软件工程思想

林锐

序

《软件工程思想》讲述“软件开发”和“做程序员”的道理，视野独特，构思新颖，内容风趣，不落窠臼，令人耳目一新。堪称难得，以至回味无穷。

作者从事了八年的软件开发工作，在他的博士学位论文完成之际写下了这本“心之所感”。虽然它探讨的是软件工程最常见的内容，但他将亲身所历的感悟写成活泼生动的文字，将软件工程的很多原则和方法融于笑谈之中，让人看得轻松，时有共鸣。尽管很薄，然其内涵不逊于厚近千页的有关教科书。

每次回浙大我都要和林锐相聚，谈学术、论社会，直面人生，“位卑未敢忘忧国”，每每至凌晨。前不久我在某大学计算机系作讲座，最后冒昧谈了几句题外话，其中之一是“学问与明理”。古人云：“读书明理”，意即读书要明白做人的道理。我以为其中的重要内涵，是要有积极的人生观，以贡献社会为己任。这也是我们的共识。林锐曾立誓做一名“真实、正直、优秀的科技人员”。他在自己困难的时候依然资助数名贫困中学生和大学生；常常躬身拾捡被乱扔于地的废纸、塑料袋，以示后生。这都会使很多的学人汗颜有加。

简言之，林锐对软件工程实践的积极思考、轻快而不失深邃的文笔及其言行，都是出色之处。

正由于此，而不仅因为是同行，我才不惭浅陋，接受他的要求，荣幸地成为本书的第一位读者，并在本来应是名人大家留文的地方谈林说森。

董军，2000 年 2 月于
朝夕室

前　言

在 60 年代计算机发展初期，程序设计是少数聪明人干的事。他们的智力与技能超群，编写的程序既能控制弱智的计算机，又能让别人看不懂、不会用。那个时期编程就跟捏泥巴一样随心所欲，于是他们很过分地把程序的集合称为软件，以便自己开心或伤心时再把程序捏个面目全非。人们就在这种美滋滋的感觉下热情地编程，结果产生了一堆问题：程序质量低下，错误频出，进度延误，费用剧增……。这些问题导致了“软件危机”。

在 1968 年，一群程序员、计算机科学家与工业界人士聚集一起共商对策。通过借鉴传统工业的成功做法，他们主张通过工程化的方法开发软件来解决软件危机，并冠以“软件工程”这一术语。三十年余年来，尽管软件的一些毛病如人类的感冒一样无法根治，但软件的发展速度超过了任何传统工业，期间并未出现真真的软件危机。这的确是前人的先见之明。如今软件工程成了一门学科。

软件工程主要讲述软件开发的道理，基本上是软件实践者的成功经验和失败教训的总结。软件工程的观念、方法、策略和规范都是朴实无华的，平凡之人皆可领会，关键在于运用。我们不可以把软件工程方法看成是诸葛亮的锦囊妙计——在出了问题后才打开看看，而应该事先掌握，预料将要出现的问题，控制每个实践环节，并防患于未然。研究软件工程永远做不到理论家那么潇洒：定理证明了，就完事。

我在读大学的十年里有八年从事软件开发，尽管编写了几十万行 C++/C 程序，也经历了若干次小不点儿大的成功和失败，可老感觉只学了些皮毛，心里慌兮兮的。在博士研究生毕业前的半年里，我告戒自己不应该再稀里糊涂地在程序堆里滚爬下去了，于是就面壁反省，做了一阵子木讷的和尚。在“打坐”时，每有心得体会便记录下来，不知不觉凑成了八章经，我就给此经书起名为《软件工程思想》。

经典的软件工程书籍厚得象砖头，或让人望而却步，或让人看了心事重重。请宽恕我的幼稚，我试图用三个问题：是什么、为什么、怎么办，来解释软件工程的道理。所以本书薄得象饺子皮——用来包“思想”这种有味道的“馅”。本书的八章经分别为：

第一章“软件工程基本观念”；

第二章“程序员与程序经理”；

第三章“项目计划与质量管理”；

第四章“可行性分析与需求分析”；

第五章“系统设计”；

第六章“C++ 面向对象程序设计”；

第七章“测试与改错”；

第八章“维护与再生工程”。

附录“大学十年”可以充当饭后的水果。

我偶尔也担心此书写得太肤浅，内容少得可怜。就象一只鸡在水里扑腾了几下，并不能产生美味的鸡汤。但是如果您化了几分钟时间翻阅本书的任意章节，您马上就愿意再化几个小时一口气读完全书，并且乐得直拍桌子：“好！很好！非常好！”

您可以把这本科技书当小说看，但在看书时请不要吃东西，免得喷了别人或者呛着自己。

如果您买了本书后觉得不值得，我一定赔偿您的损失。

致　谢

本书并不属于我博士学位论文的研究范畴，但却是我读博士学位三年来写的最有意思的作品。

首先要感谢我的导师，浙江大学计算机辅助设计与图形学（CAD&CG）国家重点实验室的石教英教授。在其他师兄弟正儿八经地“攻读”博士学位时，我“不务正业”地开了一家软件公司。当我摔了一个大跟头灰溜溜地回到陌生的实验室时，石老师仍然热情地帮助我“修成正果”。临近毕业，我心中惭愧，三年来我从来都没给石老师干过活，我这个博士生他算是白招了。我很希望大学里多一些象石老师那样开明而大度的导师。

董军博士是本书的第一位读者。我们是“君子之交”却不“淡如水”，除了漆夜长谈科技、艺术、哲学外，还不忘“吃喝玩乐”享受生活。他在品阅的同时完成了审稿工作。

彭小澎好学上进，尽管她执教的是艺术课程，却很想再学点自然科学。她常听我聊侃软件工程，不知不觉成了本书的第二位读者。她看书时只会“哼”“哈”，从未有沉思状，估计啥也没看懂。小澎是个天真未泯的大孩子，和我称兄道弟，给我带来了很多快乐。有一天中午，我们把浙江大学校门口草坪上的垃圾捡得干干净净，俩人无上光荣。我希望小澎早日“荣升”讲师，再恭敬地叫她彭老师。

北京因特国风网络软件公司的周鸿一是个真正的软件高手。他在我开发软件产品失败时给予了最多的帮助，并指正我在软件设计中存在的根深蒂固的方法错误，使我能尽早地逐步改正。我平时能说会道，但在他面前我哑口无言只有听的份，因为他的才华已全方位地超过了我。我真希望多结识象他这样的朋友。

高振华老先生是个糊涂而可爱的民营企业家，我们是忘年交。我把他干的糊涂事（投资软件公司）写进书里，作为可行性分析的案例。高先生给予我经济上的帮助，使我能够在舒适的环境中开展最后一年博士学位论文工作。尽管我读书的工资每月只有 300 元，但日子过得象神仙一样舒服。

浙江大学计算机系的杨孟洲、周昆、曾震宇、杨建、白云、金锋等同学和我合作开发软件，给了我很多技术上的帮助。我对他们深表感谢。

特别感谢父母给我起了很好听的名字。读了十年大学，现在我差不多名副其实了。

林锐，2000 年 2 月

于浙江大学 CAD&CG 国家重点实验室

目　　录

第五章 系统设计

第一章　软件工程基本观念

本章讲述软件工程的基本观念，是关于软件工程宏观上的探讨。如果你是软件公司的老板，用不着在第一线工作，那么看这一章就够了。但你一定要让员工们相信不停地工作是人生最大的快乐，并且让他们把本书看完。

1.1 节讲述软件工程的目标和常用的软件工程模型。1.2 节讲述软件开发的基本策略：“复用”、“分而治之”、“优化——折衷”，有助于指导实践者选择方法和产生新方法。1.3 节例举一些不正确的观念，取材于早期软件人员比较幼稚的想法，初学者可以引以为戒。1.4 节探讨一些有争议的观念。

看完本章，要树立这样的信念：软件开发过程中的坎坎坷坷，仿佛只是人脸的凹凸不平，用热水毛巾一把就可抹平。让我们高举程序主义、软件工程思想的伟大旗帜，紧密团结在以 Microsoft 为核心的软件公司周围，沿着比尔·盖茨的生财之道，不分白天黑夜地编程，把建设有中国特色的软件产业的伟大事业全面推向 21 世纪。

1.1 软件工程的目标与常用模型

软件工程的目标是提高软件的质量与生产率，最终实现软件的工业化生产。质量是软件需求方最关心的问题，用户即使不图物美价廉，也要求个货真价实。生产率是软件供应方最关心的问题，老板和员工都想用更少的时间挣更多的钱。质量与生产率之间有着内在的联系，高生产率必须以质量合格为前提。如果质量不合格，对供需双方都是坏事情。从短期效益看，追求高质量会延长软件开发时间并且增大费用，似乎降低了生产率。从长期效益看，高质量将保证软件开发的全过程更加规范流畅，大大降低了软件的维护代价，实质上是提高了生产率，同时可获得很好的信誉。质量与生产率之间不存在根本的对立，好的软件工程方法可以同时提高质量与生产率。

软件供需双方的代表能在餐桌上谈笑风生，归功于第一线开发人员的辛勤工作。质量与生产率的提高就指望程序员与程序经理。对开发人员而言，如果非得在质量与生产率之间分个主次不可，那么应该是质量第一，生产率第二。这是因为：（1）质量直接体现在软件的每段程序中，高质量自然是开发人员的技术追求，也是职业道德的要求。（2）高质量对所有的用户都有价值，而高生产率只对开发方有意义。（3）如果一开始就追求高生产率，容易使人急功近利，留下隐患。宁可进度慢些，也要保证每个环节的质量，以图长远利益。

软件的质量因素很多，如正确性，性能、可靠性、容错性、易用性、灵活性、可扩充性、可理解性、可维护性等等。有些因素相互重叠，有些则相抵触，真要提高质量可不容易啊！

软件工程的主要环节有：人员管理、项目管理、可行性与需求分析、系统设计、程序设计、测试、维护等，如图 1.1 所示。

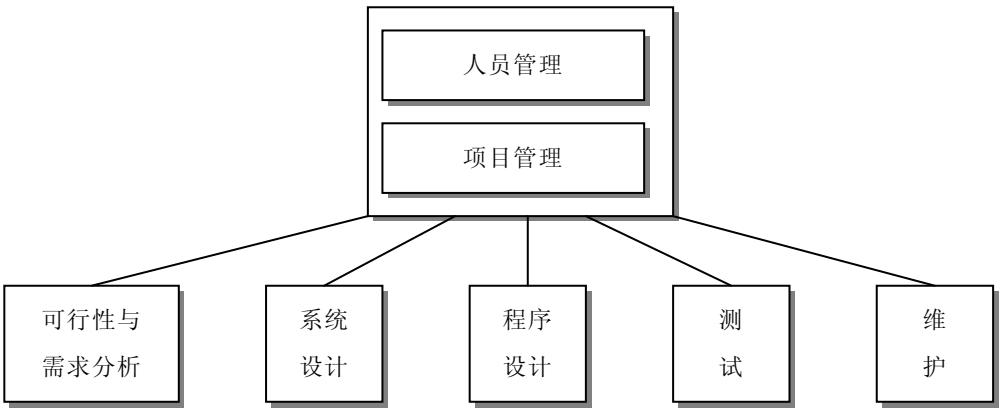


图 1.1　软件工程的主要环节

软件工程模型建议用一定的流程将各个环节连接起来，并可用规范的方式操作全过程，如同工厂的生产线。常见的软件工程模型有：线性模型（图 1.2），渐增式模型（图 1.3），螺旋模型，快速原型模型，形式化描述模型等等 [Pressmam 1999, Sommerville 1992]。

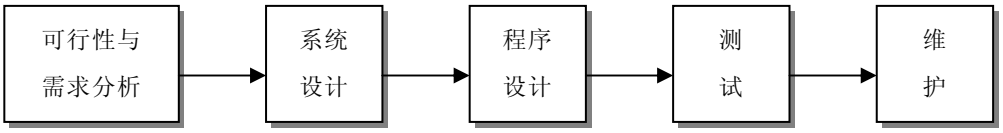


图 1.2　软件工程的线性模型

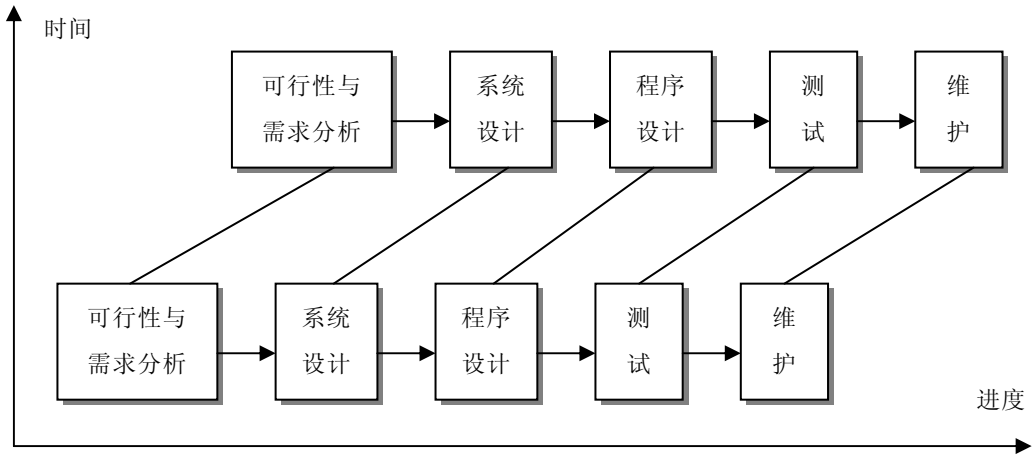
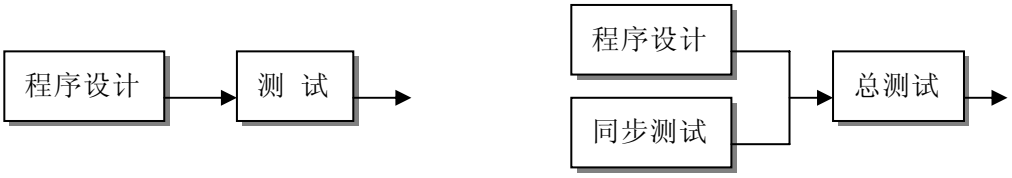


图 1.3　软件工程的渐增式模型

最早出现的软件工程模型是线性模型（又称瀑布模型）。线性模型太理想化，太单纯，已不再适合现代的软件开发模式，几乎被业界抛弃。偶而被人提起，都属于被贬对象，未被留一丝惋惜。但我们应该认识到，“线性”是人们最容易掌握并能熟练应用的思想方法。当人们碰到一个复杂的“非线性”问题时，总是千方百计地将其分解或转化为一系列简单的线性问题，然后逐个解决。一个软件系统的整体可能是复杂的，而单个子程序总是简单的，可以用线性的方式来实现，否则干活就太累了。线性是一种简洁，简洁就

是美。当我们领会了线性的精神，就不要再呆板地套用线性模型的外表，而应该用活它。例如渐增式模型实质就是分段的线性模型，如图 1.3 所示。螺旋模型则是接连的弯曲了的线性模型。在其它模型中都能够找到线性模型的影子。

套用固定的模型不是程序员的聪明之举。比如“程序设计”与“测试”之间的关系，习惯上总以为程序设计在先，测试在后，如图 1.4（a）所示。而对于一些复杂的程序，将测试分为同步测试与总测试更有效，如图 1.4（b）所示。



（a）　　（b）

图 1.4 （a）程序设计在先测试在后　　（b）测试分为同步测试与总测试

不论是什么软件工程模型，总是少不了图 1.1 中的各个环节。本书摊开具体的软件工程模型，顺序讲述人员管理、项目管理、可行性与需求分析、系统设计、程序设计、测试，以及维护与再生工程。其中程序设计部分以 C++/C 语言为例。

1.2 软件开发的基本策略

人们都有自己的世界观和方法论，能自然而然地运用于生活和工作中。同样，程序员脑子里的软件工程观念会无形地支配其怎么去做事情。软件工程三十年的发展，已经积累了相当多的方法，但这些方法不是严密的理论。实践人员不应该教条地套用方法，更重要的是学会“选择合适的方法”和“产生新方法”。有谋略才会有好的战术。几千年前，我们的祖先就在打闹之际写下了很多心得体会，被现代人很好地运用于工业和商业。本节讲述软件开发中的三种基本策略：“复用”、“分而治之”、“优化——折衷”。

1.2.1 复用

复用就是指“利用现成的东西”，文人称之为“拿来主义”。被复用的对象可以是有形的物体，也可以是无形的成果。复用不是人类懒惰的表现而是智慧的表现。因为人类总是在继承了前人的成果，不断加以利用、改进或创新后才会进步。所以当我们欢度国庆时，要搞清楚祖国远不止 50 岁，我们今天享用到的财富还有上下五千年人民的贡献。进步只是应该的，不进步则就可耻了。

复用的内涵包括了提高质量与生产率两者。由经验可知，在一个新系统中，大部分的内容是成熟的，只有小部分内容是创新的。一般地可以相信成熟的东西总是比较可靠的（即具有高质量），而大量成熟的工作可以通过复用来快速实现（即具有高生产率）。勤劳并且聪明的人们应该把大部分的时间用在小比例的创新工作上，而把小部分的时间用在大比例的成熟工作中，这样才能把工作做得又快又好。

把复用的思想用于软件开发，称为软件复用。据统计，世上已有 1000 亿多行程序，

无数功能被重写了成千上万次，真是浪费哪。面向对象（Object Oriented）学者的口头禅就是“请不要再发明相同的车轮子了” 。

将具有一定集成度并可以重复使用的软件组成单元称为软构件(Software Component)。软件复用可以表述为：构造新的软件系统可以不必每次从零做起，直接使用已有的软构件，即可组装（或加以合理修改）成新的系统。复用方法合理化并简化了软件开发过程，减少了总的开发工作量与维护代价，既降低了软件的成本又提高了生产率。另一方面，由于软构件是经过反复使用验证的，自身具有较高的质量。因此由软构件组成的新系统也具有较高的质量。利用软构件生产应用软件的过程如图 1.5 所示。

软件复用不仅要使自己拿来方便，还要让别人拿去方便，是“拿来拿去主义”。面向对象方法，Microsoft 公司的 COM 规范 [Rogerson 1999]，都能很好地用于实现大规模的软件复用。

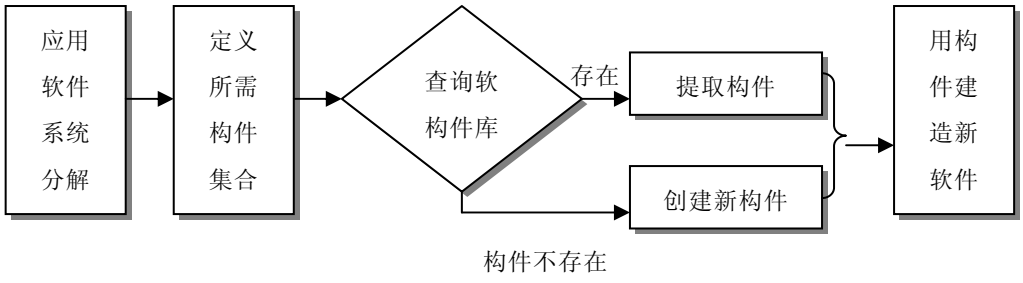


图 1.5　利用软构件生产应用软件的过程

1.2.2 分而治之

分而治之是指把一个复杂的问题分解成若干个简单的问题，然后逐个解决。这种朴素的思想来源于人们生活与工作的经验，完全适合于技术领域。软件人员在执行分而治之的时候，应该着重考虑：复杂问题分解后，每个问题能否用程序实现？所有程序最终能否集成为一个软件系统并有效解决原始的复杂问题？

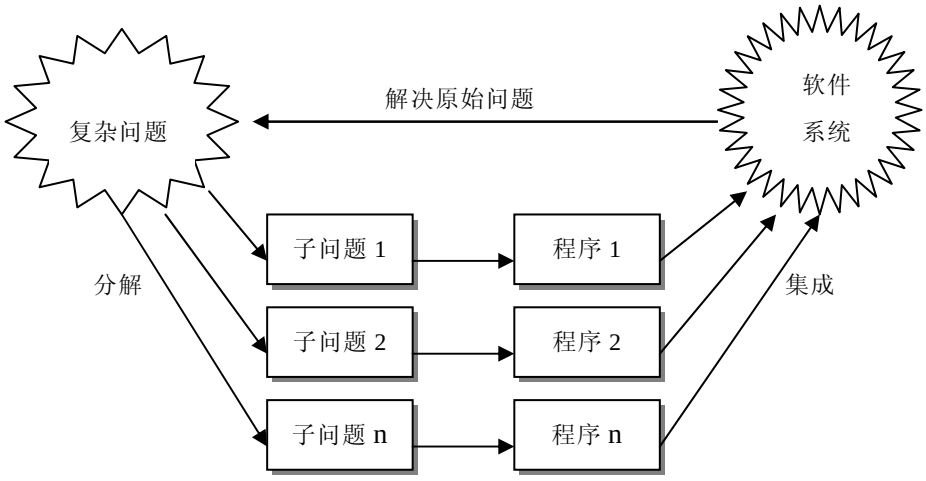


图 1.6 软件领域的分而治之策略

图 1.6 表示了软件领域的分而治之策略。诸如软件的体系结构设计、模块化设计都是分而治之的具体表现。软件的分而治之不可以“硬分硬治”。不像为了吃一个西瓜或是

一只鸡，挥刀斩成 n 块，再把每块塞进嘴里粉碎搅拌，然后交由胃肠来消化吸收，象征复杂问题的西瓜或是鸡也就此消失了。

1.2.3 优化——折衷

软件的优化是指优化软件的各个质量因素，如提高运行速度，提高对内存资源的利用率，使用户界面更加友好，使三维图形的真实感更强等等。想做好优化工作，首先要让开发人员都有正确的认识：优化工作不是可有可无的事情，而是必须要做的事情。当优化工作成为一种责任时，程序员才会不断改进软件中的算法，数据结构和程序组织，从而提高软件质量。

著名的 3D 游戏软件 Quake，能够在 PC 机上实时地绘制高度真实感的复杂场景。Quake 的开发者能把很多成熟的图形技术发挥到极致，例如把 Bresenham 画线、多边形裁剪、树遍历等算法的速度提高近一个数量级。我第一次看到 Quake 时不仅感到震动，而且深受打击。这个 PC 游戏软件的技术水平已经远胜于我所见识到的国内领先的图形学相关科研成果。这对我们日益盛行的点到完止的研发工作真是莫大的讽刺。所以当我们开发的软件表现出很多不可救药的病症时，不要怨机器差。真的是我们自己没有把工作做好，写不好字却嫌笔钝。

就假设我们经过思想教育后，精神抖擞，随时准备为优化工作干上六天七夜。但愿意做并不意味着就能把事情做好。优化工作的复杂之处是很多目标存在千丝万缕的关系，可谓数不清理还乱。当不能够使所有的目标都得到优化时，就需要“折衷”策略。

软件中的折衷策略是指通过协调各个质量因素，实现整体质量的最优。就象党支部副书记扮演和事佬的角色：“…为了使整个组织具有最好的战斗力，我们要重用几个人，照顾一些人，在万不得已的情况下委屈一批人”。

软件折衷的重要原则是不能使某一方损失关键的职能，更不可以象“舍鱼而取熊掌”那样抛弃一方。例如 3D 动画软件的瓶颈通常是速度，但如果为了提高速度而在程序中取消光照明计算，那么场景就会丧失真实感，3D 动画也就不再有意义了（如果人类全是色盲，计算机图形学将变得异常简单）。

人都有惰性，如果允许滥用折衷的话，那么一当碰到困难，人们就会用拆东墙补西墙的方式去折衷，不再下苦功去做有意义的优化。所以我们有必要为折衷制定严正的立场：在保证其它因素不差的前提下，使某些因素变得更好。

下面让我们用“优化——折衷”的策略解决“鱼和熊掌不可得兼”的难题。

问题提出：假设鱼每千克 10 元，熊掌每千克一万元。有个倔脾气的人只有 20 元钱，非得要吃上一公斤美妙的“熊掌烧鱼”，怎么办？

解决方案：化 9 元 9 角 9 分钱买 999 克鱼肉，化 10 元钱买 1 克熊掌肉，可做一道“熊掌戏鱼”菜。剩下的那一分钱还可建立奖励基金。

1.3 一些不正确的观念

本节例举并分析一些不正确的软件工程观念，可帮助初学者少犯相似的错误。

观念之一：我们拥有一套讲述如何开发软件的书籍，书中充满了标准与示例，可以帮助我们解决软件开发中遇到的任何问题。

客观情况：好的参考书无疑能指导我们的工作。充分利用书籍中的方法、技术和技巧，可以有效地解决软件开发中大量常见的问题。但实践者并不能因此依赖于书籍，这是因为：（1）现实的工作中，由于条件千差万别，即使是相当成熟的软件工程规范，常常也无法套用。（2）软件技术日新月异，没有哪一种软件标准能长盛不衰。祖传秘方在某些领域很吃香，而在软件领域则意味着落后。

观念之二：我们拥有最好的开发工具、最好的计算机，一定能做出优秀的软件。

客观情况：良好的开发环境只是产出成果的必要条件，而不是充分条件。如果拥有好环境的是一群庸人，难保他们不干出南辕北辙的事情。

观念之三：如果我们落后于计划，可以增加更多的程序员来解决。

客观情况：软件开发不同于传统的农业生产，人多不见得力量大。如果给落后于计划的项目增添新手，可能会更加延误项目。因为：（1）新手会产生很多新的错误，使项目混乱。（2）老手向新手解释工作以及交流思想都要花费时间，使实际开发时间更少。所以科学的项目计划很重要，不在乎计划能提前多少，重在恰如其分。如果用“大跃进”的方式奔向共产主义，只会产生倒退的后果。

观念之四：既然需求分析很困难，不管三七二十一先把软件做了再说，反正软件是灵活的，随时可以修改。

客观情况：对需求把握得越准确，软件的修修补补就越少。有些需求在一开始时很难确定，在开发过程中要不断地加以改正。软件修改越早代价越少，修改越晚代价越大，就跟治病一样道理。

1.4　一些有争议的观念

本节探讨一些有争议的观念，目的不在于得出“正确”或“错误”的评断，而在于争议会激发更多理性的思考。

争议之一：如果软件运行较慢，是换一台更快的计算机，还是设计一种更快的算法？

作者观点：如果开发软件的目的是为了学习或是研究，那么应该设计一种更快的算法。如果该软件已经用于商业，则需谨慎考虑：若换一台更快的计算机能解决问题，则是最快的解决方案。改进算法虽然可以从根本上提高软件的运行速度，但可能引入错误以及延误进程。技术狂毫无疑问会选择后者，因为他们觉得放弃任何可以优化的机会就等于犯罪。

类似的争议还有：是买现成的程序，还是彻底自己开发？技术人员和商业人士常常会有不同的选择。

争议之二：有最好的软件工程方法，最好的编程语言吗？

作者观点：在软件领域永远没有最好的，只有更好的。能解决问题的都是好方法或是好语言。程序员在最初学习 Basic、Fortran、 Pascal、C、C++等语言时会感觉一个比一个好，不免有喜新厌旧之举。而如今的 Visual Basic、Delphi、Visual C++、Java 等语言各有所长，真的难分优劣。开发人员应该根据客观条件，选择自己熟悉的方法和语言，才能保证合格的质量与生产率。

