第1章　对象的演化

计算机革命起源于一台机器，程序设计语言也源于一台机器。

然而计算机并不仅仅是一台机器，它是心智放大器和另一种有表述能力的媒体。这一点使它不很像机器，而更像我们大脑的一部分，更像其他有表述能力的手段，例如写作、绘画、雕刻、动画制作或电影制作。面向对象的程序设计是计算机向有表述能力的媒体发展中的一部分。

本章将介绍面向对象程序设计（OOP）的基本概念，然后讨论OOP开发方法，最后介绍使程序员、项目和公司使用面向对象程序设计方法而采用的策略。

本章是一些背景材料，如果读者急于学习这门语言的具体内容，可以跳到第 2章，然后再回过头来学习本章。

1.1　基本概念

C++包含了比面向对象程序设计基本概念更多的内容，读者应当在学习设计和开发程序之前先理解该语言所包含的基本概念。

1.1.1　对象：特性+行为[1]

第一个面向对象的程序设计语言是60年代开发的Simula-67。其目的是为了解决模拟问题。典型的模拟问题是银行出纳业务，包括出纳部门、顾客、业务、货币的单位等大量的“对象”。把那些在程序执行期间除了状态之外其他方面都一样的对象归在一起，构成对象的“类”，这就是“类”一词的来源。

类描述了一组有相同特性（数据元素）和相同行为（函数）的对象。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。这些已存在的数据类型的设计动机仅仅是为了描述机器的存储单元。程序员可以通过增添他所需要的新数据类型来扩展这个程序设计语言。该程序设计系统欢迎创建、关注新的类，对它们进行与内部类型一样的类型检查。

这种方法并不限于去模拟具体问题。尽管不是所有的人都同意，但大部分人相信，任何程序都模拟所设计系统。OOP技术能很容易地将大量问题归纳成为一个简单的解，这一发现产生了大量的OOP语言，其中最著名的是Smalltalk——C++ 之前最成功的OOP语言。

抽象数据类型的创建是面向对象程序设计中的一个基本概念。抽象数据类型几乎能像内部类型一样准确工作。程序员可以创建类型的变量（在面向对象程序设计中称为“对象”或“实例”）并操纵这些变量（称为发送“消息”或“请求”，对象根据发来的消息知道需要做什么事情）。

1.1.2　继承：类型关系

类型不仅仅说明一组对象上的约束，还说明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，一个类型可能包括比另一个类型更多的特性，也可以处理更多的消息

[1] 这一描述部分引自我对《The Tao of Objects》（Gary Entsminger著）一书的介绍。

(或对消息进行不同的处理)。继承表示了基本类型和派生类型之间的相似性。一个基本类型具有所有由它派生出来的类型所共有的特性和行为。程序员创建一个基本类型以描述系统中一些对象的思想核心。由这个基本类型派生出其他类型，表达了认识该核心的不同途径。

例如，垃圾再生机要对垃圾进行分类。这里基本类型是“垃圾”，每件垃圾有重量、价值等等，并且可以被破碎、融化或分解。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，钢可以被磁化）。另外，有些行为可以不同(纸的价值取决于它的种类和状态)。程序员可以用继承建立类的层次结构，在该层次结构中用类型术语来表述他需要解决的问题。

第二个例子是经典的形体问题，可以用于计算机辅助设计系统或游戏模拟中。这里基本类型是“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、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（计算形体的面积）。类型层次结构既体现了形体间的类似，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，这样，从问题描述到解的描述之间就不需要很多中间模型（程序语言解决大型问题，就需要中间模型）。面向对象之前的语言，描述问题的解不可避免地要用计算机术语。使用对象术语，类型层次结构是主要模型，所以可以从现实世界中的系统描述直接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，往往会被问题的简单性难住。

1.1.3 多态性

当处理类型层次结构时，程序员常常希望不把对象看作是某一特殊类型的成员，而把它看作基本类型成员，这样就可以编写不依赖于特殊类型的代码。在形体例子中，函数可以对一般形体进行操作，而不关心它们是圆、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，新添类型不影响原来的代码，这是扩展面向对象程序以处理新情况的最普通的方法。例如，可以派生出形体的一个新的子类，称为五边形，而不必修改那些处理一般形体的函数。通过派生新子类，很容易扩展程序，这个能力很重要，因为它极大地减少了软件维护的花费。(所谓“软件危机”正是由软件的实际花费远远超出人们的想象而产生的。)

如果试图把派生类型的对象看作它们的基本类型（圆看作形体，自行车看作车辆，鸬鹚看作鸟），就有一个问题：如果一个函数告诉一个一般形体去绘制它自己，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，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、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，不用修改函数调用，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？

在面向对象的程序设计中，答案是巧妙的。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引起与被调用代码的“早捆绑”，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。早捆绑意味着编译器对特定的函数名产生调用，而连接器确定调用执行代码的绝对地址。对于 OOP，在程序运行之前，编译器不确定执行代码的地址，所以，当消息发送给一般对象时，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”的思想。当给对象发送消息时，在程序

运行之前不去确定被调用的代码。编译器保证这个被调用的函数存在，并完成参数和返回值的类型检查，但是它不知道将执行的准确代码。

为了实现晚捆绑，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的地址（这一问题将在第 14章中详细介绍）。这样，每个对象就能根据一个指针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。

程序员可以用关键字 virtual表明他希望某个函数有晚捆绑的灵活性，而并不需要懂得virtual的使用机制。没有它，就不能用 C++做面向对象的程序设计。 Virtual函数（虚函数）表示允许在相同家族中的类有不同的行为。这些不同是引起多态行为的原因。

1.1.4 操作概念：OOP程序像什么

我们已经知道，用C 语言编写的过程程序就是一些数据定义和函数调用。要理解这种程序的含义，程序员必须掌握函数调用和函数实现的本身。这就是过程程序需要中间表示的原因。中间表示容易引起混淆，因为中间表示的表述是原始的，更偏向于计算机，而不偏向于所解决的问题。

因为 C++ 向 C 语言增加了许多新概念，所以程序员很自然地认为， C++程序中的main()会比功能相同的 C 程序更复杂。但令人吃惊的是，一个写得很好的C++程序一般要比功能相同的C程序更简单和容易理解。程序员只会看到一些描述问题空间对象的定义（而不是计算机的描述），发送给这些对象的消息。这些消息表示了在这个空间的活动。面向对象程序设计的优点之一是通过阅读，很容易理解代码。通常，面向对象程序需要较少的代码，因为问题中的许多部分都可以用已存在的库代码。

1.2 为什么C++会成功

C++能够如此成功，部分原因是它的目标不只是为了将 C语言转变成OOP语言（虽然这是最初的目的），而且还为了解决当今程序员，特别是那些在 C语言中已经大量投资的程序员所面临的许多问题。人们已经对OOP语言有了这样传统的看法：程序员应当抛弃所知道的每件事情并且从一组新概念和新文法重新开始，他应当相信，最好丢掉所有来自过程语言的老行装。从长远角度看，这是对的。但从短期角度看，这些行装还是有价值的。最有价值的可能不是那些已存在的代码库（给出合适的工具，可以转变它），而是已存在的头脑库。作为一个职业 C程序员，如果让他丢掉他知道的关于 C的每一件事，以适应新的语言，那么，几个月内，他将毫无成果，直到他的头脑适应了这一新范例为止。如果他能调整已有的 C知识，并在这个基础上扩展，那么他就可以继续保持高效率，带着已有的知识，进入面向对象程序设计的世界。因为每个人有他自己的程序设计模型，所以这个转变是很混乱的。因此， C++成功的原因是经济上的：转变到OOP需要代价，而转变到C++所花的代价较小。

C++的目的是提高效率。效率取决于很多东西，而语言是为了尽可能地帮助使用者，尽可能不用武断的规则或特殊的性能妨碍使用者。 C++成功是因为它立足于实际：尽可能地为程序员提供最大便利。

1.2.1 较好的C

即便程序员在C++环境下继续写C代码，也能直接得到好处，因为C++堵塞了C语言中的一

些漏洞，并提供更好的类型检查和编译时的分析。程序员必须先说明函数，使编译器能检查它们的使用情况。预处理器虚拟删除值替换和宏，这就减少了查找疵点的困难。C++有一个性能，称为references(引用)，它允许对函数参数和返回值的地址进行更方便的处理。函数重载改进了对名字的处理，使程序员能对不同的函数使用相同的名字。另外，名字空间也加强了名字的控制。许多性能使C的更安全。

