第3章

初识UML

想要理解和使用 UML，需要掌握 UML 的概念模型。UML 的概念模型主要包括基本构造块、运用于构造块的通用机制和用于组织 UML 视图的架构。UML 的概念模型支撑起了 UML 语法的整体架构和分析思想，对于普通建模用户而言，从 UML 概念模型入手能够快速掌握 UML 建模的基本思想，从而能够读懂并建立一些基本模型；在有了丰富的使用 UML 的经验后，就可以在这些概念模型之上理解 UML 的结构，从而使用更深层次的语言特征开展建模工作。

3.1 UML 构造块

构造块(building block)指的是 UML 的基本建模元素，是 UML 中用于表达的语言元素，是来自现实世界中的概念的抽象描述方法。构造块包括事物(thing)、关系(relationship)和图(diagram)三个方面的内容。事物是对模型中关键元素的抽象体现；关系是事物和事物间联系的方式；图是相关的事物及其关系的聚合表现。

3.1.1 事物

在 UML 中，事物是构成模型图的主要构造块，它们代表了一些面向对象的基本概念。事物被分为以下四种类型。

1. 结构事物

结构事物(structural thing)通常作为 UML 模型的静态部分，用于描述概念元素或物理元素。结构事物总称为类元(classifier)。常见的结构事物有类、接口、用例、协作、组件、节点等。

类(class)是对具有相同属性、相同操作、相同关系和相同语义的一组对象的描述。在 UML 图中使用矩形表示类，核心内容包括类名、属性、方法(操作)。在 UML 中的类的图示如图 3-1 所示。

接口(interface)是一组操作的集合，这些操作包括类或组件的动作，描述了元素的外部可见行为。接口仅仅定义操作的数量和特征，但不提供具体的实现方法。接口可以被类所继承，继承了某接口的类必须提供该接口所有操作的实现。接口一般不需要属性。在 UML 中接口的声明也使用矩形描述，在接口名上方使用构造型<< interface >>与类做区分，

如图 3-2 所示。内容包括接口名称和操作(一般没有属性)。

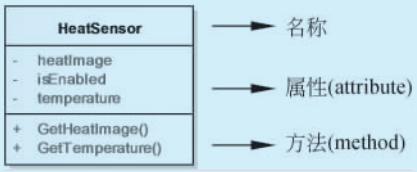

图 3-1 类

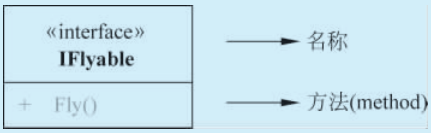

图 3-2 接口

协作(collaboration)定义了一个交互,它是在为实现某个目标而共同工作、相互配合的多个元素之间的交互动作。协作具有结构、行为和维度,一个类或对象可以参与多个协作。在 UML 图中把协作画成虚线椭圆,仅包含名称,如图 3-3 所示。这种表示法允许从外部把一个协作视为一个整体,但我们通常不使用这种表示方式,而是对协作内部更感兴趣。因此,我们往往会放大一个协作,将其内部细节引导到一些图中——特别是类图(用来表示协作的结构部分)和交互图(用来表示协作的行为部分)。

用例(use case)描述了一组动作序列,这些动作序列将作为服务由特定的参与者触发或执行,在过程中产生有价值、可观察的结果,结果可反馈给参与者或作为其他用例的参数。在 UML 图中用例表示为实线椭圆,仅包含名称,如图 3-4 所示。

组件(component)是系统中封装好的模块化部件,仅将外部接口暴露出来,内部实现被隐藏。组件可以由部件和部件之间的连接表示,其中部件也可以包含更小的组件。接口相同的部件可以互相替换。在 UML 图中将组件表示成矩形框,左边框上附着两个小矩形,框内写组件名,如图 3-5 所示。

节点(node)是在软件部署时需要的物理元素,其本质为一种计算机资源。一组组件可以存在于一个节点内,也可以从一个节点迁移到另一个节点。在 UML 图中节点用一个立方体表示,仅包含名称,如图 3-6 所示。

图 3-3 协作

图 3-4 用例

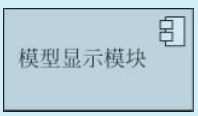

图 3-5 组件

图 3-6 节点

2. 行为事物

行为事物(behavioral thing)也称为动作事物,是 UML 模型的动态部分,用于描述 UML 模型中的动态元素,主要为静态元素之间产生的时间和空间上的行为动作,类似于句子中动词的作用。常见的行为事物有交互、状态机、活动等。

交互(interaction)描述一种行为,它产生于协作完成一个任务的多个元素之间。交互包含消息、状态和连接。在 UML 图中消息表示为实箭头,源自消息发出者,指向接收者,箭头上方写操作名。

