第 3 章　白盒测试方法

教学要求

- 掌握：白盒测试的基本概念以及相关方法。
- 理解：白盒测试工具 JUnit 的用法。
- 了解：白盒测试的必要性。

3.1　白盒测试方法

一般来说，测试任何产品有两种方法：第一种测试方法就是我们第 2 章提到的黑盒测试，这种测试方法是在已经知道了产品应该具有的功能前提下通过测试来检验每个功能是否都能正常使用。第二种测试方法是在知道产品内部工作过程的前提下通过测试来检验产品内部动作是否按照规格说明书的规定正常进行，这种方法称为白盒测试，又称为结构测试。

与黑盒测试相反，白盒测试法是把程序看成装在一个透明的盒子里，按照程序的内部的逻辑测试程序，检验程序中的每条通路是否能按预定要求正确工作。白盒测试方法的突出特点是基于被测程序的源代码，而不是软件的规格说明。和其他软件测试技术相比，白盒测试方法得到的测试用例能够做到以下几点：①保证模块中的所有独立路径至少被执行一次；②对所有逻辑值都会测试 TRUE 和 FALSE；③在上下边界及可操作范围内运行所有循环情况；④检查内部数据结构以确保其有效性。由此可知，白盒测试更容易发现软件故障。在软件测试过程中，单元测试大都采用白盒测试。常见的白盒测试方法有代码检查、逻辑覆盖、基本路径测试等。

3.1.1　代码检查

在介绍代码检查之前，需要弄清楚软件测试中的静态测试方法和动态测试方法。所谓静态测试是指在不运行被测试程序，通过其他手段，如检查、审查，达到检测目的。所谓动态测试是指通过运行和使用被测程序，发现软件故障，以达到检测的目的。以检查汽车为例，检查车的油漆、外观，打开车前盖查看里面部件是否有问题属于静态测试。发动汽车，听发动机的声音、通过驾驶检查车的部件是否有问题属于动态测试。

代码检查即静态白盒测试，在不执行程序的条件下仔细审查代码（可采用互查、走查等形式），从而找出软件故障的过程。根据经验表明，代码中 65%以上的缺陷可以通过代码检查发现出来。代码检查不仅使修复的时间和费用大幅度降低，而且黑盒测试人员还可以根据审查备注确定存在软件缺陷的特性范围。

那么，如何进行代码检查呢？一般来说，一个代码检查小组通常由四至五人组成，分别是本程序的编码人员、程序的设计人员、测试技术人员以及小组协调人等。需要注意的是，协调人的职责包括安排进程、分发材料；记录发现的所有错误；确保所有错误随后得到改正。协调人在代码检查中起主导作用。因此，协调人最好不要是程序的编码人员。

正式审查过程中有 4 个关键要素：

（1）确定问题。进行代码检查的目的就是要找出代码是否存在逻辑上的错误以及是否在代码中引入了没有在设计中指定的包。在代码检查中，参与人员必须树立正确的态度，如果程序员将代码检查视为对其个人的攻击，采取了防范的态度，那么检查过程就不会有效果。因此，软件中存在的错误应被看做是伴随着软件开发的艰难性所固有而不是编写程序的人员本身的弱点。

（2）遵守准则。为了使审查过程有条不紊的进行，在审查前就必须设定一套准则，其中包括审查地点，最好是不受外界干扰的环境；会议时间最好不超过 120 分钟；代码量以及检查代码的速度适中等。这样，审查才能保质保量的完成。

（3）提前准备。参与人员必须明确自己的职责和义务。经验表明，审查过程中找出的大部分问题是在准备期间发现的。

（4）编写审查报告。审查过程最终必须形成一个书面总结报告并及时提交，便于开发小组成员进行修改和改进。

通过正式审查不但可以及早发现软件缺陷，而且可以在讨论和交流中增进成员间的信任，为程序员之间交流经验相互学习提供平台。同时，还可以间接促进程序员更加认真仔细的编写和检查代码。

那么，代码检查应注意哪些可能存在的软件缺陷呢？首先必须对代码的规范性进行审查，如嵌套的 IF 是否正确的缩进；注释是否准确并有意义；是否使用有意义的标号。在代码检查中，有时出现编写的代码不符合某种标准和规范，虽然这些问题不影响代码正常运行，但是如果程序员能严格遵守一些语言编码标准，如电子电气工程学会（IEEE）提供的程序规范和最佳做法的文档，可以提高代码的可靠性、可移植性和易读性等。代码规范性审查有助于及早发现缺陷，帮助程序员养成良好的编程习惯。另外还要考虑以下几种类别的错误。

（1）数据的引用错误。主要包括是否引用了未初始化的变量；数组和字符串的下标是否为整数值且下标是否越界；变量是否被赋予了不同类型的值；是否为引用指针分配内存；一个数据结构是否在多个函数或者子程序中引用，在每一个引用中是否明确定义了结构等。

（2）数据类型错误。主要包括变量数据类型是否定义错误；变量的精度是否够；是否对不同数据类型进行比较或赋值等。

（3）数据声明错误。主要包括变量是否在声明的同时进行了初始化；是否正确初始化并与其类型一致；变量是否都赋予正确的长度、类型和存储类；变量名是否相似等。

（4）计算错误。主要包括计算时是否了解和考虑到编译器对类型或长度不一致的变量的转换规则；计算中是否使用了不同数据类型的变量；除数或模是否可能为零；变量的值是否超过有意义的范围；赋值的目的变量是否小于赋值表达式的值等。

（5）逻辑运算错误。主要包括表达式是否存在优先级错误；每一个逻辑表达式是否都正确地表达；逻辑计算是否如期进行；求值次序是否有疑问；逻辑表达式的操作数是否为逻辑值等。

（6）控制流程错误。主要包括程序中的语句组是否对应；程序、模块、子程序和循环能否终止；是否存在永远不停的循环；对于多分支语句，索引变量是否能超出可能的分支数目；是否存在“丢掉一个”错误，导致意外进入循环等。

（7）子程序参数错误。主要包括子程序接收的参数类型和大小与调用代码发送的是否匹配；如果子程序有多个入口点，引用的参数是否与当前入口点没有关系；常量是否当作形参传递，意外在子程序中改动；子程序是否更改了仅作为输入值的参数；每一个参数的单位是否与相应的形参匹配；如果存在全局变量，在所有引用子程序中是否有相似的定义和属性等。

（8）输入/输出错误。主要包括软件是否严格遵守外设读写数据的专用格式；软件是否处理外设未连接、不可用、或者读写过程中存储空间占满等情况；软件是否以预期的方式处理预计的错误；是否检查错误提示信息的准确性、正确性、语法和拼写等。

（9）其他错误。主要包括软件是否使用其他外语；是否处理扩展 ASCII 字符；是否需用统一编码取代 ASCII；程序编译是否产生“警告”或者“提示”信息；是否对外部接口采集的数据进行确认；标号和子程序是否符合代码的逻辑意思等。

3.1.2　覆盖测试

覆盖测试以程序内部的逻辑结构为基础设计测试用例，要求对被测程序的逻辑结构有清楚的了解。根据覆盖测试的目标不同，可分为：语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖、组合覆盖及路径覆盖。为了清楚地说明几种逻辑覆盖测试方法之间的不同，下面以一个小程序为例。

```
if((x>0)&&(y<0))   {
    z=z-(x+y);
}
if((x>2)||(z>0)) {
    z=z+5;
}
```