程序设计是自由与快乐的事情，不要发誓忠于某某主义而自寻烦恼。

争议之三：编程时是否应该多使用技巧？

作者观点：就软件开发而言，技巧的优点在于能另辟蹊径地解决一些问题，缺点是技巧并不为人熟知。若在程序中用太多的技巧，可能会留下隐患，别人也难以理解程序。鉴于一个局部的优点对整个系统而言是微不足道的，而一个错误则可能是致命的。作者建议用自然的方式编程，少用技巧。

《狼三则》的故事告诉我们“失败的技巧通常是技俩”。当我们在编程时无法判断是用了技巧还是用了技俩，那就少用。《卖油翁》的故事又告诉我们“熟能生巧”，表明技巧是自然而然产生的，而不是卖弄出来的。卖油翁的绝技是可到中央电视台表演的，而他老人家却谦虚地说：“没啥没啥，用熟了而已”。

争议之四：软件中的错误是否可按严重程度分等级？

作者观点：在定量分析时，可以将错误分等级，以便于管理。微软的一些开发小组将错误分成四个等级 [Cusumano 1996]，如表 1.1 所示。

一级严重：错误导致软件崩溃。
二级严重：错误导致一个特性不能运行并且没有替代方案。
三级严重：错误导致一个特性不能运行但有替代方案。
四级严重：错误是表面化的或是微小的。

表 1.1　错误的四个等级

上述分类是非常技术性的，并不是普适的。假设某个财务软件有两个错误：错误 A 使该软件死掉，错误 B 导致工资计算错误。按表 1.1 分类，错误 A 属一级严重，错误 B 属二级严重。但事实上 B 要比 A 严重。工资算多了或者算少了，将会使老板或员工遭受经济损失。而错误 A 只使操作员感到厌烦，并没有造成经济损失。另一个示例是操作手册写错，按表 1.1 分类则属四级严重，但这种错误可能导致机毁人亡。

开发人员应该意识到：所有的错误都是严重的，不存在微不足道的错误。这样才能少犯错误。

1.5 小 结

软件工程学科发展到今天，已经有了很多方法和规范，学之不尽。本章只在宏观上讨论了软件工程的一些思想，更具体的内容将在后面的章节论述。无论是什么好方法，贵在理解与灵活运用，而不可当成灵丹妙药，不象“吃了脑黄金或脑白金，就能使一亿人先聪明起来”。

第二章　程序员与程序经理

工作在第一线的软件开发人员是程序员和程序经理，他们决定着软件的命运。良好的程序员队伍和出色的管理是软件项目成功的必要条件。管理不是管制，不是去卡住人家的脖子，因为程序员不是一群野鸭子。管理的目的是让大家一起把工作做好，并且让各人获得各自的快乐和满足。当一个组织被出色地领导时，雇员甚至不知道他们已被领导。在项目完成时，他们会自豪地说：“看看我们通过努力取得的成绩吧”。所以管理者不能老惦记着自己是一个官，而应时刻意识到自己是责任的主要承担者。

我们经常会听到有经理头衔的人在高谈阔论：“编程我不会，做个项目还不 easy？派个人去搞系统分析，回头再叫几个程序员把需求译成程序，不就 OK 了吗？”

不懂英语的人准以为 easy 和 OK 是贬义词。要让软件项目失败很容易，只要符合下列条件之一即可：

（1）项目经理对软件一无所知；

（2）技术负责人对编程不感兴趣；

（3）真真编写代码的程序员是临时雇用的。

如果上述三个条件同时具备，就请放心失败好了。

让我们少幻想自己是比尔·盖茨，先当好程序员和程序经理再说。

2.1　了 解 程 序 员

早期的程序员干活能从软件直通硬件，个个生猛无比。又因他们的作息时间、言行举止与常人不太一样，久而久之就给人们留下了“神秘”、“孤僻”的印象。如今软件行业被炒得热火朝天，有能耐的程序员即便躲在大山岙的军工厂里也能被挖出来。而更多原本不是程序员的人操起几本“速成”、“二十一天通”等书籍也加入了这个行业。现在国内号称有上百万程序员，这支大军鱼龙混杂，已搞不清那些是正规军，那些是民兵游击队了。

真正的程序员都有如下秉性：

一、诚实

程序员在学习与工作期间几乎天天与机器打交道，压根就没有受欺骗或欺骗人的机会。勤奋的程序员在调试无穷多的程序 Bug 时，已经深深地接受了“诚实”的教育。不诚实的人，他肯定不想做、也做不好程序员。

有一名市场营销员和一名程序员都在新闻发布会上发言，将一项新技术的消息公布于众。

市场营销员说：“这项技术比电话、晶体管和原子弹三项发明加起来对世界文明的影响都要大。”

程序员说：“这项技术在有限的领域内，在有限的程度上，解决了一些技术性的问题。”

看来为了让我们的民族更加诚实，学电脑真的要从娃娃抓起。

二、简单——实用主义

有人问一个数学家，一个物理学家和一名程序员：“一个盒子有几个面？”

数学家回答说：“有六个面，因为盒子是长方体。”

物理学家回答说：“有 12 个面，分为 6 个外表面和 6 个内表面 。”

程序员回答说：“只有两个面，里面放电路板和硬盘，外面放显示器和键盘。”

目前即使最先进的计算机也不具备智能，程序员的基本工作就是把复杂的问题转化为计算机能处理的简单的程序。如果一个问题复杂到连程序员自己都不能理解，他就无法编出程序让更笨的计算机来处理。所以程序员信奉“简单——实用”主义。

也有不少做计算机“学问”的人颠倒行事。本来几句话、几行程序就能说明白的事，非得要抬高到理论创新的程度，写成玄乎的文章去评教授或者弄个博士学位。所幸在第一线工作的程序员大多是实干的。

三、爱憎分明

程序员大都喜欢技术挑战，不喜欢搞测试与维护。高水平的程序员喜欢与高水平的程序员一起工作，因为他们怕“与臭棋佬下棋，棋越下越臭”。程序员大都厌恶拉帮结派、耍政治手腕。不信，数一数你认识的程序员，有几个是党派人士？

四、工作单调但不乏味

有人问编程大师：“程序设计的真正含义是什么 ？”

大师回答说：“饿了的时候就吃，困的时候就睡，只要时机恰当就进行程序设计。”

其实程序员的生活和工作已融为一体，尽管单调却不乏味，还能独享孤独。有诗为证：

我编程三日
两耳不闻人声
只有硬盘在歌唱

结论：优秀的程序员没有理由不让人喜欢，他们远比怪僻来得可爱。

2.2 了 解 程 序 经 理

这里程序经理是指一支程序员队伍的领导者，不管他的职务是开发组长，项目经理，还是部门经理。程序经理是技术性的基层或中层干部，是软件企业得以发展的生力军。程序经理的选拔是不容草率的事。不象有些事业单位，只要政治口号喊得勤快、能左右逢缘不犯错误就可混个领导当当。也不象一些官僚机构，只有两个人的办公室也要设正主任和副主任。如果碰巧正主任姓傅，副主任姓郑，还会斗个没完没了。

在一个管理混乱的软件公司里，如果某个程序员能大喊大叫并且干劲十足，那他就能成为一名程序经理。微软公司在选择经理人员时，总是把他们的技术知识和运用技术去赚钱的能力放在首位。程序经理一般就是程序员队伍中最聪明的那个家伙。比尔·盖茨曾这样描述聪明人[Cusumano1996]：

聪明人一定反应敏捷，善于接受新事物。他能迅速进入一个新领域，给你一个头头

是道的解释。他提出的问题往往一针见血、击中要害。他能及时掌握所学知识，并且博闻强记，他能把本来认为互不相干的领域联系在一起使问题得到解决。他富有创新精神与合作精神……

好的程序经理应该具备以下几个条件：

一、技术水平是程序员队伍中的最高级别

每个程序员骨子里头都有一股傲气，如果你不能技压群雄，他们就不会听你指挥。一个技术水平较差的人被任命为程序经理真是个悲剧，就象一个略有权势的太监，表面上有人对他点头哈腰，背后却被人鄙视。

二、能做最多且最难的工作

程序经理编程要快且好。别人要干一天的活，他半天就能做完，这样才会有精力去搞管理。程序经理应负责系统分析、系统设计这类最难的开发工作，并指导不同水平的程序员把各自的工作做好。如果人手不够，程序经理要能同时干几个人的活。

三、有人格魅力

软件开发是智力创作过程，你不能指望仅通过执行规章制度来产生好的作品。很多软件公司的程序经理都不是管理专业出身的，他们也不可能为了搞好管理而成天玩弄心机。技术出色的程序经理一般少有心术不正的，所以管理的重点应是“以身作则”、“公正待人”。如果程序经理在上班时趴在桌上睡觉，其他程序员也会这样干。如果程序经理发现有两个程序员趴在机器旁睡觉，不能只对其中一个大声吼叫：“你一编程就想睡觉，看看人家，在睡觉时都想着编程。”

如果管理者没有人格魅力，就没有人信服你，团队就不会有凝聚力，乌合之众不可能开发出优秀的软件。

结论：一个有活力的软件公司的各级经理都不会这样感叹，“因为我啥也不会干，所以只好当领导。”

2.3 程序员升为经理后是否还要编程

让我们先看看 Microsoft 公司的系统软件部门与应用软件部门的领导是怎样看待这个问题的[Cusumano1996]。Windows NT 3.0 项目的软件经理娄•帕雷罗里让他手下的经理们像他一样每天花一半的时间编写代码：

我在组内制定了许多规则，其中最重要的一条是每个人都得编程，谁也别想坐在那儿发号施令……我发现管理者很容易失去目标，他们总是无法认识到问题的本质并且反应迟缓。如果你始终不放弃编写代码，你就能对项目的进展情况了如指掌，及时发现并解决问题……我大概每天花一半的时间编写代码并寻找项目的缺陷。

作为应用软件领域的经理，克里斯•彼得斯也持同样的看法。在他任 Word 项目总经理时就认为：

在一些大公司内部，各部门经理把具体操作的层次向下移。你一旦当上开发部门经理，很快就会以自己身居高位、日理万机为由放弃编程；同样地，开发小组的组长会以自己重任在肩而不愿编程；至于程序员也会觉得自己十分繁忙、分身无术而不再多编写程序。虽然我是 270 名员工的领导，似乎不再需要做什么具体的工作了，但我还是为 Word

新版本编写了一个特性。

程序员升为经理后一定要编程，这个道理已经说得很清楚了。最怕的是“虚心接受，坚决不做”；或者仅是做个样子，每天花一分钟时间编程，编译器还没运行完就关掉了。

2.4 经理与技术队伍的建设

如果是经营一个加工厂或一个饭店，经理们可以不必懂技术。因为他们的常识，以及通过耳闻目睹或者咨询都能解决实践中的问题。在软件领域，技术的力量是无穷的，一天之内就可使整个产业发生巨变。也许你在商业上很精明，但无法保证自己在技术浪潮中安然无恙。软件公司的各级经理最好既精通技术又懂管理。

一个出色的领导，加上一支技术过硬的队伍，才有可能创造业绩。不能光指望请来孙子或诸葛亮当教练，就能让弱不禁风的男足去捧世界杯。不少人总喜欢自吹中国人很聪明，最适合搞软件开发。可至今也没有做出几个很光彩的软件来，这与十三亿人口不呼应啊。新中国历来喜欢与可怜的印度相比较来展现丰富多彩的优越性，可是软件产业没法与人家比。工作在第一线的程序员与程序经理应该意识到：好兵好将都不是天生的，是后天练出来的；既要学会冷静地分析问题，又要充满激情地去工作。

软件公司总希望能物色到既精通技术又善长商业的优秀人才做经理。但已经出名了的优秀人才难以请到，也难以留住。所以把公司中的普通员工培养成为优秀人才是重要的举措。公司的老板不要对程序员抱有偏见，以为他们只配与机器打交道。一个高水平的程序员既然能学好数字逻辑，能理得清楚软件中很多象“嵌套”这类“鸡生蛋并且蛋又生了鸡”的错综复杂的关系，从理论上讲当个县长也不成问题。

现在很多女士不会烧菜，却能把菜的营养讲得头头是道。虽然这是个值得哀叹的社会问题，但我们应该有信心期待：如果她们非得天天烧菜不可，那么不久就能把菜烧得又好吃又有营养。许多程序员不懂商业，不是智力上的原因，主要是个人兴趣和环境所致。软件公司的老板应该这样鼓励有灵气的员工：“你能把技术做得那么棒，还怕搞不好管理？放心干吧！”的确，很多技术人员是在工作中领悟如何管理的，他们经过挫折与磨练，逐渐升为组长、项目经理，乃至成为公司重要的决策者。

优秀的程序员喜欢与优秀的程序员一起工作，这是一种理想的愿望。一个普通的软件公司不可能有非常多的优秀程序员，即便有，他们也不可能天天聚在一起干同一件事并且和睦得无法形容。中国自封建社会起就有喜好内斗的风俗习惯，几千年下来早已渗透到社会各个角落，那怕黄河水流断了，估计这民风也会延袭下去。要使程序员队伍稳健，必须有合理的等级制度来维护。等级制度并不限制自由和民主，它能让自以为聪明绝顶、谁也不服的人们懂得如何合作与奋斗。就象有了一架梯子，每个人才有机会爬上墙头摘下那向往已久的野花。当梯子散成一堆木棍时，只可能造就几个卖炭翁。

下面我们尝试着建立一个程序员队伍的等级制度。

把技术水平分为四级，第一级最低，第四级最高。第一级技术水平的程序员主要考核编程基本功，要求质量合格（他们主要来自刚毕业的大学生）。第二级技术水平的程序员编程质量要高，做过几个软件项目，有数年的工作经验，并能指导新手的工作。第三级技术水平的程序员主要考核系统分析与系统设计的能力，要求其技术有足够的深度和

广度。第四级技术水平的程序员是成功的软件产品的设计师，他不仅技术超群，并且能使技术转化为有价值的商品。

把管理（这里仅指软件业务的管理，不考虑行政事务）水平也分成四级。第零级最低，第三级最高。第零级管理水平的人没有管理职务，就是普通员工。第一级管理水平的人是开发小组的组长，可带领几名程序员工作。第二级管理水平的人是项目经理。第三级管理水平的人决定某些产品是否要开发，以及如何去占领市场。

每个程序员都有明确的技术级别和管理级别。技术级别与管理级别有一定的联系。一般地，第一级技术水平的人只能做普通员工；第二级技术水平的人可以当一名组长；第三级技术水平的人可以当一名项目经理；第四级技术水平的人可成为公司产品的决策者。如图 2.1 所示。本书作者目前的技术水平当属第二级，管理水平符合组长的要求。作者在读中学和大学时就曾美滋滋地当过课代表，也就是组长级别。

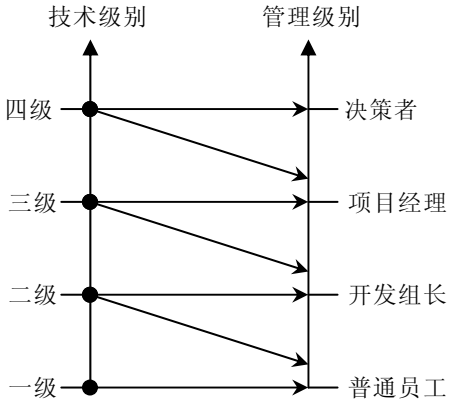


图 2.1 技术级别与管理级别

2.5 向错误与失败学习

不管是生活或工作，人们都应该向错误与失败学习，目的是让我们在短暂的健康年华中少犯错误、少失败，多做几件正确的对社会有贡献的事。

导致软件项目失败的因素很多，如果不去找借口的话，就会发现错误的根源在自己身上：知识贫乏、才能低下、经验不足、骄傲自负……。我们必须正视自身的不足与缺点，才会学到经验教训。可人们常有太多的虚荣，为了克服心理障碍，白白浪费了很多本该用于创造的精力。

假设犯错误的人是诚实的并且是勤奋的。他愿意不带虚荣地改进自己。当这个人突然面对失败时，可能觉得自己一无是处，也许会不知所措，也许会病急乱投医。程序员都有一种共同的体会：在调试程序时，时常碰到只有十几行的程序竟会产生上百个编译错误；最后发现这么多的错误其实是由某一行程序错误引发的。当我们在工作中碰到挫折时，先要冷静地分析问题（事出有因哪），找出问题的内因与外因。内因是最主要的，应该予以最先解决。

前几年，中国出现了一个叫“法轮功”的邪教，教徒达数百万之多，人民群众深受

其害。不久前，全国的主要媒体对“法轮功”进行连续数月的声讨与揭露。目睹了很多受害人的哭诉后，相信人们能够明白“法轮功”是邪恶的、反动的。但在愤怒与心痛之余，我们不禁要反思：为什么那么多人轻信邪教？人们是否接受了教训？

在电视上看到很多人的确作了深刻的检讨：“我真是后悔啊，跟错了李洪志（法轮功的头头）这个坏蛋，我对不起社会……。以后我一定要听党组织的话，党叫我干什么我就干什么，决不上坏人的当。”

我觉得这些受害人一点都没有醒悟：他只知道法轮功是个邪教，并不知道自己为什么信了邪教。有些事情只要用脑袋去想一想就能分辨是非，可人们就是不去思考，却渴望能跟对“福星”，甘愿把自己的脑袋拴在别人的裤带上。难道这就是人民的纯朴与可爱吗？回顾一下历史，在“文革”时期，亿万人民跟着合法的党组织大干伤天害理的事，一干就是十多年哪！可见世界上哪个人哪个组织都不能确保绝对的英明。

所以说“迷信”是傻子碰到骗子的结果。傻是内因，被骗是外因。傻子碰到好人未必能做出好事，傻子碰到另一个骗子就会做出另一件傻事。为了不让自己“傻”，善良的人们应该用脑子去多学一些知识，努力让自己来把握命运，不要急着把一生托给某个人或某个组织。

软件人员在遭受项目失败并开始反省时，不要只是就事论事地仅把眼光锁在特定的项目上，吃一堑应该长好几个智才对。本书作者刚刚失败过，乐意乘热讲讲感受。

我在读本科和硕士研究生时，一直信奉“创造性的事业要靠激情来推动”。我把这个口号贴在办公室里，并扔掉物理学专业天天编程。在读硕士研究生的第一年，我卖出了第一份软件。到我读博士研究生的第一年，我心想事成地获得了全国大学生电脑大赛软件展示第一名。那时候我自以为翅膀已经硬了，再回顾前些年的艰苦，不禁有“媳妇熬成婆”的悲壮感觉。于是我在杭州这个小地方略作宣传，在 1997 年 10 月份开了一家软件公司。

我开始把“振兴民族软件产业”列入日程，并且提前担忧将来钱挣得太多用不完该怎么办。半年之后，我开始为软件产品作宣传，可并没有出现订单如潮、接应不暇的形势（事实上压根就没有反应）。我已经意识到市场没找对，但仍觉得软件中的技术很有价值，准备再开创“东方不亮西方亮”的新局面。于是我向只有一面之缘尚在北大方正工作的一位朋友求助。他是真真的软件高手，当我小心翼翼地展示约 10 万行 C++代码的软件时，他竟在十几分钟内就指出多处重大的设计错误，使我目瞪口呆地意识到整个软件系统的价值为零。那种心痛啊，就象眼睁睁看着孩子被狼吃掉一样。

1998 年 10 月，这位朋友再一次从北京飞到杭州，三下五除二替我把只活了一年的公司给关闭掉。他放心不下，觉得我“恶病需用猛药补”，于是意尤未尽地把我捉到北大方正插在他管辖的部门，让我学习怎样做事情。北京寒冷的冬天可以营造一种凄凉的气氛，冲去一切可以自我原谅的借口。我并不是太爱虚荣的人，知道这次失败是我的毛病积累到一定水准忍不住喷发出来的结果。我绝不能以年纪尚轻不太懂市场与管理为理由轻率地敷衍过去。我把自己察觉到的数十个毛病列出来，日后一个一个克服掉。……本书的大部分内容取自我在一年前的教训录。

改错之后，现在我不仅不难过而且挺快乐。觉得第一次失败很浪漫，值得怀念。刚开始写这本书时，我那位北京的朋友把脚伸到杭州来散步，顺手又给了我几帖药，可以用到我毕业。看来缺点是改不完的，补短和扬长要一起来。

2.6 提高综合素责

前面给软件开发人员加了过多的赞誉。一个技术出色的程序员可以自豪，但不可以目空一切。上天不可能赋于一个人太多的优点，以致于他没有表示谦虚的余地。

我们在求学时可能太功利太挑剔，导致知识结构非常单薄，只怕到了晚年也成不了大器。当程序员擅长技术时，还要时刻留意弥补自己并不擅长的非技术才能。扬长补短才能提高综合素质。

假如能回到中学时代，我希望能把文科学好。那时侯盛传“学好数理化，走遍天下都不怕”。我读中学时很无知，鄙视一切文科，现在后悔莫及。高考语文成绩 54 分（只比我的期望值低 6 分）。写作文的最高目标就是不逃题，考试前我总是反复祈祷：我没干过坏事，保佑我作文不逃题吧！上大学的第一天我竟然无法用普通话说出“去洗澡怎么走”，只好晃动澡票与辅导员打哑语。中学的历史、地理课也被我糟踏了，考试时只会填写任课老师某年某月某日在我家乡英勇就义，比谁的成绩更接近零分。更让我沮丧的是，这些行径都不是我发明的，我顶多是个跟屁虫而已，一点回忆的自豪感都没有。

扔掉文科只学理科并不等同于“放下包袱，轻装前进”，倒象是摘掉了控制系统的机车，开不了多远就翻车了。我搞了八年的软件开发，没做出象样的软件来。倒是有同行意外发现我的文笔不错，是当作家的料。我发现自己在不该开花的地方结了一颗瘦涩的果子。曹操之子曹彰曾建议：“大丈夫当学卫青、霍去病，立功沙漠，长驱数十万众，纵横天下，何能为博士耶？”要后悔的事情太多了，只能现在做得勤快些。明知自己不成大器，但愿意亡羊补牢，力求学得更深更广。

不要让人觉得程序员只管钻研技术，可以不懂世事并且应该自由散漫。程序员不该因为幼稚而显得单纯，应该是成熟了才变得单纯，才配得上这个充满活力的职业。

2.7 小 结

本章讲述做好程序员和程序经理的一些道理，为了剥去阻碍我们进步的那些虚伪，多唠叨了几个故事。

中国经历了很多打斗、整人的革命，却没有一次赶上工业革命。在如今计算机横行的形势下，我们不能再掉队了。90 年代初期，中国出现了一些程序员英雄，曾让我们激动过、崇拜过。但这些孤胆英雄们很快地几乎全消亡了，他们只留下故事，没留下更多的价值。再一次让我们意识到“振兴民族软件产业”不能依靠几个人一朝一夕的辉煌。软件人员勤奋学习和工作，不该只图将来能做成几件事情的快意，而应力求事业长盛不衰，才能推动整个民族软件产业持久稳健地发展。

第三章　项目计划与质量管理

在可行性分析之后，项目计划与质量管理将贯穿需求分析、系统设计、程序设计、测试、维护等软件工程环节。

项目计划是要提供一份合理的进程表，让所有开发人员任务明确、步调一致，最终共同准时地完成项目。项目计划是要付诸实施的，不象用嘴巴喊政治口号，可以很夸张。软件的项目计划重在“准确”而非“快速”。

提高质量是软件工程的主要目标。但由于软件开发是一种智力创作活动，很难象传统工业那样通过执行严格的操作规范来保证软件产品的质量。世上最小心翼翼、最老实巴脚的程序员未必就能开发出高质量的软件来。程序员必须了解软件质量的方方面面(称为质量因素)，如正确性、性能、易用性、灵活性、可复用性、可理解性等等，才能在进行系统设计、程序设计时将高质量内建其中。软件的高质量并不是“管理”出来的，实质上是设计出来的，质量的管理只是一种预防和认证的手段而已。

3.1　项 目 计 划

做项目计划，如同给一个待出生的婴儿写传记那样困难。如果允许项目结束后再写计划，那就轻松多了，并且可以 100% 地准确。

历史教训让我们明白一个道理：如果一万年以后才会有一条阳光大道通向共产主义，那么现在就不要忙着砸锅炼钢赶英超美，免得在跑步奔向共产主义时把自己累死饿死。在做软件的项目计划时，应屏弃一切浮夸作风。只有“知己知彼”才能做出合理的项目计划。这里“知彼”是指要了解项目的规模、难度与时间限制。“知己”是指要了解有多少可用资源，如可调用的程序员有几个？他们的水平如何？软硬件设施如何？

3.1.1 知己知彼

首先要了解项目的规模、难度与时间限制，才可以确定应该投入多少人力、物力去做这个项目。在可行性分析阶段就要考虑这个问题。但不幸的是，人们在陷入项目不能自拔之前总难以准确地估计项目的规模与难度。这里经验起到了最重要的作用。

项目的时间限制有两类。第一类，项目应该完成的日期写在合同中，如果延期了，则开发方要作出相应的赔偿。第二类是开发自己的软件产品，虽然只确定了该产品大致的发行日期并允许有延误，但如果拖延太久则会失去商机造成损失。

项目的资源分为三类：“人”、“可复用的软构件”和“软硬件环境”，如图 3.1 所示。

（１）人是最有价值的资源。项目计划的制定者要确定开发人员的名单，要根据他们的专长进行分工。

（２）可复用的软构件是次有价值的资源。1.2.1 节论述了复用软构件可提高软件的质量与生产率。软构件并非一定要用自己的，可以向专业的软件供应商购买。

（３）软硬件环境虽然不是最重要的资源，却是必需的资源。原则上软硬件环境只要符

合项目的开发要求即可。有些项目可能要用到特殊的设备，则要事先作好准备，以免用时找不到而担搁了进程。

图 3.1 项目的资源

3.1.2 进度安排

有一位程序员忙着编写程序，经理问他还需要多久才能完成。

“明天就可以完成。”程序员立即回答。

“我想这是不切实际的，实话实说，到底还要多少时间？”经理说。

“我还想加进一些新的功能，这需要花两个星期。”程序员想了一会儿说。

“即使这样也期望过高了，只要你编完程序时告诉我一声，我也就满足了。”经理说。

几年以后，经理要退休了。在他去退休午餐会时，发现那位程序员正趴在机器旁睡觉：可怜的家伙整个晚上都在忙于编写那个程序。[James 1999]

程序员也期望每天早晨能在 7:00 准时起床，可老是一觉醒来就到中午了。项目落后于进度表乃是家常便饭，不必大惊小怪。以下一些事件经常会导致项目被延误：

（1）上级领导主管臆断，制定了不现实的期限。项目经理与程序员们被迫按照不合理的进度表开展工作。

（2）客户的需求发生了变化，但没有对进度表作出相应的修改。

（3）低估了项目的规模与难度，导致投入的人力和物力不足。

（4）并未预见到存在难以克服的技术障碍。

（5）并未预见到开发人员会发生问题，如生病，辞职等等。

（6）开发人员之间不能很好的交流、协作，导致各阶段任务难以如期完成。

所以写进程表不能象小学生写决心书那样充满幻想。以下是一些有益的建议：

（1）制定进度表的人最好就是项目负责人，他最了解项目和开发人员。进度表要经过开发小组的讨论，在得到大部数人的支持后才能实施。避免出现一厢情愿的局面。

（2）进度安排并不见得一定要符合逻辑顺序。应尽可能地先做技术难度高的事，后做难度低的事。也就是辛苦在前，轻松在后。

小时候我对一位老先生吃饭很感兴趣：他总是先把一大盒的米饭吃光了，然后再幸福地品尝一小盒菜。父母告诉我这是中国的传统美德，叫“先苦后甜”。从此我铭记在心，按此道理去学习和工作。可如今在饭店里，人们总是先把菜吃完了，最后才吃点米饭。天哪，生活真是太复杂了，我究竟该“先吃饭” 还是“先吃菜”？

