面向对象分析

5.1 面向对象方法学概述

传统的软件工程方法学部分地缓解了软件危机，许多中小规模软件项目都获得了成功。但是，人们也注意到当把这种方法学应用于大型软件产品的开发时，似乎很少取得成功。

自 20 世纪 80 年代中期起，人们开始注重面向对象分析和设计的研究，逐步形成了面向对象方法学。到了 20 世纪 90 年代，面向对象方法学已经成为人们在开发软件时首选的范型。该方法采用人类认识客观世界过程中习惯的思维方式，更加直观、自然地描述客观事物，成为一种快速高效的软件开发方法，已广泛应用于数据库系统、分布式系统、网络管理系统、人工智能等领域软件的开发。

5.1.1 面向对象的概念

1. 对象(object)

应用领域中有意义的、与所要解决的问题有关系的任何事物都可以作为对象。对象既可以是具体的物理实体的抽象，也可以是人为的概念，或者是任何有明确边界和意义的东西。面向对象方法学中的对象是由描述该对象属性的数据以及可以对这些数据施加的所有操作封装在一起构成的统一体。

对象以数据为中心，操作围绕对其数据所需要做的处理来设置，为了完成某个操作必须通过它的公有接口向对象发消息，请求它执行它的某个操作，处理它的私有数据。对象实现了数据封装，它的私有数据完全被封装在盒子内部，对外是隐藏的、不可见的，对私有数据的访问或处理只能通过公有操作进行。不同对象各自独立地处理自身的数据，彼此通过传递信息完成通信，本质上具有并行性。对象内部各种元素彼此结合得很紧密，内聚性相当强，而对象之间的耦合通常比较松，因此模块独立性好。

2. 类(class)

类就是对具有相同数据和相同操作的一组相似对象的定义。类的概念来自于人们认识自然、认识社会的过程。在这一过程中，人们主要使用两种方法：由特殊到一般的归纳法和由一般到特殊的演绎法。在归纳的过程中，从一个个具体的事物中把共同的特征抽取出来，形成一个一般的概念，这就是“归类”。在演绎的过程中又把同类的事物，根据不同的特征分成不同的小类，这就是“分类”。类的内部状态是指类集合中对象的共同状态，类的运动规律

是指类集合中对象的共同运动规律。

3. 实例(instance)

一个特定的类有许多具体的对象,这些对象都被称为实例。当使用“对象”这个术语时,既可以指一个具体的对象,也可以泛指一般的对象,但是,当使用“实例”这个术语时,必然是指一个具体的对象。

4. 消息(message)

消息是向对象发出的服务请求,通常包含接收消息的对象、消息选择符(消息名)、消息的变元(零个或多个)。消息通信与对象的封装原则密切相关。封装使对象成为各司其职、互不干扰的独立单位;消息通信则为其提供唯一合法的动态联系途径,使其行为可以互相配合,构成一个有机的系统。

5. 方法(method)

方法就是对象能执行的操作,即类中定义的服务。方法描述了对象执行操作的算法,响应消息的方法。

6. 属性(attribute)

属性就是类中定义的数据。它是对客观世界实体所具有的性质的抽象。类的每个实例都有自己特有的属性值。

7. 封装(encapsulation)

封装是在面向对象的程序中,把数据和实现操作的代码集中起来放在对象内部。一个对象好像是一个不透明的黑盒子,表示对象状态的数据和实现操作的代码与局部数据都被封装在黑盒子里面,从外面是看不见的,更不能从外面直接访问或修改这些数据和代码。

封装也就是信息隐藏,通过封装对外界隐藏了对象的实现细节。对象类实质上是抽象数据类型。类把数据说明和操作说明与数据表达和操作实现分离开了,使用者仅需要知道它的说明(值域及可对数据施加的操作)就可以使用它。

8. 继承(inheritance)

