# 第1章 对象导言

计算机革命起源于一台机器。因此，程序设计语言的起源看上去也起源于那台机器。 21

然而，计算机并不仅仅是一台机器，因为它们就像是智力放大工具（Steve Jobs[⊖]喜欢称其为“智力的自行车”）和另一种富于表现力的媒体。所以，它们渐渐地不太像机器，而更像我们大脑的一部分了，它们还挺像其他的一些富于表现力的媒体（例如写作、绘画、雕刻、动画或电影制作）。面向对象程序设计是“使计算机成为一种富于表现力的媒体”这一华彩乐章的一部分。

本章将介绍面向对象程序设计（OOP）的基本概念，包括OOP开发方法的概述。在读者阅读本书之前，我们假设读者已经有了使用过程型程序设计语言的经验，当然不一定是C语言。如果读者认为在学习本书之前需要补习程序设计方面的预备知识和C的语法知识，可以参考本书附带的光盘“Thinking in C: Foundations for C++ and Java”，这些内容也可以在*www.BruceEckel.com*中找到。

本章是一些背景和辅助材料。如果在未了解这些大背景之前就进入面向对象程序设计，许多人都会感到有困难。因此，这里为读者预先介绍有关OOP的一些基本概念。但是，还有另一些读者，在不见到某些具体结构之前，就不能理解语言的整体概念；这些人不接触某些代码就会停止不前和不知所措。如果读者属于后者，急于学习这门语言的具体内容，则可以跳过本章，这样不会妨碍写程序和学习这门语言。但是，读者以后终将回过头来补充本章的知识，这样才能理解为什么对象如此重要，以及如何用对象设计程序。

## 1.1 抽象的过程

所有的程序设计语言都提供抽象。可以说，人们能解决的问题的复杂性直接与抽象的类型和质量有关。这里“类型”指的是“要抽象的东西”。汇编语言是对底层机器的小幅度抽象。其 22
后的许多所谓“命令式”语言（例如Fortran、BASIC和C）都是对汇编语言的抽象。这些语言较之汇编语言有了很大的改进，但是它们的主要抽象仍然要求程序员按计算机的结构去思考，而不是按要解决的问题的结构去思考。程序员必须在机器模型（在“解空间”，即建模该问题的空间中，例如在计算机中）和实际上要解决的问题的模型（在“问题空间”，即问题存在的空间中）之间建立联系。程序员必须努力进行这之间的对应，这实际上是程序设计语言之外的任务，它使得程序难以编写且维护费用昂贵，并且作为一种副效应，造就了整个“程序设计方法”产业。

取代对机器建模的另一种方式是对要解决的问题进行建模。像LISP和APL这样的早期语言选取了特殊的“世界观”（“所有的问题最终都是表”或“所有的问题都是算法”），PROLOG则把所有的问题都看做决策链。还有一些语言，创造它们的目的是为了基于约束的编程和专门用于通过绘图符号来编程（后者已被证明局限太大）。这些方法中的每一种对于它所针对的特定问题都是很好的解决方案，但对于这个领域之外的问题，就笨拙难用了。

⊖ 史蒂夫·乔布斯（Steve Jobs）是苹果公司的创始人和精神领袖。——编辑注

面向对象的方法为程序员提供了在问题空间中表示各种事物元素的工具，从而向前迈进了一大步。这种表示方法是通用的，并不限定程序员只处理特定类型的问题。我们把问题空间中的事物和它们在解空间中的表示称为“对象”（当然，还需要另外一些对象，它们在问题空间中没有对应物）。其思想是允许程序通过添加新的对象类型，而使程序本身能够根据问题的术语进行调整，这样当我们读描述解决方案的代码时，也就是在读表达该问题的文字。这
23 是比以前更灵活和更强大的语言抽象。因此，OOP允许程序员用问题本身的术语来描述问题，而不是用要运行解决方案的计算机的术语来描述问题。当然这些问题的术语仍然与计算机有联系。每个对象看上去像一台小计算机，它有状态，有可以执行的运算。这似乎是现实世界中对象的很好类比，它们都有特性和行为。

一些语言的设计者认为，面向对象程序设计本身并不足以轻易解决所有程序设计问题，他们提倡结合各种方法，形成**多范型**（*multiparadigm*）的程序设计语言[⊖]。

Alan Kay总结了Smalltalk的五个基本特性，这些特性代表了纯面向对象程序设计的方法。Smalltalk是第一个成功的面向对象语言，是C++的基础语言之一。

(1) **万物皆对象。**对象可以被认为是一个奇特的变量，它能存放数据，而且可以对它“提出请求”，要求它执行对它自身的运算。理论上，我们可以在需要解决的问题中取出任意概念性的成分（狗、建筑物、服务等），把它表示为程序中的对象。

(2) **程序就是一组对象，对象之间通过发送消息互相通知做什么。**更具体地讲，可以将消息看做是对于调用某个特定对象所属函数的请求。

(3) **每一个对象都有它自己的由其他对象构成的存储区。**这样，就可以通过包含已经存在的对象创造新对象。因此，程序员可以构造出复杂的程序，而且能将程序的复杂性隐藏在对
24 象的简明性背后。

(4) **每个对象都有一个类型。**采用OOP术语，每个对象都是某个类的**实例**（instance），其中“类”（class）与“类型”（type）是同义词。类的最重要的突出特征是“能向它发送什么消息”。

(5) **一个特定类型的所有对象都能接收相同的消息。**正如后面将看到的，这实际上是一条装载语句。因为一个“circle”类型的对象也是一个“shape”类型的对象，所以保证circle能接收shape消息。这意味着，我们可以编写与shape通信的代码，该代码能自动地对符合shape描述的任何东西进行处理。这种**替换能力**（*substitutability*）是OOP的最强大的思想之一。

## 1.2 对象有一个接口

亚里士多德可能是第一个认真研究**类型**（*type*）概念的人，他提到了“鱼类和鸟类”。所有对象（虽然都具有惟一性）都是一类对象中的一员，它们有共同的特征和行为。这一思想在第一个面向对象语言Simula-67中得到了直接的应用，该语言用基本关键字**class**在程序中引入新类型。

顾名思义，创造Simula的目的是为了解决模拟问题，例如著名的“银行出纳员问题”[⊜]。其中包括出纳员、顾客、账户、交易、货币的单位等大量的“对象”。把那些在程序执行期间的状态之外其他方面都一样的对象归为“几类对象”，这就是关键字**class**（类）的来源。创建抽象数据类型是面向对象程序设计的基本思想。抽象数据类型几乎能完全像内部类型一样工

⊖ 参见Timothy Budd所写的专著《*Multiparadigm Programming in Leda*》(Addison-Wesley, 1995)。

⊜ 可以在本书的第2卷中找到这一问题的有趣实现，本书的第2卷可从www.BruceEckel.com上找到。

作。程序员可以创建类型的变量［面向对象的说法称为**对象**（*object*）或**实例**（*instance*）］和操纵这些变量（称为**发送消息**或**请求**，对象根据发来的消息推断需要做什么事情）。每个类的成员（元素）都有共性：每个账户有余额，每个出纳员都能接收存款，等等。同时，每个成员都有自己的状态，每个账户有不同的余额，每个出纳员都有名字。这样，在计算机程序中，出纳员、客户、账户、交易等，每一个都被描述为惟一的实体。这个实体就是对象，每个对象都属于一个定义了它的特性和行为的特定类。 25

