第5章 白盒测试用例设计及应用

白盒测试是基于程序内部逻辑结构，针对程序语句、路径、变量状态等进行测试的一种方法。如图 5-1 所示的程序流程图，可使用白盒测试方法检查该程序流程图中的各个分支条件是否得到满足、每条执行路径是否按预定要求正确地工作。

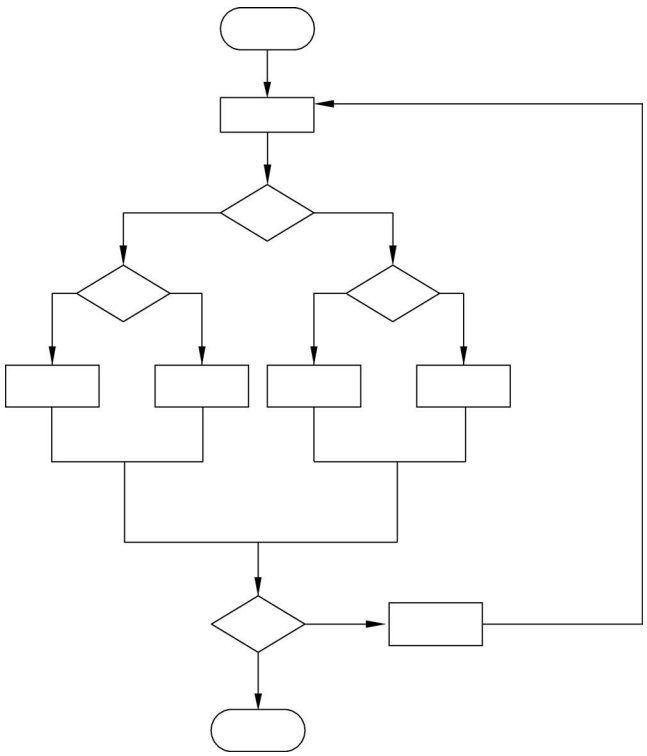


图 5-1　程序内部结构流程图

单元测试主要采用白盒测试方法，辅以黑盒测试方法。白盒测试方法应用于代码评审、单元程序代码测试，而黑盒测试方法则应用于模块、组件等大单元的功能测试。

白盒测试方法包括逻辑覆盖法和基本路径测试法，逻辑覆盖法又可分为更多不同覆盖级别的测试方法。测试时，可以根据不同的覆盖要求使用这些方法完成测试用例的设计。白盒测试方法的分类如图 5-2 所示。

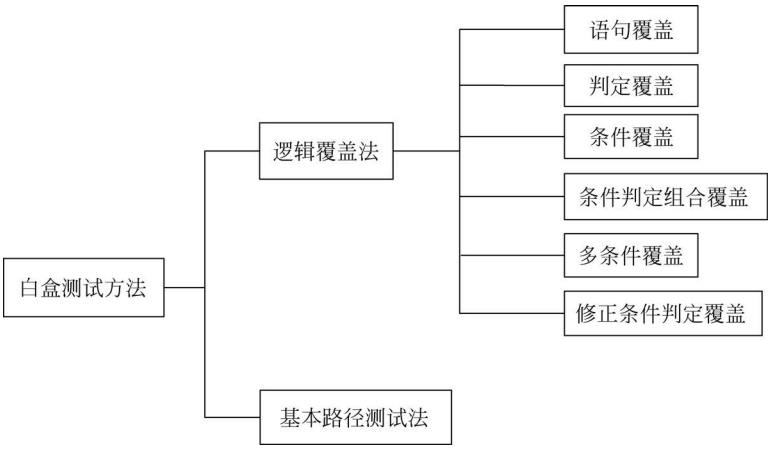


图 5-2　白盒测试方法的分类

5.1　逻辑覆盖法

逻辑覆盖通过对程序逻辑结构的遍历实现对程序的覆盖,它是一系列测试过程的总称,这组测试过程逐渐实现越来越完整的通路测试。从覆盖源程序语句的详尽程度分析,逻辑覆盖标准包括以下不同的覆盖标准:语句覆盖(Statement Coverage,SC)、判定覆盖(Decision Coverage,DC)、条件覆盖(Condition Coverage,CC)、条件判定组合覆盖(Condition/Decision Coverage,CDC)、多条件覆盖(Multiple Condition Coverage,MCC)和修正条件判定覆盖(Modified Condition Decision Coverage,MCDC)。

