第 3 章　UML 基础知识

3.1　UML 概述

3.1.1　UML 简介

面向对象的分析与设计(OOA&D)方法的发展在 20 世纪 80 年代末至 90 年代中出现了一个高潮，UML(Unified Modeling Language，统一建模语言)是这个高潮的产物。它不仅统一了 Booch、Rumbaugh 和 Jacobson 的表示方法，而且对其作了进一步的发展，并最终统一为大众所接受的标准建模语言。它是一种可视化的建模语言，能够让系统构造者用标准的、易于理解的方式建立起能够表达他们设计思想的系统蓝图，并且提供一种机制，以便于不同的人之间有效地共享和交流设计成果。

3.1.2　与 UML 相关的 5 个开发阶段

系统开发有 5 个阶段：需求分析、分析、设计、编码和测试阶段。

1. 需求分析阶段

UML 语言利用用例来捕获客户的需求。通过用例模型，就可以使那些对系统感兴趣的外部参与者与他们要求系统具备的功能(即用例)一起被建模。外部参与者和用例之间是通过关系建模的，并且相互之间存在通信关联，或者被分解为更具体的层次结构。参与者和用例是由 UML 的用例图描述的。每一个用例都是用文本进行描述的，它确定了客户的需求，即在不考虑功能如何实现的情况下，客户所企盼的系统功能。需求分析不仅软件系统需要，业务过程同样也需要。

2. 系统分析阶段

分析阶段关注的是出现在问题域中的主要抽象(类和对象)和机制。被建模的类以及类之间的关系在 UML 的类图中被明确指定和描述。为了实现用例，各类之间需要相互协作。这种协作是由 UML 中的动态模型描述的。在分析阶段，只有在问题域(现实世界的概念)中的类才被建模，这里的类并不是那些在软件系统中定义了细节和解决方案的技术类。此阶段的类有用户界面类、数据库类、通信类和并发类等。

3. 系统设计阶段

在设计阶段，分析阶段的结果被扩展为一个技术解决方案。新类被加入进来，以提供以下一些技术基础结构：用户界面、处理对象存储的数据库、与其他系统的通信、与系统中各种设备的接口等。在分析阶段获得的问题域中的类被“嵌入”到此技术基础结构中，这样就能够同时改变问题域和基础结构。设计阶段将为随后的构建阶段产生详细的规格说明。

4. 编码阶段

在编码阶段(或者称之为构建阶段)，设计阶段的类被转换为使用面向对象编程语言编制(不推荐使用过程语言)的实际代码。这一任务可能比较困难，也可能比较容易，主要取决于所使用的编程语言本身的能力。用 UML 创建分析模型和设计模型时，最好避免试图将模型转换为代码。在开发的早期阶段，模型是帮助理解和搭建系统结构的一种手段。这样，如果在早期阶段就考虑代码，势必达不到预期的目标，即创建简单的和正确的模型。所以，编码是一个单独的阶段，也就是只有到了编码阶段，模型才被转换为代码。

5. 测试阶段

通常，一个系统需要经过单元测试、集成测试、系统测试和接受性测试。单元测试是对单个类或一组类的测试，一般情况下由编程者自己完成。集成测试是集成组件和类，以校验它们是否是像指定的那样进行合作。系统测试将系统看做一个“黑盒子”，检验系统是否具有最终用户所期望的功能。接受性测试与系统测试相似，它是由客户实施的，以验证系统是否满足客户的需求。不同的测试团队使用不同的 UML 图作为他们工作的基础：单元测试团队使用类图和类规格说明；集成测试团队一般使用组件图和协作图；系统测试团队则利用用例图检验最初在这些图中定义的系统行为。

3.2 相关 UML 内容介绍

3.2.1 视图

为复杂系统建模是一项要求非常广泛的任务。理想情况下，系统由单一的图形来描述，该图形明确地定义了整个系统，并且易于人们相互交流和理解。然而，通常这是不可能实现的。因为单一的图形不可能捕获到描述系统所需的所有信息。一般来说，系统通常是从多个不同的方面来描述的：功能方面(系统的静态结构和动态交互)、非功能方面(时间需求、可靠性、部署等方面)，以及组织结构方面(任务组织结构、代码模块的映射等)。因此，系统就由许多视图来共同描述，每一个视图代表完整系统描述的投影，显示系统的一个特定方面。