（3）开发一个大的软件项目，应该将进度表分为若干个里程碑。一个里程碑之内的多个任务可以同步进行。程序员极容易沉迷于技术，要么乐不思蜀，要么焦头烂额。里程碑就象心灵的灯塔，使忙碌的人群不混乱，不迷失方向。

（4）进度表中必须留有缓冲时间，并将缓冲时间用到不确定的事情上。因为人们对即将

要做的事情知之甚少，所以要留一些时间以防不测。Microsoft 公司的一些开发小组甚至制定了“50% 缓冲规则”[Cusumano 1996]。对许多项目经理而言，容忍进度表中存在缓冲时间，不啻为观念上的一个飞跃。

（5）如果发现项目应交付的期限非常不合理，就要跟领导或跟客户据理力争，请求放宽期限、调整进度。当客户的需求发生变化时，就要对进度表作出相应的修正。不要觉得修改进度表很困难很麻烦，不修改才会产生真真的麻烦。很多人认为戒烟很困难，但马克·吐温曾说：“戒烟很容易，我一年就戒几十次。”

3.2 零缺陷质量管理的观念

“零缺陷”质量管理的观念来源于一些国际上著名的硬件生产厂商。尽管软件的开发与硬件生产有极大的差别，但我们仍可以从“零缺陷”质量管理中得到启迪。“零缺陷”质量管理至少有两个核心内容：一是高目标，二是可执行的规范。

3.2.1 高目标

人在做一件事情时，由于存在很多不确定的因素，一般不可能 100% 地达到目标。假设平常人做事能完成目标的 80%。如果某个人的目标是 100 分，那么他最终成绩可达 80 分。如果某个人的目标只是 60 分，那么他最终成绩只有 48 分。我们在考场上身经百战，很清楚那些只想混及格的学生通常都不会及格，那些想得高分的学生也常为自己的失误而捶胸顿足。

做一个项目通常需要多个人的协作。假设项目的总质量（最高为 1）是十个开发人员的工作质量之积。如果每个人的质量目标是 0.95，那么十个人的累积质量不会超过 0.19。如果每个人的质量目标是 0.9 分，那么十个人的累积质量不会超过 0.03。只有每个人都做到 1，项目总质量才会是 1。

如果没有高目标，人的堕落就很快。如果没有“零缺陷”的质量目标，也许缺陷就会成堆。

3.2.2 可执行的规范

实现 100 分显然比实现 80 分要付出更多的努力。“零缺陷”质量目标不是随心所欲提出来的，做得到才有意义。实现高目标需要一套可执行的规范来保证。

50 年代末，全国掀起了“浮夸风”。为了实现亩产数万斤推广各种方法，害得全国闹饥荒。想不到有数千年种粮经验的几亿中国农民就这么整齐地栽倒了。

好规范必须是本企业有能力执行的。一个普通企业照搬一流企业的规范未必行得通。软件工程的规范很容易从书籍中找到，但有了这些规范并不表明就能把软件做好。国内很多软件公司根本没有条件去执行业界推荐的软件工程规范。社会主义初级阶段的“草”与发达资本主义国家的“苗”的确有不同的培育方式。

软件是如此的灵活，如果没有规范来制约，就容易因无序的喜好而导致混沌；但规范如果太严密了，就会扼杀程序员生机勃勃的创造力。制定软件规范是进退两难的事。

程序员必须深入了解软件多方面的质量因素，把那些能提高软件质量因素的各种规范植入脑中，才能在各个实践环节自然而然地把高质量设计到软件中。

3.3 软件的质量因素

“运行正确”的程序就是高质量的程序吗？

不贪污的官就是好官吗？

时下老百姓对一些腐败的地方政府深痛恶绝，对“官”不再有质量期望。只要当官的不贪污，哪怕毫无政绩，也算是“好官”。也有一些精明的老百姓打出旗号：宁要贪污犯，不要大笨蛋。相比之下，程序员是够幸福的了。因为我们能通过努力，由自己来把握软件的命运。那么就不要轻易放弃提高软件质量的权利了。

“运行正确”的程序不见得就是高质量的程序。这个程序也许运行速度很低并且浪费内存；也许代码写得一塌糊涂，除了开发者本人谁也看不懂也不会使用。正确性只是反映软件质量的一个因素而已。

软件的质量因素很多，如正确性、精确性、可靠性、容错性、性能、效率、易用性、可理解性、简洁性、可复用性、可扩充性、兼容性等等（还可以列出十几个）。这些质量因素之间“你中有我，我中有他”，非常缠绵。如果程序员每天要面对那么多质量因素咬文嚼字，不久就会迂腐得象孔乙已，并且有找不到女朋友的危险。

为了便于理解，可以参照武侠小说中的武学分类，将质量因素粗略地分成几大派。你想那武学源源流长，相互渗透，谁能数得清有多少江湖派别。但想在道上混，总得知道六大门派：“少林派”、“武当派”、“峨嵋派”、“华山派”、“昆仑派”和“崆峒派”。软件质量因素的分类如图 3.2 所示。其中“正确性与精确性”排在首位，地位如同“少林派”与“武当派”；而“性能与效率”，“易用性”，“可理解性与简洁性”和“可复用性与可扩充性”亦是举足轻重的质量因素，地位仿佛“峨嵋派”，“华山派”，“昆仑派”和“崆峒派”。其它的质量因素总可以在图 3.2 中找到合适的亲缘关系，本节不再一一细表。

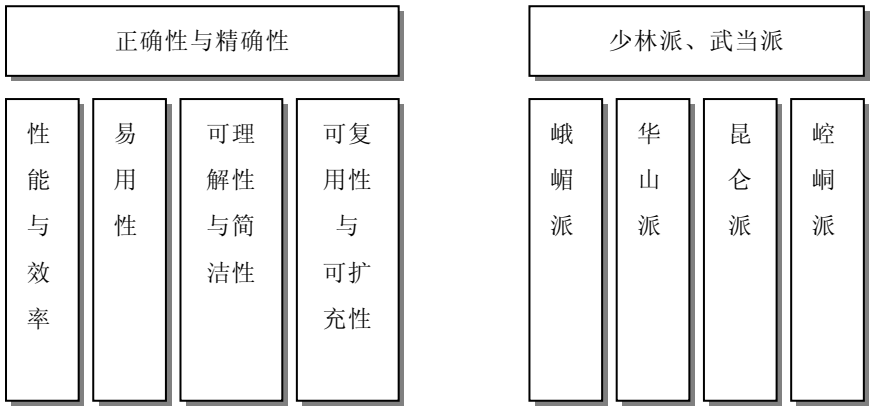


图 3.2　软件质量因素分类和武学分类

3.3.1 正确性与精确性

正确性与精确性之所以排在质量因素的第一位，是因为如果软件运行不正确或者不

精确，就会给用户造成不便甚至造成损失。机器不会主动欺骗人，软件运行不正确或者不精确一般都是人造成的。即使一个软件能 100% 地按需求规格执行，但是如果需求分析错了，那么对客户而言这个软件也存在错误。即使需求分析完全符合客户的要求，但是如果软件没有 100% 地按需求规格执行，那么这个软件也存在错误。开发一个大的软件项目，程序员要为“正确”、“精确”四个字竭尽精力。

与正确性、精确性相关的质量因素是容错性和可靠性。

容错性首先承认软件系统存在不正确与不精确的因素，为了防止潜在的不正确与不精确因素引发灾难，系统为此设计了安全措施。在一些高风险的软件系统，如航空航天、武器、金融等系统中，容错性设计非常重要。

可靠性是指在一定的环境下，在给定的时间内，系统不发生故障的概率。可靠性本来是硬件领域的术语。比如某个电子设备，一开始工作很正常，但由于工作中器件的物理性质会发生变化（如发热），慢慢地系统就会失常。所以一个设计完全正确的硬件系统，在工作中未必就是可靠的。软件在运行时不会发生物理性质的变化，人们常以为如果软件的某个功能是正确的，那么它一辈子都是正确的。可是我们无法对软件进行彻底地测试，无法根除软件中潜在的错误。平时软件运行得好好的，说不准哪一天就不正常了，如“2000 年”问题。因此把可靠性引入软件领域是有意义的。我曾买了一本关于软件可靠性的著作，此书充满了数学公式。我发现以我目前的学历实在难以看懂书上讲了些什么。请宽恕我的愚昧，我把此书给“供”起来，没敢用笔画一处记号。

3.3.2 性能与效率

用户都希望软件的运行速度高些（高性能），并且占用资源少些（高效率）。旧社会地主就是这么对待长工的：干活要快点，吃得要少点。程序员可以通过优化算法、数据结构和代码组织来提高软件系统的性能与效率。优化的关键工作是找出限制性能与效率的“瓶颈”，不要在无关痛痒的地方瞎忙乎。如果你想职称升得快，光靠增加课时能顶屁用；你就该一年写它几十篇文章，争取破格升教授。

3.3.3 易用性

易用性是指用户感觉使用软件的难易程度。用户可能是操作软件的最终用户，也可能是那些要使用源代码的程序员。现代人的生活节奏快，干啥事都想图个方便。所以把易用性作为重要的质量因素无可非议。

导致软件易用性差的根本原因是开发人员犯了“错位”的毛病：他以为只要自己用起来方便，用户也一定会满意。俗话说“王婆卖瓜，自卖自夸”。当程序员向用户展示软件时，常会得意地讲：“这个软件非常好用，我操作给你看，……是很好用吧！”软件的易用性要让用户来评价。当用户真的感到软件很好用时，一股温暖的感觉油然而生，于是就用“友好”来评价易用性。

3.3.4 可理解性与简洁性

可理解性表达了人们一种质朴的愿望：我化钱买了它，总得让我明白它是什么东西。我小时候的一个伙伴在读中学时，就因无法理解电荷之分正负，觉得很烦恼，便早早地缀学当工人。

可理解性也是对用户而言的。开发人员只有在自己思路清晰时才可能写出让别人能理解的程序。编程时还要注意不可滥用技巧，应该用自然的方式编程。我们的确不知道自己的得意之举究竟是锦上添花，还是画蛇添足。就象蒸出一笼馒头，在上面插一朵鲜花，本想弄点诗情画意，却让人误以为那是一堆热气腾腾的牛粪。

简洁是一种美，不管是自己还是用户都会有同感。在生活中，与简洁对立的是“罗里罗嗦”。中国小说中最“婆婆妈妈”的男人是唐僧。有一项民意调查：如果世上只有唐僧、孙悟空、猪八戒和沙僧这四类男人，你要嫁给哪一类？请列出优先级。调查结果表明，现代女性毫不例外地把唐僧摆在老末。

一个原始的应用问题可能很复杂，但高水平的人就能够把软件系统设计得很简洁。如果软件系统臃肿不堪，它迟早会出问题。简洁是人们对工作“精益求精”的结果。

废话大师有句名言：“如果我令你过于轻松地明白了，那你一定是误解了我说的话。”我最近有一种奇怪的体会：如果把学术文章写得很简洁，让人很容易理解，它往往中不了；只有加上一些玄乎的东西，把本来简单的弄成复杂的，才会增加投稿的命中率。事实上，我可以在 5 分钟之内说清楚三年来读博所做的工作，根本用不着写 100 多页的博士论文。我是在临近毕业时，才发觉自己完全不适合读博士学位。将来工作后，我一定要好好编程，重新做人。

3.3.5 可复用性与可扩充性

复用的一种方式是原封不动地使用现成的软构件，另一种方式是对现成的软构件进行必要的扩充后再使用。可复用性好的程序一般也具有良好的可扩充性。本书第六章将论述如何设计可复用、可扩充的 C++程序。

3.4 质 量 检 查

检查是人们不信任自己和别人的一种行为。当某些事情涉及到利益分配时，更需要有检查活动来保证公平。估计即使进入了共产主义社会，也少不了检查。

质量检查并不是要等到项目结束时才执行唯一的一次，应该在每个实践环节都要执行。对应于进度表，在每个里程碑到达时执行质量检查比较合理。质量检查的内容有二：一是作出评审，是合格还是不合格？能打多少分？二是作出建议，对质量为什么好为什么差进行分析，以便“改差为好”、“好上加好”。

以下是人们经常采用的软件质量检查措施[Pressman 1999]：

（1）事先把检查的主要内容制成一张表，使检查活动集中在主要问题上。

（2）只评审工作，不评审开发者。评审的气氛应该是融洽的。存在的错误应该被有礼貌地指出来，任何人的意见都不应被阻挠或小看。

（3）建立一个议事日程并遵循它。检查过程不能放任自由，必须排照既定的方向和日程进行。

（4）不要化太多的时间争论和辩驳。

（5）说清楚问题所在，但不要企图当场解决所有问题。

（6）对检查人员进行适当的培训。

……

做好检查工作并不是件容易的事。自古以来“上有政策，下有对策”。 虚假的质量检查还不如不检查，下面讲两个故事作为解释。

故事一

不久前我回到西北那所读了六年多的大学，惊奇地发现校园里房前屋后长满了待收割的小麦！这所大学是从事电子科技的，种小麦干啥呀？朱总理曾讲过：“目前国家粮食充足，再来三年自然灾害也不怕。”现在国泰民安，似乎用不着“深挖洞，广积粮”。我素知学校提创勤俭节约、自力更生，但与其种小麦还不如种蔬菜呢。老同学告诉我，种小麦是为了应付“211”工程（为 21 世纪选拔 100 所重点大学）的检查团，因为“211”工程有较高的绿化指标。偏偏检查赶在冬天，那时的西北极难长草。我那所大学本来就人多地少，地上一长草马上就会被谈恋爱的学生给折磨死。一到冬天，整个校园就光秃秃一片。用小麦绿化校园可谓千古绝笔，检查团的那些权贵人士早已五谷不分，岂知所见的“草坪”乃是麦田。

检查工作要预防被检查者弄虚作假。

故事二

我上高中时，班里举行一次入团评审。侯选人中有几位是好学生，有几位是坏学生。我心想“伸张正义”的机会到了，绝不能让坏蛋混进纯洁的团里。可天知道团支部书记是聪明绝顶还是蠢笨之极。他竟说：“班里还有一些同学没有入团，现在他们申请入团，有不同意的请举手。”我们都不知道该怎么办了。书记接着说：“既然没有人举手反对，就表示全部同意，请大家鼓掌欢迎。”这次入团评审不到一分钟就结束了，从此后我再也没想过争取入党。

检查工作要有科学的评审方式。

3.5 小 结

不知为什么，国内很多大的企业都喊着要进世界 500 强。如果真的实现了，世界 500 强还不全被中国霸占了。软件的项目计划和质量管理都不是用来喊叫的口号。做项目计划时切忌“冒进”，不要指望在项目陷入困境后靠增加人手来解救。软件的高质量主要是设计出来的，不是“管”出来的，更不能依赖质量检查。为此程序员要充分了解软件的质量因素，只有提高设计水平，才能开发出高质量的软件。

第四章　可行性分析与需求分析

可行性分析是要决定“做还是不做”。

需求分析是要决定“做什么，不做什么”。

即使可行性分析是客观的、科学的，但决策仍有可能是错误的。因为决策者是人，人会冲动，有赌博心态。如果可行性分析表明做某件事的成功率是 10%，失败率是 90%，倘若该事情的意义非常大，决策者也许会一拍脑袋：“豁出去，干!”于是这世界就多了一份极喜与极悲。4.1 节讲述可行性分析的四大要素：经济、技术、社会环境和人。

目前国内很多软件公司做系统集成项目，如果谈谈系统集成项目的可行性分析将很有意思。可是那些系统集成项目大多是政府机构的，由于软件行业尚不规范并且客户方存在腐败现象，所以业内流传“没有做不了的系统集成项目”。软件公司的注意力几乎全集中在“如何拿到项目订单”以及“拿到订单后如何蒙混过关”上，使我丧失了卖弄“可行性分析”的机会。既然不能正面指点一个人如何做好事，那么就规劝他不要做坏事吧。

4.2 节讲述可行性分析案例——投资软件公司失败的教训。作者本来没有资格谈论投资，但事有凑巧：近一年来我关闭了一个亏损 30 万元的软件公司（我自己的）；休克一个年亏损 200 万元的软件公司（朋友的）；扼杀一个 200 万元的投资方案（陌生人的）；踩灭一个处于萌芽状态的 100 万元的投资设想（熟人的）。鉴于现在比较富有的民营企业渴望投资软件行业的越来越多，值得谈谈这方面的可行性分析。我将讲述亲身经历后的感受，提一些建议。

不论是为客户做软件项目还是为自己做软件产品，都要进行需求分析。需求分析最恼人之处是难以在项目刚启动时搞清楚需求，如果在项目做了一大半时需求发生了变化，那将使项目陷入困境。4.3 节解释需求分析为什么困难，4.4 节讲述如何进行需求分析。本章的需求分析均不涉及编程，所以不考虑结构化、面向对象等分析方法。

4.1 可行性分析的要素

做可行性分析不能以偏盖全，也不可以什么鸡毛蒜皮的细节都加以权衡。可行性分析必须为决策提供有价值的证据。

联想集团领导人柳传志曾说：“没钱赚的事我们不干；有钱赚但投不起钱的事不干；有钱赚也投得起钱但没有可靠的人选，这样的事也不干。”柳传志为决策立了上述准则，同时也为可以行性分析指明了重点。

一般地，软件领域的可行性分析主要考虑四个要素：经济、技术、社会环境和人。本节只是泛泛地解释这四个要素，旨在建立全局分析的观念。4.2 节将结合案例围绕上述要素进行重点分析与评注。

4.1.1 经济

经济可行性分析主要包括：“成本——收益”分析和“短期——长远利益”分析。

一、成本——收益分析

成本——收益分析最容易理解，如果成本高于收益则表明亏损了，如果成本大大高于收益那就亏大了。商人都不喜欢做吃亏的事情。有些商店成天贴着“最后一天跳楼大拍卖”的标语，意思是：我准备吃大亏让你占便宜，同志，你快上钩吧。

如果是为客户做软件项目，那么收益就写在合同中。如果是做自己的软件产品，那么收益就是销售额。

人们在预估产品销售额时常常过分乐观而犯下大错。那些对你的产品说恭维话的人并不见得就是要买货的人，俗话说“嫌货才是买货人”。当你没碰到一个挑刺的人而感觉这产品好得会让你发大财时，就要做好会破产的心理准备。

如果做的是小本生意，那可得对成本进行细算。软件的成本不是指存放软件的那张光盘的成本，而是指开发成本。要考虑的成本有：

（1）办公室房租。

（2）办公用品，如桌、椅、书柜、照明电器、空调等。

（3）计算机、打印机、网络等硬件设备。

（4）电话、传真等通讯设备以及通讯费用。

（5）资料费。

（6）办公消耗，如水电费、打印复印费等。

（7）软件开发人员与行政人员的工资。

（8）购买系统软件的费用，如买操作系统、数据库、软件开发工具等。有些老板买盗版的系统软件，却按市场价算成本，可从美国佬那里赚一笔。

（9）做市场调查、可行性分析、需求分析的交际费用。

（10）公司人员培训费用。

（11）产品宣传费用。如果用 Internet 作宣传，则要考虑建设 Web 站点的费用。

（12）如果客户是政府部门，还要充分考虑用于吃喝玩乐、行贿的费用。

（13）如果公司的风水不好，会有很多莫名其妙的管理费。每戳一个红艳艳的公章都要化一把钞票。

二、短期——长远利益分析

人们喜欢吃着碗里的、看着锅里的，还想着别人家里的。短期利益和长远利益兼得是人们梦寐以求的事。在商业上，这等好事可不会轻易降临。

短期利益容易把握，风险较低。国内软件公司经常出现一窝蜂地去做信息管理系统、多媒体光盘、系统集成项目或 Internet 服务。每当我们沉迷于短期利益不思进取时，应该好好回忆童年时代那些伟大的抱负，给自己一些激励。

长远利益难以把握，风险较大。能为了长远利益不惜短期亏损的人，要么是雄心勃勃的将帅之才，要么是“纸上谈兵”、“眼高手底”的那一类庸人。国内目前有不少 Internet 企业，只投入不产出。为了成就将来的霸业，甘愿现在拼财力、比耐性。最后存活下来的几个公司将瓜分市场。

那些为长远利益奋斗的人们，你们可得把长征的路途走完啊，千万别让事业中途夭折。

4.1.2 技术

技术可行性分析至少要考虑以下几方面因素：

（1）在给定的时间内能否实现需求说明中的功能。如果在项目开发过程中遇到难以克服的技术问题，麻烦就大了。轻则拖延进度，重则断送项目。

（2）软件的质量如何？有些应用对实时性要求很高，如果软件运行慢如蜗牛，即便功能具备也毫无实用价值。有些高风险的应用对软件的正确性与精确性要求极高，如果软件出了差错而造成客户利益损失，那么软件开发方可要赔惨了。

（3）软件的生产率如何？如果生产率低下，能赚到的钱就少，并且会逐渐丧失竞争力。在统计软件总的开发时间时，不能漏掉用于维护的时间。软件维护是非常拖后腿的事，它能把前期拿到的利润慢慢地消耗光。如果软件的质量不好，将会导致维护的代价很高，企图通过偷工减料而提高生产率，是得不偿失的事。

技术可行性分析可以简单地表述为：做得了吗？做得好吗？做得快吗？

4.1.3 社会环境

社会环境的可行性至少包括两种因素：市场与政策。

市场又分为未成熟的市场、成熟的市场和将要消亡的市场。

涉足未成熟的市场要冒很大的风险，要尽可能准确地估计潜在的市场有多大？自己能占多少份额？多长时间能实现？

挤进成熟的市场，虽然风险不高，但油水也不多。如果供大于求，即软件开发公司多，项目少，那么在竞标时可能会出现恶性杀价的情形。国内第一批卖计算机的、做系统集成的公司发了财，别人眼红了也挤进来，这个行业的平均利润也就下降了。

将要消亡的市场就别进去了。尽管很多程序员怀念 DOS 时代编程的那种淋漓尽致，可现在没人要 DOS 应用软件了。学校教学尚可用用 DOS 软件，商业软件公司则不可再去开发 DOS 软件。

政策对软件公司的生存与发展影响非常大。整个 90 年代，中国电信的收费相当高，仅此一招就把国内互联网企业打得奄奄一息。某些软件行业的利润很高，但可能存在地方保护政策，使竞争不公平。政策不当将阻碍软件公司的健康发展，可最怕的还是政府干预企业的正当行为。例如：

现在家电行业竞争非常激烈，其中有一个著名企业的总裁十分了得，把对手打得节节败退。于是中央领导人就来视察该企业并作讲话："你们的业绩辉煌，得到了中央的高度重视，……但我们是社会主义国家，不是资本主义国家，你们总得给兄弟企业的同志们留口饭吃吧！"

有一次我拜访了北京大学一位研究经济学的朋友。这个年青人，还是个党员，竟然这么说："我最近在研究国内明星企业的兴衰问题，我发现了一个规律，明星企业一旦被政府领导人视察过，它就忘了自己是谁，就会做些走向死亡的蠢事。"

我实在不明白企业中为什么还要有"书记"职位。我以为"书记"乃是天下第一号可笑的官衔，"书记"本是"秘书"（secretary）的同义词，是个可有可无的行政人员的称呼，在中国竟然成了最大的官衔。每次看到新闻联播把国家主席错叫成总书记我都十分气愤：因为总书记的称喟只对几千万的党员适用，国家的新闻机构难道不面向十多亿

普通老百姓？如果我将来的工作单位还靠“书记”来管事，我每天准忙着生气，那里还有精力去编程。

4.1.4 人

有句名言：“人分四类——人物，人才，人手，人渣。”

如果一个软件公司里上述四类人齐全了，那么最好的分工是让“人物”当领导，“人才”做第一线的开发人员，“人手”做行政人员，“人渣”负责行贿。

这里只谈公司的领导与开发人员“行还是不行”。“人物”毕竟是少数，“人才”可是济济的。举重若轻的那类“人才”可以做领导，举轻若重的那类人才适合做软件开发人员。假如一群持有学士、硕士和博士文凭的毕业生到软件公司应聘，该如何录用呢？我的建议如下：

先选择本科毕业生，因为他们正当青春、干劲十足、不摆架子、不耻下问、要求不高、奉献甚多。

其次选择硕士毕业生，如果该生没象范进中举时那么老，并且在读硕士时没有天天去造文章而丢弃了编程工作，那么让有经验的学士程序员带他们煅练几个月就可以用了。

如果学士、硕士被其它公司取光了，那只好捡几个博士充数。博士到了软件公司有什么用呢？我想不出有什么用，只知道他们挺值得可怜的：从硕士读到博士出头，这六七年时间，真本事没学多少，倒学会“眼高手低”甚至“弄虚作假”；毕业时蓦然回首，发觉青春已被虚度，心灵已呈老态，唯有长叹短嘘，强把自负作自信。我也将博士毕业，就要论为三手贷贱卖了。真羡慕那些比我年轻的学士、硕士们，他们可以远走高飞，唉！

4.2 可行性分析案例——投资软件公司失败的教训

谈到软件产业，不能不提及比尔·盖茨与 Microsoft 公司。因为比尔·盖茨创建了 Microsoft 公司并成为世界首富的事实，使得无数从事软件工作的人们心存同样的梦想。有太多人急着想做中国的比尔盖茨 。有个年青人发明了一种汉字输入法，便在媒体上放言欲覆盖比尔·盖茨。中央电视台特冲动地把一个上了年纪的院士请来，让他谈谈自己与比尔·盖茨的比较，害得这位院士一个劲地辨解自己不是中国的比尔·盖茨。

近几年来，一批 Internet 英雄企业如 Yahoo、Netscape 兴起。尤如打破了秦始皇一统的天下，重返春秋战国时代。让软件人员走出了 Microsoft 的阴影，看到了阳光灿烂的软件世界。于是各色各样小不点儿的软件公司在国内遍地开花。

打破水缸的小孩子很多，但并不见得就会有司马光的业绩。由于“经济、技术、社会环境、人的因素存在差异，有些事情美国人能做成，我们模仿着做未必就能做得成功。虽然“星星之火、可以燎原”，但我们的国力薄弱，实在容不得把有限的火种扔到不毛之地。所以要进行可行性分析，如果不可行，就不要急着去做。本节三个案例是作者亲身经历的，我力求讲清楚错在哪里，并总结经验教训。希望读者看后能提高警惕，免犯相同的错误。

4.2.1 可行性分析案例之一

这个案例讲述我从开公司到关闭公司的一些经历和感受。

我从本科三年级开始编写图形程序，一见钟情后便如痴如醉，不管一切地抛弃了本科与硕士的微电子专业。1997 年春季，我到了向往已久的浙江大学 CAD&CG 国家重点实验室读博士学位。我幸福地幻想着大干一番自己喜爱的专业。开学的第一天，我兴冲冲地奔向实验室。进门不到 5 分钟，就因不懂规矩被看门的年青女子训了几次。为了不再冒犯规矩，我就老老实实地抓起一份计算机报纸并且站着阅读。突然一个气得脸色铁青的男人（机房管理员之一）对我断喝：“你在干什么！你怎么可以不经允许就翻看别人的报纸!”我就象一个情窦初开的少年飘飘然地去拥抱梦中情人，不料迎来两个耳光，此下场比《猫和老鼠》中的猫还惨。