其中&&、||是逻辑运算符，3 个输入参数是 x、y、z。其对应的程序流程图如图 3-1 所示（a、b、c、d、e 为控制流上的若干程序点）。

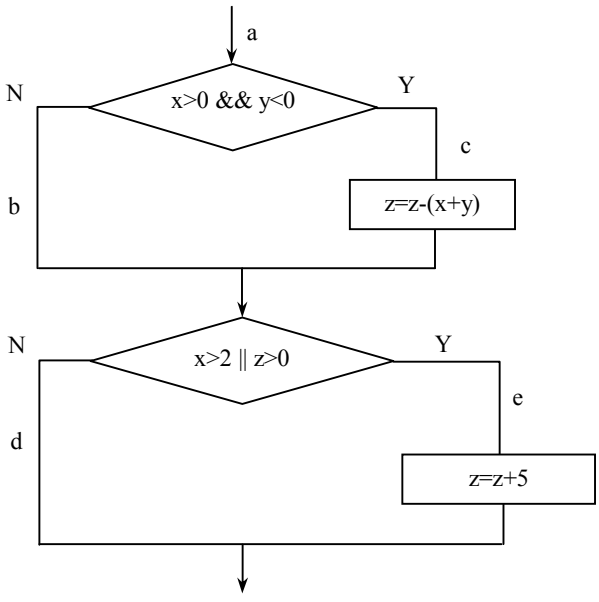


图 3-1　被测程序流程图

1. 语句覆盖

所谓语句覆盖是指设计若干个测试用例，使程序中的每个可执行语句至少被执行一次。对上述程序段，可以设计一个通过路径 ace 的测试路径。例如，当 x=4，y=-3，z=2 时（我

们不妨把它称为 test1），该程序段中的 4 个语句都得到执行，从而实现语句覆盖。如果所设计的测试用例 test2 为：x=4，y=3，z=2，则该程序执行的路径为 abe，不满足语句覆盖的条件。

根据上面的例子可以看出，语句覆盖可以保证程序段中的每个语句都能得到执行，似乎能够全面的检验每个语句。事实上，语句覆盖是一种不充分的检验方法，它是比较弱的逻辑覆盖原则。当程序段中两个判定的逻辑运算存在问题时，例如，如果第一个判断的运算符“&&”和第二个判断的运算符“||”被误写反了。这时使用测试用例 test1 仍然可以让程序按照 ace 的路径执行，却发现不了两个判断中的逻辑错误。除此之外，我们可以比较容易地找出虽然已经满足了语句覆盖，但依然存在错误的例子。如程序段中的 if(x>=0)误写成 if(x>0)，若给出的测试数据是大于 0 的数，则语句覆盖满足，程序得到执行，但是却没有发现该程序的错误。语句覆盖在测试程序时，对检查不可执行语句方面起到一定作用，但被测程序并不是语句间无序的堆积，语句之间存在着各种各样的内部联系。所以，语句覆盖并不能排除被测试程序中存在故障的风险。

2. 判定覆盖

所谓判定覆盖是指设计若干个测试用例，使得程序中的每个判定至少得到一次真值和假值，即判断中的真假分支至少均执行一次。判定覆盖又称为分支覆盖。

对上述程序段，可以设计如下两个测试用例：

test1：x=4，y=-3，z=2

test3：x=-1，y=1，z=-1

test1 执行了路径 ace，test3 执行了路径 abd，因此使得两个判断中的 4 个分支都得到检测，满足判定覆盖的条件。

还可以设计另外两个测试用例，同样满足判定覆盖条件。

test4：x=3，y=2，z=1

test5：x=1，y=-3，z=-5

test4 执行了路径 abe，test5 执行了路径 acd。

由以上两组测试用例可以看出，它们不仅满足了判定覆盖，同时也满足了语句覆盖。所以，判定覆盖要比语句覆盖更强一些，判定覆盖比语句覆盖要多几乎一倍的测试路径。但是，往往大部分的判定语句是由多个逻辑条件组合而成（如判定语句中包含 AND、OR、CASE），若仅仅判断其最终结果，而忽略每个条件的取值情况，必然会遗漏部分测试路径。例如，将第二个判断中的 z>0 误写成 z<0，测试用例 test3 依然可以执行路径 abe。满足判定覆盖却仍然无法确定判断内部条件的错误。另外，针对如 CASE 语句这种多出口判断，目前编程语言可以支持，因此判定覆盖准则可以扩充到多出口判断的情况。

3. 条件覆盖

所谓条件覆盖是指设计若干个测试用例，使得程序中每个判断中每个条件的可能值至少得到一次。因此，条件覆盖与判定覆盖相比增加了对符合判定情况的测试以及测试路径。

对上述程序段，第一个判断(x>0)&&(y<0)应考虑的情况为：

x>0 为真，记作 T1

x>0 为假，记作-T1

y<0 为真，记作 T2

y<0 为假，记作-T2

第二个判断(x>2)||(z>0)应考虑的情况为：

x>2 为真，记作 T3

x>2 为假，记作-T3

z>0 为真，记作 T4

z>0 为假，记作-T4

如表 3-1 所示，我们所设计的测试用例覆盖了以上 8 种情况，即 T1、T2、T3、T4、-T1、-T2、-T3、-T4。

表 3-1　条件覆盖（1）

测试用例	x,	y,	z	执行路径	覆盖条件
test1	4	-3	2	ace	T1　T2　T3　T4
test3	-1	1	-1	abd	-T1　-T2　-T3　-T4

通过上表可以看出，test1 和 test3 不仅覆盖了 4 个条件的 8 种情况，而且也同时覆盖了两个判断的 4 个分支。那么，条件覆盖和判定覆盖是否存在着一定的关系？如果满足条件覆盖是否就意味着一定能够满足判定覆盖呢？答案是否定的，条件覆盖并不能保证判定覆盖。条件覆盖只能保证每个条件至少有一次为真，而不考虑所有的判定结果，如表 3-2 所示。

表 3-2　条件覆盖（2）

测试用例	x,	y,	z	执行路径	覆盖条件
test4	3	2	1	abe	T1　-T2　T3　T4
test6	-1	-2	-1	abd	-T1　T2　-T3　-T4

因此，覆盖了所有条件的测试用例不一定覆盖所有的分支。为了解决这一矛盾，就引入了判定/条件覆盖，从而对条件和分支进行兼顾测试。

4. 判定/条件覆盖

所谓判定/条件覆盖是指设计若干个测试用例，使得判断中每个条件的所有（真或假）取值至少出现一次，并且每个判断的所有（真或假）判断结果也至少出现一次。

对上述程序段，当测试用例选取 test1 和 test3 就可以满足判定/条件覆盖，见表 3-1。由此可知，这两个测试数据也就是为了满足条件覆盖标准最初选取的两组数据，判定/条件覆盖并未考虑条件的组合情况，因此，有时判定/条件覆盖也并不比条件覆盖更强。

