1

Introduction

What is software testing? Why do we need to test software? Can we live without testing? How do we handle software bugs reported by the customers? Who should work hard to avoid frustrations of software failures?

Such questions are endless. But we need to find answers to such burning questions. Software organizations are regularly getting failure reports of their products and this number is increasing day by day. All of us would appreciate that it is extremely disappointing for the developers to receive such failure reports. The developers normally ask: how did these bugs escape unnoticed? It is a fact that software developers are experiencing failures in testing their coded programs and such situations are becoming threats to modern automated civilization. We cannot imagine a day without using cell phones, logging on to the internet, sending e-mails, watching television and so on. All these activities are dependent on software, and software is not reliable. The world has seen many software failures and some were even fatal to human life.

1.1 SOME SOFTWARE FAILURES

A major problem of the software industry is its inability to develop error-free software. Had software developers ever been asked to certify that the software developed is error-free, no software would have ever been released. Hence 'software crises' has become a fixture of everyday life with many well-publicized failures that have had not only major economic impact but also have become the cause of loss of life. Some of the failures are discussed in subsequent sections.

1.1.1 The Explosion of the Ariane 5 Rocket

The Ariane 5 rocket was designed by European Space Agency and it was launched on June 4, 1996. It was an unmanned rocket and unfortunately exploded only after 40 seconds of its take

off from Kourou, French Guiana. The design and development took ten long years with a cost of $7 billion. An enquiry board was constituted to find the reasons of the explosion. The board identified the cause and mentioned in its report that [LION96]: "The failure of the Ariane 5 was caused by the complete loss of guidance and altitude information, 37 seconds after start of the main engine ignition sequence (30 seconds after lift-off). This loss of information was due to specification and design errors in the software of the inertial reference system. The extensive reviews and tests carried out during the Ariane 5 development programme did not include adequate analysis and testing of the inertial reference system or of the complete flight control system, which could have detected the potential failure". A software fault in the inertial reference system was identified as a reason for the explosion by the enquiry committee. The inertial reference system of the rocket had tried to convert 64 bit floating point number of horizontal velocity to a 16 bit signed integer. However, the number was greater than 32,767 (beyond the permissible limit of 16 bit machine) and the conversion process failed.

Unfortunately, the navigation system of Ariane 4 was used in Ariane 5 without proper testing and analysis. The Ariane 5 was a high speed rocket with higher value of an internal alignment function, known as horizontal bias. This value is for the calculation of horizontal velocity. On the day of the explosion, this value was more than expectations due to different trajectory of this rocket as compared to the Ariane 4. Therefore, the main technical reason was the conversion problem at the time of converting the horizontal bias variable, and this resulted into the shutdown of the computer of the inertial reference system. When the computer shut down, it passed control to an identical, redundant unit, which was there to provide backup in case of such a failure. But the second unit had failed in the identical manner before a few milliseconds. Why wouldn't it be? It was running the same software.

The designers never thought that the particular velocity value would ever be large enough to cause trouble. After all, it never had been before. Unfortunately Ariane 5 was a faster rocket than Ariane 4. Moreover, the calculation containing the error, which shut down the computer system, actually served no purpose, once the rocket was in the air. Its only function was to align the system before launch. So it should have been turned off. But designers chose long ago, in an earlier version of the Ariane 4, to leave this function running for the first forty seconds of flight – a 'special feature' meant to make the restart of the system easy in the event of a brief hold in the countdown. Such design decisions and poor testing resulted in the explosion of Ariane 5.

1.1.2 The Y2K Problem

The Y2K problem was the most critical problem of the last century. The whole world was expecting something drastic on January 1, 2000. Significant sums of money were spent by software companies to get rid of this problem. What was the problem? It was simply the case of using two digits for the year instead of four digits. For instance, 1965 was considered as 65. The developers could not imagine the problem of year 2000. What would happen on January 1, 2000? The last two digits i.e. 00 may belong to any century like 1800, 1900, 2000, 2100, etc. The simple ignorance or a faulty design decision to use only the last two digits for the year resulted into the serious Y2K problem. Most of the software was re-tested and modified or discarded, depending on the situation.

1.1.3 The USA Star-Wars Program

'Patriot missile' was the result of the USA 'Star Wars' program. This missile was used for the first time in the Gulf war against the Scud missile of Iraq. Surprisingly, 'Patriot missiles' failed many times to hit the targeted Scud missile. One of the failures killed 28 American soldiers in Dhahran, Saudi Arabia. An investigation team was constituted to identify the cause of failure. The team re-looked at every dimension of the product and found the reason for the failure. The cause of the failure was a software fault. There was a slight timing error in the system's clock after 14 hours of its operation. Hence, the tracking system was not accurate after 14 hours of operations and at the time of the Dhahran attack, the system had already operated for more than 100 hours.

1.1.4 Failure of London Ambulance System

The software controlling the ambulance dispatch system of London collapsed on October 26-27, 1992 and also on November 4, 1992 due to software failures. The system was introduced on October 26, 1992. The London Ambulance Service was a challenging task that used to cover an area of 600 square miles and handled 1500 emergency calls per day. Due to such a failure, there was a partial or no ambulance cover for many hours. The position of the vehicles was incorrectly recorded and multiple vehicles were sent to the same location. Everywhere people were searching for an ambulance and nobody knew the reason for non-arrival of ambulances at the desired sites. The repair cost was estimated to be £9m, but it is believed that twenty lives could have been saved if this failure had not occurred. The enquiry committee clearly pointed out the administrative negligence and over-reliance on 'cosy assurances' of the software company. The administration was allowed to use this system without proper alternative systems in case of any failure. The committee also termed the possible cause of failure as [ANDE98, FINK93]: "When the system went live, it could not cope with the volume of calls and broke under the strain. The transition to a back-up computer system had not been properly rehearsed and also failed."

1.1.5 USS Yorktown Incident

The USS Yorktown - a guided missile cruiser was in the water for several hours due to the software failure in 1998. A user wrongly gave a zero value as an input which caused a division by zero error. This fault further failed the propulsion system of the ship and it did not move in the water for many hours. The reason behind this failure was that the program did not check for any valid input.

1.1.6 Accounting Software Failures

Financial software is an essential part of any company's IT infrastructure. However, many companies have suffered failures in the accounting system due to errors in the financial software. The failures range from producing the wrong information to the complete system failure. There

is widespread dissatisfaction over the quality of financial software. If a system gives information in the incorrect format, it may have an adverse impact on customer satisfaction.

1.1.7 Experience of Windows XP

Charles C. Mann shared his views about Windows XP through his article in technology review [MANN02] as: "Microsoft released Windows XP on October 25, 2001. That same day, what may be a record, the company posted 18 megabyte of patches on its website for bug fixes, compatibility updates, and enhancements. Two patches fixed important security holes. Or rather, one of them did; the other patch did not work. Microsoft advised (still advises) users to back up critical files before installing patches." This situation is quite embarrassing and clearly explains the sad situation of the software companies. The developers were either too careless or in a great hurry to fix such obvious faults.

We may endlessly continue discussing the history of software failures. Is there any light at the end of the tunnel? Or will the same scenario continue for many years to come? When automobile engineers give their views about cars, they do not say that the quality of today's cars is not better than the cars produced in the last decade. Similarly aeronautical engineers do not say that Boeing or Airbus makes poor quality planes as compared to the planes manufactured in the previous decade. Civil engineers also do not show their anxieties over the quality of today's structures over the structures of the last decade. Everyone feels that things are improving day by day. But software, alas, seems different. Most of the software developers are confused about the quality of their software products. If they are asked about the quality of software being produced by companies, they generally say, "It is getting worse day by day." It is as if we say that Boeing's planes produced in 2009 are less reliable than those produced in 1980. The blame for software bugs belongs to nearly everyone. It belongs to the software companies that rush products to market without adequately testing them. It belongs to the software developers who do not understand the importance of detecting and removing faults before customers experience them as failures. It belongs to a legal system that has given the software developers a free pass on error-related damages. The blame also belongs to universities that stress more on software development than testing.

1.2 TESTING PROCESS

Testing is an important aspect of the software development life cycle. It is basically the process of testing the newly developed software, prior to its actual use. The program is executed with desired input(s) and the output(s) is/are observed accordingly. The observed output(s) is/are compared with expected output(s). If both are same, then the program is said to be correct as per specifications, otherwise there is something wrong somewhere in the program. Testing is a very expensive process and consumes one-third to one-half of the cost of a typical development project. It is largely a systematic process but partly intuitive too. Hence, good testing process entails much more than just executing a program a few times to see its correctness.

1.2.1 What is Software Testing?

Good testing entails more than just executing a program with desired input(s). Let's consider a program termed as 'Minimum' (see Figure 1.1) that reads a set of integers and prints the smallest integer. We may execute this program using Turbo C complier with a number of inputs and compare the expected output with the observed output as given in Table 1.1.

```
LINE NUMBER   /*SOURCE CODE*/
              #include<stdio.h>
              #include<limits.h>
              #include<conio.h>
1.            void Minimum();
2.            void main()
3.            {
4.                    Minimum();
5.            }
6.            void Minimum()
7.            {
8.                    int array[100];
9.                    int Number;
10.                   int i;
11.                   int tmpData;
12.                   int Minimum=INT_MAX;
13.                   clrscr();
14.                   "printf("Enter the size of the array:");
15.                   scanf("%d",&Number);
16.                   for(i=0;i<Number;i++) {
17.                           printf("Enter A[%d]=",i+1);
18.                           scanf("%d",&tmpData);
19.                           tmpData=(tmpData<0)?-tmpData:tmpData;
20.                           array[i]=tmpData;
21.                   }
22.                   i=1;
23.                   while(i<Number-1) {
24.                           if(Minimum>array[i])
25.                           {
26.                                   Minimum=array[i];
27.                           }
28.                           i++;
29.                   }
30.                   printf("Minimum = %d\n", Minimum);
31.                   getch();
32.           }
```

Figure 1.1. Program 'Minimum' to find the smallest integer out of a set of integers

Table 1.1. Inputs and outputs of the program 'Minimum'

Test Case	Inputs		Expected Output	Observed Output	Match?
	Size	Set of Integers			
1.	5	6, 9, 2, 16, 19	2	2	Yes
2.	7	96, 11, 32, 9, 39, 99, 91	9	9	Yes
3.	7	31, 36, 42, 16, 65, 76, 81	16	16	Yes
4.	6	28, 21, 36, 31, 30, 38	21	21	Yes
5.	6	106, 109, 88, 111, 114, 116	88	88	Yes
6.	6	61, 69, 99, 31, 21, 69	21	21	Yes
7.	4	6, 2, 9, 5	2	2	Yes
8.	4	99, 21, 7, 49	7	7	Yes

There are 8 sets of inputs in Table 1.1. We may feel that these 8 test cases are sufficient for such a trivial program. In all these test cases, the observed output is the same as the expected output. We may also design similar test cases to show that the observed output is matched with the expected output. There are many definitions of testing. A few of them are given below:

(i) Testing is the process of demonstrating that errors are not present.
(ii) The purpose of testing is to show that a program performs its intended functions correctly.
(iii) Testing is the process of establishing confidence that a program does what it is supposed to do.

The philosophy of all three definitions is to demonstrate that the given program behaves as per specifications. We may write 100 sets of inputs for the program 'Minimum' and show that this program behaves as per specifications. However, all three definitions are not correct. They describe almost the opposite of what testing should be viewed as. Forgetting the definitions for the moment, whenever we want to test a program, we want to establish confidence about the correctness of the program. Hence, our objective should not be to show that the program works as per specifications. But, we should do testing with the assumption that there are faults and our aim should be to remove these faults at the earliest. Thus, a more appropriate definition is [MYER04]: **"Testing is the process of executing a program with the intent of finding faults."** Human beings are normally goal oriented. Thus, establishment of a proper objective is essential for the success of any project. If our objective is to show that a program has no errors, then we shall sub-consciously work towards this objective. We shall intend to choose those inputs that have a low probability of making a program fail as we have seen in Table 1.1, where all inputs are purposely selected to show that the program is absolutely correct. On the contrary, if our objective is to show that a program has errors, we may select those test cases which have a higher probability of finding errors. We shall focus on weak and critical portions of the program to find more errors. This type of testing will be more useful and meaningful.

We again consider the program 'Minimum' (given in Figure 1.1) and concentrate on some typical and critical situations as discussed below:

(i) A very short list (of inputs) with the size of 1, 2, or 3 elements.
(ii) An empty list i.e. of size 0.

(iii) A list where the minimum element is the first or last element.
(iv) A list where the minimum element is negative.
(v) A list where all elements are negative.
(vi) A list where some elements are real numbers.
(vii) A list where some elements are alphabetic characters.
(viii) A list with duplicate elements.
(ix) A list where one element has a value greater than the maximum permissible limit of an integer.

We may find many similar situations which may be very challenging and risky for this program and each such situation should be tested separately. In Table 1.1, we have selected elements in every list to cover essentially the same situation: a list of moderate length, containing all positive integers, where the minimum is somewhere in the middle. Table 1.2 gives us another view of the same program 'Minimum' and the results are astonishing to everyone. It is clear from the outputs that the program has many failures.

Table 1.2. Some critical/typical situations of the program 'Minimum'

S. No.		Inputs		Expected Output	Observed Output	Match?
		Size	Set of Integers			
Case 1						
A very short list with size 1, 2 or 3	A	1	90	90	2147483647	**No**
	B	2	12, 10	10	2147483647	**No**
	C	2	10, 12	10	2147483647	**No**
	D	3	12, 14, 36	12	14	**No**
	E	3	36, 14, 12	12	14	**No**
	F	3	14, 12, 36	12	12	**Yes**
Case 2						
An empty list, i.e. of size 0	A	0	–	Error message	2147483647	**No**
Case 3						
A list where the minimum element is the first or last element	A	5	10, 23, 34, 81, 97	10	23	**No**
	B	5	97, 81, 34, 23, 10	10	23	**No**
Case 4						
A list where the minimum element is negative	A	4	10, –2, 5, 23	–2	2	**No**
	B	4	5, –25, 20, 36	–25	20	**No**
Case 5						
A list where all elements are negative	A	5	–23, –31, –45, –56, –78	–78	31	**No**
	B	5	–6, –203, –56, –78, –2	–203	56	**No**
Case 6						
A list where some elements are real numbers	A	5	12, 34.56, 6.9, 62.14, 19	6.9	34 (The program does not take values for index 3,4 and 5)	**No**
	B	5.4	2, 3, 5, 6, 9	2	858993460 (The program does not take any array value)	**No**

(*Contd.*)

(*Contd.*)

S. No.		Inputs		Expected Output	Observed Output	Match?
		Size	Set of Integers			
Case 7						
A list where some elements are characters	A	5	23, 2l, 26, 6, 9	6	2 (The program does not take any other index value for 3, 4 and 5)	**No**
	B	1l	2, 3, 4, 9, 6, 5, 11, 12, 14, 21, 22	2	2147483647 (Program does not take any other index value)	**No**
Case 8						
A list with duplicate elements	A	5	3, 4, 6, 9, 6	3	4	**No**
	B	5	13, 6, 6, 9, 15	6	6	**Yes**
Case 9						
A list where one element has a value greater than the maximum permissible limit of an integer	A	5	530, 4294967297, 23, 46, 59	23	1	**No**

What are the possible reasons for so many failures shown in Table 1.3? We should read our program 'Minimum' (given in Figure 1.1) very carefully to find reasons for so many failures. The possible reasons of failures for all nine cases discussed in Table 1.2 are given in Table 1.3. It is clear from Table 1.3 that this program suffers from serious design problems. Many important issues are not handled properly and therefore, we get strange observed outputs. The causes of getting these particular values of observed outputs are given in Table 1.4.

Table 1.3. Possible reasons of failures for all nine cases

S. No.	Possible Reasons
Case 1	
A very short list with size 1, 2 or 3	While finding the minimum, the base value of the index and/or end value of the index of the usable array has not been handled properly (see line numbers 22 and 23).
Case 2	
An empty list i.e. of size 0	The program proceeds without checking the size of the array (see line numbers 15 and 16).
Case 3	
A list where the minimum element is the first or last element	Same as for Case 1.
Case 4	
A list where the minimum element is negative	The program converts all negative integers into positive integers (see line number 19).
Case 5	
A list where all elements are negative	Same as for Case 4.

(*Contd.*)

(*Contd.*)

S. No.	Possible Reasons
Case 6 A list where some elements are real numbers	The program uses scanf() function to read the values. The scanf() has unpredictable behaviour for inputs not according to the specified format. (See line numbers 15 and 18).
Case 7 A list where some elements are alphabetic characters	Same as for Case 6.
Case 8 A list with duplicate elements	(a) Same as for Case 1. (b) We are getting the correct result because the minimum value is in the middle of the list and all values are positive.
Case 9 A list with one value greater than the maximum permissible limit of an integer	This is a hardware dependent problem. This is the case of the overflow of maximum permissible value of the integer. In this example, 32 bits integers are used.

Table 1.4. Reasons for observed output

Cases	Observed Output	Remarks
1 (a)	2147483647	The program has ignored the first and last values of the list. This is the maximum value of a 32 bit integer to which a variable minimum is initialized.
1 (b)	2147483647	
1 (c)	2147483647	
1 (d)	14	The program has ignored the first and last values of the list. The middle value is 14.
1 (e)	14	
1 (f)	12	The program has ignored the first and last value of the list. Fortunately, the middle value is the minimum value and thus the result is correct.
2 (a)	2147483647	The maximum value of a 32 bit integer to which a variable minimum is initialized.
3 (a)	23	The program has ignored the first and last values of the list. The value 23 is the minimum value in the remaining list.
3 (b)	23	
4 (a)	2	The program has ignored the first and last values. It has also converted negative integer(s) to positive integer(s).
4 (b)	20	
5 (a)	31	Same as Case 4.
5 (b)	56	
6 (a)	34	After getting '.' of 34.56, the program was terminated and 34 was displayed. However, the program has also ignored 12, being the first index value.
6 (b)	858993460	Garbage value.
7 (a)	2	After getting 'l' in the second index value '2l', the program terminated abruptly and displayed 2.
7 (b)	2147483647	The input has a non digit value. The program displays the value to which variable 'minimum' is initialized.
8 (a)	4	The program has ignored the first and last index values. 4 is the minimum in the remaining list.
8 (b)	6	Fortunately the result is correct although the first and last index values are ignored.
9 (a)	1	The program displays this value due to the overflow of the 32 bit signed integer data type used in the program.

Modifications in the program 'Minimum'

Table 1.4 has given many reasons for undesired outputs. These reasons help us to identify the causes of such failures. Some important reasons are given below:

(i) The program has ignored the first and last values of the list
The program is not handling the first and last values of the list properly. If we see the line numbers 22 and 23 of the program, we will identify the causes. There are two faults. Line number 22 "i = 1;" should be changed to "i = 0;" in order to handle the first value of the list. Line number 23 "while (i<Number -1)" should be changed to "while (i<=Number-1)" in order to handle the last value of the list.

(ii) The program proceeds without checking the size of the array
If we see line numbers 14 and 15 of the program, we will come to know that the program is not checking the size of the array / list before searching for the minimum value. A list cannot be of zero or negative size. If the user enters a negative or zero value of size or value greater than the size of the array, an appropriate message should be displayed. Hence after line number 15, the value of the size should be checked as under:

```
if (Number < = 0||Number>100)
{
        printf ("Invalid size specified");
}
```

If the size is greater than zero and lesser than 101, then the program should proceed further, otherwise it should be terminated.

(iii) Program has converted negative values to positive values
Line number 19 is converting all negative values to positive values. That is why the program is not able to handle negative values. We should delete this line to remove this fault.
The modified program, based on the above three points is given in Figure 1.2. The nine cases of Table 1.2 are executed on this modified program and the results are given in Table 1.5.

```
LINE NUMBER     /*SOURCE CODE*/
                #include<stdio.h>
                #include<limits.h>
                #include<conio.h>
1.              void Minimum();
2.              void main()
3.              {
4.                      Minimum();
5.              }
6.              void Minimum()
7.              {
8.                      int array[100];
9.                      int Number;
```

(*Contd.*)

(*Contd.*)

```
10.                    int i;
11.                    int tmpData;
12.                    int Minimum=INT_MAX;
13.                    clrscr();
14.                    printf("Enter the size of the array:");
15.                    scanf("%d",&Number);
16.                    if(Number<=0||Number>100) {
17.                    printf("Invalid size specified");
18.                    }
19.                    else {
20.                    for(i=0;i<Number;i++) {
21.                              printf("Enter A[%d]=",i+1);
22.                              scanf("%d",&tmpData);
23.                              /*tmpData=(tmpData<0)?-tmpData:tmpData;*/
24.                              array[i]=tmpData;
25.                    }
26.                    i=0;
27.                    while(i<=Number-1) {
28.                              if(Minimum>array[i])
29.                              {
30.                                        Minimum=array[i];
31.                              }
32.                              i++;
33.                    }
34.                    printf("Minimum = %d\n", Minimum);
35.                    }
36.                    getch();
37.          }
```

Figure 1.2. Modified program 'Minimum' to find the smallest integer out of a set of integers

Table 1.5 gives us some encouraging results. Out of 9 cases, only 3 cases are not matched. Six cases have been handled successfully by the modified program given in Figure 1.2. The cases 6 and 7 are failed due to the scanf() function parsing problem. There are many ways to handle this problem. We may design a program without using scanf() function at all. However, scanf() is a very common function and all of us use it frequently. Whenever any value is given using scanf() which is not as per specified format, scanf() behaves very notoriously and gives strange results. It is advisable to display a warning message for the user before using the scanf() function. The warning message may compel the user to enter values in the specified format only. If the user does not do so, he/she may have to suffer the consequences accordingly. The case 9 problem is due to the fixed maximal size of the integers in the machine and the language used. This also has to be handled through a warning message to the user. The further modified program based on these observations is given in the Figure 1.3.

Table 1.5. Results of the modified program 'Minimum'

Sr. No.		Inputs		Expected Output	Observed Output	Match?
		Size	Set of Integers			
Case 1						
A very short list with size 1, 2 or 3	A	1	90	90	90	Yes
	B	2	12, 10	10	10	Yes
	C	2	10, 12	10	10	Yes
	D	3	12, 14, 36	12	12	Yes
	E	3	36, 14, 12	12	12	Yes
	F	3	14, 12, 36	12	12	Yes
Case 2						
An empty list, i.e. of size 0	A	0	–	Error message	Error message	Yes
Case 3						
A list where the minimum element is the first or last element	A	5	10, 23, 34, 81, 97	10	10	Yes
	B	5	97, 81, 34, 23, 10	10	10	Yes
Case 4						
A list where the minimum element is negative	A	4	10, –2, 5, 23	–2	–2	Yes
	B	4	5, –25, 20, 36	–25	–25	Yes
Case 5						
A list where all elements are negative	A	5	–23, –31, –45, –56, –78	–78	–78	Yes
	B	5	–6, –203, –56, –78, –2	–203	–203	Yes
Case 6						
A list where some elements are real numbers	A	5	12, 34.56, 6.9, 62.14, 19	6.9	34	**No**
	B	5.4	2, 3, 5, 6, 9	2	858993460	**No**
Case 7						
A list where some elements are alphabetic characters	A	5	23, 2l, 26, 6, 9	6	2	**No**
	B	1l	2, 3, 4, 9, 6, 5, 11, 12, 14, 21, 22	2	858993460	**No**
Case 8						
A list with duplicate elements	A	5	3,4,6,9, 6	3	3	Yes
	B	5	13, 6, 6, 9, 15	6	6	Yes
Case 9						
A list where one element has a value greater than the maximum permissible limit of an integer	A	5	530, 42949672 97, 23, 46, 59	23	1	**No**

```
LINE NUMBER     /*SOURCE CODE*/
                #include<stdio.h>
                #include<limits.h>
                #include<conio.h>
1.              void Minimum();
2.              void main()
3.              {
4.                      Minimum();
5.              }
6.              void Minimum()
7.              {
8.                      int array[100];
9.                      int Number;
10.                     int i;
11.                     int tmpData;
12.                     int Minimum=INT_MAX;
13.                     clrscr();
14.                     printf("Enter the size of the array:");
15.                     scanf("%d",&Number);
16.                     if(Number<=0||Number>100) {
17.                             printf("Invalid size specified");
18.                     }
19.                     else {
20.                     printf("Warning: The data entered must be a valid integer and
                        must be between %d to %d, INT_MIN, INT_MAX\n");
21.                     for(i=0;i<Number;i++) {
22.                             printf("Enter A[%d]=",i+1);
23.                             scanf("%d",&tmpData);
24.                             /*tmpData=(tmpData<0)?-tmpData:tmpData;*/
25.                             array[i]=tmpData;
26.                     }
27.                     i=0;
28.                     while(i<=Number-1) {
29.                             if(Minimum>array[i])
30.                             {
31.                                     Minimum=array[i];
32.                             }
33.                             i++;
34.                     }
35.                     printf("Minimum = %d\n", Minimum);
36.                     }
37.                     getch();
38.             }
```

Figure 1.3. Final program 'Minimum' to find the smallest integer out of a set of integers

Our goal is to find critical situations of any program. Test cases shall be designed for every critical situation in order to make the program fail in such situations. If it is not possible to remove a fault then proper warning messages shall be given at proper places in the program. The aim of the best testing person should be to fix most of the faults. This is possible only if our intention is to show that the program does not work as per specifications. Hence, as given earlier, the most appropriate definition is **"Testing is the process of executing a program with the intent of finding faults."** Testing never shows the absence of faults, but it shows that the faults are present in the program.

1.2.2 Why Should We Test?

Software testing is a very expensive and critical activity; but releasing the software without testing is definitely more expensive and dangerous. No one would like to do it. It is like running a car without brakes. Hence testing is essential; but how much testing is required? Do we have methods to measure it? Do we have techniques to quantify it? The answer is not easy. All projects are different in nature and functionalities and a single yardstick may not be helpful in all situations. It is a unique area with altogether different problems.

The programs are growing in size and complexity. The most common approach is 'code and fix' which is against the fundamental principles of software engineering. Watts S. Humphrey, of Carnegie Mellon University [HUMP02] conducted a multiyear study of 13000 programs and concluded that "On average professional coders make 100 to 150 errors in every thousand lines of code they write." The C. Mann [MANN02] used Humphrey's figures on the business operating system Windows NT 4 and gave some interesting observations: "Windows NT 4 code size is of 16 million lines. Thus, this would have been written with about two million mistakes. Most would have been too small to have any effect, but some thousands would have caused serious problems. Naturally, Microsoft exhaustively tested Windows NT 4 before release, but in almost any phase of tests, they would have found less than half the defects. If Microsoft had gone through four rounds of testing, an expensive and time consuming procedure, the company would have found at least 15 out of 16 bugs. This means five defects per thousand lines of code are still remaining. This is very low. But the software would still have (as per study) as many as 80,000 defects."

The basic issue of this discussion is that we cannot release a software system without adequate testing. The study results may not be universally applicable but, at least, they give us some idea about the depth and seriousness of the problem. When to release the software is a very important decision. Economics generally plays an important role. We shall try to find more errors in the early phases of software development. The cost of removal of such errors will be very reasonable as compared to those errors which we may find in the later phases of software development. The cost to fix errors increases drastically from the specification phase to the test phase and finally to the maintenance phase as shown in Figure 1.4.

If an error is found and fixed in the specification and analysis phase, it hardly costs anything. We may term this as '1 unit of cost' for fixing an error during specifications and analysis phase. The same error, if propagated to design, may cost 10 units and if, further propagated to coding, may cost 100 units. If it is detected and fixed during the testing phase, it may lead to 1000 units of cost. If it could not be detected even during testing and is found by the customer after release, the cost becomes very high. We may not be able to predict the cost of failure for

a life critical system's software. The world has seen many failures and these failures have been costly to the software companies.

The fact is that we are releasing software that is full of errors, even after doing sufficient testing. No software would ever be released by its developers if they are asked to certify that the software is free of errors. Testing, therefore, continues to the point where it is considered that the cost of testing processes significantly outweighs the returns.

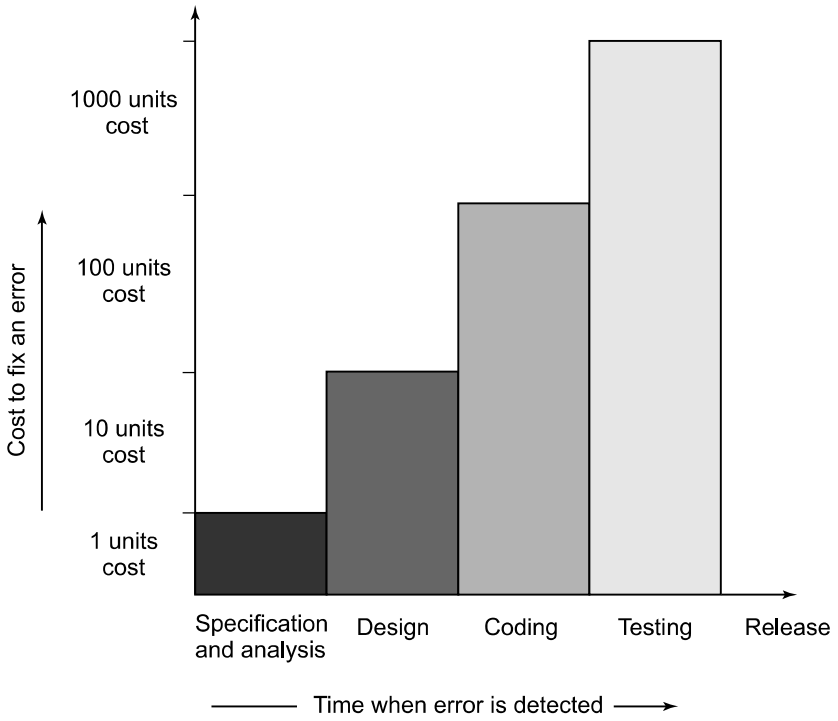


Figure 1.4. Phase wise cost of fixing an error

1.2.3 Who Should We Do the Testing?

Testing a software system may not be the responsibility of a single person. Actually, it is a team work and the size of the team is dependent on the complexity, criticality and functionality of the software under test. The software developers should have a reduced role in testing, if possible. The concern here is that the developers are intimately involved with the development of the software and thus it is very difficult for them to point out errors from their own creations. Beizer [BE1Z90] explains this situation effectively when he states, "There is a myth that if we were really good at programming, there would be no bugs to catch. If we could really concentrate; if everyone used structured programming, top down design, decision figures; if programs were written in SQUISH; if we had the right silver bullets, then there would be no bugs. So goes the myth. There are bugs, the myth says because we are bad at what we do; and if we are bad at it, we should feel guilty about it. Therefore, testing and test design amount to an admission of failures, which instils a goodly dose of guilt. The tedium of testing is just punishment for our errors. Punishment for what? For being human? Guilt for what? For not achieving human perfection? For not being able to distinguish between what another developer thinks and what he says? For not being telepathic? For not solving human communication problems that have been kicked around by philosophers and theologians for 40 centuries."

The testing persons must be cautious, curious, critical but non-judgmental and good communicators. One part of their job is to ask questions that the developers might not be able to ask themselves or are awkward, irritating, insulting or even threatening to the developers. Some of the questions are [BENT04]:

(i) How is the software?
(ii) How good is it?
(iii) How do you know that it works? What evidence do you have?
(iv) What are the critical areas?
(v) What are the weak areas and why?
(vi) What are serious design issues?
(vii) What do you feel about the complexity of the source code?

The testing persons use the software as heavily as an expert user on the customer side. User testing almost invariably recruits too many novice users because they are available and the software must be usable by them. The problem is that the novices do not have domain knowledge that the expert users have and may not recognize that something is wrong.

Many companies have made a distinction between development and testing phases by making different people responsible for each phase. This has an additional advantage. Faced with the opportunity of testing someone else's software, our professional pride will demand that we achieve success. Success in testing is finding errors. We will therefore strive to reveal any errors present in the software. In other words, our ego would have been harnessed to the testing process, in a very positive way, in a way, which would be virtually impossible, had we been testing our own software [NORM89]. Therefore, most of the times, the testing persons are different from development persons for the overall benefit of the system. The developers provide guidelines during testing; however, the overall responsibility is owned by the persons who are involved in testing. Roles of the persons involved during development and testing are given in Table 1.6.

Table 1.6. Persons and their roles during development and testing

S. No.	Persons	Roles
1.	Customer	Provides funding, gives requirements, approves changes and some test results.
2.	Project Manager	Plans and manages the project.
3.	Software Developer(s)	Designs, codes and builds the software; participates in source code reviews and testing; fixes bugs, defects and shortcomings.
4.	Testing co-ordinator(s)	Creates test plans and test specifications based on the requirements and functional and technical documents.
5.	Testing person(s)	Executes the tests and documents results.

1.2.4 What Should We Test?

Is it possible to test the program for all possible valid and invalid inputs? The answer is always negative due to a large number of inputs. We consider a simple example where a program has two 8 bit integers as inputs. Total combinations of inputs are $2^8 \times 2^8$. If only one second is

required (possible only with automated testing) to execute one set of inputs, it may take 18 hours to test all possible combinations of inputs. Here, invalid test cases are not considered which may also require a substantial amount of time. In practice, inputs are more than two and the size is also more than 8 bits. What will happen when inputs are real and imaginary numbers? We may wish to go for complete testing of the program, which is neither feasible nor possible. This situation has made this area very challenging where the million dollar question is, "How to choose a reasonable number of test cases out of a large pool of test cases?" Researchers are working very hard to find the answer to this question. Many testing techniques attempt to provide answers to this question in their own ways. However, we do not have a standard yardstick for the selection of test cases.

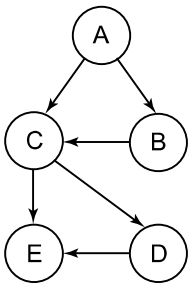
We all know the importance of this area and expect some drastic solutions in the future. We also know that every project is a new project with new expectations, conditions and constraints. What is the bottom line for testing? At least, we may wish to touch this bottom line, which may incorporate the following:

(i) Execute every statement of the program at least once.
(ii) Execute all possible paths of the program at least once.
(iii) Execute every exit of the branch statement at least once.

This bottom line is also not easily achievable. Consider the following piece of source code:

```
1. if (x > 0)
2. {
3. a = a + b;
4. }
5. if (y>10)
6. {
7. c=c+d;
8. }
```

This code can be represented graphically as:



Line Numbers	Symbol for representation
1	A
2, 3, 4	B
5	C
6, 7, 8	D
End	E

The possible paths are: ACE, ABCE, ACDE and ABCDE. However, if we choose x = 9 and y = 15, all statements are covered. Hence only one test case is sufficient for 100% statement coverage by traversing only one path ABCDE. Therefore, 100% statement coverage may not be sufficient, even though that may be difficult to achieve in real life programs.

Myers [MYER04] has given an example in his book entitled "The art of software testing" which shows that the number of paths is too large to test. He considered a control flow graph (as given in Figure 1.5) of a 10 to 20 statement program with 'DO Loop' that iterates up to 20 times. Within 'DO Loop' there are many nested 'IF' statements. The assumption is that all decisions in the program are independent of each other. The number of unique paths is nothing but the number of unique ways to move from point X to point Y. Myers further stated that executing every statement of the program at least once may seem to be a reasonable goal. However many portions of the program may be missed with this type of criteria.

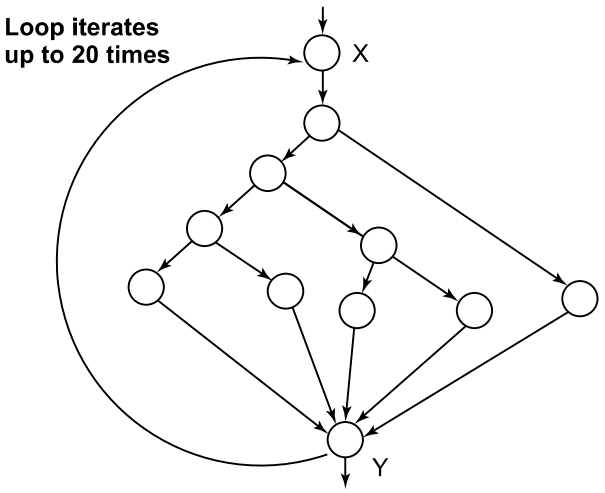


Figure 1.5. Control flow graph of a 10 to 20 statement program [MYER04]

"The total number of paths is approximately 10^{14} or 100 trillion. It is computed from $5^{20} + 5^{19} + \ldots\ldots\ldots 5^{1}$, where 5 is the number of independent paths of the control flow graph. If we write, execute and verify a test case every five minutes, it would take approximately one billion years to try every path. If we are 300 times faster, completing a test case one per second, we could complete the job in 3.2 million years." This is an extreme situation; however, in reality, all decisions are not independent. Hence, the total paths may be less than the calculated paths. But real programs are much more complex and larger in size. Hence, 'testing all paths' is very difficult if not impossible to achieve.

We may like to test a program for all possible valid and invalid inputs and furthermore, we may also like to execute all possible paths; but practically, it is quite difficult. Every exit condition of a branch statement is similarly difficult to test due to a large number of such conditions. We require effective planning, strategies and sufficient resources even to target the minimum possible bottom line. We should also check the program for very large numbers, very small numbers, numbers that are close to each other, negative numbers, some extreme cases, characters, special letters, symbols and some strange cases.

1.3 SOME TERMINOLOGIES

Some terminologies are discussed in this section, which are inter-related and confusing but commonly used in the area of software testing.

1.3.1 Program and Software

Both terms are used interchangeably, although they are quite different. The software is the superset of the program(s). It consists of one or many program(s), documentation manuals and operating procedure manuals. These components are shown in Figure 1.6.

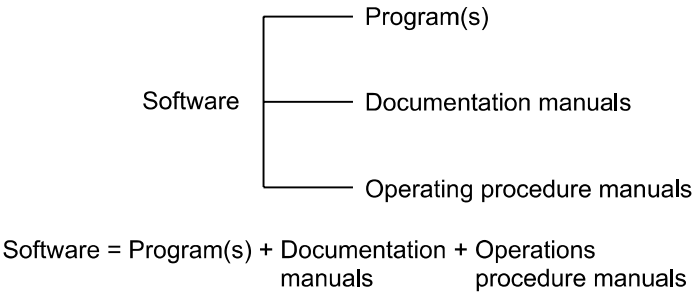


Figure 1.6. Components of the software

The program is a combination of source code and object code. Every phase of the software development life cycle requires preparation of a few documentation manuals which are shown in Figure 1.7. These are very helpful for development and maintenance activities.

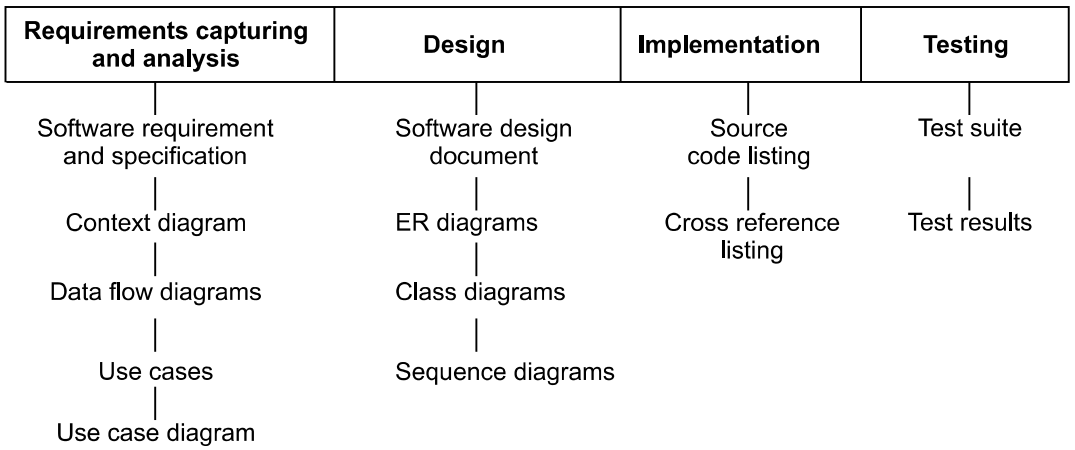


Figure 1.7. Documentation manuals

Operating procedure manuals consist of instructions to set up, install, use and to maintain the software. The list of operating procedure manuals / documents is given in Figure 1.8.

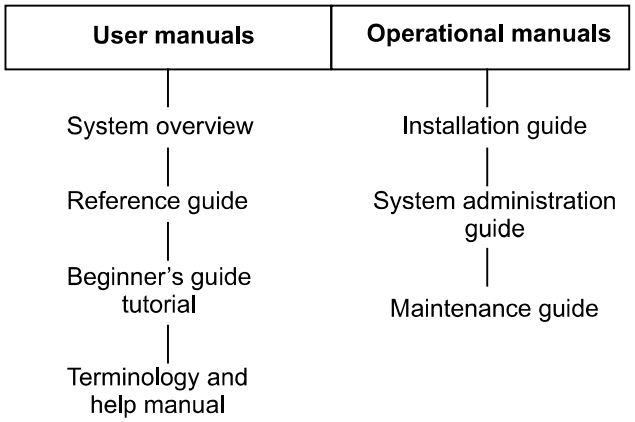


Figure 1.8. Operating system manuals

1.3.2 Verification and Validation

These terms are used interchangeably and some of us may also feel that both are synonyms. The Institute of Electrical and Electronics Engineers (IEEE) has given definitions which are largely accepted by the software testing community. Verification is related to static testing which is performed manually. We only inspect and review the document. However, validation is dynamic in nature and requires the execution of the program.

Verification: As per IEEE [IEEE01], "It is the process of evaluating the system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase." We apply verification activities from the early phases of the software development and check / review the documents generated after the completion of each phase. Hence, it is the process of reviewing the requirement document, design document, source code and other related documents of the project. This is manual testing and involves only looking at the documents in order to ensure what comes out is what we expected to get.

Validation: As per IEEE [IEEE01], "It is the process of evaluating a system or component during or at the end of development process to determine whether it satisfies the specified requirements." It requires the actual execution of the program. It is dynamic testing and requires a computer for execution of the program. Here, we experience failures and identify the causes of such failures.
Hence, testing includes both verification and validation. Thus

$$\text{Testing} = \text{Verification} + \text{Validation}$$

Both are essential and complementary activities of software testing. If effective verification is carried out, it may minimize the need of validation and more number of errors may be detected in the early phases of the software development. Unfortunately, testing is primarily validation oriented.

1.3.3 Fault, Error, Bug and Failure

All terms are used interchangeably although error, mistake and defect are synonyms in software testing terminology. When we make an error during coding, we call this a 'bug'. Hence, error / mistake / defect in coding is called a bug.

A fault is the representation of an error where representation is the mode of expression such as data flow diagrams, ER diagrams, source code, use cases, etc. If fault is in the source code, we call it a bug.

A failure is the result of execution of a fault and is dynamic in nature. When the expected output does not match with the observed output, we experience a failure. The program has to execute for a failure to occur. A fault may lead to many failures. A particular fault may cause different failures depending on the inputs to the program.

1.3.4 Test, Test Case and Test Suite

Test and test case terms are synonyms and may be used interchangeably. A test case consists of inputs given to the program and its expected outputs. Inputs may also contain pre-condition(s) (circumstances that hold prior to test case execution), if any, and actual inputs identified by some testing methods. Expected output may contain post-condition(s) (circumstances after the execution of a test case), if any, and outputs which may come as a result when selected inputs are given to the software. Every test case will have a unique identification number. When we do testing, we set desire pre-condition(s), if any, given selected inputs to the program and note the observed output(s). We compare the observed output(s) with the expected output(s) and if they are the same, the test case is successful. If they are different, that is the failure condition with selected input(s) and this should be recorded properly in order to find the cause of failure. A good test case has a high probability of showing a failure condition. Hence, test case designers should identify weak areas of the program and design test cases accordingly. The template for a typical test case is given in Table 1.7.

Table 1.7. Test case template

	Test Case Identification Number:
	Part I (Before Execution)
1.	Purpose of test case:
2.	Pre-condition(s): (optional)
3.	Input(s) :
4.	Expected Output(s) :
5.	Post-condition(s) :
6.	Written by :
7.	Date of design :
	Part II (After Execution)
1.	Output(s) :
2.	Post-condition(s) : (optional)

(*Contd.*)

(*Contd.*)

	Part II (After Execution)
3.	Pass / fail :
4.	If fails, any possible reason of failure (optional) :
5.	Suggestions (optional)
6.	Run by :
7.	Date of suggestion :

The set of test cases is called a test suite. We may have a test suite of all test cases, test suite of all successful test cases and test suite of all unsuccessful test cases. Any combination of test cases will generate a test suite. All test suites should be preserved as we preserve source code and other documents. They are equally valuable and useful for the purpose of maintenance of the software. Sometimes test suite of unsuccessful test cases gives very important information because these are the test cases which have made the program fail in the past.

1.3.5 Deliverables and Milestones

Different deliverables are generated during various phases of the software development. The examples are source code, Software Requirements and Specification document (SRS), Software Design Document (SDD), Installation guide, user reference manual, etc.

The milestones are the events that are used to ascertain the status of the project. For instance, finalization of SRS is a milestone; completion of SDD is another milestone. The milestones are essential for monitoring and planning the progress of the software development.

1.3.6 Alpha, Beta and Acceptance Testing

Customers may use the software in different and strange ways. Their involvement in testing may help to understand their minds and may force developers to make necessary changes in the software. These three terms are related to the customer's involvement in testing with different meanings.

Acceptance Testing: This term is used when the software is developed for a specific customer. The customer is involved during acceptance testing. He/she may design adhoc test cases or well-planned test cases and execute them to see the correctness of the software. This type of testing is called acceptance testing and may be carried out for a few weeks or months. The discovered errors are fixed and modified and then the software is delivered to the customer.

Alpha and Beta Testing: These terms are used when the software is developed as a product for anonymous customers. Therefore, acceptance testing is not possible. Some potential customers are identified to test the product. The alpha tests are conducted at the developer's site by the customer. These tests are conducted in a controlled environment and may start when the formal testing process is near completion. The beta tests are conducted by potential customers at their sites. Unlike alpha testing, the developer is not present here. It is carried out in an uncontrolled real life environment by many potential customers. Customers are expected to report failures, if any, to the company. These failure reports are studied by the developers and appropriate changes are made in the software. Beta tests have shown their advantages in the past and releasing a beta version of the software to the potential customer has become a

common practice. The company gets the feedback of many potential customers without making any payment. The other good thing is that the reputation of the company is not at stake even if many failures are encountered.

1.3.7 Quality and Reliability

Software reliability is one of the important factors of software quality. Other factors are understandability, completeness, portability, consistency, maintainability, usability, efficiency, etc. These quality factors are known as non-functional requirements for a software system.

Software reliability is defined as "the probability of failure free operation for a specified time in a specified environment" [ANSI91]. Although software reliability is defined as a probabilistic function and comes with the notion of time, it is not a direct function of time. The software does not wear out like hardware during the software development life cycle. There is no aging concept in software and it will change only when we intentionally change or upgrade the software.

Software quality determines how well the software is designed (quality of design), and how well the software conforms to that design (quality of conformance).

Some software practitioners also feel that quality and reliability is the same thing. If we are testing a program till it is stable, reliable and dependable, we are assuring a high quality product. Unfortunately, that is not necessarily true. Reliability is just one part of quality. To produce a good quality product, a software tester must verify and validate throughout the software development process.

1.3.8 Testing, Quality Assurance and Quality Control

Most of us feel that these terms are similar and may be used interchangeably. This creates confusion about the purpose of the testing team and Quality Assurance (QA) team. As we have seen in the previous section (1.2.1), the purpose of testing is to find faults and find them in the early phases of software development. We remove faults and ensure the correctness of removal and also minimize the effect of change on other parts of the software.

The purpose of QA activity is to enforce standards and techniques to improve the development process and prevent the previous faults from ever occurring. A good QA activity enforces good software engineering practices which help to produce good quality software. The QA group monitors and guides throughout the software development life cycle. This is a defect prevention technique and concentrates on the process of the software development. Examples are reviews, audits, etc.

Quality control attempts to build a software system and test it thoroughly. If failures are experienced, it removes the cause of failures and ensures the correctness of removal. It concentrates on specific products rather than processes as in the case of QA. This is a defect detection and correction activity which is usually done after the completion of the software development. An example is software testing at various levels.

1.3.9 Static and Dynamic Testing

Static testing refers to testing activities without executing the source code. All verification activities like inspections, walkthroughs, reviews, etc. come under this category of testing.

This, if started in the early phases of the software development, gives good results at a very reasonable cost. Dynamic testing refers to executing the source code and seeing how it performs with specific inputs. All validation activities come in this category where execution of the program is essential.

1.3.10 Testing and Debugging

The purpose of testing is to find faults and find them as early as possible. When we find any such fault, the process used to determine the cause of this fault and to remove it is known as debugging. These are related activities and are carried out sequentially.

1.4 LIMITATIONS OF TESTING

We want to test everything before giving the software to the customers. This 'everything' is very illusive and has many meanings. What do we understand when we say 'everything'? We may expect one, two or all of the following when we refer to 'everything':

(i) Execute every statement of the program
(ii) Execute every true and false condition
(iii) Execute every condition of a decision node
(iv) Execute every possible path
(v) Execute the program with all valid inputs
(vi) Execute the program with all invalid inputs

These six objectives are impossible to achieve due to time and resource constraints as discussed in section 1.2.4. We may achieve a few of them. If we do any compromise, we may miss a bug. Input domain is too large to test and there are too many paths in any program. Hence 'Everything' is impossible and we have to settle for 'less than everything' in real life situations. Some of the other issues which further make the situation more complex and complicated are given in the subsequent sub-sections.

1.4.1 Errors in the Software Requirement and Specification Document

These issues are very difficult to identify. If 6+9=20 is written in the SRS document and our program prints output as 20 when 6 and 9 are inputs, is it a bug? If the program prints output as 15, when inputs are 6 and 9, how can we interpret? In this case, the actual output is so obvious that interpretation may not require time to take a correct decision. But in most of the situations, outputs are not so obvious. Some requirements may be misunderstood and some may be missed. Ambiguities of natural languages (like English) may give more than one meaning to a sentence and make life difficult for testers. Hence, problems in writing good SRS have also become one of the problems of software testing.

1.4.2 Logical Bugs

How do we handle logical bugs? An interesting example is given in Figure 1.9. In this function, statement "d = c++;" given in line number 4 is incorrect. As per requirements, it should have

been "d = ++c"; but due to a typographical mistake and ignorance, "d = c++;" has been written. This is a logical error and cannot be detected by the compiler. Here, confusion is due to the use of prefix and postfix operators. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on the left. On the other hand, a postfix operator first assigns the value to the variable on the left and then increment the operand [BALA07]. In this function the postfix operator is used instead of the prefix operator. The function returns the integer value of 'flag'. If this function is executed on a 16 bit computer, the valid integer range for input 'c' is –32768 to 32767. Hence, there are 65536 possible inputs to this program. We may not like to create 65536 test cases. After all, who will execute those cases, if at all created, one fine day? Which input values are to be selected for the detection of this bug? Ten test cases have been given in Table 1.8 and none of them could detect this bug. How many test cases out of possible 65536 test cases will find this bug? What are the chances that we will select all those test cases or any one of them in order to find this bug? Only two test cases out of 65536 can detect this bug and are given in Table 1.9. This example shows the impossibility of testing 'everything'. If a small function can create so many problems, we may appreciate the problems of real life large and complex programs. Logical bugs are extremely difficult to handle and become one of the serious concerns of testing.

Software testing has inherent difficulties which is making it impossible to completely test the software. It can only show that bugs are in the software but it cannot show that bugs are not in the software at all. With all the limitations, software testing still is mandatory and a very useful filter to detect errors, which may further be removed. However we all know that good testing cannot make the software better, only good coding with software engineering principles makes the software better. However, good testing techniques may detect a good number of errors and their removal may improve the quality of the software.

```
1. int funct1 (int c)
2. {
3. int d, flag;
4. d = c ++ ; // should be d = ++ c; as per requirements
5. if (d < 20000)
6.          flag = 1 ;
7. else
8.          flag = 0;
9. return (flag);
10. }
```

Figure 1.9. A typical example

Table 1.8. Test cases for function of Figure 1.9

Test case	Input c	Expected output	Actual output
1.	0	1	1
2.	1	1	1
3.	20000	0	0
4.	30000	0	0

(*Contd.*)

(*Contd.*)

Test case	Input c	Expected output	Actual output
5.	–10000	1	1
6.	–20000	1	1
7.	–1	1	1
8.	–16000	1	1
9.	27000	0	0
10.	32000	0	0

Table 1.9. Typical test cases where outputs are different

Input c	Expected output	Actual output
19999	0	1
32767	Integer out of specified range	0

1.4.3 Difficult to Measure the Progress of Testing

How to measure the progress of testing? Normally we count various things to measure and interpret these counts. Is experiencing more failures good news or bad news? The answer could be either. A higher number of failures may indicate that testing was thorough and very few faults remain in the software. Or, it may be treated as an indication of poor quality of the software with lots of faults; even though many have been exposed, lots of them still remain. These counts may be illusive and may not help us to measure the progress of testing.

This difficulty of measuring the progress of testing leads to another issue i.e. when to stop testing and release the software to the customer(s)? This is a sensitive decision and should be based on the status of testing. However, in the absence of testing standards, 'economics', 'time to market' and 'gut feeling' have become important issues over technical considerations for the release of any software. Many models are available with serious limitations and are not universally acceptable.

Software companies are facing serious challenges in testing their products and these challenges are growing bigger as the software grows more complex. Hence, we should recognize the complex nature of testing and take it seriously. The gap between standards and practices should be reduced in order to test the software effectively which may result in to good quality software.

1.5 THE V SHAPED SOFTWARE LIFE CYCLE MODEL

The V shaped model is the modified form of the waterfall model with a special focus on testing activities. The waterfall model allows us to start testing activities after the completion of the implementation phase. This was popular when testing was primarily validation oriented. Now, there is a shift in testing activities from validation to verification where we want to review / inspect every activity of the software development life cycle. We want to involve the testing persons from the requirement analysis and specification phase itself. They will review the SRS document and identify the weak areas, critical areas, ambiguous areas and misrepresented areas. This will improve the quality of the SRS document and may further minimize the errors.

These verification activities are treated as error preventive exercises and are applied at requirements analysis and specification phase, high level design phase, detailed design phase and implementation phase. We not only want to improve the quality of the end products at all phases by reviews, inspections and walkthroughs, but also want to design test cases and test plans during these phases. The designing of test cases after requirement analysis and specification phase, high level design phase, detailed design phase and implementation phase may help us to improve the quality of the final product and also reduce the cost and development time.

1.5.1 Graphical Representation

The shape of the model is like the English letter 'V' and it emphasizes testing activities in every phase. There are two parts of the software development life cycle in this model i.e. development and testing and are shown in Figure 1.10. We want to carry out development and testing activities in parallel and this model helps us to do the same in order to reduce time and cost.

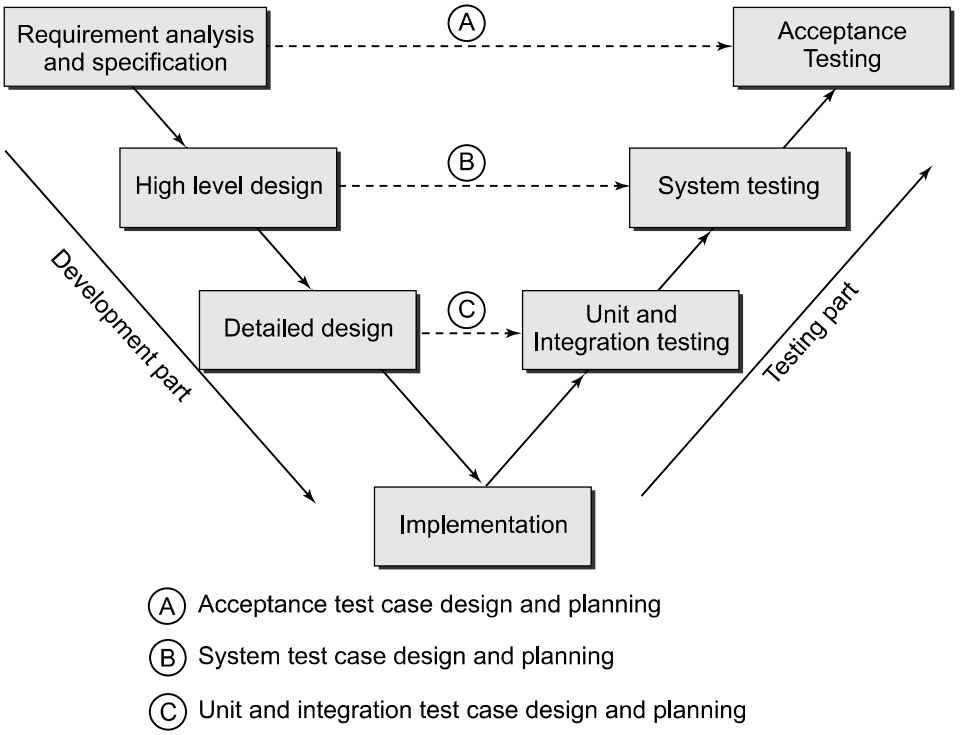


Figure 1.10. V shaped software development life cycle model

1.5.2 Relationship of Development and Testing Parts

The development part consists of the first four phases (i.e. requirements analysis and specification, high level design, detailed design and implementation) whereas the testing part has three phases (i.e. unit and integration testing, system testing and acceptance testing). The model establishes the relationship between the development and testing parts. The acceptance test case design and planning activities should be conducted along with the software requirements and specifications

phase. Similarly the system test case design and planning activities should be carried out along with high level design phase. Unit and integration test case design and planning activities should be carried out along with the detailed design phase. The development work is to be done by the development team and testing is to be done by the testing team simultaneously. After the completion of implementation, we will have the required test cases for every phase of testing. The only remaining work is to execute these test cases and observe the responses of the outcome of the execution. This model brings the quality into the development of our products. The encouragement of writing test cases and test plans in the earlier phases of the software development life cycle is the real strength of this model. We require more resources to implement this model as compared to the waterfall model. This model also suffers from many disadvantages of the waterfall model like non-availability of a working version of the product until late in the life cycle, difficulty in accommodating any change, etc. This model has also limited applications in today's interactive software processes.

MULTIPLE CHOICE QUESTIONS

Note: ***Select the most appropriate answer for the following questions.***

1.1 What is software testing?
- (a) It is the process of demonstrating that errors are not present.
- (b) It is the process of establishing confidence that a program does what it is supposed to do.
- (c) It is the process of executing a program with the intent of finding errors.
- (d) It is the process of showing the correctness of a program.

1.2 Why should testing be done?
- (a) To ensure the correctness of a program
- (b) To find errors in a program
- (c) To establish the reliability of a program
- (d) To certify the effectiveness of a program

1.3 Which phase consumes maximum effort to fix an error?
- (a) Requirements analysis and specifications
- (b) Design phase
- (c) Coding phase
- (d) Feasibility study phase

1.4 Which objective is most difficult to achieve?
- (a) Execute every statement of a program at least once
- (b) Execute every branch statement of a program at least once
- (c) Execute every path of a program at least once
- (d) Execute every condition of a branch statement of a program at least once

1.5 Software errors during coding are known as:
- (a) Bugs
- (b) Defects
- (c) Failures
- (d) Mistakes

2

Functional Testing

Software testing is very important but is an effort-consuming activity. A large number of test cases are possible and some of them may make the software fail. As we all know, if observed behaviour of the software is different from the expected behaviour, we treat this as a failure condition. Failure is a dynamic condition that always occurs after the execution of the software. Everyone is in search of such test cases which may make the software fail and every technique attempts to find ways to design those test cases which have a higher probability of showing a failure.

Functional testing techniques attempt to design those test cases which have a higher probability of making a software fail. These techniques also attempt to test every possible functionality of the software. Test cases are designed on the basis of functionality and the internal structure of the program is completely ignored. Observed output(s) is (are) compared with expected output(s) for selected input(s) with preconditions, if any. The software is treated as a black box and therefore, it is also known as black box testing as shown in Figure 2.1.

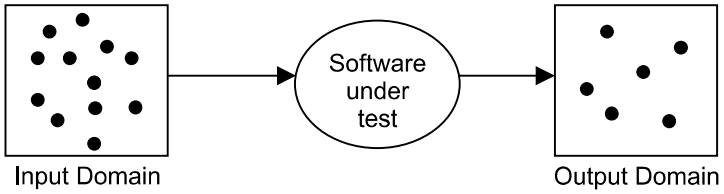


Figure 2.1. Functional (Black Box) testing

Every dot in the input domain represents a set of inputs and every dot in the output domain represents a set of outputs. Every set of input(s) will have a corresponding set of output(s). The test cases are designed on the basis of user requirements without considering the internal structure of the program. This black box knowledge is sufficient to design a good number of test cases. Many activities are performed in real life with only black box knowledge like

driving a car, using a cell phone, operating a computer, etc. In functional testing techniques, execution of a program is essential and hence these testing techniques come under the category of 'validation'. Here, both valid and invalid inputs are chosen to see the observed behaviour of the program. These techniques can be used at all levels of software testing like unit, integration, system and acceptance testing. They also help the tester to design efficient and effective test cases to find faults in the software.

2.1 BOUNDARY VALUE ANALYSIS

This is a simple but popular functional testing technique. Here, we concentrate on input values and design test cases with input values that are on or close to boundary values. Experience has shown that such test cases have a higher probability of detecting a fault in the software. Suppose there is a program 'Square' which takes 'x' as an input and prints the square of 'x'as output. The range of 'x' is from 1 to 100. One possibility is to give all values from 1 to 100 one by one to the program and see the observed behaviour. We have to execute this program 100 times to check every input value. In boundary value analysis, we select values on or close to boundaries and all input values may have one of the following:

(i) Minimum value
(ii) Just above minimum value
(iii) Maximum value
(iv) Just below maximum value
(v) Nominal (Average) value

These values are shown in Figure 2.2 for the program 'Square'.

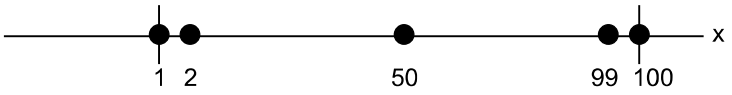


Figure 2.2. Five values for input 'x' of 'Square' program

These five values (1, 2, 50, 99 and 100) are selected on the basis of boundary value analysis and give reasonable confidence about the correctness of the program. There is no need to select all 100 inputs and execute the program one by one for all 100 inputs. The number of inputs selected by this technique is 4n + 1 where 'n' is the number of inputs. One nominal value is selected which may represent all values which are neither close to boundary nor on the boundary. Test cases for 'Square' program are given in Table 2.1.

Table 2.1. Test cases for the 'Square' program

Test Case	Input x	Expected output
1.	1	1
2.	2	4
3.	50	2500
4.	99	9801
5.	100	10000

Consider a program 'Addition' with two input values x and y and it gives the addition of x and y as an output. The range of both input values are given as:

$$100 \leq x \leq 300$$

$$200 \leq y \leq 400$$

The selected values for *x* and *y* are given in Figure 2.3.

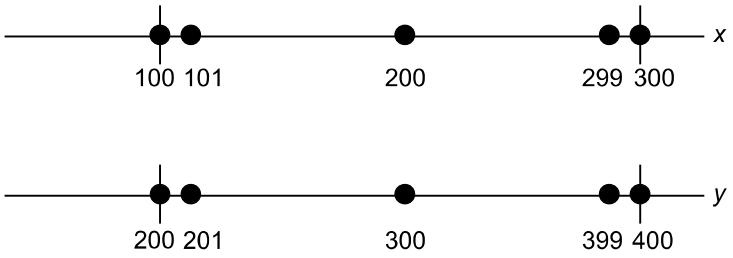


Figure 2.3. Selected values for input values x and y

The 'x' and 'y' inputs are required for the execution of the program. The input domain of this program 'Addition' is shown in Figure 2.4. Any point within the inner rectangle is a legitimate input to the program.

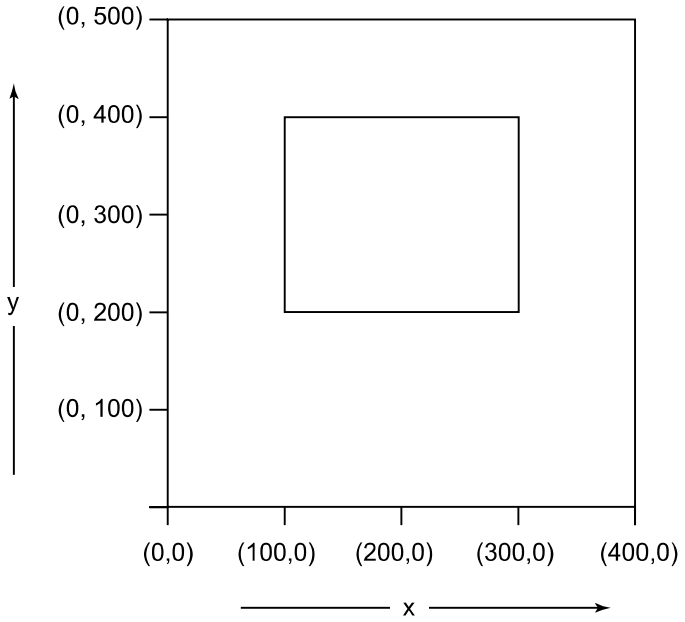


Figure 2.4. Valid input domain for the program 'Addition'

We also consider 'single fault' assumption theory of reliability which says that failures are rarely the result of the simultaneous occurrence of two (or more) faults. Normally, one fault is responsible for one failure. With this theory in mind, we select one input value on boundary (minimum), just above boundary (minimum $^{+}$), just below boundary (maximum $^{-}$), on boundary

(maximum), nominal (average) and other n-1 input values as nominal values. The inputs are shown graphically in Figure 2.5 and the test cases for 'Addition' program are given in Table 2.2.

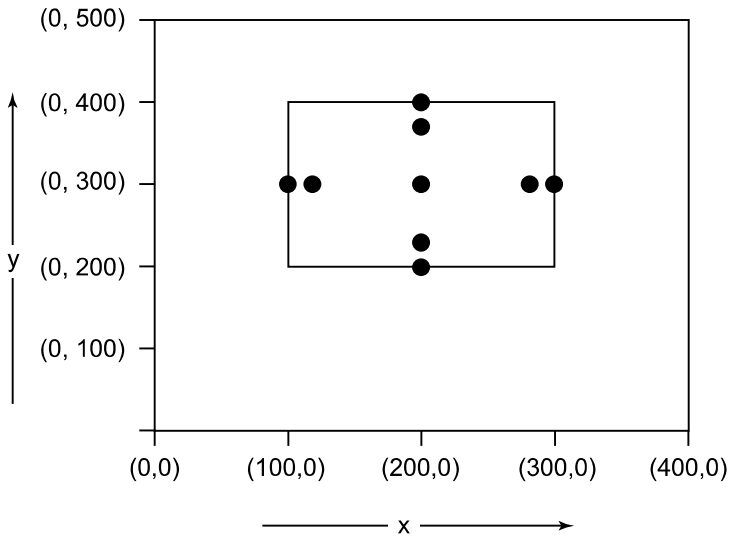


Figure 2.5. Graphical representation of inputs

Table 2.2. Test cases for the program 'Addition'

Test Case	x	y	Expected Output
1.	100	300	400
2.	101	300	401
3.	200	300	500
4.	299	300	599
5.	300	300	600
6.	200	200	400
7.	200	201	401
8.	200	300	500
9.	200	399	599
10.	200	400	600

In Table 2.2, two test cases are common (3 and 8), hence one must be selected. This technique generates 9 test cases where all inputs have valid values. Each dot of the Figure 2.5 represents a test case and inner rectangle is the domain of legitimate input values. Thus, for a program of 'n' variables, boundary value analysis yields 4n + 1 test cases.

Example 2.1: Consider a program for the determination of the largest amongst three numbers. Its input is a triple of positive integers (say x,y and z) and values are from interval [1, 300]. Design the boundary value test cases.

Solution: The boundary value test cases are given in Table 2.3.

Table 2.3. Boundary value test cases to find the largest among three numbers

Test Case	x	y	z	Expected output
1.	1	150	150	150
2.	2	150	150	150
3.	150	150	150	150
4.	299	150	150	299
5.	300	150	150	300
6.	150	1	150	150
7.	150	2	150	150
8.	150	299	150	299
9.	150	300	150	300
10.	150	150	1	150
11.	150	150	2	150
12.	150	150	299	299
13.	150	150	300	300

Example 2.2: Consider a program for the determination of division of a student based on the marks in three subjects. Its input is a triple of positive integers (say mark1, mark2, and mark3) and values are from interval [0, 100].
The division is calculated according to the following rules:

Marks Obtained (Average)	Division
75 – 100	First Division with distinction
60 – 74	First division
50 – 59	Second division
40 – 49	Third division
0 – 39	Fail

Total marks obtained are the average of marks obtained in the three subjects i.e.

$$\text{Average} = (\text{mark1} + \text{mark 2} + \text{mark3}) / 3$$

The program output may have one of the following words:
[Fail, Third Division, Second Division, First Division, First Division with Distinction]
Design the boundary value test cases.

Solution: The boundary value test cases are given in Table 2.4.

Table 2.4. Boundary value test cases for the program determining the division of a student

Test Case	mark1	mark2	mark3	Expected Output
1.	0	50	50	Fail
2.	1	50	50	Fail
3.	50	50	50	Second Division
4.	99	50	50	First Division
5.	100	50	50	First Division

(*Contd.*)

(*Contd.*)

Test Case	mark1	mark2	mark3	Expected Output
6.	50	0	50	Fail
7.	50	1	50	Fail
8.	50	99	50	First Division
9.	50	100	50	First Division
10.	50	50	0	Fail
11.	50	50	1	Fail
12.	50	50	99	First Division
13.	50	50	100	First Division

Example 2.3: Consider a program for classification of a triangle. Its input is a triple of positive integers (say a, b, c) and the input parameters are greater than zero and less than or equal to 100.
The triangle is classified according to the following rules:

Right angled triangle: $c^2 = a^2 + b^2$ or $a^2 = b^2 + c^2$ or $b^2 = c^2 + a^2$
Obtuse angled triangle: $c^2 > a^2 + b^2$ or $a^2 > b^2 + c^2$ or $b^2 > c^2 + a^2$
Acute angled triangle: $c^2 < a^2 + b^2$ and $a^2 < b^2 + c^2$ and $b^2 < c^2 + a^2$

The program output may have one of the following words:
[Acute angled triangle, Obtuse angled triangle, Right angled triangle, Invalid triangle]
Design the boundary value test cases.

