返回总目录

目　　录

第 2 章　UML 语言概述

统一建模语言（Unified Modeling Language，简称 UML）的应用领域很广泛，它可以用于商业建模（business modeling）、软件开发建模的各个阶段、也可以用于其他类型的系统。它是一种通用（general）的建模语言，具有创建系统的静态结构和动态行为等多种结构（construction）模型的能力。UML 语言本身并不复杂，也不很专业化，它具有可扩展性和通用性，适合为各种多变的系统建模。

本章主要介绍 UML 语言的概况，使大家了解 UML 的结构和基本元素。基本元素只给出简要的描述，更详细深入的讨论将在后继章节中叙述。因此，读者在本章的学习中只要知道与 UML 有关的概念即可，并不需要完全理解图例。

UML 由视图（views）、图（Diagrams）、模型元素（Model elements）和通用机制（general mechanism）等几个部分构成。

视图用来表示被建模系统的各个方面（从不同的目的出发建立，为系统建立多个模型，这些模型都反映同一个系统，且具有一致性）。视图由多个图（Diagrams）构成，它不是一个图片（graph），而是在某一个抽象层上，对系统的抽象表示。如果要为系统建立一个完整的模型图，只需定义一定数量的视图，每个视图表示系统的一个特殊的方面就可以了。另外，视图还把建模语言和系统开发时选择的方法或过程连接起来。

图由各种图片（graph）构成，用来描述一个视图的内容。UML 语言定义了 9 种不同的图的类型，把它们有机地结合起来就可以描述系统的所有视图。

模型元素代表面向对象中的类、对象、消息和关系等概念，是构成图的最基本的常用概念。一个模型元素可以用在多个不同的图中，无论怎样使用，它总是具有相同的含义和相同的符号表示。

通用机制用于表示其他信息，比如注释、模型元素的语义等。另外，它还提供扩展机制，使 UML 语言能够适应一个特殊的方法（或过程）、或扩充至一个组织或用户。

2.1　视　　图

给复杂的系统建模是一件困难和耗时的事情。从理想化的角度来说，整个系统像是一张画图，这张图画清晰而又直观地描述了系统的结构和功能，既易于理解又易于交流。但事实上，要画出这张图画几乎是不可能的，因为，一个简单的图画并不能完全反映出系统中需要的所有信息。

描述一个系统涉及到该系统的许多方面，比如：功能性方面（它包括静态结构和动态交互）、非功能性方面（定时需求、可靠性、展开性等）和组织管理方面（工作组、映射代码模块等）。

完整地描述系统，通常的做法是用一组视图反映系统的各个方面，每个视图代表完整系统描述中的一个抽象，显示这个系统中的一个特定的方面。每个视图由一组图构成，图中包含了强调系统中某一方面的信息。视图与视图之间有时会产生轻微的重叠，从而使得

一个图实际上可能是多个视图的一个组成部分。如果用不同的视图观察系统，每次只集中地观察系统的一个方面。视图中的图应该简单，易于交流，且与其他的图（图用图形符号表示，图符号代表系统中的模型元素）和视图有关联关系。

UML 中的视图包括：用例视图（Use-case view）、逻辑视图（Logical view）、组件视图（Component view）、并发视图（Concurrency View）、展开视图（Deployment View）等五种。能够使用的其他视图还有静态——动态视图、逻辑——物理视图、工作流程（workflow）等视图，但 UML 语言中并不使用这些视图，它们是 UML 语言的设计者意识中的视图，因此在未来的大多数 CASE 工具中有可能包含这些视图。

当用户选择一个 CASE 工具作图（diagram）的时候，一定要保证该工具能够容易地从一个视图导航（navigate）到另一个视图。另外，为了看清楚一个功能在图中是怎样工作的，该工具也应该具备方便地切换至用例视图或展开视图的长处，因为用例视图下可以看到该功能是怎样被外部用户描述的，展开视图下可以看到物理结构中该功能是怎样分布的（即确定在哪台计算机中得到该功能）。

2.1.1 用例视图

用例视图（Use-case view）用于描述系统应该具有的功能集。它是从系统的外部用户角度出发，对系统的抽象表示。

用例视图所描述的系统功能依靠于外部用户或另一个系统触发激活，为用户或另一个系统提供服务，实现用户或另一个系统与系统的交互。系统实现的最终目标是提供用例视图中描述的功能。

用例视图中可以包含若干个用例（use-case）。用例用来表示系统能够提供的功能（系统用法），一个用例是系统用法（功能请求）的一个通用描述。

用例视图是其他视图的核心和基础。其他视图的构造和发展依赖于用例视图中所描述的内容。因为系统的最终目标是提供用例视图中描述的功能，同时附带一些非功能性的性质，因此用例视图影响着所有其他的视图。

用例视图还可用于测试系统是否满足用户的需求和验证系统的有效性。

用例视图主要为用户、设计人员、开发人员和测试人员而设置。用例视图静态地描述系统功能，为了动态地观察系统功能，偶尔也用活动图（activity diagram）描述。

2.1.2 逻辑视图

用例视图只考虑系统应提供什么样的功能，对这些功能的内部运作情况不予考虑，为了揭示系统内部的设计和协作状况，要使用逻辑视图描述系统。

逻辑视图（Logical view）用来显示系统内部的功能是怎样设计的，它利用系统的静态结构和动态行为来刻画系统功能。静态结构描述类、对象和它们之间的关系等。动态行为主要描述对象之间的动态协作，当对象之间彼此发送消息给给定的函数时产生动态协作，一致性（persistence）和并发性（concurrency）等性质，以及接口和类的内部结构都要在逻辑视图中定义。

静态结构在类图和对象图中描述，动态建模用状态图、序列图、协作图和活动图描述。

2.1.3 组件视图

组件视图（Component view）用来显示代码组件的组织方式。它描述了实现模块（implementation module）和它们之间的依赖关系。

组件视图由组件图构成。组件是代码模块，不同类型的代码模块形成不同的组件，组件按照一定的结构和依赖关系呈现。组件的附加信息（比如，为组件分配资源）或其他管理信息（比如，进展工作的进展报告）也可以加入到组件视图中。组件视图主要供开发者使用。

2.1.4 并发视图

并发视图（Concurrency View）用来显示系统的并发工作状况。并发视图将系统划分为进程和处理机方式，通过划分引入并发机制，利用并发高效地使用资源、并行执行和处理异步事件。除了划分系统为并发执行的控制线程外，并发视图还必须处理通信和这些线程之间的同步问题。并发视图所描述的方面属于系统中的非功能性质方面。

并发视图供系统开发者和集成者（integrator）使用。它由动态图（状态图、序列图、协作图、活动图）和执行图（组件图、展开图）构成。

2.1.5 展开视图

展开视图（Deployment View）用来显示系统的物理架构，即系统的物理展开。比如，计算机和设备以及它们之间的联接方式。其中计算机和设备称为结点（node）。它由展开图表示。展开视图还包括一个映射，该映射显示在物理架构中组件是怎样展开的。比如，在每台独立的计算机上，哪一个程序或对象在运行。

展开视图提供给开发者、集成者和测试者。

2.2 图

图（diagram）由图片（graph）组成，图片是模型元素的符号化。把这些符号有机地组织起来形成的图表示了系统的一个特殊部分或某个方面。一个典型的系统模型应有多个各种类型的图。图是一个具体视图的组成部分，在画一个图时，就相当于把这个图分配给某个视图了。依据图本身的内容，有些图可能是多个视图的一部分。

