第 12 章　面向对象实现

面向对象程序的质量基本上由面向对象设计的质量决定，但是，所采用的程序语言的特点和程序设计风格也将对程序的可靠性、可重用性及可维护性产生深远影响。

目前，软件测试仍然是保证软件可靠性的主要措施，对于面向对象的软件来说，情况也是如此。面向对象测试的目标，也是用尽可能低的测试成本发现尽可能多的软件错误。但是，面向对象程序中特有的封装、继承和多态等机制，也给面向对象测试带来一些新特点，增加了测试和调试的难度。必须在实践中努力探索适合于面向对象软件的更有效的测试方法。

12.1 程序设计语言

12.1.1 面向对象语言的优点

面向对象设计的结果既可以用面向对象语言、也可以用非面向对象语言实现。使用面向对象语言时，由于语言本身充分支持面向对象概念的实现，因此，编译程序可以自动把面向对象概念映射到目标程序中。使用非面向对象语言编写面向对象程序，则必须由程序员自己把面向对象概念映射到目标程序中。例如，C 语言并不直接支持类或对象的概念，程序员只
能在结构(struct)中定义变量和相应的函数(事实上，不能直接在结构中定义函数而是要利用指针间接定义)。所有非面向对象语言都不支持一般特殊结构的实现，使用这类语言编程时要么完全回避继承的概念，要么在声明特殊化类时，把对一般化类的引用嵌套在它里面。

到底应该选用面向对象语言还是非面向对象语言，关键不在于语言功能强弱。从原理上说，使用任何一种通用语言都可以实现面向对象概念。当然，使用面向对象语言，实现面向对象概念，远比使用非面向对象语言方便，但是，方便性也并不是决定选择何种语言的关键因素。选择编程语言的关键因素，是语言的一致的表达能力、可重用性及可维护性。从面向对象观点看来，能够更完整、更准确地表达问题域语义的面向对象语言的语法是非常重要的，因为这会带来下述几个重要优点：

1.一致的表示方法

从前面章节的讲述中可以知道，面向对象开发基于不随时间变化的、一致的表示方法。这种表示方法应该从问题域到 OOA，从 OOA 到 OOD，最后从 OOD 到面向对象编程(OOP)，始终稳定不变。一致的表示方法既有利于在软件开发过程中始终使用统一的概念，也有利于维护人员理解软件的各种配置成分。

2.可重用性

为了能带来可观的商业利益，必须在更广泛的范围中运用重用机制，而不是仅仅在程序设计这个层次上进行重用。因此，在 OOA，OOD 直到 OOP 中都显式地表示问题域语义，其

意义是十分深远的。随着时间的推移，软件开发组织既可能重用它在某个问题域内的OOA结果，也可能重用相应的OOD和OOP结果。

3.可维护性

尽管人们反复强调保持文档与源程序一致的必要性，但是，在实际工作中很难做到交付两类不同的文档，并使它们保持彼此完全一致。特别是考虑到进度、预算、能力和人员等限制因素时，做到两类文档完全一致几乎是不可能的。因此，维护人员最终面对的往往只有源程序本身。

以ATM系统为例，说明在程序内部表达问题域语义对维护工作的意义。假设在维护该系统时没有合适的文档资料可供参阅，于是维护人员人工浏览程序或使用软件工具扫描程序，记下或打印出程序显式陈述的问题域语义，维护人员看到"ATM"、“账户”、“现金兑换卡”等，这对维护人员理解所要维护的软件将有很大帮助。

因此，在选择编程语言时，应该考虑的首要因素，是在供选择的语言中哪个语言能最好地表达问题域语义。一般说来，应该尽量选用面向对象语言来实现面向对象分析、设计的结果。

12.1.2 面向对象语言的技术特点

面向对象语言的形成借鉴了历史上许多程序语言的特点，从中吸取了丰富的营养。当今的面向对象语言，从20世纪50年代诞生的LISP语言中引进了动态联编的概念和交互式开发环境的思想，从20世纪60年代推出的SIMULA语言中引进了类的概念和继承机制，此外，还受到20世纪70年代末期开发的Modula_2语言和Ada语言中数据抽象机制的影响。

20世纪80年代以来，面向对象语言像雨后春笋一样大量涌现，形成了两大类面向对象语言。一类是纯面向对象语言，如Smalltalk和Eiffel等语言。另一类是混合型面向对象语言，也就是在过程语言的基础上增加面向对象机制，如C++等语言。

一般说来，纯面向对象语言着重支持面向对象方法研究和快速原型的实现，而混合型面向对象语言的目标则是提高运行速度和使传统程序员容易接受面向对象思想。成熟的面向对象语言通常都提供丰富的类库和强有力的开发环境。

下面介绍在选择面向对象语言时应该着重考察的一些技术特点。

1.支持类与对象概念的机制

所有面向对象语言都允许用户动态创建对象，并且可以用指针引用动态创建的对象。允许动态创建对象，就意味着系统必须处理内存管理问题，如果不及时释放不再需要的对象所占用的内存，动态存储分配就有可能耗尽内存。

有两种管理内存的方法，一种是由语言的运行机制自动管理内存，即提供自动回收“垃圾的机制；另一种是由程序员编写释放内存的代码。自动管理内存不仅方便而且安全，但是必须采用先进的垃圾收集算法才能减少开销。某些面向对象的语言(如C++)允许程序员定义析构函数(destructor)。每当一个对象超出范围或被显式删除时，就自动调用析构函数。这种机制使得程序员能够方便地构造和唤醒释放内存的操作，却又不是垃圾收集机制。