为便于理解,下面统一使用程序 5-1(用 C 语言书写)进行讲解,图 5-3 所示为其程序流程图。

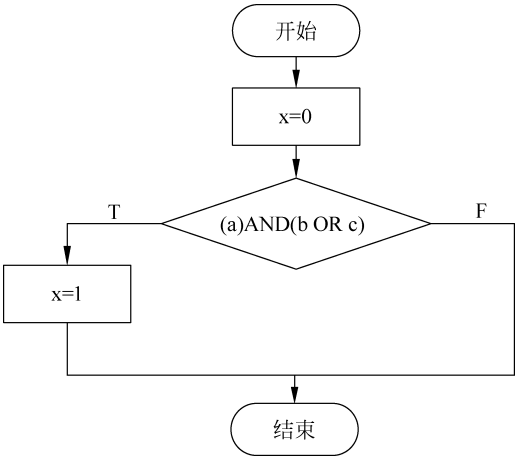


图 5-3　示例代码的流程图

程序 5-1

```
int function(bool a,bool b,bool c)
{
```

```
    int x;
    x = 0;
    if(a &&( b||c))
        x = 1;
    return x;
}
```

扫一扫

视频讲解

5.1.1 语句覆盖

语句覆盖，又称为行覆盖(Line Coverage)、段覆盖(Segment Coverage)、基本块覆盖(Basic Block Coverage)，这是最常用也是最常见的一种覆盖方式。其基本思想是设计若干测试用例，运行被测程序，使程序中每条可执行语句至少应该执行一次。

为了使程序 5-1 中的每条语句都能够至少执行一次，可以构造以下测试用例：

- a=T，b=T，c=T

从程序中的每条可执行语句都得到执行这一点看，语句覆盖的方法似乎能够比较全面地检验每条语句。但其实语句覆盖对程序执行逻辑的覆盖率很低，这是语句覆盖法最严重的缺陷。

假如这段程序中判定的逻辑运算有问题，例如，判定的第一个运算符 && 错写成运算符||，或第二个运算符||错写成运算符 &&，这时使用上述测试用例仍然可以达到 100%的语句覆盖，并且上述逻辑错误无法被检测出来。因此，一般认为语句覆盖是很弱的逻辑覆盖。

扫一扫

视频讲解

5.1.2 判定覆盖

比语句覆盖稍强的覆盖标准是判定覆盖。判定覆盖的基本思想是设计若干测试用例，运行被测程序，使程序中的每个判定都至少获得一次真值或假值，或者说使程序中的每一个取真分支和取假分支至少经历一次，因此判定覆盖又称为分支覆盖。

除了真假双值判定语句外，还有多值判定语句，如 case 语句，因此判定覆盖更一般的含义是：使每一个判定获得的每一种可能的结果至少被满足一次。

以程序 5-1 示例代码为例，构造以下测试用例即可实现判定覆盖标准：

- a=T，b=T，c=T
- a=F，b=F，c=F

应该注意到，上述两组测试用例不仅满足判定覆盖，还满足语句覆盖，从这一点看，判定覆盖比语句覆盖更强一些。但是同样地，假如这一程序段中判定的逻辑运算有问题，如表 5-1 所示，判定的第一个运算符 && 错写成运算符||或第二个运算符||错写成运算符 &&，这时使用上述测试用例仍可以达到 100%的判定覆盖，仍然无法发现上述假设的逻辑错误，因此就需要更强的逻辑覆盖标准。

表 5-1 判定覆盖

序号	a	b	c	a && (b||c)	a || (b||c)	判定覆盖/%
1	T	T	T	T	T	50
2	F	F	F	F	F	50

扫一扫

视频讲解

5.1.3 条件覆盖

在程序设计中,一个判定语句可能是由多个条件组合而成的复合判定,在图5-3所示的程序流程图中,判定a&&(b||c)包含了3个条件:a、b和c。为了更彻底地实现逻辑覆盖,可以采用条件覆盖的标准。条件覆盖的含义是:构造一组测试用例,使每一个判定语句中的每个逻辑条件的可能值至少满足一次。

按照这一定义,程序5-1要达到100%的条件覆盖,可以使用以下测试用例:

- a=F,b=T,c=F
- a=T,b=F,c=T

仔细分析可以发现,上述测试用例在满足条件覆盖的同时,把判定的两个分支也覆盖了。但是否可以说,达到了条件覆盖也就必然实现了判定覆盖呢?

