

第 8 章 剩下的就是态度问题

本书中讨论的方法可以用来检查错误和防止错误,但是这些技术并不能保证肯定可以写出无错代码,就象一个熟练的球队不可能是常胜军一样。重要的是养成好的习惯和正确的态度。

如果一个球队成天在嘴上讨论如何训练,这个球队可能有取胜的机会吗?如果这个球队的队员不断地因为工资低而牢骚满腹,或时刻担心被换下场或裁减掉,那又会怎么样呢?虽然这些问题与球赛没有直接关系,但是却影响了球员水平的发挥。

同样读音可以使用本书的所有建议,但是,如果你持疑虑的态度或者使用错误的编码习惯,那么要写出无错的代码将是很困难的。因此,你要有必胜的信心和良好的习惯,同样,你同级的同事如果没有必胜信心和良好习惯也会遇到同样的问题。

因此在本章中将指出一些编写无错代码的主要障碍。只要能意识到这些障碍,改正就很容易了。

错误不出现,我还有一招

当向程序员询问有关他们修改错误的情况时,有多少次听到他们这样的回答:“唉呀!错误消失了。”多年以前,我就曾经向我的第一个经理说过这样的话,当时,我们正在研制 Apple II 数据库产品,经理问我若已经设法找到错误项目能否就此收尾,我说:“唉呀!错误消失了。”经理停顿了片刻,然后邀请我到他的办公室坐坐。

“你说‘错误消失了’,是什么意思?”

“哎呀!你知道,我一步步地仔细查看了错误报告。错误没再出现。”

经理在椅子上向后仰了一下问:“你认为错误到哪儿去了?”

“我不知道。”我说,“我想它已被改正了吧。”

“你并不知道谁改的,是吧?”

“是的,我是不知道。”我坦诚地回答。

“好,那你不认为你应该查明到底真正发生了什么吗?”他说。“毕竟你是在和计算机打交道,错误不会自我改正。”

然后,经理进一步解释说,错误消失有三个原因:一是错误报告不对;二是错误已被别的程序员改正了;三是这个错误依然存在但没有表现出来。也就是说,作为一个专业程序员,其职责之一就是要确定错误的消失究竟属于以上三种情况中的哪一种,从而采取相应的行动,但是决不能因为错误不出现就简单地忽略了它,就万事大吉了。

在我第一次听到这个忠告的时候,也就是在 CP/M 和 Apple II 的时代,这个忠告很有价值。实际上,在这之前的几十年里,它就很有价值,而且至今它仍然很有价值。但是,直到我自己成为项目负责人,并且发现程序员普遍乐于接受测试员搞错了或有某个程序员已经为其排除了这个错误,这时我才认识到这个忠告多么有意义。

错误消失经常是因为程序员和测试员使用了不同的程序版本。如果在程序员使用的代码

中错误没有出现，就采用测试员使用的程序版本，如果错误仍未出现，就可通知测试组。但是，如果错误确实出现了，就要追踪到它早些的源程序版本，并决定如何修改它，然后再查看一下为什么在当前的源程序版本中，错误会不见了。通常错误仍然存在，只是环境有了更改从而掩盖了错误。无论什么原因，为了采取适当的步骤来改正错误，必须弄明白为什么错误消失了。

错误几乎不会“消失”

浪费精力？

当我要求程序员在老版本源程序上寻找所报告的错误时，他们经常要发牢骚，这样做似乎象是浪费时间。要是你也这么认为的话，你要明白这样做并不是要你恢复老版本源程序，而不过要你查看一下这些源程序，为你提供查错的良机，而且这也是找到错误最有效的方法。

即使你发现错误已被改正了，那些老版本源代码中将错误分离出来也是值得的。是将错误以“改正了”终结好呢？还是给错误标以“不会再产生了”并送还给测试组好呢？测试人员将怎样做呢？他们肯定不会认为这个错误已经更正了，他们只有两种选择，一种是花时间来试图提出另一组能再产生错误的用例；另一种是丢下这个错误，将其标以“不会再产生了”并希望错误已被改正。与在老版本源代码中找到了错误并以“改正了”而终结比较起来。后两种选择都不好。

及时纠正，事半功倍

我第一次参加 Excel 小组的时候，我们把所有的错误改正工作都推迟到项目的最后。这并不是说有人用枪顶着我们的脊骨说：“直到所有的特征都实现了再去改正错误”，但总有保持进度和实现特征的压力，而在修改错误方面却一点压力也没有。我曾经说过：“除非错误使系统瘫痪了或使测试组停工，否则别急着更改它，完成进度要求的各特征之后，我们有的是时间来修改错误。”简而言之，改正错误没放在优先地位。

