9

Software Testing

Software testing is a very important, challenging and essential activity. It starts along with the design of SRS document and ends at the delivery of software product to the customer. It consumes significant effort and maximum time of software development life cycle (without including the maintenance phase). The significant effort may be from one third to one half of the total effort expended till the delivery of the software product. We cannot imagine to deliver a software product without adequate testing. The existing testing techniques help us to do adequate testing. However, adequate testing has different meaning to different software testers.

9.1 What is Software Testing?

The common perception about testing is to execute a program with given input(s) and note the observed output(s). The observed output(s) is/are matched with expected output(s) of the program. If it matches, the program is correct and if it does not match, the program is incorrect for the given input(s). There are many definitions of testing and some of them are given as follows:

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

The purpose of testing as per the above definitions is to show the correctness of the program. We may select many inputs and execute the program and also demonstrate its correctness without touching critical and complex portions of the program. During testing, our objective should be to find faults and find them as early as possible. Hence, a more appropriate definition of testing is given by Myers (2004) as:

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

This definition motivates us to select those inputs which have higher probability of finding faults, although the definition is also not complete because it focuses only on the execution of the program. Nowadays, attention is also given to the activities like reviewing the documents and programs. Reviewing the documents (like SRS and SDD) helps us to find a good number of faults in the early phases of software development. Hence, testing is divided into two parts: verification and validation. Therefore, the most appropriate definition of software testing is:

> *Software testing is the process of verifying the outcomes of every phase of software development and validating the program by executing it with the intention of finding faults.*

In the initial days of programming, testing was primarily validation oriented but nowadays both are equally important and carried out in most of the software organizations.

9.1.1 Verification

Software verification is also known as static testing where testing activities are carried out without executing the program. It may include inspections, walkthroughs and reviews where documents and programs are reviewed with the purpose of finding faults. Hence, verification is the process of reviewing (rechecking) documents and program with the intention of finding faults. As per the IEEE (2001):

> *The verification is the process of evaluating the system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.*

9.1.2 Validation

Software validation is also known as dynamic testing where the program is executed with given input(s). Hence, validation involves execution of the program and usually exposes symptoms of errors. As per the IEEE (2001):

> *The validation 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.*

Validation activities are carried out with the execution of the program. We experience failures and reasons of such failures are identified.

Software testing includes both verification and validation. Both are important and complementary to each other. Effective verification activities find a good number of faults in the early phases of software development. Removing such faults will definitely provide better foundations for the implementation/construction phase. Validation activities are possible only after the implementation of module/program, but verification activities are possible in every phase of software development.

9.2 Software Verification Techniques

Software verification techniques are applicable in every phase of software development life cycle. The purpose of such techniques is to review the outcomes of every phase of software development life cycle, i.e. from requirements and analysis phase to testing phase. Most of

the software professionals are convinced about the importance and usefulness of verification techniques. They also believe that a good quality software product can be produced only after the effective implementation of verification techniques. Many verification techniques are commonly used in practice and some of them are discussed in the following subsections.

9.2.1 Peer Reviews

This is the simplest, informal and oldest verification technique. It is applicable to every document produced at any stage of software development. Programs are also reviewed using this technique without executing them. We give documents/programs to some one else and ask them to review with an objective of finding faults. Some shortcomings or faults may be identified. This technique may give very good results if the reviewer has domain knowledge, technical expertise, programming skills and involvement in the reviewing work. A report may be prepared about the faults in the documents and programs. Sometimes, faults are also reported verbally during discussion. Reported faults are examined and appropriate corrections are made in the documents and programs. Every reviewing activity improves the quality of the documents and programs. Every reviewing activity improves the quality of the documents and programs without spending significant resources. This is very effective and easily applicable to small-size documents and programs. However, effectiveness reduces, as the size of the documents and programs increases.

9.2.2 Walkthroughs

Walkthrough is a group activity for reviewing the documents and programs. It is a more formal and systematic technique than per reviews. A group of two to seven persons is constituted for the purpose of reviewing. The author of the document presents the document to the group during walkthroughs conducted in a conference room. The author may use any display mechanism (like whiteboard, projector) for presentation. Participants are not expected to prepare anything prior to such a meeting. Only the presenter who happens to be the author of the document prepares for the walkthrough. Documents and/or programs are distributed to every participant and the author presents the contents to all to make them understand. All participants are free to ask questions and write their observations on any display mechanism in such a way that everyone can see it. The observations are discussed thoroughly amongst the participants and changes are suggested accordingly. The author prepares a detailed report as per suggested changes. The report is studied, changes are approved and documents/programs are modified.

The limitations of such a technique are the non-preparation of participants prior to meeting and presentation of documents by their author(s). The author(s) may unnecessarily highlight some areas and hide a few critical areas. Participants may also not be able to ask critical and penetrating questions. Walkthroughs may create awareness amongst participants and may also find a few important faults. This technique is more expensive but systematic than peer reviews and applicable to any size of software project.

9.2.3 Inspections

This technique is more formal, systematic and effective. There are many names for this technique like formal reviews, formal technical reviews and inspections. The most popular

name is inspection and is different from peer reviews and walkthroughs. A group of three to six persons is constituted. The author of the document is not the presenter. An independent presenter is appointed for this specific purpose who prepares and understands the document. This preparation provides a second eye on the document(s) prior to meeting. The documents(s) is/are distributed to every participant in advance to give sufficient time for preparation. Rules of the meeting are prepared and circulated to all participants in advance to make them familiar with the framework of the meeting. The group is headed by an impartial moderator who should be a senior person to conduct the meetings smoothly. A recorder is also engaged to record the proceedings of the meetings.

The presenter presents the document(s) in the meeting and makes everyone comfortable with the document(s). Every participant is encouraged to participate in the deliberations as per modified rules. Everyone gets time to express his/her views, opinions about quality of document(s), potential faults and critical areas. The moderator chairs the meeting, enforces the discipline as per prescribed rules and respects views of every participant. Important observations are displayed in a way that is visible to every participant. The idea is to consider everyone's views and not to criticize them. Many times, a checklist is used to review the document. After the meeting, the moderator along with the recorder prepares a report and circulates it to all participants. Participants may discuss the report with the moderator and suggest changes. A final report is prepared after incorporating changes. The changes should be approved by the moderator. Most of the organizations use inspections as a verification technique and outcomes are very good. Many faults are generally identified in the early phases of software development. This is a popular, useful and systematic technique and is effective for every type of document and program. The comparison of all the three techniques is given in Table 9.1.

Table 9.1 Verification techniques comparison

	1	2	3
Technique	Peer reviews	Walkthrough	Inspections
Presenter	No one	Author	Someone other than the author
Number of participants	1 or 2	2 to 7 participants	3 to 6 participants
Prior preparation	Not required	Only the presenter is required to be prepared	All participants are required to be prepared
Applicability	Small-size software projects	Any size software projects	Any size software projects
Report	Optional	Compulsory	Compulsory
Advantages	Inexpensive and always finds some faults	Makes people aware about the project	Useful and finds many faults
Disadvantages	Dependent on the ability of reviewer	May find few faults	Skilled participants are needed and expensive

Verification is always more useful than validation due to its applicability in early phases of software development. It finds those faults which may not be detected by any validation technique. Verification techniques are becoming popular and more attention is given to such techniques from the beginning of the software development.

9.3 Checklist: A Popular Verification Tool

A checklist is normally used which consists of a list of important information contents that a deliverable must contain. A checklist may discipline the reviewing process and may identify duplicate information, missing information, and unclear and wrong information. Every document may have a different checklist which may be based on important, critical and difficult areas of the document. A checklist may be applicable in all verification techniques but it is more effective during inspections.

9.3.1 SRS Document Checklist

The SRS document is given in Chapter 3 (refer to Section 3.7) which is designed as per the IEEE standard 830–1998. Characteristics of good requirements are also given in Section 3.6 of Chapter 3 which may include the characteristics such as correct, unambiguous, complete, verifiable, modifiable, clear, feasible, necessary and understandable. Therefore, the SRS document checklist should address the above-mentioned characteristics. The SRS document is the source of many potential faults. It should be reviewed very seriously and thoroughly. The effective reviewing process improves the quality of the SRS document and minimizes the occurrence of many future problems. A properly designed checklist may help to achieve the objective of developing good quality software. A checklist is given in Table 9.2 which may be modified as per the requirement of the SRS document.

Table 9.2 Checklist for verifying SRS document

Part I

Reviewer name(s)	Organization	Review date	Project title

Part II

S. No.	Description	Yes/No/NA	Comments
	Document overview		
1.	Are you satisfied with defined scope?		
2.	Are the objectives of the project clearly stated?		
3.	Has IEEE std-830-1998 been followed?		
4.	Is SRS document approved by the customer?		
5.	Is the layout of the screens well designed?		
6.	Are definitions, acronyms and abbreviations defined?		
7.	Do you suggest changes in the overall description?		
8.	Are user interfaces, hardware interfaces, software interfaces and communication interfaces clearly described?		
9.	Are major product functions stated?		
10.	Are you satisfied with the list of constraints?		
11.	Do you want to add any field in any form?		

(Contd.)

Table 9.2 Checklist for a good quality software (*Contd.*)

S. No.	Description	Yes/No/NA	Comments
12.	Do you want to delete any field in any form?		
13.	Do you want to add/delete any validity check?		
14.	Is readability of the document satisfactory?		
15.	Are non-functional requirements specified?		
	Actors, use cases and use case description		
16.	Are all the actors identified?		
17.	Are all the use cases determined?		
18.	Does the name of the use case represent the function it is required to perform?		
19.	Are all the actors included in the use case description of each use case?		
20.	Are you ready to accept identified use cases and their descriptions?		
21.	Does the use case description include precondition?		
22.	Does the use case description include postcondition?		
23.	Is the basic flow in the use case complete?		
24.	Are all the alternative flows of the use case stated?		
25.	Are related use cases stated?		
	Characteristics of requirements		
26.	Are all requirements feasible?		
27.	Are requirements non-feasible due to technical problems?		
28.	Are requirements non-feasible due to inadequate resources?		
29.	Are a few requirements difficult to implement?		
30.	Are all requirements conveying only one meaning?		
31.	Are there conflicts amongst requirements?		
32.	Are all functional and non-functional requirement defined?		
33.	Are all use cases clearly understandable?		
34.	Are all forms available with adequate validity checks?		
35.	Are all requirements specified at a consistent level of detail?		
36.	Is consistency maintained throughout the document?		
37.	Are there any conflicts amongst requirements?		
38.	Are all defined requirements verifiable?		
39.	Are all stated requirements understandable?		
40.	Are redundant requirements identified?		
41.	Do you want to add a requirement?		
42.	Are all stated requirements written in a simple language?		

(*Contd.*)

Table 9.2 Checklist for a good quality software (*Contd.*)

S. No.	Description	Yes/No/NA	Comments
43.	Are there any non-verifiable words in any requirement?		
44.	Are all assumptions and dependencies defined?		
45.	Are the requirements consistent with other documents of the project?		
	General		
46.	Are you satisfied with the depth of the document?		
47.	Is logical database requirements specified?		
48.	Are design constraints described?		
49.	Is site adaptation requirement clearly defined?		
50.	Is product perspective written with adequate details?		

9.3.2 Object-Oriented Analysis Checklist

The checklist for OOA is given in Table 9.3 which may be used to verify the documents produced after OOA phase.

Table 9.3 Checklist for verifying OOA

Part I

Reviewer name(s)	Organization	Review date	Project title

Part II

S. No.	Description	Yes/No/NA	Comments
1.	Are all the classes—entity, interface and control—identified correctly?		
2.	Is any class missing per use case?		
3.	Is relationship between classes identified correctly?		
4.	Are entity classes persistent even after the end of use case?		
5.	Do control classes model flow of control in the system?		
6.	Does the name of the class convey the function it is intended to perform?		
7.	In case of association relationship, is multiplicity between classes identified correctly?		
8.	Are role names in case of association relationship between classes identified correctly?		
9.	Are all attributes identified correctly?		
10.	Does the name of the attribute convey the function it is intended to perform?		

(Contd.)

Table 9.3 Checklist for OOA (*Contd.*)

S. No.	Description	Yes/No/NA	Comments
11.	Is UML syntax of each class correct?		
12.	Are the control classes necessary?		
13.	Is naming convention followed by the attributes?		
14.	Are classes identified for each use case?		
15.	Are all attributes defined clearly?		

9.3.3 Object-Oriented Design Checklist

The OOD checklist is given in Table 9.4 which may be used to verify the documents produced after the OOD phase.

Table 9.4 Checklist for verifying OOD

Part I

Reviewer name(s)	Organization	Review date	Project title

Part II

S. No.	Description	Yes/No/NA	Comments
1.	Are all the objects in the interaction diagram the identified correctly?		
2.	Are all the classes in the interaction diagram identified correctly?		
3.	Are interaction diagrams created for each use case scenario?		
4.	Are all messages shown at right places in the interaction diagram?		
5.	Are all messages passing the parameters correctly?		
	Sequence diagram		
6.	Do the sequence diagrams have an initiating actor?		
7.	Does an actor send message to the system in the sequence diagram?		
8.	Does the sequence diagram depict the order in which messages are sent?		
9.	Are the objects in the sequence diagram destroyed when not required?		
10.	Is the focus of control for each object identified correctly?		
11.	Are all the return messages identified correctly?		
12.	Are all the parameters of the messages identified correctly?		
13.	Are all the parameter types of the message parameters identified correctly?		

(*Contd.*)

Table 9.4 Checklist for OOD (*Contd.*)

S. No.	Description	Yes/No/NA	Comments
14.	Do the messages follow the naming convention?		
15.	Is the order of the messages identified correctly?		
16.	Are all swimlanes defined properly?		
17.	Are all states identified correctly in statechart diagrams?		
18.	Are all guard conditions used at proper places?		
19.	Are activity diagrams used to model the working of all processes?		
20.	Are all stated notations used in all activity diagrams and statechart diagrams?		

9.4 Functional Testing

Functional testing techniques are validation techniques because execution of the program is essential for the purpose of testing. Inputs are given to the program during execution and the observed output is compared with the expected output. If the observed output is different than the expected output, the situation is treated as a failure of the program. Functional testing techniques may design those test cases which have higher chances of making the program fail. These test cases are designed as per functionality and internal structure of the program is not at all considered. The program is treated as a black box and only functionality of the program is considered. Functional testing is also called black box testing because internal structure of the program is completely ignored and is shown in Figure 9.1.