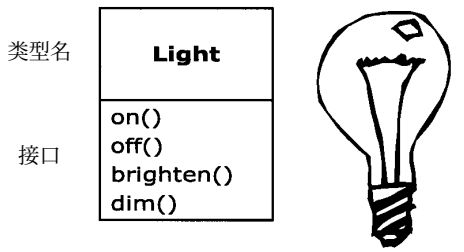
所以，虽然在面向对象的程序设计中，我们所做的工作实际上是创造新数据类型，但事实上所有的面向对象的程序设计语言都使用关键字“ class ”。当碰到“ 类型（ type）”时可以看做“ 类（class）”，反之亦然[⊖]。

由于类描述了一组有相同特性（数据元素）和相同行为（功能）的对象，因此类实际上就是数据类型，例如浮点数也有一组特性和行为。区别在于，程序员定义类是为了与具体问题相适应，而不是被迫使用已存在的数据类型，而设计这些已存在的数据类型的动机是为了表示机器中的存储单元。程序员可以通过增添专门针对自己需要的新数据类型来扩展程序设计语言。这种程序设计系统欢迎新的类，关注新的类，对它们进行与内置类型一样的类型检查。

面向对象方法并不限于模拟创建。无论我们是否同意，都不能否认任何程序都是我们正在设计的系统的一种模拟，OOP技术的使用确实可以容易地将大量问题缩减为一个简单的解决方案。

一旦建立了一个类，程序员想制造这个类的多少个对象就可以制造多少个，然后操作这些对象，就如同它们是所解决的问题中的元素。实际上，面向对象程序设计的难题之一，是在问题空间中的元素和解空间的对象之间建立一对一的映射。 26

但是，我们如何得到一个对象去为我们做有用工作呢？必须有一种方法能向对象作出请求，使得它能做某件事情，例如完成交易、在屏幕上画图或打开开关。每个对象只能满足特定的请求。可以向对象发出的请求是由它的**接口**（*interface*）定义的，而接口由类型确定。一个简单的例子可能该是电灯泡的表示了。




```
Light lt;
lt.on();
```

接口规定我们能向特定的对象发出什么请求。然而，必须有代码满足这种请求，再加上隐藏的数据，就组成了**实现**（*implementation*）。从过程型程序设计的观点看，这并不复杂。类型对每个可能的请求都有一个相关的函数，当向对象发请求时，就调用这个函数。这个过程通常概括为向对象“ 发送消息 ”（提出请求），对象根据这个消息确定做什么（执行代码）。

如上例，这个类型或称类的名字是 **Light**，这个特定的 **Light**对象的名字是 **lt**，可以对**Light**对象提出一些请求：打开它、关闭它、使它变亮或变暗。通过声明一个名字（ **lt**），可以

⊖ 一些人对此做了区分，他们认为类型（ type）确定接口，而类（class）是对这个接口的特定实现。

创建一个**Light**对象。为了向这个对象发送消息，可以说出这个对象的名字，并用句点连接对消息的请求。从使用预定义类的用户的角度看，用对象编程只需要这些工作就行了。

上图符合**统一建模语言**（Unified Modeling Language, UML）的格式，每个类由一个方框
27 表示，这个方框的顶部标有类型名，中间部分列出所关注的数据成员，底部是**成员函数**（member function属于这个对象的函数，它们能接收发送给这个对象的任何消息）。通常，只有类的名字和公共成员函数会在 UML设计图中表示出来，所以中间部分不显示。如果只对类的名字感兴趣，则底部也不需要显示。

## 1.3 实现的隐藏

把程序员划分为**类创建者**（创建新数据类型的人）和**客户程序员**[⊖]（在应用程序中使用数据类型的类的用户）是有益的。客户程序员的目标是去收集一个装满类的工具箱，用于快速应用开发。类创建者的目标是去建造类，这个类只暴露对于客户程序员是必需的东西，其他的都隐藏起来。为什么呢？因为如果是隐藏的东西，客户程序员就不能使用它，这意味着，这个类的创建者可以改变隐藏的部分，而不用担心会影响其他人。被隐藏的部分通常是对象内部的管理功能，容易受到粗心或无知的客户程序的损害，所以隐藏实现会减少程序错误。隐藏的概念怎么强调都不过分。

在任何关系中，存在一个所有的参与者都遵从的边界是重要的。当我们创建一个库时，也就与客户程序员建立了关系。他们也是程序员，但是他们是通过使用我们的库来组装应用程序，也可能建立更大的库。

如果类的所有成员对于任何人都能用，那么客户程序员就可以用这个类做其中的任何事
28 情，不存在强制规则。虽然我们可能实际上不希望客户程序员直接操纵这个类的某些成员，但是因为没有访问控制，所以就没有办法保护它，所有的东西都暴露无遗。

访问控制的第一个理由是为了防止客户程序员插手他们不应当接触的部分，也就是对于数据类型的内部实施方案是必需的部分，而不是用户为了解决他们的特定问题所需要的接口部分。这实际上是对用户的很好服务，因为这使得他们容易看到哪些部分对于他们是重要的，哪些部分是可以忽略的。

访问控制的第二个理由是允许库设计者去改变这个类的内部工作方式，而不必担心这样做会影响客户程序员。例如，库设计者可能为了容易开发而用简单的方法实现了一个特殊的类，但后来发现需要重写这个类以使得它运行得更快。如果接口和实现是严格分离和被保护的，那么库设计者就可以很容易完成重写任务，用户只需要重新连接就可以了。

C++语言使用了三个明确的关键字来设置类中的边界： **public**、**private**和**protected**。它们的使用和含义相当简明。这些**访问说明符**（*access specifier*）确定谁能用随后的定义。**public**意味着随后的定义对所有人都可用。相反， **private**关键字则意味着，除了该类型的创建者和该类型的内部成员函数之外，任何人都不能访问这些定义。 **private**在我们与客户程序员之间筑起了一道墙。如果有人试图访问一个私有成员，就会产生一个编译时错误。**protected**与**private**基本相似，只有一点不同，即继承的类可以访问 **protected**成员，但不能访问**private**成员。我们将在稍后部分介绍继承。

---

⊖ 对于这个术语，我要感谢我的朋友 Scott Meyers（名著《Effective C++》的作者 —— 编辑注）。

## 1.4 实现的重用

创建了一个类并进行了测试后，这个类（理论上）就成了有用的代码单元。重用性并不像许多人所希望的那样容易达到，产生一个好设计需要经验和洞察力。但是只要有了这样的设计，就应该提供重用。代码重用是面向对象程序设计语言的最大优点之一。 29

重用一个类最简单的方法就是直接使用这个类的对象，并且还可以将这个类的对象放到一个新类的里面。我们称之为“创建一个成员对象”。可以用任何数量和类型的其他对象组成新类，通过组合得到新类所希望的功能。因为这是由已经存在的类组成新类，所以称为**组合**（*composition*）［或者更通常地称为**聚合**（*aggregation*）］。组合常常被称为“`has-a`（有）”关系，比如“在小汽车中有发动机”一样。