5. 组合覆盖

所谓组合覆盖是指设计若干个测试用例，使得每个判定条件的各种情况至少出现一次。

对上述程序段，两个判断中的 4 个条件可能出现的组合为：

①x>0, y<0，记作 T1,T2

②x>0, y>0，记作 T1,-T2

③x<0, y<0，记作-T1,T2

④x<0, y>0，记作-T1,-T2

⑤x>2, z>0，记作 T3,T4

⑥x>2, z<0，记作 T3,-T4

⑦x<2, z>0，记作-T3,T4

⑧x<2, z<0，记作-T3,-T4

下面我们将设计 4 个测试用例，满足判定/条件覆盖。如表 3-3 所示。

表 3-3 判定/条件覆盖

测试用例	x,	y,	z	执行路径	覆盖组合	覆盖条件
test1	4	-3	2	Ace	①⑤	T1 T2 T3 T4
test3	-1	1	-1	Abd	④⑧	-T1-T2 -T3 -T4
test7	-1	-2	1	Abe	③⑦	-T1 T2 -T3 T4
test8	3	1	-1	Abe	②⑥	T1 -T2 T3 -T4

以上测试用例覆盖了所有条件组合以及 4 个分支，一般来说，满足条件组合覆盖的测试用例一定满足判定覆盖、条件覆盖和判定/条件覆盖。但是，我们同时可以看到组合覆盖线性地增加了测试用例的数量，并且只执行了 3 条路径，路径 acd 被漏掉。在测试中，只有程序的每一条路径都经得起考验，这种程序才能被称为全面检验的合格品。

6. 路径覆盖

所谓路径覆盖是指设计若干个测试用例覆盖程序中所有的路径。根据图 3-1 可知，这些路径分别为：abd，abe，acd，ace。我们选择 test3、test7、test5 和 test1 四个测试用例即可满足路径覆盖，如表 3-4 所示。

表 3-4 路径覆盖

测试用例	x,	y,	z	执行路径
test1	4	-3	2	ace
test3	-1	1	-1	abd
test5	1	-3	-5	acd
test7	-1	-2	1	abe

对于这种程序比较简短，只需要 4 个测试用例，采用路径覆盖貌似还可以罗列出来。但在实际工作中，即使一个不太复杂的程序其路径有可能是一个巨大的数字。在有些情况下，一些执行路径是不可能被执行的。如果要完全覆盖路径是不太现实的，那么如何解决这一问题呢？我们要对覆盖路径数量进行压缩，例如遇到程序中的循环体只执行一次。当然，即使做到了路径覆盖，也不能确保被测试程序的正确性。因此，为了能够发现更多的软件故障，就需要针对不同的功能采用不同的测试方法。

3.1.3 路径测试

路径测试就是从一个程序的入口开始，执行所经历的各个语句的完整过程。从广义的角度讲，任何有关路径分析的测试都可以被称为路径测试。

路径测试法是在程序控制流图的基础上，通过分析控制构造的环路复杂性，导出基本可执行路径集合，从而设计测试用例的方法。路径测试方法包括以下几个步骤。

（1）画出程序的控制流图。所谓控制流图是指程序设计中，为了突出控制流的结构，对程序流程图进行简化后的图。它主要包括节点和控制线两个图形符号。比较常见语句的控制流图，如图 3-2 所示。

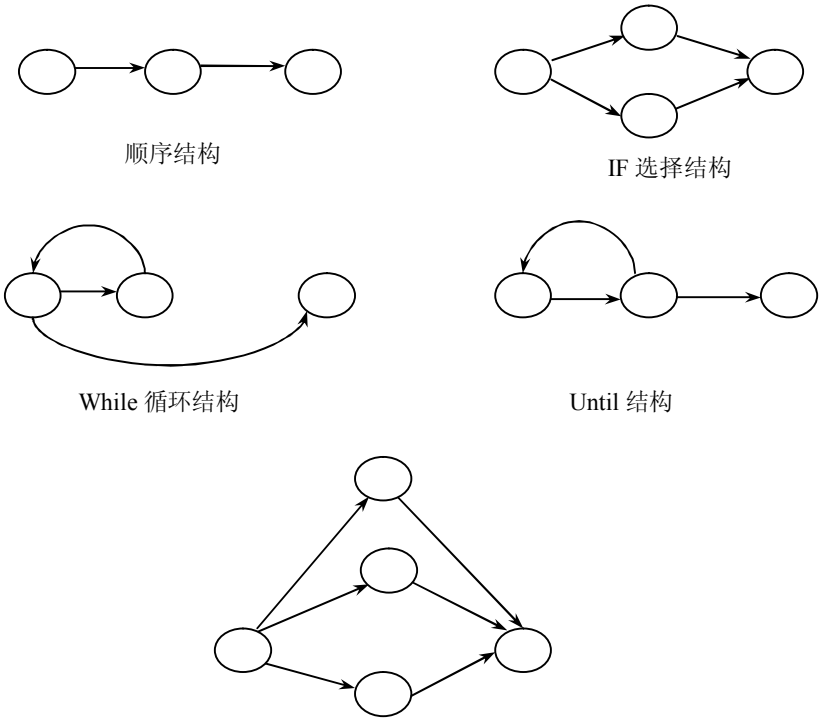


图 3-2　常见的几种控制流图

（2）程序环形复杂度：McCabe 复杂性度量。从程序的环路复杂性可导出程序基本路径集合中的独立路径条数，这是确定程序中每个可执行语句至少执行一次所必须的测试用例数目的上界。所谓独立路径就是从程序入口到出口的多次执行中，每次至少有一个语句是新的，未被重复。

（3）导出基本路径集，确定程序的独立路径。

（4）根据（3）中的独立路径，设计测试用例的输入数据和预期输出，确保基本路径集中的每一条路径的执行。

1. 程序路径表达

在对路径进行分析的时候，首先要解决的是确定每个路径以及路径数目。为了更加直观和形象地表达出每条路径对于某条路径，可采用弧序列或者节点序列的方式并引了两个运算：加和乘。

（1）弧 a 和弧 b 相加，表示为 a+b，它表明两条弧是“或”的关系，是并行的路段。

（2）弧 a 和弧 b 相乘，表示为 ab，它表明路径是先经历弧 a，接着再经历弧 b，弧 a 和弧 b 是先后相接的。

路径表达式运算满足以下规律。

加法交换律：a+b=b+a

加法结合律：a+(b+c)=(a+b)+c

加法幂运算：a+a=a

乘法结合律：a(bc)=(ab)c

分配律：a(b+c)=ab+ac　　(a+b)c=ac+bc　　(a+b)(c+d)=a(c+d)+b(c+d)

值得注意的是，路径表达式中乘法不满足交换律。

图 3-3 所示是两个简单程序的控制流图。

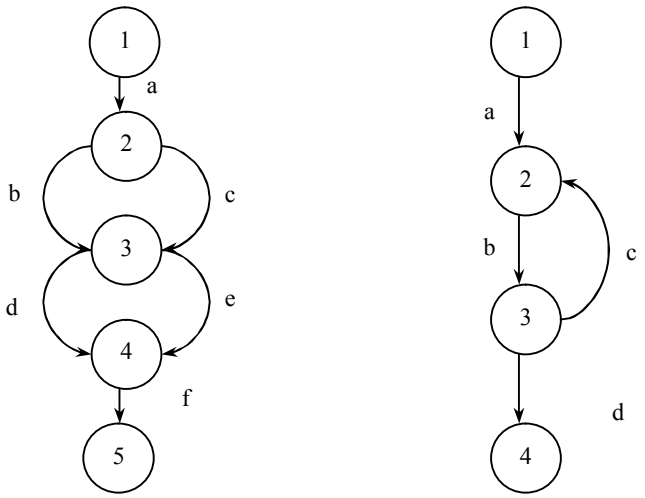


图 3-3　程序控制流图

图 3-3（a）共计 4 条路径：abdf、acef、abef、acdf。根据加和乘的法则可以知道，该程序的路径表达式为：abdf+acef+abef+acdf，即 a(b+c)(d+e)f。

