第 3 章　面向对象方法与 UML

面向对象方法是一个非常实用且强有力的软件开发方法。它起源于 20 世纪 60 年代末挪威的 K. Nyguard 等人推出的编程语言 Simula 67。在这个语言中引入了数据抽象和类的概念，但真正为面向对象程序设计奠定基础的是由 Alan Keyz 主持推出的 Smalltalk 语言，“面向对象”这个词也是 Smalltalk 首先采用的。1976 年推出了 Smalltalk-72，1978 年推出了 Smalltalk-76，由此逐步发展和完善了面向对象程序设计的概念。1981 年由 Xerox Learning Research Group 所研制的 Smalltalk-80 系统，全面地体现了面向对象程序设计语言的特征。Smalltalk 被认为是第一个真正的面向对象程序设计语言。由于 Smalltalk-80 语言的推广使用，导致了面向对象语言的蓬勃发展，有的是传统语言的扩充，有的是新开发的面向对象语言，其中有代表性的包括 Objective-C(1986)、C++ (1986)、Self(1987)、Eiffel (1987)、CLOS(1986)、Object Oriented Pascal 等。1986 年，Grady Booch 首先提出“面向对象设计”概念，从那以后，越来越多的人投入到面向对象的研究领域。一方面，面向对象向软件开发的前期阶段，包括面向对象设计、面向对象分析，是按照 OOP—OOD—OOA 的顺序发展的，而在进行系统开发时却应该按照 OOA—OOD—OOP 的顺序；另一方面，面向对象在越来越广泛的计算机软硬件领域得以发展，如面向对象程序设计方法学、面向对象数据库、面向对象操作系统、面向对象软件开发环境、面向对象的智能程序设计、面向对象的计算机体系结构等。面向对象技术已成为软件开发的主流技术。

3.1　面向对象系统的概念

3.1.1　面向对象系统的概念

为了讨论面向对象的系统，必须首先明确什么是“面向对象”？Coad 和 Yourdon 给出了一个定义：

面向对象＝对象＋类＋继承＋消息通信

如果一个系统是使用这样 4 个概念设计和实现的，则可认为这个系统是面向对象的。面向对象系统的每个成分都应是对象，计算是通过新对象的建立和对象之间的通信来执行的。

面向对象系统具有许多特色。

- 系统的定义从问题领域的实体出发，与人类习惯的思维方式一致。
- 搭建的系统结构稳定性好，修改可以局部化。
- 系统及体系结构可以使用构件组装，可复用性好。
- 软件系统容易理解，容易修改，容易测试，适合开发大型的软件产品。
- 软件体系结构严格按照信息(细节)隐蔽的原则设计，产品可维护性好。

3.1.2 对象

1. 对象的定义

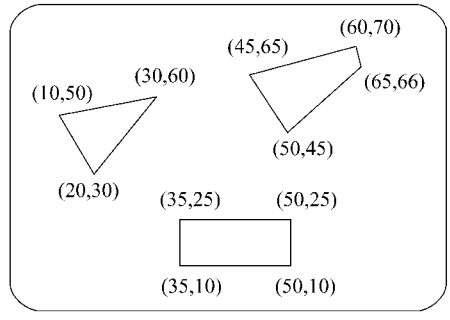
对象(Object)是系统中用来描述客观事物的实体，是构成系统的基本单位。每个对象可用它的名字、它本身的一组属性和它可以执行的一组操作来定义。

属性表征了对象的静态特征，在 C++ 中称为数据成员，一般通过封装在对象内部的数据存储来定义。一旦对象的数据存储都赋了值，这个对象的状态就确定了。

操作又称为方法或服务，它描述了对象执行的功能，因此，它表征了对象的动态行为，在 C++ 中称为成员函数。若通过消息传递，还可以为其他对象使用。

对象属性的值只能通过执行对象的操作来改变，也可以说，对象的状态只能通过执行对象的操作才能修改。

例如，在计算机屏幕上画 3 个多边形。为简化起见，每个多边形可以看做是一个由有序顶点集合定义的对象。这个顶点集合给出了多边形对象的状态，包括它的形状和它在屏幕上的位置。在多边形上的操作包括 **draw**(在屏幕上显示它)、**move**(从原来的位置上移动到一个指定的新位置)及 **contains**(检查某个指定的点是否在多边形内部)。图 3.1(a)显示了在计算机屏幕上的 3 个多边形对象和定义它们的点。图 3.1(b)给出了这些多边形对象的图形表示。