不出几日，我就发现实验室里人们大多轻言寡语、小心翼翼、井水不犯河水。初到此实验室的北方同学极为迷惑地问我：“你们浙江人是不是都这个德性？我看你不太象嘛。”

如果不允许一个男人在工作时仰天大笑，这样的地方不去也罢。

我颇费周折地考入 CAD&CG 实验室，却尚未热身就全力而退，决心自立门户（至今我都没有用实验室的计算机编过一行程序）。

那时我穷困潦倒，只有一床，一盆，一壶，一碗。我那些穷朋友们象挤牙膏一样挤一些钱资肋我。我买了一台计算机，就在宿舍里开发软件。1997 年 8 月，我去北京参加首届中国大学生电脑大赛软件展示。路费也是借的，同学为我壮胆时说：“如果不能获奖，就回到实验室干活吧。”我说一定会拿第一名，这是必须实现的近期目标。

于是我拿了第一名，号称大学生“软件明星”。回到浙大后没啥反应，我就向杭州的几个报社发一份简讯。不久有一个记者来采访我，谈了一天，我发现记者还是不太明白，干脆自己写新闻报道，并且含蓄地做了一个广告：万事俱备，只待投资。10 月份我被评上浙江省青年英才（大学生代表），又获得中国大学生跨世纪发展基金特等奖。我到东软集团（沈阳）参加“民族软件产业青年论坛”，大不咧咧地作了一次演讲。由于我能说会道，频频上电视，引来近 10 个投资者。我选择了一位年龄比我大一倍、非常精明的商人作合伙人，成立了“杭州临境软件开发有限公司”。彼时，我可谓光芒四射，卓而不群，名片上印着“以振兴民族软件产业为己任，做真实、正直、优秀的科技人员。”浙江大学想开除我，被我“晓之以理、动之以情”安抚住。

我当时想开发一套名为 Soft3D 的图形系统，此系统下至开发工具，上至应用软件，无所不包。公司名字起为“临境”有两个含义：一是表示身临其境，这是我对图形技术的追求；二是表示快到了与 SGI 公司称兄道弟的境界，这是我对事业的追求。

我从实验室挖来一位聪明绝顶的师弟做技术伙伴（他的头明显比我的大，估计其脑容量至少是我 1.5 倍）。我曾经以师兄的身份为他洗过一双袜子，他因此觉得我是个好人。我俩一拍即合，常常为 Soft3D 的设计方案自我倾倒。一想到 Microsoft 公司的二维 Windows 系统即将被 Soft3D 打击得狼狈不堪时，我们就乐不可支，冲劲十足。到了 1998 年 7 月份，我们做了一套既不是科研又不全象商品的软件，我宣传了几个月都没有人要。1998 年 10 月份，我用光了 30 万元资金，只好关闭公司。不久我被一位朋友捉到北大方正去“劳改”，两个月后我才心服口服地承认自己失败了。

在失败一周年来临之际，我客观地从可行性分析角度说明了我和投资方所犯的错误，写此文以祭我那幼年夭折的软件公司。

一、我的主要错误

（1）年青气盛，在不具备条件的情况下，想一下子做成石破天惊的事。我的设计方案技术难度很大（有一些是热门的研究课题），只有 30 万元资金的小公司根本没有财力与技术力量去做这种事。分析经济与技术可行性，即可否定我的设计方案。

（2）我以技术为中心而没有以市场为中心去做产品，以为自己喜欢的软件别人也一定喜欢。我涉足的是在国内尚不成候的市场，我无法估计这市场有多大，人们到底要什么。伙伴们跟着我瞎忙乎一整年，结果做出一个洋洋洒洒没人要的软件。分析市场可行性也可否定我的设计方案。

（3）我做到了“真实、正直”，但并没有达到优秀的程度。我曾得到很多炫目的荣誉，但学生时代的荣誉只是一种鼓励，并不是对我才能和事业的确认。正因为我不够优秀，学识浅薄，加上没有更高水平的人指点我，才会把事情搞砸了。

二、投资方的错误

（1）投资方是个精明的商人，他把我的设计方案交给美国的一个软件公司分析，结论是否定的。但他觉得我这个人很有利用价值，希望可以做成功其它事情，即使 Soft3D 软件做不成功，只要挣到钱就行。这种赌博心态使得正确的可行性分析变得毫无价值。

（2）由于我不懂商业，又象所有单纯的学生那样容易相信别人。他让我写下了不公正的合同，我竟然向他借钱买下本来就属于我的 30%技术股份。他名为投资方，实质上双方各出了一半的资金（他出 51%，我出 49%）。他在明知 Soft3D 软件不能成功的情况下，却为了占我的便宜而丧失了应有的精明，最终导致双方都损失。

关闭公司时，他搬走了所有的固定财产。我明明投入了技术，又亏了 15 万元，却一无所得。几个月后当我意识到不公平而找他协商时，他说：“只能怨你自己愚蠢，读到博士，连张合同都看不懂。”

我觉得自己在此事上虽愚但不蠢，诚实正直的品质加上不懈的努力会让我变得有智慧。自己的奋斗没什么可以后悔的，学到的远比失去的多，下一次会做得更好。

4.2.2 可行性分析案例之二

1999 年 3 月，一位与我同年同月同日同时辰出生的朋友请我帮忙，对一份长达 8 页的投资方案：

《万向为什么不投资互联？》

——“中国供应商信息网”引资方案

（以下简称“引资方案”）进行可行性分析。

万向集团是浙江省民营企业的老大，有的是钱，找它投资真的是找对了。我当时忙着对自己进行查漏补缺，正在复习本科的计算机基础课程。不耐烦地看完那份“引资方案”，觉得比我去年写的 Soft3D 设计方案荒唐十倍以上，于是毫不迟疑地否决。

朋友问：“为什么？”

我答曰：“因为内行的人都会否决。”

朋友嘲笑：“这是废话，如果投资者都是内行的话，就不用请人分析了。你要写出让农民企业家看得懂的可行性分析报告，才叫有水平。”

我说：“好，好，就看在同年同月同日同时辰的份上，帮你一把。”

于是我就为拥有几十亿资产的一群农民伯伯写了一份 6 页纸的可行性分析报告。此

“引资方案”是个很典型的不可行方案，我将摘录一部分内容作为案例进行分析。我不认识写“引资方案”的人，虽然批评很多，但是我对他（她）本人毫无恶意。由于我的可行性分析报告是写给不具备计算机基础知识的人看的，文中采用较为形象的日常事例来比喻、解释信息产业的一些现象。这些比喻在一定程度上是合理的，但可能是不严谨的。

该“引资方案”有四段主要内容，每段的主题思想简述如下。

第一段，介绍了 Internet 企业的红火，得出一些结论，这是“建议投资互联网”的关键论据之一。主要文字如下：

“国际互联网英雄企业雅虎（Yahoo）、网景（Netscape）……这些互联网企业有一个特点，就是赢利都还不多甚至微利运转。它们的年收入尚以万来计算，而股市却达到几十亿乃至百亿。……几乎所有互联网企业只要一上市，它的股价就可以青云直上，无人能用任何股市定理来推算它们将涨到什么程度。”

第二段，概括了在国内投资 Internet 企业的几大好处，这是建议万向集团投资 Internet 企业的关键论据。主要文字如下：

“……借互联网这个金蛋与广大投资者的看好，股票必能大幅上升。借美国网股的升幅样板，广大中国股民可能在短期内即可推动股票实现翻番。……”

“……相信不久的将来，互联网企业的黄金赢利期即可到来。事实上，由于进入互联网经营网站的直接成本很低，目前国内的绝大部分网站皆为小打小闹，几乎没有几个网站有雄厚的资金注入建设。……”

“……通过建设一个大网站所带来的全国乃至全世界的广告影响是无可估量的。…很容易造势，效果远远优于同等金额的电视、报纸广告。……”

第三段，该方案的作者说自己已经建立了一个公司，经营一个叫“中国供应商信息网”的站点。目前他有一些构思，希望万向集团投资 200 万元/年，按他的方案，可以很快使万向集团股票狂升，机会千载难逢。主要文字如下：

“我们经营的网站有 2 年多历史了。…年经营成本 9 万元，我只找了一个助手。…全年收入近 10 万元，虽说不亏，但我无法满足现状。…希望吸取资金 200 万。…预计年收入 1200 万至 2000 万。”

“……增加网站每日求购信息的整编工作，做到任何其它一个网站有的我们都有。”

“……增加中国及世界经济新闻版块，由每周更新到每日更新。”

“……建立网上信用卡收帐系统，一旦需要，可立即投入使用。”

第四段，他说自己是萧山人，万向集团也源于萧山。

一、对“引资方案”第一段的分析与评论

Internet 英雄企业是典型的知识经济产物。雅虎、网景公司的营业利润与其股市相比是极微的。从外表上看：一个固定资产可以被忽略的，几乎没有营业利润的 Internet 企业，可以有极高的股市价值，并且市值不停上涨。为什么能够这样？这是有极苛刻的前提条件的。但是“引资方案”一开始就立论：

几乎所有互联网企业只要一上市，它的股价就可以青云直上，无人能用任何股市定理来推算它们将涨到什么程度。

这显然是胡吹，我认为即使外行也不会相信。尽管我也不够资历去评论雅虎、网景

公司，但愿意尽力不离谱地通俗地解释雅虎、网景现象。

让我们把 Internet 想象成无边无际的网，把每个网的交织点叫做网站，在每个网站中至少可以放一台计算机。Internet 的基本功能是让网上的所有计算机能够相互通讯。由于现在的一台计算机能做十年前不敢想的事，真的无法估量上亿台计算机相联的 Internet 有多么强大的本领。Internet 极其相关产业成了美国的支柱产业。

为了在一台计算机上可以看到其它计算机的信息，必须安装一种叫浏览器的软件。浏览器的重要性如同发动机对于车辆。一个美国年轻人最先发明了浏览器。同时，计算机工业界的一位伟大人物，SGI 公司（世界最大计算机公司之一）的创始人远见卓识，投资数千万美元开发这种刚诞生的浏览器。他们成立了网景公司，一位是站在前面的年轻英雄，一位是站在后面的老英雄。网景公司刚上市的时候，浏览器还没有销售（更谈不上利润），但是浏览器实在太有价值了（能让微软公司都害怕被击跨），所以网景公司上市第一天就升值十几亿美元。而网景公司销售产品得到的利润只是用来运转公司的，远比不上其该有的价值（股市价值）。可并不是随便一个 Internet 企业都象网景公司那么辉煌。

举杭州西湖的例子吧。西湖的一些景点收门票，卖纪念品，一年不多的经营收入，只够用来打扫卫生、修修补补、种树植草。但西湖有白居易、苏东坡巨大的贡献，千年累积的“人文、历史、地理”价值何止千百亿。假设杭州西湖上市，起价 1 亿元人民币，真不知道有多少人来争抢，股值当然会狂升千百倍。而我家乡的一个鱼塘也叫“西湖”，主人从来不幻想他的“西湖”值多少个亿，但他知道卖不出鱼就会贫穷，一天不争几元钱就会挨饿。

接下去说雅虎公司吧。

由于有了浏览器，用 Internet 交流信息就十分方便，推进了 Internet 的发展。还有另一个问题，用户在用互联网之前，总得先知道要找的信息在什么地方（打电话也要先知道对方的号码），不可能对上亿台计算机挨个浏览吧。这时就需要一种叫做搜索引擎的软件系统，它专门用于存放与检索互联网信息。假如一个用户想查找一种叫“万向节”的汽车零件，只需要输入“万向节”三个字，搜索引擎就会在很快查找出全球有关“万向节”的网站地址。搜索引擎的重要性，就象一个封建皇朝的藏宝图与挖掘机。美国的几个年轻人（头头是个华裔）发明了搜索引擎，他们成立了雅虎公司，该公司得到日本软件巨头 SoftBank 公司的投资。也是由于雅虎公司的搜索引擎实在太有价值了，所以股值很快达到其投资的百倍。现在去做搜索引擎想发大财已经太晚了。人家都用导弹打飞机了，我们还做梦用弓箭射飞机。

雅虎、网景现象小结：只有在具备极大的市场需求、极优秀的知识产品、业界杰出人物的条件下，才可能使一个公司的股值远远高于其“原始投资 + 经营利润”。如果去投资一个没有价值、不产生利润的公司，那只会拖跨投资方，这是投资常识。

二、对“引资方案”第二段的分析与评论

（1）……借互联网这个金蛋与广大投资者的看好，股票必能大幅上升。借美国网股的升幅样板，广大中国股民可能在短期内即可推动股票实现翻番。

评论：用童话故事“皇帝的新装”来比喻很恰当：他是那个裁缝，投资方是那个笨蛋皇帝，股民就相当于傻乎乎的臣民。如果有那么容易的事情，身为经济学家的总理早

就做了，中国的信息产业也不会太落后。

（2）.....相信不久的将来，互联网企业的黄金赢利期即可到来。事实上，由于进入互联网经营网站的直接成本很低，目前国内的绝大部分网站皆为小打小闹，几乎没有几个网站有雄厚的资金注入建设。...

评论：他说进入互联网经营网站的直接成本很低 ，那简直是黑白颠倒，哪有成本低才小打小闹的！表明他对中国 ISP（Internet 服务供应商）发展史全无知。在整个 90 年代，有成百上千的 ISP 死掉，主要原因是经营成本（付给电信局的钱）太高而用户太少。为什么还有一些 ISP 在小打小闹？那是因为该死掉的全死啦，剩一口气的与刚出生的正在一块儿垂死挣扎。现在还能在互联网上指手划脚的都是与中国电信部门有密切关系的 ISP。为什么几乎没有几个网站有雄厚的资金注入建设 ？因为怕啦。

请看看近几年的“中国计算机报”、“计算机世界”，中国 ISP 发展史简直就是软件业界勇士与中国电信抗争的血泪史。国内最了不起的 ISP 是北京“赢海威时空”，在中国教育科研网建立之前，它就建立了中国人自己的“小型 Internet”，连《读者》也报道过那些感人事迹。投资超过千万（投资方是银行）。但由于亏损太大，在 1998 年，“赢海威时空”总裁张树新女士，一位富有才华的 ISP 创业者，被董事会强行解职，原有创业者跟着全被辞退。此事震动软件业界。还有谁的钱比银行多？有谁比中国电信更能控制互联网？

（3）通过建设一个大网站所带来的全国乃至全世界的广告影响是无可估量的。...很容易造势，效果远远优于同等金额的电视、报纸广告。

评论：这是梦话。有一篇评论说中国老百姓通过 Internet 获取信息所付出的代价是美国人的 1000 倍以上。看一个简单的算术就能明白：中国老百姓的工资不及美国人的十分之一，中国网站的有用信息远不及美国的十分之一，中国 Internet 传输速率不及美国的十分之一，电话费用又比美国的高。仅此几项乘积，代价已是美国人的 1000 倍以上。

这个数字说明，很多事情在美国能成功，在中国就难以成功。在 Internet 这个问题上，存在 1000 倍的难度，足以扼杀普通公司。

我和同事都是软件专业的博士生，Internet 对我们而言就象眼睛与耳朵一样重要。可是连我们也没有条件使用 Internet，成了专业的瞎子与聋子，普通老百姓更不用说了。如果没有群众用户，ISP 广告给谁看？过不了一年，ISP 就得饿死，怎么去产生全球影响？怎么去迎接不久将来的滚滚财源？

三、对“引资方案”第三段的分析与评论

（1）我们经营的网站有 2 年多历史了。...年经营成本 9 万元，我只找了一个助手。...全年收入近 10 万元，虽说不亏，但我无法满足现状。...希望吸取资金 200 万。...预计年收入 1200 万至 2000 万。

评论：可以看出，他的经营能力为 10 万/年，是个单干户。怎么能一下子就可以经营“年投资 200 万，并且预计年收入达千万的企业？”刚学会走路，就夸口能跑得跟子弹一样快，神仙？

（2）增加网站每日求购信息的整编工作，做到任何其它一个网站有的我们都有。

评论：这句话雅虎公司都不敢夸口。

（3）增加中国及世界经济新闻版块，由每周更新到每日更新。

评论：他能做到的，中央电视台、新华社的网站应该早做到了。

（4）建立网上信用卡收帐系统，一旦需要，可立即投入使用。

评论：这种事情，必须是公安部门、税务部门、银行、系统集成商等一起商量好了后才可以做的。

四、对“引资方案”第四段的分析与评论

他说自己是萧山人，万向集团也源于萧山，因此他首先选择万向集团。

评论：他建议万向集团投资互联网，可不是为了万向集团，也不是为了萧山的老百姓，这一点连我们做学生的都明白。商业投资不应该带有太多的感情色彩，即便是万向集团的人提出的方案，如果不可行，也要否决。

根据以上的分析，我全盘否定了该“投资方案”，并建议万向集团不要投资 ISP。可惜当初没人否定我开公司时的方案。

4.2.2 可行性分析案例之三

看了第一个案例，你可能会惋惜。看了第二个案例，你可能觉得好笑。如果看了第三个案例，你肯定会生气。

1999 年 1 月，有一个民营企业家 G 先生向我请教一个问题：“我给一个年轻人投资了 100 万元，建立一家从事环保信息应用开发的软件公司。他曾许诺一年内创利润上千万元，可是才过去 5 个月，他就把 100 万元用完了，什么也没挣到。我实在不明白是怎么回事，请你帮我分析分析。”

这位 G 先生年龄有我的 2.5 倍，曾在西北当过几十年的技术兵，性格豪爽。他投资的那个年轻人叫 Y（以下称 Y 经理），自称有英国的管理学文凭，能对公司的市场、技术、管理一把抓。G 先生喜欢说“钱我没问题”，于是想也不想就投了 100 万元，并且给 Y 经理 40%的股份。

G 先生请 Y 经理到家里坐谈。我那时突然狡猾起来，自称是 G 先生的远房亲戚，在浙大读半导体物理，特羡慕那些做软件的同龄人，渴望听听 Y 经理的高见。Y 经理果然信口开河，滔滔不绝，连绵不断，如黄河泛滥，一发而不可收拾。我激动地想去参观他的公司和产品，并表示要抛弃物理专业，立马转向软件专业。Y 经理得意而笑：“对于 IT 行业你就不懂了，我们经营的是一种理念而不是产品，这是国外最先进的思想。你可以来参观我的公司，但你看不到具体的东西，只能用心去领会。”这屁话比曹元朗（《围城》）的诗还臭。我搞软件只有 8 年功夫，说我不懂 IT 行业并不过分。可我读了 10 年大学都没听到过如此“先进”的思想。如果这是英国管理学教育的成果，我认为自己已经发现了这个曾经是“日不落帝国”的衰败的真真原因，有必要找英国首相切磋一番。

我对 G 先生说：“Y 经理根本不懂技术，为人极其浮夸。应马上关闭公司，以绝后患。那 100 万元你也亏得起，就买个教训吧。”

G 先生说：“钱我没问题，那 100 万元就算我在澳门赌博输掉了。”

1999 年 5 月，G 先生又来找我请教另一个问题。

他说：“小林啊，你上次说得很有道理，我接受了教训。”

我说：“那是好事，不论年龄大小，知错就改总是好孩子嘛。”

他叹了一口气：“最近几个月，Y 经理又花了我 100 万元。”

我当时差点给噎死，气凶凶地训 G 先生：“我早跟您讲过，Y 经理不是好东西，叫你关闭公司你不听，你老说钱没问题，亏你 200 万元活该。”

老先生象犯了错误的小孩子：“Y 经理每一次向我要钱时，都拍拍胸脯保证下个月就有利润，所以我一而再、再而三地掏钱给他，希望能救活软件公司。现在该怎么办？”

一个有 20 名职员的软件公司，程序员只有三四个，连“十羊九牧”都不如。200 万元的财务报表中，有 100 多万元用于吃喝玩乐和行贿。这种公司完全无药可救。台湾的李敖曾说过：“当你没法扶一个人上马时，也许应该拉他下马”。从 5 月份到 8 月份，我行侠仗义，替 G 先生清理软件公司，根除 Y 经理这些败类。

可是难哪，因为 G 先生投资的公司根本不把 G 先生放在眼里，又岂能让我插手。就在我想方设法卡住 Y 经理的脖子时，Y 经理总能从 G 先生那里挖出钱。G 先生就象被吸血鬼附身，却仍存幻想：“如果吸血鬼能治好我的病，就让它再吸些血吧。”

Y 经理又和一个来自深圳的骗子 H 想了注意，教唆 G 先生再投资 100 万元新建一个“指纹”公司，说利润将比环保信息的应用开发更加可观（估计要用亿来度量）。就在他们准备鉴合同之际，我偶而路过，发现异常，便强行阻止。

G 先生是个好人，但太顽固。好几次我气极想撒手不管，但又不忍心好人被坏人欺负。我曾请求 G 先生：“我求您别再说钱没有问题，您的私人财产会被人骗光。请让我把这漏洞堵住吧，好让我安心地回学校做完博士学位论文。”

到 8 月份，我和 G 先生的两个儿子，伙同黑社会的朋友，强行把那个软件公司搬回 G 先生的工厂中，辞退所有员工。现在那个软件公司被别人接管，仍然半死不活，好在每月亏损不过几万元，G 先生承受得起，我就不再去碰 G 先生的伤疤。

我以前从未玩过与人勾心斗角的游戏，此三个月的经历让我疲惫不堪。那个软件公司的员工曾透露，Y 经理的英国文凭大约是在上海或杭州某个大专培训班里混来的，《围城》中的方鸿渐买美国克莱顿大学博士文凭尚知羞耻，而 Y 经理却趾高气扬。害得我平白无故为英国教育界担心，回想起英国鬼子曾打劫过中国，倍感耻辱。

G 先生是正人君子，不防小人，实在不是现代的商人。我和他成了忘年交。G 先生第一次见到我时问我工资几何，我答曰：“300 元，够买几本书。”G 先生甚为着急：“这样的条件怎么能生活？你就搬到我家来住吧，我家条件好，你可以安心地学习，将来可为国家多作贡献。”后来他几次相邀，我就看在国家的份上住入他家。自从读中学以来，我第一次享受食来张嘴，不用洗衣的奢侈。唯一的麻烦是我得向很多朋友解释：“我不是被别人养起来了，是为了国家的利益，不得已才这么做的。G 先生是男的不是女的，并且没有待出嫁的女儿。”

4.3 需求分析为什么困难

有几种原因使需求分析变得困难：（1）客户说不清楚需求；（2）需求自身经常变动；（3）分析人员或客户理解有误。

4.3.1 客户说不清楚需求

有些客户对需求只有朦胧的感觉，当然说不清楚具体的需求。例如全国各地的很多政府机构在搞网络建设，这些单位的领导和办公人员大多不清楚计算机网络有什么用，反而要软件系统分析人员替他们设想需求。这类工程的需求是如此的主观，以致产生很多贪污腐败现象。

有些客户心里非常清楚想要什么，但却说不明白。读者可能很不以为然。就举日常生活的事例吧，比如说买鞋子。我们非常了解自己的脚，但没法说清楚脚的大小和形状。只能拿鞋子去试，试穿时感觉到舒服才会买鞋（居然也有神通广大的售货员，看一眼客户的手，就知道应该穿什么样的鞋）。

如果客户本身就懂软件开发，能把需求说得清清楚楚，这样的需求分析将会非常轻松、愉快。如果客户全不懂软件，但信任软件开发方，这事也好办。分析人员可以引导客户，先阐述常规的需求，再由客户否定不需要的，最终确定客户真正的需求。最怕的就是“不懂装懂”或者“半懂充内行”的客户，他们会提出不切实际的需求。如果这些客户甚至觉得自己是上帝的爸爸，那么沟通和协商都会很困难。

4.3.2 需求自身经常变动

唐僧曾说：“妖要是有了仁慈之心，就不再是妖，是人妖。”（《大话西游之大圣娶亲》）

连妖都会变心，别说人了。所以喜新厌旧乃人之常情，世界也因此变得多姿多彩。

软件的需求会变化吗？

答：据历史记载，没有一个软件的需求改动少于三次。唯一只改动需求两次的客户是个死人。这个可怜的家伙还是在运送第三次需求的路上被车子撞死的。[Cline 1995]

让我们先接受“需求会变动”这个事实吧，免得在需求变动时惊慌失措。明白“需求会变动”这个道理后，在进行需求分析时就要留点神：

（1）尽可能地分析清楚哪些是稳定的需求，哪些是易变的需求。以便在进行系统设计时，将软件的核心建筑在稳定的需求上，否则将会吃尽苦头。

（2）在合同中一定要说清楚“做什么”和“不做什么”。如果合同含含糊糊，日后扯皮的事情就多。要防止象韩复渠那样，在别人请他喝酒吃饭时他什么都点头（人家就更加献殷勤），吃完了他就宣布刚才答应的事都不算数，便扬长而去。

4.3.3 分析人员或客户理解有误

有个外星人间谍潜伏到地球刺探情报，它给上司写了一份报告：“主宰地球的是车。它们喝汽油，靠四个轮子滚动前进。嗓门极大，在夜里双眼能射出强光。……有趣的是，车里住着一种叫作‘人’的寄生虫，这些寄生虫完全控制了车。”

软件系统分析人员不可能都是全才。客户表达的需求，不同的分析人员可能有不同的理解。如果分析人员理解错了，可能会导致开发人员白干活，吃力不讨好。我读中学时候最怕写作文逃题，如果逃题了，不管作文写得多长，总是零分。所以分析人员写好需求说明书后，要请客户方的各个代表验证。如果问题很复杂，双方都不太明白，就有必要请开发人员快速构造软件的原型，双方再次论证需求说明书是否正确。

由于客户大多不懂软件，他们可能觉得软件是万能的，会提出一些无法实现的需求。有时客户还会把软件系统分析人员的建议或答复给想歪了。

有一个软件人员滔滔不绝地向客户讲解在“信息高速公路上做广告”的种种好处，

客户听得津津有味。最后，心动的客户对软件人员说：“好得很，就让我们马上行动起来吧。请您决定广告牌的尺寸和放在哪条高速公路上，我立即派人去做。”

为什么软件系统分析员的工资要比普通程序员高？就是因为需求分析困难嘛。

4.4 如何进行需求分析

上一节诉说了需求分析的困难，本节要知难而进。

进行需求分析不象情人之间的浪漫做法——“让我摸摸你的头发，感觉它是什么颜色。”我们要围绕两个核心问题开展需求分析：（1）应该了解什么？（2）通过什么方式去了解？

4.4.1 应该了解什么

那怕是天下最无能的市长或书记，都知道在作报告时要先从宏观上讲一、二、三、四、五，再从细节上讲 A、B、C、D、E。需求分析不象侦探推理那样从蛛丝马迹着手。应该先了解宏观的问题，再了解细节的问题，如图 4.1 所示。