继承是父类和子类之间共享数据结构和方法的一种机制,是以现存的定义内容为基础,建立新定义内容的技术,是类之间的一种关系。

继承性通常表示父类与子类的关系,继承有两种:单继承,指子类只继承一个父类的数据结构和方法;多重继承,指子类继承了多个父类的数据结构和方法。子类的公共属性和操作归属于父类,并为每个子类共享,子类继承了父类的特性。

9. 多态(polymorphism)

多态性是指多种类型的对象在相同的操作或函数、过程中取得不同结果的特性。利用多态技术,用户可发送一个通用的消息,而实现的细节则由接收对象自行决定,这样同一消

息就可调用不同的方法。多态性不仅增加了面向对象软件的灵活性，进一步减少了信息冗余，而且显著提高了软件的可复用性和可扩充性。

当扩充系统功能增加新的实体类型时，只需要派生出与新实体类相应的新的子类，并在新派生出的子类中定义符合该类需要的虚函数，完全不需要修改原有的程序代码，甚至不需要重新编译原有的程序。

10. 重载(overloading)

重载有函数重载和运算符重载两种。函数重载是指在同一作用域内的若干个参数特征不同的函数可以使用相同的函数名字；运算符重载是指同一个运算符可以施加于不同类型的操作数之上。当然，当参数特征不同或被操作数的类型不同时，实现函数的算法或运算符的语义是不相同的。重载进一步提高了面向对象系统的灵活性和可读性。

5.1.2 面向对象方法的要点

面向对象方法有4个要点。

(1) 对象。客观世界由各种对象组成，任何事物都是对象，复杂的对象可以由比较简单的对象以某种方式组合而成。因此，面向对象的软件系统是由对象组成的，软件中的任何元素都是对象，复杂的软件对象由比较简单的对象组合而成。面向对象方法用对象分解取代了传统方法的功能分解。

(2) 类。把所有对象都划分成各种对象类(简称为类)，每个对象类都定义了一组数据和一组方法。数据用于表示对象的静态属性，是对象的状态信息。类中定义的方法，是允许施加于该类对象上的操作，是该类所有对象共享的。

(3) 继承。按照子类(或称为派生类)与父类(或称为基类)的关系，把若干个对象类组成一个层次结构的系统(也称为类等级)。在这种层次结构中，通常下层的派生类具有和上层的基类相同的特性(包括数据和方法)，这种现象称为继承。

(4) 传递消息。对象彼此之间仅能通过传递消息互相联系。对象与传统的数据有本质区别，对象不是被动地等待外界对自己施加操作，而是进行处理的主体，必须发消息请求它执行它的某个操作，处理它的私有数据，不能从外界直接对它的私有数据进行操作。

5.1.3 面向对象方法学的优点

面向对象方法学的基本思想是尽可能按照人类认识世界的方法和思维方式分析和解决问题。该方法可提供更加清晰的需求分析和设计，是指导软件开发的系统方法，其优点主要表现在以下几方面。

(1) 与人类习惯的思维方法一致。

面向对象的软件技术以对象为核心，软件系统由对象组成。对象之间通过传递消息互相联系，以模拟现实世界中不同事物彼此之间的联系。面向对象的设计方法强调模拟现实世界中的概念而非算法。

面向对象方法学符合人类分析问题、解决问题的习惯思维方式。面向对象的环境提供了强有力的抽象机制，便于用户在利用计算机软件系统解决复杂问题时使用习惯的抽象思维工具，使问题空间与解空间一致，利于对开发过程各阶段综合考虑，有效地降低开发复杂

度，提高软件质量。

（2）稳定性好。

面向对象的软件系统的结构是根据问题领域的模型建立起来的，而不是基于对系统应完成的功能的分解，因此，当对系统的功能需求变化时并不会引起软件结构的整体变化，往往仅需要做一些局部性的修改。以对象为中心构造的软件系统比较稳定。