（上面的UML图用实心菱形表示组合，说明这里有一个小汽车。通常我将采用一种更简单的形式，即仅用一条不带菱形的线来表示一个关联。⊖）

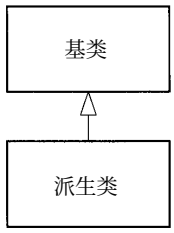
组合具有很大的灵活性，新类的成员对象通常是私有的，使用这个类的客户程序员不能接触它们。这种特点允许我们改变这些成员而不会干扰已存在的客户代码。我们还可以在运行时改变这些成员对象，动态地改变程序的行为。下面将介绍的继承没有这种灵活性，因为编译器必须在用继承方法创造的类上加入编译时限制。

因为继承在面向对象的程序设计中很重要，所以它常常得到高度重视，并且新程序员可能会产生在任何地方都使用继承的想法。这会形成拙笨和极度复杂的设计。实际上，当创建新类时，程序员应当首先考虑组合，因为它更简单和更灵活。如果采用组合的方法，设计将变得清晰。一旦我们具备一些经验之后，就能很明显地知道什么时候需要采用继承方法。 30

## 1.5 继承：重用接口

对象的思想本身是一种很方便的工具。我们可以将数据和功能通过**概念**封装在一起，使得我们能描述合适的问题空间思想，而不是被强制使用底层机器的用语。通过使用 `class`关键字，这些概念被表示为程序设计语言中的基本单元。

然而，克服许多困难去创造一个类，并随后强制性地创造一个有类似功能的全新的类，似乎并不是一种很好的方法。如果能选取已存在的类，克隆它，然后对这个克隆增加和修改，则是再好不过的事。这是**继承**（*inheritance*）带来的好处，缺点是，如果原来的类（称为**基类**、**超类**或**父类**）被修改，则这个修改过的“克隆”（称为**派生类**、**继承类**或**子类**）也会表现出这些改变。



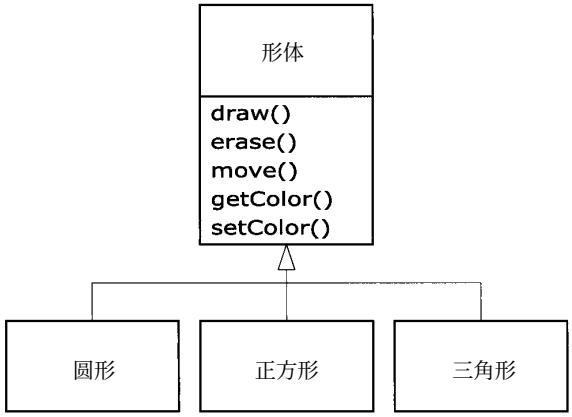

⊖ 这种形式对于大部分图通常是足够详细的，不需要特别指出何处使用聚合或称组合。

（在上面的UML图中，箭头从派生类指向基类。正如你将会看到的，可以有多个派生类。）

类型不仅仅描述一组对象上的约束，而且还描述与其他类型之间的关系。两个类型可以有共同的特性和行为，但是一个类型可以包括比另一个类型更多的特性，也可以处理更多的
31 消息（或对消息进行不同的处理）。继承表示了在基类型和派生类型之间的这种相似性。一个基类型具有所有由它派生出来的类型所共有的特性和行为。程序员创建一个基类型以描述关于系统中的一些对象的思想核心。由这个基类型，我们可以派生出其他类型来表述实现该核心的不同途径。

例如，垃圾再生机器要对垃圾进行分类。这里基类型是“垃圾”，每件垃圾有重量、价值等，并且可以被破碎、被融化或被分解。这样，可以派生出更特殊的垃圾类型，它们可以有另外的特性（瓶子有颜色）或行为（铝可以被压碎，钢可以带有磁性）。另外，有些行为可以不同（纸的价值取决于它的种类和情况）。使用继承，我们可以建立类型的层次结构，在该层次结构中用其类型术语来表述我们需要解决的问题。

第二个例子是经典的“形体”范例，可以用于计算机辅助设计系统或游戏模拟。在这里，基类型是“形体”，每个形体有大小、颜色、位置等。每个形体能被绘制、擦除、移动、着色等。由此，可以派生出特殊类型的形体：圆形、正方形、三角形等，它们中的每一个都有另外的特性和行为。例如，某些形体可以翻转。有些行为可以不同，形体面积的计算。类型层
32 次结构既体现了形体之间的相似性，又体现了它们之间的区别。

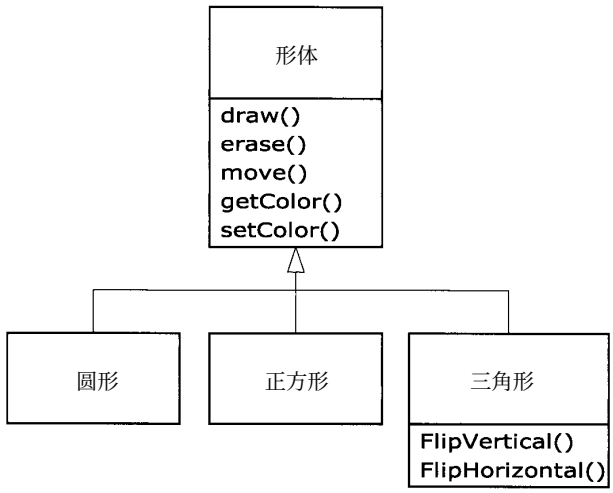



用与问题相同的术语描述问题的解是非常有益的，因为这样我们就无需在问题的描述和解的描述之间使用许多的中间模型。使用对象，类的层次结构就是最初的模型，所以能直接从实际世界中的系统描述进入代码中的系统描述。实际上，使用面向对象设计，人们的困难之一是从开始到结束过于简单。一个已经习惯于寻找复杂解的训练有素的头脑，往往会被问题本来的简单性所难住。

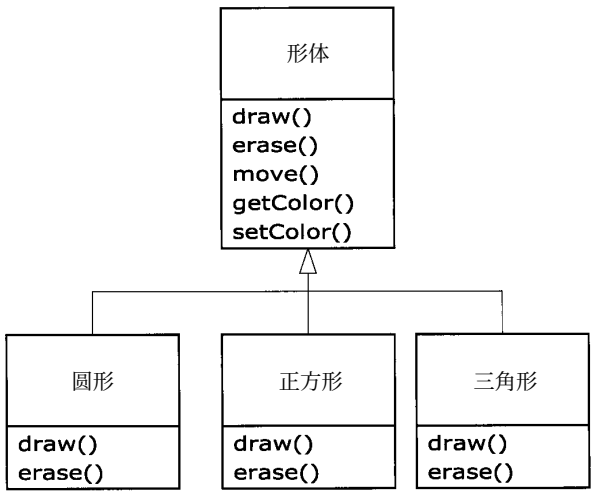
当我们从已经存在的类型来继承时，我们就创造了一个新类型。这个新类型不仅包含那个已经存在的类型的所有成员（虽然私有成员已被隐藏且不可访问），但更重要的是，它复制了这个基类的接口。也就是说，所有能够发送给这个基类对象的消息，也能够发送给这个派生类的对象。因为我们能够根据发送给一个类的消息知道这个类的类型，所以这意味着这个派生类与**这个基类是相同类型的**。在前面的例子中，“圆形是一个形体”。这种通过继承实现
33 类型等价性，是理解面向对象程序设计含义的基本途径之一。