图 3-3（b）是一个循环的控制流图，它的路径是随着循环体被执行的次数的不同而有所增加的。其路径表达式可写成：

abd+abcbd+abcbcbd+abcbcbcbd+…

其简化为：ab(1+cb+(cb)2+…)d

路径数的计算：

在路径表达式中，将所有弧均以数值 1 来代替，再进行表达式的相乘和相加运算，最后得到的数值即为该程序的路径数。

对于图 3-3（a）可知，将 a、b、c、d、e、f 以 1 代入表达式可得，该程序的路径数为：

$$N=1\times（1+1）\times（1+1）\times 1=4$$

对于图 3-3（b）可知，将 a、b、c、d 以 1 代入表达式，假设只考虑循环次数小于 3 的情况，则该程序的路径数为：

$$N=1\times（1+1+1）\times 1=3$$

2. 程序的环路复杂性

环路复杂性 V(G)的计算方式有以下三种：

第一种：V(G)=区域数目。程序控制流图中的区域数目对应其结构复杂度。所谓区域是指边界和节点包围的形状。计算区域时还要考虑图外部分，将其作为一个区域。

第二种：V(G)=E-N+2。其中 E 表示边界数目，N 表示节点数目。

第三种：V(G)=P+1。其中 P 表示判断节点数目。

如图 3-4 所示。

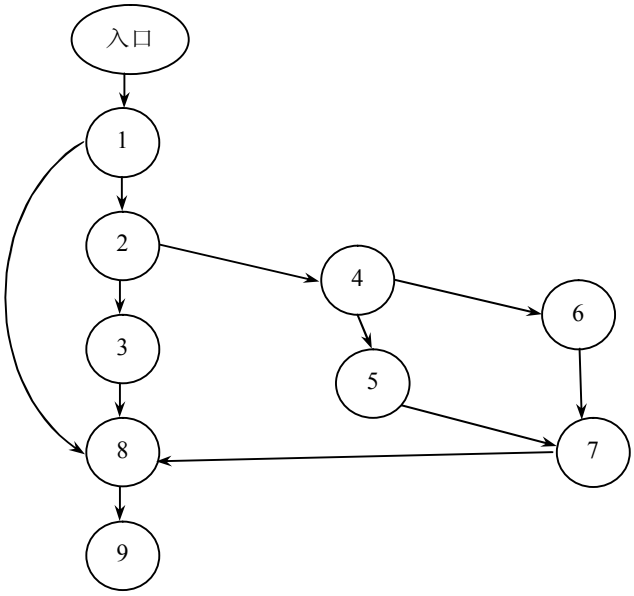


图 3-4　某程序段控制流图

方法一：该控制流图有 4 个区域。

方法二：V(G)=11-9+2=4。

方法三：V(G)=3+1=4。

根据上述计算，其控制流图最多有 4 条独立路径（以下通过节点序号表示路径），分别是：

P1=1、8、9

P2=1、2、3、8、9

P3=1、2、4、5、7、8、9

P4=1、2、4、6、7、8、9

如果程序的每个独立路径都测试过，则可以认为程序的每个语句都验证过了。

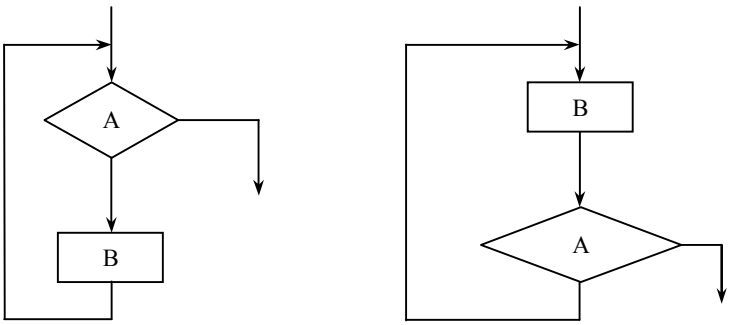
3．Z 路径覆盖

通过对路径覆盖的分析可以知道，对于路径较少且比较简单的程序而言，实现路径覆盖是可能实现的。但是，如果程序中出现多个循环或者判断的话，所涉及的路径数目也会快速增加，这就可能造成无法实现路径覆盖。为了解决这个问题，就必须去除一些次要因素，限制循环次数，从而减少路径数量。这种简化循环下的路径覆盖称为 Z 路径覆盖。

这里所说的对循环化简是指限制循环的次数。无论循环的形式和实际执行循环体的次数多少，我们只考虑循环 1 次和 0 次两种情况。即只考虑执行时进入循环体一次和跳过循环体这两种情况。

程序中比较典型的循环控制结构即为 while 和 do-while。两者的区别在于 while 是先判断，再执行；do-while 是先执行，再判断。因此，do-while 至少执行一次。两种循环的流程图，如图 3-5 所示。

根据 Z 路径覆盖的定义，如果限定循环只执行 1 次或 0 次，则图 3-5 中的两个数据流程图合并为一个。如图 3-6 所示，图 3-5（b）先执行循环体 B 一次，再判断，与图 3-6 中只执行右边的支路效果相同。



（a）while 循环结构　　（b）do…while 循环结构

图 3-5　循环结构

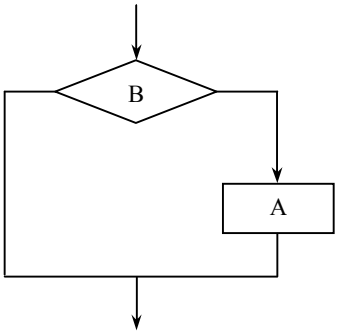


图 3-6　条件选择结构

现在通过一个具体的例子来看如何实现路径测试。有如下程序段：

```
void   sort ( int   x, int   y )
1   {
2          int   a=1,b=2;
3            while ( x--> 0 )
4          {
5                   if (y==0 )
6                              a=b-3;
7                  else
8                           if (y==1 )
9                                  a=b+5;
10                         else
11                             a=b*2;
12          }
13 }
```

步骤 1：画控制流图。如图 3-7 所示。

步骤 2：计算环形复杂度：该流程图有 4 个区域。

步骤 3：导出独立路径（用语句编号表示）。

路径 1：3→13

路径 2：3→5→6→12→3→13

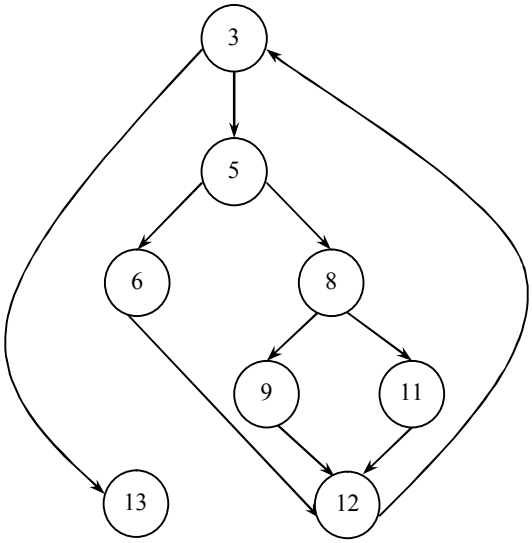


图 3-7　该程序段控制流图

路径 3：3→5→8→9→12→3→13

路径 4：3→5→8→11→12→3→13

步骤 4：设计测试用例，如表 3-5 所示。

表 3-5　设计测试用例

测试用例	输入数据	预期输出
测试用例 1	x = 0 y= 0	a = 1 b = 2
测试用例 2	x= 1 y= 0	a =-1 b = 2
测试用例 3	x =1 y = 1	a =7 b = 5
测试用例 4	x= 1 y = 2	a =4 b = 2