（3）可复用性好。

在面向对象方法所使用的对象中，数据和操作正是作为平等伙伴出现的。因此，对象具有很强的自含性。此外，对象固有的封装性和信息隐藏机制，使得对象的内部实现与外界隔离，具有较强的独立性。由此可见，对象是比较理想的可复用模块和软件成分。

（4）较易开发大型软件产品。

用面向对象方法学开发软件时，构成软件系统的每个对象就像一个微型程序，有自己的数据、操作、功能和用途。因此，可以把一个大型软件产品分解成一系列本质上相互独立的小产品来处理，这不仅降低了开发的技术难度，而且也使得对开发工作的管理变得容易多了。

（5）可维护性好。

当对软件的功能或性能的要求发生变化时，通常不会引起软件的整体变化，往往只需对局部做一些修改，自然比较容易实现。

类是理想的模块机制，它的独立性好，修改一个类通常很少会牵扯其他类。面向对象软件技术特有的继承机制，使得对软件的修改和扩充比较容易实现，通常只需要从已有类派生出一些新类，无须修改软件原有成分。面向对象软件技术的多态性机制，使得当扩充软件功能时，需要对原有代码所做的修改进一步减少，需要增加的新代码也有所减少。

面向对象的软件技术符合人们习惯的思维方式，用这种方法建立的软件系统的结构与问题空间的结构基本一致。因此，面向对象的软件系统比较容易理解。

对面向对象的软件的维护主要通过从已有类派生出一些新类来实现。因此，维护后的测试和调试工作也主要围绕这些新类进行。类是独立性很强的模块，对类的测试通常比较容易实现，如果发现错误也往往集中在类的内部，比较容易调试。

5.1.4 面向对象开发方法

目前，面向对象开发方法的研究已日趋成熟，且已有很多面向对象产品问世。开发方法有 Booch 方法、Coad 方法、OMT 方法和 UML 语言等。

（1）Booch 方法。Booch 最先描述了面向对象的软件开发方法的基础问题，指出面向对象开发是一种根本不同于传统的功能分解的设计方法。面向对象的软件分解更接近人对客观事物的理解，而功能分解只通过问题空间的转换获得。

（2）Coad 方法。Coad 方法是 Coad 和 Yourdon 于 1989 年提出的面向对象开发方法。该方法的主要优点是通过多年来大系统开发的经验与面向对象概念的有机结合，在对象、结构、属性和操作的认定方面，提出了一套系统的原则。该方法完成了从需求角度进一步进行类和类层次结构的认定。尽管 Coad 方法没有引入类和类层次结构的术语，但事实上已经在分类结构、属性、操作、消息关联等概念中体现了类和类层次结构的特征类。

（3）OMT 方法。对象建模技术（object modeling technique，OMT）是美国通用电气公司提出的一套系统开发技术。它以面向对象的思想为基础，通过对问题进行抽象，构造出一

组相关的模型，从而能够全面地捕捉问题空间的信息。该方法是一种新兴的面向对象的开发方法，开发工作的基础是对真实世界的对象建模，然后围绕这些对象使用分析模型来进行独立于语言的设计，面向对象的建模和设计促进了对需求的理解，有利于开发出更清晰、更容易维护的软件系统。该方法为大多数应用领域的软件开发提供了一种实际的、高效的保证。

(4) UML 语言。1995 年至 1997 年，软件工程领域取得重大进展，其成果超过软件工程领域过去 10 多年的总和，最重要的成果之一是统一建模语言(Unified Modeling Language，UML)的出现。UML 成为面向对象技术领域内占主导地位的标准建模语言，是一种定义良好、易于表达、功能强大且普遍适用的建模技术和方法，融入了软件工程领域的新思想、新方法和新技术。不仅支持面向对象的分析与设计，还支持从需求分析开始的软件开发全过程。不仅统一了 Booch 方法、OMT 方法、OOSE 方法的表示方法，而且对其做了进一步的发展，最终成为大众接受的标准建模语言。