1.2.2 采用渐进的学习方式

与学习新语言有关的问题是效率的问题。所有公司都不可避免地因软件工程师学习新语言而突然降低了效率。C++是对C的扩充，而不是新的文法和新的程序设计模型。程序员学习和理解这些性能，逐渐应用并继续创建有用的代码。这是C++成功的最重要的原因之一。

另外，已有的C代码在C++中仍然是有用的，但因为C++编译器更严格，所以，重新编译这些代码时，常常会发现隐藏的错误。

1.2.3 运行效率

有时，以程序执行速度换取程序员的效率是值得的。假如一个金融模型仅在短期内有用，那么快速创建这个模型比所写程序能更快速执行重要。很多应用程序都要求有一定的运行效率，所以C++在更高运行效率时总是犯错。C程序员非常重视运行效率，这让他们认为这个语言不太庞大，也不太慢。产生的代码运行效率不够时，程序员可以用C++的一些性能做一些调整。

C++不仅有与C相同的基本控制能力（和C++程序中直接写汇编语言的能力），非正式的证据指出，面向对象的C++程序的速度与用C写的程序速度相差在±10%之内，而且常常更接近。用OOP方法设计的程序可能比C的对应版本更有效。

1.2.4 系统更容易表达和理解

为适合于某问题而设计的类当然能更好地表达这个问题。这意味着写代码时，程序员是在用问题空间的术语描述问题的解（例如“把锁链放在箱子里”），而不是用计算机的术语，也就是解空间的术语，描述问题的解（例如“设置芯片的一位即合上继电器”）。程序员所涉及的是较高层的概念，一行代码能做更多的事情。

易于表达所带来的另一个好处是易于维护。据报道，在程序的整个生命周期中，维护占了花费的很大部分。如果程序容易理解，那么它就更容易维护，还能减少创建和维护文档的花费。

1.2.5 “库”使你事半功倍

创建程序的最快方法是使用已经写好的代码：库。C++的主要目标是让程序员能更容易地使用库，这是通过将库转换为新数据类型（类）来完成的。引入一个库，就是向该语言增加一个新类型。编译器负责这个库如何使用，保证适当的初始化和清除，保证函数被正确地调用，因此程序员的精力可以集中在他想要这个库做什么，而不是如何做上。

因为程序的各部分之间名字是隔离的，所以程序员想用多少库就用多少库，不会有像C语言那样的名字冲突。

• 模板的源代码重用

一些重要的类型要求修改源代码以便有效地重用。模板可以自动完成对代码的修改，因而

是重用库代码特别有用的工具。用模板设计的类型很容易与其他类型一起工作。因为模板对程序员隐藏了这类代码重用的复杂性，所以特别好用。

1.2.6 错误处理

在C语言中，错误处理声名狼藉。程序员常常忽视它们，对它们束手无策。如果正在建大而复杂的程序，没有什么比让错误隐藏在某处，且不能指出它来自何处更糟的了。 C++的异常处理（见第17章的内容）保证能检查到错误并进行处理。

1.2.7 大程序设计

许多传统语言对程序的规模和复杂性有自身的限制。例如， BASIC对于某些类型的问题能很快解决，但是如果这个程序有几页纸长，或者超出该语言的正常解题范围，那么它可能永远算不出结果。C语言同样有这样的限制，例如当程序超过 50 000行时，名字冲突就开始成为问题。简言之，程序员用光了函数和变量名。另一个特别糟糕的问题是如果 C语言中存在一些小漏洞 —— 错误藏在大程序中，要找出它们是极其困难的。

没有清楚的文字告诉程序员，什么时候他的语言会失效，即便有，他也会忽视它们。他不说“ 我的BASIC程序太大，我必须用 C重写 ”，而是试图硬塞进另外几行，增加额外的性能。所以额外的花费就悄悄增加了。

设计C++的目的是为了辅助大程序设计，也就是说，去掉小程序和大程序之间复杂性的分界。当程序员写hello-world类实用程序时，他确实不需要用OOP、模板、名字空间和异常处理，但当他需要的时候，这些性能就有用了。而且，编译器在排除错误方面，对于小程序和大程序一样有效。

1.3 方法学介绍

所谓方法学是指一组过程和启发式，用以减少程序设计问题的复杂性。在 OOP中，方法学是一个有许多实践的领域。因此，在程序员考虑采用某一方法之前，了解该方法将要解决的问题是很重要的。对于C++，有一点是确实的：它本身就是希望减少程序表达的复杂性。从而不必用更复杂方法学。对于用过程语言的简单方法所不能处理的大型问题，在 C++中用一些简单的方法就足够了。

认识到“ 方法学 ”一词含义太广是很重要的。实际上，设计和编写程序时，无论做什么都在使用一种方法。只不过因为它是程序员自己的方法而没有意识到。但是，它是程序员编程中的一个过程。如果过程是有效的，只需要用 C++做很小的调整。如果程序员对他的效率和调整程序的方法不满意，他可以考虑采用一种更正式的方法。

1.3.1 复杂性

为了分析复杂性，先假设：程序设计制定原则来对付复杂性。

原则以两种方式出现，每种方式都被单独检查。

1) 内部原则体现在程序自身的结构中，机灵而有见解的程序员可以通过程序设计语言的表达方式了解这种内部原则。

2) 外部原则体现在程序的源信息中，一般被描述为“ 设计文档 ”（不要与产品文档混淆）。

我认为，这两种形式的原则互相不一致：一个是程序的本质，是为了让程序能工作而产生

的，另一个是程序的分析，为了将来理解和维护程序而产生的。创建和维护都是程序生命期的基本组成部分。有用的程序设计方法把两者综合为最合适的方式，而不偏向任何一方。

1.3.2 内部原则

程序设计的演化（C++只是其中的一步）从程序设计模型强加于内部开始，也就是允许程序员为内存位置和机器指令取别名。这是数字机器程序设计的一次飞跃，带动了其他方面的发展，包括从初级机器中抽象出来，向更方便地解决手边问题的模型发展。不是所有这些发展都能流行，起源于学术界并延伸进计算机世界的思想常常依赖于所适应的问题。

命名子程序的创建和支持子程序库的连接技术在50年代向前飞跃发展，并且孕育出了两个语言，它们在当时产生了巨大冲击，这就是为科学工作者使用的 FORTRAN（FORmula-TRANslation）和为商业者使用的COBOL（COmmon Business-Oriented Language）。纯计算机科学中很成功的语言是Lisp（List-Processing），而面向数学的语言应当是APL（A Programming Language）。

这些语言的共同特点是对过程的使用。 Lisp和APL的创造专注于语言的高雅 —— 语言的“ mission语句 ”嵌入在处理所有任务情况的引擎中。 FORTRAN和COBOL的创造是为了解决专门的问题，当这些问题变得更复杂，有新的问题出现时，它们又得到了发展。甚至它们进入衰退期后，仍在发展：FORTRAN和COBOL的版本都面向对象进行了扩充（后时髦哲学的基本原则是：任何具有自己独特生活方式的组织，其主要目标就是使这种生活方式永存）。

命名子程序在程序设计中起了重要作用，语言的设计围绕着这一原则，特别是 Algol和Pascal。同时另外一些语言也出现了，它们成功地解决了程序设计的一些子集问题，并将它们有序排列。最有趣的两个语言是Prolog和FORTH。前者是围绕着推理机而建立的（在其他语言中常常称作库）。后者是一个可扩充语言，允许程序员重新形成这个语言，以适应所解决的问题，观念上类似于面向对象程序设计。 FORTH还可以改变语言，因而很难维护，并且是内部原则概念最纯正的表达，它强调的是问题一时的解，而不是对这个解的维护。

人们还创造了其他许多语言，以解决某一部分的程序设计问题。通常，这些语言以特定的目标开始。例如，BASIC（Beginners All-purpose Symbolic Instruction Code）是在60年代设计的，目的是使程序设计对初学者更简单。 APL的设计是为了数学处理。两种语言都能够解决其他问题，而关键在于它们是否是这些问题集合最理想的解。有一句笑话是，“ 带着锤子三年，看什么都是钉子 ”。这反映了根本的经济学真理：如果我们只有 BASIC或APL语言，特别是，当最终期限很短且这个解的生命期有限时，它就是我们问题最好的解。

然而，最终考虑两个因素：复杂性的管理和维护（将在下一部分讨论）。即这种语言首先是为某一领域开发的，而程序员又不愿花很长时间来熟悉这门语言，其结果只能使程序越来越长，使手头的问题屈服于语言。界限是模糊的：谁能说什么时候您的语言会使您失望呢？这不是马上就出现的。