2.实现整体一部分(即聚集)结构的机制

一般说来，有两种实现方法，分别使用指针和独立的关联对象实现整体一部分结构。大多数现有的面向对象语言并不显式支持独立的关联对象，在这种情况下，使用指针是最容易的实现方法，通过增加内部指针可以方便地实现关联。

3.实现一般一特殊(即泛化)结构的机制

既包括实现继承的机制也包括解决名字冲突的机制。所谓解决名字冲突，指的是处理在多个基类中可能出现的重名问题，这个问题仅在支持多重继承的语言中才会遇到。某些语言拒绝接受有名字冲突的程序，另一些语言提供了解决冲突的协议。不论使用何种语言，程序员都应该尽力避免出现名字冲突。

4.实现属性和服务的机制

对于实现属性的机制应该着重考虑以下几个方面：支持实例连接的机制；属性的可见性控制；对属性值的约束。对于服务来说，主要应该考虑下列因素：支持消息连接(即表达对象交互关系)的机制；控制服务可见性的机制；动态联编。

所谓动态联编，是指应用系统在运行过程中，当需要执行一个特定服务的时候，选择(或联编)实现该服务的适当算法的能力。动态联编机制使得程序员在向对象发送消息时拥有较大自由，在发送消息前，无须知道接受消息的对象当时属于哪个类。

5.类型检查

程序设计语言可以按照编译时进行类型检查的严格程度来分类。如果语言仅要求每个变量或属性隶属于一个对象，则是弱类型的；如果语法规定每个变量或属性必须准确地属于某个特定的类，则这样的语言是强类型的。面向对象语言在这方面差异很大，例如，Smalltalk 实际上是一种无类型语言(所有变量都是未指定类的对象)；C++和 Eiffel 则是强类型语言。混合型语言(如 C++，Objective—C 等)甚至允许属性值不是对象而是某种预定义的基本类型数据(如整数，浮点数等)，这可以提高操作的效率。

强类型语言主要有两个优点：一是有利于在编译时发现程序错误，二是增加了优化的可能性。通常使用强类型编译型语言开发软件产品，使用弱类型解释型语言快速开发原型。总的说来，强类型语言有助于提高软件的可靠性和运行效率，现代的程序语言理论支持强类型检查，大多数新语言都是强类型的。

6.类库

大多数面向对象语言都提供一个实用的类库。某些语言本身并没有规定提供什么样的类库，而是由实现这种语言的编译系统自行提供类库。存在类库，许多软构件就不必由程序员重头编写了，这为实现软件重用带来很大方便。

类库中往往包含实现通用数据结构(例如，动态数组、表、队列、栈、树等等)的类，通常把这些类称为包容类。在类库中还可以找到实现各种关联的类。

更完整的类库通常还提供独立于具体设备的接口类(例如，输入输出流)，此外，用于实现窗口系统的用户界面类也非常有用，它们构成一个相对独立的图形库。

7.效率

许多人认为面向对象语言的主要缺点是效率低。产生这种印象的一个原因是，某些早期的面向对象语言是解释型的而不是编译型的。事实上，使用拥有完整类库的面向对象语言，有时能比使用非面向对象语言得到运行更快的代码。这是因为类库中提供了更高效的算法和更好的数据结构，例如，程序员已经无须编写实现哈希表或平衡树算法的代码了，类库中已经提供了这类数据结构，而且算法先进、代码精巧可靠。

认为面向对象语言效率低的另一个理由是，这种语言在运行时使用动态联编实现多态性，这似乎需要在运行时查找继承树，以得到定义给定操作的类。事实上，绝大多数面向对象语言都优化了这个查找过程，从而实现了高效率查找。只要在程序运行时始终保持类结构不变，就能在子类中存储各个操作的正确入口点，从而使得动态联编成为查找哈希表的高效过程，不会由于继承树深度加大或类中定义的操作数增加而降低效率。

8.持久保存对象

任何应用程序都对数据进行处理，如果希望数据能够不依赖于程序执行的生命期而长时间保存下来，则需要提供某种保存数据的方法。希望长期保存数据主要出于以下两个原因：

(1)为实现在不同程序之间传递数据，需要保存数据；

(2)为恢复被中断了的程序的运行，首先需要保存数据。

一些面向对象语言(例如，c++)，没有提供直接存储对象的机制。这些语言的用户必须自己管理对象的输入输出，或者购买面向对象的数据库管理系统。

另外一些面向对象语言(例如，Smalltalk)，把当前的执行状态完整地保存在磁盘上。还有一些面向对象语言，提供了访问磁盘对象的输入输出操作。

通过在类库中增加对象存储管理功能，可以在不改变语言定义或不增加关键字的情况下，就在开发环境中提供这种功能。然后，可以从“可存储的类”中派生出需要持久保存的对象，该对象自然继承了对象存储管理功能。这就是Eiffel语言采用的策略。

理想情况下，应该使程序设计语言语法与对象存储管理语法实现无缝集成。

9.参数化类

在实际的应用程序中，常常看到这样一些软件元素(即函数、类等软件成分)，从它们的逻辑功能看，彼此是相同的，所不同的主要是处理的对象(数据)类型不同。例如，对于一个向量(一维数组)类来说，不论是整型向量，浮点型向量，还是其他任何类型的向量，针对它的数据元素所进行的基本操作都是相同的(例如，插入、删除、检索等)，当然，不同向量的数据元素的类型是不同的。如果程序语言提供一种能抽象出这类共性的机制，则对减少冗余和提高可重用性是大有好处的。