状态机(state machine)定义了对象或行为在生命周期内的状态转移规则。状态机中包

含状态、转移、条件(事件)以及活动。UML 图中的状态机表示为圆角矩形，包含状态名，如图 3-7 所示。

活动(activity)描述了一个操作执行时的过程信息。一个活动包含在操作执行过程中每一个步骤(动作)之间的先后序列关系。UML 图中的活动也表示为圆角矩形，如图 3-8 所示，它和状态机图例的区分依靠语义。

图 3-7　状态机

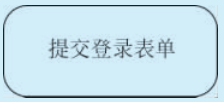

图 3-8　活动

3. 分组事物

分组事物(grouping thing)又称组织事物，是 UML 模型的组织部分，是用来组织系统设计的事物。主要的分组事物是包(如图 3-9 所示)，另外，其他基于包的扩展事物(例如子系统、层等)也可作为分组事物。

4. 注释事物

注释事物(annotation thing)又称辅助事物，是 UML 模型的解释部分。这些注释事物用来描述、说明和标注模型的任何元素，简言之就是对 UML 中元素的注释。最主要的注释事物就是注解(note)，是依附于一个元素或一组元素之上对其进行约束或解释的简单符号，内容为对元素的进一步解释文本。这些解释文本在 UML 图中可以附加到任何模型的任意位置上，用虚线连接到被解释的元素，如图 3-10 所示。当不需要显示注释时可以隐藏，也可以以链接形式放到外部文本中(如果注释很长)。几乎所有的 UML 图形元素都可以用注解来说明。

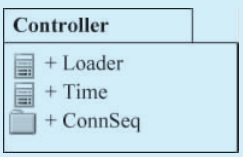

图 3-9　包

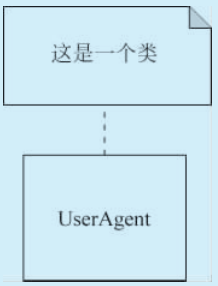

图 3-10　注解

3.1.2　关系

关系是模型元素之间具体化的语义连接，负责联系 UML 的各类事物，构造出结构良好的 UML 模型。在 UML 中有四种主要的关系。

- 关联(association)：描述不同类元的实例之间的连接。它是一种结构化的关系，指

一种对象和另一种对象之间存在联系，即“从一个对象可以访问另一个对象”。更详细地，我们说两个对象之间互相可以访问，那么这是一个双向关联，否则称为单向关联。关联中还有一种特殊情况，称作聚合关系，聚合表示两个类元的实例具有整体和部分的关系，表示整体的模型元素可能是多个表示部分的模型元素的聚合。例如，一个汽车与四个轮胎会构成关联关系，而这种关联关系同时又是聚合关系。

- 依赖(dependency)：描述一对模型元素之间的内在联系(语义关系)，若一个元素的某些特性随某一个独立元素的特性的改变而改变，则这个元素不是独立的，它依赖于上文所给的那个独立元素。例如，水管依赖热水器，对它们运送的水进行加热。
- 泛化(generalization)：类似于面向对象方法中的继承关系，是特殊到一般的一种归纳和分类关系。泛化可以添加约束条件，说明该泛化关系的使用方法或扩充方法，称为受限泛化。
- 实现(realization)：描述规格说明和其实现的元素之间的连接的一种关系。其中规格说明定义了行为的说明，真正的实现由后一个模型元素来完成。实现关系一般用于两种情况下：接口和实现接口的类和组件之间与用例和实现它们的协作之间。

这四种关系是 UML 模型中包含的最基本的关系，它们可以扩展和变形。例如关联可以扩展为聚合、组合两种特殊的关系，依赖则有导入、包含、扩展等多种关系。这些关系的具体内容会在后续章节中进行详细讲解。

3.1.3 图

当用户选择了模型所需的事物和关系之后，就需要将模型展示出来；这种展示就是通过 UML 的图来实现。图是一组模型元素的图形表示，是模型的展示效果。多数的 UML 图是由通过路径连接的图形构成的。信息主要通过拓扑结构表示，而不依赖于符号的大小或者位置(有一些例外，如顺序图)。

根据 UML 图的基本功能和作用，可以将其划分为两大类：结构图(structure diagrams)和行为图(behaviour diagrams)。结构图捕获事物与事物之间的静态关系，用来描述系统的静态结构模型；行为图则捕获事物的交互过程如何产生系统的行为，用来描述系统的动态行为模型。

在 UML 1.4 中，共包含用例图、类图、对象图、活动图、状态图、顺序图、协作图、组件图、部署图九种。另外，尽管 UML 1.4 使用包图说明规范的组织结构，但是没有对包图进行明确定义。UML 1.4 中图的结构如图 3-11 所示。

在升级到 UML 2 规范后，随着软件工程技术的变迁，对图有不同的分类方法和解释方式。UML 2 规范包含 14 种图：类图、对象图、组合结构图、组件图、部署图、包图、外廓图、用例图、活动图、状态机图、顺序图、通信图、时序图、交互概览图。UML 2 中图的结构如图 3-12 所示。