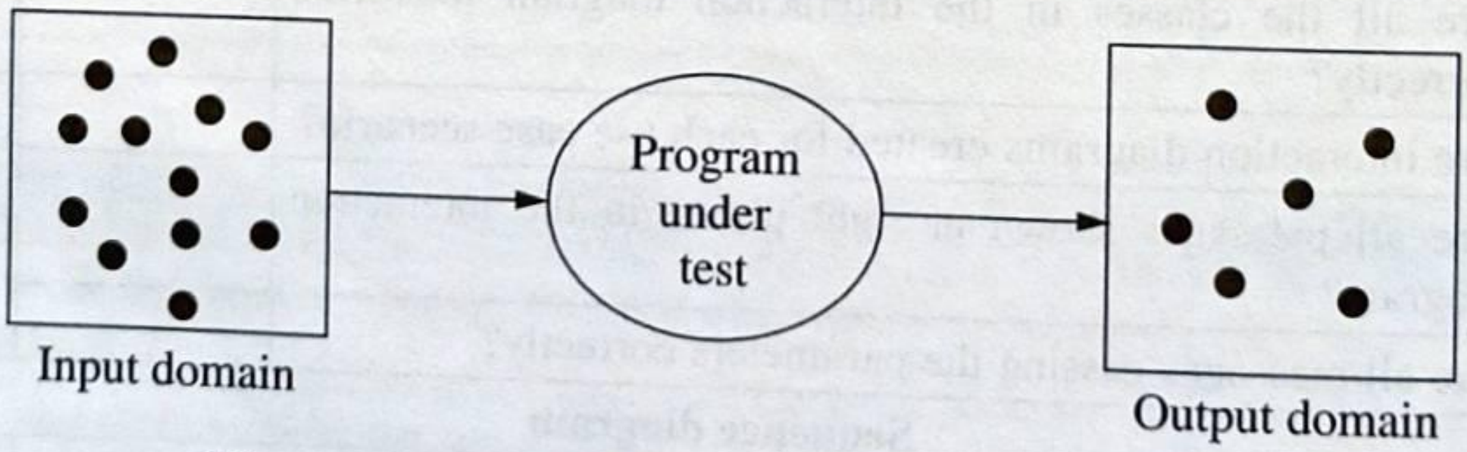

Figure 9.1 Functional (black box) testing.

Every dot of the input domain represents input(s) and every dot of the output domain represents output(s). Every dot of the input domain has a corresponding dot in the output domain domain. We consider valid and invalid inputs for the purpose of program execution and note the behaviour in terms of observed outputs. Functional testing techniques provide ways to design effective test cases to find errors in the program.

9.4.1 Boundary Value Analysis

The boundary value analysis technique focuses upon on or close to boundary values of input domain. We feel that the inputs on or close to boundary values have more chances to make the program fail after execution. Hence, test cases with input values on or close to boundary should

be designed. We consider a program 'square root' which takes a as an input value and prints the square root of a. The range of input value a is from 100 to 1000. We may execute the program for all input values from 100 to 1000 and observe the outputs of the program 900 times to see the output for every possible valid input. We may not like to execute the program for every possible valid input due to time and resource constraints. The boundary value analysis technique helps us to reduce the number of test cases by focusing only upon on or close to boundary values. On or close to boundary values cover the following:

- Minimum value
- Just above the minimum value
- Maximum value
- Just below the maximum value
- Nominal (average) value

These values are represented in Figure 9.2 for the program 'square root' with input value ranging from 100 to 1000.

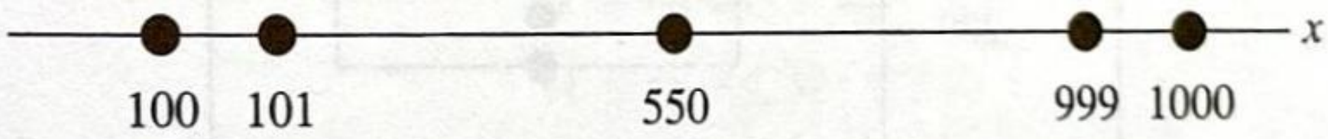

Figure 9.2 Input to the program of 'square root'.

The technique selects only five values instead of 900 values to test the program. These values of a are 100, 101, 550, 999, 1000 which cover the boundary portions of the input value except one nominal value (say 550). The nominal value represents those values which are neither on the boundary nor close to boundary. The number of test cases designed by this technique is $4n + 1$, where n is the number of input values. The test cases generated for the 'square root' program are shown in Table 9.5.

Table 9.5 Test cases of 'square root' program test cases

Test case	Input a	Expected output
1	100	10
2	101	10.05
3	550	23.45
4	999	31.61
5	1000	31.62

We consider a program 'subtraction' with two input values a and b and the output is the subtraction of a and b. The ranges of input values are shown as:

$$100 \leq a \leq 1000$$

$$200 \leq b \leq 1200$$

The selected values for a and b are given as:

$$a = (100, 101, 550, 999, 1000)$$

$$b = (200, 201, 700, 1199, 1200)$$

Both inputs (a and b) are required for the execution of the program. The input domain is shown in Figure 9.3 where any point within the rectangle is a legitimate input of the program.

The test cases are generated on the basis of 'single fault' assumption theory of reliability. This theory assumes that failures are rarely the result of simultaneous occurrence of two (or more) faults. Generally one fault is responsible for a failure. The implication of this theory is that we select one input for any of the defined states (minimum, just above minimum, just below maximum, maximum, nominal) and other input(s) as nominal values.

The test cases are $4n + 1$ (i.e. $8 + 1 = 9$) which are given in Table 9.6. The input values are also shown graphically in Figure 9.3.

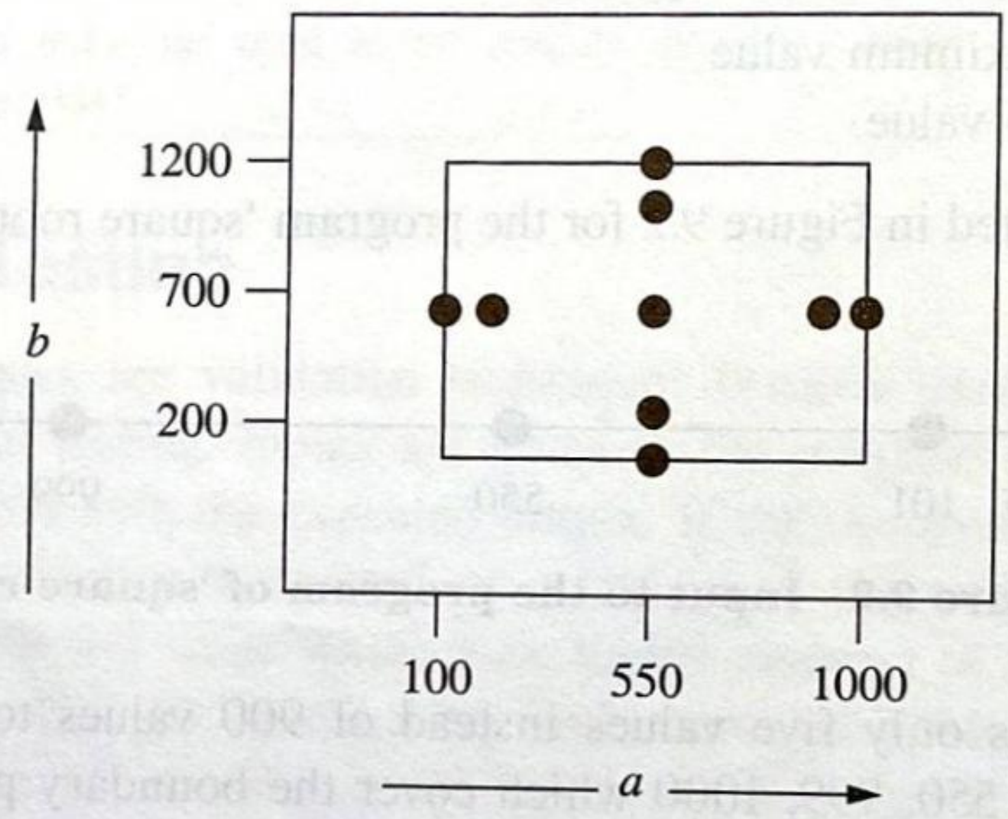

Figure 9.3 Graphical representation of inputs.

Table 9.6 Test cases for 'subtraction' program

Test case	a	b	Expected output
1	100	700	–600
2	101	700	–599
3	550	700	–150
4	999	700	299
5	1000	700	300
6	550	200	350
7	550	201	349
8	550	1199	–649
9	550	1200	–650

EXAMPLE 9.1 Consider a program to multiply and divide two numbers. The inputs may be two valid integers (say a and b) in the range of [0, 100]. Generate boundary value analysis test cases.

Solution Boundary value analysis test cases are given in Table 9.7.

Table 9.7 Boundary value analysis test cases for Example 9.1

Test case	a	b	Expected output	
1	0	50	0	0
2	1	50	50	0
3	50	50	2500	1
4	99	50	4950	1
5	100	50	5000	2
6	50	0	0	Divide by zero error
7	50	1	50	50
8	50	99	4950	0
9	50	100	5000	0

EXAMPLE 9.2 Consider a program which takes a date as an input and checks whether it is a valid date or not. Its input is a triple of day, month and year with the values in the following ranges:

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

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

$$2000 \leq \text{year} \leq 2070$$

Generate boundary value analysis test cases.

Solution Boundary value analysis test cases are given in Table 9.8.

Table 9.8 Boundary value test cases for program determining validity of date

Test case	Month	Day	Year	Expected output
1	1	15	2035	Valid date (1/15/2035)
2	2	15	2035	Valid date (2/15/2035)
3	6	15	2035	Valid date (6/15/2035)
4	11	15	2035	Valid date (11/15/2035)
5	12	15	2035	Valid date (12/15/2035)
6	6	1	2035	Valid date (6/1/2035)
7	6	2	2035	Valid date (6/2/2035)
8	6	30	2035	Valid date (6/30/2035)
9	6	31	2035	Invalid date
10	6	15	2000	Valid date (6/15/2000)
11	6	15	2001	Valid date (6/15/2001)
12	6	15	2069	Valid date (6/15/2069)
13	6	15	2070	Valid date (6/15/2070)

There are three extensions of boundary value analysis technique and are given below:

(i) **Robustness testing**

Two additional states 'just below minimum' and 'just above maximum' are added to see the behaviour of the program with invalid input values. Invalid inputs are equally important and may make the program fail when such inputs are given.

There are seven states in robustness testing and are given as:

1. Just below minimum (minimum$^-$)
2. Minimum
3. Just above minimum (minimum$^+$)
4. Nominal (average value)
5. Just below maximum (maximum$^-$)
6. Maximum
7. Just above maximum (maximum$^+$)

The total number of test cases generated for robustness testing is $6n + 1$, where n is the number of input values. Test cases for 'subtraction' program using robustness testing are 13 as given in Table 9.9.

Table 9.9 Robustness test cases for 'subtraction' program

Test case	*a*	*b*	Expected output
1	99	700	Invalid input
2	100	700	–600
3	101	700	–599
4	550	700	–150
5	999	700	299
6	1000	700	300
7	1001	700	Invalid input
8	550	199	Invalid input
9	550	200	350
10	550	201	349
11	550	1199	–649
12	550	1200	–650
13	550	1201	Invalid input

The input domain has four test cases which are outside; the other test cases lie in the legitimate input domain as shown in Figure 9.4.

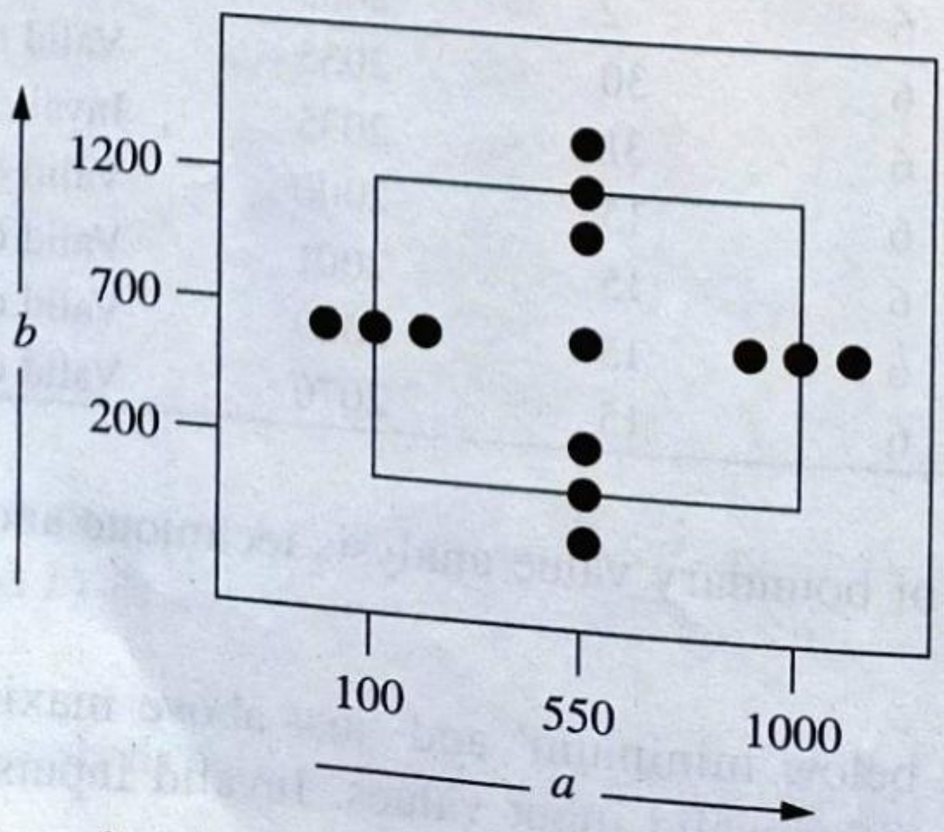

Figure 9.4 Input domain of 'subtraction' program for robustness test cases.

(ii) Worst case testing

This is another form of boundary value analysis where the single fault assumption theory of reliability is rejected. Hence, a failure may also be the result of the occurrence of more than one fault simultaneously. The result of this rejection is that one, two or all input values may have one of the following states:

- Minimum
- Minimum$^+$
- Nominal (average value)
- Maximum$^-$
- Maximum

In the worst case testing, the restriction of one input value at the above-mentioned states and other input values nominal is removed. The implication is that the number of test cases will increase from $4n + 1$ test cases to 5^n test cases, where n is the number of input values. The 'subtraction' program will have $5^2 = 25$ test cases which are shown in Table 9.10.

Table 9.10 Worst test cases for 'subtraction' program

Test case	a	b	Expected output
1	100	200	–100
2	100	201	–101
3	100	700	–600
4	100	1199	–1099
5	100	1200	–1100
6	101	200	–99
7	101	201	–100
8	101	700	–599
9	101	1199	–1098
10	101	1200	–1099
11	550	200	350
12	550	201	349
13	550	700	–150
14	550	1199	–649
15	550	1200	–650
16	999	200	799
17	999	201	798
18	999	700	299
19	999	1199	–200
20	999	1200	–201
21	1000	200	800
22	1000	201	799
23	1000	700	300
24	1000	1199	–199
25	1000	1200	–200

The inputs given in Table 9.10 are graphically represented in Figure 9.5.

