第 7 章

软 件 工 程

本章讨论软件开发和维护的全过程。之所以称为软件工程，是因为软件开发乃是一个工程性的过程。不过，软件工程具有许多使自身有别于其他工程科目的特点。

我们的讨论所关注的是大型的软件系统。开发这类系统时所面对的问题并非只是编写小程序所面对的问题的放大。譬如说，开发这样的大系统要求许多人工作很长一段时间，在此期间预期的系统需求可能会改变，做课题的人员也可能有变动。因此，软件工程包括了诸如人员编制、项目管理等这样一些更多与商务管理有关，而不是与计算机科学有关的论题。当然，我们的侧重点还是放在那些与计算机科学关系密切的论题上。

7.1 软件工程学科

为了有助于理解软件工程所涉及的问题，可以想象建造一个大型复杂的对象（例如一辆汽车，一幢楼房，甚至一座教堂）。对此，进行设计，监管它的构建全过程。如何来估算完成这个项目所需的时间、费用及其他资源的成本？如何把项目分割成几个部分以便于管理？如何保证建成的各部分相互间协调一致？如何使不同部分工作的人员相互沟通？如何检查进度？如何妥善应付各种细枝末节（如门把手的选用、壁饰的设计、彩色玻璃窗蓝色玻璃的可供量、柱子的强度、供暖管道系统的设计等等）？在一个大型软件系统的开发过程中，同样需要面对如此繁多的问题。

人们也许会认为，工程是一种成熟的领域，所以一定会有许多现成的工程技术可以用来解决软件工程里的这些问题。这种想法虽然有些道理，但是没有看到软件的特性和其他工程领域的特性之间有很多不同的地方。

区别之一，是关于采用通用预制件来建造系统的可能性问题。长期以来，在一些传统的工程领域，建造复杂的装备时都尽量采用各种现成的部件作为构件。例如设计一辆新车时，像引擎、收音机、空调、门锁等这类部件，就没有必要重新设计，完全可以利用老型号的设计方案。但是，对于软件，原先设计成的部件都针对特定的情况，就是说，它们的内部设计与具体的应用有关。因此再用到这一部件时，就需要重新设计。所以，复杂的软

件系统历来都是从头做起。

软件工程与传统工程间的另一个差异是关于容差的允许程度。传统工程领域开发的一个产品，只要在限定的范围内能完成任务就可以了。例如一台洗衣机只要在设定时间的2%的误差范围内，完成洗涤-过水-甩干这个周期就行了。可是对于软件的执行，它要么对，要么错。一个财会系统，即使容差小于2%也是不能接受的。

还有一个不同，在于衡量软件的属性缺乏定量系统作为**度量标准**（metrics）。一个机械设备的质量通常以平均无故障时间来测量，这是对设备耐受损耗的一种衡量。然而，软件不会损耗，所以这种衡量品质的方法在软件工程中也用不上。

软件特性不能以定量方式测量，这也是软件工程不像机械、电子工程那样，至今还未找到一个严格、坚实的立足地的主要原因之一。机电工程中问题的解决是建立在已确立无疑的物理科学基础之上，而软件工程仍然在寻找自身的根基。在某种意义上，软件工程今天的状态类似于机械工程在17世纪早期的状态，那时候牛顿和其他科学家尚未发现，原来诸如质量、加速度和力这样一些特性及其相互之间的关系也能以数学方式加以度量和描述。

因而，当今软件工程的研究在两个层面上进行：一部分研究者的工作指向开发那些用于直接应用的技术，可称为实践派；另一部分研究者，可称为理论派，则致力于探寻软件工程的基础的原理和理论，为将来构建更坚实的技术而努力。基于自身的原因，许多实践派从前开发和提倡的方法已经被其他的方法所替代，而新的方法随着时间的推移也会淘汰。而与此同时，理论派方面的进展也仍是云遮雾障。

对实践派和理论派两方面的成果的需求却是巨大的。我们这个社会已迷醉于计算机系统及其软件。经济、保健、政府管理、法律实施、交通运输，以及防务系统等等都已离不开大型软件系统。在这些系统中，可靠性始终是主要的问题。软件出错会导致灾难或事故，诸如：把满月升起误判为核攻击；纽约银行一天中损失5百万美元；“水手号18”空间探测器的失踪；辐射过量杀人、致残；还有电信通信大面积同时瘫痪。

一方面科学界在不断寻找开发更佳软件的方法，另一方面专业组织也间接地贡献着他们的努力，促进制定高标准的职业道德和行为准则，要求其成员遵守。例如，美国计算机协会（ACM）和美国电气及电子工程师协会（IEEE）已正式通过职业道德和职业行为准则，增强软件开发者的敬业精神，反对漠视个人职责。

美国计算机协会

美国计算机协会（ACM）成立于1947年，是致力于推动艺术、科学及信息技术应用的国际性科学和教育组织。其总部在纽约，有许多专业组，包括计算机体系结构、人工智能、生物医学计算、计算机与社会、计算机科学教育、计算机图像学、超文本/超媒体、操作系统、程序设计语言、模拟与仿真，以及软件工程等。ACM 的网站地址是 http://www.acm.org 。其职业道德和行为准则可在 http://www.acm.org/constitution/code.html中找到。

本章的其余部分将探讨软件工程的一些基本原则（如软件生命周期及模块化），预测软件工程发展的一些动向（如设计模式及开放源开发），以及考察面向对象范型在这个领域所起的影响。

 问题与练习

1. 为什么一个程序中代码行的数量并非是对程序复杂性的一种好的度量方法？
2. 哪一种技术可以用来确定一个软件中有多少错误？
3. 提出一种衡量软件质量的度量建议，并说明这种度量有什么缺点？
4. 再说出一种领域，像软件工程一样，仍在为寻找其科学基础而奋斗，所以它使用的技术更有赖于实践派的开发，而不是靠理论派。