我相信现在的 Microsoft 程序员听到上面所讲的肯定感到很逆耳，因为项目不再以这种方式研制了。这种方法存在的问题太多，最坏的是不能预言产品什么时候能够完成。你怎样估计修改 1742 个错误的时间？当然，不仅仅有 1742 个错误需要修改，因为在程序员修改旧错误时又会引起新错误。更密切相关的是，修改一个错误可能暴露其它的潜在错误，由于第一个错误的障碍，测试组未能发现这些潜在错误。

但这还不是唯一的问题。

由于没有改正错误就完成了所要求的特征，产品看上去比它实际进展情况要提前了许多。公司的重要人物测试使用内部发行版本，发现除了一些偶然的错误之外，产品工作得很好，他们很惊奇，只用了六个月的开发时间就几乎完成了一个最终产品。他们看不到存储空间溢出错误，或某些从未试用过的特征错，他们只知道代码“各特征齐全”，基本上可以工作。

到最后用几个月的时间来修改错误也往往士气不振。程序员喜欢编程序而不愿意改错，

但是在每个项目的最后，有好几个月的时间，他们除了改错无事可作。由于开发组以外的每个人都明显地知道代码已接近完成，因此改错经常具有很大的压力。

这不是自找吗？

然而，自打 Macintosh Excel 1.03 开始到撤消 —— 未宣布名字的窗口产品（由于失控的错误表造成的）为止，Microsoft 一直运行带有错误的产品，这就迫使 Microsoft 认真研究怎样开发产品。得到的结论并不使人感到惊奇：

- 不要通过把改正错误移置产品开发周期的最后阶段来节省时间。修改一年前写的代码比修改几天前写的代码更难，实际上这是浪费时间。
- “一次性”地修改错误会带来许多问题，因为发现错误越早，重复这个错误的可能性就越小。
- 错误是一种负反馈，程序开发倒是快了，却使程序员疏于检查。如果规定只有把错误全部改正之后才能增加新特征的话，那么在整个产品开发期间都可以避免程序员的疏漏，他们将忙于修改错误。反之，如果允许程序员略过错误，那就使管理失控。
- 若把错误数保持在近乎于 0 的数量上，就可以很容易地预言产品的完成时间。只需要估算一下完成 32 个特征所需的时间，而不需要估算完成 32 个特征加上改正 1742 个错误所需的时间。更好的是，你总能处于可随时交出已开发特征的有利地位。

以上这些观点并不只适用于 Microsoft 开发，而且适用于任何软件开发。如果你在发现错误时没有及时纠正，那么 Microsoft 的坏经验就是你的反面教材。你可以从自己的艰难经历中或从别人的沉痛教训中学到很多东西。到底该怎样做呢？

马上修改错误，不要推迟到最后

憨医救人

在安东尼·罗宾斯的小说《唤醒巨人》(Awaken the Giant Within) 中讲了一位医生的故事。一天，有个医生走到一条汹涌的河边，她突然听到落水者的呼救声。她环顾了四周，发现没有人去救，于是，她就跳入水中，朝着落水者游去。她将落水者救上岸，做口对口的人工呼吸，这个人刚一恢复呼吸，又从河里传来了另外两个落水者的求救声。她又一次跳入水中，把这两个人救上岸，正当她安顿好这两个人时，医生又听到另外四个落水者的求救声，然后她又听到另外八个落水者的求救声……问题是医生只忙于救人，抽不出时间到上游查明是谁把人们扔到水中。

象这个医生一样，程序员也经常忙于“治愈”错误而没有停下来判断一下是什么原因引起了这些错误。例如象上一章我们讲过的函数 `strFromUns`，由于它迫使程序员使用未受保护的数据而导致错误。但是错误总是出现在调用 `strFromUns` 的函数内，而不是在 `strFromUns` 本身。因此，你认为应该修改错误的真正根源 `strFromUns` 呢，还是修改出了错的调用 `strFromUns` 的这个函数呢？

在我把 Windows Excel 的一个特征移植到 Macintosh Excel 时（当时，它们仍是两个独

立源代码段) 也出现了类似的问题。在移植了一个特征之后, 我开始测试代码并发现有个函数得到了未预料到的 NULL 空指针。我检查了代码, 但是错误是在这个函数所调用的函数 (传出 NULL) 中呢, 还是在这个函数本身 (没有处理 NULL) 呢? 因此我找到原来的程序员并问他说明了情况。他马上编辑了该函数并说: “哦, 这个函数无处理 NULL 指针能力。”然后, 就在我站在边上看着的时候他通过插入如下代码而改正了错误, 当指针为 NULL 时 “快跳” 出来

```
if(pb == NULL)
    return(FALSE);
```

我提醒他是否这个函数不应该得到空指针 NULL, 错误在调用函数中而不在这个函数中。他回答道: “我清楚这些代码, 这样做就可以改正错误了。”但是我觉得这种解决方法好象只改正了错误的症状而没有改正错误的原因, 于是我返回我的办公室花了 10 分钟时间来追踪 NULL 空指针的来源。发现空指针 NULL 不仅是这个错误的真正根源, 而且也是另外两个已知错误的原因。

还有几次当我追踪到错误的根源时, 经常这样认为: “等一下, 修改这个函数可能是不对的。如果是这个函数出错的话, 函数在另外的地方也应该出问题呀, 可是它没有出问题呀。”我肯定你能猜出为什么函数在另外的地方能够工作, 它之所以工作是因为某个程序员已经局部性地修改了这个较为通常的错误。

修改错误要治本, 不要治表

你有无事生非的代码吗?