UML 2 中的大部分图与 UML 1.4 是大致相同的(可能在表示法上略有区别)，另一部分的图是将 UML 1 中的某些图的功能进行了细分，还增加了几种新图。下面通过表 3-1 对比 UML 1.4 与 UML 2 中图的区别。

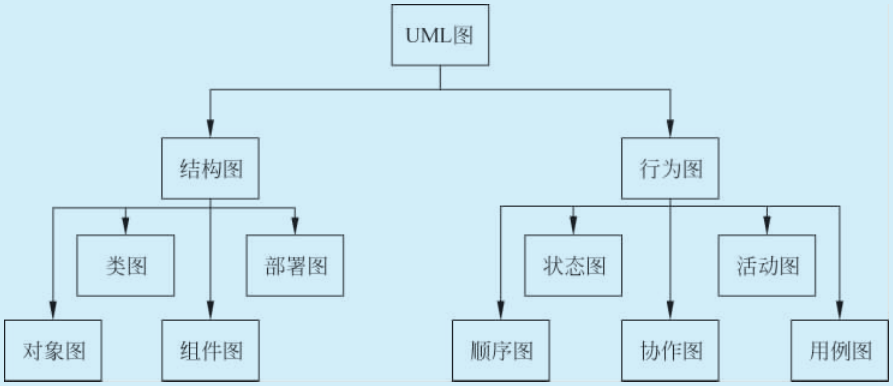

图 3-11　UML 1.4 中的图

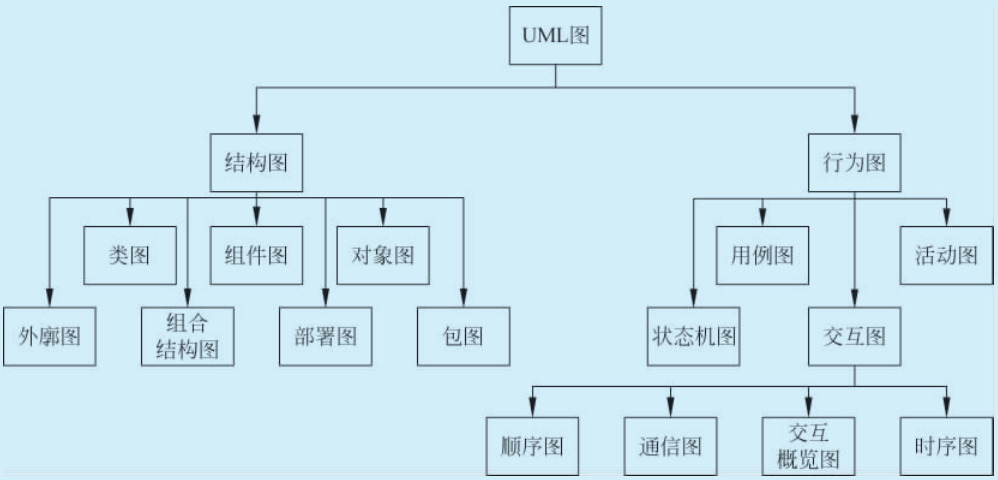

图 3-12　UML 2 中的图

表 3-1　UML 1.4 与 UML 2 不同图的对比

UML 1.4	UML 2	对 比 说 明
	包图	尽管 UML 1.4 使用包图说明规范的组织结构，但是没有对包图进行明确定义
状态图	状态机图	只是名称不同，技术上完全相同
活动图	活动图	UML 2 的活动图独立于状态机存在
	组合结构图	显示结构化类元或协作的内部结构，和普通类图之间没有严格界限
	交互图	UML 2 中的交互图是顺序图、通信图、交互概览图和时序图的统称，与活动图密切相关
协作图	通信图	UML 2 中多用更加精确的通信图来代替协作图的大部分功能；UML 2 中协作图作为一种组合结构图存在
	交互概览图	活动图的变体，合并了序列图片段和控制流构造
	时序图	UML 2 中新增的时序图是一种特殊的序列图形式，显式地表示了生命线上的状态变化和标度时间

3.2 UML 通用机制

UML 提供了四种通用机制，它们被一直地应用到模型中，描述了达到面向对象建模目的的 4 种策略，并在 UML 的不同语境下被反复运用，使得 UML 更简单并易于使用。这四种机制分别是：规格说明(specifications)、修饰(adornments)、通用划分(common divisions)和扩展机制(extensibility mechanisms)。

3.2.1 规格说明

UML 不仅仅是一个图形化的语言。恰恰相反，在每个图形符号后面都有一段描述用来说明构建模块的语法和语义。例如，在一个类的符号中暗示了一种规格说明：它提供类所有的属性、操作等信息的全面描述；虽然，有时为了表现得更直观，类图示可能只显示这些描述的一小部分。而且，从另外一个视角来看这个类，可能会有完全不同的部分，但仍然与类的基本规范保持一致。