UML 中包含用例图、类图、对象图、状态图、序列图、协作图、活动图、组件图、展开图共九种。本小节讨论九种图的基本概念。关于图的语法、含义、它们之间怎样交互等所有细节将在后继章节中叙述。使用这九种图就可以描述世界上任何复杂的事物，这就充分地显示了 UML 的多样性和灵活性。

2.2.1 用例图

用例图（use-case diagram）用于显示若干角色（actor）以及这些角色与系统提供的用例之间的连接关系，如图 2-1 所示。用例是系统提供的功能（即系统的具体用法）的描述。通常一个实际的用例采用普通的文字描述，作为用例符号的文档性质。当然，实际的用例

图也可以用活动图描述。用例图仅仅从角色（触发系统功能的用户等）使用系统的角度描述系统中的信息，也就是站在系统外部察看系统功能，它并不描述系统内部对该功能的具体操作方式。用例图定义的是系统的功能需求。关于用例图的图示方法、含义等更进一步的介绍放在第三章中。

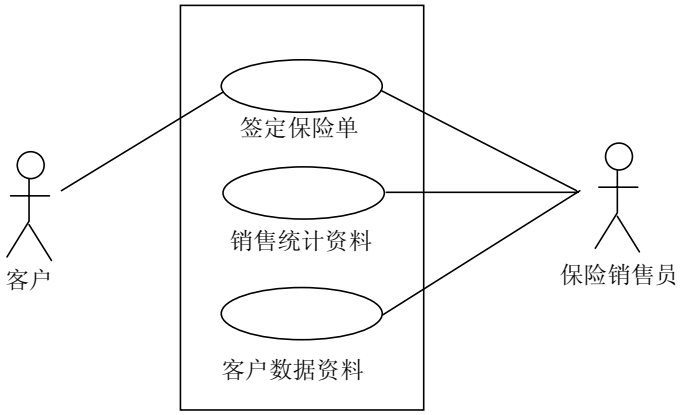


图 2.1　用例图示例

2.2.2　类图

类图（class diagram）用来表示系统中的类和类与类之间的关系，它是对系统静态结构的描述，如图 2-2 所示。

类用来表示系统中需要处理的事物。类与类之间有多种连接方式（关系），比如：关联（彼此间的连接）、依赖（一个类使用另一个类）、通用化（一个类是另一个类的特殊化）或打包（packaged）（多个类聚合成一个基本元素）。类与类之间的这些关系都体现在类图的内部结构之中，通过类的属性（attribute）和操作（operation）这些术语反映出来。在系统的生命周期中，类图所描述的静态结构在任何情况下都是有效的。

一个典型的系统中通常有若干个类图。一个类图不一定包含系统中所有的类，一个类还可以加到几个类图中。在第四章中，我们再详细讨论。

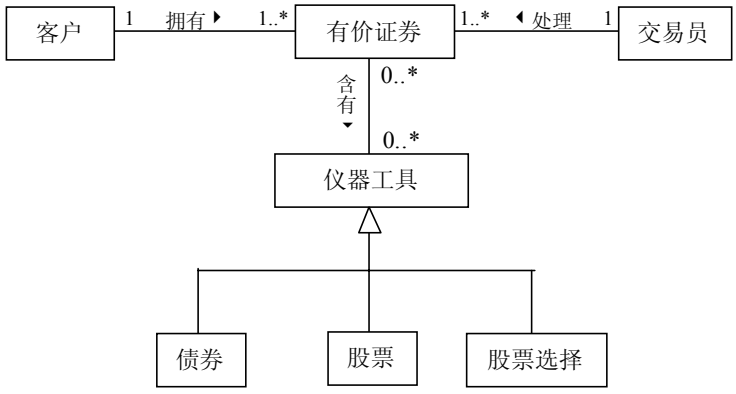


图 2-2　金融贸易类图示例

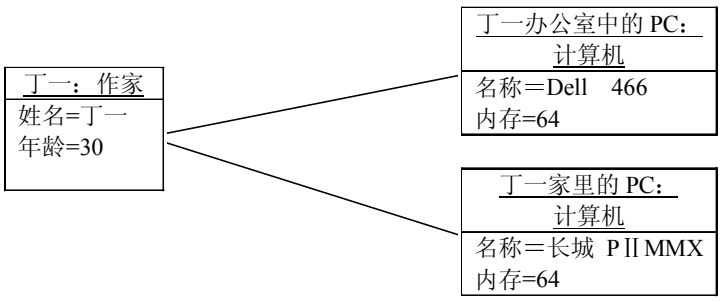
2.2.3 对象图

对象图是类图的变体。两者之间的差别在于对象图表示的是类的对象实例，而不是真实的类。对象图是类图的一个范例（example），它及时具体地反映了系统执行到某处时，系统的工作状况。

对象图中使用的图示符号与类图几乎完全相同，只不过对象图中的对象名加了下划线，而且类与类之间关系的所有实例也都画了出来，如图 2-3 所示。图 2-3（a）的类图抽象地显示各个类及它们之间的关系，图 2-3（b）的对象图则是图 2-3（a）类图的一个实例表示。

对象图没有类图重要，对象图通常用来示例一个复杂的类图，通过对象图反映真正的实例是什么，它们之间可能具有什么样的关系，帮助对类图的理解。对象图也可以用在协作图中作为其一个组成部分，用来反映一组对象之间的动态协作关系。

（a） 类图



（b） 对象图

图 2-3 对象图与类图示例

2.2.4 状态图

一般说来，状态图是对类所描述事物的补充说明，它显示了类的所有对象可能具有的状态，以及引起状态变化的事件，如图 2-4 所示。事件可以是给它发送消息的另一个对象或者某个任务执行完毕（比如，指定时间到）。状态的变化称作转移（transition）。一个转移可以有一个与之相连的动作（action），这个动作指明了状态转移时应该做些什么。

并不是所有的类都有相应的状态图。状态图仅用于具有下列特点的类：具有若干个确定的状态，类的行为在这些状态下会受到影响且被不同的状态改变。

另外，也可以为系统描绘整体状态图。关于状态图更进一步的讨论详见第五章和第八章。

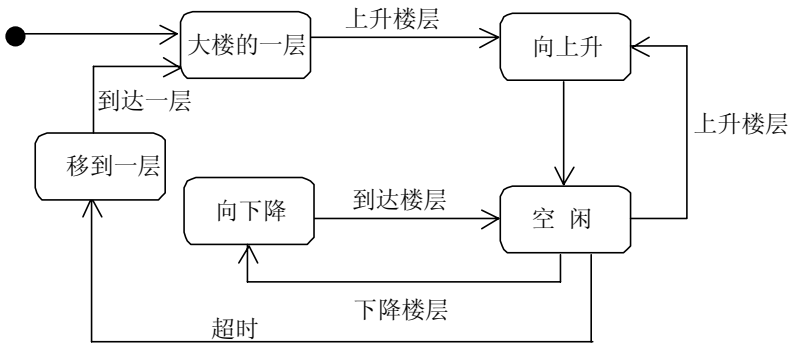


图 2.4　电梯的状态图示例

2.2.5 序列图

序列图用来反映若干个对象之间的动态协作关系，也就是随着时间的流逝，对象之间是如何交互的，如图 2-5 所示。序列图主要反映对象之间已发送消息的先后次序，说明对象之间的交互过程，以及系统执行过程中，在某一具体位置将会有什么事件发生。