所谓参数化类，就是使用一个或多个类型去参数化一个类的机制，有了这种机制，程序员就可以先定义一个参数化的类模板(即在类定义中包含以参数形式出现的一个或多个类型)，然后把数据类型作为参数传递进来，从而把这个类模板应用在不同的应用程序中，或用在同一应用程序的不同部分。Eiffel语言中就有参数化类，C++语言也提供了类模板。

10.开发环境

软件工具和软件工程环境对软件生产率有很大影响。由于面向对象程序中继承关系和动态联编等引入的特殊复杂性，面向对象语言所提供的软件工具或开发环境就显得尤其重要了。至少应该包括下列一些最基本的软件工具：编辑程序，编译程序或解释程序，浏览工具，调试器(debugger)等。

编译程序或解释程序是最基本、最重要的软件工具。编译与解释的差别主要是速度和效率不同。利用解释程序解释执行用户的源程序，虽然速度慢、效率低，但却可以更方便更灵活地进行调试。编译型语言适于用来开发正式的软件产品，优化工作做得好的编译程序能生成效率很高的目标代码。有些面向对象语言(例如Objective_c)除提供编译程序外，还提供一个解释工具，从而给用户带来很大方便。

某些面向对象语言的编译程序，先把用户源程序翻译成一种中间语言程序，然后再把中间语言程序翻译成目标代码。这样做可能会使得调试器不能理解原始的源程序。在评价调试器时，首先应该弄清楚它是针对原始的面向对象源程序，还是针对中间代码进行调试。如果是针对中间代码进行调试，则会给调试人员带来许多不便。此外，面向对象的调试器，应该能够查看属性值和分析消息连接的后果。

在开发大型系统的时候，需要有系统构造工具和变动控制工具。因此应该考虑语言本身是否提供了这种工具，或者该语言能否与现有的这类工具很好地集成起来。经验表明，传统的系统构造工具(例如，UNIX的Make)目前对许多应用系统来说都已经太原始了。

12.1.3 选择面向对象语言

开发人员在选择面向对象语言时，还应该着重考虑以下一些实际因素。

1.将来能否占主导地位

在若干年以后，哪种面向对象的程序设计语言将占主导地位呢?为了使自己的产品在若干年后仍然具有很强的生命力，人们可能希望采用将来占主导地位的语言编程。

根据目前占有的市场份额，以及专业书刊和学术会议上所做的分析、评价，人们往往能够对未来哪种面向对象语言将占据主导地位做出预测。

但是，最终决定选用哪种面向对象语言的实际因素，往往是诸如成本之类的经济因素而不是技术因素。

2.可重用性

采用面向对象方法开发软件的基本目的和主要优点，是通过重用提高软件生产率。因此，应该优先选用能够最完整、最准确地表达问题域语义的面向对象语言。

3.类库和开发环境

决定可重用性的因素，不仅仅是面向对象程序语言本身，开发环境和类库也是非常重要的因素。事实上，语言、开发环境和类库这3个因素综合起来，共同决定了可重用性。

考虑类库的时候，不仅应该考虑是否提供了类库，还应该考虑类库中提供了哪些有价值的类。随着类库的日益成熟和丰富，在开发新应用系统时，需要开发人员自己编写的代码将越来越少。

为便于积累可重用的类和重用已有的类，在开发环境中，除了提供前述的基本软件工具外，还应该提供使用方便的类库编辑工具和浏览工具。其中的类库浏览工具应该具有强大的联想功能。

4.其他因素

在选择编程语言时，应该考虑的其他因素还有：对用户学习面向对象分析、设计和编码技术所能提供的培训服务；在使用这个面向对象语言期间能提供的技术支持；能提供给开发人员使用的开发工具、开发平台、发行平台；对机器性能和内存的需求；集成已有软件的容易程度等。

12.2 程序设计风格

在本书第7章已经强调指出，良好的程序设计风格对保证程序质量的重要性。良好的程序设计风格对面向对象实现来说尤其重要，不仅能明显减少维护或扩充的开销，而且有助于在新项目中重用已有的程序代码。

良好的面向对象程序设计风格，既包括传统的程序设计风格准则，也包括为适应面向对象方法所特有的概念(例如，继承性)而必须遵循的一些新准则。

12.2.1 提高可重用性

面向对象方法的一个主要目标，就是提高软件的可重用性。正如11.3节所述，软件重

用有多个层次，在编码阶段主要涉及代码重用问题。一般说来，代码重用有两种：一种是本项目内的代码重用，另一种是新项目重用旧项目的代码。内部重用主要是找出设计中相同或相似的部分，然后利用继承机制共享它们。为做到外部重用(即一个项目重用另一项目的代码)，则必须有长远眼光，需要反复考虑精心设计。虽然为实现外部重用而需要考虑的面，比为实现内部重用而需要考虑的面更广，但是，有助于实现这两类重用的程序设计准则却是相同的。下面讲述主要的准则：

1.提高方法的内聚

一个方法(即服务)应该只完成单个功能。如果某个方法涉及两个或多个不相关的功能，则应该把它分解成几个更小的方法。

2.减小方法的规模

应该减小方法的规模，如果某个方法规模过大(代码长度超过一页纸可能就太大了)，则应该把它分解成几个更小的方法。

3.保持方法的一致性

保持方法的一致性，有助于实现代码重用。一般说来，功能相似的方法应该有一致的名字、参数特征(包括参数个数、类型和次序)、返回值类型、使用条件及出错条件等。

4.把策略与实现分开

从所完成的功能看，有两种不同类型的方法。一类方法负责做出决策，提供变元，并且管理全局资源，可称为策略方法。另一类方法负责完成具体的操作，但却并不做出是否执行这个操作的决定，也不知道为什么执行这个操作，可称为实现方法。