由于基类和派生类有相同接口，因此伴随着接口必然有一些实现。也就是说，当对象接收到一个特定的消息后必定执行一些代码。如果只是简单地继承一个类，而不做其他任何事情，来自基类接口的方法也就进入了派生类。这就意味着，派生类的对象不仅有相同的类型，而且有相同的行为，这一点并不是特别有意义的。

有两种方法能使新派生类区别于原始基类。第一种相当直接，简单地向派生类添加全新的函数。这些新函数不是基类接口的一部分。这意味着，这个基类不能做我们希望它做的事情，所以必须添加函数。继承的这种简单和原始的运用有时就是问题的完美解。但是，我们会进一步看到，基类可能也需要这些添加的函数。在面向对象程序设计中，这种设计的发现和迭代过程会经常发生。




虽然继承有时意味着向接口添加新函数，但这未必真的需要。使新类有别于基类的第二个和更重要的方法是，**改变**已经存在的基类函数的行为，这称为**重载**（*overriding*）这个函数。 34

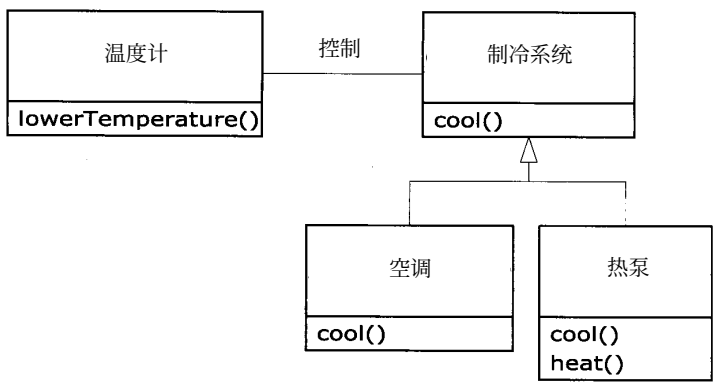



为了重载函数，可以简单地在派生类中创建新定义。相当于说：“我正在使用同一个接口函数，但是我希望它为我的新类型做不同的事情。”

### 1.5.1 is-a关系和is-like-a关系

对于继承有一些争论。继承应当只重载基类（并且不添加基类中没有的新成员函数）吗？这就意味着派生类与基类是**完全**相同的类型，因为它们有相同的接口。结果是，我们可以用派生类的对象代替基类的对象。这被认为是**纯代替**（*pure substitution*），常常被称做**代替原则**（*substitution principle*）。在某种意义上，这是对待继承的理想方法。我们常把基类和派生类之间的关系看做是一个“ *is-a*（是）”关系，因为我们可以说“ 圆形是一个形体 ”。对继承
35 的一种测试方法就是看我们是否可以说这些类有“ *is-a* ”关系，而且还有意义。

有时需要向一个派生类型添加新的接口元素，这样就扩展了接口并创建了新类型。这个新类型仍然可以代替这个基类，但这个代替不是完美的，因为这些新函数不能从基类访问。这可以描述为“ is-like-a（像）”关系；新类型有老类型的接口，但还包含其他函数，所以不能说它们完全相同。以一台空调为例。假设你的房子与制冷的全部控制连线；也就是说，它有一个允许你控制冷却的接口。设想这台空调坏了，用一台热泵代替它，这台热泵既可以制冷又可以制热，这台热泵**就像**一台空调，但它能做更多的事情。因为你的房子的控制系统仅仅是针对制冷功能设计的，所以它仅限于与新对象的制冷部分通信。新对象的接口已经被扩展，而这个已经存在的系统只知道原来的接口，并不知道扩展的部分。




很显然，基类“ 制冷系统 ”是不充分的，应当改为“ 温度控制系统 ”，使它也能包含加热功能。在这一点上，代替原则可用。上图是一个例子，它既可以发生在设计过程中，也可以发生在现实世界中。

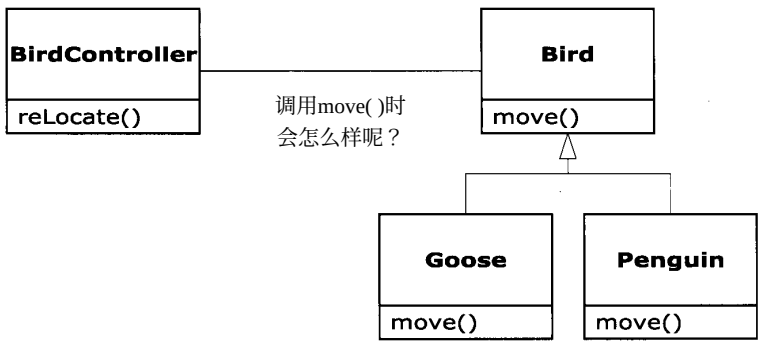
36 当我们考虑代替原则，很容易将这种方法（纯代替）看做是做事情的惟一方法。实际上，如果我们的设计能够采用这种方法，效果也很好。但是，我们还发现有时必须向派生类的接口添加新函数。通过考察，我们发现两种情况都很常见。

## 1.6 具有多态性的可互换对象

当处理类型层次结构时，程序员常常希望不把对象看做是某一特殊类型的成员，而是想把它看做是其基本类型的成员，这样就允许程序员编写不依赖于特殊类型的程序代码。在形体的例子中，函数可以对一般形体进行操作，而不关心它们是圆形、正方形还是三角形。所有的形体都能被绘制、擦除和移动，所以这些函数能简单地发送消息给一个形体对象，而不考虑这个对象如何处理这个消息。

这样，程序代码不受增添新类型的影响，而且增添新类型是扩展面向对象程序来处理新情况最普通的方法。例如，可以派生出形体的一个新的子类型，称为五边形，而不必修改那些处理一般形体的函数。通过派生新的子类型，可以很容易扩展程序，这个能力很重要，因为这会在降低软件维护费用的同时，极大地改善软件设计。

然而，如果试图把派生类型的对象看做是比它们自身更一般的基本类型（圆形看做形体，自行车看做车辆，鸬鹚看做鸟），这里就有一个问题：如果一个函数告诉一个一般的形体去绘制它自己，或者告诉一个一般的车辆去行驶，或者告诉一只一般的鸟去飞翔，则编译器在编译时就不能确切地知道应当执行哪段代码。同样的问题是，消息发送时，程序员并不想知道将执行哪段代码。绘图函数能等同地应用于圆形、正方形或三角形，对象根据它的特殊类型来执行合适的代码。如果增加一个新的子类型，不用修改函数调用，它就可以执行不同的代码。编译器不能确切地知道执行哪段代码，那么它应该怎么办呢？例如，在下图中，**BirdController**对象只是与一般的 **Bird**对象交互，并不知道它们到底是什么类型。这对于**BirdController**是方便的，因为不需要编写专门的代码来确定它正在对哪种 **Bird**工作以及它有什么样的行为。但当忽略专门的 **Bird**类型而调用**move()**时，将发生什么事情呢？会出现正确的行为吗？（**Goose**是跑、是飞、还是游泳？**Penguin**是跑、还是游泳？） 37