序列图由若干个对象组成，每个对象用一个垂直的虚线表示（线上方是对象名），每个对象的正下方有一个矩形条，它与垂直的虚线相叠，矩形条表示该对象随时间流逝的过程（从上至下），对象之间传递的消息用消息箭头表示，它们位于表示对象的垂直线条之间。时间说明和其他的注释作为脚本放在图的边缘。对序列图的讨论详见第五章和第八章。

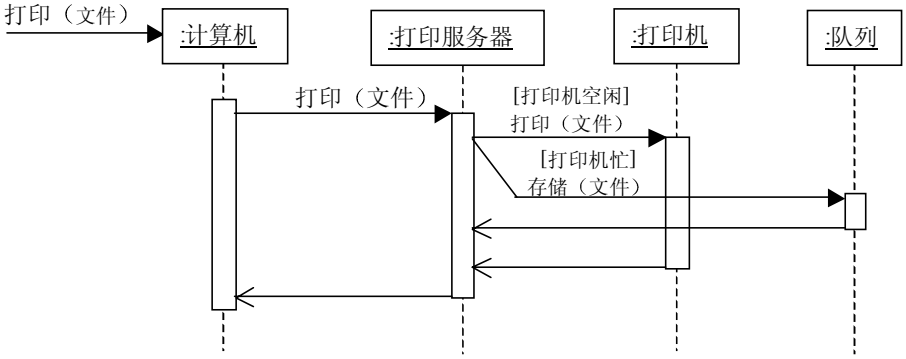


图 2.5　序列图示例

2.2.6　协作图

协作图和序列图的作用一样，反映的也是动态协作。除了显示消息变化（称为交互）外，协作图还显示了对象和它们之间的关系（称为上下文有关）。由于协作图或序列图都反映对象之间的交互，所以建模者可以任意选择一种反映对象间的协作。如果需要强调时间和序列，最好选择序列图；如果需要强调上下文相关，最好选择协作图。

协作图与对象图的画法一样，图中含有若干个对象及它们之间的关系（使用对象图或类图中的符号），对象之间流动的消息用消息箭头表示，箭头中间用标签标识消息被发送的序号、条件、迭代（iteration）方式、返回值等等。通过识别消息标签的语法，开发者可以看出对象间的协作，也可以跟踪执行流程和消息的变化情况。

协作图中也能包含活动对象，多个活动对象可以并发执行。如图 2-6 所示。第五章和第八章详细讨论协作图。

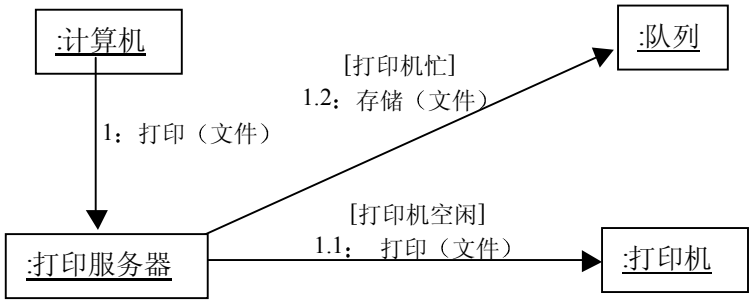


图 2.6　协作图示例

2.2.7　活动图

活动图（activity diagram）反映一个连续的活动流，如图 2-7 所示。相对于描述活动流（比如，用例或交互）来说，活动图更常用于描述某个操作执行时的活动状况。

活动图由各种动作状态（action state）构成，每个动作状态包含可执行动作的规范说明。当某个动作执行完毕，该动作的状态就会随着改变。这样，动作状态的控制就从一个状态流向另一个与之相连的状态。

活动图中还可以显示决策、条件、动作状态的并行执行、消息（被动作发送或接收）的规范说明等内容。活动图在第五章中详述。

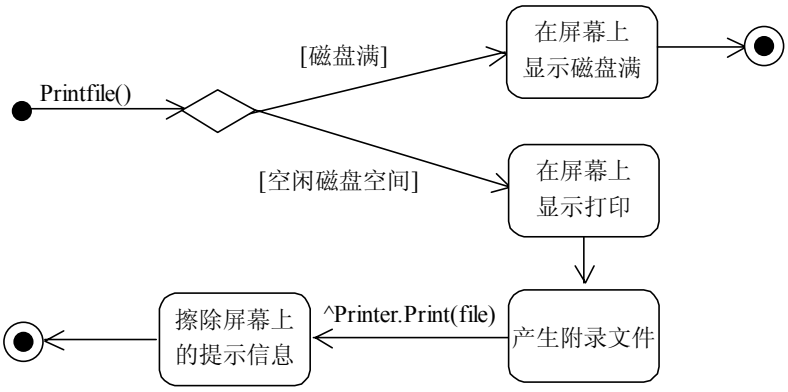


图 2.7　活动图示例

2.2.8　组件图

组件图（component diagram）用来反映代码的物理结构。

代码的物理结构用代码组件表示。组件可以是源代码、二进制文件或可执行文件组件。

组件包含了逻辑类或逻辑类的实现信息，因此逻辑视图与组件视图之间存在着映射关系。组件之间也存在依赖关系，利用这种依赖关系可以方便地很容易地分析一个组件的变化会给其他的组件带来怎样的影响。

组件可以与公开的任何接口（比如，OLE／COM 接口）一起显示，也可以把它们组

合起来形成一个包（package），在组件图中显示这种组合包。实际编程工作中经常使用组件图（如图 2-8 所示）。第六章中将进一步详述组件图。

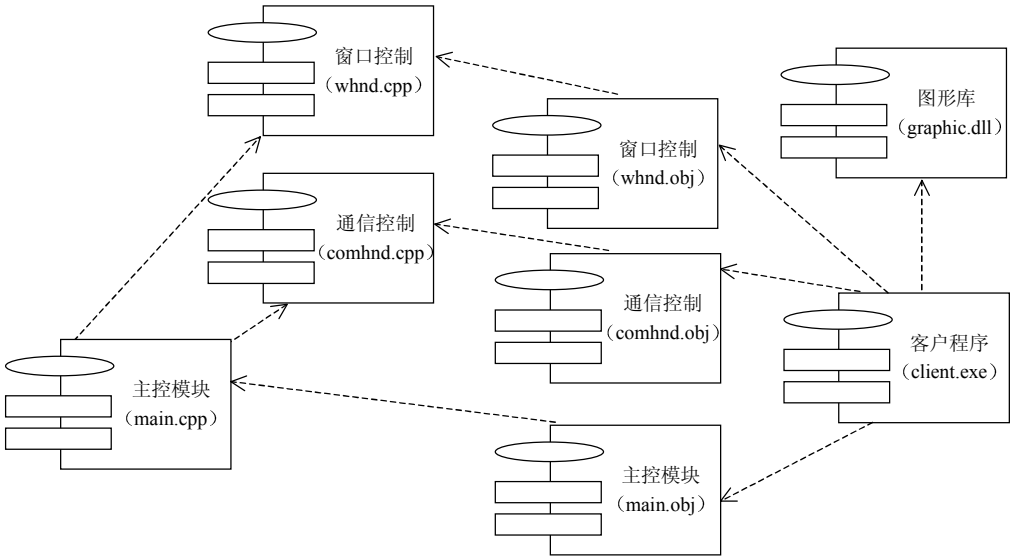


图 2.8　组件图示例

2.2.9　展开图

展开图（deployment diagram）用来显示系统中软件和硬件的物理架构。通常展开图中显示实际的计算机和设备（用结点表示），以及各个结点之间的关系（还可以显示关系的类型）。每个结点内部显示的可执行的组件和对象清晰地反映出哪个软件运行在哪个结点上。组件之间的依赖关系也可以显示在展开图中。