策略方法应该检查系统运行状态，并处理出错情况，它们并不直接完成计算或实现复杂的算法。策略方法通常紧密依赖于具体应用，这类方法比较容易编写，也比较容易理解。

实现方法仅仅针对具体数据完成特定处理，通常用于实现复杂的算法。实现方法并不制定决策，也不管理全局资源，如果在执行过程中发现错误，它们应该只返回执行状态而不对错误采取行动。由于实现方法是自含式算法，相对独立于具体应用，因此，在其他应用系统中也可能重用它们。

为提高可重用性，在编程时不要把策略和实现放在同一个方法中，应该把算法的核心部分放在一个单独的具体实现方法中。为此需要从策略方法中提取出具体参数，作为调用实现方法的变元。

5.全面覆盖

如果输入条件的各种组合都可能出现，则应该针对所有组合写出方法，而不能仅仅针对当前用到的组合情况写方法。例如，如果在当前应用中需要写一个方法，以获取表中第一个元素，则至少还应该为获取表中最后一个元素再写一个方法。

此外，一个方法不应该只能处理正常值，对空值、极限值及界外值等异常情况也应该能够作出有意义的响应。

6.尽量不使用全局信息

应该尽量降低方法与外界的耦合程度，不使用全局信息是降低耦合度的一项主要措施。

7.利用继承机制

在面向对象程序中,使用继承机制是实现共享和提高重用程度的主要途径.

(1) 调用子过程.最简单的做法是把公共的代码写成一个被其他方法调用公用方法.可以在基类中定义这个公用方法,如图12.1所示.

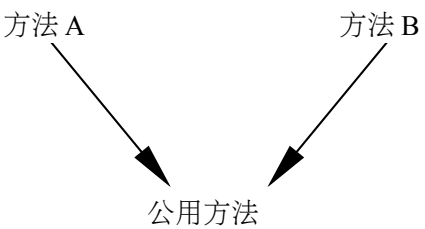

图12.1 通过调用公用方法实现代码重用

公用方法
公共代码
调用方法
公共代码

中的方法调用,如图

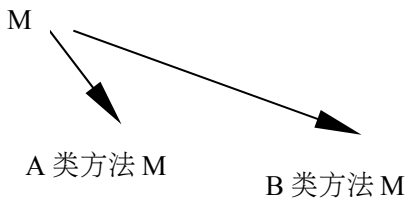

图12.2 通用因子分解实现代码重用

(2)分解因子。有时提高相似类代码可重用性的一个有效途径，是从不同类的相似方法中分解出不同的“因子”(即不同的代码)，把余下的代码作为公用方法中的公共代码，把分解出的因子作为名字相同算法不同的方法，放在不同类中定义，并被这个公用方法调用，如图 12.2 所示。使用这种途径通常额外定义一个抽象基类，并在这个抽象基类中定义公用方法 。把这种途径与面向对象语言提供的多态性机制结合起来，让派生类继承抽象基类中定义的公用方法，可以明显降低为增添新子类而需付出的工作量，因为只需在新子类中编写其特有的代码。

(3)使用委托。继承关系的存在意味着子类“即是”父类，因此，父类的所有方法和属性应该都适用于子类。仅当确实存在一般一特殊关系时，使用继承才是恰当的。继承机制使用不当将造成程序难于理解、修改和扩充。

当逻辑上不存在一般一特殊关系时，为重用已有的代码，可以利用委托机制，如本书 11.11.3 小节所述。

(4)把代码封装在类中 。程序员往往希望重用用其他方法编写的 、解决同一类应用问题的程序代码。重用这类代码的一个比较安全的途径，是把被重用的代码封装在类中。

例如，在开发一个数学分析应用系统的过程中，已知有现成的实现矩阵变换的商品软件包，程序员不想用 C++语言重写这个算法，于是他定义一个矩阵类把这个商品软件包的功能封装在该类中。

12.2.2 提高可扩充性

上一小节所述的提高可重用性的准则，也能提高程序的可扩充性 。此外，下列的面向对象程序设计准则也有助于提高可扩充性:

1.封装实现策略

应该把类的实现策略(包括描述属性的数据结构 、修改属性的算法等)封装起来，对外只提供公有的接口，否则将降低今后修改数据结构或算法的自由度。

2.不要用一个方法遍历多条关联链

一个方法应该只包含对象模型中的有限内容 。违反这条准则将导致方法过分复杂，既不易理解，也不易修改扩充。

3.避免使用多分支语句

一般说来，可以利用 DO—CASE 语句测试对象的内部状态，而不要用来根据对象类型选择应有的行为，否则在增添新类时将不得不修改原有的代码 。应该合理地利用多态性机制，根据对象当前类型，自动决定应有的行为。

4.精心确定公有方法

公有方法是向公众公布的接口。对这类方法的修改往往会涉及许多其他类，因此，修改公有方法的代价通常都比较高。为提高可修改性，降低维护成本，必须精心选择和定义公有方法。私有方法是仅在类内使用的方法，通常利用私有方法来实现公有方法。删除、增加或修改私有方法所涉及的面要窄得多，因此代价也比较低。

同样，属性和关联也可以分为公有和私有两大类，公有的属性或关联又可进一步设置为具有只读权限或只写权限两类。

12.2.3 提高健壮性

程序员在编写实现方法的代码时，既应该考虑效率，也应该考虑健壮性。通常需要在健壮性与效率之间做出适当的折衷。必须认识到，对于任何一个实用软件来说，健壮性都是不可忽略的质量指标。为提高健壮性应该遵守以下几条准则。