在面向对象的程序设计中，答案是非常新奇的：编译器并不做传统意义上的函数调用。由非OOP编译器产生的函数调用会导致与被调用代码的**早捆绑**（*early binding*），对于这一术语，读者可能还没有听说过，因为从来没有想到过它。早捆绑的意思是，编译器会对特定的函数名产生调用，而连接器将这个调用解析为要执行代码的绝对地址。在 OOP中，直到程序运行时，编译器才能确定执行代码的地址，所以，当消息被发送给一般对象时，需要采用其他的方案。

为了解决这一问题，面向对象语言采用**晚捆绑**（*late binding*）的思想。当给对象发送消息时，在程序运行时才去确定被调用的代码。编译器保证这个被调用的函数存在，并执行参数和返回值的类型检查［其中不采用这种处理方式的语言称为**弱类型**（*weakly typed*）语言］，但是它并不知道将执行的确切代码。 38

为了执行晚捆绑， C++编译器在真正调用的地方插入一段特殊的二进制代码。通过使用存放在对象自身中的信息，这段代码在运行时计算被调用函数函数体的地址（这一过程将在第15章中详细介绍）。这样，每个对象就能根据这段二进制代码的内容有不同的行为。当一个对象接收到消息时，它根据这个消息判断应当做什么。

我们可以用关键字 **virtual**声明他希望某个函数有晚捆绑的灵活性。我们并不需要懂得**virtual**使用的机制，但是没有它，我们就不能用 C++进行面向对象的程序设计。在 C++中，

必须记住添加**virtual**关键字，因为根据规定，默认情况下成员函数不能动态捆绑。 virtual函数（虚函数）可用来表示出在相同家族中的类具有不同的行为。这些不同是产生多态行为的原因。

考虑形体的例子，在本章的前面画出过类的家族（所有基于统一接口的类）。为了论述多态性，我们希望编写一段简单的代码，只涉及基类，不涉及类型的具体细节。这段代码是从特定类型信息中分离出来的，编写起来比较简单，理解起来也比较容易。如果有一个新的类型（例如**Hexagon**）通过继承添加进来，那么所写的这段代码将会适用于“ **Shape** ”的这个新类型，就像在已经存在的类型上使用一样。这样，程序就是**可扩展的**（*extensible*）。

如果用C++编写一个函数（很快，读者将会学习如何去写）：

```
void doStuff(Shape& s) {
  s.erase();
  // ...
  s.draw();
}
```

这个函数与任何 **Shape**对话，所以它独立于它正在绘制或擦除的对象的特定类型（‘ **&** ’表示“ 取这个对象的地址，传给 **doStuff( )** ”，但是现在理解这些细节并不重要）。如果在程序
39 的其他部分使用**doStuff( )**函数：

```
Circle c;
Triangle t;
Line l;
doStuff(c);
doStuff(t);
doStuff(l);
```

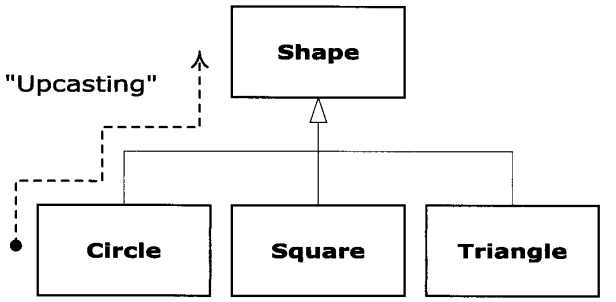
对**doStuff( )**的调用会自动正确工作，而不管调用对象的确切类型。

这真是一个令人惊讶的技术。想一想这行代码：

```
doStuff(c);
```

这里发生的事情是将**Circle**传递给对**Shape**有效的函数。因为**Circle**是一个**Shape**，所以可以由**doStuff( )**处理。这就是说，任何能由 **doStuff( )**发送给**Shape**的消息， **Circle**都能接收。所以它是完全安全的和符合逻辑的。

我们把处理派生类型就如同处理其基类型的过程称为**向上类型转换**（*upcasting*）。“ cast ”一词来自铸造领域，“ up ”一词来自于继承图的典型排列方式，基类型置于顶层，派生类向下层展开。这样，类型向基类型的转换是沿继承图向上移动，即“ 向上类型转换 ”。



面向对象程序在一些地方会包含一些向上类型转换，因为这正是我们从必须了解所处理的是什么具体类型这一桎梏中解脱出来的方式。请看在 **doStuff( )**中的代码：

```
s.erase();
// ...
s.draw();
```

40

注意，这可不是在说：“如果是**Circle**，做这件事，如果是**Square**，做这件事，等等”。如果写这种代码，要检查 **Shape**所有可能的类型，那将是很糟糕的，因为每次添加一种新的 **Shape**，都必须改变代码。这里，我们只是在说：“它是一种形体，我知道它能 **erase( )**和**draw( )**自己，如法操作，注意细节的正确性。”

**doStuff( )**代码最引人注意的地方是它能做正确的事情。对**Circle**调用**draw( )**将执行的代码不同于对**Square**或**Line**调用**draw( )**所执行的代码。但是当**draw( )**的消息发送给一个匿名**Shape**时，正确行为的发生取决于这个 **Shape**的实际类型。这是令人惊奇的，因为，如前所述，当C++编译器编译**doStuff( )**代码时，它并不知道它所处理对象的准确类型。所以以此类推，似乎最终调用的是**Shape**的**erase( )**和**draw( )**的版本，而不是特殊的**Circle**、**Square**或**Line**的版本。然而，因为多态性，一切操作都完全正确。编译器和运行系统可以处理这些细节，我们只需要知道它会这样做和知道如何用它设计程序就行了。如果一个成员函数是 **virtual**的，则当我们给一个对象发送消息时，这个对象将做正确的事情，即便是在有向上类型转换的情况下。

## 1.7 创建和销毁对象

从技术角度， OOP的论域就是抽象数据类型、继承和多态性。但是，其他一些问题也是重要的。本节对这些问题给出综述。

特别重要的是对象创建和销毁的方法。对象的数据存放在何处？如何控制对象的生命期？不同的程序设计语言有不同的行事之道。 C++采取的方法是把效率控制作为最重要的问题，所以它为程序员提供了一个选择。为了最大化运行速度，通过将对象存放在栈中或静态存储区域中，存储和生命期可以在编写程序时确定。栈是内存中的一个区域，可以直接由微处理器在程序执行期间存放数据。在栈中的变量有时称为**自动变量**（*automatic variable*）或**局部变量**（*scoped variable*）。静态存储区域简单说是内存的一个固定块，在程序开始执行以前分配。使用栈或静态存储区，可以快速分配和释放，有时这是有价值的。然而，我们牺牲了灵活性，因为程序员必须在写程序时知道对象的准确数量、生命期和类型。如果程序员正在解决一个更一般的问题，例如计算机辅助设计、仓库管理或者空中交通控制，这就太受限制了。

41

第二种方法是在称为**堆**（*heap*）的区域动态创建对象。用这种方法，可以直到运行时还不知道需要多少个对象，它们的生命期是什么和它们的准确的数据类型是什么。这些决定是在程序运行之中作出的。如果需要新的对象，直接使用 **new**关键字让它在堆上生成。当使用结束时，用关键字**delete**释放。