Solution: The boundary value analysis test cases are given in Table 2.5.

Table 2.5. Boundary value test cases for triangle classification program

Test Case	a	b	c	Expected Output
1.	1	50	50	Acute angled triangle
2.	2	50	50	Acute angled triangle
3.	50	50	50	Acute angled triangle
4.	99	50	50	Obtuse angled triangle
5.	100	50	50	Invalid triangle
6.	50	1	50	Acute angled triangle
7.	50	2	50	Acute angled triangle
8.	50	99	50	Obtuse angled triangle
9.	50	100	50	Invalid triangle
10.	50	50	1	Acute angled triangle
11.	50	50	2	Acute angled triangle
12.	50	50	99	Obtuse angled triangle
13.	50	50	100	Invalid triangle

Example 2.4: Consider a program for determining the day of the week. Its input is a triple of day, month and year with the values in the range

$1 \leq$ month ≤ 12

$1 \leq$ day ≤ 31

$1900 \leq$ year ≤ 2058

The possible outputs would be the day of the week or invalid date. Design the boundary value test cases.

Solution: The boundary value test cases are given in Table 2.6.

Table 2.6. Boundary value test cases for the program determining the day of the week

Test Case	month	day	year	Expected Output
1.	1	15	1979	Monday
2.	2	15	1979	Thursday
3.	6	15	1979	Friday
4.	11	15	1979	Thursday
5.	12	15	1979	Saturday
6.	6	1	1979	Friday
7.	6	2	1979	Saturday
8.	6	30	1979	Saturday
9.	6	31	1979	Invalid Date
10.	6	15	1900	Friday
11.	6	15	1901	Saturday
12.	6	15	2057	Friday
13.	6	15	2058	Saturday

2.1.1 Robustness Testing

This is the extension of boundary value analysis. Here, we also select invalid values and see the responses of the program. Invalid values are also important to check the behaviour of the program. Hence, two additional states are added i.e. just below minimum value (minimum value $^{-}$) and just above maximum value (maximum value $^{+}$). We want to go beyond the legitimate domain of input values. This extended form of boundary value analysis is known as robustness testing. The inputs are shown graphically in Figure 2.6 and the test cases for the program 'Addition' are given in Table 2.7. There are four additional test cases which are outside the legitimate input domain. Thus, the total test cases in robustness testing are $6n + 1$, where 'n' is the number of input values. All input values may have one of the following values:

(i) Minimum value
(ii) Just above minimum value
(iii) Just below minimum value
(iv) Just above maximum value
(v) Just below maximum value
(vi) Maximum value
(vii) Nominal (Average) value

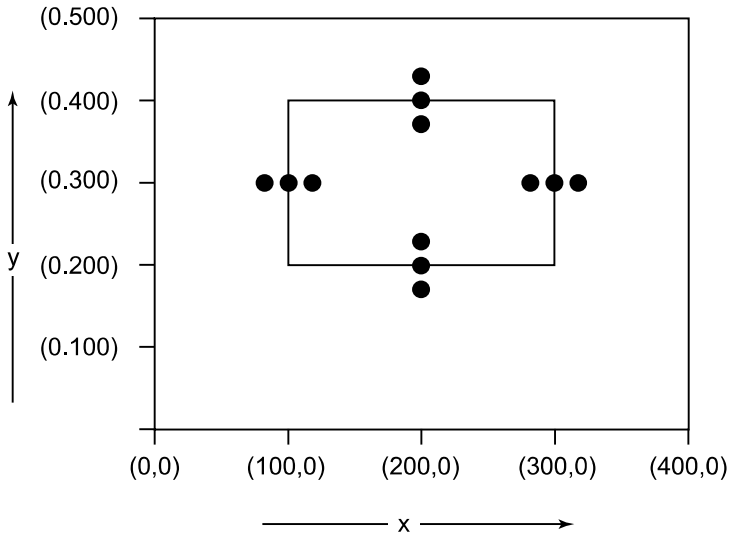


Figure 2.6. Graphical representation of inputs

Table 2.7. Robustness test cases for two input values x and y

Test Case	x	y	Expected Output
1.	99	300	Invalid Input
2.	100	300	400
3.	101	300	401
4.	200	300	500
5.	299	300	599
6.	300	300	600
7.	301	300	Invalid Input
8.	200	199	Invalid Input
9.	200	200	400
10.	200	201	401
11.	200	399	599
12.	200	400	600
13.	200	401	Invalid Input

2.1.2 Worst-Case Testing

This is a special form of boundary value analysis where we don't consider the 'single fault' assumption theory of reliability. Now, failures are also due to occurrence of more than one fault simultaneously. The implication of this concept in boundary value analysis is that all input values may have one of the following:

(i) Minimum value
(ii) Just above minimum value

(iii) Just below maximum value
(iv) Maximum value
(v) Nominal (Average) value

The restriction of one input value at any of the above mentioned values and other input values must be at nominal is not valid in worst-case testing. This will increase the number of test cases from 4n + 1 test cases to 5^n test cases, where 'n' is the number of input values. The inputs for 'Addition' program are shown graphically in Figure 2.7. The program 'Addition' will have $5^2 = 25$ test cases and these test cases are given in Table 2.8.

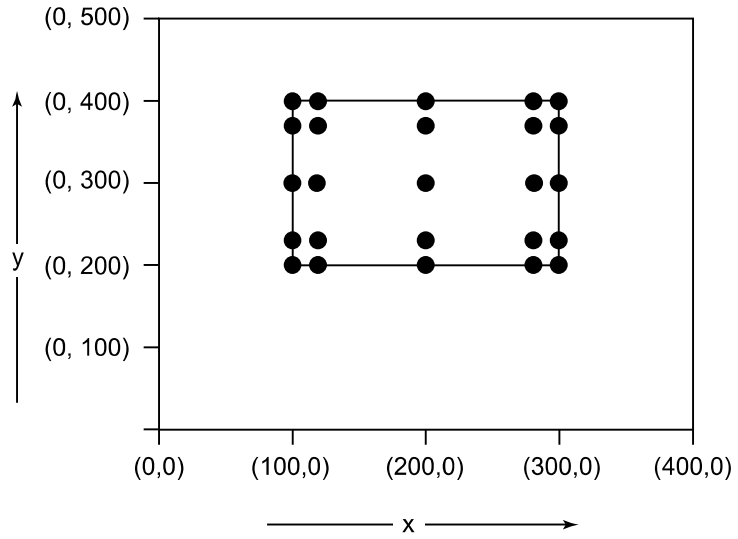


Figure 2.7. Graphical representation of inputs

Table 2.8. Worst test cases for the program 'Addition'

Test Case	x	y	Expected Output
1.	100	200	300
2.	100	201	301
3.	100	300	400
4.	100	399	499
5.	100	400	500
6.	101	200	301
7.	101	201	302
8.	101	300	401
9.	101	399	500
10.	101	400	501
11.	200	200	400
12.	200	201	401
13.	200	300	500
14.	200	399	599

(*Contd.*)

(*Contd.*)

Test Case	x	y	Expected Output
15.	200	400	600
16.	299	200	499
17.	299	201	500
18.	299	300	599
19.	299	399	698
20.	299	400	699
21.	300	200	500
22.	300	201	501
23.	300	300	600
24.	300	399	699
25.	300	400	700

This is a more comprehensive technique and boundary value test cases are proper sub-sets of worst case test cases. This requires more effort and is recommended in situations where failure of the program is extremely critical and costly [JORG07].

2.1.3 Robust Worst-Case Testing

In robustness testing, we add two more states i.e. just below minimum value (minimum value$^-$) and just above maximum value (maximum value$^+$). We also give invalid inputs and observe the behaviour of the program. A program should be able to handle invalid input values, otherwise it may fail and give unexpected output values. There are seven states (minimum $^-$, minimum, minimum $^+$, nominal, maximum $^-$, maximum, maximum $^+$) and a total of 7^n test cases will be generated. This will be the largest set of test cases and requires the maximum effort to generate such test cases. The inputs for the program 'Addition' are graphically shown in Figure 2.8. The program 'Addition' will have $7^2 = 49$ test cases and these test cases are shown in Table 2.9.

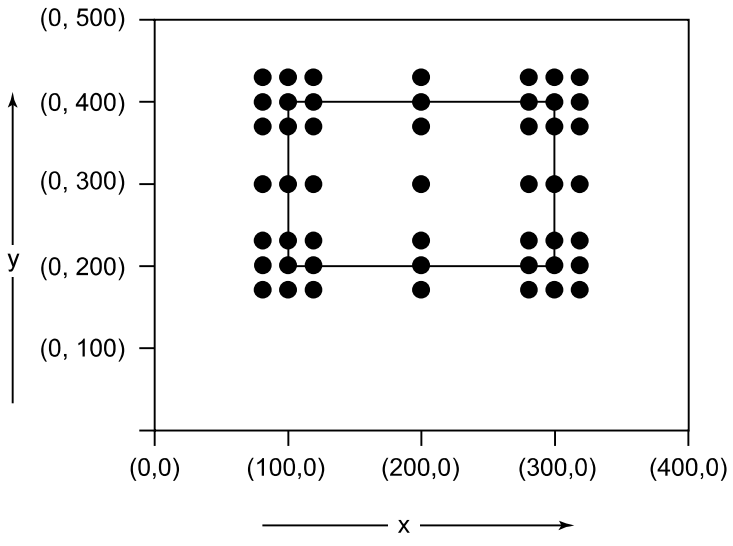


Figure 2.8. Graphical representation of inputs

Table 2.9. Robust worst test cases for the program 'Addition'

Test Case	x	y	Expected Output
1.	99	199	Invalid input
2.	99	200	Invalid input
3.	99	201	Invalid input
4.	99	300	Invalid input
5.	99	399	Invalid input
6.	99	400	Invalid input
7.	99	401	Invalid input
8.	100	199	Invalid input
9.	100	200	300
10.	100	201	301
11.	100	300	400
12.	100	399	499
13.	100	400	500
14.	100	401	Invalid input
15.	101	199	Invalid input
16.	101	200	301
17.	101	201	302
18.	101	300	401
19.	101	399	500
20.	101	400	501
21.	101	401	Invalid input
22.	200	199	Invalid input
23.	200	200	400
24.	200	201	401
25.	200	300	500
26.	200	399	599
27.	200	400	600
28.	200	401	Invalid input
29.	299	199	Invalid input
30.	299	200	499
31.	299	201	500
32.	299	300	599
33.	299	399	698
34.	299	400	699
35.	299	401	Invalid input
36.	300	199	Invalid input
37.	300	200	500
38.	300	201	501
39.	300	300	600
40.	300	399	699
41.	300	400	700
42.	300	401	Invalid input
43.	301	199	Invalid input
44.	301	200	Invalid input
45.	301	201	Invalid input
46.	301	300	Invalid input
47.	301	399	Invalid input
48.	301	400	Invalid input
49.	301	401	Invalid input

2.1.4 Applicability

Boundary value analysis is a simple technique and may prove to be effective when used correctly. Here, input values should be independent which restricts its applicability in many programs. This technique does not make sense for Boolean variables where input values are TRUE and FALSE only, and no choice is available for nominal values, just above boundary values, just below boundary values, etc. This technique can significantly reduce the number of test cases and is suited to programs in which input values are within ranges or within sets. This is equally applicable at the unit, integration, system and acceptance test levels. All we want is input values where boundaries can be identified from the requirements.

Example 2.5: Consider the program for the determination of the largest amongst three numbers as explained in example 2.1. Design the robust test cases and worst case test cases for this program.

Solution: The robust test cases and worst test cases are given in Table 2.10 and Table 2.11 respectively.

Table 2.10. Robust test cases for the program to find the largest among three numbers

Test Case	x	y	z	Expected output
1.	0	150	150	Invalid input
2.	1	150	150	150
3.	2	150	150	150
4.	150	150	150	150
5.	299	150	150	299
6.	300	150	150	300
7.	301	150	150	Invalid input
8.	150	0	150	Invalid input
9.	150	1	150	150
10.	150	2	150	150
11.	150	299	150	299
12.	150	300	150	300
13.	150	301	150	Invalid input
14.	150	150	0	Invalid input
15.	150	150	1	150
16.	150	150	2	150
17.	150	150	299	299
18.	150	150	300	300
19.	150	150	301	Invalid input

Table 2.11. Worst case test cases for the program to find the largest among three numbers

Test Case	x	y	z	Expected output
1.	1	1	1	1
2.	1	1	2	2
3.	1	1	150	150

(*Contd.*)

(*Contd.*)

Test Case	x	y	z	Expected output
4.	1	1	299	299
5.	1	1	300	300
6.	1	2	1	2
7.	1	2	2	2
8.	1	2	150	150
9.	1	2	299	299
10.	1	2	300	300
11.	1	150	1	150
12.	1	150	2	150
13.	1	150	150	150
14.	1	150	299	299
15.	1	150	300	300
16.	1	299	1	299
17.	1	299	2	299
18.	1	299	150	299
19.	1	299	299	299
20.	1	299	300	300
21.	1	300	1	300
22.	1	300	2	300
23.	1	300	150	300
24.	1	300	299	300
25.	1	300	300	300
26.	2	1	1	2
27.	2	1	2	2
28.	2	1	150	150
29.	2	1	299	299
30.	2	1	300	300
31.	2	2	1	2
32.	2	2	2	2
33.	2	2	150	150
34.	2	2	299	299
35.	2	2	300	300
36.	2	150	1	150
37.	2	150	2	150
38.	2	150	150	150
39.	2	150	299	299
40.	2	150	300	300
41.	2	299	1	299
42.	2	299	2	299
43.	2	299	150	299
44.	2	299	299	299
45.	2	299	300	300
46.	2	300	1	300
47.	2	300	2	300
48.	2	300	150	300
49.	2	300	299	300

(*Contd.*)

(*Contd.*)

Test Case	x	y	z	Expected output
50.	2	300	300	300
51.	150	1	1	150
52.	150	1	2	150
53.	150	1	150	150
54.	150	1	299	299
55.	150	1	300	300
56.	150	2	1	150
57.	150	2	2	150
58.	150	2	150	150
59.	150	2	299	299
60.	150	2	300	300
61.	150	150	1	150
62.	150	150	2	150
63.	150	150	150	150
64.	150	150	299	299
65.	150	150	300	300
66.	150	299	1	299
67.	150	299	2	299
68.	150	299	150	299
69.	150	299	299	299
70.	150	299	300	300
71.	150	300	1	300
72.	150	300	2	300
73.	150	300	150	300
74.	150	300	299	300
75.	150	300	300	300
76.	299	1	1	299
77.	299	1	2	299
78.	299	1	150	299
79.	299	1	299	299
80.	299	1	300	300
81.	299	2	1	299
82.	299	2	2	299
83.	299	2	150	299
84.	299	2	299	299
85.	299	2	300	300
86.	299	150	1	299
87.	299	150	2	299
88.	299	150	150	299
89.	299	150	299	299
90.	299	150	300	300
91.	299	299	1	299
92.	299	299	2	299
93.	299	299	150	299
94.	299	299	299	299
95.	299	299	300	300

(*Contd.*)

(*Contd.*)

Test Case	x	y	z	Expected output
96.	299	300	1	300
97.	299	300	2	300
98.	299	300	150	300
99.	299	300	299	300
100.	299	300	300	300
101.	300	1	1	300
102.	300	1	2	300
103.	300	1	150	300
104.	300	1	299	300
105.	300	1	300	300
106.	300	2	1	300
107.	300	2	2	300
108.	300	2	150	300
109.	300	2	299	300
110.	300	2	300	300
111.	300	150	1	300
112.	300	150	2	300
113.	300	150	150	300
114.	300	150	299	300
115.	300	150	300	300
116.	300	299	1	300
117.	300	299	2	300
118.	300	299	150	300
119.	300	299	299	300
120.	300	299	300	300
121.	300	300	1	300
122.	300	300	2	300
123.	300	300	150	300
124.	300	300	299	300
125.	300	300	300	300

Example 2.6: Consider the program for the determination of division of a student based on marks obtained in three subjects as explained in example 2.2. Design the robust test cases and worst case test cases for this program.

Solution: The robust test cases and worst test cases are given in Table 2.12 and Table 2.13 respectively.

Table 2.12. Robust test cases for the program determining the division of a student

Test Case	mark1	mark2	mark3	Expected Output
1.	–1	50	50	Invalid marks
2.	0	50	50	Fail
3.	1	50	50	Fail
4.	50	50	50	Second Division
5.	99	50	50	First Division
6.	100	50	50	First Division

(*Contd.*)

(*Contd.*)

Test Case	mark1	mark2	mark3	Expected Output
7.	101	50	50	Invalid marks
8.	50	–1	50	Invalid marks
9.	50	0	50	Fail
10.	50	1	50	Fail
11.	50	99	50	First Division
12.	50	100	50	First Division
13.	50	101	50	Invalid marks
14.	50	50	–1	Invalid marks
15.	50	50	0	Fail
16.	50	50	1	Fail
17.	50	50	99	First Division
18.	50	50	100	First Division
19.	50	50	101	Invalid Marks

Table 2.13. Worst case test cases for the program for determining the division of a student

Test Case	mark1	mark2	mark3	Expected Output
1	0	0	0	Fail
2.	0	0	1	Fail
3.	0	0	50	Fail
4.	0	0	99	Fail
5.	0	0	100	Fail
6.	0	1	0	Fail
7.	0	1	1	Fail
8.	0	1	50	Fail
9.	0	1	99	Fail
10.	0	1	100	Fail
11.	0	50	0	Fail
12.	0	50	1	Fail
13.	0	50	50	Fail
14.	0	50	99	Third division
15.	0	50	100	Second division
16.	0	99	0	Fail
17.	0	99	1	Fail
18.	0	99	50	Third division
19.	0	99	99	First division
20.	0	99	100	First division
21.	0	100	0	Fail
22.	0	100	1	Fail
23.	0	100	50	Second division
24.	0	100	99	First division
25.	0	100	100	First division
26.	1	0	0	Fail
27.	1	0	1	Fail
28.	1	0	50	Fail
29.	1	0	99	Fail
30.	1	0	100	Fail
31.	1	1	0	Fail
32.	1	1	1	Fail

(*Contd.*)

(*Contd.*)

Test Case	mark1	mark2	mark3	Expected Output
33.	1	1	50	Fail
34.	1	1	99	Fail
35.	1	1	100	Fail
36.	1	50	0	Fail
37.	1	50	1	Fail
38.	1	50	50	Fail
39.	1	50	99	Second division
40.	1	50	100	Second division
41.	1	99	0	Fail
42.	1	99	1	Fail
43.	1	99	50	Second division
44.	1	99	99	First division
45.	1	99	100	First division
46.	1	100	0	Fail
47.	1	100	1	Fail
48.	1	100	50	Second division
49.	1	100	99	First division
50.	1	100	100	First division
51.	50	0	0	Fail
52.	50	0	1	Fail
53.	50	0	50	Fail
54.	50	0	99	Third division
55.	50	0	100	Second division
56.	50	1	0	Fail
57.	50	1	1	Fail
58.	50	1	50	Fail
59.	50	1	99	Second division
60.	50	1	100	Second division
61.	50	50	0	Fail
62.	50	50	1	Fail
63.	50	50	50	Second division
64.	50	50	99	First division
65.	50	50	100	First division
66.	50	99	0	Third division
67.	50	99	1	Second division
68.	50	99	50	First division
69.	50	99	99	First division with distinction
70.	50	99	100	First division with distinction
71.	50	100	0	Second division
72.	50	100	1	Second division
73.	50	100	50	First division
74.	50	100	99	First division
75.	50	100	100	First division with distinction
76.	99	0	0	Fail
77.	99	0	1	Fail
78.	99	0	50	Third division
79.	99	0	99	First division
80.	99	0	100	First division
81.	99	1	0	Fail
82.	99	1	1	Fail

(*Contd.*)

(*Contd.*)

Test Case	mark1	mark2	mark3	Expected Output
83.	99	1	50	Second division
84.	99	1	99	First division
85.	99	1	100	First division
86.	99	50	0	Third division
87.	99	50	1	Second division
88.	99	50	50	First division
89.	99	50	99	First division with distinction
90.	99	50	100	First division with distinction
91.	99	99	0	First division
92.	99	99	1	First division
93.	99	99	50	First division with distinction
94.	99	99	99	First division with distinction
95.	99	99	100	First division with distinction
96.	99	100	0	First division
97.	99	100	1	First division
98.	99	100	50	First division with distinction
99.	99	100	99	First division with distinction
100.	99	100	100	First division with distinction
101.	100	0	0	Fail
102.	100	0	1	Fail
103.	100	0	50	Second division
104.	100	0	99	First division
105.	100	0	100	First division
106.	100	1	0	Fail
107.	100	1	1	Fail
108.	100	1	50	Second division
109.	100	1	99	First division
110.	100	1	100	First division
111.	100	50	0	Second division
112.	100	50	1	Second division
113.	100	50	50	First division
114.	100	50	99	First division with distinction
115.	100	50	100	First division with distinction
116.	100	99	0	First division
117.	100	99	1	First division
118.	100	99	50	First division with distinction
119.	100	99	99	First division wit distinction
120.	100	99	100	First division with distinction
121.	100	100	0	First division
122.	100	100	1	First division
123.	100	100	50	First division with distinction
124.	100	100	99	First division with distinction
125.	100	100	100	First division with distinction

Example 2.7: Consider the program for classification of a triangle in example 2.3. Generate robust and worst test cases for this program.

Solution: Robust test cases and worst test cases are given in Table 2.14 and Table 2.15 respectively.

Table 2.14. Robust test cases for the triangle classification program

Test Case	a	b	c	Expected Output
1.	0	50	50	Input values out of range
2.	1	50	50	Acute angled triangle
3.	2	50	50	Acute angled triangle
4.	50	50	50	Acute angled triangle
5.	99	50	50	Obtuse angled triangle
6.	100	50	50	Invalid triangle
7.	101	50	50	Input values out of range
8.	50	0	50	Input values out of range
9.	50	1	50	Acute angled triangle
10.	50	2	50	Acute angled triangle
11.	50	99	50	Obtuse angled triangle
12.	50	100	50	Invalid triangle
13.	50	101	50	Input values out of range
14.	50	50	0	Input values out of range
15.	50	50	1	Acute angled triangle
16.	50	50	2	Acute angled triangle
17.	50	50	99	Obtuse angled triangle
18.	50	50	100	Invalid triangle
19.	50	50	101	Input values out of range

Table 2.15. Worst case test cases for the triangle classification program

Test Case	a	b	c	Expected Output
1.	1	1	1	Acute angled triangle
2.	1	1	2	Invalid triangle
3.	1	1	50	Invalid triangle
4.	1	1	99	Invalid triangle
5.	1	1	100	Invalid triangle
6.	1	2	1	Invalid triangle
7.	1	2	2	Acute angled triangle
8.	1	2	50	Invalid triangle
9.	1	2	99	Invalid triangle
10.	1	2	100	Invalid triangle
11.	1	50	1	Invalid triangle
12.	1	50	2	Invalid triangle
13.	1	50	50	Acute angled triangle
14.	1	50	99	Invalid triangle
15.	1	50	100	Invalid triangle
16.	1	99	1	Invalid triangle
17.	1	99	2	Invalid triangle

(*Contd.*)

(*Contd.*)

Test Case	a	b	c	Expected Output
18.	1	99	50	Invalid triangle
19.	1	99	99	Acute angled triangle
20.	1	99	100	Invalid triangle
21.	1	100	1	Invalid triangle
22.	1	100	2	Invalid triangle
23.	1	100	50	Invalid triangle
24.	1	100	99	Invalid triangle
25.	1	100	100	Acute angled triangle
26.	2	1	1	Invalid triangle
27.	2	1	2	Acute angled triangle
28.	2	1	50	Invalid triangle
29.	2	1	99	Invalid triangle
30.	2	1	100	Invalid triangle
31.	2	2	1	Acute angled triangle
32.	2	2	2	Acute angled triangle
33.	2	2	50	Invalid triangle
34.	2	2	99	Invalid triangle
35.	2	2	100	Invalid triangle
36.	2	50	1	Invalid triangle
37.	2	50	2	Invalid triangle
38.	2	50	50	Acute angled triangle
39.	2	50	99	Invalid triangle
40.	2	50	100	Invalid triangle
41.	2	99	1	Invalid triangle
42.	2	99	2	Invalid triangle
43.	2	99	50	Invalid triangle
44.	2	99	99	Acute angled
45.	2	99	100	Obtuse angled triangle
46.	2	100	1	Invalid triangle
47.	2	100	2	Invalid triangle
48.	2	100	50	Invalid triangle
49.	2	100	99	Obtuse angled triangle
50.	2	100	100	Acute angled triangle
51.	50	1	1	Invalid triangle
52.	50	1	2	Invalid triangle
53.	50	1	50	Acute angled triangle
54.	50	1	99	Invalid triangle
55.	50	1	100	Invalid triangle
56.	50	2	1	Invalid triangle

(*Contd.*)

(*Contd.*)

Test Case	a	b	c	Expected Output
57.	50	2	2	Invalid triangle
58.	50	2	50	Acute angled triangle
59.	50	2	99	Invalid triangle
60.	50	2	100	Invalid triangle
61.	50	50	1	Acute angled triangle
62.	50	50	2	Acute angled triangle
63.	50	50	50	Acute angled triangle
64.	50	50	99	Obtuse angled triangle
65.	50	50	100	Invalid triangle
66.	50	99	1	Invalid triangle
67.	50	99	2	Invalid triangle
68.	50	99	50	Obtuse angled triangle
69.	50	99	99	Acute angled triangle
70.	50	99	100	Acute angled triangle
71.	50	100	1	Invalid triangle
72.	50	100	2	Invalid triangle
73.	50	100	50	Invalid triangle
74.	50	100	99	Acute angled triangle
75.	50	100	100	Acute angled triangle
76.	99	1	1	Invalid triangle
77.	99	1	2	Invalid triangle
78.	99	1	50	Invalid triangle
79.	99	1	99	Acute angled triangle
80.	99	1	100	Invalid triangle
81.	99	2	1	Invalid triangle
82.	99	2	2	Invalid triangle
83.	99	2	50	Invalid triangle
84.	99	2	99	Acute angled triangle
85.	99	2	100	Obtuse angled triangle
86.	99	50	1	Invalid triangle
87.	99	50	2	Invalid triangle
88.	99	50	50	Obtuse angled triangle
89.	99	50	99	Acute angled triangle
90.	99	50	100	Acute angled triangle
91.	99	99	1	Acute angled triangle
92.	99	99	2	Acute angled triangle
93.	99	99	50	Acute angled triangle
94.	99	99	99	Acute angled triangle
95.	99	99	100	Acute angled triangle

(*Contd.*)

(*Contd.*)

Test Case	a	b	c	Expected Output
96.	99	100	1	Invalid triangle
97.	99	100	2	Obtuse angled triangle
98.	99	100	50	Acute angled triangle
99.	99	100	99	Acute angled triangle
100.	99	100	100	Acute angled triangle
101.	100	1	1	Invalid triangle
102.	100	1	2	Invalid triangle
103.	100	1	50	Invalid triangle
104.	100	1	99	Invalid triangle
105.	100	1	100	Acute angled triangle
106.	100	2	1	Invalid triangle
107.	100	2	2	Invalid triangle
108.	100	2	50	Invalid triangle
109.	100	2	99	Obtuse angled triangle
110.	100	2	100	Acute angled triangle
111.	100	50	1	Invalid triangle
112.	100	50	2	Invalid triangle
113.	100	50	50	Invalid triangle
114.	100	50	99	Acute angled triangle
115.	100	50	100	Acute angled triangle
116.	100	99	1	Invalid triangle
117.	100	99	2	Obtuse angled triangle
118.	100	99	50	Acute angled triangle
119.	100	99	99	Acute angled triangle
120.	100	99	100	Acute angled triangle
121.	100	100	1	Acute angled triangle
122.	100	100	2	Acute angled triangle
123.	100	100	50	Acute angled triangle
124.	100	100	99	Acute angled triangle
125.	100	100	100	Acute angled triangle

Example 2.8: Consider the program for the determination of day of the week as explained in example 2.4. Design the robust and worst test cases for this program.

Solution: Robust test cases and worst test cases are given in Table 2.16 and Table 2.17 respectively.

Table 2.16. Robust test cases for program for determining the day of the week

Test Case	month	day	year	Expected Output
1.	0	15	1979	Invalid date
2.	1	15	1979	Monday
3.	2	15	1979	Thursday
4.	6	15	1979	Friday
5.	11	15	1979	Thursday
6.	12	15	1979	Saturday
7.	13	15	1979	Invalid date
8.	6	0	1979	Invalid date
9.	6	1	1979	Friday
10.	6	2	1979	Saturday
11.	6	30	1979	Saturday
12.	6	31	1979	Invalid date
13.	6	32	1979	Invalid date
14.	6	15	1899	Invalid date (out of range)
15.	6	15	1900	Friday
16.	6	15	1901	Saturday
17.	6	15	2057	Friday
18.	6	15	2058	Saturday
19.	6	15	2059	Invalid date (out of range)

Table 2.17. Worst case test cases for the program determining day of the week

Test Case	month	day	year	Expected Output
1.	1	1	1900	Monday
2.	1	1	1901	Tuesday
3.	1	1	1979	Monday
4.	1	1	2057	Monday
5.	1	1	2058	Tuesday
6.	1	2	1900	Tuesday
7.	1	2	1901	Wednesday
8.	1	2	1979	Tuesday
9.	1	2	2057	Tuesday
10.	1	2	2058	Wednesday
11.	1	15	1900	Monday
12.	1	15	1901	Tuesday
13.	1	15	1979	Monday
14.	1	15	2057	Monday
15.	1	15	2058	Tuesday

(*Contd.*)

(*Contd.*)

Test Case	month	day	year	Expected Output
16.	1	30	1900	Tuesday
17.	1	30	1901	Wednesday
18.	1	30	1979	Tuesday
19.	1	30	2057	Tuesday
20.	1	30	2058	Wednesday
21.	1	31	1900	Wednesday
22.	1	31	1901	Thursday
23.	1	31	1979	Wednesday
24.	1	31	2057	Wednesday
25.	1	31	2058	Thursday
26.	2	1	1900	Thursday
27.	2	1	1901	Friday
28.	2	1	1979	Thursday
29.	2	1	2057	Thursday
30.	2	1	2058	Friday
31.	2	2	1900	Friday
32.	2	2	1901	Saturday
33.	2	2	1979	Friday
34.	2	2	2057	Friday
35.	2	2	2058	Saturday
36.	2	15	1900	Thursday
37.	2	15	1901	Friday
38.	2	15	1979	Thursday
39.	2	15	2057	Thursday
40.	2	15	2058	Friday
41.	2	30	1900	Invalid date
42.	2	30	1901	Invalid date
43.	2	30	1979	Invalid date
44.	2	30	2057	Invalid date
45.	2	30	2058	Invalid date
46.	2	31	1900	Invalid date
47.	2	31	1901	Invalid date
48.	2	31	1979	Invalid date
49.	2	31	2057	Invalid date
50.	2	31	2058	Invalid date
51.	6	1	1900	Friday
52.	6	1	1901	Saturday

(*Contd.*)

(*Contd.*)

Test Case	month	day	year	Expected Output
53.	6	1	1979	Friday
54.	6	1	2057	Friday
55.	6	1	2058	Saturday
56.	6	2	1900	Saturday
57.	6	2	1901	Sunday
58.	6	2	1979	Saturday
59.	6	2	2057	Saturday
60.	6	2	2058	Sunday
61.	6	15	1900	Friday
62.	6	15	1901	Saturday
63.	6	15	1979	Friday
64.	6	15	2057	Friday
65.	6	15	2058	Saturday
66.	6	30	1900	Saturday
67.	6	30	1901	Sunday
68.	6	30	1979	Saturday
69.	6	30	2057	Saturday
70.	6	30	2058	Sunday
71.	6	31	1900	Invalid date
72.	6	31	1901	Invalid date
73.	6	31	1979	Invalid date
74.	6	31	2057	Invalid date
75.	6	31	2058	Invalid date
76.	11	1	1900	Thursday
77.	11	1	1901	Friday
78.	11	1	1979	Thursday
79.	11	1	2057	Thursday
80.	11	1	2058	Friday
81.	11	2	1900	Friday
82.	11	2	1901	Saturday
83.	11	2	1979	Friday
84.	11	2	2057	Friday
85.	11	2	2058	Saturday
86.	11	15	1900	Thursday
87.	11	15	1901	Friday
88.	11	15	1979	Thursday
89.	11	15	2057	Thursday

(*Contd.*)

(*Contd.*)

Test Case	month	day	year	Expected Output
90.	11	15	2058	Friday
91.	11	30	1900	Friday
92.	11	30	1901	Saturday
93.	11	30	1979	Friday
94.	11	30	2057	Friday
95.	11	30	2058	Saturday
96.	11	31	1900	Invalid date
97.	11	31	1901	Invalid date
98.	11	31	1979	Invalid date
99.	11	31	2057	Invalid date
100.	11	31	2058	Invalid date
101.	12	1	1900	Saturday
102.	12	1	1901	Sunday
103.	12	1	1979	Saturday
104.	12	1	2057	Saturday
105.	12	1	2058	Sunday
106.	12	2	1900	Sunday
107.	12	2	1901	Monday
108.	12	2	1979	Sunday
109.	12	2	2057	Sunday
110.	12	2	2058	Monday
111.	12	15	1900	Saturday
112.	12	15	1901	Sunday
113.	12	15	1979	Saturday
114.	12	15	2057	Saturday
115.	12	15	2058	Sunday
116.	12	30	1900	Sunday
117.	12	30	1901	Monday
118.	12	30	1979	Sunday
119.	12	30	2057	Sunday
120.	12	30	2058	Monday
121.	12	31	1900	Monday
122.	12	31	1901	Tuesday
123.	12	31	1979	Monday
124.	12	31	2057	Monday
125.	12	31	2058	Tuesday

2.2 EQUIVALENCE CLASS TESTING

As we have discussed earlier, a large number of test cases are generated for any program. It is neither feasible nor desirable to execute all such test cases. We want to select a few test cases and still wish to achieve a reasonable level of coverage. Many test cases do not test any new thing and they just execute the same lines of source code again and again. We may divide input domain into various categories with some relationship and expect that every test case from a category exhibits the same behaviour. If categories are well selected, we may assume that if one representative test case works correctly, others may also give the same results. This assumption allows us to select exactly one test case from each category and if there are four categories, four test cases may be selected. Each category is called an equivalence class and this type of testing is known as equivalence class testing.

2.2.1 Creation of Equivalence Classes

The entire input domain can be divided into at least two equivalence classes: one containing all valid inputs and the other containing all invalid inputs. Each equivalence class can further be sub-divided into equivalence classes on which the program is required to behave differently. The input domain equivalence classes for the program 'Square' which takes 'x' as an input (range 1-100) and prints the square of 'x' (seen in Figure 2.2) are given as:

(i) $I_1 = \{ 1 \leq x \leq 100 \}$ (Valid input range from 1 to 100)
(ii) $I_2 = \{ x < 1 \}$ (Any invalid input where x is less than 1)
(iii) $I_3 = \{ x > 100 \}$ (Any invalid input where x is greater than 100)

Three test cases are generated covering every equivalence class and are given in Table 2.18.

Table 2.18. Test cases for program 'Square' based on input domain

Test Case	Input x	Expected Output
I_1	0	Invalid Input
I_2	50	2500
I_3	101	Invalid Input

The following equivalence classes can be generated for program 'Addition' for input domain:

(i) $I_1 = \{ 100 \leq x \leq 300 \text{ and } 200 \leq y \leq 400 \}$ (Both x and y are valid values)
(ii) $I_2 = \{ 100 \leq x \leq 300 \text{ and } y < 200 \}$ (x is valid and y is invalid)
(iii) $I_3 = \{ 100 \leq x \leq 300 \text{ and } y > 400 \}$ (x is valid and y is invalid)
(iv) $I_4 = \{ x < 100 \text{ and } 200 \leq y \leq 400 \}$ (x is invalid and y is valid)
(v) $I_5 = \{ x > 300 \text{ and } 200 \leq y \leq 400 \}$ (x is invalid and y is valid)
(vi) $I_6 = \{ x < 100 \text{ and } y < 200 \}$ (Both inputs are invalid)
(vii) $I_7 = \{ x < 100 \text{ and } y > 400 \}$ (Both inputs are invalid)
(viii) $I_8 = \{ x > 300 \text{ and } y < 200 \}$ (Both inputs are invalid)
(ix) $I_9 = \{ x > 300 \text{ and } y > 400 \}$ (Both inputs are invalid)

The graphical representation of inputs is shown in Figure 2.9 and the test cases are given in Table 2.19.

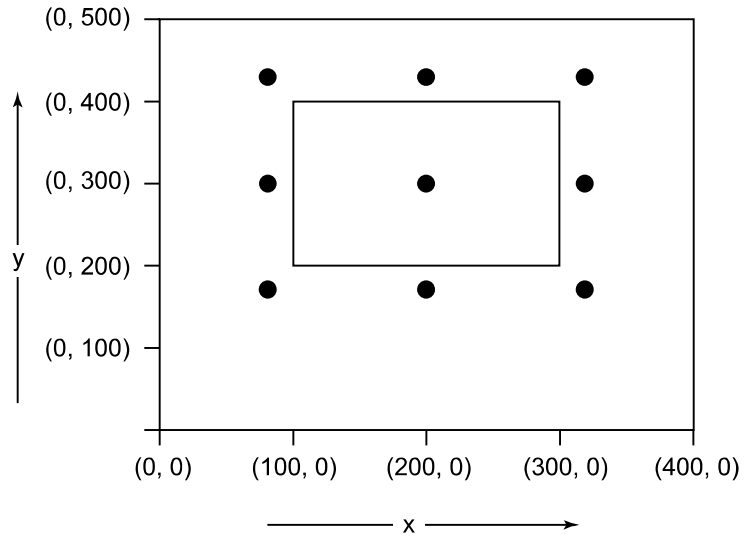


Figure 2.9. Graphical representation of inputs

Table 2.19. Test cases for the program 'Addition'

Test Case	x	y	Expected Output
I_1	200	300	500
I_2	200	199	Invalid input
I^3	200	401	Invalid input
I_4	99	300	Invalid input
I_5	301	300	Invalid input
I_6	99	199	Invalid input
I_7	99	401	Invalid input
I_8	301	199	Invalid input
I_9	301	401	Invalid input

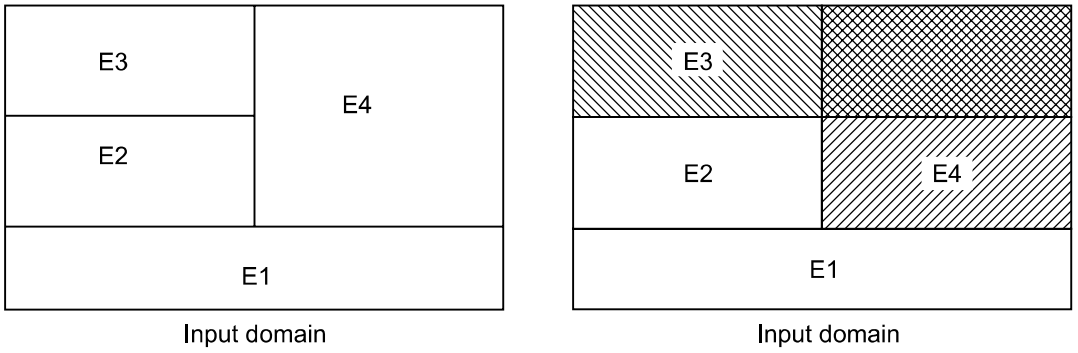
The equivalence classes of input domain may be mutually exclusive (as shown in Figure 2.10 (a)) and they may have overlapping regions (as shown in Figure 2.10 (b)).

We may also partition output domain for the design of equivalence classes. Every output will lead to an equivalence class. Thus, for 'Square' program, the output domain equivalence classes are given as:

O_1 = {square of the input number 'x'}

O_2 = {Invalid input)

The test cases for output domain are shown in Table 2.20. Some of input and output domain test cases may be the same.



(a) Four mutually exclusive equivalence classes (b) E3 and E4 have an overlapping region

Figure 2.10. Equivalence classes of input domain

Table 2.20. Test cases for program 'Square' based on output domain

Test Case	Input x	Expected Output
O_1	50	2500
O_2	0	Invalid Input

We may also design output domain equivalence classes for the program 'Addition' as given below:

O_1 = { Addition of two input numbers x and y }

O_2 = {Invalid Input}

The test cases are given in Table 2.21.

Table 2.21. Test cases for program 'Addition' based on output domain

Test Case	x	y	Expected Output
O_1	200	300	500
O_2	99	300	Invalid Input

In the above two examples, valid input domain has only one equivalence class. We may design more numbers of equivalence classes based on the type of problem and nature of inputs and outputs. Here, the most important task is the creation of equivalence classes which require domain knowledge and experience of testing. This technique reduces the number of test cases that should be designed and executed.

2.2.2 Applicability

It is applicable at unit, integration, system and acceptance test levels. The basic requirement is that inputs or outputs must be partitioned based on the requirements and every partition will give a test case. The selected test case may test the same thing, as would have been tested by another test case of the same equivalence class, and if one test case catches a bug, the other

probably will too. If one test case does not find a bug, the other test cases of the same equivalence class may also not find any bug. We do not consider dependencies among different variables while designing equivalence classes.

The design of equivalence classes is subjective and two testing persons may design two different sets of partitions of input and output domains. This is understandable and correct as long as the partitions are reviewed and all agree that they acceptably cover the program under test.

Example 2.9: Consider the program for determination of the largest amongst three numbers specified in example 2.1. Identify the equivalence class test cases for output and input domain.

Solution: Output domain equivalence classes are:

O_1= {<x, y, z > : Largest amongst three numbers x, y, z }

O_2= {<x, y, z > : Input values(s) is /are out of range with sides x, y, z }

The test cases are given in Table 2.22.

Table 2.22. Output domain test cases to find the largest among three numbers

Test Case	x	y	z	Expected Output
O_1	150	140	110	150
O_2	301	50	50	Input values are out of range

Input domain based equivalence classes are:

I_1 = { 1 ≤x ≤ 300 and 1 ≤ y ≤ 300 and 1 ≤ z ≤ 300 } (All inputs are valid)

I_2 = { x < 1 and 1 ≤ y ≤ 300 and 1 ≤ z ≤ 300 } (x is invalid , y is valid and z is valid)

I_3 = { 1 ≤ x ≤ 300 and y < 1 and 1 ≤ z ≤ 300 } (x is valid, y is invalid and z is valid)

I_4 = { 1 ≤ x ≤ 300 and 1 ≤ y ≤ 300 and z < 1 } (x is valid, y is valid and z is invalid)

I_5 = { x > 300 and 1 ≤ y ≤ 300 and 1 ≤ z ≤ 300 } (x is invalid, y is valid and z is valid)

I_6 = { 1 ≤ x ≤ 300 and y > 300 and 1 ≤ z ≤ 300 } (x is valid, y is invalid and z is valid)

I_7 = { 1 ≤ x ≤ 300 and 1 ≤ y ≤ 300 and z > 300 } (x is valid, y is valid and z is invalid)

I_8 = { x < 1 and y < 1 and 1 ≤ z ≤ 300 } (x is invalid, y is invalid and z is valid)

I_9 = { 1 ≤ x ≤ 300 and y < 1 and z < 1 } (x is valid, y is invalid and z is invalid)

I_{10} = { x < 1 and 1 ≤ y ≤ 300 and z < 1 } (x is invalid, y is valid and z is invalid)

I_{11} = { x > 300 and y > 300 and 1 ≤ z ≤ 300 } (x is invalid, y is invalid and z is valid)

I_{12} = { 1 ≤ x ≤ 300 and y > 300 and z > 300 }(x is valid, y is invalid and z is invalid)

I_{13} = { x > 300 and 1 ≤ y ≤ 300 and z > 300 } (x is invalid, y is valid and z is invalid)

I_{14} = { x < 1 and y > 300 and 1 ≤ z ≤ 300 } (x is invalid, y is invalid and z is valid)

I_{15} = { x > 300 and y < 1 and 1≤ z ≤ 300 } (x is invalid, y is invalid and z is valid)

I_{16} = {1 ≤ x ≤ 300 and y < 1 and z > 300 } (x is valid, y is invalid and z is invalid)

I_{17} = { $1 \le x \le 300$ and $y > 300$ and $z < 1$ } (x is valid, y is invalid and z is invalid)

I_{18} = { $x < 1$ and $1 \le y \le 300$ and $z > 300$ } (x is invalid, y is valid and z is invalid)

I_{19} = { $x > 300$ and $1 \le y \le 300$ and $z < 1$ } (x is invalid, y is valid and z is invalid)

I_{20} = { $x < 1$ and $y < 1$ and $z < 1$ } (All inputs are invalid)

I_{21} = { $x > 300$. and $y > 300$ and $z > 300$ } (All inputs are invalid)

I_{22} = { $x < 1$ and $y < 1$ and $z > 300$ } (All inputs are invalid)

I_{23} = { $x < 1$ and $y > 300$ and $z < 1$ } (All inputs are invalid)

I_{24} = { $x > 300$ and $y < 1$ and $z < 1$ } (All inputs are invalid)

I_{25} = { $x > 300$ and $y > 300$ and $z < 1$ } (All inputs are invalid)

I_{26} = { $x > 300$ and $y < 1$ and $z > 300$ } (All inputs are invalid)

I_{27} = { $x < 1$ and $y > 300$ and $z > 300$ } (All inputs are invalid)

The input domain test cases are given in Table 2.23.

Table 2.23. Input domain test case

Test Case	x	y	z	Expected Output
I_1	150	40	50	150
I_2	0	50	50	Input values are out of range
I_3	50	0	50	Input values are out of range
I_4	50	50	0	Input values are out of range
I_5	101	50	50	Input values are out of range
I_6	50	101	50	Input values are out of range
I_7	50	50	101	Input values are out of range
I_8	0	0	50	Input values are out of range
I_9	50	0	0	Input values are out of range
I_{10}	0	50	0	Input values are out of range
I_{11}	301	301	50	Input values are out of range
I_{12}	50	301	301	Input values are out of range
I_{13}	301	50	301	Input values are out of range
I_{14}	0	301	50	Input values are out of range
I_{15}	301	0	50	Input values are out of range
I_{16}	50	0	301	Input values are out of range
I_{17}	50	301	0	Input values are out of range
I_{18}	0	50	301	Input values are out of range
I_{19}	301	50	0	Input values are out of range

(*Contd.*)

(*Contd.*)

Test Case	x	y	z	Expected Output
I_{20}	0	0	0	Input values are out of range
I_{21}	301	301	301	Input values are out of range
I_{22}	0	0	301	Input values are out of range
I_{23}	0	301	0	Input values are out of range
I_{24}	301	0	0	Input values are out of range
I_{25}	301	301	0	Input values are out of range
I_{26}	301	0	301	Input values are out of range
I_{27}	0	301	301	Input values are out of range

Example 2.10: Consider the program for the determination of division of a student as explained in example 2.2. Identify the equivalence class test cases for output and input domains.

Solution: Output domain equivalence class test cases can be identified as follows:

O_1 = { <mark1, mark2, mark3> : First Division with distinction if average > = 75 }

O_2 = { <mark1, mark2, mark3> : First Division if 60 ≤ average ≤ 74}

O_3 = { <mark1, mark2, mark3> : Second Division if 50 ≤ average ≤ 59 }

O_4 = { <mark1, mark2, mark3> : Third Division if 40 ≤ average ≤ 49 }

O_5 = { <mark1, mark2, mark3> : Fail if average <40 }

O_6 = { <mark1, mark2, mark3> : Invalid marks if marks are not between 0 to 100 }

The test cases generated by output domain are given in Table 2.24.

Table 2.24. Output domain test cases

Test Case	mark1	mark2	mark3	Expected Output
O_1	75	80	85	First division with distinction
O_2	68	68	68	First division
O_3	55	55	55	Second division
O_4	45	45	45	Third division
O_5	25	25	25	Fail
O_6	-1	50	50	Invalid marks

We may have another set of test cases based on input domain.

I_1 = { 0 ≤ mark1 ≤ 100 and 0 ≤ mark2 ≤ 100 and 0 ≤ mark3 ≤ 100 } (All inputs are valid)

I_2 = { mark1 < 0 and 0 ≤ mark2 ≤ 100 and 0 ≤ mark3 ≤ 100 } (mark1 is invalid, mark2 is valid and mark3 is valid)

I_3 = { $0 \leq mark1 \leq 100$ and $mark2 < 0$ and $0 \leq mark3 \leq 100$ } (mark1 is valid, mark2 is invalid and mark3 is valid)

I_4 = { $0 \leq mark1 \leq 100$ and $0 \leq mark2 \leq 100$ and $mark3 < 0$ } (mark1 is valid, mark2 is valid and mark3 is invalid)

I_5 = { $mark1 > 100$ and $0 \leq mark2 \leq 100$ and $0 \leq mark3 \leq 100$ } (mark1 is invalid, mark2 is valid and mark3 is valid)

I_6 = ($0 \leq mark1 \leq 100$ and $mark2 > 100$ and $0 \leq mark3 \leq 100$ } (mark1 is valid, mark2 is invalid and mark3 is valid)

I_7 = { $0 \leq mark1 \leq 100$ and $0 \leq mark2 \leq 100$ and $mark3 > 100$ } (mark 1 is valid, mark2 is valid and mark3 is invalid)

I_8 = { $mark1 < 0$ and $mark2 < 0$ and $0 \leq mark3 \leq 100$ } (mark1 is invalid, mark2 is invalid and mark3 is valid)

I_9 = { $0 \leq mark1 \leq 100$ and $mark2 < 0$ and $mark3 < 0$ } (mark1 is valid, mark2 is invalid and mark3 is invalid)

I_{10} = { $mark1 < 0$ and $0 \leq mark2 \leq 100$ and $mark3 < 0$ } (mark1 is invalid, mark2 is valid and mark3 is invalid)

I_{11} = { $mark1 > 100$ and $mark2 > 100$ and $0 \leq mark3 \leq 100$ } (mark1 is invalid, mark2 is invalid and mark3 is valid)

I_{12} = { $0 \leq mark1 \leq 100$ and $mark2 > 100$ and $mark3 > 100$ } (mark1 is valid, mark2 is invalid and mark3 is invalid)

I_{13} = { $mark1 > 100$ and $0 \leq mark2 \leq 100$ and $mark3 > 100$ } (mark1 is invalid, mark2 is valid and mark3 is invalid)

I_{14} = { $mark1 < 0$ and $mark2 > 100$ and $0 \leq mark3 \leq 100$ } (mark1 is invalid, mark2 is invalid and mark 3 is valid)

I_{15} = { $mark1 > 100$ and $mark2 < 0$ and $0 \leq mark3 \leq 100$ }{ (mark1 is invalid, mark2 is invalid and mark3 is valid)

I_{16} = { $0 \leq mark1 \leq 100$ and $mark2 < 0$ and $mark3 > 100$ } (mark1 is valid, mark2 is invalid and mark3 is invalid)

I_{17} = { $0 \leq mark1 \leq 100$ and $mark2 > 100$ and $mark3 < 0$ } (mark1 is valid, mark2 is invalid and mark3 is invalid)

I_{18} = { $mark1 < 0$ and $0 \leq mark2 \leq 100$ and $mark3 > 100$ } (mark1 is invalid, mark2 is valid and mark3 is invalid)

I_{19} = { $mark1 > 100$ and $0 \leq mark2 \leq 100$ and $mark3 < 0$ } (mark1 is invalid, mark2 is valid and mark3 is invalid)

I_{20} = { $mark1 < 0$ and $mark2 < 0$ and $mark3 < 0$ } (All inputs are invalid)

I_{21} = { $mark1 > 100$ and $mark2 > 100$ and $mark3 > 100$ } (All inputs are invalid)

I_{22} = { $mark1 < 0$ and $mark2 < 0$ and $mark3 > 100$ } (All inputs are invalid)

I_{23} = { $mark1 < 0$ and $mark2 > 100$ and $mark3 < 0$ } (All inputs are invalid)

I_{24} = { $mark1 > 100$ and $mark2 < 0$ and $mark3 < 0$ } (All inputs are invalid)

I_{25} = {$mark1 > 100$ and $mark2 > 100$ and $mark3 < 0$ } (All inputs are invalid)

I_{26} = { mark1 > 100 and mark2 < 0 and mark3 > 100 } (All inputs are invalid)

I_{27} = { mark1 < 0 and mark2 > 100 and mark3 > 100 } (All inputs are invalid)

Thus, 27 test cases are generated on the basis of input domain and are given in Table 3.25.

Table 2.25. Input domain test cases

Test Case	mark1	mark2	mark3	Expected Output
I_1	50	50	50	Second division
I_2	-1	50	50	Invalid marks
I_3	50	-1	50	Invalid marks
I_4	50	50	-1	Invalid marks
I_5	101	50	50	Invalid marks
I_6	50	101	50	Invalid marks
I_7	50	50	101	Invalid marks
I_8	-1	-1	50	Invalid marks
I_9	50	-1	-1	Invalid marks
I_{10}	-1	50	-1	Invalid marks
I_{11}	101	101	50	Invalid marks
I_{12}	50	101	101	Invalid marks
I_{13}	101	50	101	Invalid marks
I_{14}	-1	101	50	Invalid marks
I_{15}	101	-1	50	Invalid marks
I_{16}	50	-1	101	Invalid marks
I_{17}	50	101	-1	Invalid marks
I_{18}	-1	50	101	Invalid marks
I_{19}	101	50	-1	Invalid marks
I_{20}	-1	-1	-1	Invalid marks
I_{21}	101	101	101	Invalid marks
I_{22}	-1	-1	101	Invalid marks
I_{23}	-1	101	-1	Invalid marks
I_{24}	101	-1	-1	Invalid marks
I_{25}	101	101	-1	Invalid marks
I_{26}	101	-1	101	Invalid marks
I_{27}	-1	101	101	Invalid marks

Hence, the total number of equivalence class test cases are 27 (input domain) + 6 (output domain) which is equal to 33.

Example 2.11: Consider the program for classification of a triangle specified in example 2.3. Identify the equivalence class test cases for output and input domain.

Solution: Output domain equivalence classes are:

O_1= { < a, b, c > : Right angled triangle with sides a, b, c }

O_2= { < a, b, c > : Acute angled triangle with sides a, b, c }

O_3= { < a, b, c > : Obtuse angled triangle with sides a, b, c}

O_4= { < a, b, c > : Invalid triangle with sides a, b, c, }

O_5= { < a, b, c > : Input values(s) is /are out of range with sides a, b, c }

The test cases are given in Table 2.26.

Table 2.26. Output domain test cases for triangle classification program

Test Case	a	b	c	Expected Output
O_1	50	40	30	Right angled triangle
O_2	50	49	49	Acute angled triangle
O_3	57	40	40	Obtuse angled triangle
O_4	50	50	100	Invalid triangle
O_5	101	50	50	Input values are out of range

Input domain based equivalence classes are:

I_1 = { 1 ≤ a ≤ 100 and 1 ≤ b ≤ 100 and 1 ≤ c ≤ 100 } (All inputs are valid)

I_2 = { a < 1 and 1 ≤ b ≤ 100 and 1 ≤ c ≤ 100 } (a is invalid , b is valid and c is valid)

I_3 = { 1 ≤ a ≤ 100 and b < 1 and 1 ≤ c ≤ 100 } (a is valid, b is invalid and c is valid)

I_4 = { 1 ≤ a ≤ 100 and 1 ≤ b ≤ 100 and c < 1 } (a is valid, b is valid and c is invalid)

I_5 = { a > 100 and 1 ≤ b ≤ 100 and 1 ≤ c ≤ 100 } (a is invalid, b is valid and c is valid)

I_6 = { 1 ≤ a ≤ 100 and b > 100 and 1 ≤ c ≤ 100 } (a is valid, b is invalid and c is valid)

I_7 = { 1 ≤ a ≤ 100 and 1 ≤ b ≤ 100 and c > 100 } (a is valid, b is valid and c is invalid)

I_8 = { a < 1 and b < 1 and 1 ≤ c ≤ 100 } (a is invalid, b is invalid and c is valid)

I_9 = { 1 ≤ a ≤ 100 and b < 1 and c < 1 } (a is valid, b is invalid and c is invalid)

I_{10} = { a < 1 and 1 ≤ b ≤ 100 and c < 1 } (a is invalid, b is valid and c is invalid)

I_{11} = { a > 100 and b > 100 and 1 ≤ c ≤ 100 } (a is invalid, b is invalid and c is valid)

I_{12} = { 1 ≤ a ≤ 100 and b > 100 and c > 100 }(a is valid, b is invalid and c is invalid)

I_{13} = { a > 100 and 1 ≤ b ≤ 100 and c > 100 } (a is invalid, b is valid and c is invalid)

I_{14} = { a < 1 and b > 100 and 1 ≤ c ≤ 100 } (a is invalid, b is invalid and c is valid)

I_{15} = { a > 100 and b < 1 and 1 ≤ c ≤ 100 } (a is invalid, b is invalid and c is valid)

I_{16} = {1 ≤ a ≤ 100 and b < 1 and c > 100 } (a is valid, b is invalid and c is invalid)

I_{17} = { 1 ≤ a ≤ 100 and b > 100 and c < 1 } (a is valid, b is invalid and c is invalid)

I_{18} = { a < 1 and 1≤ b ≤ 100 and c > 100 } (a is invalid, b is valid and c is invalid)

I_{19} = { a > 100 and 1 ≤ b ≤ 100 and c < 1 } (a is invalid, b is valid and c is invalid)

I_{20} = { a < 1 and b < 1 and c < 1 } (All inputs are invalid)

I_{21} = { a > 100 and b > 100 and c > 100 } (All inputs are invalid)

I_{22} = { a < 1 and b < 1 and c > 100 } (All inputs are invalid)

I_{23} = { a < 1 and b > 100 and c < 1 } (All inputs are invalid)

I_{24} = { a > 100 and b < 1 and c < 1 } (All inputs are invalid)

I_{25} = { a > 100 and b > 100 and c < 1 } (All inputs are invalid)

I_{26} = { a > 100 and b < 1 and c > 100 } (All inputs are invalid)

I_{27} = { a < 1 and b > 100 and c > 100 } (All inputs are invalid)

Some input domain test cases can be obtained using the relationship amongst a, b and c.

I_{28} = { $a^2 = b^2 + c^2$ }

I_{29} = { $b^2 = c^2 + a^2$ }

I_{30} = { $c^2 = a^2 + b^2$ }

I_{31} = { $a^2 > b^2 + c^2$ }

I_{32} = { $b^2 > c^2 + a^2$ }

I_{33} = { $c^2 > a^2 + b^2$ }

I_{34} = { $a^2 < b^2 + c^2$ }

I_{35} = { $b^2 < c^2 + a^2$ }

I_{36} = { $c^2 < a^2 + b^2$ }

I_{37} = { a = b + c }

I_{38} = { a > b + c }

I_{39} = { b = c + a }

I_{40} = { b > c + a }

I_{41} = { c = a + b }

I_{42} = { c > a + b }

I_{43} = { $a^2 < b^2 + c^2$ && $b^2 < c^2 + a^2$ && $c^2 < a^2 + b^2$ }

The input domain test cases are given in Table 2.27.

Table 2.27. Input domain test cases

Test Case	a	b	c	Expected Output
I_1	50	50	50	Acute angled triangle
I_2	0	50	50	Input values are out of range
I_3	50	0	50	Input values are out of range
I_4	50	50	0	Input values are out of range
I_5	101	50	50	Input values are out of range
I_6	50	101	50	Input values are out of range
I_7	50	50	101	Input values are out of range
I_8	0	0	50	Input values are out of range
I_9	50	0	0	Input values are out of range
I_{10}	0	50	0	Input values are out of range
I_{11}	101	101	50	Input values are out of range
I_{12}	50	101	101	Input values are out of range
I_{13}	101	50	101	Input values are out of range
I_{14}	0	101	50	Input values are out of range
I_{15}	101	0	50	Input values are out of range
I_{16}	50	0	101	Input values are out of range
I_{17}	50	101	0	Input values are out of range
I_{18}	0	50	101	Input values are out of range
I_{19}	101	50	0	Input values are out of range
I_{20}	0	0	0	Input values are out of range
I_{21}	101	101	101	Input values are out of range
I_{22}	0	0	101	Input values are out of range
I_{23}	0	101	0	Input values are out of range
I_{24}	101	0	0	Input values are out of range
I_{25}	101	101	0	Input values are out of range
I_{26}	101	0	101	Input values are out of range
I_{27}	0	101	101	Input values are out of range
I_{28}	50	40	30	Right angled triangle
I_{29}	40	50	30	Right angled triangle
I_{30}	40	30	50	Right angled triangle
I_{31}	57	40	40	Obtuse angled triangle
I_{32}	40	57	50	Obtuse angled triangle
I_{33}	40	40	57	Obtuse angled triangle
I_{34}	50	49	49	Acute angled triangle
I_{35}	49	50	49	Acute angled triangle
I_{36}	49	49	50	Acute angled triangle
I_{37}	100	50	50	Invalid triangle
I_{38}	100	40	40	Invalid triangle
I_{39}	50	100	50	Invalid triangle
I_{40}	40	100	40	Invalid triangle
I_{41}	50	50	100	Invalid triangle
I_{42}	40	40	100	Invalid triangle
I_{43}	49	49	50	Acute angled triangle

Hence, total number of equivalence class test cases are 43 (input domain) and 5 (output domain) which is equal to 48.

Example 2.12: Consider the program for determining the day of the week as explained in example 2.4. Identify the equivalence class test cases for output and input domains.

Solution: Output domain equivalence classes are:

O_1 = { < Day, Month, Year > : Monday for all valid inputs }
O_2 = { < Day, Month, Year > : Tuesday for all valid inputs }
O_3 = { < Day, Month, Year > : Wednesday for all valid inputs}
O_4 = { < Day, Month, Year > : Thursday for all valid inputs}
O_5 = { < Day, Month, Year > : Friday for all valid inputs}
O_6 = { < Day, Month, Year > : Saturday for all valid inputs}
O_7 = { < Day, Month, Year > : Sunday for all valid inputs}
O_8 = { < Day, Month, Year > : Invalid Date if any of the input is invalid}
O_9 = { < Day, Month, Year > : Input out of range if any of the input is out of range}

The output domain test cases are given in Table 2.28.

Table 2.28. Output domain equivalence class test cases

Test Case	month	day	year	Expected Output
O_1	6	11	1979	Monday
O_2	6	12	1979	Tuesday
O_3	6	13	1979	Wednesday
O_4	6	14	1979	Thursday
O_5	6	15	1979	Friday
O_6	6	16	1979	Saturday
O_7	6	17	1979	Sunday
O_8	6	31	1979	Invalid date
O_9	6	32	1979	Inputs out of range

The input domain is partitioned as given below:

(i) Valid partitions
M1: Month has 30 Days
M2 : Month has 31 Days
M3 : Month is February
D1 : Days of a month from 1 to 28
D2 : Day = 29
D3 : Day = 30
D4 : Day = 31
Y1 : 1900 ≤ year ≤ 2058 and is a common year
Y2 : 1900 ≤ year ≤ 2058 and is a leap year.

(ii) Invalid partitions
M4 : Month < 1

M5 : Month > 12
D5 : Day < 1
D6 : Day > 31
Y3 : Year < 1900
Y4 : Year > 2058

We may have following equivalence classes which are based on input domain:

(a) Only for valid input domain

I_1 = { M1 and D1 and Y1 } (All inputs are valid)
I_2 = { M2 and D1 and Y1 } (All inputs are valid)
I_3 = { M3 and D1 and Y1 } (All inputs are valid)
I_4 = { M1 and D2 and Y1 } (All inputs are valid)
I_5 = { M2 and D2 and Y1 } (All inputs are valid)
I_6 = { M3 and D2 and Y1 } (All inputs are valid)
I_7 = { M1 and D3 and Y1 } (All inputs are valid)
I_8 = { M2 and D3 and Y1 } (All inputs are valid)
I_9 = { M3 and D3 and Y1 } (All inputs are valid)
I_{10} = { M1 and D4 and Y1 } (All inputs are valid)
I_{11} = { M2 and D4 and Y1 } (All inputs are valid)
I_{12} = { M3 and D4 and Y1 } (All inputs are valid)
I_{13} = { M1 and D1 and Y2 } (All Inputs are valid)
I_{14} = { M2 and D1 and Y2 } (All inputs are valid)
I_{15} = { M3 and D1 and Y2 } (All inputs are valid)
I_{16} = { M1 and D2 and Y2 } (All inputs are valid)
I_{17} = { M2 and D2 and Y2 } (All inputs are valid)
I_{18} = { M3 and D2 and Y2 } (All inputs are valid)
I_{19} = { M1 and D3 and Y2 } (All inputs are valid)
I_{20} = { M2 and D3 and Y2 } (All inputs are valid)
I_{21} = { M3 and D3 and Y2 } (All inputs are valid)
I_{22} = { M1 and D4 and Y2 } (All inputs are valid)
I_{23} = { M2 and D4 and Y2 } (All inputs are valid)
I_{24} = { M3 and D4 and Y2 } (All inputs are valid)

(b) Only for Invalid input domain

I_{25} = { M4 and D1 and Y1 } (Month is invalid, Day is valid and Year is valid)
I_{26} = { M5 and D1 and Y1 } (Month is invalid, Day is valid and Year is valid)
I_{27} = { M4 and D2 and Y1 } (Month is invalid, Day is valid and Year is valid)
I_{28} = { M5 and D2 and Y1 } (Month is invalid, Day is valid and Year is valid)
I_{29} = { M4 and D3 and Y1 } (Month is invalid, Day is valid and Year is valid)
I_{30} = { M5 and D3 and Y1 } (Month is invalid, Day is valid and Year is valid)

I_{31} = { M4 and D4 and Y1 } (Month is invalid, Day is valid and Year is valid)
I_{32} = { M5 and D4 and Y1 } (Month is invalid, Day is valid and year is valid)
I_{33} = { M4 and D1 and Y2 } (Month is invalid, Day is valid and Year is valid)
I_{34} = { M5 and D1 and Y2 } (Month is invalid, Day is valid and Year is valid)
I_{35} = { M4 and D2 and Y2 } (Month is invalid, Day is valid and Year is valid)
I_{36} = { M5 and D2 and Y2 } (Month is invalid, Day is valid and Year is valid)
I_{37} = { M4 and D3 and Y2 } (Month is invalid, Day is valid and Year is valid)
I_{38} = { M5 and D3 and Y2 } (Month is invalid, Day is valid and Year is valid)
I_{39} = { M4 and D4 and Y2 } (Month is invalid, Day is valid and Year is valid)
I_{40} = { M5 and D4 and Y2 } (Month is invalid, Day is valid and Year is valid)
I_{41} = { M1 and D5 and Y1 } (Month is valid, Day is invalid and Year is valid)
I_{42} = { M1 and D6 and Y1 } (Month is valid, Day is invalid and Year is valid)
I_{43} = { M2 and D5 and Y1 } (Month is valid, Day is invalid and Year is valid)
I_{44} = { M2 and D6 and Y1 } (Month is valid, Day is invalid and Year is valid)
I_{45} = { M3 and D5 and Y1 } (Month is valid, Day is invalid and Year is valid)
I_{46} = { M3 and D6 and Y1 } (Month is valid, Day is invalid and Year is valid)
I_{47} = { M1 and D5 and Y2 } (Month is valid, Day is invalid and Year is valid)
I_{48} = { M1 and D6 and Y2 } (Month is valid, Day is invalid and Year is valid)
I_{49} = { M2 and D5 and Y2 } (Month is valid, Day is invalid and Year is valid)
I_{50} = { M2 and D6 and Y2 } (Month is valid, Day is invalid and Year is valid)
I_{51} = { M3 and D5 and Y2 } (Month is valid, Day is invalid and Year is valid)
I_{52} = { M3 and D6 and Y2 } (Month is valid, Day is invalid and Year is valid)
I_{53} = { M1 and D1 and Y3 } (Month is valid, Day is valid and Year is invalid)
I_{54} = { M1 and D1 and Y4 } (Month is valid, Day is valid and Year is invalid)
I_{55} = { M2 and D1 and Y3 } (Month is valid, Day is valid and Year is invalid)
I_{56} = { M2 and D1 and Y4 } (Month is valid, Day is valid and Year is invalid)
I_{57} = { M3 and D1 and Y3 } (Month is valid, Day is valid and Year is invalid)
I_{58} = { M3 and D1 and Y4 } (Month is valid, Day is valid and Year is invalid)
I_{59} = { M1 and D2 and Y3 } (Month is valid, Day is valid and Year is invalid)
I_{60} = { M1 and D2 and Y4 } (Month is valid, Day is valid and Year is invalid)
I_{61} = { M2 and D2 and Y3 } (Month is valid, Day is valid and Year is invalid)
I_{62} = { M2 and D2 and Y4 } (Month is valid, Day is valid and Year is invalid)
I_{63} = { M3 and D2 and Y3 } (Month is valid, Day is valid and Year is invalid)
I_{64} = { M3 and D2 and Y4 } (Month is valid, Day is valid and Year is invalid)
I_{65} = { M1 and D3 and Y3 } (Month is valid, Day is valid and Year is invalid)
I_{66} = { M1 and D3 and Y3 } (Month is valid, Day is valid and Year is invalid)
I_{67} = { M2 and D3 and Y3 } (Month is valid, Day is valid and Year is invalid)

I_{68} = { M2 and D3 and Y4 } (Month is valid, Day is valid and Year is invalid)
I_{69} = { M3 and D3 and Y3 } (Month is valid, Day is valid and Year is invalid)
I_{70} = { M3 and D3 and Y4 } (Month is valid, Day is valid and Year is invalid)
I_{71} = { M1 and D4 and Y3 } (Month is valid, Day is valid and Year is invalid)
I_{72} = { M1 and D4 and Y4 } (Month is valid, Day is valid and Year is invalid)
I_{73} = { M2 and D4 and Y3 } (Month is valid, Day is valid and Year is invalid)
I_{74} = { M2 and D4 and Y4 } (Month is valid, Day is valid and Year is invalid)
I_{75} = { M3 and D4 and Y3 } (Month is valid, Day is valid and Year is invalid)
I_{76} = { M3 and D4 and Y4 } (Month is valid, Day is valid and Year is invalid)
I_{77} = { M4 and D5 and Y1 } (Month is invalid, Day is invalid and Year is valid)
1_{78} = { M4 and D5 and Y2 } (Month is invalid, Day is invalid and year is valid)
I_{79} = { M4 and D6 and Y1 } (Month is invalid, Day is invalid and Year is valid)
I_{80} = { M4 and D6 and Y2 } (Month is invalid, Day is invalid and Year is valid)
I_{81} = { M5 and D5 and Y1 } (Month is invalid, Day is invalid and Year is valid)
I_{82} = { M5 and D5 and Y2 } (Month is invalid, Day is invalid and Year is valid)
I_{83} = { M5 and D6 and Y1 } (Month is invalid, Day is invalid and Year is valid)
I_{84} = { M5 and D6 and Y2 } (Month is invalid, Day is invalid and Year is valid)
I_{85} = { M4 and D1 and Y3 } (Month is invalid, Day is valid and Year is invalid)
I_{86} = { M4 and D1 and Y4 } (Month is invalid, Day is valid and Year is invalid)
I_{87} = { M4 and D2 and Y3 } (Month is invalid, Day is valid and Year is invalid)
I_{88} = { M4 and D2 and Y4 } (Month is invalid, Day is valid and Year is invalid)
I_{89} = { M4 and D3 and Y3 } (Month is invalid, Day is valid and Year is invalid)
I_{90} = { M4 and D3 and Y4 } (Month is invalid, day is valid and Year is invalid)
I_{91} = { M4 and D4 and Y3 } (Month is invalid, Day is valid and Year is invalid)
I_{92} = { M4 and D4 and Y4 } (Month is invalid, Day is valid and Year is invalid)
I_{93} = { M5 and D1 and Y3 } (Month is invalid, Day is valid and Year is invalid)
I_{94} = { M5 and D1 and Y4 } (Month is invalid, Day is valid and Year is invalid)
I_{95} = { M5 and D2 and Y3 } (Month is invalid, Day is valid and year is invalid)
I_{96} = { M5 and D2 and Y4 } (Month is invalid, Day is valid and Year is invalid)
I_{97} = { M5 and D3 and Y3 } (Month is invalid, Day is valid and Year is invalid)
I_{98} = { M5 and D3 and Y4 } (Month is invalid, Day is valid and Year is invalid)
I_{99} = { M5 and D4 and Y3 } (Month is invalid, Day is valid and Year is invalid)
I_{100} = { M5 and D4 and Y4 } (Month is invalid, Day is valid and Year is invalid)
I_{101} = { M1 and D5 and Y3 } (Month is valid, Day is invalid and Year is invalid)
I_{102} = { M1 and D5 and Y4 } (Month is valid, Day is invalid and Year is invalid)
I_{103} = { M2 and D5 and Y3 } (Month is valid, Day is invalid and Year is invalid)
I_{104} = { M2 and D5 and Y4 } (Month is valid, Day is invalid and Year is invalid)

I_{105} = { M3 and D5 and Y3 } (Month is valid, Day is invalid and Year is invalid)
I_{106} = { M3 and D5 and Y4 } (Month is valid, Day is invalid and Year is invalid)
I_{107} = { M1 and D6 and Y3 } (Month is valid, Day is invalid and Year is invalid)
I_{108} = { M1 and D6 and Y4 } (Month is valid, Day is invalid and Year is invalid)
I_{109} = { M2 and D6 and Y3 } (Month is valid, Day is invalid and Year is invalid)
I_{110} = { M2 and D6 and Y4 } (Month is valid, Day is invalid and Year is invalid)
I_{111} = { M3 and D6 and Y3 } (Month is valid, Day is invalid and Year is invalid)
I_{112} = { M3 and D6 and Y4 } (Month is valid, Day is invalid and Year is invalid)
I_{113} = (M4 and D5 and Y3 } (All inputs are invalid)
I_{114} = { M4 and D5 and Y4 } (All inputs are invalid)
I_{115} = { M4 and D6 and Y3 } (All inputs are invalid)
I_{116} = { M4 and D6 and Y4 } (All inputs are invalid)
I_{117} = { M5 and D5 and Y3 } (All inputs are invalid)
I_{118} = { M5 and D5 and Y4 } (All inputs are invalid)
I_{119} = { M5 and D6 and Y3 } (All inputs are invalid)
I_{120} = { M5 and D6 and Y4 } (All inputs are invalid)

The test cases generated on the basis of input domain are given in Table 2.29.