1.预防用户的操作错误

软件系统必须具有处理用户操作错误的能力。当用户在输人数据时发生错误，不应该引起程序运行中断，更不应该造成“死机”。任何一个接收用户输人数据的方法，对其接收到的数据都必须进行检查，即使发现了非常严重的错误，也应该给出恰当的提示信息，并准备再次接收用户的输入。

2.检查参数的合法性

对公有方法，尤其应该着重检查其参数的合法性，因为用户在使用公有方法时可能违反参数的约束条件。

3.不要预先确定限制条件

在设计阶段，往往很难准确地预测出应用系统中使用的数据结构的最大容量需求。因此不应该预先设定限制条件。如果有必要和可能，则应该使用动态内存分配机制，创建未预先设定限制条件的数据结构。

4.先测试后优化

为在效率与健壮性之间做出合理的折衷，应该在为提高效率而进行优化之前，先测试程序的性能，人们常常惊奇地发现，事实上大部分程序代码所消耗的运行时间并不多。应该仔细研究应用程序的特点，以确定哪些部分需要着重测试(例如，最坏情况出现的次数及处理时间，可能需要着重测试)。经过测试，合理地确定为提高性能应该着重优化的关键部分。如果实现某个操作的算法有许多种，则应该综合考虑内存需求、速度及实现的简易程度等因素，经合理折衷选定适当的算法。

12.3 测试策略

测试软件的经典策略是，从“小型测试”开始，逐步过渡到“大型测试”。用软件测试的专业术语描述，就是从单元测试开始，逐步进入集成测试，最后进行确认测试和系统测试。对于传统的软件系统来说，单元测试集中测试最小的可编译的程序单元(过程模块)，一旦把这些单元都测试完之后，就把它们集成到程序结构中去；在集成过程中还应该进行一系列的回归测试，以发现模块接口错误和新单元加入到程序中所带来的副作用；最后，把

软件系统作为一个整体来测试，以发现软件需求错误。测试面向对象软件的策略与上述策略基本相同，但也有许多新特点。

12.3.1 面向对象的单元测试

当考虑面向对象的软件时，单元的概念改变了。“封装”导致了类和对象的定义，这意味着类和类的实例(对象)包装了属性(数据)和处理这些数据的操作(也称为方法或服务)。现在，最小的可测试单元是封装起来的类和对象。一个类可以包含一组不同的操作，而一个特定的操作也可能存在于一组不同的类中。因此，对于面向对象的软件来说，单元测试的含义发生了很大变化。

测试面向对象软件时，不能再孤立地测试单个操作，而应该把操作作为类的一部分来测试。例如，假设有一个类层次，操作X在超类中定义并被一组子类继承，每个子类都使用操作X，但是，X调用子类中定义的操作并处理子类的私有属性。由于在不同的子类中使用操作X的环境有微妙的差别，因此有必要在每个子类的语境中测试操作X。这就说明，当测试面向对象软件时，传统的单元测试方法是不适用的，不能再在“真空，中(即孤立地)测试单个操作。

12.3.2 面向对象的集成测试

因为在面向对象的软件中不存在层次的控制结构，传统的自顶向下或自底向上的集成策略就没有意义了。此外，由于构成类的各个成分彼此间存在直接或间接的交互，一次集成一个操作到类中(传统的渐增式集成方法)通常是不现实的。

面向对象软件的集成测试主要有下述两种不同的策略。

(1)基于线程的测试(thread based testing)。这种策略把响应系统的一个输入或一个事件所需要的那些类集成起来。分别集成并测试每个线程，同时应用回归测试以保证没有产生副作用。

(2)基于使用的测试(use based testing)。这种方法首先测试几乎不使用服务器类的那些类(称为独立类)，把独立类都测试完之后，再测试使用独立类的下一个层次的类(称为依赖类)。对依赖类的测试一个层次一个层次地持续进行下去，直至把整个软件系统构造完为止。

在测试面向对象的软件过程中，应该注意发现不同的类之间的协作错误。集群测试(cluster testing)是面向对象软件集成测试的一个步骤。在这个测试步骤中，用精心设计的测试用例检查一群相互协作的类(通过研究对象模型可以确定协作类)，这些测试用例力图发现协作错误。

12.3.3 面向对象的确认测试

在确认测试或系统测试层次，不再考虑类之间相互连接的细节。和传统的确认测试一样，面向对象软件的确认测试也集中检查用户可见的动作和用户可识别的输出。为了导出确认测

试用例，测试人员应该认真研究动态模型和描述系统行为的脚本，以确定最可能发现用户交互需求错误的情景。

当然，传统的黑盒测试方法(见本书第7章)也可用于设计确认测试用例，但是，对于面向对象的软件来说，主要还是根据动态模型和描述系统行为的脚本来设计确认测试用例。

12.4 设计测试用例

目前，面向对象软件的测试用例的设计方法，还处于研究、发展阶段。与传统软件测试(测试用例的设计由软件的输入—处理—输出视图或单个模块的算法细节驱动)不同，面向对象测试关注于设计适当的操作序列以检查类的状态。

12.4.1 测试类的方法

前面已经讲过，软件测试从“小型测试”开始，逐步过渡到“大型测试”。对面向对象的软件来说，小型测试着重测试单个类和类中封装的方法。测试单个类的方法主要有随机测试、划分测试和基于故障的测试等3种。

