UNIT-1

Introduction & Functional Testing

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.

- **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.

```
/*SOURCE CODE*/
#include<stdio.h>
#include<limits.h>
#include<conio.h>
void Minimum();
void main()
{
        Minimum();
}
void Minimum()
{
        int array[100];
        int Number;
        int i;
        int tmpData;
        int Minimum=INT_MAX;
        clrscr();
        "printf("Enter the size of the array:");
        scanf("%d",&Number);
        for(i=0;i<Number;i++) {
                printf("Enter A[%d]=",i+1);
                scanf("%d",&tmpData);
                tmpData=(tmpData<0)?-tmpData:tmpData;
                array[i]=tmpData;
        }
        i=1;
        while(i<Number-1) {
                if(Minimum>array[i])
                {
                        Minimum=array[i];
                }
                i++;
        }
```

```
        printf("Minimum = %d\n", Minimum);
        getch();
    }
```

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 Size	Inputs Set of Integers	Expected Output	Observed Output	Match?
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: Size	Inputs: Set of Integers	Expected Output	Observed Output	Match?
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 Size	Set of Integers	Expected Output	Observed Output	Match?
Case 7						
A list where some elements are characters	A	5	23, 2I, 26, 6, 9	6	2 (The program does not take any other index value for 3, 4 and 5)	**No**
	B	1I	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) 1 (b) 1 (c)	2147483647 2147483647 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 (d) 1 (e)	14 14	The program has ignored the first and last values of the list. The middle value is 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) 3 (b)	23 23	The program has ignored the first and last values of the list. The value 23 is the minimum value in the remaining list.
4 (a) 4 (b)	2 20	The program has ignored the first and last values. It has also converted negative integer(s) to positive integer(s).
5 (a) 5 (b)	31 56	Same as Case 4.
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.*)

```
        int i;
        int tmpData;
        int Minimum=INT_MAX;
        clrscr();
        printf("Enter the size of the array:");
        scanf("%d",&Number);
        if(Number<=0||Number>100)  {
        printf("Invalid size specified");
        }
        else {
        for(i=0;i<Number;i++) {
                printf("Enter  A[%d]=",i+1);
                scanf("%d",&tmpData);
                /*tmpData=(tmpData<0)?-tmpData:tmpData;*/
                array[i]=tmpData;
        }
        i=0;
        while(i<=Number-1) {
                if(Minimum>array[i])
                {
                        Minimum=array[i];
                }
                i++;
        }
        printf("Minimum = %d\n", Minimum);
        }
        getch();
    }
```

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 Size	Set of Integers	Expected Output	Observed Output	Match?
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**

```
/*SOURCE  CODE*/
#include<stdio.h>
#include<limits.h>
#include<conio.h>
void Minimum();
void main()
{
        Minimum();
}
void Minimum()
{
        int array[100];
        int Number;
        int i;
        int tmpData;
        int Minimum=INT_MAX;
        clrscr();
        printf("Enter the size of the array:");
        scanf("%d",&Number);
        if(Number<=0||Number>100)  {
                printf("Invalid size specified");
        }
        else {
        printf("Warning: The data entered must be a valid integer and
        must be between %d to %d, INT_MIN, INT_MAX\n");
        for(i=0;i<Number;i++) {
                printf("Enter  A[%d]=",i+1);
                scanf("%d",&tmpData);
                /*tmpData=(tmpData<0)?-tmpData:tmpData;*/
                array[i]=tmpData;
        }
        i=0;
        while(i<=Number-1) {
                if(Minimum>array[i])
                {
                        Minimum=array[i];
                }
                i++;
        }
        printf("Minimum = %d\n", Minimum);
        }
        getch();
}
```

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.

- **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.

