Software Testing 8

Software testing is an important discipline, and consumes significant, amount of effort. A proper strategy is required to carry out testing activities systematically and effectively. Thus, testing strategy provides a framework or set of activities, which are essential for the success of the project. This may include planning, designing of test cases, execution of program with test cases, interpretation of the outcome and finally collection and management of data.

As we all know, software testing is the process of testing the software product. Effective software testing will contribute to the delivery of higher quality software products, more satisfied users, lower maintenance costs, more accurate, and reliable results. However, ineffective testing will lead to the opposite results; low quality products, unhappy users, increased maintenance costs, unreliable and inaccurate results. Hence, software testing is necessary and important activity of software development process. It is a very expensive process and consumes one-third to one-half of the cost of a typical development project. It is partly intuitive but largely systematic. Good testing involves much more than just running the program a few times to see whether it works. Thorough analysis of a program helps us to test more systematically and more effectively.

8.1 A STRATEGIC APPROACH TO SOFTWARE TESTING

Software testing is a specialised discipline requiring unique skills. Software testing should not be intuitive as for as possible and we must learn how to do it systematically. Naive managers erroneously think that any developer can test software. Some may feel that if we can program, then we can test. What is great in it [TAMR03]? This may not be the right way of thinking.

Software is everywhere. However, it is written by people-so it is not perfect. We have seen many software failures like Intel Pentium Floating Point Division bug of 1994, NASA Mars Polar Lander of 1999, Patriot Missile Defense system of 1991, Y2K Problem etc. [PATT01].

8.1.1 What is Testing?

Many people understand many definitions of testing. Few of them are given below:

1. Testing is the process of demonstrating that errors are not present.
2. The purpose of testing is to show that a program performs its intended functions correctly.
3. Testing is the process of establishing confidence that a program does what it is supposed to do.

These definitions are incorrect. They describe almost the opposite of what testing should be viewed as. Forgetting the definitions for the moment, consider that when we want to test a program, we want to add some value to the program. Adding value means raising the quality or reliability of the program. Raising the reliability of the program means finding and removing errors. Hence, we should not test a program to show that it works; rather we should start with the assumption that the program contains errors and then test the program to find as many of the errors as possible. Thus, a more appropriate definition is:

"Testing is the process of executing a program with the intent of finding errors".

Human beings are normally goal oriented. Thus, establishing the proper goal has an important psychological effect. If our goal is to demonstrate that a program has no errors, then we shall subconsciously steer towards this goal; that is , we will tend to select those inputs that have a low probability of causing the program to fail. On the other hand, if our goal is to demonstrate that a program has errors, our inputs selection will have a higher probability of finding errors. The second approach will add more value to the program than the first one. Thus, testing cannot show the absence of errors, it can only show that errors are present [DAHL72].

According to most appropriate definition, there is a fundamental entity "errors are present within the software under test". This cannot be the aim of software designers. They must have designed the software with the aim of producing it with zero errors. Therefore, whole effort of software engineering activities is to design methods and tools to eliminate errors at source. Software testing is becoming increasingly important in the earlier part of the software life cycle, aiming to discover errors before they are deeply embedded within systems. It is to be hoped that one-day software engineering will become refined to the degree that software testing will be fully integrated within each phase of software life cycle. After all, engineers building bridges do not need to test their products to destruction to predict the breaking point of their constructs. For the moment, in software, this is the only practical method open to us.

In software testing we are facing a major dilemma. On the one hand we wish to design the software product that has zero errors while on the other hand we must remain firm in our belief that any software product under testing certainly has errors, which need to be unearthed.

8.1.2 Why should we Test?

Although software testing is itself an expensive activity, yet launching of software without testing may lead to cost potentially much higher than that of testing, specially in systems where human safety is involved. No one would think of allowing automatic pilot software into service without the most rigorous testing. In so-called life critical systems, economics must not be the prime consideration while deciding whether a product should be released to a customer.

In most systems, however, it is the cost factor, which plays a major role. It is both the driving force and the limiting factor as well. In the software life cycle the earlier the errors are discovered and removed, the lower is the cost of their removal. The most damaging errors are those, which are not discovered during the testing process and therefore remain when the system 'goes live'. In commercial systems it is often difficult to estimate the costs of errors. For example, in a banking system, the potential cost of even a minor software error could be enormous. The consequential cost of lost business (which may never be recovered) can be beyond calculations.