3.2　白盒测试工具（JUnit）

软件测试在软件投入使用前，对软件需求分析、设计规格书和编码进行最后的审查，这是软件质量保证的关键步骤。大量的数据表明，软件测试的工作量往往占软件开发总工作量的40%以上，而且成本不菲。所以软件测试在整个开发过程中具有举足轻重的地位。

软件测试在软件开发过程中跨越了两个阶段:通常在编写每一个模块之后就要做必要的测试，这就叫单元测试，编码和单元测试属于软件开发过程中的同一阶段。在这个阶段之后，需要对软件系统进行各种综合的测试，即综合测试，它属于软件工程的测试阶段。

软件测试是软件开发的重要组成部分，但是很多开发者却忽略单元测试。他们认为测试应该由专门的测试人员来做。因为他们对自己写出的代码很“了解”，但他们却忽视了重要

的一点，如果成员不对自己的代码进行测试，他们怎么知道自己写的代码会按照预期的方式运行呢？

单元测试就是开发者写一段测试代码来验证自己编写的一段代码运行是否正确。一般来说，一个单元测试用来判定在给定条件写某个函数的行为。例如，如果想测试一个类型的某个函数返回的对象是否是原来预期的对象。

那么为什么要进行单元测试呢？当编写完一段代码之后，系统会进行编译，然后开始运行。如果编译都没有通过，运行就更不可能了。

如果编译通过只能说明没有语法错误，但却无法保证这段代码在任何时候都会按照自己的预期结果运行。所有的这些问题单元测试都可以解决。编写单元测试可以验证自己编写的代码是否按照预期运行。总之，单元测试可以使开发者的工作变得越来越轻松。

3.2.1 白盒测试工具介绍

JUnit 是 1997 年 Erich 和 Kent Beck 为 Java 语言创建的一个简单而有效的单元测试框架。JUnit 是 XUnit 测试体系架构的一种实现。

在 JUnit 单元测试框架设计时，设定了三个总体目标：

第一个是简化测试的编写，这种简化包括测试框架的学习和实际测试单元的编写。

第二个是使测试单元保持持久性。

第三个则是可以利用既有的测试来编写相关的测试。

要使用 JUnit，请先至 JUnit 官方网站 http://www.junit.org/，单击 Download JUnit 后出现 JUnit 下载列表（主要的 JUnit 版本为 JUnit3 和 JUnit4，本文以 JUnit3 为主进行讲解），下载 JUnit3.8.1 压缩包，如图 3-8、图 3-9 所示。

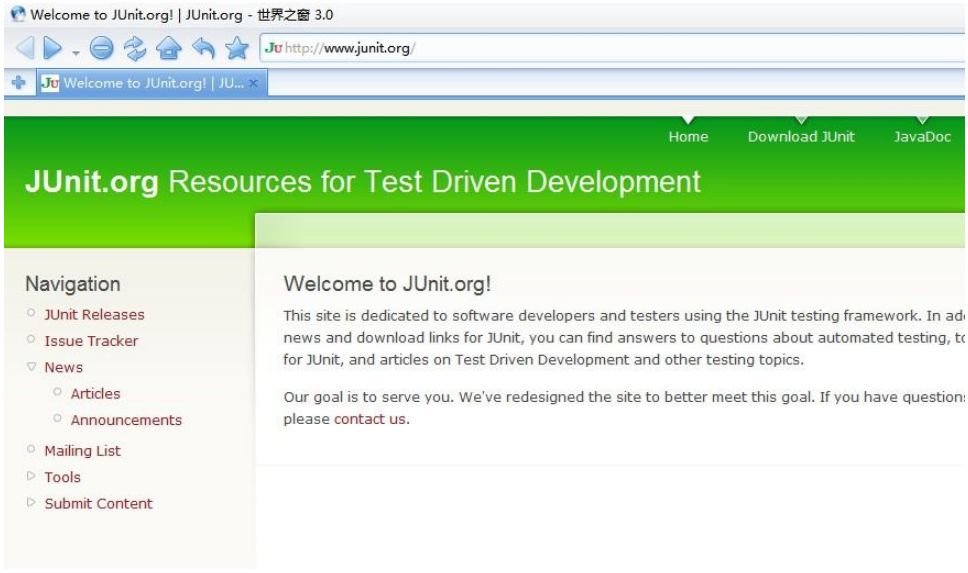


图 3-8 JUnit 官方网站

下载后解开压缩文件，当中会含有 junit.jar 文件，将这个文件复制到指定的文件夹中，如 c:\junit3.8.1\junit.jar，然后设定 CLASSPATH。

如果在 CMD 环境下可以使用 set classpath 命令进行设置，如图 3-10 所示。

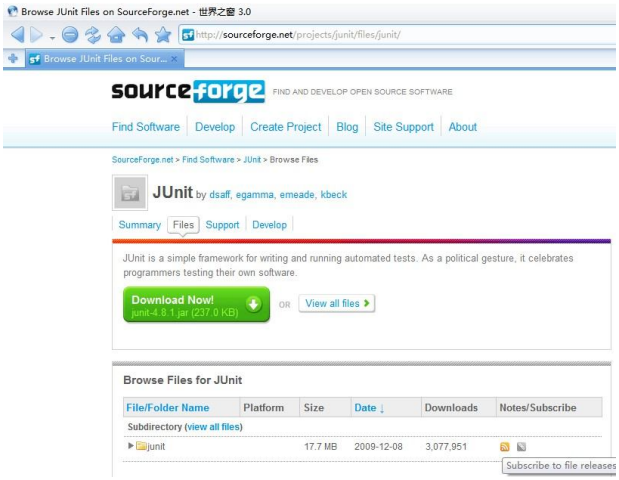


图 3-9　JUnit 下载列表

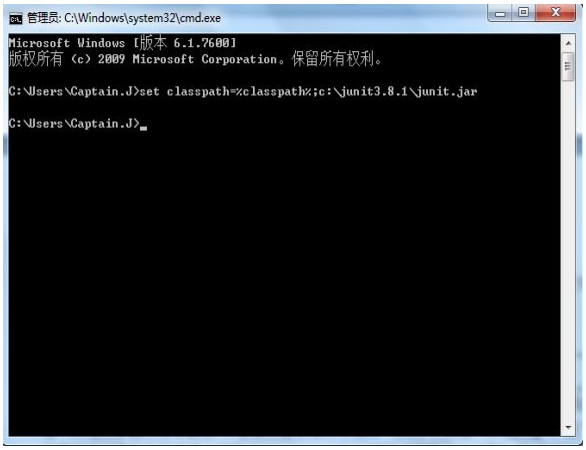


图 3-10　CMD 环境下设置 CLASSPATH

如果是在 Windows 2000/XP 环境下，请在“系统属性”→“高级”→“环境变量”中设定“系统变量”中的 CLASSPATH，如果没有就自行增加，如图 3-11 所示。

