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 5 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.

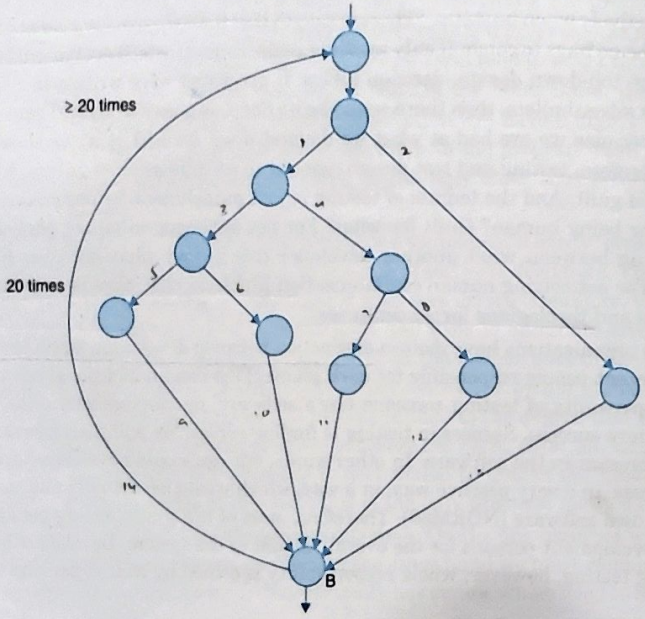

Fig. 8.1: Control flow graph [MYER79]

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 (Black box 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.

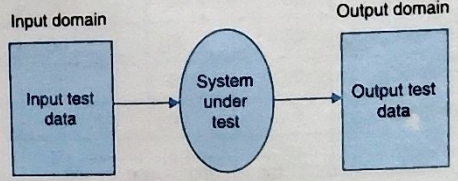

Fig. 8.3: Black box testing

1. Boundary Value
2. Equivalence Class
3. Decision Table
4. Cause Effect Graph
5. Special Value 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 chances 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.

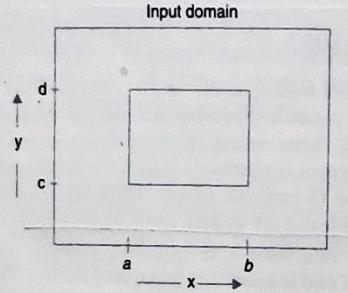

Fig. 8.4: Input domain for program having two input variables

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.

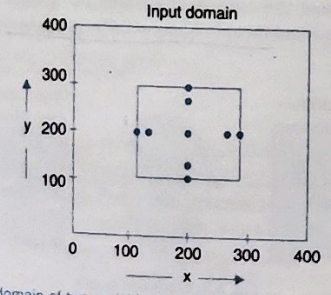

Fig. 8.5: Input domain of two variables x and y with boundaries [100, 300]

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

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

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 and 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.

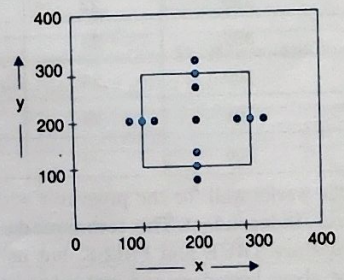

Fig. 8.6: Robustness test cases for two variables x and y with range [100, 300] each

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 values. 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.)...

0, 1, 50, 99, 100

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

(Contd.)...

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.

Solution

Robust test cases are $6n + 1$. Hence total 19 robust test cases are designed and are given below:

Test case	Month	Day	Year	Expected output
1	6	15	1899	Invalid date (outside range)
2	6	15	1900	14 June, 1900
3	6	15	1901	14 June, 1901
4	6	15	1962	14 June, 1962
5	6	15	2024	14 June, 2024
6	6	15	2025	14 June, 2025
7	6	15	2026	Invalid date (outside range)
8	6	0	1962	Invalid date
9	6	1	1962	31 May, 1962
10	6	2	1962	1 June, 1962
11	6	30	1962	29 June, 1962
12	6	31	1962	Invalid date
13	6	32	1962	Invalid date
14	0	15	1962	Invalid date
15	1	15	1962	14 January, 1962
16	2	15	1962	14 February, 1962
17	11	15	1962	14 November, 1962
18	12	15	1962	14 December, 1962
19	13	15	1962	Invalid date

Worst test cases are 5^n and n is 3. Hence 125 test cases are generated and are given below:

Test case	Month	Day	Year	Expected output
1	1	1	1900	31 December, 1899
2	1	1	1901	31 December, 1900
3	1	1	1962	31 December, 1961
4	1	1	2024	31 December, 2023
5	1	1	2025	31 December, 2024
6	1	2	1900	1 January, 1900
7	1	2	1901	1 January, 1901

(Contd.)...

Test case	Month	Day	Year	Expected output
8	1	2	1962	1 January, 1962
9	1	2	2024	1 January, 2024
10	1	2	2025	1 January, 2025
11	1	15	1900	14 January, 1900
12	1	15	1901	14 January, 1901
13	1	15	1962	14 January, 1962
14	1	15	2024	14 January, 2024
15	1	15	2025	14 January, 2025
16	1	30	1900	29 January, 1900
17	1	30	1901	29 January, 1901
18	1	30	1962	29 January, 1962
19	1	30	2024	29 January, 2024
20	1	30	2025	29 January, 2025
21	1	31	1900	30 January, 1900
22	1	31	1901	30 January, 1901
23	1	31	1962	30 January, 1962
24	1	31	2024	30 January, 2024
25	1	31	2025	30 January, 2025
26	2	1	1900	31 January, 1900
27	2	1	1901	31 January, 1901
28	2	1	1962	31 January, 1962
29	2	1	2024	31 January, 2024
30	2	1	2025	31 January, 2025
31	2	2	1900	1 February, 1900
32	2	2	1901	1 February, 1901
33	2	2	1962	1 February, 1962
34	2	2	2024	1 February, 2024
35	2	2	2025	1 February, 2025
36	2	15	1900	14 February, 1900
37	2	15	1901	14 February, 1901
38	2	15	1962	14 February, 1962
39	2	15	2024	14 February, 2024
40	2	15	2025	14 February, 2025
41	2	30	1900	Invalid date
42	2	30	1901	Invalid date

(Contd.)...

Test case	Month	Day	Year	Expected output
43	2	30	1962	Invalid date
44	2	30	2024	Invalid date
45	2	30	2025	Invalid date
46	2	31	1900	Invalid date
47	2	31	1901	Invalid date
48	2	31	1962	Invalid date
49	2	31	2024	Invalid date
50	2	31	2025	Invalid date
51	6	1	1900	31 May, 1900
52	6	1	1901	31 May, 1901
53	6	1	1962	31 May, 1962
54	6	1	2024	31 May, 2024
55	6	1	2025	31 May, 2025
56	6	2	1900	1 June, 1900
57	6	2	1901	1 June, 1901
58	6	2	1962	1 June, 1962
59	6	2	2024	1 June, 2024
60	6	2	2025	1 June, 2025
61	6	15	1900	14 June, 1900
62	6	15	1901	14 June, 1901
63	6	15	1962	14 June, 1962
64	6	15	2024	14 June, 2024
65	6	15	2025	14 June, 2025
66	6	30	1900	29 June, 1900
67	6	30	1901	29 June, 1901
68	6	30	1962	29 June, 1962
69	6	30	2024	29 June, 2024
70	6	30	2025	29 June, 2025
71	6	31	1900	Invalid date
72	6	31	1901	Invalid date
73	6	31	1962	Invalid date
74	6	31	2024	Invalid date
75	6	31	2025	Invalid date
76	11	1	1900	31 October, 1900
77	11	1	1901	31 October, 1901
78	11	1	1962	31 October, 1962

(Contd.)...

Test case	Month	Day	Year	Expected output
79	11	1	2024	31 October, 2024
80	11	1	2025	31 October, 2025
81	11	2	1900	1 November, 1900
82	11	2	1901	1 November, 1901
83	11	2	1962	1 November, 1962
84	11	2	2024	1 November, 2024
85	11	2	2025	1 November, 2025
86	11	15	1900	14 November, 1900
87	11	15	1901	14 November, 1901
88	11	15	1962	14 November, 1962
89	11	15	2024	14 November, 2024
90	11	15	2025	14 November, 2025
91	11	30	1900	29 November, 1900
92	11	30	1901	29 November, 1901
93	11	30	1962	29 November, 1962
94	11	30	2024	29 November, 2024
95	11	30	2025	29 November, 2025
96	11	31	1900	Invalid date
97	11	31	1901	Invalid date
98	11	31	1962	Invalid date
99	11	31	2024	Invalid date
100	11	31	2025	Invalid date
101	12	1	1900	30 November, 1900
102	12	1	1901	30 November, 1901
103	12	1	1962	30 November, 1962
104	12	1	2024	30 November, 2024
105	12	1	2025	30 November, 2025
106	12	2	1900	1 December, 1900
107	12	2	1901	1 December, 1901
108	12	2	1962	1 December, 1962
109	12	2	2024	1 December, 2024
110	12	2	2025	1 December, 2025
111	12	15	1900	14 December, 1900
112	12	15	1901	14 December, 1901
113	12	15	1962	14 December, 1962
114	12	15	2024	14 December, 2024

(Contd.)...

Test case	Month	Day	Year	Expected output
115	12	15	2025	14 December, 2025
116	12	30	1900	29 December, 1900
117	12	30	1901	29 December, 1901
118	12	30	1962	29 December, 1962
119	12	30	2024	29 December, 2024
120	12	30	2025	29 December, 2025
121	12	31	1900	30 December, 1900
122	12	31	1901	30 December, 1901
123	12	31	1962	30 December, 1962
124	12	31	2024	30 December, 2024
125	12	31	2025	30 December, 2025

Example 8.6

Consider the triangle problem as given in Example 8.3. Generate robust and worst test cases for this problem.

Solution

Robust test cases are:

Test case	x	y	z	Expected output
1	50	50	0	Invalid input
2	50	50	1	Isosceles
3	50	50	2	Isosceles
4	50	50	50	Equilateral
5	50	50	99	Isosceles
6	50	50	100	Not a triangle
7	50	50	101	Invalid input
8	50	0	50	Invalid input
9	50	1	50	Isosceles
10	50	2	50	Isosceles
11	50	99	50	Isosceles
12	50	100	50	Not a triangle
13	50	101	50	Invalid input
14	0	50	50	Invalid input
15	1	50	50	Isosceles
16	2	50	50	Isosceles
17	99	50	50	Isosceles
18	100	50	50	Not a triangle
19	101	50	50	Invalid input

Worst test cases are 125 and are given below:

Test case	x	y	z	Expected output
1	1	1	1	Equilateral
2	1	1	2	Not a triangle
3	1	1	50	Not a triangle
4	1	1	99	Not a triangle
5	1	1	100	Not a triangle
6	1	2	1	Not a triangle
7	1	2	2	Isosceles
8	1	2	50	Not a triangle
9	1	2	99	Not a triangle
10	1	2	100	Not a triangle
11	1	50	1	Not a triangle
12	1	50	2	Not a triangle
13	1	50	50	Isosceles
14	1	50	99	Not a triangle
15	1	50	100	Not a triangle
16	1	99	1	Not a triangle
17	1	99	2	Not a triangle
18	1	99	50	Not a triangle
19	1	99	99	Isosceles
20	1	99	100	Not a triangle
21	1	100	1	Not a triangle
22	1	100	2	Not a triangle
23	1	100	50	Not a triangle
24	1	100	99	Not a triangle
25	1	100	100	Isosceles
26	2	1	1	Not a triangle
27	2	1	2	Isosceles
28	2	1	50	Not a triangle
29	2	1	99	Not a triangle
30	2	1	100	Not a triangle
31	2	2	1	Isosceles
32	2	2	2	Equilateral
33	2	2	50	Not a triangle
34	2	2	99	Not a triangle

(Contd.)...

Test case	x	y	z	Expected output
35	2	2	100	Not a triangle
36	2	50	1	Not a triangle
37	2	50	2	Not a triangle
38	2	50	50	Isosceles
39	2	50	99	Not a triangle
40	2	50	100	Not a triangle
41	2	99	1	Not a triangle
42	2	99	2	Not a triangle
43	2	99	50	Not a triangle
44	2	99	99	Isosceles
45	2	99	100	Scalene
46	2	100	1	Not a triangle
47	2	100	2	Not a triangle
48	2	100	50	Not a triangle
49	2	100	99	Scalene
50	2	100	100	Isosceles
51	50	1	1	Not a triangle
52	50	1	2	Not a triangle
53	50	1	50	Isosceles
54	50	1	99	Not a triangle
55	50	1	100	Not a triangle
56	50	2	1	Not a triangle
57	50	2	2	Not a triangle
58	50	2	50	Isosceles
59	50	2	99	Not a triangle
60	50	2	100	Not a triangle
61	50	50	1	Isosceles
62	50	50	2	Isosceles
63	50	50	50	Equilateral
64	50	50	99	Isosceles
65	50	50	100	Not a triangle
66	50	99	1	Not a triangle
67	50	99	2	Not a triangle
68	50	99	50	Isoscless
69	50	99	99	Isoscless

(Contd.)...

Test case	x	y	z	Expected output
70	50	99	100	Scalene
71	50	100	1	Not a triangle
72	50	100	2	Not a triangle
73	50	100	50	Not a triangle
74	50	100	99	Scalene
75	50	100	100	Isosceles
76	99	1	1	Not a triangle
77	99	1	2	Not a triangle
78	99	1	50	Not a triangle
79	99	1	99	Isosceles
80	99	1	100	Not a triangle
81	99	2	1	Not a triangle
82	99	2	2	Not a triangle
83	99	2	50	Not a triangle
84	99	2	99	Isosceles
85	99	2	100	Scalene
86	99	50	1	Not a triangle
87	99	50	2	Not a triangle
88	99	50	50	Isosceles
89	99	50	99	Isosceles
90	99	50	100	Scalene
91	99	99	1	Isosceles
92	99	99	2	Isosceles
93	99	99	50	Isosceles
94	99	99	99	Equilateral
95	99	99	100	Isosceles
96	99	100	1	Not a triangle
97	99	100	2	Scalene
98	99	100	50	Scalene
99	99	100	99	Isosceles
100	99	100	100	Isosceles
101	100	1	1	Not a triangle
102	100	1	2	Not a triangle
103	100	1	50	Not a triangle
104	100	1	99	Not a triangle

(Contd.)...

Test case	x	y	z	Expected output
105	100	1	100	Isosceles
106	100	2	1	Not a triangle
107	100	2	2	Not a triangle
108	100	2	50	Not a triangle
109	100	2	99	Scalene
110	100	2	100	Isosceles
111	100	50	1	Not a triangle
112	100	50	2	Not a triangle
113	100	50	50	Not a triangle
114	100	50	99	Scalene
115	100	50	100	Isosceles
116	100	99	1	Not a triangle
117	100	99	2	Scalene
118	100	99	50	Scalene
119	100	99	99	Isosceles
120	100	99	100	Isosceles
121	100	100	1	Isosceles
122	100	100	2	Isosceles
123	100	100	50	Isosceles
124	100	100	99	Isosceles
125	100	100	100	Equilateral

8.3.2 Equivalence Class Testing

In this method, input domain of a program is partitioned into a finite number of equivalence classes such that one can reasonably assume, but not be absolutely sure, that the test of a representative value of each class is equivalent to a test of any other value. That is, if one test case in a class detects an error, all other test cases in the class would be expected to find same error. Conversely, if a test case did not detect an error, we would expect that no other test cases in the class would find an error. Two steps are required in implementing this method:

1. The equivalence classes are identified by taking each input condition and partitioning it into valid and invalid classes. For example, if an input condition specifies a range of values from 1–999, we identify one valid equivalence class [1 < item < 999]; and two invalid equivalence classes [item < 1] and [item > 999].
2. Generate the test cases using the equivalence classes identified in the previous step. This is performed by writing test cases covering all the valid equivalence classes. Then a test case is written for each invalid equivalence class so that no test contains more than one invalid class. This is to ensure that no two invalid classes mask each other.

In Fig. 8.7, both valid and invalid input domains are shown.

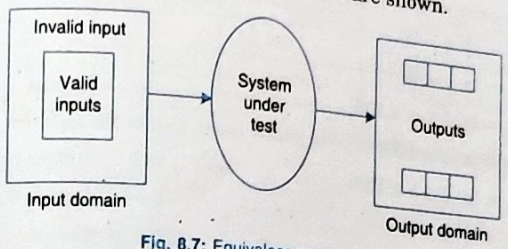

Fig. 8.7: Equivalece partitioning

The idea is to choose at least one element from each equivalent class. In the triangle problem we would certainly have a test case for equivalent triangle, and we may take (50, 50, 50) as inputs for a test case. If we do this, we would not expect to learn much from test cases such as (40, 40, 40) and (100, 100, 100). These test cases may be treated as redundant test cases.

We should not forget to have equivalent classes for invalid inputs. This is often best source of bugs. We should test different types of invalid inputs in order to get more errors. As an example, for a program that is supposed to accept any number between 1 and 99, there are at least four equivalence classes from input side. The classes are:

(*i*) Any number between 1 and 99 is valid input.

(*ii*) Any number less than 1. This include 0 and all negative numbers.

(*iii*) Any number greater than 99.

(*iv*) If it is not a number, it should not be accepted.

Most of the time, equivalence class testing defines classes of the input domain. However, equivalence classes should also be defined for output domain. Hence, we should design equivalence classes based on input and output domains.

Example 8.7

Consider the program for the determination of nature of roots of a quadratic equation as explained in Example 8.1. Identify the equivalence class test cases for output and input domains.

Solution

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

$O_1 = \{< a, b, c >$: Not a quadratic equation if $a = 0\}$

$O_2 = \{< a, b, c >$: Real roots if $(b^2 - 4ac) > 0\}$

$O_3 = \{< a, b, c >$: Imaginary roots if $(b^2 - 4ac) < 0\}$

$O_4 = \{< a, b, c >$: Equal roots if $(b^2 - 4ac) = 0\}$

The number of test cases can be derived from above relations and shown below:

Test case	a	b	c	Expected output
1	0	50	50	Not a quadratic equation
2	1	50	50	Real roots
3	50	50	50	Imaginary roots
4	50	100	50	Equal roots

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

$I_1 = \{a: a = 0\}$

$I_2 = \{a: a < 0\}$

$I_3 = \{a: 1 \le a \le 100\}$

$I_4 = \{a: a > 100\}$

$I_5 = \{b: 0 \le b \le 100\}$

$I_6 = \{b: b < 0\}$

$I_7 = \{b: b > 100\}$

$I_8 = \{c: 0 \le c \le 100\}$

$I_9 = \{c: c < 0\}$

$I_{10} = \{c: c > 100\}$

In these classes, our basic assumption is single fault theory. Hence one value is at an extreme and other values are nominal values. Input domain test cases are:

Test case	*a*	*b*	*c*	*Expected output*
1	0	50	50	Not a quadratic equation
2	– 1	50	50	Invalid input
3	50	50	50	Imaginary roots
4	101	50	50	Invalid input
5	50	50	50	Imaginary roots
6	50	– 1	50	Invalid input
7	50	101	50	Invalid input
8	50	50	50	Imaginary roots
9	50	50	– 1	Invalid input
10	50	50	101	Invalid input

Here test cases 5 and 8 are redundant test cases. If we choose any value other than nominal, we may not have redundant test cases. Hence total test cases are 10 + 4 = 14 for this problem.

