Program Testing

White Box Testing

[Introduction 1](#_Toc100590872)

[Aims of white box testing 1](#_Toc100590873)

[Fundamental Criteria 2](#_Toc100590874)

[Criteria C1 -Statement Coverage 2](#_Toc100590875)

[Criteria C2 -Branch Coverage 3](#_Toc100590876)

[Criteria C3 - Multiple Condition Coverage 6](#_Toc100590877)

[Description of Method 8](#_Toc100590878)

[Worked Examples 9](#_Toc100590879)

[Example 1 9](#_Toc100590880)

[Example 2 12](#_Toc100590881)

[Loop Testing Criteria 19](#_Toc100590882)

[Final Observations 20](#_Toc100590883)

[Solutions to Review Questions 22](#_Toc100590884)

# Introduction

In white box testing we make use of the program’s structure in deriving test cases. By looking at the code we can identify test values that will ensure that the program is fully executed. In the process we should be able to identify “errors of commission”, (i.e. bugs due to the programmer implementing a feature not required from the program specification), plus logic errors.

There are a variety of approaches to white box testing. We will focus on one main method known as Multiple Condition Coverage, (MCC), but also give guidelines and rules relating to other important requirements for white box testing.

The MCC method is an essential tool for any programmer concerned with the production of bug free software. It is best used for unit testing but can be scaled up and modified for use during other stages of the program development cycle.

# Aims of white box testing

We will start by looking at criteria upon which our techniques will be based. Criteria are simply conditions that our test cases should satisfy. Several criteria will be introduced and discussed. Eventually we will define satisfactory criteria, i.e. one that maximises our chance of identifying bugs, and then consider some examples illustrating the application of the criteria.

Our main concern is to ensure that our code is thoroughly executed. If any part of the program has not been executed under test, then the testing is not complete. Thus, our first aim is:

**to guarantee that all independent paths are exercised**

It is necessary to define the term independent paths but that will come later.

All programs, in general, are composed of branch points at which important decisions are made. Studies have shown that it is at these points in the program that mistakes are likely to be made. This realisation leads to the second aim:

**to exercise all logical decisions on true and false sides**

Loops within programs have very high probabilities of being error prone. In generating any test cases we should pay particular attention to the execution of loops structures. In particular our third aim will be to:

**to execute all loops at boundaries and within boundaries.**

Finally, as we have noted elsewhere, programs are composed of both algorithms and data structures. Our final aim should be to ensure that:

**all internal data structures are adequately exercised**.

The first three aims will lead us to the method of test design we will use in this course.

# Fundamental Criteria

Test criteria are the rules we use or objectives we attempt to achieve when we draw up our test cases. There are quite a variety of possible criteria. We will define two as our starting point.

### Criteria C1 -Statement Coverage

**Every statement in the unit must be executed at least once under some test.**

This criterion is absolutely essential. If at the end of testing there are still statements in the program that have not been exercised then the program has not been adequately tested. Unfortunately it is a very weak criterion. If we only use C1 then there is every chance that we will miss program bugs. Consider the following example:

|  |  |  |
| --- | --- | --- |
| **Correct Code** |  | **Incorrect Code** |
| if ( x< 0 ) |  | if ( x<0 ) |
| { |  | { |
| x = x + a; |  | x = x + a; |
| } |  | x = x + a; |
| x = x + a; |  | } |

In both programs we can satisfy C1 by choosing a negative value for x. With this choice the result of executing the sections of code is that the final value of x is x+2a. The error is not detected but we have chosen a value for x that satisfies C1.

This demonstrates the weakness of statement coverage. If it is possible to generate test cases that satisfy a criteria but will not identify a bug then we have problems.

However, even though statement coverage is very weak it is essential that any criteria we finally adopt must include statement coverage. If at the end of testing some code all the statements have not been executed the code has not been correctly tested. If you haven’t tried it how do you know what it does?