“只要没有破坏作用, 怎么改也行。”这似乎是某些程序员的口号。不管代码是否很好地工作, 某些程序员总要强行在代码上留下自己的痕迹。如果你曾与那些喜欢将整个文件重新格式化以适合他们口味的程序员工作过的话, 你肯定会理解我所讲的内容。尽管大多数程序员对 “清理” 代码非常谨慎, 但是, 似乎所有程序员都不同程度地做过这件事情。

清理代码的问题在于程序员总不把改进的代码作为新代码处理。例如, 有些程序员在浏览文件时, 看到下面所示的代码, 他们就把与 0 比较的测试改为与 ‘\0’ 作比较的测试 (其他程序员也可能想全部删掉测试)。

```
char* strcpy(char* pchTo, char* pchFrom)
{
    char* pchStart = pchTo;
    while( (*pchTo++ = *pchFrom++) != 0 )
        NULL;
    Return(pchStart);
}
```

把 0 改为一个空字符的问题是很容易将 ‘\0’ 错误地键入为 ‘0’, 但是有多少程序员愿

意在做了这样简单的改变之后再测试一下 strcpy 呢？这提醒我们：当你做了如此简单的更改之后，你是否也象对待新编写的代码那样进行完全的测试呢？如果没有，那么这些没有必要的代码更改就会有引入错误的危险。

你可能认为只要修改后的代码仍能通过编译，那么这些更改就不算错。例如，改变局部变量的名字怎么会引出问题呢？可是它确实能引起问题。我曾经跟踪一个错误直到一个函数，这个函数具有一个局部变量名 hPrint，它与具有同样名字的全局变量冲突。由于这个函数在不久前工作还很正常，我查看了老版本的源程序，来看一下到底当前版本改变了什么并验证我的修改是否会重新引入以前有过的错误。我就发现了清理代码问题。老版本中有个局部变量 hPrint1，但是没有 hPrint2 或 hPrint3 来解释名字中‘1’的意义。删除‘1’的程序员肯定认为 hPrint1 是人为的冗余并将其清理为 hPrint，从而引起了名字冲突，导致了错误。

为了避免犯上面的错误，要经常提醒自己：与我一起工作的程序员并非一些笨蛋。当你发现一些有明显错误或显然没有必要的代码时，上面的警句将提醒你小心从事。看上去明显有问题的代码，以后你就会发现，它这样写可能有很好但又不明显的原因。我曾经见过一段荒谬的代码，它唯一的目的是当编译程序代码生成有了错误才工作（这是极罕见的——译者注），如果你清理了这段代码那么就会引入错误。当然这样的代码应该有注释来解释一下它要实现的功能，但是，不是所有的程序员都想得那么周到。

因此，如果你发现了象下面这样的代码：

```
char chGetNext(void)
{
    int ch;           /* ch“必须”是 int 类型 */
    ch = getchar();
    return(chRemapChar(ch));
}
```

不要急于删除显然“没有必要”的 ch，清理成这样的函数：

```
char chGetNext(void)
{
    return( chRemapChar(getchar()) );
}
```

这样整理后，如果 chRemapChar 是宏，它会多次求参数的值，从而可能引入了错误。因此，保持“没有必要的”局部变量，避免没有必要的错误。

除非关系产品的成败，否则不要整理代码

把“冷门”特征打入冷宫

避免清理代码只是编写无错代码普遍原则的特例，这个普遍原则就是：如果没有必要就

不要编写（或修改）代码。这个建议看上去似乎很奇怪，但是如果你经常提出疑问：“这个特征对产品的成败有什么重要作用？”从而删掉这些特征，那么你就不会陷入困境。

某些特征加到产品中并没有价值，但是它之所以存在仅仅是为了填满特征集；另外一些特征的存在是因为大公司买主要求这些特征；还有一些特征能够存在是因为竞争者的产品具有这些特征，评审人就决定将这些特征纳入特征表中。如果有很好的市场和产品规划小组，那就不应该加入这些没有必要的特征。但是，作为一名程序员，不仅会随大流采用这些特征，甚至还可能是某些没有必要特征的发源人。

你曾经听过程序员说这样的话吗？“如果 WordSmasher 可以做……，那将是个大‘冷门’。”这个所谓“冷门”特征是因为它能提高产品的质量呢，还是因为它的实现在技术上具有挑战性？如果这个特征能提高产品的质量，那么应将该特征推迟到程序的下个版本实现，到那时将对其进行合理的评价并制定相应的进度表。如果这个特征仅仅是一种技术上的挑战，那么否决它。我的建议并不是要抑制创造力，而是要遏制那些不必要的特征以及相关错误的发展。