Example 8.8

Consider the program for determining the Previous date in a calendar as explained in Example 8.3. Identify the equivalence class test cases for output and input domains.

Solution

Output domain equivalence classes are:

$O_1 = \{< D, M, Y >: \text{Previous date if all are valid inputs}\}$

$O_2 = \{< D, M, Y >: \text{Invalid date if any input makes the date invalid}\}$

Test case	*M*	*D*	*Y*	*Expected output*
1	6	15	1962	14 June, 1962
2	6	31	1962	Invalid date

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

$I_1 = \{\text{month}: 1 \le m \le 12\}$

$I_2 = \{\text{month}: m < 1\}$

$I_3 = \{\text{month}: m > 12\}$

$I_4 = \{\text{day}: 1 \le D \le 31\}$

$I_5 = \{\text{day}: D < 1\}$

$I_6 = \{\text{day}: D > 31\}$

$I_7 = \{\text{year}: 1900 \le Y \le 2025\}$

$I_8 = \{\text{year}: Y < 1900\}$

$I_9 = \{\text{year}: Y > 2025\}$

Input domain test cases are :

Test case	*M*	*D*	*Y*	*Expected output*
1	6	15	1962	14 June, 1962
2	– 1	15	1962	Invalid input
3	13	15	1962	Invalid input
4	6	15	1962	14 June, 1962
5	6	– 1	1962	Invalid input
6	6	32	1962	Invalid input
7	6	15	1962	14 June, 1962
8	6	15	1899	Invalid input (value out of range)
9	6	15	2026	Invalid input (value out of range)

Example 8.9

Consider the triangle problem specified in example 8.3. Identify the equivalence class test cases for output and input domain.

Solution

Output domain equivalence classes are:

$O_1 = \{< x, y, z >: \text{Equilateral triangle with sides } x, y, z\}$

$O_2 = \{< x, y, z >: \text{Isosceles trinagle with sides } x, y, z\}$

$O_3 = \{< x, y, z >: \text{Scalene triangle with sides } x, y, z\}$

$O_4 = \{< x, y, z >: \text{Not a triangle with sides } x, y, z\}$

The test cases are:

Test case	x	y	z	Expected output
1	50	50	50	Equilateral
2	50	50	99	Isosceles
3	100	99	50	Scalene
4	50	100	50	Not a triangle

Input domain based classes are:

$I_1 = \{x: x < 1\}$

$I_2 = \{x: x > 100\}$

$I_3 = \{x: 1 \le x \le 100\}$

$I_4 = \{y: y < 1\}$

$I_5 = \{y: y > 100\}$

$I_6 = \{y: 1 \le y \le 100\}$

$I_7 = \{z: z < 1\}$

$I_8 = \{z: z > 100\}$

$I_9 = \{z: 1 \le z \le 100\}$

Some input domain test cases can be obtained using the relationship amongst x, y and z.

$I_{10} = \{< x, y, z >: x = y = z\}$

$I_{11} = \{< x, y, z >: x = y, x \neq z\}$

$I_{12} = \{< x, y, z >: x = z, x \neq y\}$

$I_{13} = \{< x, y, z >; y = z, x \neq y\}$

$I_{14} = \{< x, y, z >: x \neq y, x \neq z, y \neq z\}$

$I_{15} = \{< x, y, z >: x = y + z\}$

$I_{16} = \{< x, y, z >: x > y + z\}$

$I_{17} = \{< x, y, z >: y = x + z\}$

$I_{18} = \{< x, y, z >: y > x + z\}$

$I_{19} = \{< x, y, z >: z = x + y\}$

$I_{20} = \{< x, y, z >; z > x + y\}$

Test cases derived from input domain are:

Test case	x	y	z	Expected output
1	0	50	50	Invalid input
2	101	50	50	Invalid input
3	50	50	50	Equilateral
4	50	0	50	Invalid input
5	50	101	50	Invalid input
6	50	50	50	Equilateral
7	50	50	0	Invalid input
8	50	50	101	Invalid input
9	50	50	50	Equilateral
10	60	60	60	Equilateral
11	50	50	60	Isosceles
12	50	60	50	Isosceles
13	60	50	50	Isosceles
14	100	99	50	Scalene
15	100	50	50	Not a triangle
16	100	50	25	Not a triangle
17	50	100	50	Not a triangle
18	50	100	25	Not a triangle
19	50	50	100	Not a triangle
20	25	50	100	Not a triangle

8.3.3 Decision Table Based Testing

Decisions tables are useful for describing situations in which a number of combinations of actions are taken under varying sets of conditions. Decision tables have been used to represent and analyse complex logical relationships since early 1960s. We would like to show that how these may be applied to the testing and how tester may adopt the principles to his/her own situation. Some of the basic terms are shown in table 8.2.

Table 8.2: Decision table terminology.

		Entry						
Condition Stub	c_1	True				False		
	c_2	True		False		True		False
	c_3	True	False	True	False	True	False	—
Action Stub	a_1	X	X			X		
	a_2	X		X			X	
	a_3		X			X		
	a_4				X		X	X

There are four portions of a decision table namely, Conditions stub, Action Stub, Condition entries and Action entries. When conditions c_1, c_2 and c_3 are all true, actions a_1 and a_2 occur. When conditions c_1 and c_2 are true and c_3 is false, actions a_1 and a_3 occur. The decision tables in which all entries are binary are called limited entry decision tables. If conditions are allowed to have several values, the resulting tables are called Extended Entry Decision tables.

Test case design

To identify test cases with decision tables, we interpret conditions as inputs, and actions as outputs. Sometimes, conditions end up referring to equivalence classes of inputs, and actions refers to major functional processing portions of the item being tested. The rules are then interpreted as test cases. Because the decision table can mechanically be forced to be completed, we know, we have a comprehensive set of test cases. There are several techniques that produce decision tables that are more useful to testers. One helpful style is to add an action to show when a rule is logically impossible [JORG95].

Consider the decision table shown in table 8.3, we see examples of don't care entries and impossible rule usage.

Table 8.3: Decision table for triangle problem.

c_1: x, y, z are sides of a triangle?	N	Y							
c_2: x = y ?	—	Y				N			
c_3: x = z ?	—	Y		N		Y		N	
c_4: y = z ?	—	Y	N	Y	N	Y	N	Y	N
a_1: Not a triangle	X								
a_2: Scalene									X
a_3: Isosceles					X		X	X	
a_4: Equilateral		X							
a_5: Impossible			X	X		X			

If the integers x, y and z do not constitute a triangle, we do not even care about possible equalities, we may also choose conditions, but this will increase the size of the decision table as shown in table 8.4. Here, old condition (c_1: x, y, z are sides of a triangle?) has been expanded to get a more detailed view of the three inequalities of the triangle property. If any one of these fails, the three integers do not constitute sides of a triangle.

Table 8.4: Modified decision table.

Conditions											
c_1: x < y + z ?	F	T	T	T	T	T	T	T	T	T	T
c_2: y < x + z ?	—	F	T	T	T	T	T	T	T	T	T
c_3: z < x + y ?	—	—	F	T	T	T	T	T	T	T	T
c_4: x = y ?	—	—	—	T	T	T	T	F	F	F	F
c_5: x = z ?	—	—	—	T	T	F	F	T	T	F	F
c_6: y = z ?	—	—	—	T	F	T	F	T	F	T	F

(Contd.)...

a_1: Not a triangle	X	X	X								
a_2: Scalene											X
a_3: Isosceles							X		X	X	
a_4: Equilateral				X							
a_5: Impossible					X	X		X			

This is another way of representing the same thing. Table 8.3 seems to be more readable.

Example 8.10

Consider the triangle problem specified in example 8.3. Identify the test cases using the decision table of Table 8.4.

Solution

There are eleven functional test cases; three to fail triangle property, three impossible cases; one each to get equilateral, scalene triangle cases, and three to get on isosceles triangle. The test cases are given in Table 8.5.

Table 8.5: Test cases of triangle problem using decision table.

Test case	*x*	*y*	*z*	*Expected Output*
1	4	1	2	Not a triangle
2	1	4	2	Not a triangle
3	1	2	4	Not a triangle
4	5	5	5	Equilateral
5	?	?	?	Impossible
6	?	?	?	Impossible
7	2	2	3	Isosceles
8	?	?	?	Impossible
9	2	3	2	Isosceles
10	3	2	2	Isosceles
11	3	4	5	Scalene

Example 8.11

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

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

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

$$1900 \le \text{year} \le 2025$$

The possible outputs are "Previous date" and "Invalid date". Design the test cases using decision table based testing.

Solution

The input domain can be divided into following classes:

$I_1 = (M_1$: month has 30 days$)$

$I_2 = (M_2$: month has 31 days except March, August and January$)$

$I_3 = (M_3$: month is March$)$

$I_4 = (M_4$: month is August$)$

$I_5 = (M_5$: month is January$)$

$I_6 = (M_6$: month is February$)$

$I_7 = (D_1$: day = 1$)$

$I_8 = (D_2$: $2 \le$ day $\le 28)$

$I_9 = (D_3$: day = 29$)$

$I_{10} = (D_4$: day = 30$)$

$I_{11} = (D_5$: day = 31$)$

$I_{12} = (Y_1$: year is a leap year$)$

$I_{13} = (Y_2$: year is a common year$)$

The decision table is given below:

Sr. No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
C_1: Months in	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_2	M_2	M_2	M_2	M_2	M_2	M_2	M_2	M_2	M_2
C_2: days in	D_1	D_1	D_2	D_2	D_3	D_3	D_4	D_4	D_5	D_5	D_1	D_1	D_2	D_2	D_3	D_3	D_4	D_4	D_5	D_5
C_3: year in	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2
a_1: Impossible									X	X			X	X	X	X	X	X	X	X
a_2: Decrement day			X	X	X	X	X	X												
a_3: Reset day to 31	X	X									X	X								
a_4: Reset day to 30																				
a_5: Reset day to 29																				
a_6: Reset day to 28																				
a_7: decrement month	X	X									X	X								
a_8: Reset month to December																				
a_9: Decrement year																				

(Contd.)...

Sr. No.	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
C_1: Months in	M_3	M_3	M_3	M_3	M_3	M_3	M_3	M_3	M_3	M_3	M_4	M_4	M_4	M_4	M_4	M_4	M_4	M_4	M_4	M_4
C_2: days in	D_1	D_1	D_2	D_2	D_3	D_3	D_4	D_4	D_5	D_5	D_1	D_1	D_2	D_2	D_3	D_3	D_4	D_4	D_5	D_5
C_3: years in	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2
a_1: Impossible																				
a_2: Decrement day			X	X	X	X	X	X	X	X			X	X	X	X	X	X	X	X
a_3: Reset day to 31											X	X								
a_4: Reset day to 30																				
a_5: Reset day to 29	X																			
a_6: Reset day to 28		X																		
a_7: decrement month	X	X									X	X								
a_8: Reset month to December																				
a_9: Decrement year																				

Sr. No.	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
C_1: Months in	M_5	M_5	M_5	M_5	M_5	M_5	M_5	M_5	M_5	M_5	M_6	M_6	M_6	M_6	M_6	M_6	M_6	M_6	M_6	M_6
C_2: days in	D_1	D_1	D_2	D_2	D_3	D_3	D_4	D_4	D_5	D_5	D_1	D_1	D_2	D_2	D_3	D_3	D_4	D_4	D_5	D_5
C_3: years in	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2
a_1: Impossible																X	X	X	X	X
a_2: Decrement day			X	X	X	X	X	X	X	X			X	X	X					
a_3: Reset day to 31	X	X									X	X								
a_4: Reset day to 30																				
a_5: Reset day to 29																				
a_6: Reset day to 28																				
a_7: Decrement month											X	X								
a_8: Reset month to December	X	X																		
a_9: Decrement year	X	X																		

The number of test cases are equal to number of columns of the decision table. Hence 60 test cases can be generated. Here input domain is partitioned into thirteen classes. Class identification is somewhat tricky and depends on understanding of the problem. We have separate classes for March, August, January and February. Justification is given below:

(*i*) **March:** We have to find previous date. If present date is first March, the previous date may be 28 or 29 depending upon the present year. Hence separate treatment is required for March.

(*ii*) **August:** August is a month of 31 days and its previous month July is also of 31 days. This is a typical situation. Normally, 31 days months have previous month of 30 days. In order to accommodate this, August has a separate status.

(*iii*) **January:** First January present date will have 31st December as Previous date. It is similar to August month situation except decrementing the year.

(*iv*) **February:** Due to less number of days and leap year situation, February has special features and is required as a separate class.

Similary, separate classes to handle 29, 30 and 31 days have been created. First day of every month has a separate class in order to get previous month's last day. The 60 test cases are given below:

Test case	*Month*	*Day*	*Year*	*Expected output*
1	June	1	1964	31 May, 1964
2	June	1	1962	31 May, 1962
3	June	15	1964	14 June, 1964
4	June	15	1962	14 June, 1962
5	June	29	1964	28 June, 1964
6	June	29	1962	28 June, 1962
7	June	30	1964	29 June, 1964
8	June	30	1962	29 June, 1962
9	June	31	1964	Impossible
10	June	31	1962	Impossible
11	May	1	1964	30 April, 1964
12	May	1	1962	30 April, 1962
13	May	15	1964	14 May, 1964
14	May	15	1962	14 May, 1962
15	May	29	1964	28 May, 1964
16	May	29	1962	28 May, 1962
17	May	30	1964	29 May, 1964
18	May	30	1962	29 May, 1962
19	May	31	1964	30 May, 1964
20	May	31	1962	30 May, 1962

(Contd.)...

Test case	*Month*	*Day*	*Year*	*Expected output*
21	March	1	1964	29 February, 1964
22	March	1	1962	28 February, 1962
23	March	15	1964	14 March, 1964
24	March	15	1962	14 March, 1962
25	March	29	1964	28 March, 1964
26	March	29	1962	28 March, 1962
27	March	30	1964	29 March, 1964
28	March	30	1962	29 March, 1962
29	March	31	1964	30 March, 1964
30	March	31	1962	30 March, 1962
31	August	1	1964	31 July, 1964
32	August	1	1962	31 July, 1962
33	August	15	1964	14 August, 1964
34	August	15	1962	14 August, 1962
35	August	29	1964	28 August, 1964
36	August	29	1962	28 August, 1962
37	August	30	1964	29 August, 1964
38	August	30	1962	29 August, 1962
39	August	31	1964	30 August, 1964
40	August	31	1962	30 August, 1962
41	January	1	1964	31 December, 1963
42	January	1	1962	31 December, 1961
43	January	15	1964	14 January, 1964
44	January	15	1962	14 January, 1962
45	January	29	1964	28 January, 1964
46	January	29	1962	28 January, 1962
47	January	30	1964	29 January, 1964
48	January	30	1962	29 January, 1962
49	January	31	1964	30 January, 1964
50	January	31	1962	30 January, 1962
51	February	1	1964	31 January, 1964
52	February	1	1962	31 January, 1962
53	February	15	1964	14 February, 1964
54	February	15	1962	14 February, 1962
55	February	29	1964	28 February, 1964

(Contd.)...

Test case	Month	Day	Year	Expected output
56	February	29	1962	Impossible
57	February	30	1964	Impossible
58	February	30	1962	Impossible
59	February	31	1964	Impossible
60	February	31	1962	Impossible

8.3.4 Cause Effect Graphing Technique

One weakness of boundary value analysis and equivalence partitioning is that these do not explore combinations of input circumstances. These consider only single input conditions. However, combinations of inputs may result in interesting situations. These situations should be tested. If we consider all valid combinations of equivalence classes, then we will have large number of test cases, many of which will not be useful for revealing any new errors. For example, if there are "n" different input conditions, such that any combination of the input conditions is valid, we will have 2^n test cases [JALO96].

Cause effect graphing [ELME73] is a technique that aids in selecting, in a systematic way, a high-yield set of test cases. It has a beneficial effect in pointing out incompleteness and ambiguities in the specifications. The following process is used to derive test cases [MYER79].

1. The causes and effects in the specifications are identified. A cause is a distinct input condition or an equivalence class of input conditions. An effect is an output condition or a system transformation (a lingering effect that an input has on the state of the program or system). For instance, if a transaction to a program causes a master file to be updated, the alteration to the master file is a system transformation; a confirmation message would be an output condition. Causes and effects are identified by reading the specification word by word and underlining words or phrases that describe causes and effects. Each cause and effect is assigned a unique number.
2. The semantic content of the specification is analyzed and transformed into a Boolean graph linking the causes and effects. This is the cause effect graph.
3. The graph is annotated with constraints describing combinations of causes and/or effects that are impossible because of syntactic or environmental constraints.
4. By methodically tracing state conditions in the graph, the graph is converted into a limited entry decision table. Each column in the table represents a test case.
5. The columns in the decision table are converted into test cases.

The basic notation for the graph is shown in Fig. 8.8.

Think of each node as having the value 0 or 1; 0 represents the 'absent state' and 1 represents the present state. The identity function states that if c_1 is 1, e_1 is 1; else e_1 is 0. The NOT function states that if c_1 is 1, e_1 is 0 else, e_1 is 1. The OR function states that if c_1 or c_2 or c_3 is 1, e_1 is 1; else e_1 is 0. The AND function states that if both c_1 and c_2 are 1, e_1 is 1; else e_1 is 0. The AND and OR functions are allowed to have any number of inputs.

Myers [MYER79] explained this effectively with the following example. "The characters in column 1 must be an A or B. The character in column 2 must be a digit. In this situation, the file update is made. If the character in column 1 is incorrect, message x is issued. If the character in column 2 is not a digit, message y is issued".

The causes are

c_1: character in column1 is A

c_2: character in column 1 is B

c_3: character in column 2 is a digit

and the effects are

e_1: update made

e_2: message x is issued

e_3: message y is issued

The cause-effect graph is shown in Fig. 8.9.