假如选用以下两组测试用例:

- a=F,b=T,c=T
- a=T,b=F,c=F

可以发现这两组测试用例满足条件覆盖,却不能满足分支覆盖,如表5-2所示。因此,不能绝对地说条件覆盖的覆盖率一定比判定覆盖的高,反之亦然。为达到更高的覆盖率,需要同时兼顾条件覆盖和分支覆盖。

表5-2 条件覆盖

序号	a	b	c	a &&(b||c)	条件覆盖/%	判定覆盖/%
1	F	T	T	F	100	50
2	T	F	F	F		

5.1.4 条件判定组合覆盖

条件判定组合覆盖的含义是:设计足够的测试用例,使判定中每个条件的所有可能(真/假)至少出现一次,并且每个判定本身的判定结果(真/假)也至少出现一次。

对于图5-3所示的例子,选用以下两组测试用例可以符合条件判定组合覆盖标准:

- a=T,b=T,c=T
- a=F,b=F,c=F

但是条件判定组合覆盖也存在一定的缺陷。例如,判定的第一个运算符&&错写成运算符||或第二个运算符||错写成运算符&&,如表5-3所示,使用上述测试用例仍然可以达到100%的条件判定组合覆盖,无法发现这些逻辑错误。

表5-3 条件判定组合覆盖

序号	a	b	c	a||(b||c)	a &&(b&&c)	条件判定组合覆盖/%
1	T	T	T	T	T	100
2	F	F	F	F	F	

5.1.5 多条件覆盖

多条件覆盖也称条件组合覆盖，它的含义是：设计足够的测试用例，使每个判定中条件的各种可能组合都至少出现一次。显然，满足多条件覆盖的测试用例是一定满足判定覆盖、条件覆盖和条件判定组合覆盖的。

在图 5-3 所示的例子中，判定语句中包含 3 个逻辑条件，每个逻辑条件有两种可能取值，因此共有 $2^3=8$ 种可能的取值组合，对应的测试用例如表 5-4 所示，这些测试用例能够保证多条件覆盖。

表 5-4 多条件覆盖

序号	a	b	c	a&&(b||c)
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

5.1.6 修正条件判定覆盖

综上可知，当程序中的判定语句包含多个条件时，运用多条件覆盖方法进行测试，其条件取值组合数目非常大。为了节省时间和资源，提高测试效率，就必须精心设计测试用例，从数量巨大的可用测试用例中精心挑选少量的测试数据，使用这些测试数据就能够达到最好的测试效果。

修正条件判定覆盖在多条件的基础上进行数据的挑选，挑选数据的要求是：程序的判定被分解为通过逻辑操作符(AND、OR)连接的布尔条件，每个条件对于判定的结果值是独立的。

在表 5-4 所示的 8 条满足多条件覆盖的测试用例基础上，按照修正条件判定覆盖的要求选择需要的测试用例，选择结果如表 5-5 所示。

表 5-5 修正条件判定覆盖

序号	输入条件的全组合			判定结果	测试用例集		
	a	b	c	a&&(b||c)	a	b	c
1	T	T	T	T	√		
2	T	T	F	T	◇	√◇○	
3	T	F	T	T	○		√◇○
4	T	F	F	F		√◇○	√◇○
5	F	T	T	F	√		
6	F	T	F	F	◇		
7	F	F	T	F	○		
8	F	F	F	F			

图 5-3 所示的示例程序中有一个判定,该判定由 3 个条件组成,基于修正条件判定覆盖所应选取的测试数据需要使每个条件都能满足相对于判定结果的独立性要求。例如,当考虑条件 a 时,要满足 a 对于判定结果的独立性,则应该保持条件 b 和条件 c 的取值不变,仅变化条件 a 的取值,并且 a 的变化能影响判定结果的变化。从表 5-5 中可以看出,在用例 1 和用例 5 这两个测试数据中,条件 b 和条件 c 的取值均没有发生变化,仅条件 a 的取值变化影响了判定结果的变化,因此用例 1 和用例 5 达到了条件 a 的修正条件判定覆盖要求。同理,用例 2 和用例 6、用例 3 和用例 7 这两组数据也可以达到条件 a 的修正条件判定覆盖要求。

以同样的方式考虑条件 b,从表 5-5 中可以看出,在用例 2 和用例 4 中,条件 a 和条件 c 保持不变,仅条件 b 的变化影响了判定结果的变化,达到了条件 b 的修正条件判定覆盖的要求。再考虑条件 c,从表 5-5 中可以看出,在用例 3 和用例 4 这两组数据中,条件 a 和条件 b 保持不变,仅条件 c 的变化影响判定结果的变化,达到了条件 c 的修正条件判定覆盖的要求。

经过上述分析,可以得到用例集{1,2,3,4,5}(表 5-5 中用√表示)、{2,3,4,7}(表 5-5 中用○表示)、{2,3,4,6}(表 5-5 中用◇表示),这 3 组用例集均可满足修正条件判定覆盖的要求。

5.2 基本路径测试法

5.1 节使用的例子是个比较简单的程序,仅包含两条逻辑路径。但在实际问题中,即使一个不太复杂的程序,其程序路径的组合数量都是一个庞大的数字,想在测试中完全覆盖所有的路径是不现实的。