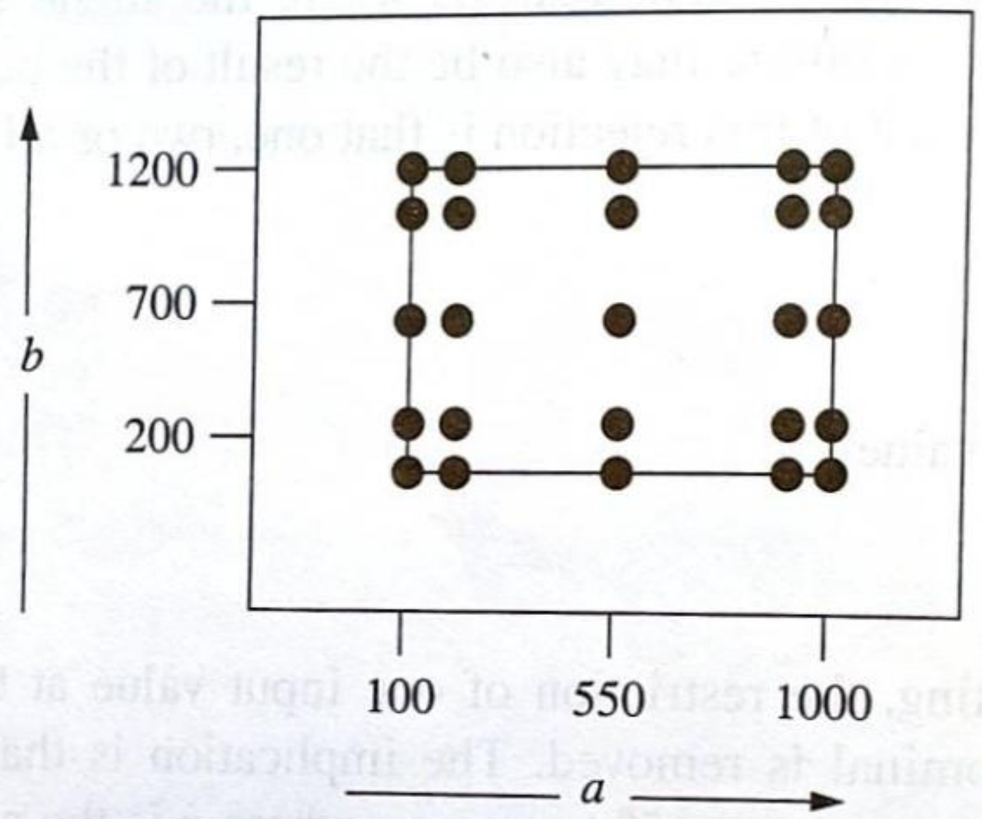

Figure 9.5 Graphical representation of inputs.

In the worst case testing, we select more number of test cases for completeness and thoroughness. This is a more effort and time-consuming technique.

(iii) Robust worst case testing

Two more states are added to check the behaviour of the program with invalid inputs. These two states are 'just below minimum' and 'just above maximum'. The total number of test cases generated by this technique is 7^n. The subtraction program has $7^2 = 49$ test cases which are given in Table 9.11.

Table 9.11 Robust worst test cases for 'subtraction' program

Test case	*a*	*b*	Expected output
1	99	199	Invalid input
2	99	200	Invalid input
3	99	201	Invalid input
4	99	700	Invalid input
5	99	1199	Invalid input
6	99	1200	Invalid input
7	99	1201	Invalid input
8	100	199	Invalid input
9	100	200	–100
10	100	201	–101
11	100	700	–600
12	100	1199	–1099
13	100	1200	–1100
14	100	1201	Invalid input
15	101	199	Invalid input

(Contd.)

Table 9.11 Robust worst test cases for 'subtraction' program (*Contd.*)

Test case	*a*	*b*	Expected output
16	101	200	–99
17	101	201	–100
18	101	700	–599
19	101	1199	–1098
20	101	1200	–1099
21	101	1201	Invalid input
22	550	199	Invalid input
23	550	200	350
24	550	201	349
25	550	700	–150
26	550	1199	–649
27	550	1200	–650
28	550	1201	Invalid input
29	999	199	Invalid input
30	999	200	799
31	999	201	798
32	999	700	299
33	999	1199	–200
34	999	1200	–201
35	999	1201	Invalid input
36	1000	199	Invalid input
37	1000	200	800
38	1000	201	799
39	1000	700	300
40	1000	1199	–199
41	1000	1200	–200
42	1000	1201	Invalid input
43	1001	199	Invalid input
44	1001	200	Invalid input
45	1001	201	Invalid input
46	1001	700	Invalid input
47	1001	1199	Invalid input
48	1001	1200	Invalid input
49	1001	1201	Invalid input

The graphical representation of the input domain of 'subtraction' program is given in Figure 9.6.

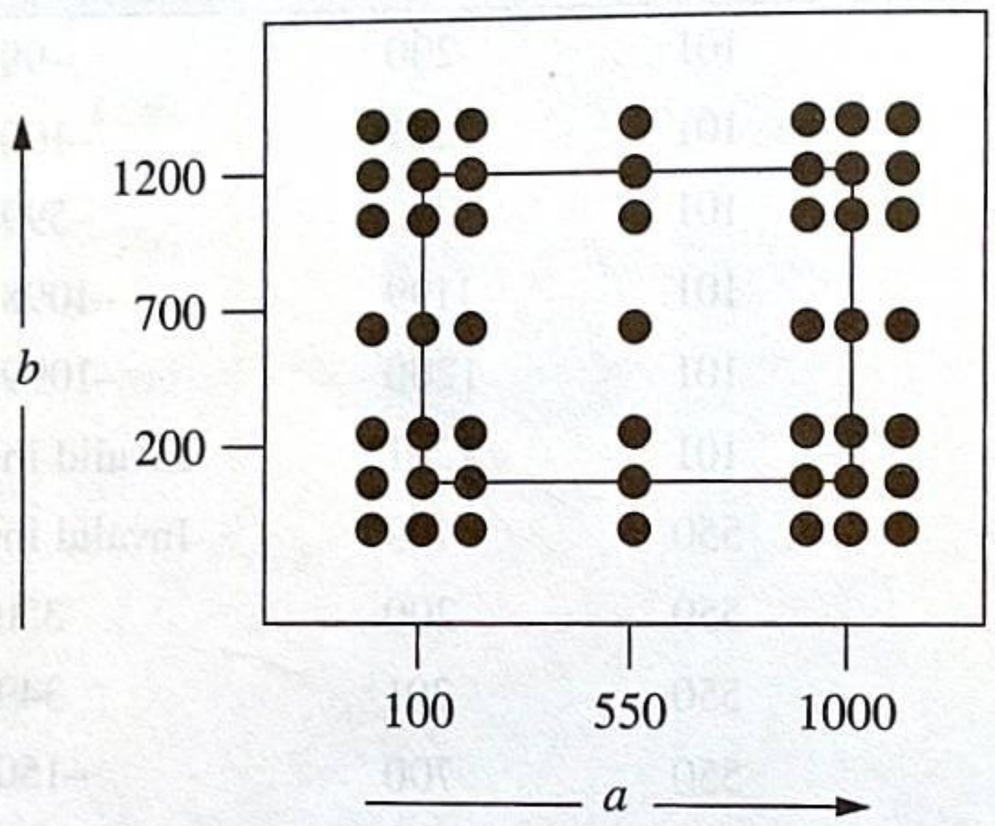

Figure 9.6 Graphical representation of inputs.

Boundary value analysis is a useful technique and may generate effective test cases. There is a restriction that input values should be independent. It is not applicable for Boolean variables because of only two available states—true and false. The technique is also easy to understand and implement for any size of program.

EXAMPLE 9.3 Consider a program to multiply and divide two numbers as explained in Example 9.1. Design the robust test cases and worst test cases for this program.

Solution Robust test cases are given in Table 9.12.

Table 9.12 Robust test cases for a program to multiply and divide two numbers

Test case	*a*	*b*	Expected output	
1	–1	50	Input values are out of range	
2	0	50	0	0
3	1	50	50	0
4	50	50	2500	1
5	99	50	4950	1
6	100	50	5000	2
7	101	50	Input values are out of range	
8	50	–1	Input values are out of range	
9	50	0	0	Divide by zero error
10	50	1	50	50
11	50	99	4950	0
12	50	100	5000	0
13	50	101	Input values are out of range	

Worst test cases are given in Table 9.13.

Table 9.13 Worst test cases for a program to multiply and divide two numbers

Test case	a	b	Expected output	
			Multiply	Divide
1	0	0	0	Undefined*
2	0	1	0	0
3	0	50	0	0
4	0	99	0	0
5	0	100	0	0
6	1	0	0	Divide by zero error
7	1	1	1	1
8	1	50	50	1
9	1	99	99	1
10	1	100	100	1
11	50	0	0	Divide by zero error
12	50	1	50	50
13	50	50	2500	1
14	50	99	4950	0
15	50	100	5000	0
16	99	0	0	Divide by zero error
17	99	1	99	99
18	99	50	4950	1
19	99	99	9801	1
20	99	100	9900	0
21	100	0	0	Divide by zero error
22	100	1	100	100
23	100	50	5000	2
24	100	99	9900	1
25	100	100	10,000	1

*0/0 is still undefined.

EXAMPLE 9.4 Consider the program for the determination of validity of the date as explained in Example 9.2. Design the robust and worst test cases for this program.

Solution Robust test cases and worst test cases are given in Tables 9.14 and 9.15, respectively.

Table 9.14 Robust test cases for program for determining the validity of the date

Test case	Month	Day	Year	Expected output
1	0	15	2035	Invalid date
2	1	15	2035	Valid date (1/15/2035)
3	2	15	2035	Valid date (2/15/2035)
4	6	15	2035	Valid date (6/15/2035)
5	11	15	2035	Valid date (11/15/2035)
6	12	15	2035	Valid date (12/15/2035)
7	13	15	2035	Invalid date
8	6	0	2035	Invalid date
9	6	1	2035	Valid date (6/1/2035)
10	6	2	2035	Valid date (6/2/2035)
11	6	30	2035	Valid date (6/30/2035)
12	6	31	2035	Invalid date
13	6	32	2035	Invalid date
14	6	15	1999	Invalid date (out of range)
15	6	15	2000	Valid date (6/15/2000)
16	6	15	2001	Valid date (6/15/2001)
17	6	15	2069	Valid date (6/15/2069)
18	6	15	2070	Valid date (6/15/2070)
19	6	15	2071	Invalid date (out of range)

Table 9.15 Worst test cases for the program determining the validity of the date

Test case	Month	Day	Year	Expected output
1	1	1	2000	Valid date (1/1/2000)
2	1	1	2001	Valid date (1/1/2001)
3	1	1	2035	Valid date (1/1/2035)
4	1	1	2069	Valid date (1/1/2069)
5	1	1	2070	Valid date (1/1/2070)
6	1	2	2000	Valid date (1/2/2000)
7	1	2	2001	Valid date (1/2/2001)
8	1	2	2035	Valid date (1/2/2035)
9	1	2	2069	Valid date (1/2/2069)
10	1	2	2070	Valid date (1/2/2070)
11	1	15	2000	Valid date (1/15/2000)
12	1	15	2001	Valid date (1/15/2001)
13	1	15	2035	Valid date (1/15/2035)
14	1	15	2069	Valid date (1/15/2069)

(Contd.)

Table 9.15 Worst test cases for the program determining the validity of the date (*Contd.*)

Test case	Month	Day	Year	Expected output
15	1	15	2070	Valid date (1/15/2070)
16	1	30	2000	Valid date (1/30/2000)
17	1	30	2001	Valid date (1/30/2001)
18	1	30	2035	Valid date (1/30/2035)
19	1	30	2069	Valid date (1/30/2069)
20	1	30	2070	Valid date (1/30/2070)
21	1	31	2000	Valid date (1/31/2000)
22	1	31	2001	Valid date (1/31/2001)
23	1	31	2035	Valid date (1/31/2035)
24	1	31	2069	Valid date (1/31/2069)
25	1	31	2070	Valid date (1/31/2070)
26	2	1	2000	Valid date (2/1/2000)
27	2	1	2001	Valid date (2/1/2001)
28	2	1	2035	Valid date (2/1/2035)
29	2	1	2069	Valid date (2/1/2069)
30	2	1	2070	Valid date (2/1/2070)
31	2	2	2000	Valid date (2/2/2000)
32	2	2	2001	Valid date (2/2/2001)
33	2	2	2035	Valid date (2/2/2035)
34	2	2	2069	Valid date (2/2/2069)
35	2	2	2070	Valid date (2/2/2070)
36	2	15	2000	Valid date (2/15/2000)
37	2	15	2001	Valid date (2/15/2001)
38	2	15	2035	Valid date (2/15/2035)
39	2	15	2069	Valid date (2/15/2069)
40	2	15	2070	Valid date (2/15/2070)
41	2	30	2000	Invalid date
42	2	30	2001	Invalid date
43	2	30	2035	Invalid date
44	2	30	2069	Invalid date
45	2	30	2070	Invalid date
46	2	31	2000	Invalid date
47	2	31	2001	Invalid date
48	2	31	2035	Invalid date
49	2	31	2069	Invalid date
50	2	31	2070	Invalid date

(*Contd.*)

Table 9.15 Worst test cases for the program determining the validity of the date (*Contd.*)

Test case	Month	Day	Year	Expected output
51	6	1	2000	Valid date (6/1/2000)
52	6	1	2001	Valid date (6/1/2001)
53	6	1	2035	Valid date (6/1/2035)
54	6	1	2069	Valid date (6/1/2069)
55	6	1	2070	Valid date (6/1/2070)
56	6	2	2000	Valid date (6/2/2000)
57	6	2	2001	Valid date (6/2/2001)
58	6	2	2035	Valid date (6/2/2035)
59	6	2	2069	Valid date (6/2/2069)
60	6	2	2070	Valid date (6/2/2070)
61	6	15	2000	Valid date (6/15/2000)
62	6	15	2001	Valid date (6/15/2001)
63	6	15	2035	Valid date (6/15/2035)
64	6	15	2069	Valid date (6/15/2069)
65	6	15	2070	Valid date (6/15/2070)
66	6	30	2000	Valid date (6/30/2000)
67	6	30	2001	Valid date (6/30/2001)
68	6	30	2035	Valid date (6/30/2035)
69	6	30	2069	Valid date (6/30/2069)
70	6	30	2070	Valid date (6/30/2070)
71	6	31	2000	Invalid date
72	6	31	2001	Invalid date
73	6	31	2035	Invalid date
74	6	31	2069	Invalid date
75	6	31	2070	Invalid date
76	11	1	2000	Valid date (11/1/2000)
77	11	1	2001	Valid date (11/1/2001)
78	11	1	2035	Valid date (11/1/2035)
79	11	1	2069	Valid date (11/1/2069)
80	11	1	2070	Valid date (11/1/2070)
81	11	2	2000	Valid date (11/2/2000)
82	11	2	2001	Valid date (11/2/2001)
83	11	2	2035	Valid date (11/2/2035)
84	11	2	2069	Valid date (11/2/2069)
85	11	2	2070	Valid date (11/2/2070)
86	11	15	2000	Valid date (11/15/2000)
87	11	15	2001	Valid date (11/15/2001)
88	11	15	2035	Valid date (11/15/2035)

Table 9.15 Worst test cases for the program determining the validity of the date (*Contd.*)

Test case	Month	Day	Year	Expected output
89	11	15	2069	Valid date (11/15/2069)
90	11	15	2070	Valid date (11/15/2070)
91	11	30	2000	Valid date (11/30/2000)
92	11	30	2001	Valid date (11/30/2001)
93	11	30	2035	Valid date (11/30/2035)
94	11	30	2069	Valid date (11/30/2069)
95	11	30	2070	Valid date (11/30/2070)
96	11	31	2000	Invalid date
97	11	31	2001	Invalid date
98	11	31	2035	Invalid date
99	11	31	2069	Invalid date
100	11	31	2070	Invalid date
101	12	1	2000	Valid date (12/1/2000)
102	12	1	2001	Valid date (12/1/2001)
103	12	1	2035	Valid date (12/1/2035)
104	12	1	2069	Valid date (12/1/2069)
105	12	1	2070	Valid date (12/1/2070)
106	12	2	2000	Valid date (12/2/2000)
107	12	2	2001	Valid date (12/2/2001)
108	12	2	2035	Valid date (12/2/2035)
109	12	2	2069	Valid date (12/2/2069)
110	12	2	2070	Valid date (12/2/2070)
111	12	15	2000	Valid date (12/15/2000)
112	12	15	2001	Valid date (12/15/2001)
113	12	15	2035	Valid date (12/15/2035)
114	12	15	2069	Valid date (12/15/2069)
115	12	15	2070	Valid date (12/15/2070)
116	12	30	2000	Valid date (12/30/2000)
117	12	30	2001	Valid date (12/30/2001)
118	12	30	2035	Valid date (12/30/2035)
119	12	30	2069	Valid date (12/30/2069)
120	12	30	2070	Valid date (12/30/2070)
121	12	31	2000	Valid date (12/31/2000)
122	12	31	2001	Valid date (12/31/2001)
123	12	31	2035	Valid date (12/31/2035)
124	12	31	2069	Valid date (12/31/2069)
125	12	31	2070	Valid date (12/31/2070)

9.4.2 Equivalence Class Testing

A program may have a large number of test cases. It may not be desirable to execute all test cases due to time and resource constraints. Many test cases may execute the same lines of source code again and again, and therefore, there is hardly any value addition. We may partition input domain into various groups on the basis of some relationship. We expect that any test case from that group will become the representative test case and produce the same behaviour. If a test case makes the program fail, then other test cases of the same group will also make the program fail. Similarly, if the output of the program is correct for one test case, the same behaviour is expected from other test cases of the group. This assumption of similar behaviour of the program for all test cases of the same group allows us to select only one test case from each group. If groups are made properly, a few test cases (equal to the number of groups) may give reasonable confidence about the correctness of the program. In this technique, each group is called an equivalence class. We may also partition the output domain into equivalence classes and generate one test case from each class.