5.2 统一建模语言 UML

5.2.1 UML 简介

统一建模语言是一种通用的可视化建模语言，可以用来描述、可视化、构造和文档化软件密集型系统的各种工件。它由信息系统和面向对象领域的 3 位著名的方法学家 Grady Booch、James Rumbaugh 和 Ivar Jacobson 提出，记录了与被构建系统有关的决策和理解，可用于对系统的理解、设计、浏览、配置、维护以及控制系统的信息。这种建模语言已经得到了广泛的支持和应用，并且已被 ISO 组织发布为国际标准。

UML 用来捕获系统静态结构和动态行为的信息。其中，静态结构定义了系统中对象的属性和方法，以及这些对象间的关系。动态行为则定义了对象在不同时间、状态下的变化以及对象间的相互通信。此外，UML 可以将模型组织为包的结构组件，使得大型系统可被分解成易于处理的单元。

UML 是独立于过程的，适用于各种软件开发方法、软件生命周期的各个阶段、各种应用领域以及各种开发工具。UML 规范没有定义一种标准的开发过程，但更适用于迭代式的开发过程。它是为支持现今大部分面向对象的开发过程而设计的。

UML 不是一种程序设计语言，但用 UML 描述的模型可以和各种编程语言相联系。可以使用代码生成器将 UML 模型转换为多种程序设计语言代码，或者使用逆向工程将程序代码转换成 UML。把正向代码生成和逆向工程这两种方式结合起来就可以产生双向工程，使得既可以在图形视图下工作，也可以在文本视图下工作。

1996 年 6 月，Booch、Rumbaugh 和 Jacobson 将 UM 更名为 UML，并发布 UML 0.9。在当时，UML 就获得了工业界、科技界和用户的广泛支持。1996 年底，UML 已经占领了面向对象技术市场 85%的份额，成为事实上的可视化建模语言的工业标准。1997 年 11 月，UML 1.1 规范被 OMG 全体成员通过，并被采纳为规范，OMG 也承担了进一步完善 UML 的工作。

在 1997—2002 年，OMG 成立的 UML 修订任务组对 UML 进行修订，陆续开发了

UML 的 1.3、1.4 和 1.5 版本。在有了若干年对 UML 的使用经验后，OMG 提出了升级 UML 的建议方案，以修正使用中发现的问题，并扩充一部分应用领域中所需的额外功能。2005 年 7 月发布了 UML 2.0 规范。在 2007—2011 年，UML 陆续发布了几个版本的规范。其中，2011 年 8 月发布的 UML 2.4.1 在 2012 年被 ISO 正式定为国际标准。2017 年 12 月，OMG 组织发布 UML 2.5.1 版本。

5.2.2 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，如图 5.1 所示。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，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，使用更深层次的语言特征开展建模工作。

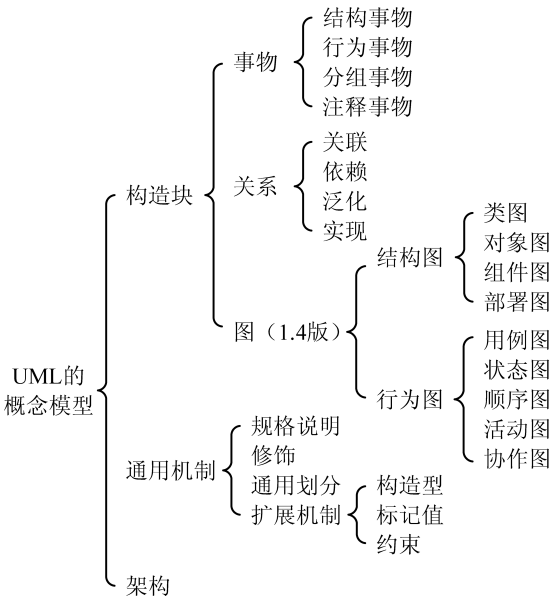

图 5.1 UML 的概念模型

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)3 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。