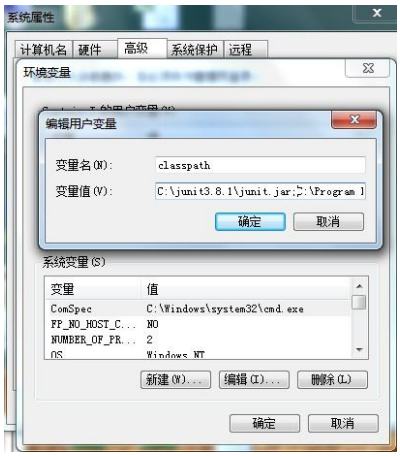


图 3-11　Windows 环境变量设置

可以通过三种方式测试 CLASSPATH 是否设置正确：文本模式测试范例。在 CMD 环境下输入 java junit.textui.TestRunner，如果出现图 3-12 所示界面表示 JUnit 安装正确。

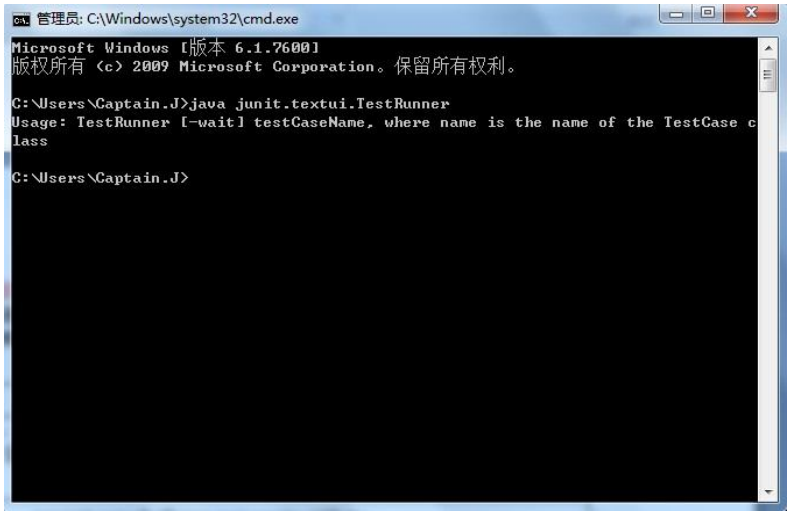


图 3-12　文本模式测试

AWT 图形模式测试范例。在 CMD 环境下输入 java junit.awtui.TestRunner，如果出现图 3-13 所示界面表示 JUnit 安装成功。

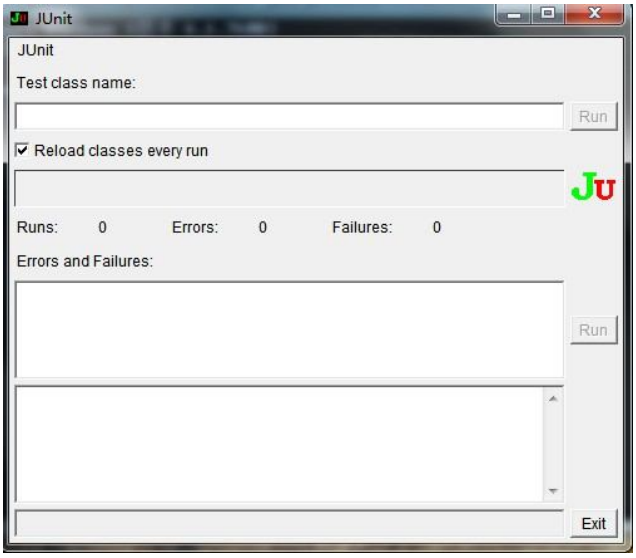


图 3-13　AWT 模式测试

Swing 图形模式测试范例。在 CMD 环境下输入 java junit.swingui.TestRunner，如果出现图 3-14 所示界面表示 JUnit 安装成功。

Eclipse 是常用的 Java 开发工具，在 Eclipse IDE 中集成了 JUnit 组件，无须另行下载和安装，但是要使用 Eclipse 中提供的运行 JUnit 单元测试用例和测试套件的图形用户界面，还要在 Eclipse 中进行一些设置。其中主要是就是类路径变量的设置。下面先看一下路径变量的具体设置步骤：

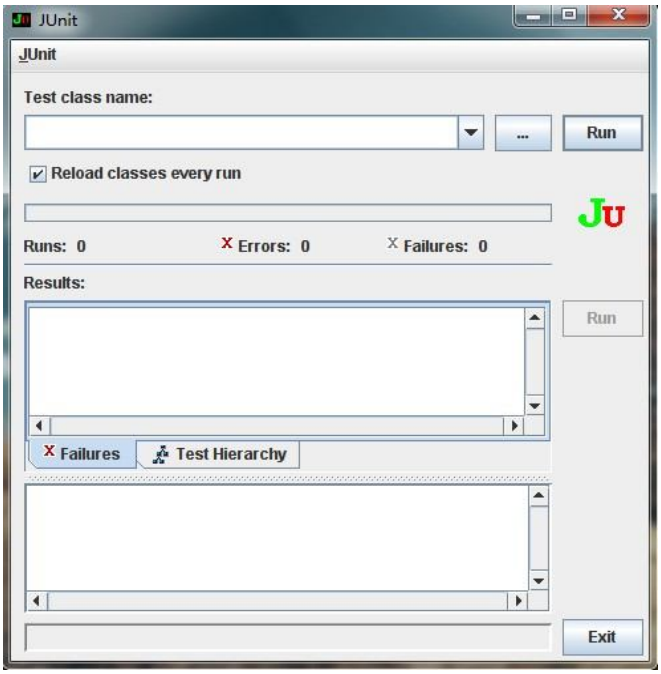


图 3-14　Swing 模式测试

（1）在主菜单栏上选择“窗口”→“首选项”，出现“首选项”对话框，如图 3-15 所示。

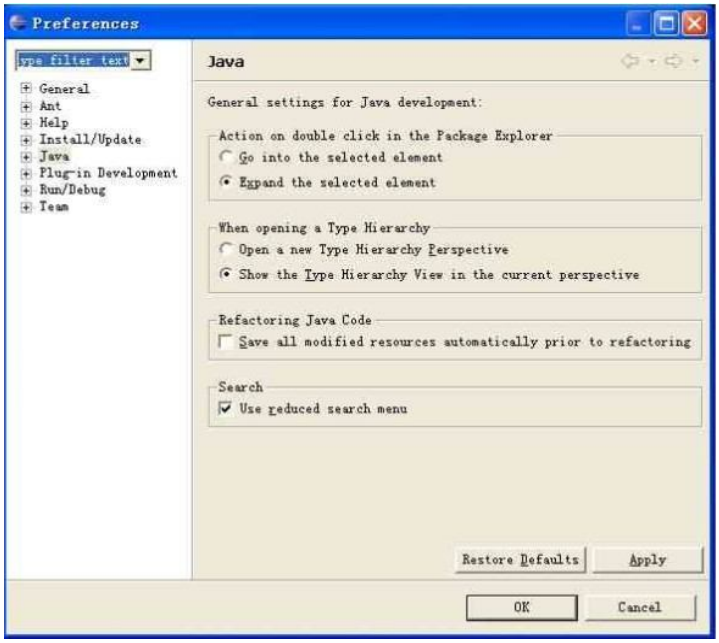


图 3-15　Eclipse 首选项

（2）展开 Java 节点，选择 Build Path→Classpath Variables，如图 3-16 所示，单击 New 按钮，在对话框中输入新的变量名 JUNIT，设置路径为 junit.jar，可以在安装目录/eclipse/plugins/org.junit_3.8.1/junit.jar 下找到 junit 压缩包，如图 3-17 所示，单击 OK 后如图 3-18 所示。