##### Review Question 1

*Consider the following section of code:*

*a = 0;*

*if ( b > 0)*

*a = 6;*

*b = c/a;*

*Does the test case b = 5 satisfy C1? Would this be a satisfactory test set for the code? If not, why not?*

##### Review Question 2

*Consider the following section of code:*

*public int average()*

*{*

*int sum = 0;*

*int n = 0;*

*x = getX();*

*while ( x>0)*

*{*

*sum = sum + x;*

*n++;*

*x = getX();*

*}*

*return sum / n ;*

*}*

*Explain how statement coverage would fail to detect a potential bug in this example.*

### Criteria C2 -Branch Coverage

Every branch alternative in the unit has to be executed at least once under some test.

An alternative definition of this criterion is:

Every decision in the unit has at least one true outcome and one false outcome.

Typical decision making statements in code are if’s and while’s. Conditions are evaluated and depending on the outcome a section of code may be executed.

If we take the above example,

|  |  |  |
| --- | --- | --- |
| **Correct Code** |  | **Incorrect Code** |
| if ( x< 0 ) |  | if ( x<0 ) |
| { |  | { |
| x = x + a; |  | x = x + a; |
| } |  | x = x + a; |
| x = x + a; |  | } |

The decision is made at the *if* statement when we check to see if x is less than 0. In order to satisfy C2 we would need to have x <0 giving a true outcome and x>= 0 to give a false result. Supposing we use the test cases x = -1 and x=1 and assume that a = 3 then running the two tests would give:

|  |  |  |
| --- | --- | --- |
| **Correct Code** | **x** | **Incorrect Code** |
| 5 | -1 | 5 |
| 4 | 1 | 1 |

##### Review Question 3

*Rework the problem with the same test values for x but with a=0. What happens?*

This criterion, branch coverage, ensures that every statement is executed and does go some way towards removing the weakness of C1 But there are still problems.

##### Review Question 4

*Consider the following section of code*

*if ( x != 0 )*

*y = 5;*

*else*

*z = z – x;*

*if ( z>1)*

*z = z/x;*

*else*

*z = 0;*

*and the test set {(x=0,z=1),(x=1,z=3)}. This set satisfies C2. Is it adequate? If not why, not?*

##### Review Question 5

*Consider the following section of code:*

*while ( z > 1 )*

*{*

*z = z / x;*

*x--;*

*}*

*Does the test set {x= 2,z= 2} satisfy C2?*

*What happens if we use {z=20, x=2} ?*

##### Review Question 6

*List all the Java control structures that involve branching, i.e. moving between alternative blocks of code.*

A more fundamental problem can occur due to the way compilers may deal with compound conditions. Consider the following example:

if (( a>1 ) && ( b=0 ))

{

x = x / a;

}

We have a compound condition, ( a>1 ) && ( b=0 ). In order to have a true outcome both conditions must be true, i.e. a>1 and b=0. We obtain a false outcome when at least one of the conditions is false.

The test set {(a=2,b=0), (a=0,b=0)} satisfy C2. Suppose we apply this to the following section of code:

if (( a>1 ) && ( b>=0 ))

{

x = x / a;

}

The result of applying our test set to both pieces of code will be indistinguishable. This is obviously a weakness of using C2.

A further problem arises because some compilers generate code that performs a lazy test i.e. for a multiple condition such as the one above only those parts of the predicate that are needed to give a result are evaluated.

To see what we mean consider the truth table for “P && Q”

|  |  |  |
| --- | --- | --- |
| **P** | **Q** | **P && Q** |
| **T** | **T** | **T** |
| **T** | **F** | **F** |
| **F** | **T** | **F** |
| **F** | **F** | **F** |

The table shows that when P is false there is no need to evaluate Q, the result is determined by P. Compilers tend to be written to generate code that exploits this feature. Thus in the above case with the multiple condition (a>0) && ( b = 0) we could generate test cases to satisfy C2 without ever executing the code for b=0. This would allow bugs to remain undiscovered.