(a) 在计算机屏幕上的三个多边形

triangle	quadrilateral1	quadrilateral2
(10,50) (30,60) (20,30)	(35,10) (50,10) (35,25) (50,25)	(45,65) (50,45) (65,66) (60,70)
draw move(Δx,Δy) contains(aPoint)	draw move(Δx,Δy) contains(aPoint)	draw move(Δx,Δy) contains(aPoint)

(b) 表示多边形的 3 个对象

图 3.1 多边形对象

2. 对象的分类

对象可以分为 5 种：物理、角色、事件、交互和规格说明。每个应用系统可以拥有某几种或所有各种对象，但也不必特意对每个对象进行分类。

(1) 物理对象(Physical Objects)——物理对象是最易识别的对象，通常可以在问题领域的描述中找到，它们的属性可以标识和测量。例如，大学课程注册系统中的学生

对象；一个网络管理系统中各种网络物理资源对象（如开关、CPU 和打印机）都是物理对象。

（2）角色对象（Roles）——一个实体的角色也可以抽象成一个单独的对象。角色对象的操作是由角色提供的技能。例如，一个面向对象系统中通常有“管理器”对象，它履行协调系统资源的角色。一个窗口系统中通常有“窗口管理器”对象，它扮演协调鼠标器按钮和其他窗口操作的角色。特别地，一个实际的物理对象可能同时承担几个角色。例如，一个退休教师同时扮演退休者和教师的角色。

（3）事件对象（Incidents）——一个事件是某种活动的一次“出现”。例如“鼠标”事件。一个事件对象通常是一个数据实体，它管理“出现”的重要信息。事件对象的操作主要用于对数据的存取。如“鼠标”事件对象有诸如光标坐标、左右键、单击、双击等信息。

（4）交互对象（Interactions）——交互表示两个对象之间的关系，这种类型的对象类似于在数据库设计时所涉及的“关系”实体。当实体之间是多对多的关系时，利用交互对象可将其简化为两个一对多的关系。例如，在大学课程注册系统中，学生和课程之间的关系是多对多的关系，可设置一个“选课”交互对象来简化它们之间的关系。

（5）规格说明对象（Specifications）——规格说明对象表明组合某些实体时的要求。规格说明对象中的操作支持把一些简单的对象组合成较复杂的对象。例如，一个“烹饪”对象定义各种调料和它们的量，以及它们组合的次序和方式。

3. 对象的特点

（1）对象是消息处理的主体。对象之间是通过消息互相通信的。

（2）对象是以数据为中心的。所有操作都与对象的属性相关，而且操作的结果往往与当时所处的状态（属性的值）有关。

（3）实现了数据封装。对象是一个黑盒，其属性值对外不可见，被完全封装在盒子内部，对属性值的访问只能通过接口中定义的（公有）接口操作进行。为了使用对象内的属性值，只需知道属性值的取值范围和可以访问该属性的接口操作，无需知道表征属性的具体数据结构和操作的实现算法。

（4）模块独立性好。由于前 3 个特点，故对象内部各种成分彼此相关，联系紧密，内聚性强。又由于完成对象功能所需的操作和相关数据结构基本上都被封装在对象内部，形成了面向对象系统的基本模块，因而它与外界的联系较少，对象之间的耦合比较松散。