问题的解开始变长，并且对于程序员更具挑战性。为了知道语言大概的限制，你得更聪明，这种聪明变成了一种标准，也就是“ 为了使该语言工作而努力 ”。这似乎是人类的操作方式，而不是遇到缺陷就抱怨，并且不再称它为缺陷。

最终，程序设计问题对于求解和维护变得太困难了，即求得的解太昂贵了。人们最终明白了，程序的复杂性超出了我们能够处理的程度。尽管一大类程序设计要求开发期间去做大部分工作并创建要求最小维护的解（或者简单地丢掉这个解，或者用不同的解替换它），但这只是问题的一部分。一般情况是，我们把软件看作是为人们提供服务的工具。如果用户的需要变化

了，服务就必须随着变化。这样，当第一版本开始运行时，项目并没有结束。项目是一个不断进化的生命体。程序的更新变成了一般程序设计问题的一个部分。

1.3.3 外部原则

为了更新和改善程序，需要更新思考问题的方法。它不只是“我们如何让程序工作”，而是“我们如何让程序工作并且使它容易改变”。这里就有一个新问题：当我们只是试图让程序工作时，我们可以假设开发组是稳定的（总之，我们可以希望这样），但是，如果我们正在考虑程序的整个生命期，就必须假设开发组成员会改变。这意味着，新组员必须以某种方式学习原程序的要点，并与老组员互相通讯（也许通过对话）。这样，该程序就需要某种形式的设计文档。

因为只想让程序工作，文档并不是必需的，所以还没有像由程序设计语言强加于程序那样的、强加于创建文档的规则。这样，如果要求文档满足特定的需要，就必须对文档强加外部原则。文档是否“工作”，这很难确定（并且需要在程序一生中验证），因此，对外部原则“最好”形式的争论.比对“最好”程序设计语言的争论更激烈。

决定外部原则时，头脑中的重要问题是“我准备解决什么问题”。问题的根本就是上面所说的“我们如何让它工作和使它容易改变”。然而，这个问题常常有多种解释：它变成了“我如何才能与FoobleBlah文档规范说明一致，以使政府会为此给我拨款”。这样，外部原则的目的是为了建立文档，而不是为了设计好的、可维护的程序。文档竟然变得比程序本身更重要了。

被问到未来一般和特殊的计算的方向时，我会从这样的问题开始：哪种解花费较少？假设这个解满足需要，价格的不同足以使程序员放弃他当前做事情的习惯方式吗？如果他的方法包括存储在项目分析和设计过程中所创建的每个文档，并且包括当项目进化时维护这些文档，那么当项目更新时，他的系统将花费很大，但是它能使新组员容易理解（假设没有那么多的使人害怕阅读的文档）。这样创建和维护方法的花费会和它打算替代方法的花费一样多。

外部结构系列的另一个极端是最小化方法。为完成设计而进行足够的分析，然后丢掉它们，使得程序员不再花时间和钱去维护它；为开始编码而做足够的设计，然后丢掉这个设计，使得程序员不再花时间和钱去维护这些文档；然后使得代码是一流的和清晰的，代码中只需要最少的注释。为了使新组员快速参与项目，代码连同注释就足够了。因为在所有这些乏味的文档上，新组员只需花费很少的时间（总之，没有人真地理解它们），所以他能较快地参与工作。

即便不维护文档，丢掉文档也不是最好的办法，因为这毕竟是程序员所做的有效工作。某些形式的文档通常是必须的（参看本章后面的描述）。

1. 通讯

对于较大的项目，期望代码像文档一样充分是不合理的，尽管我们在实际中常常这样期望。但是，代码包含了我们实际上希望外部原则所产生的事物的本质：通讯。我们只是希望能与改进这个程序的新组员通讯就足够了。但是，我们还想使花费在外部原则上的钱最少，因为最终人们只为这个程序所提供的服务付钱，而不是为它后面的设计文档付钱。为了真正有用，外部原则应当做比只产生文档更多的事情——它应当是项目组成员在创建设计时为了讨论问题而采用的通讯方法。理想的外部原则目标是使关于程序分析和设计的通讯更容易。这对于现在为这个程序而工作的人们和将来为这个程序而工作的人们是有帮助的。中心问题不只是为了能通讯，而为了产生好的设计。

人们（特别是程序员）被计算机吸引（由于机器为他们做工作），出于经济原因，要求开

发者为机器做大量工作的外部原则似乎从一开始就注定要失败。成功的方法（也就是人们习惯的方法）有两个重要的特征：

1) 它帮助人们进行分析和设计。这就是，用这种方法比用别的方法对分析和设计中的思考和通讯要容易得多。目前的效率和采用这种方法后的效率应当明显不同。否则，人们可能还留在原地。还有，它的使用必须足够简单，不需用手册。当程序员正在解决问题时，要考虑简单性，而不管他适用于符号还是技术。

2) 没有短期回报，就不会加强投资。在通向目标的可见的进展中，没有短期回报，人们就不会感到采用一种方法能使效率提高，就会回避它。不能把这个进展误认为是从一种中间形式到另一种中间形式的变换。程序员可以看到他的类，连同类之间互相发送的消息一起出现。为某人创造一种方法，就像武断的约束，因为它是简单的心理状态：人们希望感到他们正在做创造性的工作，如果某种方法妨碍他们而不是帮助他们飞快地接近目标，他们将设法绕过这种方法。

2. 量级

在方法学上反对我的观点之一是：“好了，您能够侥幸成功是因为您正在做的小项目很短。”听众对“小”的理解因人而异。虽然这种看法并不全对，但它包含一个正确的核心：我们所需要的原则与我们正在努力解决问题的量级有关。小项目完全不需要外部原则，这不同于个别程序员正在解的生命期问题的模式。涉及很多人的大项目会使人们之间有一些通讯，所以必须使通讯具有形式化方法，以使通讯有效和准确。

麻烦的是介于它们之间的项目。它们对外部原则的需要程度可能在很大程度上依赖于项目的复杂性和开发者的经验。确实，所有中等规模的项目都不需要忠实于成熟的方法，即产生许多报告和很多文档。一些项目也许这样做，但许多项目可以侥幸成功于“方法学简化”（代码多而文档少）。我们面前的所有方法学的复杂性可以减少到 80%～20%的（或更少的）规则。我们正在被方法学的细节淹没，在所解决的程序设计问题中，可能只有不足 20%的问题需要这些方法学。如果我们的设计是充分的，并且维护也不可怕，那么我们也许不需要方法学或不全部需要它。

3. OOP是结构化的吗

现在提出一个更有意义的问题。为了使通讯方便，假设方法学是需要的。这种关于程序的元通讯是必须的，因为程序设计语言是不充分的——它太原始，趋向于机器范例，对于谈论问题不很有用。例如，过程程序设计方法要求用数据和变换数据的函数作为术语谈论程序。因为这不是我们讨论实际问题的方法，所以必须在问题描述和解描述之间翻译来翻译去。一旦得到了一个解描述并且实现了它，以后无论何时对这个解描述做改变就要对问题描述做改变。这意味着必须从机器模式返回问题空间。为了得到真正可维护的程序，并且能够适应问题空间上的改变，这种翻译是必须的。投资和组织的需要似乎要求某种外部原则。过程程序的最重要的方法学是结构化技术。

现在考虑，是否解空间上的语言可以完全脱离机器模式？是否可以强迫解空间使用与问题空间相同的术语？

例如，在气候控制大楼中的空气调节器就变成了气候调节程序的空气调节器，自动调温器变成了自动调温程序，等等。（这是按直觉做的，与OOP不一致）。突然，从问题空间到解空间的翻译变成了次要问题。可以想象，在程序分析、设计和实现的每一阶段，能使用相同的术语学、相同的描述，这样，这个问题就变成了“如果文档（程序）能够充分地描述它自身，我们仍然需要关于这个文档的文档吗？”如果OOP做它所主张的事情，程序设计的形式就变成了这

样：在结构化技术中所遇到的困难在新领域中可能不复存在了。

这个论点也为一个思想实验所揭示。假设程序员需要写一些小实用程序，例如能在文本文件上完成一个操作的程序（就像在第6章的后几页上可找到的那样），它们要程序员花费几分钟，最困难的要花费几小时去写。现在假设回到50年代，这个项目必须用机器语言或汇编语言来写，使用最少的库函数，它需要许多人几星期或几个月的时间。在 50年代需要大量的外部原则和管理，现在不需要了。显然，工具的发展已经极大地增加了我们不用外部原则解决问题的复杂性（同样很显然，我们将发现的问题也更加复杂）。