正如前面所陈述，展开图用来表示展开视图，描述系统的实际物理结构。用例视图是对系统应具有的功能的描述，它们二者看上去差别很大，似乎没有什么联系。然而，如果对系统的模型定义明确，那么从物理架构的结点出发，找到它含有的组件，再通过组件到达它实现的类，再到达类的对象参与的交互，直至最终到达一个用例也是可能的。从整体来说，系统的不同视图给系统的描述应当是一致的，（如图 2-9 所示）。关于展开图的讨论放在第六章中详述。

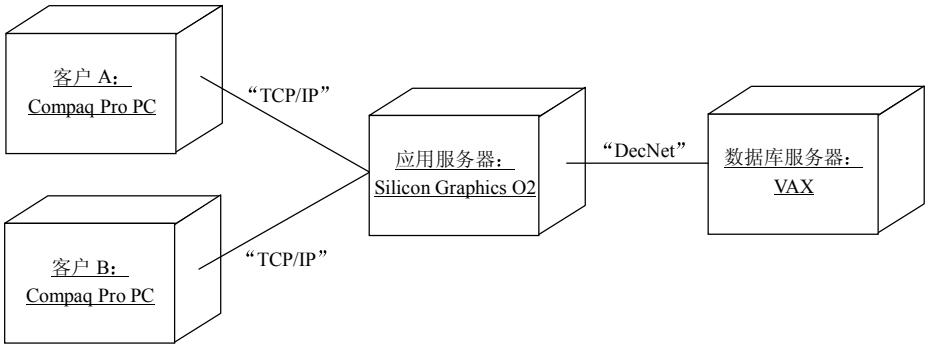


图 2.9　展开图示例

2.3 模 型 元 素

可以在图中使用的概念统称为模型元素。模型元素用语义、元素的正式定义或确定的语句所代表的准确含义来定义。模型元素在图中用其相应的视图元素（符号）表示。利用视图元素可以把图形象直观地表示出来。一个元素（符号）可以存在于多个不同类型的图中，但是具体以怎样的方式出现在哪种类型的图中要符合（依据）一定的规则。

图 2-10 给出了类、对象、状态、结点、包（package）和组件等模型元素的符号图例。

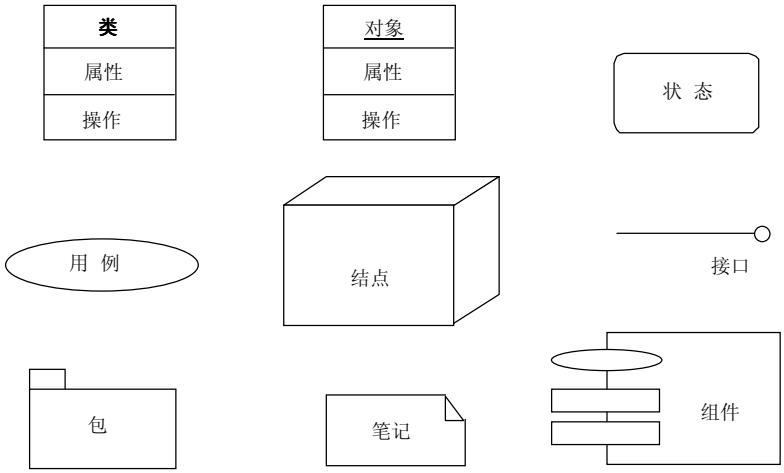


图 2.10　一些通用的模型元素符号示例

模型元素与模型元素之间的连接关系也是模型元素，常见的关系有关联（association）、通用化（generalization）、依赖（dependency）和聚合（aggregation），其中聚合是关联的一种特殊形式。这些关系的图示符号如图 2-11 所示。

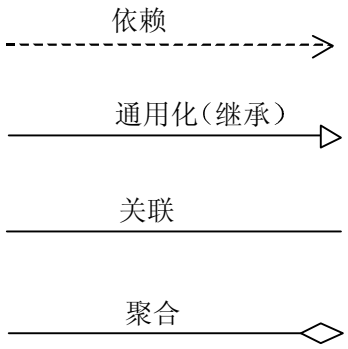


图 2.11　关系的图示符号示例

除了上述的模型元素外，模型元素还包括消息、动作和版类（stereotype）。所有模型元素的意义和允许的用法在后续章节中解释，它们的语义采用非正式的可行方式描述，而不采用 UML 语言参考手册中的正式定义。

2.4 通 用 机 制

UML 语言利用通用机制为图附加一些信息，这些信息通常无法用基本的模型元素表示。常用的通用机制有修饰（adornment）、笔记（note）和规格说明（specification）等。

2.4.1 修饰

在图的模型元素上添加修饰为模型元素附加一定的语义。这样，建模者就可以方便地把类型与实例区别开。

当某个元素代表一个类型时，它的名字被显示成黑体字；当用这个元素代表其对应类型的实例时，它的名字下面加下划线，同时还要指明实例的名字和类型的名字。

比如，类用长方形表示，其名字用黑体字书写（比如，**计算机**）。如果类的名字带有下划线，它则代表该类的一个对象（比如，丁一的计算机）。对结点的修饰方式也是一样的，结点的符号既可以是用黑体字表示的类型（比如，**打印机**），也可以是结点类型的一个实例（丁一的 HP 打印机）。其他的修饰有对各种关系的规范说明。比如重数（multiplicity），重数是一个数值或一个范围，它指明涉及到关系的类型的实例个数。修饰紧靠着模型元素书写。

2.4.2 笔记

无论建模语言怎样扩展，它不可能应用于描述任何事物。为了在模型中添加一些额外的模型元素无法表示的信息，UML 语言提供了笔记能力。笔记可以放在任何图的任意位置，并且可以含有各种各样的信息。信息的类型是字符串（UML 语言不能解释）。

如果某个元素需要一些解释或说明信息，那么就可以为该元素添加笔记。通常用虚线把含有信息的笔记与图中的一些元素联系起来，如图 2-12 所示。

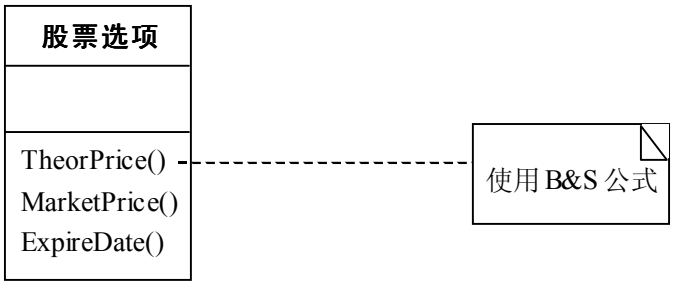


图 2.12 笔记图示

笔记中可以包含建模者的注释或问题，用以提示建模者，防止日后出现不清楚该元素的含义等情况。笔记中也可以包含版类（版类用于描述笔记的类型），版类在下一节的扩展机制中详细叙述。

2.4.3 规格说明

模型元素含有一些性质，这些性质以数值方式体现。一个性质用一个名字和一个值表示，又称作加标签值（tagged value）。加标签值用整数或字符串等类型详细说明。UML 中