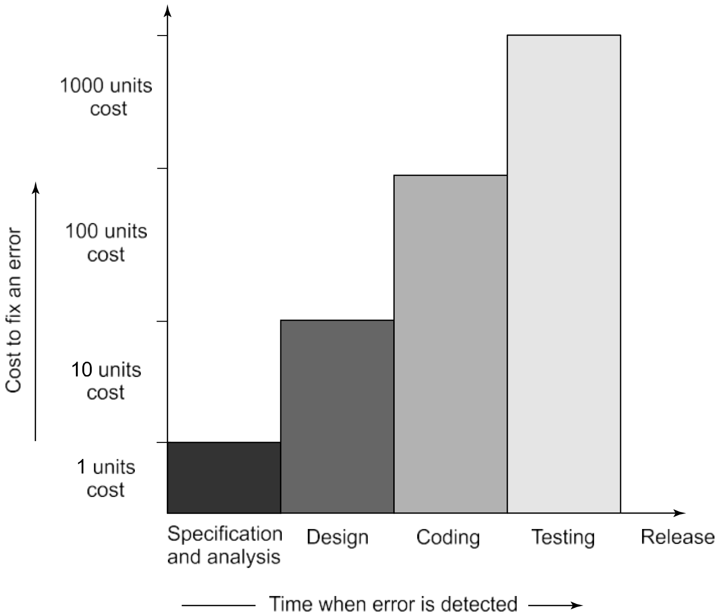


Figure 1.4. Phase wise cost of fixing an error

- ## 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.

• 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.

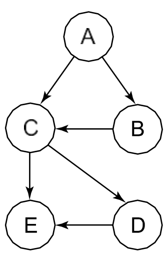
We all know the importance of this area and expect some drastic solutions in the future. We also know that every project is a new project with new expectations, conditions and constraints. What is the bottom line for testing? At least, we may wish to touch this bottom line, which may incorporate the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

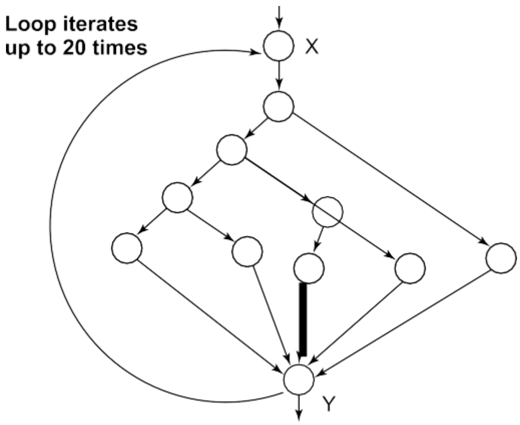


Figure 1.5. Control flow graph of a 10 to 20 statement program [MYER04]

"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.

SOME TERMINOLOGIES

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

- **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.

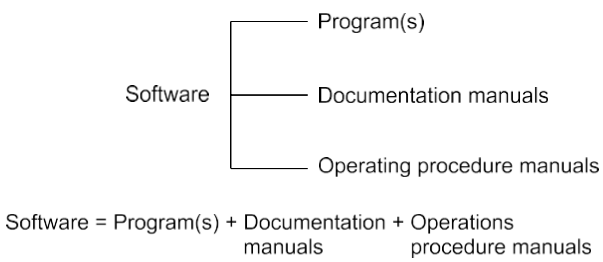


Figure 1.6. Components of the software

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.

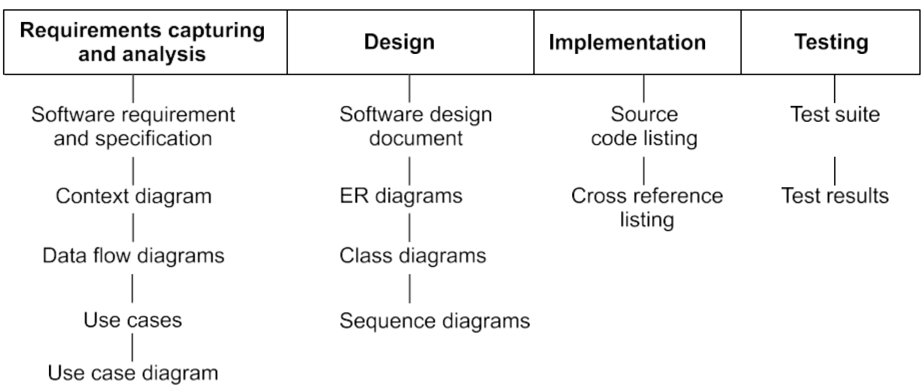


Figure 1.7. Documentation manuals

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.