这并不是说可以不需要外部原则，有用的OOP外部原则解决的问题与有用的过程程序设计外部原则所解决的问题不同，特别是，OOP方法的目标首先必须是产生好的设计。好设计不仅要促进重用，而且它与项目的各级开发者的需要是一致的。这样，开发者就会更喜欢采用这样的系统。让我们基于这些观点考虑OOP设计方法中的一些问题。

1.3.4 对象设计的五个阶段

对象的设计不限于写程序的时期，它出现在一系列阶段。有这种观点很有好处，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当像什么的理解是随着时间的推移而产生的。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[1]。同样，对象有自己的模式，通过理解、使用和重用而形成。

下面是描述，不是方法。它简直就是对象期望的设计出现时的观察结果。

1) 对象发现　这个阶段出现在程序的最初分析期间。可以通过寻找外部因素与界线、系统中的元素副本和最小概念单元而发现对象。如果已经有了一组类库，某些对象是很明显的。类之间的共同性（暗示了基类和继承类），可以立刻出现或在设计过程的后期出现。

2) 对象装配　我们在建立对象时会发现需要一些新成员，这些新成员在对象发现时期未出现过。对象的这种内部需要可能要用新类去支持它。

3) 系统构造　对对象的更多要求可能出现在以后阶段。随着不断的学习，我们会改进我们的对象。与系统中其它对象通讯和互相连接的需要，可能改变已有的类或要求新类。

4) 系统扩充　当我们向系统增添新的性能时，可能发现我们先前的设计不容易支持系统扩充。这时，我们可以重新构造部分系统，并很可能要增加新类。

5) 对象重用　这是对类的真正的重点测试。如果某些人试图在全新的情况下重用它，他们会发现一些缺点。当我们修改一个类以适应更新的程序时，类的一般原则将变得更清楚，直到我们有了一个真正可重用的对象。

对象开发原则

在这些阶段中，提出考虑开发类时所需要的一些原则：

1) 让特殊问题生成一个类，然后在解其他问题时让这个类生长和成熟。

2) 记住，发现所需要的类，是设计系统的主要内容。如果已经有了那些类，这个项目就不困难了。

3) 不要强迫自己在一开始就知道每一件事情，应当不断地学习。

4) 开始编程，让一部分能够运行，这样就可以证明或反驳已生成的设计。不要害怕过程语言风格的细面条式的代码——类分割可以控制它们。坏的类不会破坏好的类。

5) 尽量保持简单。具有明显用途的不太清楚的对象比很复杂的接口好。我们总能够从小的

[1] 参看Design Patterns:Elements of Reusable Object-Oriented Software by Erich Gamma et al., Addison-Wesley, 1995。

和简单的类开始，当我们对它有了较好地理解时再扩展这个类接口，但不可能简化已存在的类接口。

1.3.5 方法承诺什么

由于不同的原因，方法承诺的东西往往比它们能够提供的东西多得多。这是不幸的，因为当策略和不实际的期望同时出现时程序员会疑神疑鬼。一些方法的坏名声，使得程序员丢弃了手上的其他方法，忽视了一些有价值的技术。

1. 管理者的银子弹

最坏的许诺是“这个方法将解决您的所有问题”。这一许诺也很可能用这样的思想表达，即一个方法将解决实际上不存在解的问题，或者至少在程序的设计领域内没有解的问题：一个贫穷的社团文化，疲惫的、互相疏远或敌对的项目组成员；不充分的时间和资源；或试图解决一个实际上不能解的问题（资源不足）。最好的方法学，不管它许诺什么，都不解决这些或类似的任何问题。无论OOP还是C++，都无助于这样的问题。不幸的是，在这种情况下管理员是这样的人：对于银子弹的警报[1]，他是最易动摇的。

2. 提高效率的工具

这就是方法应当成为的东西。提高生产效率不仅取决于管理容易和花费不大，而且取决于一开始就创建好的设计。由于一些方法学的创造动机是为了改善维护，所以它们就片面地强调维护问题，而忽视了设计的漂亮和完整。实际上，好的设计应当是首要的目标，好的 OOP设计也应当容易维护，但这是它的附加作用。

1.3.6 方法应当提供什么

不管为特殊方法提出什么要求，它都应当提供这一节所列出的基本功能：允许为讨论这个项目将完成什么和如何做而进行通讯的约定；支持项目结构化的系统；能用某抽象形式描述项目的一组工具（使得程序员能容易地观察和操作项目）。如过去介绍过的，一个更微妙的问题是该方法对待最宝贵资源——组员积极性的“态度”。

1. 通讯约定

对于很小的项目组，可以用紧密接触的方式自然维持通讯。这是理想的请况。 C++的最大的好处之一是它可以使项目由很少的项目组成员建立，因此，明白表示的通讯能使维护变得容易，因而通讯费用低，项目组能更快地建立。

情况并不总是这样理想，有可能项目组成员很多，项目很复杂，这就需要某种形式的通讯原则。方法提供一种在项目组成员之间形成“约定”的办法。可以用两种方式看待这样的约定：

1) 敌对的　约定基于参与的当事人之间互有疑问，以使得没有人出格且每个人都做应该做的事情。约定清楚地说明，如果他们不做这些事，会出现坏的结果。这样看待任何约定，我们就已经输了，因为我们已经认为其他人是不可信赖的了。如果不能信任某人，约定并不能确保好的行为。

2) 信息的　约定是一种努力，使每个人都知道我们已经在哪些方面取得了一致的意见。这是对通讯的辅助，使得每个人能看到它并说，“是的，这是我认为我们将要做的事情”。它是协议作出后对协议的表述，只是消除误解。这类约定能最小化，并容易读。

---

[1] A reference to vampires made in The Mythical Man-Month, by Fred Brooks, Addision-Wesley, 1975。

有用的方法不鼓励敌对的约定，而重点是在通讯上。

2. 使系统结构化

结构化是系统的核心。如果一个方法能做些事情，那么它就必须能够告诉程序员：

1) 需要什么类

2) 如何把它们连接在一起，形成一个工作系统。

一个方法产生这些回答需要一个过程，即首先对问题进行分析，最终对类、系统和类之间传递的消息进行某种表述。

3. 描述工具

模型不应当比它描述的系统更复杂。一种好的模型仅提供一种抽象。

程序员一定不会使用只对特殊方法有用的描述工具。他能使自己的工具适合自己的各种需要。（例如在本章后面，建议一种符号，用于商业字处理机。）下面是有用的符号原则：

1) 方法中不含有不需要的东西。记住，“七加减二”规则的复杂性。（瞬间，人们只能在头脑中存放这么多的条目。）额外的细节就变成了负担，必须维护它，为它花钱。

2) 通过深入到描述层，人们应当能够得到所需要的信息。即我们可以在较高的抽象层上创建一些隐藏的层，仅在需要时，它们才可见。

3) 符号应当尽可能少。“过多的噱头使得软件变坏”。

4) 系统设计和类设计是互相隔离的问题。类是可重用工具，而系统是对特殊问题的解（虽然系统设计也是可重用的）。符号应当首先集中在系统设计方面。

5) 类设计符号是必须的吗？由C++语言提供的类表达对大多数情况是足够的。如果符号在描述类方面不能比用OOP语言描述有一个明显的推进，那么就不要用它。

6) 符号应当在隐藏对象的内部实现。在设计期间，这些内部实现一般不重要。

7) 保持符号简单。我们想用我们的方法做的所有事情基本上就是发现对象及其如何互相连接以形成系统。如果一个方法或符号要求更多的东西，则应当问一问，该方法花费我们的时间是否合理。

4. 不要耗尽最重要的资源

我的朋友Michael Wilk来自学术界，也许并不具备做评判的资格（从某个人那里听说的新观点），但他观察到，项目、开发组或公司拥有的最重要的资源是积极性。不管问题如何困难，过去失败多么严重，工具多么原始或不成套，积极性都能克服这些障碍。

不幸的是，各种管理技术常常完全不考虑积极性，或因为不容易度量它，就认为它是“不重要”的因素，他们认为，如果管理完善，项目就能强制完成。这种认识有压制开发组积极性的作用，因为他们会感到公司除了利益动机以外就没有感兴趣的东西了。一旦发生这种现象，组员就变成了“雇员”，看着钟，想着感兴趣的、分心的事情。