##### Review Question 7

*Produce truth tables for each of the following Boolean expressions:*

1. *P || Q*
2. *(x<y) && (y<z)*
3. *!(x>=y) && !(y>=z)*

From the above it is obvious that we need even stronger criteria against which to measure our test cases. However, just as for statement coverage, branch coverage is necessary if we are to adequately test our program.

### Criteria C3 - Multiple Condition Coverage

All possible combinations of condition outcomes in each decision must be invoked at least once under some test.

If we had the situation represented by the truth table for P and Q, above, we would need to have 4 test cases. We would need sets of values that made each of the entries in the truth table occur. This is what we mean by multiple condition coverage.

This criterion subsumes C1 and C2 and so is a stronger criterion in the sense that any test set generated to satisfy C3 must also cover C1 and C2.

The following example illustrates the method. Consider the program fragment

if (( a>1 ) && ( b=0))

x = x / a;

if (( a=2 ) && ( x>1 ))

x = x + 1;

We have two sequential if statements involving three variables a,b,x. We will assume that a and b are integers and that x is real. We wish to find a set of values for a,b,x that would form a set of test cases satisfying the criteria C3.

The first step is to draw up a set of truth tables for each condition in the code.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **a > 1** | **b = 0** | **(a>1) && (b=0)** |
| **1** | **T** | **T** | **T** |
| **2** | **T** | **F** | **F** |
| **3** | **F** | **T** | **F** |
| **4** | **F** | **F** | **F** |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **a = 2** | **x > 1** | **(a=2) && (x>1)** |
| **5** | **T** | **T** | **T** |
| **6** | **T** | **F** | **F** |
| **7** | **F** | **T** | **F** |
| **8** | **F** | **F** | **F** |

For convenience we number each of the table entries.

We must now find sets of values for a, b and x so that, if they were used to execute the code, then each of the entries in the truth table will have been satisfied at least once.

There is no easy way of doing this. Starting with the first table pick an entry. Next pick an entry in the second table. Check to ensure that the selected entries are compatible, i.e. they are not contradictory. Finally try and pick values to satisfy both entries.

In the present case we will select entries 1 and 5. This leads us to the following set of conditions:

{(a>1),(b=0),(a=2),x>1)}

Such a set of conditions is known as a path predicate equation. The solution of such equations is in general not simple. However, this example has been specially constructed to illustrate the method and it is a relatively easy task to find the required values for a, b, and x.

Obviously a must equal 2 and b must equal 0. We are left with a choice for x. In selecting a value for x we must take into account any processing that has taken place prior to reaching the second if statement. With the values we have selected for a and b the statement

x = x / a

will have been executed. This means that the input value for x will have been modified and that the condition to be satisfied is not really x>1 but rather

x/a > 1

Since a=2 we require x >2.

Thus a test case satisfying entries 1 and 5 in the above truth tables would be

{a=2,b=0,x=3}

We now repeat the process until all the entries in the truth tables have been covered.

##### Review Question 8

Complete the exercise.

## Description of Method

1. Produce truth tables for all decision expressions in the program, numbering each entry.
2. Check each truth table and remove any impossible entries.
3. Produce a minimal number of test cases to ensure that each entity in the truth tables is covered at least once; (some entries may be covered several times).

# Worked Examples

### Example 1

We will now illustrate test case generation using Multiple Condition Coverage by producing test cases for the following algorithm:

public void progressWarning(int t1, int t2, int t3)

{

boolean dataok;

float average;

String message;

if ( ( t1 < 0 ) || ( t2 < 0 ) || ( t3 < 0 ))

dataok =false;

else

dataok =true;

if ( dataok is true )

{

average = (t1 + t2 + t3) / 3.0;

if ( average >= 60.0 )

{

System.out.print( “ passing “);

if ( average < 70.0 )

System.out.println( “ but only just “);

else

System.out.println(“ very easily”);

}

else

System.out.println( “ failing”);

}

else

System.out.println( “ invalid data”);

}

An inspection of the algorithm shows that there are four conditional statements:

if ( ( t1 < 0 ) || ( t2 < 0 ) || ( t3 < 0 ))

if ( dataok is true )

if ( average >= 60.0 )

if ( average < 70.0 )

One involves multiple conditions.