It is not possible to test the software for all possible combinations of input cases. No software would ever be released by its creators if they were asked to certify that it was totally free of all errors. Testing therefore continues to the point where it is considered that the costs of the testing processes significantly outweigh the returns. Hence, when to release the software in the market, is a very important decision.

8.1.3 Who should do the Testing?

The testing requires the developers to find errors from their software. It is very difficult for software developer to point out errors from own creations. Beizer [BEIZ90] 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 only we could really concentrate, if everyone used structured programming, top-down design, decision tables, 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 failure, which instils a goodly dose of guilt. And the tedium of testing is just punishment for our errors. Punishment for what? For being human? Guilt for what? For not achieving inhuman perfection? For not distinguishing 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?"

Many organisations have made a distinction between development and testing phase 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, were we testing our own software [NORM89]. Therefore, most of the times, testing persons are different from development persons for the overall benefit of the system. Developers provide guidelines during testing, however, whole responsibility is owned by testing persons.

8.1.4 What should we Test?

We should test the program's responses to every possible input. It means, we should test for all valid and invalid inputs. Suppose a program requires two 8 bit integers as inputs. Total possible combinations are $2^8 \times 2^8$. If only one second is required to execute one set of inputs, it may take 18 hours to test all combinations. Practically, inputs are more than two and size is also more than 8 bits. We have also not considered invalid inputs where so many combinations are possible. Hence, complete testing is just not possible, although, we may wish to do so.

Another dimension is to execute all possible paths of the program. A program path can be traced through the code from the start of the program to program termination. Two paths differ if the program executes different statements in each, or executes the same statements but in different order. A program may have many paths. Myers has explained this problem with a simple example [MYER79] where he used a loop and few. IF statements as shown in Fig. 8.1.

The number of paths in the example of Fig. 8.1 are 10^{14} or 100 trillions. It is computed from $5^{20} + 5^{19} + 5^{18} + ... + 5^{1}$; where 5 is the number of paths through the loop body. If only minutes are required to test one path, it may take approximately one billion years to execute every path.

The point which we would like to highlight is that complete or exhaustive testing is just not possible. Exhaustive testing requires every statement in the program and every possible path combination to be executed at least once. So our objective is not possible to be achieved and we may have to settle for something less than that of complete testing.