为解决这一难题,需要把测试覆盖的路径数压缩到一定范围内,如程序中的循环体在测试时仅执行一次。本节介绍的基本路径测试法就是这样一种测试方法,它在程序控制流图的基础上,通过分析控制流图的环路复杂性,导出基本可执行路径的集合,然后据此设计测试用例。设计出的测试用例要保证在测试中程序的每条可执行语句至少执行一次。

5.2.1 程序的控制流图

控制流图是描述程序控制流的一种图示方式,其中基本的控制结构对应的图形符号如图 5-4 所示。在这些图形符号中,圆圈称为控制流图的节点,箭头称为控制流图的边。节点代表程序语句,边代表程序走向。

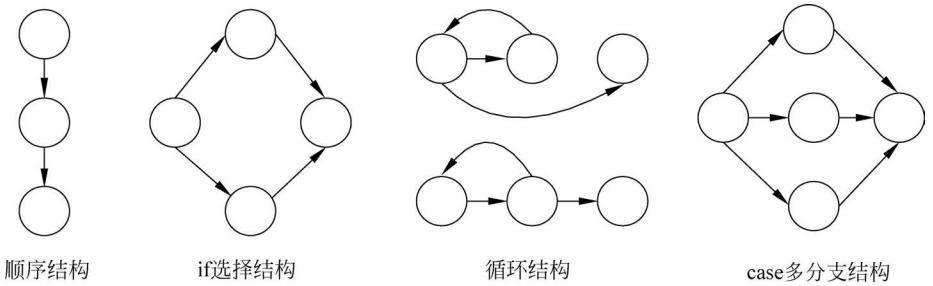
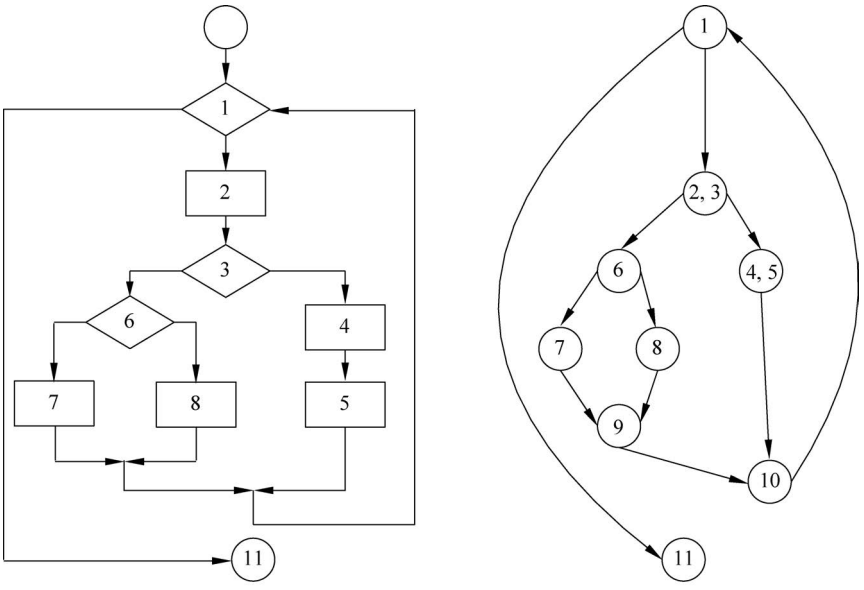


图 5-4 控制流图的图形符号

例如，图 5-5(a)所示的是一个程序流程图，它可以映射成如图 5-5(b)所示的控制流图。

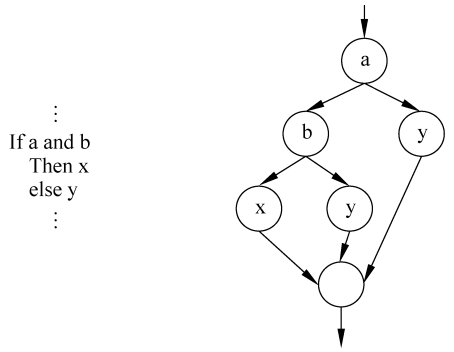


(a) 程序流程图　　(b) 控制流图

图 5-5　程序流程图和对应的控制流图

图 5-5 中，假定在程序流程图中用菱形框表示的判定条件内不包含复合条件，即每个判定都仅由一个单条件组成。另外，约定一组顺序处理框可以映射为一个单一的节点。控制流图中的箭头(边)表示了控制流的方向，类似于程序流程图中的流程线。一条边必须终止于一个节点，尤其在分支结构中分支的汇聚处，也应该添加一个汇聚节点(即使汇聚处没有执行语句)。边和节点构成的封闭图形部分叫区域，当对区域计数时，图形外的部分也应记为一个区域。