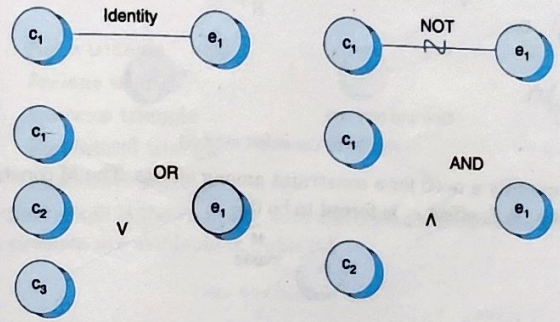

Fig. 8.8: Basic cause effect graph symbols

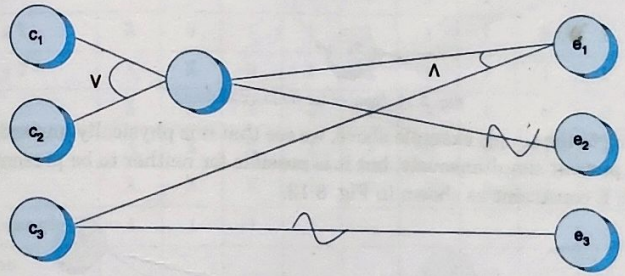

Fig. 8.9: Sample cause effect graph

Although the graph in Fig. 8.9 represents the specification, it does not contain an impossible combination of causes—it is impossible for both causes c_1 or c_2 to be set to 1 simultaneously. In most programs, certain combinations of causes are impossible because of syntactic or environmental considerations. To account for these, the notations in Fig. 8.10 is used. The **E** constraint states that it must always be true that at most one of c_1 or c_2 can be 1 (c_1 or c_2 cannot be 1 simultaneously). The **I** constraint states that at least one of c_1, c_2 and c_3 must always be (1. c_1, c_2 and c_3 cannot be 0 simultaneously). The O constraint states that one, and only one, of

c_1 and c_2 must be 1. The R constraint states that, for c_1 to be 1, c_2 must be 1 (*i.e.*, it is impossible for c_1 to be 1 and c_2 to be 0).

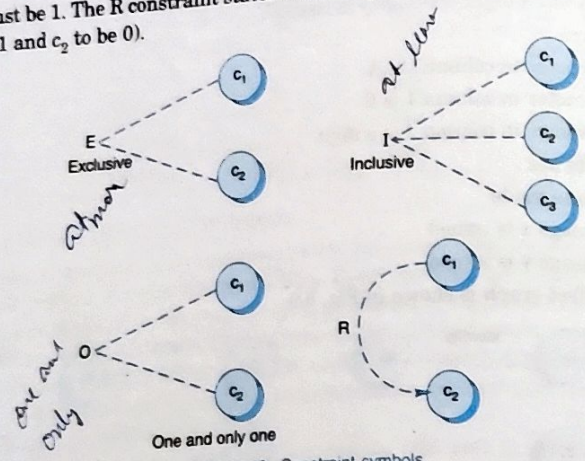

Fig. 8.10: Constraint symbols

There is frequently a need for a constraint among effects. The M constraint in Fig. 8.11 states that if effect e_1 is 1, effect e_2 is forced to be 0.

Fig. 8.11: Symbol for masks constraint

Returning to the simple example above, we see that it is physically impossible for causes c_1 and c_2 to be present simultaneously, but it is possible for neither to be present. Hence they are linked with E constraint as shown in Fig. 8.12.

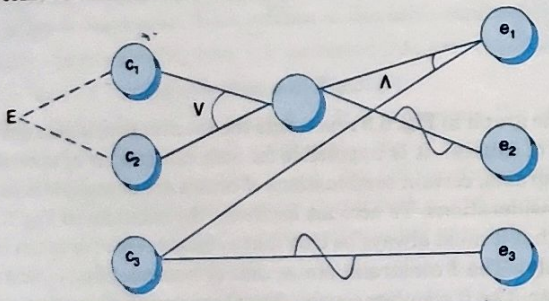

Fig. 8.12: Sample cause effect graph with exclusive constraint

Example 8.12

Consider the triangle problem specified in the example 8.3. Draw the cause-effect graph and identify the test cases.

Solution

The causes are

c_1 : side x is less than sum of sides y and z
c_2 : side y is less than sum of sides x and z
c_3 : side z is less than sum of sides x and y
c_4 : side x is equal to side y
c_5 : side x is equal to side z
c_6 : side y is equal to side z

and effects are

e_1 : Not a triangle
e_2 : Scalene triangle
e_3 : Isosceles triangle
e_4 : Equilateral triangle
e_5 : Impossible stage

The cause effect graph is shown in Fig. 8.13 and decision table is shown in table 8.6. The test cases for this problem are available in Table 8.5.

Table 8.6: Decision table

Conditions											
c_1: x < y + z ?	0	1	1	1	1	1	1	1	1	1	1
c_2: y < x + z ?	X	0	1	1	1	1	1	1	1	1	1
c_3: z < x + y ?	X	X	0	1	1	1	1	1	1	1	1
c_4: x = y ?	X	X	X	1	1	1	1	0	0	0	0
c_5: x = z ?	X	X	X	1	1	0	0	1	1	0	0
c_6: y = z ?	X	X	X	1	0	1	0	1	0	1	0
e_1: Not a triangle	1	1	1								
e_2: Scalene											1
e_3: Isosceles							1		1	1	
e_4: Equilateral				1							
e_5: Impossible					1	1		1			

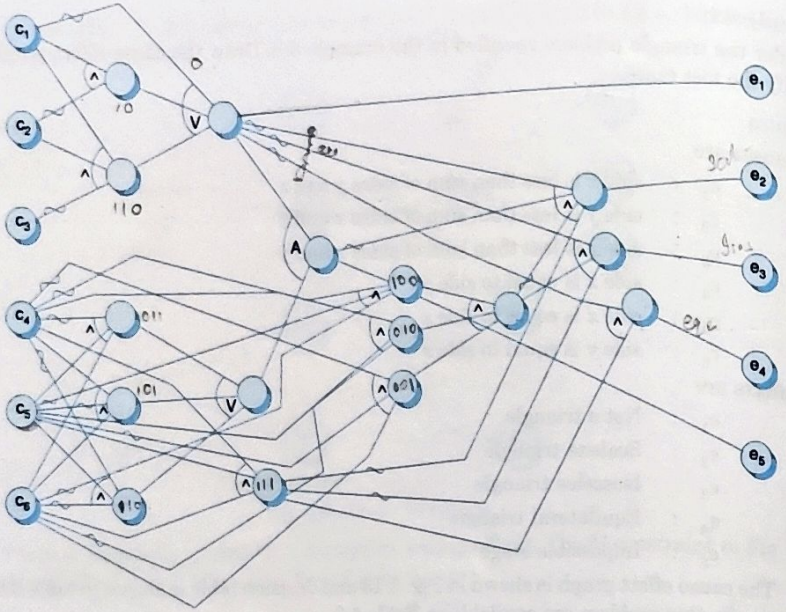

Fig. 8.13: Cause effect graph of triangle problem

Cause-effect graphing is a systematic method of generating test cases representing combinations of conditions. The alternative would be an adhoc selection of combinations, but in doing so it is likely that one would overlook many of the interesting test cases identified by the cause effect graph. Although, it does produce a set of useful test cases, it normally does not produce all of the useful test cases that might be identified.

8.3.5 Special Value Testing

It is probably the most widely practiced form of functional testing. It is the most intuitive and the least uniform type of testing. Special value testing occurs when a tester uses his or her domain knowledge, experience with similar programs and information about "Soft spots" to devise test cases. We may call this as adhoc testing.

Now guidelines are used other than to use "best engineering judgement". As a result, special value testing is heavily dependent on the abilities of the testing persons.

8.4 STRUCTURAL TESTING (White Box)

A complementary approach to functional testing is called structural/white box testing. It permits us to examine the internal structure of the program. In using this strategy, we derive test cases from an examination of the program's logic. We do not pay any attention to specifications. For instance, if the first statement of the code is "if ($x \leq 100$)", then we may try testing the program with a test case of 100.

Therefore, the knowledge to the internal structure of the code can be used to find the number of test cases required to guarantee a given level of test coverage. It would never be advisable to release a software which contained untested statements and the consequences of which might be disastrous. This goal seems to be easy, but simple objectives of structural testing are harder to achieve than may appear at first glance.

In functional testing, all specifications are being checked against the implementation, so why do we really require structural testing? It seems to be necessary because there might be parts of the code, which are not fully exercised by the functional tests. There may also be sections of the code, which are surplus to requirements. That is to say, we have checked the program against the functional tests and no errors are revealed, then further inspection by structural tests reveals a piece of code that is not even needed by the specifications and hence not examined by functional testing. This can be regarded as an error since it is a deviation from requirements. It may find those errors, which have been missed by functional testing [NORM89].

We want to look in to the program, examine the code and watch it as it runs. This activity is dynamic and is about testing a running program; therefore it is called dynamic white box testing. It we want to test the program without running it; meaning thereby examining and reviewing it; then it is called static white box testing.

Hence, static white box testing is the process of carefully and methodically reviewing the software design, architecture, or code for bugs without executing it. It is sometimes referred to as structural analysis [PATT01].

STATIC

8.4.1 Path Testing

Path testing is the name given to a group of test techniques based on judiciously selecting a set of test paths through the program. If the set of paths is properly chosen, then it means that we have achieved some measure of test thoroughness. For example, pick enough paths to assure that every source statement is executed at least once. It is most applicable to new software for (1) module testing or unit testing. It requires complete knowledge of the program's structure and (2) used by developers to unit test their own code. The effectiveness of path testing rapidly deteriorates as the size of the software under test increases. It is (3) rarely if ever, used for system (4) testing. For the developer, it is the basic test technique [BEIZ90].

This type of testing involves:

1. generating a set of paths that will cover every branch in the program.
2. finding a set of test cases that will execute every path in this set of program paths.

The two steps are not necessarily executed in sequence. Path generation (*i.e.*, step 1) can be performed through the static analysis of the program control flow and can be automated.

Flow graph

The control flow of a program can be analysed using a graphical representation known as flow graph. The flow graph is a directed graph in which nodes are either entire statements or fragments of a statement, and edges represents flow of control. If i and j are nodes in the program graph, there is an edge from node i to node j if the statement (fragment) corresponding to node j can be executed immediately after the statement (fragment) corresponding to node i.

A flow graph can easily be generated from the code of any problem. The basic constructs of flow graph are given in Fig. 8.14.

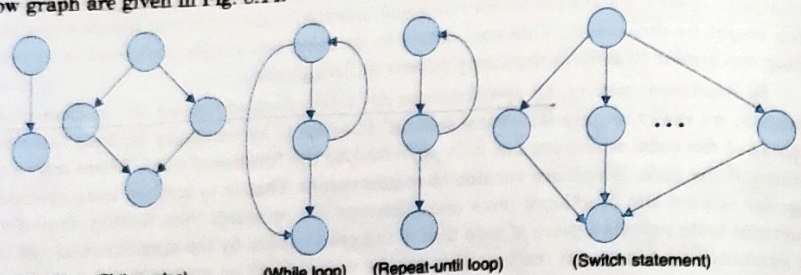

Fig. 8.14: The basic construct of the flow graph

We consider a program that generates the previous date, if a date is given as an input (for details refer Example 8.2) which is given in Fig. 8.15. Here line numbers are statements or fragments of statement. It is a matter of choice to make fragment as a separate node or to include fragments in other portion of a statement.

Our first step is to prepare a flow graph from the code. The flow graph of previous date program (given in Fig. 8.15) is generated and is given in Fig. 8.16. Such a flow graph helps us to understand the flow of control from source to destination. We want to find paths from this control flow and may like to execute every path during testing.

```
/* Program to generate the previous date given a date, assumes data
given as dd mm yyyy separated by space and performs error checks on the
validity of the current date entered. */

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

 1  int main()
 2  {
 3    int day, month, year, validDate = 0;
    /*Date Entry*/
 4    printf("Enter the day value: ");
 5    scanf("%d", &day);
 6    printf("Enter the month value: ");
 7    scanf("%d", &month);
 8    printf("Enter the year value: ");
 9    scanf("%d",&year);
    /*Check Date Validity */
10    if (year >= 1900 && year <= 2025) {
11      if (month == 1 || month == 3 || month == 5 || month == 7 ||
          month == 8 || month == 10 || month == 12) {
```

(Contd.)...


```
12        if (day >= 1 && day <= 31) {
13          validDate = 1;
14        }
15        else {
16          validDate = 0;
17        }
18      }
19      else if (month == 2) {
20        int rVal=0;
21        if (year%4 == 0) {
22          rVal=1;
23          if ((year%100)==0 && (year % 400) !=0) {
24            rVal=0;
25          }
26        }
27        if (rVal ==1 && (day >=1 && day <=29) ) {
28          validDate = 1;
29        }
30        else if (day >=1 && day <= 28 ) {
31          validDate = 1;
32        }
33        else {
34          validDate = 0;
35        }
36      }
37      else if ((month >= 1 && month <= 12) && (day >= 1 && day <= 30)) {
38        validDate = 1;
39      }
40      else {
41        validDate = 0;
42      }
43    }
    /*Prev Date Calculation*/
44    if (validDate) {
45      if (day == 1) {
46        if (month == 1) {
47          year--;
48          day=31;
49          month=12;
50        }
51        else if (month == 3) {
```

(Contd.)...

```
53          int rVal=0;
54          if (year%4 == 0) {
55            rVal=1;
56            if ((year%100)==0 && (year % 400) !=0) {
57          rVal=0;
58        }
59            }
60          if (rVal ==1) {
61            day=29;
62            month--;
63            }
64          else {
65            day=28;
66            month--;
67            }
68        }
          else if (month == 2 || month == 4 || month == 6 || month == 9 ||
          month == 11) {
69          day = 31;
70          month--;
71          }
72        else {
73            day=30;
74            month--;
75          }
76      }
77      else {
78        day--;
79      }
80      printf("The next date is: %d-%d-%d",day,month,year);
81    }
82    else {
83      printf("The entered date ( %d-%d-%d ) is invalid",day,month, year);
84    }
85    getche ();
86    return 1;
87  }
```

Fig. 8.15: Program for previous date problem

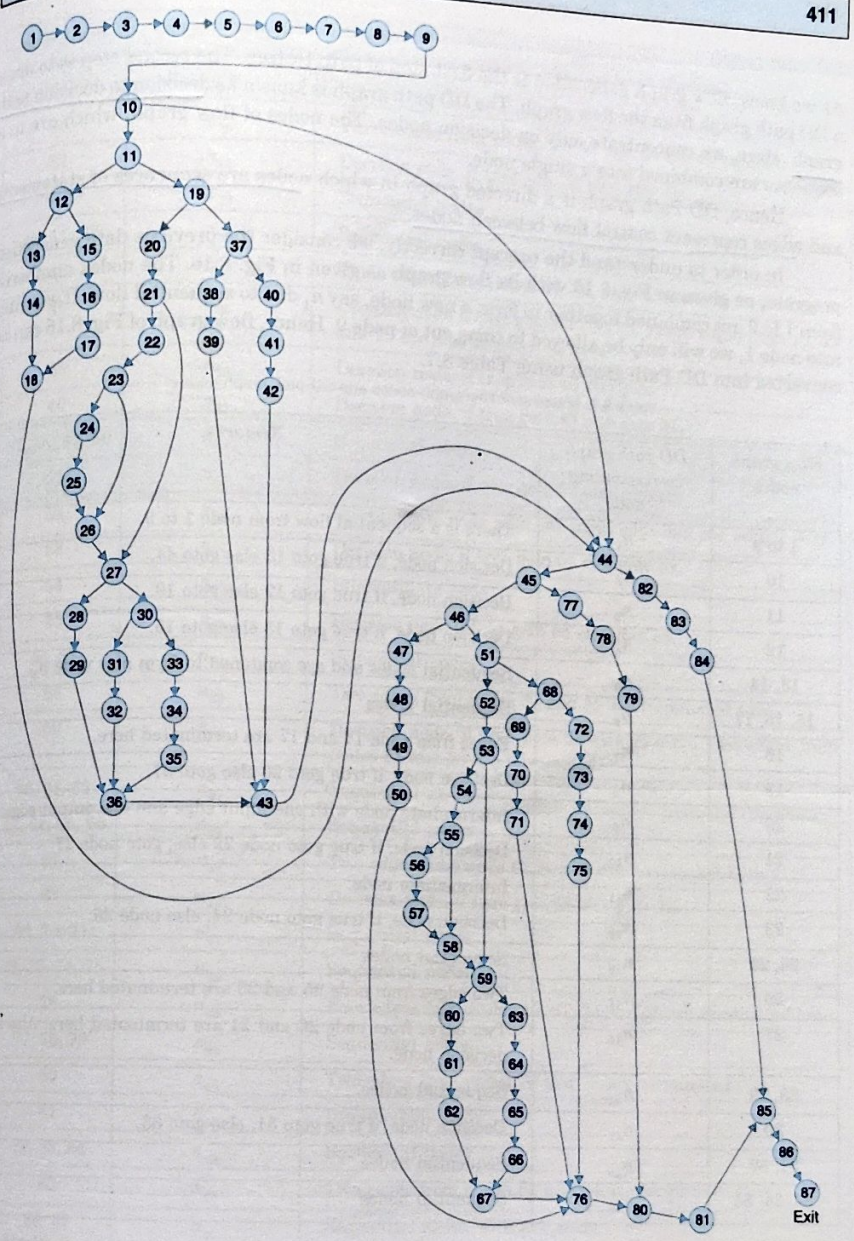

Fig. 8.16: Flow graph of previous date problem

DD path graph

As we know, flow graph generation is the first step of path testing. The second step is to draw a DD path graph from the flow graph. The DD path graph is known as decision to decision path graph. Here, we concentrate only on decision nodes. The nodes of flow graph, which are in a sequence are combined into a single node.

Hence, DD Path graph is a directed graph in which nodes are sequences of statements and edges represent control flow between nodes.

In order to understand the concept correctly, we consider the previous date generation program, as given in Fig. 8.15 with its flow graph as given in Fig. 8.16. The nodes numbered from 1 to 9 are combined together to form a new node, say n_1 due to sequential flow. If we enter into node 1, we will only be allowed to come out of node 9. Hence, flow graph of Fig. 8.16 can be converted into DD Path graph using Table 8.7.