视图是由多个图来描述的，后者强调系统的某个特定方面的信息。这里会存在一点点重叠的内容，实际上，一个图可以是多个视图的组成部分。从不同的视图来观察系统，可以使人们在某段时间内集中注意系统的一个方面。一个特定视图中的图应该足够简单，便于交流，但是一定要与其他图和视图连贯一致，因而所有视图结合在一起(通过它们

各自的图)就描述了系统的完整画面。UML 图中包含许多代表被建模系统的模型元素的图形符号。如图 3-1 所示显示了 UML 的视图,解释如下:

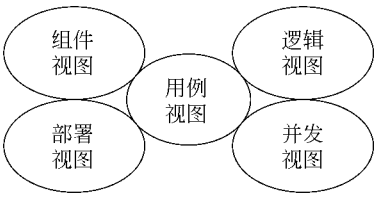

图 3-1　UML 的视图

- 用例视图(Use-case view):用例视图显示外部参与者(actor)观察到的系统功能。
- 逻辑视图(Logical view):逻辑视图从系统的静态结构和动态行为角度显示如何实现系统的功能。
- 组件视图(Component view):组件视图显示的是代码组件的组织结构。
- 并发视图(Concurrency view):并发视图显示的是系统的并发性,解决在并发系统中存在的通信和同步问题。
- 部署视图(Deployment view):部署视图显示的是系统的具体部署,也就是说,将系统部署到由计算机和设备(称之为结点(Node))组成的物理结构上。

当选择绘制 UML 图的工具时,应该确保该工具能够很容易地从一个视图导航到另一个视图。另外,为了察看一个功能在 UML 图中是如何被设计实现的,UML 工具还应该提供方便的视图切换功能,即用户可以很容易地切换到用例视图,以便察看外部用户是如何描述该功能的,或者切换到部署视图以便察看该功能是如何分布在物理结构上的,换句话说,也就是找到实现该功能的软件所在的那些计算机。

注意,UML 可以使用的其他视图有:静态-动态视图、逻辑-物理视图、工作流程视图,以及其他视图等。UML 不要求用户使用这些视图,但是,这些视图在 UML 的设计者们的脑海里是存在的,所以很可能大多数的 UML 工具都将会基于这些视图。

1. 用例视图

用例视图描述系统应该交付的功能,也就是外部参与者所看到的功能。参与者与系统打交道,参与者可以是一个用户或者是另一个系统。用例视图的使用者是客户、设计人员、开发人员以及测试人员。用例视图利用用例图进行描述,偶尔也会用活动图来描述。客户对系统的期望用法(也就是要求的功能)被当做多个用例在用例视图中进行描述,一个用例就是对系统的一个用法的通用描述。

用例视图是核心,因为它的内容驱动其他视图的开发。系统的最终目标,也就是系统将提供的功能是在用例视图中描述的,同时该视图还有其他一些非功能特性的描述,因此,用例视图将会对所有其他的视图产生影响。另外,通过测试用例视图,可以检验和最终校验系统。这种测试来自两个方面:一方面是客户,可以询问客户“这是您想要的吗?”;另一个方面就是已完成的系统,可以询问“系统是按照要求的方式运作的吗?”。

2. 逻辑视图

逻辑视图描述如何实现用例视图中提出的那些系统功能,它的使用者主要是设计人员和开发人员。与用例视图相比,逻辑视图关注系统的内部,它既描述系统的静态结构(类、对象以及它们之间的关系),也描述系统内部的动态协作关系。这种协作发生在为了实现既定功能,各对象之间进行消息传递的时刻。另外,逻辑视图也定义像永久性和并发

性这样的特性，同时还定义类的接口和内部结构。

系统的静态结构在类图和对象图中进行描述，而动态模型则在状态图、顺序图、协作图以及活动图中进行描述。

3. 组件视图

组件视图描述系统的实现模块以及它们之间的依赖关系。它的使用者主要是开发人员，它由组件图组成。组件是不同类型的代码模块，通过代码模块的结构和依赖关系来表示。组件视图中也可以添加组件的其他附加的信息，例如资源分配（为组件服务）或者其他管理信息，例如开发工作的进度报告。

4. 并发视图

并发视图处理的是将系统划分为进程和处理器。这是系统的非功能特性，该视图主要考虑资源的有效利用、代码的并行执行以及系统环境中异步事件的处理。除了将系统划分为并发执行的控制线程以外，并发视图也必须处理这些线程之间的通信和同步。