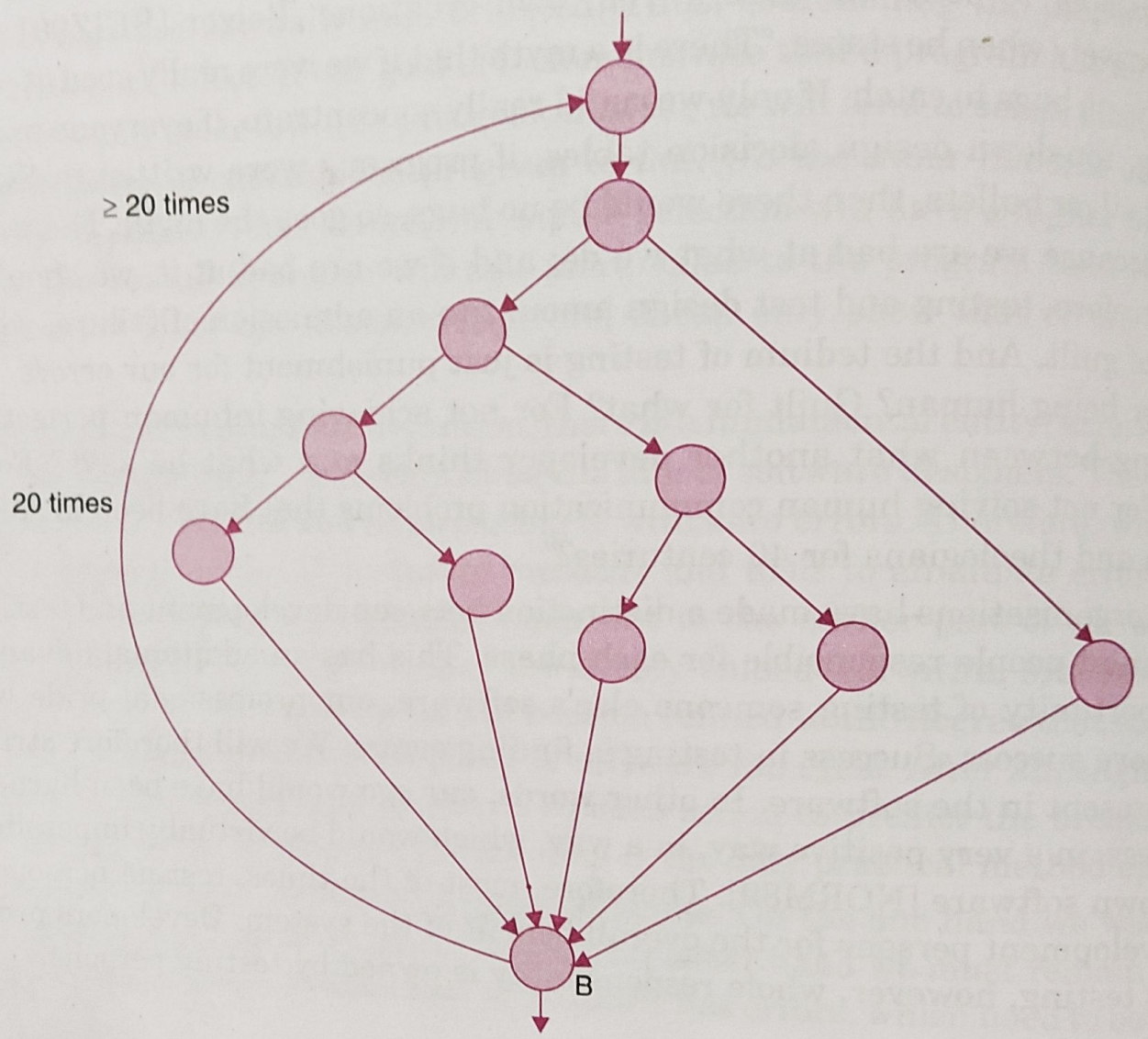

Fig. 8.1: Control flow graph [MYER79]

We may like to test those areas where probability of getting a fault is maximum. Such critical and sensitive areas are not easy to identify. Organizations should develop strategies and policies for choosing effective testing techniques rather than leaving this to arbitrary judgements of the development team.

A strategy should develop test cases for the testing of small portion of program and also develop test cases for complete system or a particular function.

8.2 SOME TERMINOLOGIES

Some terminologies are confusing and used interchangeably in literature and books. Institute of Electronics and Electrical Engineers (IEEE), USA has developed some standards which are discussed as:

8.2.1 Error, Mistake, Bug, Fault and Failure

People make errors. A good synonym is mistake. This may be a syntax error or misunderstanding of specifications. Sometimes, there are logical errors. When developers make mistakes while coding, we call these mistakes "bugs". Errors propogate from one phase to another with higher severity. A requirement error may be magnified during design, and amplified still more during coding. If it could not be detected prior to release, it may have serious implications in the field.

An error may lead to one or more faults. It is more precise to say that a fault is the representation of an error, where representation is the mode of expression, such as narrative text, data flow diagrams, ER diagrams, source code etc. Defect is a good synonym for fault. If fault is in source code, we call it a bug.

A failure occurs when a fault executes. It is the departure of the output of program from the expected output. Hence failure is dynamic. 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 how it has been exercised.

8.2.2 Test, Test Case and Test Suite

Test and Test case terms are used interchangeably. In practice, both are same and are treated as synonyms. Test case describes an input description and an expected output description. Inputs are of two types: pre conditions (circumstances that hold prior to test case execution) and the actual inputs that are identified by some testing methods. Expected outputs are also of two types: post conditions and actual outputs. Every test case will have an identification.

During testing, we set necessary preconditions, give required inputs to program, and compare the observed output with expected output to know the outcome of a test case. If expected and observed outputs are different, then, there is a failure and it must be recorded properly in order to identify the cause of failure. If both are same, then, there is no failure and program behaved in the expected manner. A good test case has a high probability of finding an error. The test case designer's main objective is to identify good test cases. The template for a typical test case is given in Fig. 8.2.

Test case ID:	
Section-I (Before execution)	*Section-II (After execution)*
Purpose:	Execution History:
Pre condition: (If any)	Result:
Inputs:	If fails, any possible reason (Optional):
Expected Outputs:	Any other observation:
Post conditions:	Any suggestion:
Written by:	Run by:
Date:	Date:

Fig. 8.2: Test case template

Test cases are valuable and useful—at least as valuable as source code. They need to be developed, reviewed, used, managed, and saved.

The set of test cases is called a test suite. We may have a test suite of all possible test case. We may have a test suite of effective/good test cases. Hence any combination of test cases may generate a test suite.