有时，技术上具有挑战性的特征能提高产品的质量，有时就不能。请小心选择。

不要实现没有战略意义的特征

不存在免费午餐

“自由”特征是那些多余性错误的另一个来源。表面上，自由特征似乎是值得的，因为这只需要很少甚至不需要做任何努力就能跳过已有的设计。怎样才能比这更好呢？具有自由特征会带来很大的问题，尽管它们对产品的成败几乎从未起过任何关键的作用。正如我在上一节讲的，你应该把任何非关键特征看成是错误的来源。程序员向程序内增加自由特征是因为他们可以增加而不是因为他们必须增加。如果它不需要你付出任何代价，那为什么不增加一个特征呢？

啊！但是这是谬论。对于程序员来说，增加自由特征可能不费事，但是对于特征来讲，它不仅仅增多了代码，还必须有人为该特征写文档，还必须有人来测试它。不要忘记还必须有人来修改该特征可能出现的错误。

当我听到某个程序员说某特征是自由的，我就知道他没有花时间来考虑纳入该特征的真实代价。

不设自由特征

灵活性滋生错误

避免错误的另一条策略是排除设计中没有必要的灵活性。这个原则贯穿本书的始终。例如，在第一章，我使用了选择编译警告以避免出现冗余的和有风险的 C 语言惯用语。在第二章，我把 ASSERT 定义为一条语句来防止在表达式中错误地使用宏。在第三章，我使用了断

言来捕获传递给 FreeMemory 的 NULL 指针，即使使用 NULL 指针调用 free 函数是合法的，我也这么做了。…… 我可以列出每一章的例子。

灵活设计的问题在于，设计越灵活，就越难发觉错误。还记得我在第 5 章中针对 realloc 所强调的那几点吗？你几乎可以扔掉 realloc 的任何输入集，可它仍将继续执行，但是它可能并没按你所希望的去执行。更糟糕的是，由于函数很灵活，因此不能插入有意义的断言验证输入的有效性。但是，如果把 realloc 分成为扩展、收缩、分配、释放存储块四个专门函数，则确认函数变元就要容易得多了。

除了过度灵活的函数之外，还应该时刻警惕着过度灵活的特征。由于灵活的特征可能产生一些没有预料到的“合法”情况，你可能会认为这些情况不需要测试甚至认为这就是合法的，因此，灵活特征同样很棘手。

例如，当我为 Apple 的 Excel 和新的 Macintosh II 机器的 Excel 增加彩色支持程序时，我要从 Windows Excel 上移植一段代码，该代码允许用户指定显示在电子表格格子内的正文颜色。例如，向一个格子内增加颜色，你应该选择已有的格子形式，如下所示（将 1234.5678 打印为\$1,234.57）：

```
$#,##0.00
```

并且在前面加上颜色声明。为了显示蓝色，用户就需要将上面的形式改为：

```
[blue]$#,##0.00
```

如果用户写了[red]，那么数据以红色显示，如此等等。

Excel 的产品说明非常清楚，颜色说明应放在数据形式的开始处，但是当我移植了这个特征打开始测试代码时，我发现下面的所有形式都工作

```
$#,##0.00[blue]
```

```
$#,##[blue]0.00
```

```
$(blue)#,##0.00
```

你可以将[blue]放在任何地方。当我向原来的程序员询问这是个错误还是个特征时，他说颜色声明可以放在任意位置“仅仅是脱离了语法分析循环。”他不认为允许一点点额外的灵活性是个错误，当时我也那么认为，于是代码就那样保留下来了。然而，回顾一下，我们不应该允许这个额外的灵活性。

不久测试组发现了六个微妙的错误，最终所有这些错误都起因于格式的语法分析程序，因为它没有料想到会发现彩色说明处于格式中间的情况。但是我们没有通过删除没有必要的灵活性来改正这个错误（这需要增加一个简单的 if 语句），而只是改正了这些特定的错误，即改正了错误的症状，从而保留了任何人已不再需要的灵活性。时至今日，Excel 仍允许将彩色说明置于你所希望的任何位置。

因此在你实现特征时要记住：不要使它们具有没有必要的灵活性，但是要容易使用。这两者是有差别的。

不允许没有必要的灵活性

移植的代码也是新代码

在把 Windows Excel 代码移植到 Macintosh Excel 的过程中，我得到了这样一条教训，人们对这种移植过来的代码，总想少做些检查。毕竟这些代码是在原来的产品中测试过的。我在把移植代码交给测试组之前就应捕获 Excel 数字格式代码中的全部错误，但是我没有这么做。我只是把代码拷贝到 Macintosh Excel，做了一些为把这些代码连接到项目中所必须的修改，然后临时测试了一下代码来验证它已被正确地连接起来了。我没有全面测试特征本身，因为我认为这已经测试过了。这是失策的，特别是在当时的情况下，Windows Excel 本身也正处于开发阶段，这就更是失策。那正是 Microsoft 小组把修改错误推迟到产品周期的最后阶段那个时期。