有许多预定义的性质，比如：文档（documentation）、响应（responsibility）、持续性（persistence）和并发性（concurrency）。

性质一般作为模型元素实例的附加规格说明，比如，用一些文字逐条列举类的响应和能力。这种规范说明方式是非正式的，并且也不会直接显示在图中，但是在某些 CASE 工具中，通过双击模型元素，就可以打开含有该元素所有性质的规格说明窗口，通过该窗口就可以方便地读取信息了。

2.5 扩 展 机 制

UML 语言具有扩展性，因此也适用于描述某个具体的方法、组织或用户。这里我们介绍三种扩展机制：版类（stereotype）、加标签值（tagged value）和约束（constrains）。这三种机制的更详细的讨论在第七章中进行。

2.5.1 版类

版类扩展机制是指在已有的模型元素基础上建立一种新的模型元素。版类与现有的元素相差不多，只不过比现有的元素多一些特别的语义罢了。版类与产生该版类的原始元素的使用场所是一样的。版类可以建立在所有的元素类型上，比如：类、结点、组件、笔记、关系（关联、通用化和依赖）。UML 语言中已经预定义了一些版类，这些预定义的版类可以直接使用，从而免去了再定义新版类的麻烦，使得 UML 语言用起来比较简单。

版类的表示方法是在元素名称旁边添加一个版类的名字。版类的名字用字符串（用双尖角括号括起来）表示，如图 2-13 所示。版类也可以用一个图形表示（比如，图标）。

具体的版类元素的图示方法有三种：第一种在元素名称之上写版类名，这是一般的表示法；第二种是在元素名称旁画出版类的图标（图形化表示）；第三种是把元素名称和版类图标合在一起。图 2-13 图示了这三种表示法。

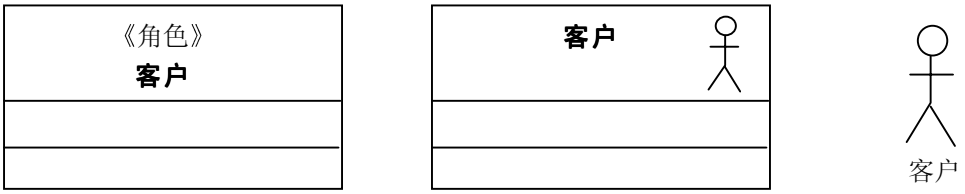


图 2-13　版类的图示方法

当一个元素与版类连接在一起后，该元素称为“指定版类的元素类型”。比如，与版类《window》相连的类就称为“window 版类的类”，这意味着该类是 window 类型的了。

当用户定义版类的时候，与之相关的类也要定义。比如，定义《window》版类时，必须定义“window 版类的类”。图 2-13 中的“客户”是具有版类《角色》的类，由于版类《角色》添加了特别的语义给“客户”类，所以该类代表的是系统的外部角色（用户）。

版类是非常好的扩展机制，它的存在避免了 UML 语言过于复杂化，同时也使 UML 语言能够适应各种需求，很多需求的新模型元素已做成了 UML 语言的基础原型（prototype），用户可以利用它添加新的语义后定义新的模型元素。

2.5.2 加标签值

在 2.4.3 节中已讨论过元素有很多性质，性质用名字和值一对信息表示。性质也称为加标签值。UML 语言中已经预定义了一定数量的性质，用户还可以为元素定义一些附加信息，即定义性质。任何一种类型的信息都可以定义为元素的性质，比如：具体的方法信息、建模进展状况的管理信息、其他工具使用的信息、用户需要给元素附加的其他各类的信息。图 2-14 表示的是仪器类的性质，其中抽象（abstract）（详细含义见第四章）是预定义的性质，作者和状态是用户定义的加标签值。

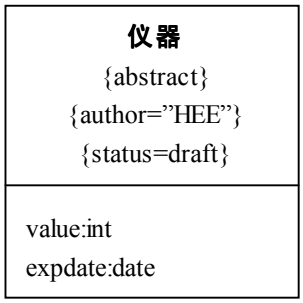


图 2.14　仪器类的性质示例

2.5.3 约束

约束是对元素的限制。通过约束限定元素的用法或元素的语义。如果在几个图中都要使用某个约束，可以在工具中声明该约束，当然，也可以在图中边定义边使用。

图 2-15 显示的是老年人（类）与一般人（类）之间的关联关系。显然，并不是所有的人都是老年人，为了表示只有 6 0 岁以上的人才能加入老年人（类），我们定义了一个约束条件：年龄属性大于 6 0 岁的人（person.age > 60）。有了这个条件，哪个人属于这种关联关系中也就自然清楚了。反过来说，假如没有约束条件，这个图就很难解释清楚。在最坏情况下，它可能会导致系统实现上的错误。

在上述例子中，约束被直接定义和应用在了需要使用的图上。当然，也可以用名字加规格说明的方法定义约束，比如，“老年人”和“person.age > 60”。UML 语言中预定义了一部分约束。这些约束的具体内容在第七章中讨论。

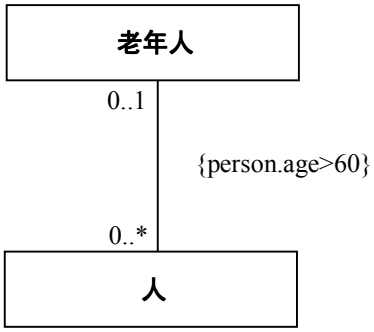


图 2.15　约束示例

2.6 用 UML 建模

用 UML 语言建造系统模型的时候，并不是只建一个模型。在系统开发的每个阶段都要建造不同的模型，建造这些模型的目的也是不同的。需求分析阶段建造的模型用来捕获系统的需求、描绘与真实世界相应的基本类和协作关系。设计阶段的模型是分析模型的扩充，为实现阶段作指导性的、技术上的解决方案。实现阶段的模型是真正的源代码（source code），编译后的源代码就变成了程序。最后是展开模型，它在物理架构上解释系统是如何展开的。

虽然这些模型各不相同，但通常情况下，后期的模型都由前期的模型扩展而来。因此，每个阶段建造的模型都要保存下来，以便出错时返回重做或重新扩展最初的分析模型。如图 2-16 所示。

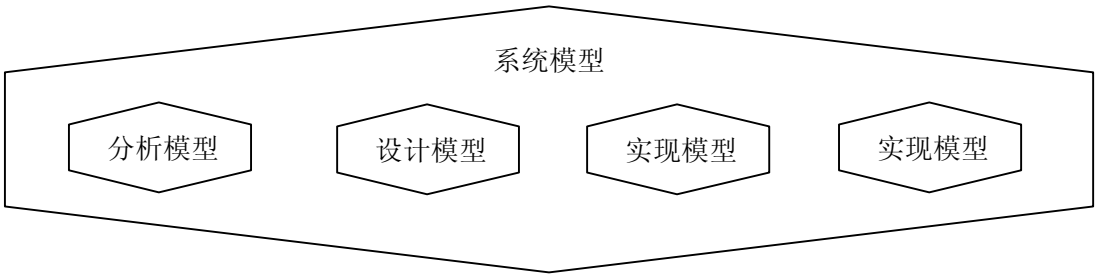


图 2.16　用多个模型描述的系统

UML 语言具有阶段独立性，也就是说，同样的通用语言和同样的图可以用在不同的阶段为不同的事情建模。这使得建模者能把更多的精力放在考虑模型的结构和适用范围上。注意，建模语言只能用于建造模型，不能用于保证系统的质量（第四章介绍）。