8.2.3 Verification and Validation

These terms are often used interchangeably but have different meanings. IEEE has given the definitions of both these which are being widely accepted by the software engineering community. Verification is primarily related to manual testing, because it requires looking at documents and reviewing them. However, validation usually requires the execution of program.

Verification: As per IEEE/ANSI, "it is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase". Hence verification activities are applied to early phases of SDLC such as requirements, design, planning etc. We check or review the documents generated after the completion of every phase in order to ensure that what comes out of that phase is what we expected to get.

Validation: As per IEEE/ANSI, "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." Therefore, validation requires actual execution of the program and is also known as computer based testing. We experience failures and identify the causes of the failures.

Hence, testing includes both verification and validation.

Testing = Verification + Validation.

Both are important and complementary to each other. Varification minimises the errors and their impact in the early phases of development. If we find more errors before execution (due to verification of the program), validation may be comparatively easy. Unfortunately, testing is primarily validation oriented.

8.2.4 Alpha, Beta and Acceptance Testing

It is not possible to predict the every usage of the software by the customer. Customer may try with strange inputs, combination of inputs and so many other things. Some output may be very clear from the developer's perspective, but customer may not understand and finally may not appriciate it. In order to avoid or minimise such situations, customer involvement is required before delivering the final product. The above mentioned three terms are related to customer's involvement in testing but have different meanings.

Acceptance testing

This term is used when the software is developed for a specific customer. A series of tests are conducted to enable the customer to validate all requirements. These tests are conducted by the end user/customer and may range from adhoc tests to well planned systematic series of tests. Acceptance testing may be conducted for few weeks or months. The discovered errors will be fixed and better quality software will be delivered to the customer.

Alpha and beta testing

The terms alpha and beta testing are used when the software is developed as a product for anonymous customers. Hence formal acceptance testing is not possible in such cases. However, some potential customers are identified to get their views about the product. The alpha tests are conducted at the developer's site by a customer. These tests are conducted in a controlled environment. Alpha testing may be started when formal testing process is near completion.

The beta tests are conducted by the customers/end users at their sites. Unlike alpha testing, developer is not present here. Beta testing is conducted in a real environment that cannot be controlled by the developer. Customers are expected to report failures, if any, to the company. After receiving such failure reports, developers modify the code and fix the bug and prepare the product for final release.

Most of the companies are following this practice firstly, they send the beta release of their product for few months. Many potential customers will use the product and may send their views about the product. Some may encounter with failure situations and may report to the company. Hence, company gets the feedback of many potential customers. The best part is that the reputation of the company is not at stake even if many failure situations are encountered.

8.3 FUNCTIONAL TESTING

As discussed earlier, complete testing is not at all possible. Thus, we may like to reduce this incompleteness as much as possible. Probably the poorest methodology is random input testing. In random input testing, some subset of all input values are selected randomly. In terms of probability of detecting errors, a randomly selected collection of test cases has little chance of being an optimal, or close to optimal, subset. What we are looking for is a set of thought processes that allow us to select a set of data more intelligently.

One way to examine this issue is to explore a strategy where testing is based on the functionality of the program and is known as functional testing. Thus, functional testing refers to testing, which involves only observation of the output for certain input values. There is no attempt to analyse the code, which produces the output. We ignore the internal structure of the code. Therefore, functional testing is also referred to as black box testing in which contents of the black box are not known. Functionality of the black box is understood completely in terms of its inputs and outputs as shown in Fig. 8.3. Here, we are interested in functionality rather than internal structure of the code. Many times we operate more effectively with black box knowledge. For example, most people successfully operate automobiles with only black box knowledge.

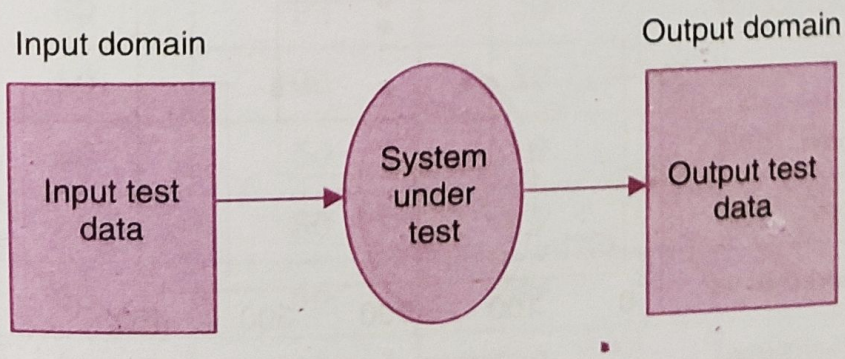

Fig. 8.3: Black box testing

There are a number of strategies or techniques that can be used to design test cases which have been found to be very successful in detecting errors.