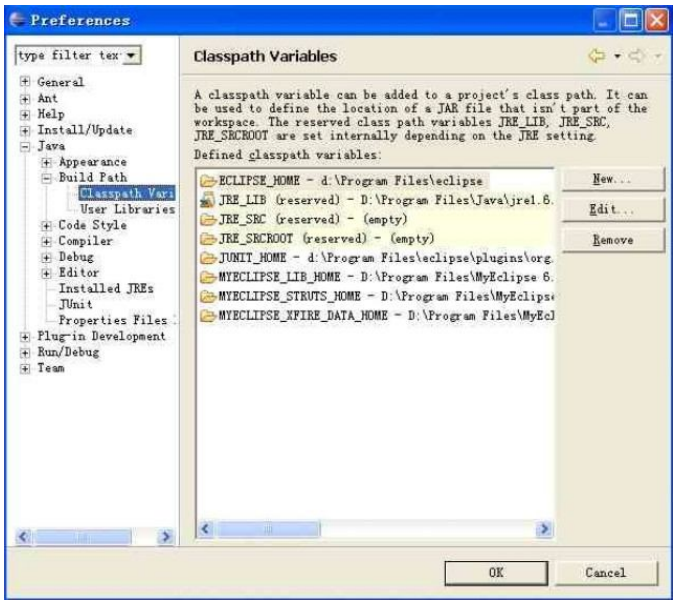


图 3-16　Classpath Variables 设置界面

图 3-17　添加 Variable 环境

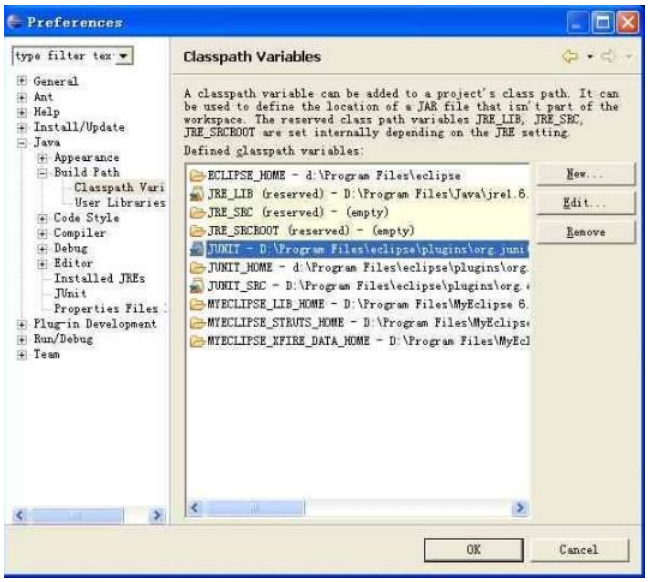


图 3-18　Variable 设置完成

（3）为了调试的需要，还要添加 JUnit 包的源代码，可以在 Eclipse 安装目录

/eclipse/plugins/org.eclipse.jdt.source_3.0.2/ 下搜索到 junitsrc.zip。为 JUnit 源代码创建一个新的环境变量 JUNIT_SRC，按照上面的步骤将其连接到 junitsrc.zip 所在的路径，如图 3-19 所示，单击 OK 按钮后 Eclipse 下 JUnit 环境设置完成。

图 3-19　Junit_SRC 设置

3.2.2　使用 JUnit 进行白盒测试

1. 独立的 JUnit 测试

（1）创建一个简单的 Java 类 Service.java，存放在 D:\junit\bo 文件夹下，类中有一个方法 calculate( )用于判断输入的三个数字能不能构成三角形。类的代码为：

```
package bo;

public class Service {
    public String calculate(int a, int b,int c) {
        String result=null;
        if(a+b<=c||a+c<=b||b+c<=a){
            result="非三角形";
        }else if(a==b&&a==c&&b==c){
            result="等边三角形";
        }else if(a!=b&&a!=c&&b!=c){
            result="一般三角形";
        }else{
            result="等腰三角形";
        }
        return result;
    }
}
```

（2）创建该类的测试类 ServiceTest.java，存放在 D:\junit\test 文件夹下。类的代码为：

```
package test;

import bo.Service;
import junit.framework.TestCase;

public class ServiceTest extends TestCase {
    public void testCalculate() {
        Service service = new Service();
        assertEquals(service.calculate(1, 1, 1), "等边三角形");
        assertEquals(service.calculate(1, 1, 2), "非三角形");
        assertEquals(service.calculate(3, 4, 5), "一般三角形");
```

```
            assertEquals(service.calculate(2, 2, 3), "等腰三角形");
        }
}
```

（3）编译源代码后，输入图 3-20 所示命令执行测试，如果显示 OK 表示测试通过。

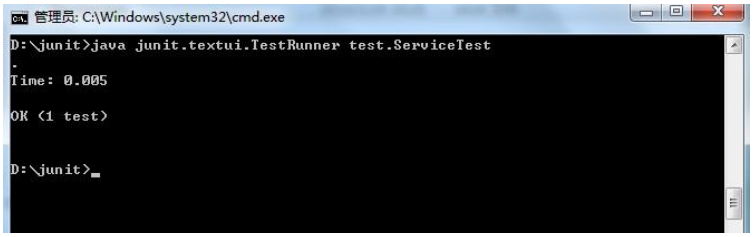


图 3-20　DOS 环境测试用过

（4）将原本正确的代码修改错误，代码如下：

```
package bo;

public class Service {
    public String calculate(int a, int b,int c) {
        String result=null;
        if(a+b<=c&&a+c<=b||b+c<=a){//此行代码中的||换成了&&，模拟程序代码错误
            result="非三角形";
        }else if(a==b&&a==c&&b==c){
            result="等边三角形";
        }else if(a!=b&&a!=c&&b!=c){
            result="一般三角形";
        }else{
            result="等腰三角形";
        }
        return result;
    }
}
```

（5）重新编译后执行测试，如图 3-21 所示，结果显示测试失败。

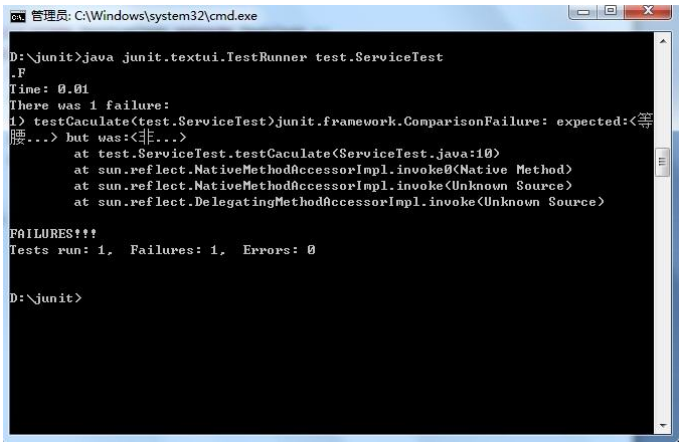


图 3-21　DOS 环境下测试失败

2. Eclipse 环境下 JUnit 测试

（1）创建一个 Java 项目 junit_test，选中项目文件单击鼠标右键，选择 Build Path→Configure Build Path，如图 3-22 所示。