若使用 UML 语言建模，建模工作一定要依照某个方法或过程进行。因为这个方法或过程列出了应进行哪些不同的步骤，及这些步骤怎样实现的大纲。建模的过程一般被分为以下几个连续的重复迭代阶段：需求分析阶段、设计阶段、实现阶段和展开阶段。与实际的建模工作相比，这是一个简单的建模过程。通常情况下，一组人聚在一起提出问题和讨论目标，就已经开始建模了。他们一起讨论并写出一个非正式的会议记要，记录可能要建造的模型的构想和应有怎样的变化。

记录会议内容的工具也很不正规，通常在笔记本和白板上书写，这种会议一直要持续到参与讨论的人感觉这些基本的模型具有一定的可行性了，才会进入下一阶段。这时形成的模型称为早期的假说。把假说用某一 CASE 工具描述，假说模型就组织起来了，同时按照建模语言的语法规则构建一个真实的图也是可行的了。再到下一阶段时，前期模型会被更详细地描述，这个阶段主要完成的工作是，细化解决问题的方案和文档，提取更多的解决问题需要的信息。这个工作可能需要几经反复才能最后完成。通过这个阶段的工作，假说也逐渐变成一个可使用的模型了。

再接下来的步骤是集成（integration）和验证（verification）模型。集成就是把同一系统中的各种图或模型结合起来，通过验证工作保证图与图之间数据的一致性。通过验证工作，保证模型能够正确地解决问题。如图 2-17 所示。

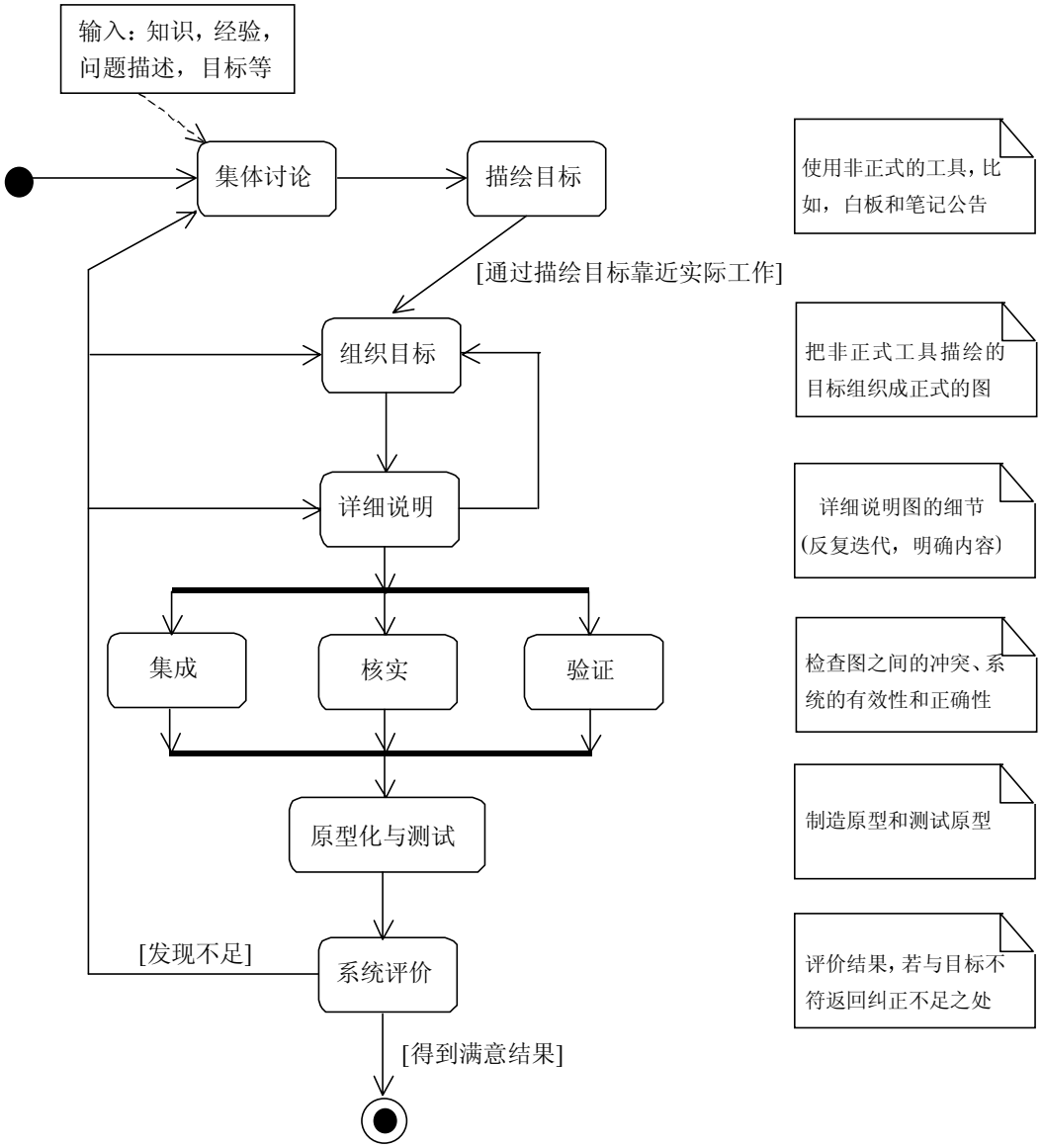


图 2.17　实际建模工作的大致流程

最后，在实际解决问题的时候，模型被实现为各种原型（prototype）。生成原型时，要对生成的原型进行评价，以便发现可能潜在的错误、遗漏的功能和开发代价过高等不足之处。如果发现了上述不足之处，那么开发人员还要返回到前期的各个阶段步骤，排除这些问题。如果问题很严重，开发者或许最终要返到刚开始的集体讨论，描绘草图的阶段，重新建模。当然，如果问题很小，开发者只需改变模型的一部分组织和规格说明。注意，把图原型化的工作，一定要在把多个图结合成原型结构之后。

原型只是一个很粗浅的东西，构建原型仅仅是为了对其进行评价，发现其中的不足之处，而对原型进一步地开发过程才算得上真正的系统开发过程。

2.7 工具的支持

使用建模语言需要相应的工具支持。即使人工在白板上画好了模型的草图，建模者也需要使用工具。因为模型中很多图的维护、同步和一致性检查等工作，人工做起来几乎是不可能的。

自从用于产生程序的第一个可视化软件问世以来，建模工具（又叫 CASE 工具）一直不很成熟。许多 CASE 工具几乎和画图工具一样，仅提供了建模语言和很少的一致性检查，增加了一些方法的知识。经过人们不断地改进，今天的 CASE 工具正在接近图的原始视觉效果，比如，Rational Rose 工具，就是一种比较现代的建模工具。但是还有一些工具仍然比较粗糙，比如一般软件中很好用的“剪切”和“粘帖”功能，在这些工具中尚未实现。另外，每种工具都有属于自己的建模语言，或至少有自己的语言定义也限制了这些工具的发展。随着统一建模语言 UML 的发布，工具制造者现在可能会花较多的时间来提高工具质量，减少定义新的方法和语言所花费的时间。

一个现代的 CASE 工具应提供下述的功能：

- 画图（draw diagrams）

CASE 工具中必须提供方便作图和为图着色的功能，也必须具有智能，能够理解图的目的，知道简单的语义和规则。这样的特点带来的方便是，当建模者不适当地或错误地使用模型元素时，工具能自动告警或禁止其操作。