实际上，不管你是怎样实现特征的，是从头开始设计实现，还是依据某个已有代码来设计实现的，你都有责任排除要加入到项目的那些代码中所存在的错误。如果 Macintosh Excel 只具有与 Windows Excel 相同的错误这可以吗？当然不可以，因为这并不能减轻这些错误的严重性。我一犯懒它就出现了。

“试一试”是个忌讳词

你也许说过多次类似这样的话：“我不知道该怎样来……”，而别的程序员回答你：“你是否试过……？”几乎可以在每个新成立的小组中听到类似这样的对话。某程序员邮出一条消息问：“我怎样才能把光标隐藏起来？”第一个人说：“试着把光标移到屏幕之外去”，另一个人建议：“把光标标志置为 0，光标像素就不可见了”，第三个人或许会说：“光标只是个位映象，因此想办法把它的宽度和高度都置为零”。试、试、试……

我承认这是个荒唐的例子，但肯定你听到过类似的对话。通常在被建议“试一试”的所有方案中，可能都不是可以采纳的合适方案。当别人告诉你试一试某件事情时，只是告诉你一个考虑过的猜测并非问题的答案。

试一试各种方案有什么错？假如试验的东西已被系统明确定义的话，那么没有任何错误。但事情常常不是这样，当程序员开始尝试某方案时，他往往会远离他们所了解的系统，进入到饥不择食地寻求解答的境界，这种解很可能有无意识的副作用，将来还要更改。程序员还有个坏习惯，就是有意识地从自由存储区读取，你此对此有何看法呢？free 肯定没有定义自由存储区中的内容是什么，但有些程序员由于某种原因感到他们需要引用自由存储区，他们一试，偏巧成功了，于是他们只好依赖 free 来实施这种行为。

因此注意听取程序员向你提出的建议，如：“你可以试一试……”等，你就会发现大多数建议利用了未定义的或病态定义的副作用。如果程序员提建议时知道怎样求解，他们就不会向你说“试一试”。例如，他们肯定会告诉你“使用 SetCursorState(INVISIBLE)系统调用。”

在找到正确的解法之前，不要一味地“试”，要花时间寻求正确的解

少试多读

几年来,在 Microsoft 的 Macintosh 程序员都能接收到 Macintosh 新闻小组在其内部网络上的一些只读编辑物。这些编辑物很有趣,但它并不十分有用,常常不能回答所提出的问题。总有一些程序员提出那些答案已清楚写在“苹果公司内用 Macintosh 手册”中的问题,但是,程序员得到的回答除了在手册中清楚地给出解的以外,往往是笼统的解决方法。幸运的是,总有几个 Macintosh 的内部专家能给出明确的答案,如:“参看 Macintosh 内部手册第 4 章,第 32 页,它上面说你应……”。

我的观点是:如果你发现你自己正在测试问题的可能解时,停下来,拿出你的手册仔细阅读。这可没有玩代码那么有趣,也没有向别人询问怎么试那么容易,但你将学到许多有关操作系统的知识和如何在它上面编程的知识。

“神圣的”进度表

当要实现相当大的特征时,某些程序员不得不花上两个星期趴在键盘上编写代码,从不着急测试他的程序。另一些程序员则在实现了十来个特征之后才停下来检查他的程序。如果这种方法能使程序员彻底全面地测试他们的代码,那么这种方法就没有什么错误。但是,这可能吗?

请考虑一下这种情况:一个程序员要用 5 天实现 5 个特征。这个程序员有两种选择:一种是实现一个特征就测试一个特征,一个一个地进行;另一种是五个五个地进行。你认为实际上哪一种方法能产生强健的代码呢?几年来,我考察了这两种编码风格。绝大多数情况下,边编写代码边测试代码的程序员较少出错。我甚至可以告诉你为什么会是这样的。

假设程序员把 5 天时间全部用来实现 5 个特征,但是随后他意识到在进度表中他没有剩下太多的时间来全面测试这些代码了。你认为程序员会用额外的一天或两天来全面测试这些代码吗?或者玩一玩代码,验证一下代码似乎是工作正常的,然后就转到下个特征呢?当然,答案要取决于程序员和工作环境。但是,带来的问题却是放弃进度计划,还是减少测试,如果放弃进度计划,大多数公司都会表示不满,而减少测试,则会失去负反馈,程序员可能更赞成保持进度计划。

即使程序员单个而不是成批地编写和测试特征,也常常由于进度原因,程序员仍要减少测试。但是当程序员成批实现特征时效果更加明显。在一批特征中只要有一个困准特征就会占用所有特征的测试时间。