方法和管理技术是建立在动机和积极性的基础上的，最宝贵的资源应当是对项目真正感兴趣。至少，应当考虑OOP设计方法对开发组士气起的作用。

5. “必”读

在选择任何方法之前，从并不想出售方法的人那儿得到意见是有帮助的。不真正地理解我们想要一种方法是为了做什么或它能为我们做什么，那么采用一种方法很容易。其他人正在用它，这似乎是很充足的理由。但是，人们有一种奇怪的心理：如果他们相信某件事能解决他们的问题，他们就将试用它（这是经验，是好的）。但是，如果它不能解决他们的问题，他们可能加倍地努力，并且开始大声宣布他们已经发现了伟大的东西（这是否定，是不好的）。这个假设是，如果见到同一条船上有其他人，就不感到孤单，即便这条船哪儿也不去。

这并不是说所有的方法学都什么也不做，而是用精神方法使程序员武装到牙齿，这些方法能使程序员保持实验精神（“它不工作，让我们再试其它方法”）和跳出否定方式（“不，这根本不是问题，每样东西都很好，我们不需要改变”）。我认为，在选择方法之前读下面的书，会为我们提供这些精神武器。

《软件的创造力》（Software Creativity，Robert Glass编，Prentice-Hall,1995）。这是我所看到的讨论整个方法学前景最好的书。它是由Glass已经写过的短文和文章以及收集到的东西组成的（P.J.Plauger是一个撰稿者），反映出他在该主题上多年的思考和研究。他们愉快地说明什么是必须的，并不东拉西扯和扫我们的兴，不说空话，而且，这里有几百篇参考文献。所有的程序员和管理者在陷入方法学泥潭之前应当读这本书[1]。

《人件》（Peopleware，Tom Demarco 和Timothy Lister 编，Dorset House,1987）。虽然他们有软件开发方面的背景，而且本书大体上是针对项目和开发组的，但是这本书的重点是人和他们的需要方面，而不是技术和技术的需要方面。他们谈论创造一个环境，在其中人们是幸福和高效率的，而不是决定人们应当遵守什么样的规则，以使得他们成为机器的合适部件。我认为，这本书后面的观点是对程序员在采用XYZ方法，然后平静地做他们总是做的事情时微笑和点头的最大贡献。

《复杂性》（Complexity, M. Mitchell Waldrop编，Simon & Schuster, 1992）。此书集中了Santa Fe, New Mexico的一组持不同观点的科学家，讨论单个原则不能解决的实际问题（经济学中的股票市场、生物学的生命原始形式、为什么人们做他们在社会中应做的事情，等等）。通过物理学、经济学、化学、数学、计算机科学、社会学和其他学科的交叉，对这些问题的一种多原则途径正在发展。更重要的是，思考这些极其复杂问题的不同方法正在形成。抛开数学的确定性，人们想写一个预言所有行为的方程，首先观察和寻找一个模式，用任何可能的手段模拟这个模式（例如，这本书编入了遗传算法）。我相信，这种思维是有用的，因为我们正在对管理越来越复杂软件项目的方法做科学观察。

1.4 起草：最小的方法

我首先声明，这一点没有证明过。我并不许诺——起草是起点，是其他思想的种子，是思想试验，尽管这是我在大量思考、适量阅读和在开发过程中对自己和其他人观察之后形成的看法。这是受我称之为“小说结构”的写作类的启示。“小说结构”出现在Robert McKee[2]的教学中，最初针对热心熟练的电影剧作家们，也针对小说家和编剧。后来我发现，程序员与这个人群有大量的共同点：他们的思想最终用某类文本形式表达，表达的结构能确定产品成功与否。有少量令人拍案称奇的上口小说，其他许多小说都很平庸，但有技巧，得到发行，大量不上口的小说得不到发表。当然，小说要描述，而程序要编写。

作家还有一些在程序设计中不太出现的约束：他们一般单独工作或可能在两个人的组中工作。这样，他们必须非常经济地使用他们的时间，放弃不能带来重要成果的方法。McKee的两个目标是将花费在电影编剧上的时间从一年减少到六个月，在这个过程中极大地提高电影编剧的质量。软件开发者可以有类似的目标。

使每个人都同意某些事情是项目启动过程中最艰苦的部分。系统的最小性应当能获得最独立的程序员的支持。

---

[1] 另外一本好“前景”的书是Object Lessons Tom Love著, SIGS Books, 1993。

[2] Through Two Arts, Inc.,12021 Wilshire Blvd. Suite 868, Los Angeles, CA 90025。

1.4.1 前提

该方法的描述是建立在两个重要前提的基础上的，在我们采用该思想的其他部分时必须仔细考虑这两个前提：

1) 与典型的过程语言（和大量已存在的语言）不同， C++语言和语言性能中有许多防护，程序员能建立自己的防护。这些防护意在防止程序员创建的程序破坏它的结构，无论在创建它的整个期间还是在程序维护期间。

2) 不管分析得如何透彻，这里还有一些有关系统的事直到设计时还没有揭示出来，更多的直到程序完成和运行时还没有揭示出来。因此，快速通过分析过程和设计过程以实现目标系统的测试是重要的。根据第一点，这比用过程语言更安全，因为C++中的防护有助于防止“面条”代码的创建。

着重强调第二点。由于历史原因，我们已经用了过程语言，因此在开始设计和实现之前开发组希望仔细地处理和了解每一微小的细节，这是值得表扬的。的确，当创建 DBMS时，彻底地了解消费者的需要是值得的。但是， DBMS是一类具有良好形式且容易理解的问题。在这一章中讨论的这类程序设计问题是wild-card变体问题，它不只简单地重新形成已知解，而是包括一个或多个wild-card因素——元素，在这里没有容易理解的先前的解，而必须进行研究[1]。在着手设计和实现之前彻底地分析wild-card问题会导致分析瘫痪，因为在分析阶段没有足够的信息解决这类问题。解这样的问题要求在整个周期中反复，要冒险（以产生感性认识，因为程序员正在做新事情，并且有较高的潜在回报）。结果是，危险由盲目“闯入”预备性实现而产生，但是它反而能减少在wild-card项目中的危险，因为人们能较早地发现一个特殊的设计是否可行。

这个方法的目标是通过建议解处理wild-card问题，得到最快的开发结果，使设计能尽早地被证明或反证。这些努力不会白费。这个方法常常建议，“建立一个解，然后再丢掉它”。用OOP，可能仍然丢掉一部分，但是因为代码被封装成类，不可避免地生产一些有用的类设计和第一次反复中发展一些对系统设计有价值的思想，它们不需要丢掉。这样，对某一问题快速地过一遍不仅产生对下一次分析、设计和实现的重复重要的信息，而且也为下一次的重复过程创建了代码基础。

这个方法的另一性能是能对项目早期部分的集体讨论提供支持。由于保持最初的文档小而简明，所以最初的文档可以由小组与动态创建该描述的领导通过几次集体讨论而创建，这不仅要求每个人的投入，而且还鼓励开发组中的每个人意见一致。也许更重要的是，它能在较高的积极性下完成一个项目（如先前注意到的，这是最基本的资源）。

表示法

作家的最有价值的计算机工具是字处理器，因为它容易支持文档的结构。对于程序设计项目，程序的结构通常是由某形式的分离的文档来表述的。因为项目变得更复杂，所以文档是必需的。这就出现了一类问题，如 Brooks[2] 所说：“数据处理的基本原则引出了试图用同步化方法维护独立文档的傻念头——而我们在程序设计文档方面的实践违反了我们自己的教义。我们典型的努力是维护程序的机器可读形式和一组独立可读的文档……。”

好的工具应当将代码和它的文档联系起来。

我们认为，使用熟悉的工具和思维模式是非常重要的， OOP的改变正面临着由它自己引起

[1] 我估计这样的项目的主要规则：如果多于一张 wild-card，则不计划它要费多长时间和它花费多少。这里有太多的自由度。

[2] The Mythical Man-Month, 出处同上。

的挑战。较早的OOP方法学已经因为使用精细的图形符号方案而受挫了。我们不可避免地要大量改变设计，因为我们必须改变设计以避免棘手的问题，所以用很难修改的符号表示设计是不好的。只是最近，才有处理这些图形符号的工具出现。容易使用设计符号的工具必须在希望人们使用这种方法之前就有了。把这种观点与在软件设计过程中要求文档这一事实结合，可以看出，最符合逻辑的工具是一个性能全面的字处理器[1]。事实上，每个公司都有了这些工具（所以试用这种方法不需要花费），大多数程序员熟悉它们，程序员习惯于用它们创建基本宏语言。这符合C++的精神，我们是建立在已有的知识和工具基础上的，而不是把它们丢掉。