Table 2.29. Input domain equivalence class test cases

Test Case	month	day	year	Expected Output
I_1	6	15	1979	Friday
I_2	5	15	1979	Tuesday
I_3	2	15	1979	Thursday
I_4	6	29	1979	Friday
I_5	5	29	1979	Tuesday
I_6	2	29	1979	Invalid Date
I_7	6	30	1979	Saturday
I_8	5	30	1979	Wednesday
I_9	2	30	1979	Invalid Date
I_{10}	6	31	1979	Invalid Date
I_{11}	5	31	1979	Thursday
I_{12}	2	31	1979	Invalid Date
I_{13}	6	15	2000	Thursday
I_{14}	5	15	2000	Monday
I_{15}	2	15	2000	Tuesday
I_{16}	6	29	2000	Thursday
I_{17}	5	29	2000	Monday
I_{18}	2	29	2000	Tuesday
I_{19}	6	30	2000	Friday
I_{20}	5	30	2000	Tuesday
I_{21}	2	30	2000	Invalid date
I_{22}	6	31	2000	Invalid date

(*Contd.*)

(*Contd.*)

Test Case	month	day	year	Expected Output
I_{23}	5	31	2000	Wednesday
I_{24}	2	31	2000	Invalid date
I_{25}	0	15	1979	Input(s) out of range
I_{26}	13	15	1979	Input(s) out of range
I_{27}	0	29	1979	Inputs(s) out of range
I_{28}	13	29	1979	Input(s) out of range
I_{29}	0	30	1979	Input(s) out of range
I_{30}	13	30	1979	Input(s) out of range
I_{31}	0	31	1979	Input(s) out of range
I_{32}	13	31	1979	Input(s) out of range
I_{33}	0	15	2000	Input(s) out of range
I_{34}	13	15	2000	Input(s) out of range
I_{35}	0	29	2000	Input(s) out of range
I_{36}	13	29	2000	Input(s) out of range
I_{37}	0	30	2000	Input(s) out of range
I_{38}	13	30	2000	Input(s) out of range
I_{39}	0	31	2000	Input(s) out of range
I_{40}	13	31	2000	Input(s) out of range
I_{41}	6	0	1979	Input(s) out of range
I_{42}	6	32	1979	Input(s) out of range
I_{43}	5	0	1979	Input(s) out of range
I_{44}	5	32	1979	Input(s) out of range
I_{45}	2	0	1979	Input(s) out of range
I_{46}	2	32	1979	Input(s) out of range
I_{47}	6	0	2000	Input(s) out of range
I_{48}	6	32	2000	Input(s) out of range
I_{49}	5	0	2000	Input(s) out of range
I_{50}	5	32	2000	Input(s) out of range
I_{51}	2	0	2000	Input(s) out of range
I_{52}	2	32	2000	Input(s) out of range
I_{53}	6	15	1899	Input(s) out of range
I_{54}	6	15	2059	Input(s) out of range
I_{55}	5	15	1899	Input(s) out of range
I_{56}	5	15	2059	Input(s) out of range
I_{57}	2	15	1899	Input(s) out of range
I_{58}	2	15	2059	Input(s) out of range
I_{59}	6	29	1899	Input(s) out of range
I_{60}	6	29	2059	Input(s) out of range
I_{61}	5	29	1899	Input(s) out of range
I_{62}	5	29	2059	Input(s) out of range
I_{63}	2	29	1899	Input(s) out of range
I_{64}	2	29	2059	Input(s) out of range
I_{65}	6	30	1899	Input(s) out of range
I_{66}	6	30	2059	Input(s) out of range
I_{67}	5	30	1899	Input(s) out of range
I_{68}	5	30	2059	Input(s) out of range

(*Contd.*)

(*Contd.*)

Test Case	month	day	year	Expected Output
I_{69}	2	30	1899	Input(s) out of range
I_{70}	2	30	2059	Input(s) out of range
I_{71}	6	31	1899	Input(s) out of range
I_{72}	6	31	2059	Input(s) out of range
I_{73}	5	31	1899	Input(s) out of range
I_{74}	5	31	2059	Input(s) out of range
I_{75}	2	31	1899	Input(s) out of range
I_{76}	2	31	2059	Input(s) out of range
I_{77}	0	0	1979	Input(s) out of range
I_{78}	0	0	2000	Input(s) out of range
I_{79}	0	32	1979	Input(s) out of range
I_{80}	0	32	2000	Input(s) out of range
I_{81}	13	0	1979	Input(s) out of range
I_{82}	13	0	2000	Input(s) out of range
I_{83}	13	32	1979	Input(s) out of range
I_{84}	13	32	2000	Input(s) out of range
I_{85}	0	15	1899	Input(s) out of range
I_{86}	0	15	2059	Input(s) out of range
I_{87}	0	20	1899	Input(s) out of range
I_{88}	0	29	2059	Input(s) out of range
I_{89}	0	30	1899	Input(s) out of range
I_{90}	0	30	2059	Input(s) out of range
I_{91}	0	31	1899	Input(s) out of range
I_{92}	0	31	2059	Input(s) out of range
I_{93}	13	15	1899	Input(s) out of range
I_{94}	13	15	2059	Input(s) out of range
I_{95}	13	29	1899	Input(s) out of range
I_{96}	13	29	2059	Input(s) out of range
I_{97}	13	30	1899	Input(s) out of range
I_{98}	13	30	2059	Input(s) out of range
I_{99}	13	31	1899	Input(s) out of range
I_{100}	13	31	2059	Input(s) out of range
I_{101}	5	0	1899	Input(s) out of range
I_{102}	5	0	2059	Input(s) out of range
I_{103}	6	0	1899	Input(s) out of range
I_{104}	6	0	2059	Input(s) out of range
I_{105}	2	0	1899	Input(s) out of range
I_{106}	2	0	2059	Input(s) out of range
I_{107}	5	32	1899	Input(s) out of range
I_{108}	5	32	2059	Input(s) out of range
I_{109}	6	32	1899	Input(s) out of range
I_{110}	6	32	2059	Input(s) out of range
I_{111}	2	32	1899	Input(s) out of range
I_{112}	2	32	2059	Input(s) out of range
I_{113}	0	0	1899	Input(s) out of range
I_{114}	0	0	2059	Input(s) out of range

(*Contd.*)

(*Contd.*)

Test Case	month	day	year	Expected Output
I_{115}	0	32	1899	Input(s) out of range
I_{116}	0	32	2059	Input(s) out of range
I_{117}	13	0	1899	Input(s) out of range
I_{118}	13	0	2059	Input(s) out of range
I_{119}	13	32	1899	Input(s) out of range
I_{120}	13	32	2059	Input(s) out of range

Hence, the total number of equivalence class test cases are 120 (input domain) + 9 (output domain) which is equal to 129. However, most of the outputs are 'Input out of range' and may not offer any value addition. This situation occurs when we choose more numbers of invalid equivalence classes.

It is clear that if the number of valid partitions of input domain increases, then the number of test cases increases very significantly and is equal to the product of the number of partitions of each input variable. In this example, there are 5 partitions of input variable 'month', 6 partitions of input variable 'day' and 4 partitions of input variable 'year' and thus leading to 5x6x4 = 120 equivalence classes of input domain.

2.3 DECISION TABLE BASED TESTING

Decision tables are used in many engineering disciplines to represent complex logical relationships. An output may be dependent on many input conditions and decision tables give a pictorial view of various combinations of input conditions. There are four portions of the decision table and are shown in Table 2.30. The decision table provides a set of conditions and their corresponding actions.

Table 2.30. Decision table

	Stubs	Entries
Condition	c_1 c_2 c_3	
Action	a_1 a_2 a_3 a_4	

Four Portions

1. Condition Stubs
2. Condition Entries
3. Action Stubs
4. Action Entries

2.3.1 Parts of the Decision Table

The four parts of the decision table are given as:

Condition Stubs: All the conditions are represented in this upper left section of the decision table. These conditions are used to determine a particular action or set of actions.

Action Stubs: All possible actions are listed in this lower left portion of the decision table.

Condition Entries: In the condition entries portion of the decision table, we have a number of columns and each column represents a rule. Values entered in this upper right portion of the table are known as inputs.

Action Entries: Each entry in the action entries portion has some associated action or set of actions in this lower right portion of the table. These values are known as outputs and are dependent upon the functionality of the program.

2.3.2 Limited Entry and Extended Entry Decision Tables

Decision table testing technique is used to design a complete set of test cases without using the internal structure of the program. Every column is associated with a rule and generates a test case. A typical decision table is given in Table 2.31.

Table 2.31. Typical structure of a decision table

Stubs	R_1	R_2	R_3	R_4
c_1	F	T	T	T
c_2	-	F	T	T
c_3	-	-	F	T
a_1	X	X		X
a_2			X	
a_3	X			

In Table 2.31, input values are only True (T) or False (F), which are binary conditions. The decision tables which use only binary conditions are known as limited entry decision tables. The decision tables which use multiple conditions where a condition may have many possibilities instead of only 'true' and 'false' are known as extended entry decision tables [COPE04].

2.3.3 'Do Not Care' Conditions and Rule Count

We consider the program for the classification of the triangle as explained in example 2.3. The decision table of the program is given in Table 2.32, where inputs are depicted using binary values.

Table 2.32. Decision table for triangle problem

Condition	c_1: a < b + c?	F	T	T	T	T	T	T	T	T	T	T
	c_2: b < c + a?	-	F	T	T	T	T	T	T	T	T	T
	c_3: c < a + b?	-	-	F	T	T	T	T	T	T	T	T
	c_4: $a^2 = b^2 + c^2$?	-	-	-	T	T	T	T	F	F	F	F
	c_5: $a^2 > b^2 + c^2$?	-	-	-	T	T	F	F	T	T	F	F
	c_6: $a_2 < b_2 + c_2$?	-	-	-	T	F	T	F	T	F	T	F
	Rule Count	32	16	8	1	1	1	1	1	1	1	1
Action	a_1 : Invalid triangle	X	X	X								
	a_2 : Right angled triangle							X				
	a_3 : Obtuse angled triangle									X		
	a_4 : Acute angled triangle										X	
	a_5 : Impossible				X	X	X		X			X

The 'do not care' conditions are represented by the '-'sign. A 'do not care' condition has no effect on the output. If we refer to column 1 of the decision table, where condition c_1: a < b + c is false, then other entries become 'do not care' entries. If c_1 is false, the output will be 'Invalid triangle' irrespective of any state (true or false) of other conditions like c_2, c_3, c_4, c_5 and c_6. These conditions become do not care conditions and are represented by '-'sign. If we do not do so and represent all true and false entries of every condition, the number of columns in the decision table will unnecessarily increase. This is nothing but a representation facility in the decision table to reduce the number of columns and avoid redundancy. Ideally, each column has one rule and that leads to a test case. A column in the entry portion of the table is known as a rule. In the Table 2.32, a term is used as 'rule count' and 32 is mentioned in column 1. The term 'rule count' is used with 'do not care' entries in the decision table and has a value 1, if 'do not care' conditions are not there, but it doubles for every 'do not care' entry. Hence each 'do not care' condition counts for two rules. Rule count can be calculated as:

$$\text{Rule count} = 2^{\text{number of do not care conditions}}$$

However, this is applicable only for limited entry decision tables where only 'true' and 'false' conditions are considered. Hence, the actual number of columns in any decision table is the sum of the rule counts of every column shown in the decision table. The triangle classification decision table has 11 columns as shown in Table 2.32. However the actual columns are a sum of rule counts and are equal to 64. Hence, this way of representation has reduced the number of columns from 64 to 11 without compromising any information. If rule count value of the decision table does not equal to the number of rules computed by the program, then the decision table is incomplete and needs revision.

2.3.4 Impossible Conditions

Decision tables are very popular for the generation of test cases. Sometimes, we may have to make a few attempts to reach the final solution. Some impossible conditions are also generated due to combinations of various inputs and an 'impossible' action is incorporated in the 'action stub' to show such a condition. We may have to redesign the input classes to reduce the impossible actions. Redundancy and inconsistency may create problems but may be reduced by proper designing of input classes depending upon the functionality of a program.

2.3.5 Applicability

Decision tables are popular in circumstances where an output is dependent on many conditions and a large number of decisions are required to be taken. They may also incorporate complex business rules and use them to design test cases. Every column of the decision table generates a test case. As the size of the program increases, handling of decision tables becomes difficult and cumbersome. In practice, they can be applied easily at unit level only. System testing and integration testing may not find its effective applications.

Example 2.13: Consider the problem for determining of the largest amongst three numbers as given in example 2.1. Identify the test cases using the decision table based testing.

Solution: The decision table is given in Table 2.33.

Table 2.33. Decision table

c_1: x > = 1?	F	T	T	T	T	T	T	T	T	T	T	T	T	T
c_2: x <= 300?	-	F	T	T	T	T	T	T	T	T	T	T	T	T
c_3: y > = 1?	-	-	F	T	T	T	T	T	T	T	T	T	T	T
c_4: y <= 300?	-	-	-	F	T	T	T	T	T	T	T	T	T	T
c_5: z > = 1?	-	-	-	-	F	T	T	T	T	T	T	T	T	T
c_6: z <= 300?	-	-	-	-	-	F	T	T	T	T	T	T	T	T
c_7: x>y?	-	-	-	-	-	-	T	T	T	T	F	F	F	F
c_8: y>z?	-	-	-	-	-	-	T	T	F	F	T	T	F	F
c_9: z>x?	-	-	-	-	-	-	T	F	T	F	T	F	T	F
Rule Count	256	128	64	32	16	8	1	1	1	1	1	1	1	1
a_1 : Invalid input	X	X	X	X	X	X								
a_2 : x is largest								X		X				
a_3 : y is largest											X	X		
a_4 : z is largest									X				X	
a_5 : Impossible							X							X

Table 2.34. Test cases of the given problem

Test Case	x	y	z	Expected Output
1.	0	50	50	Invalid marks
2.	301	50	50	Invalid marks
3.	50	0	50	Invalid marks
4.	50	301	50	Invalid marks
5.	50	50	0	Invalid marks
6.	50	50	301	Invalid marks
7.	?	?	?	Impossible
8.	150	130	110	150
9.	150	130	170	170
10.	150	130	140	150
11.	110	150	140	150
12.	140	150	120	150
13.	120	140	150	150
14.	?	?	?	Impossible

Example 2.14: Consider the problem for determining the division of the student in example 2.2. Identify the test cases using the decision table based testing.

Solution: This problem can be solved using either limited entry decision table or extended entry decision table. The effectiveness of any solution is dependent upon the creation of various conditions. The limited entry decision table is given in Table 2.35 and its associated test cases are given in Table 2.36. The impossible inputs are shown by '?' as given in test cases 8, 9, 10, 12, 13, 14, 16, 17, 19 and 22. There are 11 impossible test cases out of 22 test cases which is a very large number and compel us to look for other solutions.

Table 2.35. Limited entry decision table

Conditions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
c_1 : mark1 > = 0 ?	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
c_2 : mark1 < = 100 ?	-	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
c_3 : mark2 > = 0 ?	-	-	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
c_4 : mark2 < = 100 ?	-	-	-	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
c_5 : mark3 > = 0 ?	-	-	-	-	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
c_6 : mark3 < = 100?	-	-	-	-	-	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
c_7 : 0 ≤ avg ≤ 39 ?	-	-	-	-	-	T	T	F	T	T	T	F	F	F	F	F	F	F	F	F	F	F
c_8 : 40 ≤ avg ≤ 49 ?	-	-	-	-	-	-	T	F	F	F	F	T	T	T	T	F	F	F	F	F	F	F
c_9 : 50 ≤ avg ≤ 59 ?	-	-	-	-	-	-	-	T	F	F	F	T	T	F	F	T	T	T	F	F	F	F
c_{10} : 60 ≤ avg ≤ 74 ?	-	-	-	-	-	-	-	-	T	F	F	-	F	T	F	T	F	F	T	T	F	F
c_{11} : avg ≥ 75 ?	-	-	-	-	-	-	-	-	-	T	F	-	-	F	F	-	T	F	T	F	T	F
Rule Count	1024	512	256	128	64	32	8	4	2	1	1	4	2	1	1	2	1	1	1	1	1	1
a_1 : Invalid marks	X	X	X	X	X	X																
a_2 : First division with distinction																					X	
a_3 : First division																				X		
a_4 : Second division																		X				
a_5 : Third division															X							
a_6 : Fail											X											
a_7 : Impossible							X	X	X	X		X	X	X		X	X		X			X

There are 22 test cases corresponding to each column in the decision table. The test cases are given in Table 2.36.

Table 2.36. Test cases of the given problem

Test Case	mark1	mark2	mark3	Expected Output
1.	-1	50	50	Invalid marks
2.	101	50	50	Invalid marks
3.	50	-1	50	Invalid marks
4.	50	101	50	Invalid marks
5.	50	50	-1	Invalid marks
6.	50	50	101	Invalid marks
7.	?	?	?	Impossible
8.	?	?	?	Impossible
9.	?	?	?	Impossible
10.	?	?	?	Impossible
11.	25	25	25	Fail
12.	?	?	?	Impossible
13.	?	?	?	Impossible
14.	?	?	?	Impossible
15.	45	45	45	Third division
16.	?	?	?	Impossible
17.	?	?	?	Impossible
18.	55	55	55	Second division
19.	?	?	?	Impossible
20.	65	65	65	First division
21.	80	80	80	First division with distinction
22.	?	?	?	Impossible

The input domain may be partitioned into the following equivalence classes:

$I_1 = \{ A1 : 0 \leq mark1 \leq 100 \}$

$I_2 = \{ A2 : mark1 < 0 \}$

$I_3 = \{ A3 : mark1 > 100 \}$

$I_4 = \{ B1 : 0 \leq mark2 \leq 100 \}$

$I_5 = \{B2 : mark2 < 0 \}$

$I_6 = \{ B3 : mark2 > 100 \}$

$I_7 = \{ C1 : 0 \leq mark3 \leq 100 \}$

$I_8 = \{ C2 : mark3 < 0 \}$

$I_9 = \{ C3 : mark3 > 100 \}$

$I_{10} = \{ D1 : 0 \leq avg \leq 39 \}$

$I_{11} = \{ D2 : 40 \leq avg \leq 49 \}$

$I_{12} = \{ D3 : 50 \leq avg \leq 59 \}$

$I_{13} = \{ D4 : 60 \leq avg \leq 74\}$

$I_{14} = \{ D5 : avg \geq 75 \}$

The extended entry decision table is given in Table 2.37.

Table 2.37. Extended entry decision table

Conditions	1	2	3	4	5	6	7	8	9	10	11
c_1 : mark1 in	A1	A1	A1	A1	A1	A1	A1	A1	A1	A2	A3
c_2 : mark 2 in	B1	B1	B1	B1	B1	B1	B1	B2	B3	-	-
c_3 : mark3 in	C1	C1	C1	C1	C1	C2	C3	-	-	-	-
c_4 : avg in	D1	D2	D3	D4	D5	-	-	-	-	-	-
Rule Count	1	1	1	1	1	5	5	15	15	45	45
a_1: Invalid Marks						X	X	X	X	X	X
a_2: First Division with Distinction					X						
a_3: First Division				X							
a_4: Second Division			X								
a_5: Third Division		X									
a_6: Fail	X										

Here $2^{\text{numbers of do not care conditions}}$ formula cannot be applied because this is an extended entry decision table where multiple conditions are used. We have made equivalence classes for mark1, mark2, mark3 and average value. In column 6, rule count is 5 because "average value" is 'do not care' otherwise the following combinations should have been shown:

A1, B1, C2, D1
A1, B1, C2, D2
A1, B1, C2, D3

A1, B1, C2, D4
A1, B1, C2, D5

These five combinations have been replaced by a 'do not care' condition for average value (D) and the result is shown as A1, B1, C2, —. Hence, rule count for extended decision table is given as:

Rule count = Cartesian product of number of equivalence classes of entries having 'do not care' conditions.

The test cases are given in Table 2.38. There are 11 test cases as compared to 22 test cases given in Table 2.36.

Table 2.38. Test cases of the given problem

Test Case	mark1	mark2	mark3	Expected Output
1.	25	25	25	Fail
2.	45	45	45	Third Division
3.	55	55	55	Second Division
4.	65	65	65	First Division
5.	80	80	80	First Division with Distinction
6.	50	50	-	Invalid marks
7.	50	50	101	Invalid marks
8.	50	-	50	Invalid marks
9.	50	101	50	Invalid marks
10.	-	50	50	Invalid marks
11.	101	50	50	Invalid marks

Example 2.15: Consider the program for classification of a triangle in example 2.3. Design the test cases using decision table based testing.

Solution: We may also choose conditions which include an invalid range of input domain, but this will increase the size of the decision table as shown in Table 2.39. We add an action to show that the inputs are out of range.

The decision table is given in Table 2.39 and has the corresponding test cases that are given in Table 2.40. The number of test cases is equal to the number of columns in the decision table. Hence, 17 test cases can be generated.

In the decision table given in Table 2.39, we assumed that 'a' is the longest side. This time we do not make this assumption and take all the possible conditions into consideration i.e. any of the sides 'a', 'b' or 'c' can be longest. It has 31 rules as compared to the 17 given in Table 2.40. The full decision table is given in Table 2.41. The corresponding 55 test cases are given in Table 2.42.

Table 2.39. Decision table

Conditions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
c_1: a<b+c?	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
c_2: b<c+a?	-	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
c_3: c<a+b?	-	-	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T
c_4: a > 0?	-	-	-	F	T	T	T	T	T	T	T	T	T	T	T	T	T
c_5: a < = 100?	-	-	-	-	F	T	T	T	T	T	T	T	T	T	T	T	T
c_6: b > 0?	-	-	-	-	-	F	T	T	T	T	T	T	T	T	T	T	T
c_7: b < = 100?	-	-	-	-	-	-	F	T	T	T	T	T	T	T	T	T	T
c_8: c > 0?	-	-	-	-	-	-	-	F	T	T	T	T	T	T	T	T	T
c_9: c < = 100?	-	-	-	-	-	-	-	-	F	T	T	T	T	T	T	T	T
c_{10}: $a^2 = b^2+c^2$?	-	-	-	-	-	-	-	-	-	T	T	T	T	F	F	F	F
c_{11}: $a^2 > b^2+c^2$?	-	-	-	-	-	-	-	-	-	T	T	F	F	T	T	F	F
c_{12}: $a^2 < b^2+c^2$?	-	-	-	-	-	-	-	-	-	T	F	T	F	T	F	T	F
Rule Count	1048	1024	512	256	128	64	32	16	8	1	1	1	1	1	1	1	1
a_1 : Invalid Triangle	X	X	X														
a_2 : Input(s) out of range				X	X	X	X	X	X								
a_3 : Right angled triangle													X				
a_4 : Obtuse angled triangle															X		
a_5 : Acute angled triangle																X	
a_6 : Impossible										X	X	X		X			X

Table 2.40. Test cases

Test Case	a	b	c	Expected Output
1.	90	40	40	Invalid Triangle
2.	40	90	40	Invalid Triangle
3.	40	40	90	Invalid Triangle
4.	0	50	50	Input(s) out of Range
5.	101	50	50	Input(s) out of Range
6.	50	0	50	Input(s) out of Range
7.	50	101	50	Input(s) out of Range
8.	50	50	0	Input(s) out of Range
9.	50	50	101	Input(s) out of Range
10.	?	?	?	Impossible
11.	?	?	?	Impossible
12.	?	?	?	Impossible
13.	50	40	30	Right Angled Triangle
14.	?	?	?	Impossible
15.	57	40	40	Obtuse Angled Triangle
16.	50	49	49	Acute Angled Triangle
17.	?	?	?	Impossible

Table 2.41. Modified decision table

Conditions	1	2	3	4	5	6	7	8	9	10	11
c_1: a < b+c?	F	T	T	T	T	T	T	T	T	T	T
c_2: b < c+a?	-	F	T	T	T	T	T	T	T	T	T
c_3: c < a+b?	-	-	F	T	T	T	T	T	T	T	T
c_4: a > 0?	-	-	-	F	T	T	T	T	T	T	T
c_5: a <= 100?	-	-	-	-	F	T	T	T	T	T	T
c_6: b > 0?	-	-	-	-	-	F	T	T	T	T	T
c_7: b <= 100?	-	-	-	-	-	-	F	T	T	T	T
c_8: c > 0?	-	-	-	-	-	-	-	F	T	T	T
c_9: c <= 100?	-	-	-	-	-	-	-	-	F	T	T
c_{10}: $a^2 = b^2+c^2$?	-	-	-	-	-	-	-	-	-	T	T
c_{11}: $b^2 = c^2+a^2$?	-	-	-	-	-	-	-	-	-	T	F
c_{12}: $c^2 = a^2+b^2$?	-	-	-	-	-	-	-	-	-	-	T
c_{13}: $a^2 > b^2+c^2$?	-	-	-	-	-	-	-	-	-	-	-
c_{14}: $b^2 > c^2+a^2$?	-	-	-	-	-	-	-	-	-	-	-
c_{15}: $c^2 > a^2+b^2$?	-	-	-	-	-	-	-	-	-	-	-
Rule Count	16384	8192	4096	2048	1024	512	256	128	64	16	8
a_1 : Invalid triangle	X	X	X								
a_2 : Input(s) out of range				X	X	X	X	X	X		
a_3 : Right angled triangle											
a_4 : Obtuse angled triangle											
a_5 : Acute angled triangle											
a_6 : Impossible										X	X

(Contd.)

(*Contd.*)

Conditions	12	13	14	15	16	17	18	19	20	21	22	23	24
c_1: a < b+c?	T	T	T	T	T	T	T	T	T	T	T	T	T
c_2: b < c+a?	T	T	T	T	T	T	T	T	T	T	T	T	T
c_3: c < a+b?	T	T	T	T	T	T	T	T	T	T	T	T	T
c_4: a > 0?	T	T	T	T	T	T	T	T	T	T	T	T	T
c_5: a <= 100?	T	T	T	T	T	T	T	T	T	T	T	T	T
c_6: b > 0?	T	T	T	T	T	T	T	T	T	T	T	T	T
c_7: b <= 100?	T	T	T	T	T	T	T	T	T	T	T	T	T
c_8: c > 0?	T	T	T	T	T	T	T	T	T	T	T	T	T
c_9: c <= 100?	T	T	T	T	T	T	T	T	T	T	T	T	T
c_{10}: $a^2 = b^2+c^2$?	T	T	T	T	F	F	F	F	F	F	F	F	F
c_{11}: $b^2 = c^2+a^2$?	F	F	F	F	T	T	T	T	T	F	F	F	F
c_{12}: $c^2 = a^2+b^2$?	F	F	F	F	T	F	F	F	F	T	T	T	T
c_{13}: $a^2 > b^2+c^2$?	T	F	F	F	-	T	F	F	F	T	F	F	F
c_{14}: $b^2 > c^2+a^2$?	-	T	F	F	-	-	T	F	T	-	T	F	F
c_{15}: $c^2 > a^2+b^2$?	-	-	T	F	-	-	-	T	F	-	-	T	F
Rule Count	4	2	1	1	8	4	2	1	1	4	2	1	1
a_1 : Invalid triangle													
a_2 : Input(s) out of range													
a_3 : Right angled triangle				X					X				X
a_4 : Obtuse angled triangle													
a_5 : Acute angled triangle													
a_6 : Impossible	X	X	X		X	X	X	X		X	X	X	

Conditions	25	26	27	28	29	30	31
c_1: a < b+c?	T	T	T	T	T	T	T
c_2: b < c+a?	T	T	T	T	T	T	T
c_3: c < a+b?	T	T	T	T	T	T	T
c_4: a > 0?	T	T	T	T	T	T	T
c_5: a <= 100?	T	T	T	T	T	T	T
c_6: b > 0?	T	T	T	T	T	T	T
c_7: b <= 100?	T	T	T	T	T	T	T
c_8: c > 0?	T	T	T	T	T	T	T
c_9: c <= 100?	T	T	T	T	T	T	T
c_{10}: $a^2 = b^2+c^2$?	F	F	F	F	F	F	F
c_{11}: $b^2 = c^2+a^2$?	F	F	F	F	F	F	F
c_{12}: $c^2 = a^2+b^2$?	F	F	F	F	F	F	F

(*Contd.*)

(*Contd.*)

Conditions	25	26	27	28	29	30	31
c_{13}: $a^2 > b^2+c^2$?	T	T	T	F	F	F	F
c_{14}: $b^2 > c^2+a^2$?	T	F	F	T	T	F	F
c_{15}: $c^2 > a^2+b^2$?	-	T	F	T	F	T	F
Rule Count	2	1	1	1	1	1	1
a_1 : Invalid triangle							
a_2 : Input(s) out of range							
a_3 : Right angled triangle							
a_4 : Obtuse angled triangle			X		X	X	
a_5 : Acute angled triangle							X
a_6 : Impossible	X	X		X			

The table has 31 columns (total = 32768)

Table 2.42. Test cases of the decision table given in table 2.41

Test Case	a	b	c	Expected Output
1.	90	40	40	Invalid Triangle
2.	40	90	40	Invalid Triangle
3.	40	40	90	Invalid Triangle
4.	0	50	50	Input(s) out of Range
5.	101	50	50	Input(s) out of Range
6.	50	0	50	Input(s) out of Range
7.	50	101	50	Input(s) out of Range
8.	50	50	0	Input(s) out of Range
9.	50	50	101	Input(s) out of Range
10.	?	?	?	Impossible
11.	?	?	?	Impossible
12.	?	?	?	Impossible
13.	?	?	?	Impossible
14.	?	?	?	Impossible
15.	50	40	30	Right Angled Triangle
16.	?	?	?	Impossible
17.	?	?	?	Impossible
18.	?	?	?	Impossible
19.	?	?	?	Impossible
20.	40	50	30	Right Angled Triangle

(*Contd.*)

(*Contd.*)

Test Case	a	b	c	Expected Output
21.	?	?	?	Impossible
22.	?	?	?	Impossible
23.	?	?	?	Impossible
24.	40	30	50	Right Angled Triangle
25.	?	?	?	Impossible
26.	?	?	?	Impossible
27.	57	40	40	Obtuse Angled Triangle
28.	?	?	?	Impossible
29.	40	57	40	Obtuse Angled Triangle
30.	40	40	57	Obtuse Angled Triangle
31.	50	49	49	Acute Angled Triangle

Example 2.16: Consider a program for the determination of day of the week specified in example 2.4. Identify the test cases using decision table based testing.

Solution: The input domain can be divided into the following classes:

$I_1 =$ { M1 : month has 30 days }

$I_2 =$ { M2 : month has 31 days }

$I_3 =$ { M3 : month is February }

$I_4 =$ { M4 : month <1 }

$I_5 =$ { M5 : month > 12 }

$I_6 =$ { D1 : $1 \leq$ Day ≤ 28 }

$I_7 =$ { D2 : Day = 29 }

$I_8 =$ { D3 : Day = 30 }

$I_9 =$ { D4 : Day = 31 }

$I_{10} =$ { D5 : Day < 1 }

$I_{11} =$ { D6 : Day > 31 }

$I_{12} =$ { Y1 : $1900 \leq$ Year ≤ 2058 and is a common year }

$I_{13} =$ { Y2 : $1900 \leq$ Year ≤ 2058 and is a leap year }

I_{14} : { Y3 : Year < 1900 }

I_{15} : { Y4 : year > 2058 }

The decision table is given in Table 2.43 and the corresponding test cases are given in Table 2.44.

Table 2.43. Decision table

Test Case	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
c_1 : Months in	M1	M1	M1	M1	M1	M1	M1	M1	M1	M1	M1	M1	M1	M1	M1	M1	M1	M1	M2	M2
c_2 : Days in	D1	D1	D1	D1	D2	D2	D2	D2	D3	D3	D3	D3	D4	D4	D4	D4	D5	D6	D1	D1
c_3 : Years in	Y1	Y2	Y3	Y4	Y1	Y2	Y3	Y4	Y1	Y2	Y3	Y4	Y1	Y2	Y3	Y4	-	-	Y1	Y2
Rule Count	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	4	4	1	1
a_1 : Invalid Date													X	X						
a_2 : Day of the week	X	X			X	X			X	X									X	X
a_3 : Input out of range			X	X			X	X			X	X			X	X	X	X		

Test Case	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
c_1 : Months in	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M3
c_2 : Days in	D1	D1	D2	D2	D2	D2	D3	D3	D3	D3	D4	D4	D4	D4	D5	D6	D1
c_3 : Years in	Y3	Y4	Y1	Y2	Y3	Y4	Y1	Y2	Y3	Y4	Y1	Y2	Y3	Y4	-	-	Y1
Rule Count	1	1	1	1	1	1	1	1	1	1	1	1	1	1	4	4	1
a_1 : Invalid Date																	
a_2 : Day of the week			X	X			X	X			X	X					X
a_3 : Input out of range	X	X			X	X			X	X			X	X	X	X	

Test Case	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56
c_1 : Months in	M3	M3	M3	M3	M3	M3	M3	M3	M3	M3	M3	M3	M3	M3	M3	M3	M3	M4	M5
c_2 : Days in	D1	D1	D1	D2	D2	D2	D2	D3	D3	D3	D3	D4	D4	D4	D4	D5	D6	-	-
c_3 : Years in	Y2	Y3	Y4	Y1	Y2	Y3	Y4	Y1	Y2	Y3	Y4	Y1	Y2	Y3	Y4	-	-	-	-
Rule Count	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	24	24
a_1: Invalid Date				X				X	X			X	X						
a_2 : Day of the week	X				X														
a_3 : Input out of range		X	X			X	X			X	X			X	X	X	X	X	X

Table 2.44. Test cases of the program day of the week

Test Case	month	day	year	Expected Output
1.	6	15	1979	Friday
2.	6	15	2000	Thursday
3.	6	15	1899	Input out of range
4.	6	15	2059	Input out of range
5.	6	29	1979	Friday
6.	6	29	2000	Thursday
7.	6	29	1899	Input out of range
8.	6	29	2059	Input out of range
9.	6	30	1979	Saturday
10.	6	30	2000	Friday
11.	6	30	1899	Input out of range
12.	6	30	2059	Input out of range
13.	6	31	1979	Invalid date
14.	6	31	2000	Invalid date
15.	6	31	1899	Input out of range
16.	6	31	2059	Input out of range
17.	6	0	1979	Input out of range
18.	6	32	1979	Input out of range
19.	5	15	1979	Tuesday
20.	5	15	2000	Monday
21.	5	15	1899	Input out of range
22.	5	15	2059	Input out of range
23.	5	29	1979	Tuesday
24.	5	29	2000	Monday
25.	5	29	1899	Input out of range
26.	5	29	2059	Input out of range
27.	5	30	1979	Wednesday
28.	5	30	2000	Tuesday
29.	5	30	1899	Input out of range
30.	5	30	2059	Input out of range
31.	5	31	1979	Thursday
32.	5	31	2000	Wednesday
33.	5	31	1899	Input out of range
34.	5	31	2059	Input out of range
35.	5	0	1979	Input out of range
36.	5	32	1979	Input out of range
37.	2	15	1979	Thursday
38.	2	15	2000	Tuesday
39.	2	15	1899	Input out of range
40.	2	15	2059	Input out of range
41.	2	29	1979	Invalid date
42.	2	29	2000	Tuesday
43.	2	29	1899	Input out of range
44.	2	29	2059	Input out of range
45.	2	30	1979	Invalid date

(*Contd.*)

(*Contd.*)

Test Case	month	day	year	Expected Output
46.	2	30	2000	Invalid date
47.	2	30	1899	Input out of range
48.	2	30	2059	Input out of range
49.	2	31	1979	Invalid date
50.	2	31	2000	Invalid date
51.	2	31	1899	Input out of range
52.	2	31	2059	Input out of range
53.	2	0	1979	Input out of range
54.	2	32	1979	Input out of range
55.	0	0	1899	Input out of range
56.	13	32	1899	Input out of range

The product of number of partitions of each input variable (or equivalence classes) is 120. The decision table has 56 columns and 56 corresponding test cases are shown in Table 2.44.

2.4 CAUSE-EFFECT GRAPHING TECHNIQUE

This technique is a popular technique for small programs and considers the combinations of various inputs which were not available in earlier discussed techniques like boundary value analysis and equivalence class testing. Such techniques do not allow combinations of inputs and consider all inputs as independent inputs. Two new terms are used here and these are causes and effects, which are nothing but inputs and outputs respectively. The steps for the generation of test cases are given in Figure 2.11.

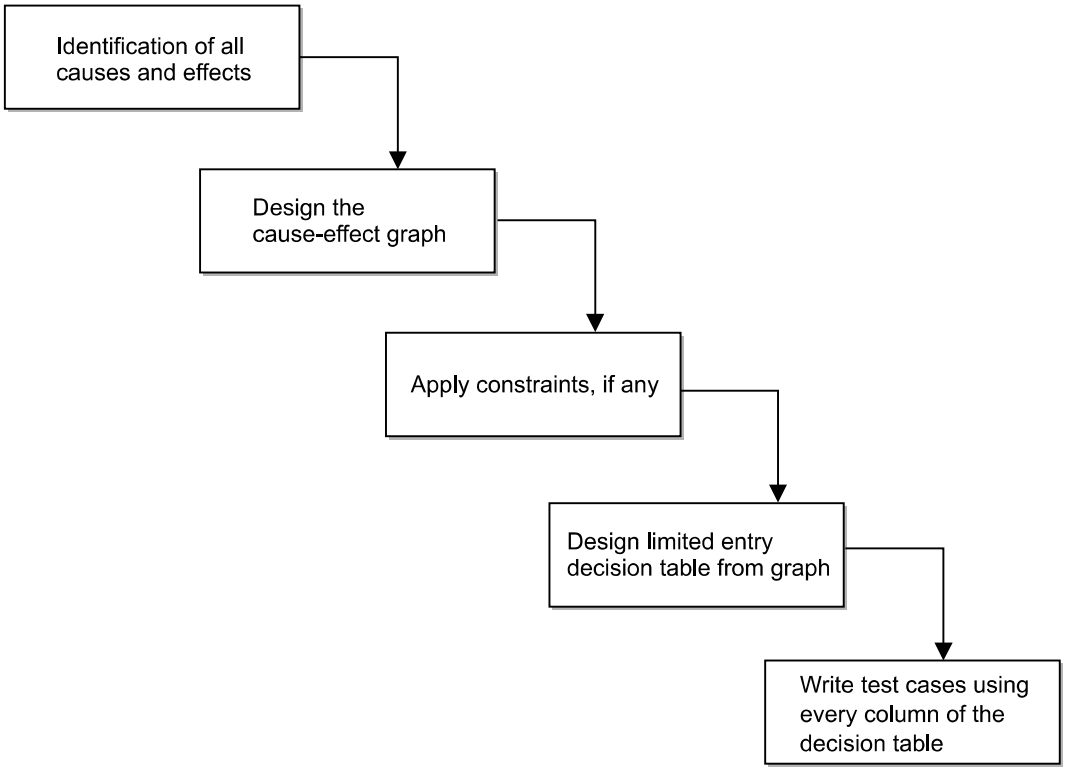


Figure 2.11. Steps for the generation of test cases

2.4.1 Identification of Causes and Effects

The SRS document is used for the identification of causes and effects. Causes which are inputs to the program and effects which are outputs of the program can easily be identified after reading the SRS document. A list is prepared for all causes and effects.

2.4.2 Design of Cause-Effect Graph

The relationship amongst causes and effects are established using cause-effect graph. The basic notations of the graph are shown in Figure 2.12.

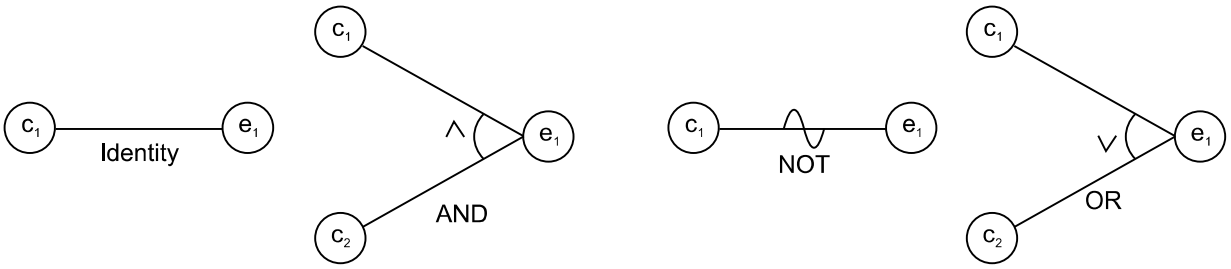


Figure 2.12. Basic notations used in cause-effect graph

In Figure 2.12, each node represents either true (present) or false (absent) state and may be assigned 1 and 0 value respectively. The purpose of four functions is given as:

(a) Identity: This function states that if c_1 is 1, then e_1 is 1; else e_1 is 0.
(b) NOT: This function states that if c_1 is 1, then e_1 is 0; else e_1 is 1.
(c) AND: This function states that if both c_1 and c_2 are 1, then e_1 is 1; else e_1 is 0.
(d) OR: This function states that if either c_1 or c_2 is 1, then e_1 is 1; else e_1 is 0.

The AND and OR functions are allowed to have any number of inputs.

2.4.3 Use of Constraints in Cause-Effect Graph

There may be a number of causes (inputs) in any program. We may like to explore the relationships amongst the causes and this process may lead to some impossible combinations of causes. Such impossible combinations or situations are represented by constraint symbols which are given in Figure 2.13.

The purpose of all five constraint symbols is given as:

(a) Exclusive
The Exclusive (E) constraint states that at most one of c_1 or c_2 can be 1 (c_1 or c_2 cannot be 1 simultaneously). However, both c_1 and c_2 can be 0 simultaneously.

(b) Inclusive
The Inclusive (I) constraints states that at least one of c_1 or c_2 must always be 1. Hence, both cannot be 0 simultaneously. However, both can be 1.

(c) One and Only One
The one and only one (O) constraint states that one and only one of c_1 and c_2 must be 1.

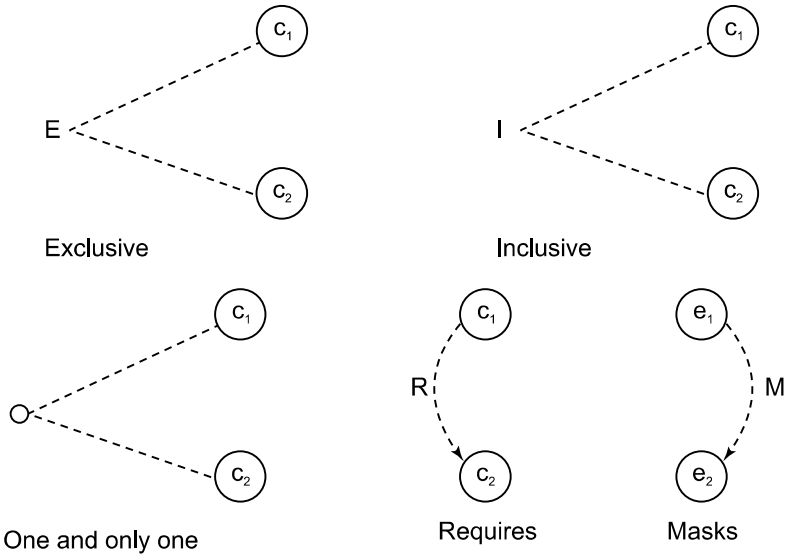


Figure 2.13. Constraint symbols for any cause-effect graph

(d) Requires
The requires (R) constraint states that for c_1 to be 1, c_2 must be 1; it is impossible for c_1 to be 1 if c_2 is 0.

(e) Mask
This constraint is applicable at the effect side of the cause-effect graph. This states that if effect e_1 is 1, effect e_2 is forced to be 0.

These five constraint symbols can be applied to a cause-effect graph depending upon the relationships amongst causes (a, b, c and d) and effects (e). They help us to represent real life situations in the cause-effect graph.

Consider the example of keeping the record of marital status and number of children of a citizen. The value of marital status must be 'U' or 'M'. The value of the number of children must be digit or null in case a citizen is unmarried. If the information entered by the user is correct then an update is made. If the value of marital status of the citizen is incorrect, then the error message 1 is issued. Similarly, if the value of number of children is incorrect, then the error message 2 is issued.

The causes are:

c_1: marital status is 'U'
c_2: marital status is 'M'
c_3: number of children is a digit

and the effects are:

e_1: updation made
e_2: error message 1 is issued
e_3: error message 2 is issued

The cause-effect graph is shown in Figure 2.14. There are two constraints exclusive (between c_1 and c_2) and requires (between c_3 and c_2), which are placed at appropriate places in the graph. Causes c_1 and c_2 cannot occur simultaneously and for cause c_3 to be true, cause c_2 has to be true. However, there is no mask constraint in this graph.

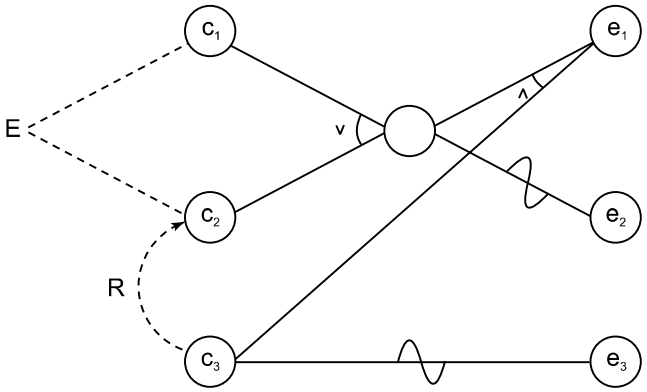


Figure 2.14. Example of cause-effect graph with exclusive (constraint) and requires constraint

2.4.4 Design of Limited Entry Decision Table

The cause-effect graph represents the relationships amongst the causes and effects. This graph may also help us to understand the various conditions/combinations amongst the causes and effects. These conditions/combinations are converted into the limited entry decision table. Each column of the table represents a test case.

2.4.5 Writing of Test Cases

Each column of the decision table represents a rule and gives us a test case. We may reduce the number of columns with the proper selection of various conditions and expected actions.

2.4.6 Applicability

Cause-effect graphing is a systematic method for generating test cases. It considers dependency of inputs using some constraints.

This technique is effective only for small programs because, as the size of the program increases, the number of causes and effects also increases and thus complexity of the cause-effect graph increases. For large-sized programs, a tool may help us to design the cause-effect graph with the minimum possible complexity.

It has very limited applications in unit testing and hardly any application in integration testing and system testing.

Example 2.17: A tourist of age greater than 21 years and having a clean driving record is supplied a rental car. A premium amount is also charged if the tourist is on business, otherwise it is not charged.

If the tourist is less than 21 year old, or does not have a clean driving record, the system will display the following message:
"Car cannot be supplied"
Draw the cause-effect graph and generate test cases.

Solution: The causes are

c_1: Age is over 21
c_2: Driving record is clean
c_3: Tourist is on business

and effects are

e_1: Supply a rental car without premium charge.
e_2: Supply a rental car with premium charge
e_3: Car cannot be supplied

The cause-effect graph is shown in Figure 2.15 and decision table is shown in Table 2.45. The test cases for the problem are given in Table 2.46.

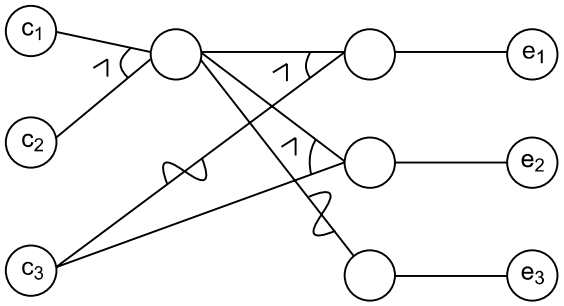


Figure 2.15. Cause-effect graph of rental car problem

Table 2.45. Decision table of rental car problem

Conditions	1	2	3	4
c_1 : Over 21 ?	F	T	T	T
c_2 : Driving record clean ?	-	F	T	T
c_3 : On Business ?	-	-	F	T
e_1 : Supply a rental car without premium charge			X	
e_2 : Supply a rental car with premium charge				X
e_3 : Car cannot be supplied	X	X		

Table 2.46. Test cases of the given decision table

Test Case	Age	Driving_record_clean	On_business	Expected Output
1.	20	Yes	Yes	Car cannot be supplied
2.	26	No	Yes	Car cannot be supplied
3.	62	Yes	No	Supply a rental car without premium charge
4.	62	Yes	Yes	Supply a rental car with premium charge.

Example 2.18: Consider the triangle classification problem ('a' is the largest side) specified in example 2.3. Draw the cause-effect graph and design decision table from it.

Solution:

The causes are:

c_1 : side 'a' is less than the sum of sides 'b' and 'c'.
c_2 : side 'b' is less than the sum of sides 'a' and 'c'.
c_3 : side 'c' is less than the sum of sides 'a' and 'b'.
c_4 : square of side 'a' is equal to the sum of squares of sides 'b' and 'c'.
c_5 : square of side 'a' is greater than the sum of squares of sides 'b' and 'c'.
c_6 : square of side 'a' is less than the sum of squares of sides 'b' and 'c'.
and effects are
e_1 : Invalid Triangle
e_2 : Right angle triangle
e_3 : Obtuse angled triangle
e_4 : Acute angled triangle
e_5 : Impossible stage

The cause-effect graph is shown in Figure 2.16 and the decision table is shown in Table 2.47.

Table 2.47. Decision table

Conditions											
c_1 : a<b+c	0	1	1	1	1	1	1	1	1	1	1
c_2 : b<a+c	X	0	1	1	1	1	1	1	1	1	1
c_3 : c<a+b	X	X	0	1	1	1	1	1	1	1	1
c_4 : $a^2=b^2+c^2$	X	X	X	1	1	1	1	0	0	0	0
c_5 : $a^2>b^2+c^2$	X	X	X	1	1	0	0	1	1	0	0
c_6 : $a^2<b^2+c^2$	X	X	X	1	0	1	0	1	0	1	0
e_1 : Invalid Triangle	1	1	1								
e_2 : Right angled Triangle							1				
e_3 : Obtuse angled triangle									1		
e_4 : Acute angled triangle										1	
e_5 : Impossible				1	1	1		1			1

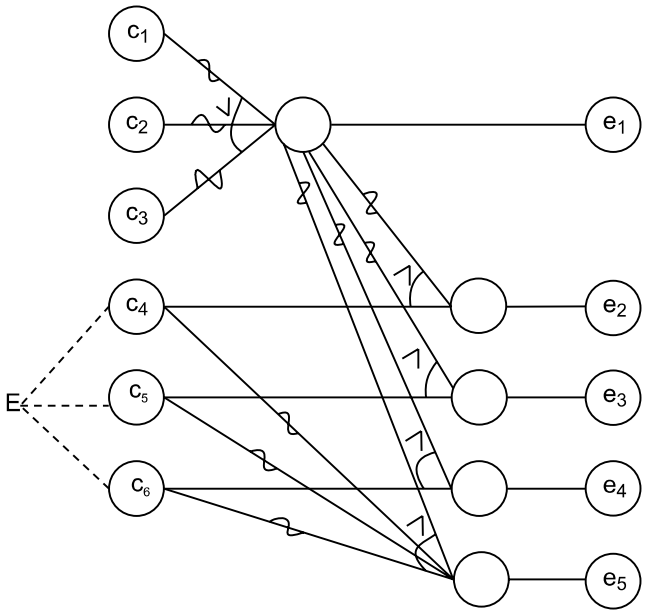


Figure 2.16. Cause-effect graph of triangle classification problem

3

Essentials of Graph Theory

Graph theory has been used extensively in computer science, electrical engineering, communication systems, operational research, economics, physics and many other areas. Any physical situation involving discrete objects may be represented by a graph along with their relationships amongst them. In practice, there are numerous applications of graphs in modern science and technology. Graph theory has recently been used for representing the connectivity of the World Wide Web. Global internet connectivity issues are studied using graphs like the number of links required to move from one web page to another and the links which are used to establish this connectivity. It has also provided many ways to test a program. Some testing techniques are available which are based on the concepts of graph theory.

3.1 WHAT IS A GRAPH?

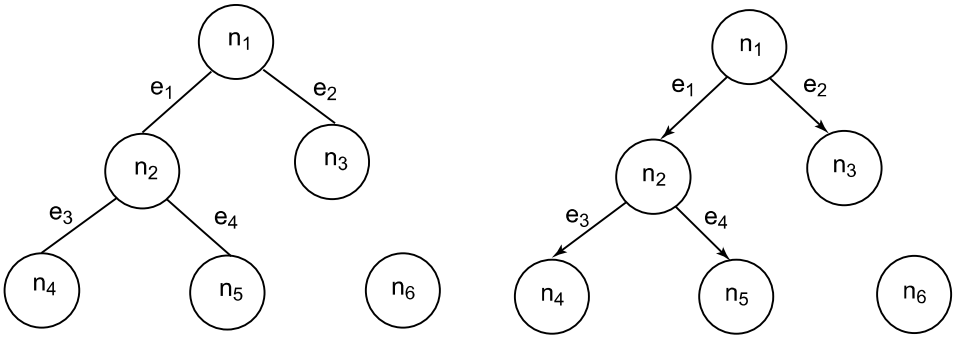
A graph has a set of nodes and a set of edges that connect these nodes. A graph G = (V, E) consists of a non-empty finite set V of nodes and a set E of edges containing ordered or unordered pairs of nodes.

$$V = (n_1, n_2, n_3 \ldots\ldots n_m) \text{ and } E = (e_1, e_2, e_3 \ldots\ldots\ldots e_k)$$

If an edge $e_i \in E$ is associated with an ordered pair $<n_i, n_j>$ or an unordered pair (n_i, n_j), where $n_i, n_j \in V$, then the e_i is said to connect the nodes n_i and n_j. The edge e_i that connects the node n_i and n_j is called incident on each of the nodes. The pair of nodes that are connected by an edge are called adjacent nodes. A node, which is not connected to any other node, is called an isolated node. A graph with only isolated nodes is known as null graph.

If in graph G = (V, E), each edge $e_i \in E$ is associated with an ordered pair of nodes, then graph G is called a directed graph or digraph. If each edge is associated with an unordered pair of nodes, then a graph G is called an undirected graph.

In diagrammatical representation of a graph, nodes are represented by circles and edges are represented by lines joining the two nodes incident on it and the same is shown in Figure 3.1.



(a) Undirected graph with 6 nodes and 4 edges

(b) Directed graph with 6 nodes and 4 edges

Figure 3.1. Undirected and directed graphs

In the Figure 3.1(a), the node and edge sets are given as:

$$V = (n_1, n_2, n_3, n_4, n_5, n_6)$$

$$E = (e_1, e_2, e_3, e_4) = ((n_1, n_2), (n_1, n_3), (n_2, n_4), (n_2, n_5))$$

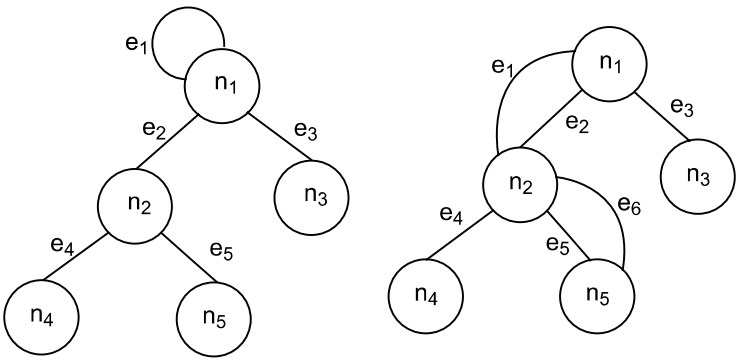
Similarly in Figure 3.1(b), the node and edge sets are

$$V = (n_1, n_2, n_3, n_4, n_5, n_6)$$

$$E = (e_1, e_2, e_3, e_4) = (<n_1, n_2>, <n_1, n_3>, <n_2, n_4>, <n_2, n_5>)$$

The only difference is that edges are the ordered pairs of nodes represented by $<n_1, n_2>$ rather than unordered pairs (n_1, n_2). For any graph (directed or undirected), a set of nodes and a set of edges between pairs of nodes are required for the construction of the graph.

An edge of a graph having the same node at its end points is called a loop. The direction of the edge is not important because the initial and final nodes are one and the same. In Figure 3.2(a), edge e_1 is a loop with node n_1 and may be represented as $e_1 = (n_1, n_1)$.



(a) Graph with loop e_1

(b) Graph with parallel edges e_1, e_2 and e_5, e_6

Figure 3.2. Undirected graphs with loop and parallel edges

If certain pairs of nodes are connected by more than one edge in a directed or undirected graph, then these edges are known as parallel edges. In Figure 3.2(b), e_1 and e_2 are parallel edges connecting nodes n_1 and n_2. Similarly e_5 and e_6 are also parallel edges connecting nodes n_2 and n_5. If a graph is a directed graph and parallel edges are in opposite directions, such edges are considered as distinct edges. A graph that has neither loops nor parallel edges is called a simple graph. A graph with one or more parallel edges is called a multigraph. These graphs are shown in Figure 3.3.

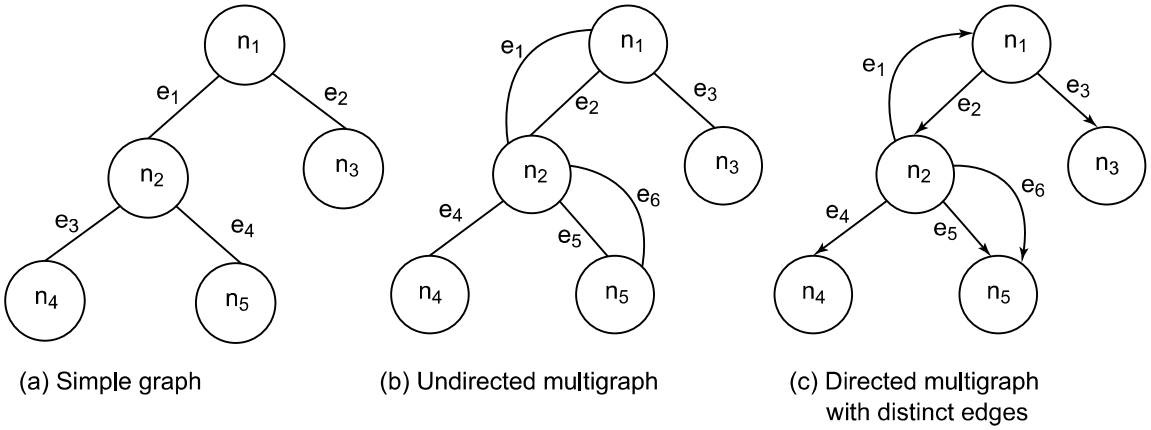


Figure 3.3. Types of graphs

A directed multigraph (see Figure 3.3 (c)) may have *distinct parallel* edges (like e_1 and e_2) and *parallel edges* (like e_5 and e_6). If numbers or weights are assigned to each edge of a graph, then such a graph is called a weighted graph.

3.1.1 Degree of a Node

The degree of a node in an undirected graph is the number of edges incident on it. However, a loop contributes twice to the degree of that node. The degree of a node 'n' is represented by deg(n). The degrees of nodes in a graph shown in Figure 3.1 (a) are given as:

$$\deg(n_1) = 2,\ \deg(n_2) = 3,\ \deg(n_3) = 1,\ \deg(n_4) = 1,\ \deg(n_5) = 1,\ \deg(n_6) = 0$$

The degree of an isolated node is always 0. In case of directed graphs, the direction of edges play an important role and indegree and outdegree of a node is calculated. Indegree of a node in a directed graph is the number of edges that are using that node as a terminal node. The indegree of a node 'n' is represented by indeg(n). The outdegree of a node in a directed graph is the number of edges that are using that node as a start node. The outdegree of a node 'n' is represented by outdeg(n). The indegrees and outdegrees of nodes in a graph shown in Figure 3.1(b) are given as:

$$\text{indeg}(n_1) = 0 \qquad \text{outdeg}(n_1) = 2$$

$$\text{indeg}(n_2) = 1 \qquad \text{outdeg}(n_2) = 2$$

$$\text{indeg}(n_3) = 1 \qquad \text{outdeg}(n_3) = 0$$

$indeg(n_4) = 1$ $outdeg(n_4) = 0$

$indeg(n_5) = 1$ $outdeg(n_5) = 0$

$indeg(n_6) = 0$ $outdeg(n_6) = 0$

The degree of a node in a directed graph is the sum of indegree and outdegree of that node. Hence, for node 'n' in a directed graph, the degree is given as:

$$deg(n) = indeg(n) + outdeg(n)$$

Few important characteristics of nodes are identified on the basis of indegree and outdegree and are given as:

(i) Source node: A node with indegree = 0 and outdegree ≥ 1
(ii) Destination node: A node with outdegree = 0 and in degree ≥ 1
(iii) Sequential node: A node with indegree = 1 and outdegree = 1
(iv) Predicate/decision node: A node with outdegree ≥ 2
(v) Junction node: indegree ≥ 2

In Figure 3.1(b), source nodes: n_1

Destination nodes: n_3, n_4, n_5

Sequential nodes: nil

Predicate nodes: n_1, n_2

There is no junction node in this graph and n_6 is an isolated node.

3.1.2 Regular Graph

If every node of a simple graph has the same degree, then the graph is called a regular graph. If the degree of a node is 'n', then it is called n-regular graph. Some of such graphs are shown in Figure 3.4.

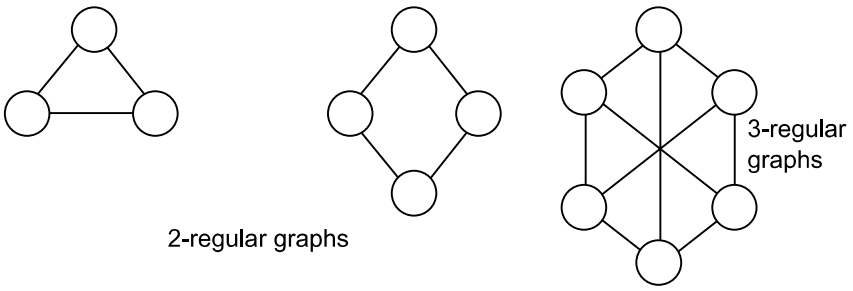


Figure 3.4. Regular graphs

3.2 MATRIX REPRESENTATION OF GRAPHS

In computer science, matrix representation of graphs is very popular due to its direct applications in programming. Each piece of information of a graph, which is shown diagrammatically, can be

converted into matrix form. Some useful matrix representations are given in the following subsections which are used commonly in testing.

3.2.1 Incidence Matrix

Consider a graph G = (V, E) where

$$V = (n_1, n_2, n_3 \ldots\ldots n_m)$$
$$E = (e_1, e_2, e_3 \ldots\ldots e_k)$$

In this graph, there are 'm' nodes and 'k' edges. An incidence matrix is a matrix with 'm' rows and 'k' columns whose elements are defined such that:

$$a(i, j) = \begin{cases} 1 \text{ if } j^{th} \text{ edge } e_j \text{ is incident on } i^{th} \text{ node } n_i \\ 0 \text{ otherwise} \end{cases}$$

The incidence matrix of a graph shown in Figure 3.1(a) is given as:

$$\begin{array}{c} \\ n_1 \\ n_2 \\ n_3 \\ n_4 \\ n_5 \\ n_6 \end{array} \begin{array}{c} \begin{array}{cccc} e_1 & e_2 & e_3 & e_4 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

The sum of entries of any column of incidence matrix is always 2. This is because a column represents an edge which has only two endpoints. If, any time, the sum is not two, then there is some mistake in the transfer of information. If we add entries of a row, which is corresponding to a node, we get the degree of that node. If all entries of a row are 0's, then the corresponding node is an isolated node. Incidence matrix of a graph may be used to calculate various properties of a graph which can further be programmed, if so desired. The incidence matrix of a directed graph is the same as the incidence matrix of an undirected graph.

3.2.2 Adjacency Matrix

It is also known as connection matrix because it represents connections in a graph. The adjacency matrix of a graph G = (V, E) with 'm' nodes is a square matrix of size m×m whose elements are defined such that:

$$a(i, j) = \begin{cases} 1 \text{ if there is an edge from nodes } n_i \text{ and } n_j \\ 0 \text{ otherwise} \end{cases}$$

The adjacency matrix of a graph shown in Figure 3.1(a) is given as:

$$\begin{array}{c} \\ n_1 \\ n_2 \\ n_3 \\ n_4 \\ n_5 \\ n_6 \end{array} \begin{array}{c} \begin{array}{cccccc} n_1 & n_2 & n_3 & n_4 & n_5 & n_6 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

If we add entries of any row, we get the degree of the node corresponding to that row. This is similar to incidence matrix. The adjacency matrix of an undirected graph is symmetric where a(i, j) = a(j, i).

If we represent connections with the name of edges, then the matrix is called graph matrix. The size is the same as adjacent matrix i.e. number of nodes in a graph. The graph matrix of a graph shown in Figure 3.1(a) is given as:

$$\begin{array}{c} \\ n_1 \\ n_2 \\ n_3 \\ n_4 \\ n_5 \\ n_6 \end{array} \begin{array}{c} \begin{array}{cccccc} n_1 & n_2 & n_3 & n_4 & n_5 & n_6 \end{array} \\ \begin{bmatrix} 0 & e_1 & e_2 & 0 & 0 & 0 \\ e_1 & 0 & 0 & e_3 & e_4 & 0 \\ e_2 & 0 & 0 & 0 & 0 & 0 \\ 0 & e_3 & 0 & 0 & 0 & 0 \\ 0 & e_4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

If connections are represented by 1, then it becomes the connection matrix.