8.3.1 Boundary Value Analysis

Experience shows that test cases that are close to boundary conditions have a higher chance of detecting an error. Here boundary condition means, an input value may be on the boundary, just below the boundary (upper side) or just above the boundary (lower side). Suppose we have an input variable x with a range from 1–100. The boundary values are 1, 2, 99 and 100.

Consider a program with two input variables x and y. These input variables have specified boundaries as:

$$a \le x \le b$$
$$c \le y \le d$$

Hence both the inputs x and y are bounded by two intervals $[a, b]$ and $[c, d]$ respectively. For input x, we may design test cases with values a and b, just above a and also just below b. Similarly for input y, we may have values c and d, just above c and also just below d. These test cases will have more chances to detect an error [JORG95]. The input domain for our program is shown in Fig. 8.4. Any point within the inner rectangle is a legitimate input to the program.

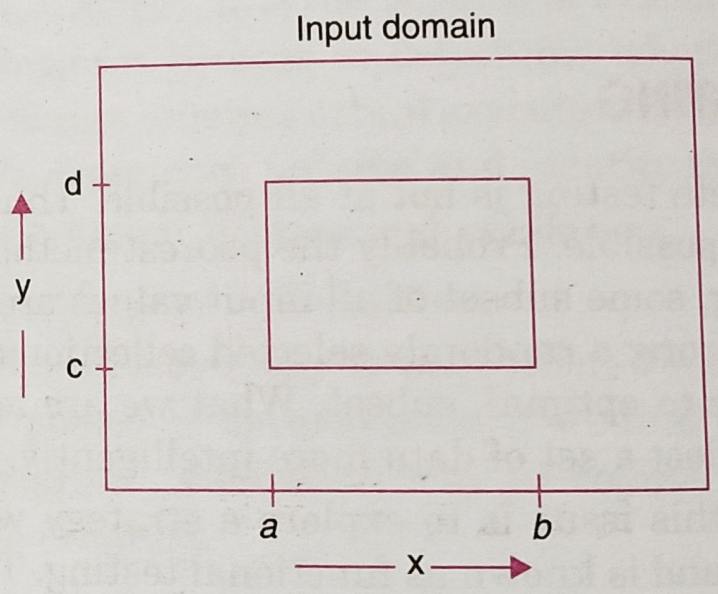

Fig. 8.4: Input domain for program having two input variables

The basic idea of boundary value analysis is to use input variable values at their minimum, just above minimum, a nominal value, just below their maximum, and at their maximum.

Here, we have an assumption of reliability theory known as "single fault" assumption