并发视图的使用者是开发人员和系统集成人员，并且该视图由动态图（状态图、协作图以及活动图）和实现图（组件图和部署图）组成。

5. 部署视图

部署视图显示系统的物理部署，例如计算机和设备（结点），以及它们之间是如何连接的。部署视图的使用者是开发人员、系统集成人员和测试人员，并且该视图由部署图表示。部署视图也包括一个显示组件如何在物理结构中部署的映射，例如，一个程序或对象在哪台计算机上执行。

3.2.2 图

图就是用来显示各种模型元素符号的实际图形，这些元素经过特定的排列组合来阐明系统的某个特定部分或方面。一般来说，一个系统模型拥有多个不同类型的图。一个图是某个特定视图的一部分。通常，图是被分配给视图来绘制的。另外，根据图中显示的内容，某些图可以是多个不同视图的组成部分。

本节只描述各种 UML 图的基本概念，而这些图的所有细节、它们的语法、确切意义，以及它们如何交互等内容请参考其他 UML 书籍。这些图都取自不同类型的系统，其目的是充分体现 UML 的多样性。

1. 用例图

用例图（Use-case Diagram）显示多个外部参与者以及它们与系统提供的用例之间的连接，如图 3-2 所示。一个用例是对系统提供的某个功能（该系统的一个特定用法）的描述。虽然实际的用例通常用普通文本来描述，并且此文本被当做该用例符号的一个文档特性，但是也可以利用一个活动图来描述用例。用例仅仅描述系统参与者从外部观察系统得到的那些功能（作为用户观察系统的系统行为），并不描述这些功能在系统内部是如何实现的。

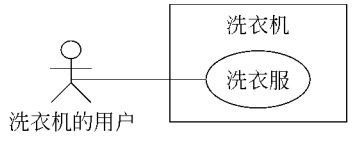

图 3-2 UML 用例图

也就是说，用例定义系统的功能需求。读者将在后面章节中看到用例图的实际应用。

代表洗衣机用户的直立小人形被称为参与者(actor)。椭圆形代表用例。注意参与者(它是发起用例的实体)可以是一个人也可以是另一个系统。还应该注意，用例位于一个代表着系统的矩形中，而参与者在矩形之外。

2. 类图

类图(Class Diagram)用来显示系统中各个类的静态结构，如图 3-3 所示。

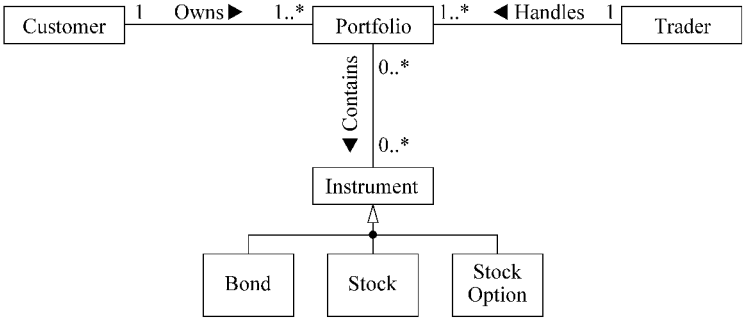

图 3-3　金融贸易系统的类图

类代表系统内处理的事物。这些类可以以多种方式相互连接在一起：关联(类互相连接)、依赖(一个类依赖/使用另一个类)、特殊化(一个类是另一个类的特化)或者打包(多个类组合为一个单元)。所有的这些关系连同每个类的内部结构都在类图中显示。其中，一个类的内部结构是用该类的属性和操作表示的。因为类图所描述的结构在系统生命周期的任何一处都是有效的，所以通常认为类图是静态的。

常常会使用特殊化(Specialize)、一般化(Generalize)、特化(Specialization)和泛化(Generalization)这几个术语来描述两个类之间的关系。例如，对于一个类 A(即父类)派生出另一个类 B(即子类)这样一个过程，也常常这样描述：类 A 可以特殊化为类 B，而类 B 可以一般化为类 A；或者类 A 是类 B 的泛化，而类 B 是类 A 的特化。