- 积累（repository）

CASE 工具中必须提供普通的积累功能，以便系统能够把收集到的模型信息存储下来。如果在某个图中改变了某个类的名称，那么这种变化必须及时地反射到使用该类的所有其他图中。

- 导航（navigation）

CASE 工具应该支持易于在模型元素之间导航的功能。也就是，使建模者能够容易地从一个图到另一个图地跟踪模型元素或扩充对模型元素的描述。

- 多用户支持

CASE 工具提供该功能使多个用户可以在一个模型上工作，但被此之间没有干扰。

- 产生代码（generate code）

一个高级的 CASE 工具一定要有产生代码的能力。该功能可以把模型中的所有信息翻译成代码框架，把该框架作为实现阶段的基础。

- 逆转（reverse）

一个高级的 CASE 工具一定要有阅读现成代码并依代码产生模型的能力，即模型可由代码生成。它与产生代码是互逆的两个过程。对开发者来说，他可以用建模工具或编程二种方法建模。

- 集成（integrate）

CASE 工具一定要能与其他工具集成，即与开发环境（比如，编辑器、编译器和调试器）和企业工具（比如，配置管理和版本控制系统）等的集成。

- 覆盖模型的所有抽象层

CASE 工具应该能够容易地从对系统的最上层的抽象描述向下导航至最低的代码层。这样，若需要获得类中一个具体操作的代码，只要在图中单击这个操作的名字即可。

- 模型互换

模型或来自某个模型的个别的图应该能够从一个工具输出，然后再输入到另一个工具。就像 JAVA 代码可在一个工具中产生，而后用在另一个工具中一样。模型互换功能也应该支持用明确定义的语言描述的模型之间的互换（输出 / 输入）。

2.7.1 绘图支持

CASE 工具应使绘图工作简单而有趣。一个高级的绘图工具能够充当 CASE 工具经历了一段很长的时间，那是因为 CASE 工具不仅必须提供优秀的选择、放置、连接和定义图中元素的机制，而且还要帮助建模者着色，形成一张正确的图。

CASE 工具还应该有理解模型元素语义的能力。这种能力能够在错误地使用模型元素时发出警告，或者提示一个具体的操作与其他操作之间可能存在不一致问题。比如，在一个模型中，若修改某个图后，将会引起该图与其他图的冲突，这时系统就会自动告警，提示建模者的修改可能出现错误。

CASE 工具也应提供图的版面设计功能。比如，允许建模者重新排列模型元素，而代表消息的线条由工具自动地重新排列，使它们彼此不会交叉。在很多 CAD 系统中，这个功能实现得很好，建模工具的制造者可以从中借鉴一些经验。

2.7.2 模型积累

CASE 工具的模型积累就是提供一个数据库。用数据库保存模型中元素的所有信息，而不考虑这些信息来自哪个图。这个积累应该含有整个模型的基本信息，这些基本信息可以通过若干个图看到。如图 2-18 所示。

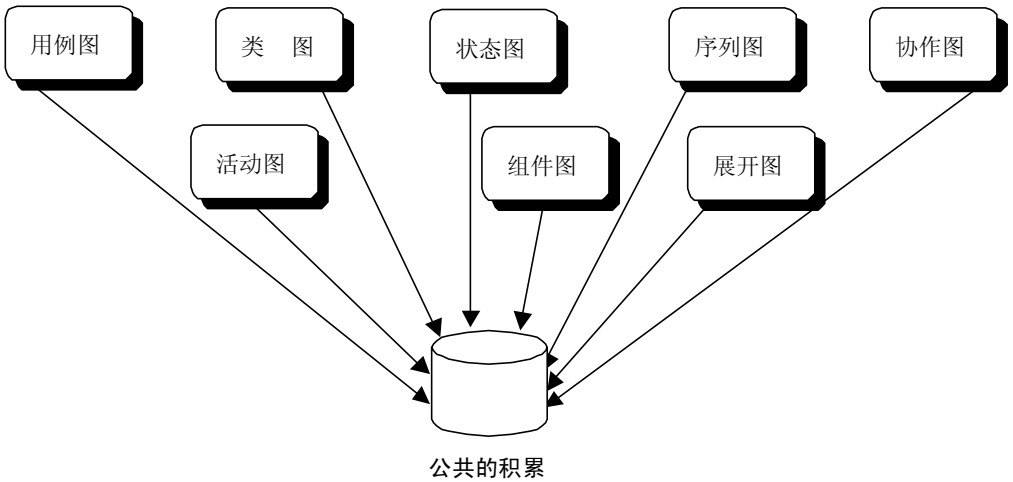


图 2.18　积累包含所有图的信息

在积累的帮助下，CASE 工具的检索不一致、鉴定、报告、重用元素或图的工作才能执行。

检索不一致的含义是，如果元素与其在其他图中的用法不一致，CASE 工具则必须警

告或阻止这种用法。如果建模者打算删除某图中的元素，而该元素又被其他图使用，同样也会产生警告开发者的信息。如果开发者一定要删除这个元素，那么所有引用了该元素的图中的该元素也将被删除。这样，开发者就一定要返回去修改引用了该元素的图，使之有效。

鉴定是指 CASE 工具可以使用积累鉴定模型。或是指出尚未详细说明的部分，或是跟踪模型，发现可能的错误或不合适的解决方法。

报告指的是 CASE 工具能自动产生所有模型元素的完全的和扩展的文档。比如，类或图的报告。这个报告与所有数据的术语分类很相似。

重用元素或图指的是利用积累支持重用功能，以便某个工程的建模解决方案或部分方案能容易地用在另一个工程中。UML 模型中的组件直接与源代码相连，因此组件或其源代码都能在不同的工程中重用。

2.7.3 导航

当把几个视图和图合起来共同描述一个系统的时候，能够方便地在视图和图之间导航是很重要的。CASE 工具一定要支持导航功能，达到方便地浏览不同的图和搜索模型元素的目的。

在 CASE 工具中表示的模型元素本身应该具有超链功能。右击元素应能弹出一个快捷菜单，上面显示普通的操作并给出可能的导航，比如，查看含有该元素的图或获取关于该元素的详细信息。图中有些部分应该能展开和折叠，就像 Windows 操作系统中的资源管理器一样，展开功能方便于观察构成包的内容（由哪些图构成），折叠功能方便于观察周围的包。

另一种控制复杂图的方式是定义过滤器，用过滤器把图中一些开发者感兴趣的方面独立表示出来或高亮显示。比如，只高亮显示图中的关系，或高亮显示类。具体高亮显示什么类型由建模者控制。有了过滤器，建模者就可以在某一时段只研究那些重要的高显部分。

2.7.4 多用户支持

CASE 工具应能让多个用户在同一个模型上协同工作。也就是说，彼此之间没有干扰。一般地，如果一个用户正在某个图上工作，那么该用户应该锁定这个图，不让其他用户同时改变这个图。更进一步地说，CASE 工具要具有识别对积累中共享元素的任何改变的能力，但是这种改变是否适当是否有效还要靠用户决定。

2.7.5 代码生成

现代 CASE 工具支持代码生成，这样建模阶段存储的有价值的部分工作就可以直接用到实现阶段，减少重复劳动。一般地，CASE 工具产生用编程语言书写的代码框架（code skeleton）和把模型转换成编程语言书写的代码（从理论上讲，包括把动态模型的一部分翻译成方法的主体部分）。而实际上，CASE 工具产生的代码通常是静态信息，比如类的声明（包括属性和方法的说明）。真正代码中的方法的主体部分（动态信息）是空缺的，它需要程序员亲自编制实现。