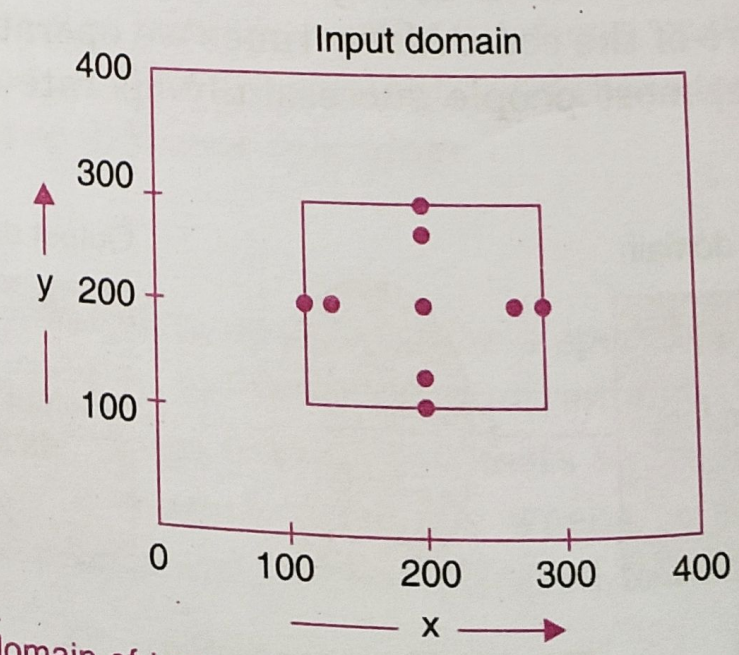

Fig. 8.5: Input domain of two variables x and y with boundaries [100, 300] each

This says that failures are rarely the result of the simultaneous occurrence of two (or more) faults. Thus, boundary value analysis test cases are obtained by holding the values of all but one variable at their nominal values and letting that variable assume its extreme values. The boundary value analysis test cases for our program with two inputs variables (x and y) that may have any value from 100–300 are: (200, 100), (200, 101), (200, 200), (200, 299), (200, 300), (100, 200), (101, 200), (299, 200) and (300, 200). This input domain is shown in Fig. 8.5. Each dot represent a test case and inner rectangle is the domain of legitimate inputs. Thus, for a program of n variables, boundary value analysis yields $4n + 1$ test cases.

Example 8.1

Consider a program for the determination of the nature of roots of a quadratic equation. Its input is a triple of positive integers (say a, b, c) and values may be from interval [0, 100]. The program output may have one of the following words:

[Not a quadratic equation; Real roots; Imaginary roots; Equal roots]

Design the boundary value test cases.

Solution

Quadratic equation will be of type:

$$ax^2 + bx + c = 0$$

Roots are real if $(b^2 - 4ac) > 0$

Roots are imaginary if $(b^2 - 4ac) < 0$

Roots are equal if $(b^2 - 4ac) = 0$

Equation is not quadratic if $a = 0$

The boundary value test cases are:

Test case	*a*	*b*	*c*	*Expected output*
1	0	50	50	Not Quadratic
2	1	50	50	Real Roots
3	50	50	50	Imaginary Roots
4	99	50	50	Imaginary Roots
5	100	50	50	Imaginary Roots
6	50	0	50	Imaginary Roots
7	50	1	50	Imaginary Roots
8	50	99	50	Imaginary Roots
9	50	100	50	Equal Roots
10	50	50	0	Real Roots
11	50	50	1	Real Roots
12	50	50	99	Imaginary Roots
13	50	50	100	Imaginary Roots

Example 8.2

Consider a program for determining the Previous date. Its input is a triple of day, month and year with the values in the range

$$1 \leq \text{month} \leq 12$$
$$1 \leq \text{day} \leq 31$$
$$1900 \leq \text{year} \leq 2025$$

The possible outputs would be Previous date or invalid input date. Design the boundary value test cases.

Solution

The Previous date program takes a date as input and checks it for validity. If valid, it returns the previous date as its output.

As we know, with single fault assumption theory, $4n + 1$ test cases can be designed which are equal to 13. The boundary value test cases are:

Test case	*Month*	*Day*	*Year*	*Expected output*
1	6	15	1900	14 June, 1900
2	6	15	1901	14 June, 1901
3	6	15	1962	14 June, 1962
4	6	15	2024	14 June, 2024
5	6	15	2025	14 June, 2025
6	6	1	1962	31 May, 1962
7	6	2	1962	1 June, 1962
8	6	30	1962	29 June, 1962
9	6	31	1962	Invalid date
10	1	15	1962	14 January, 1962
11	2	15	1962	14 February, 1962
12	11	15	1962	14 November, 1962
13	12	15	1962	14 December, 1962

Example 8.3

Consider a simple program to classify a triangle. Its input is a triple of positive integers (say x, y, z) and the data type for input parameters ensures that these will be integers greater than 0 and less than or equal to 100. The program output may be one of the following words:

[Scalene; Isosceles; Equilateral; Not a triangle]

Design the boundary value test cases.

Solution

The boundary value test cases are shown below:

Test case	x	y	z	Expected output
1	50	50	1	Isosceles
2	50	50	2	Isosceles
3	50	50	50	Equilateral
4	50	50	99	Isosceles
5	50	50	100	Not a triangle
6	50	1	50	Isosceles
7	50	2	50	Isosceles
8	50	99	50	Isosceles
9	50	100	50	Not a triangle
10	1	50	50	Isosceles
11	2	50	50	Isosceles
12	99	50	50	Isosceles
13	100	50	50	Not a triangle

Robustness testing

It is nothing but the extension of boundary value analysis. Here, we would like to see, what happens when the extreme values are exceeded with a value slightly greater than the maximum and a value slightly less than minimum. It means, we want to go outside the legitimate boundary of input domain. This type of testing is quite common in electric and electronic circuits. This extended form of boundary value analysis is called robustness testing and shown in Fig. 8.6.