Design of Equivalence Classes

In general, without considering any relationship, an input domain can be divided into at least two equivalence classes, i.e. one class of all valid inputs and other class of all invalid inputs. We may design many classes on the basis of relationships and logic of the program. The input domain and output domain may generate a good number of classes and every class gives us a test case which will represent all test cases of that class. We consider the program 'square root' which takes a as an input in the range (100–1000) and prints the square root of a. We may generate the following equivalence classes for the input domain:

$I_1 = \{100 \leq a \leq 1000\}$ (valid input range from 100 to 1000)

$I_2 = \{a < 100\}$ (any invalid input where a is less than 100)

$I_3 = \{a > 1000\}$ (any invalid input where a is more than 1000)

The above three equivalence classes are designed without considering any relationship. For example, the first class I_1 represents a valid input class where all inputs are within the specified range $\{100 \leq a \leq 1000\}$. Three test cases are generated, one from every equivalence class, and are given in Table 9.16.

Table 9.16 Test cases of 'square root' program for input domain

Test case	Input a	Expected output
I_1	500	22.36
I_2	10	Invalid input
I_3	1100	Invalid input

The output domain is also partitioned to generate equivalence classes. We generate the following output domain equivalence classes:

$O_1 = \{$Square root of input value $a\}$

$O_2 = \{$Invalid value$\}$

The test cases for the output domain are given in Table 9.17. Some of the input and output domain test cases may be the same.

Table 9.17 Test cases of 'square root' program for output domain

Test case	Input a	Expected output
O_1	500	22.36
O_2	1100	Invalid input

We consider the 'subtraction' program which takes two inputs a (range 100–1000) and b (range 200–1200) and performs the subtraction of these two input values. On the basis of the input domain, we generate the following equivalence classes:

(i) $I_1 = \{100 \leq a \leq 1000 \text{ and } 200 \leq b \leq 1200\}$ (Both a and b are valid values)
(ii) $I_2 = \{100 \leq a \leq 1000 \text{ and } b < 200\}$ (a is valid and b is invalid)
(iii) $I_3 = \{100 \leq a \leq 1000 \text{ and } b > 1200\}$ (a is valid and b is invalid)
(iv) $I_4 = \{a < 100 \text{ and } 200 \leq b \leq 1200\}$ (a is invalid and b is valid)
(v) $I_5 = \{a > 1000 \text{ and } 200 \leq b \leq 1200\}$ (a is invalid and b is valid)
(vi) $I_6 = \{a < 100 \text{ and } b < 200\}$ (Both inputs are invalid)
(vii) $I_7 = \{a < 100 \text{ and } b > 1200\}$ (Both inputs are invalid)
(viii) $I_8 = \{a > 1000 \text{ and } b < 200\}$ (Both inputs are invalid)
(ix) $I_9 = \{a > 1000 \text{ and } b > 1200\}$ (Both inputs are invalid)

The input domain test cases are given in Table 9.18.

Table 9.18 Test cases of 'subtraction' program for input domain

Test case	a	b	Expected output
I_1	600	300	300
I_2	600	100	Invalid input
I_3	600	1300	Invalid input
I_4	10	300	Invalid input
I_5	1100	300	Invalid input
I_6	10	100	Invalid input
I_7	10	1300	Invalid input
I_8	1100	100	Invalid input
I_9	1100	1300	Invalid input

The output domain equivalence classes are given as follows:

$O_1 = \{\text{subtraction of two values } a \text{ and } b\}$
$O_2 = \{\text{Invalid input}\}$

The test cases for the output domain are given in Table 9.19.

Table 9.19 Test cases of 'subtraction' program for output domain

Test case	*a*	*b*	Expected output
O_1	600	300	300
O_2	10	300	Invalid input

We have designed only one equivalence class for the valid input domain in both of the above discussed examples. We could not partition the valid input domain due to simplicity of the problem. However, programs are more complex and logic oriented. We may design more valid input domain equivalence classes on the basis of logic and relationship. The design of equivalence classes is subjective and two designers may design different classes. The objective is to cover every functionality of the program which may further cover maximum portion of the source code during testing. This technique is applicable at unit, integration system and acceptance testing levels. This is a very effective technique if equivalence classes are designed correctly and also reduces the number of test cases significantly.

EXAMPLE 9.5 Consider a program to multiply and divide two numbers. The inputs may be two valid integers (say *a* and *b*) in the range of [0, 100]. Develop test cases using equivalence class testing.

Solution The test cases are generated for output domain given in Table 9.20:

Table 9.20 Test cases for output domain

Test case	*a*	*b*	Expected output	
			Multiply	Divide
O_1	50	50	2500	1
O_2	101	50	Input values are out of range	

The equivalence classes for input domain are shown below:

(i) $I_1 = \{0 \le a \le 100 \text{ and } 0 \le b \le 100\}$ (Both *a* and *b* are valid values)
(ii) $I_2 = \{0 \le a \le 100 \text{ and } b < 0\}$ (*a* is valid and *b* is invalid)
(iii) $I_3 = \{0 \le a \le 100 \text{ and } b > 100\}$ (*a* is valid and *b* is invalid)
(iv) $I_4 = \{a < 0 \text{ and } 0 \le b \le 100\}$ (*a* is invalid and *b* is valid)
(v) $I_5 = \{a > 100 \text{ and } 0 \le b \le 100\}$ (*a* is invalid and *b* is valid)
(vi) $I_6 = \{a < 0 \text{ and } b < 0\}$ (Both inputs are invalid)
(vii) $I_7 = \{a < 0 \text{ and } b > 100\}$ (Both inputs are invalid)
(viii) $I_8 = \{a > 100 \text{ and } b < 0\}$ (Both inputs are invalid)
(ix) $I_9 = \{a > 100 \text{ and } b > 100\}$ (Both inputs are invalid)

The test cases for input domain are given in Table 9.21.

Table 9.21 Test cases for input domain

Test case	a	b	Expected output	
I_1	50	50	2500	1
I_2	50	–1	Input values are out of range	
I_3	50	101	Input values are out of range	
I_4	–1	50	Input values are out of range	
I_5	101	50	Input values are out of range	
I_6	–1	–1	Input values are out of range	
I_7	–1	101	Input values are out of range	
I_8	101	–1	Input values are out of range	
I_9	101	101	Input values are out of range	

EXAMPLE 9.6 Consider the program for determining whether the date is valid or not. Identify the equivalence class test cases for output and input domains.

Solution Output domain equivalence classes are:

O_1 = {< Day, Month, Year > : Valid}

O_2 = {< Day, Month, Year > : Invalid date if any of the inputs is invalid}

O_3 = {< Day, Month, Year > : Input is out of range if any of the inputs is out of range}

The output domain test cases are given in Table 9.22.

Table 9.22 Output domain equivalence class test cases

Test case	Month	Day	Year	Expected output
O_1	6	11	1979	Valid date
O_2	6	31	1979	Invalid date
O_3	6	32	1979	Inputs out of range

The input domain is partitioned as given below:

(i) Valid partitions

M_1: Month has 30 days

M_2: Month has 31 days

M_3: Month is February

D_1: Days of a month from 1 to 28

D_2: Day = 29

D_3: Day = 30

D_4: Day = 31

Y_1: 1900 ≤ year ≤ 2058 and is a common year

Y_2: 1900 ≤ year ≤ 2058 and is a leap year

(ii) Invalid partitions

M_4: Month < 1

M_5: Month > 12

D_5: Day < 1

D_6: Day > 31
Y_3: Year < 1900
Y_4: Year > 2058

We may have the following set of test cases which are based on input domain:

(a) Only for valid input domain

$I_1 = \{M_1$ and D_1 and $Y_1\}$ (All inputs are valid)
$I_2 = \{M_2$ and D_1 and $Y_1\}$ (All inputs are valid)
$I_3 = \{M_3$ and D_1 and $Y_1\}$ (All inputs are valid)
$I_4 = \{M_1$ and D_2 and $Y_1\}$ (All inputs are valid)
$I_5 = \{M_2$ and D_2 and $Y_1\}$ (All inputs are valid)
$I_6 = \{M_3$ and D_2 and $Y_1\}$ (All inputs are valid)
$I_7 = \{M_1$ and D_3 and $Y_1\}$ (All inputs are valid)
$I_8 = \{M_2$ and D_3 and $Y_1\}$ (All inputs are valid)
$I_9 = \{M_3$ and D_3 and $Y_1\}$ (All inputs are valid)
$I_{10} = \{M_1$ and D_4 and $Y_1\}$ (All inputs are valid)
$I_{11} = \{M_2$ and D_4 and $Y_1\}$ (All inputs are valid)
$I_{12} = \{M_3$ and D_4 and $Y_1\}$ (All inputs are valid)
$I_{13} = \{M_1$ and D_1 and $Y_2\}$ (All Inputs are valid)
$I_{14} = \{M_2$ and D_1 and $Y_2\}$ (All inputs are valid)
$I_{15} = \{M_3$ and D_1 and $Y_2\}$ (All inputs are valid)
$I_{16} = \{M_1$ and D_2 and $Y_2\}$ (All inputs are valid)
$I_{17} = \{M_2$ and D_2 and $Y_2\}$ (All inputs are valid)
$I_{18} = \{M_3$ and D_2 and $Y_2\}$ (All inputs are valid)
$I_{19} = \{M_1$ and D_3 and $Y_2\}$ (All inputs are valid)
$I_{20} = \{M_2$ and D_3 and $Y_2\}$ (All inputs are valid)
$I_{21} = \{M_3$ and D_3 and $Y_2\}$ (All inputs are valid)
$I_{22} = \{M_1$ and D_4 and $Y_2\}$ (All inputs are valid)
$I_{23} = \{M_2$ and D_4 and $Y_2\}$ (All inputs are valid)
$I_{24} = \{M_3$ and D_4 and $Y_2\}$ (All inputs are valid)

(b) Only for invalid input domain

$I_{25} = \{M_4$ and D_1 and $Y_1\}$ (Month is invalid, Day is valid and Year is valid)
$I_{26} = \{M_5$ and D_1 and $Y_1\}$ (Month is invalid, Day is valid and Year is valid)
$I_{27} = \{M_4$ and D_2 and $Y_1\}$ (Month is invalid, Day is valid and Year is valid)
$I_{28} = \{M_5$ and D_2 and $Y_1\}$ (Month is invalid, Day is valid and Year is valid)
$I_{29} = \{M_4$ and D_3 and $Y_1\}$ (Month is invalid, Day is valid and Year is valid)
$I_{30} = \{M_5$ and D_3 and $Y_1\}$ (Month is invalid, Day is valid and Year is valid)
$I_{31} = \{M_4$ and D_4 and $Y_1\}$ (Month is invalid, Day is valid and Year is valid)
$I_{32} = \{M_5$ and D_4 and $Y_1\}$ (Month is invalid, Day is valid and Year is valid)
$I_{33} = \{M_4$ and D_1 and $Y_2\}$ (Month is invalid, Day is valid and Year is valid)
$I_{34} = \{M_5$ and D_1 and $Y_2\}$ (Month is invalid, Day is valid and Year is valid)