因为这种存储是在运行时动态管理的，所以在堆上分配存储所需要的时间比在栈上创建存储的时间长得多（在栈上创建存储常常只是一条向下移动栈指针的微处理器指令，另外一条是移回指令）。动态方法做出了一般性的逻辑假设，即对象趋向于更加复杂，所以，为找出存储和释放这个存储的额外开销对于对象的创建没有重要的影响。另外，对于解决一般性的程序设计问题，最大的灵活性是主要的。

另一个问题是对象的生命期。如果在栈上或在静态存储上创建一个对象，编译器决定这个对象持续多长时间并能自动销毁它。然而，如果在堆上创建它，编译器则不知道它的生命期。在C++中，程序员必须编程决定何时销毁此对象。然后使用**delete**关键字执行这个销毁任务。作
42 为一个替换，运行环境可以提供一个称为**垃圾收集器**（*garbage collector*）的功能，当一个对象不再被使用时此功能可以自动发现并销毁这个对象。当然，使用垃圾收集器编写程序是非常方便的，但是它需要所有应用软件能忍受垃圾收集器的存在及垃圾收集的系统开销。这并不符合C++语言的设计需要，因此C++没有包括它，尽管存在用于C++的第三方垃圾收集器。

## 1.8 异常处理：应对错误

从程序设计语言出现开始，错误处理就是最重要的问题之一。因为设计一个好的错误处理方案非常困难，许多语言忽略这个问题，将这个问题转交给库的设计者，而库的设计者往往采取不彻底的措施，即可以在许多情况下起作用，但很容易被绕开，通常是被忽略。大多数错误处理方案的一个主要问题，在于一厢情愿地认为程序员会小心地遵循一些语言本身并不强制要求而是商定好的规范。如果程序员不够小心（这种情况在他们非常匆忙的时候经常发生），这些方案就会很容易被忘记。

**异常处理**（*exception handling*）将错误处理直接与程序设计语言甚至有时是操作系统联系起来。异常是一个对象，它在出错的地方被抛出，并且被一段用以处理特定类型错误的异常处理代码（exception handler）所接收。异常处理似乎是另一个并行的执行路径，在出错的时候被调用。由于它使用一个单独的执行路径，它不需要干涉正常的执行代码。因为不需经常检查错误，代码可以很简洁。另外，一个抛出的异常并不同于一个由函数返回的错误值或为了指出错误条件而由函数设置的标记，后两者可以被忽略，而异常不能被忽略，必须保证它们在某些点上进行处理。最后，异常提供了一个从错误状态中进行可靠恢复的方法。除了从这个程序中退出以外，我们常常还可以作出正确的设置，并且恢复程序执行，这有助于产
43 生更健壮的系统。

异常处理并不是面向对象的一个特性，尽管在面向对象语言中异常通常用一个对象表示。异常处理的出现早于面向对象语言。

异常处理在本卷中介绍得很少且很少使用；第2卷详尽讨论了异常处理。

## 1.9 分析和设计

面向对象范例是一种新的异于前辈的编程思考方式，许多人一开始在学习如何处理一个OOP项目时都会感到非常困难。但是了解到任何事物都被认为是对象，并且学会用面向对象的风格去进一步思考之后，我们就可以开始利用OOP所提供的所有优点创造出“好的”设计。

**方法**（*method*）［通常称为**方法论**（*methodology*）］是一系列的过程和探索，用以降低程序设计问题的复杂性。自从面向对象程序设计出现，已有许多 OOP方法被提出来。本节将让读者感受使用一种方法时应完成什么任务？

尤其在OOP中，方法论是一个充满实验的领域，因此在我们考虑采用一个方法之前，理解它试图要解决什么问题是重要的。这在C++中尤其正确，这种编程语言在表达一个程序时试图减少复杂性（同C相比）。这在实际上可能减缓对更复杂方法论的需求。相反，简单的方法可以满足在C++中处理更大类的问题，在过程型语言中用简单方法处理的问题相比起来则小很多。

认识到术语“方法论”通常太大且承诺太多也是很重要的。设计和编写一个程序时，我们所做的一切就是一个方法。它可能是我们自创的方法，我们可能没有意识到正在创造一种方法，但它确实是我们创造时经历的一个过程。如果它是一个有效的过程，只需要略加调整以和C++配合。如果我们对自己的效率和程序生产方式不满意，就可以考虑采纳一个正式的方法或在许多正式方法中选择某些部分。 44

经历开发过程时，最重要的问题是：不要迷路。这很容易做到。大部分分析和设计方法都是为了解决最大的一些问题。记住，大多数项目并不适合这一点，因此我们通常可以用一个相对小的子集成功地进行分析和设计㊀。但是采用某种过程，不论它怎么有局限，总比一上来就直接编码好得多。

在开发过程中很容易受阻，陷入“分析瘫痪状态”，这种状态中往往由于没有弄清当前阶段的所有小细节而感到不能继续了。记住，不论做了多少分析，总有系统的一些问题直到设计时才暴露出来，并且更多的问题是到编程或直到程序完成运行时才出现。因此，迅速进行分析和设计并对提出的系统执行测试是相当重要的。

这个问题值得强调。因为我们在过程型语言上的历史经验，一个项目组希望在进入设计和实现之前认真处理和理解每个细节，这是值得赞扬的。的确，在构造 DBMS时，需要彻底理解用户的需要。但是 DBMS属于能很好表述和充分理解的一类问题。在许多这种程序中，数据库结构就主要是问题之所在。本章讨论的编程问题属于所谓“不定（ wild card）”（本人的术语）类型，这种问题的解决方法不是将众所周知的解决方案简单地重组，而是包含一个或多个“不定要素”——先前没有较了解的解决方案的要素，为此，需要研究㊁。由于在分析 45
阶段没有充分的信息去解决这类问题，因此在设计和执行之前试图彻底地分析“不定型”问题会造成分析瘫痪。解决“不定型”问题需要在整个循环中反复，且需要冒风险（这是很有意义的，由于是在试图完成一些新颖的且潜在回报很高的事情）。看起来似乎有风险是由于“匆忙”进入初步实现而引起的，但这样反而能降低风险，因为我们正在较早地确定一个特定的方法对这个问题是不是可行的。产品开发也是一种风险管理。

经常有人提到“建立一个然后丢掉”。在OOP中，我们仍可以将一部分丢掉，然而由于代码被封装成类，在第一次迭代中我们将必然生成一些有用的类设计，并且产生一些不必抛弃的关于系统设计的有价值的思想。因此，在问题的第一次快速遍历中不仅要为下一遍分析、设计及实现产生关键的信息，而且为下一遍建立代码基础。

也就是说，如果我们正在考虑的是一个包含丰富细节而且需要许多步骤和文档的方法学，将很难判断什么时候停止。应当牢记我们正在努力寻找的是什么：

(1) 什么是对象？（如何将项目分成多个组成部分？）

(2) 它们的接口是什么？（需要向每个对象发送什么信息？）

只要我们知道了对象和接口，就可以编写程序了。由于各种原因我们可能需要比这些更 46
多的描述和文档，但是我们需要的信息不能比这些更少。

整个过程可以分5个阶段完成，阶段0只是使用一些结构的初始约定。

---

㊀ 一个极好的例子是Martin Fowler所写的专著《*UML Distilled*》(Addison-Wesley, 2000)，该书将复杂的UML过程简化为可管理的子集。