如果判断中的条件表达式是复合条件，即条件表达式是由一个或多个逻辑运算符(OR、AND)连接的逻辑表达式，则需要把由复合条件构成的判断拆分为一系列由单个条件构成的嵌套判断。例如，图 5-6(a)所示的程序代码段中的分支判定包含两个单条件 a 和 b，其控制流图中应该将该分支判定拆分，画成如图 5-6(b)所示的图形。



(a) 代码　　(b) 控制流图

图 5-6　复合逻辑下的控制流图

5.2.2 控制流图的环路复杂性

程序的环路复杂性即 McCabe 复杂性度量,在进行程序的基本路径测试时,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,这是确保程序中每个可执行语句至少执行一次必需的测试用例数目的上界。

独立路径是指包括一组以前没有处理过的语句或条件的一条路径。从控制流图看,一条独立路径是至少包含有一条在其他独立路径中从未有过的边的路径。例如,在图 5-5(b)所示的控制流图中,一组独立的路径如下:

```
path1: 1-11
path2: 1-2-3-4-5-10-1-11
path3: 1-2-3-6-8-9-10-1-11
path4: 1-2-3-6-7-9-10-1-11
```

从此例中可知,一条新的路径必须包含一条新边。路径 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 不能作为一条独立路径,因为它只是前面已经说明了路径的组合,没有通过新的边。

path1、path2、path3 和 path4 组成了图 5-5(b)所示的控制流图的一个基本路径集。只要设计出的测试用例能够确保这些基本路径的执行,就可以使程序中的每个可执行语句至少执行一次,每个条件的取真和取假分支也能得到测试。基本路径集不是唯一的,对于给定的控制流图,可以得到不同的基本路径集。

通常环路复杂性还可以简单地定义为控制流图的区域数。对于图 5-5(b)所示的控制流图,它有 4 个区域,其环路复杂性 V(G)=4;环路复杂性也可以通过控制流图中边的个数减去节点的个数再加 2(11−9+2=4)计算。环路复杂性是构成基本路径集的独立路径数的上界,通过计算环路复杂性可以得到应该设计的测试用例的数目。

5.2.3 基本路径测试法示例

基本路径测试法既适用于依据模块的详细设计进行测试用例设计的情形,也适合依据源程序代码进行测试用例设计的情形。应用基本路径测试法设计测试用例的主要步骤如下。

(1) 以详细设计或源代码作为基础,导出程序的控制流图 G。

(2) 计算控制流图 G 的环路复杂性 V(G)。

(3) 确定基本路径。

(4) 生成测试用例,确保基本路径集中每条路径的执行。

下面以一个简单的 C 函数为例,说明使用基本路径测试法设计测试用例的过程。此函数的程序流程图如图 5-7 所示。

1. 以详细设计或源代码为基础,导出程序的控制流图

利用图 5-4 所示的控制流图的图形符号以及控制流图的构造规则绘制控制流图,如图 5-8(a)所示,图中圆圈中的数字对应图 5-7 中的圆圈,⑭是补上去的空节点,空节点能保证控制流图中的每条边的两端都连着节点。