I_{35} = {M_4 and D_2 and Y_2} (Month is invalid, Day is valid and Year is valid)
I_{36} = {M_5 and D_2 and Y_2} (Month is invalid, Day is valid and Year is valid)
I_{37} = {M_4 and D_3 and Y_2} (Month is invalid, Day is valid and Year is valid)
I_{38} = {M_5 and D_3 and Y_2} (Month is invalid, Day is valid and Year is valid)
I_{39} = {M_4 and D_4 and Y_2} (Month is invalid, Day is valid and Year is valid)
I_{40} = {M_5 and D_4 and Y_2} (Month is invalid, Day is valid and Year is valid)
I_{41} = {M_1 and D_5 and Y_1} (Month is valid, Day is invalid and Year is valid)
I_{42} = {M_1 and D_6 and Y_1} (Month is valid, Day is invalid and Year is valid)
I_{43} = {M_2 and D_5 and Y_1} (Month is valid, Day is invalid and Year is valid)
I_{44} = {M_2 and D_6 and Y_1} (Month is valid, Day is invalid and Year is valid)
I_{45} = {M_3 and D_5 and Y_1} (Month is valid, Day is invalid and Year is valid)
I_{46} = {M_3 and D_6 and Y_1} (Month is valid, Day is invalid and Year is valid)
I_{47} = {M_1 and D_5 and Y_2} (Month is valid, Day is invalid and Year is valid)
I_{48} = {M_1 and D_6 and Y_2} (Month is valid, Day is invalid and Year is valid)
I_{49} = {M_2 and D_5 and Y_2} (Month is valid, Day is invalid and Year is valid)
I_{50} = {M_2 and D_6 and Y_2} (Month is valid, Day is invalid and Year is valid)
I_{51} = {M_3 and D_5 and Y_2} (Month is valid, Day is invalid and Year is valid)
I_{52} = {M_3 and D_6 and Y_2} (Month is valid, Day is invalid and Year is valid)
I_{53} = {M_1 and D_1 and Y_3} (Month is valid, Day is valid and Year is invalid)
I_{54} = {M_1 and D_1 and Y_4} (Month is valid, Day is valid and Year is invalid)
I_{55} = {M_2 and D_1 and Y_3} (Month is valid, Day is valid and Year is invalid)
I_{56} = {M_2 and D_1 and Y_4} (Month is valid, Day is valid and Year is invalid)
I_{57} = {M_3 and D_1 and Y_3} (Month is valid, Day is valid and Year is invalid)
I_{58} = {M_3 and D_1 and Y_4} (Month is valid, Day is valid and Year is invalid)
I_{59} = {M_1 and D_2 and Y_3} (Month is valid, Day is valid and Year is invalid)
I_{60} = {M_1 and D_2 and Y_4} (Month is valid, Day is valid and Year is invalid)
I_{61} = {M_2 and D_2 and Y_3} (Month is valid, Day is valid and Year is invalid)
I_{62} = {M_2 and D_2 and Y_4} (Month is valid, Day is valid and Year is invalid)
I_{63} = {M_3 and D_2 and Y_3} (Month is valid, Day is valid and Year is invalid)
I_{64} = {M_3 and D_2 and Y_4} (Month is valid, Day is valid and Year is invalid)
I_{65} = {M_1 and D_3 and Y_3} (Month is valid, Day is valid and Year is invalid)
I_{66} = {M_1 and D_3 and Y_3} (Month is valid, Day is valid and Year is invalid)
I_{67} = {M_2 and D_3 and Y_3} (Month is valid, Day is valid and Year is invalid)
I_{68} = {M_2 and D_3 and Y_4} (Month is valid, Day is valid and Year is invalid)
I_{69} = {M_3 and D_3 and Y_3} (Month is valid, Day is valid and Year is invalid)
I_{70} = {M_3 and D_3 and Y_4} (Month is valid, Day is valid and Year is invalid)
I_{71} = {M_1 and D_4 and Y_3} (Month is valid, Day is valid and Year is invalid)
I_{72} = {M_1 and D_4 and Y_4} (Month is valid, Day is valid and Year is invalid)
I_{73} = {M_2 and D_4 and Y_3} (Month is valid, Day is valid and Year is invalid)
I_{74} = {M_2 and D_4 and Y_4} (Month is valid, Day is valid and Year is invalid)

I_{75} = {M_3 and D_4 and Y_3} (Month is valid, Day is valid and Year is invalid)
I_{76} = {M_3 and D_4 and Y_4} (Month is valid, Day is valid and Year is invalid)
I_{77} = {M_4 and D_5 and Y_1} (Month is invalid, Day is invalid and Year is valid)
I_{78} = {M_4 and D_5 and Y_2} (Month is invalid, Day is invalid and Year is valid)
I_{79} = {M_4 and D_6 and Y_1} (Month is invalid, Day is invalid and Year is valid)
I_{80} = {M_4 and D_6 and Y_2} (Month is invalid, Day is invalid and Year is valid)
I_{81} = {M_5 and D_5 and Y_1} (Month is invalid, Day is invalid and Year is valid)
I_{82} = {M_5 and D_5 and Y_2} (Month is invalid, Day is invalid and Year is valid)
I_{83} = {M_5 and D_6 and Y_1} (Month is invalid, Day is invalid and Year is valid)
I_{84} = {M_5 and D_6 and Y_2} (Month is invalid, Day is invalid and Year is valid)
I_{85} = {M_4 and D_1 and Y_3} (Month is invalid, Day is valid and Year is invalid)
I_{86} = {M_4 and D_1 and Y_4} (Month is invalid, Day is valid and Year is invalid)
I_{87} = {M_4 and D_2 and Y_3} (Month is invalid, Day is valid and Year is invalid)
I_{88} = {M_4 and D_2 and Y_4} (Month is invalid, Day is valid and Year is invalid)
I_{89} = {M_4 and D_3 and Y_3} (Month is invalid, Day is valid and Year is invalid)
I_{90} = {M_4 and D_3 and Y_4} (Month is invalid, Day is valid and Year is invalid)
I_{91} = {M_4 and D_4 and Y_3} (Month is invalid, Day is valid and Year is invalid)
I_{92} = {M_4 and D_4 and Y_4} (Month is invalid, Day is valid and Year is invalid)
I_{93} = {M_5 and D_1 and Y_3} (Month is invalid, Day is valid and Year is invalid)
I_{94} = {M_5 and D_1 and Y_4} (Month is invalid, Day is valid and Year is invalid)
I_{95} = {M_5 and D_2 and Y_3} (Month is invalid, Day is valid and Year is invalid)
I_{96} = {M_5 and D_2 and Y_4} (Month is invalid, Day is valid and Year is invalid)
I_{97} = {M_5 and D_3 and Y_3} (Month is invalid, Day is valid and Year is invalid)
I_{98} = {M_5 and D_3 and Y_4} (Month is invalid, Day is valid and Year is invalid)
I_{99} = {M_5 and D_4 and Y_3} (Month is invalid, Day is valid and Year is invalid)
I_{100} = {M_5 and D_4 and Y_4} (Month is invalid, Day is valid and Year is invalid)
I_{101} = {M_1 and D_5 and Y_3} (Month is valid, Day is invalid and Year is invalid)
I_{102} = {M_1 and D_5 and Y_4} (Month is valid, Day is invalid and Year is invalid)
I_{103} = {M_2 and D_5 and Y_3} (Month is valid, Day is invalid and Year is invalid)
I_{104} = {M_2 and D_5 and Y_4} (Month is valid, Day is invalid and Year is invalid)
I_{105} = {M_3 and D_5 and Y_3} (Month is valid, Day is invalid and Year is invalid)
I_{106} = {M_3 and D_5 and Y_4} (Month is valid, Day is invalid and Year is invalid)
I_{107} = {M_1 and D_6 and Y_3} (Month is valid, Day is invalid and Year is invalid)
I_{108} = {M_1 and D_6 and Y_4} (Month is valid, Day is invalid and Year is invalid)
I_{109} = {M_2 and D_6 and Y_3} (Month is valid, Day is invalid and Year is invalid)
I_{110} = {M_2 and D_6 and Y_4} (Month is valid, Day is invalid and Year is invalid)
I_{111} = {M_3 and D_6 and Y_3} (Month is valid, Day is invalid and Year is invalid)
I_{112} = {M_3 and D_6 and Y_4} (Month is valid, Day is invalid and Year is invalid)
I_{113} = {M_4 and D_5 and Y_3} (All inputs are invalid)

I_{114} = {M_4 and D_5 and Y_4} (All inputs are invalid)
I_{115} = {M_4 and D_6 and Y_3} (All inputs are invalid)
I_{116} = {M_4 and D_6 and Y_4} (All inputs are invalid)
I_{117} = {M_5 and D_5 and Y_3} (All inputs are invalid)
I_{118} = {M_5 and D_5 and Y_4} (All inputs are invalid)
I_{119} = {M_5 and D_6 and Y_3} (All inputs are invalid)
I_{120} = {M_5 and D6 and Y_4} (All inputs are invalid)

The test cases generated on the basis of input domain are given in Table 9.23.

Table 9.23 Input domain equivalence class test cases

Test case	Month	Day	Year	Expected output
I_1	6	15	2035	Valid date
I_2	5	15	2035	Valid date
I_3	2	15	2035	Valid date
I_4	6	29	2035	Valid date
I_5	5	29	2035	Valid date
I_6	2	29	2035	Invalid date
I_7	6	30	2035	Valid date
I_8	5	30	2035	Valid date
I_9	2	30	2035	Invalid date
I_{10}	6	31	2035	Invalid date
I_{11}	5	31	2035	Valid date
I_{12}	2	31	2035	Invalid date
I_{13}	6	15	2000	Valid date
I_{14}	5	15	2000	Valid date
I_{15}	2	15	2000	Valid date
I_{16}	6	29	2000	Valid date
I_{17}	5	29	2000	Valid date
I_{18}	2	29	2000	Valid date
I_{19}	6	30	2000	Valid date
I_{20}	5	30	2000	Valid date
I_{21}	2	30	2000	Invalid date
I_{22}	6	31	2000	Invalid date
I_{23}	5	31	2000	Valid date
I_{24}	2	31	2000	Invalid date
I_{25}	0	15	2035	Input(s) out of range
I_{26}	13	15	2035	Input(s) out of range
I_{27}	0	29	2035	Inputs(s) out of range
I_{28}	13	29	2035	Input(s) out of range

(Contd.)

Table 9.23 Input domain equivalence class test cases (*Contd.*)

Test case	Month	Day	Year	Expected output
I_{29}	0	30	2035	Input(s) out of range
I_{30}	13	30	2035	Input(s) out of range
I_{31}	0	31	2035	Input(s) out of range
I_{32}	13	31	2035	Input(s) out of range
I_{33}	0	15	2000	Input(s) out of range
I_{34}	13	15	2000	Input(s) out of range
I_{35}	0	29	2000	Input(s) out of range
I_{36}	13	29	2000	Input(s) out of range
I_{37}	0	30	2000	Input(s) out of range
I_{38}	13	30	2000	Input(s) out of range
I_{39}	0	31	2000	Input(s) out of range
I_{40}	13	31	2000	Input(s) out of range
I_{41}	6	0	2035	Input(s) out of range
I_{42}	6	32	2035	Input(s) out of range
I_{43}	5	0	2035	Input(s) out of range
I_{44}	5	32	2035	Input(s) out of range
I_{45}	2	0	2035	Input(s) out of range
I_{46}	2	32	2035	Input(s) out of range
I_{47}	6	0	2000	Input(s) out of range
I_{48}	6	32	2000	Input(s) out of range
I_{49}	5	0	2000	Input(s) out of range
I_{50}	5	32	2000	Input(s) out of range
I_{51}	2	0	2000	Input(s) out of range
I_{52}	2	32	2000	Input(s) out of range
I_{53}	6	15	1899	Input(s) out of range
I_{54}	6	15	2059	Input(s) out of range
I_{55}	5	15	1899	Input(s) out of range
I_{56}	5	15	2059	Input(s) out of range
I_{57}	2	15	1899	Input(s) out of range
I_{58}	2	15	2059	Input(s) out of range
I_{59}	6	29	1899	Input(s) out of range
I_{60}	6	29	2059	Input(s) out of range
I_{61}	5	29	1899	Input(s) out of range
I_{62}	5	29	2059	Input(s) out of range
I_{63}	2	29	1899	Input(s) out of range
I_{64}	2	29	2059	Input(s) out of range
I_{65}	6	30	1899	Input(s) out of range

(Contd.)

Table 9.23 Input domain equivalence class test cases (*Contd.*)

Test case	Month	Day	Year	Expected output
I_{66}	6	30	2059	Input(s) out of range
I_{67}	5	30	1899	Input(s) out of range
I_{68}	5	30	2059	Input(s) out of range
I_{69}	2	30	1899	Input(s) out of range
I_{70}	2	30	2059	Input(s) out of range
I_{71}	6	31	1899	Input(s) out of range
I_{72}	6	31	2059	Input(s) out of range
I_{73}	5	31	1899	Input(s) out of range
I_{74}	5	31	2059	Input(s) out of range
I_{75}	2	31	1899	Input(s) out of range
I_{76}	2	31	2059	Input(s) out of range
I_{77}	0	0	2035	Input(s) out of range
I_{78}	0	0	2000	Input(s) out of range
I_{79}	0	32	2035	Input(s) out of range
I_{80}	0	32	2000	Input(s) out of range
I_{81}	13	0	2035	Input(s) out of range
I_{82}	13	0	2000	Input(s) out of range
I_{83}	13	32	2035	Input(s) out of range
I_{84}	13	32	2000	Input(s) out of range
I_{85}	0	15	1899	Input(s) out of range
I_{86}	0	15	2059	Input(s) out of range
I_{87}	0	20	1899	Input(s) out of range
I_{88}	0	29	2059	Input(s) out of range
I_{89}	0	30	1899	Input(s) out of range
I_{90}	0	30	2059	Input(s) out of range
I_{91}	0	31	1899	Input(s) out of range
I_{92}	0	31	2059	Input(s) out of range
I_{93}	13	15	1899	Input(s) out of range
I_{94}	13	15	2059	Input(s) out of range
I_{95}	13	29	1899	Input(s) out of range
I_{96}	13	29	2059	Input(s) out of range
I_{97}	13	30	1899	Input(s) out of range
I_{98}	13	30	2059	Input(s) out of range
I_{99}	13	31	1899	Input(s) out of range
I_{100}	13	31	2059	Input(s) out of range
I_{101}	5	0	1899	Input(s) out of range
I_{102}	5	0	2059	Input(s) out of range

(*Contd.*)

Table 9.23 Input domain equivalence class test cases (*Contd.*)

Test case	Month	Day	Year	Expected output
I_{103}	6	0	1899	Input(s) out of range
I_{104}	6	0	2059	Input(s) out of range
I_{105}	2	0	1899	Input(s) out of range
I_{106}	2	0	2059	Input(s) out of range
I_{107}	5	32	1899	Input(s) out of range
I_{108}	5	32	2059	Input(s) out of range
I_{109}	6	32	1899	Input(s) out of range
I_{110}	6	32	2059	Input(s) out of range
I_{111}	2	32	1899	Input(s) out of range
I_{112}	2	32	2059	Input(s) out of range
I_{113}	0	0	1899	Input(s) out of range
I_{114}	0	0	2059	Input(s) out of range
I_{115}	0	32	1899	Input(s) out of range
I_{116}	0	32	2059	Input(s) out of range
I_{117}	13	0	1899	Input(s) out of range
I_{118}	13	0	2059	Input(s) out of range
I_{119}	13	32	1899	Input(s) out of range
I_{120}	13	32	2059	Input(s) out of range

9.4.3 Decision Table-Based Testing

Decision tables are commonly used in engineering disciplines to represent complex logical relationships. They are effective to model situations where an output is dependent on many input conditions. Software testers have also found their applications in testing and a technique is developed which is known as decision table-based testing. There are four portions of a decision table, namely, condition stubs, condition entries, action stubs and action entries. A typical decision table is shown in Table 9.24.

Table 9.24 Portion of the decision table

	Stubs	Entries
Condition	All conditions are shown.	Inputs are shown on the basis of conditions. There are columns and each column represents a rule.
Action	All actions are shown.	Represent the outputs on the basis of various input conditions.

There are two types of decision tables. The first type is called the limited entry decision table where input values represent only the true and false condition as shown in Table 9.25.

Table 9.25 Limited entry decision table

Condition stub	Condition entry				
c_1	T	F	F	F	F
c_2	-	T	F	F	F
c_3	-	-	T	F	F
	-	-	-	T	F
a_1	X		X		
a_2		X			X
a_3	X			X	

Every column of the decision table represents a rule and generates a test case. A '-' in the condition entry represents a 'do not care' condition. An 'X' in the action entry represents the action mentioned in the corresponding action stub. In Table 9.25, if condition c_1 is true, then c_2, c_3 and c_4 become 'do not care' conditions and actions a_1 and a_3 are to be performed.

The second type of decision table is called extended entry decision table. In such a table, multiple conditions are used instead of true and false conditions. A condition may have many options instead of only true and false and such options are represented in the extended entry decision table.