（5）具有并行的特点。不同的对象各自独立地处理自身的数据，彼此通过发送消息、传递信息来完成通信。所以它们具有并行工作的特点。

对象有两个视图，分别表现在设计和实现方面。设计视图把对象看做实体，产生有关实体的声明，包括实体的属性和可以执行的操作，但不涉及实现功能的一系列动作。实现视图用对象表示在应用程序代码中的实体，是数据存储与相关操作的统一的封装体，是数据抽象和过程抽象的实例化。

3.1.3 类与封装

类（Class）是一组具有相同结构、相同服务、共同关系和共同语义的对象集合。类的定义包括类名、一组数据属性和在数据上的一组合法操作。

在一个类中，每个对象都是类的实例（Instance），它们都具有相同的属性（但具有不同的属性值），都可使用类中定义的方法。一个实例的属性称为该实例的实例变量，实例变量的值一旦确定，该实例的状态也就确定下来了。

例如，在图 3.1 中有两个四边形，它们具有相同的数据结构和相同的操作，可以基于它们定义一个如图 3.2 所示的 quadrilateral 类。该类的每个实例有同样的一组实例变量和服务。就这个意义来讲，类 Quadrilateral 提供了一个抽象，表示了所有四边形对象，定义了各个四边形实例中的实例变量和可以作用于任一实例上的一组操作。

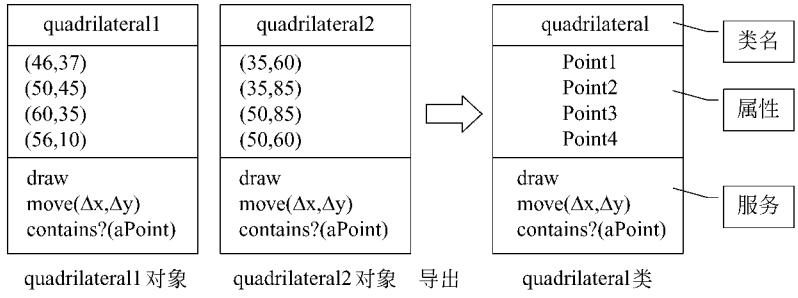


图 3.2　quadrilateral（四边形）的类定义

一旦有了类的定义，还可以创建属于该类的新实例，但必须定义类的构造函数，以便在创建新的实例时系统利用这些构造函数对它们初始化。

类的定义应遵循抽象数据类型（ADT）的原则，按照使用与实现分离的要求，封装类的数据定义和辅助操作定义。为此，必须把可提供给外部使用的操作定义在类的接口部分（在 C++ 中这部分的存取权限为 public）。

在定义类的数据结构和操作时常常会用到其他类的实例，作为它的组成部分或参数。例如，在图 3.2 中定义 Quadrilateral 类时，引用的 4 个顶点是 point 类的实例。这些实例应当受到保护不被其他对象存取，包括同一个类的其他实例。如果其他对象必须存取这些点，应在类的接口上增加一些相应的存取操作。

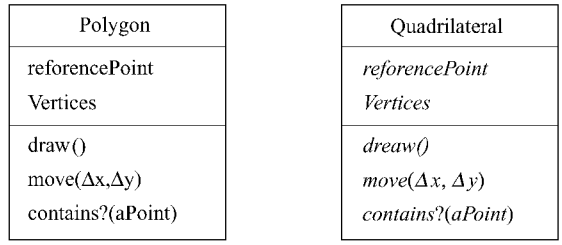
类与 Pascal 语言中的记录或 C 语言中的结构的区别在于类可提供各种级别的访问权限。此外，类中包括了数据结构和操作的定义，在语义上更完整。而在 Pascal 的记录和 C 的结构中只能对数据做结构定义，至于相关的操作需要单独定义，这将给系统的开发和修改带来很大的隐患。

3.1.4　继承

如果某几个类之间具有共性的东西（属性和行为），把它们抽取出来放在一个父类中，将各个类的特有的东西放在子类中分别描述，则可建立起子类对父类的继承。

例如，当定义 Quadrilateral 类时，如果图 3.3(a) 所示的 Polygon（多边形）类已经存在，则 Quadrilateral 类可以作为它的子类定义，如图 3.3(b) 所示。图 3.3(b) 中斜体部分表示在 Polygon 类中已经定义，并通过继承可加到 Quadrilateral 类的定义中。假如这些成员作为 Polygon 类的一部分已经过了测试，那么在 Quadrilateral 类中就无需像新写的代码那样做严格的测试了。