The adjacency matrix of a directed graph G = (V, E) with 'm' nodes is a square matrix of size m×m whose elements are such that:

$$a(i, j) = \begin{cases} 1 \text{ if there is an edge from nodes } n_i \text{ and } n_j \\ 0 \text{ otherwise} \end{cases}$$

The adjacency matrix of a directed graph may not be symmetric. If we add entries of a row, it becomes outdegree of the node and if we add entries of a column, it becomes indegree of the node. The adjacency matrix of a directed graph given in Figure 3.1(b) is given as:

$$\begin{array}{c} \\ n_1 \\ n_2 \\ n_3 \\ n_4 \\ n_5 \\ n_6 \end{array} \begin{array}{c} \begin{array}{cccccc} n_1 & n_2 & n_3 & n_4 & n_5 & n_6 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

In directed graph, direction of an edge is considered, hence adjacency matrix of a directed graph is different from the undirected graph.

3.3 PATHS AND INDEPENDENT PATHS

A path in a graph is a sequence of adjacent nodes where nodes in sequence share a common edge or sequence of adjacent pair of edges where edges in sequence share a common node. The paths in a graph shown in Figure 3.1(a) are given in Table 3.1.

Table 3.1. Paths of undirected graph in Figure 3.1(a)

S.No.	Paths from initial node to final node	Sequence of nodes	Sequence of edges
1.	n_1 to n_4	n_1, n_2, n_4	e_1, e_3
2.	n_1 to n_5	n_1, n_2, n_5	e_1, e_4
3.	n_1 to n_2	n_1, n_2	e_1
4.	n_1 to n_3	n_1, n_3	e_2
5.	n_2 to n_4	n_2, n_4	e_3
6.	n_2 to n_5	n_2, n_5	e_4
7.	n_2 to n_1	n_2, n_1	e_1
8.	n_3 to n_1	n_3, n_1	e_2
9.	n_4 to n_2	n_4, n_2	e_3
10.	n_4 to n_1	n_4, n_2, n_1	e_3, e_1
11.	n_5 to n_2	n_5, n_2	e_4
12.	n_5 to n_1	n_5, n_2, n_1	e_4, e_1
13.	n_2 to n_3	n_2, n_1, n_3	e_1, e_2
14.	n_3 to n_2	n_3, n_1, n_2	e_2, e_1
15.	n_4 to n_3	n_4, n_2, n_1, n_3	e_3, e_1, e_2
16.	n_3 to n_4	n_3, n_1, n_2, n_4	e_2, e_1, e_3
17.	n_4 to n_5	n_4, n_2, n_5	e_3, e_4
18.	n_5 to n_4	n_5, n_2, n_4	e_4, e_3
19.	n_5 to n_3	n_5, n_2, n_1, n_3	e_4, e_1, e_2
20.	n_3 to n_5	n_3, n_1, n_2, n_5	e_2, e_1, e_4

A path represented by sequence of nodes is a more popular representation technique than sequence of edges.

An independent path in a graph is a path that has at least one new node or edge in its sequence from the initial node to its final node.

If there is no repetition of an edge in a path from the initial node to the final node, then the path is called a simple path. The number of edges in a path is called the length of the path.

The direction of an edge provides more meaning to a path. Hence, paths of a directed graph seem to be more practical and useful. They are also called chains. The paths in a directed graph shown in Figure 3.1(b) are given in Table 3.2.

Table 3.2. Paths of directed graph in Figure 3.1(b)

S.No.	Paths from initial node to final node	Sequence of nodes	Sequence of edges
1.	n_1 to n_4	n_1, n_2, n_4	e_1, e_3
2.	n_1 to n_5	n_1, n_2, n_5	e_1, e_4
3.	n_1 to n_2	n_1, n_2	e_1
4.	n_1 to n_3	n_1, n_3	e_2
5.	n_2 to n_4	n_2, n_4	e_3
6.	n_2 to n_5	n_2, n_5	e_4

3.3.1 Cycles

When the initial and final nodes of a path are the same and if length ≠ 0, the path is called a cycle. Consider the graph given in Figure 3.5 having 6 nodes and 6 edges with a cycle constituted by a path n_1, n_2, n_5, n_3, n_1 of length 4.

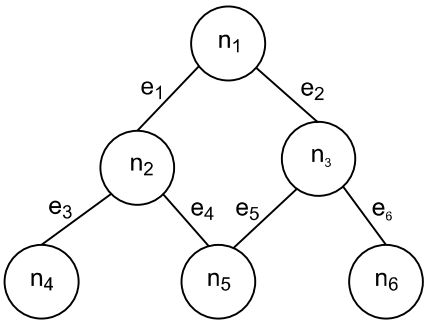


Figure 3.5. Cyclic graph

3.3.2 Connectedness of a Graph

An undirected graph is said to be connected if there is a path between every pair of nodes of the graph, otherwise it is said to be a disconnected graph. Two nodes are also said to be connected if there is a path between them.

A graph shown in Figure 3.3 (a) is a connected graph and a graph shown in Figure 3.1 (a) is a disconnected graph. The graph shown in Figure 3.6 is a disconnected graph with two portions of the graph.

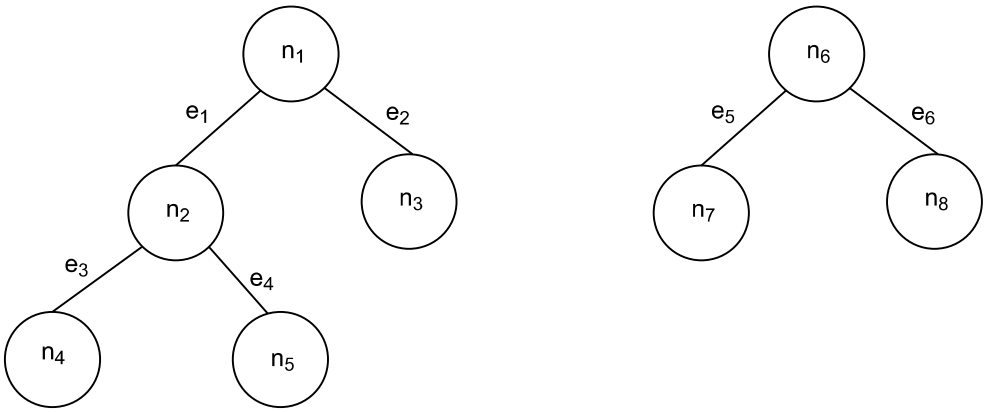


Figure 3.6. Disconnected graphs

A disconnected graph is the union of two or more disjoint portions of the graph. These disjoint portions are called the connected components of the graph.

A directed graph is said to be strongly connected if there is a path from node n_i to node n_j; where node n_i and n_j are any pair of nodes of the graph.

Every node of the graph should have a path to every other node of the graph in the strongly connected graphs. The directed graph shown in Figure 3.7 is a strongly connected graph.

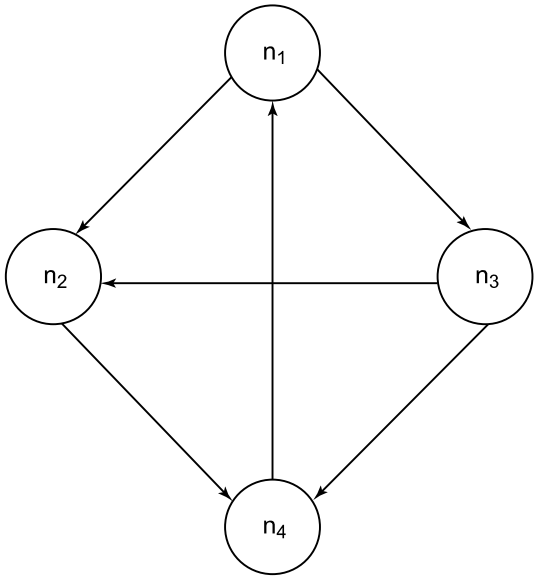


Figure 3.7. Strongly connected graph

The graph shown in Figure 3.7 has the following pair of nodes:
$<n_1, n_2>$, $<n_1, n_3>$, $<n_4, n_1>$, $<n_3, n_2>$, $<n_2, n_4>$, $<n_3, n_4>$. We identify paths for every pair of nodes.

(a) Pair $<n_1, n_2>$: Path = n_1, n_2
Pair $<n_2, n_1>$: Path = n_2, n_4, n_1

(b) Pair $<n_1, n_3>$: Path = n_1, n_3
Pair $<n_3, n_1>$: Path = n_3, n_4, n_1
(c) Pair $<n_1, n_4>$: Path = n_1, n_2, n_4
Pair $<n_4, n_1>$: Path = n_4, n_1
(d) Pair $<n_2, n_3>$: Path = n_2, n_4, n_1, n_3
Pair $<n_3, n_2>$: Path = n_3, n_2
(e) Pair $<n_2, n_4>$: Path = n_2, n_4
Pair $<n_4, n_2>$: Path = n_4, n_1, n_2
(f) Pair $<n_3, n_4>$: Path = n_3, n_4
Pair $<n_4, n_3>$: Path = n_4, n_1, n_3
Hence this graph is strongly connected.

A directed graph is said to be weakly connected, if there is a path between every two nodes when the directions of the edges are not considered. A strongly connected directed graph will also be weakly connected when we do not consider the directions of edges. Consider the graph shown in Figure 3.8 which is weakly connected.

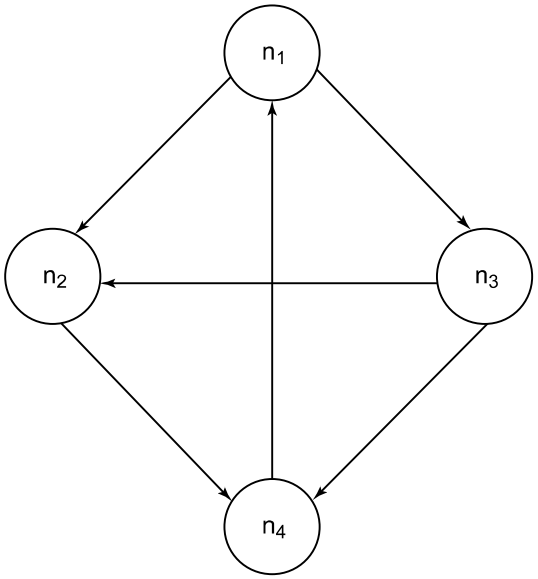
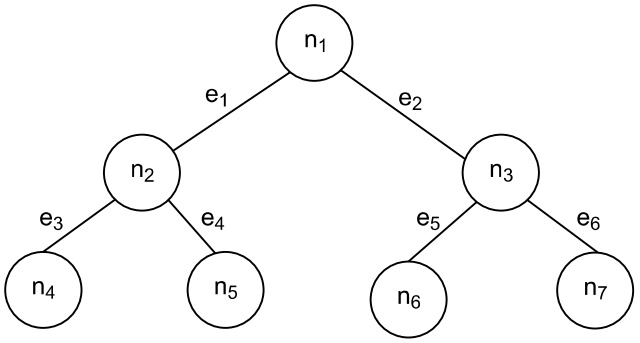


Figure 3.8. Weakly connected graph

When we do not consider the directions, it becomes an undirected graph where there is a path between every two nodes of the graph. Hence, this graph is weakly connected but not strongly connected.

Example 3.1: Consider the following undirected graph and find:

(a) The degree of all nodes
(b) The incidence matrix
(c) Adjacency matrix
(d) Paths



Solution:

This graph is an undirected graph with seven nodes and six edges

(a) The degrees of nodes are given as:

$\deg(n_1) = 2$ $\deg(n_5) = 1$

$\deg(n_2) = 3$ $\deg(n_6) = 1$

$\deg(n_3) = 3$ $\deg(n_7) = 1$

$\deg(n_4) = 1$

(b) Incidence matrix of this graph has 7×6 size which is given as:

$$\begin{array}{c} \\ n_1 \\ n_2 \\ n_3 \\ n_4 \\ n_5 \\ n_6 \\ n_7 \end{array} \begin{array}{c} \begin{array}{cccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

(c) Adjacency matrix with size 7×7 is given as:

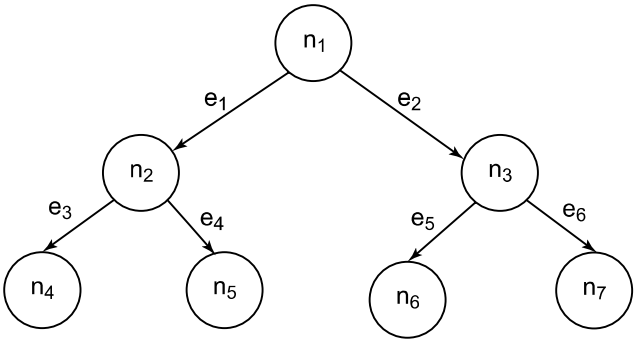
$$\begin{array}{c} \\ n_1 \\ n_2 \\ n_3 \\ n_4 \\ n_5 \\ n_6 \\ n_7 \end{array} \begin{array}{c} \begin{array}{ccccccc} n_1 & n_2 & n_3 & n_4 & n_5 & n_6 & n_7 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

(d) Paths of the graph are given as:

S. No.	Paths from initial node to final node	Sequence of nodes	Sequence of edges
1.	n_1 to n_2	n_1, n_2	e_1
2.	n_1 to n_4	n_1, n_2, n_4	e_1, e_3
3.	n_1 to n_5	n_1, n_2, n_5	e_1, e_4
4.	n_1 to n_3	n_1, n_3	e_2
5.	n_1 to n_6	n_1, n_3, n_6	e_2, e_5
6.	n_1 to n_7	n_1, n_3, n_7	e_2, e_6
7.	n_2 to n_1	n_2, n_1	e_1
8.	n_2 to n_4	n_2, n_4	e_3
9.	n_2 to n_5	n_2, n_5	e_4
10.	n_3 to n_1	n_3, n_1	e_2
11.	n_3 to n_6	n_3, n_6	e_5
12.	n_3 to n_7	n_3, n_7	e_6
13.	n_4 to n_2	n_4, n_2	e_3
14.	n_4 to n_1	n_4, n_2, n_1	e_3, e_1
15.	n_5 to n_2	n_5, n_2	e_4
16.	n_5 to n_1	n_5, n_2, n_1	e_4, e_1
17.	n_6 to n_3	n_6, n_2	e_5
18.	n_6 to n_1	n_6, n_3, n_1	e_5, e_2
19.	n_7 to n_3	n_7, n_3	e_6
20.	n_7 to n_1	n_7, n_3, n_1	e_6, e_2
21.	n_2 to n_3	n_2, n_1, n_3	e_1, e_2
22.	n_2 to n_6	n_2, n_1, n_3, n_6	e_1, e_2, e_5
23.	n_2 to n_7	n_2, n_1, n_3, n_7	e_1, e_2, e_6
24.	n_3 to n_2	n_3, n_1, n_2	e_2, e_1
25.	n_3 to n_4	n_3, n_1, n_2, n_4	e_2, e_1, e_3
26.	n_3 to n_5	n_3, n_1, n_2, n_5	e_1, e_2, e_4
27.	n_4 to n_3	n_4, n_2, n_1, n_3	e_3, e_1, e_2
28.	n_4 to n_5	n_4, n_2, n_5	e_3, e_4
29.	n_4 to n_6	n_4, n_2, n_1, n_3, n_6	e_3, e_1, e_2, e_5
30.	n_4 to n_7	n_4, n_2, n_1, n_3, n_7	e_3, e_1, e_2, e_6
31.	n_5 to n_3	n_5, n_2, n_1, n_3	e_4, e_1, e_2
32.	n_5 to n_4	n_5, n_2, n_4	e_4, e_3
33.	n_5 to n_6	n_5, n_2, n_1, n_3, n_6	e_4, e_1, e_2, e_5
34.	n_5 to n_7	n_5, n_2, n_1, n_3, n_7	e_4, e_1, e_2, e_6
35.	n_6 to n_2	n_6, n_3, n_1, n_2	e_5, e_2, e_1
36.	n_6 to n_4	n_6, n_3, n_1, n_2, n_4	e_5, e_2, e_1, e_3
37.	n_6 to n_5	n_6, n_3, n_1, n_2, n_5	e_5, e_2, e_1, e_4
38.	n_6 to n_7	n_6, n_3, n_7	e_5, e_6
39.	n_7 to n_2	n_7, n_3, n_1, n_2	e_6, e_2, e_1
40.	n_7 to n_4	n_7, n_3, n_1, n_2, n_4	e_6, e_2, e_1, e_3
41.	n_7 to n_5	n_7, n_3, n_1, n_2, n_5	e_6, e_2, e_1, e_4
42.	n_7 to n_6	n_7, n_3, n_6	e_6, e_5

Example 3.2: Consider the following directed graph and find:

(a) The degree of all nodes
(b) The incidence matrix
(c) Adjacency matrix
(d) Paths
(e) Connectedness



Solution:

This graph is a directed graph with seven nodes and six edges

(a) The degrees of all nodes are given as:

$indeg(n_1) = 0$	$outdeg(n_1) = 2$	$deg(n_1) = 2$
$indeg(n_2) = 1$	$outdeg(n_2) = 2$	$deg(n_2) = 3$
$indeg(n_3) = 1$	$outdeg(n_3) = 2$	$deg(n_3) = 3$
$indeg(n_4) = 1$	$outdeg(n_4) = 0$	$deg(n_4) = 1$
$indeg(n_5) = 1$	$outdeg(n_5) = 0$	$deg(n_5) = 1$
$indeg(n_6) = 1$	$outdeg(n_6) = 0$	$deg(n_6) = 1$
$indeg(n_7) = 1$	$outdeg(n_7) = 0$	$deg(n_7) = 1$

(b) Incidence matrix of this graph has 7 × 6 is given as:

$$\begin{array}{c} \\ n_1 \\ n_2 \\ n_3 \\ n_4 \\ n_5 \\ n_6 \\ n_7 \end{array} \begin{array}{c} \begin{array}{cccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

(c) Adjacency matrix with size 7 × 7 is given as:

$$
\begin{array}{c} \\ n_1 \\ n_2 \\ n_3 \\ n_4 \\ n_5 \\ n_6 \\ n_7 \end{array}
\begin{array}{c}
\begin{array}{ccccccc} n_1 & n_2 & n_3 & n_4 & n_5 & n_6 & n_7 \end{array} \\
\begin{bmatrix}
0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

(d) Paths of the graph are given as:

S. No.	Paths from initial node to final node	Sequence of nodes	Sequence of edges
1.	n_1 to n_2	n_1, n_2	e_1
2.	n_1 to n_4	n_1, n_2, n_4	e_1, e_3
3.	n_1 to n_5	n_1, n_2, n_5	e_1, e_4
4.	n_1 to n_3	n_1, n_3	e_2
5.	n_1 to n_6	n_1, n_3, n_6	e_2, e_5
6.	n_1 to n_7	n_1, n_3, n_7	e_2, e_6
7.	n_2 to n_4	n_2, n_4	e_3
.8.	n_2 to n_5	n_2, n_5	e_4
9.	n_3 to n_6	n_3, n_6	e_5
10.	n_3 to n_7	n_3, n_7	e_6

(e) This graph is weakly connected because there is no path between several nodes; for example, from node n_4 to n_2, n_6 to n_3, n_3 to n_1, n_2 to n_1, etc. however, if we do not consider directions, there is a path from every node to every other node which is the definition of a weakly connected graph.

3.4 GENERATION OF A GRAPH FROM PROGRAM

Graphs are extensively used in testing of computer programs. We may represent the flow of data and flow of control of any program in terms of directed graphs. A graph representing the flow of control of a program is called a program graph. The program graph may further be transformed into the DD path graph. Both of these graphs may provide foundations to many testing techniques.

3.4.1 Program Graphs

Program graph is a graphical representation of the source code of a program. The statements of a program are represented by nodes and flow of control by edges in the program graph. The definition of a program graph is [JORG07]:

> "A program graph is a directed graph in which nodes are either statements or fragments of a statement and edges represent flow of control."

The program graph helps us to understand the internal structure of the program which may provide the basis for designing the testing techniques. The basic constructs of the program graph are given in Figure 3.9.

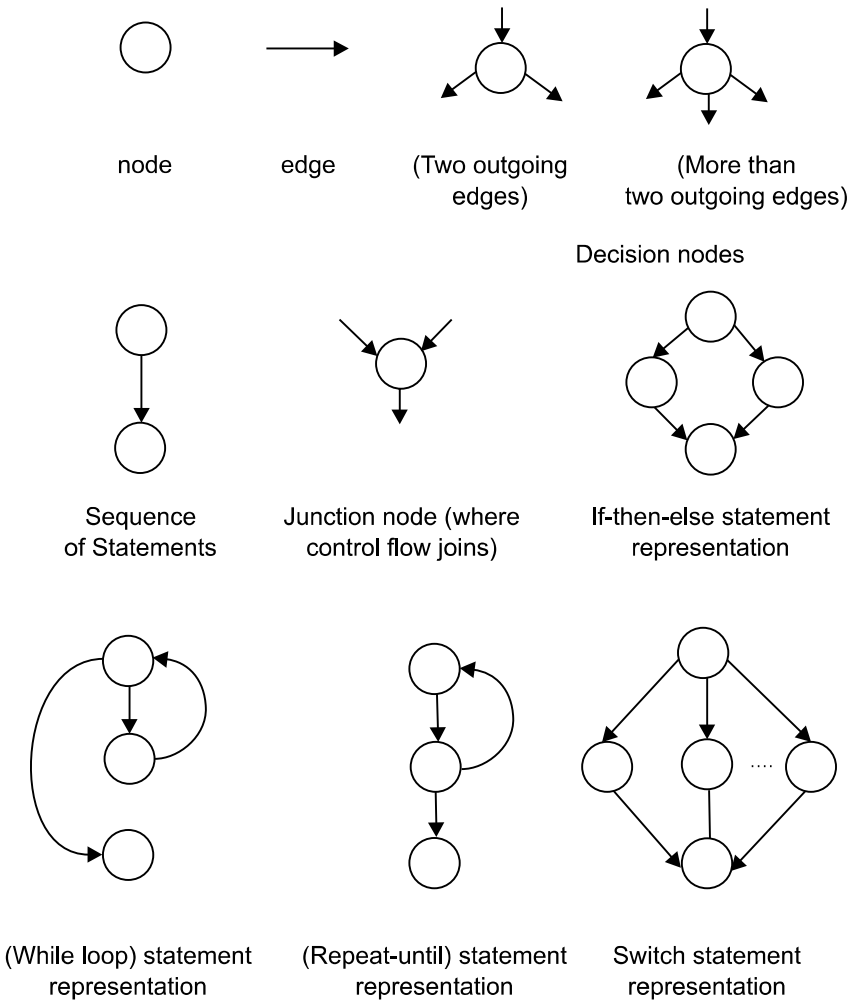


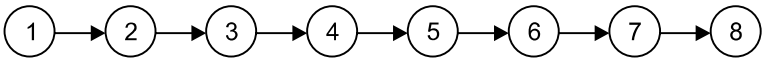
Figure 3.9. Basic constructs of a program graph

The basic constructs are used to convert a program in its program graph. We consider the program 'Square' which takes a number as an input and generates the square of the number. This program has 8 sequential statements which are represented by 8 nodes. All nodes are arranged sequentially which may lead to only one path in this program graph. Every program graph has one source node and one destination node.

We also consider a program given in Figure 3.11 that takes three numbers as input and prints the largest amongst these three numbers as output. The program graph of the program is given in Figure 3.12. There are 28 statements in the program which are represented by 28 nodes. All nodes are not in a sequence which may lead to many paths in the program graph.

```
#include<stdio.h>
1   void main()
2   {
3   int num, result;
4   printf("Enter the number:");
5   scanf("%d", &num);
6   result=num*num;
7   printf("The result is: %d", result);
8   }
```

(a) Program to find 'square' of a number



(b) Program graph for 'Square' program

Figure 3.10. Program 'Square' and its program graph

```
                #include<stdio.h>
                #include<conio.h>
1.              void main()
2.              {
3.              float A,B,C;
4.              clrscr();
5.              printf("Enter number 1:\n");
6.              scanf("%f", &A);
7.              printf("Enter number 2:\n");
8.              scanf("%f", &B);
9.              printf("Enter number 3:\n");
10.             scanf("%f", &C);
                /*Check for greatest of three numbers*/
11.             if(A>B) {
12.             if(A>C) {
13.                         printf("The largest number is: %f\n",A);
14.                         }
15.             else  {
16.                         printf("The largest number is: %f\n",C);
17.                         }
18.             }
19.             else {
20.             if(C>B) {
21.                         printf("The largest number is: %f\n",C);
22.                         }
```

(*Contd.*)

(*Contd.*)

```
23.            else  {
24.                        printf("The largest number is: %f\n",B);
25.                        }
26.            }
27.            getch();
28.            }
```

Figure 3.11. Program to find the largest among three numbers

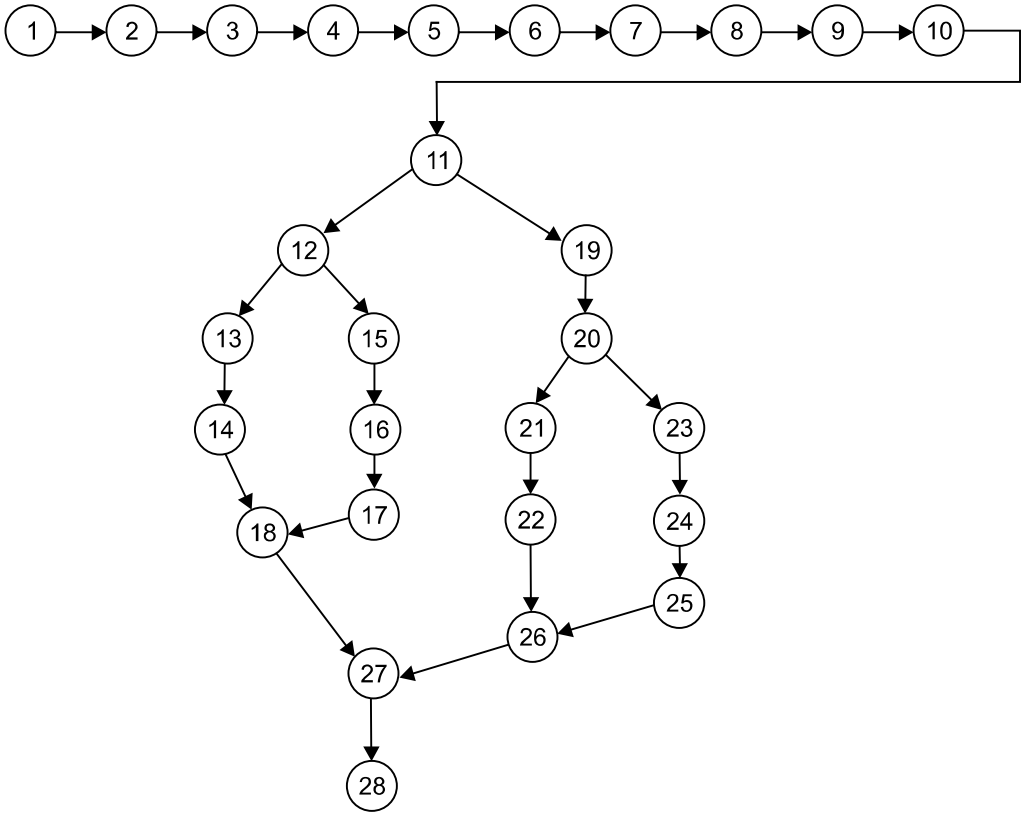


Figure 3.12. Program graph to find the largest number amongst three numbers as given in Figure 3.11.

Our example is simple, so it is easy to find all paths starting from the source node (node 1) and ending at the destination node (node 28). There are four possible paths. Every program graph may provide some interesting observations about the structure of the program. In our program graph given in Figure 3.12, nodes 2 to 10 are in sequence and nodes 11, 12, and 20 have two outgoing edges (predicate nodes) and nodes 18, 26, 27 have two incoming edges are (junction nodes).

We may also come to know whether the program is structured or not. A large program may be a structured program whereas a small program may be unstructured due to a loop in a program. If we have a loop in a program, large number of paths may be generated as shown in figure 1.5 of chapter 1. Myers [MYER04] has shown 10^{14} paths in a very small program graph due to a loop that iterates up to 20 times. This shows how an unstructured program may lead to difficulties in even finding every possible path in a program graph. Hence, testing a structured program is much easier as compared to any unstructured program.

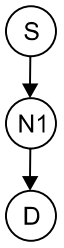
3.4.2 DD Path Graphs

The Decision to Decision (DD) path graph is an extension of a program graph. It is widely known as DD path graph. There may be many nodes in a program graph which are in a sequence. When we enter into the first node of the sequence, we can exit only from the last node of that sequence. In DD path graph, such nodes which are in a sequence are combined into a block and are represented by a single node. Hence, the DD path graph is a directed graph in which nodes are sequences of statements and edges are control flow amongst the nodes. All programs have an entry and an exit and the corresponding program graph has a source node and a destination node. Similarly, the DD path graph also has a source node and a destination node.

We prepare a mapping table for the program graph and the DD path graph nodes. A mapping table maps nodes of the program graph to the corresponding nodes of the DD path graph. This may combine sequential nodes of the program graph into a block and that is represented by a single node in the DD path graph. This process may reduce the size of the program graph and convert it into a more meaningful DD path graph. We consider program 'Square' and its program graph given in Figure 3.10. We prepare a mapping table and a DD path graph as shown in Figure 3.13. All nodes are sequential nodes except node 1 and node 8 which are source node and destination node respectively.

Program graph nodes	DD path graph corresponding nodes	Remarks
1	S	Source node
2 – 7	N1	Sequential flow
8	D	Destination node

Mapping of program graph nodes and DD path graph nodes



DD Path graph

Figure 3.13. DD path graph and mapping table of program graph in Figure 3.10

We consider a program to find the 'largest amongst three numbers' as given in Figure 3.11. The program graph is also given in Figure 3.12. There are many sequential nodes, decision nodes, junction nodes available in its program graph. Its mapping table and the DD path graph are given in Table 3.3 and Figure 3.14 respectively.

Table 3.3. Mapping of program graph nodes and DD graph nodes

Program graph nodes	DD path graph corresponding node	Remarks
1	S	Source node
2 to 10	N1	Sequential nodes, there is a sequential flow from node 2 to 10
11	N2	Decision node, if true goto 12, else goto 19
12	N3	Decision node, if true goto 13 else goto 15
13, 14	N4	Sequential nodes
15, 16, 17	N5	Sequential nodes

(*Contd.*)

(*Contd.*)

Program graph nodes	DD path graph corresponding node	Remarks
18	N6	Junction node, two edges 14 and 17 are terminated here
19	N7	Intermediate node terminated at node 20
20	N8	Decision node, if true goto 21 else goto 23
21, 22	N9	Sequential nodes
23, 24, 25	N10	Sequential nodes
26	N11	Junction node, two edges 22 and 25 are terminated here
27	N12	Junction node, two edges 18 and 26 are terminated here
28	D	Destination node

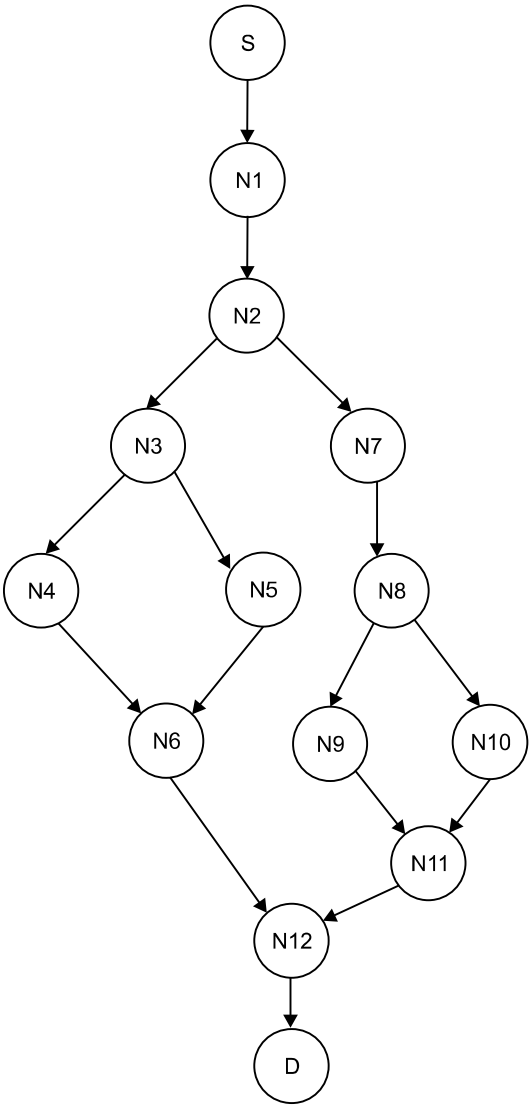


Figure 3.14. DD path graph of the program to find the largest among three numbers.

The DD path graphs are used to find paths of a program. We may like to test every identified path during testing which may give us some level of confidence about the correctness of the program.

Example 3.3: Consider the program for the determination of division of a student. Its input is a triple of positive integers (mark1, mark2, mark3) and values are from interval [0, 100].

The program is given in Figure 3.15. The output may be one of the following words:

[First division with distinction, First division, Second division, Third division, Fail, Invalid marks]. Draw the program graph and the DD path graph.

Solution:
The program graph is given in Figure 3.16. The mapping table of the DD path graph is given in Table 3.4 and DD path graph is given in Figure 3.17.

```
        /*Program to output division of a student based on the marks in three subjects*/
        #include<stdio.h>
        #include<conio.h>
1.      void main()
2.      {
3.      int mark1, mark2,mark3,avg;
4.      clrscr();
5.      printf("Enter marks of 3 subjects (between 0-100)\n");
6.      printf("Enter marks of first subject:");
7.      scanf("%d", &mark1);
8.      printf("Enter marks of second subject:");
9.      scanf("%d", &mark2);
10.     printf("Enter marks of third subject:");
11.     scanf("%d",&mark3);
12.     if(mark1>100||mark1<0||mark2>100||mark2<0||mark3>100||mark3<0) {
13.             printf("Invalid Marks! Please try again");
14.             }
15.     else {
16.     avg=(mark1+mark2+mark3)/3;
17.     if(avg<40) {
18.             printf("Fail");
19.             }
20.     else if(avg>=40&&avg<50) {
21.             printf("Third Division");
22.             }
23.     else if(avg>=50&&avg<60) {
24.             printf("Second Division");
25.             }
```

(*Contd.*)

(*Contd.*)

```
26.        else if(avg>=60&&avg<75) {
27.              printf("First Division");
28.                   }
29.        else       {
30.                   printf("First Division with Distinction");
31.                   }
32.        }
33.        getch();
34.        }
```

Figure 3.15. Source code of determination of division of a student problem

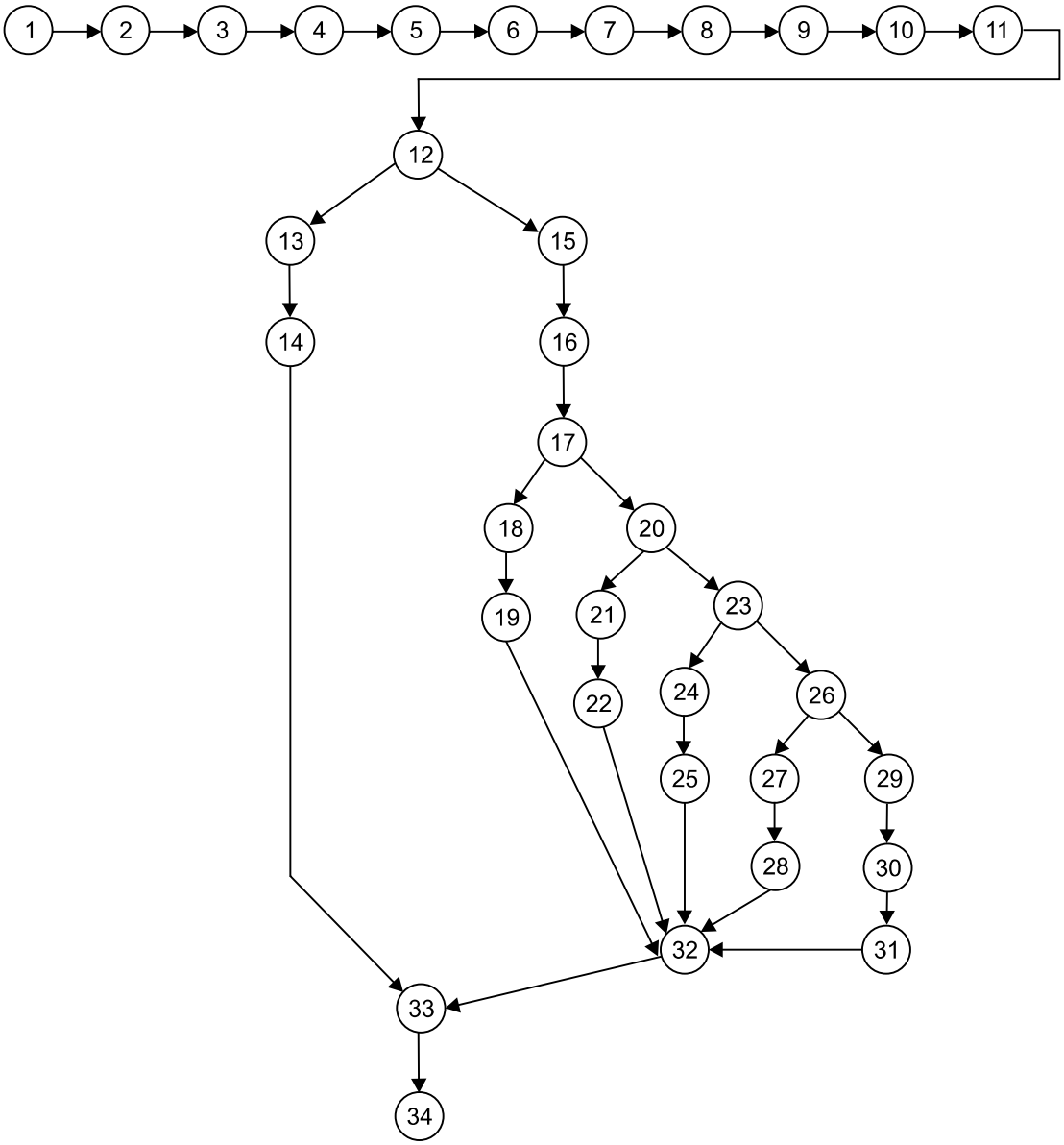


Figure 3.16. Program graph

Table 3.4. Mapping of program graph nodes and DD graph nodes

Program graph nodes	DD path graph corresponding node	Remarks
1	S	Source node
2 to 11	N1	Sequential nodes, there is a sequential flow from node 2 to 11
12	N2	Decision node, if true goto 13 else goto 15
13, 14	N3	Sequential nodes
15, 16	N4	Sequential nodes
17	N5	Decision node, if true goto 18 else goto 20
18, 19	N6	Sequential nodes
20	N7	Decision node, if true goto 21 else goto 23
21, 22	N8	Sequential nodes
23	N9	Decision node, if true goto 24 else goto 26
24, 25	N10	Sequential nodes
26	N11	Decision node, if true goto 27 else goto 29
27, 28	N12	Sequential nodes
29, 31	N13	Sequential nodes
32	N14	Junction node, five edges 19, 22, 25, 28 and 31 are terminated here
33	N15	Junction node, two edges 14 and 32 are terminated here
34	D	Destination node

Example 3.4: Consider the program for classification of a triangle. Its input is a triple of positive integers (say a, b and c) and values from the interval [1, 100]. The output may be [Right angled triangle, Acute angled triangle, Obtuse angled triangle, Invalid triangle, Input values are out of Range]. The program is given in Figure 3.18. Draw the program graph and the DD path graph.

Solution:
The program graph is shown in Figure 3.19. The mapping table is given in Table 3.5 and the DD path graph is given in Figure 3.20.

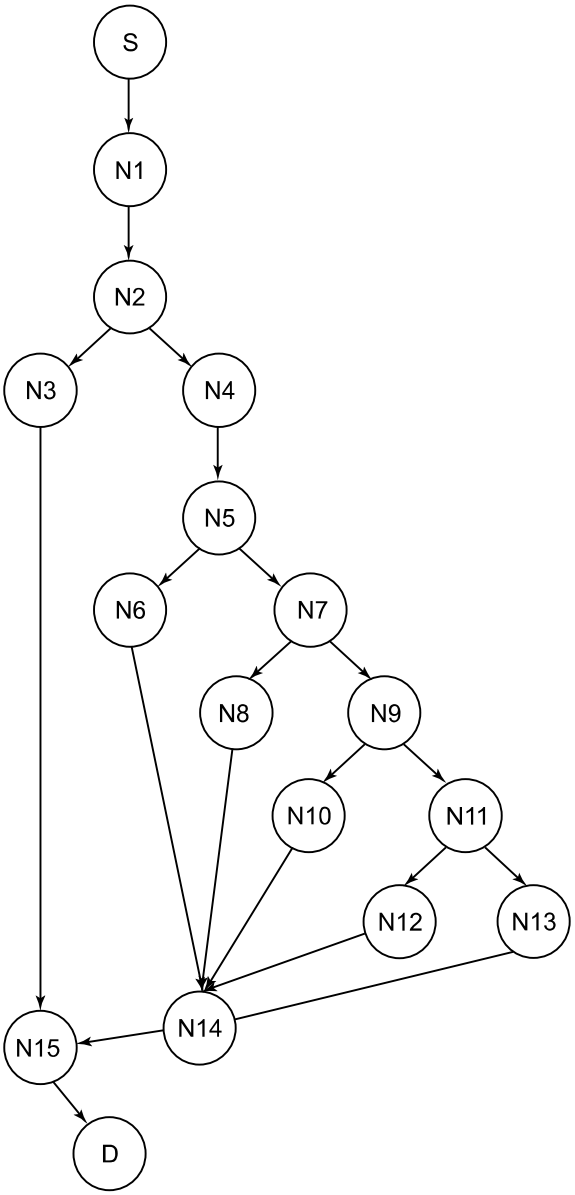


Figure 3.17. DD path graph of program to find division of a student

```
/*Program to classify whether a triangle is acute, obtuse or right angled given the sides of
the triangle*/
//Header Files
#include<stdio.h>
#include<conio.h>
#include<math.h>
1.    void main() //Main Begins
2.    {
```

(*Contd.*)

(*Contd.*)

```
3.      double a,b,c;
4.      double a1,a2,a3;
5.      int valid=0;
6.      clrscr();
7.      printf("Enter first side of the triangle:"); /*Enter the sides of Triangle*/
8.      scanf("%lf",&a);
9.      printf("Enter second side of the triangle:");
10.     scanf("%lf",&b);
11.     printf("Enter third side of the triangle:");
12.     scanf("%lf",&c);
        /*Checks whether a triangle is valid or not*/
13.     if(a>0&&a<=100&&b>0&&b<=100&&c>0&&c<=100) {
14.     if((a+b)>c&&(b+c)>a&&(c+a)>b) {
15.             valid=1;
16.             }
17.     else  {
18.             valid=-1;
19.             }
20.     }
21.     if(valid==1) {
22.     a1=(a*a+b*b)/(c*c);
23.     a2=(b*b+c*c)/(a*a);
24.     a3=(c*c+a*a)/(b*b);
25.     if(a1<1||a2<1||a3<1) {
26.          printf("Obtuse angled triangle");
27.         }
28.     else if(a1==1||a2==1||a3==1) {
29.         printf("Right angled triangle");
30.         }
31.     else {
32.         printf("Acute angled triangle");
33.        }
34.     }
35.     else if(valid==-1) {
36.             printf("\nInvalid Triangle");
37.             }
38.     else  {
39.             printf("\nInput Values are Out of Range");
```

(*Contd.*)

(*Contd.*)

```
40.            }
41.        getch();
42.        } //Main Ends
```

Figure: 3.18. Source code for classification of triangle problem

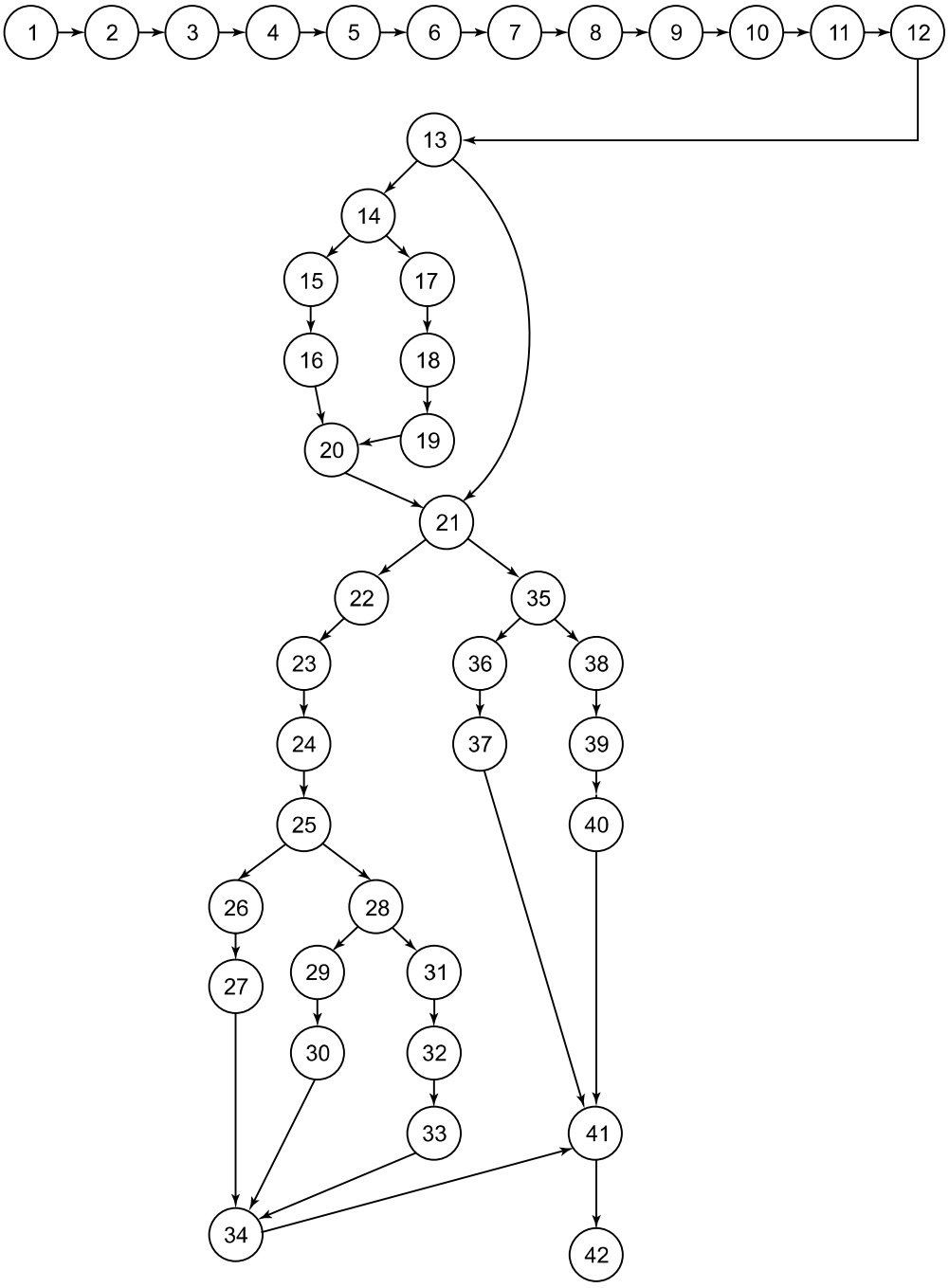


Figure 3.19. Program graph of classification of triangle problem

Table 3.5. Mapping of program graph nodes and DD graph nodes

Program graph nodes	DD path graph corresponding node	Remarks
1	S	Source node
2 to 12	N1	Sequential nodes, there is a sequential flow from node 2 to 12
13	N2	Decision node, if true goto 14 else goto 21
14	N3	Decision node, if true goto 15 else goto 17
15, 16	N4	Sequential nodes
17, 18, 19	N5	Sequential nodes
20	N6	Junction node, two edges 16 and 19 are terminated here
21	N7	Junction node, two edges 13 and 20 are terminated here. Also a decision node, if true goto 22, else goto 35
22, 23, 24	N8	Sequential nodes
25	N9	Decision node, if true goto 26 else goto 28
26, 27	N10	Sequential nodes
28	N11	Decision node, if true goto 29 else goto 31
29, 30	N12	Sequential nodes
31, 32, 33	N13	Sequential nodes
34	N14	Junction node, three edges 27, 30 and 33 are terminated here
35	N15	Decision node, if true goto 36 else goto 38
36, 37	N16	Sequential nodes
38, 39, 40	N17	Sequential nodes
41	N18	Three edges 34, 37 and 40 are terminated here.
42	D	Destination node.

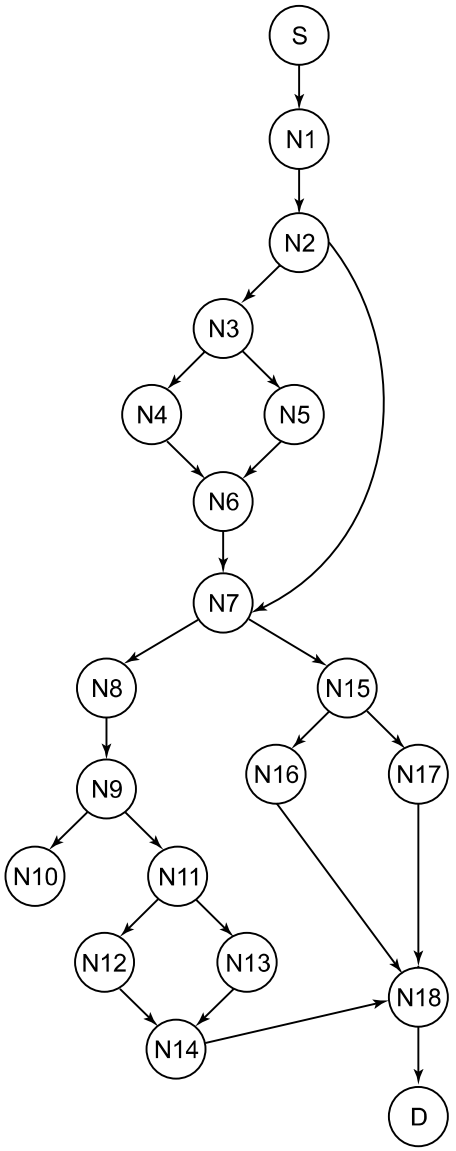


Figure 3.20. DD path graph of the program to classify a triangle

Example 3.5: Consider the program for determining the day of the week. Its input is a triple of day, month and year with the values in the range

$$1 \leq \text{month} \leq 12$$

$$1 \leq \text{day} \leq 31$$

$$1900 \leq \text{year} \leq 2058$$

The possible values of the output may be [Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Invalid date]. The program is given in Figure 3.21.

Draw the program graph and the DD path graph.

```
        /*Program to compute day of the week*/
        /*Header Files*/
        #include<stdio.h>
        #include<conio.h>
1.      void main()
2.      {
3.      int day,month,year,century,Y,y1,M,date,validDate=0,leap=0;
4.      clrscr();
5.      printf("Enter day:");
6.      scanf("%d",&day);
7.      printf("Enter month:");
8.      scanf("%d",&month);
9.      printf("Enter year (between 1900 and 2058):");
10.     scanf("%d",&year);
        /*Check whether the date is valid or not*/
11.     if(year>=1900&&year<=2058) {
12.     if(year%4==0) { /*Check for leap year*/
13.             leap=1;
14.             if((year%100)==0&&(year%400)!=0) {
15.                     leap=0;
16.             }
17.     }
18.     if(month==4||month==6||month==9||month==11){
19.     if(day>=1&&day<=30) {
20.             validDate=1;
21.             }
22.     else  {
23.             validDate=0;
24.             }
25.     }
26.     else if(month==2){
27.     if(leap==1&&(day>=1&&day<=29)) {
28.             validDate=1;
29.             }
30.     else if(day>=1&&day<=28) {
31.             validDate=1;
32.             }
33.     else  {
34.             validDate=0;
35.             }
36.     }
37.     else if((month>=1&&month<=12)&&(day>=1&&day<=31)){
```

(*Contd.*)

(*Contd.*)

```
38.                 validDate=1;
39.                 }
40.         else  {
41.                 validDate=0;
42.                 }
43.         }
44.         if(validDate) { /*Calculation of Day in the week*/
45.         if(year>=1900&&year<2000){
46.         century=0;
47.         y1=year-1900;
48.         }
49.         else {
50.         century=6;
51.         y1=year-2000;
52.         }
53.         Y=y1+(y1/4);
54.         if(month==1) {
55.         if(leap==0) {
56.                 M=0; /*for non-leap year*/
57.                 }
58.         else  {
59.                 M=6; /*for leap year*/
60.                 }
61.         }
62.         else if(month==2){
63.         if(leap==0) {
64.                 M=3; /*for non-leap year*/
65.                 }
66.         else  {
67.                 M=2; //for leap year
68.                 }
69.         }
70.         else if((month==3)||(month==11)) {
71.                 M=3;
72.                 }
73.         else if((month==4)||(month==7)) {
74.                 M=6;
75.                 }
76.         else if(month==5) {
77.                 M=1;
78.                 }
79.         else if(month==6) {
80.                 M=4;
```

(*Contd.*)

(*Contd.*)

```
81.                 }
82.         else if(month==8) {
83.                 M=2;
84.                 }
85.         else if((month==9)||(month==12)) {
86.                 M=5;
87.                 }
88.         else  {
89.                 M=0;
90.                 }
91.         date=(century+Y+M+day)%7;
92.         if(date==0) { /*Determine the day of the week*/
93.                 printf("Day of the week for [%d:%d:%d] is Sunday",day,month,year);
94.                 }
95.         else if(date==1) {
96.                 printf("Day of the week for [%d:%d:%d] is Monday",day,month,year);
97.                 }
98.         else if(date==2) {
99.                 printf("Day of the week for [%d:%d:%d] is Tuesday",day,month,year);
100.                }
101.        else if(date==3) {
102.                printf("Day of the week for [%d:%d:%d] is Wednesday",day,month,year);
103.                }
104.        else if(date==4) {
105.                printf("Day of the week for [%d:%d:%d] is Thursday",day,month,year);
106.                }
107.        else if(date==5) {
108.                printf("Day of the week for [%d:%d:%d] is Friday",day,month,year);
109.                }
110.        else  {
111.                printf("Day of the week for [%d:%d:%d] is Saturday",day,month,year);
112.                }
113.        }
114.        else  {
115.                printf("The date entered [%d:%d:%d] is invalid",day,month,year);
116.                }
117.        getch();
118.        }
```

Figure 3.21. Source code for determination of day of the week

Solution:

The program graph is shown in Figure 3.22. The mapping table is given in Table 3.6 and the DD path graph is given in Figure 3.23.

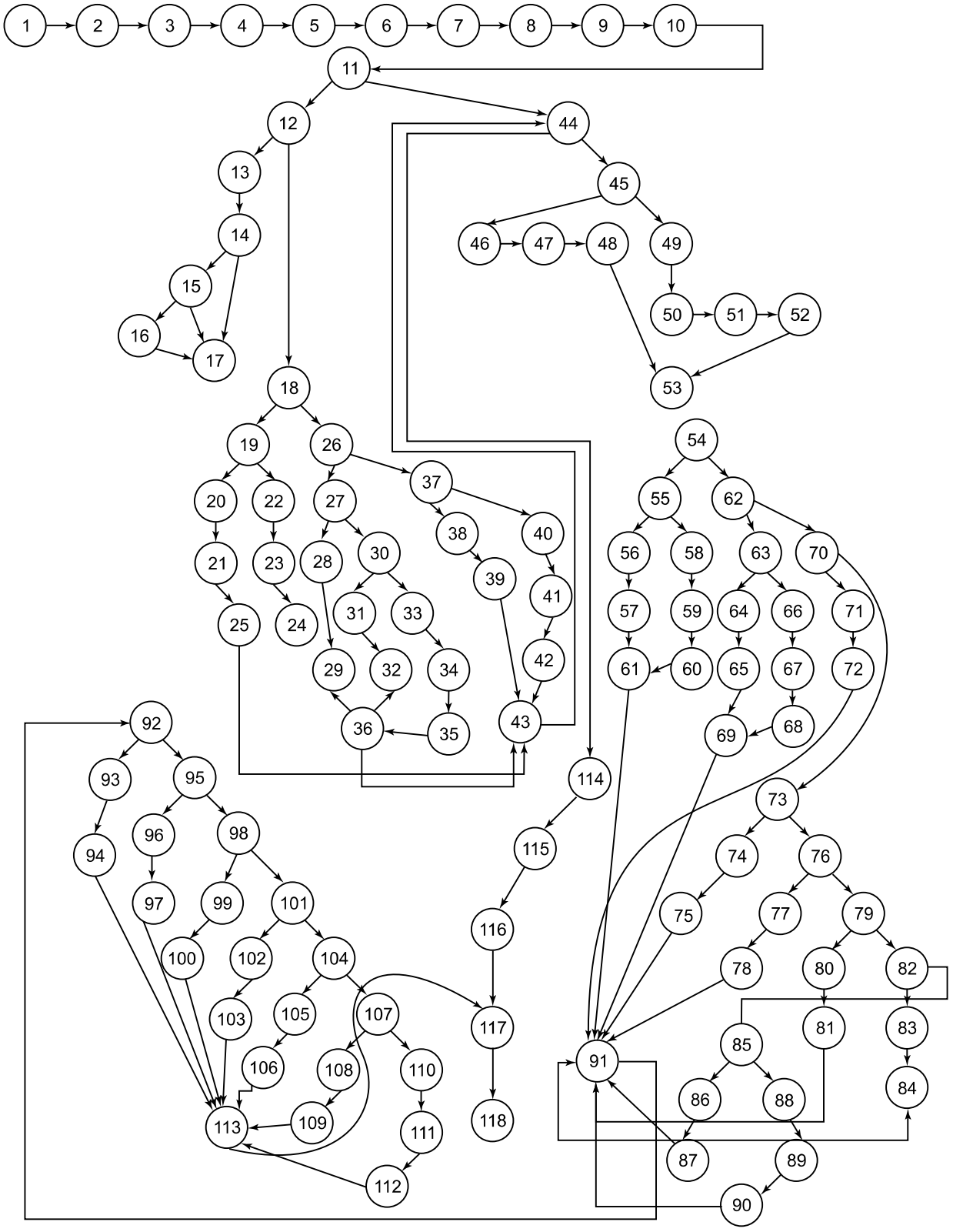


Figure 3.22. Program graph for determination of day of the week

Table 3.6. Mapping of program graph nodes to DD graph nodes

Program graph nodes	DD path graph corresponding node	Remarks
1	S	Source node
2 to 10	N1	Sequential nodes, there is a sequential flow from node 2 to 10
11	N2	Decision node, if true goto 12, else goto 44
12	N3	Decision node, if true goto 13, else goto 18
13	N4	Intermediate node terminated at node 14
14	N5	Decision node, if true goto 15, else goto 17
15, 16	N6	Sequential nodes
17	N7	Junction node, two edges 14 and 16 are terminated here
18	N8	Junction node, two edges 12 and 17 are terminated here. Also a decision node, if true goto 19, else goto 26
19	N9	Decision node, if true goto 20, else goto 22
20, 21	N10	Sequential nodes
22, 23, 24	N11	Sequential nodes
25	N12	Junction node, two edges 21 and 24 are terminated here
26	N13	Decision node, if true goto 27, else goto 37
27	N14	Decision node, if true goto 28, else goto 30
28, 29	N15	Sequential nodes
30	N16	Decision node, if true goto 31, else goto 33
31, 32	N17	Sequential nodes
33, 34, 35	N18	Sequential nodes
36	N19	Junction node, three edges 29, 32, and 35 are terminated here
37	N20	Decision node, if true goto 38, else goto 40
38, 39	N21	Sequential nodes
40, 41, 42	N22	Sequential nodes
43	N23	Four edges 25, 36, 39, and 42 are terminated here
44	N24	Junction node, two edges 11 and 43 are terminated here and also a decision node. If true goto 45, else goto 114
45	N25	Decision node, if true goto 46, else goto 49
46, 47, 48	N26	Sequential nodes
49, 50, 51, 52	N27	Sequential nodes
53	N28	Junction node, two edges 48 and 52 are terminated here
54	N29	Decision node, if true goto 55, else goto 62
55	N30	Decision node, if true goto 56, else goto 58
56, 57	N31	Sequential nodes
58, 59, 60	N32	Sequential nodes

(*Contd.*)

(*Contd.*)

Program graph nodes	DD path graph corresponding node	Remarks
61	N33	Junction node, two edges 57 and 60 are terminated here
62	N34	Decision node, if true goto 63, else goto 70
63	N35	Decision node, if true goto 64, else goto 66
64, 65	N36	Sequential nodes
66, 67, 68	N37	Sequential nodes
69	N38	Junction node, two edges 65 and 68 are terminated here
70	N39	Decision node, if true goto 71 else goto 73
71, 72	N40	Sequential nodes
73	N41	Decision node, if true goto 74 else goto 76
74, 75	N42	Sequential nodes
76	N43	Decision node, if true goto 77 else goto 79
77, 78	N44	Sequential nodes
79	N45	Decision node, if true goto 80 else goto 82
80, 81	N46	Sequential nodes
82	N47	Decision node, if true goto 83, else goto 85
83, 84	N48	Sequential nodes
85	N49	Decision node, if true goto 86, else goto 88
86, 87	N50	Sequential nodes
88, 89, 90	N51	Sequential nodes
91	N52	Junction node, nine edges 61, 69, 72, 75, 78, 81, 84, 87 and 90 are terminated here
92	N53	Decision node, if true goto 93, else goto 95
93, 94	N54	Sequential nodes
95	N55	Decision node, if true goto 96, else goto 98
96, 97	N56	Sequential nodes
98	N57	Decision node, if true goto 99, else goto 101
99, 100	N58	Sequential nodes
101	N59	Decision node, if true goto 102, else goto 104
102, 103	N60	Sequential nodes
104	N61	Decision node, if true goto 105, else goto 107
105, 106	N62	Sequential nodes
107	N63	Decision node, if true goto 108, else goto 110
108, 109	N64	Sequential nodes
110, 111, 112	N65	Sequential nodes
113	N66	Seven edges 94, 97, 100, 103, 106, 109 and 112 are terminated here
114, 115, 116	N67	Sequential nodes
117	N68	Junction node, two edges 116 and 113 are terminated here
118	D	Destination node

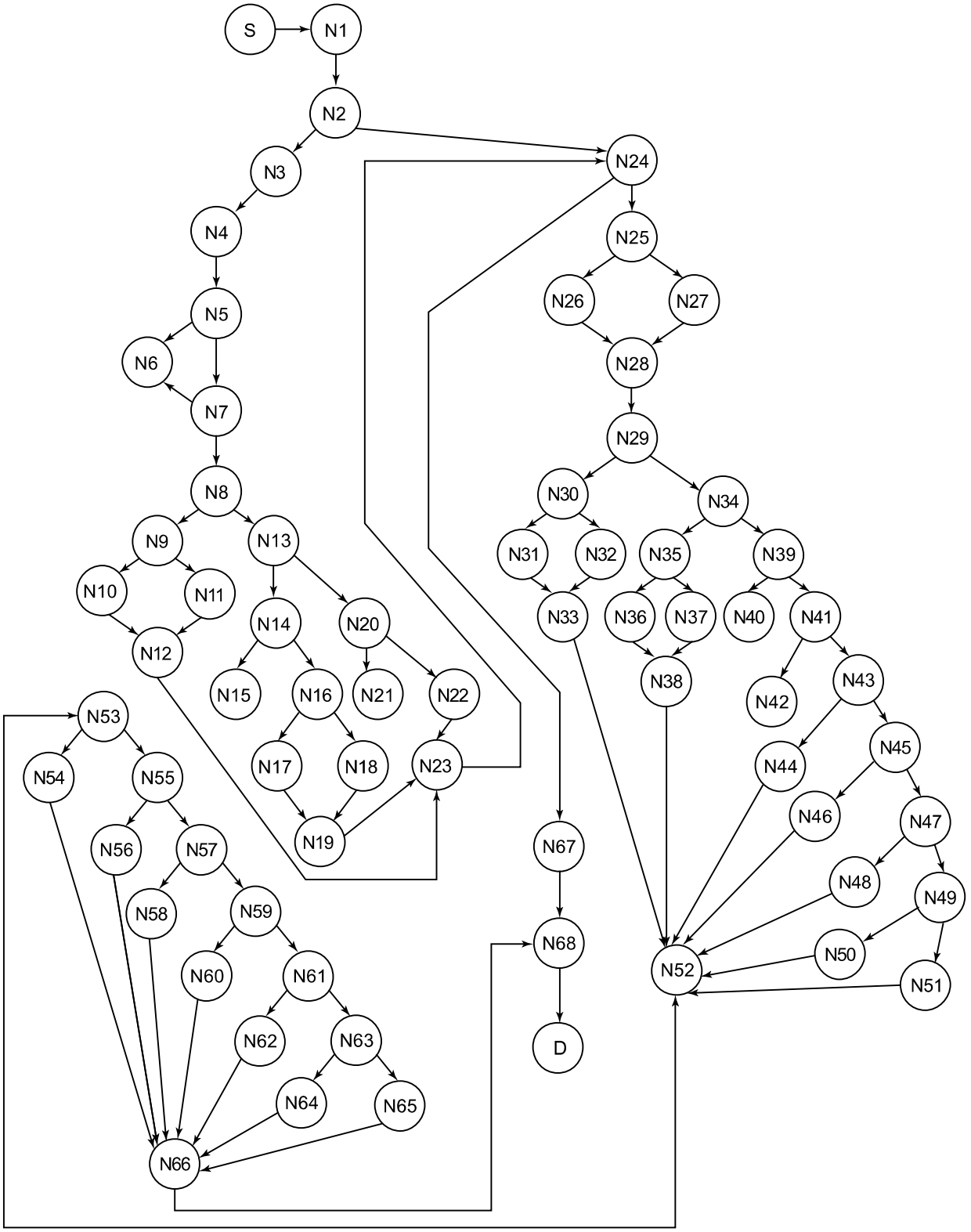


Figure 3.23. DD path graph for determination of day of the week

3.5 IDENTIFICATION OF INDEPENDENT PATHS

There are many paths in any program. If there are loops in a program, the number of paths increases drastically. In such situations, we may be traversing the same nodes and edges again and again. However, as defined earlier, an independent path should have at least one new node or edge which is to be traversed. We should identify every independent path of a program and pay special attention to these paths during testing. A few concepts of graph theory are used in testing techniques which may help us to identify independent paths.

3.5.1 Cyclomatic Complexity

This concept involves using cyclomatic number of graph theory which has been redefined as cyclomatic complexity. This is nothing but the number of independent paths through a program. McCabe [MCCA76] introduced this concept and gave three methods to calculate cyclomatic complexity.

(i) $V(G) = e - n + 2P$

where $V(G)$ = Cyclomatic complexity

G : program graph

n : number of nodes

e : number of edges

P : number of connected components

The program graph (G) is a directed graph with single entry node and single exit node. A connected graph is a program graph where all nodes are reachable from entry node, and exit node is also reachable from all nodes. Such a program graph will have connected component (P) value equal to one. If there are parts of the program graph, the value will be the number of parts of the program graph where one part may represent the main program and other parts may represent sub-programs.

(ii) Cyclomatic complexity is equal to the number of regions of the program graph.

(iii) Cyclomatic complexity

$$V(G) = \Pi + 1$$

where Π is the number of predicate nodes contained in the program graph (G).

The only restriction is that every predicate node should have two outgoing edges i.e. one for 'true' condition and another for 'false' condition. If there are more than two outgoing edges, the structure is required to be changed in order to have only two outgoing edges. If it is not possible, then this method ($\Pi + 1$) is not applicable.

Properties of cyclomatic complexity:

1. $V(G) \geq 1$
2. $V(G)$ is the maximum number of independent paths in program graph G.
3. Addition or deletion of functional statements to program graph G does not affect $V(G)$.
4. G has only one path if $V(G)=1$
5. $V(G)$ depends only on the decision structure of G.

We consider the program graph given in Figure 3.24.

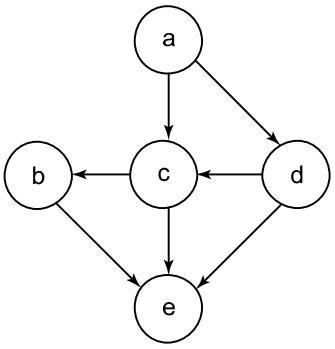
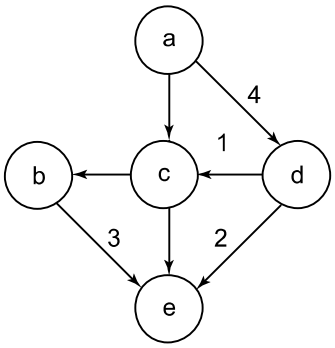


Figure 3.24. Program graph

The value of cyclomatic complexity can be calculated as:

(i) $V(G) = e - n + 2P$
$= 7 - 5 + 2$
$= 4$

(ii) V(G) = No. of regions of the graph



Hence, V(G) = 4
Three regions (1, 2 and 3) are inside and 4^{th} is the outside region of the graph

(iii) $V(G) = \Pi + 1$
$= 3 + 1 = 4$

There are three predicate nodes namely node a, node c and node d.
These four independent paths are given as:

Path 1 : ace
Path 2 : ade
Path 3 : adce
Path 4 : acbe

We consider another program graph given in Figure 3.25 with three parts of the program graph.

