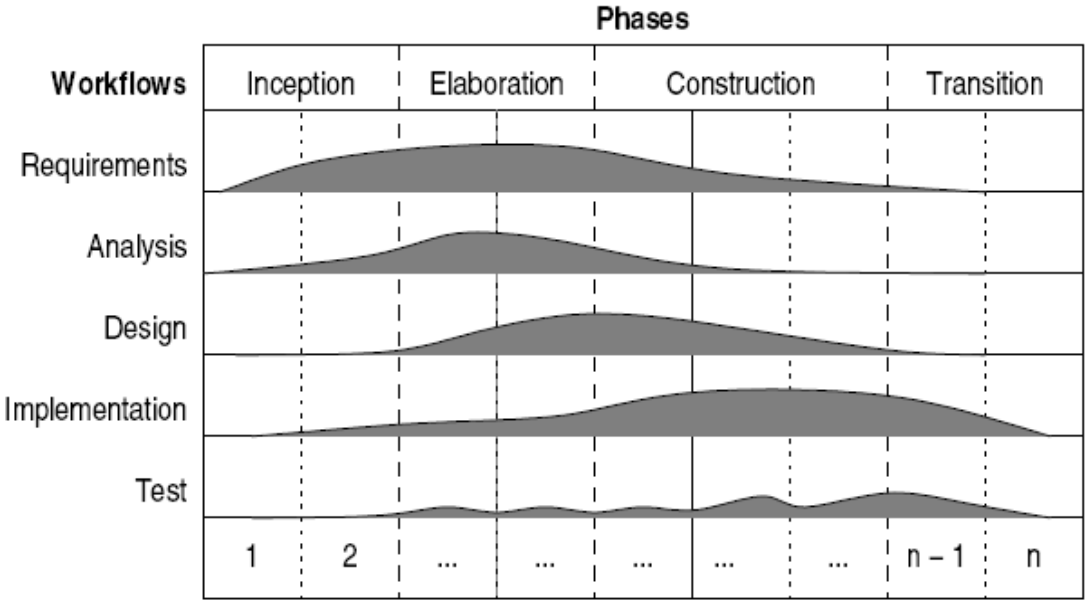


图 3.3 统一过程概览 (Jacobson 等., 1999)

过程中的四个阶段均以里程碑（*milestones*）结束，在图中没有表示出来，里程碑的意图是捕捉项目生命周期中的点，在这些点上可能进行重大的管理决策和进展评估。初始阶段主要关注项目计划和风险评估。细化阶段关心定义系统的总体架构。构造阶段是建立系统，最终得到一个能够交付给客户进行β 测试的版本。移交阶段包含了β 测试时期，以发布完整的系统而终止。

每个阶段中可能有不同次数的迭代。如图所示，一次迭代通常被组织为一个小的瀑布过程。一次迭代应该得到对所构造产品的一个增量的改进。

通常，尤其是接近项目结束时，每次迭代将会产生一个已经完全实现的功能性的增量，但是也可能是其他结果。例如，导致对系统体系结构的一些方面修改的一次迭代，如果它意味着后续的迭代能够更有效地进行，也会非常有价值。

3.4 模型在开发中的作用

统一过程认为，模型的使用在任何软件开发活动中都有重要作用。模型用在各种工作流所包含的不同的详细级别上，建立演化系统的文档。统一过程是和 UML 一起开发的，它包括非常明确的建议，说明如何使用和在什么情况下能够使用各种 UML 图，以支持每个工作流中包含的活动。

3.5 节描述了统一过程确认的各种 UML 图之间的关系。即使没有使用这个完整的过程，这些关系对于如何最佳地使用 UML 建立开发的文档，也提供了很有价值的观点。但是，在仔细考虑各种 UML 图之间的关系之前，简单考虑一下在软件开发中使用模型的其他观点是值得的。

在 3.2 节中简要讨论的演化过程很少使用软件模型，而是更注重系统代码的开发，并且在某些情况下极力地反对使用软件模型。这样的思考方法被极限编程（XP）运动再次承接。如同较早的演化模型，XP 把自己视为在软件开发中广泛使用模型的过程和清楚划分过程的反对者。但是 XP 的这两个方面是独立的，值得分别考虑。

模型的传统使用是，在编码之前产生模型，然后依照模型中包含的概要来编写代码。后续的开发，即在后面的迭代中，应该从更新模型开始，然后继续以一种有秩序的方式根据模型再更新代码。显然，这个过程只有在模型和代码保持一致时才会有用。

但是，一种典型的情景是在一次迭代的编码时，编程人员会根据他们的经验，在实现时以或多或少有意义的方式来修改设计。在这种情况下设计模型很少会和代码保持一致，结果是在下一次迭代开始时设计模型和代码不一致，因而不能保证对设计的修改能够有效地应用到代码。

这对开发团队是一个现实问题。一种解决是强调双向工程（*round-trip engineering*）的概念。这个概念建议应该开发工具，该工具能够使设计和代码保持一致，并且随着项目的发展共同演化。另一种解决是 XP 和相关方法提议的，通过只维护一套文档来消除这个问题，因为所有开发的基本产品都是源代码，XP 建议所有的工作都应该只是在代码上进行。如果团队认为使用模型有帮助，那么在迭代中可以使用模型，但是不保留模型，并且不作为系统永久文档的一部分。

因此，在软件开发的基于模型的描述和另一种只注重代码的描述之间存在一个重大的鸿沟，这两种倾向在多年以来的软件开发方面的著述中可以发现。

另一方面，在统一过程和 XP 之间也有很多类似之处。例如，二者都强调将开发建立在需求陈述基础上的重要性，而需求陈述是围绕着用户使用系统执行的任务的描述组织的。这些陈述在 UML 中被称为用例（*use cases*），在 XP 中称为情节（*stories*）。二者显然都是迭代的和增量的方法，都建议尽早实现和测试核心功能。二者都认为，随着加入更多的增量，系统的设计可能需要完全出人意料的演化，在 XP 中这一过程称为重构（*refactoring*）。

XP 明确针对小型到中型的开发。通常是十个人左右的团队，而且 XP 的核心是一组工作经验，例如结对编程、在编码之前开发自动化的测试用例和频繁的产品集成，这些可以在一个小的、有凝聚力团队的环境中采用。对比之下，统一过程则适用于各种规模的项目，并且对这类项目自顶向下的全面管理有更多表述。

3.5 UML 在统一过程中的运用

统一过程考虑了各种 UML 图在一个开发项目环境中的典型使用，在 UML 图之间存在着的某些关系是逻辑上的必然结果，即使将这些图独立地用于一个结构化过程时，也值得记住这些关系。本节将讨论和概述这些关系并将应用于后续章节的案例开发中。

3.5.1 需求

统一过程非常注重通过用例来捕获系统需求。UML 支持这一过程的略图在图 3.4 中给出。

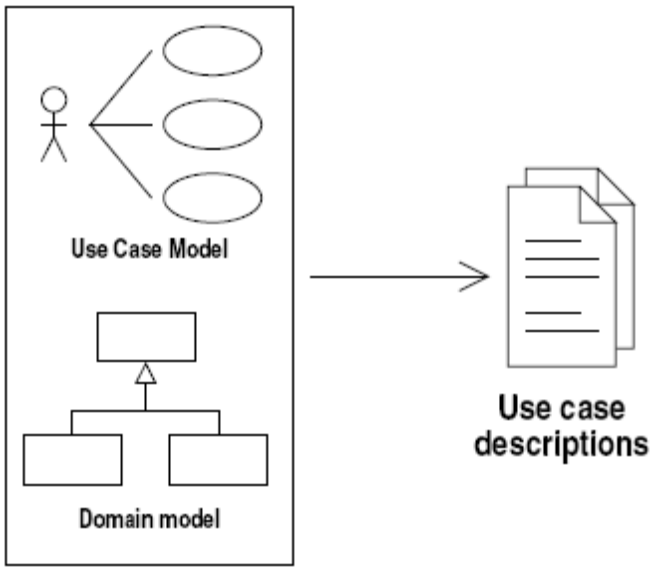


图 3.4 用 UML 捕获需求

在 UML 用例模型中捕获和记录的是系统的用例和参与者以及它们之间的关系。这通常由领域模型（*domain model*）支持，领域模型是文档化重要的业务概念及其关系的简单类图。领域模型的重要性在于，它确定了书写用例描述所使用的术语，并为消除这些描述中的不明确和不清楚提供了可能。通常，它也有助于产生列出和定义系统中关键词语的词汇表（*glossary*）。

然而，需求文档中最重要的部分是用例的文字描述，系统功能的重要细节在此处考虑并形成文档。可是 UML 并没有定义用例描述的标准形式，所以对一个项目来说，定义用于书写用例描述的模板是必要的。目前已经逐渐形成了许多不同的模板，在第 4.3 节将对用例描述可能的内容进行详细讨论。

3.5.2 用例驱动的过程

统一过程被描述为“用例驱动”的过程，这就意味着在统一过程后面的各个阶段要系统地使用用例。在分析和设计中对用例的主要使用如图 3.5 所示。

此时最重要的活动是用例的实现（*realization*）。实现是用例的高层实现，用 UML 交互图表示。它表示了系统如何将用例的功能作为对象之间的一系列交互来提供。实现中的对象来自领域模型中定义的类。

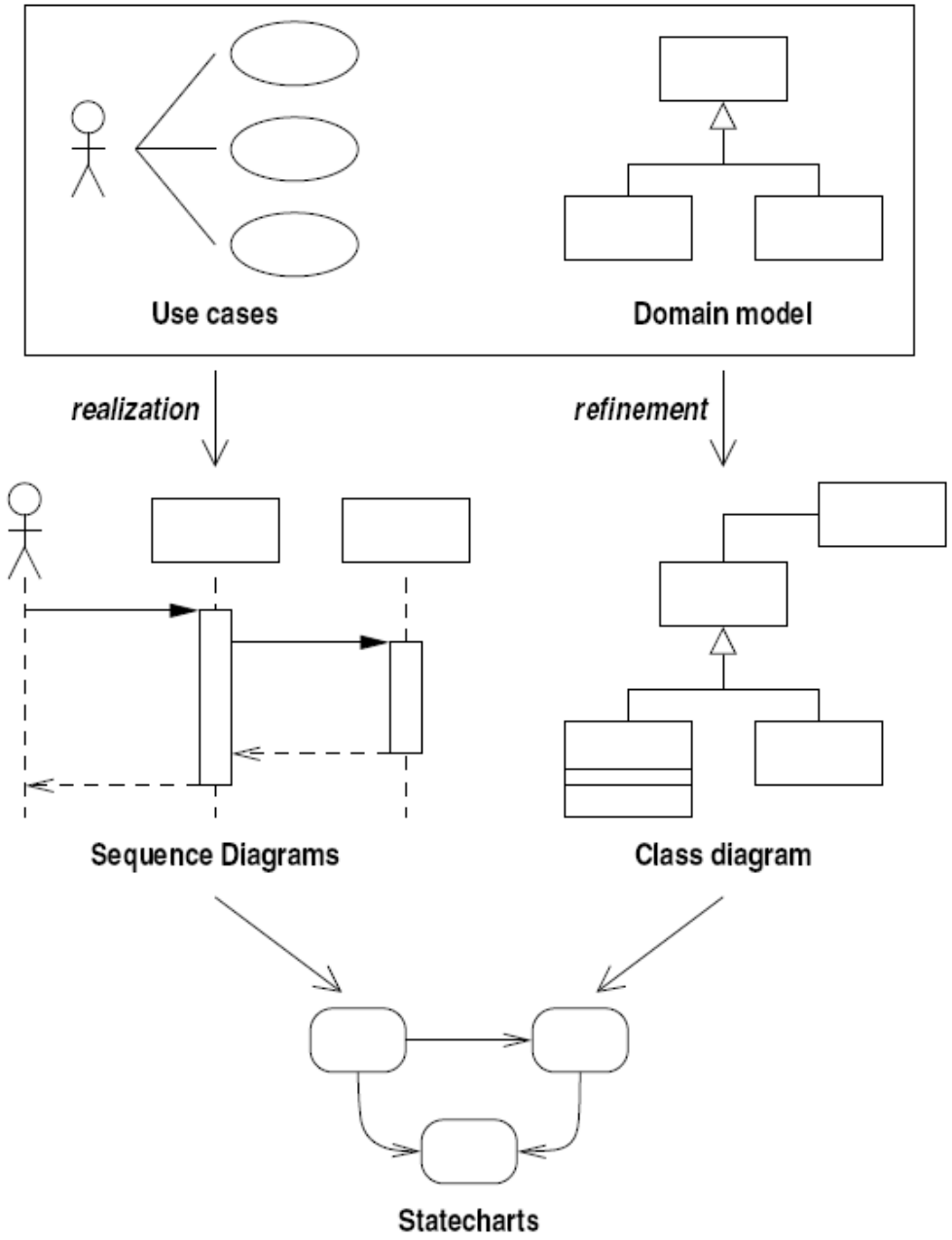


图 3.5 用例实现和细化

领域模型自然不会定义实现用例所需要的一切。实现一般揭示了对添加的类和重新确定类之间关系的需求，并强制设计人员更详细地指定支持交互所需要的属性和操作。这个细化

（*refinement*）过程把最初的领域模型发展成为一个更全面的类图，其中包含了足够的细节以形成实现的基础。

这里重要的一点是，类模型的开发不是孤立地进行的，对类模型需要做出的修改完全是由实现用例的过程推动的。这意味着，类图是以这种使设计人员能够确信，详细规约的类实际上将会支持所需要的功能的方式演进的。如果脱离对象交互的考虑产生复杂的类图，情况将不会是这样。

最后，以用例的实现和详细的类图为基础，可以为需要状态图的类开发出状态图，作为那些类的实例的生命周期的文档。图 3.5 概括地表示了用例模型的内容如何直接渗透到主要的 UML 模型产品中。

一旦产生了图 3.5 所示的模型，可以直接用这些模型来指导系统的实现，在第 7 章和第 13 章将更详细地研究这个过程。用例模型对开发过程的最后一个重大影响是产生系统的测试用例。为了准备对系统的验收测试，可以从用例中系统地导出测试：能够成功执行这些用例，是系统实现了某些对用户有用的业务功能的合理保证。

3.6 小结

- 瀑布模型把软件的开发过程设想为一个高度结构化的不同活动或阶段的序列，从需求分析开始，到测试和部署结束。
- 由于瀑布模型建议在生命周期的晚期进行测试，就将一个无法接受的风险量推迟到了项目的末尾，这时要经济地解决它通常已经太晚。
- 基于瀑布模型的项目的另一个问题是假设在项目开始就能够产生一个明确的需求陈述。
- 其他模型，例如演化模型和螺旋模型，尝试通过建议在软件开发中使用增量和迭代方法解决瀑布模型的问题。
- 当前的方法学已经采纳了这些思想，最著名的是统一过程。
- 统一过程广泛使用 UML 定义的模型，这间接地表明许多关系可以用来组织系统的文档。

3.7 习题

（1） 3.1 节讨论了在软件开发后期才测试系统所带来的风险。这是软件工程中特有的问题吗？在工程的其他分支中也面临类似问题吗？设计的制品的特性中有什么会导致这成为一个问题？其他领域的工程师是如何避免这个问题的？

（2） 3.1 节中提出，如果一个软件开发过程假定能够在项目开始产生明确的需求陈述会有问题。这对其他工程领域也是一个同样重大的问题吗？如果是，那么在设计的制品的特性中有什么会导致这成为特定领域的一个问题？其他领域的工程师是如何避免这个问题的？

（3）为什么螺旋模型提出每次迭代应该包含产生下次迭代计划的活动？

（4）你能找到增量的但没有迭代的过程或者迭代而没有增量的过程的例子吗？

（5）开源代码开发被当作是对传统开发方法的根本改变而广泛讨论，而且它能声称一些显著的成功案例，例如 Linux 操作系统。讨论如何使用本章提出的过程概念来刻画开源代码开发。

第 4 章 餐馆系统：业务建模

接下来的四章将考虑一个简单的案例，并给出一个从需求获取到实现的完整开发过程。我们将考虑一次迭代，经过统一过程标识的主要工作流之中的四个：需求、分析、设计和实现，通过这个例子说明 UML 表示法在软件开发中的使用。

由于本案例研究的意图在于强调开发的产品而不是过程，所以不会详细考虑由统一过程定义的这些工作流的结构，在真正需要的地方将在介绍 UML 表示法的同时，简略介绍开发中涉及的活动。

4.1 非形式化的需求

要开发的系统的意图：通过改进为顾客预定和分配餐桌的过程，支持一家餐馆的日常经营。这家餐馆现在采用一个手工预约系统，使用保存在一个大文件夹中的手写预约单。

图 4.1 是现在使用的预约单的一个例子，预约单中的每一行对应餐馆中一张特定的餐桌。预约是对特定一张餐桌登记的，每个预约中记录有餐具的套数，也就是预期进餐者的数目，这样就能分配一张大小适当的餐桌。这家餐馆在晚间供应三次餐点，称为“简餐”、“正餐”和“宵夜”时段。但是如同预约单所表明的，这些时段无须严格遵守，可以预约跨多个时段的时间。最后，预约中要记录联系人的姓名和电话。

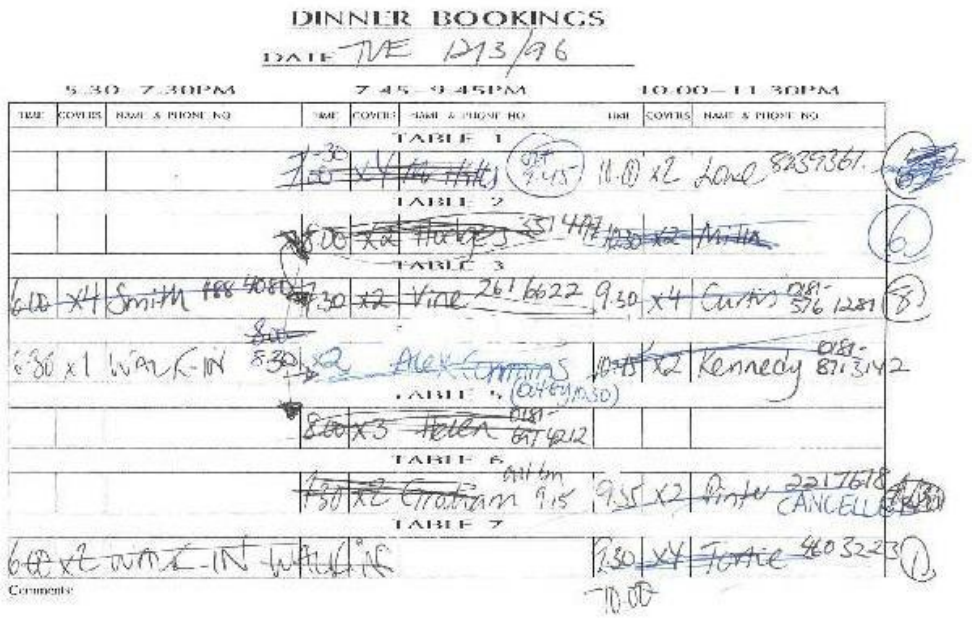


图 4.1 手工预约单

为了记录各种事情，要在预约单上加一个备注。当一行用餐者到来并在他们的餐桌就座时，就划掉相应的预约登记。如果他们就座的不是他们预约的餐桌，就画一个箭头从最初预

约的餐桌指向新餐桌。如果顾客打电话取消预约，并不能从表中真正地擦除，而是做一个预约已经取消的备注。其他的信息也可以写在预约单上，比如餐桌必须在什么时间空出来。

如果有空闲的餐桌，用餐者也可以不提前预约就进餐馆用餐，这被称为“未预约的顾客（walk-in）”，并在预约单中作为预约登记以表示餐桌被占用，但是不记录顾客的姓名或电话。

4.1.1 对计算机化系统的需要

这家餐馆的管理人员已经发现了很多与手工系统相关的问题。手工系统速度慢，预约登记单很快就变得难以理解。这可能导致经营上的问题，例如，实际上有空餐桌而由于这个预约单不是很明显，会妨碍顾客进行预约。没有备份系统：如果一张预约单被毁坏了，餐馆就没有了当晚有什么预约的记录。最后，从现有的预约单获取即使很简单的管理数据也很费时，例如餐桌的使用率。

由于这些以及其他原因，这家餐馆意欲开发一个现行预约单的自动化版本。新系统应该和现有的预约单显示同样的信息，并且格式大致相同，使餐馆员工易于转换到新系统。每当记录了新的预约或者对已有预约进行修改时，应该立即更新显示，使餐馆员工在工作时总能获得最新信息。

系统必须易于记录餐馆营业时发生的有意义的事情，例如顾客的到来。系统的操作应当尽可能是直接操作屏幕上显示的数据。例如，可以简单地将预约拖动到屏幕上一个适当的位置来改变预约的时间或者分配的餐桌。

4.1.2 定义一次迭代

迭代和增量的方法建议，系统的第一次迭代应该只交付足够使系统提供某些确实有商业价值的核心功能。在餐馆预约系统这个例子中，基本需求是餐馆在营业时记录预约和更新预约单信息的能力。如果这些功能可以使用，就有可能用这个系统代替现有的预约单，然后在后续的迭代中再开发其他功能。

4.2 用例建模

在系统可能采用的各种视图中，用例视图（*use case view*）被认为是在 UML 中起着支配作用的视图。用例视图描述的是系统外部可见的行为。因此，在软件开发始于考虑拟开发系统的需求的情况下，用例视图确立了一种强制力量，驱动和约束着后续的开发。

用例视图展示的是系统功能的结构化视图。视图定义了若干**参与者**（*actors*）和这些参与者可以参与的**用例**（*use case*）。参与者对用户与系统交互时可能充当的角色建模，一个用例则描述了用户使用系统能够完成的一项特定任务。用例视图应该包含一组定义了该系统的完整功能的用例，或者至少定义了当前迭代所规定的功能的用例，这些用例应该以在系统支持下能够完成的任务的措词给出。

理想地，用例视图对于客户、最终用户、领域专家、测试人员和其他涉及到系统的人员是容易理解的，不需要他们详细了解系统结构和实现。用例视图不描述软件系统的组织或结构，它的作用是给设计者施加约束，设计者必须设计出一个能够提供用例视图中指定的功能的结构。

4.2.1 用例

用例是系统的用户能够使用系统完成的不同任务。在这个例子中，我们将简单描述预约系统可能的一组用例，但是在真正的开发中，用例一般是由分析人员与系统未来的用户磋商确定的。

餐馆预约系统第一次迭代的意图是允许用户使用自动化的预约单。可以通过考虑在系统实现后餐馆员工能够用它来做什么，简单地草拟出这次迭代的一组初步用例。下面列出了这些用例所支持的主要任务：

1．记录一个新的预约信息（“记录预约”）。

2．取消一个预约（“取消预约”）。

3．记录一位顾客的到来（“记录到达”）。

4．将一位顾客从一张餐桌移到另一张餐桌（“调换餐桌”）。

用例不止是系统部分功能的简单描述，所以有时可以这样说：一个用例描述了系统能够为一个特定的用户做些什么：一个用例描述的是一个自包含的任务，用户总是把该任务作为他们正常工作的一部分。如果一组用例覆盖了用户使用系统完成的所有功能，这就保证了系统功能需求已经完全规约。

上面的列表简单确定和命名了一些候选的用例。为了更完全地理解每个用例中包含什么，必须像 4.3 节中说明的那样，写出每个用例的详细描述。

4.2.2 参与者

一个用例描述了系统及其用户之间的一类交互。但是，系统通常有不同种类的用户，他们能够执行系统功能的不同子集。例如，多用户系统通常定义一个角色称为“系统管理员”：这个人有权访问普通用户不能使用的功能，例如定义新用户，或者进行系统备份。

人与系统交互时能够担任的不同角色称为*参与者*（*actors*）。参与者一般对应于对系统的一个特定的访问级别，由参与者能够执行的系统功能的类别定义。在其他情况下，参与者不是如此严格定义的，而是简单对应于一组对系统有不同兴趣的人。

在餐馆预约系统的案例中，上述用例可以分成两组。第一组由与维护提前预约信息有关的用例组成。顾客将联系餐馆提前预约或取消提前预约，一般地，接待员会接听这些电话并更新预约系统中存储的信息，因此，我们能够确定一个与相应用例关联的参与者。

在第二组中有许多任务需要在餐馆营业时执行，包括记录顾客的到来，以及为了适应意料之外的业务需要将一行用餐者从一张餐桌移到另一张餐桌。这些工作可能是侍者领班的责任，因此我们能够标识另一个与这些用例关联的参与者。

区分参与者很重要，参与者是用例模型的一个构成成分，并且是系统的真正用户。一般而言，参与者和用户之间不存在一对一的对应。例如，在一个小餐馆中，一个人可以同时作为接待员和领班，可能通过使用具有不同访问特权的密码登录到系统。相反地，对应于一个参与者可能有许多真正的人：大餐馆在餐馆的每个房间或每层都可能有不同的领班。参与者甚至不一定必须是人类用户，例如，网络中的计算机可以直接互相通信，在某些系统中远程计算机可能最好能作为参与者建模。

4.2.3 用例图

用例图（*use case diagram*）以图的形式概括了系统中的不同参与者和用例，并显示了哪些参与者能够参与哪些用例。餐馆预约系统的初始用例图如图 4.2 所示，其中包括了上面确定的参与者和用例。

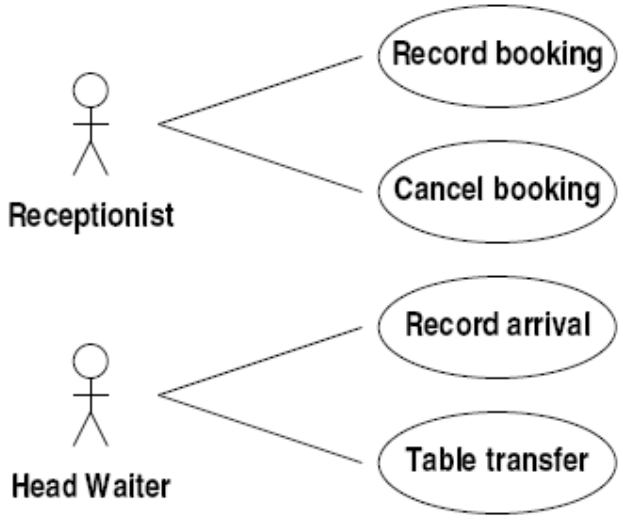


图 4.2 初始用例图

用例图最简单的形式只是显示参与者、用例和它们之间的关系。参与者用一个线条人形图标表示，用例用包含有用例名字的椭圆表示。在参与者参与的或者能够执行一个特定用例的地方，用一个连接参与者和相关用例的关联表示这种关系。

UML 允许在用例图中包含更多的结构，来定义用例之间以及参与者之间的各种关系。例如，可以定义多个用例共有的共享行为，形式化为用例的包含。

然而，在实践中不值得花费很多时间细化用例图，因为额外的关系对后面的开发起不到很大作用。如下一节所讨论的，考虑每个用例指定的行为的细节更为重要。其他的用例图表示法将在适当的地方介绍。

4.3 描述用例

用例描述了系统和它的用户之间在一定层次上的完整的交互。例如，打电话给餐馆进行预约的一位顾客，会和餐馆中一位将在系统中记录预约的店员讲话。为此，这名店员需要充

当接待员的角色并以某种方式和系统交互，虽然这并不是他们正式职位的描述。在这种情况下，该店员被认为是接待员参与者的一个实例，发生在接待员和系统之间的交互是用例的一个实例。

在用例的不同实例中会发生什么的细节，会在很多方面有所不同。例如接待员必须要输入每个新预约的特定数据，如不同顾客的姓名和电话号码，这些数据在各个实例中都不尽相同。

更值得注意的是，一个用例实例中可能会出现差错，这样将不能达到原来的目的。例如，在用户要求的时间没有合适的餐桌，用例的实例可能实际上不会产生一个新预约。用例的完整描述必须指明，在用例所有可能的实例中可能发生些什么。

用例描述可能因此而包含大量的信息，这就需要某种系统的方法来记录这些信息。但是，UML 没有定义一种描述用例的标准方式。这样做的部分原因是因为用例的意图是不拘形式地用作与系统未来用户沟通的一种辅助工具，所以重要的是开发人员应当有自由，用看来对用户有帮助并且容易理解的各种途径与用户讨论用例。尽管如此，在定义用例时能有一些可以考虑的结构还是有用的，为此，许多作者定义了用例描述的模板。一个模板实质上是一个标题列表，每个标题概括了可能记录的用例的一些信息。下面给出了两个用例描述的模板。在本章中，将简略讨论用例描述最重要的方面。

表 4.1 用例描述模板一

用例描述模板
UML 没有定义书写用例的标准模板，但是采用与项目一致的格式是有益的。此处列出可以定义用例的典型标题，然而应当强调的是，实用上更重要的是专注于写出完整的和可理解的事件路径，而不是按指定的模板填写每个部分。
名称（*Name*）
用例的名称应当用简短的动词短语表达，说明用户使用用例完成的任务。
概述或简要描述（*Summary, or short description*）
单列一节概述该用例完成什么通常是有益的。
参与者（*Actors*）
列出此用例涉及的参与者和负责发起此用例执行的主参与者。
触发器（*Triggers*）
触发者是开始此用例的事件。触发者并不必须向该系统输入事件，例如，在预约系统例子中，‘预约’用例的触发者可能是‘一个潜在的客户打给餐馆的一个预约电话’。而在另一种情况下，触发者可能是此用例中第一个系统事件。
前置条件（*Preconditions*）
前置条件概述在用例可以开始前，什么必须为真。通常前置条件说明在指定的一个用例运行前，另一个什么用例必须运行。典型的前置条件可以是‘用户已成功登录’。
后置条件（*Postconditions, or guarantees*）

后置条件概述当用例完成时什么是真。在许多情况下，这将依赖于在一个特定用例实例中发生的确切的一系列交互。区分‘最低保证’和‘成功保证’可能是实用的，前者描述在所有情况下发生什么和不发生什么，后者描述如果正常的事件路径成功地完成将会发生什么。

事件路径或场景（*Courses of events, or scenarios*）

基本的或正常的事件路径，通常应当作为不中止的交互序列出现。对事件路径中的交互通常加以编号，以便于以后参照。

可选和例外事件路径（*Alternative and exceptional courses of events*）

可选和例外事件路径可以完整地写出。然而通常只须在基本事件路径中的分叉点简单地指明可选事件流，对行为可能改变的位置予以编号，并指明导致分叉的条件。

扩展点（*Extension points*）

这部分应当列出在事件路径中可能发生扩展的位置，并给出确定扩展是否发生的条件或事件。扩展本身应当作为单独的用例写出；否则，可以指明可选的事件路径。

包含（*Inclusions*）

这部分简单地概述包含在已定义的用例中的用例。在哪些地方包含发生应当在事件路径中指明。如果显示相同信息的用例图已经使用这一节是多余的。

表 4.2 用例描述模板二

详细用例描述	
Use case name	用例名字
Related Requirements	相关需求：本用例部分或完全满足的需求。
Goal In Context	上下文中的目标：用例在系统中的位置，这个用例为什么重要。
Preconditions	前置条件：在用例执行前需要发生什么。
Successful End Condition	成功终止条件：如果用例成功执行，系统应该满足的条件。
Failed End Condition	失败终止条件：如果用例执行失败，系统应该满足的条件。
Primary Actors	主参与者：参与用例的主要参与者。通常包括触发用例执行或直接从用例执行中接收信息的参与者。
Secondary Actors	辅助参与者：参与用例执行的非主要参与者。
Trigger	触发器：由参与者触发的引起用例执行的事件。
Main Flow	主流程：描述用例正常执行时的每一个重要步骤。
Extensions	扩展：描述主流程中描述的可选步骤。

4.3.1 事件路径

用例描述必须定义在执行用例时用户和系统之间可能的交互。这些交互可以描述为对话形式：用户对系统执行一些行为，系统以某种方式响应。这样的对话一直进行到该用例实例结束。

交互可以区分为“正常”交互和其他各种情况的交互。在正常交互中，用例的主要目标可以没有任何问题并且不中断地达到，而在其他情况中一些可选的功能会被调用，或者由于出错导致不能完成正常的交互。正常情况被称为基本事件路径（*basic course of events*），其他情况称为可选的（*alternative*）或例外的（*exceptional*）事件路径，取决于它们被看作是可选的还是错误。用例描述的主要部分是对用例所指定的各种事件路径的说明。.

例如，在“记录预约”用例中，基本事件路径将描述这样的情况：一位顾客打电话进行预约，在要求的日期和时间有一张合适的餐桌是空闲的，接待员输入顾客的姓名和电话号码并记录预约。这样的事件路径，如下所示，能够以比较结构化的方式表示，以强调用户的动作和系统响应之间的交互：

记录预约：基本事件路径

1. 接待员输入要预约的日期;
2. 系统显示那一天的预约;
3. 有合适的餐桌可以使用；接待员输入顾客的姓名和电话号码、预约的时间、用餐人数和餐桌号;
4. 系统记录并显示新的预约。

在事件路径中，常常会想到包含类似“接待员询问顾客要来多少人”这样的交互。其实这是背景信息，而不是用例的基本部分。事件路径要记录的重要事情是用户输入到系统的信息，而不是该信息是如何获得的。而且，包含背景的交互会使用例比不包含时的可复用性差，而且会使系统的描述比本来需要的更复杂。

例如，假定在餐馆没有营业时顾客在电话答录机上留下了预约请求，这将由接待员在每天开始营业时处理。上面给出的基本事件路径，对接待员直接同顾客讲话或者从录音信息中取得的详细信息，同样适用：单一的用例“记录预约”将这两种情况都包括在内了。然而，如果用例描述包含对接待员和顾客的对话的引用，在处理录音信息时它将不能适用，就需要一个不同的用例。

如果在顾客要求的日期和时间没有可用的餐桌，上面描述的基本事件路径就不能完成。在这种情况下会发生什么可以通过一个可选事件路径描述，如下所示：

记录预约——没有可用的餐桌：可选事件路径

1. 接待员输入要求预约的日期;
2. 系统显示该日的预约;
3. 没有合适的餐桌可以使用，用例终止。

这看起来有些简单，但是至少告诉我们，在这一点必须可能中断基本事件路径。在后续的迭代中，可能会为这种情况定义其他的功能，例如，将顾客的请求输入到一份排队等待名

单中。注意，确定是否能够进行预约是接待员的职责，系统所能做的只是在输入预约数据后核对餐桌事实上可用。

可选事件路径描述的情况，可以作为营业的一个正常部分出现，它们并没有指出产生了误解，或者发生了错误。在另外一些情况下，也许因为某个错误或用户的疏忽而不可能完成基本事件路径，这些情况则由例外事件路径描述。

例如，我们能够预料在餐馆客满时会有许多顾客要求预约，接待员没有任何办法解决这个问题，所以要通过一个可选事件路径来描述。相反地，如果接待员错误地试图将一个预约分配到过小的不够所要求的就餐者人数的餐桌就座时，可能就要作为一个例外事件路径描述了。

记录预约——餐桌过小：例外事件路径

1. 接待员输入要求预约的日期;
2. 系统显示那一天的预约;
3. 接待员输入顾客的姓名和电话，预约的时间，用餐人数和餐桌号;
4. 输入的预约用餐人数多于要求餐桌的最大指定大小，于是系统发出一个警告讯息询问用户是否想要继续预约。
5. 如果回答“否”，用例将不进行预约而终止;
6. 如果回答“是”，预约将被输入，并附有一个警告标志。

不同类型的事件路径之间的区分是非形式的，这种区分使用例的总体描述组织得更容易理解。以同样的方式描述所有的事件路径，在后续的开发活动中就可以用类似的方式处理。因此，譬如说在不明确的情况中，不值得花费过多的时间去决定一个特定的情形是可选的还是例外，更重要的是一定要确认给出了必需行为的详细描述。

4.3.2 用户界面原型

尽管上面给出的事件路径描述了用户和系统之间的交互，但是，它们没有明确地详述这些交互是如何发生的。例如，虽然说明了接待员输入新预约的各种信息，但是没有说明是如何做的，是直接键入到预约单中，还是在对话框中填写，或者完全是通过其他的方法。
一般而言，在用例描述中详述用户界面不是个好主意。用例描述的重点是定义系统和用户之间交互的总体结构，而包含用户界面的细节会使之不清晰。另外，用户界面应该被设计得协调一致并便于使用，而这只有在合理地考虑了各种用户任务后才能做到。如果用例描述不适当地指定了用户界面的细节，可能会使用户界面设计者的工作更加困难，或者需要大量改写用例描述。

不过，对用户界面像什么样子有一个大概的看法，可能会有助于理解用例描述。在餐馆预约系统的例子中，我们知道系统是为代替现有的预约单而设计的，那么很可能屏幕设计将接近当前使用的预约单的结构，所以我们假定屏幕布局将类似于图 4.3 中所示。

屏幕的主体显示已有的预约。在屏幕左边列出的是餐桌，屏幕上部是时间。每个预约由一个显示着相关数据的淡阴影的矩形表示。预约的日期显示在屏幕的顶部。

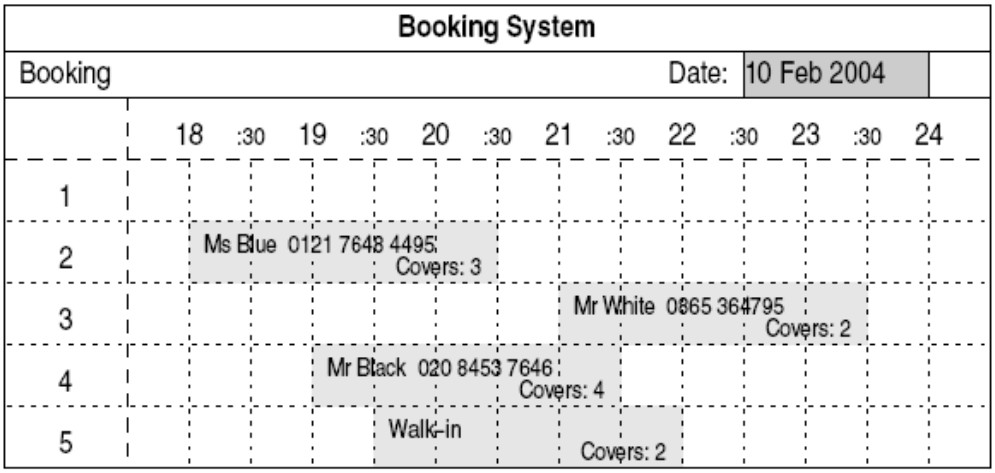


图 4.3 预约系统主屏幕的原型

这个草图没有指定数据是如何由用户输入的。对于“记录预约”用例，一种可能是用户先在日期框内键入需要的日期，得到那天已有预约的显示，然后或许可以从一个菜单中选择“记录预约”选项，并将预约数据输入到一个对话框。当完成后，应立即更新显示器，显示新的预约。

4.4 组织用例模型

记录了预约之后，接下来必须要处理的重要事件是顾客到达餐馆，这由 “记录到达”用例描述。该用例的基本事件路径如下：

记录到达：基本事件路径

1. 侍者领班输入当前日期;
2. 系统显示当天的预约;
3. 侍者领班确认一个选定的预约已经到达。
4. 系统对此进行记录并更新显示器，将顾客标记为已到达。

在这个用例中，如果系统记录中没有到达顾客的预约，可能发生一个可选事件路径。在这种情况下，如果有合适的空闲餐桌，则创建一个未经预约的登记。

记录到达——没有提前预定：可选事件路径

1. 侍者领班输入当前日期;
2. 系统显示当天的预约;
3. 系统中没有记录该顾客的预约，所以侍者领班输入预约时间、用餐人数和餐桌号，创建一个未预约登记;
4. 系统记录并显示新预约。

比较这些事件路径和为“记录预约”用例所写的事件路径，可以看到在这两个用例中存在着某些共享功能。与其多次写出相同的交互，一种更好的方法是在一个地方定义共享行为

并在需要的地方引用它。UML 定义的用例图表示法提供了一些可以这样做的方法，能够产生更简单和结构更好的用例模型。

4.4.1 用例包含

迄今描述的事件路径中，也许其中最明显的冗余是，它们全都是从参与者输入一个日期，然后系统响应，显示当天记录的预约而开始的。如果这个公共的功能能够以某种方法反映在模型的结构中将是有益的，这样，就没有必要重复写出这个公共功能。

实际上，这个交互很可能会形成一个完整的用例。例如，餐馆经理可能试图计算某个晚上要雇佣多少个服务员，那么，简单地看看当天的预约可能是估计餐馆大约会有多繁忙的一个好办法。但是，当前给出的这个模型却做不到这一点，因为检查给定日期的预约只能作为其他用例的一部分来执行。

这个理由提示我们，应该定义一个新用例，与显示给定一天的预约的任务相对应。这个用例能够被餐馆的任何工作人员执行，因而任何参与者都可以在下面对基本事件路径的描述中被提及。

显示预约：基本事件路径

1. 用户输入日期;
2. 系统显示当日的预约。

这个新用例和已经描述的用例之间的关系可以这样来描绘：只要在执行其他用例之一时就包含“显示预约”用例中的交互。

这种关系需要在用例描述和用例图中予以清晰表示。在用例描述中，如下面版本的“记录预约”用例的基本事件路径描述的，可以非形式地说明包含其他的用例。

记录预约：基本事件路径（修改）

1. 接待员执行“显示预约”用例;
2. 接待员输入顾客姓名和电话号码、预定的时间、用餐人数以及预留的餐桌;
3. 系统记录和显示新预约。

一个用例和它所包含的其他用例之间的关系在用例图中用一个连接两个用例的虚线箭头表示，称为依赖（*dependency*），用一个指定所描述关系的类型的构造型（*stereotype*）标记。图 4.4 表示了“记录预约”和“显示预约”之间的“包含（*include*）”依赖。

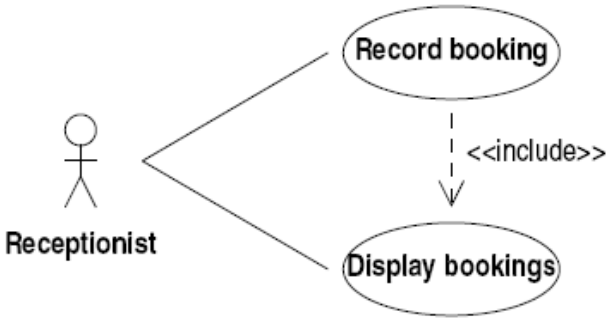


图 4.4 用例包含

注意，除了包含依赖，图 4.4 还有另外一个关联将该参与者连接到“显示预约”用例。这指明了该用例可以由接待员独立于进行新预约来操作。

4.4.2 参与者泛化

很容易更新图 4.2 中的用例图，使之包含新的“显示预约”用例。因为接待员和侍者领班都能够执行新用例，图中将包含从每个参与者到这个新用例的一个关联。

但是，这些关联表示相同的关系，因为我们假定任何人都能够显示给定日期的预约。与其两次显示实质上相同的关系，我们可以通过定义一个新的参与者表示接待员和侍者领班共有的东西来简化该图。图 4.5 中描述了这个新参与者，它表示餐馆所有员工可以共享的能力，因而称为“员工（staff）”。已有的参与者通过泛化（*generalization*）与新参与者相关，表示它们被看作是“员工”的特殊情况，定义了只能由一个员工子集共享的附加的特性。

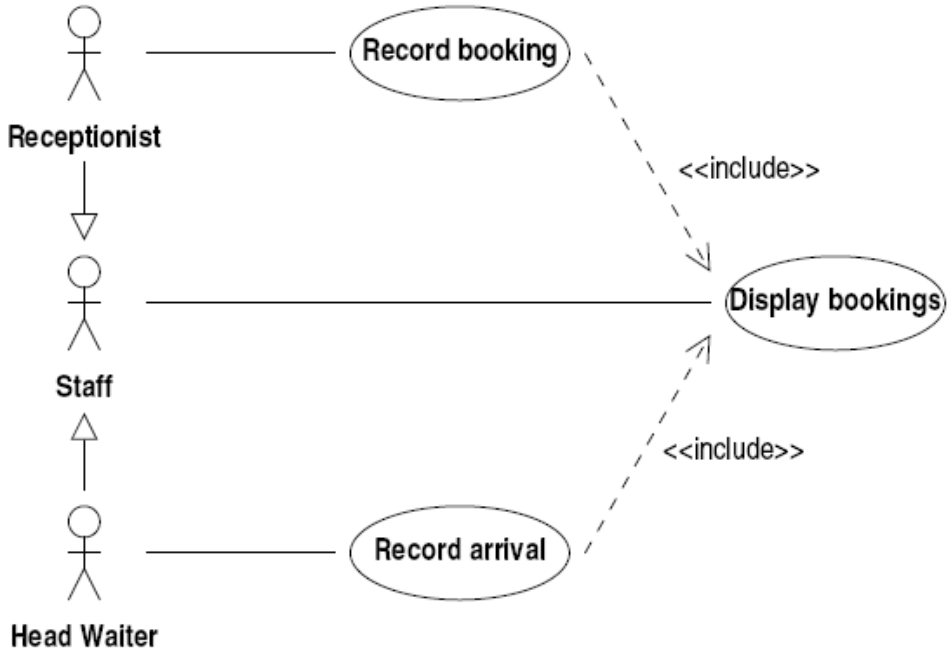


图 4.5 参与者泛化

参与者之间泛化的含意是：特化的参与者可以参与和更一般的参与者关联的所有用例。因此，图 4.5 指定了接待员和侍者领班都可以执行“显示预约”用例。另外，可以指派给特化的参与者更多的责任，图 4.5 指定只有接待员才能够记录预约，而只有侍者领班才可以记录到达，这和图 4.2 中最初的模型中定义的一样。

4.4.3 用例扩展

“记录到达”用例的可选事件路径规定，如果系统没有记录某位顾客的预约，那么侍者领班会创建一个未预约登记来表示他们在餐馆用餐。但是，将记录未预约登记表示为单独一个用例可能更好一些，因为未预约登记将会为那些从不提前进行预约的顾客创建，而且该用例可能需要独立于“记录到达”用例执行。

“记录未预约顾客”用例的基本事件路径将会被某个没有预约就来用餐的人触发。它的结构非常类似于“记录预约”用例，只是记录的细节不同。基本事件路径可以如下描述：

记录未预约顾客：基本事件路径

1. 侍者领班执行“显示预约”用例;
2. 侍者领班输入时间、用餐人数和分配给顾客的餐桌;
3. 系统记录并显示新预约。

但是，现在看起来在“记录到达”用例的可选事件路径和这个新用例的描述之间有相当多的重叠。很自然地会问：能否通过将两个用例以某种方式相关而消除重叠，或许通过上面介绍的包含依赖。

包含依赖对这种情况并不适合，因为在“记录未预约顾客”中指定的交互不是在每次执行“记录到达”时都执行。

更合适的是，它们之间有一种可选的关系：“记录未预约顾客”用例只是在“记录到达”的某些情况下被执行，也就是对该顾客没有已经记录的预约、有一张合适的空闲餐桌、并且顾客还想在餐馆用餐时才被执行。

UML 通过假定在某种情况下，“记录到达”用例可以被“记录未预约顾客”用例**扩展**（*extend*），来描述这种情形。这在用例模型中可以通过一个标记为“extend”的构造型的依赖表示，如图 4.6。注意，这种依赖和包含依赖还有其他不同，即箭头是从扩展的用例指向被扩展的用例。

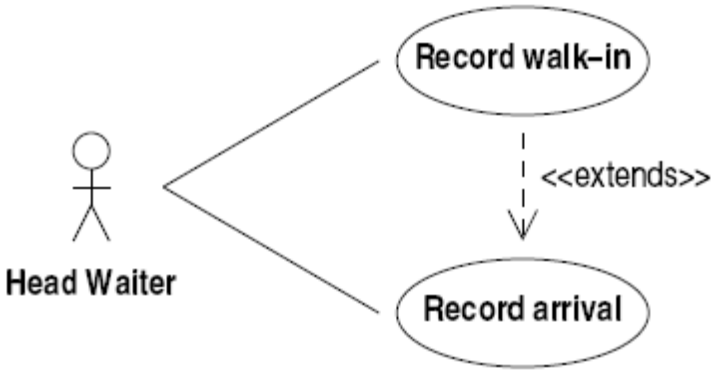


图 4.6 用例扩展

包含依赖和扩展依赖之间的区别相当微妙，并且对于这两种构造型的确切含意也有许多争论。但是，与其他的 UML 模型相比，用例模型的使用相对不那么正式，所以通常不值得花过多的时间困扰在用例之间这些关系的细节上。建立用例模型的目的是使系统的需求更容易理解，这个目的可以通过书写用例描述更好地达到，用例描述可以更清楚地说明每个用例中发生了什么。

4.5 完成用例模型

图 4.2 中剩下的两个用例很容易处理。例如，取消预约的基本事件路径可以如下指定：

取消预约:基本事件路径

1. 接待员选择要求的预约;
2. 接待员取消该预约;
3. 系统询问接待员确认取消;
4. 接待员回答“是”，系统记录取消并更新显示。

这个事件路径没有清楚地详细说明用户将如何完成这些任务。如上所述，用户界面的细节最好在后面对每个用例的功能都得到了很好的理解时再指定。

不指定用户界面细节的另一个好处是，这为系统能提供多种完成任务的方式留下了不受限制的可能性。例如，取消一个预约的一种方法可能是通过菜单选择，引出一个对话框，输入该预定的标识细节以进行选择。另一种方法是在一个预约矩形框上单击鼠标右键可能弹出一个菜单，包含一个“取消”选项。这些都是达到同样结果的方法，即取消预约，在这个阶段以一种足够通用的包含这二者的方式写出用例描述是比较好的想法。

要完成这个用例描述还有许多可选和例外事件路径要考虑。例如，可能餐馆的经营规定禁止取消过去某段时间的预约或已经记录了顾客到来的预约。对这些可选情况的说明留作习题。

“调换餐桌”用例的基本事件路径也可以独立于用户界面的细节定义如下：

调换餐桌:基本事件路径

1. 侍者领班选择需要的预约;
2. 侍者领班改变该预约的餐桌分配;
3. 系统记录改变并更新显示。

这个用例可以通过一个菜单选项调用，由用户在一个对话框中填写新的餐桌号，或者通过将预约矩形拖到它的新位置完成调换餐桌。可选和例外事件路径可以从餐馆的经营规则得到：和取消一样，应该不可能将一个过期的预约调换到新餐桌，也应该不可能将一个预约移到已被占用的餐桌。

什么时候一个用例模型完成？

最后的这两个用例的考虑可能使人联想到，使用在 4.4 节中介绍的组织机制对用例模型更进一步加细。例如，取消和调换餐桌两个用例都涉及选择一个预约并更新系统保存的关于它的一些信息。或许应该确定一个独立的选择预约的用例，包含在这两个用例之中。也许应该确定一个更通用的用例，可能叫“更新预约”，为用户提供一种一般的机制，修改与一个预约相关的数据，例如用餐人数，或者结束时间。

然而，用例分析是一项非形式化的技术，在一定时间之后再花时间寻求对模型的改进时会降低回报。这对包含关系和扩展关系尤其适用：这些关系通常与从用例产生的设计的结构特性并不相当，所以缺少一个可能的依赖的后果并不严重。

图4.7是一个完整的用例图，是上面对餐馆预约系统的第一次迭代中的用例讨论的总结。这将作为后续章节中对这个案例进一步开发的基础。

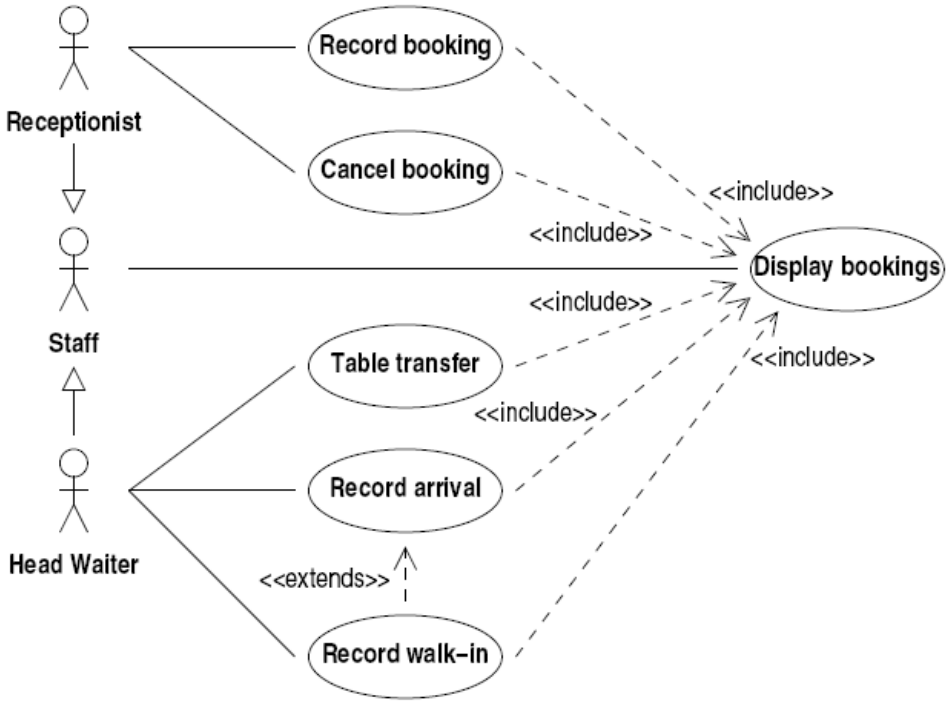


图 4.7 完成的用例图

4.6 用例建模小结

用例视图捕捉系统、子系统、类或构件对外部用户显现的行为。用例视图将系统功能划分为对参与者有意义的交互。

4.6.1 识别参与者

参与者定义了一组在功能上密切相关的角色，当某实体与系统交互时，该实体扮演这样的角色。参与者可以包括人、外部系统和设备。参与者是由角色定义的一种类型，它有实例，即承担参与者角色的具体事物。

参与者虽然在用例模型中使用，但参与者并不是系统的一部分，它们位于系统之外，是在系统外部与系统进行交互的事物。识别参与者：

1. 人员。在直接使用系统的人员发现参与者。例如启动、关闭和维护系统，从系统中获得信息，向系统提供信息。首先关注启动系统的参与者，从中找出后续与系统交互的其他参与者。从用户的角度考虑如何使用系统，特别考虑其中可能发现的三种参与者。

2. 外部系统。所有与本系统交互的外部系统都是参与者。外部系统可以是相对于正在开发的系统的其他子系统、上级系统或下级系统，即任何与当前系统协作的系统。

3. 设备。与系统交互的设备：与系统相连，向系统提供外界信息或在系统的控制下运行，这样的设备是系统的参与者。由操作系统管理的一般标准用户界面设备不包括在内。

4.6.2 用例的特性

用例是描述系统的一项功能的一组动作序列，动作序列表示参与者与系统之间的交互，系统执行该动作序列要为参与者产生结果。

用例是一种类型，它是要实例化执行的。当参与者实例与系统进行交互时，一个用例描述的功能的全部或部分才发挥作用，其中经历的动作序列即该用例的一个实例。

用例描述的是参与者所使用的一项系统功能，该功能应该相对完整。用例是一项功能的完整说明，而不能只是其中的一个片段。不能像结构化分析方法中把大的加工细分为下层的较小加工那样来细分用例。用例是不分层的，不能说上层的用例由下层的较小用例组成。

用例只描述参与者和系统彼此为对方直接做了什么事情，不描述怎么做，也不描述间接地做了些什么。用例对参与者产生结果，是指系统对参与者的动作要做出响应。

用例是相对独立的。不存在没有参与者的用例，用例不应该自动启动，也不应该主动启动另一个用例。用例的执行结果对参与者来说是可观测的和有意义的。用例比如是以动宾短语形式出现的。一个用例就是一个需求单元、分析单元、设计单元、开发单元、测试单元，甚至部署单元。