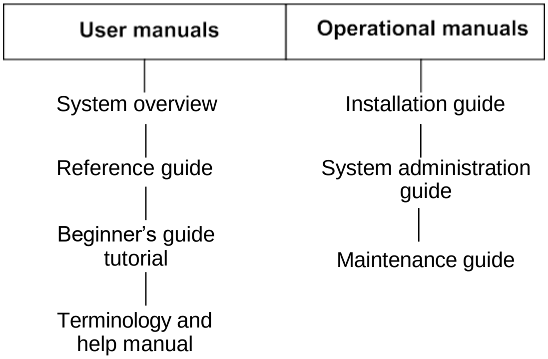


Figure 1.8. Operating system manuals

- ## 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

Testing = Verification + 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.

- ## 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.

- ## 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.

- **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.

- **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.

- **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.

- **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.

- **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.

- **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.

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'.

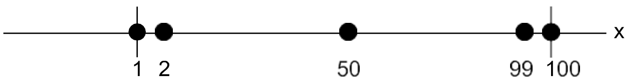


Figure 2.2. Five values for input 'x' of 'Square' program

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 xand 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.

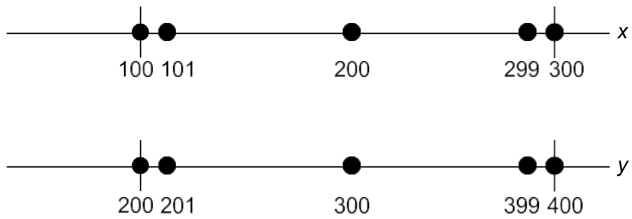


Figure 2.3. Selected values for input values x and y

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.

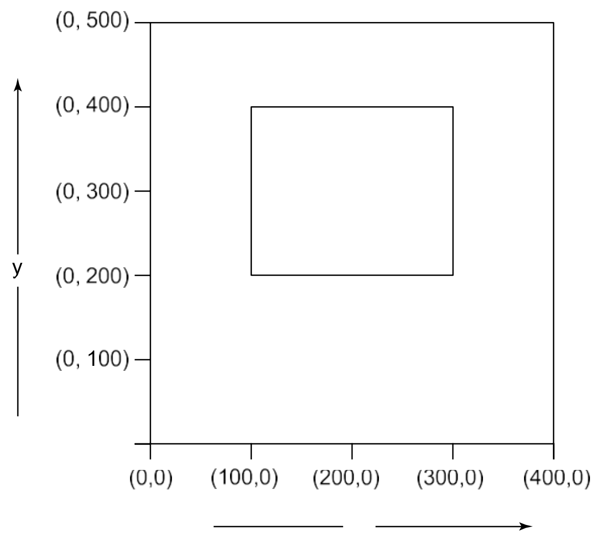


Figure 2.4. Valid input domain for the program 'Addition'

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.

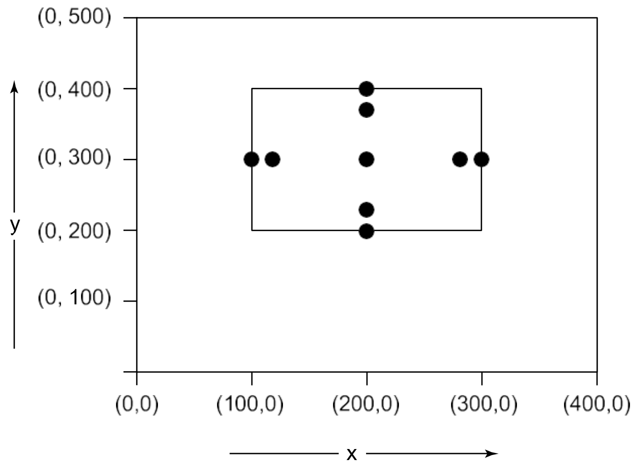


Figure 2.5. Graphical representation of inputs

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.