7.2　软件生命周期

软件工程最基本的概念就是软件生命周期。

7.2.1　周期是个整体

图7.1表示了软件生命周期。这个图显示了一个事实，就是软件一旦开发完成，它就进入了一个既被使用又被修改的循环，这个循环将终其一生。其实这种模式在许多产品的制造中很常见。不同在于，在其他产品制造中，修改阶段会更明确地称为修理阶段或维护阶段，因为损耗的部件退出使用而进行修补。

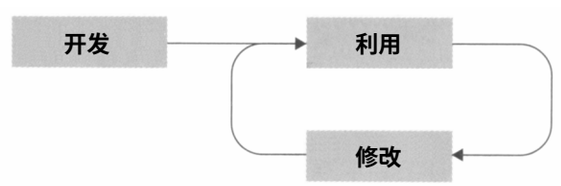

图 7.1　软件生命周期

但软件不会耗损。软件之所以进入修改阶段，或因为发现了错误；或因为软件的应用发生了改变，而要求软件作相应的改变；或因为发现上一次的修改变动在软件的其他地方带进了新问题。

无论软件因什么原因进入修改阶段，这个过程要求某人（往往不是原作者）研究底层的程序及其文档，把这个程序（至少是相关部分）读明白。不然的话，任何的改动只会带来更多的问题。即使一个程序设计精良并有良好文档，要对它达到这种理解可是一个困难的任务。实际上往往在这个阶段导致最后把软件弃之不用，其借口（太经常了）是从头开

发一个新系统要比成功修改一个现成的软件容易得多。

经验表明，在软件开发期间稍多加一点努力，会在需要对软件修改时造成很不同的后果。例如第6章讨论数据描述语句中我们已看到，在程序中如何用名字AirportAlt来替代非描述性的数值645。不难明白，一旦有必要作修改时，改变与这个名字关联的数值，要比查找并改变大量的645这个数值方便得多。因而，软件工程的大部分研究集中在软件生命周期的开发阶段，以达到这种付出-收益间的更大回报。

7.2.2 传统的开发阶段

软件生命周期开发阶段的主要步骤有分析、设计、实现及测试（见图7.2）。

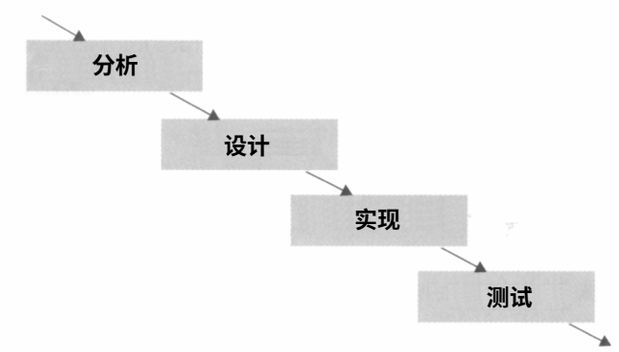

图 7.2 软件生命周期的开发阶段

分析

软件生命周期的开发阶段从分析开始——主要目的是确定所预期的系统要完成什么任务。如果系统是一个在竞争市场上销售的普通产品，那么这种分析就需要进行广泛的调研，以确定潜在用户的需求。如果系统是为专门的用户设计的，那么这个过程只要进行不太广泛的调研。

一旦确定了潜在用户的需要，就把它们汇编成新系统必须满足的一系列**需求**（requirement）。这些需求是从应用的角度来表述的，而不是以数据处理专业技术术语表达。譬如，某一条需求说数据只有被授权的人员才能访问，另一条说数据必须反映每个工作日结束时的库存状况，另一条又说数据在计算机屏幕上显示的安排形式要与当前使用的打印纸格式相符。

系统需求确定后，要转换成更为技术性的**规格说明**（specification）。例如，“数据只有被授权的人员才能访问”这样一个需求就可能变为这样一个说明：只有当键盘上键入了被认可的8位口令后，系统才能响应。或者也可能是这样的一个说明：除非经只有被授权人员才知道的程序预处理，否则数据以加密形式显示。

美国电气及电子工程师协会

美国电气及电子工程师协会（IEEE）是电机工程师、电子工程师和制造工程师的组织，建立于1963年，由美国电机工程师协会（由包括托马斯·爱迪生在内的25位电机工程师在1884年创建）与美国无线电工程师协会（建于1912年）合并而成。今天IEEE的执行中心位于新泽西州的Piscataway。协会由许多技术分会组成，如航天及电子系统协会、激光及光电子协会、机器人及自动化协会、交通运输技术协会，以及（对我们来说最重要的）计算机协会。IEEE的活动也参与了各种标准的开发、制定。特别值得指出，IEEE的努力导致了今天仍在大多数计算机上使用的数值的浮点表示形式标准。

IEEE的主页地址是http://www.ieee.org，IEEE计算机协会的主页地址是http://www.computer.org，IEEE的伦理标准在http://www.ieee.org/about/whatis/code.html。

设计

如果说分析阶段着眼于预期的系统应该做什么，那么设计阶段就着力在系统如何达到这些目标。软件系统的结构就在这个阶段建立。

对大型软件系统来说，最佳的结构是模块化的结构。这是相当确定的一个原则。正是靠了这种模块化的分解，大型系统才可能得以实现。没有这种分解，那么实现一个大型系统所需的技术细节，就完全不是人的认知能力所能及的。然而，有了模块化设计以后，只要掌握相应模块所属的细节就行。模块化设计也有利于软件以后的维护，因为修改也只要以模块为单位来进行。（例如如果要修改职工医疗补助费的计算办法，那么就只要考虑处理医疗补助的模块就可以。）