UML 的规格说明用来对系统的细节进行描述，在增加模型的规格说明时可以确定系统的更多性质，细化对系统的描述。通过规格说明，我们可以利用 UML 构建出一个可增量的模型，即首先分析确定 UML 图形，然后不断对该元素添加规格说明来完善其语义。

3.2.2 修饰

UML 中大多数的元素都有一个唯一的和直接的图形符号，用来给元素的最重要的方面提供一个可视的表达方式。例如，类的图示是有意地设计为易描绘的图形，并且类符号也揭示出类的最重要的方面，即它的名称、属性和操作。

修饰是对规格说明的文字的或图形的表示。我们已经知道，类的规格说明可能包含其他细节，诸如它是不是抽象类，它的属性和操作的可见性，这些细节中的大多数都可以通过图示或文本修饰在类的基本矩形框符号中表达。例如，这里有一个类，我们可以通过不同的修饰来标示出它是一个抽象类，拥有两个公有性的操作，一个保护性的操作和一个私有性的操作。

在 UML 中的每个元素符号都以一个基本的符号开始，在其上添加一些具有独特性的修饰。

3.2.3 通用划分

在面向对象系统建模中，通常有几种划分方法，其中最常见的两种划分是类型-实例与接口-实现。

1. 类型-实例

类型-实例(type-instance)是通用描述与某个特定元素的对应。通用描述符称为类型，特定元素称为实例，一个类型可以有多个实例。在使用过程中可以类比面向对象语言中的类和对象的关系，事实上，类和对象就是一种典型的类型-实例划分。实例的表示方法为类

型的描述符的名称下加下画线，后附冒号和类型。如Undergraduate ： Student。

2. 接口-实现

接口是一个系统或对象的行为规范，这种规范预先告知使用者或外部的其他对象这个系统或对象的某项能力及其提供的服务。通过接口，使用者可以启动该系统或对象的某个行为。实现是接口的具体行为，它负责执行接口的全部语义，是具体的服务兑现过程。例如，在借钱时我们打过一张欠条，那么这张欠条就是一种还钱的约定。但是欠条只代表一个“我会还钱”的约定，而不代表真正的还了钱。把钱真正交到借主手上才是将承诺兑现的过程。那么接口就相当于欠条，而还钱是欠条所对应的实现。

许多 UML 的构造块都有像接口-实现这样的二分法。例如，接口与实现它的类或组件、用例与实现它的协作、操作与实现它的方法等。

3.2.4 UML 扩展机制

为了扩充在某些细节方面的描述能力，UML 允许建模者在不改变整体语言风格的基础上定义一些通用性的扩展。UML 所提供的扩展很可能无法满足出现的所有要求，但是它以一种易于实现的简单方式容纳了建模者需要对 UML 所做的大部分剪裁。

UML 中的扩展机制包括构造型（stereotype）、标记值（tagged value）和约束（constraint）三种。在使用扩展机制的时候需要注意，有些扩展违反了 UML 的标准形式，使用它们也会造成逻辑上的互相影响。因此在使用扩展机制之前，建模者应当仔细权衡利弊，特别是当现有机制能够合理工作时，是否还需要应用扩展机制。

1. 构造型

构造型是将一个已有的元素模型进行修改或精化，创造出一种新的模型元素。构造型的信息内容和形式与已存在的基本模型元素相同，但拥有不同的含义与用法。

例如，业务建模领域的建模人员经常希望将业务对象和业务过程作为特殊的元素进行建模，它们在特定的环境中拥有不同于其他元素的用法。然而它们实际上可以被看作是特殊的类——同样拥有属性与操作，但是在使用上有着特殊的约束。

构造型在元素的特性描述中定义。每个构造型都从一个基本的模型元素派生而来。该构造型的所有元素都具有基本模型元素的特性。构造型的表示方法为一个双尖括号内附构造型名称，一般放在已有的基本模型元素符号上方。UML 中预定义了一些构造型供建模者使用，用户也可以根据自己的需要自行定义。例如，我们已经知道，接口实际上是<< interface>>构造型的类元素，如图 3-13 所示。

图 3-13　构造型

2. 标记值

标记值是关于模型元素本身的一个属性的定义，即一个元属性的定义。标记所定义的是用户模型中元素的特性而非运行时对象的特性。标记定义被构造型所拥有。

简单来说，当一个元素应用某种构造型时，该元素就获得了该构造型中所定义的所有标记。对每一个标记，建模者可以指定一个标记值。一般情况下，标记名、符号和值被写在注解中与模型元素连接在一起，如图 3-14 所示。