4.6.3 用例之间的关系

用例之间可以存在三种关系：包含、扩展和泛化。表示法如图 4.8 所示。

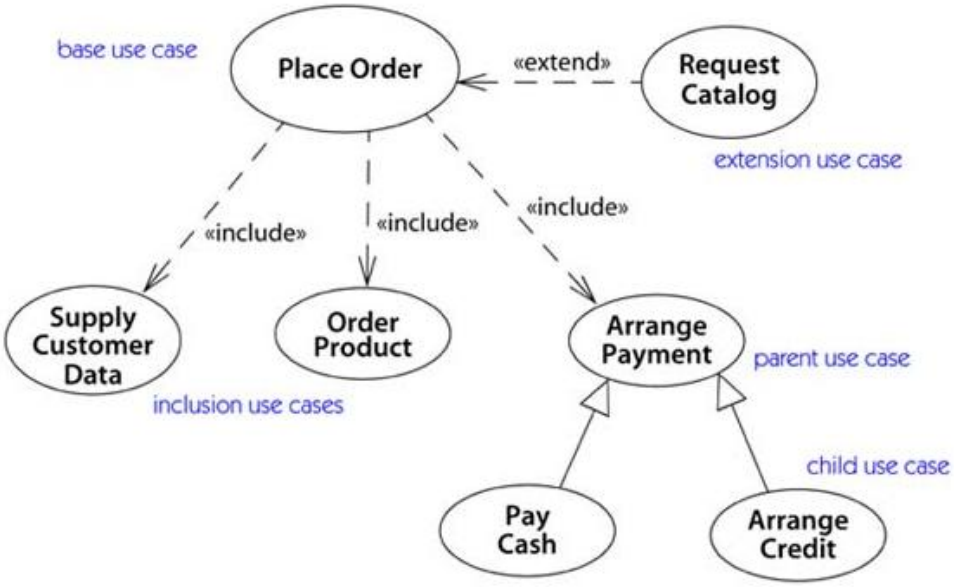


图 4.8 用例之间的关系

包含

用例中经常存在着重复的动作序列，为了避免重复，把重复的动作序列放在单独一个用例中，原有的用例（基用例）再引入该用例（供应者用例、被包含用例）。一个用例可以包含多个用例，一个用例也可以被多个用例包含。

包含关系表明：基用例在它内部说明的某一个或某一些位置上，要使用供应者用例执行的结果。

扩展

在一个或几个用例的描述中，有时存在着可选的描述系统交互行为的动作序列片段。在这种情况下，可以从用例中把可选的动作序列片段抽取出来，放在另一个用例（扩展用例）中，原来的用例（基用例）再用其进行扩展。扩展关系表明：按基用例中指定的扩展条件，把扩展用例的动作序列插入到基用例中的扩展点处。

扩展点是用例中的一个位置，在这样的位置上，如果该处的扩展条件为真，就要插入扩展用例中描述的全部动作序列或其中的一部分，并执行。执行完成后，基用例继续执行扩展点下面的行为。如果扩展条件为假，扩展不会发生。

泛化

用例之间的返回关系和类之间或参与者之间的泛化关系一样。特殊用例（子用例）继承了一般用例（父用例）的行为，还可以增加行为或覆盖父用例的行为。

4.6.4 用例图中的其他符号

用例图中其他的符号如图 4.8 所示。

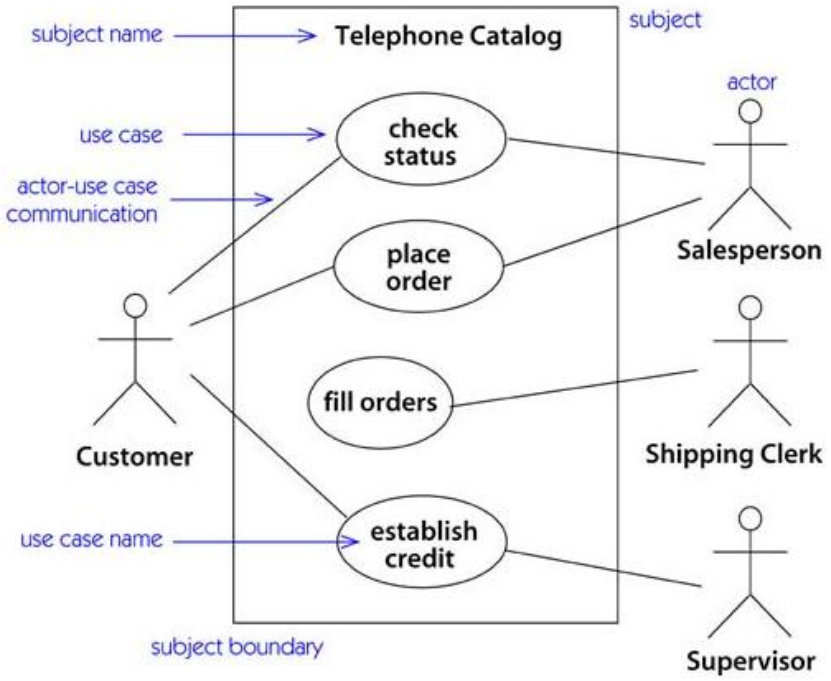


图 4.9 用例图中的符号

4.7 领域建模

用例的意图是使系统的开发人员和用户都容易理解，因此要用来自业务领域的术语进行描述，而不是使用面向实现或计算机的词汇。通常和用例建模同时进行的活动是系统地文档化在用例描述中使用的业务概念。

业务概念文档化的一种常见方法是产生一个类图，说明最重要的业务概念和它们之间的关系。这样的类图通常称为领域模型（*domain model*），对大规模的项目，领域模型通常是比较艰巨的业务建模活动的结果的一部分，较小的系统通常则可以用一个简单的领域模型充分地描述。

领域模型一般不会用到全部的类图表示法。在领域模型中，类通常表示在系统的现实世界环境中有意义的实体或概念。系统必须记录的数据作为这些类的属性建模。领域模型中还用关联和泛化描述了这些概念之间的关系。领域模型通常不包含操作，这会在后面更详细地考虑用例的实现时定义。

在餐馆预约系统中，关键的业务需求是记录顾客已经预定用餐，所以领域建模可以从标识表示预定（*reservation*）和顾客（*customer*）的两个类开始。我们知道，系统必须记录每个进行预定的顾客的姓名和电话号码，所以比较合理的是将这些作为顾客类的属性建模。

顾客已经进行了预定的事实可以通过将该顾客链接到该预定来记录，因此在领域模型中，这两个类之间的关联就建立了顾客进行预定这个事实的模型。应该指定这个关联的重数：每个预定是由一个顾客进行的，这个人的姓名和电话由系统记录，但是每个顾客可以进行多个预定。建模结果的记录如图 4.10 所示。

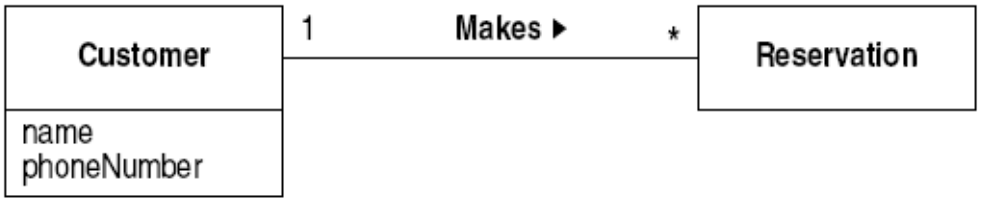


图 4.10 顾客和预定建模

然而，在预定和顾客之间还有更进一步的关系应该包含在领域模型中。这是由一个事实引起的，即预约通常是为多于一个人进行的，所有用餐的人都可以被描述为餐馆的顾客。

在领域模型包含这个关系之前，我们应该先考虑系统必须维护的信息。进行预定的人的姓名和电话必须记录，这样，如果有什么问题时可以和他们联系。但是对于到来用餐的人，重要的只是他们有多少人，餐馆没有必要记录每个用餐者的个人详细信息。因此，可能并不需要将一个预定链接到所有会来用餐的顾客，而是应该将用餐人数作为一个属性包含在 Reservation 类中。

接下来我们可以考虑对于预定必须记录的信息。预定的日期和时间是很直接的属性，可以作为属性建模。系统还必须记录分配给预定的餐桌，这可以通过将餐桌号作为预定的另一

个属性来记录，但是，还有一个选择是将每张餐桌作为一个自主对象建模，因而在领域模型中引入了一个餐桌（*table*）类。

有时很难决定是应该将一项特殊信息作为一个类还是作为一个属性包含在领域模型中。对餐桌这种情况，一个有关的考虑是，餐馆需要记录每张餐桌的其他信息，例如，餐桌可以容纳的用餐人数。在对象模型中，这项信息必须作为一个类的属性记录，而餐桌类是存储该信息的很自然的地方。图 4.11 是扩充以后包含了预定的各种特性的领域模型。

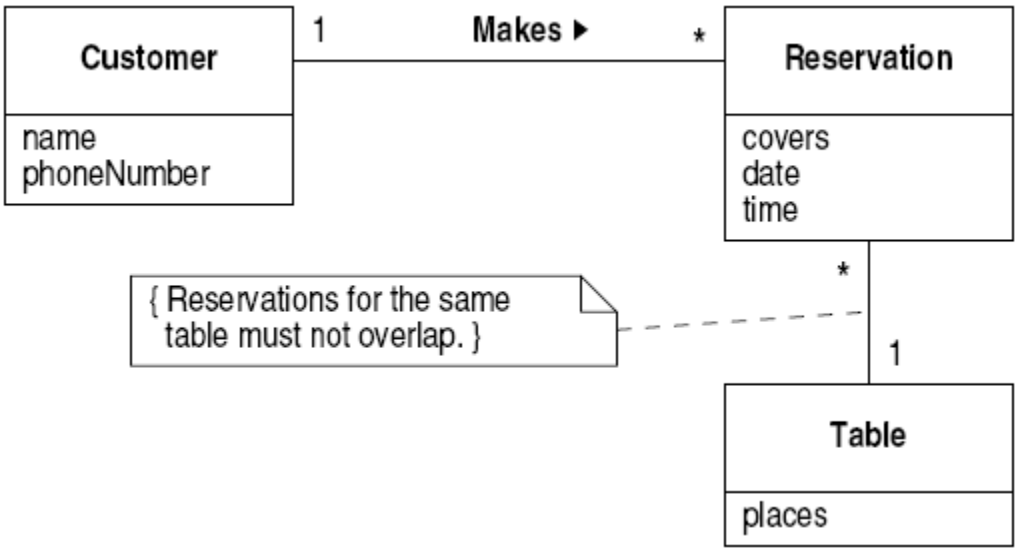


图 4.11　包含预定的特性的领域模型

Reservation 类和 Table 类之间的关联的重数指明一个预定只能对应一张餐桌。这似乎排除了餐馆为就座于多个餐桌的大团体提供预定的可能。我们假定在开发的第一次迭代中这个需求可以充分地处理，方法是对这样的团体进行多个、同时间的预约，每个预约对应一张要占用的餐桌。

在关联的另一端，这个重数断言对每张餐桌可以进行多个预定，这并不是意味着隐含会同时对一个餐桌有两个预定的意思，而是指在不同的日期和不同的时间，餐桌可以被分配给不同的顾客。

但是，餐桌不能重复预定，显然是一个重要的经营规定。这一点不能通过 UML 提供的图形化表示法表示，这样的特性是通过给相关的模型元素增加约束（*constraints*）建模的。约束是系统的所有状态都必须满足的特性，将在后面章节详细讨论。图 4.11 是在关联上用注释的形式显示了一个非形式化的约束，排除了重复预定餐桌的可能性。

对这个关联的另外一个似乎合理的约束是预定的用餐人数不能大于该预定所链接的餐桌的座位数。在大多数情况下，将会遵守这个约束，但在“记录预约”用例的描述中清楚地表明，在例外情况下，为了使大团体在那里就座，一个餐桌可以有附加的位置。因此，要增加将这种可能性排除在外的约束，将会和用例模型不一致。

领域模型没有涉及未预约的就餐者（walk-in），未预约和预定有一些共同的属性，就是存储的基本数据和到餐桌的链接，但不同的是对未预约的就餐者没有顾客信息的记录。这

表示预定和未预约应该通过定义一个一般类建模，该类记录二者的共同特性，而用特化的子类建立预定和未预约顾客的模型。对模型的细化如图 4.12 所示。

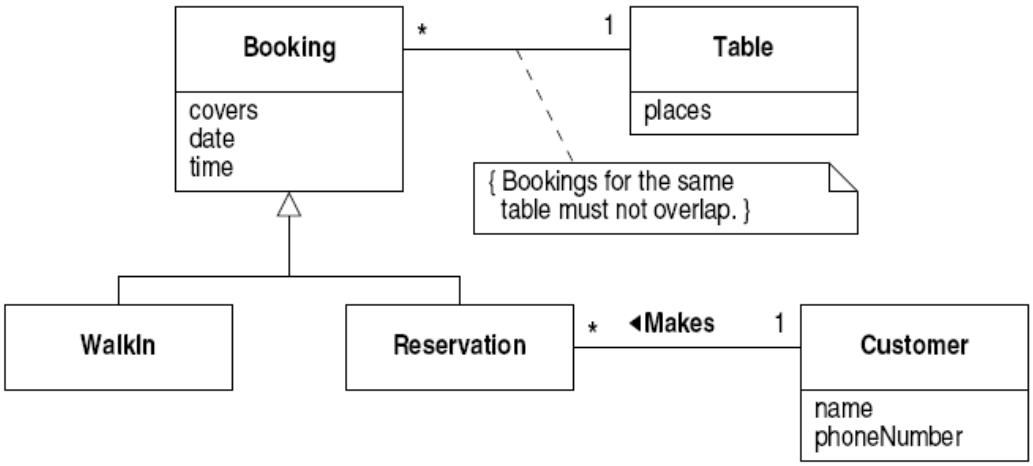


图 4.12　包含未预约用餐的领域模型

在图 4.12 中 WalkIn 类看起来是多余的，因为它没有对从 Booking 类继承的特性进行任何增加。然而，将它作为一个单独的类包含在模型中，对未来的改变起到了一种保险作用。如果后面需要在模型中增加未预约用餐者的某个特性，而预约没有该特性，在图 4.12 的设计中很容易做到。但是，如果简单地将 WalkIn 作为其父类的实例创建，那么涉及到的模型的修改会相当多。

4.7.1 领域模型的正确性

在开发领域模型时，提出对系统某些方面建模的另一种方式很常见，而且通常看起来也没有明显的理由去选择模型的一个版本，而不是选择另一个。这种例子包括图 4.12 中对餐桌号和未预约用餐者预约的处理，另外，可能会建议日期应该作为一个单独的类建模，而不是简单地作为预约的一个属性。

某些模型明显是错误的，或者是不充分的。如果根本没有包含日期，模型将不能储存预约系统需要的一项关键数据。但是，要证明一个模型的正确性或者即使是一个模型优于另一个模型，要更困难一些。从领域建模的角度看来，在将日期作为属性建模或者作为一个链接到相关预定的单独类的实例之间作出选择似乎没有太多的困难。

这个问题可以放到完整的开发中，从领域模型的目的的角度来考虑。因为设计人员最终是要确定一组对象，它们能够以有效地支持整个系统必须交付的功能的方式进行交互。因此，领域模型中可供选择的方式，从做到这一点的程度上，可以最好地评价。

然而，这个问题并不能通过孤立地检查领域模型而简单地评定。还必须通过观察领域模型中的类的实例之间的交互实际上是如何支持需要的功能，考虑模型实际上表达了什么。这是统一过程中分析和设计工作流的关键活动，将在接下来的两章中考虑。

4.8 术语表

领域模型也是对客户谈论系统所用的概念和词汇的一个详细考虑。在非正式地描写一个系统时很容易使用不明确或不一致的术语。例如，术语“预约（booking）”和“预定（reservation）”的交替使用遍及本章，但是，图 4.12 表明实际上在预约的一般概念和一个提前进行的预定之间是有区别的，因而建议对这两个术语更仔细的定义。

通常，将系统的核心词汇以一系列定义收集到一个系统术语表（*glossary*）中进行总结很有用。例如，在预约系统的开发中，迄今为止使用的术语可以列出来，并定义如下：

预约（Booking）： 分配一张餐桌给一行用餐者进餐。

用餐人数（Covers）： 预定将来用餐的人数。

顾客（Customer）： 进行预定的人。

用餐者（Diner）： 在餐馆用餐的人。

位子（Places）： 在一张特定餐桌能够就座的用餐者人数。

预定（Reservation）： 提前预约一个特定时间的餐桌。

未预约（Walk-in）： 没有提前进行的预约。

理想地，一旦创建了术语表，所有的系统文档在编辑时应该一致地使用已定义的术语。在后续的章节中，餐馆预约系统的进一步开发将使用上面定义的术语，但编辑本章前面给出的用例描述以反映这些正式定义将留作习题。

4.9 小结

- 在业务建模活动结束时，系统文档包含一个用例模型、对各个用例的文字描述、一个关键术语的术语表以及一个领域模型。
- 用例图描述了参与者和用例以及它们之间的各种关系。
- 用例表示了一类用户可以利用系统完成的典型任务。
- 参与者表示了用户在与系统交互时可以充当的角色。参与者和用例的关联，表示以给定角色工作的用户能够执行哪些任务。参与者可以通过泛化相关，以明确地表示它们共享的功能。
- 一个用例可以包含另一个用例：意思是被包含用例规定的交互构成包含用例每次执行的一部分。
- 一个用例可以扩展另一个用例：意思是扩展用例规定的交互构成被扩展用例的一次执行的一个可选部分。
- 用例描述是文字形式的文档，详细描述在执行用例时在用户和系统之间可以发生的交互。
- 每个用例描述包含一个基本事件路径，描述用例的“正常”执行，以及一组可选和例外

事件路径。

- 领域模型显示重要的业务概念、它们之间的关系和由系统维护的业务数据。领域模型表示为类图，一般只显示类、属性、关联以及泛化。
- 术语表定义业务领域中的重要术语，为每个术语提供一个一致同意的定义，定义了应该在用例描述中使用的特定业务的词汇。

4.10 习题

（1）假定餐馆经理看了图 4.2 后批评说设计不完善，因为餐馆雇不起一个专职的接待员。你如何回应？

（2）重画图 4.7 中的用例图，没有“Staff”参与者，而且不使用参与者泛化。从餐馆经营的角度来看，这两个版本的用例图描述的是相同的事实吗？你认为哪个图更清楚并且更容易理解？

（3）扩充“记录预约”用例的描述，使其包含接待员试图重复预定某个餐桌的情况。这是一个可选的还是一个例外的事件路径？

（4）系统应该不允许侍者领班对一个预定多次记录到达。考虑系统可能阻止其发生的方法，如果必要，用“记录到达”用例的一个新的事件路径描述系统的反应。

（5）写出你能想到的“显示预约”用例的任何可选或例外事件路径的描述。

（6）扩充“取消预约”和“调换餐桌”用例的描述，使之包含一个完整的可选和例外事件路径的列表。

（7）在用例描述中，对可选和例外事件路径能否进行清楚和无歧义的区分？用来自餐馆预约系统的例子给出你的答案的理由。

（8）用 4.3 节讨论的模板和 4.8 节中的术语表中列出的术语重写本章给出的非正式的用例描述。

（9）本章的讨论对如何修改预约信息很少谈及。预约只能被取消，或者移动到另一个餐桌。扩充用例模型，以允许对预约进行更一般的编辑，譬如说是通过使用对话框显示一个预约的全部信息并允许进行适当修改。

（10）按照现在的情况，领域模型将允许任意数目的一行人被分配到任意餐桌。假定餐馆决定正式规定能够加到一个餐桌的额外位子的数目为“满座数”。和以前一样，应该询问用户确认对该餐桌的预约人数过多，但决不能超过餐桌的满座数。更新进行预定的用例描述以适应这个新需求，并适当地修改领域模型。

（11）假定餐馆决定提供指定的可抽烟餐桌和无烟餐桌。需求的这个改变对本章中提出的用例模型会有什么影响？

（12）本章中的讨论没有提到预约的时间长短：只是输入了到达时间，并且我们隐含地假定所有预约有一个标准的时间长度。扩充用例模型使之允许在预约的时间长短上有更多的灵活

性，用定义一个新用例的方法调整预约的长短。这可以通过明确地输入时间完成，或者通过使用鼠标改变显示的预约矩形的长度完成。

（13）在第一次迭代中，预定由接待员手工分配到餐桌。假定已知预约的日期和时间以及用餐者人数的信息，为系统自动分配预约到餐桌的一个增量写出用例描述。

（14）扩充预约系统以支持排队等待名单。不能在要求的时间进行预定的顾客应该有放入等待名单的机会。当有餐桌可用时，可能是因为取消预约，系统应该检查在等待名单上是否有人现在能够进行预定，如果有，系统给接待员发送一个消息，由接待员和顾客联系以确定是否进行预定。扩充用例模型描述这个功能。

（15）考虑对等待名单的另一个方案说明，在有合适的餐桌可用时系统自动提醒顾客，譬如说通过发送一个文本消息给存储的电话号码。顾客随后和餐馆联系以确认他们仍然想进行预定。扩充原始的用例模型以描述这个功能。

从餐馆的业务需求的角度看，能给出什么理由要这个等待名单的一个版本而不是另一个？

（16）如果预约系统提供一个“撤销（undo）”工具，可能会很方便，这样如取消一个预约或调换餐桌的操作能够很快速并容易地撤销。如何将这个特征加入到用例模型中？对于撤销每个不同种类的操作应该有单独的用例，还是一个“撤销”用例？

（17）考虑如何改进本章中描述的系统，以允许一个预约同时有多个餐桌。描述为了表现这个需求需要对用例描述、原型用户界面和领域模型进行的修改。

（18）本章的餐馆系统案例是开发一个 PC 机上的桌面应用程序，如果要开发的是移动设备上的应用程序，那么业务建模的工作有没有什么不同？说明你的看法和理由。

4.11 实践题

（1）考虑一个网络书店系统，根据你的了解和设想建立用例模型和领域模型，给出每个用例的描述，定义术语表。

（2）考虑一个 ATM 系统，根据你的了解和设想建立用例模型和领域模型，给出每个用例的描述，定义术语表。

（3）考虑一个图书馆管理系统，根据你的了解和设想建立用例模型和领域模型，给出每个用例的描述，定义术语表。

（4）考虑校园 IC 卡管理系统，根据你的了解和设想建立用例模型和领域模型，给出每个用例的描述，定义术语表。

（5）考虑某商场的会员管理系统，根据你的了解和设想建立用例模型和领域模型，给出每个用例的描述，定义术语表。

（6）考虑某宾馆的住宿管理系统，根据你的了解和设想建立用例模型和领域模型，给出每个用例的描述，定义术语表。

（7）考虑某饮用水公司的预约送水管理系统，根据你的了解和设想建立用例模型和领域模型，给出每个用例的描述，定义术语表。

（8）考虑你熟悉的其他类似规模的系统，根据你的了解和设想建立用例模型和领域模型，给出每个用例的描述，定义术语表。

第 5 章 餐馆系统：分析

分析经常看作是软件开发中的一个明确的阶段或活动，但是，分析和设计的区分并不总是非常清晰。在面向对象方法中尤其如此：因为开发自始至终都使用相同的概念和表示法，分析和设计经常像是相互融合在一起。本章介绍的分析的观点出自统一过程，应该指出，不同的作者和方法学会给出不同的解释，甚至在有些情况下并不认为分析是一个独立的活动。

5.1 分析的目标

要定义分析的目标，有一种方法是确定分析的是什么。在完整的开发语境中，对于这个问题，一个看来合理的答案是“系统需求”。以用例描述的形式陈述的需求是定义系统外部行为非常有价值的工具，但是它们对系统的内部结构，或者如何提出一组交互的对象来支持所要求的功能并没有给出任何指导。因此，可以把分析的任务描述为是构造一个模型，说明这些交互的对象怎样才能够交付用例中规定的行为。

从工作产品来看，分析活动的典型输入是用例和领域模型。虽然这些模型描述了系统的结构和行为方面，但是这些描述并不是很完整。用例描述通过用户与系统的交互来说明从外部看到的系统功能，领域模型则定义了重要业务概念之间的关系。现在还缺少一个详细说明：表示业务实体或从业务实体导出的对象是怎样以一种能实现用例中规定的行为的方式交互的。

分析工作流借助于一种叫做实现（*realization*）的技术来解决这个问题。在实现过程中，对每个用例，应当开发一个高层交互，来说明如何通过适当类的实例的交互，产生所需要的系统行为。

面向对象系统中的类往往能够由现实世界的实体得出，根据这条指导原则，领域模型中的类通常就构成了用例实现的起点。但是，实现的过程总会导致领域模型的变更，在本章到处都可以看到这样的例子。除了实现之外，分析工作流还产生一个分析模型（*analysis model*），这是源于领域模型的类图，向其中并入了为了能够支持用例中规定的功能而增加和修改的内容。

在 UML 中用交互图（*interaction diagram*）定义用例的实现，UML 2 的交互图共有 4 种：顺序图、通信图、时序图和交互概况图。通信图（*communication diagram*）在 UML 1 中叫做协作图（*collaboration diagram*），第 2 章中说明库存控制程序的行为时已经非正式地使用过。顺序图（*sequence diagram*），在本章用于文档化用例的实现。这两种形式的图差不多是等价的，提供了表示相同信息的不同方式。顺序图清晰地说明了各种事件的发生次序，因而经常用于交互的各种事件的发生次序特别重要时。时序图和交互概况图是 UML 2 中新增加的图，在后续相关章节中介绍。

统一过程把系统架构描述（*architectural description*）的产生也放在分析工作流中。架构描述是关于系统总体结构的一些相当高层的决策的文档，并不是如何处理各个用例的局部细节。架构描述将在 5.3 节详细说明。

5.1.1 分析和设计的区别

分析最重要的任务是产生用例实现，并以此为基础，使领域模型进化为一个更全面的类图。无论是否定义了单独的分析活动，这些活动都是面向对象开发的基础。

原则上，用例实现可以很细致地进行，使得从每个用例的实现中都可以看出最终代码中的每一个交互和方法调用，并且在类模型中也展示出几乎和类实现同样多的信息。由于分析和设计自始至终都可以使用相同的技术和表示法，因此就很难给出分析结束和设计开始的一个清晰界线的正式定义。这种看法形成了是将分析和设计各自作为一个单独的活动，还是完全没有明显的分析阶段的基础。

如果要区别分析和设计，只能是非形式的，基于采取的不同观点，而不是因为技术的差异。例如，分析的重点集中在系统需求上，而在设计中重点转移到了要产生的软件的结构上。分析用对象模型表示应用领域，而在设计中我们将为设计做准备的分析模型转化为具体的软件产品。

5.2 对象设计

为了产生实现用例的交互图，必须将需要的数据和处理分配给一组对象，这些对象就可以进行交互以支持用例指定的功能。因为通常有许多不同的可能设计结果，而且哪个是最佳选择也不明显，所以，这往往是开发面向对象系统最困难也最具创造性的一个方面。

领域模型定义了一组有属性和关系的类，以此作为设计的基础是很吸引人的想法。虽然在相当大的程度上的确都是这样做的，但是要记住：领域模型有许多局限性。

首先，领域模型表示了应用领域中的重要概念。在最终的设计中对这些概念和它们的关系应该有一个相当直接的表示，这当然是面向对象设计的一个愿望，但是并不能保证情况总是这样。至少，最后的设计总会包含一些类，这些类要么虽然在领域模型中没有出现却在进一步的设计工作过程中被发现，要么是在应用领域中没有相似物的类。

其次，领域模型通常不显示操作。然而，开发设计时至关紧要的一部分就是决定每个对象应该完成什么处理，而领域模型对此没有提供任何帮助。现实世界中的行动和对象模型中的操作不同，所以即使在领域模型中增加行动，也不太可能在分析中提供多少帮助。

本节讨论对象设计的基本原则，为如何创建良好的对象结构提供一些指导。本章剩下的部分将描述该原则在餐馆预约系统的用例分析中的应用。

5.2.1 对象责任

面向对象程序设计的启示是：软件对象反映的是在现实世界或应用领域中发现的对象。这种思想在激发软件设计的某些方面证明是很有用的，尤其是在系统的静态数据结构方面。

然而，软件对象的行为和现实世界中对象的行为不同：对象通过点对点的通信进行交互，互相发送消息，而现实世界中的实体与它们的环境之间的交互、与环境中其他对象之间的交互更加丰富。

这意味着对象设计者不一定能信赖开发设计中的直觉。需要一些通用的原则或隐喻（*metaphor*）来总结面向对象的方法，并能够用于启发和指导对象的设计。对象设计者使用的最为广泛而且经久不衰的隐喻或许是：对象应该通过它们的责任（*responsibility*）来刻画。对象的责任的基本类型有两种：维护某些数据，支持某些处理。

面向对象系统中的数据并不是保存在单独一个中央数据存储库中，而是分布在系统的所有对象中。这可以用责任的术语来描述说：每个对象负责管理系统中数据的一个子集。一个对象负责的数据不仅包括它的属性值，还包括它所维护的与系统中其他对象的链接。

对象负有的另一类责任是支持某些处理，这些处理最终在它的类所实现的方法中定义。由对象进行的典型处理包括：在该对象可用的数据上进行某些计算，通过给其他对象发送消息协同进行一个较大的操作以及用它们返回的数据做些事情。

对象责任的比喻并没有给出将数据和操作分配到对象的算法，但是它的确提供了一种方式去考虑对象表示的是什么，以及在系统上下文中它应该做什么。

然而隐喻自身并不够有力，因为它几乎没有对把责任实际地分配到对象提供指导。例如，要是有一个对象，有责任存储餐馆中的餐桌的大小，可是又要更新顾客的电话号码，就有些奇怪。这两项责任似乎不应当归在一起，将它们分配到同一个对象的系统会相当难以理解和维护。

术语内聚性（*cohesion*）在软件设计中用于描述一组看来共同地属于并且组成一个合理的整体的责任的特性。负责维护餐桌大小和顾客电话号码的对象不会是内聚的：这两项任务和系统中完全不同的实体相关，而且似乎不是功能相关的。因此，看来它们应该是不同对象的责任才是适当的。相比而言，一个对象如果负责维护的所有数据都是关于顾客的，而再没有其他数据，这个对象应该是内聚的。

因而，对象设计的一个基本原则是：在进行用例实现时，设计者应该定义具有功能上内聚的责任集合的对象和类。本章剩余的部分将举例说明这条原则的应用。

5.3 软件架构

定义良好的对象应该有一组内聚的责任，这是一条很有用的原则，但是即使内聚性也是一个相当非形式的术语。也很容易举出一些对象的例子，它们是高内聚的对象，但仍是对象设计中拙劣的选择。例如，在餐馆预约系统的情况中，可能会提出：顾客对象应该负责任何与顾客有关的事宜，包括在屏幕上显示顾客信息、维护顾客的姓名和电话号码，以及将这些数据存储到一个关系数据库中。从某种意义上说，这样的对象是高内聚的，但经验表明，将如此大范围的活动并入单独一个对象中不会产生简单而且可维护的设计。

因而，除了关于对象应该如何设计的通用原则之外，能够利用过去的设计经验提供一些例子说明哪些设计选择有成效，而哪些没有成效，也很有用。与其将经验整理成抽象的原则或隐喻，倒不如把这些在过去已经使用过的并且是成功的设计策略，编纂成非常具体的设计策略的例子给出。习惯上将这样的例子称为模式（*pattern*）。

模式已经从多种不同层次上进行了描述，从处理某个特殊编码问题的低层模式，到组织系统总体结构和架构的高层的模式。术语软件架构（*software architecture*）用于指称如何将系统划分为子系统、各个子系统将是什么角色以及它们将如何彼此相关的高层决策。诸如和用户交互、数据永久存储处理有关的总体策略也可以作为系统架构的一部分来描述。

对于软件架构的详细讨论不属于本书的范围。在本节剩下的部分将描述一种特定的架构，该架构已经过证明，是一种设计使用图形用户界面并提供某种持久存储机制的大多数典型桌面应用的适用方式。

5.3.1 层次架构

用若干个层定义一个系统的架构的思想在软件工程中由来已久，并且已经在众多不同的领域应用。定义多个层的动机是将责任分配到不同的子系统，并在各个层之间定义整齐而简单的接口。在许多方面，这可以看作是对象设计中应用的同一原则的又一个应用，只是用于系统中更大的构件。

在这里将要描述的架构的基本思想之一是清楚地区分在系统中负责维护数据的部分和负责将数据向用户呈现的部分。最早明确表达这些思想的是被称为“模型－视图－控制器（*Model–View–Controller*）”架构，或 MVC，它是为用 Smalltalk 语言编写的程序定义的架构。

通常，在编写面向对象程序时，创建一个类，既保存与某些实体对应的数据，又将数据显示在屏幕上，这往往很吸引人。例如，这种方法看起来使得在数据变化时更新显示比较容易：因为同一个对象负责这两项任务，当它的状态改变时可以直接更新自己的显示以反映新的状态。

但是，当显示由多个不同对象的复杂表示组成时，这种方法不能起作用。一个对象为了合理地显示自己的状态，可能不得不感知到屏幕上其他的一切，因而大大地增加了该类的复杂性。此外，要是用户界面改变了，如果显示数据的责任广泛分布于这些对象，那么系统中所有的类也不得不改变。为了应付这些问题，MVC 架构建议将这些责任交给不同的类，由一个模型（*model*）类来负责维护数据，而由一个视图（*view*）类负责显示数据。

依照这个模式设计系统引起的结果是：系统运行时对象之间传递的消息数目会增加。例如，无论何时要更新显示，视图类在显示之前都必须从模型类获取对象最近的状态。

不过，这种方法带来的好处值得这样做。至少，这可以使对同一数据定义多个或可替换的视图变得非常容易。通过使用不同的视图，同一个应用程序能够支持多种用户界面，例如基于台式电脑的用户界面和基于移动电话的用户界面。如果数据的维护和显示在同一个类中定义，要将二者分离并安全地修改用户界面代码会困难得多。

这种在模型和视图之间进行区分的原则可以应用于系统一级，得到架构中两个分离的层次。维护系统状态和实现应用业务逻辑的类置于*应用层*（*application layer*），而与用户界面有关的类放在*表示层*（*presentation layer*）。

这两个层在如图 5.1 所示。每个层由一个包（*package*）描绘，在图中表示为一个“带标签的文件夹”图标，其中有包的名字。UML 中的包只是对其他模型元素的分组，用于定义模型中的层次结构。一个模型可以分为多个包，每个包又可以包含嵌套的包，只要认为必要或有帮助，可以嵌套到任意层。包的内容可以在图标中显示，但不是必需的。如果没有显示包的内容，则在文件夹图标内写上包的名字。

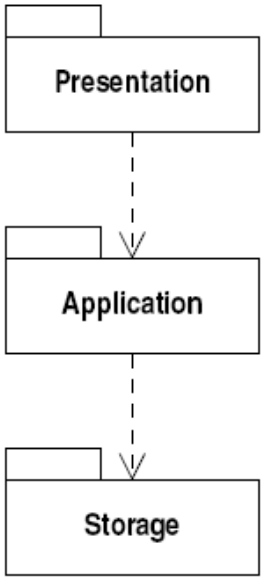


图 5.1 三层架构

图 5.1 还显示了表示层和应用层之间的依赖，说明表示层依赖或使用应用层中定义的类和其他模型元素，但反过来并非如此。这是层次结构的一个典型特征，“高”层可以使用“低”层提供的特征，但低层是独立于高层定义的。

依赖的这种不对称性反映了 MVC 架构的一个关键思想，那就是模型应该独立于视图。这反映了一个观察结果，在许多系统中，基本业务逻辑的改变远不如构成用户界面的代码改变得那么频繁。因此，能够在不影响模型类的情况下更新视图类比较理想，而图 5.1 所示的依赖性反映了这种需要。例如，这种方法使得不修改应用的核心类而开发应用程序的新界面切实可行，譬如说可能是支持基于 web 的或移动访问的界面。

图 5.1 所示的架构中的第三层负责系统中数据的持久存储。这是对象模型很少涉及的系统的一个方面：进行对象设计和用例实现时，好像是所有需要的对象都保存在内存中，而且在需要时立即就能获得。但是，这种假设是不实际的，在所有（即使是最简单的）系统中，也需要某些持久数据存储机制，既是为了保存内存容纳不下的数据，同时也是为了确保数据在会话之间或是譬如系统崩溃之后，能够持久存储。

经验表明，与不该让应用层负责数据的显示的原因相同，让应用层的类额外再负责保证它们所维护数据的持久存储也不是一个好主意。因此，一种常见的系统架构是将这个责任放在单独一层中，即在图 5.1 中的*存储*（*storage*）层。

在三层架构的基础上，可以提出对分析和设计之间的差异的另一种描述。典型地，分析只和应用层中对象的行为与交互有关：通常，应用层对每个系统都是独特的，而分析是一种论证所提议的系统事实上可行的方法。另一方面，设计与架构中所有层次上的对象设计都有关，特别是层与层之间的交互。然而，在大多情况下，很多系统的表示层和存储层的需求是共同的，因而，与分析相比，在更大程度上能够应用以前工作中积累的模式来产生设计。

5.3.2 分析类的构造型

MVC 架构在区分模型类和视图类的同时，还清楚地分出了主要任务是控制复杂交互的对象。这些对象很自然地被称为*控制器*（*controller*）。在原始的 MVC 架构中，控制器负责接收用户输入，将消息转发给模型对象以更新系统的状态，查看对象以保证保持最新的用户界面。因此，在 MVC 对象设计中，各个对象可以归类为模型、视图或控制器对象。

统一过程定义了一种与之相似的对象类别，即*边界*（*boundary*）、*控制*（*control*）和*实体*（*entity*）对象。实体对象，如同 MVC 中的模型对象，负责维护数据，但边界对象和控制对象的特点与视图和控制器略有不同。

边界对象是那些与外部用户交互的对象。它们是用户界面的抽象，负责处理输入和输出。在 MVC 中，用户输入由控制对象检测，但输出处理是视图对象的责任。统一过程中的控制对象更关注控制一个用例中涉及的应用层中的对象的交互，而且不处理输入和输出。

划分边界、控制和实体对象的意图是在分析中作为一种刻画实现中对象作用的方式，同时也提供一种构造实现的标准方式。它们是普通的类，具有附加的非形式化语义来描述它们在系统中的作用，所以在 UML 中用构造型表示。构造型可以写在类图标中，不过这三种类也有专门定义的特殊图标。这些图标可以代替通常的类图标，或类图标中的文字构造型，如图 5.2 所示。

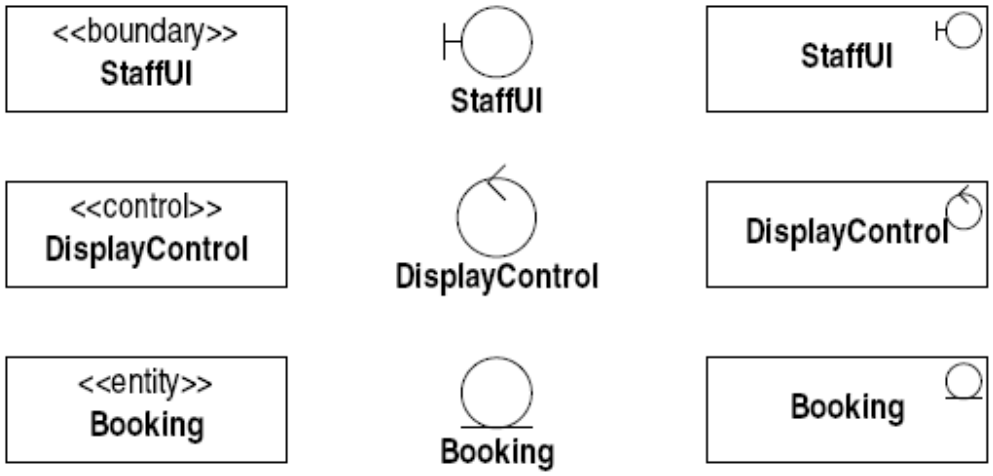


图 5.2 分析类构造型的表示法

5.4 用例实现

在餐馆预约系统中，最简单的用例可能是“显示预约”用例。它对系统也很重要，在系统中，它被其他几个用例包含，并且负责更新用户看到的显示。因此，从这里开始实现预约系统的用例比较合理。基本事件路径中指定的交互非常简单：用户输入要求的日期，系统显示所记录的当天所有的预约进行响应。

5.4.1 系统消息

在用例的执行期间，如下对用户产生的输入建模：包含一个在实现中表示用户的相关参与者的实例，把用户的动作作为参与者实例发送给系统的消息。从外部用户到系统的消息有时称为*系统消息*（*system message*），以区别于系统中的对象产生的内部消息。

因此，“显示预约”用例的实现将包含一个员工（*Staff*）参与者的实例，这是因为任何员工都能够执行这个用例。用户的交互基本上只是一个请求，即显示指定日期的预约。这可以作为一个单独的消息建模，相关的日期作为消息的参数。系统对这个消息的响应是检索请求那一天的预约并显示，然后用例实例结束。

来自用户的消息必须发送给预约系统中的某个类的实例。然而，像图 4.12 这样的领域模型通常并不包括一个适合于接收系统消息的类。整体看来，领域模型中的类表示的是业务实体，而给这样的类增加响应来自用户的消息的责任并不合适。通过这个简单例子，可以看到分析如何导致了对领域模型的增加和修改，也可以看到在整个模型中不同的类充当不同的角色。

决定如何最好地刻画接收系统消息的对象并不是非常直接。统一过程将边界对象定义为接收来自用户的消息的对象，所以，使用边界对象似乎是合理的选择。但是，边界对象属于系统架构中的表示层：如果我们试图分析应用层中对象的行为，使用边界构造型看来是一种误导。

考虑接收系统消息的对象的作用可以启发我们选择另一种方式。一般地，在一个用例中可能涉及多个系统消息，检查用户发送这些消息的次序合理正确、协调系统产生的响应都是必要的。这正是控制对象适合承担的责任，在本章剩余的部分，我们将把接收系统消息的对象描述为控制器。

逻辑上，对每个用例都可能有一个不同的控制对象，但是在简单系统中或许没有必要。如果所有用例使用单独一个控制器，那么可以将它看作是将系统作为一个整体的表示。如果以后发现这个对象大到不可行而且低内聚，那么可以将它分裂为几个单独的控制器。

图 5.3 用一个顺序图说明了“显示预约”用例中的这一个系统消息。该消息由一个表示整个系统的控制器对象接收，并用相应的 UML 构造型来表示。

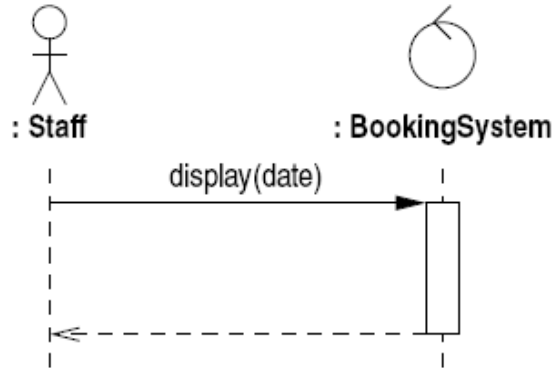


图 5.3 一个系统消息

顺序图和通信图相似，显示了在一个交互中消息是如何在对象之间传递的，但是图的结构稍有不同。顺序图的主要特征是在图中明确地表示出了贯穿交互的时间推移：一个交互从图的顶部开始，时间在图中从顶部流向底部。

在图的最顶部，显示交互开始时存在的相关对象。在这个例子中，有一个表示员工成员的参与者实例，还有一个控制对象。在交互图中，通常将这些对象用表示为角色（*roles*），而不是用实例。实际上，这两种方法没有太大差别。在 UML 1 中，角色和实例的区别是：角色名字不加下划线，实例名字要加下划线；角色如果命名的话，名字前面加斜线“/”。在 UML 2 中，二者的表示法没有区别。

图中出现的每个角色或实例都有一条生命线（*lifeline*），用一条在其图标下延伸的虚线表示。生命线说明了对象存在的时间段。在这个例子中，员工成员和预约系统对象在交互开始就存在，而且在交互期间没有销毁，所以它们的生命线从图的顶部一直延伸到底部。

图 5.3 中的交互包含一个单独的系统消息，请求显示某日的预约，其中日期是作为参数传递的。在顺序图中将消息表示为从一个对象的生命线到另一个对象生命线的箭头。当 BookingSystem 对象接收到“display”消息时，就开始一个新的激活（*activation*），用其生命线上一个拉长的矩形条表示。一个激活相当于一个对象正在处理一个消息的那个时间段。在顺序图中显示激活可以很容易看到在交互中控制流处于何处以及一个对象在处理另一消息的过程中可以怎样发送一个消息。

图 5.3 没有显示如何检索和显示所需预约信息的细节，在激活结束的地方，显示了一个从系统到用户的返回消息。这并不对应于交互中的另外一个不同的消息，而是表示在该点系统完成了用户请求的处理并将控制返回到用户。

5.4.2 存取预约

下一步是细化图 5.3，以说明系统如何检索和显示用户请求的数据。这里涉及两个动作：一是检索相关日期的预约，二是更新显示，用这些预约来替换已有预约，因此需要某些简单的控制以确保这些动作能够以正确的次序发生。控制这个用例流程的责任已经分配给了 BookingSystem 对象，因此要由它来启动这两个动作。

首先，我们必须决定 BookingSystem 对象如何得到所需要的预约。这可以通过向负责维护餐馆全部预约的对象，发送一个请求指定日期的所有预约的消息来实现。然后我们再去设想能够找到所需预约的各种算法。

然而，在设计中还没有一个类负责保持有关系统知道的全部预约。领域模型中的预约类（Booking）负责存储的是单个预约的信息，但是我们需要一种方法在系统知道的所有预约上迭代处理。一般而言，维护单个实体和实体集合的责任最好分给不同的对象。

至今确定的另一个最合适的类就是 BookingSystem 控制器类。然而，这个对象已有的责任是处理系统消息，这和维护一组数据完全不同。如果将这两个责任都分给 BookingSystem 对象，那么有可能创建一个相当不具内聚性的类，所以看来似乎有必要定义一个新类来维护预约集合。

在应用领域中，预约可以看作是餐馆自身的一个属性。因此，一种可能的策略是引入一个表示餐馆的新类，让这个类负责维护到所有预约的链接，并在请求时查找和返回特定的预约。在这个设想下，显示预约的用例的实现可以细化到下一级，如图 5.4 所示。

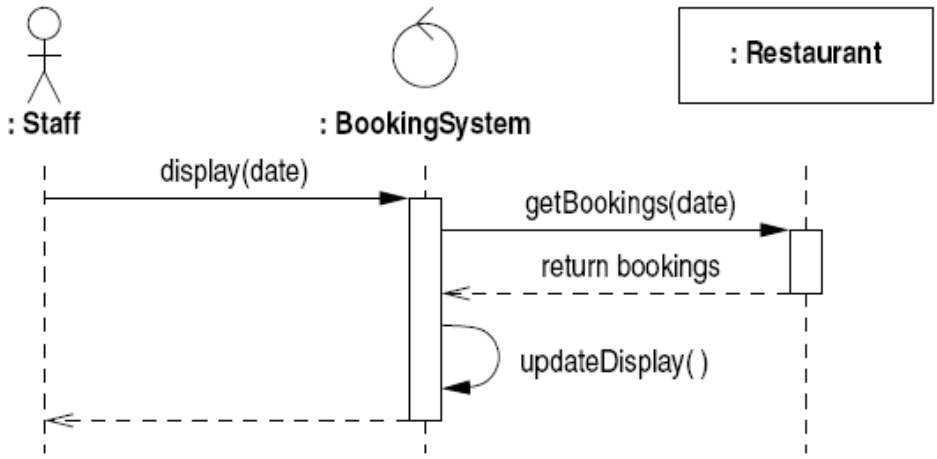


图 5.4 检索预约供显示

图中说明了系统做了些什么来响应来自员工成员的请求。图中显示了由此请求引起的这些消息来自于预约系统的生命线上与系统消息对应的激活，因而表现了过程消息调用的嵌套控制流的特征。返回消息上明确地注出了返回的数据。

接下来，发送一个消息更新当前的显示。根据系统的架构，这个消息应该从 BookingSystem 发送到表示层中的某个类，请求更新显示。图 5.4 中的“updateDisplay”消息对应了一个将此传达到表示层的方法，具体的处理机制将在下一章讨论。只要被显示的信息发生变化，这个方法就可能会在许多不同场合被调用。

5.4.3 检索预约细节

剩下要决定的问题是 Restaurant 对象如何识别返回的预约。逻辑上，需要获得每个预约对象的日期，并返回与来自参与者的消息中提供的日期相匹配的预约。这个交互如图 5.5 所示，“getDate”消息前面的星号是重数，表示这个消息在此交互中一般会多次发送给不同

的预约对象。这是 UML 1 中表示重复消息的方法，有些支持 UML 2 的建模工具仍然支持。在 UML 2 中，用交互片段来表示重复消息，我们在后续相关章节中讨论。

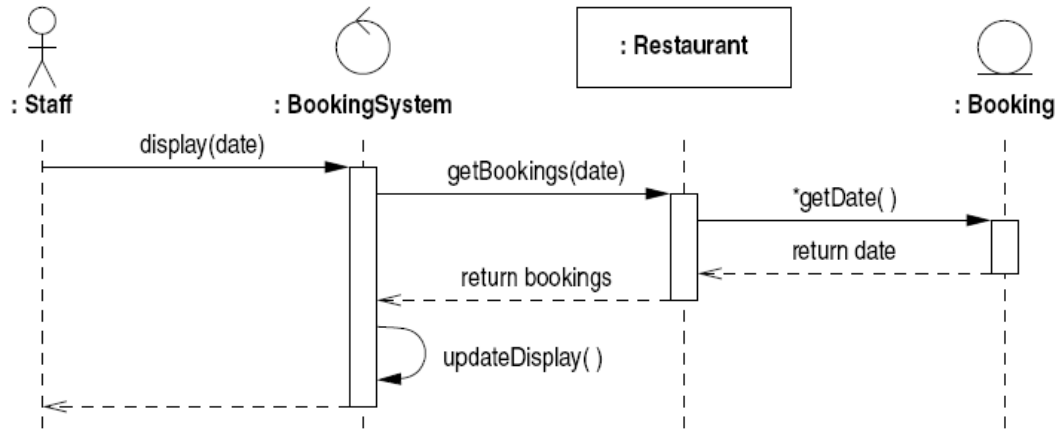


图 5.5 显示预约：基本事件路径的实现

图 5.5 从分析的层面上给出了“显示预约”用例的基本事件路径的一个完整实现。其中对于交互的每件事都没有详细说明，例如，没有指定用于返回检索到的预约的集合的数据结构，也没有指定在预约对象集合上的迭代操作进行的方式。这个细节层面会留到详细设计阶段。然而，这个图描绘了交互的总体形式，并再次保证了至今确定的类实际上能够支持这个用例。

5.4.4 细化领域模型

“显示预约”用例的实现的过程确定了两个新类和许多在类的实例之间传递的消息。可以把这些信息加入到领域模型，因而开始了将领域模型变成一个更全面的类图的细化过程，这个类图文档化了上面的分析活动的结果。图 5.6 是分析类图的一部分，其中包括图 5.5 中的新类。

图 5.6 中显示了两个新关联。第一个是从餐馆类到预约类，反映了我们指定餐馆类负责登记系统已知的所有预约的细节。由于预约是由另一个类的实例表示的，所以餐馆能够掌握这些预约的唯一方法是存储到每个预约的链接，如关联所指明的那样。

但是，还有另外一个责任，即记录当前在屏幕上显示的是哪些预约。这些预约是在图 5.5 中返回给边界对象的预约。如果每次它们显示后都没有保存，那么无论何时要更新显示，都不得不再次从餐馆对象检索当前的预约。可能有很多原因导致需要更新，例如在某个其他应用程序重写窗口之后进行刷新，所以这种方法可能会涉及很多不必要的处理。

图 5.6 采用的是另一种设计，将记住哪些预约和当前日期相关的责任交给预约系统 BookingSystem 类。BookingSystem 和 Booking 类之间的关联展现了这一信息：和 Restaurant 餐馆对象一样，预约系统 BookingSystem 通过维护一组到相关预约的链接来履行自己的责任。与此相关，预约系统 BookingSystem 类也记录了当前显示的日期 date，作为一个属性。

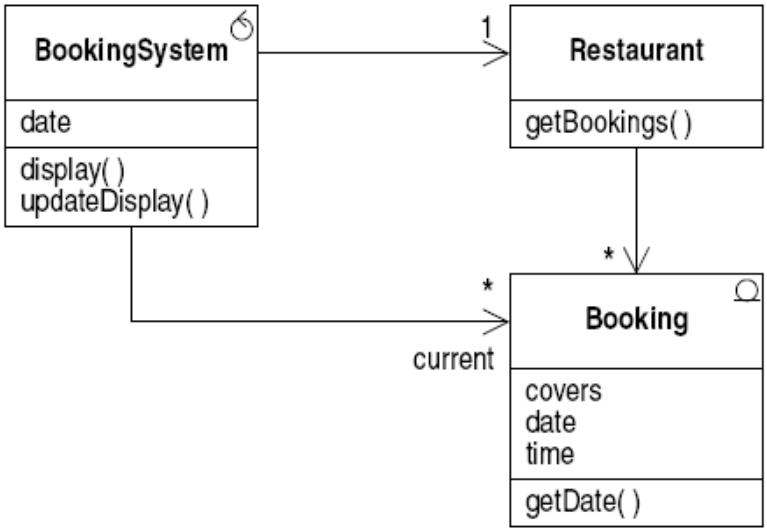


图 5.6 分析模型中的一些类

在顺序图中出现的消息也要作为各个类的操作出现在类图中。发送一个消息的目的通常是为了调用接收该消息的对象中的一个方法，这通过在接收消息的类中包含一个操作表示。为了清晰起见，操作的命名和调用它的消息相同。

类图中的关联除了维护与这些关系有关的信息外，还以对象之间的链接的形式定义了“通信信道”，沿着信道可以发送消息。因此，一种有用的一致性检查就是确认：只要从一个对象向另一对象发送消息，那么，类图中就记录有一个关联，可以作为该消息的通信信道。

领域模型可以用文档化对现实世界应用而言很重要的关系，但是这些关系并不是用来支持设计中的消息传递的。在很多情况下不需要实现这样的关联，或者只需要在消息发送的方向上给以支持。为了文档化这一事实，作为后续实现的指导，在图 5.6 中的关联上增加了消息传递方向的导航性注解。

5.5 记录新预约

现在，可以对用例模型中的其他用例重复进行上节阐述的用例实现过程，从而产生预约系统的一个完整的分析模型。在显示预约之后，下一个最基本的任务很可能就是创建新预约，因此在本节将考虑这个用例。

如前所述，我们不想在这个阶段对用户界面的细节建模。我们将假定，创建一个新预约所需的详细资料是由用户界面的某些适当的元素收集的，例如一个对话框，而用例的逻辑结构可以由一个来自用户的单个系统消息表示，该消息请求创建一个新预约，并将需要的数据作为参数传递。

和上个用例一样，这个系统消息由相同的控制器对象即预约系统 BookingSystem 实例接收。我们现在必须决定应该将创建新预约对象的责任分配给什么对象。仅有的两个可行的选择是 BookingSystem 对象和餐馆 Restaurant 对象。因为 Restaurant 对象已经具有维护系统已

知的全部预约对象集合的责任，所以看来将创建新预约的责任也指定给它能够维持高度的内聚性。

这个决策反映在图 5.7 的顺序图中，其中将新预约的详细信息（details）传递给餐馆对象，由餐馆对象实际地创建新的预约。和显示预约的情况一样，发送“updateDisplay”消息以便通知表示层系统状态已经改变，需要更新该视图的相关部分。

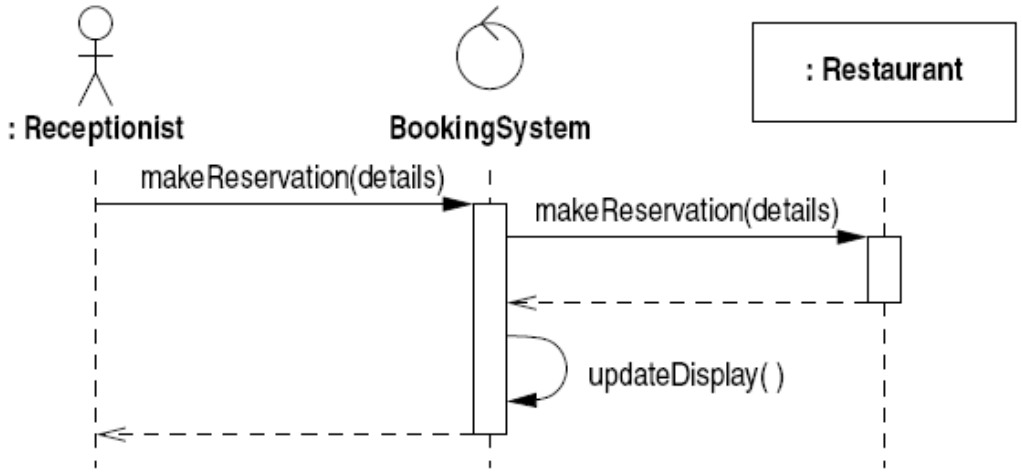


图 5.7 记录预约：初始交互

注意，在分析中，不详细说明诸如参数和数据类型这样的细节是完全可以接受的。在某个阶段，可能需要决定如何表示与新预约相关的数据以及伴随着系统消息传递什么参数。然而，在分析中，主要关注的是论证能够创建一个可行的对象设计以支持用例，这一点一旦做到了，某些细节可以推迟到稍后的阶段。