代码生成工作可被用户参数化，也就是用户给出指令，在指令中说明需要产生的代码有怎样的特点，由 CASE 工具依此指令生成。

任何类型的编程语言都能在 CASE 工具中使用。一般常用的是面向对象编程语言，比如 C++或 JAVA。但是用 SQL 或 IDL 这样的语言书写的代码也能够生成。由于不同的编程语言使用不同的代码生成器，因此 CASE 工具应有插接各种代码生成器的能力。

假如依据某个模型生成了代码之后，用户才开始编写方法的主体部分。如果在用户编程完成后，又对该模型的某个地方做了修改，那么再用修改过的模型生成代码会不会把用户的编码工作丢失呢？答案是不会丢失。因为代码中，自动生成的代码和人工编制的代码分别用不同的标记显示，当重新生成代码时，代码生成器不会触及人工编码的那一部分。

2.7.6 工程逆转

工程逆转与代码生成几乎是对立的二个功能。CASE 工具阅读和分析代码，为的是用图显示代码的结构。通常只有静态结构（比如类图）能用代码构建，动态信息是不能从代码中提取的。

工程逆转常应用于下列二种代码，一是不知其结构的代码，这个代码或许是买来的，或许是人工编制的；二是由代码生成功能产生的代码。对人工编制的未知结构的代码进行逆转时，逆转结果或是被提高或是被降低，不符合结构化的代码也能被发现出来。对于购买的类库（代码），工程逆转用来得到该类库的结构图，使该类库中的类能被直接用在建模过程时所作的图中。

代码生成和工程逆转合在一起称作往返工程（round-trip engineering）。使用往返工程，开发者能够在建模和实现之间反复操作，开发过程才真正变成了能够迭代反复的过程。

2.7.7 集成

建模工具与系统开发时需要使用的其他工具形成一个整体，就是集成。建模工具只是集成环境中的一个部分，但是对其他工具而言，它是一个真正的自然的“集线器”（Hub），如图 2-19 所示。可以集成的工具有开发环境、配置和版本控制、文档工具、测试工具、GUI 构造器、需求说明工具、工程管理和过程支持工具等七种。

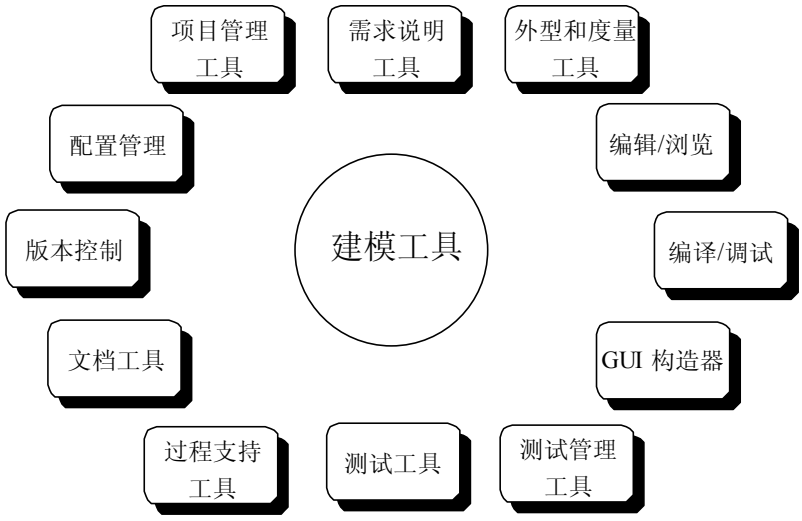


图 2.19　以建模工具为核心集成的工具箱

开发环境包括代码的编辑、编译和调试。有了这些集成的工具，就可以从图的模型元素直接进入开发环境，反过来也一样。同样，也能在编辑代码模块时，决定该模块在系统的逻辑和物理模型中的位置。

配置和版本控制工具用来控制系统的各种不同的配置，以及系统和个别元素的不同版本。版本控制包含对模型和代码两方面的控制。

文档工具能自动从模型积累中产生文档。它还能从已知的信息中产生数据资料。

测试工具主要用于管理测试过程，收集和维护测试报告。测试保证系统的有效性（系统好不好？）和正确性（系统对不对？）。模型本身所含有的供测试过程用的大量信息也应该传给测试工具。

GUI 构造器工具存储和管理图形化的用户接口，它应能自动地为模型中的类生成 GUI 表单（form）。比如，通过获取类的属性，自动生成表单中与之相应类型的域。

需求说明工具用于对于系统中非功能性方面需求的描述，比如定时需求、可靠性和展开性等（UML 中通过用例捕获系统的功能性需求）。

设计工程管理工具的目的在于帮助工程管理者制定时间安排表、资源分配计划表和跟踪工程进度表。由于建造模型占据工程的大部分时间，因此使工程管理者能够容易地检查建模工作的进度是很有意义的。

过程支持工具用于支持某个具体方法或过程的用法。在 CASE 工具中集成过程支持工具可能不是最佳的选择，因为，从本质上讲，建造的模型应是对任何方法或过程实质性部分的体现，而过程支持工具提供的用法不一定适合该方法或过程。

最后需要说明的是，并不是每个工程都要使用上述所有工具，目前市售的建模工具也没有完全集成上述所有工具。

2.7.8 模型互换

模型互换的含义是在一个工具中产生的模型能够应用于另一个工具中。执行模型互换的先决条件是把存储模型的格式标准化。为 UML 定义这种格式的工作正在进展中，尽管这种格式还不是 OMG 标准的组成部分。

当工具之间的模型互换变成现实的时候，开发环境中的建模工具会更加简单，同时这种模型也可能被集成到更高级的工具之中。

用户将是模型互换功能的最大受益者，因为他们可以不再紧紧依靠某个 CASE 工具制造商。如果他们喜欢另一种 CASE 工具，只要直接将他们的模型移植过来即可，有了统一的标准格式（版本），其他相互独立的工具（比如，文档、报告和生成数据库等工具）也能容易地开发出来。

2.7.9 小结

UML 语言用若干个视图（view）构造系统模型。每个视图代表系统的一个方面。视图用图描述，图又用模型元素的符号表示。图中包含的模型元素可以有类、对象、结点、组件、关系（关联、通用性、依赖）等，这些模型元素有具体的含义并且用图形符号表示。

UML 图包括：类图、对象图、用例图、状态图、序列图、协作图、活动图、组件图和展开图。这些图的用途和绘制这些图时应遵守的规则在后续章节中叙述。

UML 的通用机制用来给图添加信息。通用机制有：修饰（与元素名一起放置）、笔记（存储任何类型的信息）和规格说明。UML 的扩展机制有：加标签值、约束、版类（在已有模型元素的基础上定义新的模型元素）。

一个系统由多个不同类型的模型描述，每种模型都有不同的目的。分析模型描述功能需求和为真实世界的类构建模型。设计模型把分析结果转换成技术解决方案。实现模型使用面向对象的编程语言将系统编码实现。展开模型把建好的代码放置在物理架构中。上述的几个建模工作是重复迭代操作的过程，并且必须按一定的顺序进行。

在实际工程中，用户使用 UML 时需要借助工具。现代 CASE 工具应具有下列能力：绘图、存储积累信息、导航、产生报告和文档、代码生成、识别代码产生模型、与其他开发工具集成。