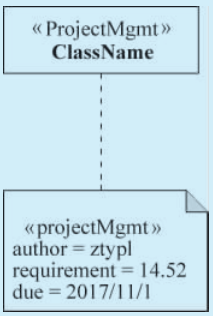

图 3-14 标记值

标记可以用来存储元素的任意信息，它是一个名称-值组合，表现为形如“property＝value”的字符串形式。在定义构造型时，建模者定义标记名来表示想要记录的一些特性；在将构造型应用给元素时，建模者需要给标记名指定标记值来存储这个元素的特性信息。例如，标记名可以是 author，表示这个标记用来存储此元素的作者姓名，而标记值则根据实际情况来填写，如 James Rumbaugh。标记值对于存储项目管理信息尤其有用，它可以用来记录开发者的信息、代码信息、日志、代码模板、代码生成说明等。

此外，标记值还提供了一种将与实现相关的附加信息与元素联系起来的方式。例如，代码生成器需要有关代码种类的附加信息以从模型中生成代码，我们可以利用标记来告诉代码生成器使用哪种实现方式。标记也可以为其他类型的插件工具所使用，如项目计划生成器和报表书写器。

3. 约束

约束是使用某种文本语言中的陈述句表达的语义条件或者限制。通常约束可以附加在任何一个或一组模型元素上，它表达了附加在元素上的额外语义信息。

每个约束包括一个约束体与一种解释语言。这里的解释语言可以是自然语言，也可以是形式化语言。如果是自然语言，则约束本身是不能自动强制遵守的。UML 提供了约束语言 OCL，但也可以使用其他形式语言。

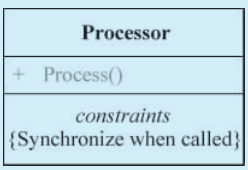

图 3-15 约束

某些常用的约束有名称，可以避免每次使用时都写出完整的复杂语句。例如，xor 就是异或约束的名称，其具体语义将在本书第 6 章讲解。

约束使用大括号（{ }）中的文本串表示，可以应用于大部分 UML 元素。图 3-15 就表示了一个类的约束。

3.3 “4＋1”架构

RUP“4＋1”架构方法采用用例驱动，在软件生命周期的各个阶段对软件进行建模，从不同视角对系统进行解读，从而形成统一软件过程架构描述。本节将主要介绍在建模过程中常用的“4＋1”架构。

3.3.1 “4+1”架构的概念和组成

“4+1”视图模型是由 Philippe Kruchten 于 1995 年在 *IEEE Software* 的一篇名叫 The 4+1 View Model of Architecture 的论文中提出的。在这个视图模型中，软件开发者从五个不同视角描述软件体系结构的一组视图模型。它们包括逻辑视图、开发视图、进程视图、物理视图和场景视图。每个视图只反映系统的某一部分，五个视图结合起来才可以描述整个系统的结构。

逻辑视图(logic view)将系统功能进行分解，它负责反映出系统内部是如何组织和协作来实现功能的。逻辑视图中包含从用户服务中提取出的对系统功能的抽象、分解和分析，通过揭示类、对象、类与对象之间的静态关系，以及对象之间如何交互的动态行为来展示各个对象如何共同实现系统的功能。逻辑视图主要对应着 UML 的类图。

开发视图(development view)主要用来描述软件的各个模块的组织方式，包括源程序、程序包、支持软件、第三方库等。开发视图面向开发人员，主要考虑软件的编程时需求，例如，模块的编写是否容易，是否可以重用，哪些成熟的框架可以应用等。对应到 UML 中，由于其描述了静态的软件组织结构，一般由有着相似功能的组件图(组件与子系统)表达。

进程视图(process view)主要描述系统的运行特性，侧重系统的性能和稳定性，关心系统的并发性、分布性、集成性的好坏，主要关注进程、线程、对象，并发、同步、通信等运行时概念。同时它为逻辑视图中类的具体操作指定进程或线程，并且对运行时单元之间的交互加以规划。进程视图主要面向系统集成人员，便于对系统进行性能测试。在 UML 中运行时分析一般采用顺序图、协作图、状态机图和协作图来完成。

物理视图(physical view)主要描述硬件配置，强调系统的安装、配置、通信、拓扑结构等问题。在考虑性能和可靠性的基础上，它将软件系统映射到指定的硬件设备上。物理视图应保证不同的硬件环境给软件的性能带来的影响最小，或者对于某种已知的物理配置而言让性能和稳定性达到最高。物理视图是综合考虑软件系统和安装、运行环境的视图。UML 中的部署图基本可以实现以上物理视图涉及的部分。

场景视图(scenarios)从项目需求入手，将四个视图结合为一个整体。可以描述一个特定视图内的构件关系，也可以描述不同视图间的构件关系。四个视图的元素需要协同工作以实现场景视图中给出的用例，它是距离用户需要最近的视图，也是软件开发中的重要驱动要素(用例驱动)。它实际上不包含新的内容(这也是“4+1”的由来)，只做四个视图的整合工作。但它是所有视图的核心，所谓用例驱动，就是指系统应当通过分析用例来决定应该提供哪些功能，所以它既是设计的核心，又是最终测试和检验的基准。UML 中的场景视图主要指用例图。