5.5.1 创建新对象

为了完成这个用例的实现，我们需要考虑这个控制器对象如何响应“makeReservation”消息。图 5.8 给出了一个顺序图说明创建新预定的过程。

在创建一个新预定 Reservation 对象之前，必需锁定预定相关的餐桌 Table 和顾客 Customer 对象。根据领域模型，每个 Reservation 对象被链接到恰好一个 Table 对象和恰好一个 Customer 对象。因此，创建一个未链接到这些对象的预定将是实现错误，因为系统状态将和领域模型中的重数说明不一致。

我们假定从用户传来的数据中包含有这些对象的文本的标识符，例如餐桌号码以及顾客的姓名和电话号码。这些数据应该由用户在指定新预定的详细信息时输入。但是，在创建预定之前，我们需要定位由这些数据标识的对象，以便在创建新预定时可以获得适当的对象引用。

在此，我们必须决定：提供对餐桌和顾客对象进行访问的责任应该由哪个对象负责。如同预约一样，将这些实体描述为是餐馆的属性很自然，因此我们可以暂时将这个责任指定给餐馆。这里很明显有一个潜在的危险：餐馆对象可能拥有相当不内聚的一组责任，但是目前

似乎几乎没有理由将这些责任分配到不同的类中。然而，作为一个未来开发中潜在的问题，应该记得这一点。

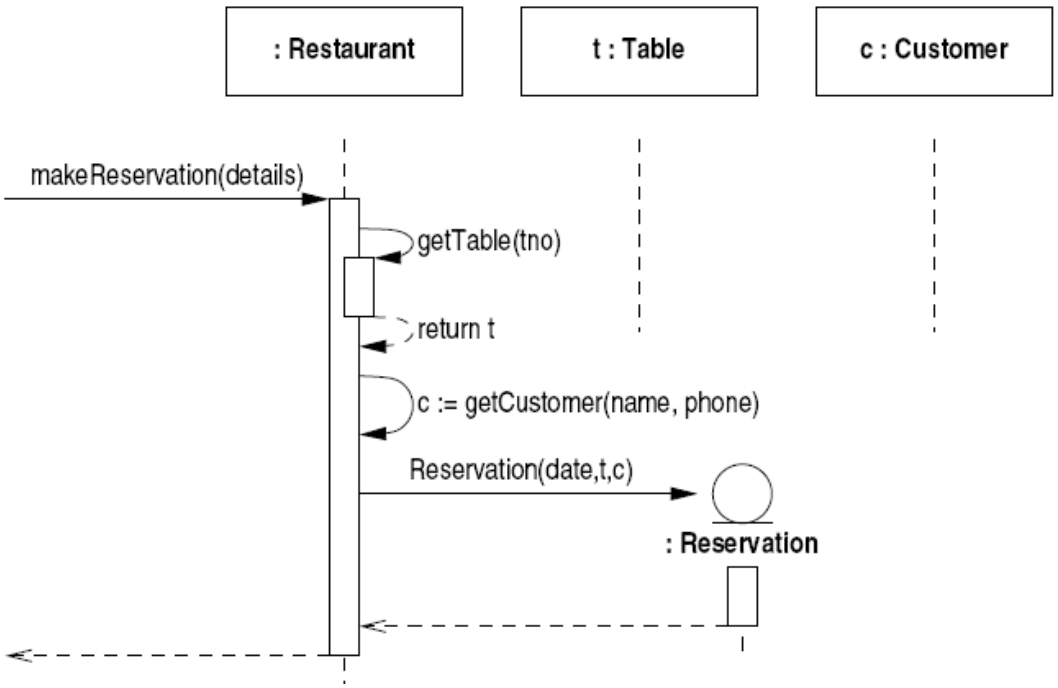


图 5.8 创建一个新预定

“getTable”和“getCustomer”消息分别返回的是与作为参数传递给它们的数据对应的餐桌和顾客对象。由这些消息返回的对象如图 5.8 中所示，为了避免混淆，它们的生命线被截短了。我们假设，参数“tno”、“name”以及“phone”可以从伴随“makeReservation”消息传递的未详细说明的“details”中提取。

图 5.8 给出了表示对象调用自己的方法的两种不同方式。“getTable”消息明确地表明为这个新消息创建了一个嵌套的激活，并且当它结束时返回数据。第二个消息“getCustomer”没有一起显示激活，而是消息自身包括了返回值，使用的是第 2 章介绍的表示法。

在交互的开始，并不存在新预定，所以它没有出现在顺序图的顶部。在顺序图中，新对象显示在相应于它们被创建时的点，下面延伸的是生命线。引起一个新对象创建的消息指向对象本身，而不是指向其生命线。这些消息通常称为构造器（*constructors*），在 UML 中可以用可选的“create”构造型表示，以突出它们与通常的消息不同。在新对象正下方的激活对应于构造函数的执行，而且非常可能在这里从新创建的对象发出消息。图 5.8 假定构造器方法和它们的类的名字相同，这是许多语言遵循的惯例。

5.5.2 记录未预约顾客的预约

未预约顾客的预约在一个顾客没有提前预定而进入餐馆用餐时创建。它们在系统中记录的方式和预定相同，但是因为它们不和顾客关联，所以在创建时需要较少的信息。“记录未预约（顾客）”用例的实现留作一个习题。

5.6 取消预约

与迄今所实现的用例相比，取消一个预约的用例结构更复杂，因为用户对用例的参与由多个与系统的交互组成。如同在第 4 章描述的，要取消一个预约，用户必须首先选择要取消的预约，然后取消它，并在最后系统提示时确认取消。这个事件路径的一种可能的实现的顺序图表示在图 5.9 中给出。

该用例分解为两个独立的部分。首先，选择需要的预约。这可以通过很多种方法完成：也许是用户点击屏幕上的预约矩形框，或者输入标识该预约的一些数据。这在图 5.9 中是通过一个来自用户的“selectBooking”消息实现的。在分析层面，由用户提供的识别预约的信息的确切特性没有详细说明，图中只是非正式地指出提供了足够的信息来确定所需的预约。

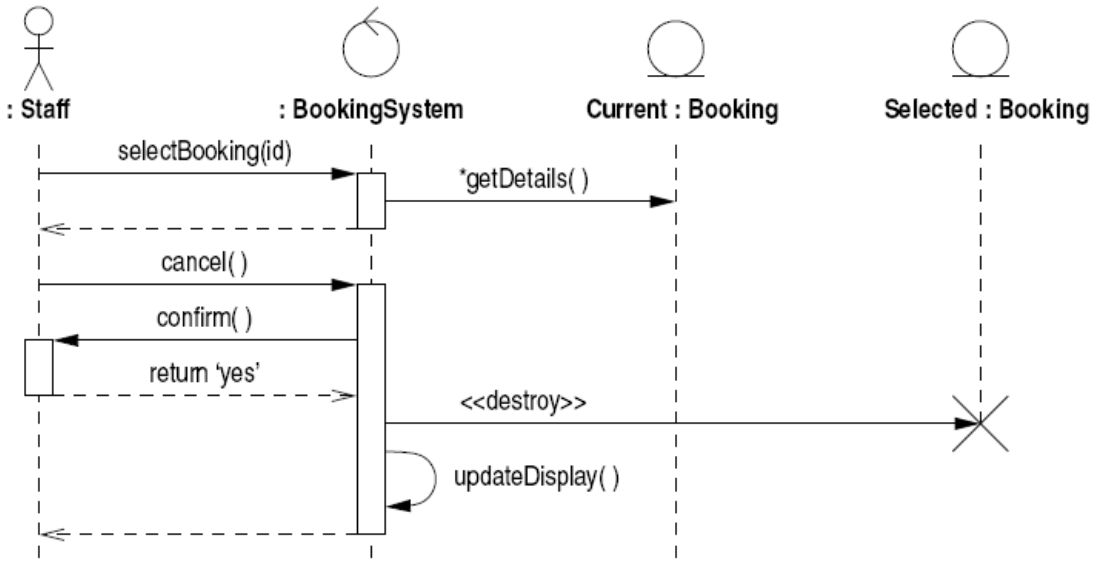


图 5.9 取消一个预约

为了查找需要的预约对象，必须检查当前显示的每一个预约。由于预约系统 BookingSystem 对象已经有了维护这组预约的责任，所以能够直接在这些预约上重复。用户界面检查所有当前预约的详细信息以查看哪一个和选择的标准匹配。在图 5.9 中这由“getDetails”消息表示。消息前面的重数表示它可以被发送零或多次，取决于当前显示的预约有多少。

在基本事件路径中，我们假定一个预约被成功选择。因为这个预约在用例的稍后阶段被涉及，所以独立显示于图的顶部。角色名（*Role names*）用来区分选定的预约和当前的预约。角色名不是为一个单独的对象命名，而是描述在交互中对象能够充当的角色。每次执行这个用例时，选定的预约可以是不同的对象。

一旦选择了需要的预约，可以在屏幕上以某种方式突出地显示出来，用户就调用取消操作。得到用户确认的过程可以通过一个从边界对象回到该用户的消息建模：这可能对应于比如说显示一个确认对话框。此时强制用户以某种方式响应这个消息，返回消息则表示用户的响应。

一旦这个消息被接收，用户界面对象就删除选定的预约，并在用例结束之前更新显示。对象删除由一个 “destroy”构造型的消息表示：当对象接收到这样一个消息时，它的生命线就由一个大大的“X”终止，表示对象的销毁。注意，对象销毁采取的形式可能因编程语言而不同，而且在具有自动无用单元收集的语言中，可能不需要明确的方法调用来删除一个对象。

5.6.1 细化领域模型

在这个交互中暗含着一个新责任，即记住哪一个是所选预约。如果这没有在某个地方记录，那么预约系统对象就会不知道要销毁的是哪个预约。图 5.9 中的顺序图假定这个责任分配给了预约系统对象：因为预约系统对象已经负责维护当前显示的预约的集合，所以这与它已有的责任结合得很紧密。

这个责任需要利用预约系统和预约类之间的一个附加关联，如图 5.10 所示，反映在逐步演化的类图中。注意这些关联的不同重数：当前可以显示多个预约，但至多只能选择其中一个。这些关联还对系统施加了更进一步的约束，也就是选中的预约必须是当前显示的预约中的一个。

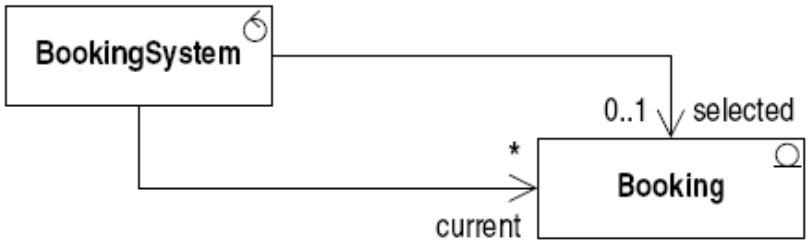


图 5.10 记录选中预约

5.7 更新预约

“记录到达”用例的结构和“取消预约”用例非常类似：用户首先选择需要的预约，接着向系统表明顾客已经到达。这个事件路径的实现如图 5.11 所示。注意选中的预约明确地作为一个预定表示，因为记录一个未预约顾客的到来是没有意义的。在基本事件路径中，我们假定用户实际上是选择了一个预定。

我们假设餐馆想要记录顾客预定的到达时间，因而必须决定什么类来负责存储这个信息。对一个未预约顾客，该信息全然没有意义：未预约顾客的“到达时间”实际上和他们到餐馆的时间相同。这表明最适当的责任分配应该是将到达时间作为预定类（Reservation）的一个属性。

然而，用户很可能选择了一个 Walk-in 预约而不是一个预定，然后试图记录到达时间。如果未预约类（WalkIn）不支持“setArrivalTime”操作，但是发给了它这个消息，那么将

出现一个运行时错误。为了防止这种情况，我们必须或者保证这个消息不发送给未预约对象，或者保证未预约类 Booking 支持这个消息。

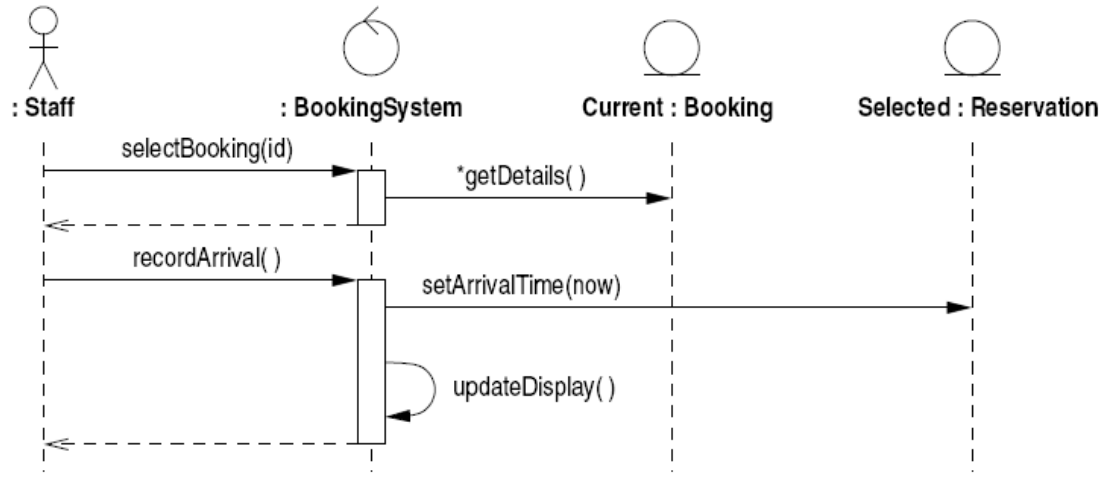


图 5.11 记录一个顾客的到达

第一种可能性需要运行时检查以确定选中的预约属于什么子类。只要可能，这样的运行时检查最好能够避免，因为会导致难以维护的复杂代码。一个更好的方法是为 Booking 继承层次中的所有类都提供“setArrivalTime”操作，但是将记录到达时间的特定责任只赋给那些此举有意义的类。阐明这种方法的类图片段在图 5.12 中给出。

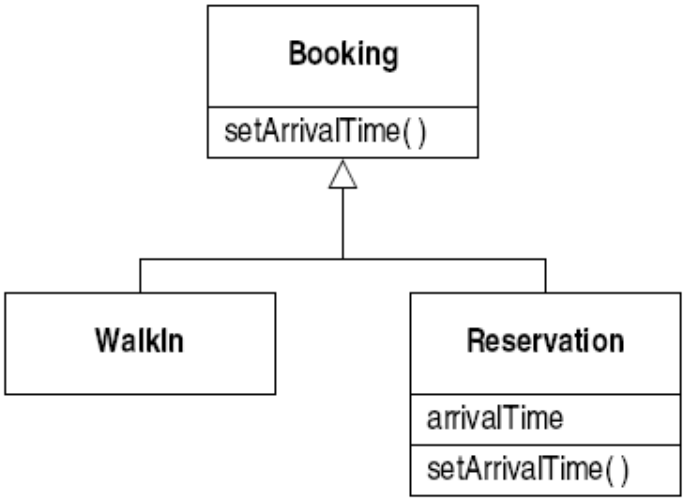


图 5.12 在类层次中分配责任

为了使之更具体一些，设想一种办法在 Booking 类中提供了“setArrivalTime”操作的缺省（预设）实现，以某种方式响应对该操作的不适当调用。这个操作可以在 Reservation 类中被覆盖，以便为 Reservation 类型对象记录到达时间。通过这种方式，为不同类型的预约选择合适的行为就由动态绑定机制自动进行处理，而不是由程序员来手工编码。

5.7.1 调换餐桌

从表面上看，调换餐桌用例的结构似乎比至今为止考虑的那些用例可能更复杂。如果我们设想调换是通过拖放一个预约完成的，用户将产生一系列相应于各种鼠标事件的消息，这看来应该反映在用例的实现中。

然而，如同之前指出的，让应用层的类承担像鼠标这样的特殊用户界面设备中的细节的义务并不是一个好的想法。在分析层面，这个用例最好用一个单一的系统消息建模，该消息提供一个新餐桌的标识符来与当前选中的预约相关联。这个方法保持了该用例的基本功能，又可以为后面用户界面的设计留有灵活性。

在这个设想下，“调换餐桌”用例的结构就非常类似于“记录到达”，该用例的实现留作一个习题。

5.8 完成分析模型

本章只是详细考虑了每个用例的基本事件路径。对可选和例外事件路径可以应用完全相同的原则来实现，给模型增加一些新的东西。一般而言，将每个事件路径的实现显示在一个不同的顺序图中会更清楚。UML 的确定义了表示顺序图中的可选控制流的符号，如后续交互图一章讨论的，但是一般将可选情况表示在单独的图中会更清晰。

图 5.13 显示了系统的类图，包括来源于用例实现过程中的信息和决策。这个类图也包括来自领域模型的信息，例如顾客、餐桌和预约类之间的关系以及不能重复预约的约束等。在类的实例之间传递消息的地方，关联在消息发送的方向是可导航的，其他关联则目前没有导航注解。

5.9 小结

- 分析可以定义为用对象模型表示应用领域和系统需求的活动。
- 基本的分析技术是产生用例实现，用例实现展现了如何通过一组交互的对象来实现用例指定的功能。
- 实现文档可以用 UML 定义的交互图记录。
- 产生用例实现将促使考虑领域模型的改变，领域模型将进化成为一个更详细的分析类模型。
- 对象设计的一个核心隐喻是让对象对系统中数据和操作的一个子集承担责任。
- 一个对象的责任应该经过选择，使得对象是内聚的：对象的责任应该以某种方式相关，通常是功能相关。
- 统一过程将架构描述作为分析产品之一。一种广泛使用的架构方式是将系统组织为若干个层，例如表示层、应用层和存储层。

- 可以给系统中的对象指派多个角色，以使系统的组织清晰明了。UML 定义了边界、控制和实体对象类构造型。
- 在实现中用户交互可以用由控制对象接收的系统消息表示。可以是每个用例有一个控制对象，或者是一个控制对象表示整个系统。

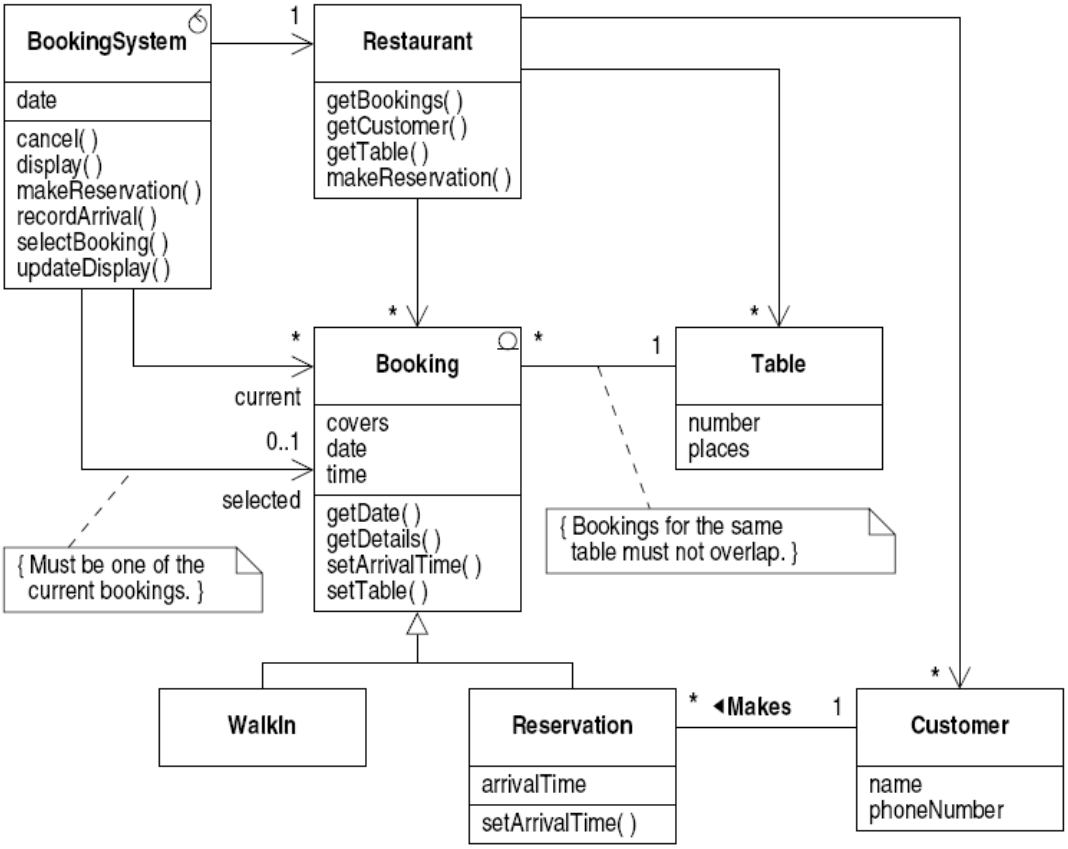


图 5.13 餐馆预约系统的分析类模型

5.10 习题

（1）图 5.4 中的餐馆 Restaurant 对象没有用分析类构造型描述，如果有的话，哪个构造型适合应用于这个对象？

（2）图 5.6 中，在预约系统 BookingSystem 类中的日期 date 属性和用户界面通过“current”关联链接到的预约中的日期属性之间应该存在什么关系？将这个关系作为一个约束加入到图中。

（3）假如有人提议删除图 5.6 中 BookingSystem 类中的 date 属性，理由是当前日期能够从由“current”关联链接的预约之一取得。对这样的建议你如何回应？

（4）为“显示预约”用例中用户提供了无效日期的异常事件路径产生一个实现。

（5）合并图 5.7 和 5.8，产生单独一个顺序图实现“记录预约” 用例。

（6）创建一个顺序图实现“记录未预约”用例。这和在上一个习题中产生的用例图有什么不同？

（7）考虑图 5.8 中的“getCustomer()”消息，如果提供的关于一个顾客的详细信息是餐馆已知的，应该返回表示那个顾客的对象。如果不是，应该创建一个新的顾客对象并返回。用不同的顺序图详细描述这两种情况。

（8）产生一个顺序图显示“调换餐桌”用例的基本事件路径的实现。假定餐桌号码作为系统消息“transfer”的一个参数提供，并且在你的图中显示如何识别这个号码对应的餐桌。

（9）为图 5.13 中的类的操作在适当的地方增加参数和返回类型的详细信息。

（10）为你在习题 4.9 的答案中定义的允许预约信息的常规编辑的用例产生实现。

（11）更新“记录预约”用例的实现，使之反映你在习题 4.10 的解答中进行的修改，即允许预约在有限程度上超过餐桌的座位数。

（12） 为习题 4.12 中描述的新功能产生实现，即允许改变预约的时间长短。

（13）为你在习题 4.13 的解答中描述的用例提供实现，指定系统如何自动为新预约分配餐桌。

5.11 实践题

以第 4 章实践题作业中得到的用例模型和领域类图为基础，进行用例实现和分析对象建模，并给出系统架构的描述。

第 6 章 餐馆系统：设计

第 5 章的用例实现阐明了如何用应用层中的对象实现系统的业务功能。设计最重要的任务就是将应用层的模型扩展到整个系统。本章讨论一些处理输入输出和持久存储的基本策略，细化分析类图，使之包含关于数据类型、消息参数等更丰富的信息。在设计阶段终结前，我们的目的是要对系统有足够详细的理解，使实现可以开始。

为了设计系统的这些方面，我们需要对系统的软硬件环境做出一些决策。在接下来的这两章，我们假定，餐馆预约系统要作为一个单用户的桌面应用系统实现。我们将它设计为一个 Java 应用程序，其用户界面是基于窗口的。对于持久性，我们假定使用关系数据库存储有关预约和顾客等持久数据。

6.1 接收用户输入

在第 5 章，来自用户的系统消息在顺序图中是作为指向一个代表预约系统的控制对象来显现的。但是，预约系统对象是一个应用层的对象，所以实际上消息并不会直接发送到这个对象。必须有某个表示层的对象，它的责任是接收用户的输入并转发给控制对象。

表示层的这个接收用户输入的对象可以很合理地描述为一个边界对象。它表示呈现给一个特定参与者的用户界面。就预约系统来说，我们假定所有用户使用相同的用户界面，因此将这个类命名为“StaffUI”。

我们假设，为了执行“显示预约”用例，用户首先要选择一个菜单选项。这引起一个对话框出现，在对话框中，用户输入需要的日期，然后单击“OK”按钮将请求提交给系统。通常，不值得对用户和标准用户界面构件（如菜单和对话框）交互的细节建模。用户界面框架提供了可复用的类来实现这些构件，而这些如何进行的细节可以留到实现时考虑。

重要的是在用户单击“OK”按钮以提交在对话框中输入的数据时所接收的消息。图 6.1 中，显示了这个边界对象 “StaffUI”，它接收这个消息，然后将处理这个消息的责任委派给先前确定的应用层中的控制对象。这是用通信图表示的，所以可以表明代表不同层的包，以阐明系统的架构。这也说明，在 UML 中，包是一个相当弱的概念，链接和消息可以轻易地跨越包的边界。

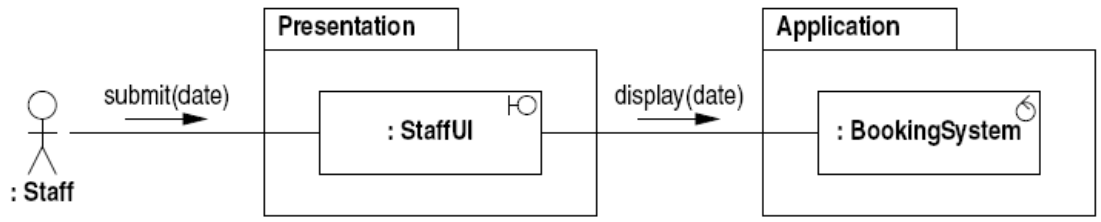


图 6.1　处理系统消息

在用户可以通过标准动作（例如由鼠标产生消息）与系统交互的情况下，经常需要由系统来处理各个用户界面事件。在这些情况下，边界对象的作用更加重要。

例如，有些用例要求用户选择一个当前显示的预约。选择预约很自然的一种方法是用户用鼠标单击该预约，这种方法支持直接操纵所显示的预约目标。用户界面将检测到这个事件，并将其作为一个“selectBooking”消息传递给预约系统，如图 6.2 所示。

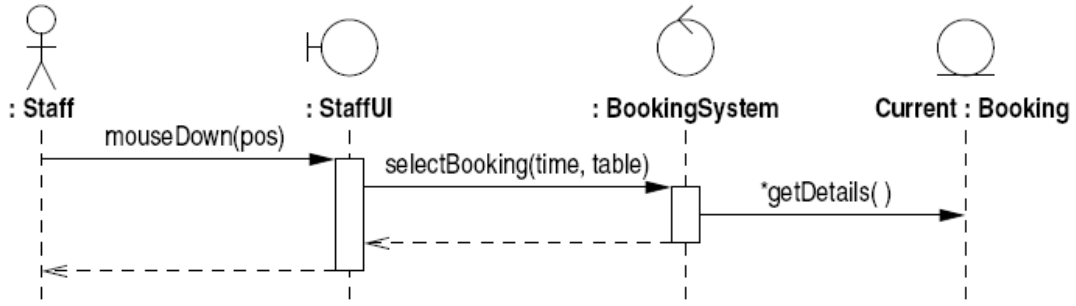


图 6.2 选择预约

与分析模型相比，图 6.2 中消息的参数信息更详细。鼠标消息的参数代表按下鼠标键的位置，以屏幕坐标表示。然而，应用层目前不需要知道任何关于界面的细节，所以在发送给预约系统控制器对象的消息上，用餐桌号和时间代替了位置，这二者将使得预约系统能够确定一个唯一的预约。

这就将鼠标坐标转化为对应用有意义的信息的责任交给了用户界面对象。这是合理的决定，因为用户界面对象也负责产生显示输出，为此，它肯定要将显示的预约的餐桌号和时间映射到相应的屏幕坐标，因此应该能够进行反向映射。

有些交互包含用户产生的多个事件。例如，我们会想到用户能够将预约从一个餐桌移动到另一个餐桌，方法是将预约的矩形从屏幕上的一个位置拖到另一个位置。这个交互将涉及用户产生的多个事件：开始是一个“mouse down”消息选择所需的预定，接着是若干“mouse move”消息，最后是一个“mouse up”消息，表明预约已经放在它最终的位置。

图 6.3 说明了这个交互，图中显示“transfer”消息只是在预约的新位置松开鼠标键的时候才被发送给预约系统。这表明，边界对象检测到的用户产生的事件和发送给控制器的系统消息之间的对应不一定必须是一对一的。这还暗示了在边界对象中一定嵌入了某些控制元素，它必须记住与用户的交互的当前状态，以及何时发送一个系统消息。

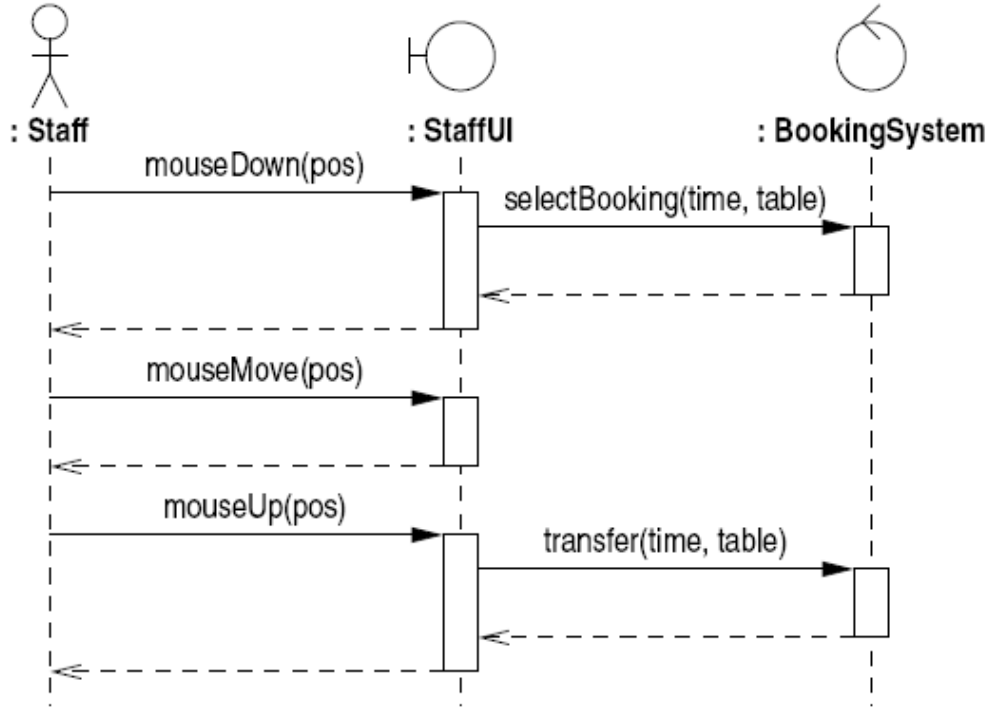


图 6.3 将预约调换到新餐桌

如前所述，这个交互假定，用户界面对象负责在屏幕坐标和对应的应用数据值之间进行转转换。尽管这个用例叫做“调换餐桌”，但是这个顺序图暗含着不需要额外代价就可以实现修改预约时间的附加操作。事实上，只实现调换餐桌可能更困难，因为这时必须检查用户将预约放在一张新餐桌但却是在不同时间段的这种例外情况。有时设计层面的考虑使得回到最初的用例模型并进行修改也是合理的，比如这个例子。对用例模型的修改留作习题。

6.2 产生输出

“StaffUI”类承担两个不同的角色：作为边界类，它接收来自用户的消息并将消息转发给控制器类；但是从第 5 章所描述的 MVC 架构的意义来说，它还充当着视图类的角色。视图类的基本责任是将应用数据或模型呈现给用户，或者换句话说，是显示系统的输出。

一般而言，对输出机制的要求是，只要应用数据的状态改变了，屏幕上对该数据的表示就要更新，使用户所看到的和系统状态是一致的。因此，对设计至关紧要的是借助某些方法让视图知道模型中的变化。

一种常见而且简单的实现方法是由视图类对应用类进行定期的查询，以发现是否有什么变化。这种技术称为轮询（*polling*），这个词也许是由民意调查而来的比拟，即提一些问题来发现人们对某些问题的看法。轮询要求表示层的类调用应用层的类，所以很适合为这个应用程序选择的层次结构。

但是，使用轮询存在很多问题。例如，从浪费的处理时间来看，可能代价高昂。采用任何合理的轮询频率，都可能在一次查询和另一次查询之间系统状态并没有改变，因而，很多轮询活动都是徒劳的。更严重的是，如果模型中有大量的数据，那么检查是否有什么变化，以及是否需要因而更新显示，是相当费力的任务。

为了避免这些问题，看来有一种更好的解决办法，即只要应用有什么变化时，由应用类来通知视图类，那么对视图的更新只是在需要时才发生。从对象设计的角度看，这是一种更好的解决方案，因为它把激发视图更新的责任放在了知道刚好什么时候需要更新的类中。

然而，在层次架构的约束下如何实施这种想法并不明显。由 MVC 架构演变的两层结构有一个基本原则：应用层应该独立于表示层。如果是这样，应用层中的类如何能够通知表示层的类需要进行更新呢？

6.2.1 应用设计模式

这是设计者面临的情况中的典型问题，对常见的设计问题的大量工作已经作为设计模式记录在解决方案中。在这种情况下，观察者（*Observer*）模式提供了一种合适的解决方法，该模式在 Gamma、Helm、Johnson 和 Vlissides 等人 1995 年所著的《设计模式》一书中定义。该模式的定义声称这个模式适用于：

1. “一个对象的变化需要改变其他对象，并且你不知道有多少对象需要改变的时候。”
2. 一个对象应该能够通知其他对象，而无需设想那些对象是谁的时候。”

目前的情况包含了这两个条件的元素：我们想要视图对象随着应用对象的变化而改变，并且我们想让应用对象能够通知视图对象这个变化，但不依赖视图对象。这个模式的定义相当抽象，本节描述的只是它的应用。

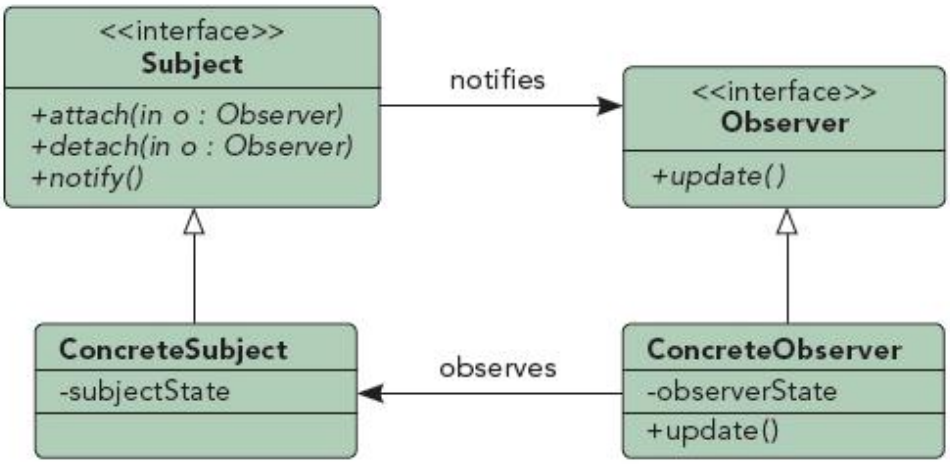


图 6.4 Observer 模式的类图

观察者模式对于在餐馆预约系统中显示更新的应用如图 6.5 所示，代表应用中不同层的包在图中也表示了出来。注意，类之间的跨包边界的关系和两个层之间的依赖性的方向是一致的。

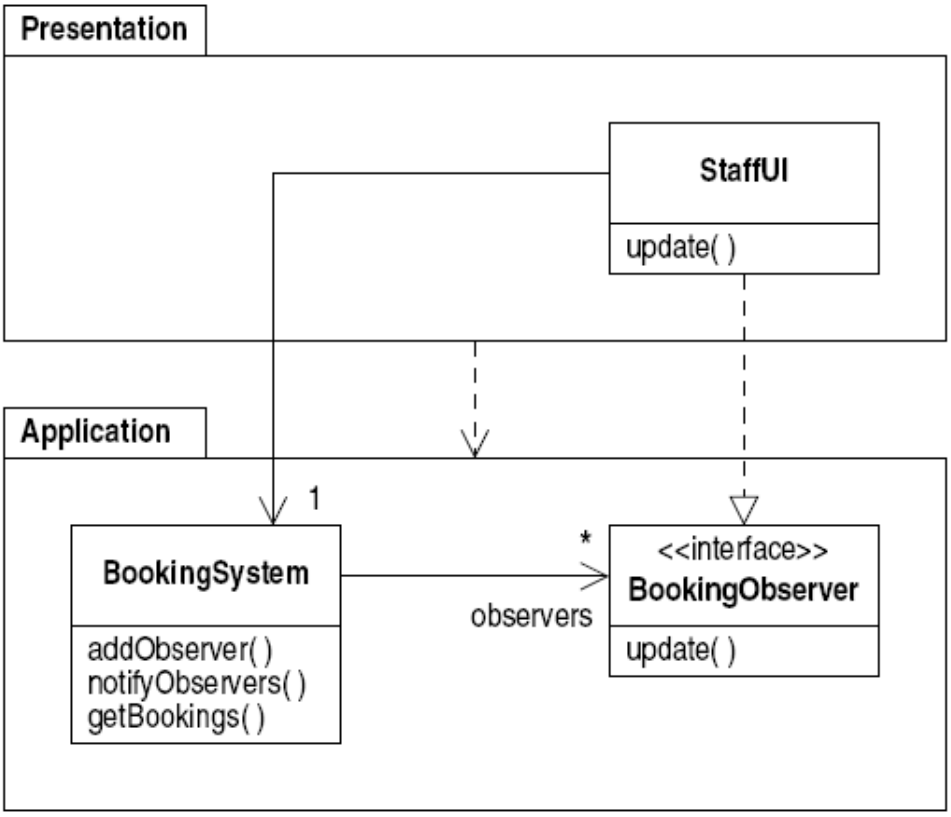


图 6.5 应用观察者模式

和许多模式一样，问题的解决依赖于多态性和动态绑定。在应用层定义一个接口BookingObserver，希望接到模型相关变化的通知的类必须实现该接口。这样的类称为观察者。这个接口非常简单，包含单独一个操作，只要显示需要更新时就调用这个操作。

预约系统 BookingSystem 类维护一组对观察者的引用，当它的状态改变时，就向每个观察者发送“update”消息。观察者可以是无论什么类，但是预约系统只通过应用层定义的 BookingObserver 接口去访问观察者，如此就保持了系统的层次结构。

表示层的“StaffUI”类实现了“BookingObserver”接口。在 UML 中，接口的实现是用带空心箭头的虚线表示的，如图 6.5 所示。在系统开始工作时，通过“addObserver”操作向预约系统的观察者列表中加入一个“StaffUI”的实例。这样，由于动态绑定，“StaffUI”对象就会收到发给观察者列表的“update”消息。

一旦“StaffUI”对象收到了“updateDisplay”消息，它就需要查明什么发生了改变。最初，我们假设用一种非常简单的方法，边界对象只是从预约系统请求一个要显示的所有预约的列表，然后彻底刷新显示。如果结果证明这种初步的方法存在性能问题，在以后可以再改进。

图 6.6 说明在实际更新显示时发生的交互。一般地，会存在多个观察者:“BookingSystem”类中的“notifyObservers”操作只是给每个观察者发送“update”消息，它代替了像图 5.4 中的“updateDisplay”消息。为了简单起见，在图 6.6 中省略了这个消息。

与图 6.5 中的对象图相比，在图 6.6 中，似乎预约系统和用户界面对象在互相直接通信，因而，应用对象依赖表示层中的对象。然而，尽管对象在通信，但这只是由于用户界面对象实现了图 6.5 中所示的观察者接口。在图 6.6 中，通过给边界对象加一个角色名强调了这一点。

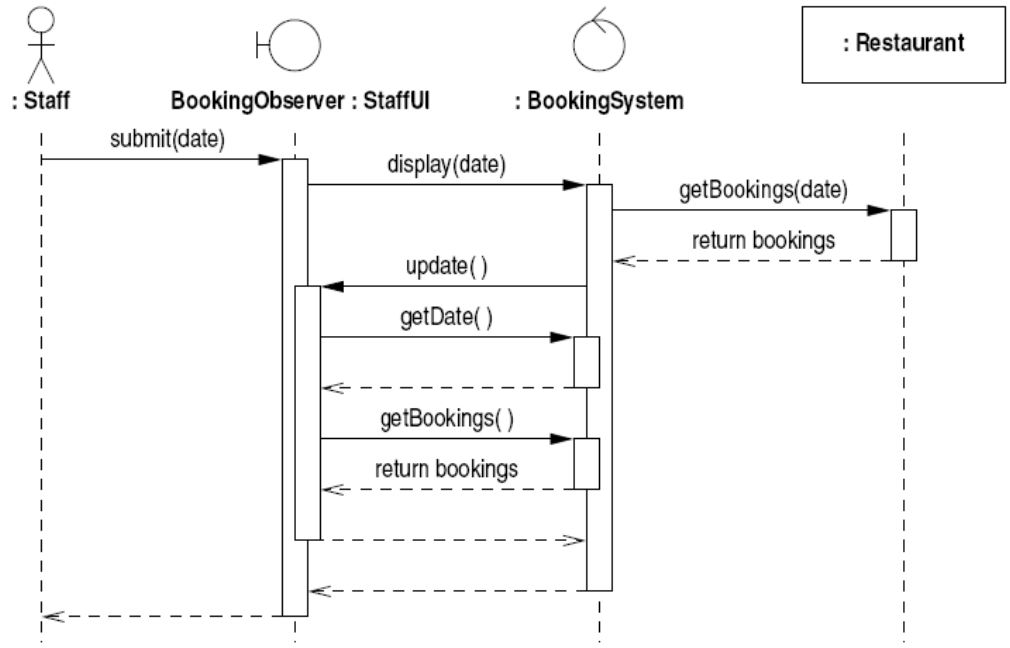


图 6.6 显示预约：设计视图

在图 6.6 的环境中，重申一下各种对象的责任可能是值得的：深刻了解这些可以容易地理解为什么这些消息是以这样的方式来交互。用户请求要显示特定某一天的预约，这个消息发给了预约系统对象。预约系统只有记录当前正在显示的那天的有关信息的责任：假定用户请求的是不同的一个日期，那么首先要从餐馆对象得到这个日期的信息，餐馆对象的责任是维护系统所知的全部预约的信息。

用户界面对象的责任只是显示一组预约，因而它必须向预约系统请求要显示的预约集合。我们没有假设用户界面记得这些预约，尽管在必要时可能在以后会进行性能的改进，让它缓存关于预约的一些信息。

6.3 持久数据存储

绝大多数的软件系统都需要某种方式存储持久数据。这个术语是指以某种方式长时间或永久地存储数据，使数据在系统关闭时不会丢失，在需要的时候可以重新装入。在餐馆预约系统的情况中，这显然是一个系统需求，例如，应该保存预约数据，并在以后某天系统重启动时再次装入。

提供持久性的存储策略有多种。一个简单的方法只是将数据写入磁盘文件，但是在大多数情况下会使用数据库管理系统。尽管有也一些面向对象的数据库，但是使用中最常见的数据库技术仍是基于关系模型的。因此，本节中的设计的基础是假设用关系数据库提供持久存储。

然而，将面向对象程序和关系数据库混搭在一起并不是简单的任务，因为这两种技术处理的数据在某些部分是以不兼容的方式建模的。在本节，对这个问题将概述一种简单的方法，但对这个问题的系统的处理超出了本书的讨论范围。本节采用了关系数据库的基本知识和相关术语。

实现分为两部分。第一，设计一个数据库模式，这是必要的，这使得系统中对象所保存的数据能够存储和检索。第二，必须设计访问数据库以及从数据库读出数据和写入数据的代码。

6.3.1 设计数据库模式

实现预约系统的持久数据存储的第一步是决定哪些数据是持久的。显然，预约 Booking 需要跨会话存储，因为这个系统整体的核心问题是捕获和记录预约信息。除此以外，还有预约链接的餐桌 Table 和顾客 Customer 对象也需要持久保存。

相比之下，在预约系统 BookingSystem 对象中保存的数据，即当前显示的预约的日期以及当前预约的集合，并不需要持久存储。当系统启动时，用户一般不会关心上次系统在使用中显示的是什么，而且，如果预约总的来说是持久的，那么就不会因为没有保存当前预约集合而丢失不能恢复的日期。应用层中剩下的“Restaurant”类并没有什么真正是自己的特性，而是作为一个系统数据的接口，因而，倘若它只保存了这些数据，“Restaurant”类就不需要作为持久的。

这些结果可以用类中的标记值（*tagged value*）在设计类图中表示。标记值是记录模型元素的特性的一种方法，采用名字-值对的形式。标记值“persistence”用来指出一个类是不是持久的，它具有两个值，非持久的类使用缺省值“transitory”，持久类使用“persistent”。记录标记值的形式是“persistence = persistent”。图 6.7 中显示的餐馆预约系统中的四个持久类，是用广泛使用的简写形式给出的标记值。

在预约系统中，不断创建的预约只有未预约 WalkIn 和提前预定 Reservation，而这些是作为“Booking”类的子类的实例保存的。这意味着“Booking”是一个抽象类，如图中所示。

将抽象类标记为持久的似乎有点不可思议，但是它所包含的一些数据由子类继承了，所以被永久保存。

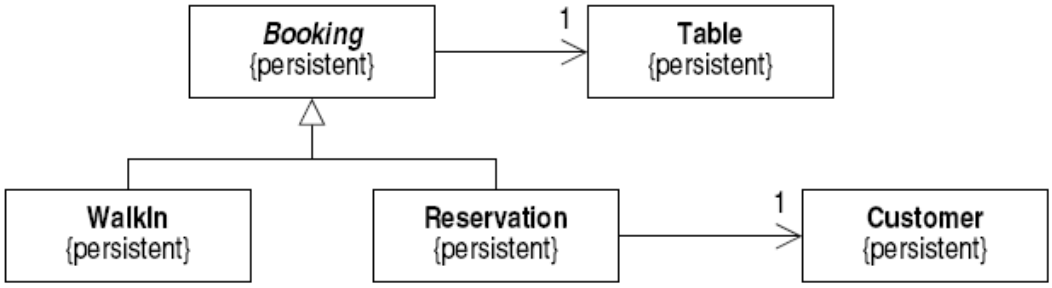


图 6.7　预约系统中的持久类

标记值“persistence”还可以应用于除了类以外的其他模型元素，特别是关联也可以标记为持久的。但是，这在类图中通常是隐含的，采用的假设是持久类之间的关联本身也是持久的。那么，在餐馆预约系统中，预约和餐桌，预定和顾客之间的关联都会作为持久的来处理。

确定了哪些数据要持久存储之后，下一步是描述持久类和关联如何保存在数据库中。这就需要在面向对象的数据模型和关系数据库使用的存储格式之间进行映射。在面向对象系统中，数据保存在互相连接的各个类的对象的结构中，然而，在关系数据库中，数据保存在表中，每个表由若干行组成。数据之间的关系通过某些数据项来支持，这些数据项称为外键，它们出现在多个表中。

对象设计和关系数据库之间最明显的相关性是类和表都定义了一种特定的数据结构，并且，保存实际数据值的类的实例似乎接近对应于表中的行。因此，将类图映射到关系数据库模式的基本策略就是用一个表来表示一个类。

然而，泛化为这种映射引入了一个复杂因素，因为泛化是一种类之间的关系，在关系数据库中没有直接的等价物。在预约系统的实现中，我们将利用“Booking”超类是抽象类的事实，因而需要存储的实例总是属于这个或那个子类，“WalkIn”或者“Reservation”。这意味着假如任何被继承的属性都保存在“WalkIn”和“Reservation”表中，那么在数据库中就不需要有一个单独“Booking”表。

因此，预约系统的关系模式将包含 4 个表，图 6.7 中的每个持久非抽象类各一个。图 5.10 中定义的这些类的属性将作为表的字段建模。为了完成关系模式的设计，我们需要考虑类图的其他重要特征，即关联。

关联的典型实现是让一个对象持有另一对象的引用（见第 7 章）。这样一个引用显然和数据库中外键的作用类似，然而，遗憾的是这些引用不能简单地存储在数据库中，这是因为引用只不过是对象在内存中的地址的一个表示。如果对象存储在数据库中，然后重新装载，就不能保证它会被放在内存中和以前相同的位置。如果以前的地址保存在其他某个对象中，那么很可能引用会被破坏，导致不可预知的运行时问题。

对象的引用不能简单地存储在数据库中，那么关联在关系模式中如何表示就不清楚了。然而，引用可以看作是实现更抽象的对象本体概念的一种方式，是一种每个对象都具有的“隐含的数据值”，并且对该对象是唯一的。

存储关联的常见办法是使对象本体在数据库中明确化，方法是给每个表一个额外的字段来表示特定实例的本体。如果一个对象持有另一对象的引用，那么相应的数据库表中可以存储被引用对象的明确标识，因而能够根据到另一对象的链接来查找引用的对象。

依据这种方法，我们提出了表 6.1 所示的关系模式。这些模式定义了预约系统中的持久数据将如何存储到一个关系数据库中。

表 6.1　餐馆预约系统的数据库模式

Table		
oid	number	places

Customer		
oid	name	phoneNumber

WalkIn				
oid	covers	date	time	table_id

Reservation					
oid	covers	date	time	table_id	customer_id

在纯面向对象程序中，本体是由语言的运行时系统自动处理的，这里的模式不同于此，它要求由程序明确地生成和操纵对象本体。对象标识符充当每个表的主键，即使在类的属性中似乎存在一个自然的键的情况下，譬如餐桌类中的餐桌号，它们也作为外键存储在其他表中，表示对象之间的链接。

6.3.2 保存和装入持久对象

定义了适当的数据库模式之后，我们还必须定义系统如何在数据库和内存之间移动持久数据。一种简单的方法是基于每一个类来处理数据：对模型中的每个持久类，定义一个相关联的映像器（*mapper*）类，其责任是在需要时将数据存储到数据库，以及根据保存的数据值重建对象。

为了跟踪类的实例，同时为了确保不创建重复的实例，需要在模型中表示出数据库模式中引入的显式对象标识符。但是，这并不需要直接引入到应用类中，因为这会将应用类限制到一种特定的持久数据存储策略中，取而代之的是为每个持久类定义一个表示持久对象的子类，这个子类新增加一个属性来保存显式的对象标识符。

图 6.8 说明了这种持久性实现的结构，该图显示了对特定的“Table”类定义的两个新类。对模型中的每个持久类可以定义相似的类。

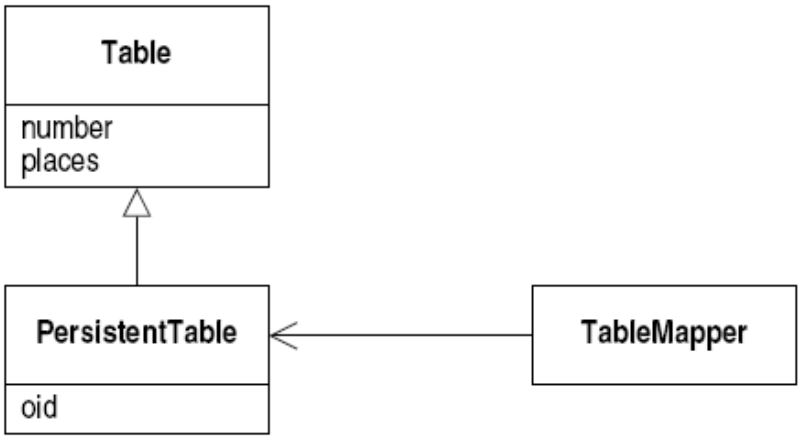


图 6.8 持久类

在运行时，由映像器类创建的餐桌对象中将包括显式的对象标识符，因此是“PersistentTable”子类的实例，由这两个类之间的关联表示。然而，由于“PersistentTable”类是“Table”类的一个子类，这对于应用层中其他想处理“Table”实例的类来说是透明的。

6.3.3 持久性和层次结构

最后，我们应该考虑上面的实现持久性的方法怎样适合整体系统的层次结构。要注意的第一个问题是：这些新的“持久”子类和映像器类依赖于它们支持的应用类，这意味着必须将它们放在应用层，而不是存储层，因为存储层是独立于应用层的。

但是，基本的应用类在很大程度上是独立于这些新类的。唯一的依赖性是由“Restaurant”类引起的，该类有责任了解系统知道的所有预约、餐桌和顾客。一旦引入了某种持久存储机制，它实际上不会自己存储这些对象，而是在需要时检索数据库。为此，必须将它链接到各种映像器类，这些类的责任是根据数据库中的数据创建对象。

现在，将应用层分为两个子包，可以使应用层的结构清晰化，其中一个包含基本的“领域”类，另一个包含支持这些类的持久存储所需要的类。假如存储层提供了独立于应用的数据库服务，那么只有持久性子包依赖于存储层。细化的系统架构如图 6.9 所示，例如对“Table”类的应用。

这种设计保持了重要的特性，即令核心应用类独立于持久存储所采取的策略。如果采用了不同的持久存储策略，“Persistence”和“Storage”包中的类将不得不改变。但是，对领域类必须要进行的唯一修改是对“Restaurant”类的修改，即考虑新的映像器类或者其等价类。

本节描述的设计旨在提供由关系数据库支持的持久性的最简单的可能方法。然而，通过进一步减少包与包之间的依赖，以及将尽可能多的代码移入到存储层，会相当大地改进详细设计。在 Craig Larman 的“*Applying UML and Patterns*”一书中描述了更复杂的方法。

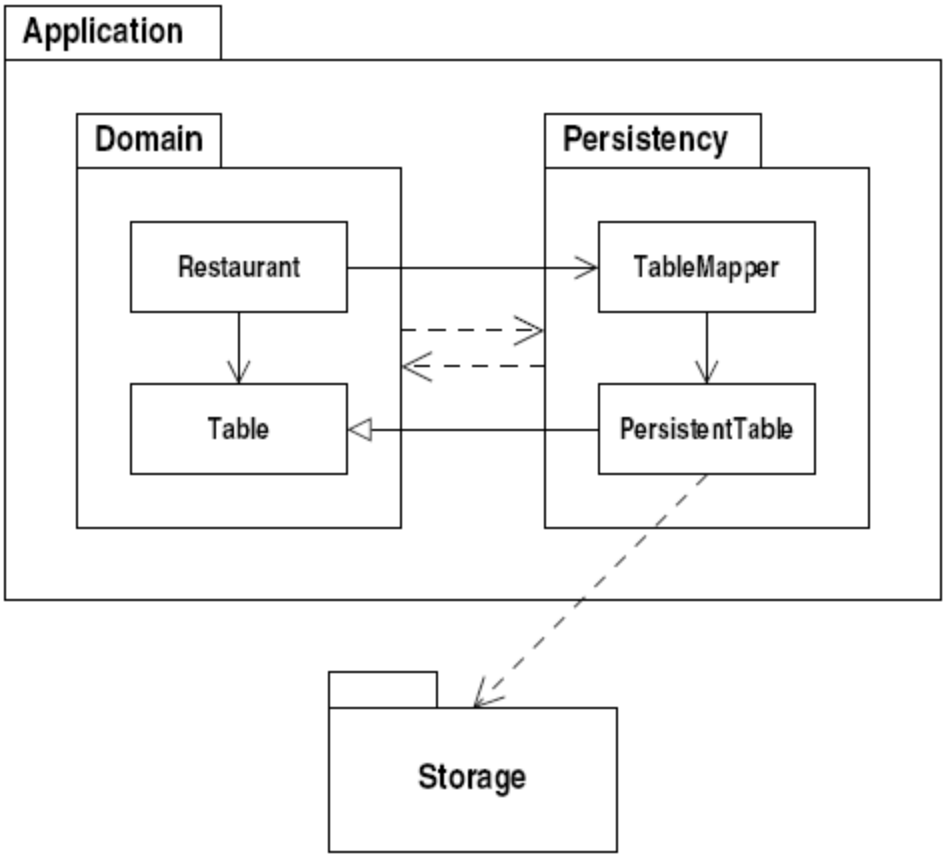


图 6.9 包括持久性的预约系统架构

6.4 设计模型

现在已经描述了餐馆预约系统的第一次迭代的整个设计。从图 5.13 的分析模型出发，加入了图 6.4 所示的元素以支持输入输出，并加入了图 6.9 所示的元素来处理持久性。因此，可以通过结合和完成这三个图画出完整的系统类图。