对控制流图进行优化,把顺序执行的节点进行合并(顺序执行的节点合并后不影响路径条数),如图 5-8(b)所示。

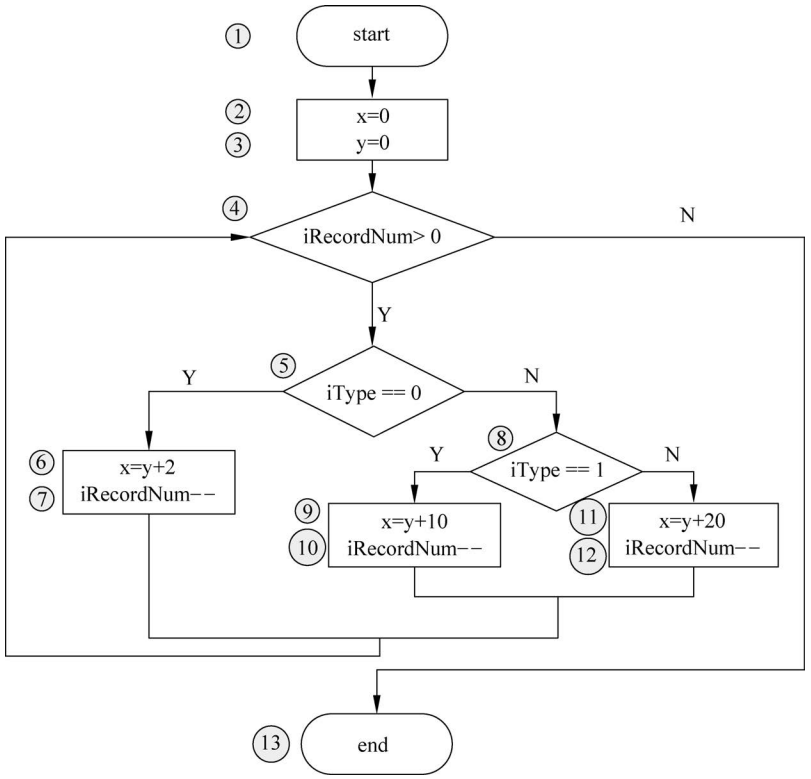
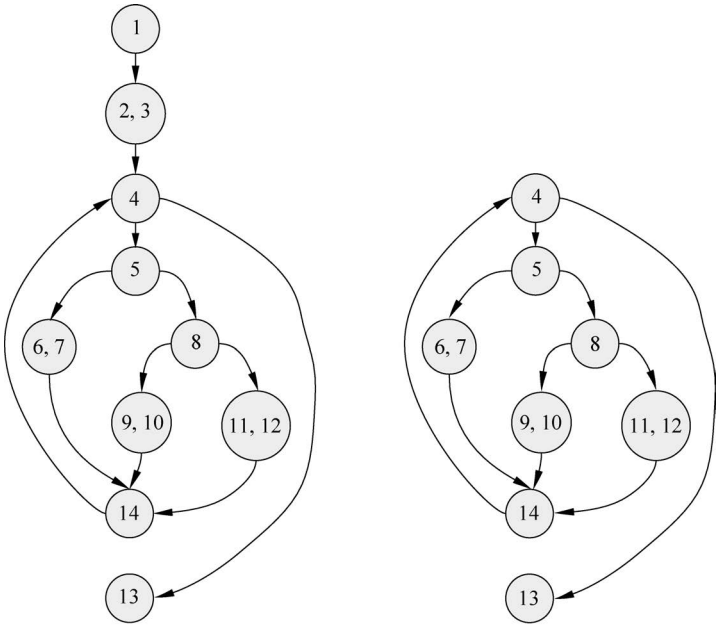


图 5-7　示例程序的程序流程图



(a) 控制流图　　(b) 优化流图

图 5-8　示例程序的控制流图

2. 计算控制流图 G 的环路复杂性 V(G)

- $V(G)=4$(区域数);
- 或 $V(G)=3$(判断节点数)$+1=4$;
- 或 $V(G)=12$(边的个数)-10(节点个数)$+2=4$。

3. 导出独立路径(用代码行号表示)

根据控制流图计算得到的环路复杂性,可知该程序的基本路径集中的路径条数为 4,具体路径描述如下:

- 路径 1:4→13
- 路径 2:4→5→6,7→14→4→13
- 路径 3:4→5→8→9,10→14→4→13
- 路径 4:4→5→8→11,12→14→4→13

4. 设计测试用例,确保基本路径集中的每条路径都被执行

设计的测试用例如表 5-6 所示。

表 5-6 测试用例表

ID	输入数据	预期结果
测试用例 1	iRecordNum=0,iType=0	x=0
测试用例 2	iRecordNum=1,iType=0	x=2
测试用例 3	iRecordNum=1,iType=1	x=10
测试用例 4	iRecordNum=1,iType=2	x=20

至此,本示例的测试用例设计完毕。下面给出本示例的源程序代码。

程序 5-2

```
int sort(int iRecordNum, int iType)
1   {
2       int x = 0;
3       int y = 0;
4       while(iRecordNum > 0)
5       {
6            if(iType == 0)
7            {
8                x = y + 2;
9                iRecordNum -- ;
10            }else if(iType == 1)
11              x = y + 10;
12            else
13              x = y + 20;
14       }
15         return x;
16   }
```

依据表 5-6 的测试用例去执行测试,给定不同的参数(用例中的输入数据)调用函数,观察函数的返回值是否与预期值(用例中的预期结果)一致,可以发现在执行测试用例 3 与测试用例 4 时,被测函数并未返回期望的数据,查找错误发现其原因是被测程序中存在死循环。

5.3 本章小结

实施单元测试能够使项目团队更快地完成工作，无数次的实践已经证明了这一点。项目开发时间越紧张，就越要进行单元测试。这个说法看似矛盾，增加了单元测试任务，对编程进度不利，但实际上，项目整体进度会因为实施单元测试而得到很大的帮助，团队可以更高质量、更快速地完成任务。本章结合实际案例，对单元测试时使用的白盒测试方法进行了详细介绍，帮助读者掌握白盒测试用例设计方法。