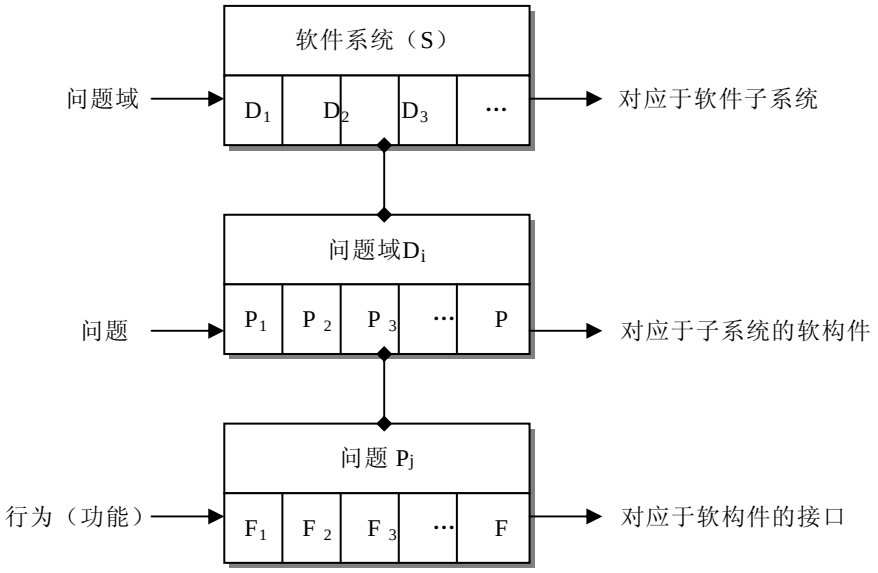


图 4.1 进行需求分析时要了解的内容

一个软件系统（记为 S）的涉及面可能很广，可以按不同的问题域（记为 D）分类，每个问题域对应于一个软件子系统。

$$S = \{ D_1, D_2, D_3, \cdots D_n \}$$

问题域D_i 由若干个问题（记为P）组成，每个问题对应于子系统中的一个软构件。

$$D_i = \{ P_1, P_2, P_3, \cdots P_m \}$$

问题P_j有若干个行为（或功能，记为F），每个行为对应于软构件中的接口。

$$P_j = \{ F_1, F_2, F_3, \cdots F_k \}$$

按图 4.1 结构写成的需求说明书，对于那些只想了解宏观需求的领导，和需要了解

细节的技术员都合适。在写需求说明书时还应该注意两个问题：

（1）最好为每个需求注释“为什么”，这样可让程序员了解需求的本质，以便选用最合适的技术来实现此需求。

（2）需求说明不可有二义性，更不能前后相矛盾。如果有二义性或前后相矛盾，则要重新分析此需求。

4.4.2 通过什么方式去了解

了解需求的方式有好几种：

（1）直接与客户交谈。如果分析人员生有足球评论员的那张“大嘴”，就非常容易侃出需求。

（2）有些需求客户讲不清楚，分析人员又猜不透，这时就要请教行家。有些高手真的很厉害，你还没有开始问，他就能讲出前因后果。让你感到“听君一席言，胜读十年书。”

（3）有很多需求可能客户与分析人员想都没有想过，或者想得太幼稚。要经常分析优秀的和蹩脚的同类软件，看到了优点就尽量吸取，看到了缺点就引以为戒。前人既然付了学费，后人就不要拒绝坐享其成。

4.5 小 结

为了阐述可行性分析的四个要素：经济、技术、社会环境和人，本章讲了几个令人垂头丧气的案例。如果您学会了客观、科学的可行性分析，在作决策时就要果断，要学习热恋中的这个年青人——“倒底行还是不行？行就结婚，不行就离婚。”

本章并没有鼓吹需求分析的难度，不是在吓唬人。如果需求分析搞错了，麻烦大哩。几十年前，我们最最伟大的领袖毛主席说了一声“人多力量大”，导致现在中国人口蹦到13亿。他老人家辉煌地走了，后人却付出了沉重的代价。

所以我们要认真地做好可行性分析和需求分析。

第五章　系 统 设 计

系统设计是把需求转化为软件系统的最重要的环节。系统设计的优劣在根本上决定了软件系统的质量。就象“一切帝国主义都是纸老虎”那样可以断定“差的系统设计必定产生差的软件系统。”所以我们要努力保证系统设计“根正苗红”，把一切左倾、右倾的设计思潮消灭在萌芽状态。

Windows NT 的一位系统设计师拥有 8 辆法拉利跑车，让 Microsoft 公司的一些程序员十分眼红。但你只能羡慕而不能愤恨，因为并不是每个程序员都有本事成为复杂软件系统的设计师。系统设计要比纯粹的编程困难得多。即便你清楚客户的需求，却未必知道应该设计什么样的软件系统——既能挣最多的钱又能让客户满意。“天下西湖三十六，最美是杭州”，千年前苏东坡大学士对西湖精采绝伦的系统设计，使杭州荣升为“天堂”，让后人只剩下赞叹和破坏的份了。

本章讲述系统设计的四方面内容：体系结构设计、模块设计、数据结构与算法设计、用户界面设计。如果将软件系统比喻为人体，那么：

（1）体系结构就如同人的骨架。如果某个家伙的骨架是猴子，那么无论怎样喂养和美容，这家伙始终都是猴子，不会成为人。

（2）模块就如同人的器官，具有特定的功能。人体中最出色的模块设计之一是手，手只有几种动作，却能做无限多的事情。人体中最糟糕的模块设计之一是嘴巴，嘴巴将最有价值但毫无相干的几种功能如吃饭、说话、亲吻混为一体，使之无法并行处理，真乃人类之不幸。

（3）数据结构与算法就如同人的血脉和神经，它让器官具有生命并能发挥功能。数据结构与算法分布在体系结构和模块中，它将协调系统的各个功能。人的耳朵和嘴巴虽然是相对独立的器官，但如果耳朵失聪了，嘴巴就只能发出“啊”“呜”的声音，等于丧失了说话的功能（所以聋子天生就是哑巴），可人们却又能用手势代替说话。人体的数据结构与算法设计真是十分神奇并且十分可笑。

（4）用户界面就如同人的外表，最容易让人一见钟情或一见恶心。象人类追求心灵美和外表美那样，软件系统也追求（内在的）功能强大和（外表的）界面友好。但随着生活节奏的加快，人们已少有兴趣去品味深藏不露的内在美。如果把 Unix 系统比作是健壮的汉子和妇人，那么 Windows 系统就象妩媚的小白脸和狐狸精。想不到 Windows 系统竟然能兴风作浪，占去大半市场。有鉴于此，我们应该鼓励女士多买化妆品（男士付钱）以获得更好的界面。

在进行系统设计时，我们要深情地关注软件的质量因素，如正确性与精确性、性能与效率、易用性、可理解性与简法性、可复用性与可扩充性等等。即使把系统设计做好了，也并不意味着就能产生好的软件系统。在程序设计、测试、维护等环节还要做大量的工作，无论哪个环节出了差错，都会把好事搞砸了。据说上帝把所有的女士都设计成天使，可是天使们在下凡时有些双脚先着地，有些脸先着地。上帝的这一疏忽让很多女孩伤透了心。我们在开发软件时，一定要吸取这个教训。

5.1 体系结构设计

杨叔子院子曾这样指点其弟子：

文学中有科学，音乐中有数学，漫画中有现代数学的拓扑学。漫画家可以“几笔”就把一个人画出来，不管怎么美化或丑化，就是活像。为什么？因为那“几笔”不是别的，而是拓扑学中的特征不变量，这是事物最本质的东西。

体系结构是软件系统中最本质的东西：

（1）体系结构是对复杂事物的一种抽象。良好的体系结构是普遍适用的，它可以高效地处理多种多样的个体需求。一提起“房子”，我们的脑中马上就会出现房子的印象（而不是地洞的印象）。“房子”是人们对住宿或办公环境的一种抽象。不论是办公楼还是民房，同一类建筑物（甚至不同类的建筑物）之间都具有非常相似的体系结构和构造方式。如果 13 亿中国人民每个人都要用特别的方式构造奇异的房子，那么 960 万平方公里的土地将会变得千疮百孔，终日不得安宁。

（2）体系结构在一定的时间内保持稳定。只有在稳定的环境下，人们才能干点事情，社会才能发展。科学告诉我们，宇宙间万物无时无刻不在运动、飞行。由于我们的生活环境在地球上保持相对稳定，以致于我们可以无忧无虑地吃饭和睡觉，压根就意识不到自己是活生生的导弹。软件开发最怕的就是需求变化，但“需求会发生变化”是个无法逃避的现实。人们希望在需求发生变化时，最好只对软件做些皮皮毛毛的修改，可千万别改动软件的体系结构。就如人们对住宿的需求也会变动，你可以经常改变房间的装璜和摆设，但不会在每次变动时都要去折墙、拆柱、挖地基。如果当需求发生变化时，程序员不得不去修改软件的体系结构，那么这个软件的系统设计是失败的。

良好的体系结构意味着普适、高效和稳定。本节将论述两种非常通用的软件体系结构：层次结构和客户机/服务器（Client/Server）结构。

5.1.1 层次结构

层次结构表达了这么一种常识：有些事情比较复杂，我们没法一口气干完，就把事情分为好几层，一层一层地去做。高层的工作总是建立在低层的工作之上。层次关系主要有两种：上下级关系和顺序相邻关系。

一、上下级关系的层次结构

我们从小学一直读到博士研究生毕业，要读 20 多年，可以分为五个层次。而范进的知识结构只有两层：“私塾”和“秀才”，但读了五十多年，如图 5.1 所示。一般地处于较高层次的学生应该懂得所有低层次的知识，而处于低层次学生无法懂得所有高层次的知识。图 5.1 的层次结构存在上下级关系，如同在军队中，上级可以命令下级，而下级不能命令上级。如果把图 5.1 的层次结构当成是一个软件系统的结构，那么上层子系统可以使用下层子系统的功能，而下层子系统不能够使用上层子系统的功能。

二、顺序相邻关系的层次结构

顺序相邻关系的层次结构表明通讯只能在相邻两层之间发生，信息只能被一层一层地顺序传递。这种层次结构的经典之作是计算机网络的 OSI 参考模型，如图 5.2 所示。为了减少设计的复杂性，大多数网络都按层（Layer）或级（Level）的方式组织。每一

层的目的都是向它的上一层提供一定的服务，而把如何实现这一服务的细节对上一层加以屏蔽。一台机器上的第 n 层与另一台机器上的第 n 层进行对话。通话的规则就是第 n 层的协议。数据不是从一台机器的第 n 层直接传送到另一台机器的第 n 层。发送方把数据和控制信息逐层向下传递。最低层是物理介质，它进行实际的通讯。接收方则将数据和控制信息逐层向上传递。

每一对相邻层之间都有接口。接口定义了下层提供的原语操作和服务。当网络设计者在决定一个网络应包含多少层，每一层应当做什么的时候，其中很重要的工作是在相邻层之间定义清晰的接口。接口可以使得同一层能轻易地用某一种实现（Implementation）来替换另一种完全不同的实现（如用卫星信道来代替所有的电话线），只要新的实现能向上层提供同一组服务就可以了。[Tanenbaum 1998]

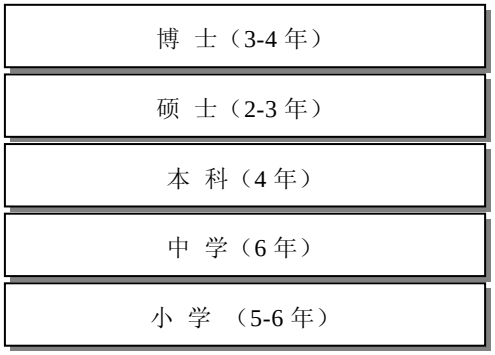


图 5.1（a）从小学读到博士存在的五个学习阶段

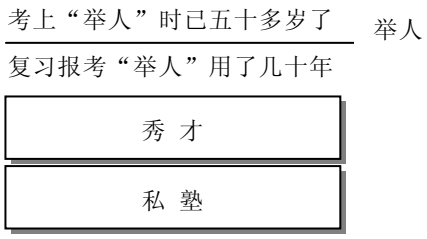


图 5.1（b）范进的知识结构

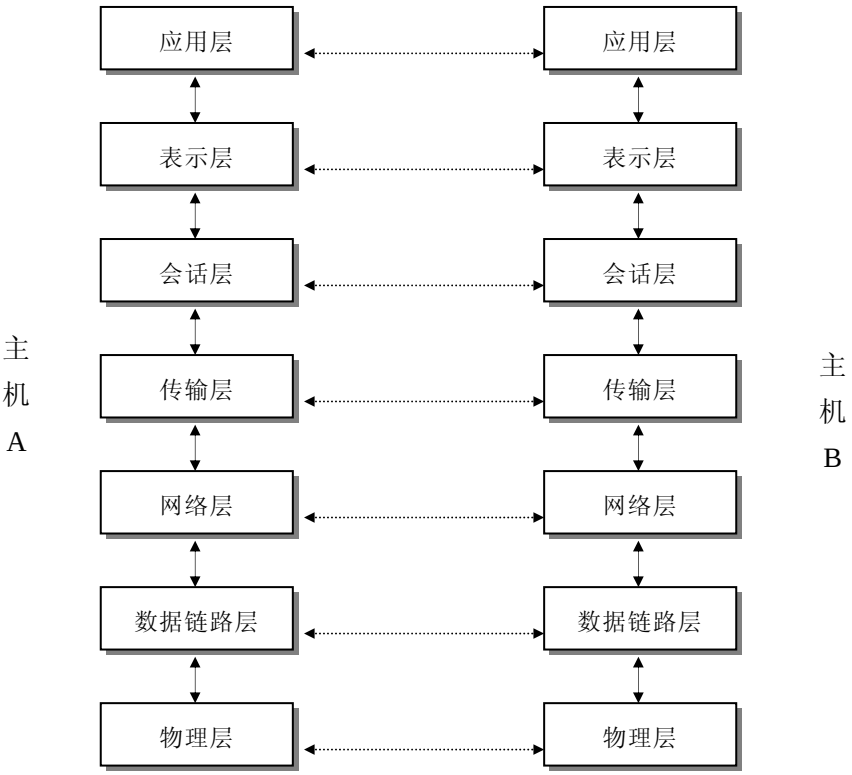


图 5.2　计算机网络的 OSI 参考模型

三、其它的层次结构

目前在大型商业应用软件系统中还流行一种包含中间件（Middleware）的层次结构，如图 5.3 所示[Jacobson 1997]。中间件支持与平台无关的分布式计算，可以用 DCOM 和 CORBA 对象来实现。

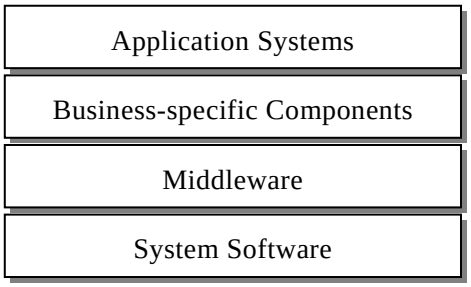
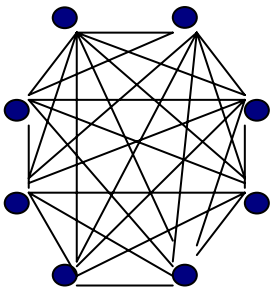


图 5.3　包含中间件的层次结构

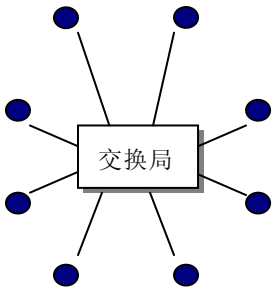
5.1.2　客户机/服务器结构

让我们先回顾一下早期的电话系统。贝尔（Alexander Graham Bell）于 1876 年申请了电话专利。那时期的电话必须一对一对地卖，用户自己在两个电话之间拉一根线。如果一个电话用户想和其它几个电话用户通话，他必须拉 n 根单独的线到每个人的房子里。于是在很短的时间内，城市里到处都是穿过房屋和树木的混乱的电话线。很明显，企图把所有的电话完全互联（如图 5.4（a）所示）是行不通的。

贝尔电话公司在 1878 年开办了第一个交换局。公司为每个客户架设一条线。打电话时，客户摇动电话的曲柄使电话公司办公室的铃响起来，操作员听到铃声以后根据要求将呼叫方和被呼叫方用跳线手工连接起来。这种集中交换式的模型如图 5.4（b）所示。很快地，贝尔系统的交换局就出现在各地。人们又要求能打城市间的长途电话，就出现了二级交换局，以后进一步发展为多个二级交换局。[Tanenbaum 1998]



5.4（a）完全互联的电话系统



5.4（b）集中交换式的电话系统

如果将图 5.4（b）中的电话看成是客户程序，将中心的交换局看成是服务程序，那么图 5.4（b）就是典型的客户机/服务器结构。注意这里客户机和服务器都是指软件而不是指硬件（一台计算机可以放多个客户机和服务器软件）。

客户机/服务器结构存在两个显然的优点：

（1）以集中的方式高效率地管理通讯。前面讲电话系统的故事就是要说明这一点。
（2）可以共享资源。比如在信息管理系统中，服务器将信息集中起来，任何客户机都可以通过访问服务器而获得所需的信息。

客户机和服务器之间的通讯以“请求——响应”的方式进行。客户机先向服务器发起“请求”（Request），服务器再响应（Response）这个请求，如图 5.5 所示。

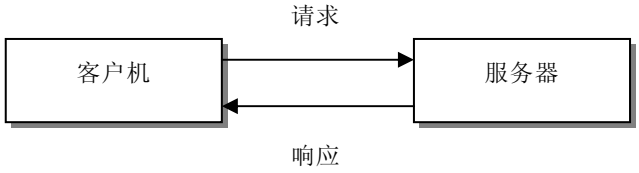


图 5.5　Client 和 Server 之间的通讯以“请求——响应”的方式进行

采用“请求——响应”这种通讯方式的基本动机是为了解决“聚集”（Rendezvous）问题。为了理解这一个问题，设想一个人试图在分离的机器上启动两个程序并让它们进行通讯。还需记住，计算机的运行速度要比人的操作速度高出许多数量级。在他启动第一个程序后，该程序开始执行并向对等程序发送消息。在几个微秒内，它便发现对等程序还不存在，于是就发出一条错误消息，然后退出。此后，他启动了第二个程序。不幸的是，当第二个程序开始执行时，它也找不到第一个程序（早已退出）。即使这两个程序连续地重新试着通讯，但由于它们的执行速度那么高，以致于它们在同一瞬间联系上的概率非常低。在客户机/服务器结构中，服务器在启动后必须（无限期地）等待客户机的“请求”，因此就形成了“请求——响应”的通讯方式。

在 Internet/Intranet 领域，目前“浏览器—Web 服务器—数据库服务器” 结构是一种非常流行的客户机/服务器结构，如图 5.6 所示。这种结构最大的优点是：客户机统一采用浏览器，这不仅让用户使用方便，而且使得客户机端不存在维护的问题。当然，软件开发布和维护的工作不是自动消失了，而是转移到了 Web 服务器端。在 Web 服务器端，程序员要用脚本语言编写响应页面。例如用 Microsoft 的 ASP 语言查询数据库服务器，将结果保存在 Web 页面中，再由浏览器显示出来。

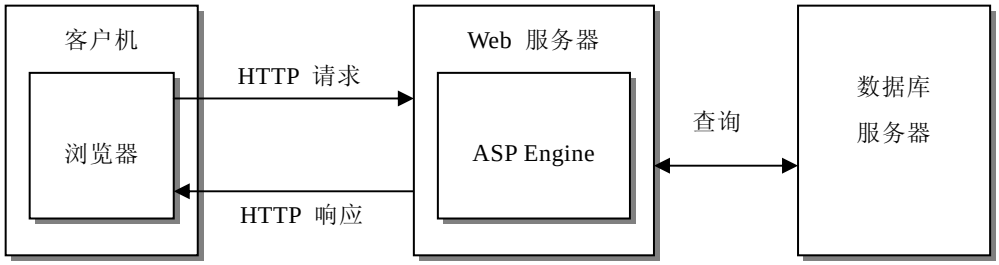


图 5.6　“浏览器—Web 服务器—数据库服务器”结构

5.2　模 块 设 计

在设计好软件的体系结构后，就已经在宏观上明确了各个模块应具有什么功能，应

放在体系结构的哪个位置。我们习惯地从功能上划分模块，保持“功能独立”是模块化设计的基本原则。因为，“功能独立”的模块可以降低开发、测试、维护等阶段的代价。但是“功能独立”并不意味着模块之间保持绝对的孤立。一个系统要完成某项任务，需要各个模块相互配合才能实现，此时模块之间就要进行信息交流。

比如手和脚是两个“功能独立”的模块。没有脚时，手照样能干活。没有手时，脚仍可以走路。但如果希望跑得快，那么迈左脚时一定要伸右臂甩左臂，迈右脚时则要伸左臂甩右臂。在设计一个模块时不仅要考虑“这个模块就该提供什么样的功能”，还要考虑“这个模块应该怎样与其它模块交流信息”。

本节将论述评价模块设计优劣的三个特征因素：“信息隐藏”、“内聚与耦合”和“封闭——开放性”。

5.2.1 信息隐藏

在一节不和谐的课堂里，老师叹气道：“要是坐在后排聊天的同学能象中间打牌的同学那么安静，就不会影响到前排睡觉的同学。”

这个故事告诉我们，如果不想让坏事传播开来，就应该把坏事隐藏起来，“家丑不可外扬”就是这个道理。为了尽量避免某个模块的行为去干扰同一系统中的其它模块，在设计模块时就要注意信息隐藏。应该让模块仅仅公开必须要让外界知道的内容，而隐藏其它一切内容。

模块的信息隐藏可以通过接口设计来实现。一个模块仅提供有限个接口（Interface），执行模块的功能或与模块交流信息必须且只须通过调用公有接口来实现。如果模块是一个 C++对象，那么该模块的公有接口就对应于对象的公有函数。如果模块是一个 COM 对象，那么该模块的公有接口就是 COM 对象的接口。一个 COM 对象可以有多个接口，而每个接口实质上是一些函数的集合。[Rogerson 1999]

美国也许是世界上丑闻最多的国家，因为它追求民主，不懂得“隐藏信息”。但美国又是软件产业最发达的国家，模块化设计的方法都是美国人倡导的，他们应该很懂得“隐藏信息”。真是前后矛盾，这些美国佬！

5.2.2 内聚与耦合

内聚（Cohesion）是一个模块内部各成分之间相关联程度的度量。耦合（Coupling）是模块之间依赖程度的度量。内聚和耦合是密切相关的，与其它模块存在强耦合的模块通常意味着弱内聚，而强内聚的模块通常意味着与其它模块之间存在弱耦合。模块设计追求强内聚，弱耦合。

一、内聚强度

内聚按强度从低到高有以下几种类型：

（1）偶然内聚。如果一个模块的各成分之间毫无关系，则称为偶然内聚。

（2）逻辑内聚。几个逻辑上相关的功能被放在同一模块中，则称为逻辑内聚。如一个模块读取各种不同类型外设的输入。尽管逻辑内聚比偶然内聚合理一些，但逻辑内聚的模块各成分在功能上并无关系，即使局部功能的修改有时也会影响全局，因此这类模块的修改也比较困难。

（3）时间内聚。如果一个模块完成的功能必须在同一时间内执行（如系统初始化），但

这些功能只是因为时间因素关联在一起，则称为时间内聚。

（4）过程内聚。如果一个模块内部的处理成分是相关的，而且这些处理必须以特定的次序执行，则称为过程内聚。

（5）通信内聚。如果一个模块的所有成分都操作同一数据集或生成同一数据集，则称为通信内聚。

（6）顺序内聚。如果一个模块的各个成分和同一个功能密切相关，而且一个成分的输出作为另一个成分的输入，则称为顺序内聚。

（7）功能内聚。模块的所有成分对于完成单一的功能都是必须的，则称为功能内聚。

二、耦合强度

耦合的强度依赖于以下几个因素：（1）一个模块对另一个模块的调用；（2）一个模块向另一个模块传递的数据量；（3）一个模块施加到另一个模块的控制的多少；（4）模块之间接口的复杂程度。

耦合按从强到弱的顺序可分为以下几种类型：

（1）内容耦合。当一个模块直接修改或操作另一个模块的数据,或者直接转入另一个模块时，就发生了内容耦合。此时，被修改的模块完全依赖于修改它的模块。

（2）公共耦合。两个以上的模块共同引用一个全局数据项就称为公共耦合。

（3）控制耦合。一个模块在界面上传递一个信号（如开关值、标志量等）控制另一个模块，接收信号的模块的动作根据信号值进行调整，称为控制耦合。

（4）标记耦合。模块间通过参数传递复杂的内部数据结构，称为标记耦合。此数据结构的变化将使相关的模块发生变化。

（5）数据耦合。模块间通过参数传递基本类型的数据，称为数据耦合。

（6）非直接耦合。模块间没有信息传递时，属于非直接耦合。

如果模块间必须存在耦合，就尽量使用数据耦合，少用控制耦合，限制公共耦合的范围，坚决避免使用内容耦合。

5.2.3 封闭——开放性

如果一个模块可以作为一个独立体被其它程序引用，则称模块具有封闭性。如果一个模块可以被扩充，则称模块具有开放性。

从字面上看，让模块具有“封闭——开放性”是矛盾的，但这种特征在软件开发过程中是客观存在的。当着手一个新问题时，我们很难一次性解决问题。应该先纵观问题的一些重要方面，同时作好以后补充的准备。因此让模块存在“开放性”并不是坏事情。“封闭性”也是需要的，因为我们不能等到完全掌握解决问题的信息后再把程序做成别人能用的模块。

模块的“封闭——开放性”实际上对应于软件质量因素中的可复用性和可扩充性。采用面向过程的方法进行程序设计，很难开发出既具有封闭性又具有开放性的模块。采用面向对象设计方法可以较好地解决这个问题。

5.3 数据结构与算法设计

学会设计数据结构与算法，可以让我们编写出高效率的程序。也许有人要问，在计算机速度日新月异的今天，为什么还需要高效率的程序？

因为我们的雄心与能力是一起增长的，技术进步最快也快不过人们欲望的增长。计算速度和存储容量上的革新仅仅提供了处理更复杂问题的有效工具，所以高效率的程序永远不会过时。

设计高效率的程序是基于良好的数据结构与算法，而不是基于编程小技巧。大多数计算机科学系在设置课程时，都重视学习基本的软件工程原理，以及数据结构与算法设计。

一般说来，数据结构与算法就是一类数据的表示及其相关的操作（这里算法不是指数值计算的算法）。从数据表示的观点来看，存储在数组中的一个有序整数表也是一种数据结构。算法是指对数据结构施加的一些操作，例如对一个线性表进行检索、插入、删除等操作。一个算法如果能在所要求的资源限制（Resource Constraints）范围内将问题解决好，则称这个算法是有效率（Efficient）的。例如一个资源限制可能是“用于存储数据的内存有限”，或者“允许执行每个子任务所需的时间有限”。一个算法如果比其它已知算法所需要的资源都少，这个算法也被称为是有效率的。算法的代价（Cost）是指消耗的资源量。一般说来，代价是由一个关键资源例如时间或空间来评估的。

毋庸置疑，人们编写程序是为了解决问题。只有通过预先分析问题来确定必须达到的性能目标，才有希望挑选出正确的数据结构。有相当多的程序员忽视了这一分析过程，而直接选用某一个他们习惯使用的，但是与问题不相称的数据结构，结果设计出一个低效率的程序。如果使用简单的设计就能够达到性能目标时，选用复杂的数据结构也是没有道理的。