一个系统一般都有多个类图——并不是所有的类都放在一个类图中——并且一个类可以参与到多个类图中。读者将会在后面章节中看到类图的应用实例。

3. 状态图

一般来说，状态图(State Diagram)是对类的描述的补充。它用于显示类的对象可能具备的所有状态，以及那些引起状态改变的事件，如图 3-4 所示。对象的一个事件可以是另一个对象向其发送的消息，例如到了某个指定的时刻，或者已经满足了某条件。状态的变化称之为转换(Transition)。一个转换也可以有一个与之相连的动作，后者用以指定完成该状态转换应该执行的操作。

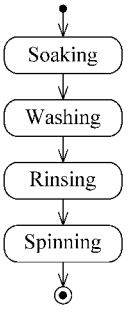

图 3-4　UML 状态图

UML 状态图如图 3-4 所示，该图能够描述上面所提及的状态。该图说明洗衣机可以从一个状态转移到另一个状态。在实际建模时，并不需要为所有的类都绘制状态图，仅对那些具有多个明确状

态的类，并且类的这些不同状态会影响和改变类的行为时才绘制类的状态图。另外，也可以为系统绘制整体状态图。

4. 顺序图

顺序图(Sequence Diagram)显示多个对象之间的动态协作，如图 3-5 所示。顺序图重点是显示对象之间发送的消息的时间顺序。它也显示对象之间的交互，就是在系统执行时，某个指定时间点将发生的事情。顺序图由多个用垂直线显示的对象组成，图中时间从上到下推移，并且顺序图显示对象之间随着时间的推移而交换的消息或函数。消息是用带消息箭头的直线表示的，并且它位于垂直对象线之间。时间说明以及其他注释放到一个脚本中，并将其放置在顺序图的页边空白处。读者将会在后面章节看到顺序图的应用实例。

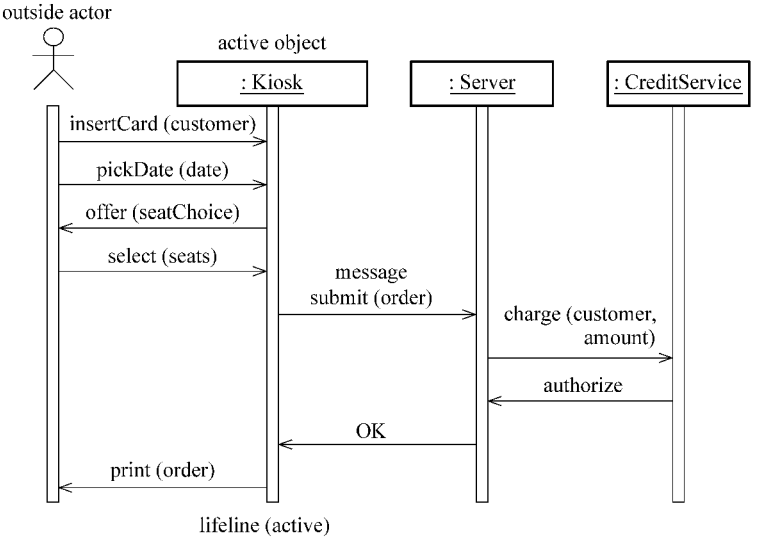

图 3-5 UML 顺序图

5. 协作图

协作图(Collaboration Diagram)像顺序图一样显示动态协作。为了显示一个协作，通常需要在顺序图和协作图之间做选择。除了显示消息的交换(称之为交互)以外，协作图也显示对象以及它们之间的关系(有时指上下文)。通常，选择顺序图还是协作图的决定条件是：如果时间或顺序是需要重点强调的方面，那么选择顺序图；如果上下文是需要重点强调的方面，那么选择协作图。顺序图和协作图都用于显示对象之间的交互。

协作图当做一个对象图来绘制，它显示多个对象以及它们之间的关系(利用类/对象图中的符号来绘制)。协作图中对象之间绘制的箭头显示对象之间的消息流向。图中的消息上放置标签，用于显示消息发送的顺序。协作图也可以显示条件、迭代和返回值等信息。当开发人员熟悉消息标签语法之后，就可以读懂对象之间的协作以及跟踪执行流程和消息交换顺序。协作图也可以包括活动对象，这些活动对象可以与其他活动对象并发地执行，如图 3-6 所示。