继承是在已有类定义的基础上定义新类的技术。例如，图 3.3 中的 Quadrilateral 类是



(a) Polygon类　　(b) Quadrilateral类（Polygon的子类）

图 3.3　使用继承的一个例子

Polygon 类的一个子类。如果限定 Quadrilateral 是矩形，还可以建立 Quadrilateral 类的子类 Rectangle，它是一种特殊的四边形，这样就形成了类继承的层次结构。

继承具有传递性，如果 Rectangle 类继承 Quadrilateral 类，Quadrilateral 类又继承 Polygon 类，则 Rectangle 类也继承了 Polygon 类。

继承从内容上可划分为 4 种。

（1）取代继承。例如“窗口”和“Windows 窗口”的关系，任何需要“窗口”的地方都可以用“Windows 窗口”来代替。此时，“窗口”可视为父类（抽象类），“Windows 窗口”作为“窗口”的子类（具体类）。

（2）内容继承。例如“四边形”与“矩形”的关系，“四边形”包括了“矩形”。

（3）受限继承。例如“鸵鸟”是一种特殊的“鸟”，它不能继承“鸟”的“会飞”的特性。这样，子类（“鸵鸟”）只能部分继承父类（“鸟”）的某些属性或操作。

（4）特化继承。例如“汽车”与“起重车”的关系。“起重车”作为“汽车”的子类可以直接使用父类的“数据＋操作”，它自己还增加了特有的“数据＋操作”，就是说，在它的接口中引入了新的能力。

3.1.5　多态性和动态绑定

对象、类、继承及消息通信表征了面向对象开发方法，但还使用了其他一些技术与它们配合，其中的两个是多态性（Polymorphism）和动态绑定（Dynamic Binding）。

多态性，也称为多形性，是指同名的函数或操作可在不同类型的对象中有各自相应的实现。例如一个“＋”操作，在整数情形执行整数加法，在浮点数情形执行浮点数加法，在字符串情形执行字符串连接。这些操作的名字相同，但实现各不相同，语言编译器根据操作对象的类型自动调用相应的实现程序。

在支持多态性的语言程序中，在函数或操作的参数表中，与形式参数对应实际参数的数据类型可能不是单一的，而是属于一个特定数据类型集合中的一个数据类型。例如，想要在屏幕上画一系列多边形，多态性允许发送消息 draw()，根据消息接收对象的类型不同，画出不同的多边形。draw() 针对的是一系列的类型（类族）而不仅仅是一个类型。

具有多态的函数或操作在运行时才根据实际的对象类型，执行相应实现程序的连接，即动态绑定。多态性的实现有两种。

（1）利用继承关系把所有数据类型当做一个抽象数据类型的子类型。

图 3.4 给出了 4 个类的继承层次。在这个继承结构中，Quadrilateral 类的接口可以响应 Polygon 类接口的所有消息。如果几个子类 Quadrilateral、Triangle 和 Rectangle 都重新定义了父类的某个函数（都用相同的函数名），如 draw()，当消息被发送到一个子类实例 p 时，在程序执行时该消息会由于子类实例的不同而被解释为不同的操作。

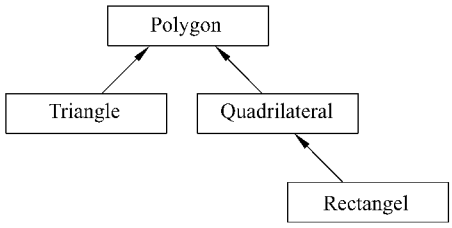


图 3.4 某些多边形类的继承层次

动态绑定保证在程序执行时实施与实例 p 连接的操作。如果 p 是 Rectangle 类的实例，则执行与 Rectangle 连接的操作；如果 p 是 Triangle 类的实例，则执行与 Triangle 连接的操作。