人们对常用的数据结构与算法的研究已经相当透彻，可以归纳出一些设计原则：

（1）每一种数据结构与算法都有其时间、空间的开销和收益。当面临一个新的设计问题时，设计者要彻底地掌握怎样权衡时空开销和算法有效性的方法。这就需要懂得算法分析的原理，而且还需要了解所使用的物理介质的特性（例如，数据存储在磁盘上与存储在内存中，就有不同的考虑）。

（2）与开销和收益有关的是时间——空间的权衡。通常可以用更大的时间开销来换取空间的收益，反之亦然。时间——空间的权衡普遍地存在于软件开发的各个阶段中。

（3）程序员应该充分地了解一些常用的数据结构与算法，避免不必要的重复设计工作。

（4）数据结构与算法为应用服务。我们必须先了解应用的需求，再寻找或设计与实际应用相匹配的数据结构。[Shaffer 1998]

5.4 用 户 界 面 设 计

某个人总有办法让自己保持心情愉快、信心十足。有一天，他向一名围棋九段和一名乒乓球世界冠军挑战，结果他全胜了。因为他跟围棋九段打乒乓球，跟乒乓球冠军下

围棋。用户界面的编程技术是人们熟悉得不得了的事，我决定讲一讲比较陌生的“用户界面设计美学”。

有位爱好书画的博士后请我欣赏钢琴演奏会。我从头到尾只听到“叮叮咚咚”的声音，实在享受不到“高雅”，就请这位朋友指点。他虽然也不懂钢琴，却从欣赏书法的角度设法解释如何欣赏音乐。可是我既不懂书法也不懂音乐，真是坐立不安。“美”似乎真的不可言传。我在读本科时，特别喜欢编写用户界面程序，并且常向同学演示、卖弄。我觉得还不过瘾，就写了一篇“用户界面设计美学”的短文[林锐 1997]。凡是路过我实验室的同学都被我逮住，被迫听完我得意之极的朗读，茫然者与痛苦者居多。不久我的朗读便所向披糜，闻声者逃之夭夭。现在我又把那篇短文摘录至此，请您忍着点吧。

5.4.1 界面设计中美的需求与导向作用

人们对美的向往和追求是与生俱有的。显然没有人愿意丑化自己的程序，也没有用户嗜好丑陋的界面。软件开发者要设计美，用户要享受美，所以界面的美是开发者与用户的共同需求。

界面美的概念很抽象，以致让人无法说清楚什么是界面的美。但它同时又很现实，以致人人都可以去欣赏和感受界面美，并且挑剔美中之不足。美学不是一种量化的学问，如果因此而轻视美学指导，必将导致在设计过程中光依赖程序员个人的经验与感觉。由于程序员接受的教育主要是如何使计算机完成工作，而不是人如何工作，因此仅靠程序员主观想象设计而成的界面往往得不到大众用户的认可。

美的界面能消除用户由感觉引起的乏味、紧张和疲劳（情绪低落），大大提高用户的工作效率，从而进一步为发挥用户技能和为用户完成任务作出贡献。从人机界面发展历史与趋势上可以看出人们对界面美的需求，以及美在界面设计中的导向作用。

界面设计已经经历了两个界限分明的时代。第一代是以文本为基础的简单交互，如常见的命令行，字符菜单等。由于第一代界面考虑人的因素太少，用户兴趣不高。随着技术的发展，出现了第二代直接操纵的界面。它大量使用图形、语音和其它交互媒介，充分地考虑了人对美的需求。直接操纵的界面使用视听、触摸等技术，让人可以凭借生活常识、经历和推理来操纵软件，愉快地完成任务。更高层次的界面甚至模拟了人的生活空间，例如虚拟现实环境。

界面的美充分体现了人机交互作用中人的特性与意图，越来越多的用户将通过具有吸引力而令人愉快的人机界面与计算机打交道。

5.4.2 界面美的内涵

本节从合适性、风格和广义美三个方面论述界面美的内涵。

一、界面的合适性

界面的合适性是指界面是否与软件功能相融洽。如果界面不适合于软件的功能，那么界面将毫无用处，界面美的内涵就无从谈起。所以界面的合适性是界面美的首要因素，它提醒设计者不要片面追求外观漂亮而导致失真或华而不实。界面的合适性既提倡外美内秀，又强调恰如其分。

合适性差的界面无疑会混淆软件意图，致使用户产生误解。即使它不损害软件功能与性能，也会使用户产生不该有的情绪波动。例如一些软件开发者喜欢为其作品加一段

动画演示，以便吸引更多用户的关注。这本是无可非议的，问题在于这演示是否合情合理。如果运行一个程序，它首先表演一套复杂的动画，在后台演奏雄壮的进行曲，电闪雷鸣之后出来的却是一个普通的文本编辑器。整个过程让用户置身于云里雾里，而结果却让用户感到惊愕而不是惊喜。合适性差的界面只会给软件带来厄运。

二、界面的风格

界面的风格有两类，一是“一致性”，二是“个性化”。

商业应用软件的界面设计注重一致性。设计者必须密切注意在相同应用领域中最流行的软件的界面，必须尊重用户使用这些软件的习惯。例如商业软件习惯于设置 F1 键为帮助热键，如果某个设计者别出心裁地让 F1 键成为程序终止的热键，那么在用户渴望得到帮助而伸手击 F1 键的一刹那，他的工作就此完蛋。相信这个用户“一朝被蛇咬，十年怕井绳”。

目前流行的软件开发工具如 Visual C++、Visual Basic、Delphi、C++ Builder、Power Builder 等，都能够快速地开发出非常相似的图形用户界面。在 Internet/Intranet 领域，浏览器几乎成了唯一的客户机程序，因为用户希望用完全一致的软件来完成千变万化的应用任务。

在娱乐领域的软件中，有个性化的界面自然比泯然于众的界面更具有吸引力。一般说来，计算机专业人员玩过的软件不计其数。界面看多了，真有种“曾经沧海难为水”的感觉。不过当我看到一个叫 Sonique 的放音乐的软件时，不禁对其界面的创意啧啧称赞，忍不住象贴美女像那样把它贴到书中，如图 5.7 所示。

图 5.7　Sonique 软件的几种界面

人们经常搞不清楚什么情况下应该追求“一致性”或“个性化”。在大白天，当人们

都穿戴整齐时，有些人喜欢只挂几片遮羞布。而当大家都赤条条地在共公浴室洗澡时，却也有人喜欢穿着衣服。

三、界面的广义美

尽管界面的美并没有增加软件的功能与性能，却又是必为可少的。用户使用界面时，除了直接的感官美感外，还有很大一部分美感是间接的，它们存在于人们的使用体验中，例如方便，实用等。与图形用户界面相比，命令行是最原始的界面，它难记又难看。但对于熟练的用户而言，他们乐于使用命令行以获得高效率。命令行因具有高效率而赢得了专业人士的喜爱，早期的 Unix 系统就是彻头彻尾的命令系统。可以说，一切有利于人机交互的界面设计因素都具有广义美。

界面设计的一些特殊考虑也体现了广义美，如设法使残障人也可以使用软件。IBM 公司在 1985 年已经创建了残障人国家支持中心。Apple 公司的专门教育办公室则提供了一些有利于残障人使用的计算机信息产品。

5.5 系统设计示例

——支持协同工作的交互式三维图形软件开发系统

本节论述“支持协同工作的交互式三维图形软件开发系统”的系统设计，作为本章的示例（ 取材于作者的博士论文工作[林锐 2000]）。

5.5.1 设计背景

图形标准在图形领域有着重要的地位，它不仅加速了 3D 应用程序的开发，而且使 3D 应用程序的可移植性更好。历史上曾出现的图形标准（或 API）有 Core、GKS、PHIGS、PEX、GL、Dore、RenderMan、Hoops、OpenGL 等等。经过竞争与淘汰，目前 OpenGL 成为国际上公认的 3D 图形工业标准，在 Unix 与 PC 平台得到广泛应用。OpenGL 提供了数百个库函数，可以方便地绘制具有真实感的 3D 图形。但是在开发交互式的 3D 图形应用程序时，图形的绘制只是一部分工作，更多的工作集中在场景数据结构、图形对象、三维交互和图形用户界面的设计上。

由于 OpenGL 与窗口系统无关，不提供任何交互手段，必须由程序员自己编写所有的交互功能。并且 OpenGL 的编程接口是低级的 C 函数，不提供可复用的对象库或者应用程序框架，开发效率不高。为了克服这些困难，人们往往在图形标准之上再建立更高级的开发工具(3D Toolkit 或 3D Engine)。基于 OpenGL 的著名的开发工具有 Open Inventor、IRIS Performer、Optimeizer/Cosmo3D 以及 GLUT 等等。Open Inventor 被誉为是交互式 3D 开发工具的“事实标准”，但显然没有一个 3D 开发工具能满足所有的应用需求，3D 的广泛应用需要更多的开发工具支持。

随着计算机图形技术与网络技术的迅猛发展，两者的结合势在必行。在商业、科研、教育、娱乐等领域，用于分布式虚拟环境（Distributed Virtual Environments, DVEs）和计算机支持协同工作（Computer Supported Cooperative Work, CSCW）的图形系统已成为研究与应用的热点。著名的 DVEs 系统有 DIVE、dVS、MR、Repo-3D 等。但是这些 DVEs

系统缺乏 3D 开发工具的交互式图形功能以及通用性，而通用的 3D 开发工具如 Open Inventor 则又不支持分布式计算和协同工作。由于在窗口系统、图形支撑库、编程语言等方面存在差异，上述 DVEs 系统和 3D 开发工具难以方便地结合使用。

我们多方面分析了 3D 需求及软硬件条件，研制完成运行于 PC 平台，支持协同工作的交互式三维图形软件开发系统，如图 5.8 所示。其中：

（1）Intra3D 2.0 是基于 OpenGL 的通用交互式三维图形软件开发工具，可用于快速开发 Window 9x/NT 下的交互式三维图形应用软件。

（2）CNC 1.0 是支持协同工作的网络通讯开发系统（Cooperative Network Communicator），其核心是支持“发布—订阅模式”与“组播模式”的服务器与 API。

结合 Intra3D 2.0 和 CNC 1.0，可以快速开发支持协同工作的交互式三维图形应用软件。

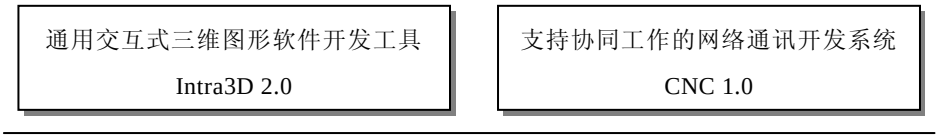


图 5.8 支持协同工作的交互式三维图形软件开发系统

5.5.2 通用交互式三维图形软件开发工具 Intra3D 2.0

Intra3D 2.0 的核心是集成了场景数据结构、图形对象、三维交互算法和图形用户界面的 C++类库与 COM（Component Object Model）对象库，支持 Visual C++、Visual Basic、Delphi 等语言的应用编程。Intra3D 的核心库分四层创建：

（1）第一层为“基础对象与函数”（Basic Objects and Functions）；

（2）第二层为“图形对象”（Graphical Objects）；

（3）第三层为“场景图与节点”（Scene Graph and Nodes）；

（4）第四层为“绘制与交互”（Rendering and Interaction）。

体系结构如图 5.9 所示，其中高层构件可以引用低层构件，但低层构件不能引用高层构件。

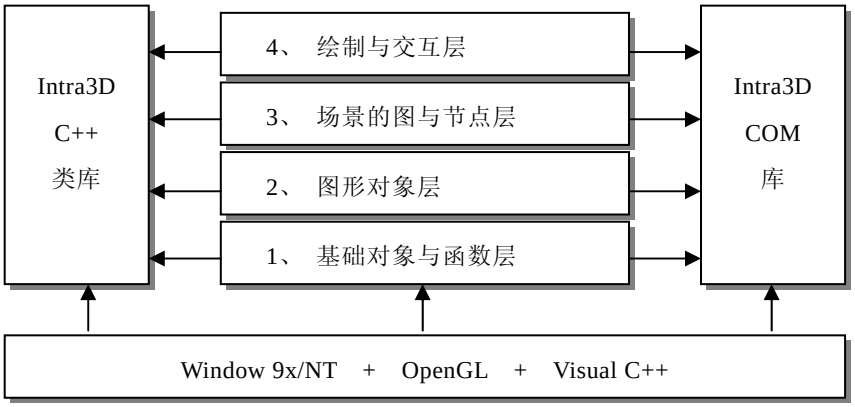


图 5.9 Intra3D 2.0 的体系结构

Intra3D 2.0 是免费软件，有配套书籍《交互式三维图形技术与程序设计》。标准版软

件约 25 兆，核心库 7 万多行 C++代码全部公开，用户可以方便地修改内核以适应不同的需求。

5.5.2.1 主要模块和功能

一、基础对象与函数层

（1）定义了用于对象引用计数的内存管理基类；

（2）矢量、矩阵与四元组运算，鼠标跟踪球算法；

（3）点阵字体与三维矢量字体输出，常用于数据可视化图形的数据标注；

（4）图像输入输出以及纹理映射，支持 BMP、GIF、JPEG、SGI、TGA 等图像格式；

（5）常用几何图元的绘制，如锥、柱、球、环等，并支持 Swept 形体，螺旋体的绘制；

（6）提供 450 余种材质，在第四层中可以交互式编辑这些材质。

二、图形对象层

图形对象能将数据转化为几何模型并可以绘制出来。Intra3D 2.0 版提供了三类图形对象：

（1）常用几何对象，如长方体、锥体、圆柱体、球体、圆环体、Swept 形体等；

（2）多边形模型对象，可用于绘制 Autodesk 公司.3ds 模型和 Wavefront 公司的.obj 模型；

（3）商业统计图形对象，如柱形图、带状图、条形图、折线图、面积图、饼图、塔形图、曲线图、曲面图、进程图、股票图等。

图形对象的开发与应用问题密切相关，用户可以使用继承方法扩充新的图形对象，而不会影响到其它三层的构件。

三、场景图与节点

场景图（Scene Graph）是有向无环图，Scene Graph 的主要节点有：（1）SceneNode 是所有节点的基类。在 SceneNode 中定义了局部坐标系以及相应的图形变换，这样便于第四层以同样的操作方式实现三维交互。（2）相机节点（CameraNode）支持平行投影与透视投影，支持多个相机切换。（3）光源节点有三种：平行光源节点(DirLightNode)、点光源节点(PointLightNode)和锥光源节点(SpotLightNode)。（4）形体节点（ShapeNode）用于引用图形对象，有关图形对象的三维交互均由 ShapeNode 处理。

四、绘制与交互层

Intra3D 的交互分两类：一类是对形体、光源和相机的直接操作，另一类是真实感属性的编辑。Intra3D 的场景视图构件（SceneView）封装了交互式绘制的所有细节，如消息处理、场景节点的遍历绘制、多重采样消锯齿、鼠标交互等。为了便于编辑真实感属性，Intra3D 定制了一些常用对话：矢量字体对话（FontDialog）、颜色对话（ColorDialog）、材质库对话（MaterialLibDialog）、材质对话（MaterialDialog）与光源对话（DirLightDialog, PointLightDialog, SpotLightDialog）。

5.5.2.2 用户界面设计

Intra3D 的场景视图构件 SceneView 用于快速创建交互式 3D 应用程序的主界面。SceneView 支持 selecting、scaling、rotating、translating、creating、deleting 等三十余种操作，并提供工具条方便于交互，如图 5.10 所示。

为了编辑真实感属性，常需在对话框中绘制 3D 图形。Microsoft 的窗口系统不提供

3D 的对话窗口。使用 Intra3D 的 Window3D 构件可在对话框中创建多个 3D 视图，图 5.11 的材质对话和图 5.12 的材质库对话都使用了 Window3D 构件。颜色编辑是 3D 图形程序中最常用的交互，材质与光源的编辑实际上是通过改变颜色分量来实现的。需要进行颜色编辑的交互均涉及 HSV 与 RGB 模式的颜色转换。Intra3D 的“绘制与交互层”实现了这些计算，并且提供彩色的滑动条用于鼠标交互。图 5.13、图 5.14 分别为点光源对话和颜色对话。

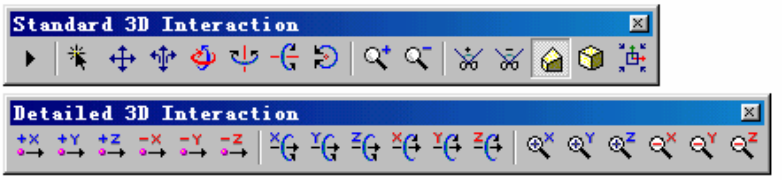


图 5. 10　用于直接操作的三维交互工具条

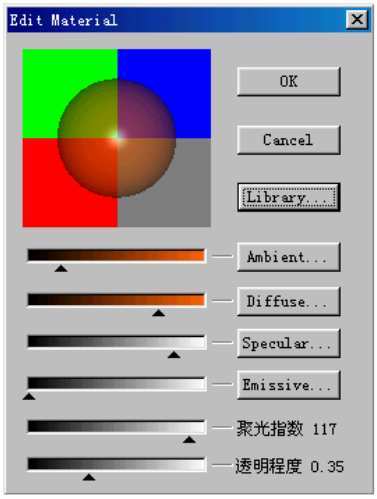


图 5.11　材质对话

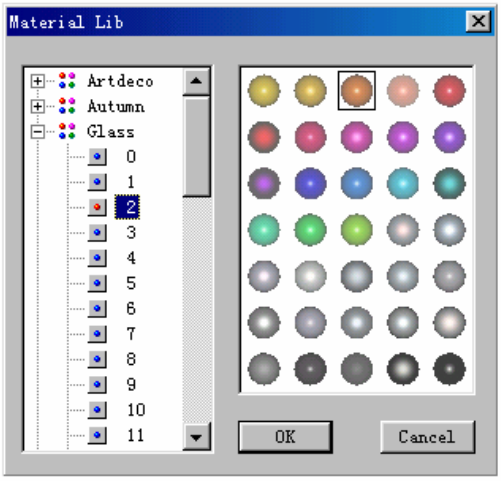


图 5.12　材质库对话

图 5.13 点光源对话

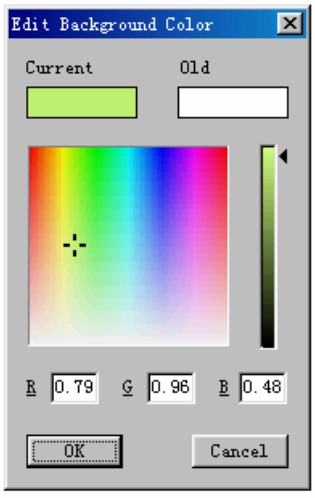


图 5.14　颜色对话

5.5.3 支持协同工作的网络通讯开发系统 CNC 1.0

最简单的协同工作模式是让两个客户机直接通讯，可以用 Socket 编程实现。假设有 n 个客户机参加协同工作，每个客户机将与所有其它的客户机通讯。那么总共存在 n(n-1)/2 个 Socket 直接通讯，并且每个客户机的变动将导致其它客户机的修改。这种 Socket 直接通讯使得协同工作的管理和客户机的程序设计变得非常困难。CNC 系统提供了支持“发布—订阅（Publish-Subscribe）”与“组播（Multicast）” 模式的服务器与 API，可以高效地管理多个组群的协同工作，并使得客户机的程序设计十分简单。CNC 1.0 的系统结构如图 5.15 所示。

CNC 服务器将客户机分组管理。在“发布—订阅”模式中，将产生数据的进程称为生产者（Producer），将接受数据的进程称为消费者（Consumer）。生产者可以向服务器发布数据，服务器保存这些数据。消费者可以向服务器订阅数据。每个客户机可能是很多数据的生产者或消费者。同一时刻，CNC 系统允许有任意多个生产者和消费者存在。CNC 的“发布—订阅” 功能是用 TCP 协议实现的。

在“组播” 模式中，服务器动态地分配每个组的组播地址。客户机可以向服务器申请加入任意组，允许向任意组播放消息，服务器不保存这些组播消息。CNC 的“组播”功能是用 UDP 协议实现的。

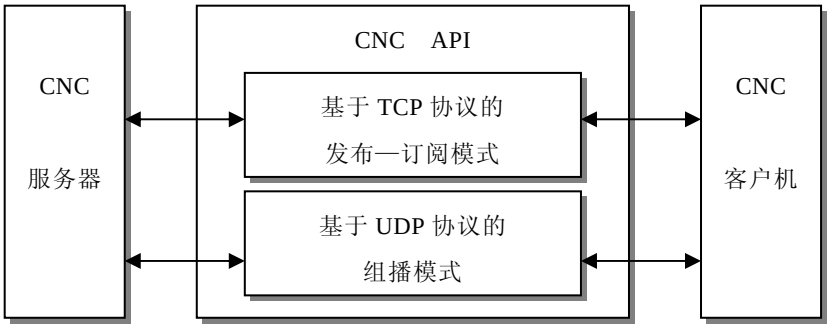


图 5.15　CNC 1.0 的系统结构

5.5.3.1　CNC 客户机的 API 设计

类 CNCClient 客户机用来实现“发布—订阅”和“组播”功能，主要接口（公有函数）如下：

```
class CNCClient
{    public:
     BOOL Connect(...);            // 连接服务器
     BOOL Disconnect();            // 与服务器断开连接
     BOOL PublishData(...);        // 向服务器发布数据
     BOOL QueryData(...);          // 向服务器查询数据
     BOOL SubscribeData(...);      // 向服务器订阅数据
     GROUPIP QueryGroupIP(...);    // 向服务器查询组播地址
     DWORD MulticastMessage(...);// 发送组播消息
     virtual void MessageResponse(...);// 响应组播消息
…
```

```
};
```

一、客户程序的“发布”协议

客户机向服务器发布的每个数据报均含有数据类型、工作组名称、数据名称、生命期和数据长度的信息。报文格式如图 5.16 所示，数据结构见 DataPublish ：

```
struct DataPublish
{
    BYTE     iDataType;         // 2 个字节数据类型，宏定义为 DATA_PUBLISH
    char     strGroupName[16]; // 16 个字节的工作组名字
    char     strDataName[16];  // 16 个字节的数据名字
    DWORD    dwLifeTime;        // 4 个字节的数据生命期，以秒为单位
    DWORD    dwLength;          // 4 个字节的数据内容的长度
    char     *pchContent;       // 数据内容
};
```

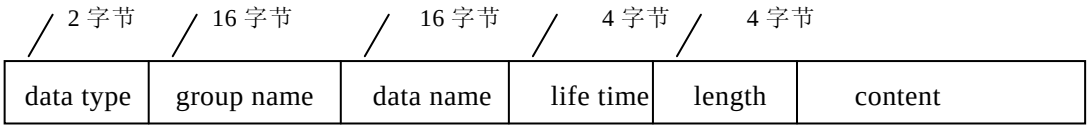


图 5.16　用于发布的报文格式

二、客户程序的“订阅”协议

客户机向服务器订阅数据分两步实现：

（1）先调用函数 QueryData 向服务器发送一个 DataQuery 格式的报文，用于查询要订阅的数据是否存在。

```
struct DataQuery
{
    BYTE     iDataType;             // 2 个字节数据类型，宏定义为 DATA_QUERY
    char     strGroupName[16];      // 16 个字节的工作组名字
    char     strDataName[16];       // 16 个字节的数据名字
};
```

（2）服务器接收到查询时，按照 DataQuery 结构中的 strGroupName 和 strDataName 进行搜索。如果该数据不存在，Server 向 Client 发送一个 FALSE 标志。如果该数据存在，服务器先向客户机发送一个 TRUE 标志，之后立即再向客户机发送该数据（DataPublish 格式）。

如果客户机得到 TRUE 标志的查询结果，就调用函数 SubscribeData 来接收服务器发送过来的数据。

三、客户程序的“组播”协议

客户机先调用函数 QueryGroupIP 向服务器发送一个 GroupAddress 格式的报文，用于查询组播地址。服务器返回相应的十进制点分式的 IP 地址。

```
struct GroupAddress
{
    BYTE     iDataType;              // 2 个字节数据类型，宏定义为 GROUP_ADDRESS
    char     strGroupName[16];       // 16 个字节的工作组名字
};
```

客户机调用函数 MulticastMessage 向指定的组（根据组播地址）播放消息。组播的数据报结构 DataMulticast 定义如下：

```
struct DataMulticast
{
    DWORD  dwContentType;    //  组播的数据报类型，由用户定义
    char     *pchContent;      //  组播的数据报内容，由用户定义
};
```

如果客户机接收到组播的消息，将自动调用函数 MessageResponse 来响应该消息。MessageResponse 是虚函数，它将根据 dwContentType 信息决定如何处理到来的组播消息，具体功能由用户定义。

5.5.3.2 CNC 服务器的设计

一、数据结构

CNC 服务器的数据结构主要由三部分组成：

（1）一张用于管理组播地址的链表。组播地址由服务器动态生成，客户机可以向服务器查询任意组的组播地址。

（2）一张用于管理线程指针的链表。服务器采用多线程并发处理技术，使客户机获得最快的响应。

（3）每个组都有一张用于管理“发布—订阅”的数据的 Hash 表。由于同一时刻，系统可能存在多个生产者与消费者，数据的存入、取出速度成为服务器性能的重要指标。Hash 表可以提供比链表更快的数据检索速度。Hash 表中的数据项结构见 DataElement ：

```
struct DataElement
{
    char     strGroupName[16]; // 工作组的名称
    char     strDataName[16];  // 数据的名称
    BYTE     iStorageType;     // 存储类型： STORAGE_FILE 或 STORAGE_MEMORY
ColeDateTime  TimeToDie;       // 作废时刻
    BOOL     bLock;            // 锁定标志： TRUE 或 FALSE
    DWORD    dwLength;         // 数据的长度
    char     *pchContent;      // 数据内容
};
```

存储类型（iStorageType）的用途：把数据全部保存在内存中将非常消耗服务器的内存资源，在很多情况下是没有必要的。为了提高内存的使用效率，服务器仅把生命期较短或者长度较短的数据保存在内存中（即为 STORAGE_MEMORY 类型），而把生命期较长

或者长度较长的数据保存在文件中（即为 STORAGE_FILE 类型）。

作废时刻（TimeToDie）的用途：客户机发布的数据均指定了生命期，服务器在接收到数据时即可计算出作废时刻。服务器将定期扫描 Hash 表，若发现有数据超出作废时刻（并且没有被锁定），即可删除此数据。

锁定标志（bLock）的用途：很多客户机可能同时订阅某个数据，而该数据可能已超出作废时刻即将被删除。为避免冲突，规定只要有客户机订阅数据，就用 iLock 标志来锁定此数据，直到订阅完成后才消除锁定。

二、多线程并发技术

服务器有一个主线程和多个子线程。主线程负责客户机的入连接请求，然后创建一个子线程来处理这个 TCP 连接。每个子线程按照 CNC API 的协议与客户机通讯。由于有多个子线程共享服务器中的数据，多线程对共享资源的同步访问成为实现的难点。CNC 主要采用了关键区、互斥对象等同步手段解决这个问题。