这个方法所用的思维模式也符合这种精神。虽然图形符号在报告中表示设计是有用的[2]，但它不能紧密地支持集体讨论。但是，每个人都懂得画轮廓，而且很多字处理器有一些画轮廓的方法，允许抓取几块轮廓，很快地移动它们。这使交互集体讨论会上快速设计很完美。另外，人们可以扩展和推倒轮廓，决定于系统中粒度的不同层次。（如后描述）因为程序员创建了设计，创建了设计文档，所以关于项目状态的报告能用一个像运行编译器一样的过程产生。

1.4.2 高概念

建立的系统，无论如何复杂，都有一个基本的目的，它服务于哪一行业和它应满足什以基本需要。如果我们看看用户界面、硬件、系统特殊的细节、编码算法和效率问题，那么我们最终会发现它有简单和直接的核心。就像好莱坞电影中所谓的高概念，我们能用一两句话描述它。这种纯描述是起点。

高概念相当重要，因为它为我们的项目定调。它是委派语句，不需要一开始就正确（可以在完全清楚它之前完善这个论述或构思设计），它只是尝试，直到它正确。例如，在空中交通控制系统中，我们可以从系统的一个高概念开始，即准备建立“控制塔跟踪飞机。”但是当将该系统用于非常小的飞机场时，也许只有一个导航员或无人导航。更有用的模型不会使得正在创建的解像问题描述那么多：“飞机到达、卸货、服务和重新装货、离去。”

1.4.3 论述(treatment)

剧本的论述是用一两页纸写的故事概要，即高概念的外层。计算机系统发展高概念和论述的最好的途径可能是组成一个小组，该小组有一个具有写能力的辅助工具。在集体讨论中能提出建议，辅助工具在与小组相连的网络计算机上或在屏幕上表达这些思想。辅助工具只起捉刀人的作用，不评价这些思想，只是简单地使它们清楚和保持它们通顺。

论述变成了初始对象发现的起点和设计的第一个雏形，它也能在拥有辅助工具的小组内完成。

1.4.4 结构化

对于系统，结构是关键。没有结构，就会任意收集无意义事件。有了结构，就有了故事。故事的结构通过特征表示，特征对应于对象，情节结构对应于系统设计。

1. 组织系统

如前所述，对于这种方法，最主要的描述工具是具有概括功能的高级字处理器。

---

[1] 我的观察是基于我最熟悉的：Microsoft Word的扩展功能，它已被用于产生了这本书的照相机准备的页。

[2] 我鼓励这种选择，即用简单的方框、线和符号，它们在画字处理的包时是可用的，而不是很难产生的无定型的形状。

从“高概念”、“论述”、“对象”和“设计”的第一层开始。当对象被发现时它们就被放在第二层子段中，放在“对象”下面。增加对象接口作为第三层子段，放在对象的特殊类下面。如果基本表述文本产生，就在相应的子段下面作为标准文本。

因为这个技术包括键入和写大纲，不依靠图画，所以会议讨论过程不受创作该描述的速度限制。

2. 特征：发现初始对象

论述包括名词和动词。当我们列出它们后，一般将名词作为类，动词或者变为这些类的方法或者变为系统设计的进程。虽然在第一遍之后程序员可能对他找出的结构不满意，但请记住，这是一个反复的过程。在将来的阶段和后面的设计中可以增加另外的类和方法，因为那时程序员对问题会有更清晰的认识。这种构造方法的要点是不需要程序员当前完全理解问题，所以不期望设计一下子展现在程序员的面前。

从简单的论述检查开始，为每个已找出的唯一名称创建“对象”中的第二层子段。取那些很显然作用于对象的动词，置它们于相应名词下面的第三层方法子段。对每个方法增加参数表（即使它最初是空的）和返回类型。这就给程序员一个雏形以供讨论与完善。

如果一个类是从另一个类继承来的，则它的第二层子段应当尽可能靠近地放在这个基类之后，它的子段名应当显示这个继承关系，就像写代码：derived:public base时应当做的。这允许适当地产生代码。

虽然能设置系统去表示从公共接口继承来的方法，但目的是只创建类和它们的公共接口，其他元素都被认为是下面实现的部分，不是高层设计。如果要表示它们，它们应当作为相应类下面的文本层注解出现。

当决策点确定后，使用修改的 Occam’s Razor办法：考虑这个选择并选择最简单的一个，因为简单的类几乎总是最好的。向类增加元素很容易，但是随着时间的推移，丢掉元素就困难了。

如果需要培植这一过程，请看一个懒程序员的观点：您应当希望哪些对象魔术般地出现，用以解决您的问题？让一些可以用的类和各种系统设计模式作为手头的参考是有用的。

我们不要总是在对象段里，分析这个论述时应当在对象和系统设计之间来回运动。任何时候，我们都可能想在任何子段下面写一些普通文本，例如有关特殊类或方法的思想或注解。

3. 情节：初始系统设计

从高概念和论述开始，会出现一些子情节。通常，它们就像“输入、过程、输出”或“用户界面、活动”一样简单。在“设计”下面，每个子情节有它自己的第二层子段。大多数故事沿用一组公共情节。在oop中，这种类似被称为“模式”。查阅在oop设计模式上的资源，可以帮助对情节的搜索。

我们现在正在创建系统的粗略草图。在集体讨论会上，小组中的人们对他们认为应该出现在系统中的活动提出建议，并且分别记录，不需要为将它与整个系统相连而工作。让整个项目组，包括机械设计人员（如果需要）、市场人员、管理人员，都出席会议。这是特别重要的，这不仅使得每个人心情舒畅，因为他们的意见已经被考虑，而且每一个人的参加对会议都是有价值的。

子情节有一组变化的阶段或状态，有在阶段之间变化的条件，有包含在每个过渡中的活动。在特殊的子情节下面，每个阶段都给出它自己的第三层子段。条件和过渡被描写为文本，放在这个阶段的标题下。情况如果理想，我们最终能够写出每个子情节的基本的东西（因为设计是反复过程），作为对象的创建并向它们发送的消息。这就变成了这个子情节的最初代码。

设计发现和对象发现过程类似，因此我们可以在会议过程中将子条目增加到这两个段中。

1.4.5 开发

这是粗设计到编译代码的最初转换，编译代码能被测试，特别是，它将证明或者反证我们的设计。这不只是一遍处理，而是一系列写和重写的开始，所以，重点是从文档到代码的转换，该转换用这样一种方法，即通过对代码中的结构或有关文字的改变重新产生文档。这样，在编码开始后（和不可避免的改变出现后）产生设计文档就变得非常容易了，而设计文档能变成项目进展的报告工具。

1. 初始翻译

通过在第一层的标题中使用标准段名“对象”和“设计”，我们就能运行我们的工具以突出这些段，并由它们产生文件头。依据我们所在的主段和正在工作的子段层，我们将完成不同的工作。最容易的办法可能是让我们的工具或宏把文档分成小块并且相应地在每一小块上工作。

对于“对象”中的每个第二层段，在段名（类名和它的基类名，如果有的话）中会有足够的信息用以自动地产生类声明，在这个类名下面的每个第三层子段，段名（成员函数名、参数表和返回类型）中会有足够的信息用以产生这个成员函数的声明。我们的工具将简单地管理这些并创建类声明。

为了使问题简单，单个类声明将出现在每个头文件中。命名这些头文件的最好办法，也许是包括这个文件名，以作为这个类的第二层段名中的标记信息。

编制情节可以更精细。每个子情节可以产生一个独立的函数，由内部 main( )调用，或者就是main( )中的一段。从一个能完成我们的工作的事情开始，更好的模式可能在以后的反复中形成。

2. 代码产生

使用自动工具（大部分字处理工具对此是合适的）：

1) 为“对象”段中描述的每个类产生一个头文件，也就是为每一个类创建一个类声明，带有公共接口函数和与它们相联系的描述块；对每个类附上在以后很容易分析的专门标号。

2) 为每个子情节产生头文件，并且将它的描述拷贝在文件的开头，作为注释块，接下来跟随函数声明。

3) 用它的概要标题层标记每个子情节、类和方法，形成加标号的、被注释的标识符：//#[1]、// #[2]等等。所有产生的文件都有文档注释，放在带有标号的专门标识块中。类名和函数声明也保留注释标记。这样，转换工具能检查、提取所有信息，并用文档描述语言更好地重新产生源文档，例如用Rich Text Format(RTF)描述语言。

4) 接口和情节在这时应当是可编译的（但不可连接），因此可以做语法检查。这将保证设计的高层完整性。文档能由当前正在编译的文件重新产生。

5) 在这个阶段，有两件事会发生。如果设计是在早期，那么我们可能需要继续加工处理集体讨论会的文档（而不是代码）或小组负责的那部分文档。然而，如果设计是完全充足的，那么我们就可以开始编码。如果在编码阶段增加接口元素，则这些元素必须连同加过标号的注释一起由程序员加标号，所以重新产生的程序能用新信息产生文档。