五个视图之间的关系如图 3-16 所示。

3.3.2 “4+1”架构要解决的问题

面对复杂的问题情境，要实现用户指定的功能、开发出满足用户需求的软件并非易事。开发过程中软件架构师是否能对各种各样的用户需求进行捕获，是否能准确地寻找出需求之间可能出现的矛盾，并且分析哪些需求是容易实现的、不易实现的、不能实现的，从而确保

重要的需求被满足。

正是因为软件设计是逻辑性极强的一种实践，是对人类智慧的高度考验，所以我们不能仅靠灵感来作为每个架构设计的策略。为了保证软件产品的功能需求，满足各种约束条件，并且在开发和使用时保障质量属性，我们在实践过程中需要依靠系统方法的指导。

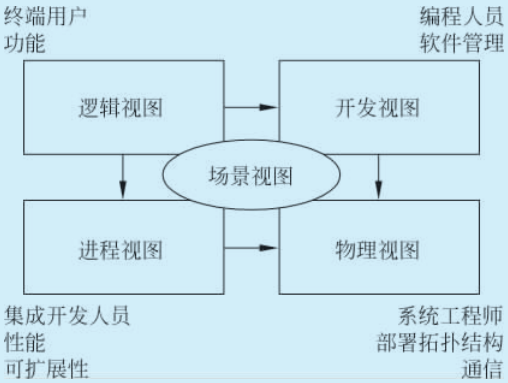

图 3-16 “4＋1”视图

从工程上简化一个问题，一种首要的思路就是分而治之。通常使用的分而治之策略有分层法、模块法等。其中，对于模块化而言，对于每个模块实行不同的较为单一的操作，透明化模块内部的信息，是一种重要的方法论。“4＋1”视图方法是一种架构设计的多重视图方法，属于一种特殊的模块法。上面已经介绍了各个视图的“单一”功能分划，以下将较为详细地介绍“4＋1”视图在开发实践中的使用方法。

3.3.3 运用“4＋1”视图方法进行软件架构设计

在软件工程的长期实践过程中，许多从业者总结出了一些“4＋1”架构的实践应用方法。其中，统一软件开发过程(Rational Unified Process，RUP)是一种成熟的、体系化、可定制的实践方法论。关于 RUP 的内容，我们在第 14 章中将对其概念和具体过程进行详细描述。本章就一般的软件开发过程，对“4＋1”视图的使用方法加以简单介绍。

一个软件项目和传统的工程项目的首要问题是一致的，那就是“做什么”“做出来的东西交给谁用”“谁付钱”。在软件术语中，这三个问题的回答被称为需求、用户和投资方。而这三者的关系比较显然：投资方希望为用户提供方便并从中谋取一些利益，缺少了投资方项目无法运行；用户实际使用这个软件产品，实现一些具体的目的，没有用户也就意味着软件产品不会被使用；需求代表着用户需要实现的目的，没有开发的软件产品往往混乱而无用。

一般来说，一个项目不会缺少投资方和实际的用户，所以三个首要问题中开发者最关心的就是用户的需求。如果没有需求，整个项目没有进行下去的目标和驱动力。所以在“4＋1”的五种视图中最先被使用的一定是场景视图。

场景视图是根据用户的需求可以直接产生和描述的，所以它是与需求关系最紧密的视图，可以在项目第一步获取需求之后立刻被使用。同样因为它代表顶层的软件产品目标，所以软件工程过程中一直通过分析各个场景来寻找功能和非功能点、检验系统是否满足要求。这些功能点和非功能点是实现部分的领航标，而根据场景进行测试是确认系统满足功能和

非功能点的重要方式。

当输出了各种场景视图之后，可以进一步使用逻辑视图来细化场景。这一步的细化包括几个方面：

（1）找到场景中的所有关键交互；

（2）使用软件术语描述出交互逻辑，注意一些场景可能是基于事件的；

（3）设计一些更下层的元素，这些元素的合理组合可以最终实现这个场景。

如果说场景视图是架构设计师与用户的通用交流语言，那么逻辑视图就是架构设计师和项目实际开发人员的通用交流语言，只是此时的表现层次仍然比较高。

逻辑视图是一个低于场景、高于详细设计的视图。这一点表现在逻辑视图仍然是静态的、注重问题分划、关注用户使用流程的。逻辑视图更多地在尝试使用编程术语描述问题，而不是解决问题。随后继续细分得到的开发视图、进程视图和物理视图则开始关注问题的具体实现。