Average = (mark1 + mark 2 + 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 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

- **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

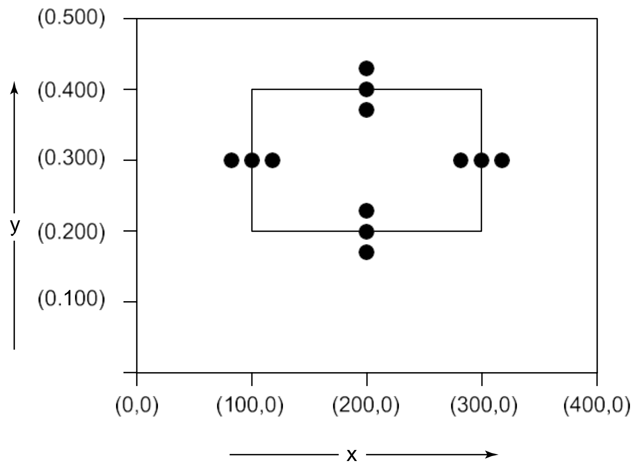


Figure 2.6. Graphical representation of inputs

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

- ## 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.

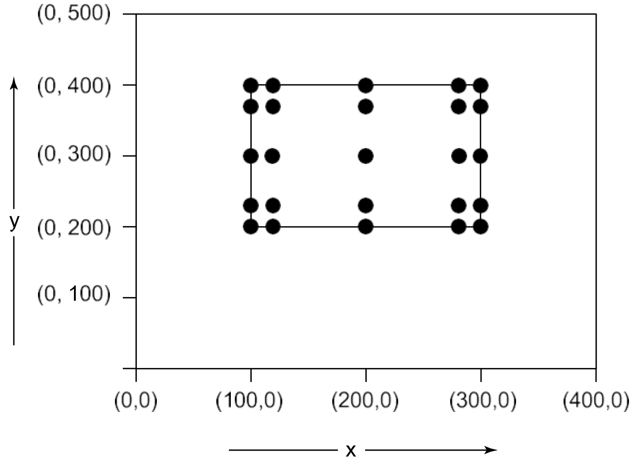


Figure 2.7. Graphical representation of inputs

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].

- **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.

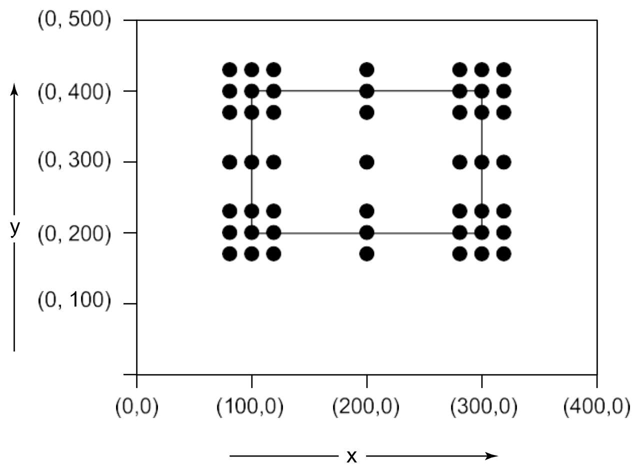


Figure 2.8. Graphical representation of inputs

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

- ### 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

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.

- ### 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.

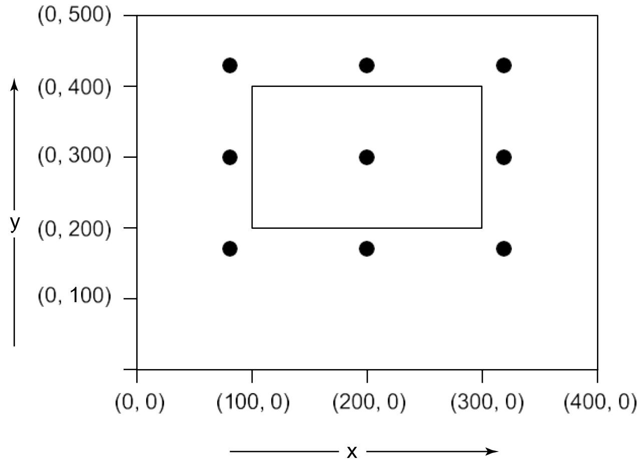


Figure 2.9. Graphical representation of inputs

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.