We consider a program which takes a date as an input and checks whether it is a valid date or not. We prepare the following classes (options) and represent them in the extended entry decision table as shown in Table 9.26.

$I_1 = \{M_1 : \text{Month has 30 days}\}$
$I_2 = \{M_2 : \text{Month has 31 days}\}$
$I_3 = \{M_3 : \text{Month is February}\}$
$I_4 = \{M_4 : \text{Month} < 1\}$
$I_5 = \{M_5 : \text{Month} > 12\}$
$I_6 = \{D_1 : 1 \leq \text{Day} \leq 28\}$
$I_7 = \{D_2 : \text{Day} = 29\}$
$I_8 = \{D_3 : \text{Day} = 30\}$
$I_9 = \{D_4 : \text{Day} = 31\}$
$I_{10} = \{D_5 : \text{Day} < 1\}$
$I_{11} = \{D_6 : \text{Day} > 31\}$
$I_{12} = \{Y_1 : 1900 \leq \text{Year} \leq 2058 \text{ and is a common year}\}$
$I_{13} = \{Y_2 : 1900 \leq \text{Year} \leq 2058 \text{ and is a leap year}\}$
$I_{14} = \{Y_3 : \text{Year} < 1900\}$
$I_{15} = \{Y_4 : \text{Year} > 2058\}$

In the decision table, various combinations of conditions are considered and that may sometimes result into an impossible action. In such a situation, we may incorporate an additional action 'impossible condition' in the action stub. We may change the design of equivalence classes to reduce the impossible conditions.

Table 9.26 Decision table of date program

Test case	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
c_1 : Months in	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_2	M_2
c_2 : Days in	D_1	D_1	D_1	D_1	D_2	D_2	D_2	D_2	D_3	D_3	D_3	D_3	D_4	D_4	D_4	D_4	D_5	D_6	D_1	D_1
c_3 : Years in	Y_1	Y_2	Y_3	Y_4	Y_1	Y_2	Y_3	Y_4	Y_1	Y_2	Y_3	Y_4	Y_1	Y_2	Y_3	Y_4	-	-	Y_1	Y_2
Rule count	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	4	4	1	1
a_1 : Invalid date													X	X						
a_2 : Valid date	X	X			X	X			X	X									X	X
a_3 : Input out of range			X	X			X	X			X	X			X	X	X	X		

Test case	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
c_1 : Months in	M_2	M_2	M_2	M_2	M_2	M_2	M_2	M_2	M_2	M_2	M_2	M_2	M_2	M_2	M_2	M_2	M_3
c_2 : Days in	D_1	D_1	D_2	D_2	D_2	D_2	D_3	D_3	D_3	D_3	D_4	D_4	D_4	D_4	D_5	D_6	D_1
c_3 : Years in	Y_3	Y_4	Y_1	Y_2	Y_3	Y_4	Y_1	Y_2	Y_3	Y_4	Y_1	Y_2	Y_3	Y_4	-	-	Y_1
Rule count	1	1	1	1	1	1	1	1	1	1	1	1	1	1	4	4	1
a_1 : Invalid date																	
a_2 : Valid date			X	X			X	X			X	X					X
a_3 : Input out of range	X	X			X	X			X	X			X	X	X	X	

Test case	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56
c_1 : Months in	M_3	M_3	M_3	M_3	M_3	M_3	M_3	M_3	M_3	M_3	M_3	M_3	M_3	M_3	M_3	M_3	M_3	M_4	M_5
c_2 : Days in	D_1	D_1	D_1	D_2	D_2	D_2	D_2	D_3	D_3	D_3	D_3	D_4	D_4	D_4	D_4	D_5	D_6	-	-
c_3 : Years in	Y_2	Y_3	Y_4	Y_1	Y_2	Y_3	Y_4	Y_1	Y_2	Y_3	Y_4	Y_1	Y_2	Y_3	Y_4	-	-	-	-
Rule count	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	24	24
a_1 : Invalid date				X				X	X			X	X						
a_2 : Valid date	X				X														
a_3 : Input out of range		X	X			X	X			X	X			X	X	X	X	X	X

Decision tables are used in situations where an output is dependent on many input conditions. Complex relationships can also be easily represented in such tables. Every column may generate a test case. The test cases of decision Table 9.26 are given in Table 9.27.

Table 9.27 Test cases of the program of date problem

Test case	Month	Day	Year	Expected output
1	6	15	1979	Valid date
2	6	15	2000	Valid date
3	6	15	1899	Input out of range
4	6	15	2059	Input out of range
5	6	29	1979	Valid date
6	6	29	2000	Valid date
7	6	29	1899	Input out of range
8	6	29	2059	Input out of range
9	6	30	1979	Valid date
10	6	30	2000	Valid date
11	6	30	1899	Input out of range
12	6	30	2059	Input out of range
13	6	31	1979	Invalid date
14	6	31	2000	Invalid date
15	6	31	1899	Input out of range
16	6	31	2059	Input out of range
17	6	0	1979	Input out of range
18	6	32	1979	Input out of range
19	5	15	1979	Valid date
20	5	15	2000	Valid date
21	5	15	1899	Input out of range
22	5	15	2059	Input out of range
23	5	29	1979	Valid date
24	5	29	2000	Valid date
25	5	29	1899	Input out of range
26	5	29	2059	Input out of range
27	5	30	1979	Valid date
28	5	30	2000	Valid date
29	5	30	1899	Input out of range
30	5	30	2059	Input out of range
31	5	31	1979	Valid date
32	5	31	2000	Valid date
33	5	31	1899	Input out of range

(Contd.)

Table 9.27 Test cases of the program of date problem (*Contd.*)

Test case	Month	Day	Year	Expected output
34	5	31	2059	Input out of range
35	5	0	1979	Input out of range
36	5	32	1979	Input out of range
37	2	15	1979	Valid date
38	2	15	2000	Valid date
39	2	15	1899	Input out of range
40	2	15	2059	Input out of range
41	2	29	1979	Invalid date
42	2	29	2000	Valid date
43	2	29	1899	Input out of range
44	2	29	2059	Input out of range
45	2	30	1979	Invalid date
46	2	30	2000	Invalid date
47	2	30	1899	Input out of range
48	2	30	2059	Input out of range
49	2	31	1979	Invalid date
50	2	31	2000	Invalid date
51	2	31	1899	Input out of range
52	2	31	2059	Input out of range
53	2	0	1979	Input out of range
54	2	32	1979	Input out of range
55	0	0	1899	Input out of range
56	13	32	1899	Input out of range

Decision tables are effectively applicable to small-size programs. As the size increases, handling becomes difficult and time consuming. We can easily apply the decision table at the unit level. System testing and integration testing do not find its applicability in any reasonable-size program.

EXAMPLE 9.7 Consider a program to multiply and divide two numbers. The inputs may be two valid integers (say a and b) in the range of [0, 100]. Develop the decision table and generate test cases.

Solution The decision table is given in Table 9.28 and the test cases are given in Table 9.29.

Table 9.28 Decision table for a program to multiply and divide two numbers

Input in valid range?	F	T	T	T	T
$a = 0$?	-	T	T	F	F
$b = 0$?	-	T	F	T	F
Input values out of range	X				
Valid output			X		X
Output undefined		X			
Divide by zero error				X	

Table 9.29 shows the test cases generated from the above decision table.

Table 9.29 Test cases for decision table shown in Table 9.28

Test case	a	b	Expected output
1	50	–1	Input values are out of range
2	0	0	Output undefined
3	0	50	0 0
4	30	0	Divide by zero error
5	50	50	2500 1

9.5 Structural Testing

Structural testing is complementary to functional testing where the source code is considered for the design of test cases rather than specifications. We focus on the internal structure of the source code and ignore the functionality of the program. Structural testing is also called white box testing and attempts to examine the source code rigorously and thoroughly to understand it correctly. The clear and correct understanding of the source code may find complex and weak areas and test cases are generated accordingly.

9.5.1 Path Testing

It is a popular structural testing technique. The source code of the program is converted into a program graph which represents the flow of control in terms of directed graphs. The program is further converted into the decision to decision (DD) path graph. Both graphs are commonly used in structural testing techniques for the generation of test cases.

A program graph is a graphical representation of the source code where statements of the program are represented by nodes and flow of control by edges. Jorgenson (2007) has defined program graph as:

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

The program graph provides the graphical view of the program and may become the foundation of many testing techniques. The fundamental constructs of the program graph are given in Figure 9.7.

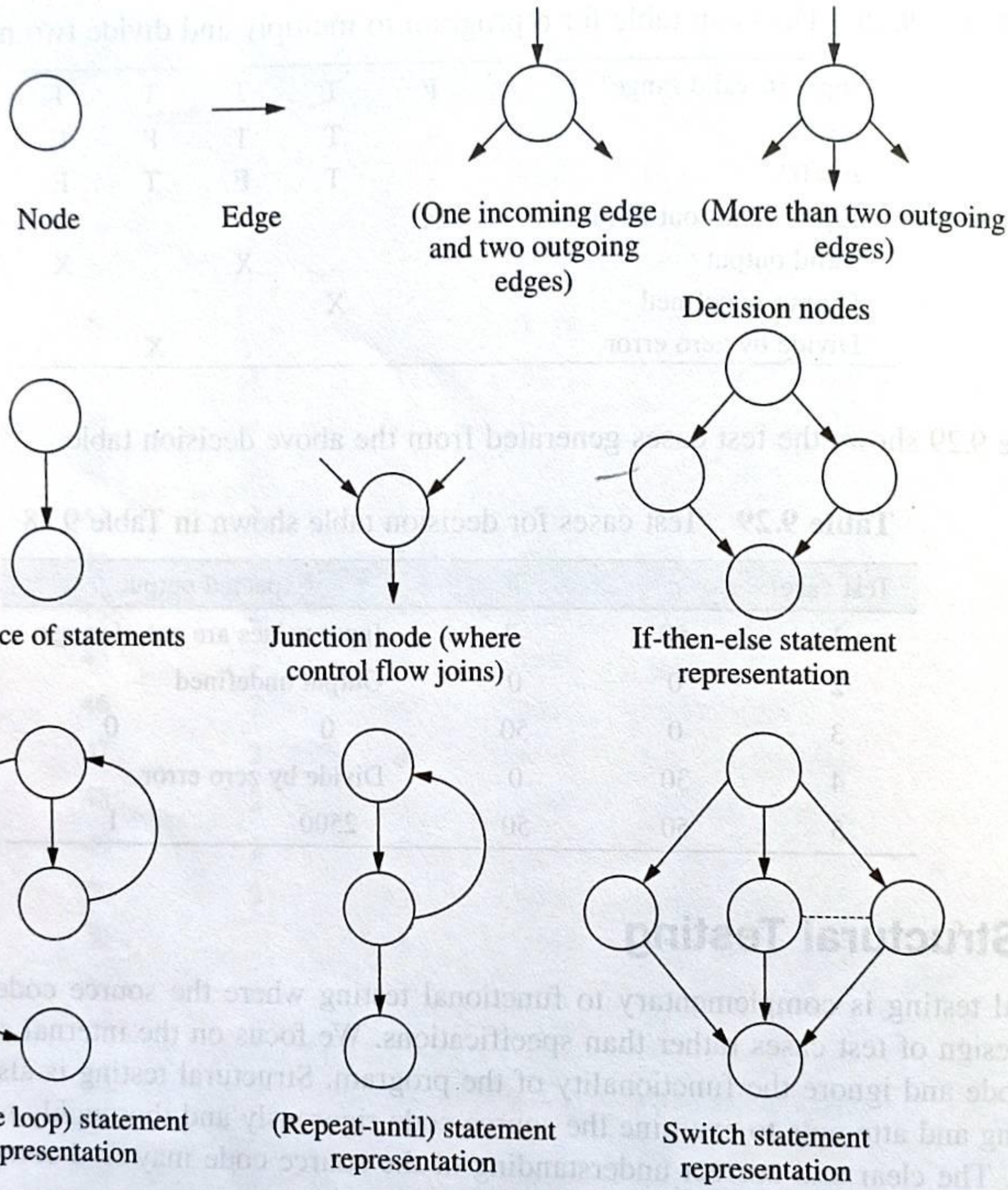

Figure 9.7 Basic constructs of a program graph.

A program can be converted into a program graph using fundamental constructs. We consider a program to determine whether a number is even or odd. The program is given in Figure 9.8 and its program graph is given in Figure 9.9. There are 16 statements in the program and hence 16 nodes in the program graph corresponding to every statement.

```
                #include<stdio.h>
                #include<conio.h>
1               void main()
2               {
3               int num;
4               clrscr() ;
5               cout<<"Enter number" ;
6               cin>>num ;
```

Figure 9.8 *(Contd.)*

```
 7      if(num%2==0)
 8      {
 9      cout<<"Number is even";
10      }
11      else
12      {
13       cout<<"Number is odd";
14       }
15      getch();
16      }
```

Figure 9.8 Program to determine whether a number is even or odd.

A path in a graph is a sequence of adjacent nodes where nodes in sequence share a common edge or sequence of adjacent pair of edges where edges in sequence share a common node. It is clear from the program graph that there are two paths in the graph.

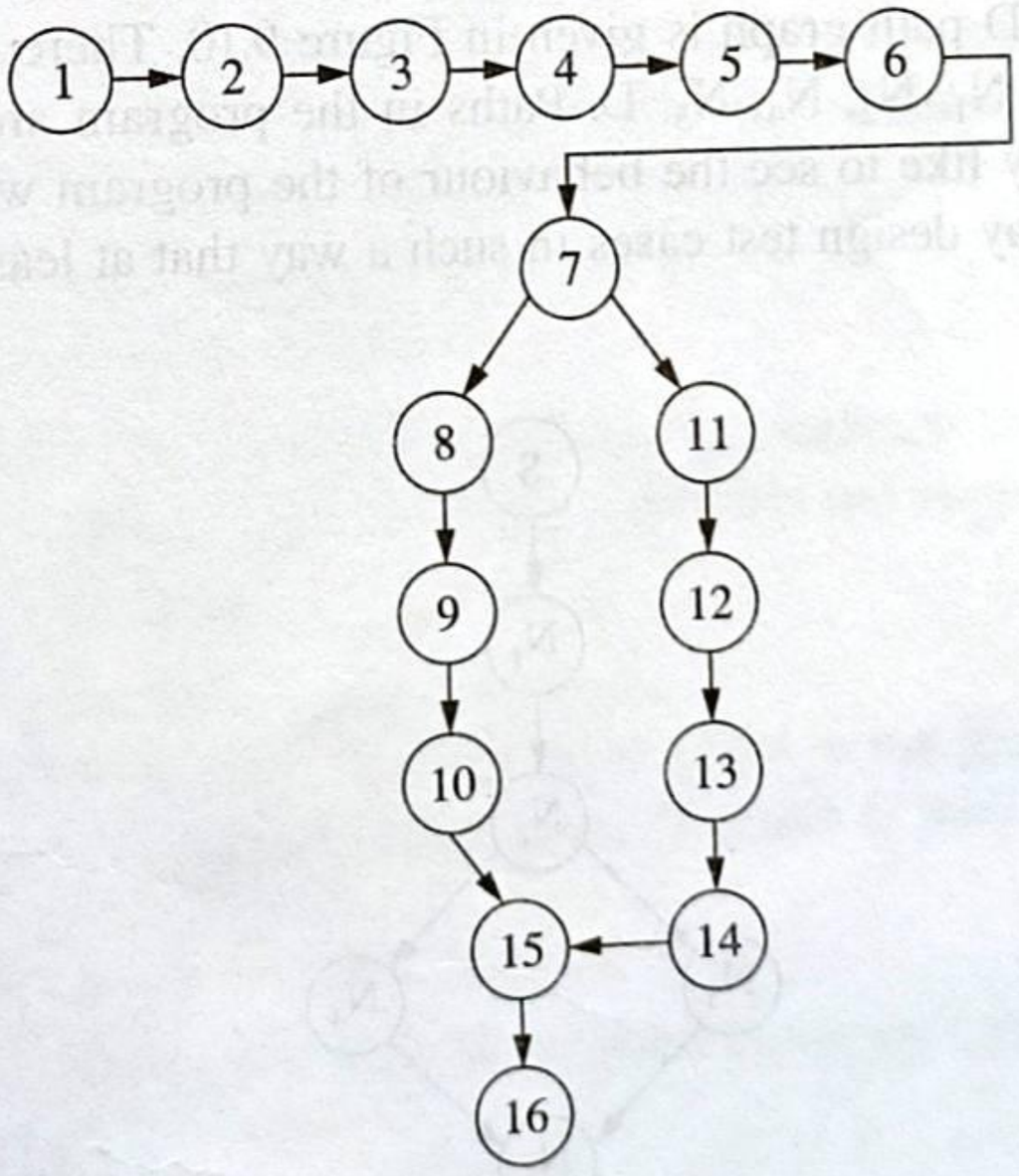

Figure 9.9 Program graph of source code given in Figure 9.8.