We produce a truth table for each. The entries in the truth tables are numbered consecutively in order to help with the second stage of the analysis.

After completing stage 1 of the process we have the following.

**(** t1 < 0 **) or (** t2 < 0 **) or (** t3 < 0 **)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | t1 < 0 | t2 < 0 | t3 < 0 |  |
| 1 | T | T | T | T |
| 2 | T | T | F | T |
| 3 | T | F | T | T |
| 4 | T | F | F | T |
| 5 | F | T | T | T |
| 6 | F | T | F | T |
| 7 | F | F | T | T |
| 8 | F | F | F | F |

**(** dataokis true **)**

|  |  |
| --- | --- |
| 9 | T |
| 10 | F |

**(** average >= 60.0 **)**

|  |  |
| --- | --- |
| 11 | T |
| 12 | F |

**(** average < 70.0 **)**

|  |  |
| --- | --- |
| 13 | T |
| 14 | F |

We must now consider each entry in a table to check that entries do not contradict each other. We need to ensure that each is logically possible.

In this example all are, so we can proceed to step three - generating test cases.

Remember our aim is to ensure that all 14 entries in the truth tables have been covered **at least once** by one of our test cases. It may be that some are used more than once.

Looking at the first table we see that the first seven of the eight entries cover invalid cases where one or more or the test scores are negative. Data that satisfy these conditions can only be paired with entry 10. Thus we require data values for t1, t2 and t3 that satisfy the following:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | t1 < 0 | t2 < 0 | t3 < 0 |  |
| 1 | T | T | T | T |
| 2 | T | T | F | T |
| 3 | T | F | T | T |
| 4 | T | F | F | T |
| 5 | F | T | T | T |
| 6 | F | T | F | T |
| 7 | F | F | T | T |

These seven will then satisfy

**(** dataokis true **)**

|  |  |
| --- | --- |
| 10 | F |

Entry number 8 will be triggered when we have three valid numbers and will satisfy entry 9.

Such data may satisfy 11 or 12 depending upon the value of average. Thus we want input values to give an average of over 60 and of under 60.

Finally entries 13 and 14 must be linked to 11 because of the structure of the program.

This gives use the following set of test cases:

|  |  |  |  |
| --- | --- | --- | --- |
| **t1** | **t2** | **t3** | **entry covered** |
| -1 | -1 | -1 | 1,10 |
| -1 | -1 | 50 | 2,10 |
| -1 | 50 | -1 | 3,10 |
| -1 | 50 | 50 | 4,10 |
| 50 | -1 | -1 | 5,10 |
| 50 | -1 | 50 | 6,10 |
| 50 | 50 | -1 | 7,10 |
| 60 | 61 | 66 | 8,9,11,13 |
| 71 | 72 | 73 | 8,9,11,14 |
| 50 | 50 | 50 | 8,9,12 |

The values used for the above test cases are not unique. Many other values could have been selected. However, any analysis of the algorithm based on multiple condition coverage would lead to the need for ten test cases.

### Example 2

This is a particularly complex program. On the basis of common sense it should be simplified by the use of supporting methods. However as a means to illustrate the application of MCC the form presented is very useful.

The method has three parameters: d, m, y representing the date. The year must be less than 2012. This is checked. There is other limited validation. All three parameters are assumed to be positive and in d is in the range 1 to 31 and m the range 1 to 12 but no checking is carried.