然而，结果产生的图将会相当庞大和复杂，在理解系统方面，也不可能对这三个独立的图中所呈现的信息有任何增加。与其用手工绘制这样的文档，不如利用支持 UML 的工具更有效。这样的工具能够对模型进行有用的语法和一致性检查，也能够轻松地产生各种图。

6.5 详细的类设计

除了定义设计系统的总体结构，还有一项关键的设计活动是详细地考虑各个类的设计。作为系统用例的实现所产生的顺序图中包含了这项活动的输入：顺序图定义了类的实例必须

能响应的消息，从而定义了每个类中必须定义的操作。各个类的设计从收集顺序图中的所有相关消息开始进行，对这些消息进行一致性检查，并在必要时增加关于参数和返回类型的信息。

在本节中，将详细考虑预约系统 BookingSystem 类，作为类的详细设计的一个例子。这个类的特征的完整清单以及全部的参数和类型信息，如图 6.10 所示。

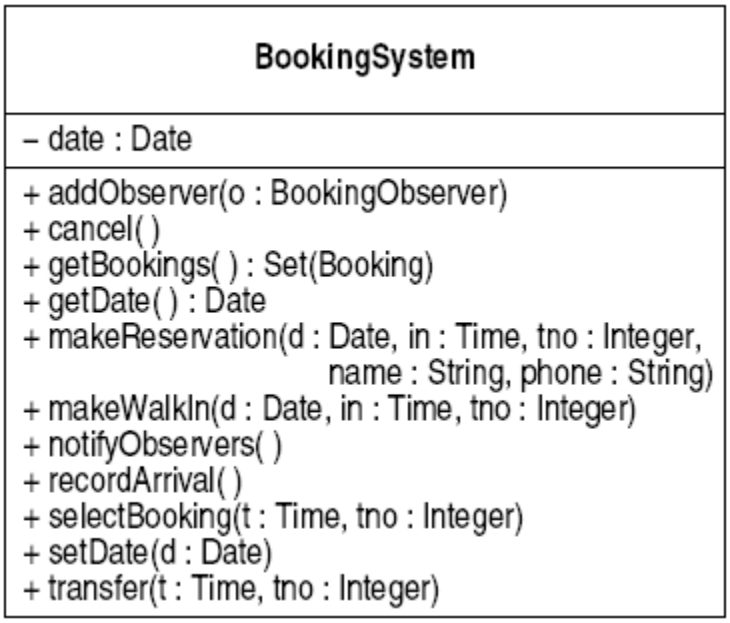


图 6.10　预约系统类的特征

这个类的基本责任是维护当前预约的集合，即用户能够在屏幕上看到的预约，该类支持的操作主要是对这个集合的操作，或是对集合中各个预约的操作。这些预约自身并不是作为这个类的一个属性建模，而是以预约系统类 BooingSystem 和预约类 Booking 之间的一个关联建模，如图 5.13 所示。

除了这些预约之外，该类还需要记录正在显示的预约的日期，如图 5.6 所示。这是作为一个属性存储的，指定为“Date”类型。Date 不是一个内置的 UML 类型，而是我们假设目标编程语言或者会提供，或者很容易定义的一个合适类型。

然而，这引出了下面的问题：如果“Date”作为一个类实现，那为什么在类图中不将其作为一个类显示出来，并将当前日期作为到预约系统类的关联建模，而是作为属性存储呢？答案涉及到 UML 和面向对象编程语言在数据类型和类之间所作的一个根本区别。

数据类型实例是“纯粹的值”：常见的例子是编程语言提供的基本类型，例如 int 和 char。它们不同于对象，不认为它们具有本体，可以自由地拷贝和复制。例如，相同的整数可以存在于内存中的不同位置，而每个对象都是唯一的，存储在一个独特的位置。

通过将“Date”作为一个属性类型，而不是一个类，我们表明日期将像整数一样处理：在实现中它们将被自由拷贝，不同对象持有的日期容易进行比较，等等。由于日期最可能由

一个 Date 类实现，因此，在类图中将其作为一个单独的类显示也不是错误的，但是这不能准确反映日期的预期用途。

在诸如图 6.1 这样的图中，向预约系统发送的是“display”消息请求它显示特定日期的预约。既然已经更详细地考虑了执行显示的机制，那么对这个操作，这似乎是个相当不适当的名字：显示数据是表示层的功能，所以在应用层中有一个称为“display”的操作看起来相当不合适。因而，给这个操作重新起一个新的名字“setDate”。

这个类中有许多维护当前预约集合的操作。“getBookings”操作返回预约集合。返回类型表明，将返回一个预约的集合，但是还没有指定将来要使用的数据结构。这个细节可以在实现中决定，并对类图相应地加以调整。

“makeReservation”和“makeWalkIn”操作创建新预约，为此所需要的参数现在已经详细地写了出来。可能有争论的是这些消息中的日期参数是不必要的，因为总是在系统当前显示的日期进行预约。尽管对于迄今描述的系统的确如此，但是这是一个相当脆弱的假设：例如，餐馆可能要求增加功能，借此顾客可以通过文字形式的详细信息向餐馆进行预约。在这种情况下用户界面就会完全不同，顾客还可以请求进行往后任何日期的预约。将日期参数包括在这些操作中，系统就在某种程度上是“耐久的”：可以无需对系统的应用层进行任何修改而实现文字的场景。

另外一组操作提供了选择一个特定预约并对预约执行各种操作的功能。可以通过指定时间和餐桌来选择一个预约，而“cancel”、“recordArrival”和“transfer”操作将以它们所代表的各种用例所规定的方式来更新这个预约。

最后，“addObserver”和“notifyObservers”操作是图 6.5 所示的观察者模式定义的接口的一部分。新的观察者可以用“addObserver” 向预约系统注册，而在系统状态改变时调用“notifyObservers”操作向所有已注册的观察者发送“update”消息。实际上，在如图 5.4 所示的分析图中的“updateDisplay”消息已由“notifyObservers”消息代替。

6.6 动态行为建模

一个完整的设计应该指定系统中类的结构和行为这两个方面。类图定义预约系统保存的数据以及数据项彼此相关的方式，因而给出了系统静态结构相当全面的描述。关于对象的行为的一些信息则由实现用例定义并在顺序图中显示，其中显示了特定交互所涉及的对象和消息。但是，在交互图中不能捕获单个对象的各个操作之间的各种关系。

例如，如果预约系统对象在收到确定取消预约的“selectBooking”消息之前，先收到了“cancel”消息，那么系统就不知道要取消哪个预约。另外，如果一个预约已经取消了，那就应该不能接收将该预约调换到另一张餐桌的消息，因为对已经取消的预约不允许进行这个操作。这些类型的问题在交互图中不能处理，交互图适合显示可以预期发生的交互，但不适合指定那些不应该发生的交互。

6.6.1 消息的顺序

一般而言，消息发送给对象的顺序会依赖对象的环境。例如，发送给预约系统的消息最终依赖系统用户所采取的行为，而不是依赖系统中执行的处理。对象一般不能支配什么消息发送给它们，或者什么时候发送。因此，原则上我们需要明确说明，对象应该如何响应任何可能的消息序列。

本章和第 5 章中绘制的顺序图详细地说明了一个对象如何响应一个特定的消息序列。但是，在顺序图中显示的消息序列只是示例，如何恰当地归纳这些消息则留给了读者。例如，在图 6.3 中，可以理解在一次交互中可以出现多个“mouseMove”消息，取决于用户移动鼠标多频繁或者多快，而“mouseDown”和“mouseUp” 消息在这个交互中只出现一次。

另外，某些序列被假定是不可能的，譬如包含两个连续的“MouseDown”消息而中间没有“MouseUp”，但是这些无法通过任何顺序图的集合明确地排除。合适的动态建模表示法应该清楚无二义地表明，一个对象在整个生命期中预期接收什么消息序列。

6.6.2 与历史有关的行为

某些消息在不同时间会在一个对象上引起不同的响应。例如，在记录顾客到达餐馆时，应该相应地设置预约的到达时间。然而，如果随后同样的消息再次发送给相同的对象，将不会改变对象的状态，因为一个预定到达多次没有意义。

为了简单起见，这里我们忽略了错误地设置第一次到达时间的可能性。这种情况也许可以用一个“修改到达时间”的新用例较好地予以处理。通过某种方法使一个操作不能出现在用户界面上看来或许也能阻止消息被发送两次。这是可能的，但并不能改变普遍的情况：毕竟，可能存在另外的界面，例如从移动电话向餐馆发送文本消息所提供的界面，在这个界面上不能禁止一个操作。

一种更好的方法是让对象负责检查自己的当前状态下没有意义的消息。由于一个消息的效果可以依赖于先前已经发送给它的消息，这意味着对象必须以某种方式知道自己的历史，或者知道自己已经接收的消息。

6.6.3 指定行为

因此，对象行为有两个方面在交互图中没有捕获，但是这需要作为系统设计的一部分明确说明。

1. 对象预期接收什么消息序列。

2. 对象如何响应消息，尤其是这个响应如何依赖于对象的历史，即它已经接收的消息。

通过形式化对象在不同时间可以处于多个不同状态之一的概念，我们可以指明需要的行为。如果我们假定对象可以响应接收到的消息从而改变状态，那么对象在给定时间的状态将依赖于到那时为止它已经接收到的消息。另外，如果对象对消息的响应可能根据它的状态而不同，我们就能够指明上面描述的对象行为的两个方面。

可以用 UML 的状态图（*statechart*）表示法指定对象的行为。为一个类定义一个状态图就指明了该类的所有对象的行为。在下面两节，通过为餐馆预约系统中的两个类定义状态图非正式地介绍状态图表示法。

6.7 预约系统的状态图

如上面所指出的，预约系统 BookingSystem 类显示出的最重要的依赖状态的行为与预约的选择有关。某些消息，如“recordArrival”，只有在已经选择了一个预约的条件下才被切实地处理。这种情况的基本动态在图 6.11 的状态图中定义。

图 6.11 一个简单的状态图

图 6.11 显示了状态图的主要特征。不严密地讲，状态对应于对象在等待接收消息的一段时间，状态用圆角矩形表示。事件对应于一个对象可以接收的消息。转换是连接两个状态的箭头，通常标记着事件的名字。

在任一给定时间，对象总是处于它可能的状态之一。当它接收到一个消息对应于从它当前状态出发的转换上的事件时，该转移被激发，对象进入转换另一端的状态。例如，假定当前没有选择的预约，预约系统处于图 6.11 左部所示的“NotSelected”状态。如果现在发生图 5.11 所示的交互，预约系统会先收到一个“selectBooking”消息，这会引起标记着该消息名字的转换被激发，而预约系统对象将迁移到“Selected” 被选中状态。然后，接收到“recordArrival”消息，标记为“recordArrival”的转换激发。但是，预约系统仍然处于接收消息之前的状态，换句话说，在这个消息被处理之后，对象还处于被选中状态。消息“transfer”和“cancel”也只有在预约处于被选中状态时才有意义。“transfer”和“recordArrival”的行为方式相同，但是“cancel”的结果略有不同。一旦取消了一个预约，它将从显示中消除并被删除，所以不会再存在一个选中的预约。因此，在状态图中，标记着“cancel”的转换必须将系统移回到“NotSelected”状态，如图 6.12 所示。

6.7.1 非确定性

图 6.12 并没有详细说明 “selectBooking”消息的全部结果，而只是举例说明了预约一开始被选择的情况。用户也可能在已经选择了一个预约时又选择另一个：这可以通过给图 6.12 增加一个“Selected ”状态上的循环转换来建模。但是，还有可能收到“selectBooking”却没有选中预约。例如，如果用户在一个没有显示预约的屏幕位置单击鼠标，就会发生这种情况，那么，传递给预约系统的时间和餐桌参数就没有对应的预约。图 6.13 说明了收到

“selectBooking”消息的所有可能结果。假定如果在屏幕上一个空位置单击鼠标，已选择的预约仍处于选定状态。

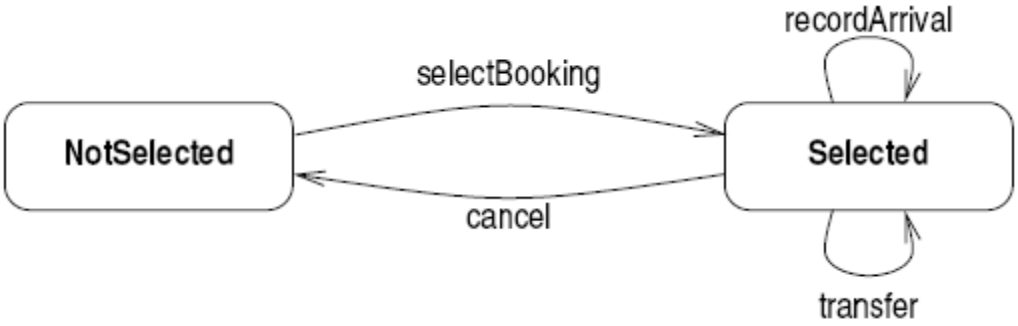


图 6.12　在选中预约上执行一个操作

图 6.13 选择预约中的非确定性

“NotSelected”状态上的循环转换表示没有选中预约和用户在空位置单击鼠标的情况。假若这样，系统仍处于“NotSelected”状态。相反，如果用户在一个预约上单击鼠标，那么到“Selected”状态的转换被激发。

然而，图 6.13 没有明确化这些细节，只是显示了从“NotSelected”状态出发的两个不同转换，并没有区分在哪种情况下一个转换被激发而不是另一个。状态图中有引起歧义的特性使其中两个由同样事件标记的转换将会被激发，称为非确定性（*non-deterministic*）。

6.7.2 监护条件

在被建模的系统中真正存在非确定性的情况下，像图6.13那样的状态图是完全适合的。然而，大多数系统是确定的，这时，在给定状态下接收一个给定的事件将总是导致同样的结果。在预约系统的情况中，它完全依赖于和“selectBooking”消息一起传递的参数：如果光标在一个预约上，就会到达“Selected”状态，否则，系统将停留在“NotSelected”状态。

为相关转换增加监护条件（*guard condition*）可以在状态图中表明这些事实。图 6.14 说明了如何指定监护条件以解决图 6.13 中的不明确性。实质上，通过明确地指明在哪种情况一个转换而不是另一个将会被激发，这个监护条件就将上面非形式化的讨论并入了状态图。

监护条件写在转换上标注的事件后的方括号中。带有监护条件的转换只有在相应的事件被检测到并且条件为真时才可以激发。在图 6.14 中，在任何时候，都只有一个监护条件可以为真，所以消除了图 6.13 中的非确定性。

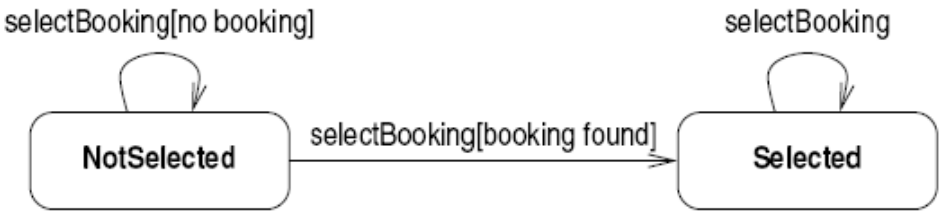


图 6.14　通过监护条件消除非确定性

注意，“Selected”状态上的循环转换上不需要监护条件。这是因为在这两种情况任一种之下，系统仍然停留在预约被选择的状态：如果在指定位置没有发现预约，如上面所叙述的，原来选择的预约仍是被选中的。

6.7.3 动作

如果需要，可以在状态图中记录一个新预约被选择的事实，方法是在相应的转换上包含动作（*action*）。动作写在转换的标注上，在事件名字和监护条件（如果有的话）之后，前面加斜线：图 6.15 是带有动作的状态图，强调新预约将被选择的情况。

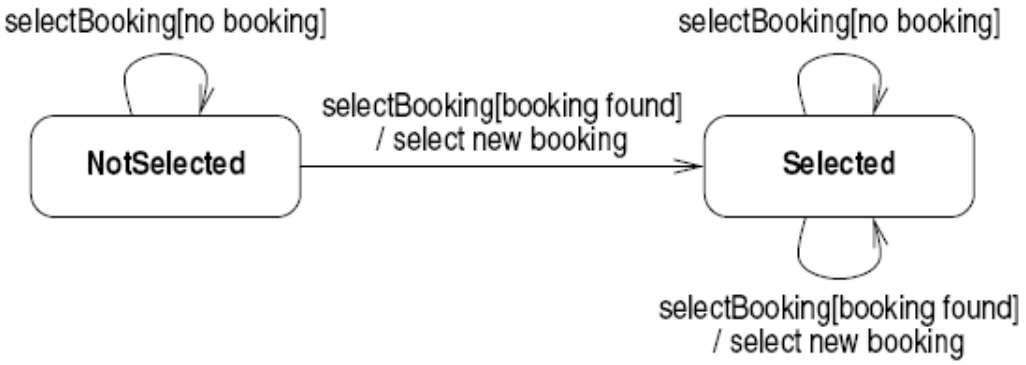


图 6.15　在状态图中包括动作

6.7.4 组合状态

除“setDate”以外，图 6.12 和 6.15 为预约系统类中定义的所有相关操作都定义了转换。然而，如果将它们合起来全放在单独一个图中，将相当混乱和难以理解，部分是因为图 6.14 中的消息、监护条件和动作的重复。

为了画出更清晰、更简单的状态图，可以使用组合状态（*composite state*）将相关状态集合到一起并使它们共享的行为明显化。图 6.156 显示了预约系统类的一个完整的状态图，其中利用了组合状态。

组合状态用大的状态图标表示，其中包含了两个嵌套的“子状态”。由两个子状态共享的转换被附在组合状态上：那么这些转换就适用于所有嵌套的子状态。

例如，图 6.15 表明，无论预约系统处于什么状态，如果接收一个“selectBooking”消息，并且在光标位置发现了一个预约，那么结果是相同的：预约被选择并且预约系统最后在“Selected”状态。在图 6.16 中，这个共享行为用从组合状态出发的单个转换表示，和图 6.15

中的两个转换具有完全相同的事件名字、监护条件和动作，该转换结束于“Selected”状态。这一个转换代替了图 6.15 中两个标记相同的转换，但指定的是完全相同的行为。

因此，一个自组合状态出发的转换等价于若干个来自每个嵌套子状态的标注相同的转换。同样的技术被用来指定接收 “setDate”消息的结果：无论系统处于什么状态，结果都是返回到“NotSelected ”状态。这样，在用户输入新日期时，已选择预约的任何记录都丢失了。

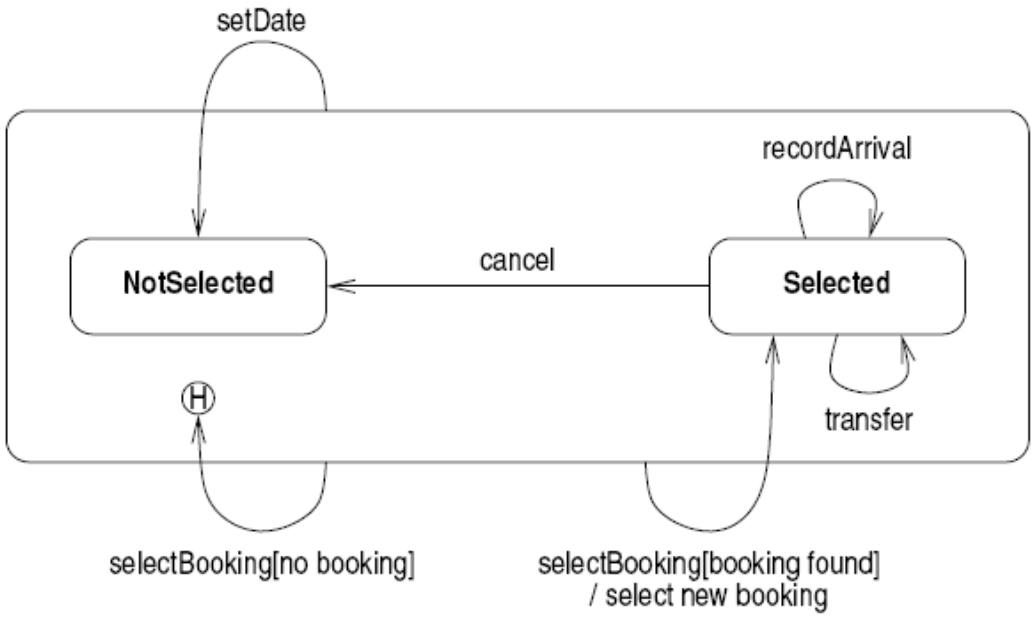


图 6.16　预约系统类的一个完整状态图

如果收到一个“selectBooking”消息，但是没有所选择的预约时，情况就不这么简单了。从图 6.16 可以看到，在某种意义上，这个事件在两种状态下具有相同的结果：系统仍然停留在已处的状态。将这个共享的行为也用从父状态出发的一个转换表示是美好的，但是由于这两个转换结束于不同的状态，所以不能像在其他情况下那样简单地处理。

对于像这样的情况，可以在组合状态中使用一个特殊的历史状态，用一个圆圈中的“H”表示。历史状态的作用就像是系统上次所处的子状态的引用。在图 6.16 中，如果收到“selectBooking”消息，但没有所选择的预约，会激发从子状态出发的转换，到达历史状态。系统将停在“Selected”或者“notSelected”状态之一，取决于转换之前它在什么状态。

6.8 预定的状态图

预定类是用状态图概括一个类的对象的行为的另一个例子。预定的确显示出了依赖于状态的行为：一旦记录了到来者，就不可能取消预约，或者再次记录到达。总结这一行为的状态图如图 6.17 所示。

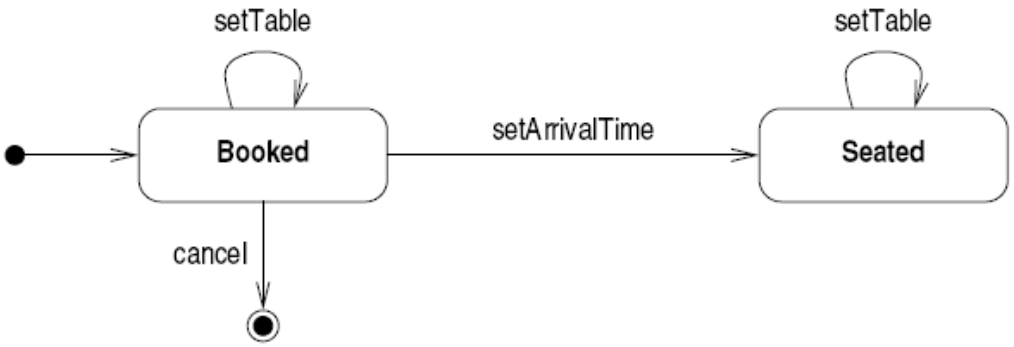


图 6.17　预定类的状态图

图中显示了两个状态，“Booked”状态对应于已经进行了预定，但是顾客还没有到达餐馆的时候，在顾客到达并且系统记录了他们的到达时间后，到达“Seated”状态。

在任何时间改变分配给一个预定的餐桌都是可能的，所以，每个状态上都出现有一个“setTable”转换。如果觉得值得，那么可以用一个组合状态和单个转移来代替它。

图 6.17 还说明了状态图表示法的一个新特征，*初始状态*和*终止状态*的使用。初始状态用一个小黑球表示，带有一个无标注的到状态图中另一状态的转换。这个转换对应于为对象调用构造函数，作用是显示一个新创建的对象处于什么状态。因此，图 6.17 指定了当预定被创建后将进入“Booked”状态。

终止状态表示为由圆圈包围的小黑球，对应于对象被销毁的点。图 5.9 表明，在预约被取消时销毁预约对象，所以到达终止状态的转换上标注着“cancel”事件。一旦已经记录了到来者，就不可能取消预约，隐含着该预约对象永远都不会销毁。这可能对应于一个业务需求：将旧的预约信息存档，或许在以后的管理中使用。

在图 6.17 中没有显示从“Seated”状态出发的“cancel”转移，但是在预定处于该状态时，也可能接收一个 cancel 消息，或许是因为代码中某些未被检测出的错误。对这种情况有两种可能的响应：一是直接忽略该消息，二是将其视为一个错误，并以某种方式抛出一个异常或其他。

状态图的语义是：如果检测到一个事件，并且不存在从当前状态出发的转换标注为该事件，那么这个事件就被忽略。这使得状态图更容易绘制，因为不必在状态上包括没有重要影响的事件。另一方面，这确实意味着必须在状态图中加入某些东西，以表示某些事件在某些状态下必须不被检测的事实。比较容易的方法是简单地在状态图中增加一个“Error”状态，并让错误的事件标记一个到错误状态的转移，在那里可以定义需要的错误处理。

6.8.1 何时不画状态图

没有必要为系统中的每一个类绘制状态图。一般说来，只为那些具有“有意义的”行为的类绘制状态图：典型地，这些是预期以某个固定次序接收消息的类，或者显示出依赖状态的行为，在不同时间以不同方式响应相同消息的类。

例如，看起来可以为顾客类画出状态图，显示例如在未来某个时间有预定的顾客和没有的顾客之间的区别。像这种顾客之间的区别当然可以识别，但是，从系统的角度看，它们是

完全不相关的。顾客类接口中的操作可以在任何时候被调用，并且在所有时间都有同样的结果。因此，画一个顾客类的状态图说明这一区别并不会对系统设计文档增加有用的信息。

6.9 小结

- 系统的输入可以由表示层中的一个边界对象处理，它向应用层中的控制器发送一个适当的系统消息。
- 当系统状态的变化可能要求显示改变时，应用层可以用观察者模式定义的方式通知表示层。
- 如果由关系数据库提供持久存储，那么，可以通过为每个持久类定义一个数据库表并增加显式的对象标识符来创建模式。
- 可以为每个持久类定义映像器类，其责任是向数据库中存入并从数据库中装入持久对象。
- 在设计工作流中进行的详细类设计要向类中加入来自所有顺序图的操作，并使诸如参数、返回类型和可见性这样的特性明确化。
- 对显示出依赖状态的行为的类，可以用状态图指定其行为。状态图说明可以发送给一个对象的消息序列，以及对象在不同时间可能对消息做出的响应。
- 状态图显示了一个类的实例可以处于的不同状态、在改变状态时可以依循的转换，和触发状态改变的事件。初始状态和终止状态表示对象的创建和销毁。
- 监护条件指定了一个转换发生的条件，动作规定了对象在特定状态下响应接收到的消息时做些什么。
- 通过分解出共享的行为，可以用组合状态简化状态图的结构。

6.10 习题

（1）将第 4 章的“调换餐桌”用例重写成一个更一般的“移动预约”用例。由于这个修改，还需要对系统文档进行什么改动？

（2）如果你有 UML 工具，如 Rational Rose，请为餐馆预约系统创建一个完整的设计模型，将本章中讨论的所有要点都包含进去。

（3）图 6.10 中，“BookingSystem”类的操作都是 public 的。但是，其中许多操作并不需要整个系统可见，而只是在图 6.9 所示的“Domain”包中可见即可。修改图 6.9，使只有在包外可见的操作是 public 的，其余的使用“包可见性”，用“～”表示。

（4）为餐馆预约系统中的其他类产生一个类似于图 6.10 的图。

（5）图 5.9 显示了确认取消预约时用户和预约系统之间的交互。这个交互和预约系统计划的层次结构一致吗？如果不一致，请解释为什么，并修改顺序图，表明应该如何获得对取消的确认。

（6）不使用组合状态，为预约系统类画一个等价于图 6.16 的状态图。从可读性的角度比较你的答案和图 6.16。

（7）在图 6.17 中增加一个错误状态，表明如果试图取消已经就座的预定时，必须进行某些错误处理。

（8）图 Ex6.8 所示的“Table”类的状态图是否是预约系统的一部分有用文档？如果是，请完成它。如果不是，请解释原因。

图 Ex6.8 提议的餐桌的状态图

6.11 实践题

（1）学习使用一种支持 UML 2 的建模工具，掌握用例图、类图、顺序图和通信图的画法。

（2） 用建模工具绘制你的实践作业中的各种图。

第 7 章 餐馆系统：实现

前面几章讨论了餐馆预约系统的设计，本章描述该系统实现的某些方面。第 1 章阐明了对象模型的语义如何保证在设计及其实现之间存在紧密连接，本章举例说明将 UML 设计模型转变为面向对象语言代码的一些简单而系统的技术。

预约系统是一大类交互式单用户应用程序的典型代表，这些应用具有图形化的用户界面，并通过输入设备（例如鼠标和键盘）检测用户的交互。这样的应用中包含大量的低层代码，处理应用代码和输入输出设备之间错综复杂的交互。

由于这种代码是大多数应用共有的，因而发展出了提供核心输入和输出功能的标准实现框架。现在，应用程序员通常只需要将实现特定应用的功能写入一个通用框架，而不用从零开始编写整个应用程序。7.3 节讨论了框架的一般概念，并在 7.4 节介绍了将预约系统集成到 Java 应用程序框架的细节。

7.1 实现图

分析和设计活动产生的文档描述了软件应用的逻辑结构，基本上是将系统作为可能细分到若干个包中的一组类看待。这些类的实例的动态行为通过交互图和状态图进一步定义。

在实现系统时，用选定的编程语言以某种方式表示这些类。这时，系统第一次呈现出实物形态：一组源代码文件。接下来编译源代码，产生各种目标文件、可执行文件或库文件。最后，这些文件在一个或多个处理器上执行，也可能结合其他的资源。

UML 定义了两种实现图以文档化系统物理结构的各个方面。构件图（*component diagram*）文档化系统的物理构件以及它们之间的关系，部署图（*deployment diagram*）文档化这些构件如何映射到实际的处理器上。

7.1.1 构件

程序员一般在谈到类和实现类的代码时好像它们是相同的东西，但这二者之间的区别是很重要的。如果开发的程序要在多个环境中运行，也许是在不同的操作系统平台上，那么同样的类可能需要以多种方式、甚至是用不同的编程语言来实现。

为了明确这个区别，UML 定义了构件（*component*）的概念。构件是表示系统部件的物理实体。构件有很多不同类型，包括源代码文件、可执行文件、库、数据库表等等，通常用构造型来明确构件表示哪种实体。

图 7.1 显示了构件的 UML 表示法，它是一个边上嵌着小矩形的矩形框。类和实现类的构件之间的关系可以用二者之间的依赖建模。这种依赖性有时用构造型“trace”标示，如果图的含意已经很清楚，也会省略构造型。

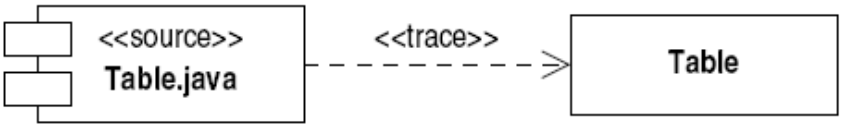


图 7.1 实现一个类的构件

UML 2 的构件表示法有所不同，如下图所示。构件用矩形图标表示，矩形框内写上构件的名字，加上<<component>>构造型，或（和）在矩形框右上角加小图标。在各种 UML 建模工具中，这几种表示法都有使用。

7.1.2 构件图

组成系统的源文件可以显示在构件图上。构件图显示了用依赖链接的构件，这种依赖典型地表示构件之间的编译依赖。在两个源文件之间，如果要编译一个，另一个必须是可用的，那么二者之间存在编译依赖。在一个类使用另一个类，例如，作为一个属性的类型或者超类时就会发生这种情况。因而构件图文档化了系统的构造需求，并且，例如能够形成生成系统 make 文件的输入。

图 7.2 显示了餐馆预约系统的部分构件图，展示了表示层和应用层中的领域类。注意，构件图中保留了分析和设计模型中定义的包结构。除了预约类层次中的类放在单独一个源文件中之外，这个图基本上文档化了类和源文件之间的对应关系，如 Java 程序中常见的那样。

UML 2 扩充了 UML 1 的构件图，增加了许多新的模型元素，用以支持基于构件的开发。详细内容见后续章节。

7.1.3 部署图

部署图表明在部署系统时，系统中的构件如何映射到处理器。餐馆预约系统最初的意图是作为在单个 PC 上运行的单用户应用来部署的，所以，图 7.3 所示的部署图有点无关紧要。进行处理的节点在部署图中用立方体表示，在该节点上部署的构件显示在立方体“内”。

当系统在网络上部署时，部署图更具有说明性，可以在图中显示网络中的不同节点。因而部署图的目的是表明在多个不同节点上的处理的物理分布。

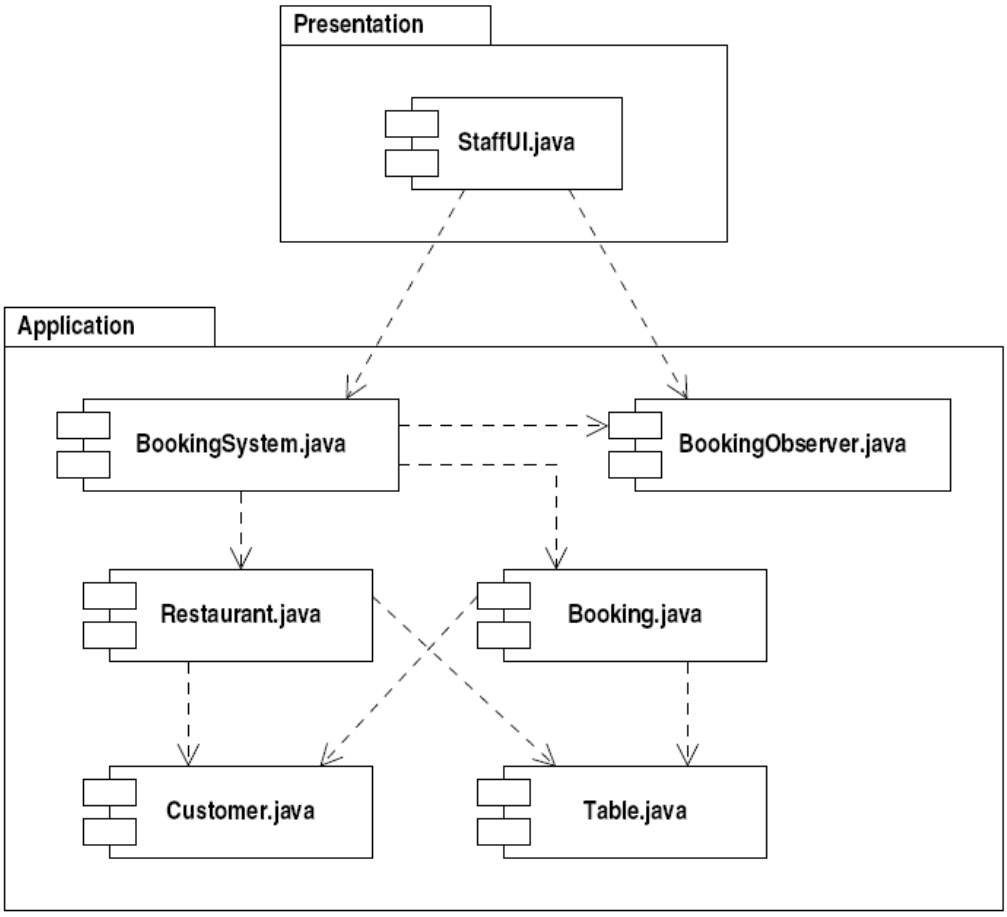


图 7.2　预约系统的构件图

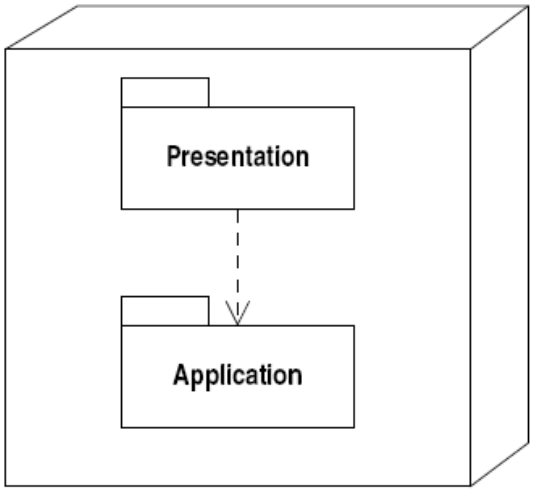


图 7.3　预约系统的部署图

7.2 实现策略

图 7.2 中的构件图显示了实现预约系统必须创建的源文件。然而，构件之间的依赖对源文件创建和测试的次序施加了特定的约束，并说明了实现的两种基本方法。

自顶向下实现从高层构件开始，按照依赖性箭头的方向，在图 7.2 中 “向下”进行。这种策略的优点是，能够在过程的早期测试系统的总体设计。缺点是需要为低层类创建桩（*stubs*）或临时实现，随着开发的进行，在后面才用这些类的真正实现代替。

自底向上实现从低层构件开始，在图中“向上”进行。这种方法能够让单个构件的开发和设计更容易：当实现一个类时，所有它依赖的类都已经实现了，可以不需要开发桩就能对它进行编译和测试。 然而，自底向上方法的风险是将整个可执行程序的生成一直推迟到实现中相当晚的阶段。

这两种方法的折衷是采用一种更迭代的方法，而且不是从实现类而是从实现用例考虑。用这种方法，对每个类，开发者实现的是各个类中支持单个用例所需要的那些特征，然后充分地测试。然后一个接一个地实现更多的用例，对这些类增加所需要的另外的特征。

7.3 应用框架

像餐馆预约系统这样的应用程序，包含了一定数量的特定于应用的功能，这些功能与专用于该餐馆的业务对象和规则的实现有关，但是，也存在着大量和其他使用基于窗口的图形用户界面的应用程序共享的功能。

应用程序员几乎没有必要编写这些执行通用低层功能的代码。大多数编程语言和环境现在支持一种重要的复用层，使处理例如输入输出的代码能够以类的框架（*framework*）的形式复用。典型地，交互式图形应用的框架会支持以下功能。

1．管理应用与其环境之间的交互。在窗口环境中，框架可以支持应用程序使用的窗口的创建和随后的管理。在 applet 的情况下，框架可以提供浏览器启动和终止在网页上运行 applet 所必需的功能。

2．提供检测用户输入并将这些输入以若干标准的良好定义的消息的形式提交给应用程序。输入可能直接由物理设备产生，譬如鼠标和键盘，或者以用户界面窗口部件为中介，譬如菜单项和按钮。

3．提供一个图形函数库，能产生输出并在应用程序控制的窗口中显示。

术语“框架”是比喻的使用，它意味着两件事。第一，框架代码可以被想象为是围绕着特定于应用的代码，就和包围着照片的相框一样。第二，框架提供了一个完整的但又只是一个骨架似的应用程序，可以作为一个支持结构使用，在上面建立完整而专门的应用程序。

框架在使应用程序员免于关注低层方面的作用，可以形象化地予以表示，如图 7.4 所示。这个非形式的图说明了，框架用什么方式使应用程序员可以避开用于处理输入输出的低层应

用编程接口（API）。它还说明了，框架如何通过提供到低层功能的标准接口而可以在许多不同的应用中复用。

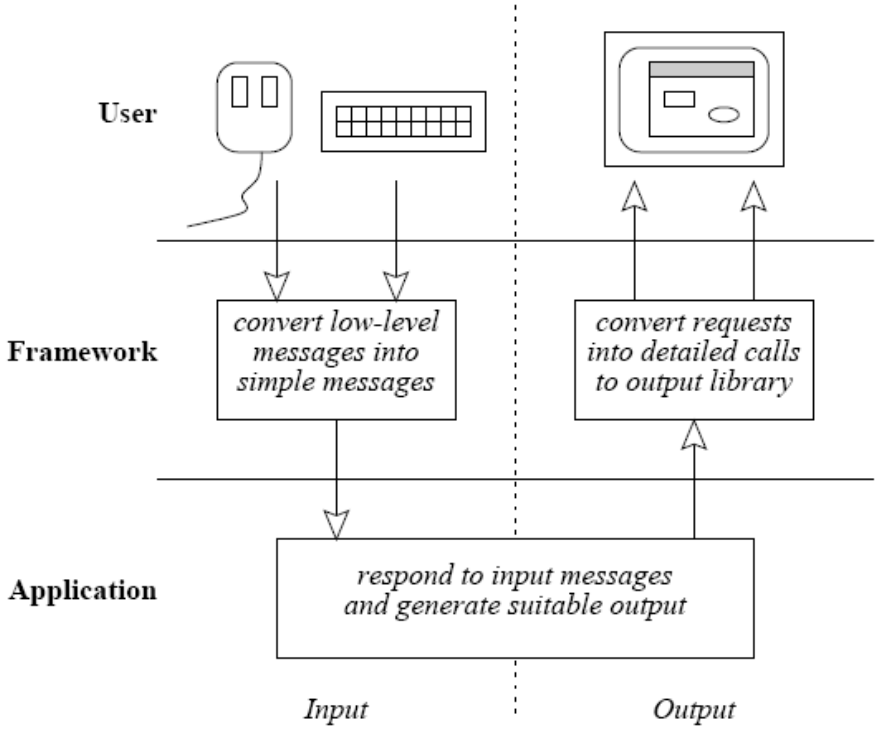


图 7.4　用户界面框架的作用

7.3.1 热点

框架作用的这种一般描述并没有解释，程序员如何将应用代码和框架提供的代码集成到一起。面向对象的框架往往借助于热点（*hotspot*）达到这个目的。在框架中，热点是应用程序要特殊化的类，如图 7.5 所示。 图中的虚线指示框架类和特定于特殊应用代码之间的分界线。

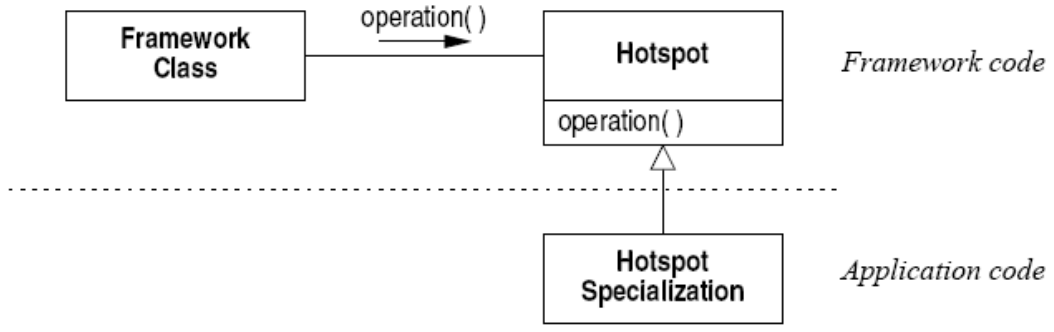


图 7.5 热点类

假定热点类提供了显示窗口及窗口内容的基本功能。这些功能在所有应用程序中必须提供，并且在许多标准情况下执行。例如，如果用户打开了另一个窗口使应用窗口变暗了，然

后又关闭或移动了该窗口时，那么就需要重画该应用窗口的内容以恢复正确的显示。框架将检测这些事件，并在适当的时候会向窗口类，即热点，发送一个消息，通知它刷新自己的显示。

一般地，热点类定义若干由框架在标准时间调用的操作。这种情形在图 7.5 中表示为从其他框架类发送到 HotSpot 的消息。应用程序员的基本任务是定义热点类的特化，并覆盖这些操作，以实现一个特殊应用的特定功能。

热点类中的操作可能是抽象的。在产生完整的应用程序之前，应用程序员必须提供这些操作的实现。但是，在大多数情况下，可以为操作写一个明智的缺省实现，即使是琐细的什么都不做的一个实现。假若这样，就可以从框架生成一个完整的应用程序，即使价值不高，而应用程序员要做的全部工作就是覆盖所关注的操作。

确切地说，需要覆盖哪些操作取决于该框架所提供的实现。图 7.6 所示的是 HotSpot 接收一个指示用户移动了鼠标的消息。在这种情况下，不存在框架可以提供的缺省功能，所以热点类为这个操作提供了一个空实现。应用程序在特化的类中覆盖了这个操作，而动态绑定机制保证了在框架类发出一个消息时，会执行特定于应用的代码。

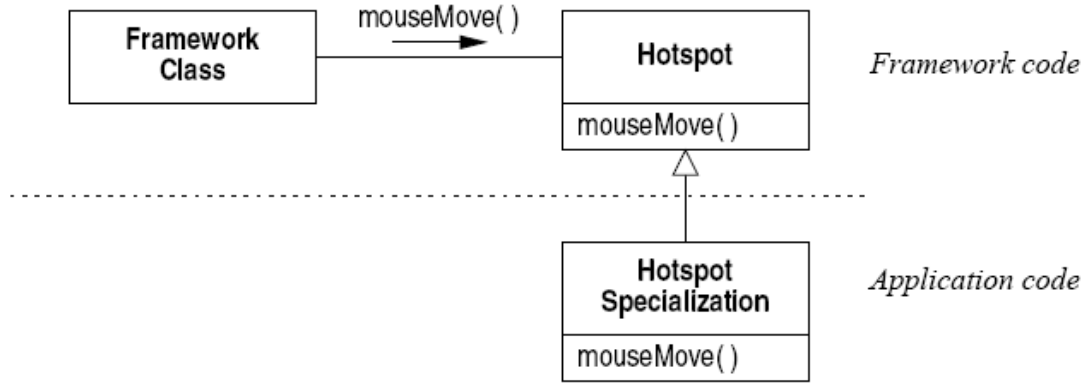


图 7.6　覆盖框架操作

在其他情况下，也许框架可能为一个操作提供有实际价值的缺省实现。图 7.7 说明了框架请求重新显示一个窗口及其内容的情形。在这种情况下，框架能够提供一些通用代码以重画窗口背景、窗口的边界和滚动条等等。这些代码在图 7.7 中的“redisplay”方法中提供。另一方面，在窗口中显示特定于应用的内容的代码则必须由程序员提供。

如果程序员通过覆盖 redisplay 方法提供这些代码，就会丢失由框架提供的通用代码。原则上，这个方法可以复制到覆盖函数中，但是这将消除使用框架所带来的许多益处。在这样的情况下，可以采用一种不同的机制。在图 7.7 中，热点类定义的第二个方法称为“displayContent”，该方法的目的是在窗口中显示特定于应用的内容。这里给了一个空的缺省实现，程序员覆盖的是这个方法。

Redisplay 函数在适当的地方调用增加的这个 displayContent 函数，以便将特定于应用的代码集成到通用框架中。那么，在运行时，当收到一个 redisplay 消息时，首先执行热点的 redisplay 函数；它调用显示内容的函数 displayContent，而通过使用动态绑定，这个特化的函数将被执行。

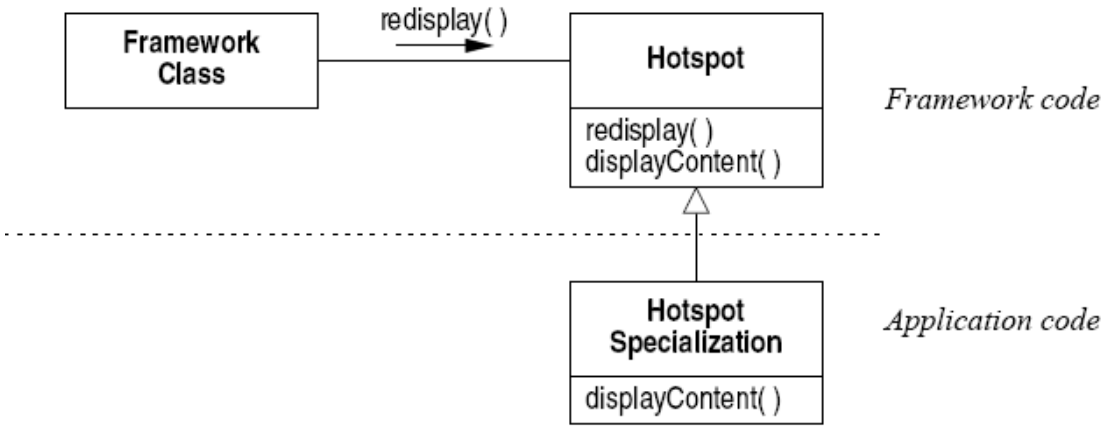


图 7.7　覆盖“回调”方法

像图 7.7 中的“displayContent”这种操作称为回调（*callback*）函数或钩子（*hook*）函数。它们提供了一种技术，借此，应用程序员能够通过重定义框架方法调用的其他操作来扩充框架方法的功能。

7.3.2 控制的倒置

框架的使用带来了不同于传统风格的一种特殊编程风格。在传统模型中，程序员编写一个“主程序”，规定了应用程序内的整个控制流。应用可以被分解为一组类和函数，其中一些可以由库提供，但是实质上控制仍在程序员手中，程序员决定在程序运行时用户可以做什么。

然而，当使用框架时，这个关系就颠倒过来了，这种情况通常被称为控制倒置（*inversion of control*）。程序中的控制流驻留在框架代码中，程序员只是提供一些函数，在过程中某些合理确定的地方被调用。运行时的控制在用户手中，而不在程序员手中，程序员的工作是提供代码，以适当的方式响应用户的动作。因此，这种编程风格有时称为事件驱动。

7.4 Java GUI 框架

作为使用框架的一个很简单的例子，本节描述如何用 Java 的 GUI 框架支持餐馆预约系统如图 4.3 所示的图形用户界面。该界面的结构很常见，由带有一个标题和一个菜单栏的单窗口组成。显示有当前日期，窗口的其余部分由显示当前预约的区域占据，并可以在此检测鼠标事件。

7.4.1 用 UML 文档化框架

Java 提供了 GUI 的 API 包 java.awt 和 javax.swing。可以用 UML 图文档化它们的结构和功能。通常，这作为一种理解框架如何被组织到一起以及如何使用框架的方法，可能非常有帮助。例如，图 7.8 展现了部分 GUI 类（图中没有使用包符号）。

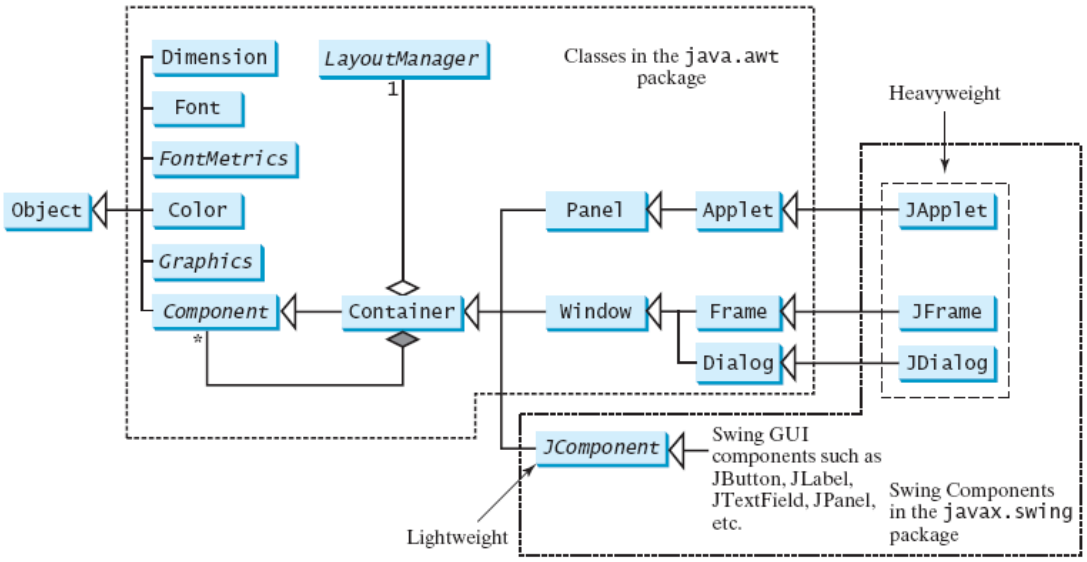
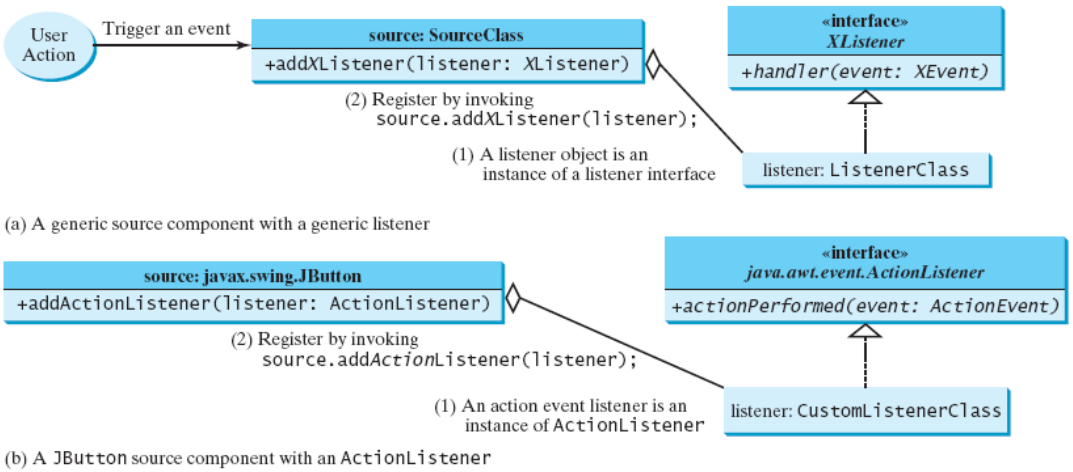


图 7.8　一些 java GUI 类（简化的）

组成用户界面的元素被称为组件，并被定义为抽象类“Component”的子类。某些组件，如容器（Container）组件可以包含其他组件，如“Container”和“Component”类之间的关联所示。早期的 Java 程序中使用 AWT 框架处理用户界面，现在基本都使用相应的 Swing 组件

边框（JFrame）是一种特殊的容器，表示应用程序窗口，它包含标题、菜单栏，以及各种特定于平台的属性。菜单栏由 JMenuBar 类（JComponent 的子类）定义。JPanel 定义了一块屏幕区域，上面可以产生任何类型的图形输出，并能够处理用户输入事件。边框中可以包含多个 JPanel 对象，形成特定的屏幕布局。之间没有预先定义的关系。

用户产生的事件在子包 `java.awt.event` 中定义，它定义了各种事件类和相应的监听器接口。监听器接口由检测对应事件。组件可以与一组监听器相关，以检测用户在该组件上产生的事件，产生事件的组件被成为源组件。Java 的事件处理机制如下图所示。



7.4.2 集成预约系统和 Java GUI 框架

在餐馆预约系统的实现中，为了利用 Java GUI，我们必须确定需要特化的相关热点类。这些类中最重要的两个是“JFrame”类和“JPanel”类。

画布提供了用户界面的主显示区域必需的基本功能，即显示图形化资料和检测用户产生的事件的能力。在预约系统的设计中，这些任务是“StaffUI”类的责任。因此，一种切实的应用选择是将“StaffUI”作为 `javax.swing.JPanel` 的一个子类实现，并通过重定义该类中适当的方法支持所需功能。

画布需要有一个容器来放置，所以我们另外需要一个“JFrame”的子类，作为预约系统的主应用窗口。至今，这个类还没有在设计中标明，因为它支持的例如“最小化窗口”这样的功能是由纯粹的通用操作组成的，这些操作在预约系统的用例中没有显式给出。这些新类如图 7.9 所示，它文档化了预约系统的表示层与 Java 应用框架集成在一起的方法。