三、Winsock 的使用

CNC 1.0 运行于 Windows 9x/NT 系统下，底层的网络通讯程序用 Winsock 编写。Winsock 有两种工作方式：阻塞方式和非阻塞方式。阻塞方式的优点是编程简单，可靠性好。缺点是容易使应用程序阻塞住，不能处理其它事务。非阻塞方式是利用 Windows 消息机制实现的。优点是在数据到来的时候，系统向应用程序窗口发送消息，使得应用程序不必总在等待数据，提高了工作效率。缺点是在发送和接收数据时，应用程序并不将事情做完（不阻塞），以至于应用程序要维护复杂的状态机。

鉴于阻塞方式和非阻塞方式各有优缺点，CNC 服务器采用了混合方式。主线程采用非阻塞的消息驱动方式，可以快速响应客户机的入连接。在子线程中，仍采用非阻塞的消息驱动方式接受客户机的请求，只有在响应请求时，采用阻塞的方式一次性地完成数据的发送或接收。

5.5.4 应用示例

图 5.17、图 5.18 是参加协同工作的两个客户程序示例，这两个程序均用 Intra3D 2.0 和 CNC 1.0 开发。图 5.17 的客户程序向 CNC 服务器订阅 .3ds 和 .obj 格式的多边形模型数据并执行交互式绘制。图 5.18 的客户程序向 CNC 服务器订阅商业统计图形数据并执行交互式绘制。另有一个客户机（数据源）向 CNC 服务器发布各种数据，并用组播来通知各个客户机当前发布了什么数据（短消息）。

Intra3D 2.0 和 CNC 1.0 目前已经可以实用，请与作者联系，免费索取软件。

5.6 小 结

让我们用著名 3D 游戏软件 Quake 的设计师 Michael Abrash 的话总结本章：“所有真正杰出的设计一旦被设计好，看起来都是那么的简单和显而易见。但是在获得杰出设计的过程中，需要付出令人难以置信的努力。”[Abrash 1998]

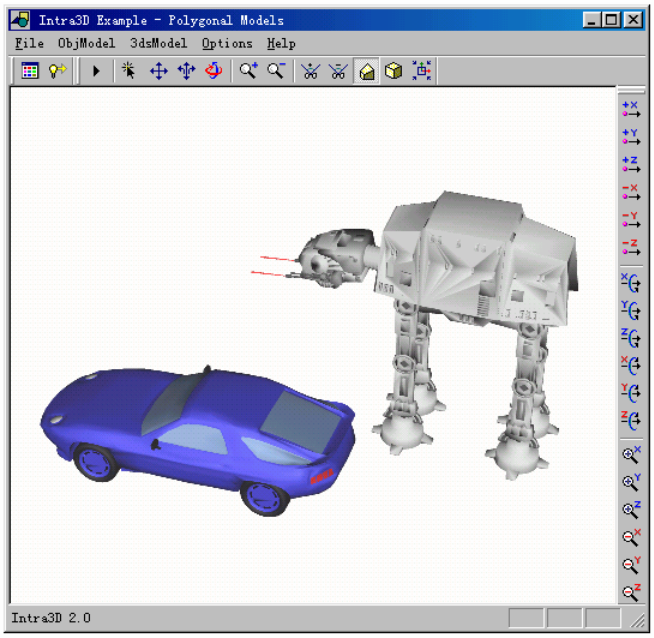


图 5.17　绘制.3ds 和.obj 模型的客户程序

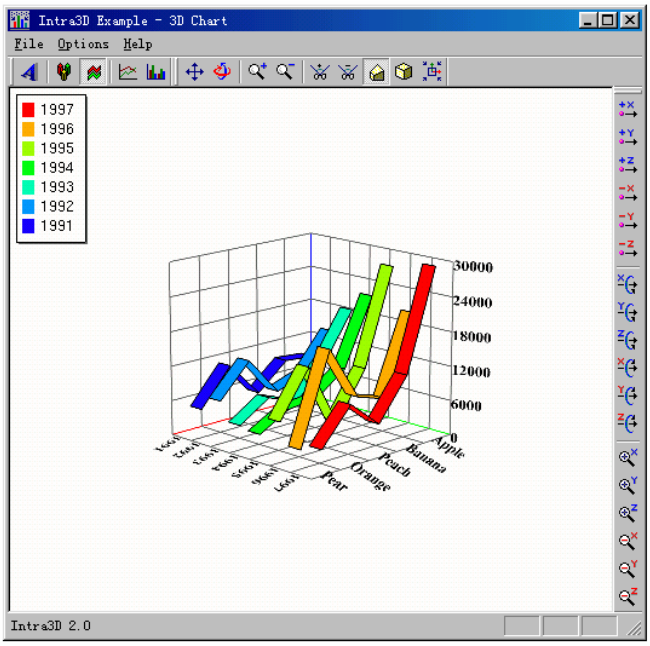


图 5.18　绘制商业统计图形的客户程序

第六章　C++面向对象程序设计

六年前，我刚热恋“面向对象”（Object-Oriented）时，一口气记住了近十个定义。六年后，我从几十万行程序中滚爬出来准备写点心得体会时，却无法解释什么是“面向对象”，就象说不清楚什么是数学那样。

软件工程中的时髦术语“面向对象分析”和“面向对象设计”，通常是针对“需求分析”和“系统设计”环节的。“面向对象”有几大学派，就象如来佛、上帝和真主用各自的方式定义了这个世界，并留下一堆经书来解释这个世界。

有些学者建议这样找“对象”：分析一个句子的语法，找出名词和动词，名词就是对象，动词则是对象的方法（即函数）。

当年国民党的文人为了对抗毛泽东的《沁园春·雪》，特意请清朝遗老们写了一些对仗工整的诗，请蒋介石过目。老蒋看了气得大骂：“娘希匹，全都有一股棺材里腐尸的气味。”

我看了几千页的软件工程资料，终于发现自己有些“弱智”，无法理解“面向对象”的理论，同时醒悟到“编程是硬道理。”

面向对象程序设计语言很多，如 Smalltalk、Ada、Eiffel、Object Pascal、Visual Basic、C++等等。C++语言最讨人喜欢，因为它兼容 C 语言，并且具备 C 语言的性能。近几年，一种叫 Java 的纯面向对象语言红极一时，不少人叫喊着要用 Java 革 C++的命。我认为 Java 好比是 C++的外甥，虽然不是直接遗传的，但也几分象样。外甥在舅舅身上玩耍时洒了一泡尿，俩人不该为此而争吵。

关于 C++程序设计的书藉非常多，本章不讲 C++的语法，只讲一些小小的编程道理。如果我能早几年明白这些小道理，就可以大大改善数十万行程序的质量了。

6.1　C++面向对象程序设计的重要概念

早期革命影片里有这样一个角色，他说：“我是党代表，我代表党，我就是党。”后来他给同志们带来了灾难。

会用 C++的程序员一定懂得面向对象程序设计吗？

不会用 C++的程序员一定不懂得面向对象程序设计吗？

两者都未必。就象坏蛋入党后未必能成为好人，好人不入党未必变成坏蛋那样。

我不怕触犯众怒地说句大话：“C++没有高手，C 语言才有高手。”在用 C 和 C++编程 8 年之后，我深深地遗憾自己不是 C 语言的高手，更遗憾没有人点拨我如何进行面向对象程序设计。我和很多 C++程序员一样，在享用到 C++语法的好处时便以为自己已经明白了面向对象程序设计。就象挤掉牙膏卖牙膏皮那样，真是暴殄天物呀。

人们不懂拼音也会讲普通话，如果懂得拼音则会把普通话讲得更好。不懂面向对象程序设计也可以用 C++编程，如果懂得面向对象程序设计则会把 C++程序编得更好。本节讲述三个非常基础的概念：“类与对象”、“继承与组合”、“虚函数与多态”。理解这些

概念，有助于提高程序的质量，特别是提高“可复用性”与“可扩充性”。

6.1.1 类与对象

对象（Object）是类（Class）的一个实例（Instance）。如果将对象比作房子，那么类就是房子的设计图纸。所以面向对象程序设计的重点是类的设计，而不是对象的设计。

类可以将数据和函数封装在一起，其中函数表示了类的行为（或称服务）。类提供关键字 public、protected 和 private 用于声明哪些数据和函数是公有的、受保护的或者是私有的。这样可以达到信息隐藏的目的，即让类仅仅公开必须要让外界知道的内容，而隐藏其它一切内容（参见 5.2.1 节“信息隐藏”）。我们不可以滥用类的封装功能，不要把它当成火锅，什么东西都往里扔。

类的设计是以数据为中心，还是以行为为中心？

主张“以数据为中心”的那一派人关注类的内部数据结构，他们习惯上将 private 类型的数据写在前面，而将 public 类型的函数写在后面，如表 8.1(a)所示。

主张“以行为为中心”的那一派人关注类应该提供什么样的服务和接口，他们习惯上将 public 类型的函数写在前面，而将 private 类型的数据写在后面，如表 8.1(b)所示。

```
Class A
{
  private:
    int    i, j;
    float  x, y;
    …
  public:
    void Func1(void);
    void Func2(void);
    …
}
```

表 8.1(a) 以数据为中心的书写风格

```
class A
{
  public:
    void Func1(void);
    void Func2(void);
    …
  private:
    int    i, j;
    float  x, y;
    …
}
```

表 8.1(b)以行为为中心的书写风格

很多 C++教课书主张在设计类时“以数据为中心”。我坚持并且建议读者在设计类时“以行为为中心”，即首先考虑类应该提供什么样的函数。Microsoft 公司的 COM 规范的核心是接口设计，COM 的接口就相当于类的公有函数[Rogerson 1999]。在程序设计方面，咱们不要怀疑 Microsoft 公司的风格。

设计孤立的类是比较容易的，难的是正确设计基类及其派生类。因为有些程序员搞不清楚“继承”（Inheritance）、“组合”（Composition）、“多态”（ Polymorphism）这些概念。

6.1.2 继承与组合

如果 A 是基类，B 是 A 的派生类，那么 B 将继承 A 的数据和函数。示例程序如下：

```
class A
{
```

```
  public:
    void   Func1(void);
    void   Func2(void);
};

class B : public A
{
  public:
    void   Func3(void);
    void   Func4(void);
};

// Example
main()
{
    B   b;          // B 的一个对象
    b.Func1();      // B 从 A 继承了函数 Func1
    b.Func2();      // B 从 A 继承了函数 Func2
    b.Func3();
    b.Func4();
}
```

这个简单的示例程序说明了一个事实：C++的“继承”特性可以提高程序的可复用性。正因为“继承”太有用、太容易用，才要防止乱用“继承”。我们要给“继承”立一些使用规则：

一、如果类 A 和类 B 毫不相关，不可以为了使 B 的功能更多些而让 B 继承 A 的功能。不要觉得“白吃白不吃”，让一个好端端的健壮青年无缘无故地吃人参补身体。

二、如果类 B 有必要使用 A 的功能，则要分两种情况考虑：

（1）若在逻辑上 B 是 A 的“一种”（a kind of ），则允许 B 继承 A 的功能。如男人（Man）是人（Human）的一种，男孩（Boy）是男人的一种。那么类 Man 可以从类 Human 派生，类 Boy 可以从类 Man 派生。示例程序如下：

```
class Human
{
    …
};
class Man : public Human
{
    …
};
class Boy : public Man
```

```
{
    …
};
```

（2）若在逻辑上 A 是 B 的“一部分”（a part of），则不允许 B 继承 A 的功能，而是要用 A 和其它东西组合出 B。例如眼（Eye）、鼻（Nose）、口（Mouth）、耳（Ear）是头（Head）的一部分，所以类 Head 应该由类 Eye、Nose、Mouth、Ear 组合而成，不是派生而成。示例程序如下：

```
class Eye
{
  public:
    void    Look(void);
};
class Nose
{
  public:
    void    Smell(void);
};
class Mouth
{
  public:
    void    Eat(void);
};
class Ear
{
  public:
    void    Listen(void);
};

// 正确的设计，冗长的程序
class Head
{
  public:
    void        Look(void)      {   m_eye.Look();   }
    void        Smell(void)     {   m_nose.Smell();   }
    void        Eat(void)       {   m_mouth.Eat();   }
    void        Listen(void)    {   m_ear.Listen();   }
  private:
    Eye         m_eye;
    Nose        m_nose;
```

```
    Mouth      m_mouth;
    Ear        m_ear;
};
```

如果允许 Head 从 Eye、Nose、Mouth、Ear 派生而成，那么 Head 将自动具有 Look、Smell、Eat、Listen 这些功能：

```
// 错误的设计
class Head : public Eye, public Nose, public Mouth, public Ear
{
};
```

上述程序十分简短并且运行正确，但是这种设计却是错误的。很多程序员经不起“继承”的诱惑而犯下设计错误。

一只公鸡使劲地追打一只刚下了蛋的母鸡，你知道为什么吗？

因为母鸡下了鸭蛋。

本书 3.3 节讲过“运行正确”的程序不见得就是高质量的程序，此处就是一个例证。

6.1.3 虚函数与多态

除了继承外，C++的另一个优良特性是支持多态，即允许将派生类的对象当作基类的对象使用。如果 A 是基类，B 和 C 是 A 的派生类，多态函数 Test 的参数是 A 的指针。那么 Test 函数可以引用 A、B、C 的对象。示例程序如下：

```
class A
{
  public:
    void   Func1(void);
};

void Test(A *a)
{
    a->Func1();
}

class B : public A
{
    …
};
class C : public A
{
    …
};
```

```
// Example
main()
{
    A   a;
    B   b;
    C   c;
    Test(&a);
    Test(&b);
    Test(&c);
};
```

以上程序看不出“多态”有什么价值，加上虚函数和抽象基类后，“多态”的威力就显示出来了。

C++用关键字 virtual 来声明一个函数为虚函数，派生类的虚函数将覆盖（override）基类对应的虚函数的功能。示例程序如下：

```
class A
{
  public:
    virtual void   Func1(void){ cout<< “This is A::Func1 \n”}
};

void Test(A *a)
{
    a->Func1();
}

class B : public A
{
  public:
    virtual void   Func1(void){ cout<< “This is B::Func1 \n”}
};
class C : public A
{
  public:
    virtual void   Func1(void){ cout<< “This is C::Func1 \n”}
};

// Example
main()
{
```

```
    A   a;
    B   b;
    C   c;
    Test(&a);  // 输出  This is A::Func1
    Test(&b);  // 输出  This is B::Func1
    Test(&c);  // 输出  This is C::Func1
};
```

如果基类 A 定义如下：

```
class A
{
  public:
    virtual void   Func1(void)=0;
};
```

那么函数 Func1 叫作纯虚函数，含有纯虚函数的类叫作抽象基类。抽象基类只管定义纯虚函数的形式，具体的功能由派生类实现。

结合“抽象基类”和“多态”有如下突出优点：

（1）应用程序不必为每一个派生类编写功能调用，只需要对抽象基类进行处理即可。这一招叫“以不变应万变”，可以大大提高程序的可复用性（这是接口设计的复用，而不是代码实现的复用）。

（2）派生类的功能可以被基类指针引用，这叫向后兼容，可以提高程序的可扩充性和可维护性。以前写的程序可以被将来写的程序调用不足为奇，但是将来写的程序可以被以前写的程序调用那可了不起。

6.2　良好的编程风格

内功深厚的武林高手出招往往平淡无奇。同理，编程高手也不会用奇门怪招写程序。良好的编程风格是产生高质量程序的前提。

6.2.1 命名约定

有不少人编程时用拼音给函数或变量命名，这样做并不能说明你很爱国，却会让用此程序的人迷糊（很多南方人不懂拼音，我就不懂）。程序中的英文一般不会太复杂，用词要力求准确。

匈牙利命名法是 Microsoft 公司倡导的 [Maguire 1993]，虽然很烦琐，但用习惯了也就成了自然。没有人强迫你采用何种命名法，但有一点应该做到：自己的程序命名必须一致。

以下是我编程时采用的命名约定：

（1）宏定义用大写字母加下划线表示，如 MAX_LENGTH；

（2）函数用大写字母开头的单词组合而成，如 SetName, GetName ；

（3）指针变量加前缀 p，如 *pNode ；
（4）BOOL 变量加前缀 b，如 bFlag ；
（5）int 变量加前缀 i，如 iWidth ；
（6）float 变量加前缀 f，如 fWidth ；
（7）double 变量加前缀 d，如 dWidth ；
（8）字符串变量加前缀 str，如 strName ；
（9）枚举变量加前缀 e，如 eDrawMode ；
（10）类的成员变量加前缀 m_，如 m_strName, m_iWidth ；
对于 int, float, double 型的变量，如果变量名的含义十分明显，则不加前缀，避免烦琐。如用于循环的 int 型变量 i,j,k ； float 型的三维坐标（x,y,z）等。

6.2.2 使用断言

程序一般分为 Debug 版本和 Release 版本，Debug 版本用于内部调试，Release 版本发行给用户使用。

断言 assert 是仅在 Debug 版本起作用的宏，它用于检查“不应该”发生的情况。以下是一个内存复制程序，在运行过程中，如果 assert 的参数为假，那么程序就会中止（一般地还会出现提示对话，说明在什么地方引发了 assert）。

```
//复制不重叠的内存块
void    memcpy(void *pvTo, void *pvFrom, size_t size)
{
      void *pbTo = (byte *) pvTo;
      void *pbFrom = (byte *) pvFrom;
      assert( pvTo != NULL    &&    pvFrom != NULL );
      while(size - - > 0 )
            *pbTo + + = *pbFrom + + ;
      return (pvTo);
}
```

assert 不是一个仓促拼凑起来的宏，为了不在程序的 Debug 版本和 Release 版本引起差别，assert 不应该产生任何副作用。所以 assert 不是函数，而是宏。程序员可以把 assert 看成一个在任何系统状态下都可以安全使用的无害测试手段。

很少有比跟踪到程序的断言，却不知道该断言的作用更让人沮丧的事了。你化了很多时间，不是为了排除错误，而只是为了弄清楚这个错误到底是什么。有的时候，程序员偶尔还会设计出有错误的断言。所以如果搞不清楚断言检查的是什么，就很难判断错误是出现在程序中，还是出现在断言中。幸运的是这个问题很好解决，只要加上清晰的注释即可。这本是显而易见的事情，可是很少有程序员这样做。这好比一个人在森林里，看到树上钉着一块“危险”的大牌子。但危险到底是什么？树要倒？有废井？有野兽？除非告诉人们“危险”是什么，否则这个警告牌难以起到积极有效的作用。难以理解的断言常常被程序员忽略，甚至被删除。[Maguire 1993]

以下是使用断言的几个原则：
（1）使用断言捕捉不应该发生的非法情况。不要混淆非法情况与错误情况之间的区别，

后者是必然存在的并且是一定要作出处理的。

（2）使用断言对函数的参数进行确认。

（3）在编写函数时，要进行反复的考查，并且自问：“我打算做哪些假定？”一旦确定了的假定，就要使用断言对假定进行检查。

（4）一般教科书都鼓励程序员们进行防错性的程序设计，但要记住这种编程风格会隐瞒错误。当进行防错性编程时，如果“不可能发生”的事情的确发生了，则要使用断言进行报警。

6.2.3 new、delete 与指针

在 C++中，操作符 new 用于申请内存，操作符 delete 用于释放内存。在 C 语言中，函数 malloc 用于申请内存，函数 free 用于释放内存。由于 C++兼容 C 语言，所以 new、delete、malloc、free 都有可能一起使用。new 能比 malloc 干更多的事，它可以申请对象的内存，而 malloc 不能。

C++和 C 语言中的指针威猛无比，用错了会带来灾难。对于一个指针 p，如果是用 new 申请的内存，则必须用 delete 而不能用 free 来释放。如果是用 malloc 申请的内存，则必须用 free 而不能用 delete 来释放。

在用 delete 或用 free 释放 p 所指的内存后，应该马上显式地将 p 置为 NULL，以防下次使用 p 时发生错误。示例程序如下：

```
void Test(void)
{
    float *p;
    p = new float[100];
    if(p==NULL) return;
    …// do something
    delete p;
    p=NULL;    // 良好的编程风格

    // 可以继续使用 p
    p = new float[500];
    if(p==NULL) return;
    …// do something else
    delete p;
    p=NULL;
}
```

我们还要预防“野指针”，“野指针”是指向“垃圾”内存的指针，主要成因有两种：

（1）指针没有初始化。

（2）指针指向已经释放的内存，这种情况最让人防不胜防，示例程序如下：

```
class A
{
```

```
public:
    void Func(void){…}
};
void Test(void)
{
    A   *p;
        {
            A   a;
            p = &a;    // 注意 a 的生命期
        }
    p->Func();      // p 是“野指针”，程序出错
}
```

6.2.4 使用 const

在定义一个常量时，const 比 #define 更加灵活。用 const 定义的常量含有数据类型，该常量可以参与逻辑运算。例如：

```
const int       LENGTH = 100;     // LENGTH 是 int 类型
const float     MAX=100;          // MAX 是 float 类型
#define         LENGTH   100      // LENGTH 无类型
#define         MAX      100      // MAX 无类型
```

除了能定义常量外，const 还有两个“保护”功能：

一、强制保护函数的参数值不发生变化

以下程序中，函数 f 不会改变输入参数 name 的值，但是函数 g 和 h 都有可能改变 name 的值。

```
void f(String   s);     // pass by value
void g(String &s);      // pass by referance
void h(String *s);      // pass by pointer
main()
{
    String   name=“Dog”;
    f(name);        // name 的值不会改变
    g(name);        // name 的值可能改变
    h(name);        // name 的值可能改变
}
```

对于一个函数而言，如果其‘&’或‘*’类型的参数只作输入用，不作输出用，那么应当在该参数前加上 const，以确保函数的代码不会改变该参数的值（如果改变了该参数的值，编译器会出现错误警告）。因此上述程序中的函数 g 和 h 应该定义成：

```
void   g(const String &s);
```

```
void    h(const String *s);
```

二、强制保护类的成员函数不改变任何数据成员的值

以下程序中，类 stack 的成员函数 Count 仅用于计数，为了确保 Count 不改变类中的任何数据成员的值，应将函数 Count 定义成 const 类型。

```
class Stack
{
  public:
    void            push(int elem);
    void            pop(void);
    int         Count(void)    const;  // const 类型的函数
  private:
    int         num;
    int         data[100];
};

int Stack::Count(void)    const
{
    ++ num;    // 编译错误，num 值发生变化
    pop();     // 编译错误，pop 将改变成员变量的值
    return num;
}
```

6.2.5 其它建议

（1）不要编写一条过分复杂的语句，紧凑的 C++/C 代码并不见到能得到高效率的机器代码，却会降低程序的可理解性，程序出错误的几率也会提高。

（2）不要编写集多种功能于一身的函数，在函数的返回值中，不要将正常值和错误标志混在一起。

（3）不要将 BOOL 值 TRUE 和 FALSE 对应于 1 和 0 进行编程。大多数编程语言将 FALSE 定义为 0，任何非 0 值都是 TRUE。Visual C++将 TRUE 定义为 1，而 Visual Basic 则将 TRUE 定义为-1。示例程序如下：

```
BOOL      flag;
…
if(flag)            { // do something }    // 正确的用法
if(flag==TRUE) { // do something }    // 危险的用法
if(flag==1)         { // do something }    // 危险的用法
if(!flag)           { // do something }    // 正确的用法
if(flag==FALSE) { // do something }  // 不合理的用法
if(flag==0)         { // do something }    // 不合理的用法
```

（4）小心不要将“= =”写成“=”，编译器不会自动发现这种错误。
（5）不要将 123 写成 0123，后者是八进制的数值。
（6）将自己经常犯的编程错误记录下来，制成表格贴在计算机旁边。

6.3 小 结

C++/C 程序设计如同少林寺的武功一样博大精深，我练了 8 年，大概只学到二三成。所以无论什么时候，都不要觉得自己的编程水平天下第一，看到别人好的技术和风格，要虚心学习。

本章的内容少得可怜，就象口渴时只给你一颗杨梅吃，你一定不过瘾。我借花献佛，推荐一本好书：Marshall P. Cline 著的《C++ FAQs》[Cline 1995]。你看了后一定会赞不绝口。

会编写 C++/C 程序，不要因此得意洋洋，这只是程序员基本的技能要求而已。如果把系统分析和系统设计比作“战略决策”，那么编程充其量只是“战术”。如果指挥官是个大笨蛋，士兵再勇敢也会吃败仗。所以我们程序员不要只把眼光盯在程序上，要让自己博学多才。我们应该向北京胡同里的小孩们学习，他们小小年纪就能指点江山，评论世界大事。

第七章　测试与改错

编程大师说：“任何一个程序，无论它多么小，总存在着错误。”

初学者不相信大师的话，他问：“如果一个程序小得只执行一个简单的功能，那会怎样？”

“这样的一个程序没有意义，”大师说，“但如果这样的程序存在的话，操作系统最后将失效，产生一个错误。”

但初学者不满足，他问：“如果操作系统不失效，那么会怎样？”

“没有不失效的操作系统，”大师说，“但如果这样的操作系统存在的话，硬件最后将失效，产生一个错误。”

初学者仍不满足，再问：“如果硬件不失效，那么会怎样？”

大师长叹一声道：“没有不失效的硬件。但如果这样的硬件存在的话，用户就会想让那个程序做一件不同的事，这件事也是一个错误。”

没有错误的程序世间难求。[James 1999]

错误是一种严重的程序缺陷。测试的目的是为了发现尽可能多的缺陷，并期望通过改错来把缺陷统统消灭，以期提高软件的质量。但关于测试与改错实在没有什么高明的方法值得大书特书，也不能表现出程序员的聪明才智。相反地，它们带来了更多的牢骚与痛苦。因此在教学和开发实践中，测试与改错总是被当作万般无奈的工作踢到角落里。

医生可以把他的错误埋葬在地下了事，但程序员不能。我们必须要学会测试与改错，并且把测试与改错工作做好。

7.1 对测试的理解

测试的道理并不深奥，计算机专业人员都应该明白。但就是这么简单的事，计算机专业的博士们也未必都已经理解。

有一天，一位比我聪明，编程比我快，学习能力比我强的计算机专业博士生恭恭敬敬地请我坐好，并且史无前例地削了苹果请我吃，为的是向我请教“软件工程”问题。你必定以为这位仁兄好学之极。非也，我和他同事三年来从未探讨过“软件工程”问题。只因为他明天要去应聘，参加面试，生怕被人问倒，就央我当晚为他恶补一把“软件工程”。他还特地问我“什么是白盒测试和黑盒测试？应该由谁来执行？”（有公司曾经这样面试应聘者）当我解释完测试的道理时，他叹了一口气说：“这些玩意儿我读大学十年来都没搞过，怎么能讲得出道理来。唉，就去碰碰运气吧。”我有“兔死狐悲”的感觉。我们这一群博士生三年来尽干些自欺欺人的事，到毕业时学问既不深也不博。个个意志消沉，老气横秋。长此以往，总有一天招聘会的大门前将贴出标语“博士与狗不得入内”。

以下是关于测试的几个重要观念。

7.1.1 测试的目的

测试的目的是为了发现尽可能多的缺陷。

这里缺陷是一种泛称，它可以指功能的错误，也可以指性能低下，易用性差等等。测试总是先假设程序中存在缺陷，再通过执行程序来发现并最终改正缺陷。理解测试的目的是个很重要的意识问题。如果说测试的目的是为了说明程序中没有缺陷，那么测试人员就会向这个目标靠拢，因而下意识地选用一些不易暴露错误的测试示例。这样的测试是虚假的。