不过，模块的概念是有区别的。如果以传统的强制范型来处理设计任务，那么模块由过程组成，一个模块化设计的开发就是要做到确定系统必须执行的各种任务。相反，如果设计任务由面向对象的观点来处理，那么模块就看作对象，设计过程就变成确定系统中的实体（对象），以及这些实体如何行为。

实现

设计的实现涉及到程序的实际编写、数据文件的创建，以及数据库的开发。在这里我们看到了一个**软件分析员**（有时也称为系统分析员）的任务与一个**程序设计员**的任务之间的不同。软件分析员参与整个开发过程，任务的重点在分析和设计阶段。程序设计员的任务主要在实现阶段。最狭义地说，程序员负责编写程序，以实现由软件分析员做出的说明。

作了以上这样一个区分，我们还要指出，其实在计算业界并没有一个总权威来操控术语的使用。许多有着软件分析员头衔的人实际上是程序员，而许多有程序员（或是高级程

序员）头衔的人，从广义上讲实际是软件分析员。我们很快就会看到，术语上的这种模糊是因为今天软件开发过程的各个步骤实际上往往是交叉重叠的。

测试

因为系统的每个模块通常都是在实现的同时也作了测试，所以测试与实现联系最紧。确实，在一个精心设计的系统中每一个模块都能独立于其他模块来进行测试。为了模拟目标模块与系统其余模块间的相互作用，别的模块只需采用简化版本（有时称为插桩模块）就可以。当然，只要各种不同的模块都测试完成并组合成功，对部件的这种测试就成为了对整个系统的测试。

遗憾的是，一个系统的测试和排错极难顺利完成。经验表明，大型的软件系统即使经过严格的测试，还是有可能包含大量的错误。其中许多错误可能在系统的一生使用中都不会被检测出来，而另外一些错误可能会造成重大故障。排除这些错误也是软件工程的目标之一。错误不灭，探究不止。

7.2.3 当今趋势

软件工程早期的方法强调以一个严格的顺序，按分析、设计、实现和测试分阶段进行。考虑是，在开发大型软件系统时顾及到做更改有太多的风险。因此，软件工程师坚持，一定先完成对系统的整个分析，再开始设计。同样，设计完成后再开始实现。结果就是开发过程称之为**瀑布模型**（waterfall model），反映了开发过程只按一个方向进行。

你会注意到软件开发的分析、设计、实现及测试四阶段与Polya确立的问题求解四阶段（4.3节）有相似之处。毕竟，开发一个大型软件也就是解决一个问题。另一方面，软件开发的传统瀑布式方法又全然不同于“自由变向”式的“摸着石子过河”这种想象型的问题求解过程。瀑布方法寻求一种高度结构化的环境，开发过程循序进行；而想象型问题求解方式，寻求一种非结构化的环境，只要跟随灵感发挥，可以随意放弃原先的思路，无须理由。

近年来，软件工程技术开始反映出这种潜在的矛盾，软件开发出现一种**渐进模型**（incremental model）。按照这种模型，所需的软件系统以一种递进的模式来构建——产品最先以一种简化版本呈现，功能有限。一旦这个版本通过测试或经未来用户的评估，更多的功能就以递增的方式增加进去，再通过测试，直到系统完全成功。例如为学校开发一个学生成绩档案系统，那么一开始只要做进能查看一小部分学生记录样本的东西就可以。等这个版本能够工作，再加进更多的功能，如可以增添和修改记录的功能。就这样以渐进的方式把更多的功能添加进去。

渐进模型反映出软件开发采用**原型方法**（prototyping）的一种趋势，就是把所要的系统先做成一个非完整的版本，称为**原型**（prototype），并加以评估。在渐进模型中，这些原型演化成最后的完整系统，这个过程称为**进化式原型法**（evolutionary prototyping）。在

另外一些情况中，原型可能会弃而不用，以使最后设计有全新的实现。这种方法称为**抛弃式原型法**（throwaway prototyping）。快速原型方法就属于这种抛弃式原型，在开发的早期就很快做成一个所需的系统的简单实例。这个原型可能只有不多几个屏幕图像构成，只是演示系统如何与用户交互作用，它会有哪些功能。其目的并非在于做成一个真正的工作版本，而是用来理清项目所涉及的各方相互沟通的一个演示工具。例如，快速原型有利于在分析阶段就敲定系统的需求，也便于向潜在客户进行销售推介。

今天，软件工程方法学涉及广泛的观念范畴。其一端是传统的瀑布模型，以工作在各自办公室的管理者和程序员的想象力为特征，每个都完成整个软件开发任务明确分工的一部分。另一端的模型（戏称为**极限编程**，extreme programming）乃是十来个成员为一个集体，在共同的工作场所自由地交换想法、相互协助，通过每天经历的设计、实现及测试的轮转，以递进的方式开发一个软件包。

软件工程的另一个发展乃是软件开发过程本身利用计算机技术，这就导致了所谓**计算机辅助软件工程**（computer-aided software engineering，CASE）。这些计算机化的系统称为CASE工具，包含有项目计划工具（帮助成本估算、进度安排及人员分配）、项目管理工具（帮助监控开发进度）、文档工具（帮助书写和组织文档）、原型和模拟工具（帮助原型开发）、界面设计工具（帮助图形用户界面开发），以及编程工具（帮助程序编写和查错）。这些工具有些不过就是其他应用中的文字处理器、电子报表、e-mail这样一些软件。而有些就相当复杂，专门为软件工程环境所设计。例如，有些CASE工具有代码生成器，只要给定系统某个部分的描述说明，它就能产生实现这部分的高级语言程序。

问题与练习

1. 系统需求和系统说明有什么区别？
2. 简要说明软件生命周期开发阶段的四个步骤（分析、设计、实现及测试）。
3. 简要说明软件开发传统瀑布模型与较新的原型模型之间的区别。