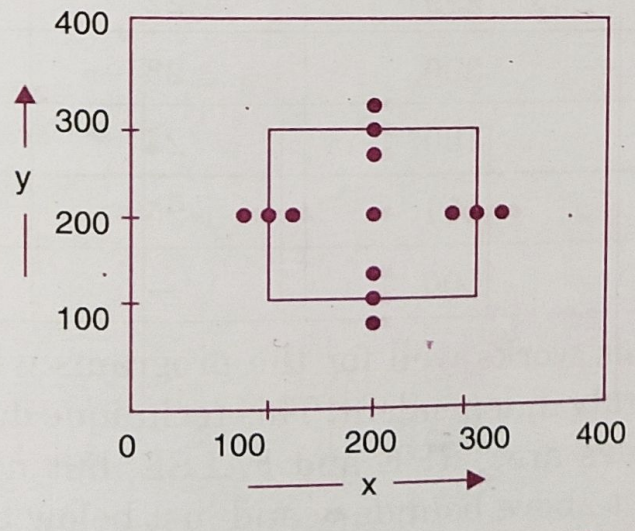

Fig. 8.6: Robustness test cases for two variables *x* and *y* with range [100, 300] each

There are four additional test cases which are outside the legitimate input domain. Hence total test cases in robustness testing are $6n + 1$, where n is the number of input variables. So, 13 test cases are:

(200, 99), (200, 100), (200, 101), (200, 200), (200, 299), (200, 300), (200, 301), (99, 200), (100, 200), (101, 200), (299, 200), (300, 200), (301, 200)

Worst-case testing

If we reject "single fault" assumption theory of reliability and may like to see what happens when more than one variable has an extreme value. In electronic circuits analysis, this is called "worst case analysis". It is more thorough in the sense that boundary value test cases are a proper subset of worst case test cases. It requires more effort. Worst case testing for a function of n variables generates 5^n test cases as opposed to $4n + 1$ test cases for boundary value analysis. Our two variable example will have $5^2 = 25$ test cases and are given in Table 8.1.

Table 8.1: Worst case test inputs for two variable example.

Test case number	Inputs		Test case number	Inputs	
	x	y		x	y
1	100	100	14	200	299
2	100	101	15	200	300
3	100	200	16	299	100
4	100	299	17	299	101
5	100	300	18	299	200
6	101	100	19	299	299
7	101	101	20	299	300
8	101	200	21	300	100
9	101	299	22	300	101
10	101	300	23	300	200
11	200	100	24	300	299
12	200	101	25	300	300
13	200	200	—		

Boundary value analysis works well for the programs with independent input variables. Here input values should be truly independent. This technique does not make sense for boolean variables where extreme values are TRUE and FALSE, but no clear choice is available for other values like nominal, just above boundary and just below boundary.

Example 8.4

Consider the program for the determination of nature of roots of a quadratic equation as explained in Example 8.1. Design the Robust test cases and worst test cases for this program.

Solution

As we know, robust test cases are $6n + 1$. Hence, in 3 variable input cases total number of test cases are 19 as given below:

Test case	*a*	*b*	*c*	*Expected output*
1	– 1	50	50	Invalid input
2	0	50	50	Not quadratic equation
3	1	50	50	Real roots
4	50	50	50	Imaginary roots
5	99	50	50	Imaginary roots
6	100	50	50	Imaginary roots
7	101	50	50	Invalid input
8	50	– 1	50	Invalid input
9	50	0	50	Imaginary roots
10	50	1	50	Imaginary roots
11	50	99	50	Imaginary roots
12	50	100	50	Equal roots
13	50	101	50	Invalid input
14	50	50	– 1	Invalid input
15	50	50	0	Real roots
16	50	50	1	Real roots
17	50	50	99	Imaginary roots
18	50	50	100	Imaginary roots
19	50	50	101	Invalid input

In case of worst test case total test cases are 5^n. Hence, 125 test cases will be generated in worst test cases. The worst test cases are given below:

Test case	*a*	*b*	*c*	*Expected output*
1	0	0	0	Not quadratic
2	0	0	1	Not quadratic
3	0	0	50	Not quadratic
4	0	0	99	Not quadratic
5	0	0	100	Not quadratic
6	0	1	0	Not quadratic

(Contd.)...