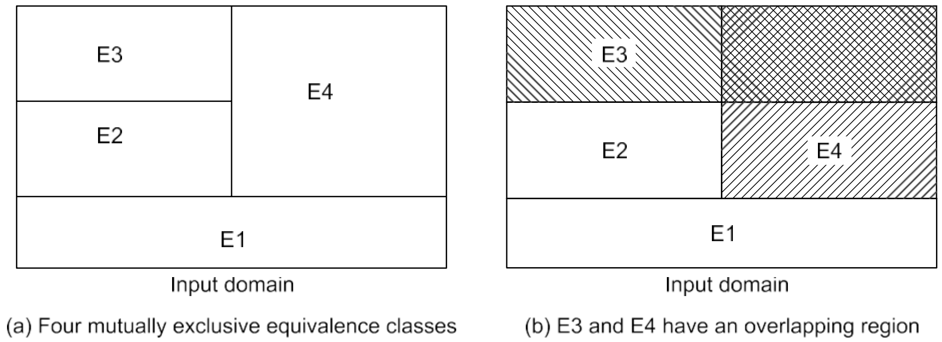


Figure 2.10. Equivalence classes of input domain

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.

- **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:

$I_1 = \{ 1 \le x \le 300 \text{ and } 1 \le y \le 300 \text{ and } 1 \le z \le 300 \}$ (All inputs are valid)

$I_2 = \{ x < 1 \text{ and } 1 \le y \le 300 \text{ and } 1 \le z \le 300 \}$ (x is invalid , y is valid and z is valid)

$I_3 = \{ 1 \le x \le 300 \text{ and } y < 1 \text{ and } 1 \le z \le 300 \}$ (x is valid, y is invalid and z is valid)

$I_4 = \{ 1 \le x \le 300 \text{ and } 1 \le y \le 300 \text{ and } z < 1 \}$ (x is valid, y is valid and z is invalid)

$I_5 = \{ x > 300 \text{ and } 1 \le y \le 300 \text{ and } 1 \le z \le 300 \}$ (x is invalid, y is valid and z is valid)

$I_6 = \{ 1 \le x \le 300 \text{ and } y > 300 \text{ and } 1 \le z \le 300 \}$ (x is valid, y is invalid and z is valid)

$I_7 = \{ 1 \le x \le 300 \text{ and } 1 \le y \le 300 \text{ and } z > 300 \}$ (x is valid, y is valid and z is invalid)

$I_8 = \{ x < 1 \text{ and } y < 1 \text{ and } 1 \le z \le 300 \}$ (x is invalid, y is invalid and z is valid)

$I_9 = \{ 1 \le x \le 300 \text{ and } y < 1 \text{ and } z < 1 \}$ (x is valid, y is invalid and z is invalid)

$I_{10} = \{ x < 1 \text{ and } 1 \le y \le 300 \text{ and } z < 1 \}$ (x is invalid, y is valid and z is invalid)

$I_{11} = \{ x > 300 \text{ and } y > 300 \text{ and } 1 \le z \le 300 \}$ (x is invalid, y is invalid and z is valid)

$I_{12} = \{ 1 \le x \le 300 \text{ and } y > 300 \text{ and } z > 300 \}$(x is valid, y is invalid and z is invalid)

$I_{13} = \{ x > 300 \text{ and } 1 \le y \le 300 \text{ and } z > 300 \}$ (x is invalid, y is valid and z is invalid)

$I_{14} = \{ x < 1 \text{ and } y > 300 \text{ and } 1 \le z \le 300 \}$ (x is invalid, y is invalid and z is valid)

$I_{15} = \{ x > 300 \text{ and } y < 1 \text{ and } 1 \le z \le 300 \}$ (x is invalid, y is invalid and z is valid)

$I_{16} = \{1 \le x \le 300 \text{ and } y < 1 \text{ and } z > 300 \}$ (x is valid, y is invalid and z is invalid)

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

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

I_{19} = { $x > 300$ and $1 \leq y \leq 300$ 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)

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

1_{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 = { $0 \le mark1 \le 100$ and $mark2 < 0$ and $0 \le mark3 \le 100$ } (mark1 is valid, mark2 is invalid and mark3 is valid)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

I_{19} = { $mark1 > 100$ and $0 \le mark2 \le 100$ 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 = { $1 \le a \le 100$ and $1 \le b \le 100$ and $1 \le c \le 100$ } (All inputs are valid)

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

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

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

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

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

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

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

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

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

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

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

I_{13} = { $a > 100$ and $1 \le b \le 100$ 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)

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

1_{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.