目前高校的科技成果鉴定会普遍存在类似的虚假现象。我在读硕士时就亲身经历过这样的事。我们的项目是研究集成电路制造过程中的成品率问题。当时国内大多数工厂的集成电路成品率只有百分之几，我编写的示例程序可以将集成电路的成品率优化到98%。示例效果是如此的好，以致一位评委（某厂的总工程师）不无讽刺地说：“采用你们的成果，我们可要发大财了。”这个项目就轻易地通过了鉴定，并且不久后获得了电子工业部科技进步二等奖。这就象在考试时通过作弊取得了好成绩而被表扬。我那时尚且纯真，羞愧之余，不禁对高校科研成果的水平和真实性大失所望（现在我已不再失望，因为很少抱希望）。

一个成功的测试示例在于发现了至今尚未发现的缺陷。

测试并不仅是个技术问题，更是个职业道德问题。

7.1.2 测试的心理要求

测试主要是由人而不是由机器执行，这就不免与心理因素相关。为了测试的真实性，对测试的心理要求是“无情”。这似乎太残酷了。开发人员不能很好地测试自己的程序是因为做不到无情。而测试人员如果做到了无情却会引起开发人员的愤怒，遭人白眼。

尽管已经明白了测试的目的是为了发现尽可能多的缺陷，但当测试人员真的发现了一堆缺陷时，却不可乐颠颠地跑去恭喜那个倒霉的开发者，否则会打架的。

7.1.3 测试的真理

测试只能证明缺陷存在，而不能证明缺陷不存在。

这个真理告诉我们，对于一个复杂的系统而言，无论采取什么样的测试手段都不能证明缺陷已经不复存在。“彻底地测试”只是一种理想。在实践中，测试要考虑时间、费用等限制，不允许无休止地测试。

7.1.4 测试与质量的关系

测试有助于提高软件的质量，但是提高软件的质量不能依赖于测试。测试与质量的关系很象在考试中“检查”与“成绩”的关系。

学习好的学生，在考试时通过认真检查能减少因疏忽而造成的答题错误，从而“提高”了考试成绩（取得他本来就该得的好成绩）。

而学习差的学生，他原本就不会做题目，无论检查多么细心，也不能提高成绩。

所以说，软件的高质量是设计出来的，而不是靠测试修补出来的。

7.2 测试人员的选择

测试需要开发人员参与吗？

测试需要独立的测试小组吗？

测试需要用户参与吗？

让我们先看一看 Microsoft 公司关于测试的经验教训，再回答上述问题。

7.2.1 Microsoft 公司的经验教训

在 80 年代初期，Microsoft 公司的许多软件产品出现了“Bug”。比如，在 1981 年与 IBM PC 机一起推出的 BASIC 软件，用户在用“.1”（或者其他数字）除以 10 时，就会出错。在 FORTRAN 软件中也存在破坏数据的“Bug”。由此激起了许多采用 Microsoft 操作系统的 PC 厂商的极大不满，而且很多个人用户也纷纷投诉。

Microsoft 公司的经理们发觉很有必要引进更好的内部测试与质量控制方法。但是遭到很多程序设计师甚至一些高级经理的坚决反对，他们固执地认为在高校学生、秘书或者外界合作人士的协助下，开发人员可以自己测试产品。在 1984 年推出 Mac 机的 Multiplan（电子表格软件）之前，Microsoft 曾特地请 Arthur Anderson 咨询公司进行测试。但是外界公司一般没有能力执行全面的软件测试。结果，一种相当厉害的破坏数据的“Bug”迫使 Microsoft 公司为它的 2 万多名用户免费提供更新版本，代价是每个版本 10 美元，一共化了 20 万美元，可谓损失惨重。

痛定思痛后，Microsoft 公司的经理们得出一个结论：如果再不成立独立的测试部门，软件产品就不可能达到更高的质量标准。IBM 和其它有着成功的软件开发历史的公司便是效法的榜样。但 Microsoft 公司并不照搬 IBM 的经验，而是有选择地采用了一些看起来比较先进的方法，如独立的测试小组，自动测试以及为关键性的构件进行代码复查等。Microsoft 公司的一位开发部门主管戴夫·穆尔回忆说：“我们清楚不能再让开发部门自己测试了。我们需要有一个单独的小组来设计测试，运行测试，并把测试信息反馈给开发部门。这是一个伟大的转折点。”

但是有了独立的测试小组后，并不等于万事大吉了。自从 Microsoft 公司在 1984 年与 1986 年之间扩大了测试小组后，开发人员开始“变懒”了。他们把代码扔在一边等着测试，忘了唯有开发人员自己才能阻止错误的发生、防患于未来。此时，Microsoft 公司历史上第二次大灾难降临了。原定于 1986 年 7 月发行的 Mac 机的 Word 3.0，千呼万唤方于 1987 年 2 月问世。这套软件竟然有 700 多处错误，有的错误可以破坏数据甚至摧毁程序。一下子就使 Microsoft 名声扫地。公司不得不为用户免费提供升级版本，费用超过了 100 万美元。[Cusumano 1995]

7.2.2 测试人员的分工

从 Microsoft 公司的教训中可知，公司内部对产品的测试（称为 α 测试），需要开发人员与独立的测试小组共同参与。开发人员应该执行“白盒”测试，即测试源程序的逻辑结构以及实现细节（“白盒”是指看得见程序的内部结构）。而独立测试小组应该执行“黑盒”测试，即按照规格说明来测试程序是否符合要求（“黑盒”是指看不见程序的内

部结构)。比如在测试一个模块时，“白盒”测试方法要对模块的所有代码进行单步跟踪测试。而“黑盒”测试方法只需测试模块的接口是否符合要求，它关心程序的外部表现而不是内部的实现细节。

小型的软件公司可能没有条件设立独立的测试小组，也有可能测试小组人员不多而忙不过来。这时，可以让开发小组的成员相互测试对方的程序。

这里要强调的是，α测试不能依赖于开发人员或者测试小组中的任意一方，必须是双方共同参与。“白盒测试”必须由开发者自己执行，因为别的测试人员无法了解到程序的内部实现细节。而“黑盒测试”必须由独立的测试人员执行，因为开发者难以做到客观、公正。开发者在测试自己的程序时存在一些弊病：

（1）开发者对自己的程序印象深刻，并总以为是正确的（自信是应该的)。倘若在设计时就存在理解错误，或因不良的编程习惯而流下隐患，那么他本人很难发现这类错误。

（2)开发者对程序的功能、接口十分熟悉，他自己几乎不可能因为使用不当而引发错误，这与大众用户的情况不太相似，所以自己测试程序难以具备典型性。

（3）程序设计有如艺术设计，开发者总是喜欢欣赏程序的成功之处，而不愿看到失败之处。让开发者去做“蓄意破坏”的测试，就象杀自己的孩子一样难以接受。即便开发者非常诚实，但“珍爱程序”的心理让他在测试时不知不觉地带入了虚假成份。

软件产品正式发行前，在公司外部邀请一些用户对产品进行测试，称为β测试。β测试的涉及面最广，最能反映用户的真实愿望，但花费的时间最长，不好控制。一般地，软件公司与β测试人员之间有一种互利的协议。即β测试人员无偿地为软件公司作测试，定期递交测试报告，提出批评与建议。而软件公司将向β测试人员免费赠送或者以很大的优惠价格发行软件的正式版本。

7.3 测试的主要内容与常用方法

有一次文学考试，问高尔基是哪国人。一考生乐极而吟：“尔基啊尔基，你若不姓高，我怎知你是中国人。”这是一种瞎猜法。如果这种方法用于软件测试，人累死也测不出什么结果来。

不论是对软件的模块还是整个系统，总有共同的内容要测试，如正确性测试，容错性测试，性能与效率测试，易用性测试，文档测试等。“白盒测试”是指开发人员从程序内部对上述内容进行测试，而“黑盒测试”是指独立的测试人员从程序外部对上述内容进行测试。很多软件工程教材讲述了各种各样的测试方法并例举了不少示例[Pressman 1997] [Sommerville 1992] [杨文龙 1997]。本节简明地讲述常用的测试方法及其道理。

7.3.1 正确性测试

正确性测试又称功能测试，它检查软件的功能是否符合规格说明。由于正确性是软件最重要的质量因素，所以其测试也最重要。

基本的方法是构造一些合理输入，检查是否得到期望的输出。这是一种枚举方法。倘若枚举空间是无限的，那可惨了，还不如回家种土豆有盼头。测试人员一定要设法减少枚举的次数，否则没好日子过。关键在于寻找等价区间，因为在等价区间中，只需用

任意值测试一次即可。等价区间的概念可表述如下：

记（A, B）是命题 f(x) 的一个等价区间，在（A, B）中任意取 x1 进行测试。

如果 f (x1) 错误，那么 f (x) 在整个（A, B）区间都将出错。

如果 f (x1) 正确，那么 f (x) 在整个（A, B）区间都将正确。

上述测试方法称为等价测试，来源于人们的直觉与经验，可令测试事半功倍。

还有一种有效的测试方法是边界值测试。即采用定义域或者等价区间的边界值进行测试。因为程序员容易疏忽边界情况，程序也“喜欢”在边界值处出错。

例如测试 $f(x)=\sqrt{x}$ 的一段程序。凭直觉等价区间应是（0, 1）和（1, +∞）。可取 x=0.5 以及 x=2.0 进行等价测试。再取 x=0 以及 x=1 进行边界值测试。

有一些复杂的程序，我们难以凭直觉与经验找到等价区间和边界值，这时枚举测试就相当有难度。

在用“白盒测试”方式进行正确性测试时，有个额外的好处：如果测试发现了错误，测试者（开发人员）马上就能修改错误。越早改正错误，付出的代价就越低。所以大多数软件公司要求程序员在写完程序时，马上执行基于单步跟踪的“白盒测试”。

7.3.2 容错性测试

容错性测试是检查软件在异常条件下的行为。容错性好的软件能确保系统不发生无法意料的事故。

比较温柔的容错性测试通常构造一些不合理的输入来引诱软件出错，例如：

（1）输入错误的数据类型，如“猴”年“马”月。

（2）输入定义域之外的数值，上海人常说的“十三点”也算一种。

粗暴一些的容错性测试俗称“大猩猩”测试，除了不能拳打脚踢嘴咬，什么招术都可以使出来。这里我举不出例子，因为我没有对程序粗暴过，并且这辈子也不打算学会粗暴。

7.3.3 性能与效率测试

性能与效率测试主要是测试软件的运行速度和对资源的利用率。有时人们关心测试的“绝对值”，如数据送输速率是每秒多少比特。有时人们关心测试的“相对值”，如某个软件比另一个软件快多少倍。

在获取测试的“绝对值”时，我们要充分考虑并记录运行环境对测试的影响。例如计算机主频，总线结构和外部设备都可能影响软件的运行速度；若与多个计算机共享资源，软件运行可能慢得像蜗牛爬行。

在获取测试的“相对值”时，我们要确保被测试的几个软件运行于完全一致的环境中。硬件环境的一致性比较容易做到（用同一台计算机即可）。但软件环境的因素较多，除了操作系统，程序设计语言和编译系统对软件的性能也会产生较大的影响。如果是比较几个算法的性能，就要求编程语言和编译器也完全一致。

性能与效率测试中很重要的一项是极限测试，因为很多软件系统会在极限测试中崩溃。例如，连续不停地向服务器发请求，测试服务器是否会陷入死锁状态不能自拔；给程序输入特别大的数据，看看它是否吃得消。

7.3.4 易用性测试

易用性测试没有一个量化的指标，主观性较强。调查表明，当用户不理解软件中的某个特性时，大多数人首先会向同事、朋友请教。要是再不起作用，就向产品支持部门打电话。只有 30%的用户会查阅用户手册。[Cusumano 1995]

一般认为，如果用户不翻阅手册就能使用软件，那么表明这个软件具有较好的易用性。

7.3.5 文档测试

文档测试主要检查文档的正确性、完备性和可理解性。好多人甚至不知道文档是软件的一个组成部分。

正确性是指不要把软件的功能和操作写错，也不允许文档内容前后矛盾。

完备性是指文档不可以“虎头蛇尾”，更不许漏掉关键内容。有些学生在证明数学题时，喜欢用“显然”两字蒙混过关。文档中很多内容对开发者可能是“显然”的，但对用户而言不见得都是“显然”的。

文档不可以写成散文、诗歌或者侦探、言情小说，要让大众用户看得懂，能理解。

很多程序员能编写出好程序，却写不出清晰的文档。不要说自己以前语文学得差，现在已没救了，找借口不是办法。没有人天生就能写出好程序，都是练出来的。同理，若第一次写不好文档，就多写几次文档，慢慢地就会写出好文档来。我上大学前不会说普通话，不会写作文，现在我极能说会写，当个秘书或书记已绰绰有余。

7.4 改 错

在软件测试时如果发现了错误，必须请程序员改错，否则测试工作就白干了。

改错是个大悲大喜的过程，一天之内可以让人在悲伤的低谷和喜悦的颠峰之间跌荡起伏。如果改过上万个程序错误，那么少男少女们不必经历失恋的挫折也能变得成熟起来。

我从大三开始真正接受改错的磨练，已记不清楚多少次汗流浃背、湿透板凳。改不了错误时，恨不得撞墙。改了错误时，比女孩子朝我笑笑还开心。

在做本科毕业设计时，一天夜里，一哥们流窜到我的实验室，哈不拢嘴地对我嚷嚷：“你知道什么叫茅塞顿开吗？”

我象白痴似的摇摇头。

他说：“今天我化了十几个小时没能干掉一个错误，刚才我去了厕所五分钟，一切都解决了。”

他还用那没洗过的手拉我，一定要请我吃“肉夹馍”。那得意劲儿仿佛同时谈了两个女朋友。

在本节，我要替程序员们总结关于改错的几点思想方法：

（1）要有勇气。东北有个林场工人，工作勤奋，一人能干几个人的活。前三十年是伐树劳模，受到周总理的接见。忽有一天醒悟过来，觉得自己太对不起森林，决心补救错误。

后三十年成了植树劳模，受到朱总理的接见。此大勇也。

程序中的错误只有开发者自己才能找出并改掉。如果因畏惧而拖延，会让你终日心情不定，食无味，睡不香。所以长痛不如短痛，要集中精力对付错误。

（2）不可蛮干。都说急中生智，我不信。我认为大多数人着急了就会蛮干，早把“智”丢到脑后。不仅人如此，动物也如此。

我们经常看到，蜜蜂或者苍蝇想从玻璃窗中飞出，它们会顶着玻璃折腾几个小时，却不晓得从旁边轻轻松松地飞走。我原以为蜜蜂和苍蝇长得太小，视野有限，以致看不见近在咫尺的逃生之窗，所以只好蛮干。可是有一天夜里，有只麻雀飞进我的房间，它的逃生方式竟然与蜜蜂一模一样。我用灯光照着那扇打开的窗户为其引路，并向它打手势，对它说话，均无济于事。它是到天亮后才飞走的，这一宿我俩都没息好。

（3）找出错误的根源。有人问阿凡提：“我肚子痛，应该用什么药？”阿凡提说：“应该用眼药水，因为你眼睛不好，吃了脏东西才肚子痛。”

我们应该运用归纳、推理等方法尽早确定错误的根源。

（4）在改错之后一定要马上进行重新测试，以免引入新的错误。有人在马路上捡到钱包后得意忘形，不料自己却被汽车撞倒。改了一个程序错误固然是喜事，但要防止乐极生悲。更加严格的要求是：不论原有程序是否绝对正确，只要对此程序作过改动（哪怕是微不足道的），都要进行重新测试。

7.5 小 结

优秀的程序员敢于声称自己的代码没有错误，这种自信让人羡慕不已。一个错误自身也许很微小，但是程序存在错误这件事很严重。能否做好测试与改错工作，思想认识和办事态度是最关键的。

程序员应该把测试当成份内之事，不要依赖于外界的“黑盒测试”。“黑盒测试”就象通过提问题来判断一个人是否是个疯子，但无法知道他为什么成了疯子。让程序员对所有的代码执行单步跟踪测试听起来很费时间，但习惯了你就感觉不到有什么不方便。单步跟踪测试将使你以后的日子更轻松。

程序出了错误一定要改错，但是“编写优质无错”的程序才是根本的解决之道。在此，我竭力建议大家阅读 Steve Maguire 著的《Writing Clean Code : Microsoft Techniques for Developing Bug-free C Programs》（有中文译本，[Maguire 1993]）。我深受此书的教诲，获益非浅。

第八章　维护与再生工程

编程大师曾说：“哪怕程序只有三行长，总有一天你也不得不对它维护。”

很多软件产品不是一次性的买卖。比如在电信、金融等领域，有些软件系统要用十几年，对软件进行维护是必不可少的。8.1 节将介绍“软件维护的常识”，对维护活动进行分类，并解释为什么维护比较困难。

软件公司的经理们没有哪一个喜欢被维护的费用吓一跳，但软件维护的代价通常是高昂的。7.2 节将说明影响维护代价的一些主要技术因素与非技术因素。

如果希望提高已有软件的质量并且提高商业竞争力，却又无法靠维护来实现，只好对已有软件进行全部或者部分的改造，这种活动叫再生工程（Reengineering）。7.3 节将解释什么是再生工程，并论述再生工程的三种类型：重构（Restructure）、逆向工程（Reverse Engineering）和前向工程（Forward Engineering）。

8.1 软件维护的常识

对软件而言，“维护”是个不太直观的术语，因为软件产品在重复使用时不会被磨损，并不需要进行像对车辆或电器那样的维护。软件维护是人们对既丰富多彩又会令人心酸的活动的统称。其中丰富多彩的活动是指那些反映客观世界变化、能使软件系统更加完善的修改和扩充工作。令人心酸的活动是指那些永无修止、并且改了旧错却引起新错让人欲哭无泪的工作。

一些学者将软件维护划分为主要的三类：纠错性维护（Corrective maintenance）、适应性维护（Adaptive maintenance）和完善性维护（Perfective maintenance）：

（1）纠错性维护。由于前期的测试不可能揭露软件系统中所有替在的错误，用户在使用软件时仍将会遇到错误，诊断和改正这些错误的过程称为纠错性维护。

（2）适应性维护。由于新的硬件设备不断推出，操作系统和编译系统也不断地升级，为了使软件能适应新的环境而引起的程序修改和扩充活动称为适应性维护。

（3）完善性维护。在软件的正常使用过程中，用户还会不断提出新的需求。为了满足用户新的需求而增加软件功能的活动称为完善性维护。

Lientz 和 Swanson 调查发现（1980 年），完善性维护约占 65%，适应性维护约占 18%，纠错性维护约占 17%[Sommerville 1992]。上述调查已是 20 年前的事了，我们不必太关心具体的比例，心里有数即可。

以下一些因素将导致维护工作变得困难：

（1）软件人员经常流动，当需要对某些程序进行维护时，可能已找不到原来的开发人员。只好让新手去“攻读”那些程序。

（2）人们一般难以读懂他人的程序。在勉强接受这类任务时，心里不免嘀咕：“我又不是他肚子里的虫子，怎么知道他如何编程。”

（3）当没有文档或者文档很差时，你简直不知道如何下手。

（4）很多程序在设计时没有考虑到将来要改动，程序之间相互交织，触一而牵百。即使有很好的文档，你也不敢轻举妄动，否则你有可能陷进错误堆里。
（5）如果软件发行了多个版本，要追踪软件的演化非常困难。
（6）维护将会产生不良的副作用，不论是修改代码、数据或文档，都有可能产生新的错误。
（7）维护工作毫无吸引力。高水平的程序员自然不愿主动去做，而公司也舍不得让高水平的程序员去做。带着低沉情绪的低水平的程序员只会把维护工作搞得一塌糊涂。

8.2 维护的代价及其主要因素

软件维护是既破财又费神的工作。看得见的代价是那些为了维护而投入的人力与财力。而看不见的维护代价则更加高昂，我们称之为“机会成本”，即为了得到某种东西所必须放弃的东西[Mankiw 1999]。把很多程序员和其它资源用于维护工作，必然会耽误新产品的开发甚至会丧失机遇，这种代价是无法估量的。

影响维护代价的非技术因素主要有：
（1）应用域的复杂性。如果应用域问题已被很好地理解，需求分析工作比较完善，那么维护代价就较低。反之维护代价就较高。
（2）开发人员的稳定性。如果某些程序的开发者还在，让他们对自己的程序进行维护，那么代价就较低。如果原来的开发者已经不在，只好让新手来维护陌生的程序，那么代价就较高。
（3）软件的生命期。越是早期的程序越难维护，你很难想像十年前的程序是多么的落后（设计思想与开发工具都落后）。一般地，软件的生命期越长，维护代价就越高。生命期越短，维护代价就越低。
（4）商业操作模式变化对软件的影响。比如财务软件，对财务制度的变化很敏感。财务制度一变动，财务软件就必须修改。一般地，商业操作模式变化越频繁，相应软件的维护代价就越高。

影响维护代价的技术因素主要有：
（1）软件对运行环境的依赖性。由于硬件以及操作系统更新很快，使得对运行环境依赖性很强的应用软件也要不停地更新，维护代价就高。
（2）编程语言。虽然低级语言比高级语言具有更好的运行速度，但是低级语言比高级语言难以理解。用高级语言编写的程序比用低级语言编写的程序的维护代价要低得多（并且生产率高得多）。一般地，商业应用软件大多采用高级语言。比如，开发一套 Windows 环境下的信息管理系统，用户大多采用 Visual Basic、Delphi 或 Power Builder 来编程，用 Visual C++的就少些，没有人会采用汇编语言。
（3）编程风格。良好的编程风格意味着良好的可理解性，可以降低维护的代价。
（4）测试与改错工作。如果测试与改错工作做得好，后期的维护代价就能降低。反之维护代价就升高。
（5）文档的质量。清晰、正确和完备的文档能降低维护的代价。低质量的文档将增加维护的代价（错误百出的文档还不如没有文档）。

8.3 再生工程

再生工程主要出于如下愿望：（1）在商业上要提高产品的竞争力；（2）在技术上要提高产品的质量。但这种愿望无法靠软件的维护来实现，因为：（1）软件的可维护性可能极差，实在不值得去做；（2）即使软件的可维护性比较好，但也只是治表不治本。再生工程干脆对已有软件进行全部或部分的改造，赋予软件新的活力。

在对待一个不良之徒时，可以进行思想教育并给予他关心和帮助，这种方式类似于“软件维护”；也可以把他关进监狱，送去劳改，这种方式相当于软件的“再生工程”；如果此人坏透顶了，就毙掉算了。

再生工程与维护的共同之处是没有抛弃原有的软件。如果把维护比作“修修补补”，那么再生工程就算是“痛改前非”。再生工程并不见得一定比维护的代价要高，但再生工程在将来获取的利益却要比通过维护得到的多。

再生工程主要有三种类型：重构、逆向工程和前向工程。

8.3.1 重构

重构一般是指通过修改代码或数据以使软件符合新的要求。重构通常并不推翻原有软件的体系结构，主要是改造一些模块和数据结构。重构的一些好处如下：

（1）使软件的质量更高，或使软件顺应新的潮流（标准）。

（2）使软件的后续（升级）版本的生产率更高。

（3）降低后期的维护代价。

要注意的是，在代码重构和数据重构之后，一定要重构相应的文档。

8.3.2 逆向工程

逆向工程来源于硬件世界。硬件厂商总想弄到竞争对手产品的设计和制造“奥秘”。但是又得不到现成的档案，只好拆卸对手的产品并进行分析，企图从中获取有价值的东西。我的很多同学从事集成电路设计工作，他们经常解剖国外的集成电路，甚至不作分析就原封不动地复制该电路的版图，然后投入生产，并美其名曰“反向设计”（Reverse Design）。

软件的逆向工程在道理上与硬件的相似。但在很多时候，软件的逆向工程并不是针对竞争对手的，而是针对自己公司多年前的产品。期望从老产品中提取系统设计、需求说明等有价值的信息。

8.3.3 前向工程

前向工程也称预防性维护，由 Miller 倡导。他把这个术语解释成“为了明天的需要，把今天的方法应用到昨天的系统上”。[Pressman 1999]

乍看起来，主动去改造一个目前运行得正常的软件系统简直就是“惹事生非”。但是软件技术发展如此迅速，与其等待一个有价值的产品逐渐老死，还不如主动去更新，以获取更大的收益。其道理就同打预防性针一样。所以，预防性维护是“吃小亏占大便宜”

的事。

8.4 小 结

大学科研机构里的软件维护工作恐怕是做得最差的了。几乎每一批新的研究生都会把毕业生留下的软件臭骂一通，然后全部推到重做。到他毕业该走时，就轮到别人骂他的工作了。如此轮回，最终没有什么成果留下。

如果希望软件系统能活下，必须要对它进行维护。如果希望软件系统有效益，则必须设法降低维护的代价。

参 考 文 献

[Abrash 1998] Michael Abrash, 图形程序开发人员指南（前导工作室译），机械工业出版社，1998

[Cline 1995] Marshall P. Cline, Greg A. Lomow, C++ FAQs, Addison-Wesley, 1995

[Comer 1996] Douglas E. Comer, David L. Stevens, Internetworking With TCP/IP, Vol III, Prentice-Hall,1996

[Cusumano 1995] Michael A. Cusumano, Richard W. Selby, 微软的秘密（程化 等译），北京大学出版社，1995

[Jacobson 1997] Ivar Jacobson, Martin Griss, Software Reuse, 世界图书出版公司，1997

[James 1999] Geoffery James, 编程之道（郭海 等译），清华大学出版社，1999

[Kruglinski 1999] David J. Kruglinski, Scot Wingo, Programming Visual C++, 北京希望电子出版社, 1999

[林锐 1996] 林锐，蔡文立，微机科学可视化系统设计，西安电子科技大学出版社，1996

[林锐 1997] 林锐，戴玉宏，图形用户界面设计与技术，西安电子科技大学出版社，1997

[林锐 2000] 林锐，支持协同工作的交互式三维图形软件开发系统与可视化平台，浙江大学博士论文，2000

[Maguire 1993] Steve Maguire, Writing Clean Code（姜静波 等译），电子工业出版社，1993

[Mankiw 1999] N. Gregory Mankiw, 经济学原理（梁小明译），北京大学出版社，1999

[Norman 1996] Ronald J. Norman, Object-Oriented Systems Analysis And Design, Prentice-Hall, 1996

[Pressman 1997] Roger S. Pressman, Software Engineering: A Practitioner's Approach (Fourth Edition), McGraw-Hill, 1997

[Rogerson 1999] Dale Rogerson, COM 技术内幕（杨秀章 译），清华大学出版社，1999

[Shaffer 1999] Clifford A. Shaffer, 数据结构与算法分析（张铭，刘晓丹译），电子工业出版社，1999

[Sommerville 1992] Ian Sommerville, Software Engineering, Addison-Wesley, 1992

[Summit 1996] Steve Summit, C Programming FAQs, Addison-Wesley, 1996

[杨文龙 1997] 杨文龙，姚淑珍，吴云，软件工程，电子工业出版社，1997

[Shaw 1996] Mary Shaw, David Garlan, Software Architecture, Prentice-Hall, 1996

[Tanenbaum 1998] Andrew S. Tanenbaum ,计算机网络（第三版，熊桂喜等译），清华大学出版社，1998