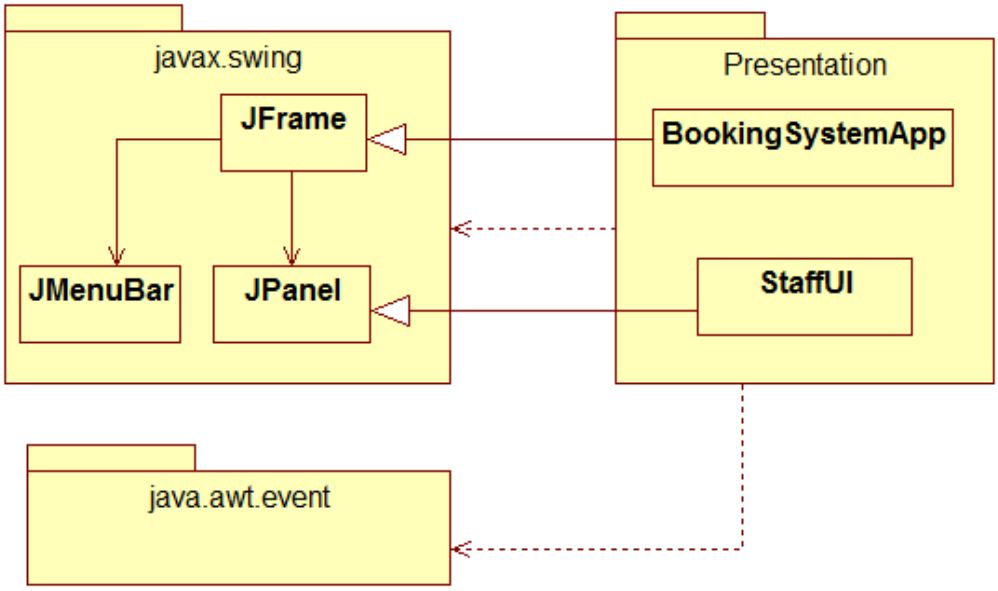


图 7.9 将 Java GUI 框架用于表示层

在预约系统中使用 Java GUI API 还涉及了许多更低层的细节，然而一旦将总体策略展现在图 7.9 中，这些细节可以通过研究应用的源代码更好地理解。

7.5 类的实现

下面几节描述预约系统的 Java 实现的一些方面，说明设计中的各种 UML 特征如何映射到代码。这里提介绍的技术并不是唯一的转换途径，但是比较清晰、简单易懂，而且强调了设计模型与编程语言结构之间的密切关系。

7.5.1 类

UML 类图中的类毫无意外作为面向对象语言中的类实现。图 7.10 显示了预约类，它具有在开发早期确定的属性和操作的一个子集。

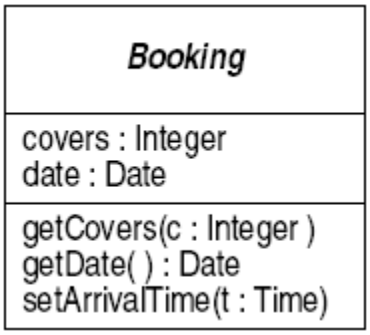


图 7.10 Booking 类

这个类可以自然地作为一个包含等价的域和方法的 Java 类实现。图 7.10 所说明的预定类的 Java 类实现大概如下。

```
//-------------------------------------------------
public abstract class Booking
{
  protected int covers ;
  protected Date date ;
  protected Booking(int c, Date d) {
    covers = c ;
    Date = d ;
  }
  public Date getDate() {
    return date ;
  }
  public void setArrivalTime(Time t) { }
  public void setCovers(int c) {
    covers = c ;
  }
}
//-------------------------------------------------
```

由于在设计类图中 booking 类是抽象的，所以它被实现为 Java 抽象类。定义了一个构造函数初始化属性的值，但是构造函数声明为 protected，所以这个类的实例只能由子类创建。注意，在类图中经常省略构造函数，但在类的实现中构造函数是必需的。类的属性在 Java 中用域表示， UML 的数据类型在必要的地方要转换为适当的 Java 的等价类型。类的操作在 Java 中定义为方法，并且应当有相应的实现，在这里相当琐细。

“setArrivalTime”方法只对 Reservation 有意义，将它包含在 Booking 超类中是使所有的预约共享一个公共接口，但是提供的是空实现，因而与它不相关的 walk-in 类可以忽略它。

在实现一个类时，需要考虑其成员的可见性。如果设计者没有指定可见性，那么实现属性和操作的经验规则是：将属性转换为实现类的私有域，将操作转换为实现类的公有方法。这反映了类的实现广泛采用的一种策略，它规定类的数据应该是私有的，并且只能通过它的操作接口访问。这个规则的例外是在该类的子类实例需要访问该类的属性时。在这种情况下通常指定属性的可见性是 protected，像这个例子中一样。

7.5.2 泛化

图 7.11 显示了餐馆预约系统设计中一个泛化的例子，这里在图 7.10 中定义的 Booking 类被代表两种不同预约的子类扩展。

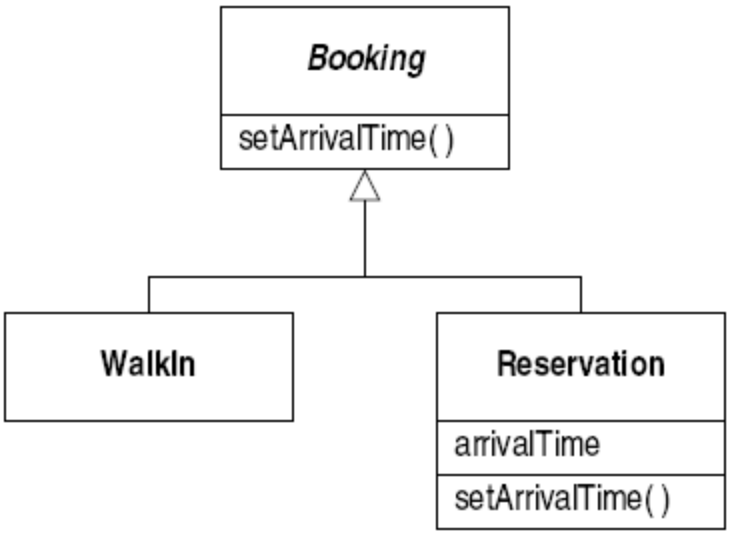


图 7.11　预约系统中的泛化

泛化可以用 Java 中的继承实现。UML 中的泛化允许无论哪里需要的超类实例都可以用一个子类实例代替，并从超类继承类的成员，而 Java 的继承语义保留了这两个特性，如下面“WalkIn”和“Reservation”类的大致实现所说明的。

```
//-------------------------------------------------
  public class WalkIn extends Booking
  {
    public WalkIn(...) {
      super(...) ;
    }
  }
  public class Reservation extends Booking
  {
    Time arrivalTime ;
    public Reservation(...) {
```

```
    super(...) ;
    arrivalTime = null ;
  }
  public void setArrivalTime( Time t ) {
    arrivalTime = t ;
  }
}
//-------------------------------------------------
```

“WalkIn”类在这里的实现相当小，因而可能会质问这究竟是否需要定义为单独一个类。支持这里遵循的方法的理由将在后续章节更详细地讨论。

7.5.3 类的重数

定义一个类时，默认情况下对系统可以创建的该类的实例数目没有限制。通常所需要的就是如此，例如，预约系统就需要创建和操纵预约类、餐台类和顾客类的多个实例。

然而，在某些情况下，规定只能创建某个类的一个实例是有用的：例如，用户界面和预约系统类，由于它们在系统中起着中心的协调作用，我们只想各要一个实例。如图 7.12 所示，用在类图标的右上角包含相应的重数，可以说明这个决定。只能实例化一次的类称为单件（*Singleton*）类。

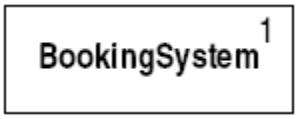


图 7.12　类重数的符号

使一个类只被创建一个实例的方式来实现类是可能的，实现的标准方法记录在称为单件（*Singleton*）的设计模式中。如果一个类作为单件实现，那么至多只能创建它的一个实例，并且这个实例可以由其他类在需要时获得。

单件模式的主要思想是使类的构造函数不可访问，因而使用单件类的类就不能创建该类的实例。单件类把单独一个实例作为静态数据域保存，客户可以通过一个静态方法获得该实例并在第一次调用这个静态方法时初始化这个唯一的实例。这个模式在预约系统类的应用由下面的代码要点说明。

```
//-------------------------------------------------------------
public class BookingSystem
{
private static BookingSystem uniqueInstance ;
public static BookingSystem getInstance() {
if (uniqueInstance == null) {
uniqueInstance = new BookingSystem() ;
```

```
}
return uniqueInstance ;
}
private BookingSystem() { ... }
}
//-------------------------------------------------------------
```

7.6 关联的实现

在类图中，关联把类联系在一起。尽管所有面向对象编程语言都提供了对类的概念的直接支持，但常用语言没有一种提供可以直接用以实现关联的特征。因此，在实现一个设计时，想一想如何处理关联的问题是必要的。

关联的作用是定义系统运行时连接对象的链接的特性。链接表明一个对象知道另外某个对象的存在和位置。此外，在必要时链接还可以作为信道，使消息沿着信道发送到链接的对象。

这提示了对象之间的链接可以使用引用实现。引用，实质上是对象的地址，当然记录了该对象的位置和本体，并且借助于引用可能调用到所链接的对象的成员函数，从而模拟了消息传递。因此，实现两个类之间的关联的一个简单策略就是：在类中声明一个域保存对所关联的类的实例的引用。

然而，关联具有某些逻辑特性，不能用这种有点过于简单化的方法处理，而需要在关联的实现中仔细考虑。这些特性在本节的剩余部分简单考虑。在后续章节将给出对实现关联的各种方法的更系统的处理。

7.6.1 双向性

关联连接类，链接连接对象，但是除非使用导航性注解，否则对链接能够在哪个方向遍历就没有限制。例如，协作图中的链接可以作为两个对象之间的通信信道，并且消息可以沿着这个信道在任一个方向发送，如设计需要所要求的那样。这种特性一般表达为关联和链接是双向的。

另一方面，引用是单向的。如果对象 *x* 持有对象 *y* 的引用，这就使 *x* 可以访问 *y*，并能够调用 *y* 的成员函数，但 *y* 对 *x* 根本没有任何了解。为了实现一个双向链接，需要两个引用，一个从 *x* 到 *y*，另一个从 *y* 回到 *x*。为了说明其不同，图 7.13 显示了三个对象之间的链接，以及与之对照的在两个方向上实现链接所需要的引用。引用通过表示访问方向的箭头与链接相区分。

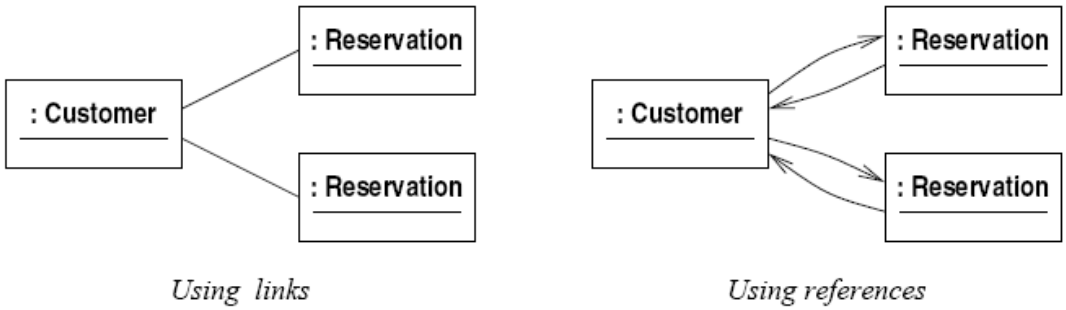


图 7.13 用引用实现双向链接

尽管的确可能以这种方式用一对引用实现链接，但是有许多理由说明这是不方便的，并应该尽可能避免。首先，为了支持引用的环状结构，相关的类声明必须互相引用。这只能以增加两个类之间的耦合为代价而实现，对结果代码的简单性和可维护性有不利影响。

其次，在链接的整个生命期中，实现链接的两个引用必须保持相互一致。它们必须一起被创建和销毁，并且不能被独立地改变。然而，对引用编程是很容易引起错误的，而所需引用数目的成双结对也增加了潜在问题。即使定义了维护引用的一个安全的方案，并始终坚持这个方案，维护两个引用也可能有相当大的开销。

7.6.2 关联的单向实现

幸而，依赖于系统的动态特性，通常并不需要在两个方向上实现关联。可以看到消息从预定发送到顾客对象，但是永远不会从顾客到预定。为了提供支持，很明显在预定中必然需要保存对其顾客的引用，但是让每个顾客对象都保存对他们的预定的引用，可能不能获得直接的益处。

因此，通过只在实际使用关联的方向提供引用，可以简化关联的实现。在实践中，需要在两个方向支持的关联相对地几乎没有。例如，在预约系统的第一次迭代中，消息从预定对象传递到顾客对象，但是决不会从顾客到预定。

因此我们可以决定，为了简化实现，只在从预定到顾客的方向上实现这个关联。这个决定可以用导航性注解的形式加入到类图中，如图 7.14 所示。

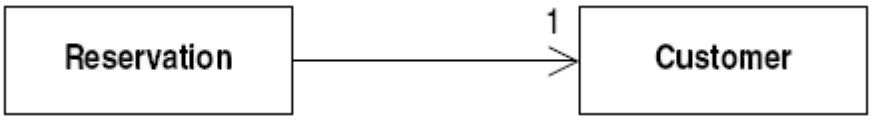


图 7.14 使关联成为单向的

这个关联简单的单向实现可以在预定类中用一个数据成员保存对所链接的顾客的引用，如下所示，同时还给出了构造函数中初始化该引用的代码。

```
//-----------------------------------------
  public class Reservation
```

```
  {
    private Customer customer ;
    public Reservation( Customer c )
    {
      customer = c ;
    }
  }
//-----------------------------------------
```

选择只在一个方向上实现关联，我们在目前的易实现性和未来的可修改性之间做了一个折衷。现在不可能在从顾客到预定的方向上遍历这个关联。未来对系统的任何修改，例如想要检索一个特定顾客的所有预定，将会比本来不用这种方法更难以实现，因为我们将不得不退回去增加反向遍历链接的能力。

7.6.3 实现重数约束

在用引用实现关联时，类图包含的关于关联的重数信息并不总是能自动地保持。例如，图 7.14 中的关联声明，每个预定对象总是和恰好一个顾客对象关联。然而，Java 中的引用可以始终具有 null 值，不引用任何对象。因此，如果在系统运行的任何时候都是如此，系统将与关联指定的重数不一致。

重数约束只能通过增加代码来保持，这段代码将在适当的地方明确地检查该变量没有存储 null 值。例如，如果试图创建一个没有与顾客对象相关联的预定，下面的代码会抛出一个异常。

```
//-----------------------------------
  public class Reservation
  {
    private Customer customer ;
    public Reservation( Customer c )
    {
      if (c == null) {
        // throw NullCustomerException
      }
      customer = c ;
    }
  }
//-----------------------------------
```

当关联指定一个大于 1 的重数时，如图 7.15 所示的例子，会出现与重数有关的问题。这里的问题是：单个变量显然不能保存对多个链接的对象的引用。

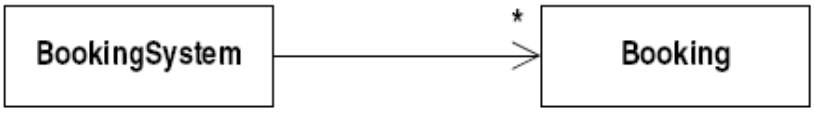


图 7.15　一对多的关联

为了实现一对多的关联，一个类必须定义一种适当的数据结构，以保存未指定数目的引用，而不是一个单引用变量。在 Java 中，各种容器类可以适用，ArrayList 类通常是一个适当而简单的选择，ArrayList 可以动态增长，以容纳所需数量的数据值。采用这种策略的“BookingSystem”类的代码概要如下。

```
//----------------------------------------
  public class BookingSystem ;
  {
    private ArrayList current = new ArrayList() ;
    public void addBooking(Booking b) {
      current.add (b) ;
    }
  }
//----------------------------------------
```

7.7 操作的实现

类图包含了静态信息，例如类的定义及其成员，可以通过将其转换为实现的结构特征实现。类图还包含了每个类中的操作的定义，但是没有表述每个操作进行的处理。这种动态信息包含在实现用例所产生的交互图中，有时候在以各个类为基础的状态图中得到总结。

在实现一个操作的时候，首先应该整理显示相应消息的交互图。这些图表明了当执行该操作时发送了其他哪些消息，因而该操作的实现应该调用哪些类的方法。如果不同的交互图显示了不同的行为，就必须实现适当的控制结构，在涉及的不同情况之间进行判别。此外，如果所考虑的类存在状态图，那么可以将状态图用作实现类中操作的指导，如下面所说明的那样。

7.7.1 状态图的实现

状态图提供了有关一个对象在其生存期中如何行为的动态信息。这些信息可以很容易地转换为代码，反映在类的方法的实现中，这些方法在被调用时需要知道对象当前的状态，以决定如何反应。

如同对类图那样，我们也希望用一种系统的方法将包含在状态图中的信息转换为代码。这节用图 6.15 所示的预约系统 BookingSystem 类的状态图举例阐明一种可能的方法。

这种方法的基本思想是：在预约系统类中定义的一个域中明确记录预约系统的当前状态。这个状态图中定义了两个不同的状态，“NotSelected”和“Selected”，这些状态被定义为预约系统类中的常量，并用一个“state”域保存系统当前状态所对应的值。

图 6.15 没有为系统指定初始状态，但合理的假设是在系统启动时没有被选择的预约，因而状态变量应该被设置为“NotSelected”。下面类的部分定义说明了这种情况，以及表示预约系统状态的常量值的定义。

```
//-----------------------------------
  public class BookingSystem
  {
    final static int NotSelected = 0 ;
    final static int Selected = 1 ;
    int state ;
    BookingSystem()
    {
      state = NotSelected ;
    }
  }
//-----------------------------------
```

发送给预约系统的消息有可能在不同的时间引起不同的行为，这取决于它当前的状态。通过将每个方法组织为一个 switch 语句，可以在类的实现中反映这种特性。switch 语句对状态变量的每个可能值都有一个 case，每个 case 的代码指定了系统在那个特定状态下如果接收到了该消息时做些什么。

如果遵循惯例，预约系统类中的每个方法将具有下面的模板指定的形式。在一些方法中，很多 case 可能都是空，不过，包含所有这些 case 会使实现的结构清晰一致并易于修改。

```
//-------------------------------------------------
  public void operation()
  {
    switch (state) {
      case NotSelected:
        // Specific actions for ‘Not Selected’ state
        break ;
      case Selected:
        // Specific actions for ‘Selected’ state
        break ;
    }
  }
//-------------------------------------------------
```

在每个不同的 case 中实现的特定行为有许多都能够直接来源于状态图中包含的动作，或者来自说明各种操作外部行为的顺序图。实现的详细资料最好通过餐馆预约系统的代码来研究。对状态图的这种实现方法的更详细描述后续实现策略章节中给出。

7.8 小结

- 许多软件开发都在应用框架的环境中进行。框架提供了半完全的、通用的应用程序，并使程序员避免使用低层的 API。
- 面向对象的框架典型地将定义一些“热点”类。为了开发应用程序，程序员必须特化这些类，并覆盖各种操作，以提供特定于应用的功能。
- 设计中的类自然映射到实现中的类，泛化则映射到继承。
- 编程语言中不包含与关联相同的特征。关联可以借助类中的域实现，该域保存对关联类实例的引用。
- 状态图可以通过使状态在实现中显式化，以系统地指导类中各个操作的实现。

7.9 习题

（1）假如这家餐馆将预约系统部署在多台 PC 上，每台 PC 都连接到一台单独的机器，该机器上维护预约信息的一个共享数据库。绘制一个部署图说明这种新配置。

（2）假如这家餐馆要改进预约系统，使预约信息可以显示在手持设备上，该设备通过无线网络连接到运行应用层的 PC 上。绘制一个部署图，说明系统的这种配置。

（3） 图 7.5、7.6 和 7.7 略微改变了 UML 的规则，在类图中包括了消息。对这些图各绘制一个相应的顺序图，图中显示框架类的一个实例和热点特化类的一个实例，以及在每次交互中将发送的消息。

（4）扩充预约系统的实现，以支持对预约的一般编辑，如在习题 4.9 和 5.10 的答案中描述的那样。

（5）扩充预约系统的实现，以支持超过餐台大小的预约的处理，如在习题 4.10 和 5.11 的答案中描述的那样。

（6）扩充预约系统的实现，允许调整预约的时间长短，如在习题 4.12 和 5.12 的答案中描述的那样。

（7）扩充预约系统的实现，允许为预约自动分配餐台，如在习题 4.13 和 5.13 的答案中描述的那样。

（8）扩充预约系统的实现，以支持等待名单的功能，如在习题 4.14 和 5.14 的答案中描述的那样。

（9）扩充预约系统的实现，允许对多张餐台进行预约，如在习题 4.16 和 5.15 的答案中描述的那样。

7.10 实践题

（1）你使用的 UML 建模工具是否支持自动代码生成（正向工程）和逆向工程？生成的代码是如何实现 UML 设计模型中的元素的？自动代码生成有什么功能局限性？

（2）选择一种开发工具平台和编程语言，实现你所设计的系统的一些核心类和基本框架。你为什么选择这种技术或语言？它们有些什么特性？为UML 设计模型的实现提供了一些什么支持？

第 8 章 类图和对象图

系统的静态模型描述系统所操纵的数据块之间在结构上的关系。它们描述数据如何分配到对象之中，这些对象如何分类，以及它们之间可以具有什么关系。静态模型并不描述系统的行为，也不描述系统中的数据如何随着时间而演进，这些方面由各种动态模型描述。

对象图（*object diagram*）和类图（*class diagram*）是两种最重要的静态模型。对象图提供系统的一个“快照”，显示在给定时间实际存在的对象以及它们之间的链接。可以为一个系统绘制多个不同的对象图，每个都代表系统在一个给定时刻的状态。对象图展示系统在给定时间持有的数据，这些数据可以表示为对象、在对象中存储的属性值或者对象之间的链接。

理解系统的一个方面是了解哪些对象图表示了系统可能的有效状态，哪些对象图表示的是无效状态。例如，考虑图 8.1 所示的两个对象图，它们描述一个假想的大学档案系统中的学生和课程对象。

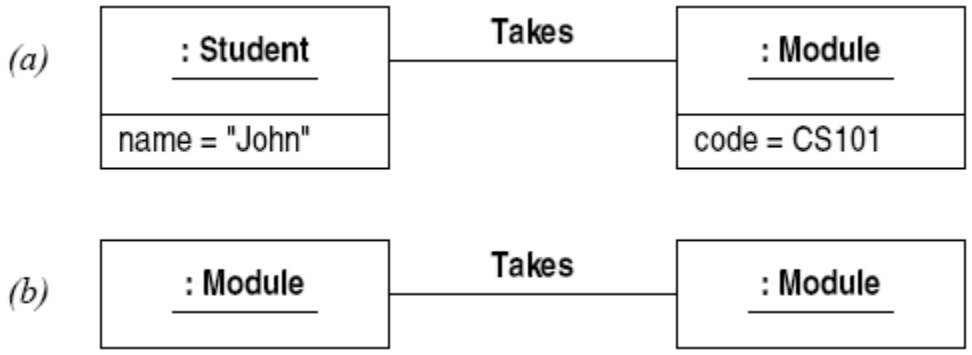


图 8.1 有效对象图(a)和无效对象图(b)

图 8.1*(a)* 显示的是特定一名学生选修一门课程的情形。假设系统的责任之一是记录哪些学生选修哪些课程，那么这个对象图就表示了系统的一种合法状态。图 8.1*(a)*所示的特定链接必定会有不存在的时候，甚至在系统运行的整个时间，都根本不会出现这个特殊的对象结构，但是，它的确表示了可能出现的情形，如果现实世界的实际情况需要这种情形。

图 8.1*(b)*的情况则不同。一门课程选修另一门课程是没有意义的，因此在大学档案系统的例子中这个图是完全没有意义的。我们不能问它描述的情形是真或是假，因为它根本不能描述一种有意义的情景。

然而，想要规约系统，不可能通过考虑所有可能的对象图并将它们分为合法或不合法来进行，因为的确存在着太多的对象图。为了说明任何表示系统合法状态的对象图都必须具有的性质，需要一种更一般的方法。为此，UML 使用类图。

类图作为一种系统规约， 除了别的方面以外，它规定可以存在什么类型的对象，这些对象封装什么数据，以及系统中的对象如何彼此关联在一起。例如，学生档案系统的一个适当类图可以清楚地表明，图 8.1*(a)*表示了系统的一种可能状态，而 8.1*(b)*是不可能的。

本章描述 UML 类图的特征，并说明如何用类图指定软件系统某些结构上的特性。本章使用的例子是假想的，只是为了讨论系统的静态结构，而丝毫不涉及该结构应该支持的处理。这种简化在介绍表示法的细节时有好处，但是不能作为如何使用和开发类图的实际可行的说明。如第 5 章的例子所指出的，类图的开发通常与交互图密切相关，交互图有助于标识系统中所需要的类。

8.1 数据类型

和众多程序设计语言一样，UML 定义了许多基本*数据类型*（*data type*），也提供了用以定义新类型的机制。数据类型代表简单的、无结构的数据种类，例如数值、字符和布尔值。数据类型一般用于指定类中的属性的类型和操作参数的类型。

数据类型的实例称为*数据值*。数据值不同于对象，数据值没有本体的概念。例如，整数 2 的两个不同当前值，存储在不同的位置，会被认为是相同的数据值，并且在检测是否相等时会返回“true”。但是，两个即使状态相同的对象也被认为是不同的。

UML 中可以使用的数据类型分为三类。首先，有许多预定义类型，主要有整数、字符串和布尔值。UML 定义这些类型的表示法如图 8.2 所示，实际上只是在类型表达式中使用其名字。UML 中没有定义这些类型的值，但假定是不言自明的。

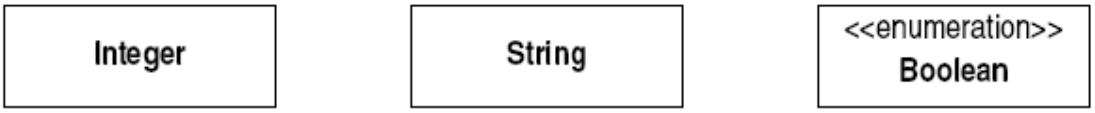


图 8.2　UML 中的预定义数据类型

数据值没有结构，所以数据类型不定义属性或操作。然而，可以为数据类型定义纯函数：它们返回数据值，但是不能修改参数。

数据类型还可以定义为枚举（*enumeration*），实质上和许多编程语言提供的枚举类型相同。枚举的值是一组命名的*枚举字面值*。布尔值用一个预定义的枚举表示，如图 8.2 所示。该类型的字面值是 true 和 false 两个值。用户定义的枚举可以通过将枚举字面值列举在图标下部的栏中定义，如图 8.3 所示。

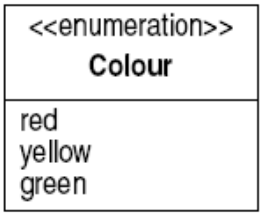


图 8.3　用户定义的枚举

最后，模型可以使用*编程语言数据类型*。这就允许特定编程语言的类型可以在 UML 中使用。这在从代码逆向构造一个类时，或者在实现设计时（例如可能必须指定使用指针或数组类型）非常有用。

8.1.1 重数

在 UML 中，有许多地方必须声明给定实体在某些情况下可以出现多少次。这用*重数*（*multiplicity*）表示。重数是一个整数集合，该集合的每个成员代表指定实体可能出现的数量。

在 UML 中重数作为数据类型来定义，用*重数范围*表示。范围由用圆点隔开的一对整数表示，例如 `0..9`。特殊符号“`*`”的意思是“无限制的”，可以用于表示没有上界的范围。

因此，`0..100` 表示从 `0` 到 `100` 的所有整数，包括 `0` 和 `100`，而 `0..*`表示所有的非负整数。两端数字相同的重数范围，如 `1..1`，用单个的数字 `1` 表示。

通常，重数由包含数字范围和单独数字的列表给定，最好以升序排列。例如，假定一个给定实体是可选的，但如果出现，则至少必须是三个。这个重数可以表示为 `0,3..*`。

8.2 类

简化描述无限数量的对象的基本技术是将相似的对象分组。共享某些特性和行为的对象放在一个组中，并为每个组定义一个*类*。类不描述个体的特殊性质，而是指定组中所有对象共享的公共特征。

例如，图 8.1 显示了代表假想的大学档案系统中的学生和课程的对象。在这样一个系统中，将会为每个学生存储相同的一些信息，如姓名和地址，而对各个学生实际存储的数据值则不同。通过定义一个学生类，我们可以一次性地规定系统将存储的关于学生的信息的结构。

这同样适用于操作：例如，修改地址的操作，潜在地适用于任何学生，或者换句话说，每个学生对象必须将此作为自己的操作提供。通过将操作描述为学生类的一部分，我们说明了它对所有学生对象的适用性。

在 UML 中，用一个矩形框来图形化地表示一个类。类的名字用粗体写在矩形框的上部。接下来的几节将描述表示类的对象的特性的方法。然而，在许多情况下，只将类的名字包含在图标中就已足够。

由一个类描述其结构的各个对象称为该类的*实例*。另外，也可以说这些对象“属于一个类”。每个对象都是某个类的实例，这是对象定义的不可缺少的部分，类决定了该对象中将存储什么数据。为了强调类和实例之间的关系，UML 用相同的图标表示对象和类。在类名前加冒号，并加下划线，而且实例也可以命名。图 8.4 显示了一个“Student”类以及它的两个实例。

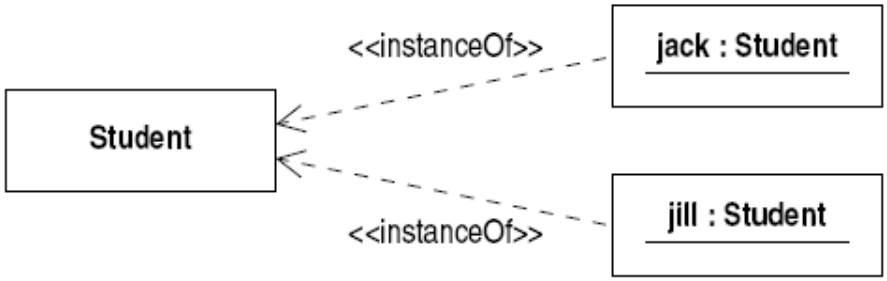


图 8.4 类及其实例

图中的虚线称为依赖（*dependency*），是 UML 表示两个模型元素之间未指定种类的关系的表示法。依赖上所附的标注，或构造型（*stereotype*），指出描述的是什么特殊关系。图 8.4 中，“*instanceOf*”依赖表明对象是指定类的实例。

然而，对象是指定类的实例的事实已经通过在每个对象的标注中使用该类的名字表达了，所以图 8.4 所示的表示法很少使用。实际上，类和对象图标在正常情况下是不会一起出现在同一个图中的。

8.2.1 类重数

重数的概念可以应用于类：类的重数规定了在任一给定时间可以存在的该类的实例数。类的缺省重数是“零或多个”，意指可以创建的实例数目没有限制。缺省重数不需要在图中表示出来。

但是，在有些情况下，想要对系统运行时任何给定时间能够存在的实例数目规定某些限制。为了做到这点，类可以包含一个重数注解，放在类图标中的右上角。这种表示法最常见的使用是说明一个类是单件类，即只能有一个实例。例如，学生档案系统可能需要记录大学本身的某些信息，因而要定义一个类保存这些信息。但是，因为系统完全是在一所大学内部，所以这个类有多个实例是没有意义的。这个约束用图 8.5 所示的类符号表示。

图 8.5 一个显示类重数的单实例类

8.3 用类描述对象

每个对象都包含某些数据以及若干处理这些数据的操作。给定类的所有对象包含相同的数据项和相同的操作，虽然数据项具有不同的实际值。这种公共结构通过对象所属类中的若干特征定义，包括属性（*attribute*）和操作（*operation*），属性描述的是类的实例中的数据项。

8.3.1 属性

属性是对类的每个实例所维护的数据域的描述。属性必须命名。除了名字，还可以提供其他信息，例如属性描述的数据的类型，或者属性的缺省初值。属性在类图标中类名下面的一栏中显示。如果显示了属性类型，类型和名字之间用冒号隔开，后面写等号和初值。但是，在设计的早期阶段，最常见的是只显示属性的名字。图8.6的例子说明了类的属性的表示法。

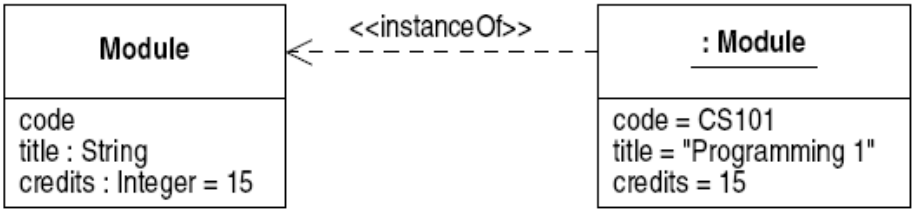


图 8.6 属性和属性值

图 8.6 显示了表示课程的一个类 Module，它有三个属性对课程的编号、名字和学分建模。图中还显示了该类的一个实例，用以说明在对象图标中指定属性值的方式。第一个属性名为“code”，表示该大学内用以引用该课程的标识符，这个属性没有指定类型，表示还没有决定如何将此代码存储为具体的数据。

属性的类型可以用数据类型表示，通常却不用类作为属性的类型。例如，如果一个模型中定义了表示学生和课程的类，那么将学生选修的课程作为学生类中的一个属性，其类型为“Module”，这似乎很有吸引力。对这种情况建模更好的方法是使用学生类和课程类之间的一个关联，如 8.4 节描述的。

图 8.6 中显示的属性具有实例作用域，意思是类的每个实例可以存储该属性的一个不同值。然而，有些数据描述的不是个别实例，而是所有当前实例的集合。例如，我们可能想记录系统所知的课程的总数，一种方法如图 8.7 所示。

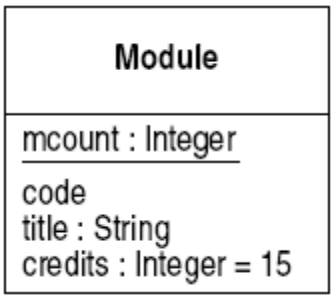


图 8.7 具有类作用域的属性

加下划线的属性，例如图 8.7 中的“mcount”，称为具有类作用域。这意味着该属性只有单独一份，可以由该类的所有实例访问。具有类作用域的属性对应于编程语言中的静态（static）变量或类变量。

属性也可以有明确定义的重数。属性的缺省重数是“正好一个”，暗指类的每个实例正好保存每个属性的一个值。但是，在有些情况下，这个缺省值并不合适。假定扩充课程类，包含一个课程结束时记录考试日期的属性。因为有些课程可能没有这样的考试，类的一些实

例不需要保存数据值，在此意义上这个属性就是可选的。这可以通过在属性名字后面增加重数注解来表示，如图 8.8 所示。

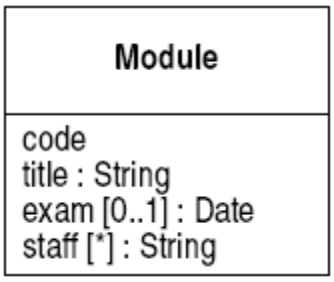


图 8.8 属性重数

图 8.8 中还出现了一个属性 staff，其重数是“多个”，目的是保存讲授该课程的教员的姓名。具有大于 1 的重数的属性在许多方面等价于一个数组，这也可以直接用适当的语言类型指定。

8.3.2 操作

和属性一样，操作也可以显示在类图标中。这些是类的每个实例都提供的操作，例如，课程类可能提供了设置和检索课程名称以及学生报名选修该课程的操作。操作显示在类图标底部的一栏中，如图 8.9 所示。在类图标中，属性和操作栏可以都省略，或省略二者之一，并且空栏不必在图标中显示。

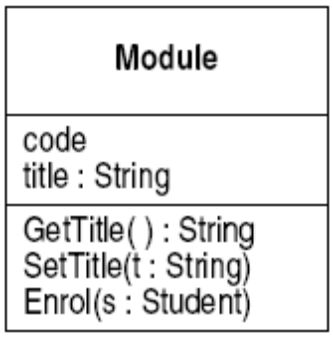


图 8.9　带有属性和操作的课程类

操作有名字，还可以有参数和返回结果，如同编程语言中的函数一样。操作的参数和返回类型可以是数据类型的名字，像指定属性类型所使用的那样，或者是类的名字。和属性一样，除了操作名字之外，其他的所有信息都是可选的。根据开发过程到达的阶段，可以提供适当的或多或少的信息。

在类图中省略操作很常见。原因是在孤立地考虑一个类时，很难决定该类应该提供什么操作。必要的操作是通过考虑系统的全局行为如何由组成系统的对象网络实现而发现的。这种分析是在构造系统的动态模型时进行的，并且只有接近设计过程结束时才可能汇集一个类的操作的完整定义。

图 8.9 中的操作具有实例作用域，意思是它们将应用于该类的各个实例上，而且只能在已经创建了实例时调用。也可以定义具有类作用域的操作；这些操作可以独立地被该类的任何实例调用，但作为结果它们只能访问同样具有类作用域的属性。图 8.10 显示了一个操作，返回当前存在的课程类的实例数目。

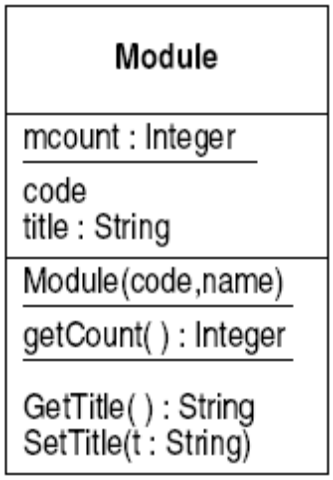


图 8.10　具有类作用域的操作

在 UML 中，构造函数（即创建类的新实例的操作）也具有类作用域，尽管它们有些和其他操作稍微不同的特性。图 8.10 显示了一个构造函数，遵循一般惯例，构造函数和类具有相同的名字。

8.3.3 标识对象

在第 2 章，对象本体（*identity*）的概念是作为对象模型的一个本质部分引入的。对象本体是对象一个无须明确表示的隐含性质，它使得该对象能够和系统中的其他所有对象区分开来。在面向对象程序设计语言中，对象在内存中的地址不能由任何其他对象共享，通常被用作对象本体的实现。

对象本体不用 UML 的数据类型表示，因此不能在模型中明确给出。特别地，对象的本体和对象的任何属性截然不同。很可能两个不同对象的所有属性的值都相同，如图 8.11 所示，但仍然是不同的对象，可以由它们的不同本体区分。

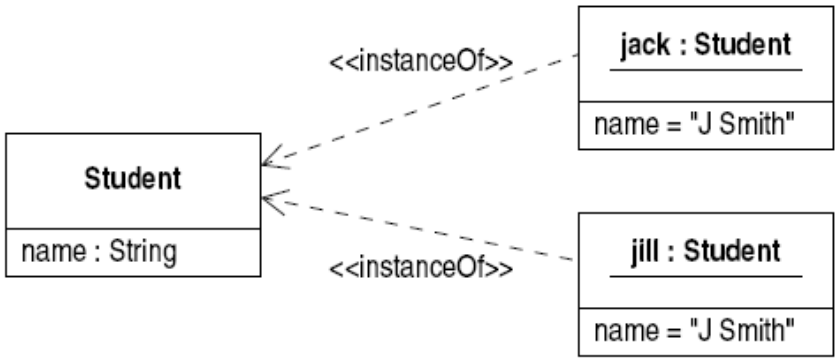


图 8.11 状态相同的不同对象

在模型中，对象本体文字上可以用在该对象的类名前的“对象名”表示，譬如图 8.11 中的“Jack”和“Jill”。然而对象名完全不同于对象可以有的任何属性。

当然，在很多情况下，对象的确具有标识性的属性。例如，学生在注册一门课程时可能分配一个唯一的识别号码，这个识别号码是现实世界的一个数据，印在学生的证件上。因此它非常适合包含在模型中，作为学生类的一个属性，如图 8.12 所示。然而，这样的属性并不能代替本体，每个“Student”类的实例既有本体又有一个明确的识别号码。而且要注意，图中并没有明确该类的每个实例都应该具有一个不同的识别号码。

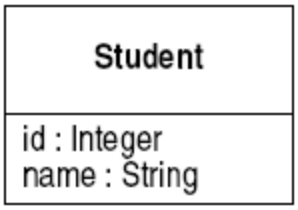


图 8.12 具有标识性属性的类

8.3.4 特征的可见性

类中的每个特征可以有一个相关的可见性（*Visibility*），指定该特征可以被其他类利用的程度。在设计语言中很少应用可见性的概念，除非是在定义方法体。然而，该表示法的确允许明确定义程序语言类的对应特性，并且 UML 的可见性符号的含意本来就源自于诸如 Java 和 C++这样的语言中的等价表示法。

UML 定义了 4 种可见性，即公有（public），保护（protected），私有（private），以及包（package）。私有特征只能在该特征的所属类中使用，保护特征还可以在其所属类的子孙类中使用，而公有特征可以在任何类中使用，具有包可见性的特征可以由所属类的同一个包中的其他类使用。

可见性的图形化表示是在特征的名字前面加一个符号，例如图 2.1 所示。公有属性用“+”表示，保护用“#”，私有用“－”，而包用“~”。对可见性没有定义缺省值，所以如果类中没有显示可见性，就认为是未定义的。然而，常见的经验规则是假定属性是私有的，如果要在子孙类中使用则是保护的，而操作是公有的。

8.4 关联

上一节解释了如何用类描述系统中发现的各种不同对象。对象图另一个主要的结构特征是对象之间存在链接（*link*）。如同将相似的对象集合在一起并用单个类描述一样，相关的链接也可以用一个单独结构描述，称为关联（*Association*）。

两个对象之间的链接是对象之间的某种联系的模型，如一个学生选修一门特定的课程。通常，由链接表达的思想可以用一种更一般的有关类之间的关系描述。在这个例子中，该关系是学生可能选修课程。这种更抽象的关系由关联建模。

因此关联涉及了多个类，并对类之间的关系建模。关联在 UML 中用连接相关类的直线表示。例如，图 8.13 显示了一个关联，对公司雇用人员的建模。

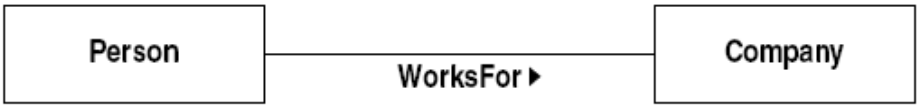


图 8.13　对雇用建模的关联

这个关联标注有一个名字“WorksFor”，显示在关联的中间。选择关联的名字时，通常要使之和相关类的名字连接在一起产生的结果接近于描述该关联的有意义的英语句子，如“Person works for company”。在这个例子中，该关联代表了人为公司工作的事实。

关联名字旁边的小三角形指出了读这个句子的方向。在这个例子中这是必要的，因为逆向的关联，即公司为人工作，也是有意义的关系，但是完全不同，所以有必要对这二者进行区分。

8.4.1 链接

两个类之间存在关联说明这两个类的实例在运行时可以链接。图 8.14 给出了图 8.13 所指定的关联的一个例子。链接可以用相关的关联名标注，加下划线以强调这个链接是该关联的一个实例。

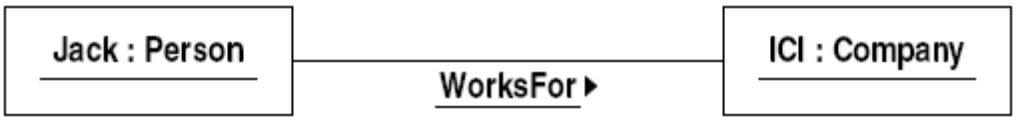


图 8.14 雇用关系的一个实例

8.4.2 关联端点的特性

除了名字之外，与关联有关的大部分信息都在关联端点定义，即关联与类相交的位置。关联的每个端点都可以标注一个角色名（*role name*）。角色名经过选择，描述从关联另一端的视角看到的这一端的对象。例如，在公司工作的人会很自然地将公司描述为自己的雇主，那么这就是关联的 “Company”端的一个合适的角色名。同样地，另一端可以标注角色名“雇员”。

关联的两端都可以指定重数，该重数指定关联另一端的类的一个实例可以链接多少个对象。图 8.15 显示了“worksFor”关联，并增加了角色名和重数注解。这个图规定一个人只能为一个公司工作，而一个公司可以有 0 或多个雇员。

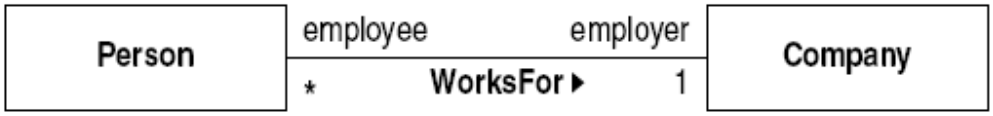


图 8.15 关联两端的注解

像这个例子暗含的一样，重数注解可以和角色名连在一起读，或者与关联的类的名字一起读，对关联模型化的关系进行更详尽的描述。然而，重数注解的准确含意与给定时间可以存在的链接的数目有关。

例如，图 8.15 中的关联规定 Person 类的任何给定实例必须链接到恰好一个 Company 类的实例。由此推断，图 8.16 所示的情形有两处是不合法的：一是表示 Jack 的对象同时链接到了两个公司对象，二是表示 Jill 的对象没有链接到任何公司对象。

图 8.15 中的图还表明一个公司对象必须链接到零或多个人对象。实际上，这对公司对象可以具有的链接数目根本没有限制。

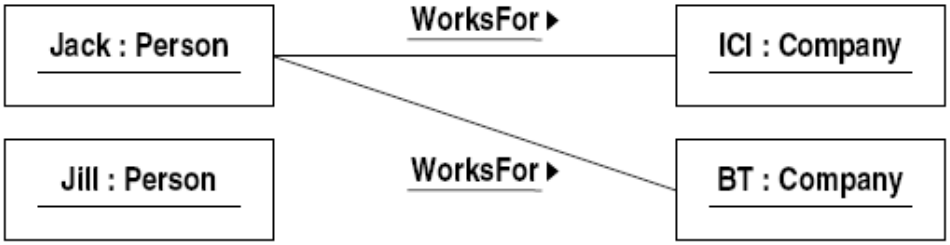


图 8.16　违反重数规约

在类的属性和连接到类的关联的另一端之间，存在着许多类似之处。例如，一个人的名字及其所工作的公司都是系统必须记录的关于这个人的数据，并且很自然地都可以看作这个人的特性。UML 中的惯例是将具有数据类型值的特性作为属性建模，而将值是另一个类的实例的特性作为关联建模，但这并不应该隐藏这两个概念之间的共性。

另外的一个类似处是，像属性一样，关联端点可以附有可见性注解，写在角色名之前。可见性符号及其含意和为属性定义的一样。如果必要，关联端点也可以定义为类作用域。

8.4.3 导航性

导航性（*navigability*）的概念是在第 2 章介绍的，在第 2 章导航性用于记录系统中的某个链接只能从一个方向遍历，或只能从一个方向发送消息的事实。导航性用链接上的箭头表示，并在相应的关联中使用同样的箭头。

没有箭头的关联，如图 8.13 所示的，通常被认为是在两个方向都可以导航的。注意：关联名字上的三角形符号定义了读关联名字的方向，与导航性没有丝毫关系，导航性是由关联自身上的箭头定义的。如果后来明确地决定雇员对象不需要保存相关公司对象的引用，这就表明，关联可能要重定义为只在一个方向可导航，如图 8.17 所示。

图 8.17　只在一个方向可导航的关联

这并没有改变关联的含意：系统仍然将雇员对象和他们工作的公司联系在一起。然而，它只是说没有必要直接查找给定一个人工作的公司，并且不能用这个关联的实例直接从 Person 对象向 Company 对象发送消息。

8.4.4 不同种类的关联

一般的 UML 表示法允许一个关联连接任意多个类，如 8.10 节所讨论的。然而，实际上使用的关联大多数是二元的，只连接两个类。原则上，任何情况都可以只用二元关联建模，而且与涉及大量类的关联相比，二元关联更容易理解和用常规编程语言实现。

二元关联一般连接两个不同的类，但是在很多情况下，对象也可以链接同类的对象。这种情形可以通过使用自关联（*self-association*）建模，即两端连接到同一个类的二元关联。图 8.18 显示了“Person”类的一个自关联的例子和由此导出的一些可能的链接。两个对象之间的链接表示一个人是另一个人的经理。

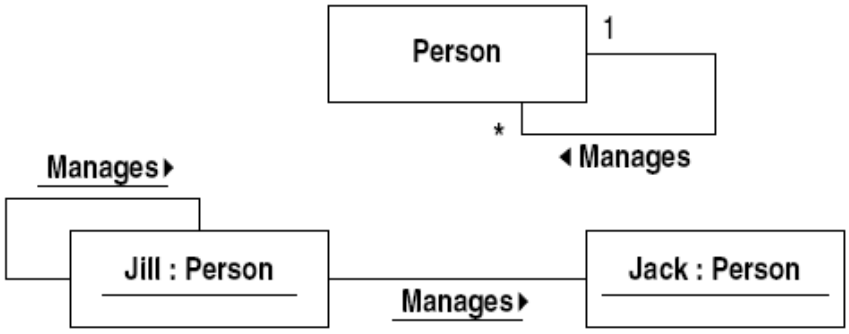


图 8.18 类的自关联

“Manages”关联说明每个人有恰好由一个经理，包括某些人是自己的经理的可能性，如图 8.18 中的 Jill。一般地，一个自关联为对象链接到自己留有可能性，如果这对特定的关联没有意义，可以使用一个明确的约束将其排除在外。

8.4.5 标注关联

重数信息通常应该显示在关联上，但关联名和角色名是可选的。设计人员可以选择必要的详细级别，以使图容易理解。

一种需要某些文字标注的情形是同一对类之间有多个关联的时候。在这种情况下，需要标注来说明关联，并保证对每个关联理解正确的重数。图 8.19 显示了 Person 类和 Company 类之间的另一个关联，表示人在作为公司雇员的同时还可以是公司的顾客。注意，设计者在这里如何决定用一个角色名标注足以让每个关联的含意都清晰。

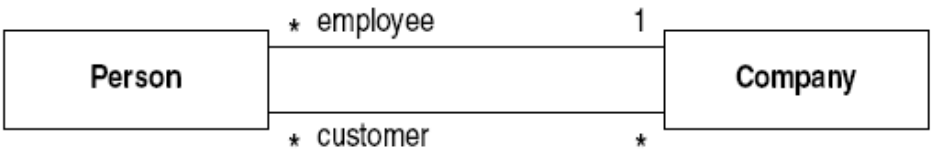


图 8.19 区分两个关联

角色名有助于区分自关联的两端，如图 8.20 所示。

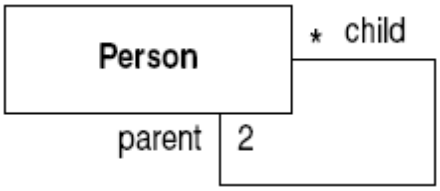


图 8.20　区分自关联的两端

8.4.6 具体化关联

考虑图 8.15 定义的“WorksFor”关联，假设一个人作为不同的角色，被同一公司雇用了两次。试图表示这种情形的对象图在图 8.21 给出：例如，也许 Jill 被雇用为在白天当讲师，因为收入微薄，在晚上不得不兼职作吧台服务员。

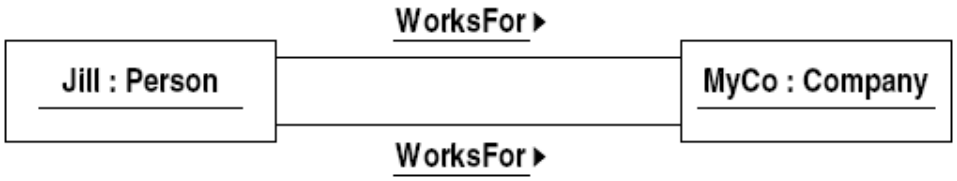


图 8.21 被同一公司雇用两次

是否允许用“WorksFor”关联描述这个对象图表示的情形实际上并不清楚。关联的重数表明 Jill 只能为一家公司工作，而争论在于图 8.21 中的图保留了这个特性。Jill 可以有两份合同，然而是和同一家公司 MyCo 订立的，所以表面上她只为一家公司工作。另一方面，Jill 具有两个“WorkFor”链接：这似乎容许了为多个公司工作的可能性，虽然在这个特殊情况中显示的两个链接连到同一个对象。这可能暗示，描述的这种情况只是偶然地与图 8.15 中的重数一致，因此描述的情形是不合法的。

答案完全依赖于 UML 是如何定义链接的。链接通常被理解为对象的简单元组，如同一组坐标或关系数据库表中的一行。在二元关联的情况下，这意味着链接只是由该链接连接的对象对，如（Jill, MyCo）。这进一步隐含着图 8.21 中的两个链接是相同的，因为它们链接的是同一对对象，因此，不能用该图记录两个不同的雇用合同。如果这是该图的目的，那么即使最乐观地说也是令人误解的，因为它暗示这对对象之间有两个不同的链接。

因此，链接只是记录两个对象之间是否拥有一种特定的关系。不能存在同一个关联有两个链接实例连接的是同一对对象，而必须寻求另外的某种方法对类似于图 8.21 中的情形建模。

我们需要一种方法区分两个链接，即使它们共享相同的特性。如果链接就像对象一样可以具有本体，这就有可能，但是元组只是由它们的组成成分确定，所以没有这样的本体。

然而，一种解决方案是用具有本体的对象代替链接。为此，我们需要找出一个似乎合理的类，它的实例和我们希望描述的链接一一对应。在这个例子中，Jill 可能对她的各个工作

签订不同的合同，如果是这样，表示合同的类就可以代替简单的“WorksFor”关系，如图 8.22 所示。

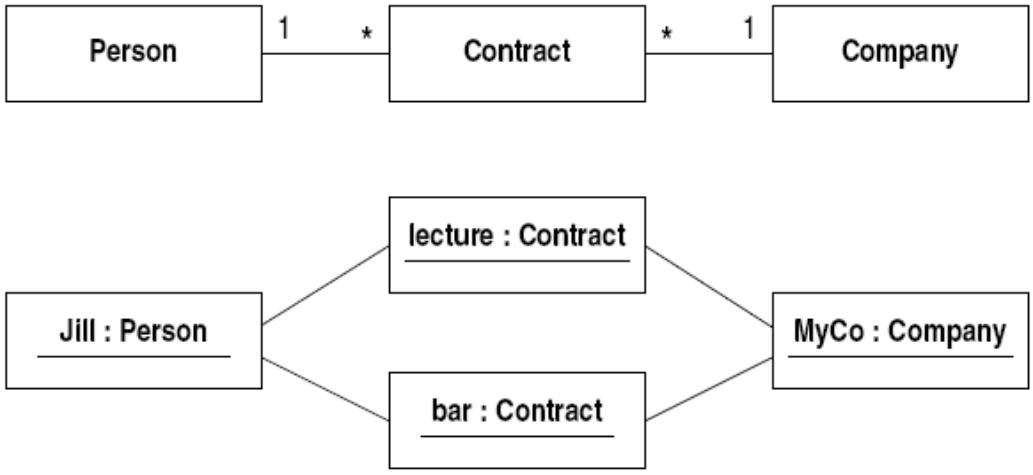


图 8.22 用一个中间类解决多重链接

术语具体化（*reification*）有时用来指这种用类代替关联的技术。除了允许多重链接，该技术还允许与关系相关的信息作为新的中间类的属性保存，例如一个特定合同支付的薪水。

然而，具体化可能改变模型的含意。在新模型中，一个人可以有多份合同，但并不妨碍这些合同是和两个或多个公司签订的。最初的“WorksFor”关系并不允许这种情况，它只允许一个人为单独一家公司工作。这个限制可以通过在图 8.22 中增加一个明确的约束表达，如后续章节所说明的。

具体化通常在数据库设计中用于消除模型中的多对多的关系。对每个多对多关系，引入一个新链接类，并用两个一对多关系代替该多对多关系，例如从图 8.15 到图 8.22 的转换。然而，在 UML 中没有消除多对多关系的特殊需要，因而通常只是在有必要产生一个更准确的模型时才进行具体化。

8.5 泛化和特化

应用程序中包含许多密切相关的类，这很常见。这些类可能或许共享一些特性和关系，或许可以自然地把它们看作是代表相同事物的不同种类。例如，考虑一个银行，向顾客提供各种账户，包括活期账户、存款账户以及在线账户。该银行操作的一个重要方面是一个顾客事实上可以拥有多个账户，这些账户属于不同的类型。对此建模的类图在图 8.23 中给出。