㊁ 我估计这样的项目有一条经验规则：如果不定因素不止一个，在没有创建一个能工作的原型之前，不要计划它将用多长时间和将花费多少。这里的自由度太大了。

### 1.9.1 第0阶段：制定计划

我们必须首先决定在此过程中应当有哪些步骤。这听起来简单（事实上，所有听起来都挺简单的），但是人们常常在开始编码之前没有考虑这一问题。如果计划是“让我们一开始就编码”，那很好（有时，当我们对问题充分理解时，这是合适的）。至少，我们承认这是一个计划。

在这个阶段，我们可能还要决定一些另外的过程结构，但不是全部。可以理解，有些程序员喜欢用“休假方式”工作，也就是在开展他们的工作过程中，没有强制性的结构。“想做的时候就做”。这可能在短时间内是吸引人的，但是我发现，在进程中设立一些里程碑可以帮助集中我们的注意力，激发我们的热情，而不是只注意“完成项目”这个单一的目标。另外，里程碑将项目分成更细的阶段使得风险减小（此外里程碑还提供更多庆祝的机会）。

当我开始研究小说结构时（有一天我也要写小说），我最初是反对结构思想的，我觉得自己在写作时，直接下笔千言就行了。但是，稍后我认识到，在写涉及计算机的文字时，本身结构足够清晰，所以不需要多想。但是我仍然要组织文字结构，虽然在我头脑中是半意识的。即便我们认为自己的计划只是一上来就开始编码，在后续阶段仍然需要不断询问和回答一些问题。

#### 1.9.1.1 任务陈述

47 无论建造什么系统，不管如何复杂，都有其基本的目的，有其要处理的业务，有其所满足的基本需要。通过依次审视用户界面、硬件或系统的特殊细节、算法编码和效率问题，我们将最终找出它的核心，通常简单而又直接。就像来自好莱坞电影的所谓**高层概念**（*high concept*），我们能用一句或两句话表述。这种纯粹的表述是起点。

高层概念相当重要，因为它设定了项目的基调，这是一种任务陈述。我们不必一开始就让它正确（我们也许正处于在项目变得完全清晰之前的最后阶段），但是要不停地努力直至它越来越正确。例如：在一个空中交通指挥系统中，我们可以从关于正在建立的系统的一个高层概念入手：“塔楼程序跟踪飞机”。但是当我们将这一系统收缩以适用于一个非常小的机场时，考虑将发生什么情况；可能只有一个控制人员甚至什么都没有。一个更有用的模型不应当像它描述问题那样多地关注正在创建的解决方案，例如“飞机到达、卸货、维修、重新装货和离开等”。

### 1.9.2 第1阶段：我们在做什么

在上一代程序设计［称为**过程型设计**（*procedural design*）］中，这一阶段称为“建立**需求分析**（*requirements analysis*）和**系统规范说明**（*system specification*）”。这些当然是容易迷路的地方。它们是一些名字很吓人的文档，本身可能就是大项目。当然，它们的目的是好的。需求分析说的是“制定一系列指导方针，我们将利用它了解任务什么时候完成且用户什么时候满足”。系统规范说明指出，“这是程序将做什么（不是现在）以满足需求的一个描述”。需求分析实际上是我们和用户之间的一个合同（即使用户在我们的公司工作或是另一些对象或系统）。系统规范说明是对问题的一个顶层探测，且在一定程度上要说明项目是否能做和将需多少时间。由于这两个问题需要人们的共识（并且因为他们通常会随时间的推移而改变意见），我认为最好将它们尽可能地保持最小限度（理想情况下，是列表和基本的图表）以节省时间。

48 我们可能有其他的限制，如需要将它们扩展为大一些的文档，但是小而简洁的最初文档，可以在由一个动态创建描述的领导者的带领下，通过很少几次头脑风暴 (brainstorming)讨论而得

到。这不仅需要每个人的投入，而且能激励小组中每个成员参与，使他们意见一致。也许，最重要的是，它可以使项目以极大的热情开始。

这一阶段中我们有必要把注意力始终放在核心问题上：确定这个系统要做什么。为此，最有价值的工具是一组所谓的“用例（use case）”。用例指明了系统中的关键特性，它们将展现我们使用的一些基本的类。它们实际上是对类似下述这些问题的描述性回答⊖：

1)“谁将使用这个系统？”

2)“执行者用这个系统做什么？”

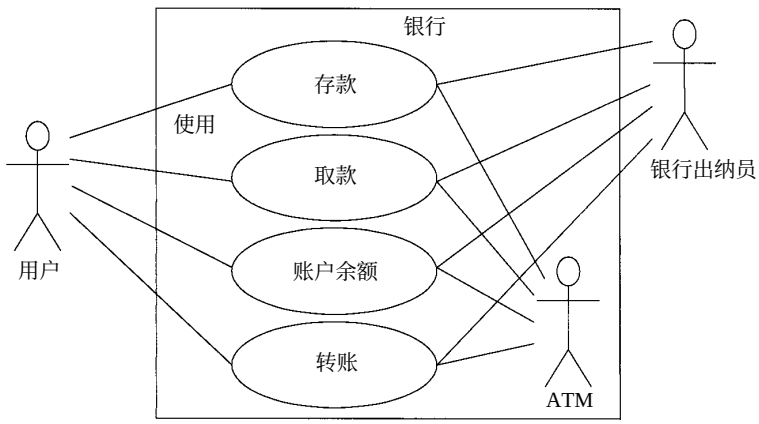
3)“执行者如何用这个系统工作？”

4)“如果其他人也做这件事，或者同一个执行者有不同的目标，该怎么办？（揭示变化）”

5)“当使用这个系统时，会发生什么问题？（揭示异常）”

例如，如果设计一个自动取款机，此系统的一个特定功能方面的用例能够描述这台自动取款机在任何可能情况下的行为。这些“情况”每一个称为**情节**（*scenario*），而用例可以认为是情节的集合。我们可以把情节认为是以“如果 ...系统将怎样？”开头的问题。例如，“如果一个用户在24小时内刚刚存了一张支票，且在此支票之外该账户中没有足够的钱能满足提款要求，这时自动取款机怎么办？”。

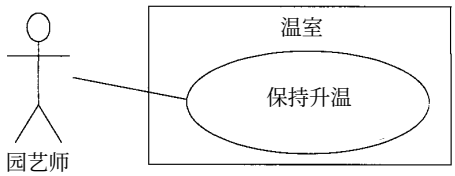
下面的用例图特意进行了简化，以防止我们过早地陷入到系统的实现细节问题中去。 49




每个小人代表一个“执行者（actor）”，它通常是一个人或其他类型的自由代理（甚至可以是其他计算机系统，如“ATM”中的情况）。方框代表系统的边界。椭圆代表用例，是对此系统能完成的有价值工作的描述。在执行者和用例之间的直线代表交互。

只要符合用户的使用感受，系统实际上如何实现并不重要。

用例不必十分复杂，即便底层系统非常复杂。这只是为了表示用户眼中的系统形象。例如：




⊖ 感谢James H Jarrett的帮助。

50 通过确定用户在系统中可能有的所有交互行为，用例就生成了需求规范说明。我们试图找到系统的完整用例，完成之后，我们就得到了系统任务的核心内容。注意力集中在用例上的好处是，它们总是将我们带回到要点部分而不至于留心那些对完成任务无关紧要的问题。也就是说，如果得到了全部用例就可以描述系统并进入到下一个阶段。在最初的尝试中我们很难完全得到特征，但这已经很好了。任何事物随着时间的推移都会自己暴露出来，如果在这一点上就要求系统规范说明完美将会永远止步不前。