如果我们拥有前端编译器，我们确实可以对类和函数自动进行编译，但是它是大作业，并且这个语言正在演化。使用明确的标号，是相当不安全的，商业浏览工具能用以检验是否所有的公共函数都已经形成文档了（也就是，它们已加标号了）。

1.4.6 重写

这类似于重写电影剧本以完善它，使它更好。在程序设计中，这是重复过程，我们的程序从好到更好，在第一遍中还没有真正理解的问题变得清楚了。我们的类从在单个项目中使用进化为可重用的资源。

从工具的观点看，转换该过程略微复杂，我们希望能分解头文件，使我们能重新整理这些文件使它们成为设计文档，包括在编码过程中已经做过的全部改变。在设计文档中对设计有任何改变，头文件也必须完全重建，这样就不会丢失为了得到在第一遍反复中编译所需要的头文件而做的任何工作。因此，我们的工具不仅应当能找出加标号的信息，将它们变成段层和文本层，而且还应当能发现其他信息，为其他信息加标号和存放其他信息，例如在每个文件开头的#include。我们应当记住，头文件表达了类设计而我们必须能够由设计文档重新产生头文件。

我们还应当注意文本层注解和讨论，它们最初产生时被转换为加标号的注释，比在设计演化过程中程序员修改的内容更多。基本上，这些注解和讨论被收集并放在各自的地方，因此设计文档反映了新的信息。这就允许我们去改变这些信息，并且返还到已产生的头文件中。

对于系统设计（main()和任何支持函数），我们也可能想获得整个文件，添加段标识符，例如A、B、C等等，作为加标号的注释（不用行号，因为行号会改变），并附上段描述（然后返还main()文件，作为加标号的文本）。

我们必须知道什么时候停止，什么时候重复设计。理想的情况是，我们达到了目标功能，处在完善和增加新功能的过程中，最后期限到来，强迫我们停止并发出我们的版本（记住，软件是预约业务）。

1.4.7 逻辑

我们会周期性地希望知道项目在什么地方需要重新整理文档。如果是在网上使用自动化工具，这个过程无关紧要。经常地整集和维护设计文档是项目领导者或管理者的责任，而项目组或个人只对文档的一小部分负责（也就是他们的代码和注释）。

对于补充功能，例如类图表，可以用第三方工具生成，并自动地添加到文档中。

在任何时候，都可以通过简单地“刷新”文档生成当前的报告。这样可以看到程序各部分的情况，也支援了项目组，并为最终用户文档提供了直接的更新。这些文档对于使新组员尽快参加工作也很有价值。

单个文档比由某些分析、设计和实现方法而产生的所有文档更合理。虽然一个较小的文档缺乏印象，但它是“活的”。相反，例如一个分析文档只是对于项目的特殊阶段有价值，然后很快变得陈旧了。对于知道将被丢掉的文档，人们很难对它再作更大的努力。

1.5 其他方法

当前有大量的形式化方法（不下20种）可用，由程序员选择[1]。有一些并不完全独立，它们有共同的思想基础，在某个更高层上，它们都是相同的。在最低层，很多方法都受语言的缺省表现约束，所以每个方法大概只能满足简单的项目。真正的好处是在较高层上，一个方法可用于实时硬件控制器，但可能不容易适合档案数据库的设计。

每种方法都有它的啦啦队，在我们对大规模方法采用之前应当更好地懂得它的语言基础，

[1] 这些是下书中的综述：Object Analysis and Design:Description of Methods, edited by Andrew T.F.Hutt of Object Management Group(OMG),john Wiley & Sons, 1994。

以此来认识一个方法是否适合我们的特殊风格，或者我们是否真需要一个方法。下面是对最流行的三种方法的描述，主要是供参考，不是购买比较。如果读者想了解更多的方法，有许多书籍可以参考，还有许多学习班可以参加。

1.5.1 Booch

Booch方法 [1] 是最早、最基本和最广泛被引用的一个方法。因为它是较早发展的，所以对各种程序设计问题都有意义。它的焦点是在OOP的类、方法和继承的单独性能上，步骤如下：

1. 在抽象的特定层上确定类和对象

这是可预见的一小步。我们用自然语言声明问题和解，并且确定关键特性，例如形成类的基本名词。如果我们在烟花行业，可能想确定工人、鞭炮、顾客，进而，可能需要确定化学药剂师、组装者、处理者，业余鞭炮爱好者和专业鞭炮者、购买者和观众。甚至更专门的，能确定年轻观众、老年观众、少年观众和父母观众。

2. 确定它们的语义

这是在相应的抽象层上定义类。如果我们计划创建一个类，我们应当确定类的相应观众。例如，如果创建鞭炮类，那么谁将观察它，化学药剂师还是观众？前者想知道在结构中有什么化学药剂，而后者将对鞭炮爆炸时释放的颜色和形状感兴趣。如果化学药剂师问起一个鞭炮产生主颜色的化学药剂，最好不要回答“有点冷绿和红”。同样，观众会对鞭炮点火后喷出的只是化学方程式感到迷惑不解。也许，我们的程序是为了眼前的市场，化学药剂师和观众都会用它，在这种情况下，鞭炮应当有主题属性和客体属性，并且能以和观察者相应的外观出现。

3. 确定它们之间的关系（CRC 卡片）

定义一个类如何与其他类互相作用。将关于每个类的信息制成表格的常见的方法是使用类，责任，协作（Class, Responsibility, Collaboration, CRC)卡片。这是一种小卡片（通常是一个索引卡），在它上面写上这个类的状态变量、它的责任（也就是它发送和接受的消息）和对与它互相作用的其他类的引用。为什么需要索引卡片？理由是我们应当能在一张小卡片上存放我们需要知道的关于一个类的所有内容，如果不能满足这点，那么这个类就太复杂了。理想的类应当能在扫视间被理解，索引卡片不仅实际可行，而且刚好符合大多数人思考问题合理的信息量。一种不涉及主要技术改革的解决办法对于每个人都可用的（就像本章前面描述的草稿方法中的文档结构化一样)。

4. 实现类

现在我们知道做什么了，投入精力进行编码。在大多数项目中， 编码将影响设计。

5. 反复设计

这样的设计过程给人一种类似著名的程序开发瀑布流方法的感觉。现在对这种方法的看法是有分歧的。在第一遍查看主要的抽象是否可以将类清晰地分离之后，可能需要重复前三个步骤。Booch写了一个“粗糙旅程完型设计过程”。如果这些类真正反映了这个解的自然语言描述，那么程序有完型的观点应当是可能的。也许要记住的最重要的事情是，通过缺省 —— 实际上是通过定义，如果修改了一个类，它的超类和子类应当仍然工作。不需要害怕修改，它不会破坏程序，对结果的任何改变将限制在子类和 /或被改变的这个类的特定协作者中。为了这个类而对CRC卡片的扫视也许是我们需要检验新版本的唯一线索。

---

[1] 参看Object-Oriented Design with Applications by Grady Booch, Benjamin Cummings, 1991.有关C++更新的版本。

1.5.2 责任驱动的设计（RDD）

这个方法[1]也用CRC卡片。在这里，正如名称蕴涵，卡片的重点在于责任的授权，而不是外观。这可由下面例子说明：Booch方法可以产生雇员——银行雇员——银行经理的继承，而在RDD中，这可能出现经理——金融经理——银行经理的继承。银行经理的主要责任是经理的责任，所以这种继承反映了这种关系。

更形式化地说，RDD包含如下内容：

1) 数据或状态： 对每个类的数据或状态变量的描述。

2) 池和源： 数据池和源的标识；处理或产生数据的类。

3) 观察者或观点： 观点或观察者类，用以隔离硬件依赖。

4) 辅助工具或帮助器： 辅助工具或帮助器类，例如连接表，它们包含很少的状态信息并简单地帮助其他类工作。

1.5.3 对象建模技术（OMT）

对象建模技术[2]（OMT）对过程增加一个或多个复杂层。 Booch方法强调类的功能表现，简单地定义它们作为自然语言解的轮廓。 RDD进了一步，强调类的责任超过强调类的表现。OMT用详细地绘制图表的方法，不仅描述类，而且还描述系统的各种状态，如下所述：