(2) 利用模板机制把所有可能的数据类型用一个参数化的数据类型来代替。

作为动态绑定的例子，再考虑 Polygon 类中的操作 contains(aPoint)。这个操作可以在类层次的各层重新实现，使得各个子类的特征能够得到使用。例如，假定一个 Rectangle 有某些边与屏幕的边平行，这时，检查一个点是否包含在矩形内，比检查一个点是否在一个一般的四边形内的效率要高一些。

如果我们有一个多态 Polygon 实例的表，如图 3.5 所示，并且想要看一个点 p(可能是鼠标点取的位置)是否在它们中的某一个内，那么可以遍历这个表，给表中的每个实例 P 发送消息 P. concains(p)。如果某个实例 P 响应这个消息，则动态绑定执行与实例 P 连接的操作。如果 P 属于 Rectangle 类，则执行与 Rectangle 连接的操作，而不使用与 Quadrilateral 类或 Polygon 类连接的操作。

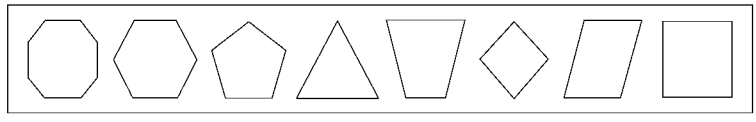


图 3.5 一系列多边形类型

3.1.6 消息通信

消息是一个对象向另一个对象传递的信息。有 4 类消息：

- 发送对象请求接收对象提供服务。
- 发送对象激活接收对象。
- 发送对象询问接收对象。
- 发送对象仅传送信息给接收对象。

消息的使用类似于函数调用，消息中指定了某一个实例、一个操作名和一个参数表(可能是空的)，如 quadrilateral1. move(15, 20) 。接收消息的实例执行消息中指定的操作，并将形式参数与参数表中相应的值结合起来。

系统功能的实现就是一组对象通过执行对象自身的操作和消息通信来完成的。如图 3.6 所示，如果想在屏幕上移动一个 Shape 实例，可调用操作 moveTo(Point)，该操作发送一个类内消息 erase()，把这个实例在原处擦去，再发送一个类内消息 draw()，并在新的位置画出它。而存取参考点 ReferencePoint 的函数则是类间消息。

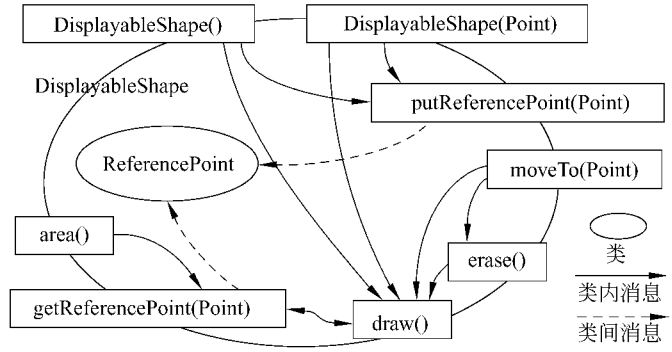


图 3.6 消息通信的例子

3.1.7 对象生存周期

对象的生存周期属于对象的实现视图。在传统的过程性程序中，首先需要定义相关的自定义数据类型，然后定义属于某种数据类型的变量，再使用这些变量参加程序的运算。在面向对象的程序中，参加程序运行的不是类，而是属于某种类的对象(实例)。与一般意义的变量不同的是，对象有它的生存周期。在程序执行过程中，它们根据需要被创建；一旦它们的使命完成不再需要它们时，对象将被销毁。从这个意义上讲，对象具有动态性。

对象的创建称为实例化。在此过程中，系统自动调用该类的构造函数，对新创建的对象初始化，即给对象的所有属性赋初值，确定该对象的初始状态。

对象具有状态和行为。对象的状态保存在对象内部的属性中，对象的行为通过相关的操作来表现。当然，对象中某些属性可以与状态无关，例如对象标识符。但对象中属性值的改变，只能通过该对象的操作。