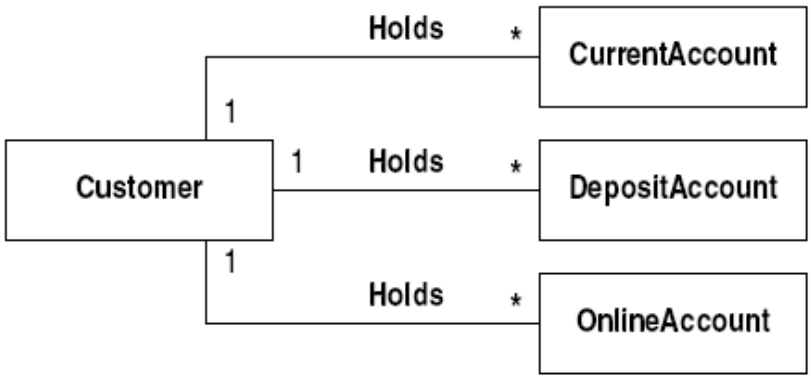


图 8.23　对银行账户建模的类图

然而，这个模型中存在两个重大问题。第一，模型中的关联过多。从顾客的角度看，拥有一个账户只是一种简单的关系，并不会因为可以拥有不同种类的账户受到很大的影响。但是，图 8.23 中的模型将此用三个不同的关联建模，因而破坏了模型概念上的简单性。更糟的是，如果增加一种新类型的账户，就不得不向模型中增加一个新关联，以允许顾客持有新类型的账户。

第二，不同种类的账户被作为完全不相关的类建模。然而，它们很可能会有大量的公共结构，因为它们可能定义许多类似的属性和操作。如果我们的建模表示法可以提供一种方式明确地表示这种公共结构，那将是非常理想的。

利用泛化（*generalization*）的概念，可以克服这些难题。所有面向对象的设计表示法都提供了泛化的概念，这个概念与程序设计语言的继承概念密切相关。使用泛化可以重画图 8.23 所示的模型，解决上面讨论的两个缺陷。

直觉上，这个例子中接下来的是，我们对账户是什么以及持有账户涉及什么，要有一个一般概念。除此之外，我们可以设想一系列不同种类的账户，像上面所列举的那些一样，尽管它们有差异，却仍共享大量的功能。我们可以将这些直觉形式化，定义一个一般的“Acount”类，对各种账户共有的东西建模，然后将代表特定种类账户的类表示为这个一般类的特化（*specialization*）。

因此，泛化是一种类之间的关系，在这种关系中，一个类被看作是一般类，而其他一些类被看作是它的特例。图 8.24 显示了一般类“Account”和代表图 8.23 中提到的特殊账户类型的三个类。泛化用一个从特殊类指向一般类的箭头表示，如果有多个特殊类，可以从每个向一般类画一个箭头，另一种方法是可以将这些箭头合并成有多个“尾巴”的一个单箭头，如图 8.24 所示。

更一般的类被称为超类，而特殊的类被称为它的子类。术语“泛化”暗含着一种从子类到超类的观点，试图创建一个更一般的类表示一组类的某些公共特征。从另一个方向看，有时更自然地看作是创建反映一个类的特殊情况的一组子类。这个过程被称为特化，有时被用作泛化关系的另一个名字。

泛化纯粹是类之间的关系，并不指定这些类的实例之间任何种类的链接或关系。因此，角色名、重数和可见性符号并不适用于泛化。唯一可以附在泛化关系上的标注是判别器

（*discriminator*），描述区别各种子类的特性。图 8.24 中泛化的一个可能的判别器或许是提供不了多少信息的“账户类型”。然而，判别器很少在简单的泛化情况中使用，在此它们不能增加多少该图表达的信息。

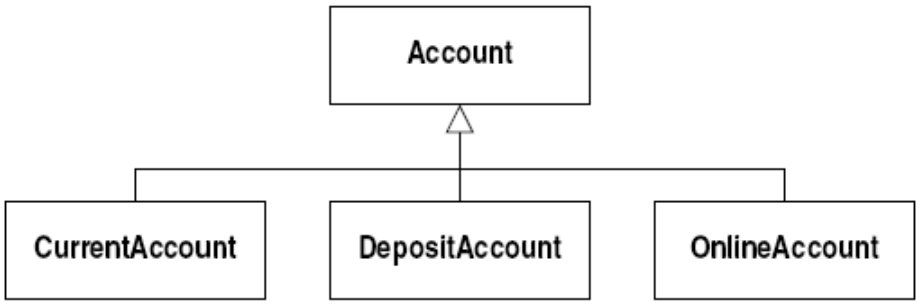


图 8.24 使用泛化的银行账户类

8.5.1 泛化的含义

对泛化的含义的一种常见说法是它表示了类之间的分类关系，尤其是用“是一种”（‘is a kind of’）所表达的关系。所以，例如我们可以正确地说，“活期账户是一种账户”，这一事实表明这两个类由泛化关系连接是适当的，如图 8.24 中那样。

但确切地说，这个定义给了泛化一个非形式的解释，它是以被建模的现实世界的实体为基础的。它建议在建模中什么时候适合使用泛化，但是泛化在模型中使用时是什么含意，还需要一个更精确的说法。

在 UML 中，泛化是通过可替换性（*substitutability*）的概念阐述的。这意味着在任何需要一个超类实例的地方，都可以毫无问题地用一个子类实例来替代。利用可替换性，我们可以简化图 8.23 中的图，用一个关联代替图中的三个关联，如图 8.25 所示。直观上，这个图规定顾客能够拥有任何数目的账户，并且，这些账户可以是如子类所定义的各种不同类型的账户。

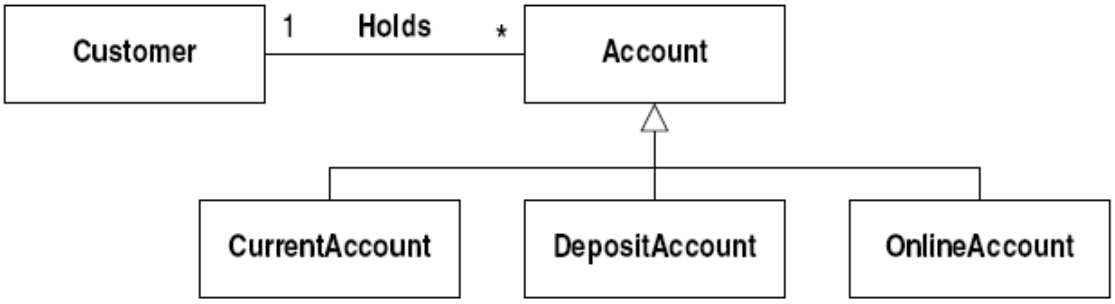


图 8.25 “拥有账户”的单一关系

这种替换如下。图 8.25 中的关联隐含着顾客“Customer”和账户“Account”类在运行时可以链接。因为可替换性，这些链接中的任何“Account”实例都可以用“Account”的任一子类的实例代替。这个替换过程可能产生如图 8.26 所示的链接，图中顾客链接到他们实际拥有的各种账户。

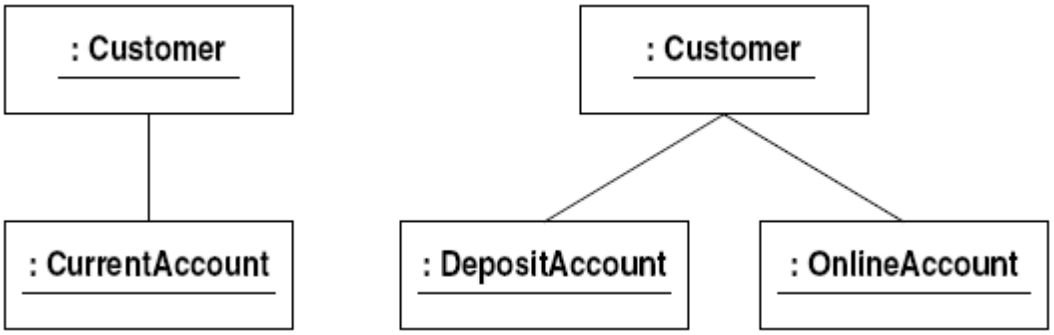


图 8.26　拥有各种账户

比较图 8.25 和 8.26，显然在某些情况下，即使没有明确的连接两个对象所属的两个类的关联，两个对象仍然可能链接。在有一个关联连接到这些链接对象的超类的情况下，这就是可能的。换句话说，图 8.26 中的顾客 Customer 和活期账户 CurrentAccount 对象之间的链接是图 8.25 中的 Customer 类和 Account 类之间的关联的一个实例。

这种现象是多态性的一种形式。多态性的意思是“很多形式”或“很多形状”，是面向对象编程语言的一种普遍特征。在这个例子中，“很多形式”指的是账户类的各种子类。

8.5.2 抽象类

在模型中引入超类通常是为了定义一些相关类的共享特征。超类的作用是通过使用可替换性原则对模型进行总体简化，而不是定义一个全新的概念。可是结果发现，常见的是不需要创建层次中的根类实例，因为所有需要的对象可以更准确地描述为其中一个子类的实例。

账户层次为此提供了一个例子。在一个银行系统中，可能一个账户必须是活期账户、存款账户或其他特定类型的账户。这意味着不存在作为根类的账户类的实例，或更准确地说，在系统运行时，不存在应该要创建的“Account”类的实例。

诸如“Account”这样的类，没有自己的实例，称为抽象类。在 UML 中将类名字写成斜体来表示抽象类，如图 8.27 所示。

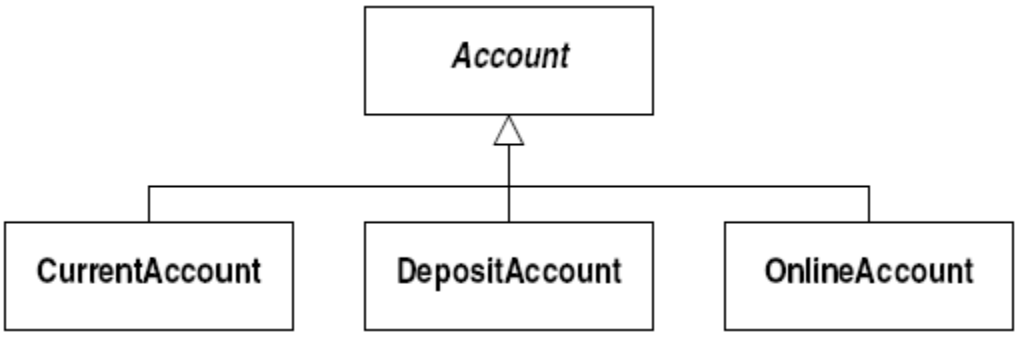


图 8.27　具有抽象根类的账户层次

不应该因为抽象类没有实例，就认为它们是多余的，就可以从类图中除去。抽象类，或层次中的根类的作用，一般是定义所有其子孙类的公共特征。这对于产生清晰和结构良好的

类图效果显著。根类还为层次中的所有类定义了一个公共接口，使用这个公共接口可以简化客户模块的编程。像有实例的具体类一样，抽象类能够提供这些好处。

8.5.3 泛化层次

如果需要，可以在多个层面上进行特化。图 8.28 显示了一个这样的例子，其中，银行引入了两种不同的活期账户：为单独顾客提供的个人账户 PersonalAccount，为公司提供的商业账户 BusinessAccount。结果，无差别的活期账户 CurrentAccount 不再可用，因此这个类被重定义为抽象类。图中的三个点称为省略号（*ellipsis*），表示除了显示的这些子类还可能有其他子类，在这个例子中还有较早出现过的在线账户 OnlineAccount 类。

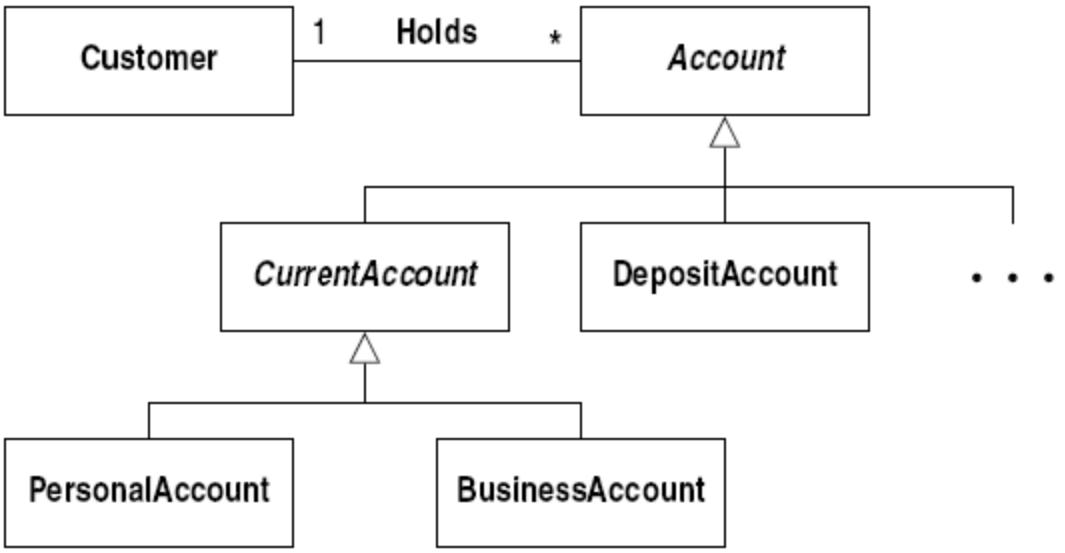


图 8.28 泛化层次

图 8.28 中，活期账户“CurrentAccount”类是“Account”类的子类，同时还是“PersonalAccount”的超类。因此，术语“子类”和“超类”是相对的术语，描述在特定的泛化关系中一个类所扮演的角色，而不是类自身的固有性质。

这样的层次可以发展到需要的那么多层。层次中一个类的祖先是向上遍历层次所发现的所有类，其子孙是那些从该类向下遍历所发现的类。这里“向上”和“向下”的意思分别是“向更一般的类”和“向更特殊的类”。尽管层次通常画的是一般类在它们的子类上面，但在表示法中并没有这样的要求。超类总是泛化关系所指向的那个类。

可替换性不是只适用于直接子类，而是适用于所有子孙类。和图 8.28 一致，持有一个账户的顾客可以持有个人账户 PersonalAccount 以及存款账户 DepositAccount，因为这两个类都是顶层的账户类的子孙类。这意味着当层次中加入新的类时，它们立即可以使用“Holds”关联。在图 8.23 中则不是这样：加入新的账户类也需要定义一个新的关联。

在新的子类中增加功能，并使之不需要修改系统的其他部分就可以使用这种能力，是面向对象程序设计方法的强大力量之一。如 UML 中的泛化所表达的，正是可替换性原则使之成为可能。

8.6 属性和操作的继承

可替换性原则的一个结果是类的实例必须具有其祖先类所规定的全部特性。如果不是这样，试图利用没有的特性之一将会失败，并且该对象也不能替换超类对象。

继承（*inheritance*）是一个类的特性自动被其所有子孙类定义的过程。在上一节，关心的特性是参与给定关联指定的链接的能力。类的其他特性包括拥有的各种属性和操作，这些特性也被子类继承，和面向对象编程语言中的方式相同。

更准确地说，在一个类的祖先中定义的所有属性和操作也是这个类自己的特征。这提供了一种手段，藉此，一些类所共享的公共特征能够在一处定义而在许多不同类中都可以使用。为了举例说明继承，图 8.29 显示了部分账户层次，其中增加了属性和操作。

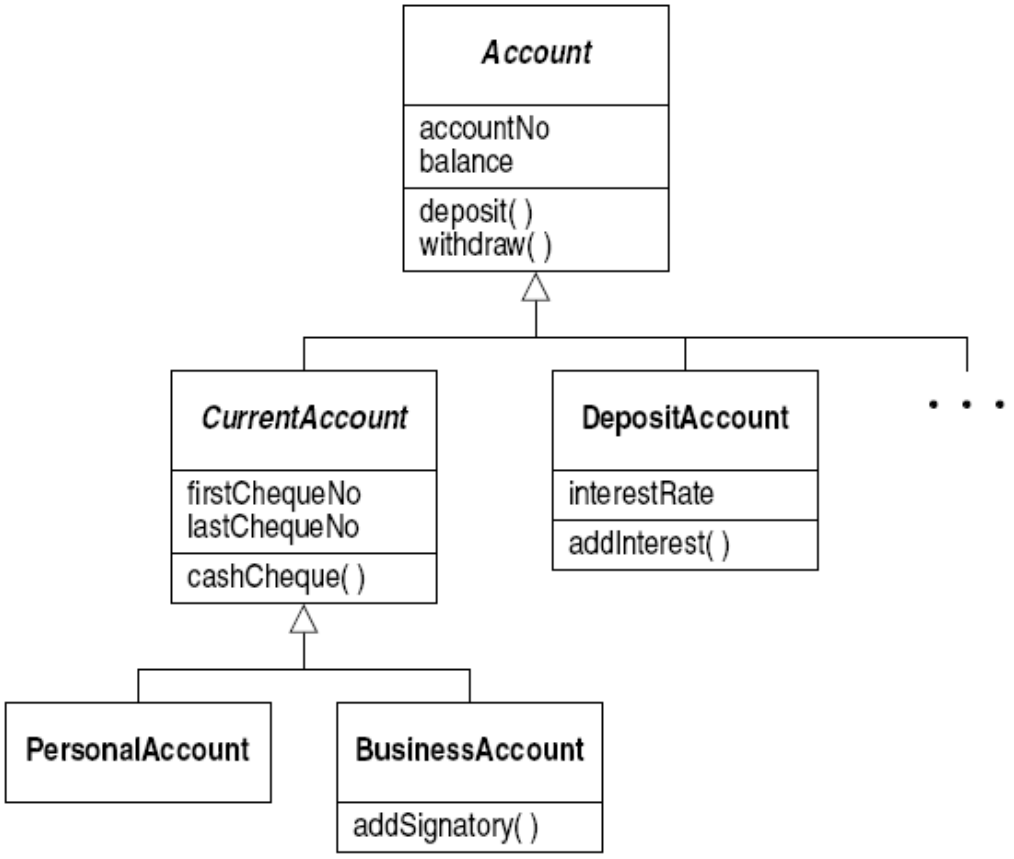


图 8.29　有继承的账户层次

这个图说明，所有账户都有账号 accountNo 和余额 balance 属性，以及存入 deposit 操作和提取 withdraw 操作。这些特征在根类“Account”中定义，但是由于继承的结果，隐含地出现在层次中其他的每个类中。

因此，继承意味着由多个类共享的特征并不一定要在每个类中全部写出，而是可以在层次中向上移到适当的超类中。一般而言，这有避免重复和使层次更清晰和易读的作用。

注意，在很多程序设计语言中，术语“继承”指的就是 UML 中称为“泛化”的类之间的关系。UML 对上面描述的特定机制保留了这个术语，这种特定机制是使用泛化的结果。

8.6.1 向子类中增加特征

子类常常不同于它们的超类，是由于它们需要定义额外的特征以支持它们自己特殊的专门行为。例如，假如活期账户提供了一个支票簿功能。为了实现这一功能，活期账户类的实例要记录为给定账户发放的支票号码的范围，另外，还需要提供一个对该账户提取支票的操作。

商业账户是对活期账户的进一步特化，区别是它们是由公司而不是由个人拥有。对商业账户，记录账户签字人可能是必要的，即允许签支票的人，并要提供一个操作为账户增加新的签字人。

支持这些需求的其他属性和操作如图 8.29 所示。为一个类定义的特征的完整列表可以通过将继承的特征和在该类自身中定义的特征结合在一起得到。例如，商业账户的属性有“accountNo”、“balance”、“firstChequeNo”和“lastChequeNo”，该类支持的操作是“deposit”、“withdraw”、“cashCheque”和“addSignatory”。

8.6.2 在子类中覆盖操作

上面的例子已经说明，在继承层次中，除了从祖先继承的属性和操作之外，如何指定需要的新属性和操作来定义一个类。然而，经常出现的是一个类需要的是所继承操作的一个修改版本，而不是直接增加一个新操作。

例如，假如只要在一个在线账户上进行存款和提款，就向账户持有人发送一个电子邮件消息以确认该事务的细节。为了实现这点，在线账户类就必须重定义存款和提款操作，以包含这个新功能。图 8.30 显示了对这两个操作的覆盖以及保存顾客的电子邮件地址的属性。

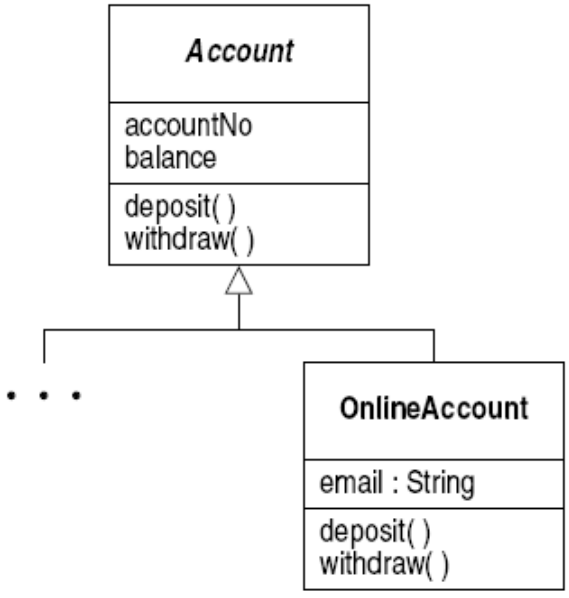


图 8.30　覆盖了提款操作的部分账户层次

如果一个类提供了对继承的操作的重定义，就称为覆盖（*override*）了继承的特征。将被覆盖操作的名字在重定义它的子类中写出来表示覆盖，如图 8.30 中的在线账户类所示。注意，如果只是为了表示一个特征是从超类继承的则不必这样做。

8.6.3 抽象操作

某些抽象类包含了操作，但在层次中的那个点不能实现它。例如，图 8.31 显示了一个经典的泛化的例子。这个层次表示了一个图形系统中可能定义的各种形状。

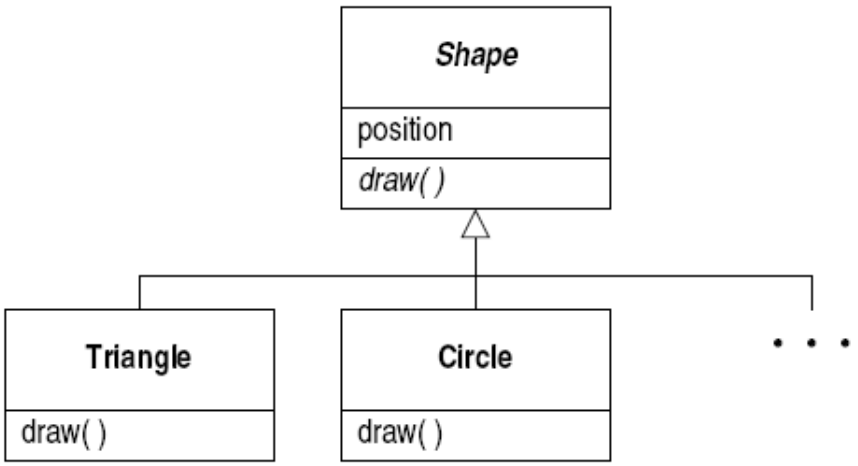


图 8.31 包括抽象类和抽象操作的形状的层次

根类“Shape”将定义所有形状公共的特性，例如位置，以及所有形状都必须提供的操作，例如绘制图形的操作。然而，“Shape”类是一个抽象类，因为不可能有一个形状，它不是三角形，不是圆形，也不是任何其他特定种类的形状。另外，不可能给出通用的绘制形状操作的实现。各个子类需要通过调用适当的图形原语，为自己定义这个操作。然而，draw 操作应该是 Shape 类的一部分，以表明“Shape”类的所有子孙都提供了这个操作。为了表示它是不能实现的，就以斜体书写，如图 8.31 所示。

这样的操作称为抽象操作，任何包含抽象操作的类必定是抽象类。在子孙类中，抽象操作必须用非抽象操作覆盖，如图 8.31 所示。任何类如果既不包含覆盖继承的抽象操作的操作，也没有继承这样的一个覆盖操作，那么这个类自身就是一个抽象类。

8.7 聚合

关联可以用于对象之间任何类型的关系的建模，关联的名字给出了建模的关系是什么的精确信息。但是，UML 挑出了一个特殊关系来专门对待，即“*part of*（部分-整体）”关系，表示一个对象是另一对象的一部分时二者之间的关系，或反过来，一个对象由一组其他对象聚集而成时的关系。

聚合（*aggregation*）是 UML 中对这种关系使用的术语。聚合只是关联的一个特例，在关联中表示“包括”另一端对象的对象一端加一个菱形符号表示。图 8.32 显示了一个聚合的典型例子，其中定义了电子邮件消息（MailMessage）有一个标题（Header）、一个正文（Body）以及若干附件（Attachment）。

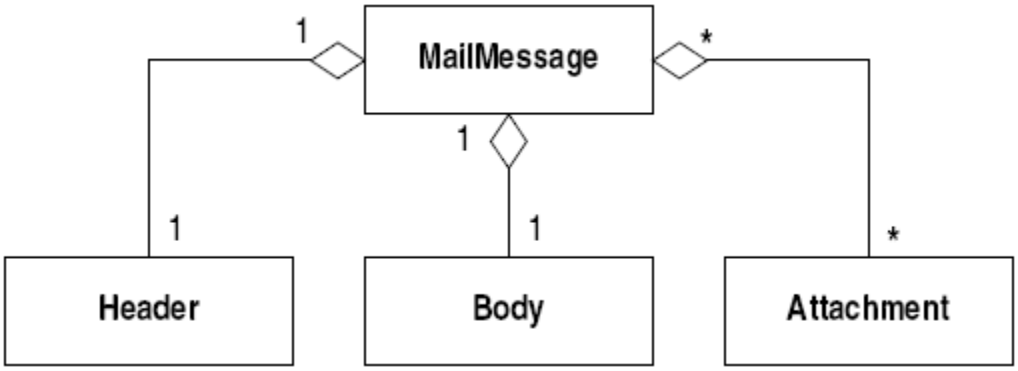


图 8.32 聚合的例子

如图 8.32 所示，重数注解可以用于聚合关系，与普通关联的使用方式相同。除了表示一个消息中标题、正文体和附件的不同数目之外，图 8.32 中指定的重数还说明：不同于标题和正文，一个附件可以同时作为多个消息的一部分。

8.7.1 聚合的含义

在类似这种情况下，在某种意义上，使用聚合达到的目的只是非正式地暗示一种对象形成另一种对象的部分而已。除了已经通过使用关联隐含的约束之外，它并没有向模型施加任何约束，因而如果对聚合的适用性存在任何怀疑时，一个好的准则是完全不考虑它。

聚合的确起作用的地方是：系统中的对象在运行时以层次方式组织。在第 2 章讨论的库存控制系统中已经给出了这样的一个例子。在那个系统中，定义了一个构件的一般概念：构件可以是单个零件，也可以是包含多个子构件的组件。图 2.12 中的类图定义了这些类之间的关系。然而，如图 8.33 所示的那样的对象结构是图 2.12 的模型完全合法的实例，其中所有的链接都是“contains”关联的实例。

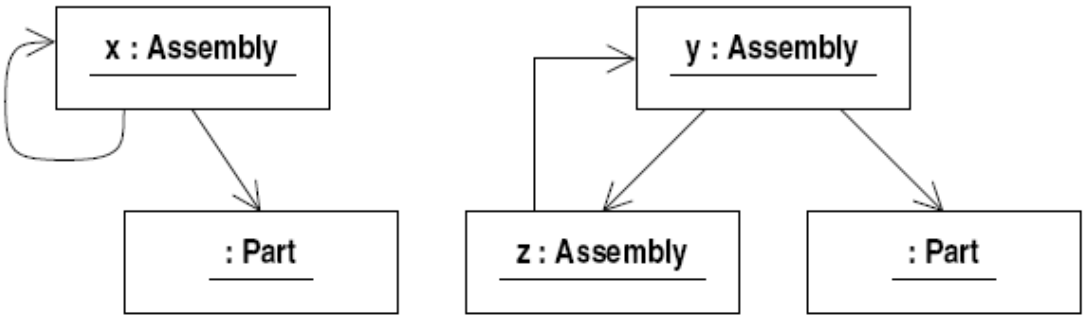


图 8.33 循环的对象结构

图 8.33 所示的对象结构的问题是：它们表示一个组件直接或非直接地是自身的一部分。在应用到实际对象，如零件和组件时，这不是一个有意义的概念，并且还意味着遍历零件层

次的操作也会循环。这些问题可以通过使用聚合消除，如图 8.34 所示，其中为图 2.12 中最初定义的关联增加了一个聚合符号。

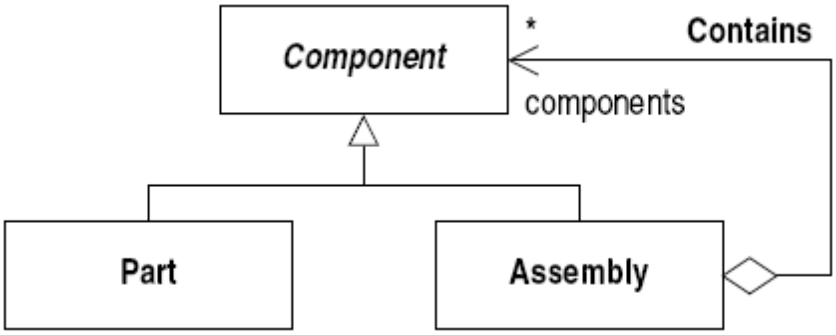


图 8.34 指定无回路的结构

聚合有两个形式性质，意味着图 8.33 中的对象结构不是图 8.34 中的类图的合法实例。第一个性质是反对称性（*antisymmetry*），意思是作为聚合的实例的链接不能将对象连接到自己。这排除了图 8.33 所示的第一种情况，名为 *x* 的组件连接到自己。

第二个性质是传递性（*transitivity*）：这是根据观察结果，如果 A 是 B 的一部分，B 是 C 的一部分，那么 A 也是 C 的一部分。更形式地说，如果对象 A 链接到 B，B 链接到 C，并且其链接是同一个聚合的实例，那么应该认为 A 也链接到 C。

聚合的这些性质按照下面的方式共同作用，排除了图 8.33 中所示的第二种可能性。因为 y 链接到 z，并且 z 链接到 y（通过一个不同的链接），所以传递性性质隐含了应该认为 y 链接到自己，然而，这被反对称性排除了，因而所描述的结构不能作为系统的合法状态。

因此，聚合在指定系统的合法状态不应该包括循环对象结构时扮演着有用的角色。然而，聚合的这些形式性质未必和整体与部分关系的直观概念有任何联系。

例如，祖先关系具有传递性和反对称性的性质：一个人的祖先的祖先仍是这个人的祖先，谁也不能是自己的祖先。这些约束可以使用聚合文档化，如图 8.35 所示，但是对此非形式的解释会使人想到人是自己祖先的一部分，就没有什么意义。

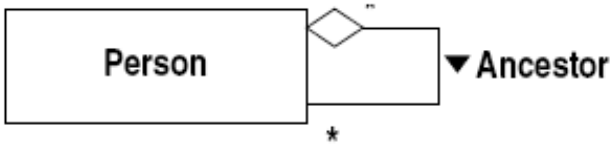


图 8.35 聚合的适用性

应该注意，上面描述的反对称性和传递性的性质，只适用于聚合使对象能够链接到自己的类的实例的情况中。当聚合是自关联时，例如祖先关系，或者泛化使得可以创建递归的对象结构时，如组件，会出现这种情况。 在不同类之间的整体—部分关系的情况下，如图 8.32 所说明的情况，聚合与一般的关联所指定的几乎没有差别。

8.8 组合

组合是一种强形式的聚合，其中“部分”对象依赖于“整体”对象。这种依赖性表现在两个方面。第一，“部分”对象一次只能属于一个组合对象，第二，当组合对象销毁时，它的所有从属部分都必须同时销毁。

在图 8.32 中给出的电子邮件消息的例子中，将消息和它的标题及正文之间的关系作为组合关系建模可能是合理的，因为很可能一旦消息已经被删除，就既不存在标题，也不存在正文体了，并且在它们存在时它们属于唯一的一个消息。如图 8.36 所示，组合的表示法类似于聚合，只是关系“整体”端的菱形是实心的。

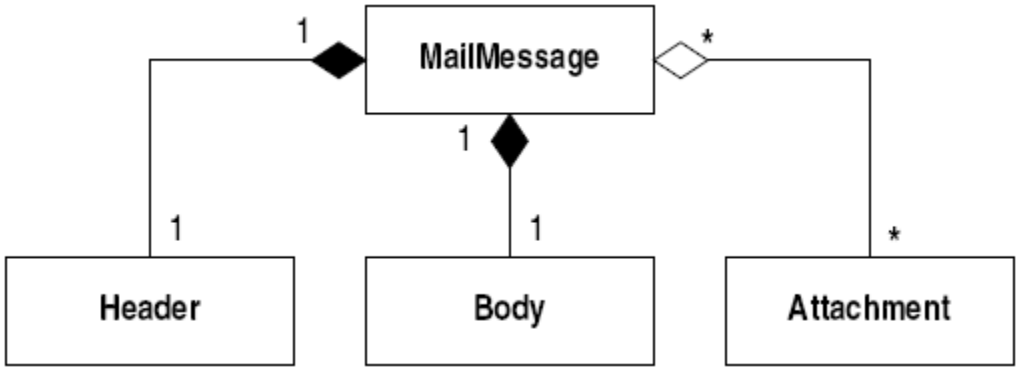


图 8.36 组合的使用

但是，消息及其附件之间的关系用组合建模可能不太恰当。首先，在同一时间，附件可以属于多个消息，其次，很可能附件可以保存，因此它们的生命期将超过所附属的消息的生命期。

然而，组合的基本概念没有强到足以强制组合对象（composite object）具有某些自然性质。例如，考虑图 8.37 中的类图，它给出了计算机某些方面的一个简单模型。该图表明计算机是一个处理器和若干端口的组合，并且端口必须连接到处理器，但是图中并没有规定一个自然约束，即连接在一起的端口和处理器必须属于相同的计算机。一个实例图，如果表示一个端口是一台计算机的一部分，但是连接到了另一台计算机的处理器，将是这个类图的合法实例。

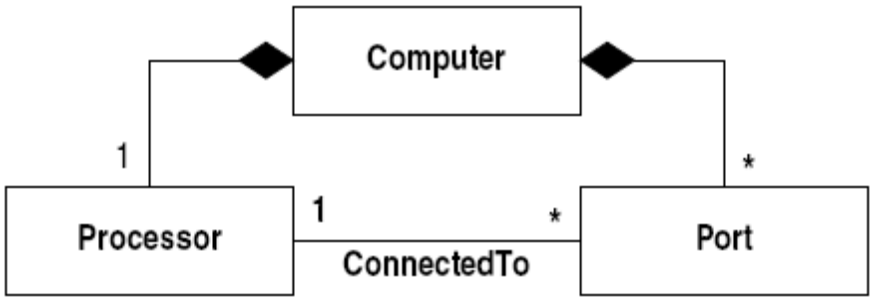


图 8.37 计算机的结构

为了更准确地对这种情形建模，我们需要一种方式表明端口和处理器之间的关联应该被包含在组合之中，以使得只有属于同一个组合物（composite）的对象才可以被连接。这可以用组合的另一种图形表示法描述，将形成组合对象的组成部分的类和关联放在计算机的类图标之内，如图 8.38 所示。注意，图 8.37 中与组合关系关联的重数现在被表示为嵌套类的类重数。

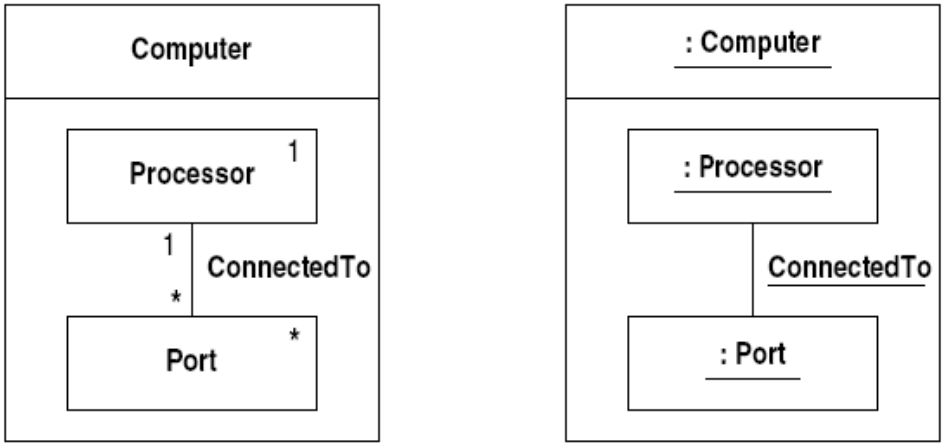


图 8.38　作为组合对象的计算机

图 8.38 还显示了一个计算机的实例，作为一个组合对象：通过物理上将“ConnectedTo”链接包含在对象中，这个图使端口和处理器必须属于同一台计算机的约束看起来很清晰。

图 8.38 中，类和关联，它们组成该组合对象的部分，显示在类图标中通常包含属性的一栏中。属性值和组合对象的这些部分共有的特性是它们只能一次属于一个对象，并且在该对象销毁时也被销毁，它们几乎可以被看作是组合的一种特例。

当然也有想要链接不同组合对象的部分的情况。例如，考虑连接成一个网络的一些计算机。物理联接可能通过连接端口对完成，而这可以用类图中端口类上的关联建模。但是，如果这个关联完全画在“计算机”组合对象框中，这将隐含着端口只有属于同一计算机时才可以连接。然而，如果我们试图对网络建模，这显然不是我们想要的，因为这个模型的重点将是显示不同计算机之间的链接。为了说明这个可能性，链接端口的关联必须以这样一种方式绘制：它经过组合对象框的外面，如图 8.39 所示。这意味着，如果需要，由关联的实例所链接的端口可以属于不同的组合对象。

UML 2.0 引入了组合结构图（*composite structure diagram*），对组合结构的元素和元素之间的复杂关系建模，详见后续章节。

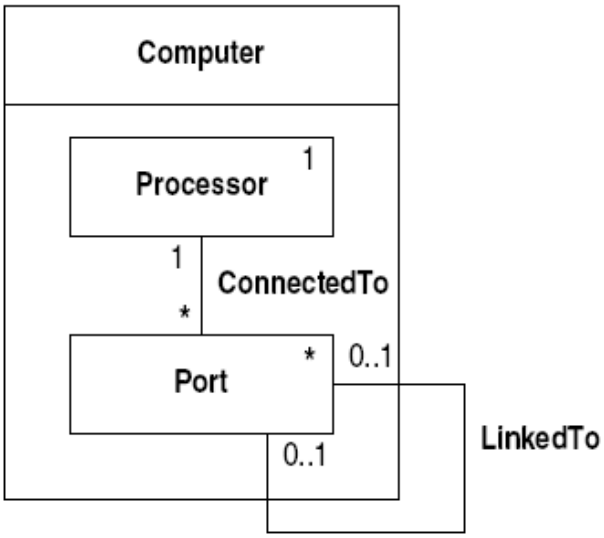


图 8.39 将计算机连接在一起

8.9 关联类

属性描述类实例的特性。例如，学生类可能有一个属性“姓名”，于是每个学生对象将包含一个指定该学生姓名的数据值。然而，有时记录某些信息是必要的，这些信息似乎是与两个对象之间的链接相关的，比单独考虑任一个对象更自然。

考虑图 8.40 所示的学生和他们选修的课程之间的关联的例子，假定系统需要记录学生在所有选修的课程中获得的所有分数。至今所介绍的类图表示法不能使我们很容易地对这种情况建模。

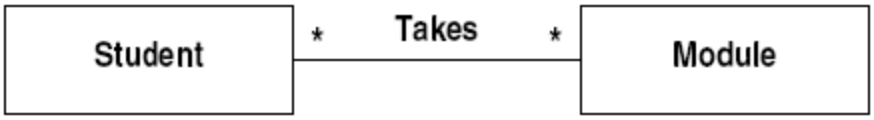


图 8.40 记录考试结果的简单模型

给学生类增加一个“分数”属性并不够，因为学生通常选修许多课程，因此需要为每个学生记录多个分数。允许保存一组分数的属性可以解决记录多个分数这个问题，但是不能保留哪个分数是哪门课程所得的信息。这个问题可能通过将每个成绩与某种课程标识符关联而解决。然而，这将是一个拙劣的想法，因为它将复制某些由已有关联建模的信息，通过在学生对象中存储课程数据，还引入了潜在的一致性问题。

因为每门课程可以被许多学生选修，如果我们考虑将学生某课程得到的分数保存在课程对象中会引起同样的问题。因此看来学生的分数不能自然地保存在这个模型中的任何一个类中。

解决这个问题的一种可能的方法是考虑将数据与两个对象之间的链接联系在一起，而不是与单个对象联系在一起。直觉上这是一个有吸引力的提议：学生只有在选修课程时才获得分数，而这正是由该链接模型化的关系。

关联类（*association class*）提供了一种将数据值和链接联系在一起的方法。关联类是 UML 中单独的一个模型元素，它同时具有关联和类二者所有的特性。特别地，一个关联类可以像关联一样连接两个类，可是同时又和类一样具有属性，以保存明确属于链接的数据。

图 8.41 显示了能够保存学生得到的一门课程的分数的关联类，这个类代替了图 8.40 中定义的“Takes”关联。关联类被表示为一个关联和一个类图标，二者由虚线连接。这两部分必须具有相同的名字，但为了消除冗余，并不需要在两处都显示。关联类的标注是像关联的风格那样选择动词，还是像类的风格一样用名词，这由设计人员自由决定，但通常取决于是将标注写在类图标中还是写在关联附近。

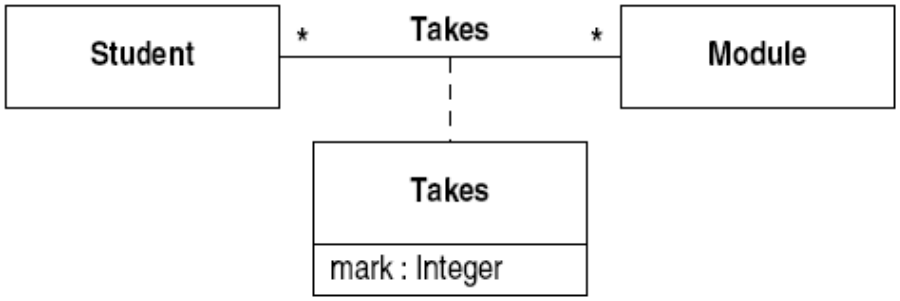


图 8.41　用关联类存储分数

关联类同时拥有关联和类二者的所有特性，所以在记录一个学生的分数时，将会既存在一个学生和课程对象之间的链接，还存在与此链接联系的一个“Takes”类的实例，其中将分数记录为一个属性值。因此，这个模型允许在每次学生选修课程时记录一个分数，如最初要求的一样。

存储学生分数的另一种方法是使用 8.4 节讨论的具体化技术。为此，将不是用一个关联类代替图 8.40 中的关联，而是用一个新的类和一对关联代替。每次完成一门课程时，应该创建一个中间类的实例，并提供一个记录学生获得分数的地方。图 8.42 中给出了“Attempt”类，以及将其连接到学生和课程类的关联。

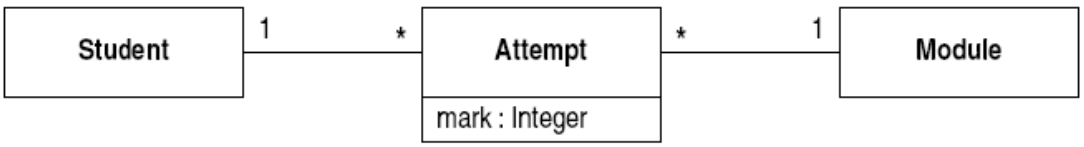


图 8.42　用具体化关联存储分数

但是，图 8.41 和 8.42 所示的模型并不完全等价。主要的区别涉及到一个特定的学生和课程之间可以存在的链接的数目。例如，假如允许学生补考不及格的课程，那么可能要求系统记录学生对一个课程的所有考试的有关数据。

图 8.42 中的模型不用修改就可以处理这种新的需求。这个模型允许任意数目的“Attempt”对象链接到相同的学生和课程，每个记录一个不同的分数。但是，这在图 8.41 所示的关联

类中是不可能的。它与等价的关联共享的性质是：在一个给定的学生和课程之间只能存在一个链接，所以必须对模型进行修改以适应新的需求。

这个例子说明关联类具有普通关联的所有性质。它们也享有类的所有性质，尤其是参与另外的关联的能力。

例如，假设系统更进一步增强以支持产生成绩单，列出选修一门课程的所有学生及其得分。对每个选修课程的学生，存在一个“Takes”的实例，记录他们的分数，所以建模成绩单的一种自然的方法是利用一个关联，将成绩单类“MarkSheet”链接到包含分数的“Takes”关联类，如图 8.43 所示。

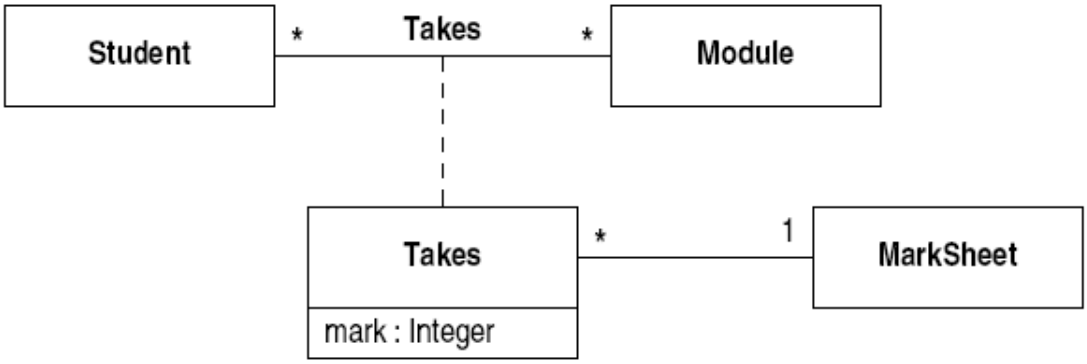


图 8.43　参与一个关联的关联类

8.10 N-元关联

迄今为止，举出的所有关联的例子都是二元的，链接正好两个类。然而，关联的概念比这更一般，原则上，可以通过一个关联链接任意数目的类。例如，对上节描述的我们需要记录学生参加课程考试的情形，又一种建模方法是使用一个三元关联，如图 8.44 所示。

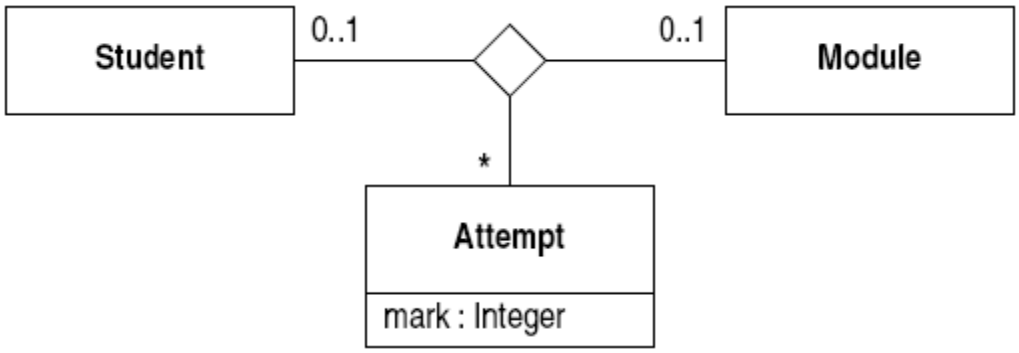


图 8.44　三元关联

术语 n-元关联（*n-ary association*）用于指一个链接任意多个类的一般关联，三元指连接恰好三个类的关联。n 元关联用一个连接参与关联的各个类的菱形描述。这些类中的每个

都有一个关联端点，可以记录许多关联端点的普通特性，包括角色名，重数注解以及导航性箭头。但是，聚合和组合不能和 n 元关联一起使用。

对一般的 n 元关联，重数注解的含意比二元关联更复杂一些。给定关联端的重数定义了该端的类可以连接到实例元组的实例数目，该元组从关联的其他各端点各取一个实例。例如，图 8.44 中的重数“多”规定了每对学生和课程对象可以被链接到零个或多个“Attempt”对象。这如同图 8.42，允许一个学生多次考一门课程。

然而，通过改变重数，我们可以施加约束，让一个学生只能考一次给定课程。在这种情况下，包括一个学生对象和一个课程对象的一对只能链接到至多一个“Attempt”对象，而这可以通过将该关联在“Attempt”端的重数改为“0..1”来指定。

“Module”类旁边的重数“0..1”可以用类似的方式解释。它表明任何给定的一对学生和考试对象将被链接到至多一个课程对象。这就形式化了一个考试对象只保存一门课程的分数的需求。这里需要可选的重数：模型不应该要求把一对无关的学生对象和考试对象链接到一个课程。

8.11 限定关联

考虑下面基于类似 Unix 文件系统的某些细节的例子。文件系统包含若干文件，每个文件通过一个唯一内部标识符为系统所知，这个内部标识符不同于用户看到的任何文件名。从用户的角度看，文件可以被命名并放在目录中。一个文件可以出现在多个目录中，并在每次出现在一个目录中时可以给文件一个不同的名字。倘若一个文件每次出现都用一个不同的名字，一个文件甚至可以在同一个目录下出现多次，从而给了用户访问相同文件的多种方式。在一个目录内，每个名字只能使用一次。然而，可以在不同的目录中使用同一个名字，并且不一定在每次都标识同一个文件。

因此，文件和目录之间的基本关系可以通过一个多对多的关联建模：一个目录可以保存多个文件，一个文件可以出现在多个目录中。这个用来识别文件的名字不是文件对象的属性：文件没有唯一的名字，而代替的是文件每次出现在一个目录中时被赋予一个名字。因而文件的名字是文件和目录之间的链接的属性，所以很自然地尝试用一个包含文件名字的关联类对这种情形建模，如图 8.45 所示。

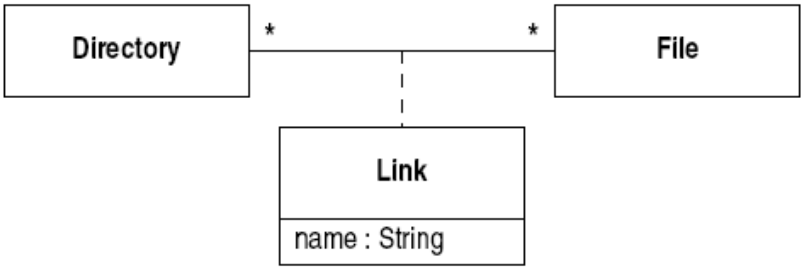


图 8.45　Unix 文件名的一个简单模型

然而，这个图在两个方面是对该情形的事实的不准确的反映。第一，它没有表达一个目录中文件名必须不同的事实。根据这个模型，很可能连接到一个特定目录的所有文件的“name”属性具有相同的值。第二，图 8.45 没有考虑到相同的文件和目录之间的多个链接的可能性。一个给定的文件可以在一个目录中以不同的名字出现多次。然而，如 8.4 节讨论的，关联的语义只允许给定一对对象之间有一个链接。

这两个问题都可以通过使用限定符（*qualifier*）解决，如图 8.46 所示。限定符是在某部分信息可以用键从一组对象中唯一确定一个对象的情况下使用的。键的相关性质是在给定语境下，每个键值只能出现一次，并且必须以某种方式确定由键描述的单个对象。在这个例子中，文件名充当了一个键。语境由目录提供，在目录中一个键值（文件名）只能出现一次。目录中的各个文件名命名一个唯一的文件。

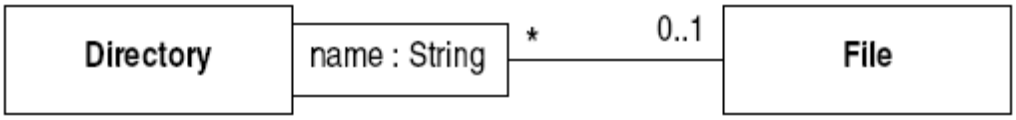


图 8.46 用限定符对文件名建模

关联类的某个属性如果具有能够充当键的特性，则在 UML 中称为限定符。限定符写在一个类旁边的小方框中，该类定义限定符值挑选对象的语境。关联线连接到限定符框，而不是连到类，如图 8.46 所示，并且在关联的那端能够使用正常的重数符号范围。

图 8.46 的部分含意和图 8.45 相同，即文件和目录由一个多到多的包容关系连接，并且一个文件在一个目录中的每次出现都有一个名字。但是，图 8.46 中的关联看起来不像是一个多对多的关系：在文件端的重数是“可选”，而不是“多”。为了使限定关联模型化的那种情形更容易理解，图 8.47 显示了与图 8.46 所示的关联一致的对象之间的一些限定链接。

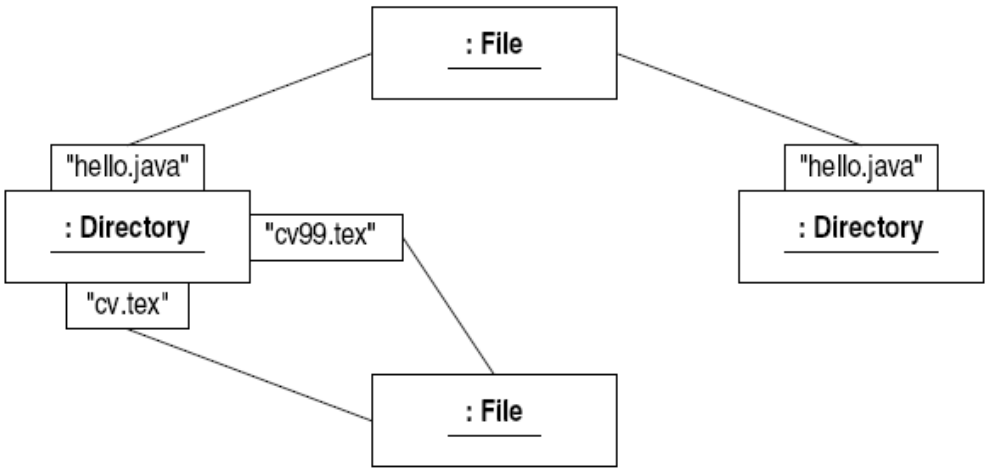


图 8.47　对象之间的限定链接

在直观上，限定关联的含意是，具有限定符的类的实例，维护从限定符的值到该关联另一端的类的实例之间的映射。在这个例子中，每个目录将维护一个从文件名到文件对象的映射。

限定符的值，即文件名，写在目录对象相邻的方框中，与限定符和类邻接的方式相同，而链接的末端是到这些方框，而不是到对象本身。图 8.46 中的任选重数指出这样一个事实，可以链接到各个限定符的值的文件至多只有一个，因为在一个目录中文件名必须是唯一的。但是，这仍然定义了目录和文件之间的一种多对多的关系，因为多个文件名可以附属于单独一个目录对象，如图 8.47 所示。

这个图还表明了单个文件能够以相同的名字出现在不同的目录下，或者以不同的名字在同一目录中出现多次。这样，在这个例子中，限定符的使用解决了对图 8.45 中的初始模型所指出的两个问题。

8.11.1 限定符和标识符

通常，限定符的使用和用于标识现实世界中的对象的属性有关。例如，图 8.48 表示了一个学生注册系统的一部分，其中学生是大学所知道的，每个学生有一个唯一的学号 id。

图 8.48　一个标识属性

然而，这个模型并没有明确每个学生的学号是唯一的。为了在图中包含这个约束，一般将这个属性重写为大学类上的一个限定符，如图 8.49 所示。

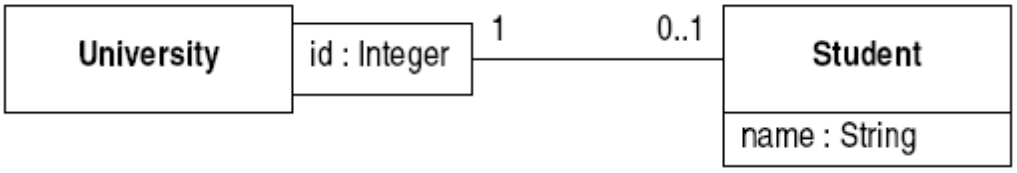


图 8.49 标识符作为限定符

由于对象有本体，因而永远没有必要只是为了区分类的不同实例而向数据模型中引入包含唯一标识符的属性。但是，如果它们在现实世界中存在，那么应该包括而且通常可以用限定符建模。