1.随机测试

下面通过银行应用系统的例子，简要地说明这种测试方法。该系统的account(账户)类有下列操作：open(打开)，setup(建立)，deposit(存款)，withdraw(取款)，balance(余额)，summarize(清单)，creditLimit(透支限额)和close(关闭)。上列每个操作都可以应用于account类的实例，但是，该系统的性质也对操作的应用施加了一些限制，例如，必须在应用其他操作之前先打开账户，在完成了全部操作之后才能关闭账户。即使有这些限制，可做的操作也有许多种排列方法。一个account类实例的最小行为历史包括下列操作：

open·setup·deposit·withdraw·close

这就是对account类的最小测试序列。但是，在下面的序列中可能发生许多其他行为：

open·setup·deposit·[deposit | withdraw | balance | summarize|creditLimit]^{n}·withdraw·close

从上列序列可以随机地产生一系列不同的操作序列，例如：

测试用例#r1

: open·setup·deposit·deposit·deposit·balance·summarize·withdraw·close

测试用例#r2：open·setup·deposit·withdraw·deposit·balance·creditLimit·withdraw·close

执行上述这些及另外一些随机产生的测试用例，可以测试类实例的不同生存历史。

2.划分测试

与测试传统软件时采用等价划分方法类似，采用划分测试(partition testing)方法可以减少测试类时所需要的测试用例的数量。首先，把输入和输出分类，然后设计测试用例以测试划分出的每个类别。下面介绍划分类别的方法。

(1)基于状态的划分

这种方法根据类操作改变类状态的能力来划分类操作。再一次考虑account类，状态操作包括deposit和withdraw，而非状态操作有balance，summarize和creditLimit。设计测试用例，以分别测试改变状态的操作和不改变状态的操作。例如，用这种方法可以设计出如下的测试用例：

测试用例#p1：open·setup·deposit·deposit·withdraw·withdraw·close

测试用例#p2：open·setup·deposit·summarize·creditLimit·withdraw·close

测试用例#P1改变状态，而测试用例#P2测试不改变状态的操作(在最小测试序列中的操作除外)。

(2)基于属性的划分

这种方法根据类操作使用的属性来划分类操作。对于account类来说，可以使用属性balance来定义划分，从而把操作划分成3个类别：

·使用balance的操作；

·修改balance的操作；

·不使用也不修改balance的操作。

然后，为每个类别设计测试序列。

(3)基于功能的划分

这种方法根据类操作所完成的功能来划分类操作。例如，可以把account类中的操作分类为初始化操作(open，setup)，计算操作(deposit，withdraw)，查询操作(balance，summarize，creditLimit)和终止操作(close)。然后为每个类别设计测试序列。

3.基于故障的测试

基于故障的测试(fault based testing)与传统的错误推测法类似，也是首先推测软件中可能有的错误，然后设计出最可能发现这些错误的测试用例。例如，软件工程师经常在问题的边界处犯错误，因此，在测试SQRT(计算平方根)操作(该操作在输入为负数时返回出错信息)时，应该着重检查边界情况：一个接近零的负数和零本身。其中“零本身”用于检查程序员是否犯了如下错误：

把语句if(x>=0)calculate_square_root();

误写成if(x>0)calculate_square_root();

为了推测出软件中可能有的错误，应该仔细研究分析模型和设计模型，而且在很大程度上要依靠测试人员的经验和直觉。如果推测得比较准确，则使用基于故障的测试方法能够用相当低的工作量发现大量错误；反之，如果推测不准，则这种方法的效果并不比随机测试技术的效果好。

12.4.2 集成测试方法

开始集成面向对象系统以后，测试用例的设计变得更加复杂。在这个测试阶段，必须对类间协作进行测试。为了举例说明设计类间测试用例的方法，我们扩充12.4.1小节引入的银行系统的例子，使它包含图12.3所示的类和协作。图中箭头方向代表消息的传递方向，箭头线上的标注给出了作为由消息所蕴含的协作的结果而调用的操作。