对象之间可以并行工作，各自独立处理自己的数据，同时改变自己的状态。对象之间还具有通信能力，它们通过互相发送消息进行通信，协同完成工作流程。

各种语言执行对象销毁的方式各有不同。例如，C++ 语言在类定义中可以定义析构函数，在销毁对象时，系统自动调用这个析构函数释放该对象占有的存储空间，再撤销该对象；在 Java 语言中有无用单元自动回收的功能，只要没有从其他对象对它的引用，该对象将自动撤销，其占有的存储空间作为垃圾自动回收。此外，在一个操作中创建并使用的局部对象，在操作执行结束时将自行释放。

3.2 统一建模语言 UML 概述

UML(Unified Modeling Language)是一种定义良好、易于表达、功能强大且普遍适用的建模语言。它融入了软件工程领域的新思想、新方法和新技术。它的作用域不仅支持面向对象系统的分析与设计，还支持从需求分析开始的系统开发的全过程。

3.2.1 什么是建模

几十年系统开发的实践证明，建模是系统成功的一个基本因素。模型的实质就是对现

实的简化或抽象，它滤掉了非本质的细节，集中描绘复杂问题或结构的本质，使得问题更容易理解。模型可以帮助人们按照实际情况或按照人们所需要的样式对系统进行直观的描绘；模型允许人们细致地说明系统的结构和行为；模型给出一个指导人们构造系统的模板；模型可以通过文档的方式把人们的决策正式记载下来。

我们开发系统，无非是想用系统来刻画客观事物及其联系，解决实际问题。人脑中形成了对客观世界的正确认识之后，如果能正确地映射到系统成分上，那么系统就一定是对客观事物的正确描述和映射，因而是正确的。如果这个过程出现了问题，无论是在人脑认识阶段（需求分析），还是在人的认识到系统成分的映射阶段（系统和软件设计），都会使系统或软件开发失败。

因此，在建模过程中通常应当注意以下一些原则：

（1）选择创建的模型应能反映所要处理和解决的问题。

（2）根据观察者的角色和观察的原因，可选择用不同详细程度表示的模型。

（3）模型建造要基于现实、反映现实。

（4）为了完整地理解和表达系统，需要用一组从不同视角描述系统的模型。例如，为了完整理解面向对象系统的体系结构，应使用几个互补和连锁的视图，例如，用例图、设计视图、进程视图、实现视图和实施视图。

（5）构造模型的基本技术是抽象，应抓住与问题有关的主要特征，从抽象到具体，逐步引进问题详细可行的解决方案。

（6）不要追求绝对的真实和完美，只须从预期目标的角度看其是否充分。

（7）应当分阶段刻画问题的关键方面，略去相对次要的因素。

（8）建模语言应支持人的由模糊到清晰、由粗到细、逐渐完善的认识过程。

（9）应采用可视化图形建模语言，例如 UML。

3.2.2 UML 发展历史

20 世纪 90 年代，在软件系统业界流行着几十种面向对象的开发方法，形成百家争鸣的局面。其中著名的 3 种方法是 OMT（Rumbaugh）、Booch 和 OOSE（Jacobson）。每种方法都有自己的开发过程、表示符号和侧重点。OMT 的强项在分析，而弱项在设计；Booch 91 的强项在设计，而弱项在分析；OOSE 的强项在行为分析，而弱项在其他领域。

随着时间的推移，Booch 方法吸收了很多 OMT 和 OOSE 以及其他优秀的分析技术。Rumbaugh 方法也接受了很多 Booch 方法在设计方面的优秀技术，这就形成了所谓的 OMT-2 方法。虽然这些方法开始互相靠拢，但它们仍然保持各自的表示法。