7.3 模块化

7.2节中有一句关键性的陈述：为了修改一个软件，就必须读懂这个程序，至少得弄明白相关的那部分。即使对小的程序，要达到这样的理解也往往是相当困难；更何况对于大型的软件系统，如果它没有模块化，那几乎是不可能的。**模块化**（modularity）就是把软件分割成易于处理的几个较小的部分，每个部分只执行整个任务的一部分。

7.3.1 模块实现

模块化可以以不同的方式达到。第5章和第6章中我们已经看到如何按照过程来组成模

块，并作为积木块（抽象工具）用来构建更大的系统。第6章讲到面向对象范型时，也遇到过模块化。那里模块是采取对象的形式，每个对象及其内部结构都与别的对象的内容无关。事实上，正是这种内在的模块结构导致了软件开发中面向对象方法的流行。

在利用过程组成模块结构的情况中，**结构图**（structure chart）是用来表示这种结构的一种传统工具。在图中每个模块（过程）用一个矩形表示，模块之间的关系用连接矩形的箭头表示。图7.3中的结构图表示了用于一个因特网“邮购”业务活动的模块结构，客户访问公司网站，浏览目录，并下订单。图中显示了名为OverseeWebSite的模块使用ProcessIncomingCustomer、OverseeBrowsing及ProcessOrder三个模块作为抽象工具来完成其任务。更确切一点说，当客户访问网站时，过程OverseeWebSite调用过程ProcessIncomingCustomer执行获得客户身份信息所需的步骤。然后调用过程OverseeBrowsing让客户阅读目录，并做出订单。一旦产生了订单，OverseeWebSite调用过程ProcessOrder来监控出货及付款活动。

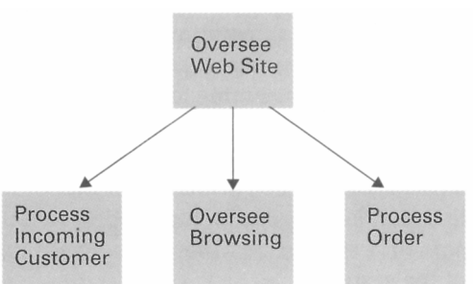

图 7.3　用于简单因特网“邮购”业务的一个结构图

过程性组织的软件系统用结构图来表示，而面向对象系统的结构常常按照对象的类及这些类之间的关系，用**类图**（class diagram）来表示。图7.4就是上述因特网邮购系统的一个简单类图。图中表示系统由Customer、Catalogue及OrderForm三类对象构成，它们由称为CurrentlyBrowsing及CurrentlyConstructing的两个关系所联系。其基本思路是，Customer类对象包含关于一个具体客户的数据及过程，Catalogue类对象包含向客户显示公司产品所需的数据及过程，OrderForm类对象包含为跟踪客户要订购的物品所需的数据及过程。关系CurrentlyBrowsing表示客户与目录间的联系（客户会浏览公司的目录），关系CurrentlyConstructing表示客户与其订单间的联系（一个客户会建立一个订单）。关系CurrentlyConstructing的连接线两端的数字表示只能有一个客户与一个订单关联，相反，一个订单也只能与一个客户关联。不同于一对一的关系，关系CurrentlyBrowsing是多对一的，它上面的数字表明许多客户可以同时浏览同一个目录。（7.5节讨论实体关系图时，我们再会回到这些概念。）

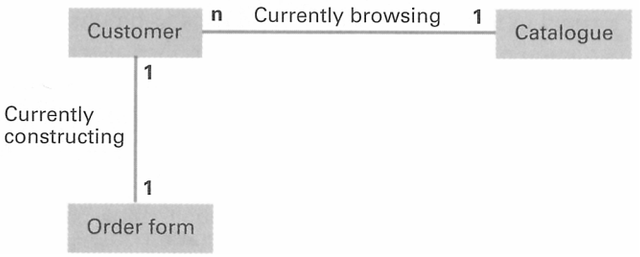

图 7.4 一个用于简单因特网“邮购”业务的类图

为了表述面向对象设计，近年来在建立标准的标记体系方面有了很大进展，最突出的例子是**统一建模语言**（Unified Modeling Language，UML），这是一个能表示各种面向对象概念的系统。图7.4的类图中使用的记号就是按照UML的约定。

7.3.2 耦合

前面介绍了模块化，强调说这是让人们获得易于应付的软件开发的一条途径。主要的想法是，今后的修改只要涉及不多几个模块，只要把修改过程的注意力放在系统的有关部分就行。当然，这里有个前提，就是一个模块中的修改不会无意中影响到系统的其他模块。因此，模块之间做到最大可能的独立，是模块化系统设计的一个目标。但是，有一件事不利于实现这个目标，就是为了构成一个协调一致的系统，模块之间一定会有联系。这种联系称为模块间**耦合**（coupling）。所以，最大化独立的目标取决于最小化的耦合。

实际上，模块间耦合的出现有几种形式。一种是**控制耦合**，出现在一个模块传递控制给另一个模块时，例如有关过程的调用/返回关系中出现的耦合关系。另一种形式是**数据耦**

现实中的软件工程

以下情节是现实的软件工程中常见的例子。XYZ公司雇用一家软件工程公司开发、安装了一套全公司的集成软件系统，以满足公司的数据处理所需。作为系统的一部分，XYZ公司又做了一套PC网络用来给员工访问这个全公司系统。这样，每个员工在办公桌上有了一台PC。很快这些PC不但用来访问新的数据管理系统，也成为员工各自可改制的工具用来提高自己的工作产出。例如，某个员工开发一个电子报表程序提高工作效率。不幸，这样一些自制的应用可能设计并不完善，也没经过通盘测试，还可能隐藏了一些全然不了解的特性。随着时间推移，这些自制的应用程序融合进了公司内部事务中。而同时，开发这些应用的员工也会因升迁、调任或跳槽，而不在原位，可是使用这些程序的其他同事却并不懂得这些程序。结果，一个以精心设计、协调一致开局的系统变成了一个设计疏漏、文档缺损、出错频繁的拼凑物。