进程视图、开发视图和物理视图不太容易分出先后顺序。虽然在“4＋1”视图中它们是不同的模块，然而它们的内容却是紧密相关的。开发视图关注各种程序包的使用，进程视图关注运行时概念，物理视图关注程序和运行库、系统软件对物理机器的要求和配合方式。有一些开发经验的读者，可能立刻就意识到它们之间密不可分的关系，例如，运行库存在着线程支持、线程安全问题；一些物理硬件支持并发而另外一些可能不支持；程序包的静态依赖关系在运行时会成为对象、进程和线程等。所以这三个视图需要合理地并用，负责每个视图的开发小组需要经常交流以确保三个视图间的内容一致。

对绝大多数面向对象软件开发过程来说，上述“4＋1”视图软件架构设计方法都是适用的。理解了本节的内容后，对读者对后续章节的理解有很大的帮助，并且后面将反复提及本节中使用的名词。

视频讲解

3.4 UML 建模工具

所谓“工欲善其事，必先利其器”，有了好的建模方法就需要有好的建模工具提供支持。经过多年的发展，目前已经出现了很多 UML 建模工具。本节主要介绍几个常用的 UML 建模工具。

1. Enterprise Architect

Enterprise Architect 是 Sparx Systems 公司的旗舰产品（见图 3-17）。它为用户提供一个高性能、直观的工作界面，联合 UML 2.0 最新规范，为台式机工作人员、开发和应用团队打造先进的软件建模方案。Enterprise Architect 的基础构建于 UML 2.0 规范之上，不仅如此，使用 UML Profile 还可以扩展建模范围，与此同时，模型验证将确保其完整性。利用 EA，设计人员可以充分利用所有 UML 2 中的图表的功能。EA 具备源代码的前向和反向工程能力，支持多种通用语言，包括 C++、C＃、Java、Delphi、VB. Net、Visual Basic 和 PHP，也可从源代码中获取完整框架。

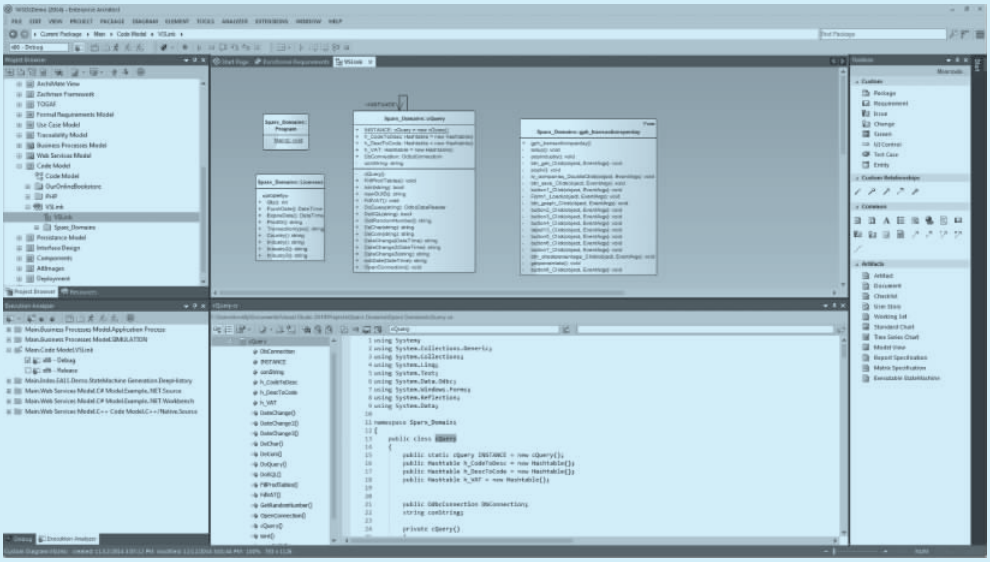

图 3-17 Enterprise Architect 界面

2. Rational Rose

视频讲解

Rational Rose 是由 Rational 公司研发的一种面向对象的可视化建模工具(见图 3-18)。Rose 为开发许多软件应用程序(包括 Ada、ANSI C++、C++、CORBA、Java、Java EE、Visual C++ 和 Visual Basic 等)提供了一系列的模型驱动功能。Rational Rose 可以满足绝大多数的建模环境的需求,是国际知名的建模工具。Rose 有很强的校验功能,可以方便地检查出模型中的许多逻辑错误,还支持多种语言的双向工程,可以自动维护 C++、Java、VB、PB、Oracle 等语言和系统的代码。由于 UML 是在 Rational 公司诞生的,这样的渊源使得 Rational Rose 力挫当前市场上很多基于 UML 可视化建模的工具。Rational Rose 自推出以来就受到了业界的瞩目,并一直引领着可视化建模工具的发展,是比较经典的 UML 建模工具。Rational Rose 的最新一次版本发布于 2003 年,因此不支持 UML 2 规范。后来 Rational 公司被 IBM 收购,又推出了新的 Rational 系列产品用于建模。