使用进度表的缺点是程序员会给速度比测试还高的优先级,本质上就是进度获得了比写正确代码还高的优先级。我的经验是,如果程序员按照进度的时间来编写某个特征的代码,那么即使减少测试,他也要按进度“完成”该特征。他会想到:“如果在代码中有某些未知的错误,测试组会通知我的。”

为了避免这一陷阱,尽量编写和测试小块代码,不要用进度作为借口跳过测试这一步。

**尽量编写和测试小块代码。
即使测试代码会影响进度，也要坚持测试代码**

名实难符

第 5 章曾解释过，`getchar` 的名字经常使得程序员认为，该函数返回一个字符，它实际上返回一个 `int`。同样，程序员经常认为，测试组会测试他们的代码，这是他们的工作，除此之外，他们还干什么呢？其实这种看法是错误的。无论程序员们怎么认为，测试组的存在并非是为了测试程序员写的代码，他们是为了保护公司，最终使用户不受低劣产品的损害。

如果和房屋建筑过程比较一下，就很容易理解测试的作用。在房屋建筑中，建设者建房，检查员检查它。但是检查员并不“测试”这些房屋：电气工程师决不会亲自去安装房屋的电线，也决不会在不接通电源，不测试保险盒，不用万用表检查每个出线口之前就交付线路。这个电气工程师决不会认为：“我不必做这些测试，如果有问题，检查员会通知我的。”有这种想法的电气工程师会很快发现他们难以找到工作。

就象上述房屋检查员一样，程序测试员不负责测试程序的主要理由是，他们不具备必要的工具和技巧，虽然有例外，但这是一条原则。尽管和计算机界的说法不一样，但是测试员测试用户的代码不可能要比用户自己测试的更好。测试员能够加入断言来捕获有问题的数据流吗？测试员能够对存储管理程序进行像第 3 章中那样的子系统测试吗？测试员能够使用调试程序来逐次逐条指令地通过代码，以检查每条代码路径是否都按照设计的要求工作吗？而现状是，尽管程序员测试他们的代码要比测试员有效得多，但是他们却不做，这就是因为计算机界有这些说法。

但是不要误解我的意思，测试组在开发过程中起着重要的作用，但决不是程序员所想象的那种作用。他们在检验产品时，寻找使程序失败的缺陷，证实产品是否与以前推出的产品不兼容，提醒发展部门改进产品性能，利用产品在实际使用中的情况来证实产品的这些特征是非常有用的。所有上述的都跟测试无关，仅仅是在产品中注入质量。

因此，请记住第 2 章中所说的，如果要持续不断地写出无错代码，就必须抓住要害并不受其控制，不要依靠测试组来发现错误，因为这不是他们的工作。

测试代码的责任不在测试员身上，而是程序员自己的责任

重复劳动

如果程序员负有测试代码的责任，那么就自然出现了这个问题：“程序员和测试员在做重复的努力吗？”可能是。但是再问一遍，当然不是。程序员测试代码，是从里向外测试，而测试员则是从外向里测试。

例如，程序员测试代码时，总是从测试每个函数开始，逐次逐条指令（或行）地通过各条代码路径，验证代码和数据流，逐步向外移动来证实函数能够在子系统中与其它函数一道

正常操作，最后程序员利用单元测试来验证各个独立的子系统之间能够正确地相互配合。通过单元测试，还能检测内部数据结构的状态。

另一方面，测试员却把代码作为一个黑盒子，从程序的各个输入处进行测试以寻找错误，测试员也可能利用回归测试来证实所有报告的错误都已排除。然后，测试员逐步向里推进，利用代码覆盖工具，来检查在全局测试中执行了多少内部代码，随之获得的信息产生新的测试，来执行未接触到的代码。

这是使用两个不同“算法”测试程序的例子。之所以这样，是因为程序员强调的是代码而测试员强调的是特征，两者从不同的方位考虑问题，这就增加了发现未知错误的机会。

遭白眼的测试员

读者是否注意到，当测试组发现一个错误后，有多少程序员发出宽慰的叹息，他们会说：“嗨！我很高兴程序在交付之前测试出了这个错误。”然而另有一些程序员，在测试员报告他们程序中的错误特别是指出一段代码中的多个错误时，他们却对此忿恨不满。我曾经见到过这种程序员怒发冲冠，也曾听到有些项目负责人说为什么测试员让我不得安宁，这是测试错误，因为我们已经删掉了这个数据。”有一次，我还阻止过一位项目负责人和一位测试负责人之间的拳打脚踢，原因是项目负责人已经处于推迟交付产品的巨大压力之下，而测试组还在继续报告错误，这使他很不安。

这听起来是否是很愚蠢？的确是很愚蠢。在我们没有注意到这个产品是在非难和压力下交付之前，容易觉得这是多么荒唐可笑。但是设身处地想一想，如果你被错误包围着，交付期已过数月，便很容易认为这些测试员的确是坏家伙。