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

- ### 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.

- **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].

- **'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.

- **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.

- **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 : $0 \le avg \le 39$?	-	-	-	-	-	T	T	F	T	T	T	F	F	F	F	F	F	F	F	F	F	F
c_8 : $40 \le avg \le 49$?	-	-	-	-	-	-	T	F	F	F	F	T	T	T	T	F	F	F	F	F	F	F
c_9 : $50 \le avg \le 59$?	-	-	-	-	-	-	-	T	F	F	F	T	T	F	F	T	T	T	F	F	F	F
c_{10} : $60 \le avg \le 74$?	-	-	-	-	-	-	-	-	T	F	F	-	F	T	F	T	F	F	T	T	F	F
c_{11} : $avg \ge 75$?	-	-	-	-	-	-	-	-	-	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:

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

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

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

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

I_5 = {B2 : mark2 < 0 }

I_6 = { B3 : mark2 > 100 }

I_7 = { C1 : 0 ≤ mark3 ≤ 100 }

I_8 = { C2 : mark3 < 0 }

I_9 = { C3 : mark3 > 100 }

I_{10} = { D1 : 0 ≤ avg ≤ 39 }

I_{11} = { D2 : 40 ≤ avg ≤ 49 }

I_{12} = { D3 : 50 ≤ avg ≤ 59 }

I_{13} = { D4 : 60 ≤ avg ≤ 74}

I_{14} = { D5 : avg ≥ 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.

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.

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.

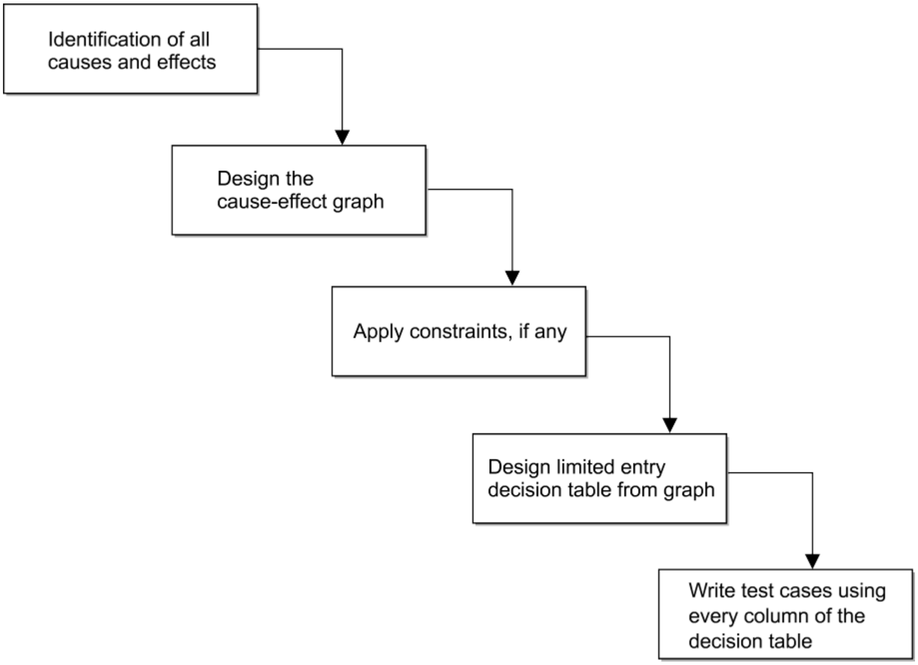


Figure 2.11. Steps for the generation of test cases

- ### 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.

- ### 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.