3. Rational Software Architect

视频讲解

IBM Rational Software Architect 是 IBM 在 2003 年 2 月并购 Rational 以来,首次发布的 Rational 产品(见图 3-19)。RSA 全面升级了之前的 Rose 工具,可以对系统进行建模、设计并维护架构、测试以及管理项目生命周期等操作。RSA 是 Rose 的升级替代品,因此支持使用 UML 来确保软件开发项目中的众多相关者不断沟通,并使用定义的规范来启动开发,并且支持最新的 UML 2 规范。RSA 支持 Java、C++、C#、EJB、WSDL、XSD、CORBA 和 SQL 等语言的正向工程和 Java、C++ 和. NET 的逆向工程。

4. StarUML

视频讲解

StarUML 是由 MKLab 开发的一款开源 UML 工具(见图 3-20)。该工具曾被遗弃过一

段时间，直到 2014 年发布了重新编写的 2.0.0 版本。该项目的目标是取代较大型的商业应用，如 Rational Rose。StarUML 目前支持大多数在 UML 2.0 中指定的图类型。目前暂时缺少时序图和交互概览图。

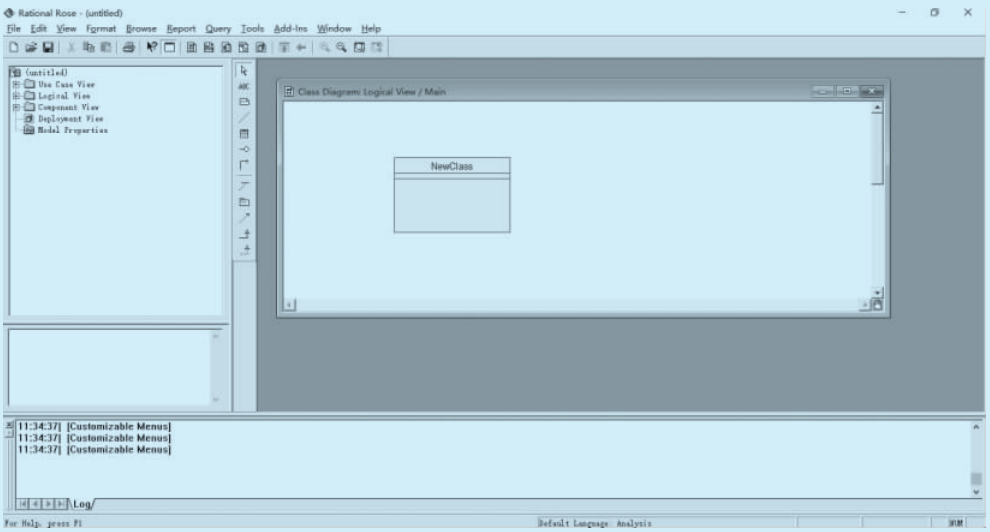

图 3-18　Rational Rose 界面

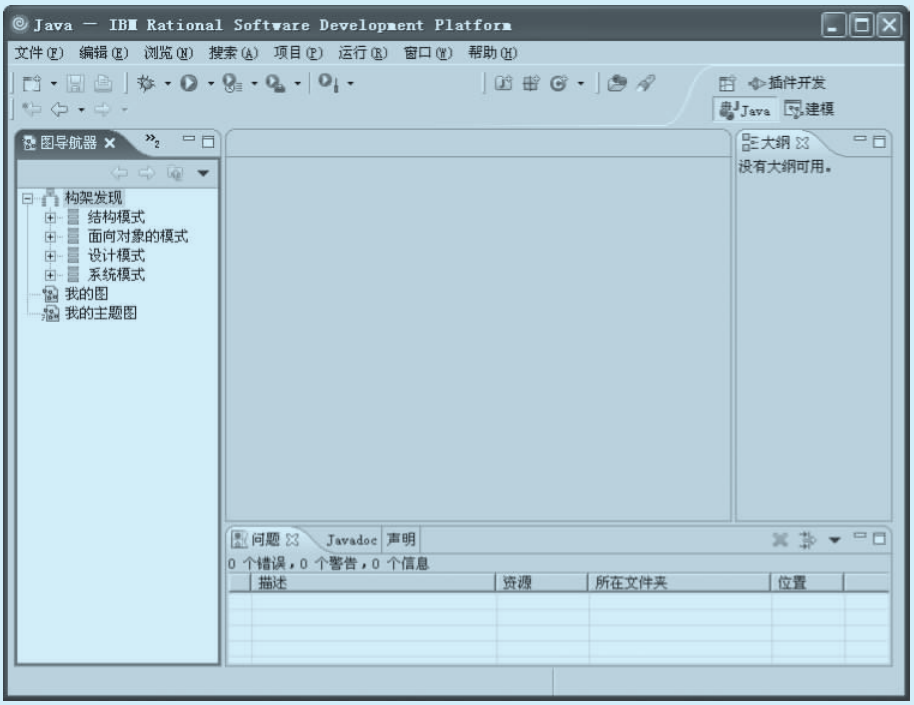

图 3-19　Rational Software Architect 界面