#### The Code

public class aDate

{

int day;

int month;

int year;

. . .

void setDateTomorrow (int d , int m, int y)

{

int error = 0;

if (m == 12)

{

if (d < 31 )

day = d+1 ;

else

{

day = 1;

.month = 1;

if ( y >= 2012)

{

error++;

System.out.println( " Error Year out of range!! ");

}

else

year = y + 1;

}

}

else

if (m == 2 )

{

if (d <28 )

day = d+1;

else

if (d == 28)

{

if ( ((y % 4)==0) && (((y %100) !=0)|| ((y % 400)==0) ) )

d = 29;

else

{

day = 1;

month = 3;

}

}

else

if (d == 29)

{

day = 1 ;

month = 3;

}

else

{

error++;

System.out.println( " Cannot have Feb " + d );

}

}

else

if ((m == 4) ||(m == 6)||(m== 9)|| (m == 11) )

{

if ( d < 30)

.day=d+1 ;

else

if (d == 30)

{

day = 1;

m+1;

}

else

{

error++;

System.out.println( " Too many days in month "+m);

}

}

else

{

if (d < 31)

day= d+1;

else

if (d == 31)

{

day =1;

.m+1;

}

else

{

error++;

System.out.println( " Too many days in month "+m);

}

}

if (error == 0)

System.out.println( "Tomorrow is " + day + " "+month + " " +year);

else

System.out.println( " ERROR in input data" );

}

…

}

#### Conditions in Program

Inspection of the code gives use 14 conditional statements :

(m == 12)

(d < 31 )

(y >= 2012)

(m == 2 )

(d <28 )

(d == 28)

((y % 4)==0) && (((y %100) !=0)|| ((y % 400)==0) ))

(d == 29)

((m == 4) || (m == 6) || (m == 9) || (m == 11) )

( d < 30)

(d == 30)

(d < 31)

(d == 31)

(error == 0)

We must write out a truth table for each of these. We have 14 truth tables. Care will need to be taken with the two involving multiple conditions.

#### Truth Tables

|  |  |
| --- | --- |
|  | m == 12 |
| 1 | **T** |
| 2 | **F** |

|  |  |
| --- | --- |
|  | d < 31 |
| 3 | **T** |
| 4 | **F** |

|  |  |
| --- | --- |
|  | y >= 2012 |
| 5 | **T** |
| 6 | **F** |

|  |  |
| --- | --- |
|  | m == 2 |
| 7 | **T** |
| 8 | **F** |

|  |  |
| --- | --- |
|  | d <28 |
| 9 | **T** |
| 10 | **F** |

|  |  |
| --- | --- |
|  | d == 28 |
| 11 | **T** |
| 12 | **F** |

|  |  |  |  |
| --- | --- | --- | --- |
|  | (y % 4)==0) | (y %100) !=0 | (y % 400)==0 |
| 13 | **T** | **T** | **F** |
| 14 | **T** | **F** | **T** |
| 15 | **F** | **T** | **F** |
| 16 | **T** | **F** | **F** |

##### Review Question 9

*Why are there only 4 entries in the above table and not 8?*

|  |  |
| --- | --- |
|  | d == 29 |
| 17 | **T** |
| 18 | **F** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | m == 4 | m == 6 | m == 9 | m == 11 |
| 19 | **T** | **F** | **F** | **F** |
| 20 | **F** | **T** | **F** | **F** |
| 21 | **F** | **F** | **T** | **F** |
| 22 | **F** | **F** | **F** | **T** |
| 23 | **F** | **F** | **F** | **F** |

##### Review Question 10

*Why are there only 5 entries in the above table and not 16?*

|  |  |
| --- | --- |
|  | d < 30 |
| 24 | **T** |
| 25 | **F** |

|  |  |
| --- | --- |
|  | d == 30 |
| 26 | **T** |
| 27 | **F** |

|  |  |
| --- | --- |
|  | d < 31 |
| 28 | **T** |
| 29 | **F** |

|  |  |
| --- | --- |
|  | d == 31 |
| 30 | **T** |
| 31 | **F** |

|  |  |
| --- | --- |
|  | error == 0 |
| 32 | **T** |
| 32 | **F** |

#### Test Case Generation

We must ensure that each entry in the above tables is covered at least once in our test set. Remember some may need to be used more than once. But we are trying to produce the smallest test set that covers all the entries.

The control flow of the program is quite complex so when we attempt to generate the test set it is very important to have the code available. This will enable us to eliminate impossible paths that could be generated by simply considering the table entries.