8.12 接口

在 UML 中，接口是操作的一个命名集合。接口用于表现诸如类或构件的实体的行为的特点。UML 中的接口可以看作是如 Java 中接口的概念的推广。接口的表示法使用的图形符号与类相同，但是包含一个将类元（*classifier*）标识为接口的构造型，如图 8.50 所示。

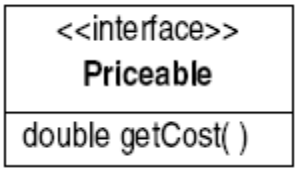


图 8.50　接口

接口之间可以存在泛化关系，其中一个接口被定义为另一接口的特化的“子接口”。接口和类之间的关系是一种实现（*realization*），其中类实现了接口。这意味着，这个类声明了或者从其他类继承了该接口中定义的所有操作。图 8.51 给出了实现的表示法，表明库存控制例子中的目录条目类实现了图 8.50 中定义的“Priceable”接口。

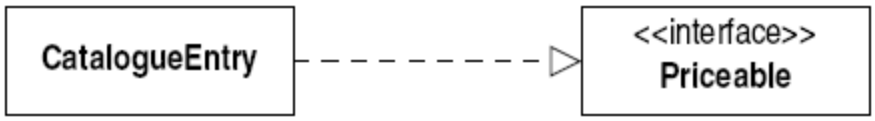


图 8.51 类实现接口

实现的另一种符号如图 8.52 所示。接口用一个小圆圈表示，标注着接口的名字，并用一条线连接实现它的类。

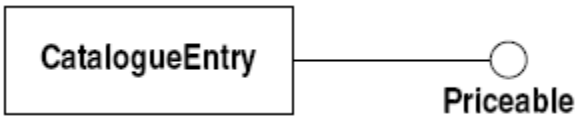


图 8.52　实现接口的另一种符号

接口表示法的一个有用的应用是明确地表示一个类的哪些特征被另一个类使用。图 8.53 表示零件类只依赖于目录条目类中定义的价格功能。类和接口符号之间的依赖意味着该类只使用接口中指定的操作。

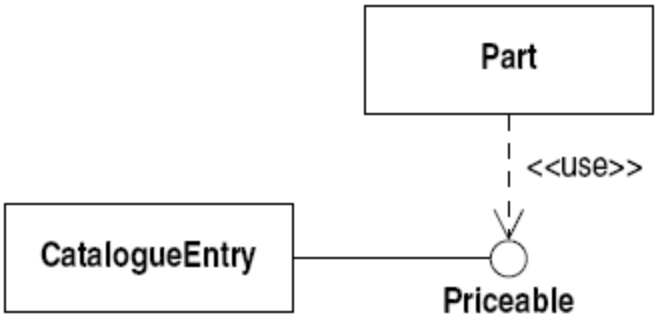


图 8.53　接口的依赖性

这个信息还可以附加在关联一端的角色名上。使用这个表示法，图 8.54 表示了和图 8.53 相同的信息。这里，接口用作一个接口说明符（*interface specifier*），为角色名指派一个“类型”。这个类型指出角色名邻近的类的哪些特征被作为所示关联的结果而使用。

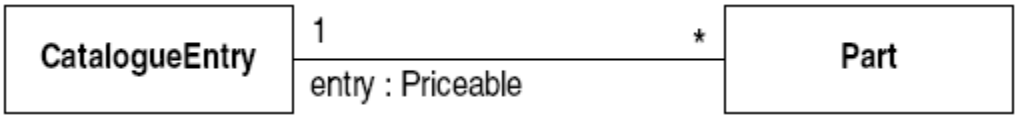


图 8.54　使用接口说明符

UML2.0 丰富了接口的概念，尤其是在构件图（*component diagram*）中使用时，详见后续章节。

8.13 模板

模板是一种参数化的模型元素。使用模板的常见例子是定义容器类。容器类是一种能够保存很多数据项的数据结构，如列表、集合或树。定义这种数据结构的代码独立于存储的数据的实际类型，但没有模板的语言难以表达必需的一般性级别。

例如，假如已经写了一个实现整数列表的类。通过物理地拷贝源代码，并将其编辑为引用字符串而不是整数，可以重用该类提供字符串列表。然而，这种方法的一个主要问题是需要维护代码的两个拷贝，对数据结构代码的任何修改将不得不在多处进行。

对这个问题，传统的面向对象解决方法是定义能够保存类层次中根类对象的数据结构。Java 为此使用“Object”类，例如 Vector 这样的数据结构定义为保存 “Object”实例的引用。因为 Java 中的每个类都是“Object”的子孙，所以这意味着借助多态性，任何类型的数据都能够保存在该数据结构中。

这种方法的局限性是它不保证正确类型的数据保存在一个数据结构中。疏忽的编程会导致在向量中保存意想不到的数据类型，并且因为向量中的对象的运行时类型信息丢失，这种错误会难以恢复。实际使用如“Vector”这样的类时，利用包装函数保证所有插入到向量中的对象都是一个类的，因此在从向量中删除时，可以安全地恢复到它们的原始类型。

用模板解决这个问题采用了一种不同的方法。例如，通过使用模板类，程序员可以定义一种保存未定义类型，例如 T 类型元素的数据结构。为了使用这样的数据结构，程序员必须指定在该数据结构的应用中用什么类型代替 T。这个过程非常容易让人回想到普通的参数绑定，所以像 T 这样的类型参数称为*模板参数*。

UML 中模板类的符号表示和普通类相似，只不过在类图标一角上的一个虚线矩形中显示了模板参数，如图 8.55 所示。由模板形成的一个类可以被表示为依赖于该模板，依赖上的“*bind*”构造型则给出了模板参数绑定的有关信息。

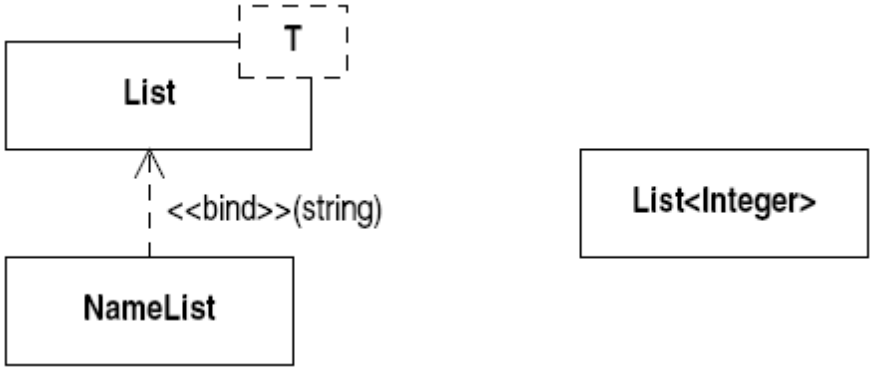


图 8.55 模板类的表示法

图 8.55 还给出了表示模板的“匿名”实例化的表示法，其中并没有给新类自己的名字，而只是将其称为“整数的列表”。这和 C++中使用的表示法相同。Java 6 中已经引入了泛型编程的概念，API 中提供了一组泛型类，程序员也可以编写自己的泛型类。

8.14 小结

- 静态模型描述系统中的数据之间结构上的关系。对象图描述特定时间存在的对象，以及它们之间的链接。
- UML 定义了许多标准数据类型，并允许有用户定义的枚举。数据值，即数据类型的实例，和对象不同，因为它们没有本体。
- 类图显示了系统的数据在所有时间都必须满足的一般特性。给定一个类图，我们可以判定一个对象图是否代表所说明的系统的可能状态。
- 这些类描述了系统中不同种类的对象。这些类，除了名字，还有描述该类的实例的对象的状态和行为的属性和操作，即该类描述的对象的状态和行为。
- 对象之间的链接由类图上的关联定义。关联表明了哪些类可以由给定种类的链接连接，以及给定的对象可以链接到多少个对象。这个信息由重数注解给出。可以标注关联，关联的每个端点也可以有角色名。
- 泛化定义了类之间的一种关系，其中一个类，即子类的实例，可以替换、或被看作是另一个类，即超类的实例。给定类可以有任意多个互斥的子类。这个过程每当需要时可以进行，形成泛化层次。
- 超类的属性和操作由它的子类继承。子类可以定义附加的属性和操作以表达它们的特化性质。如果必要，继承的操作的定义可以在子类中覆盖。
- 到一个类的关联可以被到该类的任何子类实例的链接实例化。这导致多态性，这里单个关联实际上能够指定不同类的对象之间的链接。
- 抽象类指的是没有实例的类。抽象类表示一个局部概念，它的引入有助于构成一个泛化层次。

- 聚合是关联的一种特殊形式，意图在于捕获整体－部分关系的语义。聚合可以用于禁止对象图中的回路和循环。
- 组合表示了一种更强形式的聚集，其中一个类的实例的生存期包含在另一个类的实例的生存期之中。
- 在有些情况下，类图中的信息被看作是属于一个链接比看作属于个别对象更好一些。这可以利用关联类建模。关联类同时具有关联和类的特性，如果需要也可以参与另外的关联。
- 在一项数据作为检索关联类对象的键的情况下，可以使用限定关联。
- 接口定义了可以由类支持的操作的集合。支持这些操作的类说成实现了该接口。
- 类可以参数化，从而产生模板类的定义。通过将参数绑定到模板参数，模板类提供了可复用性。

8.15 习题

（1）画出表示下面的类和对象的 UML 图标，在适当的地方，用枚举和编程语言类型指定属性类型。

(*a*) 一个表示位置的类，具有给出点的 *x* 和 *y* 坐标的属性。此外，给出这个类的两个实例，坐标（0，0）的原点，以及点（100，－40）。

(*b*) 一个表示计数器的类，具有设置或将计数器重置为零的操作，递增和递减指定数量存储的值的操作，以及返回当前值的操作。

(*c*) 一个表示开关的类，能够打开或关闭。

(*d*) 一个表示红绿灯的类，具有记录当前照亮的颜色的属性。

（2） 为下面的关联定义重数。

(*a*) “Has on loan（借出）”，链接图书馆系统中的人（people）和书（book）。

(*b*) “Has read（已读）”，链接人和书。

(*c*) “Is occupying（占据）”链接棋子和棋盘上的方格。

(*d*) “Spouse（配偶）”，链接“Person”类和“Person”类。

(*e*) “Parent（父母）”，链接“Person”类“Person”类。

（3）在图 Ex8.3 中，说明哪个对象图是给定类图的合法实例。假定对象图中的所有链接是相应类图中的关联的实例。如果对象图不是合法实例，解释为什么。

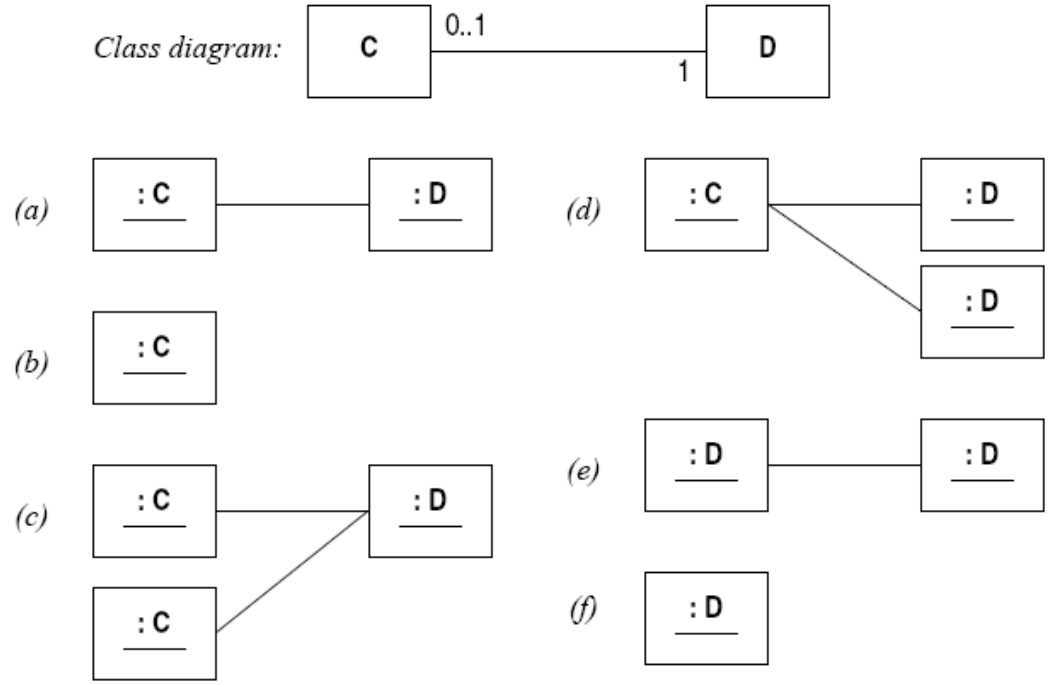


图 Ex8.3 一个“可选”关联以及一些候选对象图

（4）对图 Ex8.4 中给出的图重做上面的问题。

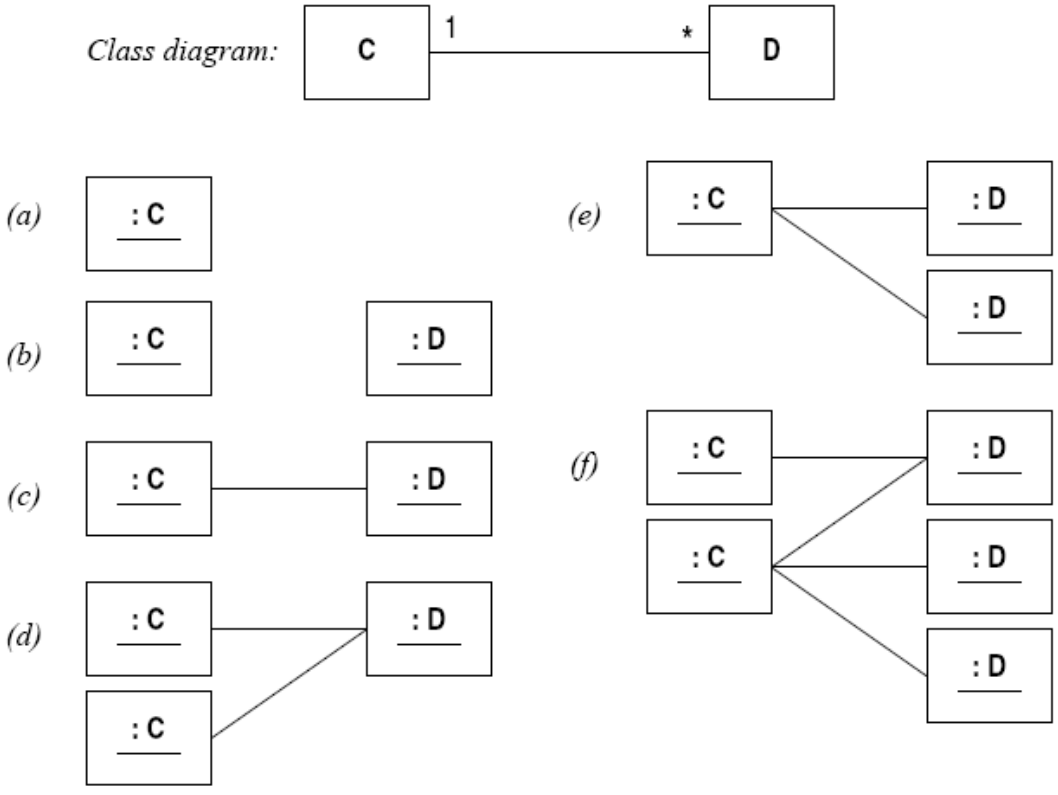


图 Ex8.4 “多”关联和一些候选对象图

（5）公司可以雇用多人，人也可以为多个公司工作。每个公司有一个总经理，公司中的每个雇员有一个经理，经理可以管理多个下属的雇员。为图 Ex8.5 中的类图加上适当的标注，使上述含意清楚。

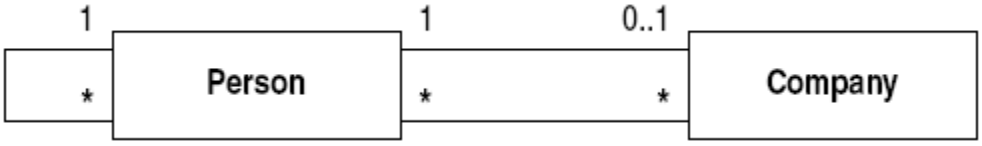


图 Ex8.5 与雇用相关的关联

（6）假定你正在写一个维护家谱的程序。

(*a*) 图 8.20 中的类图是一个保存关于人的祖先信息的适当的模型吗？如果不是，解释为什么不是，并对这个图进行适当的修改。

(*b*) 扩充这个图，以便它能够保存关于婚姻的信息。模型不仅应该能够记录结婚，而且能够记录离婚和再婚，可能是对同一个人。

（7）改进图 8.22，使该模型可以记录 Jill 的职别和薪水。除了更新该类图之外，还要改进对象图，表示所显示的两个合同的信息。

（8）在图 Ex8.8 中，说明哪个对象图是所给类图的有效实例。如果某个对象图不是有效实例，请解释为什么。

（9）在图 8.31 中，如果省略了 shape 类中的抽象的 draw 操作，会有什么后果？这是可替换性失灵了吗？或是这里涉及了某些其他原理？

（10）图 8.20 中的“parent”关联是否具有反对称性和传递性的性质？它是否能够正确地或有效地表示为一个聚合？

（11）使用组合表示法的“嵌套类”形式重画图 8.36 的类图。画一个组合对象表示一个具有标题和正文，以及两个附件的消息。

（12）画一个对象图，是图 8.37 的实例，表示一个端口是一台计算机的一部分，但是连接到另一计算机的处理器。

（13）图 8.44 中的图是否指定了“Attempt”类的每个实例恰好链接到一个学生和一门课程？如果没有，能不能修改关联上的重数来这样规定？

（14）扩充描述雇员为公司工作的图 8.15，使得它能够保存雇员的薪水

（*a*）作为属性保存

（*b*）使用关联类保存。

可以提出什么论据分别支持这两种模型？

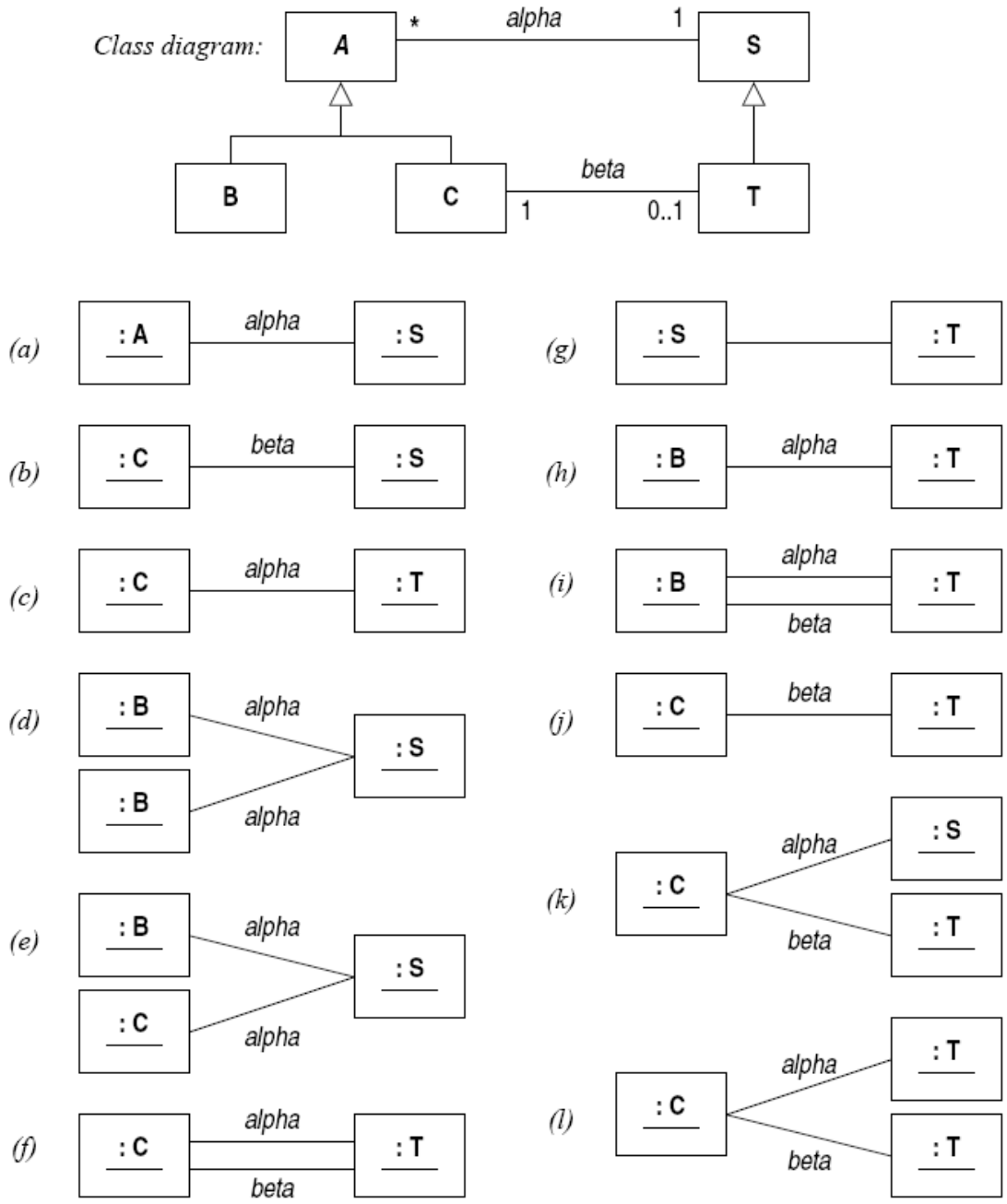


图 Ex8.8 泛化和候选的对象图

(15)画一个类图概括下列关于图书馆的事实。讨论你的设计决策,以及你的模型的局限性。

对图书馆(library)保存的每本书(book),目录(catalogue)包括书名(title)、作者名(author's name)和该书的 ISBN 号。图书馆中的一种书可能有许多册(copy),每册有一个唯一的登记号码(accession number)。图书馆有很多注册读者(registered reader),发给每个读者若干借书卡(ticket)。系统记录每个读者的姓名(name)和地址(address)和已经发给他们的借书卡的数目。读者可以用他们的每张借书卡借一本书,系统记录读者借了哪本书和必须还书的日期。

（16）图 Ex8.16 是一个文件系统某些方面的模型，其中目录包含子目录和文件，文件系统由根目录下的一组文件组成，用户可以拥有目录和文件，可以读文件并有一个主目录。

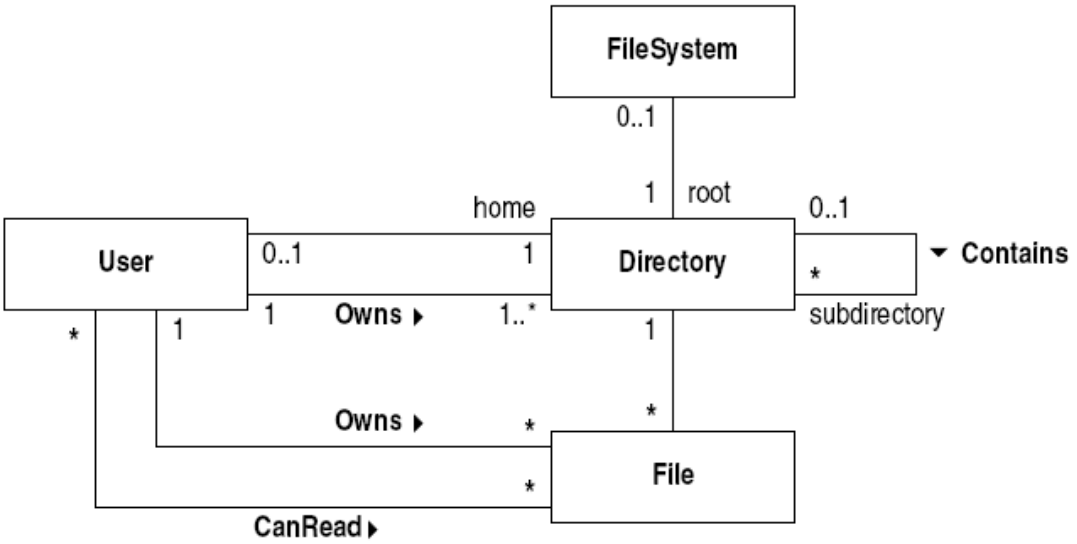


图 Ex8.16　文件系统的类图

(*a*) 画一个与该类图一致的对象图，显示这个文件系统，一个对应于你的账户的用户对象，你的主目录，一个称为 mail 的子目录，你的主目录下一个称为.login 的文件，和 mail 目录下一个称为 message1 的文件。

(*b*) 如果引入一个新的“Node”类，该文件系统的说明会更像 UNIX。“Node”是“File”和“Directory”类的超类。重画类图，用这个新的类减少图中关联的数目。

(*c*) 引入“Node”类对你在*(a)*部分所画的对象图是否有什么影响？

(*d*) 考虑在这些图中是否可以正确地使用组合或聚合。

（17）这是从图形工作站的绘图工具文档中摘录的内容。

包中的对象分为简单对象（*primitive objects*）和复合对象（*compound objects*）。简单对象是：弧（*arc*），椭圆（*ellipse*），折线（*polyline*），多边形（*polygon*），方框（*box*）和文本（*text*）。

简单对象可以被移动、旋转、垂直或水平翻转，复制或擦除。文本简单对象不能翻转。

复合对象（*compound object*）由简单对象组成。组成复合对象的简单对象不能独立地被修改，但是可以被作为一个实体操作；复合对象可以被移动、旋转、水平或垂直翻转，复制或擦除。包含任何方框的复合对象只能以 90 度旋转。

根据这些描述，画一个类图，使用泛化表示绘图包中各种不同类型的图形对象之间的关系。讨论你的模型的局限性，以及你所作的重要的设计决策。

（18）用一个类图描述下面的情况，并讨论在你的图中聚合可能的使用。

客户（customer）向供应商（supplier）发出一份订单（order），订单定购各种不同零件（part）。一个订单由若干订单行（order line）组成；每行指定供应商目录（catalogue）

中的一种特定零件，并说明要定购多少。作为对订单的响应，供应商安排一次发货（deliver），由所有定购的零件组成。（对这个习题，可以忽略不同零件种类之间的区别）。

（19） 画一个类图表示下面的信息，尤其注意限定关联的使用。解释你所作的假设或设计决策。

UK 银行系统由许多银行（bank）组成。每个银行包括若干支行（branch），每个支行由一个唯一的类别码（sort code）标识。银行保存账户（account），每个账户有一个唯一的账号（account number）；另外，每个账户由该银行的一个特定支行持有。某些账户允许开支票（cheque），每张支票由唯一的支票号（cheque number）确定。

（20）根据下面对编程语言部分语法的描述，构造一个类图说明用该语言编写的程序的结构。

一个模块（*module*）由一组命名的特征（*features*）组成。特征可以是一个变量（*variable*）、一个例程（*routine*）或一个嵌套的模块。例程由声明部分（*declaration part*）和语句部分（*statement part*）组成。例程局部的特征可以在声明部分声明，语句部分则包括一个非空的语句序列。语句可以是循环（*loop*）、条件（*conditional*）或赋值（*assignment*），每个赋值包含一个要赋值的变量的引用。

（21）画一个类图对下面描述的系统建模：

一家公司决定计算机化文档在它的各个办公室间的传递，并通过安装一个电子办公桌（*electronic desks*）网络来实现。每个办公桌提供下列服务:

记事簿（*blotting pad*），能够保存用户当前处理的文档。记事簿提供了基本的字处理设施。

文件柜（*filing cabinet*），模拟现实的文件柜。文件柜分成多个抽屉，每个抽屉分为多个文件夹。文档可以存储在抽屉中，或者存储在抽屉的文件夹中。

邮件服务（*mail service*），允许用户和网络上的其他用户通信。每个办公桌配有三个托盘（*tray*），对应于传统办公室中的 IN（收），OUT（发）和 PENDING（未决）文件盘。网络会自动将新邮件放入用户的 IN 托盘，并定时从 OUT 托盘取走文档并将它们邮寄给接收者。

文档可以在邮件托盘和记事簿之间移动，也可以在记事簿和文件柜之间移动。没有直接在托盘和文件柜之间移动文档的设施。在任何给定时间记事簿上只能有一个文档。

（22）下面是关于作用域和标识符的一些事实：

很多程序设计语言都定义了作用域（scope）的概念。每个标识符（identifier）都在某一作用域中声明，在同一作用域中声明两个或多个同名的标识符是错误的。然而，作用域可以嵌套，在内层作用域中可以用外层作用域已经使用的名字定义标识符。

画两个类图对这些事实建模，一个使用限定符，一个不用。比较得到的两个图，并判断哪一个是对所描述情形的更准确的表示。

（23）构造一个类图概括下面描述的窗口管理器（window manager）的情况：

当运行系统时，工作在桌面（desktop）上发生，桌面即窗口管理器所占据的屏幕空间。任何时候都可以运行多个应用程序（application），每个应用程序在屏幕上显示为一个图标（icon）或者显示为一个应用窗口（application window）。当窗口管理器开始工作时，控制应用程序（control application）起动；终止控制应用程序将停止当前会话（session）。当你使用应用程序时，你的桌面上出现两种窗口：应用窗口，以及应用窗口中包含的窗口，通常称为文档窗口（document window）。所有的窗口都包含一个标题栏（title bar）、最大化按钮（maximize button）和最小化按钮（minimize button）、控制菜单框（control-menu box），以及可选的水平和垂直滚动条（scroll bar）。另外，应用窗口包含一个菜单条（menu bar）；从菜单条中选择的操作可以影响应用窗口或其中包含的任何文档窗口。

（24）网络用户被授权使用某几个工作站。对每台这样的机器，用户有一个账户和密码。画一个描述这种情形的类模型，并讨论你所作的假设。

（25）设想一个有很多书籍的图书馆。每本书都包含一个参考书目（bibliography），每个参考书目由许多对其他书籍的引用（reference）组成。典型地，一本书可以被多处引用，因此一个引用可以出现在多个参考书目中。为这种情形画一个类图，并讨论在图中聚合可能的使用。

（26）为下面描述的电子布告栏（Electronic Noticeboard，EN）系统构造一个类图：

EN 系统的设计是用来帮助一个计算机系统的一组用户之间进行通信的。它能够公布布告使所有用户可以阅读，并允许在用户之间进行讨论。

当用户登录到 EN，呈现给他们的是一个用户工作区，由布告栏（*noticeboard*）和讨论组（*discussion groups*）两个区域组成。用户必须选择他们想要访问的区域，在以后任何时候都可能在两个区域之间自由移动。

布告栏包括一个布告列表，用户可以选择阅读任何已有布告，或者向布告栏增加新的布告。布告必须有终止日期（expiry date），在此日期之后它将被存档，并且不再出现在标准布告栏中。默认情况下，所有布告向所有用户一直显示到它们的终止日期；但是，用户可以选择从他们的私人布告栏视图中删除指定的布告，虽然不推荐这样做。

系统还维护有讨论组，各个讨论组论及一个特定的主题（topic）。每个讨论包括若干跟帖（contribute）。如果有的话，用户可以选择他们想要阅读的讨论。默认情况下，一个特定讨论中只有未阅读的跟帖被呈现给用户。在一个用户发帖时讨论开始。其他用户可以通过跟帖响应，依次能够产生更多跟帖。如果认为跟帖已经远离了讨论最初的主题，它就可以被确定为一个新讨论组的发帖；那么通过新组和旧组它都是可访问的。

所有的布告和跟帖和它们的日期一起存档，对跟帖来说，如果有的话，还要将跟帖的记录存档。

开始新讨论的用户可以指定只有 EN 系统注册用户的一个子集能够访问该组。可能允许用户具有只读、读写或无权访问一个组。对比之下，所有的布告对所有用户都是可读的。如果用户具有最初发帖的组的阅读权，那么用户可以阅读已存档的帖子。

（27）根据下面对 Emacs 文本编辑器中的 info 系统的描述中包含的信息，构造一个类图。在需要的地方，给出你的设计决策的理由。

Emacs 中的 info 系统提供了一个简单的超文本阅读工具，能够在编辑器中浏览在线文档。info 系统包括许多文件（file），每个文件大致对应一个单独的文档。每个文件被分为若干节点（node），每个节点包括少量的文本，描述一个特定主题。系统中的一个节点被确定为目录（directory）节点：它包含 info 系统中可利用的文件的有关信息，并在系统起动时呈现给用户。

节点通过链接（link）连在一起：每个链接联系两个节点，并且提供了使用户能够通过链接从一个节点移动到另一节点的操作，以这样一种方式，可能浏览全部文档。有三种重要的链接，称为向上（*up*），下一个（*next*）和上一个（*previous*）链接：这些名字暗示了文档中的节点将以层次方式组织，但并不是系统强制的方式。除了链接，节点还可以包含一个菜单（menu）。菜单由许多条目组成，每个条目连接到系统中另一个节点。

系统运行时，在历史清单中保存所有已访问过的节点的记录，使用户可以重回他们到达这个文档的路径。

第 9 章 状态图

在一个交互中，可能发送给单个对象一个或多个消息，并且，这些消息以特定的顺序被接收。但是，在另一个交互中，同一个对象可能接收完全不同的消息。根据每个交互的详细情况，特定消息发送到对象的顺序也可能随着情况的不同而改变。考虑对象能够参与的所有可能的交互，我们可以看到，在一个对象的整个生存期中，它必须能够合理地响应次序变动范围相当大的消息。

在第 8 章，我们已经看到，对象图并不是用来详细说明系统所有可能的状态的。首先，的确存在着太多的状态，不能用文档穷举；其次，除了要知道可能的状态是什么，我们还需要知道哪些状态是不可能的，或者不合法的。出于同样的原因，顺序图和通信图也不是用来描述对象能够参与的所有可能交互的。

对这两种情况，解决方案是相同的：使用形式更抽象的表示法详细说明系统，而不是举例说明。在 UML 中，对象的行为规格说明是通过为对象定义状态机（*state machine*）来给出的。状态机说明了对象对它在生存期期间可能检测到的事件的响应。在 UML 中，状态机通常是用一种称为状态图（*statechart diagram*）或状态机图的图来文档化的。

交互图和状态机给出的是系统动态行为的两个互补的视图。交互图显示了在较短的一段时间内在系统中的对象之间传递的消息，通常是在单个用户产生的事务期间，因此这些图必需描述很多对象，即特定事务中所涉及的那些对象。而状态图自始至终在单独一个对象的整个生存期中跟踪该对象，指定该对象能够接收的所有可能的消息序列以及它对这些消息的响应。

9.1 依赖状态的行为

许多对象展现出了依赖状态的行为。不严密地说，这意味着对象在不同时间会对相同的刺激做出不同的响应。例如，考虑一个简单 CD 播放机的行为，播放机中有一个装 CD 的抽屉盒，如果当前有播放的 CD，就装在抽屉里。还包括一个界面，界面包含三个按钮，标明“装入（load）”、“播放（play）”和“停止（stop）”。如果当前抽屉关着，装入按钮使之打开，如果是打开的，则使之关闭。停止按钮使播放机停止正在进行的播放。如果没有正在播放的 CD 时，按下停止按钮不起作用。最后，播放按钮播放抽屉中的 CD，如果按下播放按钮时抽屉是打开的，则先关闭抽屉后再开始播放。

这个 CD 播放机至少在两个方面表现出了依赖状态的行为。例如，如果抽屉开着，按下装入按钮将关闭抽屉，而抽屉关着的时候，按下装入按钮将打开抽屉。另外，如果正在播放 CD，按下停止按钮就停止播放，但是如果没有播放 CD，按这个按钮没有作用。

在这个例子中，我们至少可以标识出 CD 播放机的三个不同状态。按下“装入”按钮引起的不同结果表明我们需要区别“打开（open）”和“关闭（closed）”状态，而按下“停

止”按钮的不同结果表明存在第三个状态，可能标记为“正在播放（playing）”，它不同于上面两个状态。同样值得注意的是，CD 播放机可以响应事件而改变状态。例如，重复地按下装入按钮将引起 CD 播放机在打开和关闭状态之间转移。

这个例子中的三个状态符合 CD 播放机的实际状态中可观察到的差异，但是情况并不总是如此。区分状态的基本原则：处于一个特定状态的对象时，对至少一个事件的响应和对象处于其他状态时对该事件的响应不同。因而识别的状态可能与容易发现的对象外部特征对应，也可能并不与之对应。

用于行为建模的状态的概念应该区别于第 2 章所讨论的状态，在第 2 章，对象的状态被定义为在给定时间其属性的值的整体。状态的行为概念比这个更广泛：在两个时间一个对象的属性很可能不同，可是却处在相同的行为状态。对此，CD 播放机的“关闭”状态就是一个例子：抽屉中有 CD 或者没有 CD 可以被认为是 CD 播放机不同的属性值，但是在两种情况下我们都可以认为播放机处于关闭状态。

行为状态的识别并不是一个严格的过程。状态的不同，是通过处于不同状态的对象对事件的响应不同来区分的，但是将什么看作是不同的响应，在某种程度上却是一个需要判断的问题。行为状态的重要特性是，第一，一个对象有若干个可能的状态，并且在任何给定时间恰好处于这些状态中的一个。第二，对象可以改变状态，通常，它在给定时间所处的状态会由它的历史决定。最后，在不同时间，一个对象可能依赖其状态对同一刺激做出不同的响应。

9.2 状态、事件和转移

状态图（*statechart diagram*，通常简称为 *statechart*），显示一个对象可能的状态，它能够检测到的事件，以及它对这些事件的响应。因此，为了构造对象的状态图，我们必须首先初步确定对象能够处于什么状态以及它能够检测什么事件。比如 CD 播放机的例子，我们已经标识了打开、关闭和正在播放状态，这将作为开发状态图的基础。

软件术语一般假定对象检测到的事件就是发送给它的消息。然而，在刚开始设计时不必要这么具体：只需要对象能够检测的外部事件这个更一般的概念。在 CD 播放机的例子中，能够检测到的外部事件只是按下三个按钮。因此，CD 播放机的状态机将包括至少三个事件：“装入（load）”、“播放（play）”和“停止（stop）”。

一般而言，检测到一个事件可能导致对象从一个状态移动到另一状态，这样的移动称为转移（*transition*）。例如，如果 CD 播放机处于打开状态，按下装入按钮将引起抽屉关闭，而 CD 播放机转移到关闭状态。状态图上显示的基本信息是实体的可能状态以及状态之间的转移，或换句话说，检测各种事件的路径引起系统从一个状态转移到另一状态。

描述 CD 播放机的基本模型的状态图如图 9.1 所示。系统的状态以圆角矩形表示，其中写着状态的名字。状态转移用连接两个状态的箭头表示，箭头上标注事件的名字。这种箭头的含意是如果系统在处于箭头尾的时候接收到该事件，它将转到箭头的头所指向的状态。因此，事件通常将在状态图中出现多次，该对象可能在多个不同的状态检测到同样的事件。

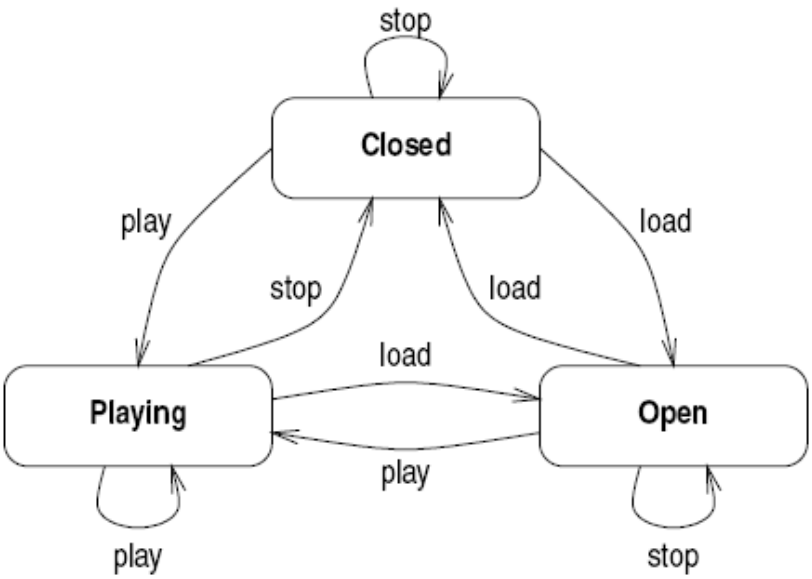


图 9.1　CD 播放机的一个简单状态机

在图 9.1 中，从每个状态都有三个出发的箭头，每一个事件都可以被 CD 播放机检测到。这种完备性不是状态机的基本性质，而只是反映了 CD 播放机的用户在任何时候都可以按下三个按钮中的任何一个。如果事件没有引起状态的改变，那么相应的转移只是在一个状态上形成回路。例如，在 CD 播放机已经处于正在播放状态，检测到播放事件时发生的转移。这样的转移被称为*自转移*（*self-transition*）。

事件如同消息一样，能够带有数据，写在消息名字后面的括号中。播放机的例子中没有需要携带附加数据的事件，但在 9.10 节考虑的例子中将看到这样的例子。

9.2.1 状态机的执行

一个简单状态机（如图 9.1）可以被认为是按照下面的方式“执行”。在任何给定时刻，对象恰好处于图中所示的状态之一。这个状态被称为*激活状态*（*active state*）。任何从激活状态出发的转移都是一个候选*激发*。例如，如果 CD 播放机的激活状态是“Open”状态，那么候选激发的转移有该状态上的自转移、标记着“load”的到关闭状态的转移，以及标记着“play”的到达“Playing”状态的转移。

能够引起转移激发的事件称为*触发器*（*trigger*）。当检测到一个事件时，将激发从激活状态出发的标注着该事件名字的转移。被激发的转移另一端的状态就成为了激活状态，这个过程将重复进行，只不过现在的候选激发变成了从新激活状态出发的转移。

从当前状态出发的事件如果没有标注为所检测到事件名字的，就忽略该事件，不激发任何转移，当前状态仍是激活状态。如果有必要指定在某个状态激活的情况下检测到一个特定事件是错误，那么可以定义一个错误状态，并增加一个到错误状态的转移，用被禁止的事件的名字加以标注。

9.3 初始状态和终止状态

图 9.1 中的图描述了 CD 播放机在使用时的运作机能，但是没有说明机器在开关时发生什么。我们假定关机器时它不表现出任何行为，当开机时它总是直接到关闭状态。

可以通过向状态图中加入*初始状态*（*initial state*）表示后一种行为；初始状态用黑色的圆点表示。从初始状态出发的转移表示创建或初始化对象时进入的状态。CD 播放机的初始状态如图 9.2 所示，初始状态上的转移表示播放机在开机后总是处于 Closed 状态。注意：从初始状态出发的转移上不应该写任何事件。

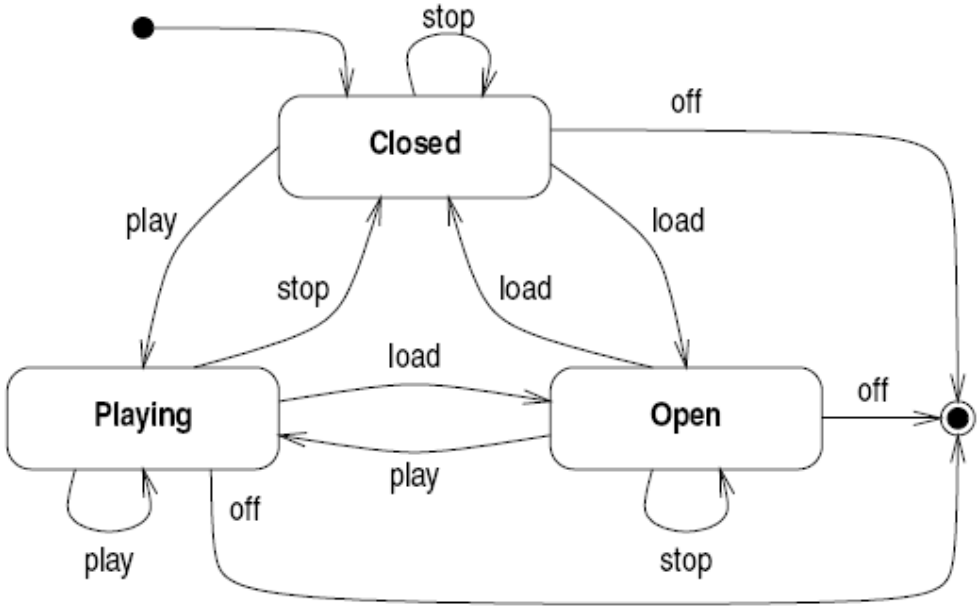


图 9.2 初始状态和终止状态

状态图还可以表示*终止状态*（*final state*）。终止状态代表对象在响应撤销、关闭或其他终止事件时到达的状态，终止状态用大圆圈中加一个小圆点表示。一般而言，可以从许多不同的状态到达终止状态。在 CD 播放机的例子中，引起到达终止状态的事件是关播放机。我们可以用一个新事件“关机（off）”对此建模。可以在任何时刻关掉播放机，因此，用 “off” 事件标注的转移将终止状态连到所有其他状态。终止状态的含意依赖于状态图所描述的对象的特性。如果是一个软件对象（类的实例）到达终止状态，那么它将完全被销毁：如果它有析构函数，那么将调用析构函数，并回收对象占用的内存。

然而，图 9.2 显然不应该被解释说 CD 播放机在每次关机时都被实际地销毁：这样的设计不可能制造出来投入市场。实际模拟的是控制 CD 播放机的软件的行为：当关掉播放机时终止控制程序，机器不再响应任何事件，直到再次开机。

9.4 监护条件

图 9.2 的状态图对 CD 播放机的行为给出了一个过分简单的描述。有一个问题是，当按下播放按钮时，播放机并不总是进入正在播放状态，而是在检测到该事件时如果抽屉中有 CD 才进入正在播放状态，否则，如果抽屉还没有关闭就只是关闭抽屉并且进入关闭状态。这意味着准确的模型应该包含从 Closed 到 Open 状态的两个标注为“play”的转移。在给定时刻，实际上沿哪个转移前进取决于该时间抽屉中的内容。

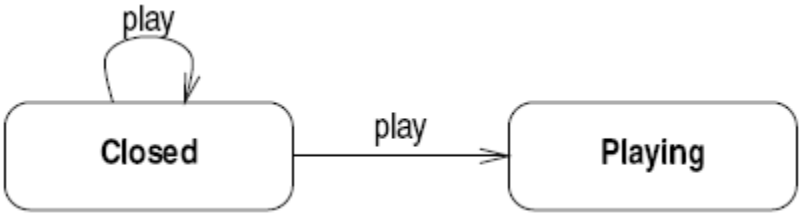


图 9.3 按下“播放”时的两种可能后果

图 9.3 显示了在 CD 播放机抽屉关着的时候按下“play”按钮的两种可能的后果。这是一个非确定状态图的例子。这个图表明播放事件可以触发两个可能的转移，但并没有说明什么时候激发其中的一个转移而不是另一个。

原则上，不确定的图没有任何错误，但是如果建模的系统事实上是确定的，那么不确定的图必定是遗漏了系统的某些信息。在 CD 播放机的例子中，当按钮按下时不存在真正的非确定性，因为播放机接下来的行为由抽屉中的内容确定。更准确的模型应该表明是什么导致沿着一个转移而不是另一个转移前进，以消除图 9.3 中的非确定性。

在状态图上，这样的信息可以通过为播放转移增加监护条件（*guard condition*）来表示，表明在什么情况下将激发该转移。监护条件是转移的规格说明的一部分，写在标注该转移的事件名字之后，并用方括号括起来。监护条件通常以非正式的英语写出，如这里一样，但是如果要求，可以用更形式化的符号，如用 OCL 语言写出。

图 9.4 所示的 CD 播放机的扩充状态图包括了监护条件，区分了抽屉中有没有 CD 的情况。为了简单起见，在这个图中省略了与当前讨论无关的初始状态和终止状态。

监护条件对状态机执行的影响如下。当检测到一个事件时，将对该事件名字标注的转移上的监护条件求值。如果转移有监护条件，那么只有在求值为真时这个转移才会激发。如果所有监护条件都是假值，并且没有无监护的转移，就忽略该事件。

如果多个转移都有值为真的监护条件，那么它们中只能有一个被激发。在这种情况下，非确定性再次引入到状态机，通常，要对一组流出转移上的监护条件进行选择，以使得在任何给定时间，为真的不能超过一个。

例如，假定 CD 播放机正处于打开状态时按下了播放按钮。发生的第一件事情是关闭抽屉；这是必要的，以便机器能够检测是否有碟片。重要的是要注意在这个时候，尽管事实上抽屉是关闭的，但 CD 播放机并不是在关闭状态。它仍然是在打开状态，评估播放转移上的

监护条件的值，看应该激发哪个转移。这阐明了前面提出的一点，CD 播放机的状态机中的状态不必要和 CD 播放机的实际状态恰好对应。

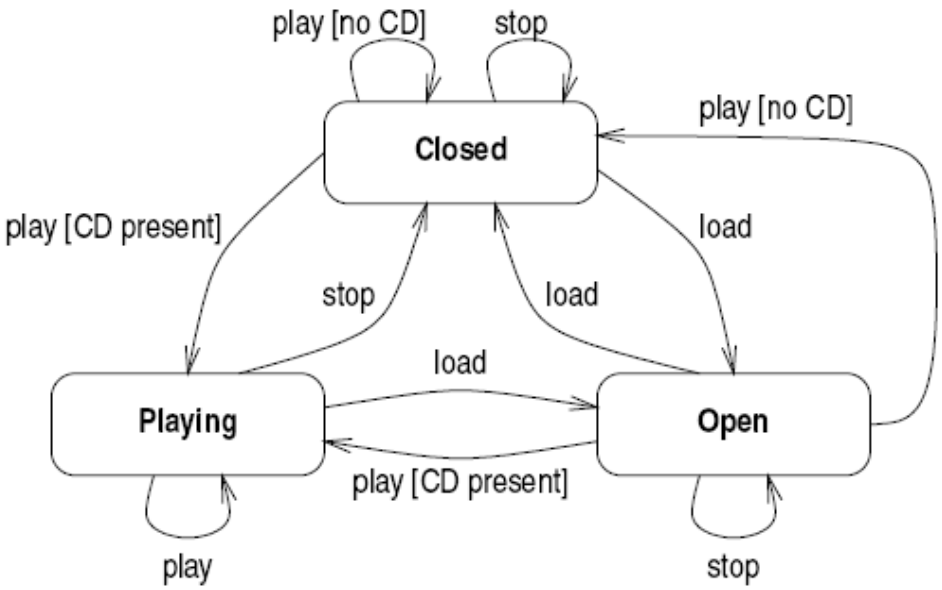


图 9.4　使用监护条件区分转移

如果有 CD，将激发从打开状态到正在播放状态的转移。状态机直接从打开转到正在播放状态，并且不经过关闭状态，即使是暂时地。如果经过，那么它必须检测到第二个事件以激发它到正在播放状态的转移。然而，只有单独一个事件，即按下播放按钮事件，是将它从打开状态转移到正在播放状态所必需的。如果需要，抽屉关闭的物理事实在状态图上可以作为动作建模，如下节所述。

9.5 动作

状态图能够说明对象响应检测到的特定事件时做些什么。这通过在图中的相关转移上增加动作（*action*）来表明。动作写在事件名字之后，前面加斜线“/”。图 9.5 所示的是 CD 播放机的状态图，加入了表示实际打开和关闭抽屉的动作。

动作可以用英语以伪代码的方式描述，也可以使用目标编程语言的符号。转移经常既带有条件，还带有动作。如果是这样，条件紧跟在事件名字之后，动作写在条件的后面，即“`event [guard condition] /action`”

动作被看作是简短的、自包含的一段处理，所花费的完成时间可以忽略。动作的定义特征是它在转移到达新状态之前完成。这隐含着动作不能由对象可能检测到的任何其他事件中断，而必须总是执行完成。在这个意义上，不是原子的动作，或者对象处于给定状态时执行的处理，不能用动作描述，而应该用*活动*描述，如第 9.6 节所描述的。

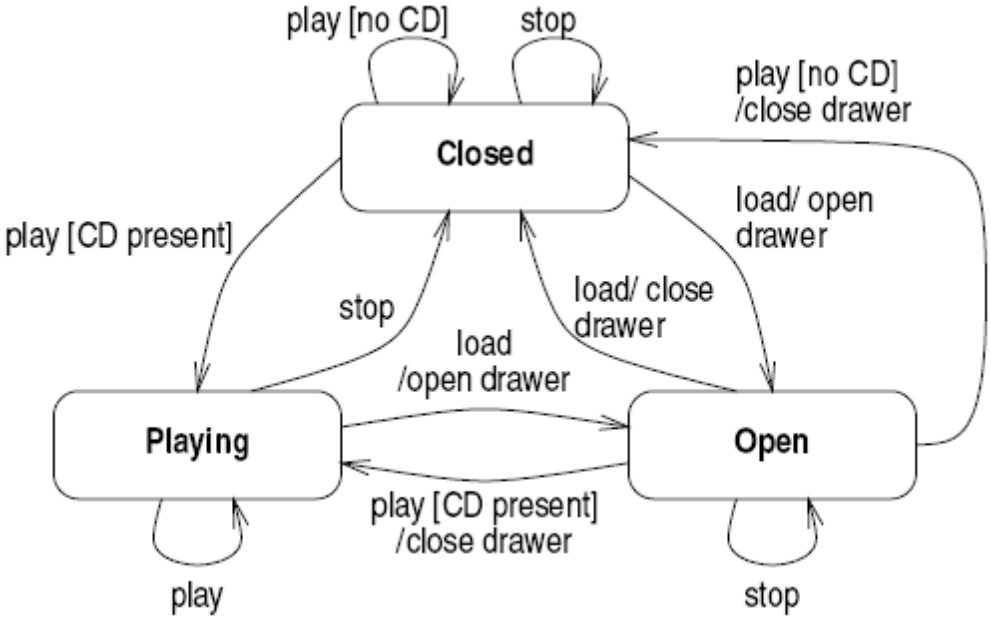


图 9.5　操纵 CD 播放机抽屉的动作

9.5.1 入口和出口动作

假定每次抽屉中有 CD 时如果按下播放按钮，CD 播放机的播放头都将自己定位到当前曲目的开头。这可以在状态图上表示，方法是在每个标注“play”的到达“Playing”状态的转移上写一个适当的动作。但是，这相当不妥而且累赘，可以用更节省的方法达到相同的效果，即在“Playing”状态中包含一个入口动作（*entry action*），如图 9.6 所示。

每当一个状态变为激活状态时，就执行入口动作，紧接在通向该状态的转移上的动作完成之后。例如，如果 CD 播放机处于打开状态，这时按下播放按钮，会关闭抽屉并激发到正在播放状态的转移。结果正在播放状态变成激活的，正在播放状态中的入口动作会立即被执行。

状态还可以有出口动作（*exit action*），只要离开该状态的转移激发时就会执行。图 9.6 中的出口动作表示只要执行了引起停止 CD 播放的动作，首先发生的事情就是抬起 CD 播放机的播放头。

注意，自转移被看作是状态改变。当一个状态上的自转移激发时，这个状态暂时不再是激活的，然后被再次激活。这意味着，当沿着自转移前进时，如果该状态存在入口动作和出口动作，则首先执行出口动作，接着执行入口动作。在图 9.6 中，这意味着当 CD 正在播放时按下播放按钮的结果是播放头回到当前曲目的开头，从而重新开始这个曲目。实际上很多 CD 播放机都展现出了这种行为。

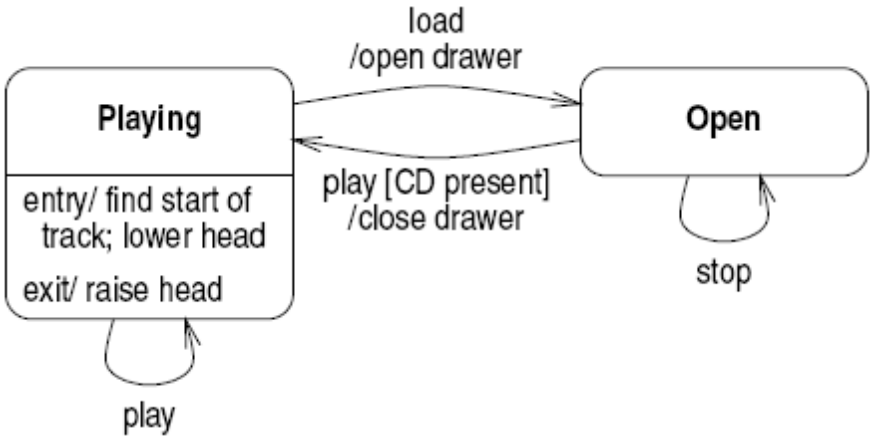


图 9.6　入口和出口动作

9.6 活动

显然，当处于正在播放状态时，CD 播放机正在做某些事情，即播放 CD 的当前曲目。要花费时间完成的持续操作可以被表示为状态中的*活动*（*activity*）。和动作一样，活动也写在状态之中，前面加上“do”标记，如图 9.7 所示。