Table 8.7: Mapping of flow graph nodes and DD path graph nodes

Flow graph nodes	*DD path graph corresponding node*	*Remarks*
1 to 9	n_1	There is a sequential flow from node 1 to 9.
10	n_2	Decision node, if true goto 13 else goto 44.
11	n_3	Decision node, if true goto 12 else goto 19.
12	n_4	Decision node, if true goto 13 else goto 15.
13, 14	n_5	Sequential nodes and are combined to form new node n_5.
15, 16, 17	n_6	Sequential nodes.
18	n_7	Edges from node 14 and 17 are terminated here.
19	n_8	Decision node, if true goto 20 else goto 37.
20	n_9	Intermediate node with one input edge and one output edge.
21	n_{10}	Decision node, if true goto node 22 else, goto node 27
22	n_{11}	Intermediate node
23	n_{12}	Decision node, if true goto node 24, else node 26.
24, 25	n_{13}	Sequential nodes
26	n_{14}	Two edges from node 25 and 23 are terminated here.
27	n_{15}	Two edges from node 26 and 21 are terminated here. Also a decision node.
28, 29	n_{16}	Sequential nodes.
30	n_{17}	Decision node, if true goto 31, else goto 33.
31, 32	n_{18}	Sequential nodes
33, 34, 35	n_{19}	Sequential nodes

(Contd.)...

Flow graph nodes	*DD path graph corresponding node*	*Remarks*
36	n_{20}	Three edges from nodes, 29, 32 and 35 are terminated here.
37	n_{21}	Decision node, if true goto 38 else goto 40.
38, 39	n_{22}	Sequential nodes
40, 41, 42	n_{23}	Sequential nodes
43	n_{24}	Three edges from nodes 36, 39, and 42 are terminated here.
44	n_{25}	Decision node if true goto 45 else 82. Three edges from 18, 43 and 10 are also terminated here.
45	n_{26}	Decision node, if true goto 46 else goto 77.
46	n_{27}	Decision node, if true goto 47 else goto 51.
47, 48, 49, 50	n_{28}	Sequential nodes
51	n_{29}	Decision node, if true goto 52 else goto 68.
52	n_{30}	Intermediate node with one input edge and one output edge.
53	n_{31}	Decision node, if true goto 54 else goto 59.
54	n_{32}	Intermediate node
55	n_{33}	Decision node if true goto 56 else goto 58.
56, 57	n_{34}	Sequential nodes
58	n_{35}	Two edges from nodes 57 and 55 are terminated here.
59	n_{36}	Decision node, if true goto 60 else goto 63. Two edges from nodes 58 and 53 are terminated.
60, 61, 62	n_{37}	Sequential nodes
63, 64, 65, 66	n_{38}	Sequential nodes
67	n_{39}	Two edges from node 62 and 66 are terminated here.
68	n_{40}	Decision node, if true goto 69 else goto 72.
69, 70, 71	n_{41}	Sequential nodes
72, 73, 74, 75	n_{42}	Sequential nodes
76	n_{43}	Four edges from nodes 50, 67, 71 and 75 are terminated here.
77, 78, 79	n_{44}	Sequential nodes
80	n_{45}	Two edges from nodes 76 and 79 are terminated.
81	n_{46}	Intermediate node
82, 83, 84	n_{47}	Sequential nodes
85	n_{48}	Two edges from nodes 81 and 84 are terminated here.
86, 87	n_{49}	Sequential nodes with exit node.

The DD path graph is given in Fig. 8.17.

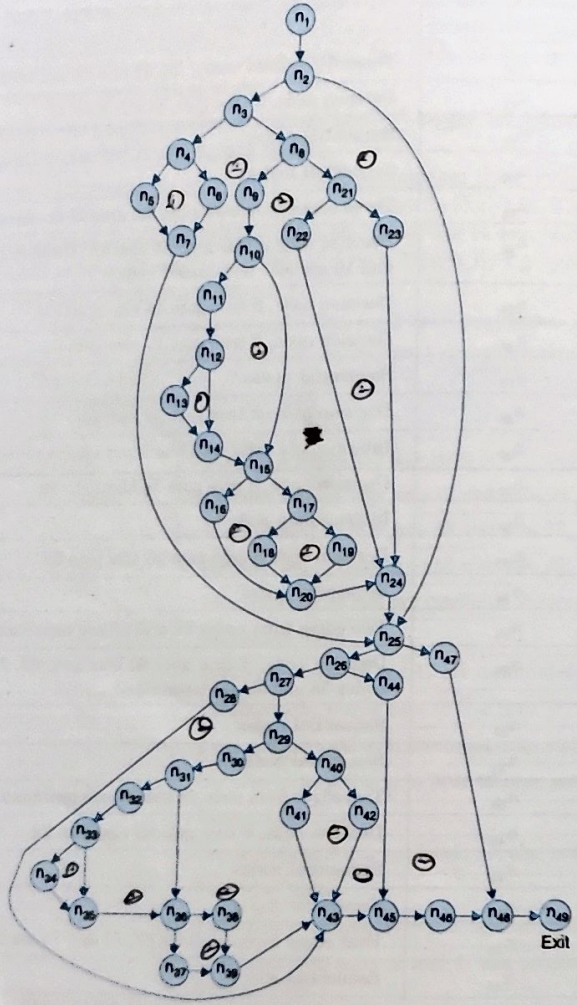

Fig. 8.17: DD path graph of previous date problem

Independent paths

The DD path graph is used to find independent paths. We are interested to execute all independent paths at least once during path testing.

An independent path is any path through the DD path graph that introduces at least one new set of processing statements or new conditions. Therefore, an independent path must move along at least one edge that has not been traversed before the path is defined.

We consider the previous date problem and its DD path graph which is given in Fig. 8.17. The independent paths are found and are given in Fig. 8.18. There are 18 independent paths.

	Independent paths of previous date problem
1	$n_1, n_2, n_{25}, n_{47}, n_{48}, n_{49}$
2	$n_1, n_2, n_3, n_4, n_5, n_7, n_{25}, n_{47}, n_{48}, n_{49}$
3	$n_1, n_2, n_3, n_4, n_6, n_7, n_{25}, n_{47}, n_{48}, n_{49}$
4	$n_1, n_2, n_3, n_8, n_{21}, n_{22}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
5	$n_1, n_2, n_3, n_8, n_{21}, n_{23}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
6	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{15}, n_{17}, n_{19}, n_{20}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
7	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{15}, n_{17}, n_{18}, n_{20}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
8	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{11}, n_{12}, n_{13}, n_{14}, n_{15}, n_{17}, n_{18}, n_{20}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
9	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{11}, n_{12}, n_{14}, n_{15}, n_{17}, n_{18}, n_{20}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
10	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{15}, n_{16}, n_{20}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
11	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{15}, n_{16}, n_{20}, n_{24}, n_{25}, n_{26}, n_{44}, n_{45}, n_{46}, n_{48}, n_{49}$
12	$n_1, n_2, n_3, n_8, n_9, n_{11}, n_{12}, n_{14}, n_{15}, n_{16}, n_{20}, n_{24}, n_{25}, n_{26}, n_{27}, n_{28}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
13	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{11}, n_{12}, n_{14}, n_{15}, n_{16}, n_{20}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{40}, n_{41}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
14	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{11}, n_{12}, n_{14}, n_{15}, n_{16}, n_{20}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{40}, n_{42}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
15	$n_1, n_2, n_3, n_8, n_{21}, n_{22}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{30}, n_{31}, n_{36}, n_{38}, n_{39}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
16	$n_1, n_2, n_3, n_8, n_{21}, n_{22}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{30}, n_{31}, n_{36}, n_{37}, n_{39}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
17	$n_1, n_2, n_3, n_8, n_{21}, n_{22}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{30}, n_{31}, n_{32}, n_{33}, n_{34}, n_{35}, n_{36}, n_{37}, n_{39}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
18	$n_1, n_2, n_3, n_8, n_{21}, n_{22}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{30}, n_{31}, n_{32}, n_{33}, n_{35}, n_{36}, n_{37}, n_{39}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$

Fig. 8.18: Independent paths of previous date problem

It is quite interesting to use independent paths in order to ensure that

(i) Every statement in the program has been executed at least once.

(ii) Every branch has been exercised for true and false conditions.

There are high quality commercial tools that generate the DD path graph of a given program. The vendors make sure that the products work for wide variety of programming languages. In practice, it is reasonable to make DD Path graphs for programs upto about 100 source lines. Beyond that, we should go for a standard tool.

Example 8.13

Consider the 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 is given in Fig. 8.19. The output may have one of the following words:

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

Draw the flow graph and DD path graph. Also find independent paths from the DD path graph.

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
1     int main()
2   {
3     int a,b,c,validInput=0,d;
4     double D;
5     printf("Enter the 'a' value: ");
6     scanf("%d",&a);
7     printf("Enter the 'b' value: ");
8     scanf("%d",&b);
9     printf("Enter the 'c' value: ");
10    scanf("%d",&c);
11    if ((a >= 0) && (a <= 100) && (b >= 0) && (b <= 100) && (c >= 0)
        && (c <= 100)) {
12      validInput = 1;
13      if (a == 0) {
14        validInput = -1;
15      }
16    }
17    if (validInput==1) {
18      d = b*b - 4*a*c;
19      if (d == 0) {
20        printf("The roots are equal and are r1 = r2 = %f\n",
                -b/(2*(float) a));
21      }
22      else if ( d > 0 ) {
23        D=sqrt(d);
24        printf("The roots are real and are r1 = %f and r2 = %f\n",
                (-b-D)/(2* a), (-b+D)/(2* a));
25      }
26      else {
27        D=sqrt(-d)/(2*a);
28        printf("The roots are imaginary and are r1 = (%f,%f) and
                r2 = (%f,%f)\n", -b/(2.0*a),D,-b/(2.0*a),-D);
29      }
30    }
31    else if (validInput == -1) {
```

(Contd.)...


```
32      printf("The values do not constitute a Quadratic equation.");
33    }
34    else {
35      printf("The inputs belong to invalid range.");
36    }
37    getche();
38    return 1;
39  }
```

Fig. 8.19: Code of quadratic equation problem

Solution

The flow graph is given below:

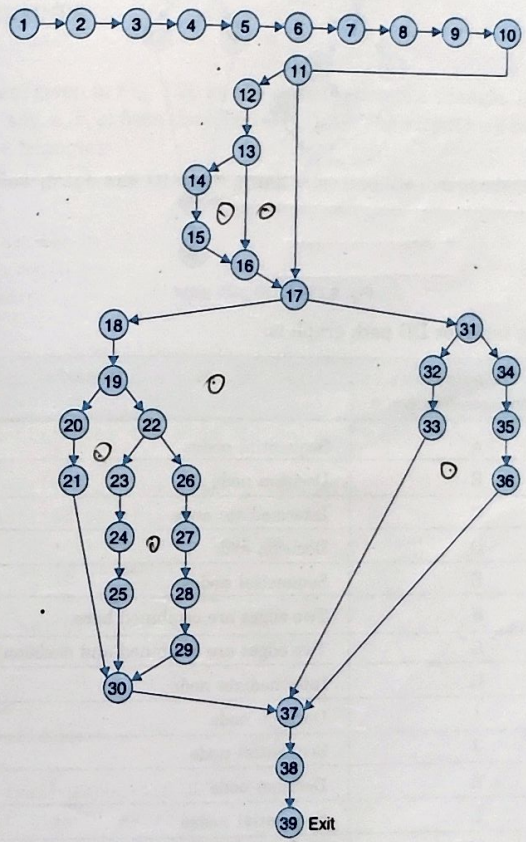

Fig. 8.19(a): Program flow graph

DD Path graph is given below :

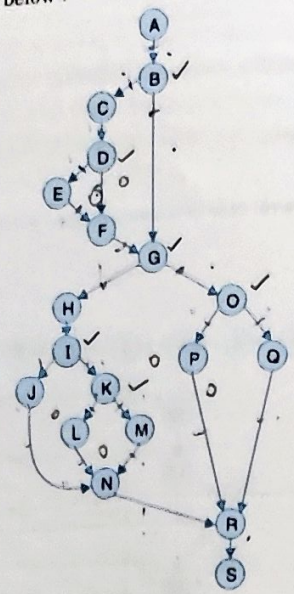

Fig. 8.19(b): DD path graph

The mapping table for DD path graph is:

Flow graph nodes	*DD path graph corresponding nodes*	*Remarks*
1 to 10	A	Sequential nodes
11	B	Decision node
12	C	Intermediate node
13	D	Decision node
14, 15	E	Sequential nodes
16	F	Two edges are combined here
17	G	Two edges are combined and decision node
18	H	Intermediate node
19	I	Decision node
20, 21	J	Sequential node
22	K	Decision node
23, 24, 25	L	Sequential nodes
26, 27, 28, 29	M	Sequential nodes

(Contd.)...

30	N	Three edges are combined
31	O	Decision node
32, 33	P	Sequential nodes
34, 35, 36	Q	Sequential nodes
37	R	Three edges are combined
38, 39	S	Sequential nodes with exit node.

Independent paths are:

(*i*) ABGOQRS
(*ii*) ABGOPRS
(*iii*) ABCDFGOQRS
(*iv*) ABCDEFGOPRS
(*v*) ABGHIJNRS
(*vi*) ABGHIKLNRS
(*vii*) ABGHIKMNRS

Example 8.14

Consider a program given in Fig. 8.20 for the classification of a triangle. Its input is a triple of positive integers (say, *a*, *b*, *c*) from the interval [1, 100]. The output may be [Scalene, Isosceles, Equilateral, Not a triangle].

Draw the flow graph and DD path graph. Also find the independent paths from the DD path graph.

```
#include <stdio.h>
#include <conio.h>
1   int main()
2   {
3     int a,b,c,validInput=0;
4     printf("Enter the side 'a' value: ");
5     scanf("%d",&a);
6     printf("Enter the side 'b' value: ")
7     scanf("%d",&b);
8     printf("Enter the side 'c' value:");
9     scanf("%d",&c);
10    if ((a > 0) && (a <= 100) && (b > 0) && (b <= 100) && (c > 0)
          && (c <= 100)) {
11      if ( (a + b) > c) && ((c + a) > b) && ((b + c) > a)) {
12        validInput = 1;
13      }
14    }
15    else {
16      validInput = -1;
17    }
18    If (validInput==1) {
19      If ((a==b) && (b==c)) {
20        printf("The trinagle is equilateral");
21      }
22      else if ( (a == b) || (b == c) || (c == a) ) {
```

(Contd.)...