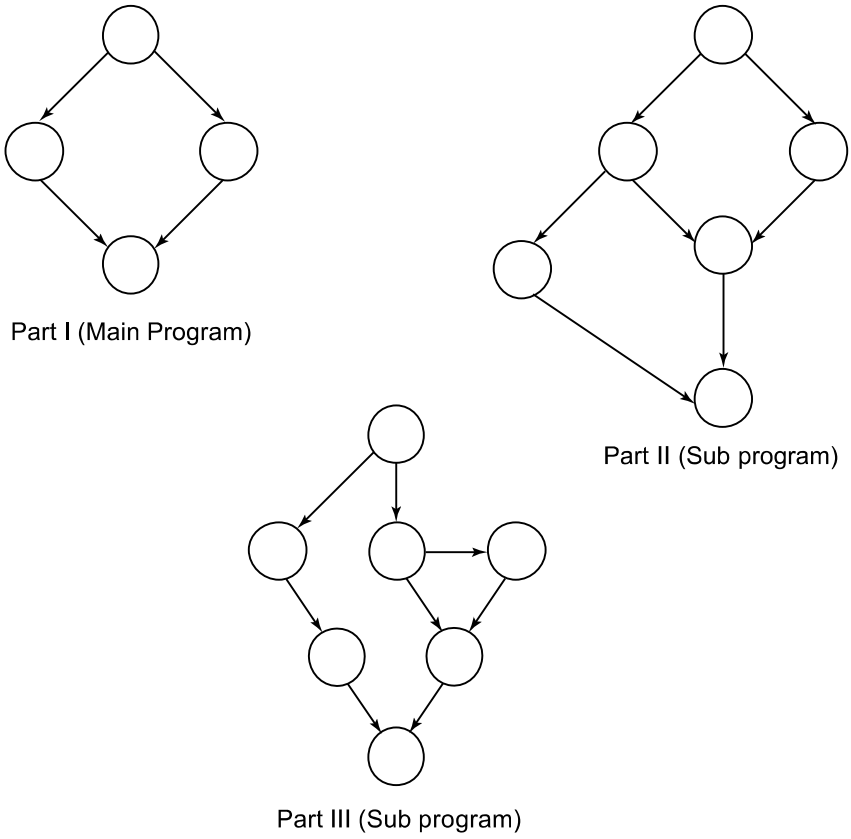


Figure 3.25. Program graph with 3 connected components

$$V(G) = e - n + 2P`$$
$$= (4+7+8) - (4+6+7) + 2x3$$
$$= 19 - 17 + 6$$
$$= 8$$

We calculate the cyclomatic complexity of each part of the graph independently.

$$V(G - \text{Part I}) = 4 - 4 + 2 = 2$$
$$V(G - \text{Part II}) = 7 - 6 + 2 = 3$$
$$V(G - \text{Part III}) = 8 - 7 + 2 = 3$$

Hence, V (G – Part I U G – Part II U G – Part III)

$$= V(G - \text{Part I}) + V(G - \text{Part II}) + V(G - \text{Part III})$$

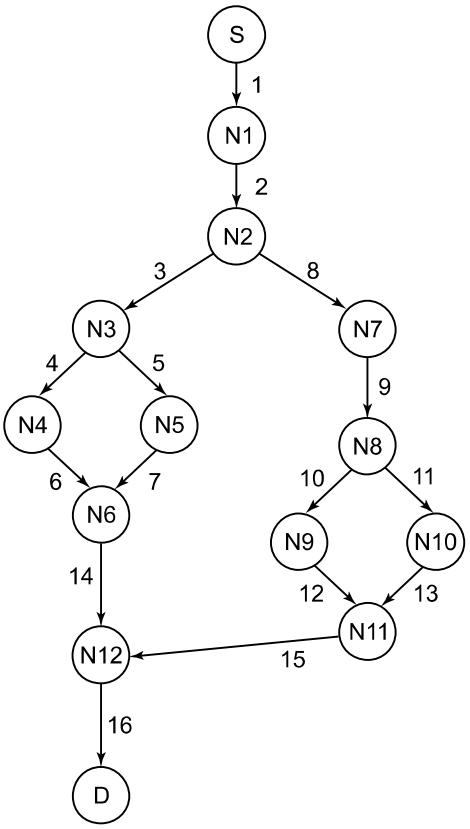
In general, the cyclomatic complexity of a program graph with P connected components is equal to the summation of their individual cyclomatic complexities. To understand this, consider graph G_i where $1 \leq i \leq P$ denote the P connected components of a graph, and e_i and n_i are the number of edges and nodes in the i^{th} connected component of the graph. Then, we may have the following equation:

$$V(G) = e - n + 2P = \sum_{i=1}^{P} e_i - \sum_{i=1}^{P} n_i + 2P$$

$$= \sum_{i=1}^{P} (e_i - n_i + 2) = \sum_{i=1}^{P} V(G_i)$$

The cyclomatic complexity is a popular measure to know the complexity of any program. It is easy to calculate and immediately provides an insight to the implementation of the program. McCabe suggested an upper limit for this cyclomatic complexity i.e. 10 [MACC76]. If this exceeds, developers have to redesign the program to reduce the cyclomatic complexity. The purpose is to keep the size of the program manageable and compel the testers to execute all independent paths. This technique is more popular at module level and forces everyone to minimize its value for the overall success of the program. There may be situations where this limit seems unreasonable; e.g. when a large number of independent cases follow a selection function like switch or case statement.

Example 3.6: Consider the following DD path graph (as given in Figure 3.14) and calculate the cyclomatic complexity. Also find independent paths.



Solution:

(i) $V(G) = e - n + 2P$

$= 16 - 14 + 2$

$= 4$

(ii) V (G) = Number of Regions
= 4

(iii) V (G) = Π + 1
= 3 (N2, N3, N8) + 1
= 4

There are 4 independent paths as given below:

(i) S, N1, N2, N3, N4, N6, N12, D
(ii) S, N1, N2, N3, N5, N6, N12, D
(iii) S, N1, N2, N7, N8, N9, N11, N12, D
(iv) S, N1, N2, N8, N10, N11, N12, D

Example 3.7: Consider the problem for determination of division of a student with the DD path graph given in Figure 3.17. Find cyclomatic complexity and also find independent paths.

Solution:
Number of edges (e) = 21
Number of nodes (n) = 17

(i) V(G) = e – n + 2P = 21 – 17 + 2 = 6
(ii) V(G) = Π + 1 = 5 + 1 = 6
(iii) V(G) = Number of regions = 6

Hence cyclomatic complexity is 6 meaning there are six independent paths in the DD path graph.
The independent paths are:

(i) S, N1, N2, N3, N15, D
(ii) S, N1, N2, N4, N5, N6, N14, N15, D
(iii) S, N1, N2, N4, N5, N7, N8, N14, N15, D
(iv) S, N1, N2, N4, N5, N7, N9, N10, N14, N15, D
(v) S, N1, N2, N4, N5, N7, N9, N11, N12, N14, N15, D
(vi) S, N1, N2, N4, N5, N7, N9, N11, N13, N14, N15, D

Example 3.8: Consider the classification of triangle problem given in Example 3.2 with its DD path graph given in Figure 3.20. Find the cyclomatic complexity and also find independent paths.

Solution:
Number of edges (e) = 25
Number of nodes (n) = 20

(i) V(G) = e – n + 2P = 25 – 20 + 2 = 7
(ii) V(G) = Π + 1 = 6 + 1 = 7
(iii) V(G) = Number of regions = 7

Hence cyclomatic complexity is 7. There are seven independent paths as given below:

(i) S, N1, N2, N7, N15, N17, N18, D
(ii) S, N1, N2, N7, N15, N16, N18, D
(iii) S, N1, N2, N7, N8, N9, N11, N13, N14, N18, D
(iv) S, N1, N2, N7, N8, N9, N11, N12, N14, N18, D
(v) S, N1, N2, N7, N8, N9, N10, N14, N18, D
(vi) S, N1, N2, N3, N5, N6, N7, N8, N9, N10, N14, N18, D
(vii) S, N2, N3, N4, N6, N7, N8, N9, N10, N14, N18, D

Example 3.9: Consider the DD path graph given in Figure 3.23 for determination of the day problem. Calculate the cyclomatic complexity and also find independent paths.

Solution:
Number of edges (e) = 96
Number of nodes (n) = 70

(i) $V(G) = e - n + 2P$

$= 96 - 70 + 2 = 28$

(ii) V(G) = Number of regions = 28
(iii) $V(G) = \Pi + 1$

$= 27 + 1 = 28$

Hence, there are 28 independent paths.
The independent paths are:

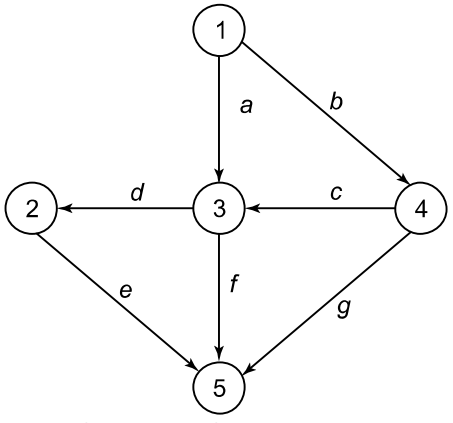
(i) S, N1, N2, N24, N25, N27, N28, N29, N34, N39, N41, N43, N45, N47, N49, N51, N52, N53, N54, N66, N68, D
(ii) S, N1, N2, N24, N25, N27, N28, N29, N34, N39, N41, N43, N45, N47, N49, N50, N52, N53, N54, N66, N68, D
(iii) S, N1, N2, N24, N25, N27, N28, N29, N34, N39, N41, N43, N45, N47, N48, N52, N53, N54, N66, N68, D
(iv) S, N1, N2, N24, N25, N27, N28, N29, N34, N39, N41, N43, N45, N46, N52, N53, N54, N66, N68, D
(v) S, N1, N2, N24, N25, N27, N28, N29, N34, N39, N41, N43, N44, N52, N53, N54, N66, N68, D
(vi) S, N1, N2, N24, N25, N27, N28, N29, N34, N39, N41, N42, N52, N53, N54, N66, N68, D
(vii) S, N1, N2, N24, N25, N27, N28, N29, N34, N39, N40, N52, N53, N54, N66, N68, D
(viii) S, N1, N2, N24, N25, N27, N28, N29, N34, N35, N37, N38, N52, N53, N54, N66, N68, D
(ix) S, N1, N2, N24, N25, N27, N28, N29, N34, N35, N36, N38, N52, N53, N54, N66, N68, D
(x) S, N1, N2, N24, N25, N27, N28, N29, N30, N32, N33, N52, N53, N54, N66, N68, D
(xi) S, N1, N2, N24, N25, N27, N28, N29, N30, N31, N33, N52, N53, N54, N66, N68, D
(xii) S, N1, N2, N24, N25, N26, N28, N29, N30, N31, N33, N52, N53, N54, N66, N68, D
(xiii) S, N1, N2, N24, N67, N68, D

(xiv) S, N1, N2, N3, N8, N13, N20, N22, N23, N24, N67, N68, D
(xv) S, N1, N2, N3, N8, N13, N20, N21, N23, N24, N67, N68, D
(xvi) S, N1, N2, N3, N8, N13, N14, N16, N18, N19, N23, N24, N67, N68, D
(xvii) S, N1, N2, N3, N8, N13, N14, N16, N17, N19, N23, N24, N67, N68, D
(xviii) S, N1, N2, N3, N8, N13, N14, N15, N19, N23, N24, N67, N68, D
(xix) S, N1, N2, N3, N8, N9, N11, N12, N23, N24, N67, N68, D
(xx) S, N1, N2, N3, N8, N9, N10, N12, N23, N24, N67, N68, D
(xxi) S, N1, N2, N3, N8, N9, N10, N12, N23, N24, N67, N68, D
(xxii) S, N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, N12, N23, N24, N67, N68, D
(xxiii) S, N1, N2, N24, N25, N26, N28, N29, N30, N31, N52, N53, N55, N57, N59, N61, N63, N65, N66, N68, D
(xxiv) S, N1, N2, N24, N25, N26, N28, N29, N30N N31, N52, N53, N55, N57, N59, N61, N63, N64, N66, N68, D
(xxv) S, N1, N2, N24, N25, N26, N28, N29, N30, N31, N52, N53, N55, N57, N59, N61, N62, N66, N68, D
(xxvi) S, N1, N2, N24, N25, N26, N28, N29, N30, N31, N52, N53, N55, N57, N59, N60, N66, N68, D
(xxvii) S, N1, N2, N24, N25, N26, N28, N29, N30, N31, N52, N53, N55, N57, N58, N66, N68, D
(xxviii) S, N1, N2, N24, N25, N26, N28, N29, N30, N31, N52, N53, N55, N56, N66, N68, D

3.5.2 Graph Matrices

The graphs are commonly used in testing to find independent paths. Cyclomatic complexity also gives us the number of independent paths in any graph. When the size of the graph increases, it becomes difficult to identify those paths manually. We may do so with the help of a software tool and graph matrices may become the basis for designing such a tool.

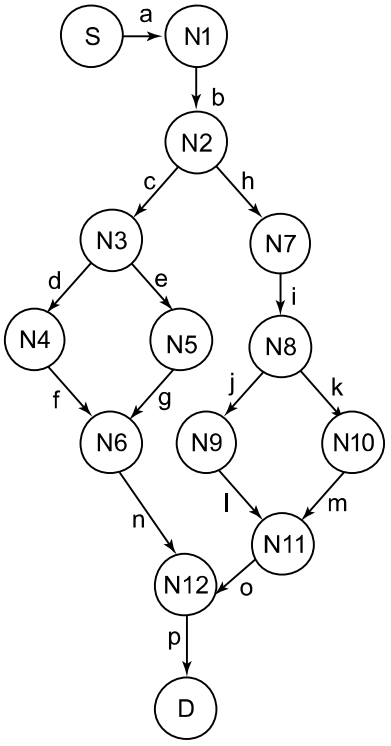
A graph matrix is a square matrix with one row and one column for every node of the graph. The size of the matrix (number of rows and number of columns) is the number of nodes of the graph. Some examples of program graphs and their graph matrices are given in Figure 3.26.



Cyclomatic Complexity = 4

	1	2	3	4	5
1			a	b	
2					e
3		d			f
4			c		g
5					

(a)



	S	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10	N11	N12	D
S		a												
N1			b											
N2				c				h						
N3					d	e								
N4							f							
N5							g							
N6													n	
N7									i					
N8										j	k			
N9												l		
N10												m		
N11													o	
N12														p
D														

(b)

Figure 3.26. Program graphs and graph matrices

Graph matrix is the tabular representation of a program graph. If we assign weight for every entry in the table, then this may be used for the identification of independent paths. The simplest weight is 1, if there is a connection and 0 if there is no connection. A matrix with such weights is known as connection matrix. A connection matrix for Figure 3.26 (b) is obtained by replacing each entry with 1, if there is a link and 0 if there is no link.

We do not show 0 entries for simplicity and blank space is treated as 0 entry as shown in Figure 3.27.

	S	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10	N11	N12	D	
S		1													1-1=0
N1			1												1-1=0
N2				1				1							2-1=1
N3					1	1									2-1=1
N4							1								1-1=0
N5							1								1-1=0
N6													1		1-1=0
N7									1						1-1=0
N8										1	1				2-1=1
N9												1			1-1=0
N10												1			1-1=0
N11													1		1-1=0
N12														1	1-1=0
D															
															3+1=4

Figure 3.27. Connection matrix for program graph shown in Figure 3.26(b)

The connection matrix can also be used to find cyclomatic complexity as shown in Figure 3.27. Each row with more than one entry represents a predicate node and cyclomatic complexity is predicate nodes plus one (Π+1).

As we know, each graph matrix expresses a direct link between nodes. If we take the square of the matrix, it shows 2-links relationships via one intermediate node. Hence, square matrix represents all paths of two links long. The K^{th} power of matrix represents all paths of K links long. We consider the graph matrix of the program graph given in Figure 3.26 (a) and find its square as shown below:

	1	2	3	4	5
1			a	b	
2					e
3		d			f
4			c		g
5					

[A]

	1	2	3	4	5
1		ad	bc		af+bg
2					
3					de
4		cd			cf
5					

$[A]^2$

There are two paths af and bg of two links between node 1 and node 5. There is no one link path from node 1 to node 5. We will get three links paths after taking the cube of this matrix as given below:

	1	2	3	4	5
1		bcd			ade+bcf
2					
3					
4					cde
5					

$[A]^3$

There are two 3-links paths from node 1 to node 5 which are ade and bcf. If we want to find four links paths, we extend this and find $[A]^4$ as given below:

	1	2	3	4	5
1					bcde
2					
3					
4					
5					

$[A]^4$

There is only one four links path – bcde, which is from node 1 to node 5. Our main objective is to use the graph matrix to find all paths between all nodes. This can be obtained by summing A, A^2, A^3….A^{n-1}. Hence, for the above examples, many paths are found and are given in Figure 3.28.

One link paths	:	a, b, c, d, e, f, g
Two links paths	:	ad, bc, af, bg, de, cd, cf
Three links paths	:	bcd, ade, bcf, cde
Four links paths	:	bcde

node 1 to node 2	:	ad, bcd
node 1 to node 3	:	a, bc
node 1 to node 4	:	b
node 1 to node 5	:	af, bg, ade, bcf, bcde
node 2 to node 1	:	-
node 2 to node 3	:	-
node 2 to node 4	:	-
node 2 to node 5	:	e
node 3 to node 1	:	-
node 3 to node 2	:	d
node 3 to node 4	:	-
node 3 to node 5	:	f, de
node 4 to node 1	:	-
node 4 to node 2	:	cd
node 4 to node 3	:	c
node 4 to node 5	:	g, cf, cde
node 5 to all other nodes	:	-

Figure 3.28. Various paths of program graph given in Figure 3.26(a)

As the cyclomatic complexity of this graph is 4, there should be 4 independent paths from node 1 (source node) to node 5 (destination node) given as:

Path 1 : af

Path 2 : bg

Path 3 : ade

Path 4 : bcf

Although 5 paths are shown, bcde does not contain any new edge or node. Thus, it cannot be treated as an independent path in this set of paths. This technique is easy to program and can be used easily for designing a testing tool for the calculation of cyclomatic complexity and generation of independent paths.

Example 3.10: Consider the program graph shown in Figure 3.29 and draw the graph and connection matrices. Find out the cyclomatic complexity and two/three link paths from a node to any other node.

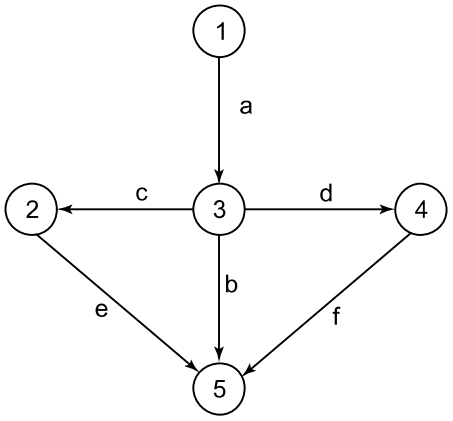


Figure 3.29. Program graph

Solution:

	1	2	3	4	5
1			a		
2					e
3		c		d	b
4					f
5					

Graph Matrix (A)

	1	2	3	4	5	
1			1			1-1=0
2					1	1-1=0
3		1		1	1	3-1=2
4					1	1-1=0
5						
						2+1=3

Connection Matrix

The graph and connection matrices are given below:

Cyclomatic complexity = e-n+2P = 6-5+2= 3

There are 3 regions in the program graph. The formula predicate node+1 (Π+1) is not applicable because predicate node 3 has three outgoing edges.

We generate square and cube matrices for [A]

	1	2	3	4	5
1		ac		ad	ab
2					
3		C		d	ce+df
4					
5					

$[A^2]$

	1	2	3	4	5
1					ace+adf
2					
3					
4					
5					

$[A^3]$

This indicates that there are the following two and three link paths:

Two links paths	ac, ad, ab, ce, df
Three links paths	ace, adf

The independent paths are:

1. ab
2. adf
3. ace

Example 3.11: Consider the DD path graph for determination of division problem shown in Figure 3.30 and draw the graph and connection matrices.

Solution: The graph and connection matrices are given in Figure 3.31 and Figure 3.32 respectively.

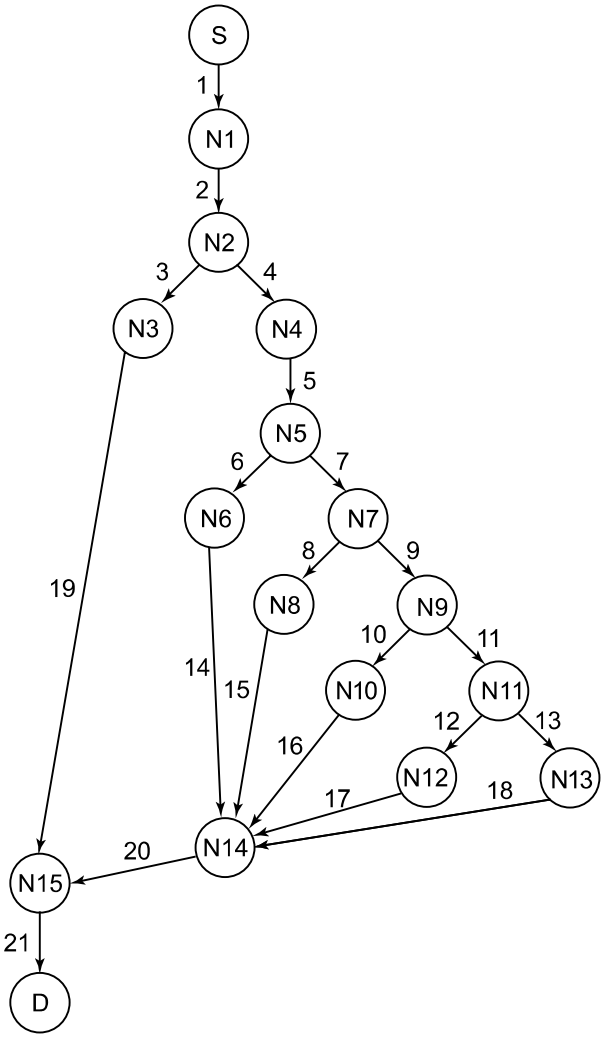


Figure 3.30. DD path graph for determination of division problem

	S	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10	N11	N12	N13	N14	N15	D
S		1															
N1			2														
N2				3	4												
N3																19	
N4						5											
N5							6	7									
N6															14		
N7									8	9							
N8															15		
N9											10	11					
N10															16		
N11													12	13			
N12															17		
N13															18		
N14																20	
N15																	21
D																	

Figure 3.31. Graph matrix for determination of division problem

	S	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10	N11	N12	N13	N14	N15	D	
S		1																1-1=0
N1			1															1-1=0
N2				1	1													2-1=1
N3																1		1-1=0
N4						1												1-1=0
N5							1	1										2-1=1
N6															1			1-1=0
N7									1	1								2-1=1
N8															1			1-1=0
N9											1	1						2-1=1
N10															1			1-1=0
N11													1	1				2-1=1
N12															1			1-1=0
N13															1			1-1=0
N14																1		1-1=0
N15																	1	1-1=0
D																		5+1=6

Figure 3.32. Connection matrix for determination of division problem

Example 3.12: Consider the DD path graph shown in Figure 3.33 for classification of triangle problem and draw the graph and connection matrices.

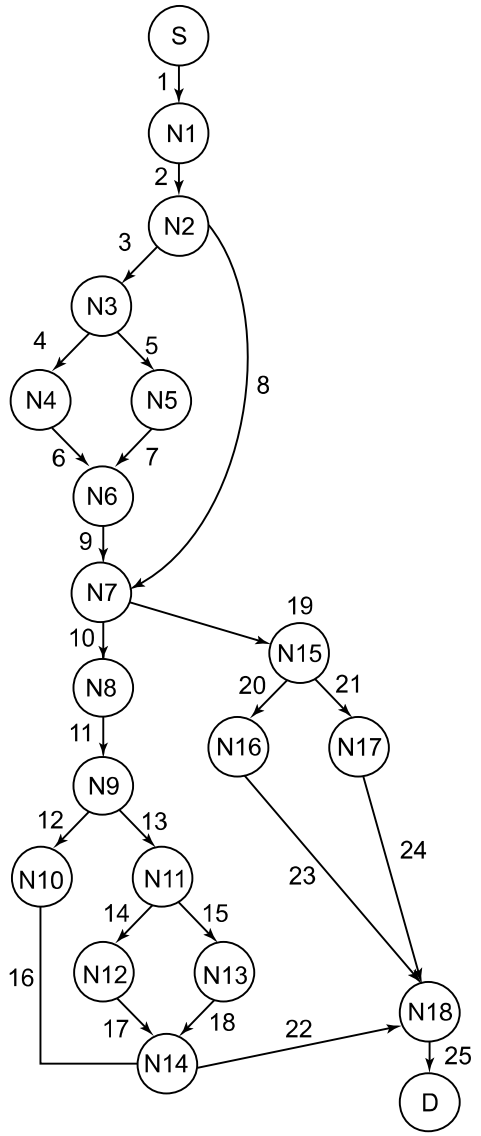


Figure 3.33. DD path graph for classification of triangle problem

Solution:

The graph and connection matrices are shown in Figure 3.34 and Figure 3.35 respectively.

	S	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10	N11	N12	N13	N14	N15	N16	N17	N18	D
S		1																		
N1			2																	
N2				3				8												
N3					4	5														
N4							6													
N5							7													
N6								9												
N7									10							19				

(*Contd.*)

(*Contd.*)

	S	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10	N11	N12	N13	N14	N15	N16	N17	N18	D
N8										11										
N9											12	13								
N10															16					
N11													14	15						
N12															17					
N13															18					
N14																			22	
N15																	20	21		
N16																			23	
N17																			24	
N18																				25
D																				

Figure 3.34. Graph matrix for classification of triangle problem

	S	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10	N11	N12	N13	N14	N15	N16	N17	N18	D	
S		1																			1-1=0
N1			1																		1-1=0
N2				1				1													2-1=1
N3					1	1															2-1=1
N4							1														1-1=0
N5							1														1-1=0
N6								1													1-1=0
N7									1							1					2-1=1
N8										1											1-1=0
N9											1	1									2-1=1
N10															1						1-1=0
N11													1	1							2-1=1
N12															1						1-1=0
N13															1						1-1=0
N14																			1		1-1=0
N15																	1	1			2-1=1
N16																			1		1-1=0
N17																			1		1-1=0
N18																				1	1-1=0
D																					
																					6+1=7

Figure 3.35 Connection matrix for classification of triangle problem

4

Structural Testing

Structural testing is considered more technical than functional testing. It attempts to design test cases from the source code and not from the specifications. The source code becomes the base document which is examined thoroughly in order to understand the internal structure and other implementation details. It also gives insight in to the source code which may be used as an essential knowledge for the design of test cases. Structural testing techniques are also known as white box testing techniques due to consideration of internal structure and other implementation details of the program. Many structural testing techniques are available and some of them are given in this chapter like control flow testing, data flow testing, slice based testing and mutation testing.

4.1 CONTROL FLOW TESTING

This technique is very popular due to its simplicity and effectiveness. We identify paths of the program and write test cases to execute those paths. As we all know, path is a sequence of statements that begins at an entry and ends at an exit. As shown in chapter 1, there may be too many paths in a program and it may not be feasible to execute all of them. As the number of decisions increase in the program, the number of paths also increase accordingly.

Every path covers a portion of the program. We define ‘coverage’ as a ‘percentage of source code that has been tested with respect to the total source code available for testing’. We may like to achieve a reasonable level of coverage using control flow testing. The most reasonable level may be to test every statement of a program at least once before the completion of testing. Hence, we may write test cases that ensure the execution of every statement. If we do so, we have some satisfaction about reasonable level of coverage. If we stop testing without achieving this level (every statement execution), we do unacceptable and intolerable activity which may lead to dangerous results in future. Testing techniques based on program coverage criterion may provide an insight about the effectiveness of test cases. Some of such techniques are discussed which are part of control flow testing.

4.1.1 Statement Coverage

We want to execute every statement of the program in order to achieve 100% statement coverage. Consider the following portion of a source code along with its program graph given in Figure 4.1.

```
        #include<stdio.h>
        #include<conio.h>

1.      void main()
2.      {
3.      int a,b,c,x=0,y=0;
4.      clrscr();
5.      printf("Enter three numbers:");
6.      scanf("%d %d %d",&a,&b,&c);
7.      if((a>b)&&(a>c)){
8.              x=a*a+b*b;
9.      }
10.     if(b>c){
11.             y=a*a-b*b;
12.     }
13.     printf("x= %d y= %d",x,y);
14.     getch();
15.     }
```

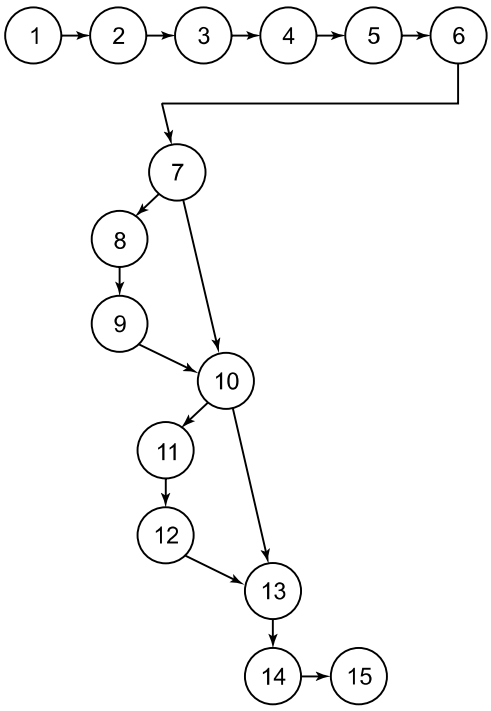


Figure 4.1. Source code with program graph

If, we select inputs like:

a=9, b=8, c=7, all statements are executed and we have achieved 100% statement coverage by only one test case. The total paths of this program graph are given as:

(i) 1–7, 10, 13–15
(ii) 1–7, 10–15
(iii) 1–10, 13–15
(iv) 1–15

The cyclomatic complexity of this graph is:

$$V(G) = e - n + 2P = 16 - 15 + 2 = 3$$

$$V(G) = \text{no. of regions} = 3$$

$$V(G) = \Pi + 1 = 2 + 1 = 3$$

Hence, independent paths are three and are given as:

(i) 1–7, 10, 13–15
(ii) 1–7, 10–15
(iii) 1–10, 13–15

Only one test case may cover all statements but will not execute all possible four paths and not even cover all independent paths (three in this case).

The objective of achieving 100% statement coverage is difficult in practice. A portion of the program may execute in exceptional circumstances and some conditions are rarely possible, and the affected portion of the program due to such conditions may not execute at all.

4.1.2 Branch Coverage

We want to test every branch of the program. Hence, we wish to test every 'True' and 'False' condition of the program. We consider the program given in Figure 4.1. If we select a = 9, b = 8, c = 7, we achieve 100% statement coverage and the path followed is given as (all true conditions):

Path = 1–15

We also want to select all false conditions with the following inputs:

a = 7, b = 8, c = 9, the path followed is

Path = 1–7, 10, 13–15

These two test cases out of four are sufficient to guarantee 100% branch coverage. The branch coverage does not guarantee 100% path coverage but it does guarantee 100% statement coverage.

4.1.3 Condition Coverage

Condition coverage is better than branch coverage because we want to test every condition at least once. However, branch coverage can be achieved without testing every condition.

Consider the seventh statement of the program given in Figure 4.1. The statement number 7 has two conditions (a>b) and (a>c). There are four possibilities namely:

(i) Both are true
(ii) First is true, second is false
(iii) First is false, second is true
(iv) Both are false

If a > b and a > c, then the statement number 7 will be true (first possibility). However, if a < b, then second condition (a > c) would not be tested and statement number 7 will be false (third and fourth possibilities). If a > b and a < c, statement number 7 will be false (second possibility). Hence, we should write test cases for every true and false condition. Selected inputs may be given as:

(i) a = 9, b = 8, c = 7 (first possibility when both are true)
(ii) a = 9, b = 8, c = 10 (second possibility – first is true, second is false)
(iii) a = 7, b = 8, c = 9 (third and fourth possibilities- first is false, statement number 7 is false)

Hence, these three test cases out of four are sufficient to ensure the execution of every condition of the program.

4.1.4 Path Coverage

In this coverage criteria, we want to test every path of the program. There are too many paths in any program due to loops and feedback connections. It may not be possible to achieve this

goal of executing all paths in many programs. If we do so, we may be confident about the correctness of the program. If it is unachievable, at least all independent paths should be executed. The program given in Figure 4.1 has four paths as given as:

(i) 1–7, 10, 13–15
(ii) 1–7, 10–15
(iii) 1–10, 13–15
(iv) 1–15

Execution of all these paths increases confidence about the correctness of the program. Inputs for test cases are given as:

S. No.	Paths Id.	Paths	Inputs			Expected Output
			a	b	c	
1.	Path-1	1-7,10, 13-15	7	8	9	x=0 y=0
2.	Path-2	1-7, 10-15	7	8	6	x=0 y=-15
3.	Path-3	1-10, 13-15	9	7	8	x=130 y=0
4.	Path-4	1-15	9	8	7	x=145 y=17

Some paths are possible from the program graph, but become impossible when we give inputs as per logic of the program. Hence, some combinations may be found to be impossible to create.

Path testing guarantee statement coverage, branch coverage and condition coverage. However, there are many paths in any program and it may not be possible to execute all the paths. We should do enough testing to achieve a reasonable level of coverage. We should execute at least (minimum level) all independent paths which are also referred to as basis paths to achieve reasonable coverage. These paths can be found using any method of cyclomatic complexity.

We have to decide our own coverage level before starting control flow testing. As we go up (statement coverage to path coverage) in the ladder, more resources and time may be required.

Example 4.1: Consider the program for the determination of the division of a student. The program and its program graph are given in Figure 3.15 and 3.16 of chapter 3 respectively. Derive test cases so that 100% path coverage is achieved.

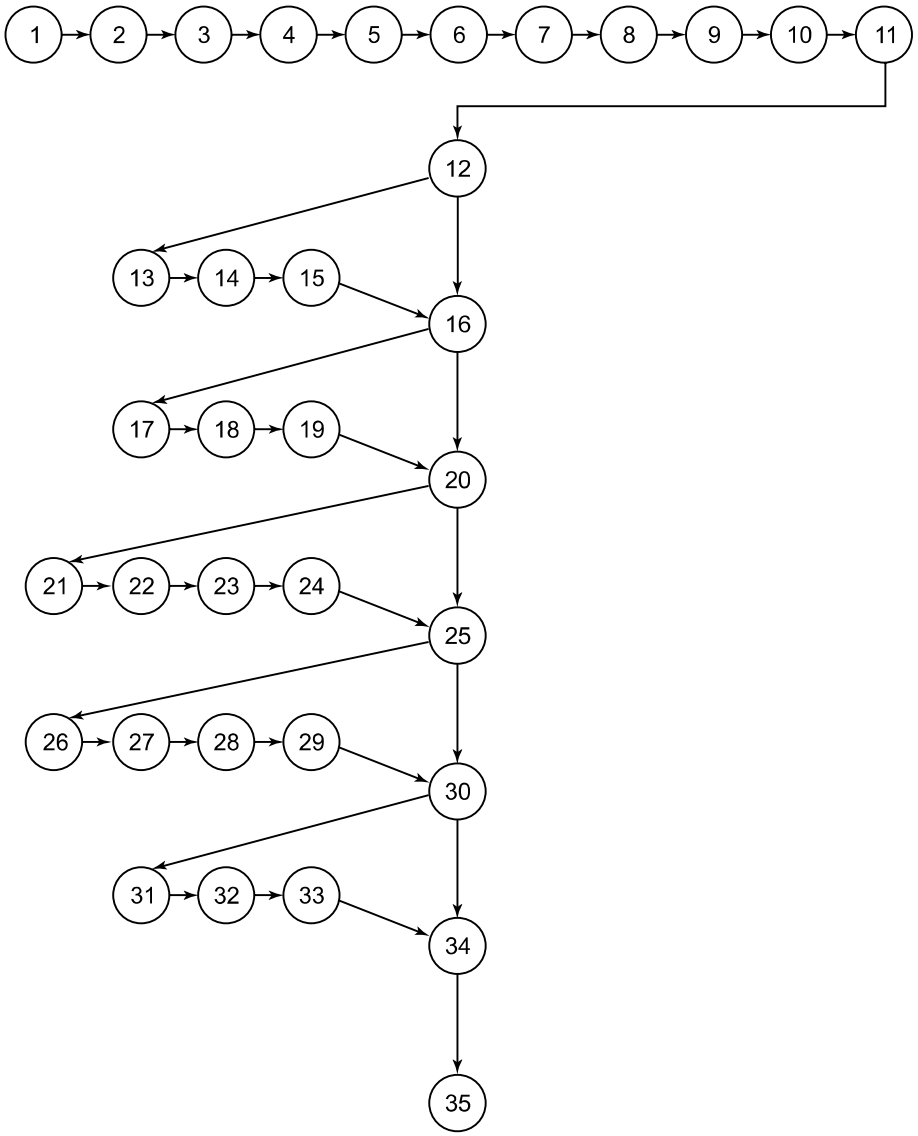
Solution:
The test cases are given in Table 4.1.

Table 4.1. Test cases

S. No.	mark1	mark2	mark3	Expected output	Paths
1.	30	-1	20	Invalid marks	1-14, 33, 34
2.	40	20	45	Fail	1-12, 15-19, 32, 33,34
3.	45	47	50	Third division	1-13, 15-17, 20-22, 32-34
4.	55	60	57	Second division	1-12, 15-17, 20, 23, 26-28, 32-34
5.	65	70	75	First division	1-12, 15-17, 20, 23, 26-28,32-34
6.	80	85	90	First division with distinction	1-12, 15-17, 20, 23, 26, 29-34

Example 4.2: Consider the program and program graph given below. Derive test cases so that 100% statement coverage and path coverage is achieved.

```
        /*Program to validate input data*/
        #include<stdio.h>
        #include<string.h>
        #include<conio.h>
1.      void main()
2.      {
3.      char fname[30],address[100],Email[100];
4.      int valid=1,flag=1;
5.      clrscr();
6.      printf("Enter first name:");
7.      scanf("%s",fname);
8.      printf("\nEnter address:");
9.      scanf("%s",address);
10.     printf("\nEnter Email:");
11.     scanf("%s",Email);
12.     if(strlen(fname)<4||strlen(fname)>30){
13.             printf("\nInvalid first name");
14.             valid=0;
15.     }
16.     if(strlen(address)<4||strlen(address)>100){
17.             printf("\nInvalid address length");
18.             valid=0;
19.     }
20.     if(strlen(Email)<8||strlen(Email)>100){
21.             printf("\nInvalid Email length");
22.             flag=0;
23.             valid=0;
24.     }
25.     if(flag==1){
26.             if(strchr(Email,'.')==0||strchr(Email,'@')==0){
27.                     printf("\nEmail must contain . and @ characters");
28.                     valid=0;
29.             }
30.     }
31.     if(valid) {
32.     printf("\nFirst name: %s \t Address: %s \t Email:
        %s",fname,address,Email);
33.     }
34.     getch();
35.     }
```



Solution:

The test cases to guarantee 100% statement and branch coverage are given in Table 4.2.

Table 4.2. Test cases for statement coverage

S. No.	First name	Address	Email	Expected output	Paths
1.	ashok	E-29, east-ofkailash	abc@yahoo.com	First name: ashok Address: E-29, east-ofkailash Email: abc@yahoo.com	1–12, 16, 20, 25, 31–35
2.	ruc	E29	abc	Invalid first name Invalid address length Invalid email length	1–25, 30, 31, 34, 35
3.	ruc	E-29	abc@yahoocom	Invalid first name Invalid address length Email must contain . and @ character	1–20, 25–31, 34, 35

(*Contd.*)

Total paths of the program graph are given in Table 4.3.

Table 4.3. Test cases for path coverage

S. No.	First name	Address	Email	Expected output	Paths
1.	-	-	-	-	1–35
2.	-	-	-	-	1–30, 34,35
3.	-	-	-	-	1–25, 30–35
4.	ruc	E29	abc	Invalid first name Invalid address length Invalid email length	1–25, 30, 31, 34, 35
5.	-	-	-	-	1–20, 25–35
6.	ruc	E-29	abc@yahoocom	Invalid first name Invalid address length Email must contain . and @ character	1–20, 25–31, 34, 35
7.	-	-	-	-	1–20, 25, 30–35
8.	ruc	E-29	Abs@yahoo.com	Invalid first name Invalid address length	1–20, 25, 30, 31, 34, 35
9.	-	-	-	-	1–16, 20–35
10.	-	-	-	-	1–16, 20–31, 34, 35
11.	-	-	-	-	1–16, 20–25, 30–35
12.	ruc	E-29, east-ofkailash	Abs	Invalid first name Invalid email length	1–16, 20–25, 30, 31, 34, 35
13.	-	-	-	-	1–16, 20, 25–35
14.	ruc	E-29, east-ofkailash	abc@yahoocom	Invalid first name Email must contain . and @ character	1–16, 20, 25–31, 34, 35
15.	-	-	-	-	1–16, 20, 25, 31–35
16.	ruc	E-29, east-ofkailash	abc@yahoo.com	Invalid first name	1–16, 20, 25, 30, 31, 34, 35
17.	-	-	-	-	1–12, 16–35
18.	-	-	-	-	1–12, 16–31, 34,35
19.	-	-	-	-	1–12, 16–25, 30–35
20.	ashok	E29	Abc	Invalid address length Invalid email length	1–12, 16–25, 30, 31, 34, 35
21.	-	-	-	-	1–12, 16–20, 25–35

(*Contd.*)

(*Contd.*)

S. No.	First name	Address	Email	Expected output	Paths
22.	ashok	E29	abc@yahoocom	Invalid address length Email must contain . and @ character	1–12, 16–20, 25–31, 34, 35
23.	-	-	-	-	1–12, 16–20, 25, 30–35
24.	ashok	E29	abc@yahoo.com	Invalid address length	1–12, 16–20, 25, 30, 31, 34, 35
25.	-	-	-	-	1–12, 16, 20–35
26.	-	-	-	-	1–12, 16, 20–31, 34, 35
27.	-	-	-	-	1–12, 16, 20–25, 30–35
28.	ashok	E-29, east-ofkailash	Abs	Invalid email length	1–12, 16, 20–25, 30, 31, 34, 35
29.	-	-	-	-	1–12, 16, 20, 25–35
30.	ashok	E-29, east-ofkailash	Abcyahoo.com	Email must contain . and @ character	1–12, 16, 20, 25–31, 34, 35
31.	ashok	E-29, east-ofkailash	abc@yahoo.com	First name: ashok Address: E-29, east-ofkailash Email: abc@yahoo.com	1–12, 16, 20, 25, 31–35
32.	-	-	-	-	1–12, 16, 20, 25, 30, 31, 34, 35

Example 4.3: Consider the program for classification of a triangle given in Figure 3.10. Derive test cases so that 100% statement coverage and path coverage is achieved.

Solution:
The test cases to guarantee 100% statement and branch coverage are given in Table 4.4.

Table 4.4. Test cases for statement coverage

S. No.	a	b	c	Expected output	Paths
1.	30	20	40	Obtuse angled triangle	1–16,20–27,34,41,42
2.	30	40	50	Right angled triangle	1–16,20–25,28–30,34,41,42
3.	40	50	60	Acute angled triangle	1–6,20–25,28,31–34,41,42
4.	30	10	15	Invalid triangle	1–14,17–21,35–37,41,42
5.	102	50	60	Input values out of range	1–13,21,35,38,39,40–42

Total paths of the program graph are given in Table 4.5.

Table 4.5. Test cases for path coverage

S. No.	a	b	c	Expected output	Paths
1.	102	–1	6	Input values out of range	1–13,21,35,38,39,40–42
2.	-	-	-	-	1–14,17–19,20,21,35,38,39,40–42
3.	-	-	-	-	1–16,20,21,35,38,39,40–42
4.	-	-	-	-	1–13,21,35,36,37,41,42
5.	30	10	15	Invalid triangle	1–14,17–21,35–37,41,42
6.	-	-	-	-	1–16,20,21,35–37,41,42
7.	-	-	-	-	1–13,21–25,28,31–34,41,42
8.	-	-	-	-	1–14,17–25,28,31–34,41,42
9.	40	50	60	Acute angled triangle	1–16,20–25,28,31–34,41,42
10.	-	-	-	-	1–13,21–25,28–30,34,41,42
11.	-	-	-	-	1–14,17–25,28–30,34,41,42
12.	30	40	50	Right angled triangle	1–16,20–25,28–30,34,41,42
13.	-	-	-	-	1–13,21–27,34,41,42
14.	-	-	-	-	1–14,17–27,34,41,42
15.	30	20	40	Obtuse angled triangle	1–16,20–27,34,41,42

Thus, there are 15 paths, out of which 10 paths are not possible to be executed as per the logic of the program.

4.2 DATA FLOW TESTING

In control flow testing, we find various paths of a program and design test cases to execute those paths. We may like to execute every statement of the program at least once before the completion of testing. Consider the following program:

```
1. # include < stdio.h>
2. void main ()
3. {
4. int a, b, c;
5. a = b + c;
6. printf ("%d", a);
7. }
```

What will be the output? The value of 'a' may be the previous value stored in the memory location assigned to variable 'a' or a garbage value. If we execute the program, we may get an unexpected value (garbage value). The mistake is in the usage (reference) of this variable without first assigning a value to it. We may assume that all variables are automatically assigned to zero initially. This does not happen always. If we define at line number 4, 'static int a, b, c', then all variables are given zero value initially. However, this is a language and compiler dependent feature and may not be generalized.

Data flow testing may help us to minimize such mistakes. It has nothing to do with data-flow diagrams. It is based on variables, their usage and their definition(s) (assignment) in the program. The main points of concern are:

(i) Statements where variables receive values (definition).
(ii) Statements where these values are used (referenced).

Data flow testing focuses on variable definition and variable usage. In line number 5 of the above program, variable 'a' is defined and variables 'b' and 'c' are used. The variables are defined and used (referenced) throughout the program. Hence, this technique concentrates on how a variable is defined and used at different places of the program.

4.2.1 Define/Reference Anomalies

Some of the define / reference anomalies are given as:

(i) A variable is defined but never used / referenced.
(ii) A variable is used but never defined.
(iii) A variable is defined twice before it is used.
(iv) A variable is used before even first-definition.

We may define a variable, use a variable and redefine a variable. So, a variable must be first defined before any type of its usage. Define / reference anomalies may be identified by static analysis of the program i.e. analyzing program without executing it. This technique uses the program graphs to understand the 'define / use' conditions of all variables. Some terms are used frequently in data flow testing and such terms are discussed in the next sub-section.

4.2.2 Definitions

A program is first converted into a program graph. As we all know, every statement of a program is replaced by a node and flow of control by an edge to prepare a program graph. There may be many paths in the program graph.

(i) Defining node
A node of a program graph is a defining node for a variable υ, if and only if, the value of the variable υ is defined in the statement corresponding to that node. It is represented as DEF (υ, n) where υ is the variable and n is the node corresponding to the statement in which υ is defined.

(ii) Usage node
A node of a program graph is a usage node for a variable υ, if and only if, the value of the variable υ is used in the statement corresponding to that node. It is represented as USE (υ, n), where 'υ' is the variable and 'n' in the node corresponding to the statement in which 'υ' is used.
A usage node USE (υ, n) is a predicate use node (denoted as P-use), if and only if, the statement corresponding to node 'n' is a predicate statement otherwise USE (υ, n) is a computation use node (denoted as C-use).

(iii) Definition use Path
A definition use path (denoted as du-path) for a variable 'υ' is a path between two nodes 'm' and 'n' where 'm' is the initial node in the path but the defining node for variable 'υ' (denoted as DEF (υ, m)) and 'n' is the final node in the path but usage node for variable 'υ' (denoted as USE (υ, n)).

(iv) Definition clear path
A definition clear path (denoted as dc-path) for a variable 'υ' is a definition use path with initial and final nodes DEF (υ, m) and USE (υ, n) such that no other node in the path is a defining node of variable 'υ'.

The du-paths and dc-paths describe the flow of data across program statements from statements where values are defined to statements where the values are used. A du-path for a variable 'υ' may have many redefinitions of variable 'υ' between initial node (DEF (υ, m)) and final node (USE (υ, n)). A dc-path for a variable 'υ' will not have any definition of variable 'υ' between initial node (DEF (υ, m)) and final node (USE (υ, n)). The du-paths that are not definition clear paths are potential troublesome paths. They should be identified and tested on topmost priority.

4.2.3 Identification of du and dc Paths

The various steps for the identification of du and dc paths are given as:

(i) Draw the program graph of the program.
(ii) Find all variables of the program and prepare a table for define / use status of all variables using the following format:

S. No.	Variable(s)	Defined at node	Used at node

(iii) Generate all du-paths from define/use variable table of step (iii) using the following format:

S. No.	Variable	du-path(begin, end)

(iv) Identify those du-paths which are not dc-paths.

4.2.4 Testing Strategies Using du-Paths

We want to generate test cases which trace every definition to each of its use and every use is traced to each of its definition. Some of the testing strategies are given as:

(i) Test all du-paths
All du-paths generated for all variables are tested. This is the strongest data flow testing strategy covering all possible du-paths.

(ii) Test all uses

Find at least one path from every definition of every variable to every use of that variable which can be reached by that definition.

For every use of a variable, there is a path from the definition of that variable to the use of that variable.

(iii) Test all definitions

Find paths from every definition of every variable to at least one use of that variable; we may choose any strategy for testing. As we go from 'test all du-paths' (no. (i)) to 'test all definitions' (no.(iii)), the number of paths are reduced. However, it is best to test all du-paths (no. (i)) and give priority to those du-paths which are not definition clear paths. The first requires that each definition reaches all possible uses through all possible du-paths, the second requires that each definition reaches all possible uses, and the third requires that each definition reaches at least one use.

4.2.5 Generation of Test Cases

After finding paths, test cases are generated by giving values to the input parameter. We get different test suites for each variable.

Consider the program given in Figure 3.11 to find the largest number amongst three numbers. Its program graph is given in Figure 3.12. There are three variables in the program namely A, B and C. Define /use nodes for all these variables are given below:

S. No.	Variable	Defined at node	Used at node
1.	A	6	11, 12, 13
2.	B	8	11, 20, 24
3.	C	10	12, 16, 20, 21

The du-paths with beginning node and end node are given as:

Variable	du-path (Begin, end)
A	6, 11 6, 12 6, 13
B	8, 11 8, 20 8, 24
C	10, 12 10, 16 10, 20 10, 21

The first strategy (best) is to test all du-paths, the second is to test all uses and the third is to test all definitions. The du-paths as per these three strategies are given as:

	Paths	Definition clear?
All du paths and all uses (Both are same in this example)	6–11 6–12 6–13 8–11 8–11, 19, 20 8–11, 19, 20, 23, 24 10–12 10–12, 15, 16 10, 11, 19, 20 10, 11, 19–21	Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes
All definitions	6–11 8–11 10–12	Yes Yes Yes

Here all du-paths and all-uses paths are the same (10 du-paths). But in the 3rd case, for all definitions, there are three paths.
Test cases are given below:

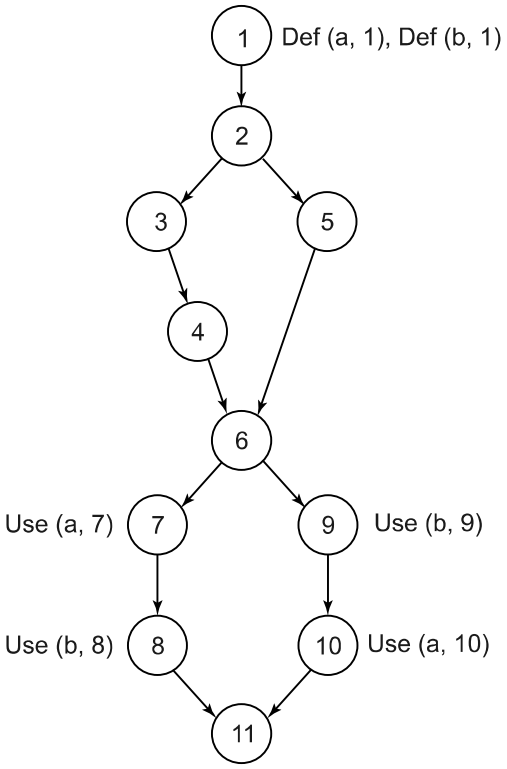
Test all du-paths

S. No.	Inputs			Expected Output	Remarks
	A	B	C		
1.	9	8	7	9	6–11
2.	9	8	7	9	6–12
3.	9	8	7	9	6–13
4.	7	9	8	9	8–11
5.	7	9	8	9	8–11, 19, 20
6.	7	9	8	9	8–11, 19, 20, 23, 24
7.	8	7	9	9	10–12
8.	8	7	9	9	10–12, ,15, 16
9.	7	8	9	9	10, 11, 19, 20
10.	7	8	9	9	10, 11, 19–21

Test All definitions

S. No.	Inputs			Expected Output	Remarks
	A	B	C		
1.	9	8	7	9	6–11
2.	7	9	8	9	8–11
3.	8	7	9	9	10–12

In this example all du-paths and all uses yield the same number of paths. This may not always be true. If we consider the following graph and find du paths with all three strategies, we will get a different number of all-du paths and all-uses paths.



Def/Use nodes table

S. No.	Variables	Defined at node	Used at node
1.	a	1	7, 10
2.	b	1	8, 9

The du paths are identified as:

S. No.	Variables	du-paths (Begin, end)
1.	a	1, 7 1, 10
2.	b	1, 8 1, 9

The du-paths are identified as per three testing strategies:

	Paths	Definition clear?
All du paths (8 paths)	1–4, 6, 7 1, 2, 5–7 1–4, 6, 9, 10 1, 2, 5, 6, 9, 10 1–4, 6, 7, 8 1, 2, 5–8 1–4, 6, 9 1, 2, 5, 6, 9	Yes Yes Yes Yes Yes Yes Yes Yes

(*Contd.*)

(*Contd.*)

	Paths	Definition clear?
All uses	1–4, 6, 7	Yes
(4 paths)	1–4, 6, 9, 10	Yes
	1–4, 6-8	Yes
	1–4, 6, 9	Yes
All definitions	1–4, 6, 7	Yes
(2 paths)	1–4, 6–8	Yes

Hence the number of paths is different in all testing strategies. When we find all du-paths, some paths may become impossible paths. We show them in order to show all combinations.

Example 4.4: Consider the program for the determination of the division problem. Its input is a triple of positive integers (mark1, mark2, mark3) and values for each of these may be from interval [0, 100]. The program is given in Figure 3.15. The output may have one of the options given below:

(i) Fail
(ii) Third division
(iii) Second division
(iv) First division
(v) First division with distinction
(vi) Invalid marks

Find all du-paths and identify those du-paths that are definition clear. Also find all du-paths, all-uses and all-definitions and generate test cases for these paths.

Solution:

(i) The program graph is given in Figure 3.16. The variables used in the program are mark1, mark2, mark3, avg.
(ii) The define/ use nodes for all variables are given below:

S. No.	Variable	Defined at node	Used at node
1.	mark1	7	12, 16
2.	mark2	9	12, 16
3.	mark3	11	12, 16
4.	avg	16	17, 20, 23, 26

(iii) The du-paths with beginning and ending nodes are given as:

S. No.	Variable	Du-path (begin, end)
1.	mark1	7, 12 7, 16
2.	mark2	9, 12 9, 16
3.	mark3	11, 12 11, 16

(*Contd.*)

(*Contd.*)

S. No.	Variable	Du-path (begin, end)
4.	Avg	16, 17
		16, 20
		16, 23
		16, 26

(iv) All du-paths, all-uses and all-definitions are given below:

	Paths	Definition clear?
All du-paths and all-uses	7–12	Yes
	7–12, 15, 16	Yes
	9–12	Yes
	9–12, 15, 16	Yes
	11, 12	Yes
	11, 12, 15, 16	Yes
	16, 17	Yes
	16, 17, 20	Yes
	16, 17, 20, 23	Yes
	16, 17, 20, 23, 26	Yes
All definitions	7–12	Yes
	9–12	Yes
	11, 12	Yes
	16, 17	Yes

Test cases for all du-paths and all-uses are given in Table 4.6 and test cases for all definitions are given in Table 4.7.

Table 4.6. Test cases for all du-paths and all-uses

S. No.	mark1	mark2	mark3	Expected Output	Remarks
1.	101	50	50	Invalid marks	7–12
2.	60	50	40	Second division	7–12, 15, 16
3.	50	101	50	Invalid marks	9–12
4.	60	70	80	First division	9–12, 15, 16
5.	50	50	101	Invalid marks	11, 12
6.	60	75	80	First division	11, 12, 15, 16
7.	30	40	30	Fail	16, 17
8.	45	50	50	Third division	16, 17, 20
9.	55	60	50	Second division	16, 17, 20, 23
10.	65	70	70	First division	16, 17, 20, 23, 26

Table 4.7. Test cases for all definitions

S. No.	mark1	mark2	mark3	Expected Output	Remarks
1.	101	50	50	Invalid marks	7–12
2.	50	101	50	Invalid marks	9–12
3.	50	50	101	Invalid marks	11, 12
4.	30	40	30	Fail	16, 17

Example 4.5: Consider the program of classification of a triangle. Its input is a triple of positive integers (a, b and c) and values for each of these may be from interval [0, 100]. The program is given in Figure 3.18. The output may have one of the options given below:

(i) Obtuse angled triangle
(ii) Acute angled triangle
(iii) Right angled triangle
(iv) Invalid triangle
(v) Input values out of range

Find all du-paths and identify those du-paths that are definition clear. Also find all du-paths, all-uses and all definitions and generate test cases from them.

Solution:

(i) The program graph is given in Figure 3.19. The variables used are a, b, c, a1, a2, a3, valid.
(ii) Define / use nodes for all variables are given below:

S. No.	Variable	Defined at node	Used at node
1.	a	8	13, 14, 22, 23, 24
2.	b	10	13, 14, 22, 23, 24
3.	c	12	13, 14, 22-24
4.	a1	22	25. 28
5.	a2	23	25, 28
6.	a3	24	25, 28
7.	valid	5, 15, 18	21, 35

(iii) The du-paths with beginning and ending nodes are given as:

S. No.	Variable	du-path (Begin, end)
1.	a	8, 13 8, 14 8, 22 8, 23 8, 24
2.	b	10, 13 10, 14 10, 22 10, 23 10, 24
3.	c	12, 13 12, 14 12, 22 12, 23 12, 24
4.	a1	22. 25 22, 28

(*Contd.*)

(*Contd.*)

S. No.	Variable	du-path (Begin, end)
5.	a2	23, 25
		23, 28
6.	a3	24, 25
		24, 28
7.	Valid	5, 21
		5, 35
		15, 21
		15, 35
		18, 21
		18, 35

All du-paths are given in Table 4.8 and the test cases for all du-paths are given in Table 4.9.

Table 4.8. All du-paths

All du-paths	Definition clear?	All du paths	Definition clear?
8–13	Yes	12–14, 17–22	Yes
8–14	Yes	12, 13, 21, 22	Yes
8–16, 20–22	Yes	12–16, 20–23	Yes
8–14, 17–22	Yes	12–14, 17–23	Yes
8–13, 21,22	Yes	12, 13, 21–23	Yes
8–16, 20–23	Yes	12–16, 20–24	Yes
8–14, 17–23	Yes	12–14, 17–24	Yes
8–13, 21–23	Yes	12, 13, 21–24	Yes
8–16, 20–24	Yes	22–25	Yes
8–14, 17–24	Yes	22–25, 28	Yes
8–13, 21–24	Yes	23–25	Yes
10–13	Yes	23–25, 28	Yes
10–14	Yes	24, 25	Yes
10–16, 20–22	Yes	24, 25, 28	Yes
10–14, 17–22	Yes	5–16, 20, 21	No
10–13, 21,22	Yes	5–14, 17–21	No
10–16, 20–23	Yes	5–13, 21	Yes
10–14, 17–23	Yes	5–16, 20, 21, 35	No
10–13, 21–23	Yes	5–14, 17–21, 35	No
10–16, 20–24	Yes	5–13, 21, 35	Yes
10–14, 17–24	Yes	15, 16, 20, 21	Yes
10–13, 21–24	Yes	15, 16, 20, 21, 35	Yes
12, 13	Yes	18–21	Yes
12–14	Yes	18–21, 35	Yes
12–16, 20–22	Yes		

We consider all combinations for the design of du-paths. In this process, test cases corresponding to some paths are not possible, but these paths are shown in the list of 'all du-paths'. They may be considered only for completion purpose.

Table 4.9. Test cases for all du-paths

S. No.	A	b	c	Expected output	Remarks
1.	30	20	40	Obtuse angled triangle	8–13
2.	30	20	40	Obtuse angled triangle	8–14
3.	30	20	40	Obtuse angled triangle	8–16, 20–22
4.	-	-	-	-	8–14, 17–22
5.	-	-	-	-	8–13, 21,22
6.	30	20	40	Obtuse angled triangle	8–16, 20–23
7.	-	-	-	-	8–14, 17–23
8.	-	-	-	-	8–13, 21–23
9.	30	20	40	Obtuse angled triangle	8–16, 20–24
10.	-	-	-	-	8–14, 17–24
11.	-	-	-	-	8–13, 21–24
12.	30	20	40	Obtuse angled triangle	10–13
13.	30	20	40	Obtuse angled triangle	10–14
14.	30	20	40	Obtuse angled triangle	10–16, 20–22
15.	-	-	-	-	10–14, 17–22
16.	-	-	-	-	10–13, 21,22
17.	30	20	40	Obtuse angled triangle	10–16, 20–23
18.	-	-	-	-	10–14, 17–23
19.	-	-	-	-	10–13, 21–23
20.	30	20	40	Obtuse angled triangle	10–16, 20–24
21.	-	-	-	-	10–14, 17–24
22.	-	-	-	-	10–13, 21–24
23.	30	20	40	Obtuse angled triangle	12, 13
24.	30	20	40	Obtuse angled triangle	12–14
25.	30	20	40	Obtuse angled triangle	12–16, 20–22
26.	-	-	-	-	12–14, 17–22
27.	-	-	-	-	12, 13, 21, 22
28.	30	20	40	Obtuse angled triangle	12–16, 20–23
29.	-	-	-	-	12–14, 17–23
30.	-	-	-	-	12, 13, 21–23
31.	30	20	40	Obtuse angled triangle	12–16, 20–24
32.	-	-	-	-	12–14, 17–24
33.	-	-	-	-	12, 13, 21–24
34.	30	20	40	Obtuse angled triangle	22–25

(*Contd.*)

(*Contd.*)

S. No.	A	b	c	Expected output	Remarks
35.	30	40	50	Right angled triangle	22–25, 28
36.	30	20	40	Obtuse angled triangle	23–25
37.	30	40	50	Right angled triangle	23–25, 28
38.	30	20	40	Obtuse angled triangle	24, 25
39.	30	40	50	Right angled triangle	24, 25, 28
40.	30	20	40	Obtuse angled triangle	5–16, 20, 21
41.	30	10	15	Invalid triangle	5–14, 17–21
42.	102	–1	6	Input values out of range	5–13, 21
43.	-	-	-	-	5–16, 20, 21, 35
44.	30	10	15	Invalid triangle	5–14, 17–21, 35
45.	102	-1	6	Input values out of range	5–13, 21, 35
46.	30	20	40	Obtuse angled triangle	15, 16, 20, 21
47.	-	-	-	-	15, 16, 20, 21, 35
48.	30	10	15	Invalid triangle	18–21
49.	30	10	15	Invalid triangle	18–21, 35

The 'all-uses' paths are given in Table 4.10 and the test cases for all du-paths are given in Table 4.11. The 'all-definitions' paths and the test cases are given in Tables 4.12 and 4.13 respectively.

Table 4.10. All uses paths for triangle classification problem

All uses	Definition clear?	All uses	Definition clear?
8–13	Yes	12–16, 20–24	Yes
8–14	Yes	22–25	Yes
8–16, 20–22	Yes	22–25, 28	Yes
8–16, 20–23	Yes	23–25	Yes
8–16, 20–24	Yes	23–25, 28	Yes
10–13	Yes	24, 25	Yes
10–14	Yes	24, 25, 28	Yes
10–16, 20–22	Yes	5–16, 20, 21	No
10–13, 21–23	Yes	5–14, 17–21, 35	No
10–16, 20–24	Yes	15, 16, 20, 21	Yes
12,13	Yes	15, 16, 20, 21, 35	Yes
12–14	Yes	18–21	Yes
12–16, 20, 21, 22	Yes	18–21, 35	Yes
12–16, 20–23	Yes		

Table 4.11. Test cases for all uses paths

S. No.	a	b	c	Expected output	Remarks
1.	30	20	40	Obtuse angled triangle	8–13
2.	30	20	40	Obtuse angled triangle	8–14
3.	30	20	40	Obtuse angled triangle	8–16, 20–22
4.	30	20	40	Obtuse angled triangle	8–16, 20–23
5.	30	20	40	Obtuse angled triangle	8–16, 20–24
6.	30	20	40	Obtuse angled triangle	10–13
7.	30	20	40	Obtuse angled triangle	10–14
8.	30	20	40	Obtuse angled triangle	10–16, 20–22
9.	30	20	40	Obtuse angled triangle	10–13, 21–23
10.	30	20	40	Obtuse angled triangle	10–16, 20–24
11.	30	20	40	Obtuse angled triangle	12,13
12.	30	20	40	Obtuse angled triangle	12–14
13.	30	20	40	Obtuse angled triangle	12–16, 20, 21, 22
14.	30	20	40	Obtuse angled triangle	12–16, 20–23
15.	30	20	40	Obtuse angled triangle	12–16, 20–24
16.	30	20	40	Obtuse angled triangle	22–25
17.	30	40	50	Right angled triangle	22–25, 28
18.	30	20	40	Obtuse angled triangle	23–25
19.	30	40	50	Right angled triangle	23–25, 28
20.	30	20	40	Obtuse angled triangle	24, 25
21.	30	40	50	Right angled triangle	24, 25, 28
22.	30	20	40	Obtuse angled triangle	5–16, 20, 21
23.	30	10	15	Invalid triangle	5–14, 17–21, 35
24.	30	20	40	Obtuse angled triangle	15, 16, 20, 21
25.	-	-	-	-	15, 16, 20, 21, 35
26.	30	10	15	Invalid triangle	18–21
27.	30	10	15	Invalid triangle	18–21, 35

Table 4.12. All definitions paths for triangle classification problem

All definitions	Definition clear?
8–13	Yes
10–13	Yes
12, 13	Yes
22–25	Yes
23–25	Yes
24,25	Yes
5–16, 20, 21	No
15, 16, 20, 21	Yes
18–21	Yes

Table 4.13. Test cases for all definitions paths

S. No.	a	b	c	Expected output	Remarks
1.	30	20	40	Obtuse angled triangle	8–13
2.	30	20	40	Obtuse angled triangle	10–13
3.	30	20	40	Obtuse angled triangle	12, 13
4.	30	20	40	Obtuse angled triangle	22–25
5.	30	20	40	Obtuse angled triangle	23–25
6.	30	20	40	Obtuse angled triangle	24,25
7.	30	20	40	Obtuse angled triangle	5–16, 20, 21
8.	30	20	40	Obtuse angled triangle	15, 16, 20, 21
9.	30	10	15	Invalid triangle	18–21

Example 4.6: Consider the program given in Figure 3.21 for the determination of day of the week. Its input is at triple of positive integers (day, month, year) from the interval

$1 \leq \text{day} \leq 31$

$1 \leq \text{month} \leq 12$

$1900 \leq \text{year} \leq 2058$

The output may be:

[Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday]

Find all du-paths and identify those du-paths that are definition clear. Also find all du-paths, all-uses and all-definitions and generate test cases for these paths.

Solution:

(i) The program graph is given in Figure 3.22. The variables used in the program are day, month, year, century, Y, Y1, M, date, validDate, leap.

(ii) Define / use nodes for all variables are given below:

S. No.	Variable	Defined at node	Used at node
1.	Day	6	19, 27, 30, 37, 91 93, 96, 99, 102 105, 108, 111, 115
2.	Month	8	18, 26, 37, 54 62, 70, 73, 76, 79 82, 85, 93, 96, 99 102, 105, 108, 111, 115
3.	Year	10	11, 12, 14, 45, 47 51, 93, 96, 99, 102 105, 108, 111, 115
4.	Century	46, 50	91
5.	Y	53	91
6.	Y1	47, 51	53

(*Contd.*)

(*Contd.*)

S. No.	Variable	Defined at node	Used at node
7.	M	56, 59, 64 67, 71, 74 77, 80, 83 86, 89	91
8.	Date	91	92, 95, 98, 101, 104, 107
9.	ValidDate	3, 20, 23 28, 31, 34, 38, 41	44
10.	Leap	3, 13, 15	27, 55, 63

(iii) The du-paths with beginning and ending nodes are given as:

S. No.	Variable	du-path (begin, end)
1.	Day	6, 19 6, 27 6, 30 6, 37 6, 91 6, 93 6, 96 6, 99 6, 102 6, 105 6, 108 6, 111 6, 115
2.	Month	8, 18 8, 26 8, 37 8, 54 8, 62 8, 70 8, 73 8, 76 8, 79 8, 82 8, 85 8, 93 8, 96 8, 99 8, 102 8, 105 8, 108 8, 111 8, 115

(*Contd.*)

(*Contd.*)

S. No.	Variable	du-path (begin, end)
3.	Year	10, 11
		10, 12
		10, 14
		10, 45
		10, 47
		10, 51
		10, 93
		10, 96
		10, 99
		10, 102
		10, 105
		10, 108
		10, 111
		10, 115
4.	Century	46, 91
		50, 91
5.	Y	53, 91
6.	Y1	47, 53
		51, 53
7.	M	56, 91
		59, 91
		64, 91
		67, 91
		71, 91
		74, 91
		77, 91
		80, 91
		83, 91
		86, 91
		89, 91
8.	Date	91, 92
		91, 95
		91, 98
		91, 101
		91, 104
		91, 107
9.	ValidDate	3, 44
		20, 44
		23, 44
		28, 44
		31, 44
		34, 44
		38, 44
		41, 44

(*Contd.*)

(*Contd.*)

S. No.	Variable	du-path (begin, end)
10.	Leap	3, 27 3, 55 3, 63 13, 27 13, 55 13, 63 15, 27 15, 55 15, 63

There are more than 10,000 du-paths and it is neither possible nor desirable to show all of them. The all uses paths and their respective test cases are shown in Table 4.14 and Table 4.15 respectively. The 'all definitions' paths are shown in Table 4.16 and their corresponding test cases are given in Table 4.17.