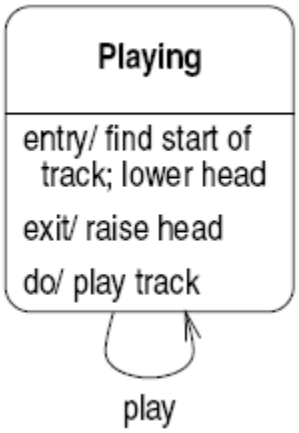


图 9.7　播放曲目的活动

动作和活动之间的区别是这样的，动作被认为是瞬时的，活动不同，是发生在一段延续的时期之内。当状态成为激活状态时，它的入口动作被执行，然后开始它的活动，并且在状态处于激活的整个期间该活动都持续运行。

对象必须在完成入口动作之后才能响应事件。然而，活动可以被任何激发离开该活动所在状态的转移的事件中断。例如，在曲目结束之前如果检测到“停止”事件，那么播放曲目的活动会被中断并停止。当离开一个状态的转移激发时，在执行出口动作之前，活动的执行被中断。

9.6.1 完成转移

除了被事件中断之外，有些活动也会自动结束。例如，图 9.7 中的 Playing 状态中的活动，如果到了曲目结束时就会出现这种情况。在某些情况下，活动终止会引起状态转移，状态图应该指定接下来哪个状态成为激活状态。

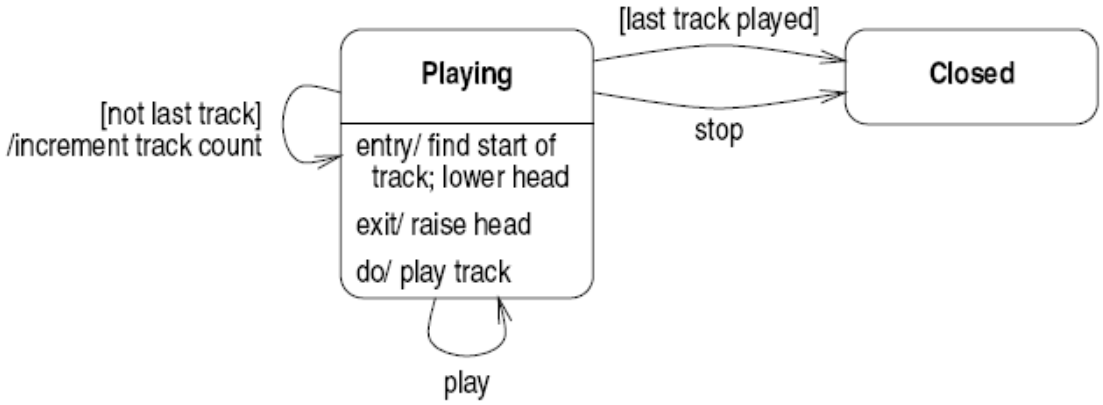


图 9.8 完成转移

可以利用*完成转移*（*completion transition*）做到这点。完成转移是没有事件标注的转移。在状态的内部活动正常终止，没有被外部事件中断时，可以触发完成转移。图 9.8 显示了 CD 播放机具有两个完成转移的正在播放状态，一个转移从正在播放状态通向关闭状态，另一个是正在播放状态上的自转移。当 CD 播放机正在播放时，用户可以按下播放或停止按钮中断当前曲目，如果这两个事件都没有被检测到，当前曲目最后会结束。在这种情况下，没有检测到外部事件，所以仅有完成转移是激发的候选转移。下来发生什么将取决于刚刚结束的曲目是否是 CD 的最后一个曲目。

完成转移带有监护条件以区分这两种情况。如果刚刚播放完了最后一个曲目，到关闭状态的转移将激发，CD 播放机将完全停止播放。否则，将激发自转移：曲目计数器递增，再次进入正在播放状态，CD 播放机将开始播放 CD 上的下一个曲目。

9.6.2 内部转移

如上所述，自转移被认为是状态改变，所以如果图 9.8 中的任一个自转移激发，正在播放状态中的活动将终止，并且在再次进入该状态之前将执行状态的出口动作，然后执行入口动作，重新开始状态的活动。

有时，需要对让对象停在同一状态，但是不触发状态的改变以及入口与出口动作的执行的这样的事件建模。例如，假定 CD 播放机有一个“信息（info）”按钮，当按下时显示当前曲目剩余的时间，但是不中断正在进行的曲目播放。

这可以作为正在播放状态中的*内部转移*（*internal transition*）来建模。内部转移写在状态之中，标注为引起该转移的事件的名字，如图 9.9 所示。和自转移不同，内部转移不会引起状态的改变，因此也不会触发入口和出口动作。

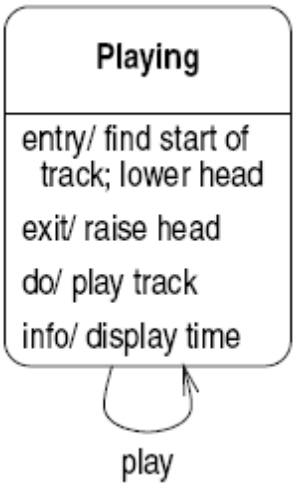


图 9.9　有内部转移的状态

9.7 组合状态

图 9.5 相当混乱和难以理解，其中还存在着一些冗余，某些转移以实质上相同的形式出现了不止一次。如果状态图要在实际中可以用于复杂系统，就需要某些简化图的方法。允许一个状态包含若干子状态，就是一种简化技术。这些子状态，因为它们共享了某些特性，这些特性可以被更简明地表示为单独一个“超状态”的特性，而组合在一起放入一个状态中。

状态可以和其他状态共享的一个性质是它们的行为，或换句话说是它们参与的转移。例如，当 CD 播放机处于打开或关闭状态时，如果抽屉中有 CD，它对 play 事件的响应是一样的，即转到正在播放状态，并播放 CD。稍不明显的是，即使没有 CD 的时候，响应也是相同的：播放机结束于关闭状态。这可能涉及状态的改变，也可能不涉及，取决于抽屉原先是打开的或关闭的，但是事件的实际结果是相同的。

图 9.10 所示的 CD 播放机的状态图用超状态析出了这个公共行为。图中引入了一个名为“未播放（not playing）”的新状态，而打开和关闭状态现在显示为这个新状态的子状态。“未播放”状态被认为是由两个嵌套的子状态组成的一个组合状态（*composite state*）。

这个新状态的存在只是为了集合 CD 播放机的相关状态，并没有引入任何新行为的可能性。组合状态具有下面的特性。第一，如果组合状态是激活的，那么它的子状态中只有一个必须也是激活的。所以在图 9.10 中，如果 CD 播放机没有播放，那么它必须处于 Open 或 Closed 两个状态中的一个。

第二，在对象处于组合状态时检测到的事件可以触发从组合状态本身出发的转移，或者从当前激活的子状态出发的转移。例如，假设 CD 播放机处于关闭状态。如果检测到一个装入事件，将激发通向打开状态的转移，而打开状态将成为激活的。然而，这是未播放状态的一个内部转移，所以它仍然是有效的，只不过有一个不同的激活子状态。

但是假如检测到的是 play 事件。不存在从关闭状态出发的标注为“播放”的转移，但是有从未播放状态出发的这样的转移。因为这个状态也是激活的，所以这些转移将被激活，并且根据抽屉中是否有 CD，它们之中的一个或其他将激发。如果有 CD，正在播放状态将变成激活状态。如果没有 CD，关闭状态将变成激活的，不过是通过从未播放状态出发的自转移。

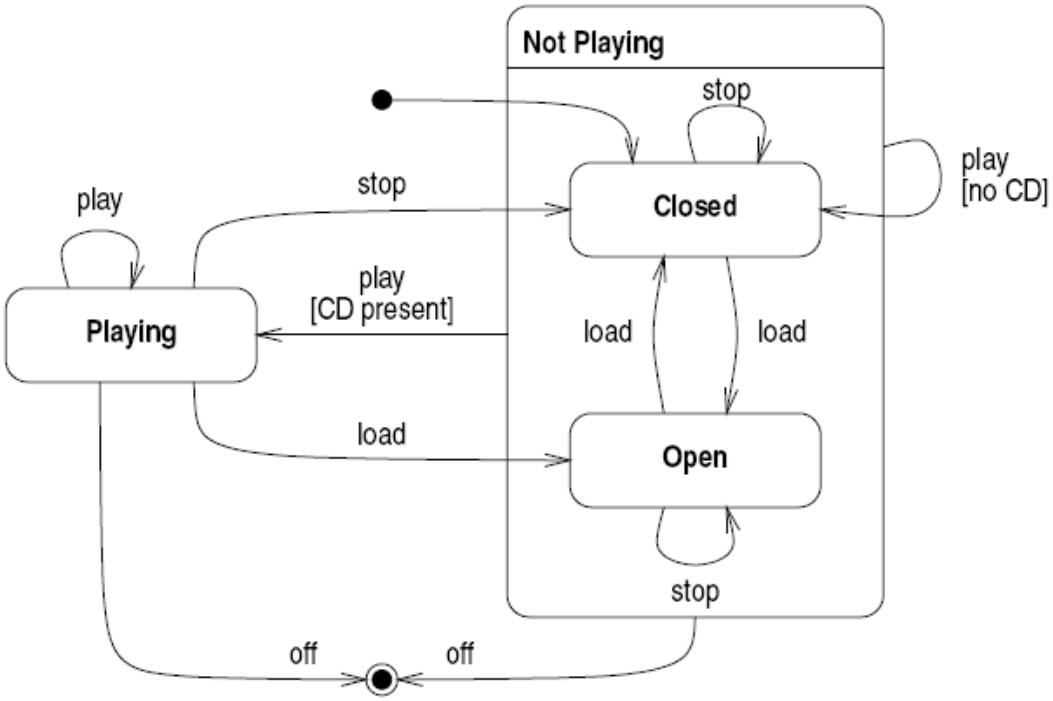


图 9.10　使用子状态的 CD 播放机

子状态完全是正常的状态，并且到达它们的转移能够自由地穿过超状态的边界。图 9.10 中的从 Playing 状态出发的 stop 和 load 转移说明了这个特性。它们穿过了“Not Playing”状态的边界，但是在形式上和含意上对图 9.5 都没有改变。转移也可以连接一个超状态中的若干子状态，如 Open 状态和 Closed 状态之间的装入转移所示例的一样。最后，转移也可以从子状态到达超状态之外的一个状态，尽管图 9.10 中没有包含相应的例子。

9.7.1 组合状态的特性

组合状态中的嵌套状态形成了一种“子状态图”，并且，除了普通的状态，组合状态还可以包含初始状态和终止状态。组合状态中的初始状态表示：如果到达组合状态的转移终止于组合状态的边界时，该默认子状态即成为激活状态。组合状态中的终止状态表明状态中正在进行的活动已经完成。到达终止状态使得从组合状态出发的完成转移能够激发。

组合状态也可以有自己的入口和出口动作。这些状态被激活的方式与简单状态每当状态变成激活的或不再是激活的时方式完全相同。

例如，假设按下 CD 播放机上的暂停按钮会引起播放被中断。当再次按下这个按钮时，从暂停的位置开始继续播放，也就是说，和按下播放按钮的情况不同，曲目不用重新开始。图 9.11 中的状态图模拟了这种行为。

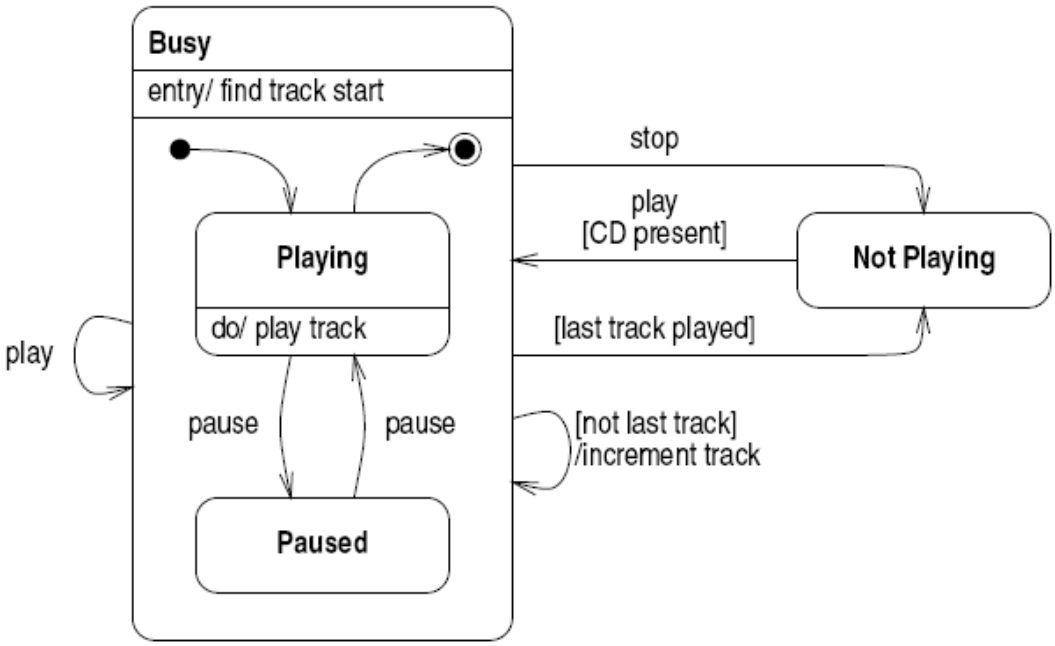


图 9.11 “暂停（pause）”按钮的建模

通过下面一些详细的事件序列可以更好地理解这个图。假如 CD 播放机处于未播放状态而且抽屉中有 CD 的时候，按下了播放按钮，那么标注为“播放”的转移将激发，使标记为“忙碌（busy）”的状态成为激活状态。因而执行该状态的入口动作，定位到当前曲目的开头。但是，这个转移没有指定忙碌状态的哪个子状态变成激活的，所以从初始状态到正在播放状态的转移激发，使正在播放状态成为激活状态。因此，开始播放当前曲目的活动。

如果用户没有做任何事情打断这个过程，在曲目结束时，忙碌状态中从正在播放状态到终止状态的完成转移将激发，这是状态的活动终止时的正常行为。接着触发从组合状态出发的一个完成转移。假如还有另外的曲目要播放，将激发忙碌状态上的自转移，曲目计数器递增，并再次进入忙碌状态。如前所述，这将引起定位到新曲目的开头并开始播放。

现在，假如用户在曲目结束之前按下了暂停按钮，这将中断播放曲目的活动，并引起到达“暂停（paused）”状态的转移激发。当用户再次按下暂停按钮时，回到正在播放状态的转移激发，并重新开始播放该曲目的活动。然而，在这种情况下，所有的转移都是忙碌状态内部的，所以不会触发定位到曲目开头的入口动作。因此，播放头不会移动，播放是从中断的那点重新开始，如所需要的那样。

9.8 历史状态

假如 CD 播放机的行为如同图 9.11 所描述的，并且用户在 CD 播放机处于暂停状态时按下了播放按钮，那么这将激发忙碌状态上的标记为“播放”的自转移，因而将退出暂停状态，再次进入忙碌状态。入口动作将导致找到曲目的开头，因为自转移只是通向组合状态的，因此沿着从初始状态出发的转移前进，使机器停留在正在播放状态，播放 CD。

但是，假如 CD 播放机展现出的实际行为不是这样的，而是在 CD 播放机暂停时按下播放按钮将重新开始该曲目，但播放机仍然处于暂停状态。重新开始播放之前，用户必须再次按下暂停按钮。对此建模的一种方法是将忙碌状态上标注为“播放”的自转移用两个自转移代替，一个在正在播放状态上，一个在暂停状态上。

然而，如果到组合状态的转移能够“记住”上次组合状态激活时哪个子状态是激活的，并能够自动返回到那个子状态，就可能避免重复同样的转移。如果 CD 播放机是在播放，那么按下“播放”它应该从曲目开始继续播放，但如果播放机暂停，那么它将一直暂停到再次按下暂停按钮。

使用如图 9.12 所示的*历史状态*可以达到这个效果。历史状态由圆圈中一个大写字母“H”表示，并且只能出现在组合状态之内。到达历史状态的转移引起组合状态中最近的激活子状态再次成为激活的。于是，如果在 CD 播放机暂停时按下“播放”按钮，将沿忙碌状态上的自转移进行，终止在历史状态。这将引起一个到上个激活子状态的隐含转移，在这个例子中即暂停状态，如同所需要的那样。

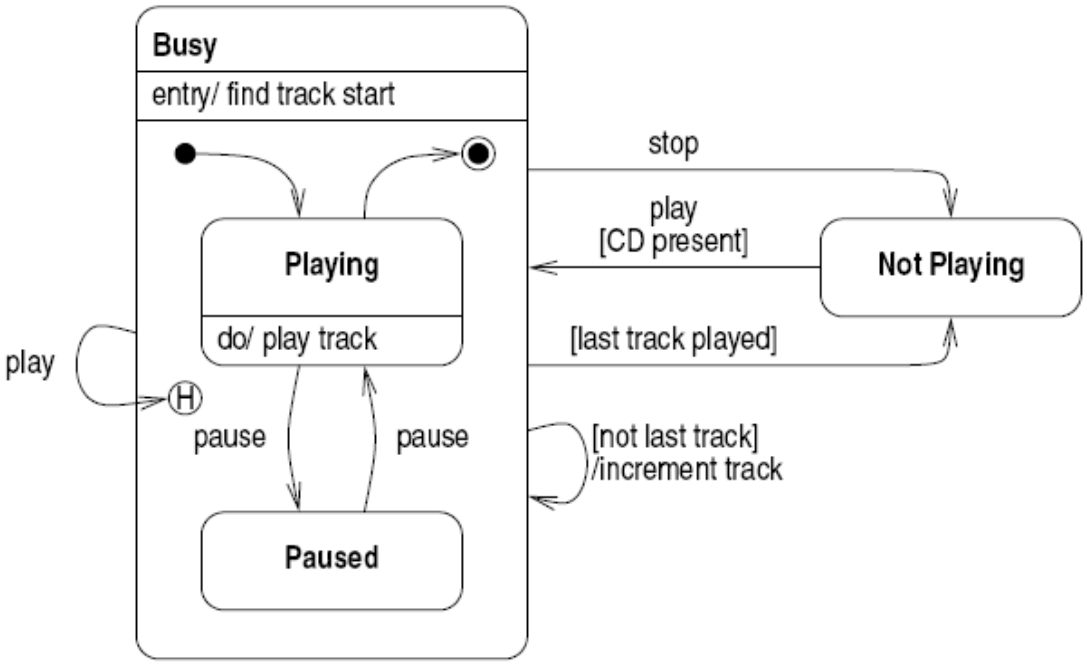


图 9.12　历史状态

在 CD 播放机暂停时，如果按下了停止按钮，随后按下“播放”，图 9.12 规定，CD 播放机将回复到正在播放状态。如果要求它应该仍然是暂停的，即使播放停下来又再开始，这可以通过将播放转移的末端从忙碌状态的边界延伸到历史状态来建模。

这引发了一个问题，如果历史状态是忙碌状态的第一个激活子状态，将出现什么情况：根据定义，在这种情况下应该没有记忆的历史。在这种情况下，我们必须指定一个默认状态成为激活的。这可以通过从历史状态向需要的默认状态画一个无标注的转移实现，在这个例子中，默认的是正在播放状态。

9.9 CD 播放机总结

图 9.13 显示了一个描述 CD 播放机的行为的完整的状态图，结合了本章讨论的许多要点。这个图源于图 8.10 和 8.12 的合并。在“未播放”状态中加入了一个初始状态，还加入了另一个历史状态，表示在没有 CD 播放时按下停止按钮没有作用，不会引起播放机的状态改变。

对这个图进一步的扩充和修改建议作为本章后面的习题。

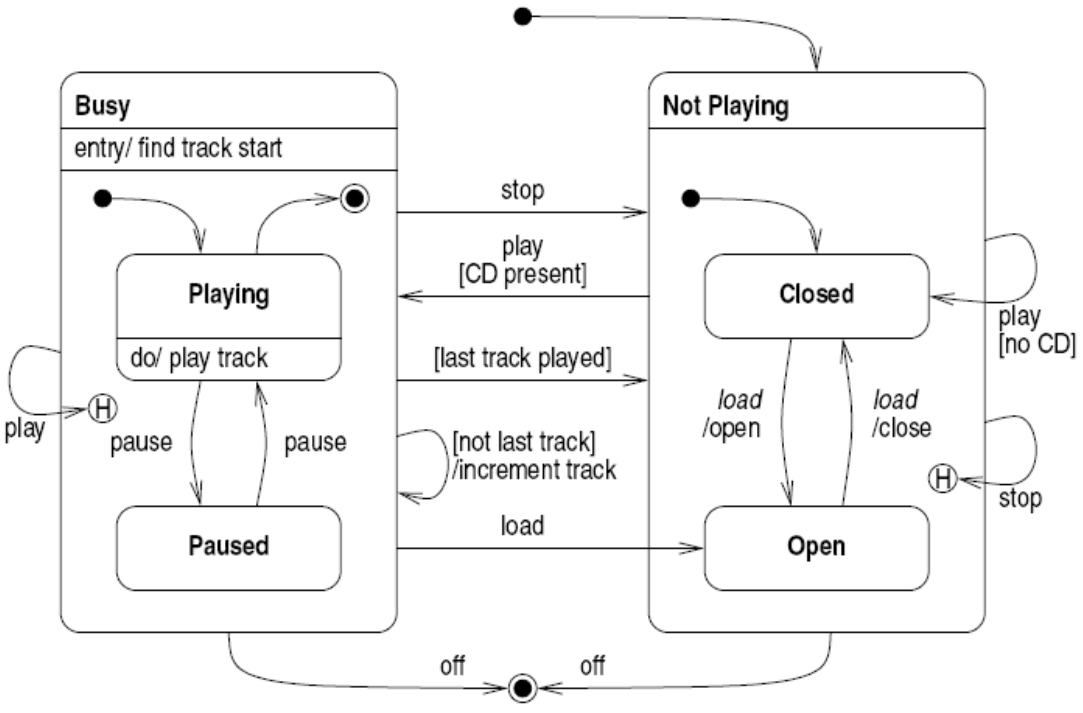


图 9.13　CD 播放机完整的状态图

9.10 实际中的动态建模

本节举例说明一个有用的构造状态图的过程，并以此说明如何用来自交互图的信息如何能够导出状态图。使用的例子是下面几段描述的自动售票机。

售票机可以接受乘客输入货币和其他，并在成功交易结束时输出需要的车票以及找的零钱。没有正在进行的交易时，机器显示“需要准确钱数”信息或者“可找零钱”信息。显示的信息决定了下一位顾客是必须输入确切数目的货币为所选的车票付款，还是在输入超过所需的钱数时机器能够找零。

机器的界面由几个按钮组成，每个按钮对应给定的一种车票。如果用户按下这些按钮之中的一个，机器就显示要输入的钱数，即车票的价钱。随着用户向机器输入货币，显示的数目就根据用户输入的数量减少。输入的钱数一超过或等于车票的价钱，就发生下面的两件事情之一。如果机器最初显示的是“可找零钱”信息，就输出需要的车票以及所需找的零钱。如果显示的是“需要准确钱数”信息，并且用户输入的恰好是所需数目的货币，将只发售一张车票。如果用户输入了过多的货币，将返回所有输入的货币。

在交易开始，用户可以选择在选择车票种类之前先输入货币。如果在最终选择了车票的时候，已经输入了足够付款的货币，就像前面的情况一样输出货币和找的零钱。如果输入的钱数少于所需车票的价钱，机器将显示剩余的费用并像上面的情况一样继续进行。

在任何情况下，如果在 30 秒内没有收到用户的输入，交易将被终止，已经输入了多少钱都会被返回给顾客。“取消”按钮也可以使用户明确地表示终止交易。

9.10.1 状态机和事件序列

状态图概括了一个对象能够接收的所有可能的事件序列。然而，要识别为准确建立对象的行为模型所需要的所有状态，有时相当困难。这里介绍的技术一次只考虑一个事件序列，逐步建立起所需状态图的完整描述，从而避免了这种困难。

从交互图中可以获得多个各自独立的事件序列。到达一个对象的消息，如果按照对象接收它们的顺序组织起来，就构成了这样的一个序列。在该对象的状态图上，必须可能找到对应于各个这样的事件序列的路径。

在构造状态图时，我们可以从选取一个序列，并定义一个只表示该序列的简单的状态图开始。然后，将更多的事件序列集成到这个初步的状态图中，用这种方法，可以用逐步的方式建立起一个完整的状态图。

本节剩下的部分将以售票机的例子说明这个过程。我们考虑售票机设计的一些典型事务，并逐步建立起一个完整的状态图。正式的对象交互图在此没有画出，因为这些事务中只涉及一个对象，即售票机本身，而且发送的消息只是由用户产生的事件序列。

9.10.2 付款之前选择车票

假如用户首先选择了特定种类的车票，然后相继输入了三枚硬币。这三枚硬币的总值超过了车票的价钱，所以机器输出车票以及需要找的零钱，并回到空闲状态，等待下一次交易。在这种情况下，机器接收到的是包含四个事件的序列：一个“车票（ticket）”事件，随后三个“硬币（coin）”事件。这些事件每一个都有相关的数据，即所选车票的价钱和输入的各个硬币的价值，但是我们将暂时忽略这些细节。

假定每个事件对应于两个状态之间的一个转移，就可以直接为任何单独的事件序列画出一个非常简易的状态图；在实行中，我们在序列的开始和结束以及每对事件之间放上状态。在现在的例子中，我们得到图 9.14 所示的状态图。因为这只是一个初步的图，所以没有为这些状态加上标注。

图 9.14　售票机的初步状态图

尽管这个状态机是所考虑的单个事件序列的精确的模型，但是还需要对它进行一些调整，才足以作为售票机的基础。首先，图 9.14 只定义了一次交易，然而售票机能够一个接一个地执行重复的交易。为了对此建模，我们可以合并图 9.14 中的第一个状态和最后一个状态。因为这些状态代表了没有正在进行交易的情形，我们将这个新状态标记为“空闲（Idle）”。

其次，在上面示例的交易中，硬币的数目实质上是任意的。在一次交易中可以输入任意数目的硬币：需要的确切数目取决于所选车票的价钱和输入硬币的面值。需要用某种循环来表示输入任意数目硬币的可能性。这可以通过将图 9.14 中的三个中间的状态合成为一个带自转移的状态实现。

最初交易的状态图的一个改进版本如图 9.15 所示，其中并入了这些修改。这个图中只包含两个状态，它是由图 9.14 最初的五个状态导出的。机器从空闲状态出发，当用户选择车票时，沿着通向“付票款”状态的转移前进。在随后输入硬币的时候，可能发生两件事情之一。如果输入的钱币总量足够付清所选车票的价钱，则跟随的是回到空闲状态的转移。但是如果还需要更多的钱，就沿着付款状态上的自转移循环，并且必须输入更多的硬币才能继续处理。

图 9.15 使用了前面介绍的一些状态图表示法来阐明机器在这次交易中的行为。首先，各种消息附上了参数，用以传送车票的价钱和输入机器的每个硬币的面值。其次，有两个从付款状态出发的标注为“coin”的转移。它们由一个非正式的监护条件区分，该条件检查是否已经输入了足够为所选车票付款的钱。最后，在回到空闲状态的转移上还显示了动作，如果已经输入了足够的钱，机器将执行该动作。在这个例子中，我们假定机器能够输出需要找的零钱。

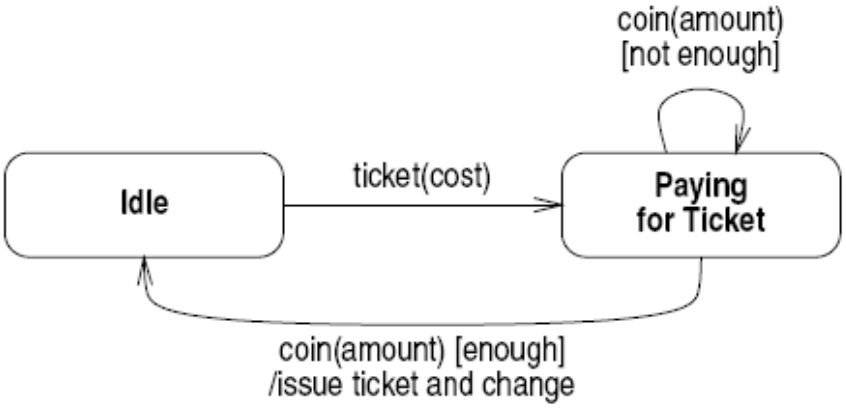


图 9.15　改进的状态图

9.10.3 选择车票之前付款

我们现在已经处理了一种可能的交易，并构造了一个初步的状态图。下一步是考虑第二种交易，并尽可能将它集成到已有的状态图中，并在必要时扩充状态图。

除了首先选择车票，售票机的说明还允许用户先输入钱，再选择需要的票。对应于这个交易的事件序列是以若干硬币事件开始，然后在接收到车票事件时输出车票和找的零钱，并结束交易。我们可以假定，交易从图 9.15 中已经标识的空闲状态开始。然而，第一个事件存在一个问题，因为图 9.15 中不包含从空闲状态出发的标注为“硬币”的转移。我们因此需要用一个适当的转移扩充状态图，通向一个新状态。

同前面的情况中一样，可以输入任意数目的硬币，而我们可以利用这个新状态上的自转移对此建模。最后，会接收一个车票事件，而机器将输出票和找的零钱，并返回到空闲状态。图 9.16 所示的状态图增加了这些内容。

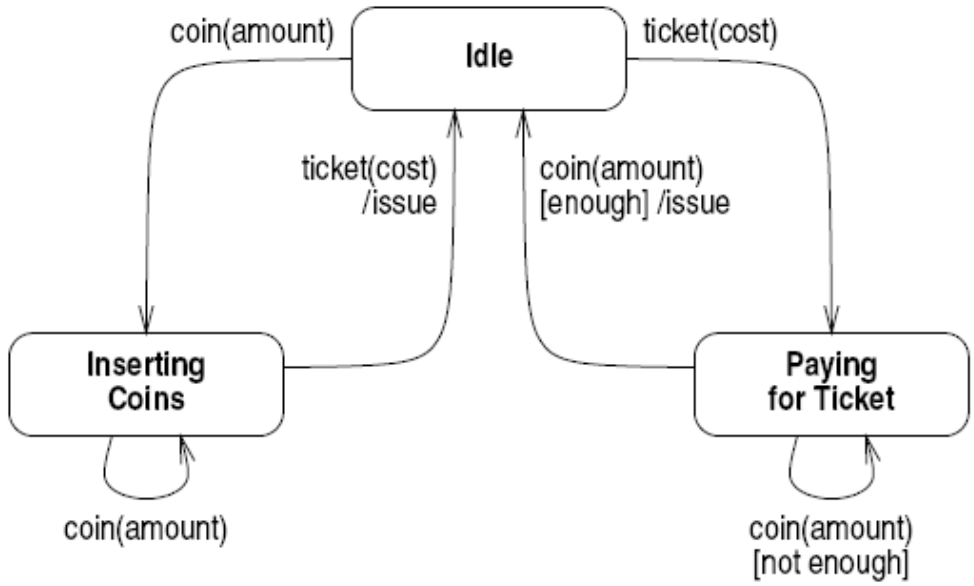


图 9.16　合并第二个交易

9.10.4 集成交易

至今所考虑的两个交易互为镜像。在第一个交易中，在输入硬币之前选择车票类型，在第二个中，输入所有硬币之后选择车票类型。然而，售票机的说明书允许第三种可能性，在选择车票类型之前先输入一些硬币，不过还需要更多的硬币补足票价。图 9.16 并不满足这种可能性：一旦已经输入硬币，选择车票类型使状态机回到空闲状态，没有输入更多硬币继续交易的可能性。

需要的是增加一个标注为“车票”的转移，从图 9.16 中的“插入硬币”状态出发。这个转移应该用一个条件和已有的转移区分开。必要的条件和在用户付票款时区分插入硬币是否会引起机器回到空闲状态的条件相同，即是否已经输入了足够的钱数付清所选车票的价钱。

这个新转移可以通向一个新状态：为了满足售票机的说明，新状态将必须允许输入硬币直到总计达到票价，随之将售出车票，机器返回到空闲状态。然而，这正好是已有的“付票款”状态提供的行为，所以可以定义新的转移到达这个状态，如图 9.17 所示。状态的名字也有所改变，以便更准确地反映它们之间的相关差异，即用户是否已经选择了车票类型。

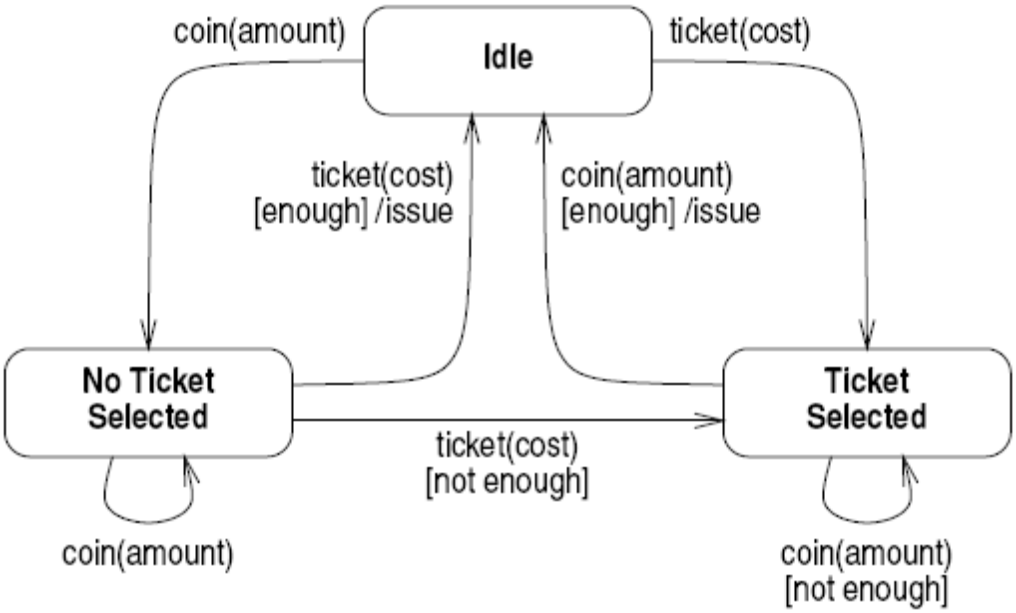


图 9.17　集成两个交易

从任何给定的事件序列对应于穿过状态图的一条连通路径的意义上来说，图 9.17 的状态图概括了在以输出车票结束的交易期间用户能够产生的所有可能的事件序列。在一个交易中，可以输入任意数目的硬币，车票类型可以选择一次，并且这些事件能够以任何次序发生。

然而，售票机的用户的行为并不总是如此明智。容易想象，某个用户选择了一种车票，然后在输入硬币之前改变了主意又选择了另一种车票。图 9.17 的状态图不允许这种可能性。一旦选择了车票种类，机器就进入“已选车票（ticket selected）”状态，而且并没有从这个状态出发的车票转移。在用户实际能做的和状态图所规定的之间在这里似乎存在一个矛盾。

解释这种现象的一种方法是注意尽管用户实际上可以重复地按下车票选择按钮，但是这未必意味着机器接收了“车票”事件。例如，情况可能是一旦进入了“已选车票”状态，就使车票选择按钮无效，并且只有再次到达空闲状态时才重新激活。

如果不是这样，就需要在状态图中给出额外车票事件的明确说明。这能够以多种方式实现，如果这样的事件是不被接受的，可能通过引入一个错误状态实现，或者如果是允许的，用“已选车票”状态上的一个自转移实现。当然，在实际例子中，在这些选项之间的选择将由正在建模的售票机的实际行为决定。

图 9.17 准确地模拟了售票机的基本行为。为了完成模型，表明能够终止交易的其他方式，即由于按下取消按钮或者由于超时而终止，以及在机器要求输入恰好等于票价的情况下其他的行为，都是必要的。在完成售票机状态图之前，将先介绍对这些特征建模的所需符号。

9.11 时间事件

如果 30 秒都没有收到来自用户的输入，售票机将超时：当前交易将被终止，输入的钱被退回给顾客。超时应该被作为一个转移建模，因为它将改变售票机的状态，从交易中的一个中间状态回到空闲状态。然而，应该用什么事件标注这样一个转移并不明显：毕竟，要点是这样的转移必须在没有检测到事件的时候精确地激发。UML 定义了专门的时间事件，可以用于这些情况中。图 9.18 显示了一个转移，在进入“未选择车票（no ticket selected）”状态 30 秒后将激发。这可以如此理解，想象每个状态的一个隐含活动是执行一个计时器，在每次进入该状态时复位。一旦计时器已经运行了时间事件所规定的一段时间，就激发从状态出发的标注有时间事件的转移。在图 9.18 中，注意，每次输入硬币时计时器复位，因为自转移被认为是状态的改变，并触发状态的入口动作。

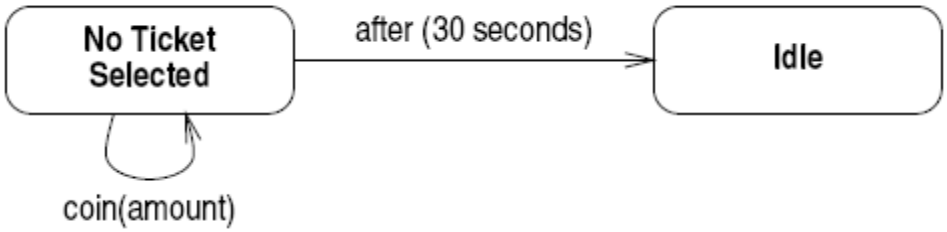


图 9.18　时间事件

在关键字“after”之后，可以给出任何一段时间作为参数。时间事件的另一种形式由关键字“when”后跟一个指定的时间点组成，这定义了一个在到达规定时间时将激发的转移。

9.12 活动状态

在图 9.17 中，两个转移在交易顺利完成后回到空闲状态。在每种情况中，都有必要检查机器是否能够退回任何需要找的零钱。如果可以，应该输出找的钱和车票，如果不能，应该退回输入的钱。

这个行为可以用一对具有适当监护条件和动作的转移表示，但这些不得不在每个回到空闲状态的路线上重复。为了避免这种重复，可以使用*活动状态*（*activity state*）简化状态图的结构，如图 9.19 所示。

活动状态表示对象执行某些内部处理的一段时间。照此，它在状态图上表示为只包含一个活动的状态。在图 9.19 中，只要顾客对一个交易的输入一旦完成，活动状态就成为激活的，对应于机器计算它是否能够返回为完成该交易所需要找的零钱。

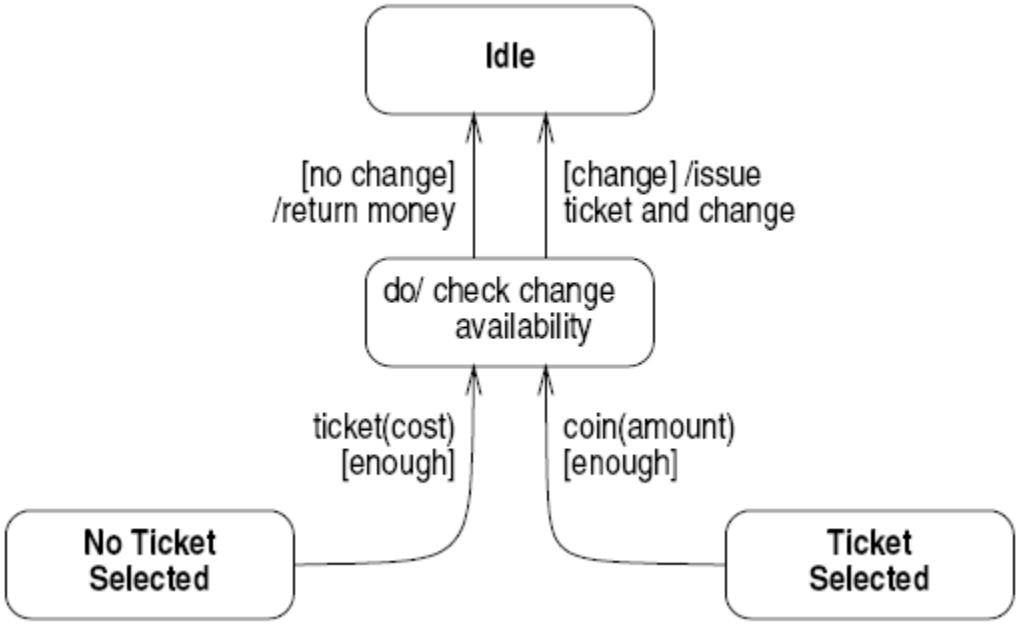


图 9.19 活动状态

活动状态表示的内部处理通常不能被外部事件中断，在这些情况下，离开活动状态的唯一转移将是完成交易。在图 9.19 中，有两个这样的转移，通过表明是否有找零头的监护条件区分。在每个情况下，这些转移还带有相应的动作。

活动状态在状态图中应该慎用，因为状态图的目的通常是说明对象对外部事件的响应，而不是对内部处理详细地建模。然而，有时它们很有用，如在图 9.19 中，作为简化状态图结构的方法。

9.13 售票机总结

图 9.20 给出了售票机的一个完整状态图。它合并了前面几节指出的各种要点，还显示出了用户在交易中间按下取消按钮的结果。

为了减少说明交易由于超时或取消而中断所需的转移的数目，图 9.20 中包括了一个组合状态，其意图是对应于交易正在进行的时间段。

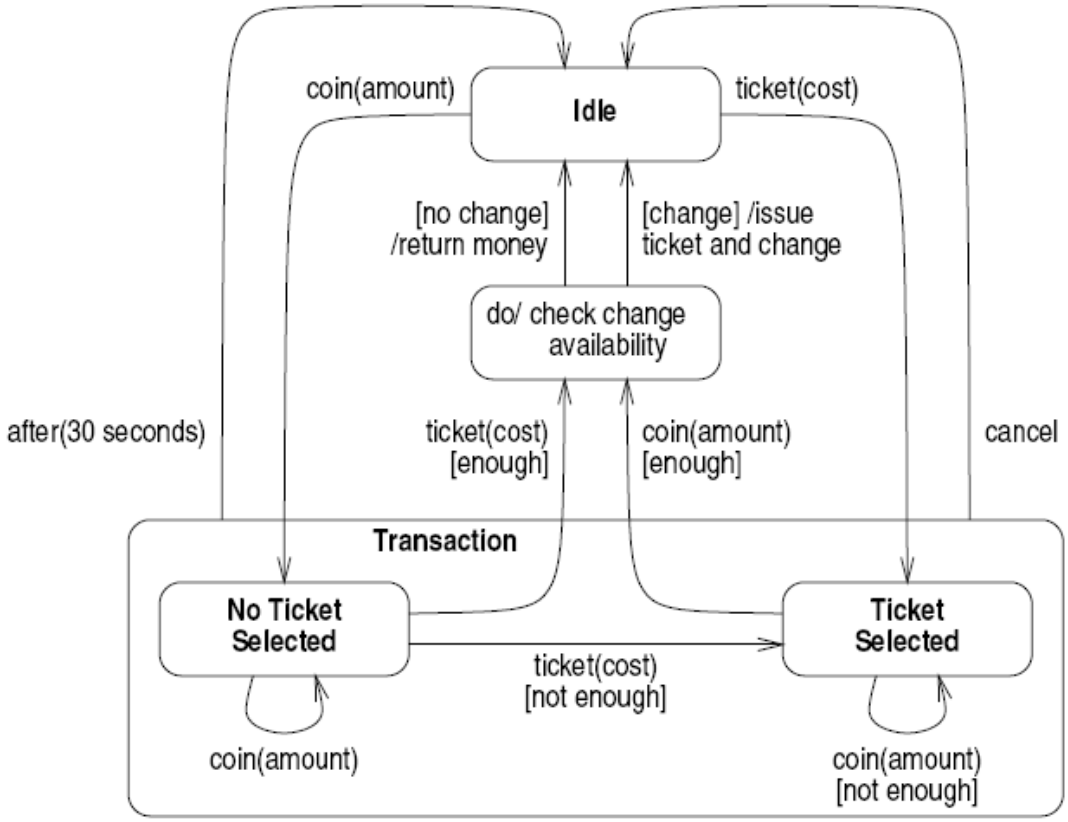


图 9.20　售票机的完整状态图

9.14 小结

- 状态图提供了可以在交互图上说明的对象的行为方面的规约。
- 状态图表明了对象在整个生命期期间能够检测到的事件以及对象对这些事件的响应。
- 一般地，对象展示出依赖于状态的行为。状态的不同由检测到一个由对象处于什么状态而决定可能有不同结果的事件来区分。
- 检测到一个事件可以引起一个转移激发，对象从一个状态转到另一个状态。
- 监护条件能够用于表示在特定时刻，一组转移中的哪个转移实际上被激发。
- 动作说明了对象对事件的响应。状态的入口和出口动作分别等价于所有到来和离开转移上的动作。

- 状态可以包含*活动*，活动发生在对象处于该状态的整个时间。活动可以被用户产生的外部事件中断。如果活动没有中断而结束，将跟随离开该状态的一个完成转移。
- 组合状态可以用于简化复杂的状态图。从超状态出发的转移同等地适用于所有嵌套的子状态。到超状态的转移的结果通过超状态内的初始状态指定。
- 状态图可以通过依次考虑源于对象交互图的各个事件序列来开发。首先开发一个简单的状态图模拟一个这样的序列，然后在考虑其他序列时再进行必要的扩充。

9.15 习题

（1） 图 9.13 说明的 CD 播放机在检测到下列事件序列后会处于什么状态？假定测试时一直有 CD。

(*a*) 初始化（initialize），装入（load）。

(*b*) 初始化（initialize），装入（load），播放（play），停止（stop）。

(*c*) 初始化（initialize），装入（load），播放（play），暂停（pause），播放（play）。

(*d*) 初始化（initialize），播放（play），停止（stop），装入（load）。

(*e*) 初始化（initialize），装入（load），暂停（pause），播放（play）。

（2） 这个问题引用了图 9.13 中给出的 CD 播放机的动态模型。假如在播放机处于打开状态，抽屉中没有 CD，并且抽屉是打开的，这时关掉播放机，然后马上再开机。在这些操作之后，CD 播放机处于什么状态？CD 播放机的抽屉的物理状态是什么？如果现在用户按下装入按钮，会发生什么？为了关闭 CD 播放机的抽屉，用户必须按下哪个按钮？

（3）画出图 9.12 的修改版本，对即使在按下“停止”然后按下“播放”时 CD 播放机应该仍然保持在暂停的需求建模。

（4） 在图 9.13 的未播放状态中，是否需要从历史状态出发的默认转移？如果不需要，为什么？如果需要，它应该到哪个子状态？

（5） 本章 CD 播放机的描述中，对它如何记录哪个是当前曲目的细节还不明确。假定 CD 播放机有一个称为“曲目计数器（track counter）”的属性，其行为如下。

抽屉中没有 CD 的时候，曲目计数器置为 0。当检测到 CD 时，曲目计数器置为 1；这在检测到装入或播放事件后实际关闭抽屉的时候发生。无论何时进入忙碌状态，曲目计数器决定定位于哪个曲目开头，并从而确定播放哪个曲目。

标注为“前进（forward）”和“后退（back）”的两个按钮允许用户手工调节曲目计数器。如果抽屉中没有 CD，这些按钮不起作用。否则，按下“前进”曲目计数器递增，而按下“后退”减少。当 CD 播放机处于忙碌状态时，按下这两个按钮之一会使播放头立即移动到所要求曲目的开始。

扩充图 9.13 中的状态图，以对此行为建模。

（6） 在 9.10 节讨论的售票机的例子中，假如一旦选择了车票类型，车票选择按钮就无效，直到交易结束时才重新激活。另外，假如已经输入了足够购买所需车票的钱，硬币的投币口就关闭，只有在输出任何车票和找零头时才重新打开。用入口和出口动作在图 9.20 的状态图上明确表示这种行为。

（7） 去掉活动状态，重画图 9.20，并从清晰性和易理解性的角度对得到的状态图和图 9.20 进行比较。

（8） 修改图 9.20 的状态图，使得在交易开始检查找零头的可用性，并显示一个适当的信息，如 9.10 节所说明的。

（9）重画图 Ex9.9 给出的状态图，去掉所有的组合状态，并用剩余状态之间的等价转移带替所有到达和从组合状态出发的转移。

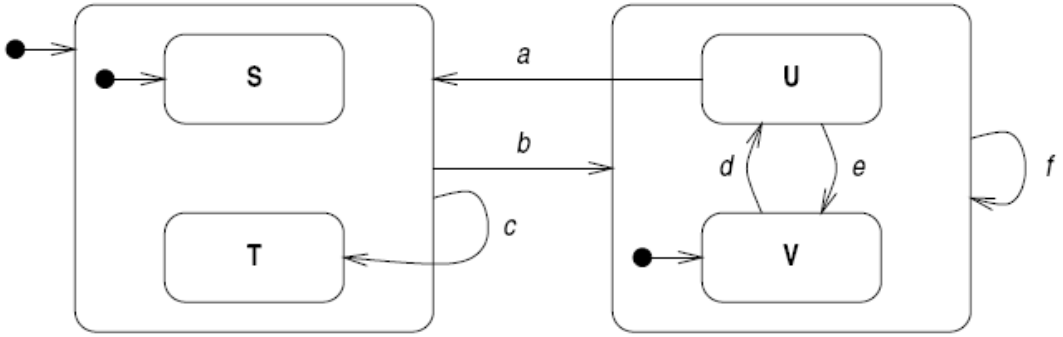


图 Ex9.9 使用嵌套状态的状态图

（9）找出一个接受但另一个不接受的事件序列，说明图 Ex9.10 中的两个状态图的含意不等价。假定序列从初始状态开始。

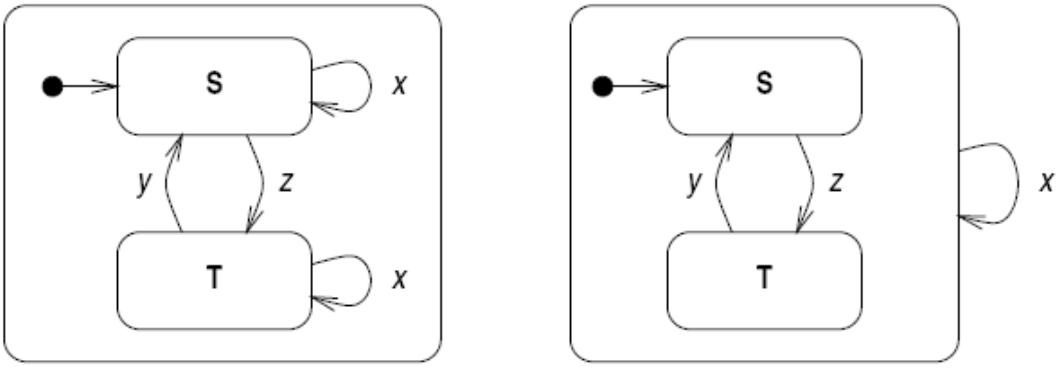


图 Ex9.10 两个不等价的状态图

（11） 阅览室中的灯是由一个分别标记为“On”、“Off”和“Dim”三个开关的面板控制。“On”把灯打开到它们的最大亮度，“Off”关灯。还有一个中等的亮度，在演示幻灯

片和其他投影材料时使用。“Dim”开关将亮度从最亮降低到这个中等亮度；再次按下“On”开关，将恢复最大亮度。画一个状态图对这个阅览室的照明系统的行为建模。

（12） 窗口管理系统中的窗口能够以三种状态之一显示：*最大化*，占据整个屏幕；*正常*，在屏幕的给定位置上显示为一个给定大小的有边界的窗口；*图标化*，显示为一个小图标。当窗口打开时，它显示为一个正常的窗口，除非选择了*自动最小化*，在这种情况下它将被显示为一个图标。正常的窗口和图标可以被最大化；最大化的窗口和正常窗口可以被最小化，或缩小为一个图标。最大化的窗口可以恢复到正常大小，图标可以恢复到最小化之前的大小。图标和正常窗口可以移动，正常窗口还可以调整大小。无论怎样显示，窗口总可以被关闭。画一个状态图表达有关窗口显示的这些事实。

（13） 下面给出了对自动柜员机（ATM）的行为的描述。产生一个描述其行为的状态图。列出你由于描述的二义性、不清晰或不完整而必须做出的假设。

用户通过插入一个银行卡开始在 ATM 上的事务。假定该卡是机器可读的，用户被提示输入他们的个人身份号码（PIN）。一旦输入了这个号码，就呈现给用户一个菜单，包含下列选项：显示账户余额，取款并要收据，以及取款不要收据。如果用户选择了取款选项之一，将提示他们输入取款的金额，输入的金额必须是 10 的倍数。

ATM 向银行的远程计算机发送事务详细资料时验证用户的 PIN。如果 PIN 是无效的，给用户重新输入 PIN 的选择，并重试所选的事务。如果新的 PIN 还是无效的，重复这一过程。一旦输入了三个无效的 PIN，事务终止，并且机器将吞入用户卡。

如果输入了有效的 PIN，进一步的处理取决于选择的事务类型。对“显示余额”事务，在屏幕上显示余额，在用户确认之后，回到事务菜单。如果用户超过了可以从该账户提取的金额，取款事务可能失败；在这种情况下，显示一个错误信息，在用户确认后，回到事务菜单。否则，退回用户卡，输出钱，如果要求收据接着是收据。在要求用户输入的任一点，除了简单的确认，都提供了一个“取消”选项。如果选择了取消，退回用户卡，并终止他们和 ATM 的交互。

（14） 一个简单的数字手表包括一个显示小时和分钟的显示器，小时和分钟之间由一个闪烁的冒号隔开，还提供了两个按钮（A 和 B），能够使显示器更新。

(*a*) 为了给显示的小时数加 2，应该执行下列动作，其中按钮 B 给小时显示加 1：

按下 A；按下 B；按下 B；按下 A；按下 A。

画一个简单的状态图准确表示这个事件序列。

(*b*) 在上面的交互中，显示的小时可以增加任何需要的数量，并且每当需要时整个交互可以重复。重画状态图以并入这些泛化。

(*c*) 为了增加手表显示的分钟数，可以按两次按钮 A，接着重复按下按钮 B，每次给显示的分钟数加 1。为这个手表画一个完整的状态图，并入对显示的小时和分钟的更新。为你的状态图中的状态给出有意义的名字，并对任何标注为“按下 B”的转移加上适当的动作。

(*d*) 这个手表随后被增强，并入了闹钟，而下面的交互被提议作为设置闹钟时间的方法：

按下 A；按下 A；按下 B （重复）；按下 A；按下 B (重复) ；按下 A。

意思是用户快速地连续两次按下按钮 A，像鼠标的“双击”一样。解释这个建议将如何向数字手表的状态图中引入非确定性。说明你怎样通过向状态图引入额外的状态消除这种非确定性。

（15） 画一个状态图总结下面对 Java 线程的生命周期中可能出现的一些事件的描述中给出的信息。

当线程被创建时，它并不立即运行，而是停留在 *New Thread* 状态。当线程处于这个状态时，它只能被起动或停止。除了 *start* 或 *stop* 之外，调用任何方法都没有意义，并且会导致抛出异常。

start 方法引起为该线程分配系统资源，并调用线程的 *run* 方法。在此时，线程处于 *Running* 状态。

如果调用的线程的 *sleep* 或 *suspend* 方法，线程将变成不可运行的。*Sleep* 方法有一个参数指定线程休眠的时间长度；当过去这么多时间，线程又开始运行。如果调用了 *suspend* 方法，那么线程只有在调用 *resume* 方法时才能再次运行。

线程可以以两种方式死亡。当它的 *run* 方法正常退出时它自然死亡。在任何时候通过调用它的 *stop* 方法也可以杀死线程。