```
23          printf("The triangle is isosceles");
24        }
25        else {
26          printf("The trinagle is scalene");
27        }
28      }
29      else if (validInput == 0) {
30        printf("The values do not constitute a Triangle");
31      }
32      else {
33        printf("The inputs belong to invalid range");
34      }
35      getche();
36      return 1;
37    }
```

Fig. 8.20: Code of triangle classification problem

Flow graph of trinagle problem is:

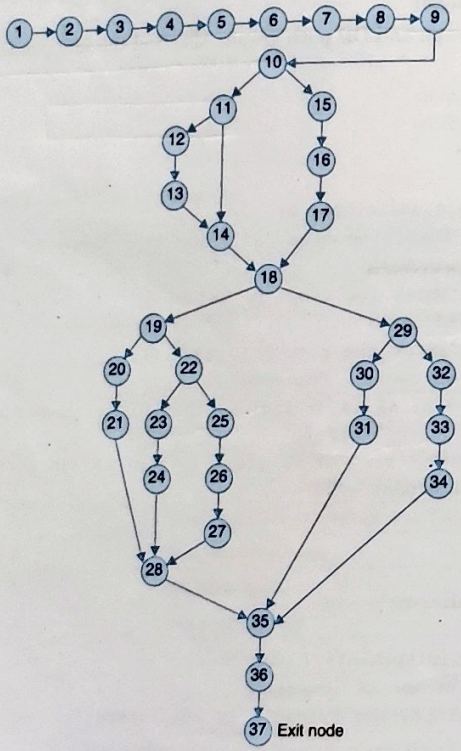

Fig. 8.20(a): Program flow graph

The mapping table for DD path graph is:

Flow graph nodes	*DD path graph corresponding nodes*	*Remarks*
1 to 9	A	Sequential nodes
10	B	Decision node
11	C	Decision node
12, 13	D	Sequential nodes
14	E	Two edges are joined here
15, 16, 17	F	Sequential nodes
18	G	Decision nodes plus joining of two edges
19	H	Decision node
20, 21	I	Sequential nodes
22	J	Decision node
23, 24	K	Sequential nodes
25, 26, 27	L	Sequential nodes
28	M	Three edges are combined here
29	N	Decision node
30, 31	O	Sequential nodes
32, 33, 34	P	Sequential nodes
35	Q	Three edges are combined here
36, 37	R	Sequential nodes with exit node

DD path graph is given in Fig. 8.20 (*b*).

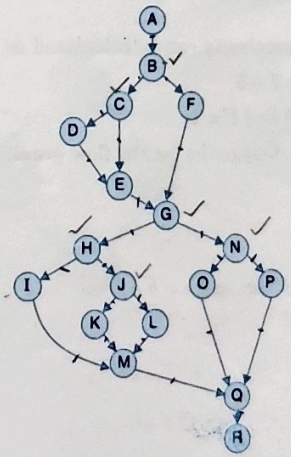

Fig. 8.20(b): DD path graph

Independent paths are:

(*i*) ABFGNPQR
(*ii*) ABFGNOQR
(*iii*) ABCEGNPQR
(*iv*) ABCDEGNOQR
(*v*) ABFGHIMQR
(*vi*) ABFGHJKMQR
(*vii*) ABFGHJLMQR

8.4.2 Cyclomatic Complexity

The cyclomatic complexity is also known as structural complexity because it gives internal view of the code. This approach is used to find the number of independent paths through a program. This provides us the upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once and every condition has been executed on its true and false side. If a program has backward branch then it may have infinite number of paths. Although it is possible to define a set of algebraic expressions that gives the total number of possible paths through a program, however, using total number of paths has been found to be impractical. Because of this, the complexity measure is defined in terms of independent paths—that when taken in combination will generate every possible path [MCCA76]. An independent path is any path through the program that introduces at least one new set of processing statements or a new condition.

McCabe's cyclomatic metric [MCCA 76] V(G) of a graph G with n vertices, e edges, and P connected components is $V(G) = e - n + 2P$.

Given a program we will associate with it a directed graph that has unique entry and exit nodes. Each node in the graph corresponds to a block of code in the program where the flow is sequential and the arcs correspond to branches taken in the program. This graph is classically known as flow graph and it is assumed that each node can be reached by the entry node and each node can reach the exit node. For example, a flow graph shown in Fig. 8.21 with entry node '*a*' and exit node '*f*'.

The value of cyclomatic complexity can be calculated as

$$V(G) = 9 - 6 + 2 = 5$$

Here $e = 9, n = 6$ and $P = 1$

There will be five independent paths for the flow graph illustrated in Fig. 8.21.

path 1 : *acf*
path 2 : *abef*
path 3 : *adcf*
path 4 : *abeacf* or *abeabef*
path 5 : *abebef*

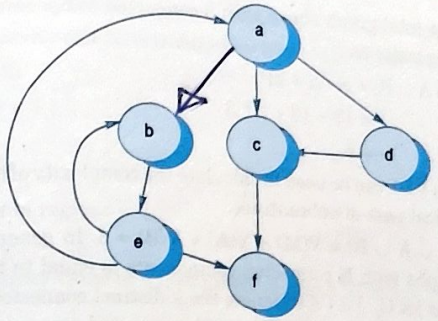

Fig. 8.21

Notice that the sequence of an arbitrary number of nodes always has unit complexity and that cyclomatic complexity conforms to our intuitive notion of minimum number of paths. Several properties of cyclomatic complexity are stated below:

1. $V(G) \geq 1$
2. V(G) is the maximum number of independent paths in graph G.
3. Inserting and deleting functional statements to G does not affect V(G).
4. G has only one path if and only if $V(G) = 1$.
5. Inserting a new row in G increases V(G) by unity.
6. V(G) depends only on the decision structure of G.

The role of P in the complexity calculation $V(G) = e - n + 2P$ is required to be understood correctly. We define a flow graph with unique entry and exit nodes, all nodes reachable from the entry, and exit reachable from all nodes. This definition would result in all flow graphs having only one connected component. One could, however, imagine a main program M and two called subroutines A and B having a flow graph shown in Fig. 8.22.

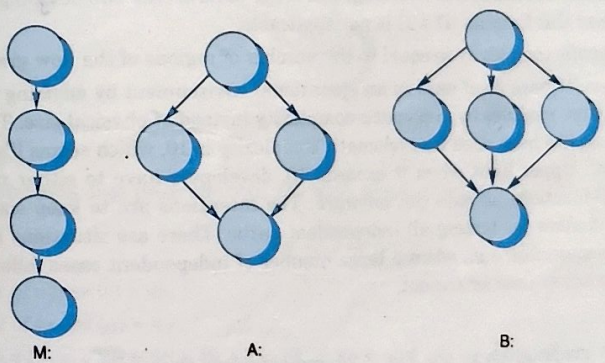

Fig. 8.22

Let us denote the total graph above with 3 connected components as M ∪ A ∪ B. Since P = 3, we calculate complexity as

$$V(M \cup A \cup B) = e - n + 2P$$
$$= 13 - 13 + 2 * 3$$
$$= 6$$

This method with $P \neq 1$ can be used to calculate the complexity of a collection of programs, particularly a hierarchical nest of subroutines.

Notice that V(M ∪ A ∪ B) = V(M) + V(A) + V(B) = 6. In general, the complexity of a collection C of flow graphs with K connected components is equal to the summation of their complexities. To see this let C_i, $1 \leq i \leq K$ denote the k distinct connected component, and let e_i and n_i be the number of edges and nodes in the ith-connected component. Then

$$V(C) = e - n + 2p = \sum_{i=1}^{k} e_i - \sum_{i=1}^{k} n_i + 2K$$

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

Since the calculation V = e − n + 2P can be quite tedious for a developer, an effort has been made to simplify the complexity calculations. Two alternate methods are available for the complexity calculations.

1. Cyclomatic complexity V(G) of a flow graph G is equal to the number of predicate (decision) nodes plus one [MILL72].

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

Where Π is the number of predicate nodes contained in the flow graph G.

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

2. Cyclomatic complexity is equal to the number of regions of the flow graph.

These results have been used in an operational environment by advising developers to limit their software modules by cyclomatic complexity instead of physical size. The particular upper bound that has been used for cyclomatic complexity is 10, which seems like reasonable, but not magical. Upper limit when it exceeds 10, developers have to either recognise and modularise sub-functions or redo the software. The intentions are to keep size of modules manageable and allow for testing all independent paths. There are situations in which this limit seems unreasonable: *e.g.*, when a large number of independent cases follow a selection function like switch or case statement.

Example 8.15

Consider a flow graph given in the Fig. 8.23 and calculate the cyclomatic complexity by all three methods.

Solution

Cyclomatic complexity can be calculated by any of the three methods.

1. V(G) = e − n + 2P
   = 13 − 10 + 2 = 5
2. V(G) = Π + 1
   = 4 + 1 = 5
3. V(G) = number of regions
   = 5

Therefore, complexity value of a flow graph in Fig. 8.23 is 5.

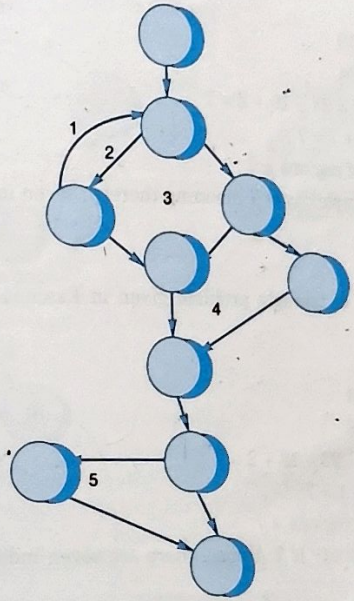

Fig. 8.23: [MCCA76]

Example 8.16

Consider the previous date program with DD path graph given in Fig. 8.17. Find cyclomatic complexity.

Solution

Number of edges (*e*) = 65

Number of nodes (*n*) = 49

(*i*) V(G) = e − n + 2P = 65 − 49 + 2 = 18

(*ii*) V(G) = Π + 1 = 17 + 1 = 18

(*iii*) V(G) = Number of regions = 18.

Therefore cyclomatic complexity is 18. This value is quite high and indicates that there is a need to redesign the program. If we review the code, it is clear that we may have two separate modules, one for "checking the validity of date" and another for "previous date calculation". Actually, these two functions are independent and should be placed in different modules. If we do so, cyclomatic complexity will be reduced to 10 and 8 respectively (split occurs at n_{25}). Hence, this method gives us some idea about the structure of the code and modularity of the program.

Example 8.17

Consider the quadratic equation problem given in Example 8.13 with its DD path graph. Find the cyclomatic complexity:

Solution

Number of nodes = 19

Number of edges = 24

(*i*) V(G) = $e - n + 2P = 24 - 19 + 2 = 7$

(*ii*) V(G) = $\Pi + 1 = 6 + 1 = 7$

(*iii*) V(G) = Number of regions = 7

Hence cyclomatic complexity is 7 meaning thereby, seven independent paths in the DD path graph.

Example 8.18

Consider the classification of triangle problem given in Example 8.14. Find the cyclomatic complexity.

Solution

Number of edges = 23

Number of nodes = 18

(*i*) V(G) = $e - n + 2P = 23 - 18 + 2 = 7$

(*ii*) V(G) = $\Pi + 1 = 6 + 1 = 7$

(*iii*) V(G) = Number of regions = 7

The cyclomatic complexity is 7. Hence, there are seven independent paths as given in Example 8.14.

8.4.3 Graph Matrices

Whenever graphs are used for testing, we are interested to find independent paths. The objective is to trace all links of the graph at least once. Path tracing is not an easy task and is subject to errors. If the size of graph increases, it becomes difficult to do path tracing manually. In practice, it is always advisable to go for testing tool. To develop such a tool, a data structure, called graph matrix can be quite helpful.

A graph matrix is a square matrix with one row and one column for every node in the graph. The size of the matrix (*i.e.*, the number of rows and columns) is equal to the number of nodes in the flow graph. Some examples of graphs and associated matrices [BEIZ90] are shown in Fig. 8.24.

In the graph matrix, there is a place to put every possible direct connection between any node and any other node. A connection from node *i* to node *j* does not imply a connection from

node *j* to node *i*. In Fig. 8.24 (*c*) the (5, 6) entry is m but the (6, 5) entry is c. If there are several links between two nodes, then the entry is a sum ; the '+' sign denotes parallel links.

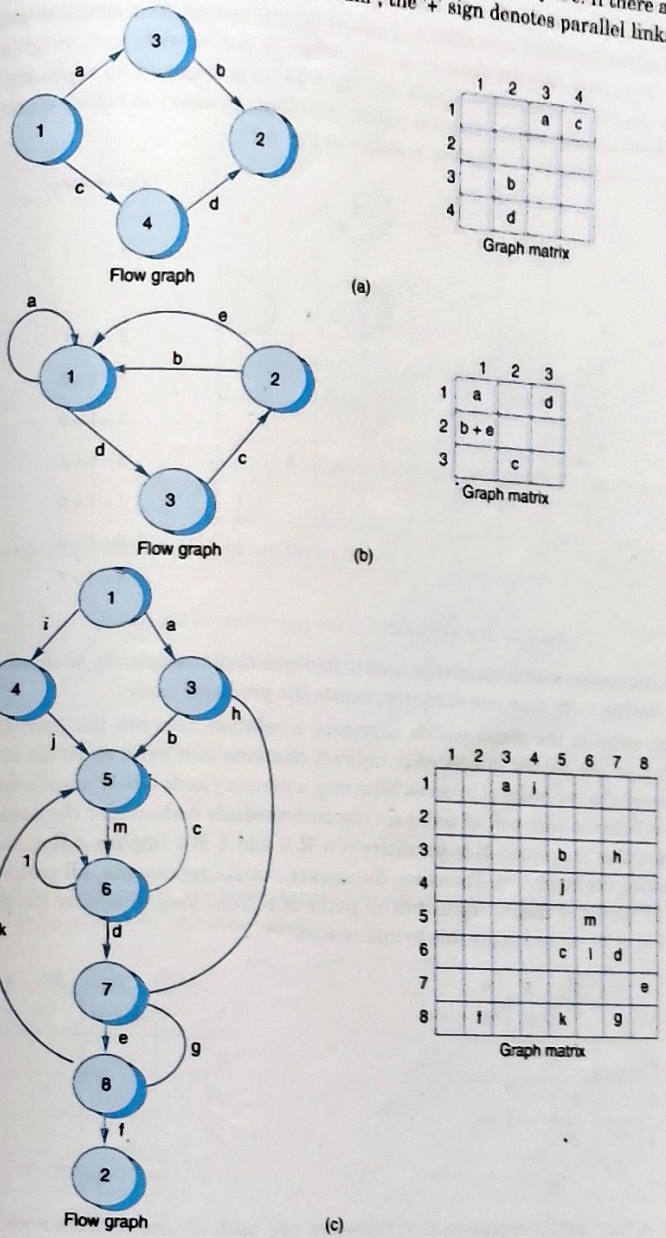

Fig. 8.24: Flow graphs and graphs matrices

Graph matrix is nothing but the tabular representation of a flow graph. It does not seem to be useful in the present form. If we assign weight to each entry, the graph matrix can be used for evaluating useful information required during testing. The simplest weight is 1, if there is a connection and 0 if there is no connection. A matrix with such weights is called a connection matrix. A connection matrix for Fig. 8.24 (c) is obtained by replacing each entry with 1, if there is a link and 0 if there is no link [BEIZ90]. As usual, to reduce clutter we do not write down 0 entries and this matrix is shown in Fig. 8.25.

	1	2	3	4	5	6	7	8	Connections
1			1	1					2 – 1 = 1
2									
3					1		1		2 – 1 = 1
4					1				1 – 1 = 0
5						1			1 – 1 = 0
6					1	1	1		3 – 1 = 2
7								1	1 – 1 = 0
8		1			1		1		3 – 1 = 2
									6 + 1 = 7

Fig. 8.25: Connection matrix of flow graph shown in Fig. 8.24 (c)

The connection matrix can also be used to find cyclomatic complexity as shown in Fig. 8.25. Each row having more than one entry represents the predicate node.

Each entry in the graph matrix expresses a relation between the pair. It is a direct relation, but we are usually interested in indirect relations that exist by virtue of intervening nodes between the two nodes of interest. Squaring a matrix yields a new matrix that expresses the relation between each pair of nodes via one intermediate node under the assumption that node is transitive. A relation **R** is transitive if a **R** b and b **R** c implies a **R** c. Most relations used in testing are transitive. Therefore, the square matrix represents, all paths of two links long. The K^{th} power of matrix represents all paths of K links long. Consider the graph matrix shown in Fig. 8.24 (a) and apply this to that matrix.

	1	2	3	4
1			a	c
2				
3		b		
4		d		

[A]

	1	2	3	4
1		ab + cd		
2				
3				
4				

$[A]^2$

The square matrix represent that there are two path ab and cd from node 1 to node 2.

Example 8.19

Consider the flow graph shown in the Fig. 8.26 and draw the graph and connection matrices. Find out cyclomatic complexity and two/three link paths from a node to any other node.

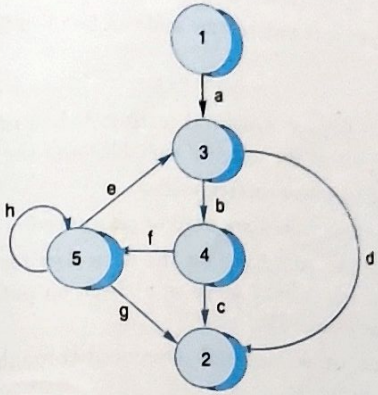

Fig. 8.26: Flow graph [BEIZ90]

Solution

The graph and connection matrices are given below:

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

Graph matrix (A)

	1	2	3	4	5	Connections
1			1			1 – 1 = 0
2						
3		1		1		2 – 1 = 1
4		1			1	2 – 1 = 1
5		1	1		1	3 – 1 = 2
						4 + 1 = 5

Connection matrix

To find two link paths, we have to generate a square of graph matrix [A] and for three link paths, a cube of matrix [A] is required.

	1	2	3	4	5
1		ad		ab	
2					
3		bc			bf
4		fg	fe		fh
5		ed + hg	he	eb	h^2

$[A^2]$

	1	2	3	4	5
1		abc			afb
2					
3		bfg	bfe		bfh
4		fed + fhg	fhe	feb	fh^2
5		ebc + hed + h^2g	h^2e	heb	ebf + h^3

$[A^3]$

This indicates that there is a two links path "*ad*" and three-link path "*abc*" available from node 1 to node 2.

Our main objective is to use matrix operations to obtain the set of all paths between all nodes. This can be obtained by summing A, A^2, A^3, ... A^{n-1}.

These operations are easy to programme and can be used for designing testing tools.

8.4.4 Data Flow Testing

Data flow testing is another form of structural testing. It has nothing to do with data flow diagrams. Here, we concentrate on the usage of variables and the focus points are:

(*i*) Statements where variables receive values.

(*ii*) Statement where these values are used or referenced.

Flow graphs are also used as a basis for the data flow testing as in the case of path testing. Some times, we feel that it may serve as a check on path testing and is treated as another form of path testing [JORG95].

As we know, variables are defined and referenced throughout the program. We may have few define/reference anomalies:

(*i*) A variable is defined but not used/referenced.

(*ii*) A variable is used but never defined.

(*iii*) A variable is defined twice before it is used.

These anomalies can be identified by static analysis of code *i.e.,* analysing code without executing it. In order to formalise the approach of data flow testing, some definitions are required, which are discussed below [JORG 95].

Definitions

The definitions refer to a program P that has a program graph G (P) and a set of program variables V. The G (P) has a single entry node and a single exit node. The set of all paths in P is PATHS (P).

(*i*) **Defining node:** Node $n \in$ G (P) is a defining node of the variable $v \in$ V written as DEF(v, n), iff the value of the variable v is defined at the statement fragment corresponding to node n.

(*ii*) **Usage node:** Node $n \in$ G (P) is a usage node of the variable $v \in$ V, written as USE (v, n), iff the value of the variable v is used at the statement fragment corresponding to node n. A usage node USE (v, n) is a predicate use (denoted as p) iff statement n is a predicate statement otherwise USE (v, n) is a computation use (denoted as c).

(*iii*) **Definition use:** A definition use path with respect to a variable v (denoted du-path) is a path in PATHS (P) such that, for some $v \in$ V, there are define and usage nodes DEF (v, m) and USE (v, n) such that m and n are the initial and final nodes of the path.

(*iv*) **Definition clear:** A definition clear path with respect to a variable v (denoted dc-path) is a definition use path in PATHS (P) with initial and final nodes DEF (v, m) and USE (v, n), such that no other node in the path is a defining node of v.

The du-paths and dc paths describe the flow of data across source statements from points at which the values are defined to points at which the values are used. The du-paths that are not definition clear are potential trouble spots.

Hence, our objective is to find all du-paths and then identify those du-paths which are not dc-paths. The steps are given in Fig. 8.27. We may like to generate specific test cases for du-paths that are not dc-paths.

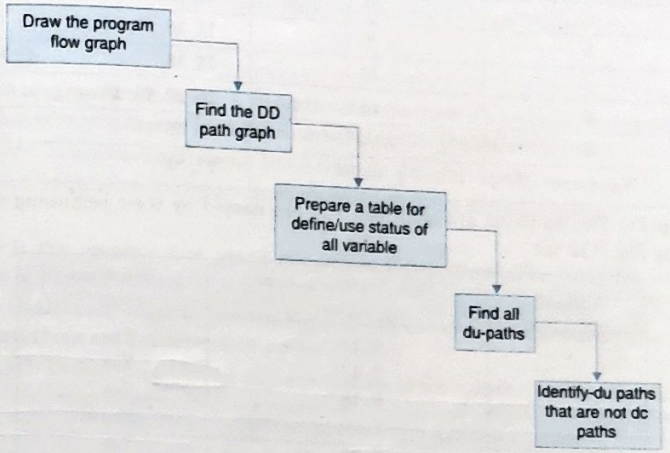

Fig. 8.27: Steps for data flow testing

One simple approach is to test every du-path at least once. This is known as du testing strategy. In this strategy, chance to cover all edges of the flow graph is very high; although it does not guarantee 100% coverage. This is also effective for error detection specifically for du-paths that are not definition clear paths.

Example 8.20

Consider the program of 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 for each of these may be from interval [0, 100]. The program is given in Fig. 8.19. The output may have one of the options given below:

(*i*) Not a quadratic equation

(*ii*) real roots

(*iii*) imaginary roots

(*iv*) equal roots

(*v*) invalid inputs.

Find all du-paths and identify those du-paths that are definition clear.

Solution

Step I: The program flow graph is given in Fig. 8.19 (*a*). The variables used in the program are a, b, c, d, validinput, D.

Step II: DD path graph is given in Fig. 8.19 (*b*). The cyclomatic complexity of this graph is 7 indicating there are seven independent paths.

Step III: Define/Use nodes for all variables are given below:

Variable	*Defined at node*	*Used at node*
a	6	11, 13, 18, 20, 24, 27, 28
b	8	11, 18, 20, 24, 28
c	10	11, 18
d	18	19, 22, 23, 27
D	23, 27	24, 28
Validinput	3, 12, 14	17, 31

Step IV: The du-paths are identified and are named by their beginning and ending nodes using Fig. 8.19 (*a*).

Variable	*Path (beginning, end) nodes*	*Definition clear ?*
a	6, 11	Yes
	6, 13	Yes
	6, 18	Yes
	6, 20	Yes
	6, 24	Yes
	6, 27	Yes
	6, 28	Yes
b	8, 11	Yes
	8, 18	Yes
	8, 20	Yes
	8, 24	Yes
	8, 28	Yes
c	10, 11	Yes
	10, 18	Yes
d	18, 19	Yes
	18, 22	Yes
	18, 23	Yes
	18, 27	Yes
D	23, 24	Yes
	23, 28	Path not possible
	27, 24	Path not possible
	27, 28	Yes
Validinput	3, 17	no
	3, 31	no
	12, 17	no
	12, 31	no
	14, 17	Yes
	14, 31	Yes

Total du-paths are 26 out of which 4 paths are not definition clear paths. Two path < 27, 24 >, < 23, 28 > are impossible paths. Our emphasis should be to generate test cases for all 26 du-paths, and if not possible, then, at least for 4 du-paths that are not definition clear paths.

Example 8.21

Consider the program given in Fig. 8.20 for the classification of a triangle. Its input is a triple of positive integers (say *a*, *b*, *c*) from the interval [1, 100]. The output may be:

[Scalene, Isosceles, Equilateral, Not a triangle, Invalid inputs].

Find all du-paths and identify those du-paths that are definition clear.

Solution

Step I: The program flow graph is given in Fig. 8.20 (*a*). The variables used in the program are *a*, *b*, *c*, validinput.

Step II: DD path graph is given in Fig. 8.20 (*b*). The cyclomatic complexity of the graph is 7 and, thus, there are 7 independent paths.

Step III: Define/use nodes for all variables are given below:

Variable	*Defined at node*	*Used at node*
a	6, 5	10, 11, 19, 22
b	7	10, 11, 19, 22
c	9	10, 11, 19, 22
Validinput	3, 12, 16	18, 29

Step IV: The du-paths are identified and are named by their beginning and ending nodes using Fig. 8.20 (*a*).

Variable	*Path (beginning, end) nodes*	*Definition clear ?*
a	5, 10	Yes
	5, 11	Yes
	5, 19	Yes
	5, 22	Yes
b	7, 10	Yes
	7, 11	Yes
	7, 19	Yes
	7, 22	Yes
c	9, 10	Yes
	9, 11	Yes
	9, 19	Yes
	9, 22	Yes

Variable	Path (beginning, end) nodes	Definition clear ?
Validinput	3, 18	no
	3, 29	no
	12, 18	no
	12, 29	no
	16, 18	Yes
	16, 29	Yes

Hence total du-paths are 18 out of which four paths are not definition clear.

8.4.5 Mutation Testing

Mutation testing is a fault based technique that is similar to fault seeding, except that mutations to program statements are made in order to determine properties about test cases. It is basically a fault simulation technique. In this technique, multiple copies of a program are made, and each copy is altered; this altered copy is called a mutant. Mutants are executed with test data to determine whether the test data are capable of detecting the change between the original program and the mutated program. A mutant that is detected by a test case is termed "killed" and the goal of mutation procedure is to find a set of test cases that are able to kill groups of mutant programs [FRIE 95].

Mutants are produced by applying mutant operators. An operator is essentially a grammatical rule that changes a single expression to another expression. The new expression should be syntactically legal according to the language. If one or more mutant operators are applied to all expressions in a program, the result is a large set of mutants, all of which must be killed by the test cases or shown to be equivalent to the original expression.

When we mutate code there needs to be a way of measuring the degree to which the code has been modified. For example, if the original expression is $x + 1$ and the mutant for that expression is $x + 2$, that is a lesser change to the original code than a mutant such as (c*22), where both the operand and the operator are changed. We may have a ranking scheme, where a first order mutant is a single change to an expression, a second order mutant is a mutation to a first order mutant, and so on. High order mutants becomes intractable and thus in practice only low order mutants are used.

One difficulty associated with whether mutants will be killed is the problem of reaching the location; if a mutant is not executed, it cannot be killed. Special test cases are to be designed to reach a mutant. For example, suppose, we have the code

Read (a, b, c);
If $(a > b)$ and $(b = c)$ then
$x: = a*b*c$; {make mutants; m_1, m_2, m_3.....}

To execute this, input domain must contain a value such that a is greater than b and b equals c. If input domain does not contain such a value, then all mutants made at this location should be considered equivalent to the original program, because the statement $x: = a*b*c$ is dead code (code that cannot be reached during execution). If we make the mutant $x + y$ for

$x + 1$, then we should take care about the value of y which should not be equal to 1 for designing a test case.

The manner by which a test suite is evaluated (scored) via mutation testing is as follows: for a specific test suite and a specific set of mutants, there will be three types of mutants in the code *i.e.*, killed or dead, live, equivalent. The sum of the number of live, killed, and equivalent mutants will be the total number of mutants created. The score associated with a test suite T and mutants M is simply

$$\frac{\#\text{ killed}}{\#\text{ total} - \#\text{ equivalent}} \times 100\%$$

This equation allows us to determine the likelihood that for a particular mutation adequate test suite will catch real faults based on how well the suite-killed mutants. Note that this scheme does not penalize a test suite if it is of a large size than a different test suite. For instance, suppose that test suite A with 100 test cases had a score of 50%, and test suite B with 25 test cases had a score of 49%. Although A has a better mutation score, it is also four times as large as B, and for the small increase in mutation score there is a large increase in testing costs.

8.5 LEVELS OF TESTING

Our emphasis during testing is to examine and modify the source code. There are three levels of testing *i.e.*, individual module to the entire software system. At one end, we attempt to test modules in all possible ways so as to detect any errors. From there, we combine to form aggregates of modules and test their detailed structure and functions. At the end, we may ignore the internal structure of the software and concentrate on how it responds to the typical kind of operations that will be requested by the user. These three levels of testing are usually referred to as unit testing, integration testing, and system testing [JONE 90] as shown in Fig. 8.28.

Out of the three traditional levels of testing, unit testing is best understood. The testing methods discussed so far in this chapter are directly applicable to unit testing. System testing is understood better than integration testing, but both need clarification. The bottom up approach sheds some insight: test the individual units, and then integrate these into subsystems until the entire system is tested. System testing is something that the customer understands, and it often borders on customer acceptance testing. Generally, system testing is functional rather than structural. This is mostly due to the absence of a structural basis for system test cases. In the traditional view, integration testing is what's leftover; it is not unit testing, and it is not system testing. Most of the usual discussions on integration testing concentrate on order in which units are integrated: top-down, bottom-up, or the big bang (everything at once) of the three phases, integration is the least well understood [JORG 95].

Errors located during testing may fall into several categories [DEUT 82]. Those that require immediate attention are usually of a nature that they crash the software under consideration, and testing cannot continue until they are removed. Some errors need to be corrected before testing is complete but can be ignored during the immediate testing schedule. In general, the importance of removing an error is proportional to its severity, the frequency with which it

occurs, and the degree to which the customer is aware of it. There are errors that may be acceptable to the user when compared to the cost of correcting them at the moment; these will be held until some future release date or until the customer becomes concerned enough to request a change.