Table 4.14. All uses paths for determination of the day of week problem

All uses	Definition clear?
6–19	Yes
6–18, 26, 27	Yes
6–18, 26, 27, 30	Yes
6–18, 26, 37	Yes
6–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91	Yes
6–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91–93	Yes
6–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 96	Yes
6–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 99	Yes
6–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 101, 102	Yes
6–21, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 101, 104, 105	Yes
6–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 101,104, 107, 108	Yes
6–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 101, 104, 107, 110, 111	Yes
6–11, 44, 114, 115	Yes
8–18	Yes
8–18, 26	Yes
8–18, 26, 37	Yes
8–21, 25, 43–48, 53, 54	Yes
8–21, 25, 43–48, 53, 54, 62	Yes
8–25, 43–48, 53, 54, 62, 70	Yes
8–21, 25, 43–48, 53, 54, 62, 70, 73	Yes
8–21, 25, 43–48, 53, 54, 62, 70, 73, 76	Yes

(*Contd.*)

(*Contd.*)

All uses	Definition clear?
8–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79	Yes
8–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79, 82	Yes
8–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79, 82, 85	Yes
8–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 93	Yes
8–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 96	Yes
8–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 99	Yes
8–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 101, 102	Yes
8–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 101, 104, 105	Yes
8–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 101, 104, 107, 108	Yes
8–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 101, 104, 107, 110, 111	Yes
8–11, 44, 114, 115	Yes
10, 11	Yes
10–12	Yes
10–14	Yes
10–21, 25, 43–45	Yes
10–21, 25, 43–47	Yes
10–21, 25, 43–45, 49–51	Yes
10–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91–93	Yes
10–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 96	Yes
10–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 99	Yes
10–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 101, 102	Yes
10–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 101, 104, 105	Yes
10–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 101, 104, 107, 108	Yes
10–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 101, 104, 107, 110, 111	Yes
10, 11, 44, 114, 115	Yes
46–48, 53–57, 61, 91	Yes
50–57, 61, 91	Yes
53–61, 91	Yes
47, 48, 53	Yes
51–53	Yes
56, 57, 61, 91	Yes
59–61, 91	Yes

(*Contd.*)

(*Contd.*)

All uses	Definition clear?
64, 65, 69, 91	Yes
67–69, 91	Yes
71, 72, 91	Yes
74, 75, 91	Yes
77, 78, 91	Yes
80, 81, 91	Yes
83, 84, 91	Yes
86, 87, 91	Yes
89, 90, 91	Yes
91, 92	Yes
91, 92, 95	Yes
91, 92, 95, 98	Yes
91, 92, 95, 98, 101	Yes
91, 92, 95, 98, 101, 104	Yes
91, 92, 95, 98, 101, 104, 107	Yes
3–11, 44	No
20, 21, 25, 43, 44	Yes
23–25, 43, 44	Yes
28, 29, 36, 43, 44	Yes
31, 32, 36, 43, 44	Yes
34–36, 43, 44	Yes
38, 39, 43, 44	Yes
41–44	Yes
3–18, 26, 27	No
3–18, 26, 37–39, 43–48, 53–55	No
3–18, 26, 27, 30–32, 36, 43–48, 53, 54, 62, 63	No
13–18, 26, 27	No
13–18, 26, 37–39, 43–48, 53–55	No
13–18, 26, 27, 30–32, 36, 43–48, 53, 54, 62, 63	No
15–18, 26, 27	Yes
15–18, 26, 37–39, 43–48, 53–55	Yes
15–18, 26, 27, 30–32, 36, 43–48, 53, 54, 62, 63	Yes

Table 4.15. Test cases for all uses

S. No.	Month	Day	Year	Expected output	Remarks
1.	6	15	1900	Friday	6–19
2.	2	15	1900	Thursday	6–18, 26, 27
3.	2	15	1900	Thursday	6–18, 26, 27, 30
4.	7	15	1900	Sunday	6–18, 26, 37
5.	6	15	1900	Friday	6–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91
6.	6	10	1900	Sunday	6–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91–93
7.	6	11	1900	Monday	6–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 96
8.	6	12	1900	Tuesday	6–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 99
9.	6	13	1900	Wednesday	6–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 101, 102
10.	6	14	1900	Thursday	6–21, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 101, 104, 105
11.	6	15	1900	Friday	6–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 101,104, 107, 108
12.	6	16	1900	Saturday	6–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 101, 104, 107, 110, 111
13.	6	15	2059	Invalid Date	6–11, 44, 114, 115
14.	6	15	1900	Friday	8–18
15.	2	15	1900	Thursday	8–18, 26
16.	1	15	1900	Monday	8–18, 26, 37
17.	6	15	1900	Friday	8–21, 25, 43–48, 53, 54
18.	6	15	1900	Friday	8–21, 25, 43–48, 53, 54, 62
19.	6	15	1900	Friday	8–25, 43–48, 53, 54, 62, 70
20.	4	15	1900	Sunday	8–21, 25, 43–48, 53, 54, 62, 70, 73
21.	6	15	1900	Friday	8–21, 25, 43–48, 53, 54, 62, 70, 73, 76
22.	6	15	1900	Friday	8–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79
23.	9	15	1900	Saturday	8–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79, 82

(*Contd.*)

(*Contd.*)

S. No.	Month	Day	Year	Expected output	Remarks
24.	9	15	1900	Saturday	8–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79, 82, 85
25.	6	10	1900	Sunday	8–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 93
26.	6	11	1900	Monday	8–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 96
27.	6	12	1900	Tuesday	8–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 99
28.	6	13	1900	Wednesday	8–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79,80, 81, 91, 92, 95, 98, 101, 102
29.	6	14	1900	Thursday	8–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79,80, 81, 91, 92, 95, 98, 101, 104, 105
30.	6	15	1900	Friday	8–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79, 80, 81, 91, 92, 95, 98, 101, 104, 107, 108
31.	6	16	1900	Saturday	8–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79, 80, 81, 91, 92, 95, 98, 101, 104, 107, 110, 111
32.	6	15	2059	Invalid Date	8–11, 44, 114, 115
33.	6	15	1900	Friday	10, 11
34.	6	15	1900	Friday	10–12
35.	6	15	1900	Friday	10–14
36.	6	15	1900	Friday	10–21, 25, 43–45
37.	6	15	1900	Friday	10–21, 25, 43–47
38.	6	15	2009	Monday	10–21, 25, 43–45, 49–51
39.	6	10	1900	Sunday	10–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91–93
40.	6	11	1900	Monday	10–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 96
41.	6	12	1900	Tuesday	10–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 99
42.	6	13	1900	Wednesday	10–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 101, 102
43.	6	14	1900	Thursday	10–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 101, 104, 105
44.	6	15	1900	Friday	10–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 101, 104, 107, 108

(*Contd.*)

(*Contd.*)

S. No.	Month	Day	Year	Expected output	Remarks
45.	6	16	1900	Saturday	10–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 101, 104, 107, 110, 111
46.	6	15	2059	Invalid Date	10, 11, 44, 114, 115
47.	1	15	1900	Monday	46–48, 53–57, 61, 91
48.	1	15	2009	Thursday	50–57, 61, 91
49.	1	15	2009	Thursday	53–61, 91
50.	6	15	1900	Friday	47, 48, 53
51.	6	15	2009	Monday	51–53
52.	1	15	2009	Thursday	56, 57, 61, 91
53.	1	15	2000	Saturday	59–61, 91
54.	1	15	2009	Thursday	64, 65, 69, 91
55.	2	15	2000	Tuesday	67–69, 91
56.	3	15	2009	Sunday	71, 72, 91
57.	4	15	2009	Wednesday	74, 75, 91
58.	5	15	2009	Friday	77, 78, 91
59.	6	15	2009	Monday	80, 81, 91
60.	8	15	2009	Saturday	83, 84, 91
61.	9	15	2009	Tuesday	86, 87, 91
62.	7	15	2009	Wednesday	89, 90, 91
63.	5	7	2009	Sunday	91, 92
64.	6	7	2009	Monday	91, 92, 95
65.	7	7	2009	Tuesday	91, 92, 95, 98
66.	8	7	2009	Wednesday	91, 92, 95, 98, 101
67.	9	7	2009	Thursday	91, 92, 95, 98, 101, 104
68.	10	7	2009	Friday	91, 92, 95, 98, 101, 104, 107
69.	6	15	1900	Friday	3–11, 44
70.	6	15	1900	Friday	20, 21, 25, 43, 44
71.	6	31	2009	Invalid Date	23–25, 43, 44

(*Contd.*)

(*Contd.*)

S. No.	Month	Day	Year	Expected output	Remarks
72.	2	15	2000	Tuesday	28, 29, 36, 43, 44
73.	2	15	2009	Sunday	31, 32, 36, 43, 44
74.	2	30	2009	Invalid Date	34–36, 43, 44
75.	8	15	2009	Saturday	38,39, 43, 44
76.	13	1	2009	Invalid Date	41–44
77.	2	15	1900	Thursday	3–18, 26, 27
78.	1	15	1900	Monday	3–18, 26, 37–39, 43–48, 53–55
79.	2	15	1900	Thursday	3–18, 26, 27, 30–32, 36, 43–48, 53, 54, 62, 63
80.	2	15	1900	Thursday	13–18, 26, 27
81.	1	15	1900	Monday	13–18, 26, 37–39, 43–48, 53–55
82.	2	15	1900	Thursday	13–18, 26, 27, 30–32, 36, 43–48, 53, 54, 62, 63
83.	2	15	1900	Thursday	15–18, 26, 27
84.	1	15	1900	Monday	15–18, 26, 37–39, 43–48, 53–55
85.	2	15	1900	Thursday	15–18, 26, 27, 30–32, 36, 43–48, 53, 54, 62, 63

Table 4.16. All definitions paths for determination of the day of week problem

All definitions	Definition clear?
6–19	Yes
8–18	Yes
10, 11	Yes
46–48, 53–57, 61, 91	Yes
50–57, 61, 91	Yes
53–57, 61, 91	Yes
47, 48, 53	Yes
51–53	Yes
56, 57, 61, 91	Yes
59, 60, 61, 91	Yes
64, 65, 69, 91	Yes
67–69, 91	Yes
71, 72, 91	Yes
74, 75, 91	Yes

(*Contd.*)

(*Contd.*)

All definitions	Definition clear?
77, 78, 91	Yes
80, 81, 91	Yes
83, 84, 91	Yes
86, 87, 91	Yes
89–91	Yes
91, 92	Yes
3–11, 44	No
20, 21, 25, 43, 44	Yes
23–25, 43, 44	Yes
28, 29, 36, 43, 44	Yes
31, 32, 36, 43, 44	Yes
34–36, 43, 44	Yes
38, 39, 43, 44	Yes
41–44	Yes
3–18, 26, 27	No
13–18, 26, 27	No
15–18, 26, 27	Yes

Table 4.17. Test cases for all definitions

S. No.	Month	Day	Year	Expected output	Remarks
1.	6	15	1900	Friday	6–19
2.	6	15	1900	Friday	8–18
3.	6	15	1900	Friday	10, 11
4.	1	15	1900	Monday	46–48, 53–57, 61, 91
5.	1	15	2009	Thursday	50–57, 61, 91
6.	1	15	2009	Thursday	53–57, 61, 91
7.	6	15	1900	Friday	47, 48, 53
8.	6	15	2009	Monday	51–53
9.	1	15	2009	Thursday	56, 57, 61, 91
10.	1	15	2000	Saturday	59, 60, 61, 91
11.	1	15	2009	Thursday	64, 65, 69, 91
12.	2	15	2000	Tuesday	67–69, 91
13.	3	15	2009	Sunday	71, 72, 91
14.	4	15	2009	Wednesday	74, 75, 91
15.	5	15	2009	Friday	77, 78, 91
16.	6	15	2009	Monday	80, 81, 91
17.	8	15	2009	Saturday	83, 84, 91

(*Contd.*)

(*Contd.*)

S. No.	Month	Day	Year	Expected output	Remarks
18.	9	15	2009	Tuesday	86, 87, 91
19.	7	15	2009	Wednesday	89–91
20.	6	15	2009	Monday	91, 92
21.	6	15	2059	Invalid Date	3–11, 44
22.	6	15	1900	Friday	20, 21, 25, 43, 44
23.	6	31	2009	Invalid Date	23–25, 43, 44
24.	2	15	2000	Tuesday	28, 29, 36, 43, 44
25.	2	15	2009	Sunday	31, 32, 36, 43, 44
26.	2	30	2009	Invalid Date	34–36, 43, 44
27.	8	15	2009	Saturday	38, 39, 43, 44
28.	13	1	2009	Invalid Date	41–44
29.	2	15	1900	Thursday	3–18, 26, 27
30.	2	15	1900	Thursday	13–18, 26, 27
31.	2	15	1900	Thursday	15–18, 26, 27

4.3 SLICE BASED TESTING

Program slicing was introduced by Mark Weiser [WEIS84] where we prepare various subsets (called slices) of a program with respect to its variables and their selected locations in the program. Each variable with one of its location will give us a program slice. A large program may have many smaller programs (its slices), each constructed for different variable subsets. The slices are typically simpler than the original program, thereby simplifying the process of testing of the program. Keith and James [KEIT91] have explained this concept as:

> "Program slicing is a technique for restricting the behaviour of a program to some specified subset of interest. A slice S(υ, n) of program P on variable υ, or set of variables, at statement n yields the portions of the program that contributed to the value of υ just before statement n is executed. S (υ, n) is called a slicing criteria. Slices can be computed automatically on source programs by analyzing data flow. A program slice has the added advantage of being an executable program."

Hence, slices are smaller than the original program and may be executed independently. Only two things are important here, variable and its selected location in the program.

4.3.1 Guidelines for Slicing

There are many variables in the program but their usage may be different in different statements. The following guidelines may be used for the creation of program slices.

1. All statements where variables are defined and redefined should be considered. Consider the program for classification of a triangle (given in Figure 3.18) where variable 'valid' is defined at line number 5 and redefined at line number 15 and line number 18.

5 int valid = 0
15 valid = 1
18 valid = –1

Hence, we may create S(valid, 5), S(valid, 15) and S(valid, 18) slices for variable 'valid' of the program.

2. All statements where variables receive values externally should be considered. Consider the triangle problem (given in Figure 3.18) where variables 'a', 'b' and 'c' receive values externally at line number 8, line number 10 and line number 12 respectively as shown below:
   8 scanf ("%lf", &a);
   10 scanf ("%lf", &b);
   12 scanf ("%lf", &c);
   Hence, we may create S(a, 8), S(b, 10) and S(c, 12) slices for these variables.
3. All statements where output of a variable is printed should be considered. Consider the program to find the largest amongst three numbers (given in Figure 3.11) where variable 'C' is printed at line number 16 and 21 as given below:
   16 printf ("The largest number is: % f \n", C);
   21 printf ("The largest number is: % f \n", C)
   Hence, we may create S(C, 16) and S(C, 21) as slices for 'C' variable
4. All statements where some relevant output is printed should be considered. Consider the triangle classification program where line number 26, 29, 32, 36 and 39 are used for printing the classification of the triangle (given in Figure 3.18) which is very relevant as per logic of the program. The statements are given as:
   26 printf ("Obtuse angled triangle");
   29 printf ("Right angled triangle");
   32 printf ("Acute angled triangle");
   36 printf ("\nInvalid triangle");
   39 printf ("\nInput Values out of Range");
   We may create S(a1, 26), S(a1, 29), S(a1, 32), S(valid, 36) and S(valid, 39) as slices. These are important slices for the purpose of testing.
5. The status of all variables may be considered at the last statement of the program. We consider the triangle classification program (given in figure 3.18) where line number 42 is the last statement of the program. We may create S(a1, 42), S(a2, 42), S(a3, 42), S(valid, 42), S(a, 42), S(b,42) and S(c, 42) as slices.

4.3.2 Creation of Program Slices

Consider the portion of a program given in Figure 4.2 for the identification of its slices.

1. a = 3;
2. b = 6;
3. c = b^2;
4. d = $a^2 + b^2$;
5. c = a + b;

Figure 4.2. Portion of a program

We identify two slices for variable 'c' at statement number 3 and statement number 5 as given in Figure 4.3.

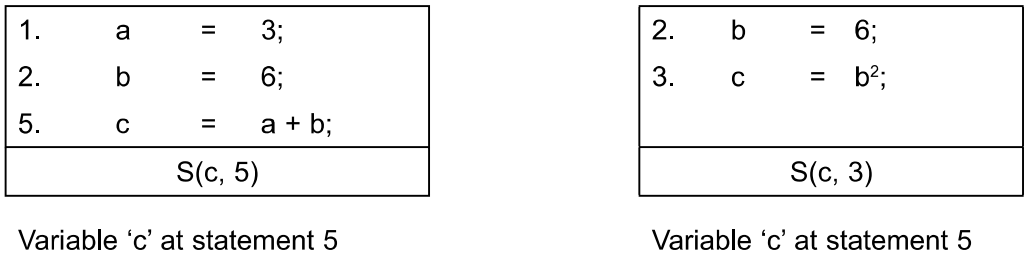


Figure 4.3. Two slices for variable 'c'

Consider the program given in Figure 4.4.

```
1.  void main ( )
2.  {
3.  int a, b, c, d, e;
4.  printf ("Enter the values of a, b and c \ n");
5.  scanf ("%d %d %d", & a, &b, &c);
6.  d = a+b;
7.  e = b+c:
8.  printf ("%d", d);
9.  printf ("%d", e);
10. }
```

Figure 4.4. Example program

Many slices may be created as per criterion (mentioned in section 4.3.1) of the program given in the Figure 4.4. Some of these slices are shown below:

```
1.  main ( )
2.  {
3.  int a, b, c, d, e;
4.  printf ("Enter the values of a, b and c \ n");
5.  scanf ("%d %d %d", &a, &b, &c);
7.  e = b + c;
9.  printf ("%d", e);
10. }
```

Slice on criterion S (e, 10) = (1, 2, 3, 4, 5, 7, 9, 10)

```
1.  main ( )
2.  {
3.  int a, b, c, d, e;
4.  printf (“Enter the values of a, b and c \ n”);
5.  scanf (“%d %d %d”, &a, &b, &c);
6.  d = a + b;
8.  printf (“%d”, d);
10. }
```

Slice on criterion S (d,10) = (1, 2, 3, 4, 5, 6, 8, 10)

```
1. main ( )
2. {
3. int a, b, c, d, e;
4. printf (“Enter the values of a, b and c \ n”);
5. scanf (“%d %d %d”, &a, &b, &c);
7.  e = b + c;
10. }
```

Slice on criterion S (e,7) = (1, 2, 3, 4, 5, 7,10)

```
1.  main ( )
2.  {
3.  int a, b, c, d, e;
4.  printf (“Enter the values of a, b and c \ n”);
5.  scanf (“%d %d %d”, &a, &b, &c);
6.  d = a + b;
10. }
```

Slice on criterion S (d,6) = (1, 2, 3, 4, 5, 6, 10)

```
1.  main ( )
2.  {
3.  int a, b, c, d, e;
4.  printf (“Enter the values of a, b and c \ n”);
5.  scanf (“%d %d %d”, &a, &b, &c);
10. }
```

Slice on criterion S (a, 5) = (1, 2, 3, 4, 5, 10)

We also consider the program to find the largest number amongst three numbers as given in Figure 3.11. There are three variables A, B and C in the program. We may create many slices like S (A, 28), S (B, 28), S (C, 28) which are given in Figure 4.8.

Some other slices and the portions of the program covered by these slices are given as:

S (A, 6) = {1– 6, 28}
S (A, 13) = {1–14, 18, 27, 28}
S (B, 8) = {1– 4, 7, 8, 28}
S (B, 24) = {1–11, 18–20, 22–28}
S (C, 10) = {1– 4, 9, 10, 28}
S (C, 16) = {1–12, 14–18, 27, 28}
S (C, 21) = {1–11, 18–22, 26–28}

It is a good programming practice to create a block even for a single statement. If we consider C++/C/Java programming languages, every single statement should be covered with curly braces { }. However, if we do not do so, the compiler will not show any warning / error message. In the process of generating slices we delete many statements (which are not required in the slice). It is essential to keep the starting and ending brackets of the block of the deleted statements. It is also advisable to give a comment 'do nothing' in order to improve the readability of the source code.

```
    #include<stdio.h>
    #include<conio.h>
1.  void main()
2.  {
3.  float A,B,C;
4.  clrscr();
5.  printf("Enter number 1:\n");
6.  scanf("%f", &A);
7.  printf("Enter number 2:\n");
8.  scanf("%f", &B);
9.  printf("Enter number 3:\n");
10. scanf("%f", &C);
11. if(A>B) {
12. if(A>C) {
13. printf("The largest number is: %f\n",A);
14.             }
18. }
27. getch();
28. }
```

(a) S(A, 28) ={1–14, 18, 27, 28}

```
    #include<stdio.h>
    #include<conio.h>
1.  void main()
2.  {
3.  float A,B,C;
4.  clrscr();
5.  printf("Enter number 1:\n");
6.  scanf("%f", &A);
7.  printf("Enter number 2:\n");
8.  scanf("%f", &B);
9.  printf("Enter number 3:\n");
10. scanf("%f", &C);
11. if(A>B) { /*do nothing*/
18. }
19. else {
20. if(C>B) { /*do nothing*/
22.             }
23. else  {
24. printf("The largest number is: %f\n",B);
25.     }
26. }
27. getch();
28. }
```

(b) S(B, 28) ={1–11, 18–20, 22–28}

(*Contd.*)

(*Contd.*)

```
        #include<stdio.h>
        #include<conio.h>
 1.     void main()
 2.     {
 3.     float A,B,C;
 4.     clrscr();
 5.     printf("Enter number 1:\n");
 6.     scanf("%f", &A);
 7.     printf("Enter number 2:\n");
 8.     scanf("%f", &B);
 9.     printf("Enter number 3:\n");
 10.    scanf("%f", &C);
 11.    if(A>B) { /*do nothing*/
 18.    }
 19.    else {
 20.    if(C>B) {
 21.                printf("The largest number is: %f\n",C);
 22.                }
 26.    }
 27.    getch();
 28.    }
```

(c) S(C, 28)={1–11, 18–22, 26–28}

Figure 4.5. Some slices of program in Figure 3.11

A statement may have many variables. However, only one variable should be used to generate a slice at a time. Different variables in the same statement will generate a different program slice. Hence, there may be a number of slices of a program depending upon the slicing criteria. Every slice is smaller than the original program and can be executed independently. Each slice may have one or more test cases and may help us to focus on the definition, redefinition, last statement of the program, and printing/reading of a variable in the slice. Program slicing has many applications in testing, debugging, program comprehension and software measurement. A statement may have many variables. We should use only one variable of a statement for generating a slice.

4.3.3 Generation of Test Cases

Every slice should be independently executable and may cover some lines of source code of the program as shown in previous examples. The test cases for the slices of the program given in Figure 3.3 (to find the largest number amongst three numbers) are shown in Table 4.18. The generated slices are S(A, 6), S(A, 13), S(A, 28), S(B, 8), S(B, 24), S(B, 28), S(C, 10), S(C, 16), S(C, 21), S(C, 28) as discussed in previous section 4.3.1.

Table 4.18. Test cases using program slices of program to find the largest among three numbers

S. No.	Slice	Lines covered	A	B	C	Expected output
1.	S(A, 6)	1–6, 28	9			No output
2.	S(A, 13)	1–14, 18, 27, 28	9	8	7	9
3.	S(A, 28)	1–14, 18, 27, 28	8	8	7	9
4.	S(B, 8)	1–4, 7, 8, 28		9		No output
5.	S(B, 24)	1–11, 18–20, 22–28	7	9	8	9
6.	S(B, 28)	1–11, 19, 20, 23–28	7	9	8	9
7.	S(C, 10)	1–4, 9, 10, 28			9	No output
8.	S(C, 16)	1–12, 14–18, 27, 28	8	7	9	9
9.	S(C, 21)	1–11, 18–22, 26–28	7	8	9	9
10.	S(C, 28)	1–11, 18–22, 26–28	7	8	9	9

Slice based testing is a popular structural testing technique and focuses on a portion of the program with respect to a variable location in any statement of the program. Hence slicing simplifies the way of testing a program's behaviour with respect to a particular subset of its variables. But slicing cannot test a behaviour which is not represented by a set of variables or a variable of the program.

Example 4.7: Consider the program for determination of division of a student. Consider all variables and generate possible program slices. Design at least one test case from every slice.

Solution:
There are four variables – mark1, mark2, mark3 and avg in the program. We may create many slices as given below:

S (mark1, 7) = {1–7, 34}

S (mark1, 13) = {1–14, 33, 34}

S (mark2, 9) = {1–5, 8, 9, 34}

S (mark2, 13) = {1–14, 33, 34}

S (mark3, 11) = {1–5, 10, 11, 34}

S (mark3, 13) = {1–14, 33, 34}

S (avg, 16) = {1–12, 14–16, 32, 34}

S (avg, 18) = {1–12, 14–19, 32–34}

S (avg, 21) = {1–12, 14–17, 19–22, 29, 31–34}

S (avg, 24) = {1–12, 14–17, 19, 20, 22–25, 29, 31–34}

S (avg, 27) = {1–12, 14–17, 19, 20, 22, 23, 25–29, 31–34}

S (avg, 30) = {1–12, 14–17, 19, 20, 22, 23, 25, 26, 28–34}

The program slices are given in Figure 4.6 and their corresponding test cases are given in Table 4.19.

```
    #include<stdio.h>
    #include<conio.h>
1.  void main()
2.  {
3.  int mark1, mark2,mark3,avg;
4.  clrscr();
5.  printf("Enter marks of 3 subjects
    (between 0-100)\n");
6.  printf("Enter marks of first
    subject:");
7.  scanf("%d", &mark1);
34. }
```

(a) S(mark1,7)/S(mark1,34)

```
    #include<stdio.h>
    #include<conio.h>
1.  void main()
2.  {
3.  int mark1, mark2,mark3,avg;
4.  clrscr();
5.  printf("Enter marks of 3 subjects (between
    0-100)\n");
8.  printf("Enter marks of second subject:");
9.  scanf("%d", &mark2);
34. }
```

(b) S(mark2,9)/S(mark2,34)

```
    #include<stdio.h>
    #include<conio.h>
1.  void main()
2.  {
3.  int mark1, mark2,mark3,avg;
4.  clrscr();
5.  printf("Enter marks of 3 subjects
    (between 0-100)\n");
10. printf("Enter marks of third subject:");
11. scanf("%d",&mark3);
34. }
```

(c) S(mark3,11)/S(mark3,34)

```
    #include<stdio.h>
    #include<conio.h>
1.  void main()
2.  {
3.  int mark1, mark2,mark3,avg;
4.  clrscr();
5.  printf("Enter marks of 3 subjects (between
    0-100)\n");
6.  printf("Enter marks of first subject:");
7.  scanf("%d", &mark1);
8.  printf("Enter marks of second subject:");
9.  scanf("%d", &mark2);
10. printf("Enter marks of third subject:");
11. scanf("%d",&mark3);
12. if(mark1>100||mark1<0||mark2>100||mark2<0||
    mark3>100||mark3<0){
13. printf("Invalid Marks! Please try again");
14. }
33. getch();
34. }
```

(d) S(mark1,13)/S(mark2,13)/S(mark3,13)

(*Contd.*)

(*Contd.*)

```
        #include<stdio.h>
        #include<conio.h>
1.      void main()
2.      {
3.      int mark1, mark2,mark3,avg;
4.      clrscr();
5.      printf("Enter marks of 3 subjects (between
        0-100)\n");
6.      printf("Enter marks of first subject:");
7.      scanf("%d", &mark1);
8.      printf("Enter marks of second subject:");
9.      scanf("%d", &mark2);
10.     printf("Enter marks of third subject:");
11.     scanf("%d",&mark3);
12.     if(mark1>100||mark1<0||mark2>100||mark2
        <0||mark3>100||mark3<0){ /* do nothing*/
14.             }
15.     else {
16.     avg=(mark1+mark2+mark3)/3;
17.     if(avg<40){
18.             printf("Fail");
19.         }
32.     }
33.     getch();
34.     }
```

(e) S(avg,18)

```
        #include<stdio.h>
        #include<conio.h>
1.      void main()
2.      {
3.      int mark1, mark2,mark3,avg;
4.      clrscr();
5.      printf("Enter marks of 3 subjects (between
        0-100)\n");
6.      printf("Enter marks of first subject:");
7.      scanf("%d", &mark1);
8.      printf("Enter marks of second subject:");
9.      scanf("%d", &mark2);
10.     printf("Enter marks of third subject:");
11.     scanf("%d",&mark3);
12.     if(mark1>100||mark1<0||mark2>100||mark2
        <0||mark3>100||mark3<0){
14.     } /* do nothing*/
15.     else {
16.     avg=(mark1+mark2+mark3)/3;
17.     if(avg<40){ /* do nothing*/
19.             }
20.     else if(avg>=40&&avg<50) {
21.             printf("Third Division");
22.         }
29.     else { /* do nothing*/
31.             }
32.     }
33.     getch();
34.     }
```

(f) S(avg,21)

```
        #include<stdio.h>
        #include<conio.h>
1.      void main()
2.      {
3.      int mark1, mark2,mark3,avg;
4.      clrscr();
5.      printf("Enter marks of 3 subjects (between
        0-100)\n");
```

(*Contd.*)

(*Contd.*)

```
6.      printf("Enter marks of first subject:");
7.      scanf("%d", &mark1);
8.      printf("Enter marks of second subject:");
9.      scanf("%d", &mark2);
10.     printf("Enter marks of third subject:");
11.   scanf("%d",&mark3);
12.   if(mark1>100||mark1<0||mark2>100||mark2
      <0||mark3>100||mark3<0) {
      /* do nothing*/
14.             }
15.   else {
16.   avg=(mark1+mark2+mark3)/3;
17.   if(avg<40) { /* do nothing*/
19.             }
20.   else if(avg>=40&&avg<50) {
      /* do nothing*/
22.             }
23.   else if(avg>=50&&avg<60) {
24.             printf("Second Division");
25.             }
29.   else      { /* do nothing*/
31.             }
32.   }
33.   getch();
34.   }
```

(g) S(avg,24)

```
6.      printf("Enter marks of first subject:");
7.      scanf("%d", &mark1);
8.      printf("Enter marks of second subject:");
9.      scanf("%d", &mark2);
10.     printf("Enter marks of third subject:");
11.   scanf("%d",&mark3);
12.   if(mark1>100||mark1<0||mark2>100||mar
      k2<0||mark3>100||mark3<0) {
      /* do nothing*/
14.             }
15.   else {
16.   avg=(mark1+mark2+mark3)/3;
17.   if(avg<40) { /* do nothing*/
19.             }
20.   else if(avg>=40&&avg<50) {
      /* do nothing*/
22.             }
23.   else if(avg>=50&&avg<60) {
25.             }
26.   else if(avg>=60&&avg<75) {
27.             printf("First Division");
28.             }
29.   else      { /* do nothing*/
31.             }
32.   }
33.   getch();
34.   }
```

(h) S(avg,27)

```
      #include<stdio.h>
      #include<conio.h>
1.    void main()
2.    {
3.    int mark1, mark2,mark3,avg;
4.    clrscr();
5.    printf("Enter marks of 3 subjects (between 0-100)\n");
6.    printf("Enter marks of first subject:");
7.    scanf("%d", &mark1);
8.    printf("Enter marks of second subject:");
```

(*Contd.*)

(*Contd.*)

```
9.    scanf("%d", &mark2);
10.   printf("Enter marks of third subject:");
11.   scanf("%d",&mark3);
12.   if(mark1>100||mark1<0||mark2>100||mark2<0||mark3>100||mark3<0) { /* do nothing*/
14.           }
15.   else {
16.   avg=(mark1+mark2+mark3)/3;
17.   if(avg<40) { /* do nothing*/
19.           }
20.   else if(avg>=40&&avg<50) {/* do nothing*/
22.           }
23.   else if(avg>=50&&avg<60) {/* do nothing*/
25.           }
26.   else if(avg>=60&&avg<75) {/* do nothing*/
28.           }
29.   else    {
30.           printf("First Division with Distinction");
31.           }
32.   }
33.   getch();
34.   }
```

(i) S(avg,30)/S(avg,34)

Figure 4.6. Slices of program for determination of division of a student

Table 4.19. Test cases using program slices

S. No.	Slice	Line covered	mark1	mark2	mark3	Expected output
1.	S(mark1, 7)	1–7, 34	65			No output
2.	S(mark1, 13)	1–14, 33, 34	101	40	50	Invalid marks
3.	S(mark1, 34)	1–7, 34	65			No output
4.	S(mark2, 9)	1–5, 8, 9, 34		65		No output
5.	S(mark2, 13)	1–14, 33, 34	40	101	50	Invalid marks
6.	S(mark2, 34)	1–5, 8, 9, 34		65		No output
7.	S(mark3, 11)	1–5, 10, 11, 34			65	No output
8.	S(mark3, 13)	1–14, 33, 34	40	50	101	Invalid marks

(*Contd.*)

(*Contd.*)

S. No.	Slice	Line covered	mark1	mark2	mark3	Expected output
9.	S(mark3, 34)	1–5, 10, 11, 34			65	No output
10.	S(avg, 16)	1–12, 14–16, 32, 34	45	50	45	No output
11.	S(avg, 18)	1–12, 14–19, 32–34	40	30	20	Fail
12.	S(avg, 21)	1–12, 14–17, 19–22, 29, 32–34	45	50	45	Third division
13.	S(avg, 24)	1–12, 14–17, 19, 20, 22–25, 29, 31–34	55	60	57	Second division
14.	S(avg, 27)	1–12, 14–17, 19, 20, 22, 23, 25–29, 31–34	65	67	65	First division
15.	S(avg, 30)	1–12, 14–17, 19, 20, 22, 23, 25, 26, 28–34	79	80	85	First division with distinction
16.	S(avg, 34)	1–12, 14–17, 19, 20, 22, 23, 25, 26, 28–34	79	80	85	First division with distinction
17.	S(avg, 16)	1–12, 14–16, 32, 34	45	50	45	No output

Example 4.8: Consider the program for classification of a triangle. Consider all variables and generate possible program slices. Design at least one test case from every slice.

Solution:

There are seven variables 'a', 'b', 'c', 'a1', 'a2', 'a3' and 'valid' in the program. We may create many slices as given below:

i. S (a, 8) = {1–8, 42}
ii. S (b, 10) = {1–6, 9, 10, 42}
iii. S (c, 12) = {1–6, 11, 12, 42}
iv. S (a1, 22) = {1–16, 20–22, 34, 42}
v. S (a1, 26) = {1–16, 20–22, 25–27, 34, 41, 42}
vi. S (a1, 29) = {1–16, 20–22, 25, 27–31, 33, 34, 41, 42}
vii. S (a1, 32) = {1–16, 20–22, 25, 27, 28, 30–34, 41, 42}
viii. S (a2, 23) = {1–16, 20, 21,23, 34, 42}
ix. S (a2, 26) = {1–16, 20, 21, 23, 25–27, 34, 41, 42)
x. S (a2, 29) = {1–16, 20, 21, 23, 25, 27–31, 33, 34, 41, 42}
xi. S (a2, 32) = {1–16, 20, 21, 23, 25, 27, 28, 30–34, 41, 42}
xii. S (a3, 26) = {1–16, 20, 21, 24–27, 34, 41, 42}
xiii. S (a3, 29) = {1–16, 20, 21, 24, 25, 27–31, 33, 34, 41,42}
xiv. S (a3, 32) = {1–16, 20, 21, 24, 25, 27, 28, 30–34, 41, 42}
xv. S (valid, 5) = {1–5, 42}
xvi. S (valid, 15) = {1–16, 20, 42}
xvii. S (valid, 18) = {1–14, 16–20, 42}
xviii. S (valid, 36) = {1–14, 16–20, 21, 34–38, 40–42}
xix. S (valid, 39) = {1–13, 20, 21, 34, 35, 37–42}

The test cases of the above slices are given in Table 4.20.

Table 4.20. Test cases using program slices

S. No.	Slice	Path	a	b	c	Expected output
1.	S(a, 8)/S(a,42)	1–8, 42	20			No output
2.	S(b, 10)/S(b,42)	1–6, 9, 10, 42		20		No output
3.	S(c, 12)/S(c,42)	1–6, 11, 12, 42			20	No output
4.	S(a1, 22)	1–16, 20–22, 34, 42	30	20	40	No output
5.	S(a1, 26)	1–16, 20–22, 25–27, 34, 41, 42	30	20	40	Obtuse angled triangle
6.	S(a1, 29)	1–16, 20–22, 25, 27–31, 33, 34, 41, 42	30	40	50	Right angled triangle
7.	S(a1, 32)	1–16, 20–22, 25, 27, 28, 30–34, 41, 42	50	60	40	Acute angled triangle
8.	S(a1, 42)	1–16, 20–22, 34, 42	30	20	40	No output
9.	S(a2, 23)	1–16, 20, 21, 23, 34, 42	30	20	40	No output
10.	S(a2, 26)	1–16, 20, 21, 23, 25–27, 34, 41, 42	40	30	20	Obtuse angled triangle
11.	S(a2, 29)	1–16, 20, 21, 23, 25, 27–31, 33, 34, 41, 42	50	40	30	Right angled triangle
12.	S(a2, 32)	1–16, 20, 21, 23, 25, 27, 28, 30–34, 41, 42	40	50	60	Acute angled triangle
13.	S(a2, 42)	1–16, 20, 21, 23, 34, 42	30	20	40	No output
14.	S(a3, 24)	1–16, 20, 21, 24, 34, 42	30	20	40	No output
15.	S(a3, 26)	1–16, 20, 21, 24–27, 34, 41, 42	20	40	30	Obtuse angled triangle
16.	S(a3, 29)	1–16, 20, 21, 24, 25, 27–31, 33, 34, 41, 42	40	50	30	Right angled triangle
17.	S(a3, 32)	1–16, 20, 21, 24, 25, 27, 28, 30–34, 41, 42	50	40	60	Acute angled triangle
18.	S(a3, 42)	1–16, 20, 21, 24, 34, 42	30	20	40	No output
19.	S(valid,5)	1–2, 5, 42				No output
20.	S(valid,15)	1–16, 20, 42	20	40	30	No output
21.	S(valid,18)	1–14, 16–20, 42	30	10	15	No output
22.	S(valid,36)	1–14, 16–20, 21, 34–38, 40–42	30	10	15	Invalid triangle
23.	S(valid,39)	1–13, 20, 21, 34, 35, 37–42	102	–1	6	Input values out of range
24.	S(valid,42)	1–14, 16–20, 42	30	10	15	No output

Example 4.9. Consider the program for determination of day of the week given in Figure 3.13. Consider variables day, validDate, leap and generate possible program slices. Design at least one test case from each slice.

Solution:
There are ten variables – day, month, year, century Y, Y1, M, date, valid date, and leap. We may create many slices for variables day, validDate and leap as given below:

1.	S(day, 6)	=	{1–6, 118}
2.	S(day, 93)	=	{1–11, 18–21, 25, 43–48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78–81, 88, 90–94, 113, 117, 118}
3.	S(day, 96)	=	{1–11, 18–21, 25, 43–48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78–81, 88, 90–92, 94–97, 110, 112, 113, 117, 118}
4.	S(day, 99)	=	{1–11, 18–21, 25, 43–48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78–81, 88, 90–92, 94, 95, 97–100, 110, 112, 113, 117, 118}
5.	S(day, 102)	=	{1–11, 18–21, 25, 43–48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78–81, 88, 90–92, 94, 95, 97, 98, 100–103, 110, 112, 113, 117, 118}
6.	S(day, 105)	=	{1–11, 18–21, 25, 43–48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78–81, 88, 90–92, 94, 95, 97, 98, 100, 101, 103–106, 110, 112, 113, 117, 118}
7.	S(day, 108)	=	{1–11, 18–21, 25, 43–48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78–81, 88, 90–92, 94, 95, 97, 98, 100, 101, 103, 104, 106–110, 112, 113, 117, 118}
8.	S(day, 111)	=	{1–11, 18–21, 25, 43–48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78–81, 88, 90–92, 94, 95, 97, 98, 100, 101, 103, 104, 106, 107, 109–113, 117, 118}
9.	S(day, 115)	=	{1–11, 43, 44, 113–118}
10.	S(day, 118)	=	{1–6, 118}
11.	S(validDate,3)	=	{1–3, 118}
12.	S(validDate,20)	=	{1–11, 18–21, 25, 43, 118}
13.	S(validDate,23)	=	{1–11, 18, 19, 21–25, 43, 118}
14.	S(validDate,28)	=	{1–13, 17, 18, 25, 26–29, 36, 40, 42, 43, 118}
15.	S(validDate,31)	=	{1–11, 18, 25, 26, 27, 29–33, 35, 36, 40, 42, 43, 118}
16.	S(validDate,34)	=	{1–11, 18, 25, 26, 27, 29, 30, 32–36, 40, 42, 43, 118}
17..	S(validDate,38)	=	{1–11, 18, 25, 26, 36–40, 42, 43, 118}
18.	S(validDate,41)	=	{1–11, 18, 25, 26, 36, 37, 39–43, 118}
19.	S(validDate,118)	=	{1–11, 18, 25, 26, 36, 37, 39–43, 118}
20.	S(leap,3)	=	{1–3, 118}
21.	S(leap,13)	=	{1–13, 17, 43, 118}
22.	S(leap,15)	=	{1–17, 43, 118}
23.	S(leap,118)	=	{1–17, 43, 118}

The test cases for the above slices are given in Table 4.21.

Table 4.21. Test cases using program slices

S. No.	Slice	Lines covered	Month	Day	Year	Expected output
1.	S(day, 6)	1–6, 118	6	-	-	No output
2.	S(day, 93)	1–11, 18–21, 25, 43–48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78–81, 88, 90–94, 113, 117, 118	6	13	1999	Sunday
3.	S(day, 96)	1–11, 18–21, 25, 43–48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78–81, 88, 90–92, 94–97, 110, 112, 113, 117, 118	6	14	1999	Monday
4.	S(day, 99)	1–11, 18–21, 25, 43–48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78–81, 88, 90–92, 94, 95, 97–100, 110, 112, 113, 117, 118	6	15	1999	Tuesday
5.	S(day, 102)	1–11, 18–21, 25, 43–48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78–81, 88, 90–92, 94, 95, 97, 98, 100–103, 110, 112, 113, 117, 118	6	16	1999	Wednesday
6.	S(day, 105)	1–11, 18–21, 25, 43–48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78–81, 88, 90–92, 94, 95, 97, 98, 100, 101, 103–106, 110, 112, 113, 117, 118	6	17	1999	Thursday
7.	S(day, 108)	1–11, 18–21, 25, 43–48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78–81, 88, 90–92, 94, 95, 97, 98, 100, 101, 103, 104, 106–110, 112, 113, 117, 118	6	18	1999	Friday
8.	S(day, 111)	1–11, 18–21, 25, 43–48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78–81, 88, 90–92, 94, 95, 97, 98, 100, 101, 103, 104, 106, 107, 109–113, 117, 118	6	19	1999	Saturday
9.	S(day, 115)	1–11, 43, 44, 113–118	6	31	2059	Invalid Date
10.	S(day, 118)	1–6, 118	6	19	1999	Saturday
11.	S(validDate,3)	1–3, 118	-	-	-	No output
12.	S(validDate,20)	1–11, 18–21, 25, 43, 118	6	15	2009	No output
13.	S(validDate,23)	1–11, 18, 19, 21–25, 43, 118	6	31	2009	No output
14.	S(validDate,28)	1–13, 17, 18, 25, 26–29, 36, 40, 42, 43, 118	2	15	2000	No output

(*Contd.*)

(*Contd.*)

S. No.	Slice	Lines covered	Month	Day	Year	Expected output
15.	S(validDate,31)	1–11, 18, 25, 26, 27, 29–33, 35, 36, 40, 42, 43, 118	2	15	2009	No output
16.	S(validDate,34)	1–11, 18, 25, 26, 27, 29, 30, 32–36, 40, 42, 43, 118	2	29	2009	No output
17.	S(validDate,38)	1–11, 18, 25, 26, 36–40, 42, 43, 118	8	15	2009	No output
18.	S(validDate,41)	1–11, 18, 25, 26, 36, 37, 39–43, 118	13	15	2009	No output
19.	S(validDate,118)	1–11, 18, 25, 26, 36, 37, 39–43, 118	13	15	2009	No output
20.	S(leap,3)	1–3, 118	-	-	-	No output
21.	S(leap,13)	1–13, 17, 43, 118	8	15	2000	No output
22.	S(leap,15)	1–17, 43, 118	8	15	1900	No output
23.	S(leap,118)	1–17, 43, 118	8	15	1900	No output

4.4 MUTATION TESTING

It is a popular technique to assess the effectiveness of a test suite. We may have a large number of test cases for any program. We neither have time nor resources to execute all of them. We may select a few test cases using any testing technique and prepare a test suite. How do we assess the effectiveness of a selected test suite? Is this test suite adequate for the program? If the test suite is not able to make the program fail, there may be one of the following reasons:

(i) The test suite is effective but hardly any errors are there in the program. How will a test suite detect errors when they are not there?

(ii) The test suite is not effective and could not find any errors. Although there may be errors, they could not be detected due to poor selection of test suite. How will errors be detected when the test suite is not effective?

In both the cases, we are not able to find errors, but the reasons are different. In the first case, the program quality is good and the test suite is effective and in the second case, the program quality is not that good and the test suite is also not that effective. When the test suite is not able to detect errors, how do we know whether the test suite is not effective or the program quality is good? Hence, assessing the effectiveness and quality of a test suite is very important. Mutation testing may help us to assess the effectiveness of a test suite and may also enhance the test suite, if it is not adequate for a program.

4.4.1 Mutation and Mutants

The process of changing a program is known as mutation. This change may be limited to one, two or very few changes in the program. We prepare a copy of the program under test and make a change in a statement of the program. This changed version of the program is known as a

mutant of the original program. The behaviour of the mutant may be different from the original program due to the introduction of a change. However, the original program and mutant are syntactically correct and should compile correctly. To mutate a program means to change a program. We generally make only one or two changes in order to assess the effectiveness of the selected test suite. We may make many mutants of a program by making small changes in the program. Every mutant will have a different change in a program. Consider a program to find the largest amongst three numbers as given in Figure 3.11 and its two mutants are given in Figure 4.7 and Figure 4.8. Every change of a program may give a different output as compared to the original program.

Many changes can be made in the program given in Figure 3.11 till it is syntactically correct. Mutant M_1 is obtained by replacing the operator '>' of line number 11 by the operator ' ='. Mutant M_2 is obtained by changing the operator '>' of line number 20 to operator '<'. These changes are simple changes. Only one change has been made in the original program to obtain mutant M_1 and mutant M_2.

```
        #include<stdio.h>
        #include<conio.h>
1.      void main()
2.      {
3.      float A,B,C;
4.      clrscr();
5.      printf("Enter number 1:\n");
6.      scanf("%f", &A);
7.      printf("Enter number 2:\n");
8.      scanf("%f", &B);
9.      printf("Enter number 3:\n");
10.     scanf("%f", &C);
        /*Check for greatest of three numbers*/
11.     if(A>B){   ← if(A=B) { mutated statement ('>' is replaced by '=')
12.     if(A>C) {
13.             printf("The largest number is: %f\n",A);
14.             }
15.     else  {
16.             printf("The largest number is: %f\n",C);
17.             }
18.     }
19.     else {
20.     if(C>B) {
21.             printf("The largest number is: %f\n",C);
22.             }
23.     else  {
24.             printf("The largest number is: %f\n",B);
25.             }
26.     }
```

(Contd.)

(*Contd.*)

```
27.         getch();
28.         }
```

M_1 : First order mutant

Figure 4.7. Mutant$_1$ (M_1) of program to find the largest among three numbers

```
            #include<stdio.h>
            #include<conio.h>
1.          void main()
2.          {
3.          float A,B,C;
4.          clrscr();
5.          printf("Enter number 1:\n");
6.          scanf("%f", &A);
7.          printf("Enter number 2:\n");
8.          scanf("%f", &B);
9.          printf("Enter number 3:\n");
10.         scanf("%f", &C);
            /*Check for greatest of three numbers*/
11.         if(A>B) {
12.         if(A>C) {
13.                 printf("The largest number is: %f\n",A);
14.                 }
15.         else  {
16.                 printf("The largest number is: %f\n",C);
17.                 }
18.         }
19.         else {
20.         if(C>B) {   ← if(C<B) { mutated statement ('>' is replaced by '<')
21.                 printf("The largest number is: %f\n",C);
22.                 }
23.         else  {
24.                 printf("The largest number is: %f\n",B);
25.                 }
26.         }
27.         getch();
28.         }
```

M_2 : First order mutant

Figure 4.8. Mutant$_2$ (M_2) of program to find the largest among three numbers

The mutants generated by making only one change are known as first order mutants. We may obtain second order mutants by making two simple changes in the program and third order mutants by making three simple changes, and so on. The second order mutant (M_3) of the program given in Figure 3.11 is obtained by making two changes in the program and thus changing operator '>' of line number 11 to operator '<' and operator '>' of line number 20 to '≥' as given in Figure 4.9. The second order mutants and above are called higher order mutants. Generally, in practice, we prefer to use only first order mutants in order to simplify the process of mutation.

```
        #include<stdio.h>
        #include<conio.h>
1.      void main()
2.      {
3.      float A,B,C;
4.      clrscr();
5.      printf("Enter number 1:\n");
6.      scanf("%f", &A);
7.      printf("Enter number 2:\n");
8.      scanf("%f", &B);
9.      printf("Enter number 3:\n");
10.     scanf("%f", &C);
        /*Check for greatest of three numbers*/
11.     if(A>B) {   ← if(A<B) { mutated statement (replacing '>' by '<')
12.     if(A>C) {
13.             printf("The largest number is: %f\n",A);
14.             }
15.     else  {
16.             printf("The largest number is: %f\n",C);
17.             }
18.     }
19.     else {
20.     if(C>B) {   ← if(C≥B) { mutated statement (replacing '>'by '≥')
21.             printf("The largest number is: %f\n",C);
22.             }
23.     else  {
24.             printf("The largest number is: %f\n",B);
25.             }
26.     }
27.     getch();
28.     }
```

M_3 : Second order mutant

Figure 4.9. Mutant$_3$ (M_3) of program to find the largest among three numbers

4.4.2 Mutation Operators

Mutants are produced by applying mutant operators. An operator is essentially a grammatical rule that changes a single expression to another expression. The changed expression should be grammatically correct as per the used language. If one or more mutant operators are applied to all expressions of a program, we may be able to generate a large set of mutants. We should measure the degree to which the program is changed. If the original expression is x + 1, and the mutant for that expression is x + 2, that is considered as a lesser change as compared to a mutant where the changed expression is (y * 2) by changing both operands and the operator. We may have a ranking scheme, where a first order mutant is a single change to an expression, a second order mutant is a mutation to a first order mutant, and so on. Higher order mutants become difficult to manage, control and trace. They are not popular in practice and first order mutants are recommended to be used. To kill a mutant, we should be able to execute the changed statement of the program. If we are not able to do so, the fault will not be detected. If x – y is changed to x – 5 to make a mutant, then we should not use the value of y to be equal to 5. If we do so, the fault will not be revealed. Some of the mutant operators for object oriented languages like Java, C++ are given as:

(i) Changing the access modifier, like public to private.
(ii) Static modifier change
(iii) Argument order change
(iv) Super Keyword change
(v) Operator change
(vi) Any operand change by a numeric value.

4.4.3 Mutation Score

When we execute a mutant using a test suite, we may have any of the following outcomes:

(i) The results of the program are affected by the change and any test case of the test suite detects it. If this happens, then the mutant is called a killed mutant.
(ii) The results of the program are not affected by the change and any test case of the test suite does not detect the mutation. The mutant is called a live mutant.
The mutation score associated with a test suite and its mutants is calculated as:

$$\text{Mutation Score} = \frac{\text{Number of mutants killed}}{\text{Total number of mutants}}$$

The total number of mutants is equal to the number of killed mutants plus the number of live mutants. The mutation score measures how sensitive the program is to the changes and how accurate the test suite is. A mutation score is always between 0 and 1. A higher value of mutation score indicates the effectiveness of the test suite although effectiveness also depends on the types of faults that the mutation operators are designed to represent.

The live mutants are important for us and should be analyzed thoroughly. Why is it that any test case of the test suite not able to detect the changed behaviour of the program? One of the reasons may be that the changed statement was not executed by these test cases. If executed,

then also it has no effect on the behaviour of the program. We should write new test cases for live mutants and kill all these mutants. The test cases that identify the changed behaviour should be preserved and transferred to the original test suite in order to enhance the capability of the test suite. Hence, the purpose of mutation testing is not only to assess the capability of a test suite but also to enhance the test suite. Some mutation testing tools are also available in the market like Insure++, Jester for Java (open source) and Nester for C++ (open source).

Example 4.10: Consider the program to find the largest of three numbers as given in figure 3.11. The test suite selected by a testing technique is given as:

S. No.	A	B	C	Expected Output
1.	6	10	2	10
2.	10	6	2	10
3.	6	2	10	10
4.	6	10	20	20

Generate five mutants (M_1 to M_5) and calculate the mutation score of this test suite.

Solution:
The mutated line numbers and changed lines are shown in Table 4.22.

Table 4.22. Mutated statements

Mutant No.	Line no.	Original line	Modified Line
M_1	11	if(A>B)	if (A<B)
M_2	11	if(A>B)	if(A>(B+C))
M_3	12	if(A>C)	if(A<C)
M_4	20	if(C>B)	if(C=B)
M_5	16	printf("The Largest number is:%f\n",C);	printf("The Largest number is:%f\n",B);

The actual output obtained by executing the mutants M_1-M_5 is shown in Tables 4.23-4.27.

Table 4.23. Actual output of mutant M_1

Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	6
2.	10	6	2	10	6
3.	6	2	10	10	10
4.	6	10	20	20	20

Table 4.24. Actual output of mutant M_2

Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	10
2.	10	6	2	10	10
3.	6	2	10	10	10
4.	6	10	20	20	20

Table 4.25. Actual output of mutant M_3

Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	10
2.	10	6	2	10	2
3.	6	2	10	10	6
4.	6	10	20	20	20

Table 4.26. Actual output of mutant M_4

Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	10
2.	10	6	2	10	10
3.	6	2	10	10	10
4.	6	10	20	20	10

Table 4.27. Actual output of mutant M_5

Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	10
2.	10	6	2	10	10
3.	6	2	10	10	2
4.	6	10	20	20	20

$$\text{Mutation Score} = \frac{\text{Number of mutants killed}}{\text{Total number of mutants}}$$
$$= \frac{4}{5}$$
$$= 0.8$$

Higher the mutant score, better is the effectiveness of the test suite. The mutant M_2 is live in the example. We may have to write a specific test case to kill this mutant. The additional test case is given in Table 4.28.

Table 4.28. Additional test case

Test case	A	B	C	Expected output
5.	10	5	6	10

Now when we execute the test case 5, the actual output will be different from the expected output (see Table 4.29), hence the mutant will be killed.

Table 4.29. Output of added test case

Test case	A	B	C	Expected output	Actual output
5.	10	5	6	10	6

This test case is very important and should be added to the given test suite. Therefore, the revised test suite is given in Table 4.30.

Table 4.30. Revised test suite

Test case	A	B	C	Expected output
1.	6	10	2	10
2.	10	6	2	10
3.	6	2	10	10
4.	6	10	20	20
5.	10	5	6	10

Example 4.11: Consider the program for classification of triangle given in Figure 3.18. The test suite A and B are selected by two different testing techniques and are given in Table 4.31 and Table 4.32, respectively. The five first order mutants and the modified lines are given in Table 4.33. Calculate the mutation score of each test suite and compare their effectiveness. Also, add any additional test case, if required.

Table 4.31. Test suite A

Test case	a	b	c	Expected output
1.	30	40	90	Invalid triangle
2.	30	20	40	Obtuse angled triangle
3.	50	40	60	Acute angled triangle
4.	30	40	50	Right angled triangle
5.	-1	50	40	Input values are out of range
6.	50	150	90	Input values are out of range
7.	50	40	-1	Input values are out of range

Table 4.32. Test suite B

Test case	a	b	c	Expected output
1.	40	90	20	Invalid triangle
2.	40	30	60	Obtuse angled triangle
3.	40	50	60	Acute angled triangle
4.	30	40	50	Right angled triangle
5.	-1	50	40	Input values are out of range
6.	30	101	90	Input values are out of range
7.	30	90	0	Input values are out of range

Table 4.33. Mutated lines

Mutant No.	Line no.	Original line	Modified Line
M_1	13	if(a>0&&a<=100&&b>0&&b<=100&&c>0&&c<=100) {	if(a>0||a<=100&&b>0&&b<=100&&c>0&&c<=100) {
M_2	14	if((a+b)>c&&(b+c)>a&&(c+a)>b) {	if((a+b)>c&&(b+c)>a&&(b+a)>b) {
M_3	21	if(valid==1) {	if(valid>1) {
M_4	23	a2=(b*b+c*c)/(a*a);	a2=(b*b+c*c)*(a*a);
M_5	25	if(a1<1||a2<1||a3<1) {	if(a1>1||a2<1||a3<1) {

Solution:

(a) Test cases for Test Suite A

The actual outputs of mutants M_1-M_5 on test suite A are shown in Tables 4.34-4.38.

Table 4.34. Actual output of M_1(A)

Test case	a	b	c	Expected output	Actual output
1.	30	40	90	Invalid triangle	Invalid triangle
2.	30	20	40	Obtuse angled triangle	Obtuse angled triangle
3.	50	40	60	Acute angled triangle	Acute angled triangle
4.	30	40	50	Right angled triangle	Right angled triangle
5.	–1	50	40	Input values are out of range	Invalid triangle
6.	50	150	90	Input values are out of range	Invalid triangle
7.	50	40	–1	Input values are out of range	Invalid triangle

Table 4.35. Actual output of M_2(A)

Test case	a	b	c	Expected output	Actual output
1.	30	40	90	Invalid triangle	Invalid triangle
2.	30	20	40	Obtuse angled triangle	Obtuse angled triangle
3.	50	40	60	Acute angled triangle	Acute angled triangle
4.	30	40	50	Right angled triangle	Right angled triangle
5.	–1	50	40	Input values are out of range	Input values are out of range
6.	50	150	90	Input values are out of range	Input values are out of range
7.	50	40	–1	Input values are out of range	Input values are out of range

Table 4.36. Actual output of M_3(A)

Test case	a	b	c	Expected output	Actual output
1.	30	40	90	Invalid triangle	Invalid triangle
2.	30	20	40	Obtuse angled triangle	Input values are out of range
3.	50	40	60	Acute angled triangle	Input values are out of range
4.	30	40	50	Right angled triangle	Input values are out of range
5.	–1	50	40	Input values are out of range	Input values are out of range
6.	50	150	90	Input values are out of range	Input values are out of range
7.	50	40	–1	Input values are out of range	Input values are out of range

Table 4.37. Actual output of M_4(A)

Test case	a	b	c	Expected output	Actual output
1.	30	40	90	Invalid triangle	Invalid triangle
2.	30	20	40	Obtuse angled triangle	Obtuse angled triangle
3.	50	40	60	Acute angled triangle	Acute angled triangle
4.	30	40	50	Right angled triangle	Right angled triangle
5.	–1	50	40	Input values are out of range	Input values are out of range
6.	50	150	90	Input values are out of range	Input values are out of range
7.	50	40	–1	Input values are out of range	Input values are out of range

Table 4.38. Actual output of M_5(A)

Test case	a	b	c	Expected output	Actual output
1.	30	40	90	Invalid triangle	Invalid triangle
2.	30	20	40	Obtuse angled triangle	Acute angled triangle
3.	50	40	60	Acute angled triangle	Obtuse angled triangle
4.	30	40	50	Right angled triangle	Right angled triangle
5.	–1	50	40	Input values are out of range	Input values are out of range
6.	50	150	90	Input values are out of range	Input values are out of range
7.	50	40	–1	Input values are out of range	Input values are out of range

Two mutants are M_2 and M_4 are live. Thus, the mutation score using test suite A is 0.6.

$$\text{Mutation Score} = \frac{\text{Number of mutants killed}}{\text{Total number of mutants}}$$
$$= \frac{3}{5}$$
$$= 0.6$$

(b) Test cases for Test Suite B

The actual outputs of mutants M_1-M_5 on test suite B are shown in Tables 4.39-4.43.

Table 4.39. Actual output of M_1(B)

Test case	a	b	c	Expected output	Actual output
1.	40	90	20	Invalid triangle	Invalid triangle
2.	40	30	60	Obtuse angled triangle	Obtuse angled triangle
3.	40	50	60	Acute angled triangle	Acute angled triangle
4.	30	40	50	Right angled triangle	Right angled triangle
5.	–1	50	40	Input values are out of range	Invalid triangle
6.	30	101	90	Input values are out of range	Obtuse angled triangle
7.	30	90	0	Input values are out of range	Invalid triangle

Table 4.40. Actual output of M_2(B)

Test case	a	b	c	Expected output	Actual output
1.	40	90	20	Invalid triangle	Obtuse angled triangle
2.	40	30	60	Obtuse angled triangle	Obtuse angled triangle
3.	40	50	60	Acute angled triangle	Acute angled triangle
4.	30	40	50	Right angled triangle	Right angled triangle
5.	–1	50	40	Input values are out of range	Input values are out of range
6.	30	101	90	Input values are out of range	Input values are out of range
7.	30	90	0	Input values are out of range	Input values are out of range

Table 4.41. Actual output of M_3(B)

Test case	a	b	c	Expected output	Actual output
1.	40	90	20	Invalid triangle	Invalid triangle
2.	40	30	60	Obtuse angled triangle	Input values are out of range
3.	40	50	60	Acute angled triangle	Input values are out of range
4.	30	40	50	Right angled triangle	Input values are out of range
5.	–1	50	40	Input values are out of range	Input values are out of range
6.	30	101	90	Input values are out of range	Input values are out of range
7.	30	90	0	Input values are out of range	Input values are out of range

Table 4.42. Actual output of M_4(B)

Test case	a	b	c	Expected output	Actual output
1.	40	90	20	Invalid triangle	Invalid triangle
2.	40	30	60	Obtuse angled triangle	Obtuse angled triangle
3.	40	50	60	Acute angled triangle	Acute angled triangle
4.	30	40	50	Right angled triangle	Right angled triangle
5.	–1	50	40	Input values are out of range	Input values are out of range
6.	30	101	90	Input values are out of range	Input values are out of range
7.	30	90	0	Input values are out of range	Input values are out of range

Table 4.43. Actual output of M_5(B)

Test case	a	b	c	Expected output	Actual output
1.	40	90	20	Invalid triangle	Invalid triangle
2.	40	30	60	Obtuse angled triangle	Acute angled triangle
3.	40	50	60	Acute angled triangle	Obtuse angled triangle
4.	30	40	50	Right angled triangle	Right angled triangle
5.	–1	50	40	Input values are out of range	Input values are out of range
6.	30	101	90	Input values are out of range	Input values are out of range
7.	30	90	0	Input values are out of range	Input values are out of range

$$\text{Mutation Score} = \frac{\text{Number of mutants killed}}{\text{Total number of mutants}}$$
$$= \frac{4}{5}$$
$$= 0.8$$

The mutation score of Test suite B is higher as compared to the mutation score of test suite A, hence test suite B is more effective in comparison to test suite A. In order to kill the live mutant (M_4), an additional test case should be added to test suite B as shown in Table 4.44.

Table 4.44. Additional test case

Test case	a	b	c	Expected output
8.	40	30	20	Obtuse angled triangle

The revised test suite B is given in Table 4.45.

Table 4.45. Revised test suite B

Test case	a	b	c	Expected output
1.	40	90	20	Invalid triangle
2.	40	30	60	Obtuse angled triangle
3.	40	50	60	Acute angled triangle
4.	30	40	50	Right angled triangle
5.	-1	50	40	Input values are out of range
6.	30	101	90	Input values are out of range
7.	30	90	0	Input values are out of range
8.	40	30	20	Obtuse angled triangle

MULTIPLE CHOICE QUESTIONS

Note: ***Select the most appropriate answer for the following questions.***

4.1 Which is not a structural testing technique?
- (a) Mutation testing
- (b) Data flow testing
- (c) Slice based testing
- (d) Decision table based testing

4.2 Which is a structural testing technique?
- (a) Data flow testing
- (b) Control flow testing
- (c) Mutation testing
- (d) All of the above

4.3 Data flow testing is related to:
- (a) ER diagrams
- (b) Data flow diagrams
- (c) Data dictionaries
- (d) None of the above

4.4 Mutation testing is related to:
- (a) Fault seeding
- (b) Fault severity
- (c) Fault impact analysis
- (d) None of the above

4.5 Mutation score does not indicate anything about:
- (a) Size of a test suite
- (b) Effectiveness of a test suite
- (c) Performance of a test suite
- (d) Usefulness of a test suite

7

Selection, Minimization and Prioritization of Test Cases for Regression Testing

Software maintenance is becoming important and expensive day by day. Development of software may take a few years (2 to 4 years), but the same may have to be maintained for several years (10 to 15 years). Software maintenance accounts for as much as two-thirds of the cost of software production [BEIZ90].

Software inevitably changes, whatever well-written and designed initially it may be. There are many reasons for such changes:

(i) Some errors may have been discovered during the actual use of the software.
(ii) The user may have requested for additional functionality.
(iii) Software may have to be modified due to change in some external policies and principles. When European countries had decided to go for a single European currency, this change affected all banking system software.
(iv) Some restructuring work may have to be done to improve the efficiency and performance of the software.
(v) Software may have to be modified due to change in existing technologies.
(vi) Some obsolete capabilities may have to be deleted.

This list is endless but the message is loud and clear i.e. 'change is inevitable'. Hence, software always changes in order to address the above mentioned issues. This changed software is required to be re-tested in order to ensure that changes work correctly and these changes have not adversely affected other parts of the software. This is necessary because small changes in one part of the software program may have subtle undesired effects in other seemingly unrelated parts of the software.

7.1 WHAT IS REGRESSION TESTING?

When we develop software, we use development testing to obtain confidence in the correctness of the software. Development testing involves constructing a test plan that describes how we should test the software and then, designing and running a suite of test cases that satisfy the

requirements of the test plan. When we modify software, we typically re-test it. This process of re-testing is called regression testing.

Hence, regression testing is the process of re-testing the modified parts of the software and ensuring that no new errors have been introduced into previously tested source code due to these modifications. Therefore, regression testing tests both the modified source code and other parts of the source code that may be affected by the change. It serves several purposes like:

- Increases confidence in the correctness of the modified program.
- Locates errors in the modified program.
- Preserves the quality and reliability of the software.
- Ensures the software's continued operation.

We typically think of regression testing as a software maintenance activity; however, we also perform regression testing during the latter stage of software development. This latter stage starts after we have developed test plans and test suites and used them initially to test the software. During this stage of development, we fine-tune the source code and correct errors in it, hence these activities resemble maintenance activities. The comparison of development testing and regression testing is given in Table 7.1.

Table 7.1. Comparison of regression and development testing

S.No.	Development Testing	Regression Testing
1.	We write test cases.	We may use already available test cases.
2.	We want to test all portions of the source code.	We want to test only modified portion of the source code and the portion affected by the modifications.
3.	We do development testing just once in the lifetime of the software.	We may have to do regression testing many times in the lifetime of the software.
4.	We do development testing to obtain confidence about the correctness of the software.	We do regression testing to obtain confidence about the correctness of the modified portion of the software.
5.	Performed under the pressure of release date.	Performed in crisis situations, under greater time constraints.
6.	Separate allocation of budget and time.	Practically no time and generally no separate budget allocation.
7.	Focus is on the whole software with the objective of finding faults.	Focus is only on the modified portion and other affected portions with the objective of ensuring the correctness of the modifications.
8.	Time and effort consuming activity (40% to 70%).	Not much time and effort is consumed as compared to development testing.

7.1.1 Regression Testing Process

Regression testing is a very costly process and consumes a significant amount of resources. The question is "how to reduce this cost?" Whenever a failure is experienced, it is reported to the software team. The team may like to debug the source code to know the reason(s) for this

failure. After identification of the reason(s), the source code is modified and we generally do not expect the same failure again. In order to ensure this correctness, we re-test the source code with a focus on modified portion(s) of the source code and also on affected portion(s) of the source code due to modifications. We need test cases that target the modified and affected portions of the source code. We may write new test cases, which may be a 'time and effort consuming' activity. We neither have enough time nor reasonable resources to write new test cases for every failure. Another option is to use the existing test cases which were designed for development testing and some of them might have been used during development testing. The existing test suite may be useful and may reduce the cost of regression testing. As we all know, the size of the existing test suite may be very large and it may not be possible to execute all tests. The greatest challenge is to reduce the size of the existing test suite for a particular failure. The various steps are shown in Figure 7.1. Hence, test case selection for a failure is the main key for regression testing.