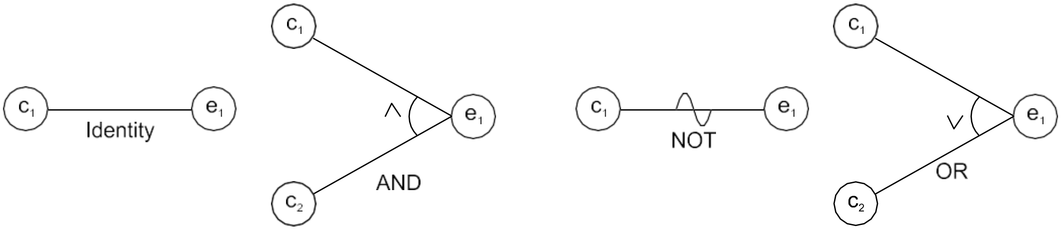


Figure 2.12. Basic notations used in cause-effect graph

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:

(i) Identity: This function states that if c_1 is 1, then e_1 is 1; else e_1 is 0.
(ii) NOT: This function states that if c_1 is 1, then e_1 is 0; else e_1 is 1.
(iii) AND: This function states that if both c_1 and c_2 are 1, then e_1 is 1; else e_1 is 0.
(iv) 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.

- ### 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)(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.

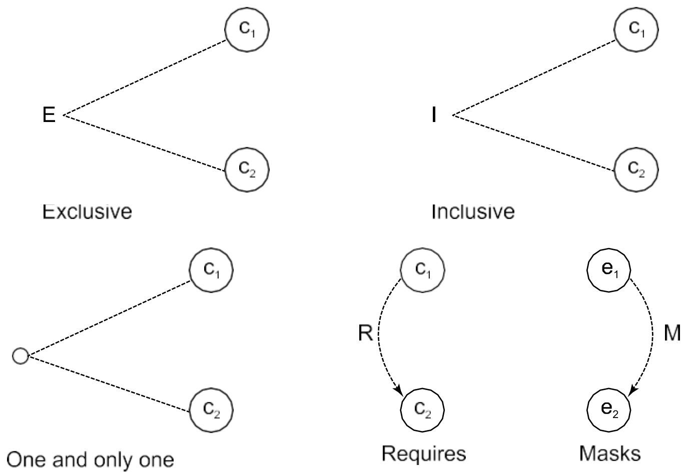


Figure 2.13. Constraint symbols for any cause-effect graph

(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.

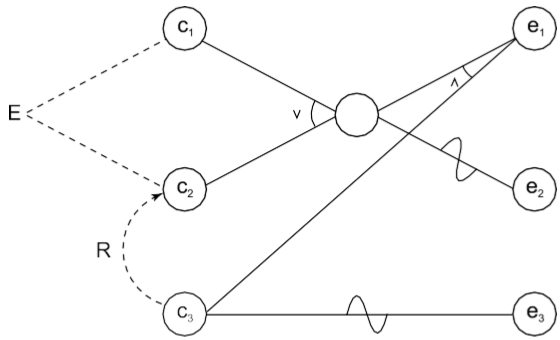


Figure 2.14. Example of cause-effect graph with exclusive (constraint) and requires constraint

- ### 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.

- ### 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.

- ### 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.

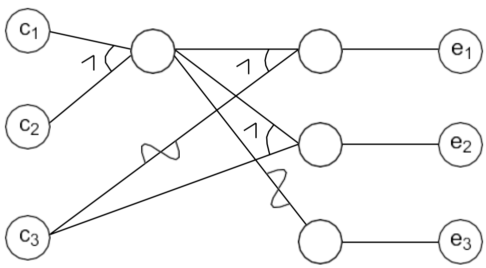


Figure 2.15. Cause-effect graph of rental car problem

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

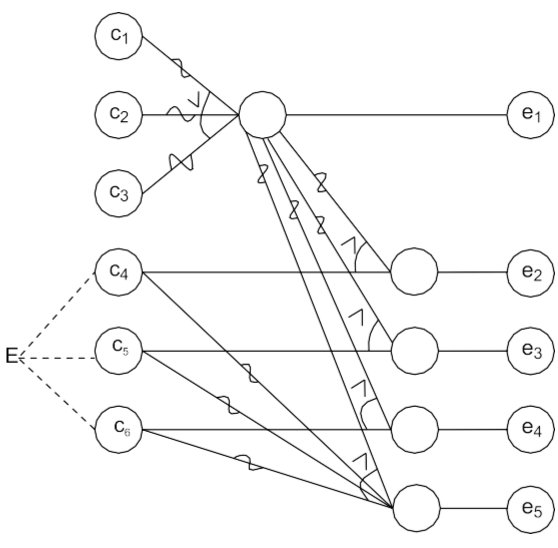


Figure 2.16. Cause-effect graph of triangle classification problem