Finally, there are errors, usually non-reproducible, for which insufficient evidence exists to evaluate them. They are flagged for ongoing consideration until they become more tractable. Non-reproducible errors are a frequent problem with concurrent software.

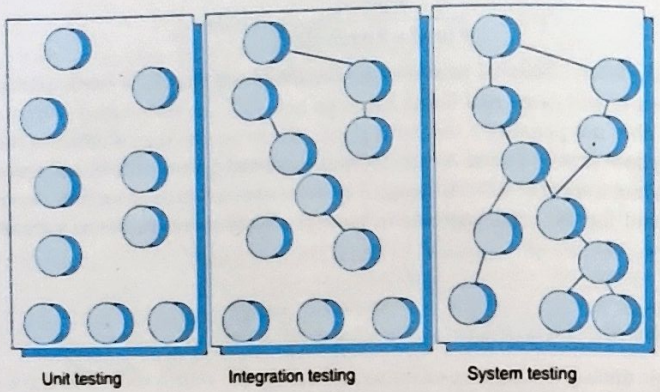

Fig. 8.28: Levels of testing

Number of other activities are also carried out alongwith testing. These include test documentation, product modification, and discussions with the customer to schedule installation, preparation of installation and training material etc. When the product has finally been released by the quality assurance group at the end of testing, we want to move immediately into the installation portion of product delivery.

8.5.1 Unit Testing

Unit testing is the process of taking a module and running it in isolation from the rest of the software product by using prepared test cases and comparing the actual results with the results predicted by the specifications and design of the module. One purpose of testing is to find (and remove) as many errors in the software as practical. There are number of reasons in support of unit testing than testing the entire product [JONE 90].

1. The size of a single module is small enough that we can locate an error fairly easily.
2. The module is small enough that we can attempt to test it in some demonstrably exhaustive fashion.
3. Confusing interactions of multiple errors in widely different parts of the software are eliminated.

There are problems associated with testing a module in isolation. How do we run a module without anything to call it, to be called by it or, possibly, to output intermediate values obtained during execution? One approach is to construct an appropriate driver routine to call it and, simple stubs to be called by it, and to insert output statements in it.

Stubs serve to replace modules that are subordinate to (called by) the module to be tested. A stub or dummy subprogram uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns [PRES 97].

This overhead code, called scaffolding represents effort that is important to testing, but does not appear in the delivered product as shown in Fig. 8.29 [JONE 90]. If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many modules cannot be adequately unit tested with simple overhead software. In such cases, complete testing can be postponed until the integration test step (where drivers and stubs are also used). A second approach is to generate the scaffolding automatically by means of a test harness. A test harness among other things, allows us to own a single unit in isolation, while simulating the rest of the software system environment by providing appropriate input, output, parameters, and interaction for the unit. A third and rather ineffective technique is to omit unit testing and simply to allow incremental addition of modules to a partially integrated product, hoping that the integration testing will also provide sufficient coverage of the module's structure. This technique is usually inadequate but nevertheless it is often recommended in the literature. The white box testing approaches are normally used for unit testing and the steps can be conducted in parallel for multiple modules.

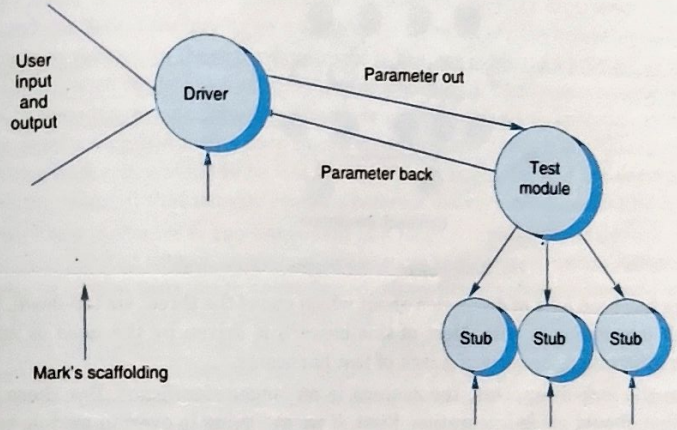

Fig. 8.29: Scaffolding required testing a program unit (module)

8.5.2 Integration Testing

The purpose of unit testing is to determine that each independent module is correctly implemented. This gives little chance to determine that the interface between modules is also correct, and for this reason integration testing must be performed. One specific target of integration testing is the interface whether parameters match on both sides as to type, permissible ranges, meaning and utilization [COLL 88].

There are several classical integration strategies that really have little basis in a rational methodology. Top down-integration proceeds down the invocation hierarchy, adding one module

at a time until an entire tree level is integrated; and thus it eliminates the need for drivers. The bottom-up strategy works similarly from the bottom and has no need of stubs. A sandwich strategy runs from top and bottom concurrently, meeting somewhere in the middle. All the three approaches are shown in Fig. 8.30.

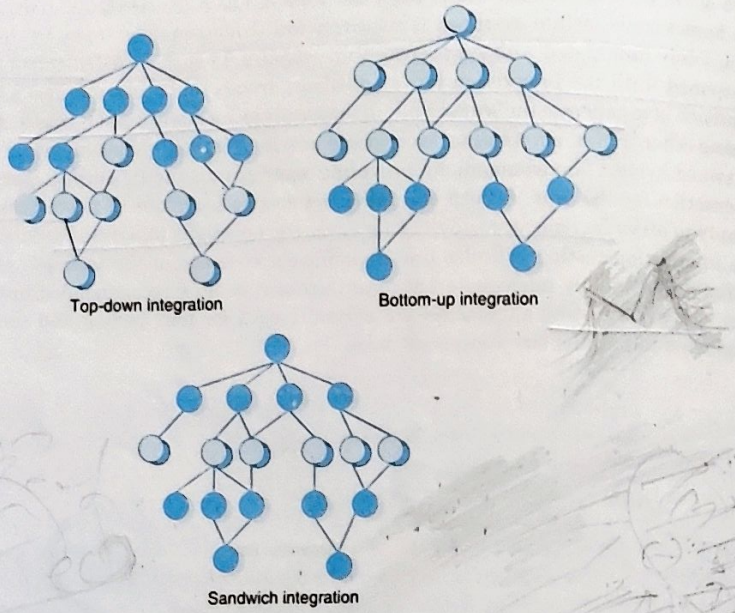

Fig. 8.30: Three different integration approaches

There has been a lot of discussion about which one of the three; viz top-down, bottom up or sandwich integration is best. Most of this concern is driven by the need to mix module testing into integration, because of a lack of test harnesses.

When the lack disappears, the concern is no longer significant. But there are a few principles that should guide integration. First, if we are going to overlap module testing and integration testing, then we will find that not all modules will be ready for integration at the same time.

Integration should follow the lines of first putting together those subsystems that are of great concern. This prioritisation might dictate top-down integration if control and the user interface were the most worrisome or complex part of software. This can occur, for instance, if the customer is anxious to see a running program early in the testing phase. In another situation, the machine interface and performance might be of special interest, and then bottom-up integration would be dictated. With bottom-up integration we stand better chance of experiencing a high degree of concurrency during our integration. It has also been said that top-down integration is an exercise in faith that everything will work out well, whereas bottom-up integration shows that things really do work.

With integration testing, we move slowly away from structural testing and toward functional testing, which treats a module as an impenetrable mechanism for performing a function. As the aggregated modules become larger and larger, we lose our ability to think about path coverage and domains, and must be satisfied with simply determining that the product seems to do what we intended. That is, we treat the product as a black box and verify that specified inputs produce appropriate outputs, without concern for the internal structure of the software. However, even in functional tests we can and should continue to choose test data that represent important or unusual conditions [JONE 90]. In attempting to bridge the gap from small units to a large software system, we may not be able to jump from structural tests to functional tests immediately, while maintaining the principle of adding one unit at a time. One way of testing intermediate levels of the product is to isolate utility, or internal functions defined during the architectural design stage and to use them as the basis for functional testing.

Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly.

8.5.3 System Testing

Of the three levels of testing, the system level is closet to everyday experience. We test many things; a used car before we buy it, an on-line cable network service before we subscribe, and so on. A common pattern in these familiar forms is that we evaluate a product in terms of our expectations; not with respect to a specification or a standard. Consequently, goal is not to find faults, but to demonstrate performance. Because of this we tend to approach system testing from a functional standpoint rather than from a structural one. Since it is so intuitively familiar, system testing in practice tends to be less formal than it might be, and is compounded by the reduced testing interval that usually remains before a delivery deadline [JORG 95].

As we know, software is one component of a large computer based system. Ultimately, software is incorporated with other system components (*e.g.*, new hardware, information), and thus, a series of special tests are to be conducted. Many times, software products are designed to run on a variety of hardware configurations. The software should actually be tested on many different hardware set-ups, although the full range of memory, processor, operating system, and peripheral possibilities may be too large for complete testing. There are many types of specifications, and we should be aware of those as we perform system testing. For instance, there may be a specified level of performance required of the software. This may involve measurement of response time under various loads and operating conditions. It may also require measurement of main and disk memory usage. Software reliability should also be measured during all other tests of the integrated product. If minimum and average up-time behaviour of the product were specified, then these should be met. The time and effort needed to recover from failures should also be recorded and compared with specifications. These specifications should represent customer's wants and needs, and during system testing, we can try to see if the requirements and specifications really do coincide.

Petschenik gives some guidelines for choosing test cases during system testing [PETS 85]. The first is that testing the system's capabilities is more important than testing its components. This implies that failures that are catastrophic should be looked for whereas failures that are merely annoying need not worry us. The idea is that a user can deal with a badly formatted report, but probably cannot deal with unavailability of the report.

Petschenik's second rule is that testing the usual is more important than testing the exotic. This can be accomplished by subjecting the software to the kind of use that is representative of actual use, as described in the operational profile. The user may be able to help with this kind of testing. In fact, software engineers may exhibit blind spots that cause them to notice exotic problems; while they ignore problems that user would spot immediately.

Third, if we are testing after modification of an existing product, we should test old capabilities rather than new ones. The rationale here is that the user is not depending on the new functions of the software, and would not be paralysed if these were not right. But a failure in the old functionality could do just that-paralyse the user's entire operation.

The rationale for Petschenik's testing priorities is that exhaustive testing of functional capabilities is incomplete with the kind of short update cycle imposed on many software products. Just as dependability is more important to the user than the total correctness, the basic and existing functionality also weigh much more heavily than total functionality. Foremost we should avoid disrupting current usage by introducing a new release of the product that would not support current functions.

During system testing, we should evaluate a number of attributes of the software that are vital to the user and are listed in Fig. 8.31. These represent the operational correctness of the product and may be part of the software specifications [JONE 90].

Usable	Is the product convenient, clear, and predictable?
Secure	Is access to sensitive data restricted to those with authorization?
Compatible	Will the product work correctly in conjunction with existing data, software, and procedures?
Dependable	Do adequate safeguards against failure and methods for recovery exist in the product?
Documented	Are manuals complete, correct, and understandable?

Fig. 8.31: Attributes of software to be tested during system testing

8.6 VALIDATION TESTING

It refers to test the software as a complete product. This should be done after unit and integration testing. Here, we want to test the software with the perspective of the customers and may like to ensure that the software meets the expectations of the customers. We may also check all enterics of validation criteria section of the SRS document.

It may not be possible to predict every usage of the software by the customer. We may plane very systematically and seriously ; but behaviour of customers may very drastically. They may try strange inputs, combination of inputs and so on. Some results may be very obvious for developers and may not require any explanation, however, customer may not understand correctly and hence may not appriciate it. In order to avoid or minimise such situations, the involvement of customer is required during validation testing. Alpha, beta and acceptance testing are nothing but the various ways of involving customers during testing.

The alpha and beta testing techniques are used when the software is developed for anonymous customers. Compilers, operating systems, CASE tools etc. come under this category. Here, as such customers are not available, but, potential customers are identified for the purpose of testing. These potential customers are invited in the premises of the company and are requested to use the software. The process is carried out under the guidance of developers and hence only controlled environment is provided. The beta testing has become very popular. Companies are going for beta releases of their products. The potential customers test the software in their respective premises across the globe. The beta testing is conducted in a real environment, without any control of developers. The developers receive the failure reports/ suggestions, and may modify the code, whereever necessary. With this strategy, the company gets the feedback of many potential customers without involving the reputation of the company.

The acceptance testing is popular with customised software. Here, customer is available to check the software as per expectations. These tests may range from adhoc tests to well planned systematic series of tests. The duration of this type of validation testing may be few weeks or months. The identified buys will be fixed and improved software will be delivered to the customer.

The IEEE has developed a standard (IEEE std. 1059—1993) entitled "IEEE guide for software verification and validation" to provide specific guidance about planning and documenting the tasks required by the standard so that the customer may write an effective plan [IEEE 93]. Validation testing improves the quality of software product in terms of functional capabilities and quality attributes. The quality attributes may vary from project to project. These attributes serve the customer's need for a software product that is capable of meeting its objectives with adequate performance with no unexpected side effects. Some of the attributes are accuracy, completeness, consistency, correctness, efficiency, maintainability, portability, reliability, testability, usability etc. Some customers may focus on such attributes during validation testing. A systematic plan may result into good quality product and happy customers.

8.7 THE ART OF DEBUGGING

As discussed earlier, the goal of testing is to identify errors (bugs) in the program. The process of testing generates symptoms, and a program's failure is a clear symptom of the presence of an error. After getting a symptom, we begin to investigate the cause and place of that error. After identification of place, we examine that portion to identify the cause of the problem. This process is called debugging.

Hence, debugging is the activity of locating and correcting errors. It can start once a failure has been detected. Unfortunately, going from the detection of a failure to correcting the error that is responsible, is far from trivial. It is one of the least understood activities in software development and is practiced with the least amount of discipline. It is often approached with much hope and little planning [GHEZ 94].

8.7.1 Debugging Techniques

Most developers have learned through experience several techniques for debugging. Generally these are applied in a trial and error manner. Debugging is not an easy process. This is probably due to human psychology rather than software technology. Error removal requires humility to even admit the possibility of errors in the code we have created and requires an open mind that is willing to see what the software actually does rather than it should do. Commenting on human aspect of debugging, Shneiderman [SHNE 80], states:

"It is one of the most frustrating parts of programming. It has elements of problem solving or brain teasers, coupled with the annoying recognition that we have made a mistake. Heightened anxiety and the unwillingness to accept the possibility of errors, increases the task difficulty. Fortunately, there is a great sigh of relief and a lessening of tension when the bug is ultimately corrected".

However, Pressman [PRES 97] explained few characteristics of bugs that provide some clues.

1. "The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located in other part. Highly coupled program structures may complicate this situation.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptom may actually be caused by nonerrors (e.g. round-off inaccuracies).
4. The symptom may be caused by a human error that is not easily traced.
5. The symptom may be a result of timing problems rather than processing problems.
6. It may be difficult to accurately reproduce input conditions (e.g. a real time application in which input ordering is indeterminate).
7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware with software inextricably.
8. The symptom may be due to causes that are distributed across a number of tasks running on different processors".

A number of popular techniques are given in table 8.8.

In general, none of these techniques should be used without a thorough prior analysis of symptoms the error resulting in a hypothesis concerning the cause of the errors. For instance, if two modules behave properly when operated separately, and a failure occurs when they are integrated, their interface should be checked for consistency.

8.7.2 Debugging Approaches

In heart of debugging process is not the debugging tools, but the underlying approaches used to deduce the cause of the error.

Table 8.8: A comparison of various debugging techniques

Techniques	*Features*	*Advantages*	*Disadvantages*
1. Core dumps	A printout of all registers and relevant memory locations is obtained and studied. All dumps should be well documented and retained for possible use on subsequent problems.	1. The complete contents of memory at a crucial instant of time are obtained for study. 2. Can be cost-effective if used to explore validity of a well-formulated error hypothesis.	1. Require some CPU time, significant I/O time, and much analysis time. 2. Wasteful if used indiscriminately (*i.e.*, at noncrucial instant or without an error theory or debugging plan.) 3. Hexadecimal numbers are cumbersome to interpret and it is difficult to determine the address of source-language variables.
2. Traces	Essentially similar to core dumps, except the printout contains only certain memory and register contents and printing is conditional on some event occurring. Typical conditioning events are entry, exit, or use of (1) a particular subroutine, statement, macro, or database; (2) communication with a terminal, printer, disk, or other peripheral; (3) the value of a variable or expression; and (4) timed actuations (periodic or random) in certain real-time systems. A special problem with trace programs is that the conditions are entered in the source language and any changes require a recompilation.		
3. Print statements	The standard print statement in the language being used is sprinkled throughout the program to output values of key variables.	1. This is a simple way to test whether a particular variable changes, as it should after a particular event. 2. A sequence of print statements portrays the dynamics of variable changes.	1. They are cumbersome to use on large programs. 2. If used indiscriminately, they can produce copious data to be analysed, much of which are superfluous.
4. Debugging Programs	A program which runs concurrently with the program under test and provides commands to (1) examine memory and registers; (2) stop execution of the program at a particular point; (3) search for references to particular constants, variables, registers.	1. Terminal-oriented real-time program. 2. Considerable flexibility to examine dynamics of operation.	1. Generally works on a machine language program. 2. Higher-level-language versions must work with inter-preters. 3. More commonly used on microcomputers than large computers.

One obvious techniques is that of trial and error. The debugger looks at the error symptoms, reaches a snap judgement as to where in the code the underlying error might be, and jumps into and roam around in the program with one or more debugging techniques from the standard pit of tools. Obviously, this is slow and wasteful approach.

The second approach, called backtracking, is to examine the error symptoms to see where they are first noticed. One then backtracks in the program flow of control to a point where the symptoms have disappeared. Generally, this process brackets the location of the error in the program. Subsequent careful study of the bounded segment of the code generally reveals the cause. Another obvious variation of backtracking is forward tracking, where we use, print statements or other means to examine a succession of intermediate results to determine at what point the result first become wrong. If we assume that we know the correct values of the variables at several key points within a program, we can adopt a binary search type of strategy. A set of inputs are injected near the middle of the program and the output is examined. If the output is correct the error is in the first half of the program; and if output is wrong, the error is in the second half of the program. This process is repeated as many times as it is feasible to bracket the erroneous portion of the code for final analysis [SHOO 87].

The third approach could be to insert watch points (output statements) at the appropriate place in the program. We can use a software to insert watch points in a program without modifying the program manually. This eliminates many practical problems involved with adding adhoc-debugging statements to the program manually. For example, in the manual modes, we have to be sure that after finding and fixing the error, we remove any debugging statements we have inserted.

The fourth approach is more general and called induction and deduction [MYER 79]. The inductive approach comes from the formulation of a single working hypothesis based on the data, on the analysis of existing data, and on especially collected data to prove or disprove the working hypothesis. A description of the steps follows; a flowchart of their application sequence is shown in Fig. 8.32.

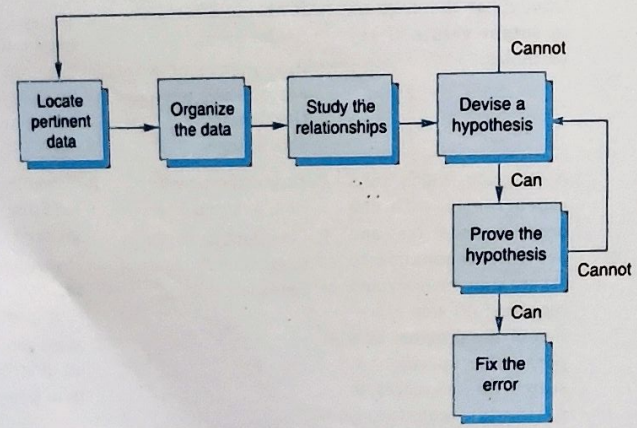

Fig. 8.32: The inductive debugging process [MYER79]

Induction approach

1. **Locate the pertinent data:** A major mistake made when debugging a program is failing to take account of all available data or symptoms about the problem. The first step is the enumeration of all that is known about what the program did correctly, and what it did incorrectly (i.e., the symptoms that led one to believe that an error exists). Additional valuable clues are provided by similar, but different, test cases that do not cause the symptoms to appear.
2. **Organize the data:** Remembering that induction implies that one is progressing from the specific to the general, the second step is the structuring of the pertinent data to allow one to observe patterns. Of particular importance is the search for contradictions (i.e., "the error occurs only when the customer has no outstanding balance in his margin account").
3. **Devise a hypothesis:** The next steps are to study the relationships among the clues and devise, using the patterns that might be visible in the structure of the clues, one or more hypotheses about the cause of the error. If one cannot devise a theory more data is necessary, possibly obtained by devising and executing additional test cases. If multiple theories seem possible the most probable one is selected first.
4. **Prove the hypothesis:** A major mistake at this point, given the pressures under which debugging is usually performed, is skipping this step by jumping to conclusions and attempting to fix the problem. However, it is vital to prove the reasonableness of the hypothesis before proceeding. A failure to do this often results in the fixing of only a symptom of the problem, or only a portion of the problem. The hypothesis is proved by comparing it to the original clues or data, making sure that this hypothesis completely explains the existence of the clues. If it does not either the hypothesis is invalid, or the hypothesis is incomplete, or multiple errors are present.

Deduction approach

The process of deduction begins by enumerating all causes or hypotheses, which seem possible. Then, one by one, particular causes are ruled out until a single one remains for validation. A description of the steps follows; a flowchart of their sequence appears in Fig. 8.33 [MYER79].

1. **Enumerate the possible causes or hypotheses:** The first step is to develop a list of all conceivable causes of the error. They need not be complete explanations; they are merely theories through which one can structure and analyse the available data.
2. **Use the data to eliminate possible causes:** By a careful analysis of the data, particularly by looking for contradictions, one attempts to eliminate all but one of the possible causes. If all are eliminated, additional data are needed (e.g. by devising additional test cases) to devise new theories. If more than one possible cause remains, the most probable cause, the prime hypothesis, is selected first.