每当我看见程序员向测试人员发火时，我总是把他们拉到一旁并问他们：你们为什么要测试人员为程序员所犯的错误负责呢，和测试员发火毫无道理，他们仅仅是报信者。

每当测试员向你报告你的代码中有某个错误时，你最先的反应是震惊和不相信，你本来就没想到测试员会在你的代码中发现错误；你的第二个反应应该是表示感谢，因为测试员帮助你避免交付错误。

不要责怪测试员发现了你的错误

不存在荒谬的错误

有时你会听到程序员抱怨某个错误太荒谬，或者抱怨某个测试员经常报告一些愚蠢的错误。如果你听到这样的抱怨时，制止并提醒他，测试员并不判断错误的严重性，也不说这些错误是否值得排除。测试员必须报告所有的错误，不管是愚蠢的还是不愚蠢的，尽管测试员知道，有些愚蠢的错误可能是某个严重问题的副作用。

但是真正的问题是，程序员在测试这个代码时为什么没有捕获这些错误呢？即使这些错误很轻微并且不值得排除，但找出错误的根源也是非常重要的，以避免将来出现类似的错误。

一个错误可能很轻微。但是它的存在本身就很严重。

建立自己的优先顺序

如果往前翻几页重温一下本书的主要观点，你就会惊奇地发现，其中有些观点似乎是相互矛盾的。然而当你仔细思考以后，你可能又不那么认为了。总之，程序员要经常和编写快速代码和紧凑代码这样相互矛盾的目标打交道。

因此问题是，当面临两个可能的实现时究竟选哪一个？可以肯定要在快速算法和小巧算法之间做出选择是比较困难的，但是，要在快速算法与可维护算法之间、或者在小巧但有风险的算法与较大但易于测试的算法之间作出选择时，你会做出怎样的选择呢？肯定有些程序员会不假思索地回答这些问题，但也有一些程序员不能确定到底选择哪一种。如果几星期之后再问他们同样的问题，他们将会给出不同的答案。

程序员之所以不能确定这种互易的理由是，因为他们除了知道象大小或速度这些非常普通的优先次序之外，不知道他们自己的优先顺序是什么。但是在程序设计中如果没有明确的优先顺序，就象是盲人骑瞎马一样，在每个转弯处都要停下来并问问自己：“现在我该怎么办？”这时你往往做出错误的选择。

然而有些程序员，他们清楚地知道自己的优先顺序，但是由于他们的优先顺序不正确或相抵触，在关键问题上他们没有认真思考因此不断地作出错误的选择。例如，许多有经验的程序员仍受 70 年代末期提倡的优先顺序的影响，那时存储空间很少，微机运行很慢，为了写有用的程序，必须要用可维护性来换取空间和速度。但是现在，程序越来越复杂，而 RAM 的容量却越来越大，计算机运行速度也不断加快，以至于即使用很差的算法，大多数任务也都能按时完成。因此现在的优先顺序不同了，不再用可维护性来换取空间和速度，否则就会得到在速度上并不明显地快但又不可维护的程序。仍还有一些程序员把程序大小和速度奉为神灵，把它们看作是产品成败的关键。由于这些程序员的有限顺序已经过时，因此他们一直在做着错误的实现选择。

因此，只要你还没有考虑过你的优先顺序，那么你就要坐下来为你自己（如果你是项目负责人，就为你的小组）认真地建立优先级列表，从而使你能够在完成项目目标的过程中不断地作出最佳选择。注意我说的是“项目目标”。你的优先级列表不应该反映你想要做的，而应反映你应该做的。例如，某些程序员可能把“个人表达方式”列为最高优先级，这样对程序员或者对产品有利吗？这些程序员接不接受命名标准？同不同意使用{}的定位风格，还是自搞一套呢？

应当指出，没有“正确”的方法来确定你的优先级序列，但是所选定的优先顺序将决定代码的风格和质量。让我们看一看约克和吉尔两个程序员的优先级列表：

约克的优先级列表

正确性
全局效率
大小
局部效率
个人方便性

吉尔的优先级列表

正确性
可测试性
全局效率
可维护性 / 明晰性
一致性

可维护性 / 明晰性

个人表达方式

可测试性

一致性

大小

局部效率

个人表达方式

个人方便性

这些优先顺序将怎样影响约克和吉尔的代码呢？两人都首先集中在写出正确代码上，这仅仅是他们在优先级排列上唯一的相同之处。可以看出，约克非常重视大小和速度，对编写清晰代码关心一般，几乎不怎么考虑测试代码是否容易。

而吉尔则把更多的注意力放在编写正确的代码上，只是在大小和速度危及到产品是否成功时，才把它们作为考虑对象。吉尔认为，除非能够很容易地测试代码，否则就无法验证代码是否正确，因此吉尔把可测试性放在优先级排列顺序列表中很高的位置。