合，这是指模块间的数据共享。

图7.3的结构图已经反映出过程方法的控制耦合。习惯上，结构图中的数据耦合用附加的箭头表示，如图7.5所示。这个图显示了当请求服务时传递给一个模块的数据项，以及当请求的任务完成时传回给原模块的数据项。具体来讲，图中显示，模块OverseeWebSite会收到由过程ProcessIncomingCustomer传来的有关每个客户的信息，并把这个信息传递至过程OverseeBrowsing。而且这个过程会报告产生的订单，订单再被送到过程ProcessOrder。

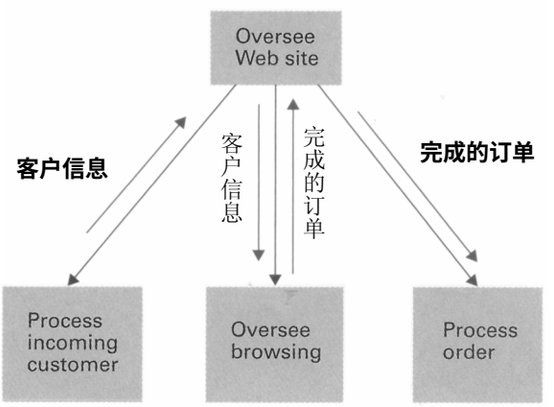

图 7.5　显示数据耦合的结构图

数据耦合的最小化是面向对象方法的主要优点之一。毕竟，对象的一个引人入胜的概念就是把处理一个特定数据项的例程都收集进一个模块中。于是，一个面向对象系统中大部分的模块间耦合均采取对象间通信的形式，通常把它们作为控制耦合来处理。这样，请求一个对象执行一个任务，本质上就是请求在对象中执行一个方法（过程）。这种请求所导致的传递给对象的控制，类似于强制式方法中传递给一个过程的控制。

在面向对象设计中，表示对象间通信的一种方法，就是在一个类图中增加信息，构成一个**协作图**（collaboration diagram）。协作图本质上也是一种类图，它还表示出系统中各种不同的对象怎样相互协作。图7.6就是那个因特网“邮购”业务例子的简单协作图（使用与UML一致的记号）。图中显示，一个Customer类对象向OrderForm对象发送一个消息，告诉它要在订单中增加一项，而一个Catalogue类对象会发送一个有具体商品信息的消息给Customer对象。

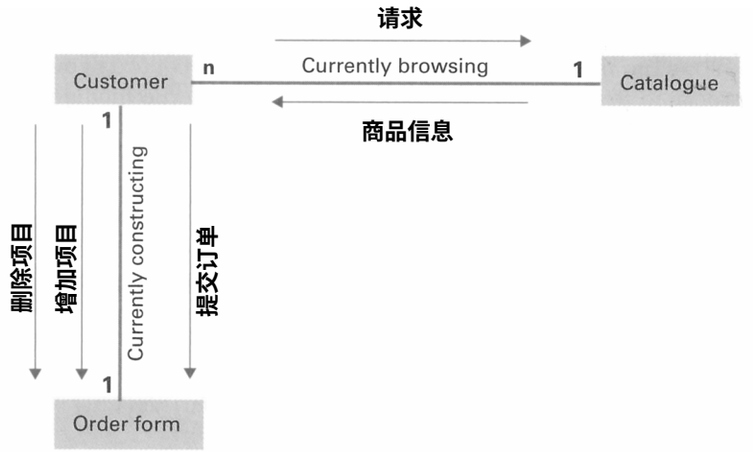

图 7.6　简单因特网“ 邮购 ”业务的协作图

不管涉及哪类耦合，程序员应当力求做到在最后写就的程序中，耦合是显然明了的。被掩盖的耦合称为**隐性耦合**（implicit coupling），常是许多软件错误的根源。一种常见的隐性耦合形式是因使用**全局数据**而造成——全局数据元素理当可为全系统中的模块所用。这与局部数据元素不同，局部数据元素只能在某一个模块中使用，除非明确地传递给另一个模块。大多数高级语言提供实现全局数据和局部数据两者的方法。全局数据的问题在于，它的改变往往是由于执行一个模块产生的副作用所造成，而并非有意地要这样做。（软件行话中**副作用**，是指软件所实行的一个行为，它并非是从写就的程序中能明显地看到。）转而，阅读程序的人除非读通模块的内部细节，否则他也不会意识到一个模块会改变全局数据。这就降低了模块作为一种抽象工具的使用价值。（所以要在程序中使用注解语句就是一个例子，可以用它来帮助读者读懂程序。）

7.3.3　内聚性

就像模块间的耦合应最小化，同样重要的是，每个模块中的内部汇集应当最大化。术语**内聚性**（cohesion）就是指这种内部的汇集，或者说，一个模块内部各部分的关联程度。为充分意识到内聚性的重要，不但要看一个系统起初的开发，更应考虑整个软件生命周期。如果一定要在一个模块中作改变，那么存在于模块中各种不同的活动会搞乱原本一个简单的过程。所以，软件设计者除了要寻求低的模块间耦合，还应力求达到高的模块中内聚性。

弱形式的内聚性称为**逻辑内聚性**。模块中的这种内聚性是因为其内部元素执行逻辑上相似的活动所引起。例如，一个模块执行系统所有与外界的通信。粘合这样一个模块的“ 胶水 ”乃是模块中所有处理通信的活动。但是通信的主题却各不相同，有的是获取数据，有的是报告错误。