6. 活动图

活动图(Activity Diagram)用于显示一系列顺序的活动，如图 3-7 所示。

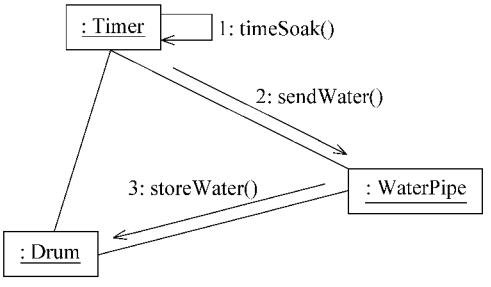

图 3-6　UML 协作图

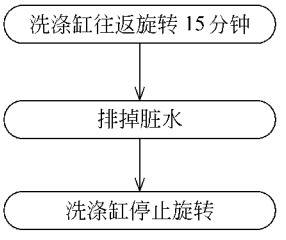

图 3-7　UML 活动图

尽管活动图也可以用于描述像用例或交互这类的活动流程，但是一般来说，它主要还是用于描述在一个操作内执行的那些活动。活动图由多个动作状态组成，后者包含将被执行的活动(即一个动作)的规格说明。当动作完成后，动作状态将会改变，转换为一个新的状态(在状态图内，状态在进行转换之前需要标明显式的事件)。于是，控制就在这些互相连接的动作状态之间流动。同时，在活动图中也可以显示决策和条件，以及动作状态的并发执行。另外，活动图也可以包含那些被发送或接收的消息的规格说明，这些消息是被执行动作的一部分。

7. 组件图

组件图是用代码组件来显示代码物理结构的。其中，组件可以是源代码组件、二进制组件或一个可执行的组件。因为一个组件包含它所实现的一个或多个逻辑类的相关信息，于是就创建了一个从逻辑视图到组件视图的映射。根据组件图中显示的那些组件之间的依赖关系，可以很容易地分析出其中某个组件的变化将会对其他组件产生什么样的影响。另外，组件也可以用它们输出的任意的接口来表示，例如 OLE/COM 接口；并且它们可以被聚集在包内。一般来说，组件图用于实际的编程工作中，如图 3-8 所示。

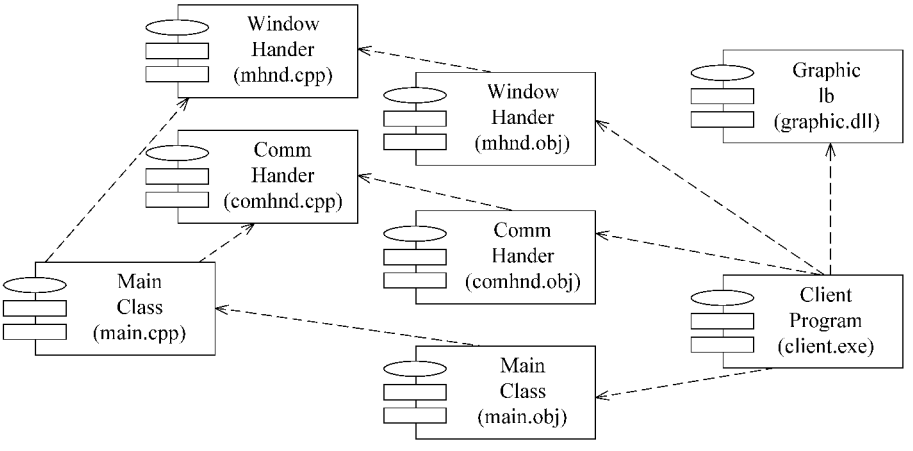

图 3-8　显示代码组件之间依赖关系的组件图

8. 部署图

UML 部署图显示了基于计算机系统的物理体系结构。它可以描述计算机，展示它们之间的连接，以及驻留在每台机器中的软件。每台计算机用一个立方体来表示，立方体之间的连线表示这些计算机之间的通信关系。图 3-9 是部署图的一个例子。

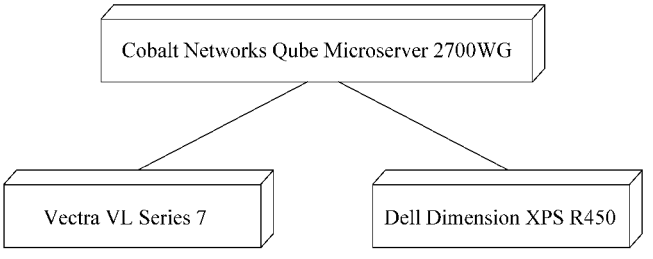

图 3-9　UML 部署图