The following shows a possible set of test cases. They are not unique!

1. Select entries : ***1,3, 32***

{ m==12, d < 31, error == 0 }

**DAY= 15 ; MONTH = 12 ; YEAR = 1950**

1. Select entries ***: 1,4,5, 33***

{ m==12, d=31, y>=2012,error == 1 }

**DAY= 31; MONTH = 12 ; YEAR = 2012**

1. Select entries : ***1,4,6, 32***

{ m==12, d = 31,y <2012, error == 0 }

**DAY= 31 ; MONTH = 12 ; YEAR = 1960**

1. Select entries: ***2, 7, 9, 32***

{m !=12,m==2,d<28,error == 0}

**DAY= 15 ; MONTH = 2 ; YEAR = 1950**

1. Select entries ***: 2,7, 10,11,13,32***

{m !=12,m==2,d>=28, d ==28, y%4==0,y%100!=0, y%400 !=0,error == 0}

**DAY= 28 ; MONTH = 2 ; YEAR = 1996**

1. Select entries : ***2,7,10,11,14,32***

{m !=12,m==2,d<28, d ==28, y%4==0,y%100==0, y%400 ==0,error == 0}

**DAY= 28 ; MONTH = 2 ; YEAR = 2000**

1. Select entries : ***2,7,10,11,15,32***

{m !=12,m==2,d<28, d ==28, y%4!=0,y%100!=0, y%400 !=0,error == 0}

**DAY= 28 ; MONTH = 2 ; YEAR = 1995**

1. Select entries : ***2,7,10,11,16,32***

{m !=12,m==2,d<28, d ==28, y%4==0,y%100==0, y%400 !=0,error == 0}

**DAY= 28 ; MONTH = 2 ; YEAR = 1900**

1. Select entries: ***2,7,10,12,17,32***

{m!=12, m ==2, d >=28, d !=28.d ==29,error==0}

**DAY= 29 ; MONTH = 2 ; YEAR = 1992**

1. Select entries: ***2,7,10,12,18,33***

{m!=12, m ==2, d >=28, d !=28.d !=29,error==1}

**DAY= 31 ; MONTH = 2 ; YEAR = 1950**

1. Select entries: ***2,8,19,24,32***

{m!=12,m!=2,m==4, d <30,error==0}

**DAY= 15 ; MONTH = 4; YEAR = 1950**

1. Select entries : ***2,8,19, 25,26,32***

{m!=12,m!=2,m==4, d >=30, d==30, error==0}

**DAY= 30 ; MONTH = 4; YEAR = 1950**

1. Select entries :***2,8,19,25,27,33***

{m!=12,m!=2,m==4, d >=30, d!=30, error==1}

**DAY= 31 ; MONTH = 4 ; YEAR = 1950**

1. Select entries***: 2,8,20,24,32***

{m!=12,m!=2,m==6, d <30,error==0}

**DAY= 15 ; MONTH = 6 ; YEAR = 1950**

1. Select entries : ***2,8,20, 25,26,32***

{m!=12,m!=2,m==6, d >=30, d==30, error==0}

**DAY= 30 ; MONTH = 6 ; YEAR = 1950**

1. Select entries ***:2,8,20,25,27,33***

{m!=12,m!=2,m==6, d >=30, d!=30, error==1}

**DAY= 31 ; MONTH = 6 ; YEAR = 1950**

1. Select entries: ***2,8,21,24,32***

{m!=12,m!=2,m==9, d <30,error==0}

**DAY= 15 ; MONTH = 9 ; YEAR = 1950**

1. Select entries : ***2,8,21, 25,26,32***

{m!=12,m!=2,m==9, d >=30, d==30, error==0}

**DAY= 30 ; MONTH = 9 ; YEAR = 1950**

1. Select entries :***2,8,21,25,27,33***

{m!=12,m!=2,m==9, d >=30, d!=30, error==1}

**DAY= 31 ; MONTH = 9 ; YEAR = 1950**

1. Select entries***: 2,8,22,24,32***

{m!=12,m!=2,m==11, d <30,error==0}

**DAY= 15 ; MONTH = 11 ; YEAR = 1950**

1. Select entries : ***2,8,22, 25,26,32***

{m!=12,m!=2,m==11, d >=30, d==30, error==0}

**DAY= 30 ; MONTH = 11 ; YEAR = 1950**

1. Select entries ***:2,8,22,25,27,33***