如果遇到了困难，我们可以通过使用一个近似工具启动这个阶段：用很少的段落描述此系统，然后寻找名词和动词。名词往往意味着执行者、用例的上下文（如“休息室”）或在用例中使用的物品。动词往往意味着执行者和用例之间的交互行为，表示用例中的特定步骤。我们还将发现名词和动词在设计阶段将对应产生对象和消息（注意用例描述子系统间的交互，因此“名词和动词”技术只能作为集体讨论工具，因为它不生成用例）[⊖]。

用例和执行者之间的边界能指出用户界面的存在，但不能定义这样的用户界面。为了定义和创建用户界面，可参看 Larry Constantine和Lucy Lockwood所著的《Software for Use》(Addison Wesley Longman, 1999) 或访问*www.ForUse.com*。

虽然这有点像魔术，但此时进行某种基本的进度安排是重要的。我们现在有了创建目标的总体概念，所以我们可能产生它需要多长时间的概念。这里涉及大量因素。如果估算了一个长时间表，则公司可能决定不创建它（并把资源用在更合理的项目上去，这是好事）。或者管理人员可能已经决定这个项目应当花多少时间，并且试图改变我们做出的估计。但是最好
51 一开始有一个准确的时间表，解决早期决心的问题。已经有大量的努力以产生精确建立时间表的技术（就像预测股票市场的技术），然而，最好的方法或许是依靠我们的经验和直觉。得到实际上将花多少时间的估计，加倍，再加上百分之十。我们的直觉也许是对的，我们能及时得到能用的产品。“加倍”将使产品更好，加百分之十用于最后的润色和细化[⊜]。然而，我们需要解释它，克服抱怨和当我们拿出这个时间表时发生种种事情，最终完成这一问题。

### 1.9.3 第2阶段：我们将如何建立对象

在这一阶段，我们必须做出设计，描述这些类和它们如何交互。确定类和交互的出色技术是**类职责协同**（*Class-Responsibility-Collaboration, CRC*）卡片。此技术的部分价值是它非常简单：只要有一组 3到5英寸的空白卡片，在上面书写。每张卡片描述一个类，在卡片上写的内容是：

(1) 类的名字。这很重要，因为名字体现了类行为的本质，所以有一目了然的作用。

(2) 类的职责：它应当做什么。通常，它可以仅由成员函数的名字陈述（因为在好的设计中，这些名字应当是描述性的），但并不产生其他的注记。如果需要开始这个过程，请从一个
52 懒程序员的立场看这个问题：你希望有什么样的对象魔术般地出现，把你的问题全部解决？

(3) 类的协同：它与其他类有哪些交互？“交互”是非常宽泛的术语。它可以是一些已经

---

⊖ 更多有关用例的内容可以在Schneider & Winters所写的专著《*Applying Use Cases*》(Addison-Wesley, 1998)和Rosenberg所写的专著《*Use Case Driven Object Modeling with UML*》(Addison-Wesley, 1999)中找到。

⊜ 我个人观点后来已经变了。加倍和增加百分之十将给出相当准确的估计（假设这里没有太多的不定要素），但是我们仍然需要勤奋工作，以及时完成。如果我们希望时间真的花这么长，并且在这个过程中得到乐趣，我认为，正确的增加是3到4倍。

存在的其他对象对这个类的对象提供的服务。协同还应当考虑这个类的观众。例如如果创建了**Firecracker**（鞭炮），那么谁将观察它，是**Chemist**（药剂师）还是**Spectator**（观众）？前者希望知道鞭炮由什么化学成分组成，后者对鞭炮爆炸后的颜色和形状有反应。

我们可能想让卡片更大一些，因为我们希望从中得到全部信息，但是它们是非常小的，这不仅能保持我们的类小，而且能防止过早地陷入过多的细节。如果一张小卡片上放不下类所需要的信息，那么这个类就太复杂了（或者是考虑过细了，或者应当创建多个类）。理想的类应当一目了然。CRC卡片的思想是帮助我们找到设计的第一印象，使得我们能得到总体概念，然后精炼我们的设计。

CRC卡片的最大的好处之一是在交流中。在一个组中，最好实时进行交流，而不是用计算机。每个人负责几个类（起初它们没有名字或其他信息）。每次只解决一个情节，决定发送什么消息给不同的对象以满足每个情节，这样就能作出一个比较形象的对问题的模拟。当我们经历了这个过程后，就会找出我们所需要的类以及它们的职责和协同，这样，我们同时也填写好这些卡片。当我们完成所有用例后，就有了一个相当完整的设计的第一印象。

在我开始用CRC卡片之前，当提出最初的设计时，我最成功的咨询经验就是站在一个没有OOP经验的项目组前，在白板上描述对象。我们讨论对象应当如何互相通信，擦除其中的一些，用其他的对象替换它们。实际上我是在白板上管理所有的“ CRC卡片 ”。项目组（他们知道项目的目标）真正地在做这个设计，他们“ 拥有 ”这个设计，而不是获得既成的设计。我所做的 53
所有事情就是通过提问正确的问题，提炼这些假设，并且从项目组得到反馈，修改这些假设来指导这个过程。这个过程的真正好处是项目组学习了如何做面向对象的设计，不是通过复审抽象的例子，而是通过在一个设计上工作，这对于他们是最有兴趣的。

制作了一组CRC卡片之后，我们可能希望用 UML[⊖]创建这个设计的更形式化的描述。我们并不非要用UML，但它可能有帮助，特别是如果我们要将一个图表挂在墙上，让大家一起思考时，这是一个很好的想法。除了 UML之外的另一选择是对象及其接口的文字描述，这或许依赖于我们的程序设计语言，也就是代码本身[⊜]。

UML还提供了另外一种图形符号来描述系统的动态模型。在一个系统或子系统的状态转换占主导地位，以至于它们需要自己的图表的情况下，这是有帮助的（例如在控制系统中）。我们可能还需要描述数据结构，因为系统或子系统中数据结构是重要因素（例如数据库）。

当已经描述了对象及其接口后，第 2阶段就要完成了。这时已经知道了对象中的大多数，通常会有对象漏掉，直到第3阶段才被发现。这没问题。我们关心的是最终能找到所有的对象。在这个阶段较早地发现它们是好的。因为 OOP提供了充分的结构，所以如果我们稍迟发现它们也可以。事实上，对象设计可能在程序设计全过程的五个阶段中都会发生。

#### 1.9.3.1 对象设计的五个阶段

对象的设计生命期不仅仅限于写程序的时间。实际上，它出现在一系列阶段上。接受这种观点很有好处，因为我们不再期望设计立刻尽善尽美，而是认识到，对对象做什么和它应 54
当像什么的理解，会随着时间的推移而呈现。这个观点也适用于不同类型程序的设计。特殊类型程序的模式是通过一次又一次地求解问题而形成的（设计模式在第 2卷介绍）。同样，对象有自己的模式，通过理解、使用和重用而形成。

---

⊖ 对于初学者，我推荐上面提到过的专著《*UML Distilled*》。

⊜ Python (www.Python.org)常常被用做“ 可执行伪代码 ”。