你认为这两个程序员哪个更可能：

- 使得所选择的编译程序都能自动捕获错误并报警？虽然为了使用安全环境可能需要额外做点工作。
- 使用断言和子系统作调试检查？
- 走查每一条代码路径从微观验证所有刚写的新代码？
- 用安全函数界面取代有风险的函数界面？虽然在每个调用点可能要额外声称 1~2 条以上的指令。
- 使用可移植类型，以及当用到移位的情况下而用除法（例如使用/4 代替>>2）？
- 避免使用第 7 章中的效率技巧？

这是个充满问题的表吗？我不这样认为。我问你：“你认为吉尔和约克谁会读这本书并按照书中的建议取做？”吉尔和约克谁会去阅读《程序设计风格要素》或者其他指导性书籍，并按照书中的建议去做呢？

读者应该注意到，由于约克的优先顺序安排，他会把注意力集中在对产品不利的代码上，他回在如何使每一行代码尽量快和小上浪费时间，而对产品长期健全却很少考虑。吉尔却相反，根据她的优先顺序，她把注意力集中在产品上，而不是代码上，除非证明（或显然）确实需要考虑大小和速度，否则她不考虑大小和速度。

现在想一想，代码和产品哪个对你的公司更重要？因此你的优先顺序应该是怎样的？

建立自己优先级列表并坚持之

说出道道

你是否看过别人写的代码，并奇怪他们为什么这样写呢？你是否就此代码问过他们，而后他们说：“哎呀，我不知道我为什么这样写，我猜我当时感觉到这样写正确吧。”

我经常评审代码，寻找帮助程序员改进技术的方法。我发现“哎呀，我不知道”这样的回答相当普遍，我还发现作出这种回答的程序员没有建立明确的优先顺序，他们的决定似乎

具有随意性。相反地，具有明确优先顺序的程序员，精确地知道他们为什么选择这个实现，并且当问及他为什么这样实现时，他能够说出道道。

小结

本章还没有提到一个很重要的观点，这就是 你必须养成经常询问怎样编写代码的习惯。本书就是长期坚持询问一些简单问题所得的结果。

- 我怎样才能自动检测出错误？
- 我怎样才能防止错误？
- 这种想法和习惯是帮助我编写无错代码呢还是妨碍了我编写无错代码？

本章的所有观点都是询问最后一个问题所产生的结果。审视一下自己的观念很重要，这些观念就反映了个人考虑问题的优先次序。如果你认为测试组的存在是为了测试你的代码，那么在编写无错代码方面你就会继续有麻烦，因为你的观念在某种程度上告诉你，在测试方面马虎点是可以的。如果你没有在观念上想着要编写无错代码，那你怎么可能会试着编写无错代码呢？

如果你想编写无错代码，就应该清除妨碍你达到这一目标的观念，清除的方法就是反问一下自己，自己的观念对达到目标是有益的还是有害的。

要点：

- 错误既不会自己产生，也不会自己改正。如果你得到了一个错误报告，但这个错误不再出现了。不要假设测试员发生了幻觉，而努力查找错误，甚至要恢复程序的老版本。
- 不能“以后”再修改错误。这是许多产品被取消的共同教训。如果你发现错误的时候就及时地更正了错误，那你的项目就不会遭受毁灭性的命运。当你的项目总是保持近似于 0 个错误时，怎么可能会有一系列的错误呢？
- 当你跟踪查到一个错误时，总要问一下自己，这个错误是否会是一个大错误的症状。当然，修改一个刚刚追踪到的症状很容易，但是要努力找到真正的起因。
- 不要编写没有必要的代码。让你的竞争者去清理代码，去实现“冷门”但无价值的特征，去实现自由特征。让他们花大量的时间去修改由于这些无用代码所引起的所有没有必要的错误。
- 记住灵活与容易使用并不是一回事。在你设计函数和特征时，重点是使之容易使用；如果它们仅仅是灵活的，象 `realloc` 函数和 Excel 中的彩色格式特征那样，那么就没法使得代码更加有用；相反地，使得发现错误变得更困难了。
- 不要受“试一试”某个方案以达到预期结果的影响。相反，应把花在尝试方案上的时间用来寻找正确的解决方法。如果必要，与负责你操作系统的公司联系，这比提出一个在将来可能会出问题的古怪实现要好。
- 代码写得尽量小以便于全面测试。在测试中不要马虎。记住，如果你不测试你的代

码，就没有人会测试你的代码了。无论怎样，你也不要期望测试组为你测试代码。

- 最后，确定你们小组的优先级顺序，并且遵循这个顺序。如果你是约克，而项目需要吉尔，那么至少在工作方面你必须改变习惯。

课题：

说服你们程序设计组建立或采纳一个优先级列表。如果你们公司具有不同层次的人才（例如初级程序设计员，程序设计员，高级程序设计员，程序设计分析员），你可能要考虑不同的层次使用不同的优先级列表，为什么？