{m!=12,m!=2,m==11, d >=30, d!=30, error==1}

**DAY= 31 ; MONTH = 11 ; YEAR = 1950**

1. Select entries : ***2,8,23,28,32***

{m!=12,m!=2,m!=4,6,911, d <31, error==0}

**DAY= 15 ; MONTH = 7 ; YEAR = 1950**

1. Select entries : ***2,8,23,29,30,32***

{m!=12,m!=2,m!=4,6,911, d >=31, d == 31, error==0}

**DAY= 31 ; MONTH = 7 ; YEAR = 1950**

1. Select entries ***: 2,8,23,29,31,33***

{m!=12,m!=2,m!=4,6,911, d >=31, d != 31, error==1}

**DAY= 32 ; MONTH = 7 ; YEAR = 1950**

In summary we have 25 test cases which we list in the following table:

|  |  |  |  |
| --- | --- | --- | --- |
| **d** | **m** | **y** | **Entries covered** |
| 15 | 12 | 1950 | 1,3, 32 |
| 31 | 12 | 2012 | 1,4,5, 33 |
| 31 | 12 | 1960 | 1,4,6, 32 |
| 15 | 2 | 1950 | 2, 7, 9, 32 |
| 28 | 2 | 1996 | 2,7, 10,11,13,32 |
| 28 | 2 | 2000 | 2,7,10,11,14,32 |
| 28 | 2 | 1995 | 2,7,10,11,15,32 |
| 28 | 2 | 1900 | 2,7,10,11,16,32 |
| 29 | 2 | 1992 | 2,7,10,12,17,32 |
| 31 | 2 | 1950 | 2,7,10,12,18,33 |
| 15 | 4 | 1950 | 2,8,19,24,32 |
| 30 | 4 | 1950 | 2,8,19, 25,26,32 |
| 31 | 4 | 1950 | 2,8,19,25,27,33 |
| 15 | 6 | 1950 | 2,8,20,24,32 |
| 30 | 6 | 1950 | 2,8,20, 25,26,32 |
| 31 | 6 | 1950 | 2,8,20,25,27,33 |
| 15 | 9 | 1950 | 2,8,21,24,32 |
| 30 | 9 | 1950 | 2,8,21, 25,26,32 |
| 31 | 9 | 1950 | 2,8,21,25,27,33 |
| 15 | 11 | 1950 | 2,8,22,24,32 |
| 30 | 11 | 1950 | 2,8,22, 25,26,32 |
| 31 | 11 | 1950 | 2,8,22,25,27,33 |
| 15 | 7 | 1950 | 2,8,23,28,32 |
| 31 | 7 | 1950 | 2,8,23,29,30,32 |
| 32 | 7 | 1950 | 2,8,23,29,31,33 |

***Finally remember that when documenting testing you should also indicate what the expected output from the code will be. ( See notes on documenting testing)***

# Loop Testing Criteria

Loop constructs play an important part in most programs. Unfortunately loops are a highly fault prone part of any program. Because of this we need to think very carefully about how we generate test cases to ensure that the loop has been adequately tested.

A typical sequence of actions relating to a simple loop can be represented as

prepare to execute loop

execute loop

follow on statements effected by loop

Errors caused by faults in designing the loop can occur in any one of the three sections, e.g.

1. We may not set up the program correctly to execute the loop - some initialisation may be faulty.
2. Bugs may occur inside the loop. One particular type of bug relates to re-initialisation - the loop may appear to execute correctly if only executed once but because values have been changed and not reset the processing may go wrong on the second and subsequent passes.
3. The loop exit conditions may be badly formed.
4. The processing that occurs after the loop may depend on the result of executing the loop. If the loop may be skipped these statements may not execute correctly.