和测试单个类相似，测试类协作可以使用随机测试方法和划分测试方法，以及基于情景的测试和行为测试来完成。

1.多类测试

Kirani和Tsai建议使用下列步骤，以生成多个类的随机测试用例。‘

·对每个客户类，使用类操作符列表来生成一系列随机测试序列。这些操作符向服务器类实例发送消息。

·对所生成的每个消息，确定协作类和在服务器对象中的对应操作符。

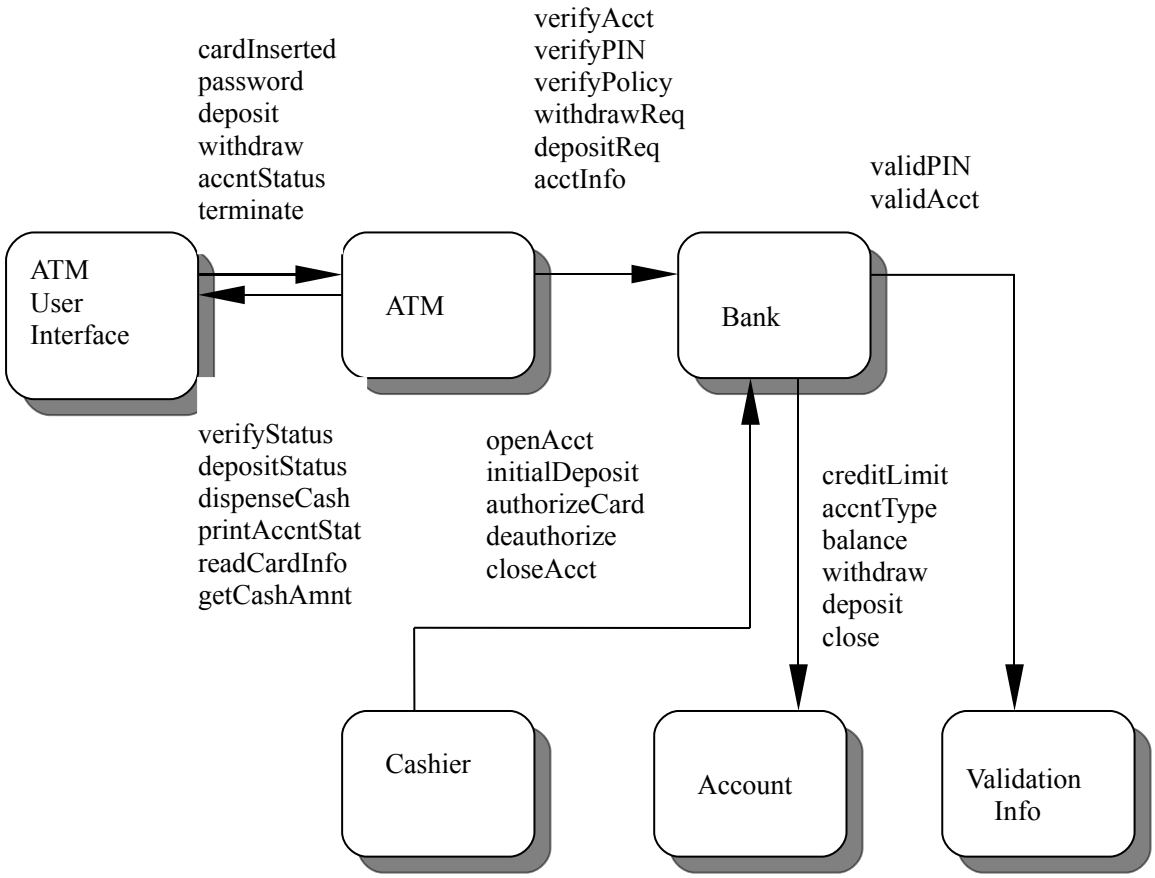

图 12.3　银行系统的类 - 协操作

·对服务器对象中的每个操作符(已经被来自客户对象的消息调用)，确定传递的消息。

·对每个消息，确定下一层被调用的操作符，并把这些操作符结合进测试序列中。

为了说明怎样用上述步骤生成多个类的随机测试用例，考虑 Bank 类相对于 ATM 类(见图 12.3)的操作序列：

verifyAcct·verifyPIN·[(verifyPolicy·withdrawReq)|depositReq|acct1nfoREQ]^{n}

对 Bank 类的随机测试用例可能是：

测试用例#r3：verifyAcct·verifyPIN·depositReq

为了考虑在上述这个测试中涉及的协作者，需要考虑与测试用例#r3 中的每个操作相关联的消息。Bank 必须和 ValidationInfo 协作以执行 verifyAcct 和 verifyPIN，Bank 还必须和 Account 协作以执行 depositReq。因此，测试上面提到的协作的新测试用例是：

测试用例#r4：

$verifyAcct_{Bank}$·[$validAcct_{VlidationInfo}$]·$veilfyPIN_{Bank}$·[$validPIN_{VlidationInfo}$]·depositReq·[$deposit_{account}$]

多个类的划分测试方法类似于单个类的划分测试方法(见 12.4.1 节)。但是，对于多类测试来说，应该扩充测试序列以包括那些通过发送给协作类的消息而被调用的操作。另一种划分测试方法，根据与特定类的接口来划分类操作。如图 12.3 所示，Bank 类接收来自 ATM 类和 Cashier 类的消息，因此，可以通过把 Bank 类中的方法戈 II 分成服务于 ATM 的和服务于 Cashier 的两类来测试它们。还可以用基于状态的划分(见 12.4.1 节)，进一步精化划分。

2.从动态模型导出测试用例

在本书第 9 章中已经讲过，怎样用状态转换图作为表示类的动态行为的模型。类的状态图可以帮助我们导出测试该类(及与其协作的那些类)的动态行为的测试用例。图 12.4 给出了前面讨论过的 account 类的状态图，从图可见，初始转换经过了 empty acct 和 setup acct 这两个状态，而类实例的大多数行为发生在 working acct 状态中，最终的 withdraw 和 close 使得 account 类分别向 nonworking acct 状态和 dead acct 状态转换。