5.4 课后习题

1. 填空题

（1）白盒测试又称为_____或_____。白盒测试将测试对象看作一个透明的盒子，按照_____的结构测试程序，检验_____是否都能按预定要求正确工作，而不注重它的_____。通过在不同点_____，确定实际的状态是否与预期的状态一致。

（2）白盒测试方法包括_____和_____。

2. 单项选择题

（1）白盒测试根据程序的（　　）设计测试用例，黑盒测试是根据软件的规格说明设计测试用例。

A. 内部逻辑　　B. 功能　　C. 输入数据　　D. 应用范围

（2）关于黑盒测试与白盒测试的区别，下列说法正确的是（　　）。

A. 白盒测试侧重于程序结构，黑盒测试侧用于功能

B. 白盒测试可以使用自动化测试工具，黑盒测试不能使用工具

C. 白盒测试需要开发人员参与，黑盒测试不需要

D. 黑盒测试比白盒测试应用更广泛

（3）下列关于测试方法的叙述不正确的是（　　）。

A. 从某种角度上讲，白盒测试与黑盒测试都属于动态测试

B. 功能测试属于黑盒测试

C. 对功能的测试通常是要考虑程序的内部结构

D. 结构测试属于白盒测试

（4）在进行单元测试时，常用的方法是（　　）。

A. 采用白盒测试，辅以黑盒测试

B. 采用黑盒测试，辅以白盒测试

C. 只采用白盒测试

D. 只采用黑盒测试

（5）如果一个判定中的复合条件表达式为(a>1)‖(b<=3)，则为了达到100%的条件覆盖率，至少需要设计（　　）个测试用例。

A. 1　　B. 2　　C. 3　　D. 4

3. 问答题

（1）白盒测试必须遵循哪些原则？

(2) 什么是基本路径测试?

(3) 简述黑盒测试与白盒测试的区别。

4. 实践题

(1) 图形分析程序。

程序5-3将任意输入的两个正整数值分别存入x和y中,据此完成图形分析的功能:若x和y值相同,则提示"可以构建圆形或正方形";若$2<|x-y|\leqslant 5$,则提示"可以构建椭圆";若$|x-y|>5$,则提示"可以构建矩形"。根据给出的程序代码(Python语言),设计能满足语句覆盖要求的测试用例。

程序5-3

```
def judgment_graphics(x, y):
 if x == y:
    print('可以构建圆形或正方形')
 elif math.fabs(x - y)> 2 and math.fabs(x - y) <= 5:
    print('可以构建椭圆形')
 elif math.fabs(x - y) > 5:
    print('可以构建矩形')
 else:
    print('您输入的数据不在判断范围内')
```

(2) 累加计算程序。

一个求累加计算和R的程序流程图如图5-9所示,程序功能是:如果累加计算和$R=\sum_{K=0}^{|N|}K$(其中R和K初始化为0)不大于给定的最大整数值(Max),则输出实际的计算结果R,否则给出错误信息。参照此流程图,设计测试用例分别满足分支覆盖和条件覆盖。