These considerations lead us to the following coverage criteria, C4, for testing loops:

design test cases to cover the following situations, (if possible)

1. skipping the loop
2. one pass through the loop then exit
3. two passes through the loop then exit
4. m passes through the loop, where m<n and n is the maximum number of passes
5. n-1,n, n+1 passes through the loop, n is the maximum number of passes
6. If there are any excluded values treat as split ranges.

It may not be possible to apply all the above to particular programs but you should check to ensure that C4 has been met as far as possible.

# Final Observations

The white box criteria presented in these notes are by no means the only criteria that exist. There are many other approaches. Criteria C3 and C4 are very powerful and if used correctly will lead to “good” testing. However, it should be noted that no single method, be it white box based or black box based, is perfect. Indeed studies have shown that the best we can expect from a single method is the identification of about 66% of bugs in a program. Combining several methods will give a higher percentage of bugs detected but will not guarantee 100% effectiveness.

You should always remember the objectives of testing, namely

***To find errors in the program logic***

After generating test cases it is worth reviewing the test data to see if any obvious data values or situations have been missed out. Should you find such situations then generate test cases to cover them. In the last resort the technique of *error guessing* based on experience is a valid part of the testing process.

# Solutions to Review Questions

##### Review Question 2

The case b = 5 satisfies C1 but would not reveal the problem that when b<0 a division by zero exception would occur.

##### Review Question 2

Using statement coverage we would not need to consider the case when the first value of x was less than or equal to 0, i.e. the no data case. This would cause the program to crash at the line average = sum / n.

##### Review Question 3

The tests fail to identify the bug in the incorrect code. When designing test cases you should always think about special cases.

##### Review Question 4

The test set is inadequate since it will not show up the potential for a division by zero.

This example illustrates more than a weakness in branch testing. It demonstrates the need to look at code after applying test case generation methods and ensure that potential weaknesses have been identified.

##### Review Question 5

With (x= 2, z = 2) we will enter the loop once. After the first pass through z = 1 and x = 1 so we will not re-enter the loop. The condition (z>1) has had both true and false outcomes so C2 is satisfied.

With (x = 2, z = 20) we will enter the loop . After the first pass z= 10 and x = 1 so we re-enter the loop. After this pass z=10 and x = 0. We re-enter the loop and attempt to divide z by 0 !

Loops require particular care when testing. The notes include guidelines for testing loops.

##### Review Question 6

if, while, do-while, for, for-each, switch, try-catch

##### Review Question 7

Produce truth tables for each of the following Boolean expressions:

|  |  |  |
| --- | --- | --- |
| **P** | **Q** | **P or Q** |
| **T** | **T** | **T** |
| **T** | **F** | **T** |
| **F** | **T** | **T** |
| **F** | **F** | **F** |

|  |  |  |
| --- | --- | --- |
| **x< y** | **y<z** | **(x<y) and (y<z)** |
| **T** | **T** | **T** |
| **T** | **F** | **F** |
| **F** | **T** | **F** |
| **F** | **F** | **F** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **x>=y** | **not(x>=y)** | **y>=z** | **not(y>=z)** | **not(x>=y) and not(y>=z)** |
| **T** | **F** | **T** | **F** | **F** |
| **T** | **F** | **F** | **T** | **F** |
| **F** | **T** | **T** | **F** | **F** |
| **F** | **T** | **F** | **T** | **T** |

Do you notice anything about the last two tables? Try and select some values for x, y and z.

##### Review Question 8

In this example we can pair up entries 2 and 6; 3 and 7; 4 and 8. Solutions can then be found. A possible but not unique set of test cases are:

|  |  |  |
| --- | --- | --- |
| a | b | x |
| 2 | 1 | 1 |
| 1 | 0 | 2 |
| 1 | 1 | 1 |

##### Review Questions 9 & 10

Because the variables in the condition statements are related. This constrains the values they can take.