(1) 事物。

在 UML 中，事物是构成模型图的主要构造块，代表了一些面向对象的基本概念。事物被分为以下 4 种类型。

① 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。

② 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,主要是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。

③ 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。另外,其他基于包的扩展事物(如子系统、层等)也可作为分组事物。

④ 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,内容是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,连接被解释的元素。几乎所有的 UML 图形元素都可以用注解来说明。

(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。UML 中有 4 种主要的关系。

① 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"。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,称为聚合/组合关系,聚合/组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。

② 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,它依赖于该独立元素。

③ 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,此类泛化称为受限泛化。

④ 实现(realization)描述规格说明和其实现的元素之间的连接的关系。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:接口和实现接口的类和组件之间;用例和实现它们的协作之间。

(3) 图(1.4 版)。

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;这种展示通过 UML 的图实现。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 图由通过路径连接的图形构成。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。

UML 1.4 共包含 9 种图,见图 5.1。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。随着软件工程技术的变迁，人们对图有不同的分类方法和解释方式。在升级到 UML 2.0 规范后，共包含 14 种图，大部分与之前版本相同，表示法上略有区别，对部分图的功能进行了细分，增加了几个新的图，如图 5.2 所示。UML 2.0 中增加了包图、组合结构图、时间图以及交互概览图。另外，状态机图是由原来的状态图改名而来的，通信图是由原来的协作图改名而来的。

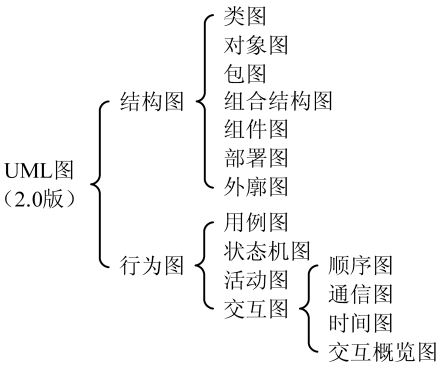

图 5.2　UML 2.0 中的图

2. 通用机制

UML 提供了 4 种通用机制，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。

(1) 规格说明(specifications)。

UML 不仅仅是一个图形化的语言，而且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。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，可以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。

(2) 修饰(adornments)。

UML 中大多数的元素都有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。修饰是对规格说明的文字的或图形的表示。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) 通用划分(common divisions)。

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。

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。类和对象就是一种典型的类型-实例划分。

接口是一个系统或对象的行为规范。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。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。

(4) 扩展机制(extensibility mechanisms)。

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、标记值和约束 3 种。

① 构造型(stereotype)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。UML 中预定义了一些构造型(如接口)供建模者使用，用户也可以根据自己的需要自

行定义。

② 标记值(tagged value)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。标记值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。标记值定义被构造型所拥有。

③ 约束(constraint)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。

3. 架构

UML 标准只是提出了这些图形的语法模型和语义模型,并没有针对这些图形的使用提供很好的支持。为了有效地利用这些模型,需要结合不同的软件工程过程,定义组织图形的架构。

一种被大家广泛接受的 UML 架构源自 Rational 统一过程(参见 1.5 节)提供的"4+1"视图架构模型。"4+1"视图架构模型是由 Philippe Kruchten 于 1995 年在 *IEEE Software* 的一篇名为 *The 4+1 View Model of Architecture* 的论文中提出的。在这个模型中,软件开发者从 5 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、实现视图、进程视图、部署视图和用例视图。每个视图只反映系统的某一部分,5 个视图结合起来才可以描述整个系统的结构,5 个视图之间的关系如图 5.3 所示。