较强形式的内聚性称为**功能内聚性**。这表示模块中所有部分都集中围绕执行某一个活动。图7.3中的模块OverseeBrowsing，如果它包含了与客户通信、获取及显示产品信息、构建客户订单诸细节，那么它在功能上并不内聚。然而，如果这些细节孤立于其他模块，并用作为抽象工具，那么，模块OverseeBrowsing的每一步都能够呈现在监控浏览过程的整个情况中，而使得模块功能上更内聚。

在面向对象设计中，因为一个对象中的方法常松散地执行相关的活动——惟一的共同约束是它们都是由同一个对象执行的活动，所以全部对象通常都只是逻辑上内聚。例如在我们因特网业务的例子中，每个订单对象可能都会包含一个方法，在订单上增添项目、从订单上删减项目，以及提交订单。所以这样一个对象只会构成一个逻辑上内聚的模块。但是，软件设计者应当力求做到使一个对象中的每个方法都在功能上内聚。也就是说，即使一个对象在整体上只是逻辑内聚，其中的每一个方法也应只执行一个功能上内聚的任务（见图7.7）。

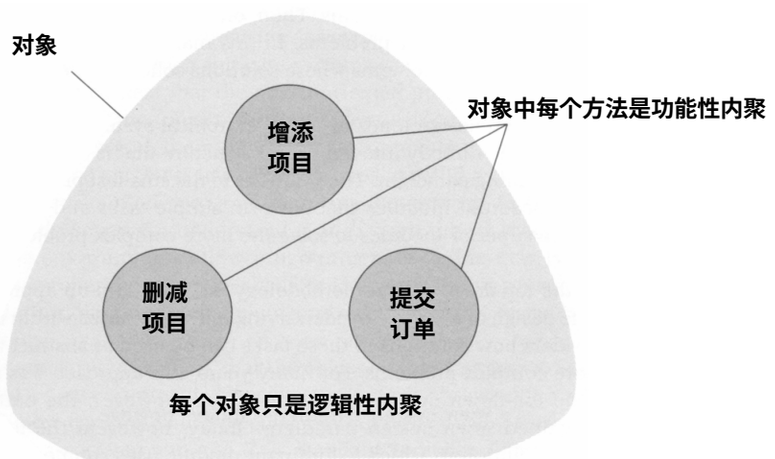

图 7.7　简单因特网“ 邮购 ”业务例子中表示订单的一个对象中的逻辑内聚性和功能内聚性

问题与练习

1. 以章、节及整体之间的耦合程度来说，小说与百科全书有什么不同？内聚性呢？
2. 一场体育比赛常分为几个单元，例如棒球比赛分成局，网球比赛也分成局。分析一下这种“ 模块 ”间的耦合。在哪种意义上这些单元是内聚的？
3. 内聚性最大化与耦合性最小化的目标是否一致？也就是说，随着内聚度增大，耦合度是否必然趋于减小？
4. 扩展图7.6中的协作图，加进Customer类对象与Catalogue类对象之间应传递的其他消息。

7.4 设计方法论

软件系统设计方法的开发是软件工程的主要探求目标之一。我们已经说过这个领域的一些话题，如瀑布模型及原型方法。本节将讨论另外一些话题，以及当前研究的若干方向。

7.4.1 自顶向下和自底向上

与系统设计有关的最有名的策略也许是自顶向下方法。这个方法的要点在于：一个复杂的问题的解决不能企图一步即就，而应当先把问题分解成较小的、易于处理的子问题，然后可以把这些子问题再分解成更小的问题。以这样的方式，一个复杂的问题就变成为一组较简单问题的集合，这些简单问题的答案合在一起解决了原先那个复杂的问题。

自顶向下设计的结果导致一个逐步求精的分级系统，常常可直接转换为与强制性程序设计方法相协调的模块化设计。分级系统中最小问题的解决就成为过程模块，每个过程模块执行一个简单的任务，并由较高层的模块作为抽象工具，用来解决系统中比较复杂的问题。

与自顶向下方法相反的是自底向上方法。在这种方法中，系统的设计先从确定系统各个具体任务开始，接着再考虑解决这些任务的方法如何能作为抽象工具用来解决比较复杂的问题。许多年里这种方法被认为不如自顶向下方法。然而现在自底向上方法又得到了推崇。这种变化的一个理由是，自顶向下方法寻求这样一种解决办法，就是其中一个主宰的模块利用一些子模块，而每个子模块又依赖子子模块，等等。但是对许多系统来说，最佳设计并非分级类型。例如，对于客户机/服务器模式，或者涉及平行处理应用的系统这些情况来说，更好的系统设计的解决办法也许是由两个或多个对等的相互作用的模块来构成的系统。

自底向上设计受到更大关注的另一个原因在于，软件工程开发方法当前的一个趋势是，利用预制、现成的构件来构建复杂的软件系统。而自底向上设计更加符合这个目标。

7.4.2 设计模式

为了寻找能利用现成构件来构建软件的途径，软件工程师从建筑领域汲取了灵感。其中值得注意的是Christopher Alexander等人著的一本书《A Pattern Language》（模式语言），这本书描绘了社区设计的一组模式。每一个模式都包括有问题的描述及推荐的解决方案。问题都具有普遍意义，建议的方案也是通用性的，它们都照顾到问题的普遍性质，而不是仅仅针对某个具体的案例。

例如，书中有一个模式称为“静隅”（“Quiet Backs”），是要在商业区设计一个“恬静之处”，能够暂避闹市的喧哗，获得片刻恬静的功效。有些案例设计成所有的楼房都朝向一条主干道，楼后小街就成了安静地。另一些案例中，通过公园、小溪，乃至教堂来获