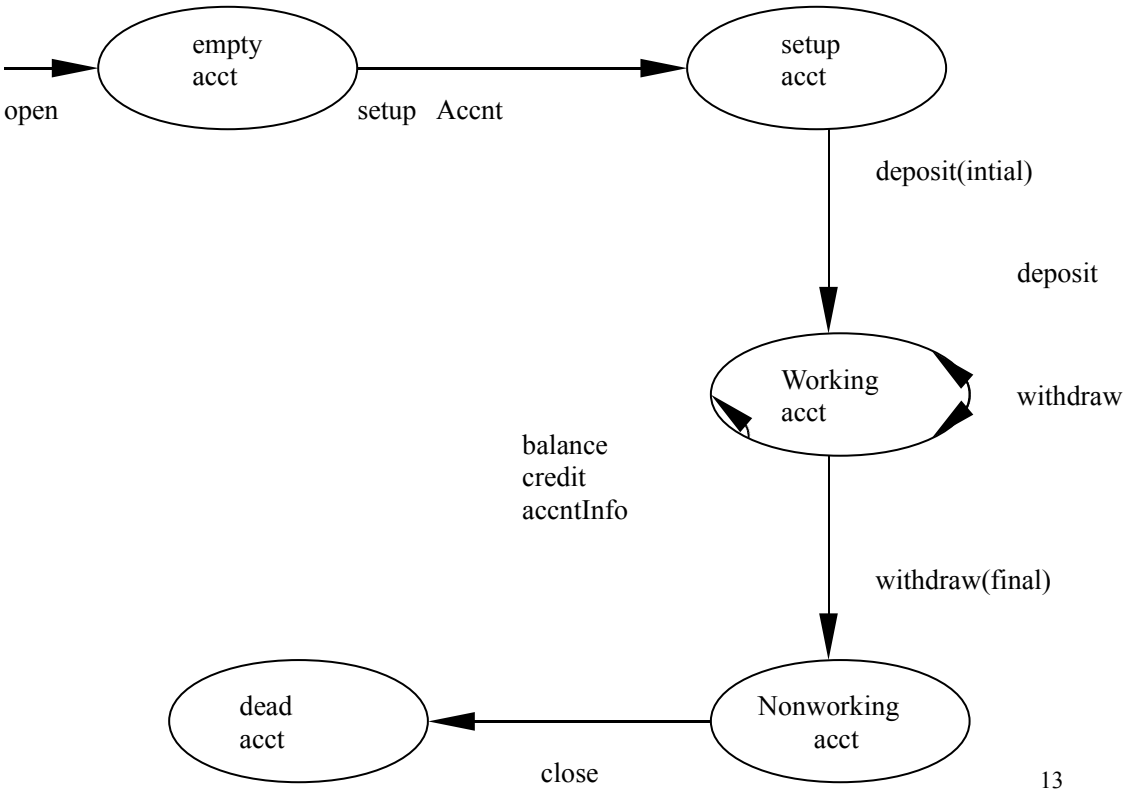

图 12.4 account 类的状态转换图

设计出的测试用例应该覆盖所有状态，也就是说，操作序列应该使得 account 类实例遍历所有允许的状态转换：

测试用例#s1：open·setupAccnt·deposit(initial)·withdraw(final)·close 应该注意，上面列出的序列与 12.4.1 节讨论的最小测试序列相同。向最小序列中加入附加的测试序列，可以得出其他测试用例：

测试用例#s2：open·setupAccnt·deposit(initial)·deposit·balance·credit·withdraw(final)·close

测试用例#s3：open·setupAccnt·deposit(initial)·deposit·withdraw·accntInfo·withdraw(final)·close

还可以导出更多测试用例，以保证该类的所有行为都被适当地测试了。在类的行为导致与一个或多个类协作的情况下，应该使用多个状态图去跟踪系统的行为流。

12.5 小　结

面向对象方法学把分析、设计和实现很自然地联系在一起了。虽然面向对象设计原则上不依赖于特定的实现环境，但是实现结果和实现成本却在很大程度上取决于实现环境。因此，直接支持面向对象设计范式的面向对象程序语言、开发环境及类库，对于面向对象实现来说是非常重要的。

为了把面向对象设计结果顺利地转变成面向对象程序，首先应该选择一种适当的程序设计语言。面向对象的程序设计语言非常适合用来实现面向对象设计结果。事实上，具有方便的开发环境和丰富的类库的面向对象程序设计语言，是实现面向对象设计的最佳选择。

良好的程序设计风格对于面向对象实现来说格外重要。它既包括传统的程序设计风格准则，也包括与面向对象方法的特点相适应的一些新准则。

面向对象方法学使用独特的概念和技术完成软件开发工作，因此，在测试面向对象程序的时候，除了继承传统的测试技术之外，还必须研究与面向对象程序特点相适应的新的测试技术。

面向对象测试的总目标与传统软件测试的目标相同，也是用最小的工作量发现最多的错误。但是，面向对象测试的策略和技术与传统测试有所不同，测试的焦点从过程构件(传统模块)移向了对象类。

一旦完成了面向对象程序设计，就开始对每个类进行单元测试。测试类时使用的方法主要有随机测试、划分测试和基于故障的测试。每种方法都测试类中封装的操作。应该设计测试序列以保证相关的操作受到充分测试。检查对象的状态(由对象的属性值表示)，以确定是否存在错误。

可以采用基于线程或基于使用的策略完成集成测试。基于线程的测试，集成一组相互协作以对某个输入或某个事件作出响应的类。基于使用的测试，从那些不使用服务器类的类开始，按层次构造系统。设计集成测试用例，也可以采用随机测试和划分测试方法。此外，从

动态模型导出的测试用例，可以测试指定的类及其协作者。

面向对象系统的确认测试也是面向黑盒的，并且可以应用传统的黑盒方法完成测试工作。但是，基于情景的测试是面向对象系统确认测试的主要方法。

习 题 12

1.面向对象实现应该选用哪种程序设计语言?为什么?
2.面向对象程序设计语言主要有哪些技术特点?
3.选择面向对象程序设计语言时主要应该考虑哪些因素?
4.良好的面向对象程序设计风格主要有哪些准则?
5.测试面向对象软件时，单元测试、集成测试和确认测试各有哪些新特点?
6.测试面向对象软件时，主要有哪些设计单元测试用例的方法?
7.测试面向对象软件时，主要有哪些设计集成测试用例的方法?
8.测试面向对象软件时，主要有哪些设计确认测试用例的方法?
9.试用C++语言实现(编程并测试)本书习题11第4题要求设计的定货系统。