1) 对象模型，“什么”，对象图表： 该对象模型类似于由Booch方法和RDD产生的那些模型；对象的类通过责任相关联。

2) 动态模型，“何时”，状态图表： 该动态模型描写了系统的与时间有关的状态。不同的状态是通过转变相关联的；包含时间有关状态的一个例子是实时传感器，它从外部世界收集数据。

3) 功能模型，“如何”，数据流程表： 该功能模型跟踪数据流。它的理论是，在程序的最低层上，实际的工作是通过使用过程而完成的，因此程序的低层行为最好通过画数据流来理解，而不是通过画它的对象来理解。

1.6 为向OOP转变而采取的策略

如果我们决定采用OOP，我们的下一个问题可能是“我如何才能使得管理员、同事、部门、伙伴开始使用OOP？”想一想，作为独立的程序员，应当如何学习使用新语言和新程序设计。正如我们以前所做的，首先训练和做例子，再通过一个试验项目得到一个基本的感觉，不要做太混乱的事情，然后尝试做一个“真实世界”的实际有用的项目。在我们的第一个项目中，我们通过读、向上司问问题、与朋友切磋等方式，继续我们的训练。基本上，这就是许多作者建议的从C转到C++的方法。转变整个公司当然应当采用某个动态的小组，但回忆个人是如何做这件事的，能在转变的每一步中起到有益的作用。

1.6.1 逐步进入OOP

当向OOP和C++转变时，有一些方针要考虑：

1. 训练

第一步是一些形式的培训。记住公司在平常的C代码上的投资，并且当每个人都在为这些遗留的东西而为难时，应努力在6到9个月内不使公司完全陷入混乱。对一个小组进行培训，更

[1] 参看Designing Object-Oriented Software by Rebecca Wirfs-Brock et al., Prentice Hall, 1990。

[2] 参看Object-Oriented Modeling and Design by James Rumbaugh et al., Prentice Hall,1991。

适宜的情况是，这个小组成员是一些勤奋好学、能很好地在一起工作的人们，当他们学习 C++时，能形成他们自己的支持网。

有时建议采用另一种方法，即在整个公司层一起训练，包括为策略管理员而开设的概论课程，以及为项目建设者而开设的设计课程和编程课程。对于较小的公司或较大公司的部门，用他们做事情的方法去做基本的改变是非常好的。然而代价较高，所以一些公司可能选择以项目层训练开始，做飞行员式的项目（可能请一个外面的导师），然后让这个项目组变成公司其他人的老师。

2. 低风险项目

首先尝试一个低风险项目，并允许出错。一旦我们已经得到了一些经验，我们就将这第一个小组的成员安插到其他项目组中，或者用这个组的成员作为 OOP的技术支柱。第一个项目可能不能正确工作，所以该项目在事情的安排上应当不是非常重要的。它应当是简单的、自包含的和有教学意义的。这意味着它应当包括创建对公司的其他程序员学习 C++有意义的类。

3. 来自成功的模型

从草稿开始之前，挑一些好的面向对象设计的例子。很可能有些人已经解决过我们的问题，如果他们还没有正确地解决它，我们可以应用已经学到的关于封装的知识，来修改存在的设计，以适合我们自己的需要。这是设计模式 [1] 的一般概念。

4. 使用已存在的类库

转变为C++的主要经济动机是容易使用以类库形式存在的代码，最短的应用开发周期是除了main()以外不必自己写任何东西。然而，一些新程序员不理解这个，不知道已存在的类库，或由于对语言的迷恋希望写可能已经存在的类。如果我们努力查找和重用其他人在转变过程中的早期代码，那么我们在OOP和C++方面将是最优的。

5. 不要用C++重写已存在的代码

虽然用C++编译C代码通常会有（有时是很大的）好处，它能发现老代码中的问题，但是把时间花在对已存在的功能代码进行 C++重写上，通常不是时间的最佳利用。如果代码是为重用而编写的，会有很大的好处。但是，有可能出现这种情况：在最初的几个项目中，并不能看到效率如您梦想的一样增长，除非这是新项目。当然，如果项目是从头开始的， C++和OOP最好。

1.6.2 管理障碍

对于管理员，他的任务就是为他的小组获得资源，使他的小组克服通往成功路上的障碍。并且，他应当努力创造高产的和令人愉快的环境，以使得他的小组最大可能完成他所要求的奇迹。转变到C++的过程包含有下面的三类障碍，如果不花费何代价，那真是奇怪的。虽然对于C程序员（也许对于其他过程语言的程序员）小组，可证明选择 C++比选择其他OOP语言代价低，但这也不是免费的。我们应当认识到，在我们试图动员我们的公司转移到 C++和着手转移之前，还有一些障碍。

1. 启动代价

这个代价要比得到C++编译器大得多。如果投资培训（也许为了指导第一个项目），并且如果选购解决问题的类库，而不是试图自己建立这些类库，那么可以将中长期代价减到最低。这些都是很花钱的，必须在制定计划时充分考虑。另外，当学习新语言连同程序设计环境时，

[1] 参看Camma et al., 出处同上。

还会有损失效率的隐含代价。培训和指导确实能使这些代价降到最低，但是组员们必须克服他们自己的困惑去理解这些问题。在这个过程中，他们将会犯更多的错误（这是一个特征，因为失败是成功之母）和有更低的效率。尽管如此，对于一些类型的程序设计问题、正确的类和正确的开发环境，即使我们还在学习C++，也可能比我们仍然用C语言时有更高的效率（即便考虑到我们正在犯更多的错误和每天写更少行的代码）。

2. 性能问题

一个共同的问题是“ OOP不会使我的程序更大且更慢吗？ ”回答是“ 不一定 ”。大多数传统的OOP语言是在实验和在头脑中快速建立原型的情况下设计的，而不侧重于普通操作。这样，它们就本质地决定了在规模上的明显增长和在速度上的明显降低。然而， C++是在已有生产程序的情况下设计的。当我们的焦点是建立快速原型时，我们可以尽快地把构件拉在一起，而忽略效率问题。如果我们正在使用任何第三方库，这些库通常已经被厂商优化过了，我们用快速开发方法时，无论如何这都不成问题。我们有一个我们喜欢的系统时，如果它足够小和足够快，这就行了。如果达不到这种要求，我们就开始用一个有益的工具进行调整，首先寻求加速，这可以通过简单地运用建立在C++中的一些特性做到。如果这还不行，应当寻求修改低层来实现，所以使用特定类的原有代码不需要改变。只有这一切都无法解决这个问题时才需要改变设计。性能在设计中的地位如此重要，以致于这一因素必然是主要设计标准的指标之一。通过快速原型较早地发现这一点是有好处的。

如本章较早所述， C和C++之间在规模和速度上的差距常常是± 10%，并且通常更接近。实际上我们可以用C++得到在规模和速度上超过C的系统，因为我们为C++ 所做的设计可以完全不同于为C所做的设计。

在C和C++之间比较规模和速度的证据至今还只是估计，也许还会继续如此。尽管一些人建议对相同的项目用C和C++同时做，但也许不会有公司把钱浪费在这里，除非它非常大并且对这个研究项目感兴趣。即便如此，它也希望钱花得更好。已经从C（或其他过程语言）转到C++的程序员几乎一致都有在程序设计效率上很大提高的个人经验，这是我们能找到的最引人注目的证据。

3. 普遍的设计错误

当项目组开始使用OOP和C++时，程序员们将会出现一系列普遍的设计错误。这经常会发生，因为在早期项目的设计和实现过程中从专家们那里得到的反馈太少，公司中没有专家。程序员似乎会觉得，在这个周期中，他懂得OOP太早了并开始了一条不好的道路。有时，对这个语言有经验的人认为显而易见的事情可能是新手们在内部激烈争论的主题。大量的这类问题都能通过聘用外部专家培训和指导来避免。

1.7 小结

这一章希望读者对面向对象程序设计和 C++的广泛问题有一定的感性认识，包括为什么OOP是不同的；为什么C++特别不同；什么是OOP方法的概念和为什么应当（或不应当）使用一个概念；提出最小化方法的建议（这是我发展的方法），它允许以最小费用开始OOP项目；对其他方法的讨论；最后当公司转到OOP和C++时会遇到的各种问题。

OOP和C++可能不会对每个人都适合。对自己的需要做出估计，并决定 C++是否能很好地满足自己的要求，或者是否能很好地离开别的程序设计系统，这是很重要的。如果程序员知道，他的需要对可以预见的未来是非常特别的，如果他有特殊的约束，不能由 C++满足，那么可以用它研究替代物。即便他最终选择了C++作为他的语言，他至少应当懂得这些选项是什么，并应当对为什么取这个方向有清晰的看法。