3. **Refine the remaining hypothesis:** The possible cause at this point might be correct, but it is unlikely to be specific enough to pinpoint the error. Hence, the next step is to use the available clues to refine the theory (*e.g.* "error in handling the last transaction in the file") to something more specific (*e.g.*, "the last transaction in the buffer is overlaid with the end-of-file indicator").

4. **Prove the remaining hypothesis:** This vital step is identical to step 4 in the induction method.

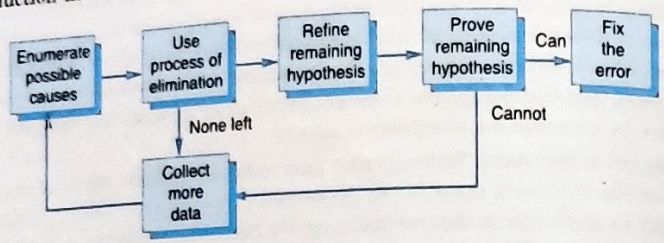

Fig. 8.33: The deductive debugging process [MYER79]

8.7.3 Debugging Tools

Each of the debugging approaches can be supplemented with debugging tools. We can apply wide variety of debugging compilers, dynamic debugging aids, automatic test case generators, memory dumps and cross reference maps. However, tools are not substitute for careful evaluation based on a complete software design document and clear source code.

Compiler is an effective tool for checking and diagnostics. Of course, it checks only syntax errors and particular kind of runtime errors. Compiler should give proper and detailed messages of errors that will be of great help to the debugging process. Compiler can give all such informations in the attribute table, which is printed alongwith the listing. The attributes table contains various level of warnings which have been picked up by the compiler scan and which are noted. Therefore, compilers are coming with error-detection features and there is no excuse for compilers without meaningful error messages.

8.8 TESTING TOOLS

One way to improve the quality and quantity of testing is to make the process as pleasant as possible for the tester. This means that tools should be as concise, powerful and natural as possible.

The two broad categories of software testing tools are: static and dynamic. Most tool functions fall cleanly into one category or the other, but there are some exceptions like symbolic evaluation systems and mutation analysis systems (which actually run interpretively). There are different types of tools available and some are listed below [VICK 84].

(*i*) Static analysers, which examine programs systematically and automatically.

(*ii*) Code inspectors, who inspect programs automatically to make sure they adhere to minimum quality standards.

(*iii*) Standards enforcers, which impose simple rules on the developer.

(*iv*) Coverage analysers, which measure the extent of coverage.

(*v*) Output comparators, used to determine whether the output in a program is appropriate or not.

(*vi*) Test file/data generators, used to set up test inputs.

(*vii*) Test harnesses, used to simplify test operations.

(*viii*) Test archiving systems, used to provide documentation about programs.

8.8.1 Static Testing Tools

Static testing tools are those that perform analysis of the programs without executing them at all.

Static analysers

A static analyser operates from a precomputed database of descriptive information derived from the source text of the program. The idea of a static analyser is to prove allegations, which are claims about the analysed programs that can be demonstrated by systematic examination of all the cases.

There is a close relation to code inspectors, but static analysers are stronger. Typical cases include FACES, DAVE, RXVP, PL/I, checkout compiler, LINT on PWB/Unix and so on [MILL79]. All of these systems are language dependent in the sense that they apply to a particular language, and also often to a particular system. Many applications using static analysers find 0.1–0.2% NCSS (Non-comment source statements) deficiency reports. Some of these are real, and others are spurious in the sense that these are false warnings that are later ignored after interpretation. These are language dependent and require high initial tool investment costs.

Code inspectors

A code inspector does a simple job of enforcing standards in a uniform way for many programs. These can be single statement or multiple statement rules. It is also possible to build code inspector assistance programs that force the inspector to do a good job by linking him to the process through an interactive environment. The AUDIT system is available which imposes some minimum conditions on the program. Code inspection activity is found in some COBOL tools (like AORIS librarian system) and in some parts of tools like RXVP.

Standard enforcers

This tool is like a code inspector, except that the rules are generally simpler. The main distinction is that a full-blown static analyser looks at whole programs, whereas a standard enforcer looks at only single statements.

Since only single statements are treated, the standards enforced tend to be cosmetic ones; even so, they are valuable because they enhance the readability of the programs. It seems well established that the readability of a program is an indirect indicator of its quality.

Other tools

Related tools are used to catch bugs indirectly through listings of the program that highlight the mistakes.

One example is a program generator that is used (mostly in COBOL environments, but possibly in others as well) to produce the proforma parts of each source module. Use of such a

method ensures that all programs look alike, which in itself enhances the readability of the programs.

Another example is using structured programming preprocessors that produce attractive print output. Such augmented program listings typically have automatic indentation, indexing features, and in some cases much more.

8.8.2 Dynamic Testing Tools

Dynamic testing tools seek to support the dynamic testing process. Besides individual tools that accomplish these functions, a few integrated systems group the functions under a single implementation.

A test consists of a single invocation of the test object and all of the execution that ensues until the test object returns control to the point where the invocation was made. Subsidiary modules called by the test object can be real or they can be simulated by testing stubs.

A series of tests is normally required to test one module or to test a set of modules. Test support tools must perform these functions:

(*i*) Input setting: selecting of the test data that the test object reads when called.

(*ii*) Stub processing: handling outputs and selecting inputs when a stub is called.

(*iii*) Results display: providing the tester with the values that the test object produces so that they can be validated.

(*iv*) Test coverage measurement: determining the test effectiveness in terms of the structure of the program.

(*v*) Test planning: helping the tester to plan tests so they are both efficient and also effective at forcing discovery of defects.

Coverage analyzers (execution verifiers)

A coverage analyser or execution verifier (or automated testing analyser, or automated verification system, etc.) is the most common and important tool for testing. It is often relatively simple. One of the common strategies for testing involves declaring a minimum level of coverage, ordinarily expressed as a percentage of the elemental segments that are exercised in aggregate during the testing process. This is called CI coverage, where CI denotes the minimum level of testing coverage so that every logically separate part of the program has been exercised at least once during the set of tests. Unit testing usually requires at least 85–90 of CI coverage.

Most often, CI is measured by planting subroutine calls-called software probes-along each segment of the program. The test object is then executed and some kind of run-time system is used to collect the data, which are then reported to the user in fixed-format reports. Use of coverage analysis can be incorporated into most quality assurance situations, although it is more difficult when there is too little space or when non real-time operation is inequivalent to real-time operation (an artifact of the instrumentation process).

Output comparators

Output comparators are used in dynamic testing-both single-module and multiple-module (system level) varieties to check that predicted and actual outputs are equivalent. This is also done during regression testing. The typical output comparator system objective is to identify

differences between two files; the old and the new output from a program. Typical operating systems for the better minicomputers often have an output comparator, sometimes called a file comparator, built-in.

Test file generators

A test file generator creates a file of information that is used as the program and does so based on commands given by the user and/or from data descriptions (in a COBOL) program's data definition section, for example). Mostly, this is a COBOL-oriented idea in which the file of test data is intended to simulate transaction inputs in a data base management situation. This idea can be adapted to other environments.

Test data generators

The test data generation problem is a difficult one, and at least for the present is one for which no general solution exists. On the other hand, there is a practical need for methods to generate test data that meet a particular objective, normally to execute a previously unexercised segment in the program.

One of the practical difficulties with test data generation is that it requires generation of sets of inequalities that represent the conditions along a chosen path, and the reality is that:

(*i*) Paths are too long and produce very complex formulas.

(*ii*) Formula sets are non-linear.

(*iii*) Many paths are illegal (not logically possible).

Practical approaches to automatic test data generation run into very difficult technical limits.

In practice, the techniques of variational test data generation are often quite effective. The test data are derived (rather than created) from an existing path that comes near the intended segment for which test data are to be found. This is often very easy to do, apparently because programs' structures tend to assist in the process. Automatically generating test data is effectively impossible, but good R and D work is now being done on the problem.

Test harness systems

A test harness system is one that is bound (*i.e.*, link-edited and relocated) around the test object and that (1) permits easy modification and control of test inputs and outputs and (2) provides for online measurement of CI coverage values. Some test harnesses are batch oriented, but the high degree of interaction available in a full-interactive system makes it seem very attractive in practical use. Modern thinking favours interactive test harness systems, which tend to be the focal point for installing many other kinds of analysis support.

Test-archiving systems

The goal of a test-archiving system is to keep track of series of tests and to act as the basis for documenting that the tests have been done and that no defects were found during the process.

A typical design involves establishing both procedures for handling files of test information and procedures for documenting which tests were run when and with what effect. Test archive systems are mainly developed on a system-specific/application-specific basis.

8.8.3 Characteristics of Modern Tools

The characteristics of modern software testing tools differ somewhat from previous systems. Modern tools are modular and highly adaptable to different environments. They also tend to make use of the facilities provided by most operating systems rather than provide those capabilities internally.

Case tools that support DFD, ERD, and structure chart diagramming, offer minimal data dictionary support, are available in the market. Project and Instaplan Management tools like MS Project, a configuration management tool like PVCS, and testing tool like Visual Text and flow graph generator, commercially software design tools like object modelling technique (OMT) and CTT development environment are also available in the market.

REFERENCES

[BEIZZ90] Beizer B., "*Software Testing Techniques*", Van Nostrand Reinhold, New York, 1990.

[BERT94] Bertolino A. and Marre M., "*Automatic Generation of Path Covers Based on the Control Flow Analysis of Computer Programs*", IEEE Trans on Software Engineering, Vol. 20, No. 2, Dec. 1994.

[BIEM94] Bieman J.M. and Off L.M., "*Measuring Functional Cohesion*", IEEE Trans. On Software Engineering, Vol. SE–20, No. 8, pp. 644–657, Aug. 1994.

[COLL88] Collofello J.S., "*Introduction to Software Verification and Validation*", SEI–CM–13–1.1, Software Engineering Institute, Pittsburgh, P.A., USA.

[DAHL72] Dahl O.J., Dijkstra E. W. and Hoare C.A.R., "*Structure Programming*", Academic Press, New York, 1972.

[DEUT82] Deutsch M.S., "*Software Verification and Validation*", Prentice-Hall, Englewood Cliffs, NJ.

[ELME73] Elmendorf W.R., "*Cause–Effect Graphs in Functional Testing*", Poughkeepsie, NY; IBM System Development Division TR–00. 2487, 1973.

[FRIE95] Friedman M.A. and Voas Jeffrey M., "*Software Assessment*", John Wiley and Sons, 1995.

[GHEZ94] Ghezzi C. et al., "*Fundamentals of Software Engineering*", Prentice–Hall, 1994.

[IEEE93] IEEE Std. 1059–1993, "*IEEE Guide for Software Verification and Validation Plans*", IEEE Standards Board, 1993.

[JALO96] Jalote P., "*An Integrated Approach to Software Engineering*", Narosa, Delhi, 1996.

[JONE90] Jones G.W., "*Software Engineering*", John Wiley and Sons, 1990.

[JORG95] Jorgensen P.C.,"*Software Testing: A Craftsman's Approach*", CRC Press, USA, 1995.

[MCCA76] McCabe T.J., "*A Complexity Metric*", IEEE Transactions on Software Engineering, SE–2,4, 308–320, December, 1976.

[MILL72] Mills H.D., "*Mathematical Foundations for Structured Programming*", Federal System Division, IBM Corp, Gaithershurg, MD, FSC 72–6012, 1972.

[MILL77] Miller E. F., "*Tutorial: Program Testing Techniques*", COMPSAC' 77, IEEE Computer Society, 1977.

[MILL79] Miller E.F. and Howden W.E., "*Software Testing and Validation Techniques*", IEEE Computer Society, 1980.

[MYER79] Myers G.J., "*The Art of Software Testing*", New York, Wiley, Interscience, 1979.

[NORM89] Norman Parrington and Marc Roper, "*Understanding Software Testing*", John Wiley and Sons, 1989.

[PATT01] Patton R., "*Software Testing*", Techmedia, 2001.

[PETS85] Petschenik N.H., "*Practical Properties in Software Testing*", IEEE Software 2 (5): 18—23.

[PRES97] Pressman R.S., "*Software Engineering: A Practitioner's Approach*", McGraw Hill, New York, 1997.

[RAJA04] Rajani R. and Oak P., "*Software Testing*", Tata McGraw Hill, 2004.

[SHNE80] Shneiderman B., "*Software Psychology*", Winthrop Publishers, 1980.

[SHOO87] Shooman M.L., "*Software Engineering*", McGraw Hill, NY, 1987.

[SOMM96] Sommerville I., "*Software Engineering*", Addison–Wesley, 1996.

[TAMR03] Tamres L., "*Introducing Software Testing*", Pearson Eduction, 2003.

[VICK84] Vick C.R. and Ramamoorthy C.V., "*Handbook of Software Engineering*", Van Nostrand Reinhold Company, New York.

[WEIS82] Weiser M., "*Program Slicing, Proc*", of 5th Int. Conf. On Software Engineering, March 1981, pp. 439–449.

MULTIPLE CHOICE QUESTIONS

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

8.1. Software testing is:

(*a*) the process of demonstrating that errors are not present.

(*b*) the process of establishing confidence that a program does what it is supposed to do.

(*c*) the process of executing a program to show that it is working as per specifications.

(*d*) the process of executing a program with the intent of finding errors.

8.2. Software mistakes during coding are known as:

(*a*) failures (*b*) defects

(*c*) bugs (*d*) errors.

8.3. Functional testing is known as:

(*a*) structural testing (*b*) behaviour testing

(*c*) regression testing (*d*) none of the above.

8.4. For a function of n variables, boundary value analysis yields:

(*a*) $4n + 3$ test cases (*b*) $4n + 1$ test cases

(*c*) $n + 4$ test cases (*d*) none of the above.

8.5. For a function of two variables, how many test cases will be generated by robustness testing ?

(*a*) 9 (*b*) 13

(*c*) 25 (*d*) 42.

8.6. For a function of n variables robustness testing of boundary value analysis yields:

(*a*) $4n + 1$ (*b*) $4n + 3$

(*c*) $6n + 1$ (*d*) none of the above.

8.7. Regression testing is primarily related to:

(*a*) functional testing (*b*) data flow testing

(*c*) development testing (*d*) maintenance testing.

8.8. A node with indegree = 0 and outdegree $\neq 0$ is called

(*a*) source node (*b*) destination node

(*c*) transfer node (*d*) none of the above.

8.9. A node with indegree ≠ 0 and outdegree = 0 is called
(a) source node
(b) predicate node
(c) destination node
(d) none of the above.

8.10. A decision table has
(a) four portions
(b) three portions
(c) five portions
(d) two portions.

8.11. Beta testing is carried out by
(a) users
(b) developers
(c) testers
(d) all of the above.

8.12. Equivalence class partitioning is related to
(a) structural testing
(b) blackbox testing
(c) mutation testing
(d) all of the above.

8.13. Cause-effect graphing technique is one form of
(a) maintenance testing
(b) structural testing
(c) function testing
(d) regression testing.

8.14. During validation
(a) process is checked
(b) product is checked
(c) developer's performance is evaluated
(d) the customer checks the product.

8.15. Verification is
(a) checking the product with respect to customer's expectations
(b) checking the product with respect to specifications
(c) checking the product with respect to the constraints of the project
(d) all of the above.

8.16. Validation is
(a) checking the product with respect to customer's expectations
(b) checking the product with respect to specification
(c) checking the product with respect to constraints of the project
(d) all of the above.

8.17. Alpha testing is done by
(a) customer
(b) tester
(c) developer
(d) all of the above.

8.18. Site for Alpha testing is
(a) software company
(b) installation place
(c) any where
(d) none of the above.

8.19. Site of Beta testing is
(a) software company
(b) user's site
(c) any where
(d) all of the above.

8.20. Acceptance testing is done by
(a) developers
(b) customers
(c) testers
(d) all of the above.

8.21. One fault may lead to
(a) one failure
(b) no failure
(c) many failures
(d) all of the above.

8.22. Test suite is
(a) set of test cases
(b) set of inputs
(c) set of outputs
(d) none of the above.

8.23. Behavioural specifications are required for:
(a) modelling
(b) verification
(c) validation
(d) none of the above.

8.24. During the development phase, the following testing approach is not adopted
(a) unit testing
(b) bottom up testing
(c) integration testing
(d) acceptance testing.

8.25. Which is not a functional testing technique?
(a) boundary value analysis
(b) decision table
(c) regression testing
(d) none of the above.

8.26. Decision tables are useful for describing situations in which:
(a) an action is taken under varying sets of conditions
(b) number of combinations of actions are taken under varying sets of conditions
(c) no action is taken under varying sets of conditions
(d) none of the above.

8.27. One weakness of boundary value analysis and equivalence partitioning is
(a) they are not effective
(b) they do not explore combinations of input circumstances
(c) they explore combinations of input circumstances
(d) none of the above.

8.28. In cause effect graphing technique, cause and effect are related to
(a) input and output
(b) output and input
(c) destination and source
(d) none of the above.

8.29. DD Path graph is called as
(a) design to design path graph
(b) defect to defect path graph
(c) destination to destination path graph
(d) decision to decision path graph.

8.30. An independent path is
(a) any path through the DD path graph that introduces at least one new set of processing statements or new conditions
(b) any path through the DD Path graph that introduces at most one new set of processing statements or new conditions
(c) any path through the DD Path graph that introduces one and only one new set of processing statements or new conditions.
(d) none of the above.

8.31. Cyclomatic complexity is developed by
(a) B.W. Boehm
(b) T.J. McCabe
(c) B.W. Littlewood
(d) Victor Basili.

8.32. Cyclomatic complexity is denoted by
(a) $V(G) = e - n + 2P$
(b) $V(G) = \Pi + 1$
(c) $V(G) =$ number of regions of the graph
(d) all of the above.

8.33. The equation $V(G) = \Pi + 1$ of cyclomatic complexity is applicable only if every predicate node has
(*a*) two outgoing edges (*b*) three or more outgoing edges.
(*c*) no outgoing edges (*d*) none of the above.

8.34. The size of the graph matrix is
(*a*) number of edges in the flow graph
(*b*) number of nodes in the flow graph
(*c*) number of paths in the flow graph
(*d*) number of independent paths in the flow graph.

8.35. Every node is represented by
(*a*) one row and one column in graph matrix (*b*) two rows and two columns in graph matrix
(*c*) one row and two columns in graph matrix (*d*) none of the above.

8.36. Cyclomatic complexity is equal to
(*a*) number of independent paths (*b*) number of paths
(*c*) number of edges (*d*) none of the above.

8.37. Data flow testing is related to
(*a*) data flow diagrams (*b*) E-R diagrams
(*c*) data dictionaries (*d*) none of the above.

8.38. In data flow testing, objective is to find
(*a*) all dc-paths that are not du-paths (*b*) all du-paths
(*c*) all du-paths that are not dc-paths (*d*) all dc-paths.

8.39. Mutation testing is related to
(*a*) fault seeding (*b*) functional testing
(*c*) fault checking (*d*) none of the above.

8.40. The overhead code required to be written for unit testing is called
(*a*) drivers (*b*) stubs
(*c*) scaffolding (*d*) none of the above.

8.41. Which is not a debugging technique ?
(*a*) core dumps (*b*) traces
(*c*) print statements (*d*) regression testing.

8.42. A break in the working of a system is called
(*a*) defect (*b*) failure
(*c*) fault (*d*) error.

8.43. Alpha and Beta testing techniques are related to
(*a*) system testing (*b*) unit testing
(*c*) acceptance testing (*d*) integration testing.

8.44. Which one is not the verification activity ?
(*a*) reviews (*b*) path testing
(*c*) walkthrough (*d*) acceptance testing.

8.45. Testing the software is basically
(*a*) verification (*b*) validation
(*c*) verification and validation (*d*) none of the above.

8.46. Integration testing techniques are
(*a*) topdown (*b*) bottom up
(*c*) sandwich (*d*) all of the above.

8.47. Functionality of a software is tested by
(*a*) white box testing (*b*) black box testing
(*c*) regression testing (*d*) none of the above.

8.48. Top down approach is used for
(*a*) development (*b*) identification of faults
(*c*) validation (*d*) functional Testing.

8.49. Thread testing is used for testing
(*a*) real time systems (*b*) object oriented systems
(*c*) event driven systems (*d*) all of the above.

8.50. Testing of software with actual data and in the actual environment is called
(*a*) alpha testing (*b*) beta testing
(*c*) regression testing (*d*) none of the above.

EXERCISE

8.1. What is software testing? Discuss the role of software testing during software life cycle and why is it so difficult?

8.2. Why should we test? Who should do the testing?

8.3. What should we test? Comment on this statement. Illustrate the importance of testing.

8.4. Defined the following terms:
(*i*) fault (*ii*) failure
(*iii*) bug (*iv*) mistake

8.5. What is the difference between
(*i*) Alpha testing and beta testing
(*ii*) Development and regression testing
(*iii*) Functional and structural testing.

8.6. Discuss the limitations of testing. Why do we say that complete testing is impossible?

8.7. Briefly discuss the following
(*i*) Test case design, Test and Test suite
(*ii*) Verification and Validation
(*iii*) Alpha, beta and acceptance testing.

8.8. Will exhaustive testing (even if possible for very small programs) guarantee that the program is 100% correct?

8.9. Why does software fail after it has passed from acceptance testing? Explain.

8.10. What are various kinds of functional testing? Describe any one in detail.

8.11. What is a software failure? Explain necessary and sufficient conditions for software failure. Mere presence of faults means software failure. Is it true? If not, explain through an example, a situation in which a failure will definitely occur.

8.12. Explain the boundary value analysis testing technique with the help of an example.

8.13. Consider the program for the determination of next date in a calendar. Its input is a triple of day, month and year with the following range

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

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

$1900 \leq \text{year} \leq 2025$

The possible outputs would be Next date or invalid input date. Design boundary value, robust and worst test cases for this programs.

8.14. Discuss the difference between worst test case and adhoc test case performance evaluation by means of testing. How can we be sure that the real worst case has actually been observed?

8.15. Describe the equivalence class testing method. Compare this with boundary value analysis technique.

8.16. Consider a program given below for the selection of the largest of numbers.