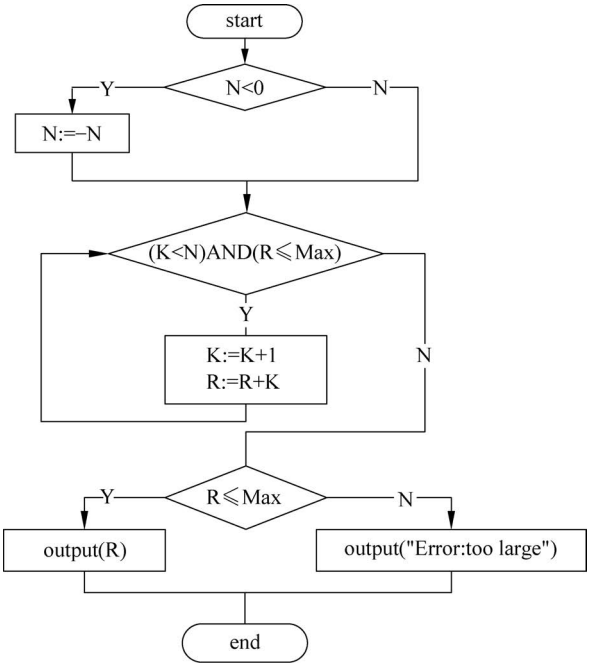


图5-9 累加计算公式的程序流程图

(3) 为图 5-10 所示的程序分别设计满足语句覆盖、判定覆盖、条件覆盖、条件判定组合覆盖、多条件覆盖与修正条件判定覆盖的测试用例。

(4) 为图 5-11 所示的程序设计符合基本路径测试法的测试用例。

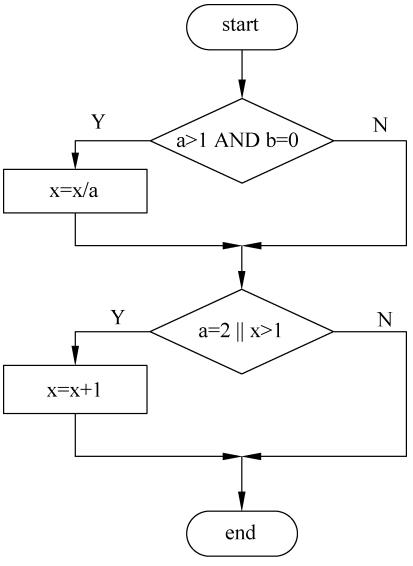


图 5-10　实践题(3)程序流程图

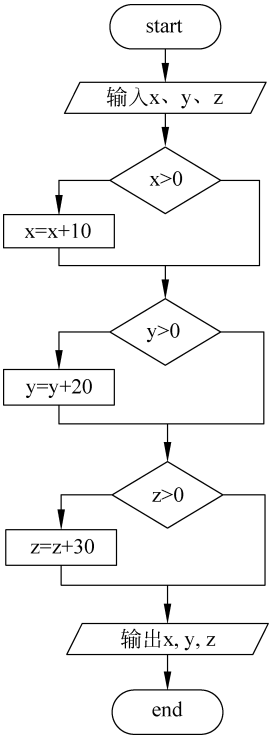


图 5-11　实践题(4)程序流程图

(5) 程序 5-4 是用 C 语言编写的三角形形状判断程序，请按照基本路径测试法为此程序设计测试用例。

要求：①画出其控制流图；②计算其环形复杂度；③写出所有基本路径；④为每一条独立路径各设计一组测试用例。

程序 5-4

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main()
{
  int a,b,c;
  printf("输入三角形的三条边边长:");
  scanf("%d %d %d",&a,&b,&c);

  if(a<=0||b<=0||c<=0)
      printf("不符合条件,请重新输入 a,b,c\n");
  else if(a+b<=c||abs(a-b)>=c)
      printf("不是三角形\n");
```

```
    else if(a == b&&a == c&&b == c)
        printf("这个三角形为等边三角形\n");
    else if(a == b||a == c||b == c)
        printf("这个三角形为等腰三角形\n");
    else
        printf("这个三角形为一般三角形\n");
}
```

(6) 图 5-12 所示的程序流程图描述了这样的功能：最多输入 50 个值(以－1 作为输入结束标志),计算这些值中可以作为学生分数的有效数据的个数及其总分和平均分。

要求：①画出其控制流图；②计算其环形复杂度；③写出所有的基本路径；④为每一条独立路径各设计一组测试用例。

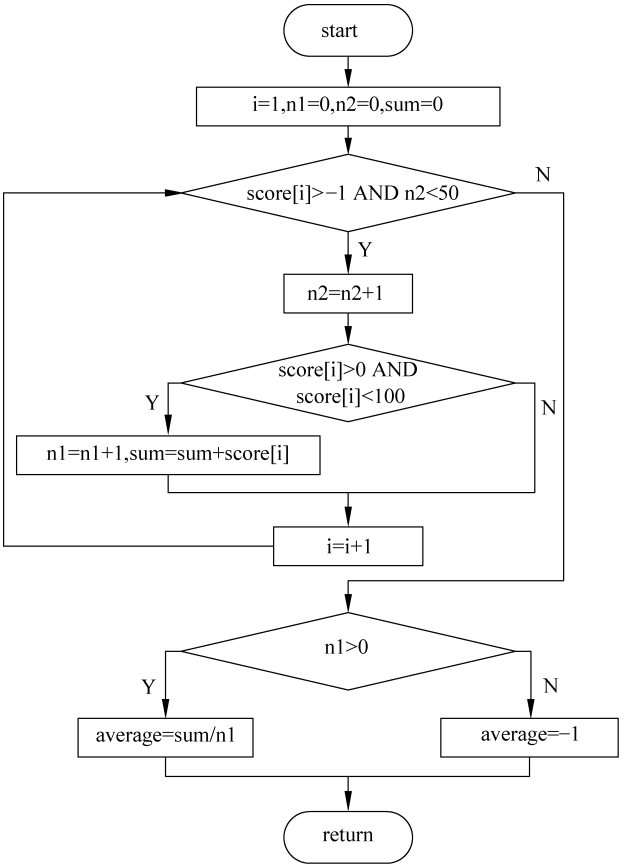


图 5-12　计算有效分数的个数及其总分和平均分