得“静隅”的效果。

对我们的讨论来说，要点在于：Alexander的工作致力于确定普遍性的问题，提出解决的样板。今天，许多软件工程师在设计大型软件系统时正是采用同样的方法。具体说，就是利用设计模式作为提供通用构建块的方法来构建软件系统。

正是在这种背景下，面向对象编程方法显得尤其有用。因为对象构成了完备、自含的单元，明确定义了与其环境的接口，所以它们是构建块的最佳候选。一个对象一旦设计成能完成某个任务，它就可以在任何要求提供这种服务的程序中用来完成该任务。并且，读过6.5节的读者会懂得，通过继承，面向对象编程环境使程序员能够定义新的对象类型，扩展了原先定义的对象的特性。从这个意义上来说，预制的对象定义可以方便地改制成符合一个特定应用的需要。

所以一点也不奇怪，面向对象编程语言C++、Java及C#伴随有一组预制“模板”，程序员可以方便地利用它们来构成执行一定任务的对象的实现。具体地说，C++有C++ Standard Template Library（C++标准模板库）；Sun Microsystem公司提供的Java编程环境有Java Application Programmer Interface（Java应用程序员接口，API）；C#程序员可以享用Microsoft提供的.Net Framework Class Library（.Net框架类库）。所有这些集合本质上都是设计模式的集合，最后都能用来作为对象。

利用设计模式开发软件的另一新兴领域是所谓**部件建构**（component architecture）。这种方法是用称为部件的单元来构建软件，而部件可以由一个以上的对象所构成。在部件建构模型中，传统的程序员被“部件装配员”所替代，由装配员把部件装配成软件。在图形界面中常常用图标来表示部件。部件装配员不必参与部件内部的编程，他只要在预制的部件集合里选用相关的部件，再把它们“连”成所要的功能。

Sun Microsystems和Microsoft两家公司都为部件装配员提供了构建软件的工具。Sun公司产品中，部件称为Java Beans，这与Java编程语言赖以命名的Java文体相一致。Microsoft的方法则是包含在称为.NET（念成“dot-NET”）的软件开发环境里。

尽管软件工程领域对设计模式和部件建构表示出兴奋，我们可得指出，Alexander本人却对他的模式在建筑上的成果并不满意。简单说来，他认为借助模式设计的系统缺乏个性。从上世纪80年代早期以来，他就致力于寻取这种难以捉摸品质的途径。不过软件工程师们倒不以为然，他们认为软件开发的目标是准确和效率，不牵连诸如漂亮、个性这些品质。所以他们坚持认为，设计模式在软件工程领域一定会证明比在建筑领域更成功。而至今取得的成果确实有力地支持了这种预期。

7.4.3 开放源开发技术

文化上的一些重要的进展有时倒并不是发生在主流文化中，它们会逐步演化，变为明确的信条，作为一个重要贡献最终被更传统的社会体制所承认。例如，爵士乐现在公认是音乐界的宝贵财富，许多现代艺术起先都被斥为异端，说不定高尔夫运动有朝一日也会成

为奥林匹克一个赛项。

软件开发中也有这样的情况。一种被计算机发烧友们采用多年的即兴开发技术，现在也成为了正统的软件开发技术。这种技术实际上就是一种进化式原型开发技术——不同在于这种原型开发是在一个公开的平台上进行。它就是所谓**开放源开发技术**（open-source development），虽然兴许叫它为公开的进化式原型技术更好。

某种意义上，开放源开发技术是beta测试（7.6节）的一种扩充。只不过传统的beta测试只能报告问题，而这里的情况是，“beta测试者”可以对软件作修改，并报告这些修改。整个的过程是这样的：原作者编写了一个软件最初版本（通常都是为了自用），并把其源代码及文档张贴到因特网，别人就可下载使用。因为其他用户得到源代码及文档，所以他们也能根据自己的需要对这个软件进行修改，或增强功能，或改正所发现的错误。他们并且把改动情况报告给原作者，原作者把这些改动合并进张贴的版本。这个扩充的版本又可作进一步的修改。实际情况中，一个软件包可以在一个星期里经历几次演化、扩充。

也许有人会认为，开放源开发的参与者并无组织，也无报酬，所以他们不会全力投入大型高质量的软件系统开发。但事实证明这种臆想不对。Linux操作系统就是一个最好例证，它是现在公认的最可信赖的操作系统之一。实际上可以认为Linux的原创者Linus Torvald是开放源开发技术之父，他正是通过这种技术引导了Linux的早期开发。

开放源开发存在一个问题，它的最终产品所有权难以由一个单独实体来持有。这就使它还没有成为更流行的软件开发技术。终究，开放源开发要求软件源代码广泛传播，就难以实施法定管理。

另一个问题是，由商界培养出来的管理者们显然难以接受这种“撒手不管”的管理风格，而这恰恰是开放源开发成功所必须的。开放源开发看来靠的是一种自由发挥的氛围，对创造力的表现欲和成就感激励每个参与者的热情。但是管理者总倾向于行使某种控制，给这种热情泼冷水。例如，管理者为了保有所有权，会保留软件的关键部分。而开放源开发得靠提供完整的软件系统，给参与者下载、使用，并最终根据自己的需要修改。有时候，管理者为避免出错的难堪会推后软件新版本的发表，而实际上，发现错误正是刺激许多开放源贡献者兴趣的诱因。

不管开放源开发是否能成为商品软件开发的流行模式，必须认识到它确实是一个现存的、可行的方法，值得软件工程领域研究者重视。要想更多地知道什么因素造成不同场合下开放源开发的成败得失，必须对此领域有更加全面的深入了解。