```
main()
  {
   float A,B,C;
   printf("Enter three values\n");
   scanf("%f%f%f", &A,&B,&c);
   printf("\n Largest value is");
   if (A>B)
        {
             if(A>C)
                  printf("%f\n",A);
             else
                  printf("%f\n",c);
        }
        else
        {
             if(C>B)
                  printf("%f\n",C);
             else
                  printf("%f\n",B);
        }
  }
```

(*i*) Design the set of test cases using boundary value analysis technique and equivalence class testing technique.

(*ii*) Select a set of test cases that will provide 100% statement coverage.

(*iii*) Develop a decision table for this program.

8.17. Consider a small program and show, why is it practically impossible to do exhaustive testing?

8.18. Explain the usefulness of decision table during testing. Is it really effective? Justify your answer.

8.19. Draw the cause effect graph of the program given in exercise 8.16.

8.20. Discuss cause effect graphing technique with an example.

8.21. Determine the boundary value test cases for the extended triangle problem that also considers right angle triangles.

8.22. Why does software testing need extensive planning? Explain.

8.23. What is meant by test case design? Discuss its objectives and indicate the steps involved in test case design.

8.24. Let us consider an example of grading the students in an academic institution. The grading is done according to the following rules:

Marks obtained	*Grade*
80–100	Distinction
60–79	First division
50–59	Second division
40–49	Third division
0–39	Fail

Generate test cases using equivalence class testing technique.

8.25. Consider a program to determine whether a number is 'odd' or 'even' and print the message

NUMBER IS EVEN

Or

NUMBER IS ODD

The number may be any valid integer.

Design boundary value and equivalence class test cases.

8.26. Admission to a professional course is subject to the following conditions:

(*a*) Marks in Mathematics >= 60

(*b*) Marks in Physics >= 50

(*c*) Marks in Chemistry >= 40

(*d*) Total in all three subjects >= 200

Or

Total in Mathematics and Physics >= 150.

If aggregate marks of an eligible candidate are more than 225, he/she will be eligible for honours course, otherwise he/she will be eligible for pass course. The program reads the marks in the three subjects and generates the following outputs:

(*a*) Not eligible.

(*b*) Eligible to pass course.

(*c*) Eligible to honours course.

Design test cases using decision table testing technique.

8.27. Draw the flow graph for program of largest of three numbers as shown in exercise 8.16. Find out all independent paths that will guarantee that all statements in the program have been tested.

8.28. Explain the significance of independent paths. Is it necessary to look for a tool for flow graph generation, if program size increases beyond 100 source lines?

8.29. Discuss the structural testing. How is it different from functional testing?

8.30. What do you understand by structural testing? Illustrate important structural testing techniques.

8.31. Discuss the importance of path testing during structural testing.

8.32. What is cyclomatic complexity? Explain with the help of an example.

8.33. Is it reasonable to define "thresholds" for software modules? For example, is a module acceptable if its $V(G) \leq 10$? Justify your answer.

8.34. Explain data flow testing. Consider an example and show all "du" paths. Also identify those "du" paths that are not "dc" paths.

8.35. Discuss the various steps of data flow testing.

8.36. If we perturb a value, changing the current value of 100 by 1000, what is the effect of this change? What precautions are required while designing the test cases?

8.37. What is the difference between white and black box testing? Is determining test cases easier in black or white box testing? Is it correct to claim that if white box testing is done properly, it will achieve close to 100% path coverage?

8.38. What are the objectives of testing? Why is the psychology of a testing person important?

8.39. Why does software fail after it has passed all testing phases? Remember, software, unlike hardware does not wear out with time.

8.40. What is the purpose of integration testing? How is it done?

8.41. Differentiate between integration testing and system testing.

8.42. Is unit testing possible or even desirable in all circumstances? Provide examples to Justify your answer?

8.43. Petschenik suggested that a different team than the one that does integration testing should carry out system testing. What are some good reasons for this?

8.44. Test a program of your choice, and uncover several program errors. Localise the main route of these errors, and explain how you found the courses. Did you use the techniques of Table 8.8? Explain why or why not.

8.45. How can design attributes facilitate debugging?

8.46. List some of the problems that could result from adding debugging statements to code. Discuss possible solutions to these problems.

8.47. What are various debugging approaches? Discuss them with the help of examples.

8.48. Researchers and practitioners have proposed several mixed testing strategies intended to combine advantages of the various techniques discussed in this chapter. Propose your own combination, perhaps also using some kind of random testing at selected points [GHEE 94].

8.49. Design a test set for a spell checker. Then run it on a word processor having a spell checker, and report on possible inadequacies with respect to your requirements.

8.50. 4 GLs represent a major step forward in the development of automatic program generators. Explain the major advantages and disadvantages in the use of 4 GLs. What are the cost impacts of applications of testing and how do you justify expenditures for these activities.