1995 年开始，Booch 和刚转入 Rational 公司的 Rumbaugh 在公司高层的主持下将他们各自的面向对象建模方法统一为 Unified Method V0.8。一年之后，Ivar Jacobson 加入其中，共同将该方法统一为 UML 0.9。这三位杰出的专家被称为“三友（Three Amigos）”。很快 UML 获得了工业界、科技界和应用界的广泛支持，1996 年 10 月在美国就已有七百多家公司表示支持采用 UML 作为建模语言。1996 年年底，UML 已稳居面向对象技术市场 85%的份额，成为可视化建模语言事实上的工业标准。

1996 年，一个由建模专家组成的国际性组织“UML 伙伴组织”开始同“三友”一起工作，计划提议将 UML 作为 OMG 的标准建模语言。1997 年 1 月，伙伴组织向 OMG 提交了最

初的提案 UML 1.0。经过了 9 个月的紧张修订，于 1997 年 9 月提交了最终提案 UML1.1，这个提案在 1997 年 11 月被 OMG 正式采纳为对象建模标准。

UML 在完善过程中，吸收了百家之长，包含了来自很多其他方法的最佳思想，如图 3.7 所示。到 2005 年，推出了 UML 2.0。该工作分为 4 个部分：UML 2.0 超级结构、UML 2.0 基础结构、UML 2.0 对象约束语言(OCL)和 UML 2.0 图的交换。2009 年发布了 UML 2.2。图 3.8 描述了 UML 的发展历史。

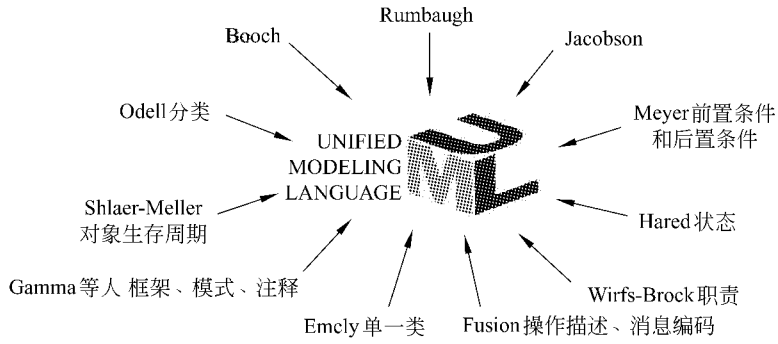


图 3.7 UML 吸收了许多面向对象方法的优秀想法

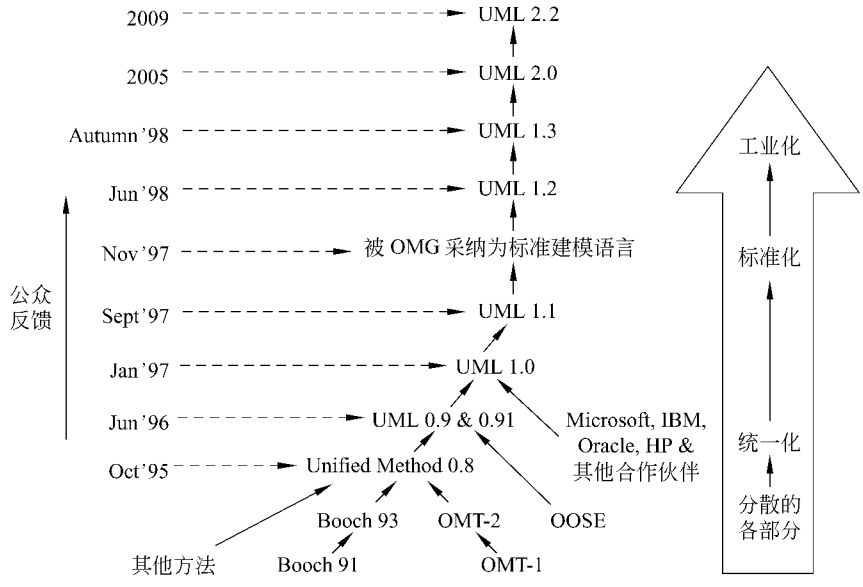


图 3.8 UML 的发展历史

3.2.3 UML 的特点

有人认为，标准建模语言 UML 的出现，是面向对象方法在 20 世纪 90 年代中期所取得的最重要的成果。其主要特点可以归结为以下 6 点。

(1) 统一标准。UML 不仅统一了 Booch、OMT 和 OOSE 等方法中的基本概念，还吸取了面向对象技术领域中其他流派的长处，其中包括非 OO 方法的影响。UML 使用的符号表示考虑了各种方法的图形表示，删掉了大量易引起混乱的、多余的和极少使用的符号，也