Test case	a	b	c	Expected output
7	0	1	1	Not quadratic
8	0	1	50	Not quadratic
9	0	1	99	Not quadratic
10	0	1	100	Not quadratic
11	0	50	0	Not quadratic
12	0	50	1	Not quadratic
13	0	50	50	Not quadratic
14	0	50	99	Not quadratic
15	0	50	100	Not quadratic
16	0	99	0	Not quadratic
17	0	99	1	Not quadratic
18	0	99	50	Not quadratic
19	0	99	99	Not quadratic
20	0	99	100	Not quadratic
21	0	100	0	Not quadratic
22	0	100	1	Not quadratic
23	0	100	50	Not quadratic
24	0	100	99	Not quadratic
25	0	100	100	Not quadratic
26	1	0	0	Equal roots
27	1	0	1	Imaginary
28	1	0	50	Imaginary
29	1	0	99	Imaginary
30	1	0	100	Imaginary
31	1	1	0	Real roots
32	1	1	1	Imaginary
33	1	1	50	Imaginary
34	1	1	99	Imaginary
35	1	1	100	Imaginary
36	1	50	0	Real roots

(Contd.)

Test case	a	b	c	Expected output
37	1	50	1	Real roots
38	1	50	50	Real roots
39	1	50	99	Real roots
40	1	50	100	Real roots
41	1	99	0	Real roots
42	1	99	1	Real roots
43	1	99	50	Real roots
44	1	99	99	Real roots
45	1	99	100	Real roots
46	1	100	0	Real roots
47	1	100	1	Real roots
48	1	100	50	Real roots
49	1	100	99	Real roots
50	1	100	100	Real roots
51	50	0	0	Equal roots
52	50	0	1	Imaginary
53	50	0	50	Imaginary
54	50	0	99	Imaginary
55	50	0	100	Imaginary
56	50	1	0	Real roots
57	50	1	1	Imaginary
58	50	1	50	Imaginary
59	50	1	99	Imaginary
60	50	1	100	Imaginary
61	50	50	0	Real roots
62	50	50	1	Real roots
63	50	50	50	Imaginary
64	50	50	99	Imaginary
65	50	50	100	Imaginary
66	50	99	0	Real root

(Contd.)...

Test case	a	b	c	Expected output
67	50	99	1	Real root
68	50	99	50	Imaginary
69	50	99	99	Imaginary
70	50	99	100	Imaginary
71	50	100	0	Real roots
72	50	100	1	Real roots
73	50	100	50	Equal roots
74	50	100	99	Imaginary
75	50	100	100	Imaginary
76	99	0	0	Equal roots
77	99	0	1	Imaginary
78	99	0	50	Imaginary
79	99	0	99	Imaginary
80	99	0	100	Imaginary
81	99	1	0	Real roots
82	99	1	1	Imaginary
83	99	1	50	Imaginary
84	99	1	99	Imaginary
85	99	1	100	Imaginary
86	99	50	0	Real Roots
87	99	50	1	Real roots
88	99	50	50	Imaginary
89	99	50	99	Imaginary
90	99	50	100	Imaginary
91	99	99	0	Real roots
92	99	99	1	Real roots
93	99	99	50	Imaginary roots
94	99	99	99	Imaginary
95	99	99	100	Imaginary
96	99	100	0	Real roots

Test case	a	b	c	Expected output
97	99	100	1	Real roots
98	99	100	50	Imaginary
99	99	100	99	Imaginary
100	99	100	100	Imaginary
101	100	0	0	Equal roots
102	100	0	1	Imaginary
103	100	0	50	Imaginary
104	100	0	99	Imaginary
105	100	0	100	Imaginary
106	100	1	0	Real roots
107	100	1	1	Imaginary
108	100	1	50	Imaginary
109	100	1	99	Imaginary
110	100	1	100	Imaginary
111	100	50	0	Real roots
112	100	50	1	Real roots
113	100	50	50	Imaginary
114	100	50	99	Imaginary
115	100	50	100	Imaginary
116	100	99	0	Real roots
117	100	99	1	Real roots
118	100	99	50	Imaginary
119	100	99	99	Imaginary
120	100	99	100	Imaginary
121	100	100	0	Real roots
122	100	100	1	Real roots
123	100	100	50	Imaginary
124	100	100	99	Imaginary
125	100	100	100	Imaginary

Example 8.5

Consider the program for the determination of previous date in a calendar as explained in Example 8.2. Design the robust and worst test cases for this program.