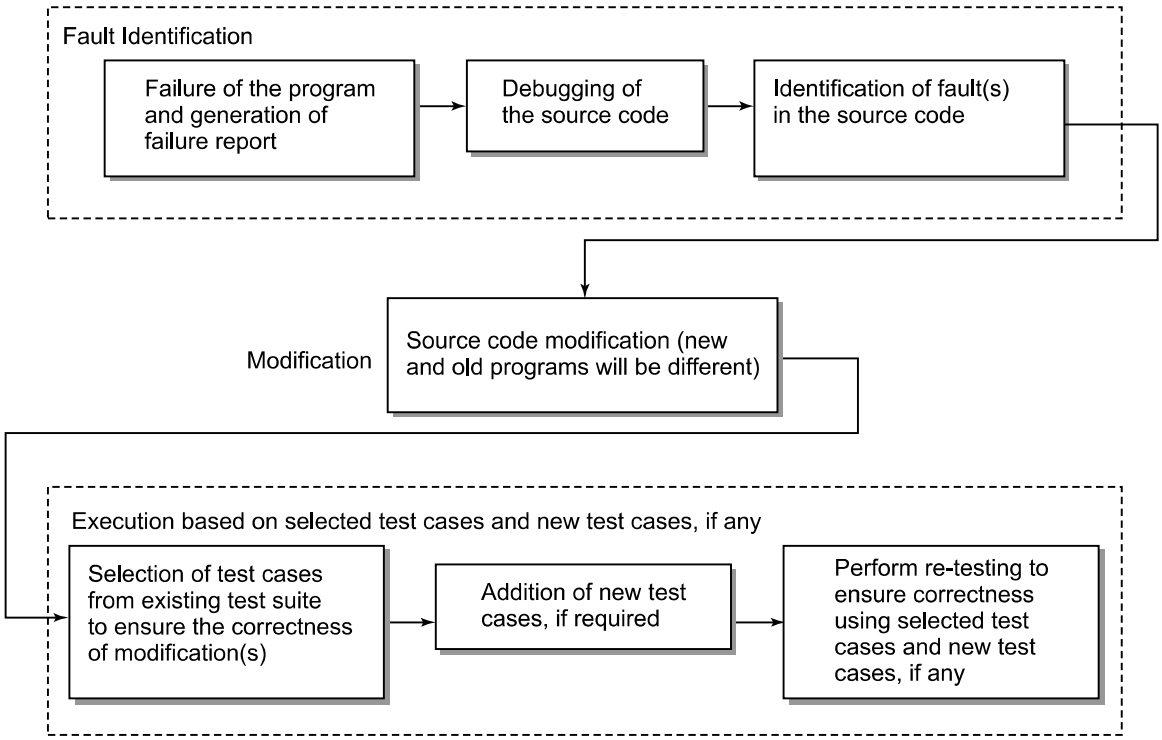


Figure 7.1. Steps of regression testing process

7.1.2 Selection of Test Cases

We want to use the existing test suite for regression testing. How should we select an appropriate number of test cases for a failure? The range is from "one test case" to "all test cases". A 'regression test cases' selection technique may help us to do this selection process. The effectiveness of the selection technique may decide the selection of the most appropriate test cases from the test suite. Many techniques have been developed for procedural and object oriented programming languages. Testing professionals are, however, reluctant to omit any test case from a test suite that might expose

a fault in the modified program. We consider a program given in Figure 7.2 along with its modified version where the modification is in line 6 (replacing operator '*' by '-'). A test suite is also given in Table 7.2.

```
1.  main( )                          1.  main ( )
2.  {                                2.  {
3.  int a, b, x, y, z;               3.  int a, b, x, y, z;
4.  scanf ("%d, %d", &a, &b);        4.  scanf ("%d, %d", &a, &b);
5.  x = a + b ;                      5.  x = a + b;
6.  y = a* b;                        6.  y = a – b;
7.  if (x ≥ y) {                     7.  if (x ≥ y) {
8.  z = x / y ;                      8.  z = x / y ;
9.  }                                9.  }
10. else {                           10. else {
11. z = x * y ;                      11. z = x * y ;
12. }                                12. }
13. printf ("z = %d \ n", z );       13. printf ("z = %d \ n", z);
14. }                                14. }
```

(a) Original program with fault in line 6. (b) Modified program with modification in line 6.

Figure 7.2. Program for printing value of *z*

Table 7.2. Test suite for program given in Figure 7.2

Set of Test Cases S. No.	Inputs		Execution History
	a	b	
1	2	1	1, 2, 3, 4, 5, 6, 7, 8, 9, 13, 14
2	1	1	1, 2, 3, 4, 5, 6, 7, 8, 9, 13, 14
3	3	2	1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14
4	3	3	1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14

In this case, the modified line is line number 6 where 'a*b' is replaced by 'a-b'. All four test cases of the test suite execute this modified line 6. We may decide to execute all four tests for the modified program. If we do so, test case 2 with inputs a = 1 and b = 1 will experience a 'divide by zero' problem, whereas others will not. However, we may like to reduce the number of test cases for the modified program. We may select all test cases which are executing the modified line. Here, line number 6 is modified. All four test cases are executing the modified line (line number 6) and hence are selected. There is no reduction in terms of the number of test cases. If we see the execution history, we find that test case 1 and test case 2 have the same execution history. Similarly, test case 3 and test case 4 have the same execution history. We choose any one test case of the same execution history to avoid repetition. For execution history 1 (i.e. 1, 2, 3, 4, 5, 6, 7, 8, 10, 11), if we select test case 1, the program will execute well, but if we select test case 2, the program will experience 'divide by zero' problem. If several test cases execute a particular modified line, and all of these test cases reach a particular

affected source code segment, minimization methods require selection of only one such test case, unless they select the others for coverage elsewhere. Therefore, either test case 1 or test case 2 may have to be selected. If we select test case 1, we miss the opportunity to detect the fault that test case 2 detects. Minimization techniques may omit some test cases that might expose fault(s) in the modified program. Hence, we should be very careful in the process of minimization of test cases and always try to use safe regression test selection technique (if at all it is possible). A safe regression test selection technique should select all test cases that can expose faults in the modified program.

7.2 REGRESSION TEST CASES SELECTION

Test suite design is an expensive process and its size can grow quite large. Most of the times, running an entire test suite is not possible as it requires a significant amount of time to run all test cases. Many techniques are available for the selection of test cases for the purpose of regression testing.

7.2.1 Select All Test Cases

This is the simplest technique where we do not want to take any risk. We want to run all test cases for any change in the program. This is the safest technique, without any risk. A program may fail many times and every time we will execute the entire test suite. This technique is practical only when the size of the test suite is small. For any reasonable or large sized test suite, it becomes impractical to execute all test cases.

7.2.2 Select Test Cases Randomly

We may select test cases randomly to reduce the size of the test suite. We decide how many test cases are required to be selected depending upon time and available resources. When we decide the number, the same number of test cases is selected randomly. If the number is large, we may get a good number of test cases for execution and testing may be of some use. But, if the number is small, testing may not be useful at all. In this technique, our assumption is that all test cases are equally good in their fault detection ability. However, in most of the situations, this assumption may not be true. We want to re-test the source code for the purpose of checking the correctness of the modified portion of the program. Many randomly selected test cases may not have any relationship with the modified portion of the program. However, random selection may be better than no regression testing at all.

7.2.3 Select Modification Traversing Test Cases

We select only those test cases that execute the modified portion of the program and the portion which is affected by the modification(s). Other test cases of the test suite are discarded.

Actually, we want to select all those test cases that reveal faults in the modified program. These test cases are known as fault revealing test cases. There is no effective technique by which we can find fault revealing test cases for the modified program. This is the best selection approach, which we want, but we do not have techniques for the same. Another lower objective

may be to select those test cases that reveal the difference in the output of the original program and the modified program. These test cases are known as modification revealing test cases. These test cases target that portion of the source code which makes the output of the original program and the modified program differ. Unfortunately, we do not have any effective technique to do this. Therefore, it is difficult to find fault revealing test cases and modification revealing test cases.

The reasonable objective is to select all those test cases that traverse the modified source code and the source code affected by modification(s). These test cases are known as modification traversing test cases. It is easy to develop techniques for modification traversing test cases and some are available too. Out of all modification traversing test cases, some may be modification revealing test cases and out of some modification revealing test cases, some may be fault revealing test cases. Many modification traversing techniques are available but their applications are limited due to the varied nature of software projects. Aditya Mathur has rightly mentioned that [MATH08]:

> *"The sophistication of techniques to select modification traversing tests requires automation. It is impractical to apply these techniques to large commercial systems unless a tool is available that incorporates at least one safe test minimization technique. Further, while test selection appears attractive from the test effort point of view, it might not be a practical technique when tests are dependent on each other in complex ways and that this dependency cannot be incorporated in the test selection tool".*

We may effectively implement any test case selection technique with the help of a testing tool. The modified source code and source code affected by modification(s) may have to be identified systematically and this selected area of the source code becomes the concern of test case selection. As the size of the source code increases, the complexity also increases, and need for an efficient technique also increases accordingly.

7.3 REDUCING THE NUMBER OF TEST CASES

Test case reduction is an essential activity and we may select those test cases that execute the modification(s) and the portion of the program that is affected by the modification(s). We may minimize the test suite or prioritize the test suite in order to execute the selected number of test cases.

7.3.1 Minimization of Test Cases

We select all those test cases that traverse the modified portion of the program and the portion that is affected by the modification(s). If we find the selected number very large, we may still reduce this using any test case minimization technique. These test case minimization techniques attempt to find redundant test cases. A redundant test case is one which achieves an objective which has already been achieved by another test case. The objective may be source code coverage, requirement coverage, variables coverage, branch coverage, specific lines of source

code coverage, etc. A minimization technique may further reduce the size of the selected test cases based on some criteria. We should always remember that any type of minimization is risky and may omit some fault revealing test cases.

7.3.2 Prioritization of Test Cases

We may indicate the order with which a test case may be addressed. This process is known as prioritization of test cases. A test case with the highest rank has the highest priority and the test case with the second highest rank has the second highest priority and as so on. Prioritization does not discard any test case. The efficiency of the regression testing is dependent upon the criteria of prioritization. There are two varieties of test case prioritization i.e. general test case prioritization and version specific test case prioritization. In general test case prioritization, for a given program with its test suite, we prioritize the test cases that will be useful over a succession of subsequent modified versions of the original program without any knowledge of modification(s). In the version specific test case prioritization, we prioritize the test cases, when the original program is changed to the modified program, with the knowledge of the changes that have been made in the original program.
Prioritization guidelines should address two fundamental issues like:

(i) What functions of the software must be tested?
(ii) What are the consequences if some functions are not tested?

Every reduction activity has an associated risk. All prioritization guidelines should be designed on the basis of risk analysis. All risky functions should be tested on higher priority. The risk analysis may be based on complexity, criticality, impact of failure, etc. The most important is the 'impact of failure' which may range from 'no impact' to 'loss of human life' and must be studied very carefully.

The simplest priority category scheme is to assign a priority code to every test case. The priority code may be based on the assumption that "test case of priority code 1 is more important than test case of priority code 2." We may have priority codes as follows:

Priority code 1	:	Essential test case
Priority code 2	:	Important test case
Priority code 3	:	Execute, if time permits
Priority code 4	:	Not important test case
Priority code 5	:	Redundant test case

There may be other ways for assigning priorities based on customer requirements or market conditions like:

Priority code 1	:	Important for the customer
Priority code 2	:	Required to increase customer satisfaction
Priority code 3	:	Help to increase market share of the product

We may design any priority category scheme, but a scheme based on technical considerations always improves the quality of the product and should always be encouraged.

7.4 RISK ANALYSIS

Unexpected behaviours of a software programme always carry huge information and most of the time they disturb every associate person. No one likes such unexpected behaviour and everyone prays that they never face these situations in their professional career. In practice, the situation is entirely different and developers do face such unexpected situations frequently and, moreover, work hard to find the solutions of the problems highlighted by these unexpected behaviours.

We may be able to minimize these situations, if we are able to minimize the risky areas of the software. Hence, risk analysis has become an important area and in most of the projects we are doing it to minimize the risk.

7.4.1 What is Risk?

Tomorrow's problems are today's risks. Therefore, a simple definition of risk is a problem that may cause some loss or threaten the success of the project, but, which has not happened yet. Risk is defined as the "probability of occurrence of an undesirable event and the impact of occurrence of that event." To understand whether an event is really risky needs an understanding of the potential consequences of the occurrences / non-occurrences of that event. Risks may delay and over-budget a project. Risky projects may also not meet specified quality levels. Hence, there are two things associated with risk as given below:

(i) Probability of occurrence of a problem (i.e. an event)
(ii) Impact of that problem

Risk analysis is a process of identifying the potential problems and then assigning a 'probability of occurrence of the problem' value and 'impact of that problem' value for each identified problem. Both of these values are assigned on a scale of 1 (low) to 10 (high). A factor 'risk exposure' is calculated for every problem which is the product of 'probability of occurrence of the problem' value and 'impact of that problem' value. The risks may be ranked on the basis of its risk exposure. A risk analysis table may be prepared as given in Table 7.3. These values may be calculated on the basis of historical data, past experience, intuition and criticality of the problem. We should not confuse with the mathematical scale of probability values which is from 0 to 1. Here, the scale of 1 to 10 is used for assigning values to both the components of the risk exposure.

Table 7.3. Risk analysis table

S. No.	Potential Problem	Probability of occurrence of problem	Impact of that Problem	Risk Exposure
1.				
2.				
3.				
4.				

The case study of 'University Registration System' given in chapter 5 is considered and its potential problems are identified. Risk exposure factor for every problem is calculated on the

basis of 'probability of occurrence of the problem' and 'impact of that problem'. The risk analysis is given in Table 7.4.

Table 7.4. Risk analysis table of 'University Registration System'

S. No.	Potential Problems	Probability of occurrence of problem	Impact of that Problem	Risk Exposure
1.	Issued password not available	2	3	6
2.	Wrong entry in students detail form	6	2	12
3.	Wrong entry in scheme detail form	3	3	9
4.	Printing mistake in registration card	2	2	4
5.	Unauthorised access	1	10	10
6.	Database corrupted	2	9	18
7.	Ambiguous documentation	8	1	8
8.	Lists not in proper format	3	2	6
9.	Issued login-id is not in specified format	2	1	2
10.	School not available in the database	2	4	8

The potential problems ranked by risk exposure are 6, 2, 5, 3, 7, 10, 1, 8, 4 and 9.

7.4.2 Risk Matrix

Risk matrix is used to capture identified problems, estimate their probability of occurrence with impact and rank the risks based on this information. We may use the risk matrix to assign thresholds that group the potential problems into priority categories. The risk matrix is shown in Figure 7.3 with four quadrants. Each quadrant represents a priority category.

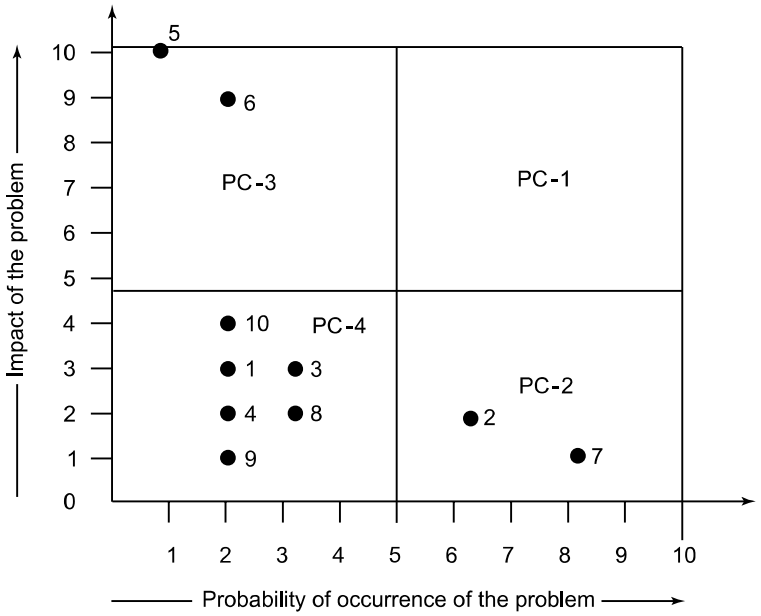


Figure 7.3. Threshold by quadrant

The priority category in defined as:

Priority category 1 (PC-1)	=	High probability value and high impact value
Priority category 2 (PC-2)	=	High probability value and low impact value
Priority category 3 (PC-3)	=	Low probability value and high impact value
Priority category 4 (PC-4)	=	Low probability value and low impact value

In this case, a risk with high probability value is given more importance than a problem with high impact value. We may change this and may decide to give more importance to high impact value over the high probability value and is shown in Figure 7.4. Hence, PC-2 and PC-3 will swap, but PC-1 and PC-4 will remain the same.

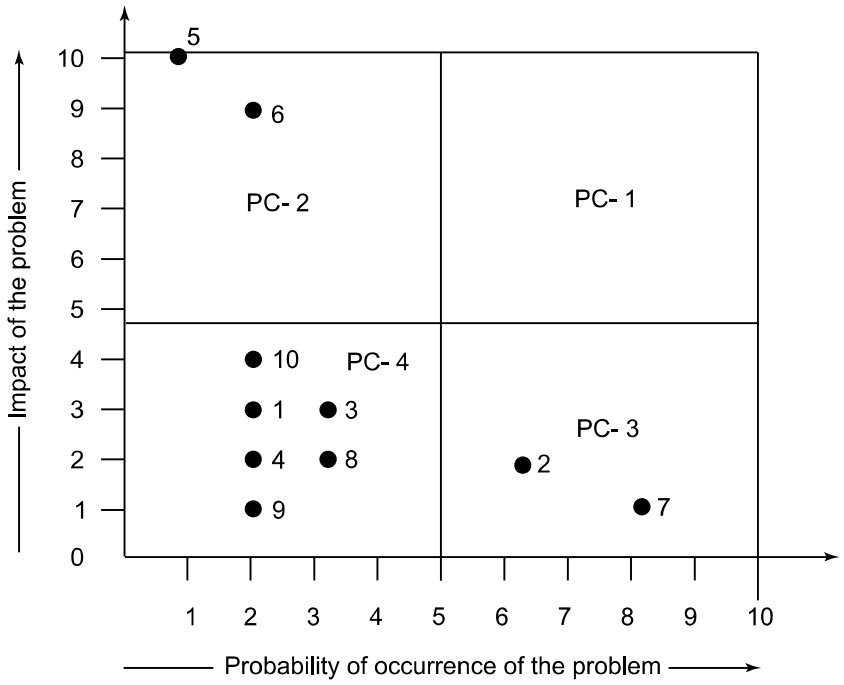


Figure 7.4. Alternative threshold by quadrant

There may be situations where we do not want to give importance to any value and assign equal importance. In this case, the diagonal band prioritization scheme may be more suitable as shown in Figure 7.5. This scheme is more appropriate in situations where we have difficulty in assigning importance to either 'probability of occurrence of the problem' value or 'impact of that problem' value.

We may also feel that high impact value must be given highest priority irrespective of the 'probability of occurrence' value. A high impact problem should be addressed first, irrespective of its probability of occurrence value. This prioritization scheme is given in Figure 7.6. Here, the highest priority (PC-1) is assigned to high impact value and for the other four quadrants; any prioritization scheme may be selected. We may also assign high priority to high 'probability of occurrence' values irrespective of the impact value as shown in Figure 7.7. This scheme may not be popular in practice. Generally, we are afraid of the impact of the problem. If the impact value is low, we are not much concerned. In the risk analysis table (see Table 7.4), ambiguous documentations (S. No. 7) have high 'probability of occurrence of problem' value (8), but

impact value is very low (1). Hence, these faults are not considered risky faults as compared to 'unauthorized access' (S. No. 5) where 'probability of occurrence' value is very low (1) and impact value is very high (10).

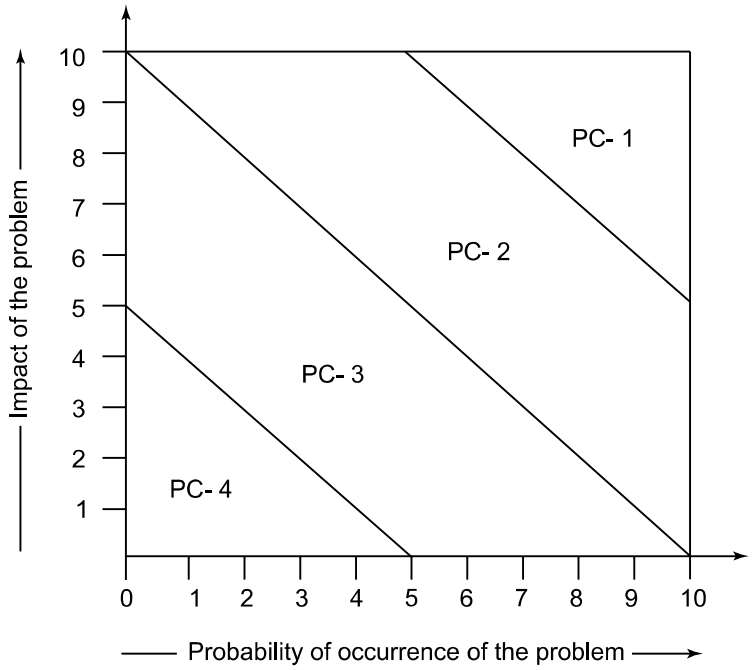


Figure 7.5. Threshold by diagonal quadrant

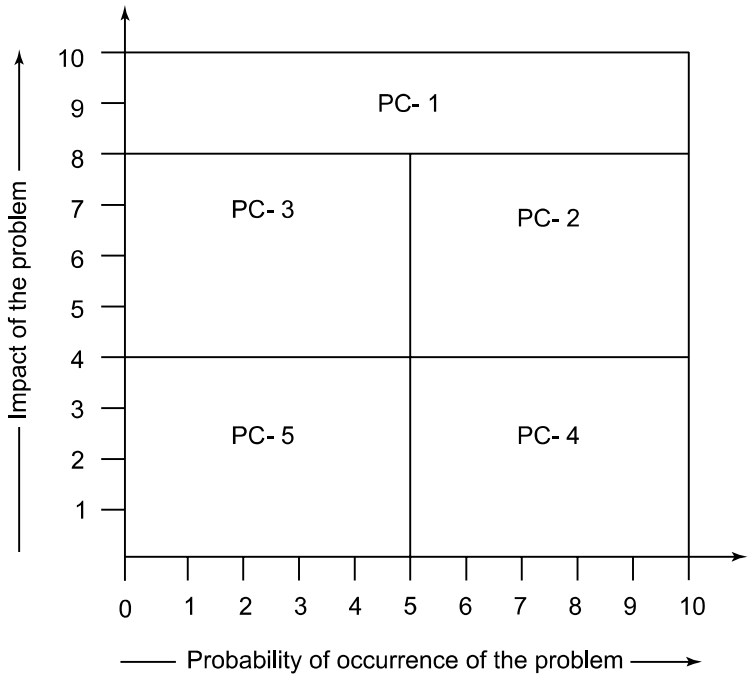


Figure 7.6. Threshold based on high 'Impact of Problem' value

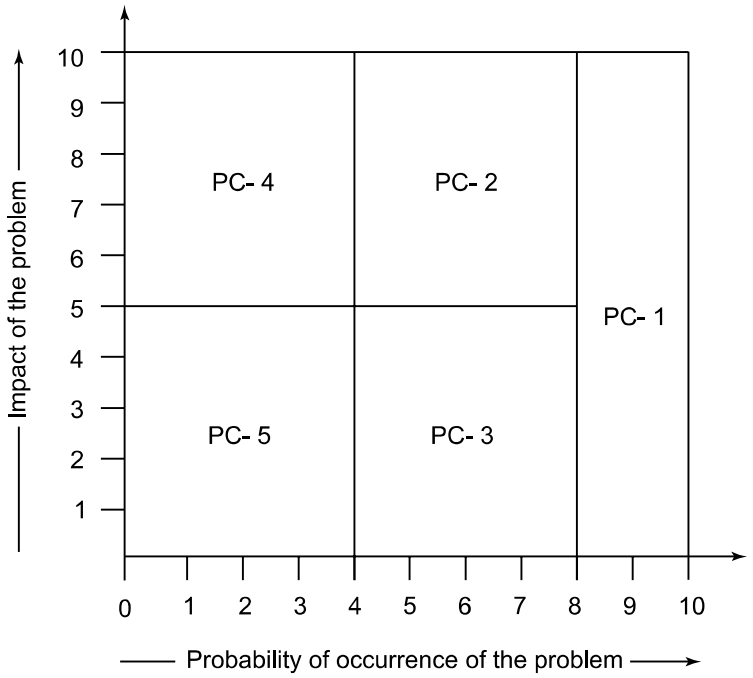


Figure 7.7. Threshold based on high 'probability of occurrence of problem' value

After the risks are ranked, the high priority risks are identified. These risks are required to be managed first and then other priority risks in descending order. These risks should be discussed in a team and proper action should be recommended to manage these risks. A risk matrix has become a powerful tool for designing prioritization schemes. Estimating the probability of occurrence of a problem may be difficult in practice. Fortunately, all that matters when using a risk matrix is the relative order of the probability estimates (which risks are more likely to occur) on the scale of 1 to 10. The impact of the problem may be critical, serious, moderate, minor or negligible. These two values are essential for risk exposure which is used to prioritize the risks.

7.5 CODE COVERAGE PRIORITIZATION TECHNIQUE

We consider a program P with its modified program P′ and its test suite T created to test P. When we modify P to P′, we would like to execute modified portion(s) of the source code and the portion(s) affected by the modification(s) to see the correctness of modification(s). We neither have time nor resources to execute all test cases of T. Our objective is to reduce the size of T to T′ using some selection criteria, which may help us to execute the modified portion of the source code and the portion(s) affected by modification(s).

A code coverage based technique [KAUR06, AGGA04] has been developed which is based on version specific test case prioritization and selects T′ from T which is a subset of T. The technique also prioritizes test cases of T′ and recommends use of high priority test cases first and then low priority test cases in descending order till time and resources are available or a reasonable level of confidence is achieved.

7.5.1 Test Cases Selection Criteria

The technique is based on version specific test case prioritization where information about changes in the program is known. Hence, prioritization is focused around the changes in the modified program. We may like to execute all modified lines of source code with a minimum number of selected test cases. This technique identifies those test cases that:

(i) Execute the modified lines of source code at least once
(ii) Execute the lines of source code after deletion of deleted lines from the execution history of the test case and are not redundant.

The technique uses two algorithms – one for 'modification' and the other for 'deletion'. The following information is available with us and has been used to design the technique:

(i) Program P with its modified program P′.
(ii) Test suite T with test cases t1, t2, t3,…..tn.
(iii) Execution history (number of lines of source code covered by a test case) of each test case of test suite T.
(iv) Line numbers of lines of source code covered by each test case are stored in a two dimensional array (t_{11}, t_{12}, t_{13},……t_{ij}).

7.5.2 Modification Algorithm

The 'modification' portion of the technique is used to minimize and prioritize test cases based on the modified lines of source code. The 'modification' algorithm uses the following information given in Table 7.5.

Table 7.5. Variables used by 'modification' algorithm

S. No.	Variable name	Description
1.	T1	It is a two dimensional array and is used to store line numbers of lines of source code covered by each test case.
2.	modloc	It is used to store the total number of modified lines of source code.
3.	mod_locode	It is a one-dimensional array and is used to store line numbers of modified lines of source code.
4.	nfound	It is a one-dimensional array and is used to store the number of lines of source code matched with modified lines of each test case.
5.	pos	It is a one-dimensional array and is used to set the position of each test case when nfound is sorted.
6.	candidate	It is a one-dimensional array. It sets the bit to 1 corresponding to the position of the test case to be removed.
7.	priority	It is a one-dimensional array and is used to set the priority of the selected test case.

The following steps have been followed in order to select and prioritize test cases from test suite T based on the modification in the program P.

Step I: Initialization of variables

Consider a hypothetical program of 60 lines of code with a test suite of 10 test cases. The execution history is given in Table 7.6. We assume that lines 1, 2, 5, 15, 35, 45, 55 are modified.

Table 7.6. Test cases with execution history

Test case Id	Execution history
T1	1, 2, 20, 30, 40, 50
T2	1, 3, 4, 21, 31, 41, 51
T3	5, 6, 7, 8, 22, 32, 42, 52
T4	6, 9, 10, 23, 24, 33, 43, 54
T5	5, 9, 11, 12, 13, 14, 15, 20, 29, 37, 38, 39
T6	15, 16, 17, 18, 19, 23, 24, 25, 34, 35, 36
T7	26, 27, 28, 40, 41, 44, 45, 46
T8	46, 47, 48, 49, 50, 53, 55
T9	55, 56, 57, 58, 59
T10	3, 4, 60

The first portion of the 'modification' algorithm is used to initialize and read values of variables T1, modloc and mod_locode.

First portion of the 'modification' algorithm

1. Repeat for i=1 to number of test cases
   (a) Repeat for j=1 to number of test cases
      (i) Initialize array T1[i][j] to zero
2. Repeat for i=1 to number of test cases
   (a) Repeat for j=1 to number of test cases
      (i) Store line numbers of line of source code covered by each test case.
3. Repeat for i=1 to number of modified lines of source code
   (a) Store line numbers of modified lines of source code in array mod_locode.

Step II: Selection and prioritization of test cases

The second portion of the algorithm counts the number of modified lines of source code covered by each test case (nfound).

Second portion of the 'modification' algorithm

2. Repeat for all true cases
   (a) Repeat for i=1 to number of test cases
      (i) Initialize array nfound[i] to zeroes
      (ii) Set pos[i] =i
   (b) Repeat for i=1 to number of test cases
      (i) Initialize l to zero

```
(ii)  Repeat for j=1 to length of the test case
            If candidate[i] ≠ 1 then
            Repeat for k=1 to modified lines of source code
                  If t1[i][j]=mod_locode[k] then
                        Increment nfound[i] by one
                         Increment l by one
```

The status of test cases covering modified lines of source code is given in Table 7.7.

Table 7.7. Test cases with number of matches found

Test Cases	Numbers of lines matched	Number of Matches (nfound)
T1	1, 2	2
T2	1	1
T3	5	1
T4	-	0
T5	5, 15	2
T6	15, 35	2
T7	45	1
T8	55	1
T9	55	1
T10	-	0

Consider the third portion of 'modification' algorithm. In this portion, we sort the nfound array and select the test case with the highest value of nfound as a candidate for selection. The test cases are arranged in increasing order of priority.

Third portion of the '"modification' algorithm

```
(c)  Initialize l to zero
(d)  Repeat for i=0 to number of test cases
     (i)  Repeat for j=1 to number of test cases
                If nfound[i]>0 then
                      t=nfound[i]
                      nfound[i]=nfound[j]
                      nfound[j]=t
                      t=pos[i]
                      pos[i]=pos[j]
                      pos[j]=t
(e)  Repeat for i=1 to number of test cases
     (i)  If nfound[i]=1 then
                Increment count
(f)  If count = 0 then
     (i)  Goto end of the algorithm
(g)  Initialize candidate[pos[0]] = 1
(h)  Initialize priority[pos[0]]= m+1
```

The test cases with less value have higher priority than the test cases with higher value. Hence, the test cases are sorted on the basis of number of modified lines covered as shown in Table 7.8.

Table 7.8. Test cases in decreasing order of number of modified lines covered

Test Cases	Numbers of lines matched	Number of Matches (nfound)	Candidate	Priority
T1	1, 2	2	1	1
T5	5, 15	2	0	0
T6	15, 35	2	0	0
T2	1	1	0	0
T3	5	1	0	0
T7	45	1	0	0
T8	55	1	0	0
T9	55	1	0	0
T4	-	0	0	0
T10	-	0	0	0

The test case with candidate=1 is selected in each iteration. In the fourth portion of the algorithm, the modified lines of source code included in the selected test case are removed from the mod_locode array. This process continues until there are no remaining modified lines of source code covered by any test case.

Fourth portion of the 'modification' algorithm

```
(a) Repeat for i=1 to length of selected test cases
    (i) Repeat for j=1 to modified lines of source code
                If t1[pos[0]][i] = mod[j] then
                    mod[j] = 0
```

Since test case T1 is selected and it covers 1 and 2 lines of source code, these lines will be removed from the mod_locode array.

mod_locode = [1, 2, 5, 15, 35, 45, 55] - [1, 2] = [5, 15, 35, 45, 55]

The remaining iterations of the 'modification' algorithm are shown in tables 7.9-7.12.

Table 7.9. Test cases in descending order of number of matches found (iteration 2)

Test Cases	Number of matches (nfound)	Matches found	Candidate	Priority
T5	2	5, 15	1	2
T6	2	15, 35	0	0
T3	1	5	0	0
T7	1	45	0	0
T8	1	55	0	0
T9	1	55	0	0
T2	0	-	0	0
T4	0	-	0	0
T10	0	-	0	0

mod_locode = [5, 15, 35, 45, 55] – [5, 15] = [35, 45, 55]

Table 7.10. Test cases in descending order of number of matches found (iteration 3)

Test Cases	Number of matches (nfound)	Matches found	Candidate	Priority
T6	1	35	1	3
T7	1	45	0	0
T8	1	55	0	0
T9	1	55	0	0
T2	0	-	0	0
T3	0	-	0	0
T4	0	-	0	0
T10	0	-	0	0

mod_locode = [35, 45, 55] – [35] = [45, 55]

Table 7.11. Test cases in descending order of number of matches found (iteration 4)

Test Cases	Number of matches (nfound)	Matches found	Candidate	Priority
T7	1	45	1	4
T8	1	55	0	0
T9	1	55	0	0
T2	0	-	0	0
T3	0	-	0	0
T4	0	-	0	0
T10	0	-	0	0

mod_locode = [45, 55] – [45] = [55]

Table 7.12. Test cases in descending order of number of matches found (iteration 5)

Test Cases	Number of matches (nfound)	Matches found	Candidate	Priority
T8	1	55	1	5
T9	1	55	0	0
T2	0	-	0	0
T3	0	-	0	0
T4	0	-	0	0
T10	0	-	0	0

mod_locode = [55] – [55] = [Nil]

Hence test cases T1, T5, T6, T7 and T8 need to be executed on the basis of their corresponding priority. Out of ten test cases, we need to run only 5 test cases for 100% code coverage of modified lines of source code. This is 50% reduction of test cases.

7.5.3 Deletion Algorithm

The 'deletion' portion of the technique is used to (i) update the execution history of test cases by removing the deleted lines of source code (ii) identify and remove those test cases that cover only those lines which are covered by other test cases of the program. The information used in the algorithm is given in Table 7.13.

Table 7.13. Variables used by 'deletion' algorithm

S. No.	Variable	Description
1.	T1	It is a two-dimensional array. It keeps the number of lines of source code covered by each test case i.
2.	deloc	It is used to store the total number of lines of source code deleted.
3.	del_locode	It is a one-dimensional array and is used to store line numbers of deleted lines of source code.
4.	count	It is a two-dimensional array. It sets the position corresponding to every matched line of source code of each test case to 1.
5.	match	It is a one-dimensional array. It stores the total count of the number of 1's in the count array for each test case.
6.	deleted	It is a one-dimensional array. It keeps the record of redundant test cases. If the value corresponding to test case i is 1 in deleted array, then that test case is redundant and should be removed.

Step I: Initialization of variables

We consider a hypothetical program of 20 lines with a test suite of 5 test cases. The execution history is given in Table 7.14.

Table 7.14. Test cases with execution history

Test case Id	Execution history
T1	1, 5, 7, 15, 20
T2	2, 3, 4, 5, 8, 16, 20
T3	6, 8, 9, 10, 11, 12, 13, 14, 17, 18
T4	1, 2, 5, 8, 17, 19
T5	1, 2, 6, 8, 9, 13

We assume that line numbers 6, 13, 17 and 20 are modified, and line numbers 4, 7 and 15 are deleted from the source code. The information is stored as:

delloc = 3
del_locode = [4, 7, 15]
modloc = 4
mod_locode = [6, 13, 17, 20]

First portion of the "deletion" algorithm

```
1. Repeat for i=1 to number of test cases
   (a) Repeat for j=1 to length of test case i
       (i) Repeat for l to number of deleted lines of source code
           If T1[i][j]=del_locode then
              Repeat for k=j to length of test case i
                T1[i][k]=T1[i][k+1]
              Initialize T1[i][k] to zero
              Decrement c[i] by one
```

After deleting line numbers 4, 7, and 15, the modified execution history is given in Table 7.15.

Table 7.15. Modified execution history after deleting line numbers 4, 7 and 15

Test case Id	Execution history
T1	1, 5, 20
T2	2, 3, 5, 8, 16, 20
T3	6, 8, 9, 10, 11, 12, 13, 14, 17, 18
T4	1, 2, 5, 8, 17, 19
T5	1, 2, 6, 8, 9, 13

Step II: Identification of redundant test cases

We want to find redundant test cases. A test case is a redundant test case, if it covers only those lines which are covered by other test cases of the program. This situation may arise due to deletion of a few lines of the program.

Consider the second portion of the 'deletion' algorithm. In this portion, the test case array is initialized with line numbers of lines of source code covered by each test case.

Second portion of the 'deletion' algorithm

```
2. Repeat for i=1 to number of test cases
   (a) Repeat for j=1 to number of test cases
       (i) Initialize array t1[i][j] to zero
       (ii) Initialize array count[i][j] to zero
3. Repeat for i=1 to number of test cases
   (a) Initialize deleted[i] and match [i] to zero
4. Repeat for i=1 to number of test cases
   (a) Initialize c[i] to number of line numbers in each test case i
   (b) Repeat for j=1 to c[i]
   (c) Initialize t1[i][j] to line numbers of line of source code covered by each test case
```

The third portion of the algorithm compares lines covered by each test case with lines covered by other test cases. A two-dimensional array count is used to keep the record of line number matched in each test case. If all the lines covered by a test case are being covered by some other test case, then that test case is redundant and should not be selected for execution.

Third portion of the 'deletion' algorithm

```
5. Repeat for i=1 to number of test cases
   (a) Repeat for j=1 to number of test cases
       (i) If i≠j and deleted[j]≠1 then
                    Repeat for k=1 to until t1[i][k]≠0
                      Repeat for l=1 until t1[j][l]≠0
                    If t1[i][k]=t1[j][l] then
                          Initialize count [i][k]=1
   (b) Repeat for m=1 to c[i]
       (i) If count[i][m]=1 then
                    Increment match[i] with 1
   (c) If match[i]=c[i] then
       (i) Initialize deleted[i] to 1
6. Repeat for i=1 to number of test cases
   (a) If deleted[i] =1 then
       Remove test case i (as it is a redundant test case)
```

On comparing all values in each test case with all values of other test cases, we found that test case 1 and test case 5 are redundant test cases. These two test cases do not cover any line which is not covered by other test cases as shown in Table 7.16.

Table 7.16. Redundant test cases

Test Case	Line Number of LOC	Found In Test Case	Redundant Y/N
T1	1	T4	Y
	5	T2	Y
	20	T2	Y
T5	6	T3	Y
	8	T3	Y
	9	T3	Y
	1	T4	Y
	2	T2	Y
	13	T3	Y

The remaining test cases are = [T2, T3, T4] and are given in Table 7.17.

Table 7.17. Modified table after removing T1 and T5

Test case Id	Execution history
T2	2, 3, 5, 8, 16, 20
T3	6, 8, 9, 10, 11, 12, 13, 14, 17, 18
T4	1, 2, 5, 8, 17, 19

Now we will minimize and prioritize test cases using 'modification' algorithm given in section 7.5.2. The status of test cases covering the modified lines is given in Table 7.18.

Table 7.18. Test cases with modified lines

Test Cases	Number of lines matched (found)	Number of matches (nfound)
T2	20	1
T3	6, 13, 17	3
T4	17	1

Test cases are sorted on the basis of number of modified lines covered as shown in tables 7.19-7.20.

Table 7.19. Test cases in descending order of number of modified lines covered

Test Cases	Number of matches (nfound)	Numbers of lines matched	Candidate	Priority
T3	3	6, 13, 17	1	1
T2	1	20	0	0
T4	1	17	0	0

mod_locode = [6, 13, 17, 20] – [6, 13, 17] = [20]

Table 7.20. Test cases in descending order of number of modified lines covered (iteration 2)

Test Cases	Number of matches (nfound)	Numbers of lines matched	Candidate	Priority
T2	1	20	1	2
T4	0	-	0	0

Hence, test cases T2 and T3 are needed to be executed and redundant test cases are T1 and T5.

Out of the five test cases, we need to run only 2 test cases for 100% code coverage of modified code coverage. This is a 60% reduction. If we run only those test cases that cover any modified lines, then T2, T3 and T4 are selected. This technique not only selects test cases, but also prioritizes test cases.

Example 7.1: Consider the algorithm for deletion and modification of lines of source code in test cases. Write a program in C to implement, minimize and prioritize test cases using the above technique.

Solution:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int t1[50][50]={0};
int count[50][50]={0};
int deleted[50],deloc,del_loc[50],k,c[50],l,num,m,n,match[50],i,j;
clrscr();
for(i=0;i<50;i++){
deleted[i]=0;
match[i]=0;
}

printf("Enter the number of test cases\n");
scanf("%d",&num);
for(i=0;i<num;i++){
      printf("Enter the length of test case %d\n",i+1);
      scanf("%d",&c[i]);
      printf("Enter the values of test case\n");
      for(j=0;j<c[i];j++){
            scanf("%d",&t1[i][j]);
            }
  }

printf("\nEnter the deleted lines of code:");
scanf("%d",&deloc);

for(i=0;i<deloc;i++)
      {
            scanf("%d",&del_loc[i]);
      }
for(i=0;i<num;i++){
      for(j=0;j<c[i];j++){
            for(l=0;l<deloc;l++){
            if(t1[i][j]==del_loc[l]){
                  for(k=j;k<c[i];k++){
                  t1[i][k]=t1[i][k+1];
                  }
                  t1[i][k]=0;
                  c[i]--;
            }
      }
```

```
        }
}
printf("Test case execution history after deletion:\n");
for(i=0;i<num;i++){
        printf("T%d\t",i+1);
        for(j=0;j<c[i];j++){
        printf("%d ",t1[i][j]);
        }
printf("\n");
}
for(i=0;i<num;i++){
        for(j=0;j<num;j++){
                if(i!=j&&deleted[j]!=1){
                for(k=0;t1[i][k]!=0;k++){
                for(l=0;t1[j][l]!=0;l++){
                if(t1[i][k]==t1[j][l])
                        count[i][k]=1;
        }
        }
        }
}
for(m=0;m<c[i];m++)
        if(count[i][m]==1)
                match[i]++;
if(match[i]==c[i])
        deleted[i]=1;
}
for(i=0;i<num;i++)
if(deleted[i]==1)
printf("Remove Test case %d\n",i+1);
getch();
}
```

OUTPUT

```
Enter the number of test cases
5
Enter the length of test case 1
5
Enter the values of test case
1 5 7 15 20
Enter the length of test case 2
7
Enter the values of test case
2 3 4 5 8 16 20
Enter the length of test case 3
10
```

```
Enter the values of test case
6 8 9 10 11 12 13 14 17 18
Enter the length of test case 4
6
Enter the values of test case
1 2 5 8 17 19
Enter the length of test case 5
6
Enter the values of test case
1 2 6 8 9 13
Enter the deleted lines of code:3
4 7 15
Test case execution history after deletion:
T1   1 5 20
T2   2 3 5 8 16 20
T3   6 8 9 10 11 12 13 14 17 18
T4   1 2 5 8 17 19
T5   1 2 6 8 9 13
Remove Test case 1
Remove Test case 5
```

```
/*Program for test case selection for modified lines using regression test case selection algorithm*/

#include<stdio.h>
#include<conio.h>

void main()
{
int t1[50][50];
int count=0;
int candidate[50]={0},priority[50]={0},m=0,pos[50],found[50][50],k,t,c[50],l,num,n,index[50],i,j,modnu
m,nfound[50],mod[50];
clrscr();
printf("Enter the number of test cases:");
scanf("%d",&num);
for(i=0;i<num;i++){
      printf("\nEnter the length of test case%d:",i+1);
      scanf("%d",&c[i]);
      }

for(i=0;i<50;i++)
      for(j=0;j<50;j++)
            found[i][j]=0;
for(i=0;i<num;i++)
      for(j=0;j<c[i];j++){
            t1[i][j]=0;
      }
for(i=0;i<num;i++){
      printf("Enter the values of test case %d\n",i+1);
      for(j=0;j<c[i];j++){
```

```
            scanf("%d",&t1[i][j]);
            }
            pos[i]=i;
  }
printf("\nEnter number of modified lines of code:");
scanf("%d",&modnum);
printf("Enter the lines of code modified:");
for(i=0;i<modnum;i++)
      scanf("%d",&mod[i]);
while(1)
{
count=0;
for(i=0;i<num;i++) {
      nfound[i]=0;
      pos[i]=i;
      }
for(i=0;i<num;i++){
l=0;
      for(j=0;j<c[i];j++){
      if(candidate[i]!=1){
      for(k=0;k<modnum;k++) {
            if(t1[i][j]==mod[k]){
                  nfound[i]++;
                  found[i][l]=mod[k];
                  l++;
                  }
      }
      }
}
}

l=0;
for(i=0;i<num;i++)
for(j=0;j<num-1;j++)
      if(nfound[i]>nfound[j]){
            t=nfound[i];
            nfound[i]=nfound[j];
            nfound[j]=t;
            t=pos[i];
            pos[i]=pos[j];
            pos[j]=t;
        }

for(i=0;i<num;i++)
      if(nfound[i]>0)
            count++;
if(count==0)
```

```
        break;

candidate[pos[0]]=1;
priority[pos[0]]=++m;

printf("\nTestcase\tMatches");
for(i=0;i<num;i++) {
        printf("\n%d\t\t%d",pos[i]+1,nfound[i]);
        getch();
        }

for(i=0;i<c[pos[0]];i++)
        for(j=0;j<modnum;j++)
                if(t1[pos[0]][i]==mod[j]){
                        mod[j]=0;
                }

printf("\nModified Array:");
for(i=0;i<modnum;i++){
        if(mod[i]==0){
                continue;
                }
        else {
                printf("%d\t",mod[i]);
                }
        }
}

count=0;
printf("\nTest case selected.....\n");
for(i=0;i<num;i++)
        if(candidate[i]==1){
                printf("\nT%d\t Priority%d\n ",i+1,priority[i]);
                count++;
                }
if(count==0){
        printf("\nNone");
        }
getch();
}
```

OUTPUT

```
Enter the number of test cases:10
Enter the length of test case1:6
Enter the length of test case2:7
Enter the length of test case3:8
Enter the length of test case4:8
```

```
Enter the length of test case5:12
Enter the length of test case6:11
Enter the length of test case7:8
Enter the length of test case8:7
Enter the length of test case9:5
Enter the length of test case10:3
Enter the values of test case 1
1 2 20 30 40 50
Enter the values of test case 2
1 3 4 21 31 41 51
Enter the values of test case 3
5 6 7 8 22 32 42 52
Enter the values of test case 4
6 9 10 23 24 33 43 54
Enter the values of test case 5
5 9 11 12 13 14 15 20 29 37 38 39
Enter the values of test case 6
15 16 17 18 19 23 24 25 34 35 36
Enter the values of test case 7
26 27 28 40 41 44 45 46
Enter the values of test case 8
46 47 48 49 50 53 55
Enter the values of test case 9
55 56 57 58 59
Enter the values of test case 10
3 4 60
Enter the number of modified lines of code:7
Enter the lines of code modified:1 2 5 15 35 45 55
Test case    Matches
1          2
5          2
6          2
3          1
2          1
7          1
8          1
9          1
4          0
10         0
Modified Array:5    15   35   45   55
Test case    Matches
5          2
6          2
3          1
7          1
8          1
9          1
```