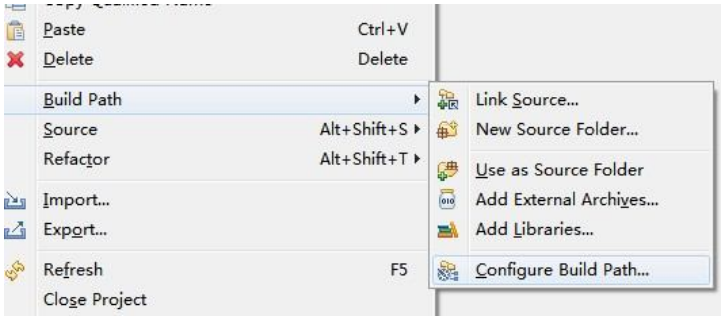


图 3-22　Build Path 设置（1）

（2）在弹出的图 3-23 所示窗口中单击 Add Variable，在图 3-24 所示界面中选择上一节新建的 JUnit 后单击 OK 按钮。

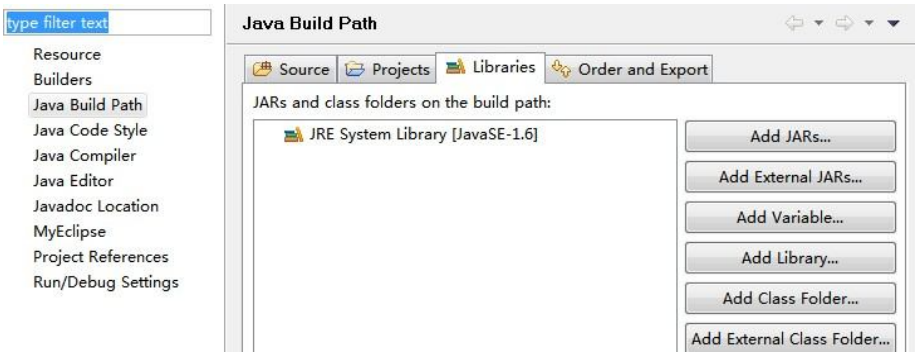


图 3-23　Build Path 设置（2）

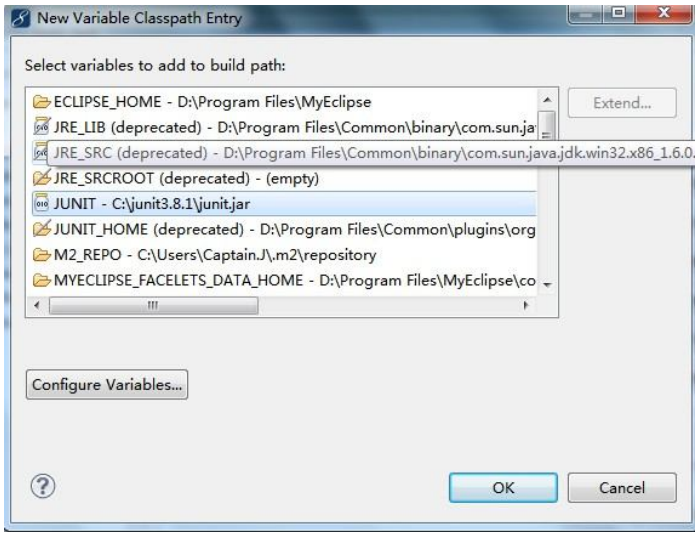


图 3-24　Build Path 设置（3）

（3）在项目中新建 Service.java，如图 3-25 所示。

Service.java

```
package bo;

public class Service {
    public String caculate(int a, int b,int c) {
        String result=null;
        if(a+b<=c||a+c<=b||b+c<=a){
            result="非三角形";
        }else if(a==b&&a==c&&b==c){
            result="等边三角形";
        }else if(a!=b&&a!=c&&b!=c){
            result="一般三角形";
        }else{
            result="等腰三角形";
        }
        return result;
    }
}
```

图 3-25　Service.java

（4）在项目中新建 ServiceTest.java，如图 3-26 所示。

ServiceTest.java

```
package test;

import bo.Service;

public class ServiceTest extends TestCase {
    public void testCaculate() {
        Service service = new Service();
        assertEquals(service.caculate(1, 1, 1), "等边三角形");
        assertEquals(service.caculate(1, 1, 2), "非三角形");
        assertEquals(service.caculate(3, 4, 5), "一般三角形");
        assertEquals(service.caculate(2, 2, 3), "等腰三角形");
    }
}
```

图 3-26　ServiceTest.java

（5）选中 ServiceTest.java 文件单击鼠标右键后选择 Debug As→JUnit Test，如图 3-27 所示。

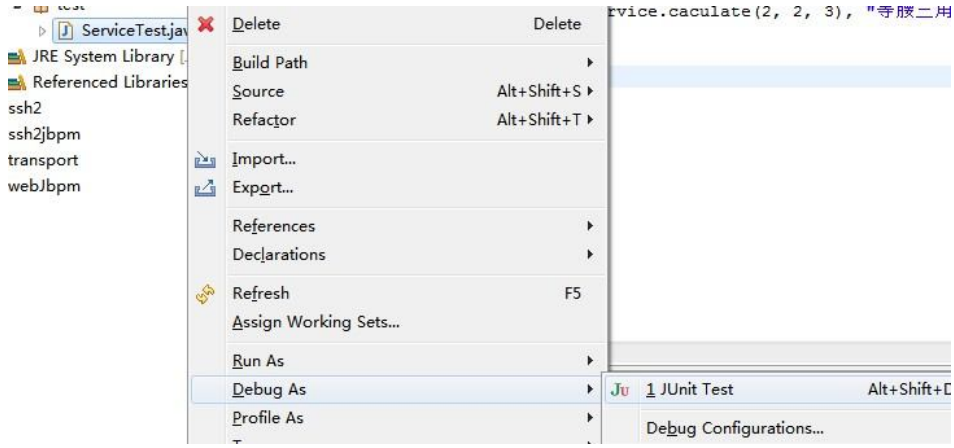


图 3-27　测试用例

（6）如果图 3-28 中的进度条为绿色则表示测试通过。

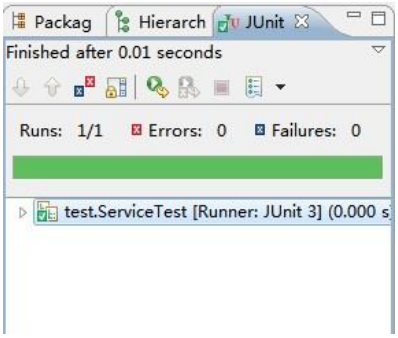


图 3-28　测试通过

本章小结

本章介绍了白盒测试的相关概念，详细介绍了几种白盒测试方法，如代码检查、语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖、组合覆盖、路径覆盖和路径测试等。全面分析了这几种白盒测试的优缺点以及相关用法。并介绍了白盒测试工具 JUnit 的使用方法。

实训习题

练习 1．什么是白盒测试？它有哪些测试方法？

练习 2．本章所介绍的几种代码覆盖的优缺点分别是什么。

练习 3．用不同的覆盖测试，设计下面程序段的测试用例。

```
Begin
    if(a>1)or(y<6)   then   c=c+x;
      if(a<25)and(c>0)   then   c=c*y;
```

练习 4．什么是 JUnit？