添加了一些新符号，提供了标准的面向对象模型元素的定义和表示法，并已经成为 OMG 的标准。

(2) 面向对象。UML 支持面向对象技术的主要概念，它提供了一批基本的表示模型元素的图形和方法，能简洁明了地表达面向对象的各种概念和模型元素。

(3) 可视化，表达能力强大。UML 是一种图形化语言，用 UML 的图形元素能清晰地表示系统的逻辑模型或实现模型。它不只是一堆图形符号，在每个图形表示符号后面，都有良好定义的语义。UML 还提供了语言的扩展机制，用户可以根据需要增加定义自己的构造型、标记值和约束等，它的强大表达能力使它可以用于各种复杂类型的软件系统的建模。

(4) 独立于过程。UML 是系统建模语言，不依赖特定的开发过程。

(5) 容易掌握使用。UML 概念明确，建模表示法简洁明了，图形结构清晰，容易掌握使用。实际上，只要着重学习 3 个方面的主要内容(UML 的基本模型元素、组织模型元素的规则、UML 语言的公共机制)，基本就了解了 UML，剩下的就是实践的问题了。

(6) 与编程语言的关系。用 Java、C++ 等编程语言可以实现一个系统。支持 UML 的一些 CASE 工具(如 Rose)可以根据 UML 所建立的系统模型自动产生 Java、C++ 等代码框架，还支持这些程序的测试及配置管理等环节的工作。

3.2.4 UML 的视图

一个软件系统往往可以从不同的角度对其进行观察，从某个角度观察到的系统就构成了系统的一个视图。每个视图都是整个系统描述的一个投影，若干个不同的视图可以完整地描述出所建造的系统。每种视图用若干幅图来描述，一幅图包含了系统的某一特殊方面的信息。UML 把视图划分为 4 个主题域：结构(Structural)、动态(Dynamic)、物理(Physical)和模型管理(Model Management)。结构主题域描述了系统中的结构成员及其相互关系，包括静态视图、设计视图和用例视图；动态主题域描述了系统的行为或其他随时间变化的行为，包括状态机视图、活动视图和交互视图；物理主题域描述了系统中的计算资源及其总体结构上的部署，包括部署视图；模型管理主题域描述层次结构中模型自身的组织(利用包来组织模型)，包括模型管理视图和剖面。

(1) 用例视图(Use Case View)　用例视图由一组用例图构成，其基本组成部件是用例、参与者和系统，用于从系统的外部视角描述参与者与系统的交互，进行系统的功能建模。用例视图的意图是列出系统中的用例和参与者，并显示哪个参与者参与了哪个用例的执行。用例的行为用动态视图，特别是用交互视图来表示。

(2) 静态视图(Static View)　静态视图用类图表示，主要描述系统中的类以及类之间的相互关系。它用于建立系统的逻辑结构模型(应用领域的视角)与物理结构模型(系统实现的视角)。在静态视图中不描述依赖于时间的系统行为。

(3) 设计视图(Design View)　设计视图由一组内部结构图、通信图和构件图实现。它用于对应用系统自身的设计层面的结构建模，例如，将设计出的体系结构表示为结构化类元(classifier)、为实现功能所需的协作，以及具有良好接口定义的构件的组装。

(4) 状态机视图(State Machine View)　状态机视图用状态机图表示。每个状态机图用于对一个类实例的整个生存周期的个体行为建模，它描述了该实例的一组状态和这些状态之间的迁移。其中，每个状态描述该实例在其生存周期中满足某种条件的一个时间段的