```
4          0
2          0
1          0
10         0
Modified Array:35   45   55
Test case    Matches
6          1
7          1
8          1
9          1
5          0
1          0
2          0
3          0
4          0
10         0
Modified Array:45   55
Test case    Matches
7          1
8          1
9          1
4          0
5          0
6          0
1          0
2          0
3          0
10         0
Modified Array:55
Test case    Matches
8          1
9          1
3          0
4          0
5          0
6          0
7          0
1          0
2          0
10         0
Modified Array:
Test case selected.....
T1   Priority1
T5   Priority2
T6   Priority3
T7   Priority4
T8   Priority5
```

8

Software Testing Activities

We start testing activities from the first phase of the software development life cycle. We may generate test cases from the SRS and SDD documents and use them during system and acceptance testing. Hence, development and testing activities are carried out simultaneously in order to produce good quality maintainable software in time and within budget. We may carry out testing at many levels and may also take help of a software testing tool. Whenever we experience a failure, we debug the source code to find reasons for such a failure. Finding the reasons for a failure is a very significant testing activity and consumes a huge amount of resources and may also delay the release of the software.

8.1 LEVELS OF TESTING

Software testing is generally carried out at different levels. There are four such levels namely unit testing, integration testing, system testing and acceptance testing as shown in Figure 8.1. The first three levels of testing activities are done by the testers and the last level of testing (acceptance) is done by the customer(s)/user(s). Each level has specific testing objectives. For example, at the unit testing level, independent units are tested using functional and/or structural testing techniques. At the integration testing level, two or more units are combined and testing is carried out to test the integration related issues of various units. At the system testing level, the system is tested as a whole and primarily functional testing techniques are used to test the system. Non-functional requirements like performance, reliability, usability, testability, etc. are also tested at this level. Load/stress testing is also performed at this level. The last level i.e. acceptance testing, is done by the customer(s)/user(s) for the purpose of accepting the final product.

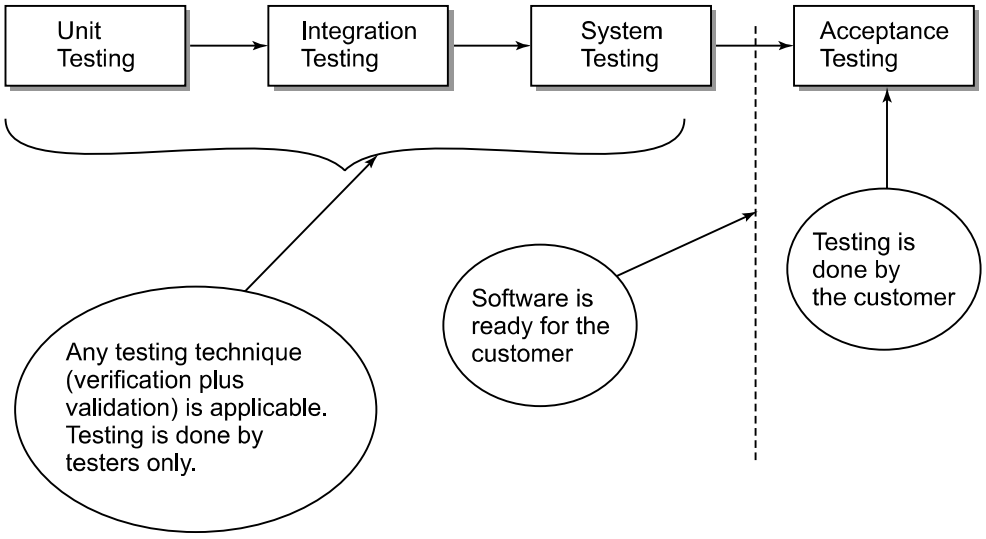


Figure 8.1. Levels of testing

8.1.1 Unit Testing

We develop software in parts / units and every unit is expected to have a defined functionality. We may call it a component, module, procedure, function, etc., which will have a purpose and may be developed independently and simultaneously. A. Bertolino and E. Marchetti have defined a unit as [BERT07]:

> "A unit is the smallest testable piece of software, which may consist of hundreds or even just few lines of source code, and generally represents the result of the work of one or few developers. The unit test cases' purpose is to ensure that the unit satisfies its functional specification and / or that its implemented structure matches the intended design structure. [BEIZ90, PFLE01]."

There are also problems with unit testing. How can we run a unit independently? A unit may not be completely independent. It may be calling a few units and also be called by one or more units. We may have to write additional source code to execute a unit. A unit X may call a unit Y and a unit Y may call a unit A and a unit B as shown in Figure 8.2(a). To execute a unit Y independently, we may have to write additional source code in a unit Y which may handle the activities of a unit X and the activities of a unit A and a unit B. The additional source code to handle the activities of a unit X is called 'driver' and the additional source code to handle the activities of a unit A and a unit B is called 'stub'. The complete additional source code which is written for the design of stub and driver is called scaffolding.

The scaffolding should be removed after the completion of unit testing. This may help us to locate an error easily due to small size of a unit. Many white box testing techniques may be effectively applicable at unit level. We should keep stubs and drivers simple and small in size to reduce the cost of testing. If we design units in such a way that they can be tested without writing stubs and drivers, we may be very efficient and lucky. Generally, in practice, it may be difficult and thus the requirement of stubs and drivers may not be eliminated. We may only minimize the requirement of scaffolding depending upon the functionality and its division in various units.

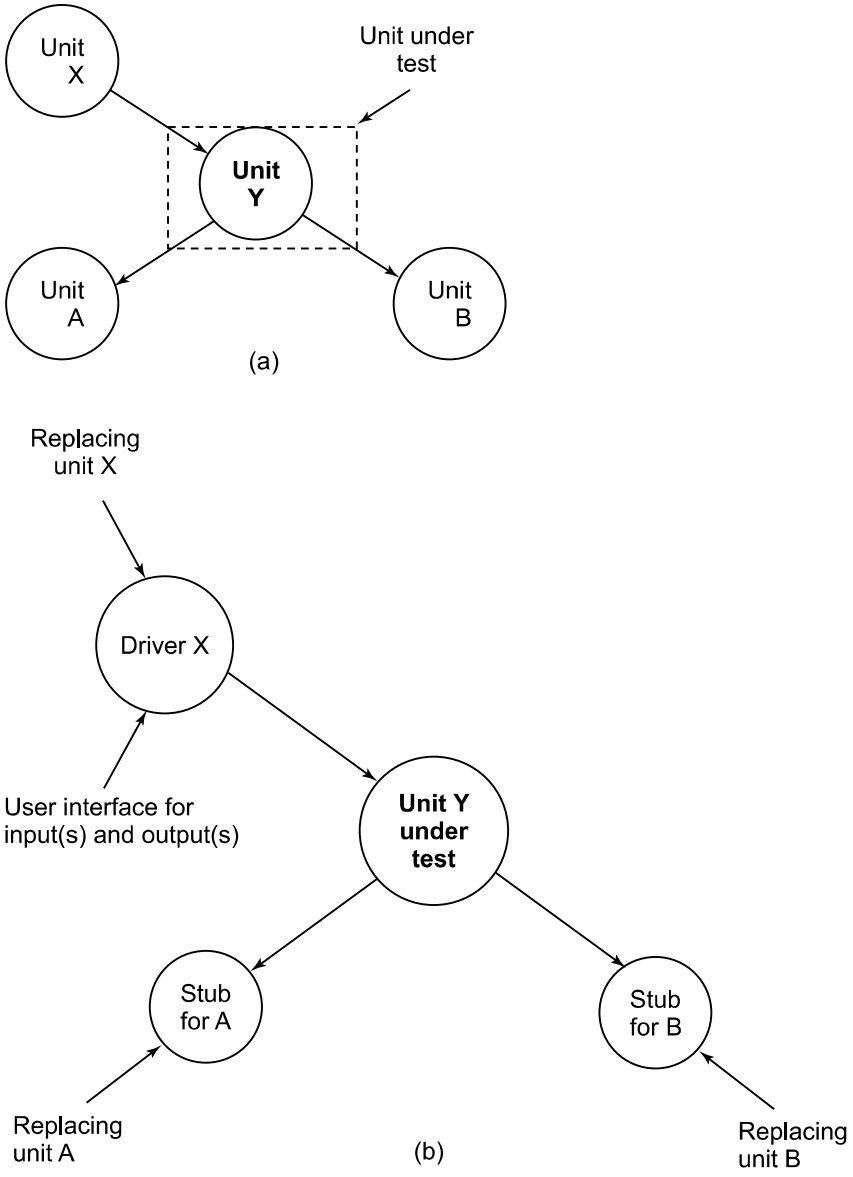
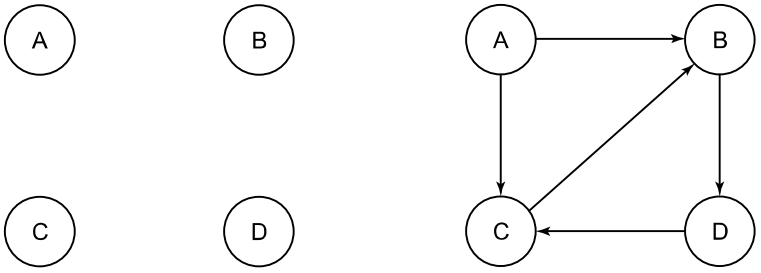


Figure 8.2. Unit under test with stubs and driver

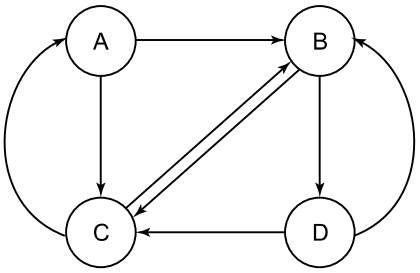
8.1.2 Integration Testing

A software program may have many units. We test units independently during unit testing after writing the required stubs and drivers. When we combine two units, we may like to test the interfaces amongst these units. We combine two or more units because they share some relationship. This relationship is represented by an interface and is known as coupling. The coupling is the measure of the degree of interdependence between units. Two units with high coupling are strongly connected and thus, dependent on each other. Two units with low coupling are weakly connected and thus have low dependency on each other. Hence, highly coupled units are heavily dependent on other units and loosely coupled units are comparatively less dependent on other units as shown in Figure 8.3.



(a) Uncoupled: No dependencies

(b) Loosely coupled units: Few dependencies



(c) Highly coupled units: Many dependencies

Figure 8.3. Coupling amongst units

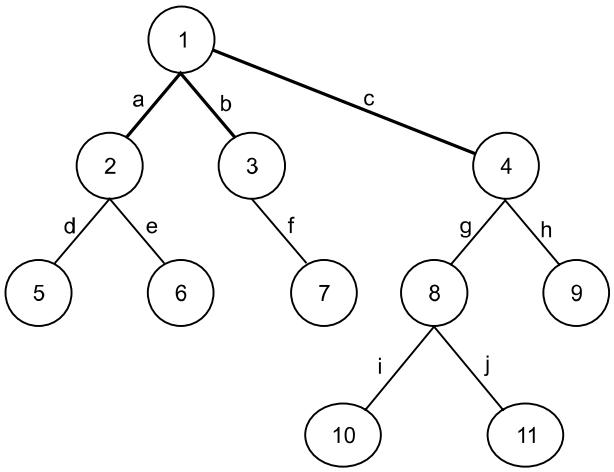
Coupling increases as the number of calls amongst units increases or the amount of shared data increases. A design with high coupling may have more errors. Loose coupling minimizes the interdependence, and some of the steps to minimize coupling are given as:

(i) Pass only data, not the control information.
(ii) Avoid passing undesired data.
(iii) Minimize parent/child relationship between calling and called units.
(iv) Minimize the number of parameters to be passed between two units.
(v) Avoid passing complete data structure.
(vi) Do not declare global variables.
(vii) Minimize the scope of variables.

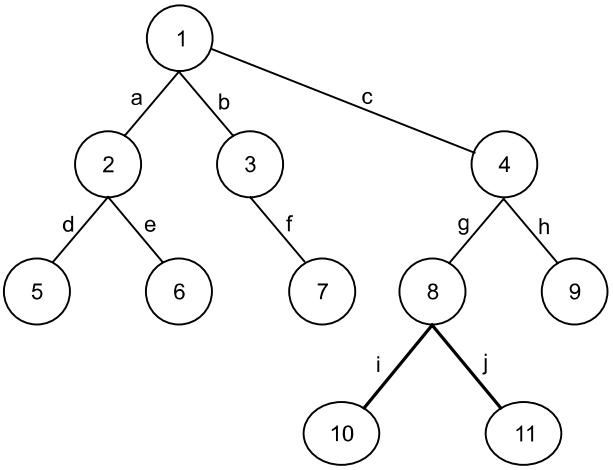
Different types of coupling are data (best), stamp, control, external, common and content (worst). When we design test cases for interfaces, we should be very clear about the coupling amongst units and if it is high, a large number of test cases should be designed to test that particular interface.

A good design should have low coupling and thus interfaces become very important. When interfaces are important, their testing will also be important. In integration testing, we focus on the issues related to interfaces amongst units. There are several integration strategies that really have little basis in a rational methodology and are given in Figure 8.4. Top down integration starts from the main unit and keeps on adding all called units of the next level. This portion should be tested thoroughly by focusing on interface issues. After completion of integration testing at this level, add the next level of units and so on till we reach the lowest level units (leaf units). There will not be any requirement of drivers and only stubs will be designed. In bottom-up integration, we start from the bottom, (i.e. from leaf units) and keep on adding upper level units till we reach the top (i.e. root node). There will not be any need of stubs. A sandwich

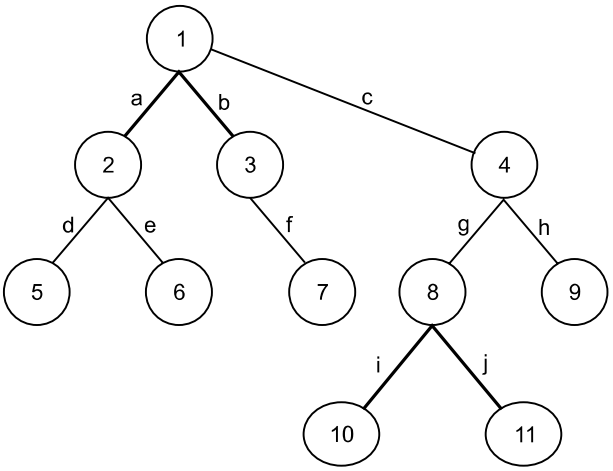
strategy runs from top and bottom concurrently, depending upon the availability of units and may meet somewhere in the middle.



(a) Top down integration (focus starts from edges a, b, c and so on)



(b) Bottom up integration (focus starts from edges i, j and so on)



(c) Sandwich integration (focus starts from a, b, i, j and so on)

Figure 8.4. Integration approaches

Each approach has its own advantages and disadvantages. In practice, sandwich integration approach is more popular. This can be started as and when two related units are available. We may use any functional or structural testing techniques to design test cases.

Functional testing techniques are easy to implement with a particular focus on the interfaces and some structural testing techniques may also be used. When a new unit is added as a part of integration testing, then the software is considered as a changed software. New paths are designed and new input(s) and output(s) conditions may emerge and new control logic may be invoked. These changes may also cause problems with units that previously worked flawlessly.

8.1.3 System Testing

We perform system testing after the completion of unit and integration testing. We test complete software along with its expected environment. We generally use functional testing techniques, although a few structural testing techniques may also be used.

A system is defined as a combination of the software, hardware and other associated parts that together provide product features and solutions. System testing ensures that each system function works as expected and it also tests for non-functional requirements like performance, security, reliability, stress, load, etc. This is the only phase of testing which tests both functional and non-functional requirements of the system. A team of the testing persons does the system testing under the supervision of a test team leader. We also review all associated documents and manuals of the software. This verification activity is equally important and may improve the quality of the final product.

Utmost care should be taken for the defects found during the system testing phase. A proper impact analysis should be done before fixing the defect. Sometimes, if the system permits, instead of fixing the defects, they are just documented and mentioned as the known limitations. This may happen in a situation when fixing is very time consuming or technically it is not possible in the present design, etc. Progress of system testing also builds confidence in the development team as this is the first phase in which the complete product is tested with a specific focus on the customer's expectations. After the completion of this phase, customers are invited to test the software.

8.1.4 Acceptance Testing

This is the extension of system testing. When the testing team feels that the product is ready for the customer(s), they invite the customer(s) for demonstration. After demonstration of the product, customer(s) may like to use the product to assess their satisfaction and confidence. This may range from adhoc usage to systematic well-planned usage of the product. This type of usage is essential before accepting the final product. The testing done for the purpose of accepting a product is known as acceptance testing. This may be carried out by the customer(s) or persons authorized by the customer(s). The venue may be the developer's site or the customer's site depending on mutual agreement. Generally, acceptance testing is carried out at the customer's site. Acceptance testing is carried out only when the software is developed for a particular customer(s). If we develop 'standardised' software for anonymous users at large

(like operating systems, compilers, case tools, etc.), then acceptance testing is not feasible. In such cases, potential customers are identified to test the software and this type of testing is called alpha / beta testing. Beta testing is done by many potential customers at their sites without any involvement of developers / testers. However, alpha testing is done by some potential customers at the developer's site under the direction and supervision of developers testers.

8.2 DEBUGGING

Whenever a software fails, we would like to understand the reason(s) for such a failure. After knowing the reason(s), we may attempt to find the solution and may make necessary changes in the source code accordingly. These changes will hopefully remove the reason(s) for that software failure. The process of identifying and correcting a software error is known as debugging. It starts after receiving a failure report and completes after ensuring that all corrections have been rightly placed and the software does not fail with the same set of input(s). The debugging is quite a difficult phase and may become one of the reasons for the software delays.

Every bug detection process is different and it is difficult to know how long it will take to detect and fix a bug. Sometimes, it may not be possible to detect a bug or if a bug is detected, it may not be feasible to correct it at all. These situations should be handled very carefully. In order to remove bugs, developers should understand that a problem prevails and then he/she should do the classification of the bug. The next step is to identify the location of the bug in the source code and finally take the corrective action to remove the bug.

8.2.1 Why Debugging is so Difficult?

Debugging is a difficult process. This is probably due to human involvement and their psychology. Developers become uncomfortable after receiving any request of debugging. It is taken against their professional pride. Shneiderman [SHNE80] has rightly commented on the human aspect of debugging as:

> "It is one of the most frustrating parts of programming. It has elements of problem solving or brain teasers, coupled with the annoying recognition that we have made a mistake. Heightened anxiety and the unwillingness to accept the possibility of errors, increase the task difficulty. Fortunately, there is a great sigh of relief and a lessening of tension when the bug is ultimately corrected."

These comments explain the difficulty of debugging. Pressman [PRES97] has given some clues about the characteristics of bugs as:

> "The debugging process attempts to match symptom with cause, thereby leading to error correction. The symptom and the cause may be geographically remote. That is, symptom may appear in one part of program, while the cause may actually be located in other part. Highly coupled program structures may further complicate this situation. Symptom may also disappear temporarily

when another error is corrected. In real time applications, it may be difficult to accurately reproduce the input conditions. In some cases, symptom may be due to causes that are distributed across a number of tasks running on different processors."

There may be many reasons which may make the debugging process difficult and time consuming. However, psychological reasons are more prevalent over technical reasons. Over the years, debugging techniques have substantially improved and they will continue to develop significantly in the near future. Some debugging tools are available and they minimize the human involvement in the debugging process. However, it is still a difficult area and consumes a significant amount of time and resources.

8.2.2 Debugging Process

Debugging means detecting and removing bugs from the programs. Whenever a program generates an unexpected behaviour, it is known as a failure of the program. This failure may be mild, annoying, disturbing, serious, extreme, catastrophic or infectious. Depending on the type of failure, actions are required to be taken. The debugging process starts after receiving a failure report either from the testing team or from users. The steps of the debugging process are replication of the bug, understanding the bug, locating the bug, fixing the bug and retesting the program. These steps are explained below:

(i) Replication of the bug: The first step in fixing a bug is to replicate it. This means to recreate the undesired behaviour under controlled conditions. The same set of input(s) should be given under similar conditions to the program and the program after execution, should produce a similar unexpected behaviour. If this happens, we are able to replicate a bug. In many cases, this is simple and straight forward. We execute the program on a particular input(s) or we press a particular button on a particular dialog, and the bug occurs. In other cases, replication may be very difficult. It may require many steps or in an interactive program such as a game, it may require precise timing. In worst cases, replication may be nearly impossible. If we do not replicate the bug, how will we verify the fix? Hence, failure to replicate a bug is a real problem. If we cannot do it, any action, which cannot be verified, has no meaning, howsoever important it may be. Some of the reasons for non-replication of a bug are:

- The user incorrectly reported the problem.
- The program has failed due to hardware problems like memory overflow, poor network connectivity, network congestion, non-availability of system buses, deadlock conditions, etc.
- The program has failed due to system software problems. The reason may be the usage of a different type of operating system, compilers, device drivers, etc. There may be any of the above-mentioned reasons for the failure of the program, although there is no inherent bug in the program for this particular failure.

Our effort should be to replicate the bug. If we cannot do so, it is advisable to keep the matter pending till we are able to replicate it. There is no point in playing with the source code for a situation which is not reproducible.

(ii) Understanding the bug

After replicating the bug, we may like to understand the bug. This means, we want to find the reason(s) for this failure. There may be one or more reasons and is generally the most time consuming activity. We should understand the program very clearly for understanding a bug. If we are the designers and source code writers, there may not be any problem for understanding the bug. If not, then we may have serious problems. If readability of the program is good and associated documents are available, we may be able to manage the problem. If readability is not that good, (which happens in many situations) and associated documents are not proper and complete, the situation becomes very difficult and complex. We may call the designers; if we are lucky, they may be available with the company and we may get them. In case of the designers not being available, the situation becomes challenging and in practice many times, we have to face this and struggle with the source code and documents written by the persons not available with the company. We may have to put effort in order to understand the program. We may start from the first statement of the source code to the last statement with a special focus on critical and complex areas of the source code. We should be able to know where to look in the source code for any particular activity. The source code should also tell us the general way in which the program acts.

The worst cases are large programs written by many persons over many years. These programs may not have consistency and may become poorly readable over time due to various maintenance activities. We should simply do the best and try to avoid making the mess worse. We may also take the help of source code analysis tools for examining large programs. A debugger may also be helpful for understanding the program. A debugger inspects a program statement-wise and may be able to show the dynamic behaviour of the program using a breakpoint. The breakpoints are used to pause the program at any time needed. At every breakpoint, we may look at values of variables, contents of relevant memory locations, registers, etc. The main point is that in order to understand a bug, program understanding is essential. We should put the desired effort before finding the reasons for the software failure. If we fail to do so, unnecessarily, we may waste our effort, which is neither required nor desired.

(iii) Locate the bug

There are two portions of the source code which need to be considered for locating a bug. The first portion of the source code is one which causes the visible incorrect behaviour and the second portion of the source code is one which is actually incorrect. In most of the situations, both portions may overlap and sometimes, both portions may be in different parts of the program. We should first find the source code which causes the incorrect behaviour. After knowing the incorrect behaviour and its related portion of the source code, we may find the portion of the source code which is at fault. Sometimes, it may be very easy to identify the problematic source code (the second portion of the source code) with manual inspection. Otherwise, we may have to take the help of a debugger. If we have 'core dumps', a debugger can immediately identify the line which fails. A 'core dumps' is the printout of all registers and relevant memory locations. We should document them and also retain them for possible future use. We may provide breakpoints while replicating the bug and this process may also help us to locate the bug.

Sometimes simple print statements may help us to locate the sources of the bad behaviour. This simple way provides us the status of various variables at different locations of the program with a specific set of inputs. A sequence of print statements may also portray the dynamics of variable changes. However, it is cumbersome to use in large programs. They may also generate superfluous data which may be difficult to analyze and manage.

We may add check routines in the source code to verify the correctness of the data structures. This may help us to know the problematic areas of the source code. If execution of these check routines is not very time consuming, then we may always add them. If it is time consuming, we may design a mechanism to make them operational, whenever required.

The most useful and powerful way is to inspect the source code. This may help us to understand the program, understand the bug and finally locate the bug. A clear understanding of the program is an absolute requirement of any debugging activity. Sometimes, the bug may not be in the program at all. It may be in a library routine or in the operating system, or in the compiler. These cases are very rare, but there are chances and if everything fails, we may have to look for such options.

(iv) Fix the bug and re-test the program

After locating the bug, we may like to fix the bug. The fixing of a bug is a programming exercise rather than a debugging activity. After making necessary changes in the source code, we may have to re-test the source code in order to ensure that the corrections have been rightly done at right place. Every change may affect other portions of the source code too. Hence an impact analysis is required to identify the affected portion and that portion should also be re-tested thoroughly. This re-testing activity is called regression testing which is a very important activity of any debugging process.

8.2.3 Debugging Approaches

There are many popular debugging approaches, but success of any approach is dependent upon the understanding of the program. If the persons involved in debugging understand the program correctly, they may be able to detect and remove the bugs.

(i) Trial and Error Method

This approach is dependent on the ability and experience of the debugging persons. After getting a failure report, it is analyzed and the program is inspected. Based on experience and intelligence, and also using the 'hit and trial' technique, the bug is located and a solution is found. This is a slow approach and becomes impractical in large programs.

(ii) Backtracking

This can be used successfully in small programs. We start at the point where the program gives an incorrect result such as an unexpected output is printed. After analyzing the output, we trace backward the source code manually until a cause of the failure is found. The source code, from the statement where symptoms of the failure is found, to the statement where the cause of failure is found, is analyzed properly. This technique brackets the locations of the bug in the program. Subsequent careful study

of the bracketed location may help us to rectify the bug. Another obvious variation of backtracking is forward tracking, where we use print statements or other means to examine a succession of intermediate results to determine at what point the result first became wrong. These approaches (backtracking and forward tracking) may be useful only when the size of the program is small. As the program size increases, it becomes difficult to manage these approaches.

(iii) Brute Force

This is probably the most common and efficient approach to identify the cause of a software failure. In this approach, memory dumps are taken, run time traces are invoked and the program is loaded with print statements. When this is done, we may find a clue by the information produced which leads to identification of cause of a bug. Memory traces are similar to memory dumps, except that the printout contains only certain memory and register contents and printing is conditional on some event occurring. Typically conditional events are entry, exit or use of one of the following:

- A particular subroutine, statement or database
- Communication with I/O devices
- Value of a variable
- Timed actions (periodic or random) in certain real time system.

A special problem with trace programs is that the conditions are entered in the source code and any changes require a recompilation. A huge amount of data is generated, which, although may help to identify the cause, but may be difficult to manage and analyze.

(iv) Cause Elimination

Cause elimination is manifested by induction or deduction and also introduces the concept of binary partitioning. Data related to error occurrence are organized to isolate potential causes. Alternatively, a list of all possible causes is developed and tests are conducted to eliminate each. Therefore, we may rule out causes one by one until a single one remains for validation. The cause is identified, properly fixed and re-tested accordingly.

8.2.4 Debugging Tools

Many debugging tools are available to support the debugging process. Some of the manual activities can also be automated using a tool. We may need a tool that may execute every statement of a program at a time and print values of any variable after executing every statement of the program. We will be free from inserting print statements in the program manually. Thus, run time debuggers are designed. Fundamentally, a run time debugger is similar to an automatic print statement generator. It helps us to trace the program path and the defined variables without having to put print statements in the source code. Every compiler available in the market comes with run time debugger. It allows us to compile and run the program with a single compilation, rather than modifying the source code and recompiling as we try to narrow down the bug.

Run time debuggers may detect bugs in the program, but may fail to find the causes of failures. We may need a special tool to find causes of failures and correct the bug. Some errors like memory corruption and memory leaks may be detected automatically. The automation was the modification in the debugging process because it automated the process of finding the bug. A tool may detect an error and our job is to simply fix it. These tools are known as 'automatic debugger' and are available in different varieties. One variety may be a library of functions that may be connected into the program. During execution of the program, these functions are called and the debugger looks for memory corruption and other similar issues. If anything is found, it is reported accordingly.

Compilers are also used for finding bugs. Of course, they check only syntax errors and particular types of run time errors. Compilers should give proper and detailed messages of errors that will be of great help to the debugging process. Compilers may give all such information in the attribute table, which is printed along with the listing. The attribute table contains various levels of warnings which have been picked up by the compiler scan and which are noted. Hence, compilers come with an error detection feature and there is no excuse to design compilers without meaningful error messages.

We may apply a wide variety of tools like run time debugger, automatic debugger, automatic test case generators, memory dumps, cross reference maps, compilers, etc. during the debugging process. However, tools are not the substitute for careful examination of the source code after thorough understanding.

8.3 SOFTWARE TESTING TOOLS

The most important effort-consuming task in software testing is to design the test cases. The execution of these test cases may not require much time and resources. Hence, the designing part is more significant than the execution part. Both parts are normally handled manually. Do we really need a tool? If yes, where and when can we use it – in the first part (designing of test cases) or second part (execution of test cases) or both? Software testing tools may be used to reduce the time of testing and to make testing as easy and pleasant as possible. Automated testing may be carried out without human involvement. This may help us in the areas where a similar dataset is to be given as input to the program again and again. A tool may undertake repeated testing, unattended (and without human intervention), during nights or on weekends.

Many non-functional requirements may be tested with the help of a tool. We want to test the performance of a software under load, which may require many computers, manpower and other resources. A tool may simulate multiple users on one computer and also a situation when many users are accessing a database simultaneously.

There are three broad categories of software testing tools i.e. static, dynamic and process management. Most of the tools fall clearly into one of these categories but there are a few exceptions like mutation analysis system which falls in more than one category. A wide variety of tools are available with different scope and quality and they assist us in many ways.

8.3.1 Static Software Testing Tools

Static software testing tools are those that perform analysis of the programs without executing them at all. They may also find the source code which will be hard to test and maintain. As we

all know, static testing is about prevention and dynamic testing is about cure. We should use both the tools but prevention is always better than cure. These tools will find more bugs as compared to dynamic testing tools (where we execute the program). There are many areas for which effective static testing tools are available, and they have shown their results for the improvement of the quality of the software.

(i) Complexity analysis tools

Complexity of a program plays a very important role while determining its quality. A popular measure of complexity is the cyclomatic complexity as discussed in chapter 4. This gives us the idea about the number of independent paths in the program and is dependent upon the number of decisions in the program. A higher value of cyclomatic complexity may indicate poor design and risky implementation. This may also be applied at the module level, and higher cyclomatic complexity value modules may either be redesigned or may be tested very thoroughly. There are other complexity measures also which are used in practice like Halstead software size measures, knot complexity measure, etc. Tools are available which are based on any of the complexity measures. These tools may take the program as an input, process it and produce a complexity value as output. This value may be an indicator of the quality of design and implementation.

(ii) Syntax and semantic analysis tools

These tools find syntax and semantic errors. Although the compiler may detect all syntax errors during compilation, early detection of such errors may help to minimize other associated errors. Semantic errors are very significant and compilers are helpless in finding such errors. There are tools in the market that may analyze the program and find errors. Non-declaration of a variable, double declaration of a variable, 'divide by zero' issue, unspecified inputs and non-initialization of a variable are some of the issues which may be detected by semantic analysis tools. These tools are language dependent and may parse the source code, maintain a list of errors and provide implementation information. The parser may find semantic errors as well as make an inference as to what is syntactically correct.

(iii) Flow graph generator tools

These tools are language dependent and take the program as an input and convert it to its flow graph. The flow graph may be used for many purposes like complexity calculation, paths identification, generation of definition use paths, program slicing, etc. These tools assist us to understand the risky and poorly designed areas of the source code.

(iv) Code comprehension tools

These tools may help us to understand unfamiliar source code. They may also identify dead source code, duplicate source code and areas that may require special attention and should be reviewed seriously.

(v) Code inspectors

Source code inspectors do the simple job of enforcing standards in a uniform way for many programs. They inspect the programs and force us to implement the guidelines of good programming practices. Although they are language dependent, most of the guidelines of good programming practices are similar in many languages. These tools

are simple and may find many critical and weak areas of the program. They may also suggest possible changes in the source code for improvement.

8.3.2 Dynamic Software Testing Tools

Dynamic software testing tools select test cases and execute the program to get the results. They also analyze the results and find reasons for failures (if any) of the program. They will be used after the implementation of the program and may also test non-functional requirements like efficiency, performance, reliability, etc.

(i) Coverage analysis tools

These tools are used to find the level of coverage of the program after executing the selected test cases. They give us an idea about the effectiveness of the selected test cases. They highlight the unexecuted portion of the source code and force us to design special test cases for that portion of the source code. There are many levels of coverage like statement coverage, branch coverage, condition coverage, multiple condition coverage, path coverage, etc. We may like to ensure that at least every statement must be executed once and every outcome of the branch statement must be executed once. This minimum level of coverage may be shown by a tool after executing an appropriate set of test cases. There are tools available for checking statement coverage, branch coverage, condition coverage, multiple conditions coverage and path coverage. The profiler displays the number of times each statement is executed. We may study the output to know which portion of the source code is not executed. We may design test cases for those portions of the source code in order to achieve the desired level of coverage. Some tools are also available to check whether the source code is as per standards or not and also generate a number of commented lines, non-commented lines, local variables, global variables, duplicate declaration of variables, etc. Some tools check the portability of the source code. A source code is not portable if some operating system dependent features are used. Some tools are Automated QA's time, Parasoft's Insure++ and Telelogic's Logicscope.

(vi) Performance testing tools

We may like to test the performance of the software under stress / load. For example, if we are testing a result management software, we may observe the performance when 10 users are entering the data and also when 100 users are entering the data simultaneously. Similarly, we may like to test a website with 10 users, 100 users, 1000 users, etc. working simultaneously. This may require huge resources and sometimes, it may not be possible to create such real life environment for testing in the company. A tool may help us to simulate such situations and test these situations in various stress conditions. This is the most popular area for the usage of any tool and many popular tools are available in the market. These tools simulate multiple users on a single computer. We may also see the response time for a database when 10 users access the database, when 100 users access the database and when 1000 users access the data base simultaneously. Will the response time be 10 seconds or 100 seconds or even 1000 seconds? No user may like to tolerate the response time in minutes. Performance testing includes load

and stress testing. Some of the popular tools are Mercury Interactive's Load Runner, Apache's J Meter, Segue Software's Silk Performer, IBM Rational's Performance Tester, Comuware's QALOAD and AutoTester's AutoController.

(vii) Functional / Regression Testing Tools

These tools are used to test the software on the basis of its functionality without considering the implementation details. They may also generate test cases automatically and execute them without human intervention. Many combinations of inputs may be considered for generating test cases automatically and these test cases may be executed, thus, relieving us from repeated testing activities. Some of the popular available tools are IBM Rational's Robot, Mercury Interactive's Win Runner, Comuware's QA Centre and Segue Software's Silktest.

8.3.3 Process Management Tools

These tools help us to manage and improve the software testing process. We may create a test plan, allocate resources and prepare a schedule for unattended testing for tracking the status of a bug using such tools. They improve many aspects of testing and make it a disciplined process. Some of the tools are IBM Rational Test Manager, Mercury Interactive's Test Director, Segue Software's Silk Plan Pro and Compuware's QA Director. Some configuration management tools are also available which may help bug tracking, its management and correctness like IBM Rational Software's Clear DDTs, Bugzilla and Samba's Jitterbug.

Selection of any tool is dependent upon the application, expectations, quality requirements and available trained manpower in the organization. Tools assist us to make testing effective, efficient and performance oriented.

8.4 SOFTWARE TEST PLAN

It is a document to specify the systematic approach to plan the testing activities of the software. If we carry out testing as per a well-designed systematic test plan document, the effectiveness of testing will improve and that may further help to produce a good quality product. The test plan document may force us to maintain a certain level of standards and disciplined approach to testing. Many software test plan documents are available, but the most popular document is the IEEE standard for Software Test Documentation (Std 829 – 1998). This document addresses the scope, schedule, milestones and purpose of various testing activities. It also specifies the items and features to be tested and features which are not to be tested. Pass/fail criteria, roles and responsibilities of persons involved, associated risks and constraints are also described in this document. The structure of the IEEE Std 829 – 1998 test plan document is given in [IEEE98c]. All ten sections have a specific purpose. Some changes may be made as per requirement of the project. A test plan document is prepared after the completion of the SRS document and may be modified along with the progress of the project. We should clearly specify the test coverage criteria and testing techniques to achieve the criteria. We should also describe who will perform testing, at what level and when. Roles and responsibilities of testers must be clearly documented.

9

Object Oriented Testing

What is object orientation? Why is it becoming important and relevant in software development? How is it improving the life of software developers? Is it a buzzword? Many such questions come into our mind whenever we think about object orientation of software engineering. Companies are releasing the object oriented versions of existing software products. Customers are also expecting object oriented software solutions. Many developers are of the view that structural programming, modular design concepts and conventional development approaches are old fashioned activities and may not be able to handle today's challenges. They may also feel that real world situations are effectively handled by object oriented concepts using modeling in order to understand them clearly. Object oriented modeling may improve the quality of the SRS document, the SDD document and may help us to produce good quality maintainable software. The software developed using object orientation may require a different set of testing techniques, although few existing concepts may also be applicable with some modifications.

9.1 WHAT IS OBJECT ORIENTATION?

We may model real world situations using object oriented concepts. Suppose we want to send a book to our teacher who does not stay in the same city, we cannot go to his house for delivery of the book because he stays in a city which is 500 km away from our city. As we all know, sending a book is not a difficult task. We may go to a nearby courier agent (say Fast Track Courier) and ask to deliver the book to our teacher on his address. After this, we are sure that the book will be delivered automatically and also within the specified time (say two days). The agents of Fast Track Courier will perform the job without disturbing us at all.

Our objective is that we want to send a book to our teacher who does not stay in our city. This objective may be simply achieved, when we identify a proper 'agent' (say Fast Track Courier) and give a 'message' to send the book on an address of the teacher. It is the 'responsibility' of the identified agent (Fast Track Agent) to send the book. There are many 'methods' or ways to

perform this task. We are not required to know the details of the operations to be performed to complete this task. Our interest is very limited and focused. If we investigate further, we may come to know that there are many ways to send a book like using train network, bus network, air network or combinations of two or more available networks. The selection of any method is the prerogative of our agent (say, agent of Fast Track Company). The agent may transfer the book to another agent with the delivery address and a message to transfer to the next agent and so on. Our task may be done by a sequence of requests from one agent to another.

In object orientation, an action is initiated by sending a message (request) to an agent who is responsible for that action. An agent acts as a receiver and if it accepts a message (request), it becomes its responsibility to initiate the desired action using some method to complete the task.

In real world situations, we do not need to know all operations of our agents to complete the assigned task. This concept of 'information hiding' with respect to message passing has become very popular in object oriented modeling. Another dimension is the interpretation of the message by the receiver. All actions are dependent upon the interpretation of the received message. Different receivers may interpret the same message differently. They may decide to use different methods for the same message. Fast Track Agency may use air network while another agency may use train network and so on. If we request our tailor to send the book, he may not have any solution for our problem. The tailor will only deny such requests. Hence, a message should be issued to a proper agent (receiver) in order to complete the task. Object orientation is centered around a few basic concepts like objects, classes, messages, interfaces, inheritance and polymorphism. These concepts help us to model a real world situation which provides the foundation for object oriented software engineering.

9.1.1 Classes and Objects

We consider the same example of sending a book to a teacher. The selection of the courier company is based on its reputation and proximity to our house. A courier management system is required in order to send a book to a teacher. All courier types (such as book, pen, etc.) may be combined to form a group and this group is known as a class. All objects are instances of a class. The class describes the structure of the instances which include behaviour and information. In our example, courier (containing courier details) is a class and all courier types are its objects as shown in Figure 9.1.

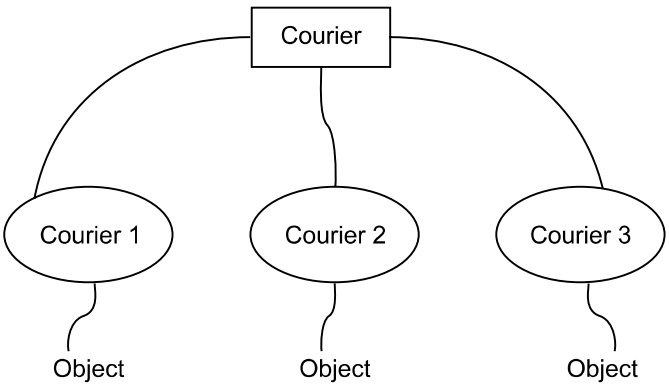


Figure 9.1. Class and its objects

All objects have unique identification and are distinguishable. There may be four horses having same attributes colour, breed and size but all are distinguishable due to their colour. The term identity means that objects are distinguished by their inherent existence and not by descriptive properties [JOSH03].

What types of things become objects? Anything and everything may become an object. In our example, customer, courier and tracking are nothing but objects. The class of an object provides the structure for the object i.e. its state and operations. A courier class is shown in Figure 9.2 with eight attributes and four operations.

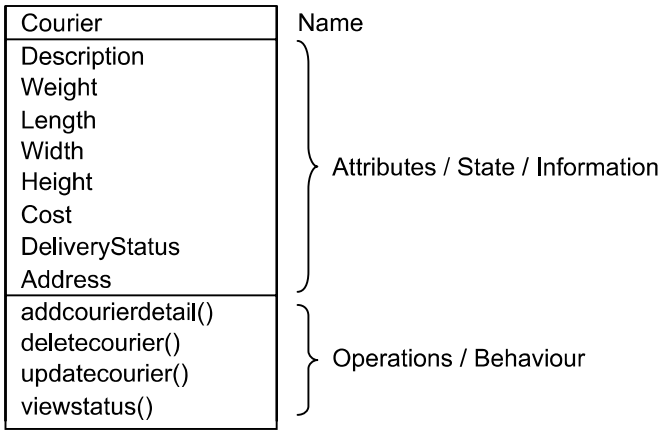


Figure 9.2. Class courier

An attribute (or information / state) is a data value held by the object of a class. The courier may have different height, weight, width, length, description and it may be delivered or not. The attributes are shown as the second part of the class courier as given in Figure 9.2. Operations (or behaviour) are the functions which may be applied on a class. All objects of a class have the same operations. A class courier shown in Figure 9.2 has four operations namely 'addcourierdetail', 'deletecourier', 'updatecourier' and 'viewstatus'. These four operations are defined for a Class Courier in the Figure 9.2. In short, every object has a list of functions (operation part) and data values required to store information (Attribute part).

I. Jacobson has defined a class as [JACO98]:

> "A class represents a template for several objects and describes how these objects are structured internally. Objects of the same class have the same definition both for their operations and for their information structures."

In an object oriented system, every object has a class and the object is called an instance of that class. We use object and instance as synonyms and an object is defined as [JACO98]:

> "An instance is an object created from a class. The class describes the (behaviour and information) structure of the instance, while the current state of the instance is defined by the operations performed on the instance."

9.1.2 Inheritance

We may have more information about Fast Track Courier Company not necessarily because it is a courier company but because it is a company. As a company, it will have employees, balance sheet, profit / loss account and a chief executive officer. It will also charge for its

services and products from the customers. These things are also true for transport companies, automobile companies, aircraft companies, etc. Since the category 'courier company' is a more specialized form of the category 'company' and any knowledge of a company is also true for a courier company and subsequently also true for Fast Track Courier Company.

We may organize our knowledge in terms of hierarchy of categories as shown in Figure 9.3.

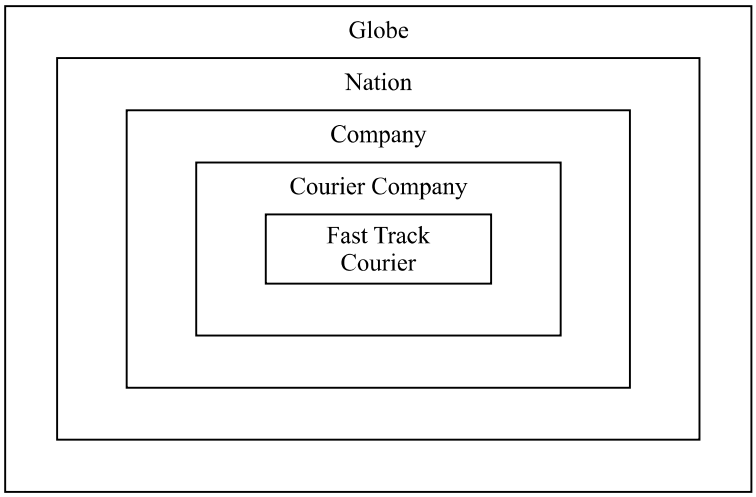


Figure 9.3. The categories around fast track courier

Fast Track Courier is a specialized category of a courier company; however, 'courier company' is a specialized category of a company. Moreover, a nation may have many companies and around the globe, we have many nations; all knowledge gathered so far, may not be directly applicable to Fast Track Company. Knowledge of a more general category, which is also applicable to a specialized category, is called inheritance. We may say that Fast Track Courier will inherit attributes of the category 'courier company' and 'courier company' will inherit the attributes of the category 'company'. This category is nothing but the class in the object oriented system. There is another tree-like structure used to represent a hierarchy of classes and is shown in Figure 9.4.

Information of a courier company is available to Fast Track Company because it is a sub-class of the class 'courier company'. The same information of a courier company is also applicable to Air World Courier and Express Courier because they are also sub-classes of the class 'courier company'. All classes inherit information from the upper classes. Hence information from a base class is common to all the derived classes; however, each derived class also has some additional information of its own. Each derived class inherits the attributes of its base class and this process is known as inheritance. In general, low level classes (known as sub-classes or derived classes) inherit state and behaviour from their high level class (known as a super class or base class).

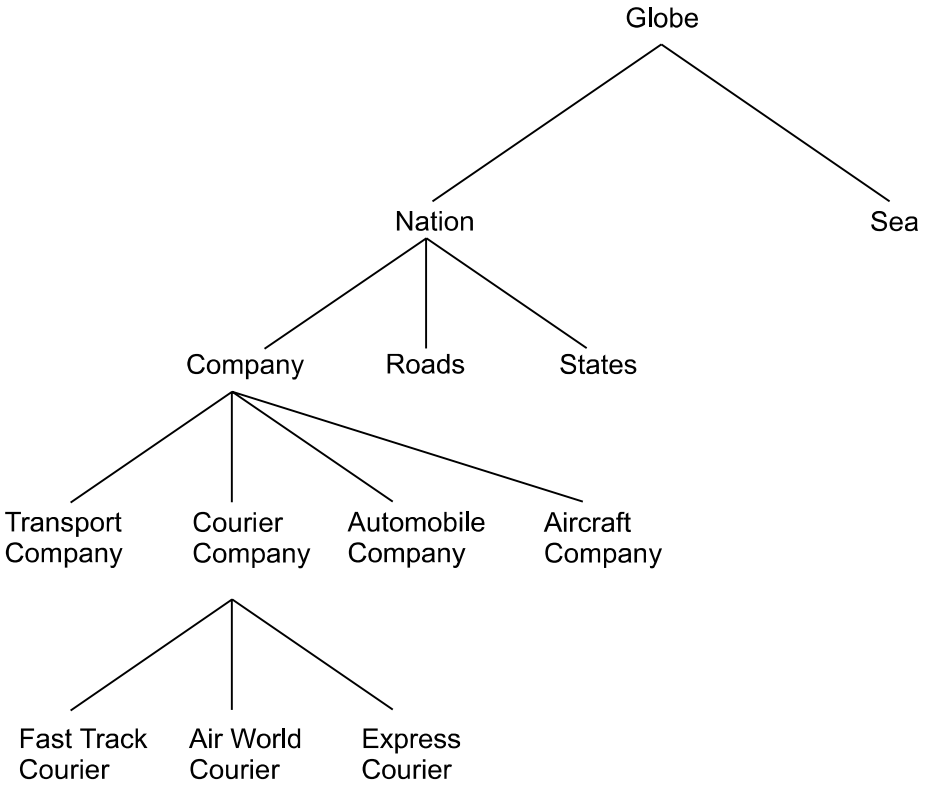


Figure 9.4. A class hierarchy

9.1.3 Messages, Methods, Responsibility, Abstraction

Objects communicate through passing messages. A message is a request for performing an operation by some object in the system. A message may consist of the identification of the target object, name of the requested operation and other relevant information for processing the request. An object which originates a message is called the sender and the object which receives a message is called the receiver. An object may send a message to another object or even to itself to perform designed functions. A 'method' is the sequence of steps (or set of operations) to be performed to fulfil the assigned task. There may be many methods available for any task. It is the responsibility of the receiver of the message to choose an appropriate method to complete a task effectively and efficiently. In Figure 9.2, four methods, 'addcourierdetail', 'deletecourier', 'updatecourier' and 'viewstatus' are available for courier class. In order to retrieve the delivery status of the courier, the 'viewstatus' method must be invoked.

Responsibility is an important concept of an object oriented system. Behaviour of an object is described in terms of responsibilities. Fast Track Courier is free to use any method to send the book without our involvement and interference. This type of independence increases the level of abstraction. This improves the independence amongst the objects which is very important for solving any complex problem. The complexity of a problem is managed using the right level of abstraction which is the elimination of the irrelevant and the amplification of the essentials. We learn driving a car by knowing driving essentials like steering wheel,

ignition, clutch, break, gear system without knowing any details of the type of engine, batteries, control system, etc. These details may not be required for a learner and may create unnecessary confusion. Hence, abstraction concept provides independence and improves the clarity of the system.

9.1.4 Polymorphism

The dictionary meaning of polymorphism is 'many forms'. In the real world, the same operations may have different meanings in different situations. For example, 'Human' is a sub-class of 'Mammal'. Similarly 'Dog', 'Bird', 'Horse' are also sub-classes of 'Mammal'. If a message 'come fast' is issued to all mammals, all may not behave in the same way. The horse and dog may run, the bird may fly and the human may take an aircraft. The behaviour of mammals is different on the same message. This concept is known as polymorphism, where the same message is sent to different objects irrespective of their class, but the responses of objects may be different. When we abstract the interface of an operation and leave the implementation details to sub-classes, this activity is called polymorphism. This operation is called polymorphic operation. We may create a super class by pulling out important states, behaviours and interfaces of the classes. This may further simplify the complexity of a problem. An object may not need to know the class of another object to whom it wishes to send a message, when we have polymorphism. This may be defined as [JACO98]:

> "Polymorphism means that the sender of a stimulus (message) does not need to know the receiving instance's class. The receiving instance can belong to an arbitrary class."

Polymorphism is considered to be an important concept of any object oriented programming language. As we all know, arithmetic operators such as +, =, - are used to operate on primary data types such as int, float, etc. We may overload these operators so that they may operate in the same way on objects (user defined data types) as they operate on primary data types. Thus, the same operators will have multiple forms.

9.1.5 Encapsulation

Encapsulation is also known as information hiding concept. It is a way in which both data and functions (or operations) that operate on data are combined into a single unit. The only way to access the data is through functions, which operate on the data. The data is hidden from the external world. Hence, it is safe from outside (external) and accidental modifications. For example, any object will have attributes (data) and operations which operate on the specified data only.

If data of any object needs to be modified, it may be done through the specified functions only. The process of encapsulating the data and functions into a single unit simplifies the activities of writing, modifying, debugging and maintaining the program.

In a university, every school may access and maintain its data on its own. One school is not allowed to access the data of another school directly. This is possible only by sending a request to the other school for the data. Hence, the data and functions that operate on the data are

specific to each school and are encapsulated into a single unit that is the school of a university.

9.2 WHAT IS OBJECT ORIENTED TESTING?

Object oriented programming concepts are different from conventional programming and have become the preferred choice for a large scale system design. The fundamental entity is the class that provides an excellent structuring mechanism. It allows us to divide a system into well-defined units which may then be implemented separately. We still do unit testing although the meaning of unit has changed. We also do integration and system testing to test the correctness of implementation. We also do regression testing in order to ensure that changes have been implemented correctly. However, many concepts and techniques are different from conventional testing.

9.2.1 What is a Unit?

In conventional programming, a unit is the smallest portion of the program that can be compiled and executed. We may call it a module, component, function or procedure. In object oriented system, we have two options for a unit. We may treat each class as a unit or may treat each method within a class as a unit. If a class is tested thoroughly, it can be reused without being unit tested again. Unit testing of a class with a super class may be impossible to do without the super classes' methods/variables. One of the solutions is to merge the super class and the class under test so that all methods and variables are available. This may solve the immediate testing problem and is called flattening of classes. But classes would not be flattened in the final product, so potential issues may still prevail. We may have to redo flattening after completion when dealing with multiple inheritance. If we decide to choose method as a unit, then these issues will be more complicated and difficult to implement. Generally, classes are selected as a unit for the purpose of unit testing.

9.2.2 Levels of Testing

We may have 3 or 4 levels of testing depending on our approach. The various testing levels are:

(i) Method testing (Unit testing)
(ii) Class testing (Unit testing)
(iii) Inter-class testing (Integration testing)
(iv) System testing

In order to test a class, we may create an instance of the class i.e. object, and pass the appropriate parameters to the constructor. We may further call the methods of the object passing parameters and receive the results. We should also examine the internal data of the object. The encapsulation plays an important role in class testing because data and function (operations) are combined in a class. We concentrate on each encapsulated class during unit testing but each function may be difficult to test independently. Inter-class testing considers the

parameter passing issues between two classes and is similar to integration testing. System testing considers the whole system and test cases are generated using functional testing techniques.

Integration testing in object oriented system is also called inter-class testing. We do not have hierarchical control structure in object orientation and thus conventional integration testing techniques like top down, bottom up and sandwich integration cannot be applied. There are three popular techniques for inter-class testing in object oriented systems. The first is the thread based testing where we integrate classes that are needed to respond to an input given to the system. Whenever we give input to a system, one or more classes are required to be executed that respond to that input to get the result. We combine such classes which execute together for a particular input or set of inputs and this is treated as a thread. We may have many threads in a system, depending on various inputs. Thread based testing is easy to implement and has proved as an effective testing technique. The second is the use case based testing where we combine classes that are required by one use case.

The third is the cluster testing where we combine classes that are required to demonstrate one collaboration. In all three approaches, we combine classes on the basis of a concept and execute them to see the outcome. Thread based testing is more popular due to its simplicity and easy implementability.

The advantage of object oriented system is that the test cases can be generated earlier in the process, even when the SRS document is being designed. Early generation of test cases may help the designers to better understand and express requirements and to ensure that specified requirements are testable. Use cases are used to generate a good number of test cases. This process is very effective and also saves time and effort. Developers and testers understand requirements clearly and may design an effective and stable system. We may also generate test cases from the SDD document. Both the teams (testers and developers) may review the SRS and the SDD documents thoroughly in order to detect many errors before coding. However, testing of source code is still a very important part of testing and all generated test cases will be used to show their usefulness and effectiveness. We may also generate test cases on the basis of the availability of the source code.

Path testing, state based testing and class testing are popular object oriented testing techniques and are discussed in subsequent sections.

9.3 PATH TESTING

As discussed earlier, path testing is a structural testing technique where the source code is required for the generation of test cases. In object oriented testing, we also identify paths from the source code and write test cases for the execution of such paths. Most of the concepts of conventional testing such as generating test cases from independent paths are also applicable in object oriented testing.

9.3.1 Activity Diagram

The first step of path testing is to convert source code into its activity diagram. In Unified Modeling Language (UML), activity diagram is used to represent sequences in which all

activities are performed. This is similar to a flow graph which is the basis of conventional path testing. Activity diagram may be generated from a use case or from a class. It may represent basic flow and also possible alternative flows. As shown in Figure 9.5, the start state is represented by a solid circle and the end state is represented by a solid circle inside a circle. The activities are represented by rectangles with rounded corners along with their descriptions. Activities are nothing but the set of operations. After execution of these set of activities, a transition takes place to another activity. Transitions are represented by an arrow. When multiple activities are performed simultaneously, the situation is represented by a symbol 'fork'. The parallel activities are combined after the completion of such activities by a symbol 'join'. The number of fork and join in an activity diagram are the same. The branches are used to describe what activities are performed after evaluating a set of conditions. Branches may also be represented as diamonds with multiple labelled exit arrows. A guard condition is a boolean expression and is also written along with branches. An activity diagram consisting of seven activities is shown in Figure 9.5.

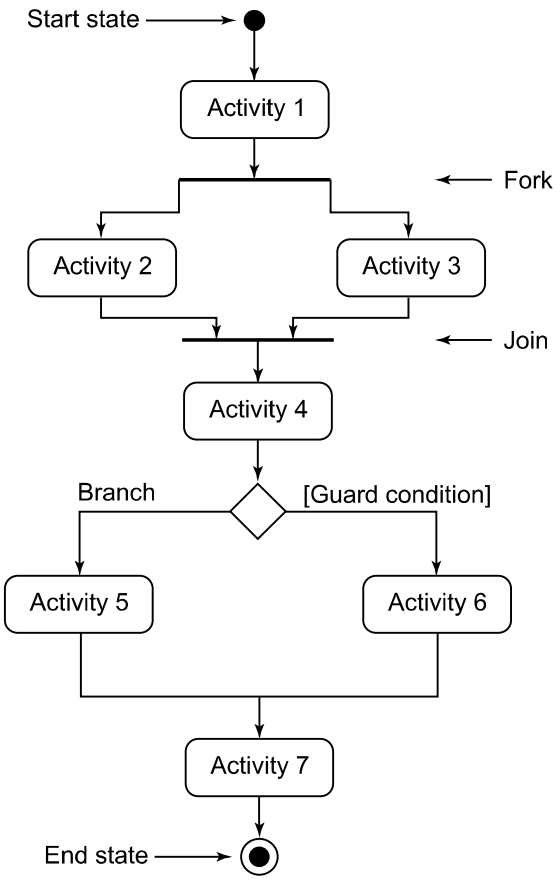
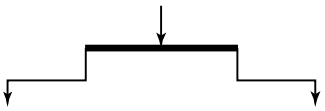
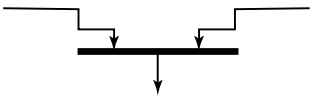
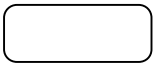
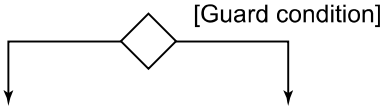


Figure 9.5. An example of an activity diagram

In the activity diagram given in Figure 9.5, Activity 2 and Activity 3 are performed simultaneously and combined by a join symbol. After Activity 4, a decision is represented by a diamond symbol and if the guard condition is true, Activity 5 is performed, otherwise Activity 6 is performed. The fork has one incoming transition (Activity 1 is split into sub-activities) and two outgoing transitions. Similarly join has two incoming transitions and one outgoing transition. The symbols of an activity diagram are given in Table 9.1.

Table 9.1. Symbols of an activity diagram

S. No.	Symbol	Notation	Remarks
1.	Fork		To represent multiple parallel activities i.e. an activity is split into two or more activities.
2.	Join		To represent the combination of two or more parallel activities after completion of respective activities.
3.	Transition		To represent transfer of flow of control from one activity to another.
4.	Activity		To represent a set of operations known as an activity.
5.	Start		To represent start state of an activity diagram.
6.	End		To represent end state of an activity diagram.
7.	Branch		To represent the transfer of flow on the basis of evaluation of boolean expression known as guard condition.

An activity diagram represents the flow of activities through the class. We may read the diagram from top to bottom i.e. from start symbol to end symbol. It provides the basis for the path testing where we may like to execute each independent path of the activity diagram at least once.

We consider the program given in Figure 9.6 for determination of division of a student. We give marks in three subjects as input to calculate the division of a student. There are three methods in this program – getdata, validate and calculate. The activity diagram for validate and calculate functions is given in Figure 9.7 and Figure 9.8.

```
#include<iostream.h>
#include<conio.h>

class student
{
int mark1;
int mark2;
int mark3;
public:
void getdata()
{
cout<<"Enter marks of 3 subjects (between 0-100)\n";
cout<<"Enter marks of first subject:";
cin>>mark1;
```

(*Contd.*)

(*Contd.*)

```
cout<<"Enter marks of second subject:";
cin>>mark2;
cout<<"Enter marks of third subject:";
cin>>mark3;
}
void validate()
{
if(mark1>100||mark1<0||mark2>100||mark2<0||mark3>100||mark3<0){
      cout<<"Invalid Marks! Please try again";
      }
else{
      calculate();
}
}
void calculate();
};

void student::calculate()
{
int avg;
avg=(mark1+mark2+mark3)/3;
if(avg<40) {
      cout<<"Fail";
}
else if(avg>=40&&avg<50){
      cout<<"Third Division";
}
else if(avg>=50&&avg<60){
      cout<<"Second Division";
}
else if(avg>=60&&avg<75){
      cout<<"First Division";
}
else{
      cout<<"First Division with Distinction";
}
}
void main()
{
clrscr();
student s1;
s1.getdata();
s1.validate();
getch();
}
```

Figure 9.6. Program to determine division of a student

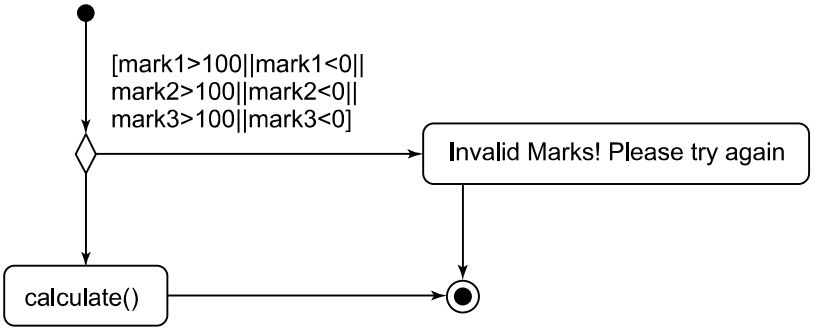


Figure 9.7. Activity diagram of function validate()

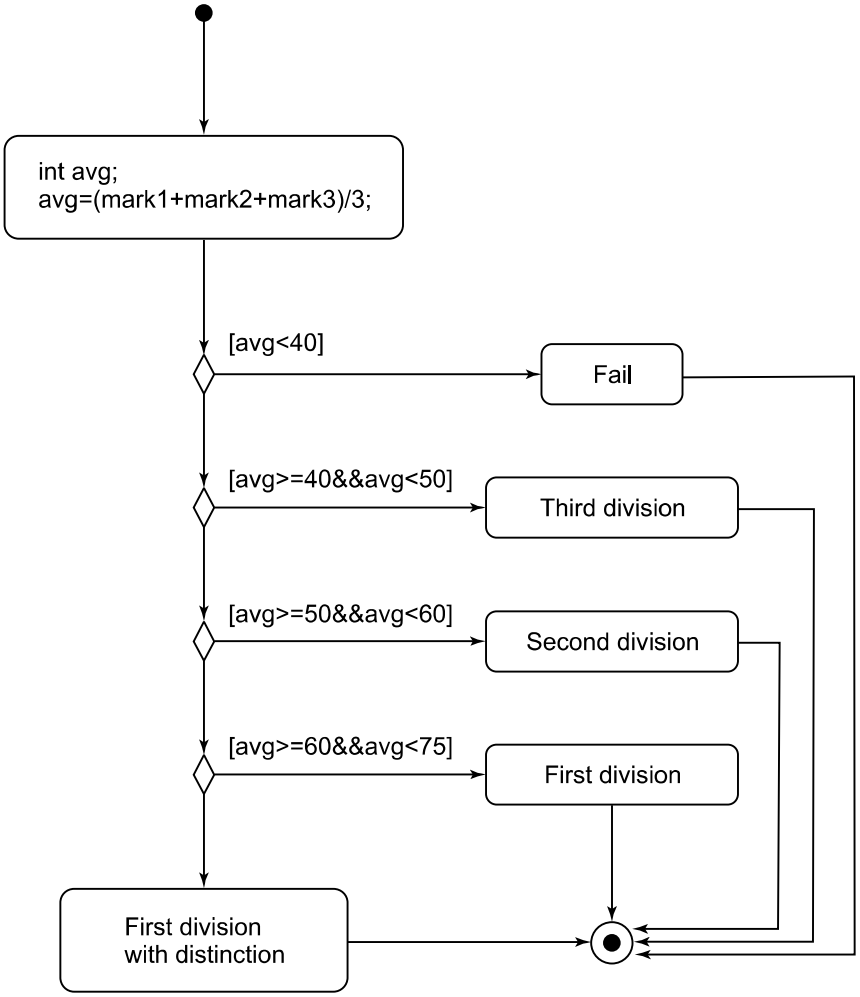


Figure 9.8. Activity diagram of function calculate()

9.3.2 Calculation of Cyclomatic Complexity

As defined earlier in chapter 4, cyclomatic complexity of a graph is given as:

$$V(G)=e-n+2P$$

Where e: number of edges of a graph G

n: number of nodes of a graph G

P: number of connected components

The same concepts of a flow graph are applicable to an activity diagram, for the calculation of cyclomatic complexity. Nodes of a flow graph are represented as branches, activities, initial state and end state in an activity diagram. The edges of a flow graph are represented as transitions in the activity diagram. We may calculate the cyclomatic complexity in the same way and cyclomatic complexity of an activity diagram given in Figure 9.5 is 2. Hence, there are two independent paths in the activity diagram.

We consider the activity diagram given in Figure 9.7 for validate function and cyclomatic complexity is calculated as

Cyclomatic complexity = e-n+2P =transitions – activities/branches +2P

= 5 – 5 +2

= 2

Similarly, for activity diagram for calculate function given in Figure 9.8, cyclomatic complexity is:

Cyclomatic complexity = e-n+2P

= 15 – 12 + 2

= 5

Hence, there are two and five independent paths of validate and calculate functions, respectively.

9.3.3 Generation of Test Cases

After the identification of independent paths, we may generate test cases that traverse all independent paths at the time of executing the program. This process will ensure that each transition of the activity diagram is traversed at least once.

In general, path testing may detect only errors that result from executing a path in the program. It may not be able to detect the errors due to omissions of some important characteristics of the program. It is heavily dependent on the control structure of the program and if we execute all paths (if possible), an effective coverage is achieved. This effective coverage may contribute to the delivery of good quality maintainable software.

Test cases from activity diagrams of validate and calculate functions (refer to figures 9.7 and 9.8) are shown in Table 9.2 and Table 9.3.

Path testing is very useful in object oriented systems. An activity diagram provides a pictorial view of a class which helps us to identify various independent paths. However, as the size of a class increases, design of an activity diagram becomes complex and difficult. This technique is applicable to the classes of reasonable size.

Table 9.2. Test cases for validate function

Test case	mark1	mark2	mark3	Path
1.	101	40	50	Invalid marks
2.	90	75	75	calculate()

Table 9.3. Test cases for calculate function

Test case	mark1	mark2	mark3	Path
1.	40	30	40	Fail
2.	45	47	48	Third division
3.	55	57	60	Second division
4.	70	65	60	First division
5.	80	85	78	First division with distinction

Example 9.1: Consider the program given in Figure 9.9 for determination of the largest amongst three numbers. There are three methods in this program – getdata, validate and maximum. Design test cases for validate and maximum methods of the class using path testing.

```
#include<iostream.h>
#include<conio.h>
class greatest
{
float A;
float B;
float C;
public:
void getdata()
{
cout<<"Enter number 1:\n";
cin>>A;
cout<<"Enter number 2:\n";
cin>>B;
cout<<"Enter number 3:\n";
cin>>C;
}
void validate()
{
if(A<0||A>400||B<0||B>400||C<0||C>400){
     cout<<"Input out of range";
}
else{
maximum();
}
}
void maximum();
};
void greatest::maximum()
{
/*Check for greatest of three numbers*/
if(A>B) {
if(A>C) {
     cout<<A;
```

```
    }
else {
    cout<<C;
    }
}
else {
if(C>B) {
    cout<<C;
    }
else {
    cout<<B;
    }
}
}
void main()
{
clrscr();
greatest g1;
g1.getdata();
g1.validate();
getch();
}
```

Figure 9.9. Program to determine largest among three numbers

Solution:

The activity diagram for validate and calculate functions is given in Figure 9.10 and Figure 9.11 and their test cases are shown in Table 9.4 and Table 9.5.

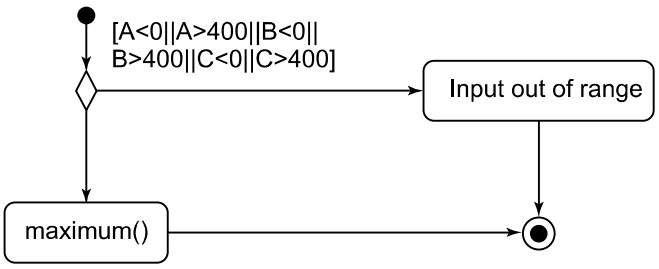


Figure 9.10. Activity diagram for function validate()

Table 9.4. Test cases of activity diagram in Figure 9.10

Test case	A	B	C	Path
1.	500	40	50	Input out of range
2.	90	75	75	maximum()

Cyclomatic complexity = e-n+2P =transitions – activities/branches +2P

= 5 – 5 + 2

= 2

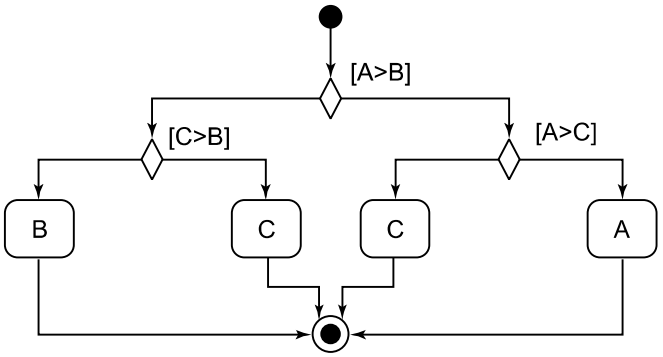


Figure 9.11. Activity diagram for function maximum()

Cyclomatic complexity = e – n + 2P = transitions – activities/branches +2P

11 – 9 + 2 = 4

Table 9.5. Test cases of activity diagram in Figure 9.11

Test case	A	B	C	Expected output
1.	100	87	56	100
2.	87	56	100	100
3.	56	87	100	100
4.	87	100	56	100

9.4 STATE BASED TESTING

State based testing is used as one of the most useful object oriented software testing techniques. It uses the concept of state machine of electronic circuits where the output of the state machine is dependent not only on the present state but also on the past state. A state represents the effect of previous inputs. Hence, in state machine, the output is not only dependent on the present inputs but also on the previous inputs. In electronic circuits, such circuits are called sequential circuits. If the output of a state is only dependent on present inputs, such circuits are called combinational circuits. In state based testing, the resulting state is compared with the expected state.

9.4.1 What is a State Machine?

State machines are used to model the behaviour of objects. A state machine represents various states which an object is expected to visit during its lifetime in response to events or methods

along with its responses to these events or methods. A state is represented by rectangles with rounded corners and transitions are represented by edges (arrows). Events and actions are represented by annotations on the directed edges. A typical state machine is shown in Figure 9.12 and descriptions of its associated terms are given in Table 9.6.

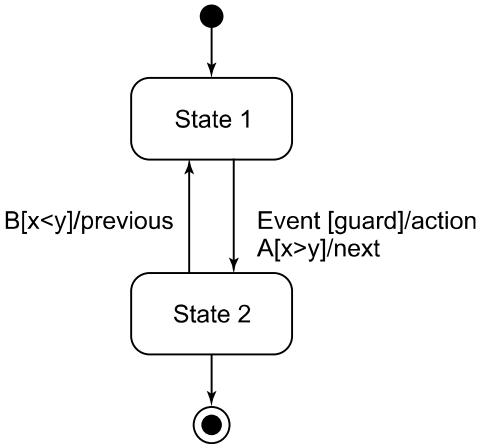


Figure 9.12. A typical state machine diagram

Table 9.6. Terminologies used in state chart diagram

S. No.	Terminologies used in statechart diagram	Description	Remarks
1.	State	Abstract situation in the life cycle of an entity that occurs in response to occurrence of some event.	State1, state2
2.	Event	An input (a message or method call).	A,B
3.	Action	An output or the result of an activity.	Next, previous
4.	Transition	Change of state after occurrence of an event.	When x>y and A is the input, state is changed from state1 to state2
5.	Guard condition	Predicate expression with an event, stating a Boolean restriction for a transition to fire.	Two predicate expressions x>y and x<y

In the Figure 9.12, there are two states – state1 and state2. If at state1, input A is given and (x>y), then state1 is changed to state2 with an output 'next'. At state2, if the input is B and (x<y), then state2 is changed to state1 with an output 'previous'. Hence, a transition transfers a system from one state to another state. The first state is called the accepting state and another is called the resultant state. Both states (accepting and resultant) may also be the same in case of self-loop conditions. The state in question is the current state or present state. Transition occurs from the current state to the resultant state.

We consider an example of a process i.e. program under execution that may have the following states:

» **New:** The process is created
» **Ready:** The process is waiting for the processor to be allocated.
» **Running:** The process is allocated to the processor and is being executed.
» **Time expired:** The time allocated to the process in execution expires.
» **Waiting:** The process is waiting for some I/O or event to occur.
» **Terminated:** The process under execution has completed.

The state machine for life cycle of a process is shown in Figure 9.13. There are six states in the state machine – new, ready, running, time expired, waiting and terminated. The waiting state is decomposed into three concurrent sub-states – I/O operation, child process and interrupt process. The three processes are separated by dashed lines. After the completion of these sub-states the flow of control joins to the ready state.

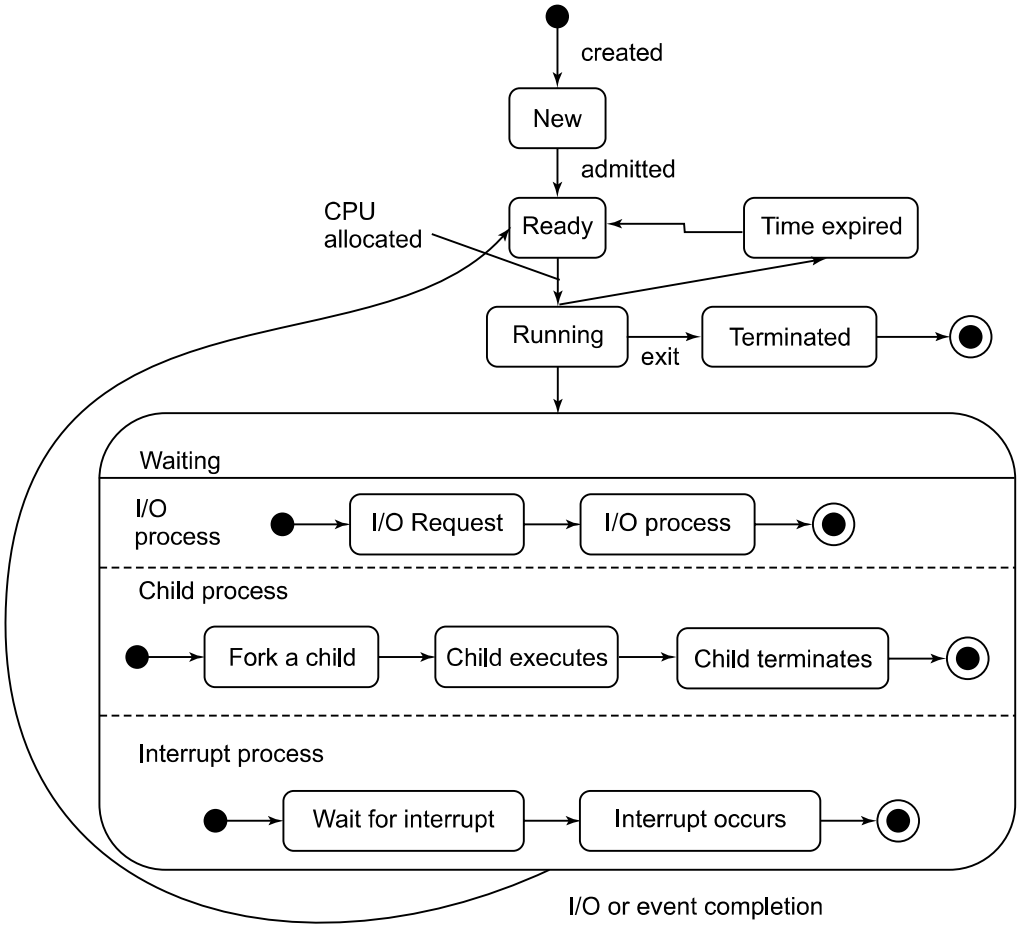


Figure 9.13. Typical life cycle of a process

9.4.2 State Chart Diagram

In Unified Modeling Language (UML), a state machine is graphically represented by a state chart diagram. It shows the flow of control from one state to another state. Here too, states are represented by rectangles with rounded corners and transitions are represented by edges (arrows).

Two special states are used i.e. α (alpha) and ω (omega) state for representing the constructor and destructor of a class. These states may simplify testing of multiple constructors, exception handling and destructors. Binder [BIND99] has explained this concept very effectively as:

> "The α state is a null state representing the declaration of an object before its construction. It may accept only a constructor, new, or a similar initialization message. The ω state is reached after an object has been destructed or deleted, or has gone out of scope. It allows for explicit modeling and systematic testing of destructors, garbage collection, and other termination actions."

Alpha and omega states are different from start state and end state of a state chart diagram. These are additional states to make things more explicit and meaningful.

We consider an example of a class 'stack' where two operations – push and pop, are allowed. The functionality of a stack suggests three states – empty, holding and full. There are four events – new, push, pop and destroy, with the following purposes:

(i) **New:** Creates an empty stack.
(ii) **Push:** Push an element in the stack, if space is available.
(iii) **Pop:** Pop out an element from the stack, if it is available.
(iv) **Destroy:** Destroy the stack after the completion of its requirement i.e. instance of the stack class is destroyed.

The state chart diagram for class stack is given in the Figure 9.14.

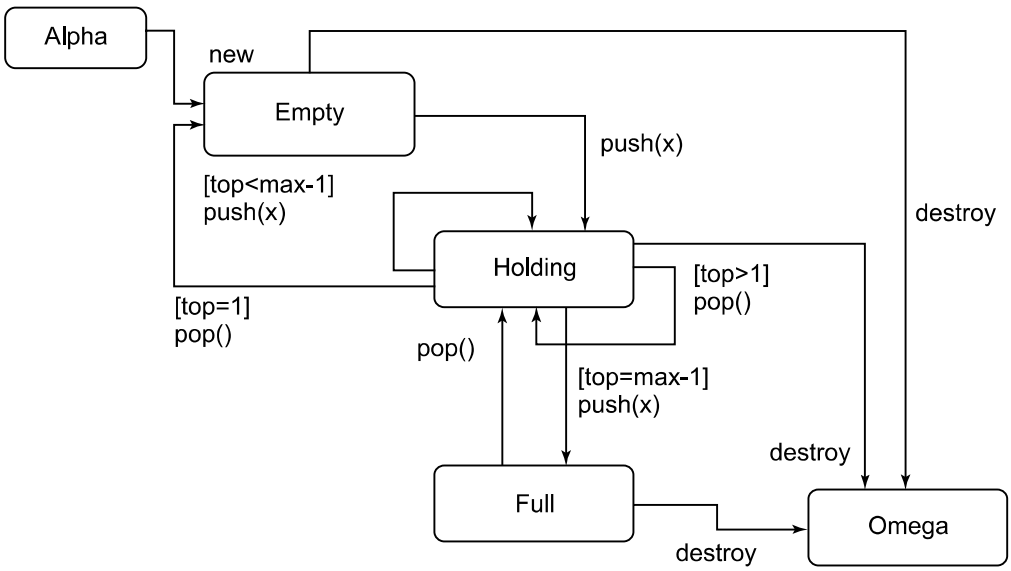


Figure 9.14. State chart diagram for class stack

9.4.3 State Transition Tables

State chart diagrams provide a graphical view of the system and help us to understand the behaviour of the system. Test cases may be designed on the basis of understanding the behaviour. However, drawing a large state chart diagram is difficult, risky and error prone. If

states are more than 10 or 15, it is difficult to keep track of various transitions. In practice, we may have to handle systems with 100 states or more. State transition tables are used when the number of states is more these tables and provide information in a compact tabular form. In state transition tables, rows represent the present acceptable state and columns represent the resultant state. The state transition table of a class stack is given in Table 9.7.

State transition tables represent every transition, event and action and may help us to design the test cases.

Table 9.7. State transition table for stack class

State	Event/method	Resultant state				
		Alpha	Empty	Holding	Full	Omega
Alpha	new		√			
	push(x)					
	pop()					
	destroy					
Empty	new					
	push(x)			√		
	pop()					
	destroy					√
Holding	new					
	push(x)			√	√	
	pop()		√	√		
	destroy					√
Full	new					
	push(x)					
	pop()			√		
	destroy					√

9.4.4 Generation of Test Cases

There are many possible testing strategies for the generation of test cases. We may identify paths from the state chart diagram and execute all of them. This seems to be difficult in practice due to a large number of paths. Another option is to exercise all transitions at least once; this may mean all events' coverage, all states' coverage and all actions' coverage. A state chart diagram and state transition tables may help us to do so and we may be able to generate a good number of systematic and planned test cases. The test cases for stack class are given in Table 9.8. Here we generate test cases for each independent path in the state transition diagram given in Figure 9.12. Some illegal transitions are also shown in Table 9.9 in order to give an idea about undesired actions. What will happen if an illegal event is given to a state? These test cases are also important and may help to find faults in the state chart diagram.

Table 9.8. Test cases for class stack

Test case id	Test case input		Expected result	
	Event (method)	Test condition	Action	State
1.1	New			Empty
1.2	Push(x)			Holding
1.3	Pop()	Top=1	Return x	Empty
1.4	destroy			Omega
2.1	New			Empty
2.2	Push(x)			Holding
2.3	Pop()	Top>1	Return x	holding
2.4	destroy			Omega
3.1	New			Empty
3.2	Push(x)	Top<max-1		Holding
3.3	Push(x)			holding
3.4	destroy			Omega
4.1	New			Empty
4.2	Push(x)			Holding
4.3	Push(x)	Top=max-1		Full
4.4	Pop()			Holding
4.5	destroy			Omega
5.1	New			Empty
5.2	Push(x)			Holding
5.3	Push(x)	Top=max-1		Full
5.4	destroy			omega
6.1	New			empty
6.2	destroy			Omega

Table 9.9. Illegal test case for class stack

Test case id	Test condition		Expected result
	Test state	Test event	Action
7.0	Empty	New	Illegal exception
8.0	Empty	Pop()	Illegal exception
9.0	Holding	New	Illegal exception
10.0	Holding	Push (top=max)	Illegal exception
11.0	Holding	Pop (top=0)	Illegal exception
12.0	Full	New	Illegal exception
13.0	Full	Push	Illegal exception
14.0	Omega	any	Illegal exception

Example 9.2: Consider the example of withdrawing cash from an ATM machine. The process consists of the following steps:

(i) The customer will be asked to insert the ATM card and enter the PIN number.
(ii) If the PIN number is valid, the withdrawal transaction will be performed:
  (a) The customer selects amount.
  (b) The system verifies that it has sufficient money to satisfy the request; then the appropriate amount of cash is dispensed by the machine and a receipt is issued.
  (c) If sufficient amount is not available in the account, a message "Balance not sufficient" is issued.
(iii) If the bank reports that the customer's PIN is invalid, then the customer will have to re-enter the PIN.

Draw a Statechart diagram and generate test cases using state based testing.

Solution:

State chart diagram for withdrawal of cash from an ATM machine is shown in Figure 9.15 and test cases are given in Table 9.10.

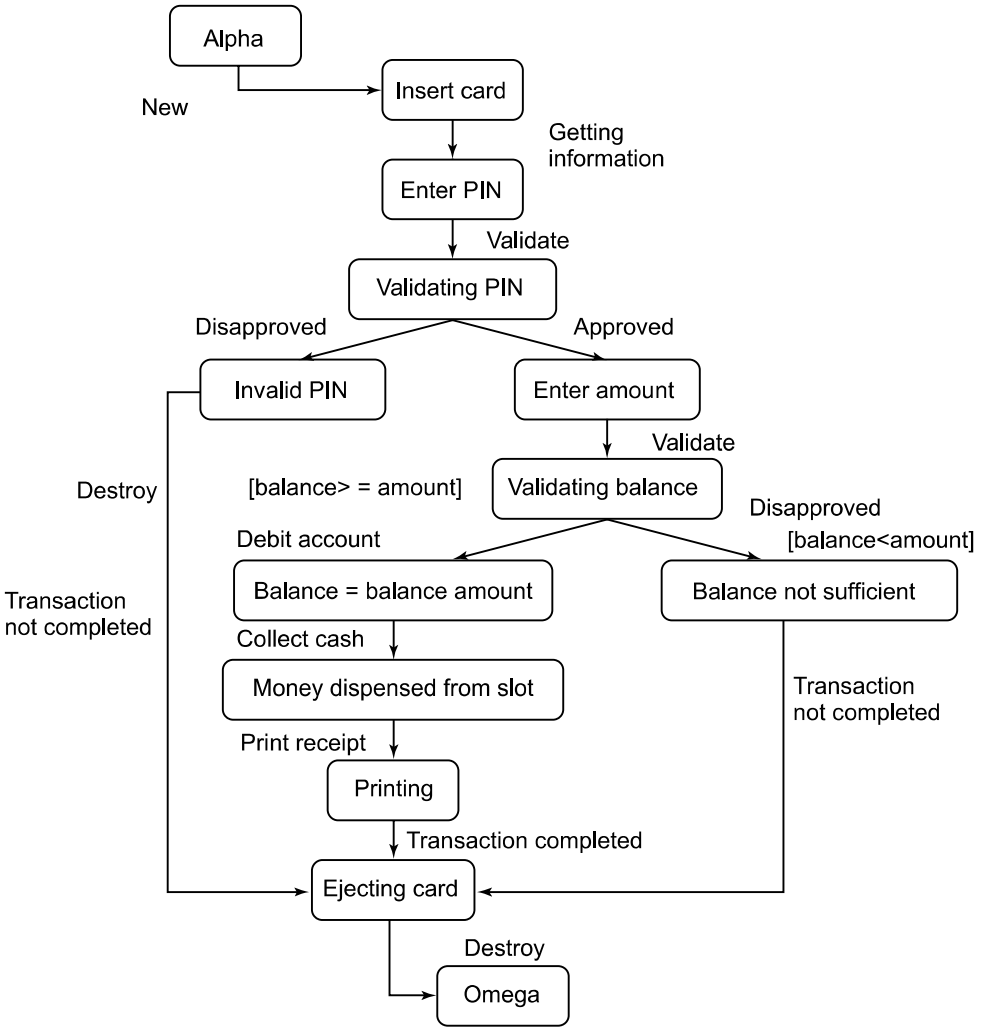


Figure 9.15. State chart diagram of withdrawal from ATM

Table 9.10. Test cases of withdrawal from ATM

Test case ID	Test case input		Expected output	
	Event	Test condition	Action	State
1.1	New			Insert card
1.2	Getting information			Enter pin
1.3	Validate			Validating PIN
1.4	Disapproved			Invalid pin
1.5	Transaction not completed		Collect card	Ejecting card
1.6	Destroy			Omega
2.1	New			Insert card
2.2	Getting information			Enter pin
2.3	Validate			Validating
2.4	Approved			Enter amount
2.5	Validate			Validating balance
2.6	Disapproved	Balance<amount		Balance not sufficient
2.7	Transaction not completed		Collect card	Ejecting card
2.8	Destroy			Omega
3.1	New			Insert card
3.2	Getting information			Enter pin
3.3	Validate			Validating
3.4	Approved			Enter amount
3.5	Validate			Validating balance
3.6	Debit amount	Balance>=amount		Balance=balance-amount
3.7	Collect cash			Money dispensed from slot
3.8	Print receipt		Collect receipt	Printing
3.9	Transaction completed		Collect card	Ejecting card
3.10	Destroy			Omega

9.5 CLASS TESTING

A class is very important in object oriented programming. Every instance of a class is known as an object. Testing of a class is very significant and critical in object oriented testing where we want to verify the implementation of a class with respect to its specifications. If the implementation is as per specifications, then it is expected that every instance of the class may behave in the specified way. Class testing is similar to the unit testing of a conventional system. We require stubs and drivers for testing a 'unit' and sometimes, it may require significant

effort. Similarly, classes also cannot be tested in isolation. They may also require additional source code (similar to stubs and drivers) for testing independently.

9.5.1 How Should We Test a Class?

We want to test the source code of a class. Validation and verification techniques are equally applicable to test a class. We may review the source code during verification and may be able to detect a good number of errors. Reviews are very common in practice, but their effectiveness is heavily dependent on the ability of the reviewer(s).

Another type of testing is validation where we may execute a class using a set of test cases. This is also common in practice but significant effort may be required to write test drivers and sometime this effort may be more than the effort of developing the 'unit' under test. After writing test cases for a class, we must design a test driver to execute each of the test cases and record the output of every test case. The test driver creates one or more instances of a class to execute a test case. We should always remember that classes are tested by creating instances and testing the behaviour of those instances [MCGR01].

9.5.2 Issues Related to Class Testing

How should we test a class? We may test it independently, as a unit or as a group of a system. The decision is dependent on the amount of effort required to develop a test driver, severity of class in the system and associated risk with it and so on. If a class has been developed to be a part of a class library, thorough testing is essential even if the cost of developing a test driver is very high.

Classes should be tested by its developers after developing a test driver. Developers are familiar with the internal design, complexities and other critical issues of a class under test and this knowledge may help to design test cases and develop test driver(s). Class should be tested with respect to its specifications. If some unspecified behaviours have been implemented, we may not be able to test them. We should always be very careful for additional functionalities which are not specified. Generally, we should discourage this practice and if it has been implemented in the SRS document, it should immediately be specified. A test plan with a test suite may discipline the testers to follow a predefined path. This is particularly essential when developers are also the testers.

9.5.3 Generating Test Cases

One of the methods of generating test cases is from pre and post conditions specified in the use cases. As discussed in chapter 6, use cases help us to generate very effective test cases. The pre and post conditions of every method may be used to generate test cases for class testing. Every method of a class has a pre-condition that needs to be satisfied before the execution. Similarly, every method of a class has a post-condition that is the resultant state after the execution of the method. Consider a class 'stack' given in Figure 9.16 with two attributes (x and top) and three methods (stack(), push(x), pop()).

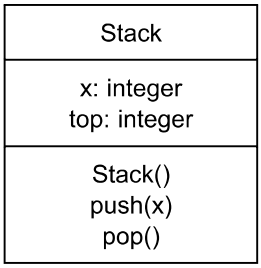


Figure 9.16. Specification for the class stack

We should first specify the pre and post conditions for every operation/method of a class. We may identify requirements for all possible combinations of situations in which a pre-condition can hold and post-conditions can be achieved. We may generate test cases to address what happens when a pre-condition is violated [MCGR01]. We consider the stack class given in Figure 9.16 and identify the following pre and post conditions of all the methods in the class:

(i) Stack::Stack()
  (a) Pre=true
  (b) Post: top=0
(ii) Stack::push(x)
  (a) Pre: top<MAX
  (b) Post: top=top+1
(iii) Stack::pop()
  (a) Pre: top>0
  (b) Post: top=top-1

After the identification of pre and post conditions, we may establish logical relationships between pre and post conditions. Every logical relationship may generate a test case. We consider the push() operation and establish the following logical relationships:

1. (pre condition: top<MAX; post condition: top=top+1)
2. (pre condition: not (top<MAX) ; post condition: exception)

Similarly for pop() operation, the following logical relationships are established:

3. (pre condition: top>0; post condition: top=top-1)
4. (pre condition: not (top>0) ; post condition: exception)

We may identify test cases for every operation/method using pre and post conditions. We should generate test cases when a pre-condition is true and false. Both are equally important to verify the behaviour of a class. We may generate test cases for push(x) and pop() operations (refer Table 9.11 and Table 9.12).

Table 9.11. Test cases of function push()

Test input	Condition	Expected output
23	top<MAX	Element '23' inserted successfully
34	top=MAX	Stack overflow

Table 9.12. Test cases of function pop()

Test input	Condition	Expected output
-	top>0	23
-	top=0	Stack underflow

Example 9.3. Consider the example of withdrawing cash from an ATM machine given in example 9.2. Generate test cases using class testing.

Solution:

The class ATMWithdrawal is given in Figure 9.17.

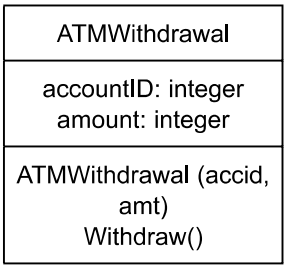


Figure 9.17. Class ATM withdrawal

The pre and post conditions of function Withdraw() are given as:

```
ATMWirthdrawal::Withdraw()
Pre: true
Post: if(PIN is valid) then
      if (balance>=amount) then
         balance=balance-amount
      else
         Display "Insufficient balance"
    else
       Display "Invalid PIN"
(true, PIN is valid and balance>=amount)
(true, PIN is valid and balance<amount)
(true, PIN is invalid)
```

Test cases are given in Table 9.13.

Table 9.13. Test cases for function withdraw()

S. No.	AccountID	Amount	Expected output
1.	4321	1000	Balance update/Debit account
2.	4321	2000	Insufficient balance
3.	4322	-	Invalid PIN