问题与练习

1. 玩拼图游戏时，把碎片倒在台上，你会按自顶向下方法来拼吗？如果先看了整体图的样子，你会改变主意吗？
2. 在早先一些章节里讲过的软件结构中找出一些可以作为设计模式的东西。
3. 设计模式在软件工程的整个工作过程中起什么作用？

4. 传统程序员与部件装配员之间有什么区别？
5. 传统的进化式原型方法与开放源开发方法之间有什么不同？

7.5 行业工具

软件工程创造了各种不同的标记系统用于系统分析和设计过程。前面我们已经讲到过结构图、类图及协作图等。此外还有**数据流程图**（dataflow diagram），这是系统中数据流动路径的一种图形表示。它确定了一个系统中数据的源、目的地及处理点。图中不同的符号有专门的意思：箭头表示数据路径，粗线表示数据源和数据目的地，圆（许多软件工程师昵称为泡泡）表示数据的加工处，两条平行直线表示数据存储。符号都由所表示的实体的名称标记。图7.8就是前面那个简单因特网“邮购”业务的数据流程图。

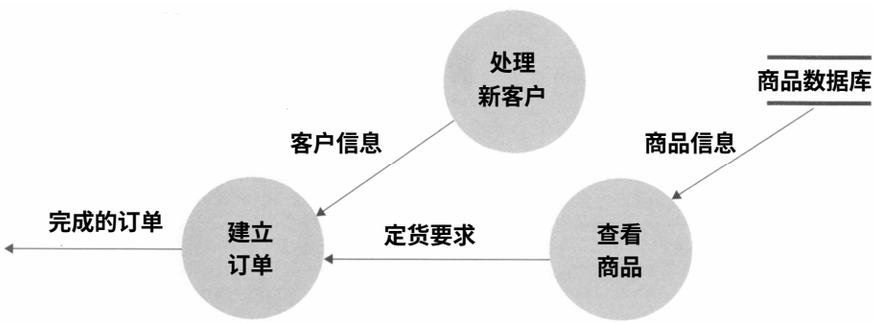

图 7.8 一个简单因特网“邮购”业务的数据流程图

软件开发的数据流方法起源于强制性编程范型的来龙去脉中。其思路是，沿着预期的系统中数据路径，可以发现数据在哪些地方进行合并、拆分或修改。因为在系统的这些地方需要计算活动，所以这些活动或活动组应当形成过程化模块。结果，对数据流的注意，帮助我们发现了系统的模块化结构。

虽然数据流分析源于强制编程方式，但是它对面向对象环境同样有用。具体说，确定系统中的数据项有助于确定对象，确定对数据的改变有助于确定这些对象执行的活动。

实体-关系图（entity-relationship diagram）是分析、设计软件系统的另一种工具。系统中的信息项（实体）及这些信息项之间的关系由图形表示。我们举一个有关大学教师、学生及课程信息的软件系统作例子，来看其中一部分的实体关系图。

首先确定涉及的数据项。实体Professor表示学校里的教授，实体Student表示学生，实体Class表示课程。Professor实体的每个具体值相应有姓名、地址、教师证号、工资等；Student实体的每个具体值相应有姓名、地址、学生证号、各科成绩的平均积分点等；Class实体的

系统设计惨剧

对系统良好设计的严格要求可以从一个例证中得到说明，这就是Therac-25（上世纪80年代中期医学界使用的一台计算机化电子加速器放射治疗仪）所遇到的问题。该机器的设计缺陷导致了六宗放射过量事件——其中三宗导致人员死亡。设计的缺陷包括：（1）机器界面的蹩脚设计，使操作员在机器的放射量还没有调整到正常值时就能开动。（2）硬件和软件之间的协调配合差，导致一些安全功能缺损。

更加近期的一些例子，有因设计不当引起的大面积停电、电话服务中断、金融业务重大出错、空间探测器失踪，以及因特网瘫痪等。要了解更多这方面问题，可查阅Risks Forum（风险论坛，网页地址：http://catless.ncl.as.uk/Risks）。

每个具体值相应有课程代码（如历史101）、学年学期、教室、日期时间等。

确定了系统中的实体，就可以来考虑实体间的关系。我们首先注意到，每个教授讲授课程，每个学生修习课程，所以确定实体Professor和Class间的关系为Teaches，实体Student和Class间的关系为Attends。（注意实体用名词命名，关系用动词命名。）

为了表示这些实体和关系，我们使用图7.9所示的实体-关系图。图中的实体用矩形，关系用菱形表示。图中显示了教授与课程间用关系Teaches关联，学生与课程间用关系Attends关联。

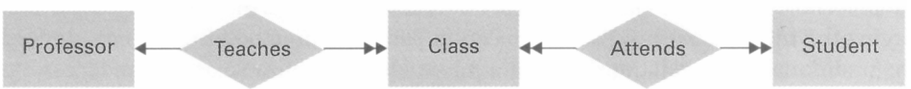

图 7.9 一个实体-关系图

然而，这个例子中的两种关系有不同的结构。Professor与Class间的关系是**一对多关系**，每个教授教几门课，而每门课只有一位教授讲授。相反，Student与Class间的关系是**多对多关系**，因为每个学生可修几门课，而每门课由几个学生选修。在作者与书籍的关系中也可以看到一对多和多对多的例子。作者与小说间传统上是一对多关系，因为一个作家可能写几本小说，而每部小说只有一个作者。不过教科书作者与教科书之间是一种多对多关系，因为一本教科书往往由几个作者合著。

还有一类是**一对一关系**。图7.6就是一个例子，其中每个Customer只与一个OrderForm关联，而每个OrderForm也只与一个Customer关联。所以总结说来，在实体之间可能出现的关系有三种基本类型，就是一对一、一对多（或多对一，按观察点不同而异）及多对多，如图7.10所示。