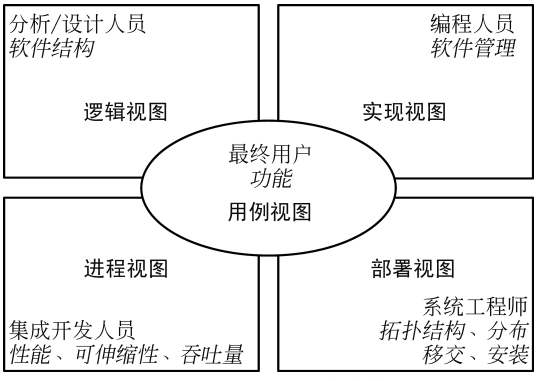

图 5.3 "4+1"视图架构模型

(1) 用例视图(use-case view)是建模过程的起点和依据,面向最终用户,描述系统的功能性需求。所有其他视图都是从用例视图派生而来的,该视图把系统的基本需求捕获为用例并提供构造其他视图的基础。

(2) 逻辑视图(logical view)面向系统分析和设计人员,描述软件结构。它来自功能需求,用于描述问题域的结构。作为类和对象的集合,它的重点是展示对象和类是如何组成系统、实现所需系统行为的。

(3) 进程视图(process view)面向系统集成人员,描述系统性能、可伸缩性、吞吐量等信息。其目标是为系统中的可执行线程和进程建模,使它们作为活动类。事实上,它是逻辑视图面向进程的变体,包含所有相同的工件。

(4) 实现视图(implementation view)面向编码人员,描述系统的组装和配置管理。其目标是对组成基于系统的物理代码的文件和构件进行建模。

(5) 部署视图(deployment view)面向系统工程师,描述系统的拓扑结构、分布、移交、安装等信息。建模的目标是把组件物理地部署到一组物理的、可计算的结点(如计算机)上。

软件项目和传统的工程项目的首要问题是一致的,都是用户的需求。如果没有需求,整个项目就没有进行下去的目标和驱动力。因此在"4+1"的5种视图中最先被使用的是用例视图。用例视图是根据用户的需求可以直接产生和描述的,因此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用例来寻找功能和非功能点、检验系统是否满足要求。

当输出了用例视图之后,可以进一步使用逻辑视图来细化场景。这一步的细化包括3个方面:找到用例中的所有关键交互;使用软件术语描述交互逻辑;设计更下层的元素。逻辑视图是架构设计师和项目实际开发人员的通用交流语言,是一个低于用例、高于详细设计的视图。逻辑视图是静态、注重问题分化、关注用户使用流程的。它更多地在尝试使用编程术语描述问题,而不是解决问题。

进程视图、实现视图和部署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,但是它们的内容却是紧密相关的。实现视图关注各种程序包的使用,进程视图关注运行时概念,部署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。这3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3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。

5.2.3 UML的应用范围

UML以面向对象的方式来描述系统。最广泛的应用是对软件系统进行建模,但它同样适用于许多非软件领域的系统。从理论上说,任何具有静态结构和动态行为的系统都可以使用UML进行建模。从软件生命周期来看,UML适用于系统开发的全过程,它的应用贯穿于从需求分析到系统建成后测试的各个阶段。

- 需求分析阶段。可以用用例捕获用户的需求。通过用例建模,可以描述对系统感兴趣的外部角色及其对系统的功能要求(用例)。
- 分析阶段。分析阶段主要关心问题域中的基本概念(如抽象、类和对象等)和机制,需要识别这些类以及它们相互间的关系,可以用UML的逻辑视图和动态视图来描述。类图描述系统的静态结构,协作图、顺序图、活动图和状态图描述系统的动态行为。
- 设计阶段。把分析阶段的结果扩展成技术解决方案,加入新的类来定义软件系统的技术方案细节。设计阶段使用与分析阶段类似的方式使用UML。
- 构造(编码)阶段。把来自设计阶段的类转换成某种面向对象程序设计语言的代码,指导并减轻编码工作。
- 测试阶段。作为测试阶段的依据,不同测试小组使用不同的UML图作为他们工作的依据。其中,类图指导单元测试;构件图和协作图指导集成测试;用例图指导系统