These paths are 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 16 and 1, 2, 3, 4, 5, 6, 7, 11, 12, 13, 14, 15, 16. Every program graph has one source node and one destination node. In the program graph given in Figure 9.9, nodes 1 to 6 are in sequence, node 7 has two outgoing edges (predicate node) and node 15 is a junction node.

A program graph can be converted into a DD program graph. There are many nodes in the program graph which are in a sequence like nodes 1 to 6 of the program graph given in Figure 9.9. When we enter into the sequence through the first node, we can exit only from the last node of the sequence. In the DD path graph, all nodes which are in a sequence are combined and represented by a single node. The DD path graph is a directed graph in which nodes are the

sequence of statements and edges are the control flow amongst the nodes. All DD path graphs have a source node and a destination node similar to the program graph. A mapping table is prepared to match the nodes of the DD path graph to the corresponding nodes of the program graph. All sequential nodes are combined into a single node which reduces the size of the DD path graph.

Mapping table of the program graph given in Figure 9.9 is presented in Table 9.30.

Table 9.30 DD path graph

S. No.	Program graph nodes	DD path graph corresponding nodes	Comments
1	1	S	Source node
2	2–6	N_1	Sequential nodes
3	7	N_2	Decision node
4	8–10	N_3	Sequential nodes
5	11–14	N_4	Sequential nodes
6	15	N_5	Junction node
7	16	D	Destination node

The corresponding DD path graph is given in Figure 9.10. There are two paths, namely, S, N_1, N_2, N_3, N_5, D and S, N_1, N_2, N_4, N_5, D. Paths in the program are identified from the DD path graph easily. We may like to see the behaviour of the program when every identified path is executed. Hence, we may design test cases in such a way that at least every identified path is executed during testing.

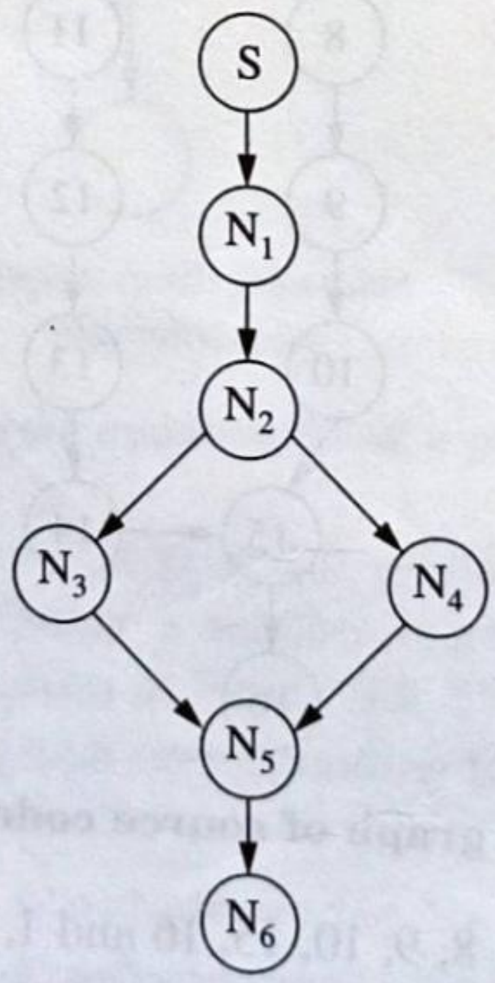

Figure 9.10 DD path graph.

Role of Independent Paths

The DD path graph is used to find independent paths. An independent path is a path through the DD path graph that introduces at least one new node or edge in its sequence from initial node to its final node. As we know, there are many paths in any program. If there are loops in the program (which is very common), the number of paths may increase and the same set of nodes

and edges may be traversed again and again. It may not be desirable to execute every path of any reasonable size program due to the repetition of the same set of statements and consumption of significant amount of time and resources. We may at least wish to execute every independent path of the DD path graph to ensure the minimum level of coverage of the source code and ascertain some confidence about the correctness of the program.

The number of independent paths can be found using a concept of graph theory which is called *cyclomatic number*. The same concept was redefined by McCabe (1976) as cyclomatic complexity for the identification of independent paths. There are three ways to calculate the cyclomatic complexity which are given below:

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

where

V(G) = cyclomatic complexity

G = graph

n = number of nodes

e = number of edges

P = number of connected components

The graph (G) is a directed graph with a single entry node and a single exit node. If this graph is a connected graph, the value of P will be 1. If there are parts of the graph, the value of P will be the number of parts of the graph.

(ii) $V(G)$ = number of regions of the program graph

(iii) $V(G)$ = number of predicate nodes (π) + 1

This is applicable only when the predicate node has two outgoing edges ("true" or "false"). If there are more than two outgoing edges, then this method is not applicable.

Properties of cyclomatic complexity:

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

The cyclomatic complexity of the DD path graph given in Figure 9.10 can be calculated as:

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

$= 7 - 7 + 2$

$= 2$

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

Hence, $V(G) = 2$

One is the inner region constituted by nodes N_2, N_3, N_4 and N_5 and the second is the outer region of the graph.

(iii) $V(G) = \pi + 1$

$= 1 + 1 = 2$

The cyclomatic complexity is 2 which is calculated by all the three methods. Hence, there are two independent paths of this graph, namely, S, N_1, N_2, N_3, N_5, D and S, N_1, N_2, N_4, N_5, D.

Cyclomatic complexity also provides some insight into the complexity of a module. McCabe (1976) proposed an upper limit of cyclomatic complexity to 10 with significant supporting evidence. We may go up to 15 in today's scenario of new programming languages, availability of CASE tools, effective validation and verification techniques, and implementation of software engineering principles and practices. If the limit further exceeds, it is advisable to redesign the module and maintain the cyclomatic complexity within prescribed limits.

Issues in Path Testing

In practice, we should test every path of the program. However, this number may be too large in most of the programs due to the presence of loops and feedback connections. We may have to set a lower objective which may be set to execute at least all independent paths. We consider the program to determine "whether a number is even or odd" (given in Figure 9.8) along with its program graph (given in Figure 9.9). We find that there are two independent paths as given below:

Path 1: S, N_1, N_2, N_3, N_5, D

Path 2: S, N_1, N_2, N_4, N_5, D

We may design test cases in such a way that both paths are executed. The test cases are given in Table 9.31.

Table 9.31 Test cases for program given in Figure 9.8

S. No.	Path ID	Paths	Inputs	Expected output
1	Path 1	S, N_1, N_2, N_3, N_5, D	6	Number is even
2	Path 2	S, N_1, N_2, N_4, N_5, D	7	Number is odd

The program graph may generate a few paths which are impossible in practice. When we give inputs, some paths may not be possible to traverse due to logic of the program. Hence, some paths are impossible to create and traverse and cannot be implemented. Path testing ensures 100% statement and branch coverage and guarantees a reasonable level of confidence about the correctness of the program.

Generation of Paths Using Activity Diagram

We may also generate paths from the activity diagram. The details of an activity diagram are available in Chapter 7 (refer to Section 7.1). Activity diagram represents the flow of activities and is similar to the program graph. It may be generated from the use cases or from the classes. We may convert a source code into its activity diagram and may find the number of independent paths. The concept of cyclomatic complexity is still applicable with slight modifications. Nodes of the program graph are represented as branches/activities/initial state/end state of an activity diagram. The edges of the program graph are represented as transitions of the activity diagram. Cyclomatic complexity is modified as

$$\text{Cyclomatic complexity} = \text{Transitions} - \text{Activities/branches} + 2P$$

The cyclomatic complexity of the activity diagram given in Figure 7.2 is calculated as:

$$\text{Cyclomatic complexity} = 8 - 8 + 2 = 2$$

There are two independent paths which is also evident from the activity diagram.

After the identification of independent paths, we design test cases to execute these paths. This ensures that every transition and every activity/branch of an activity diagram are traversed, at least once, during execution. An activity diagram provides the pictorial view of the class and helps us to identify independent paths. This technique is effectively applicable to a class of any reasonable size.

EXAMPLE 9.8 Write a program for finding the roots of a quadratic equation. Draw the activity diagram and calculate the cyclomatic complexity. Generate the test cases on the basis of cyclomatic complexity.

Solution The program for finding the roots of a quadratic equation is given below:

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

1   void main()
2   {
3   int a,b,c,validInput=0,d;
4   clrscr();
5   cout<<"Enter values of a, b & c:\n";
6   cin>>a>>b>>c;
7   if((a>=0)&&(a<=100)&&(b>=0)&&(b<=100)&&(c>=0)&&(c<=100)){
8      validInput=1;
9      if(a==0){
10     validInput=-1;
11     }
12  }
13  if(validInput==1){
14     d=b*b-4*a*c;
15     if(d==0){
16     cout<<"Equal roots";
17     }
18     else if(d>0){
19     cout<<"Real roots";
20     }
21     else{
22     cout<<"Imaginary roots";
23     }
24  }
25  else if(validInput==-1){
26     cout<<"Not quadratic";
27     }
28  else {
29     cout<<"The inputs are out of range";
30  }
31  getch();
32  }
```

The activity diagram for finding the roots of a quadratic equation is given in Figure 9.11.

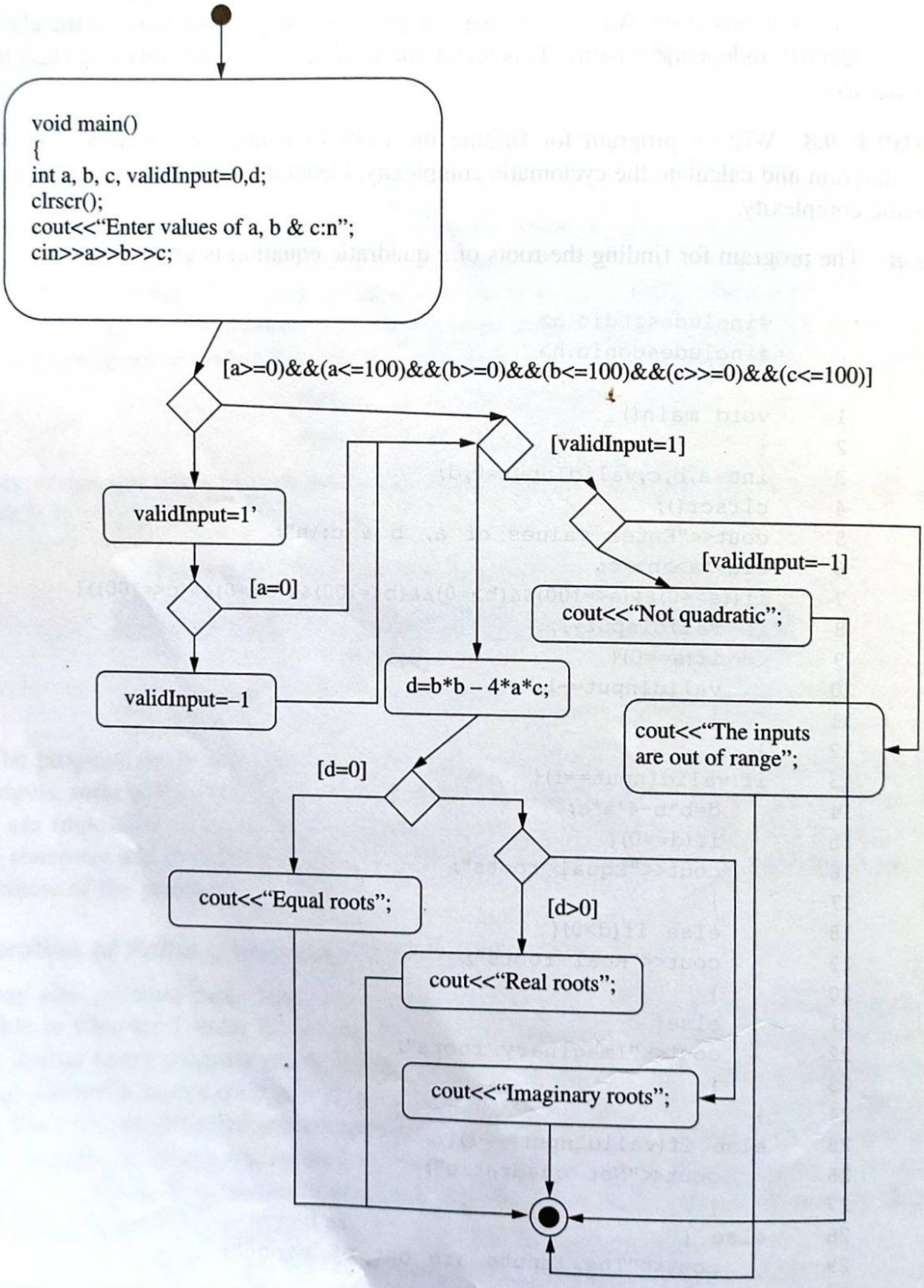

Figure 9.11 Activity diagram for finding the roots of a quadratic equation.

Cyclomatic complexity = 22 – 17 + 2 = 7

The test cases for independent paths are given in Table 9.32.

Table 9.32 Test cases for independent paths

S. No.	*a*	*b*	*c*	Expected output
1	101	50	50	Input values not in range
2	-	-	-	-
3	-	-	-	-
4	0	50	50	Not quadratic
5	99	0	0	Equal roots
6	50	50	1	Real roots
7	50	50	50	Imaginary roots

9.6 Class Testing

A class is a fundamental entity in object-oriented software development activities. Developers define the attributes and operations of a class with utmost care. Operations are also known as methods of a class. The class is treated as a unit which is similar to the module/function of conventional programming. During testing, we may like to generate test cases on the basis of structure of a class. Therefore, implementation of a class is tested with respect to its specifications. We create instances of a class for the purpose of testing and test the behaviour of such instances. The classes cannot be tested in isolation. We may have to write additional source code which is too expensive and consumes huge amount of effort. In such situations, only inspections of classes are recommended which may find faults. Classes are usually tested by their developers due to their familiarity and understandability of the source code. Developers write additional source code (stubs and drivers) and make the classes executable. We test classes with respect to their specifications. If some unspecified behaviour is implemented, we may not be able to test it. Therefore, unspecified behaviour should never be implemented. If it is necessary to implement, changes should first be made in the SRS document and subsequently in analysis and design documents. The systematic approach may help us to know the deviations and incorporate them at proper places without any confusion. Hence, a disciplined and systematic procedure may generate test cases for additional specifications and further test them accordingly.

The most popular way to design test cases is from pre- and postconditions as specified in the use cases. Every operation of a class may have pre- and postconditions. An operation (method) will be initiated only when a precondition is true. After the successful execution of the operation, one or more postconditions may be generated. In class testing, we identify pre- and postconditions for every operation of the class and establish logical relationships amongst them. We should establish logical relationships when a precondition is true and also when a precondition is false. Every logical relationship (true and false) will generate a test case and finally we may be able to test the behaviour of a class for every operation and all in also possible conditions.

We consider a class 'queue' as given in Figure 9.12 with three attributes (x, front, rear) and three operations (queue( ), insert(x), remove( )).

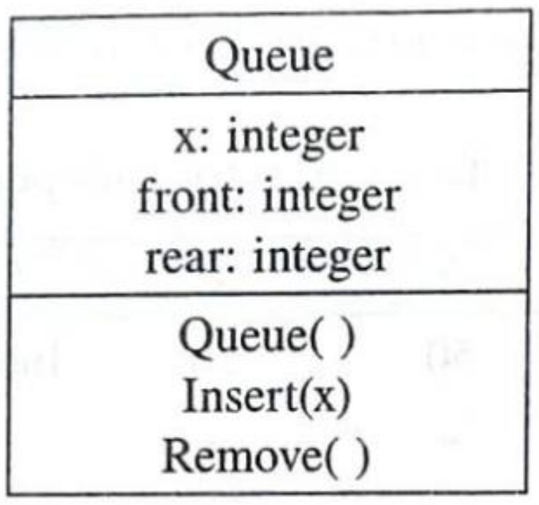

Figure 9.12 Structure of the class 'queue'.

We identify all pre- and postconditions for every operation of class 'queue'. There are three operations and their pre- and postconditions are given as:

(i) Operation Queue: The purpose of this operation is to create a queue
Queue::queue( )
(a) Pre = true
(b) Post: front = 0, rear = 0

(ii) Operation Insert(x): This operation inserts an item x into the queue.
Queue::insert(x)
(a) Pre: rear < MAX
(b) Post: rear = rear + 1

(iii) Operation Remove( ): This operation removes an item from the front side of the queue.
Queue::remove( )
(a) Pre: front > 0
(b) Post: front = front + 1

We establish logical relationships between identified pre- and postconditions for insert(x) and remove() operations.

Insert(x):

1. (precondition: rear < MAX; postcondition: rear = rear + 1)
2. (precondition: not (rear < MAX); postcondition: exception)

Similarly for remove( ) operation, the following logical relationships are established:

3. (precondition: front > 0; postcondition: front = front + 1)
4. (precondition: not (front > 0); postcondition: exception)

Test cases are generated for every established logical relationship of pre- and postconditions. Test cases for insert(x) and remove( ) operations are given in Table 9.33.

Table 9.33 Test cases for two operations of class queue

S. No.	Operation	Test input	Condition	Expected output
1	Insert(x)	23	Rear < MAX	Element '23' inserted successfully
2	Insert(x)	34	Rear = MAX	Stack overflow
3	Remove( )	-	Front > 0	23
4	Remove( )	-	Front = rear	Stack underflow

test cases. The test cases for 'queue' class are given in Table 9.35 which are based on the state transition table as given in Table 9.34.

Table 9.35 Test cases

Test case ID	Test case input		Expected result	
	Event (method)	Test condition	Action	State
1.1	New			Empty
1.2	Insert(x)			Holding
1.3	Remove( )	Front = 1	Return x	Empty
1.4	Destroy			ω
2.1	New			Empty
2.2	Insert(x)			Holding
2.3	Remove( )	Front > 1	Return x	Holding
2.4	Destroy			ω
3.1	New			Empty
3.2	Insert(x)			Holding
3.3	Insert(x)	Rear < MAX – 1		Holding
3.4	Destroy			ω
4.1	New			Empty
4.2	Insert(x)			Holding
4.3	Insert(x)	Rear = MAX – 1		Full
4.4	Remove( )			Holding
4.5	Destroy			ω
5.1	New			Empty
5.2	Insert(x)			Holding
5.3	Insert(x)	Rear = MAX – 1		Full
5.4	Destroy			ω
6.1	New			Empty
6.2	Destroy			ω

These test cases are systematic and cover every functionality of the 'queue' class. The state-based testing is simple, effective and systematically generates a good number of test cases.

9.8 Mutation Testing

Mutation testing is a useful testing technique to determine the effectiveness of an existing test suite of any program. Generally, it is applicable at the unit level, but we can also apply it at the system level without any difficulty. Why do we need such a technique? Why do we want to understand the adequacy of our test cases? As we all know, there may be a large number of test cases for any program. We normally do not execute large-size test suite completely due to

time and resource constraints. We attempt to reduce the size on the basis of some techniques. However, if we execute all test cases and do not find any fault in the program, there are the following possibilities:

(i) Test suite is effective, but there are no faults in the program.
(ii) Test suite is not effective, although there are faults in the program.

In both possibilities, we could not make our program fail, although reasons of non-failure of the program are different. In the first case, the quality of the program is good and there are no faults in the program. However, in the second case, there are faults in the program but test cases are not able to find them. Hence, mutation testing helps us to know the effectiveness of an existing test suite with reasonable level of accuracy.

9.8.1 Mutation Testing and Mutants

We prepare many copies of the program and make a change in every copy. The process of making change in the program by one or more changes is called *mutation* and the changed program is called a *mutant*. Each mutant is different than the original program by one or more changes. This change should not make the program grammatically incorrect. The mutant must be compiled and executed in the same way as the original program. A change should be a logical change in the program. We may change an arithmetic operator (+, –, *, \) or may change a Boolean relation with another one (replace > with >= or == with <=). There are many ways to make a change in the program till it is syntactically correct. Each mutant will have a unique change, which will make it different from other mutants. Consider a program to find the smallest amongst three numbers as given in Figure 9.14. The two mutants of the same program are also given in Figures 9.15 and 9.16.

```
    #include<iostream.h>
    #include<conio.h>
1   class smallest
2   {
3   private:
4   int a,b,c;
5   public:
6   void getdata()
7   {
8   cout<<"Enter first number: ";
9   cin>>a;
10  cout<<"\nEnter second number: ";
11  cin>>b;
12  cout<<"\nEnter third number: ";
13  cin>>c;
14  }
```

Figure 9.14 (*Contd.*)

```
15      void compute();
16      };
17      void smallest::compute()
18      {
19      if(a<b)
20      {
21          if(a<c)
22          {
23              cout<<"\nThe smallest number is:"<<a;
24          }
25          Else
26          {
27              cout<<"\nThe smallest number is:"<<c;
28          }
29      }
30      Else
31      {
32       if(c<b)
33       {
34            cout<<"\nThe smallest number is:"<<c;
35      }
36       Else
37       {
38            cout<<"\nThe smallest number is:"<<b;
39      }
40      }
41      }
42      void main()
43      {
44      clrscr();
45      smallest s;
46      s.getdata();
47      s.compute();
48      getch();
49      }
```

Figure 9.14 Program to find the smallest amongst three numbers.

The mutant M_1 is created by replacing the operator '<' of line number 19 by the operator '='. The mutant M_2 is created by replacing the operator '<' of line number 32 by the operator '>'. These

changes are simple and both mutants are different than the original program. Both mutants are also different than each other but are syntactically correct. The mutants are known as first-order mutants. We may also get second-order mutants by making two changes in the program and third-order mutants by making three changes and so on. The second-order and above mutants are called higher-order mutants. However, in practice, we generally use only the first-order mutants in order to simplify the process of mutation. The second-order mutants can be created by making two changes in the program as shown in Figure 9.17.

```
19  if(a<b) ← if(a=b) //mutated statement where '<' is replaced by '='
```

Figure 9.15 Mutant (M_1) of program to find the smallest amongst three numbers.

```
32  if(c<b) ← if(c>b) //mutated statement where operator '<' is
    replaced by operator '>'
```

Figure 9.16 Mutant (M_2) of program to find the smallest amongst three numbers.

```
19  if(a<b) ← if(a>b) //mutated statement where '<' is replaced
    by '>'

32  if(c<b) ← if(c=b) //mutated statement where operator '<' is
    replaced by operator '='
```

Figure 9.17 Second-order mutant of program to find the smallest amongst three numbers.

9.8.2 Mutation Operators

We use mutation operators to create mutants. Operators are used to change a grammatical expression to another expression without making the changed expression incorrect as per syntax of the implementation language. There are many ways to apply mutation operators and a large number of mutants may be created. If an expression 'a+6' is changed to 'a+10', it is considered as a lesser change as compared to an expression 'b*6', where both operator and operands are changed. The first order mutants are more popular in practice as compared to the higher-order mutants because they are easy to understand, implement, manage and control. Some of the examples of mutation operators are given as follows:

(i) access modifier change like private to public
(ii) static modifier change
(iii) Change of arithmetic operator with another one like '\' with '*' or '!+' with '–'
(iv) Change of Boolean relation with another one like '<' with '>' or '=' with '=='
(v) Delete a statement
(vi) Change of argument order
(vii) Change of any operand by a numeric value
(viii) Type case operator insertion
(ix) Type case operator deletion
(x) Change of super keyword

9.8.3 Mutation Score

We execute mutants with existing test suite and observe the behaviour. If the existing test suite is able to make a mutant fail (any observed output is different than the actual output), the mutant is treated as a killed mutant. If the test suite is not able to kill the mutant, the mutant is treated as equivalent to the original program and such mutants are called *equivalent mutants*. Equivalent mutants are also called *live mutants*. The mutation score of the test suite is calculated as:

$$\text{Mutation score} = \frac{\text{Number of mutants killed}}{\text{Total number of mutants}}$$

The total number of mutants is equal to the number of killed mutants plus the number of live mutants. A mutation score range is from 0 to 1. The higher value of a mutation score shows the effectiveness of the test suite. If the mutation score is 0, it indicates that the test suite is not able to detect any introduced fault in mutants and is useless for the program. The live mutants are important and should be studied thoroughly. Special test cases should be designed to kill live mutants. The new test cases which are able to kill live mutants should be added to the original test suite to enhance its capability. The mutation testing not only assesses the capability of the test suite but also enhances the capability by adding new test cases. Some of the popular tools are insure++, Jester for Java, nester for C++, MuJava tool, Mothra, etc.

We consider the program given in Figure 9.14 to find the smallest of three numbers. Table 9.36 shows the test suite available to test the program.

Table 9.36 Test suite to test the program given in Figure 9.14

S. No.	A	B	C	Expected output
1	3	1	5	1
2	6	2	8	2
3	4	5	7	4
4	8	6	3	3
5	9	10	6	6

Six mutants are created as per details given in Table 9.37.

Table 9.37 Mutants generated

Mutant No.	Line No.	Original line	Modified line
M_1	19	if(a<b)	if(a=b)
M_2	32	if(c<b)	if(c>b)
M_3	21	if(a<c)	if(a!=c)
M_4	32	if(c<b)	if(c=b)
M_5	19	if(a<b)	if(a>b)
M_6	32	if(c<b)	if(c<(a+b))

The actual output of all mutants with existing test suite is given in Table 9.38.

Table 9.38 Actual output of all mutants

S. No.	A	B	C	Expected output	Actual output of M_1	Actual output of M_2	Actual output of M_3	Actual output of M_4	Actual output of M_5	Actual output of M_6
1	3	1	5	1	1	5	1	1	3	1
2	6	2	8	2	2	8	2	2	6	2
3	4	5	7	4	5	4	4	4	5	4
4	8	6	3	3	3	6	3	6	3	3
5	9	10	6	6	6	6	9	6	6	6

The existing test suite kills five mutants (M_1 to M_5), but fails to kill mutant M_6. Hence, a mutation score is calculated as given below:

$$\text{Mutation score} = \frac{\text{Number of mutants killed}}{\text{Total number of mutants}}$$

$$= \frac{5}{6}$$

$$= 0.83$$

Effectiveness of a test suite is directly related to the mutation score. Mutant M_6 is live here and an additional test case is to be written to kill this mutant. The additional test case is created to kill this mutant M_6 and is given in Table 9.39.

Table 9.39 Additional test case to kill mutant M_6

S. No.	A	B	C	Expected output
6	6	4	5	4

When we execute this test case, we get the observed behaviour of the program given in Table 9.40.

Table 9.40 Observed behaviour of the program

S. No.	A	B	C	Expected output	Actual output
6	6	4	5	4	5

This additional test case is very important and must be added to the given test suite. Therefore, the revised test suite is given as in Table 9.41.

Table 9.41 Revised test suite

S. No.	A	B	C	Expected output
1	3	1	5	1
2	6	2	8	2
3	4	5	7	4
4	8	6	3	3
5	9	10	6	6
6	6	4	5	4

9.9 Levels of Testing

There are four levels in testing, namely, unit testing, integration testing, system testing and acceptance testing. Software testers are responsible for the first three levels of testing and customers are responsible for the last level (acceptance testing) of testing. Testing at each level is important and has unique advantages and challenges. At the unit level, individual units are tested using functional and/or structural testing technique. At the integration level, two or more units are combined to test the issues related to integration of units. At the system level, the complete system is tested using primarily functional testing techniques. Non-functional requirements such as reliability, testability, performance, etc. can be tested at this level only. At the acceptance level, customers test the system as per their expectations. Their testing strategy may range from ad hoc testing to well-planned systematic testing.

9.9.1 Unit Testing

There are two ways to define a unit in object-oriented testing. We may consider either each class as a unit or each operation of the class as a unit for the purpose of unit testing. How can we test a class at the unit testing level which has a parent class? The operations and attributes of the parent class will not be available which will prohibit class testing. The solution is to merge the parent class and the class under test which will make all operations and attributes available for testing. This type of merging is called *flattening of classes*. We may test such classes after flattening. After completion of testing, we should redo flattening because the final product should not have flattened classes. The issues of inheritance are to be handled carefully. If we decide to select a method as a unit, these issues will be more challenging and difficult to implement. Hence, in practice, classes are generally used as a unit at the level of unit testing.

We create instance of class (object) and pass the desired parameters to the constructor. We may also call each operation of the object with appropriate parameters and note the actual outputs. The encapsulation is important because attributes and operations are combined in a class. We focus on the encapsulated class; however, operations within the classes are the smallest available units for testing. Operations as a unit are difficult to test due to inheritance and polymorphism. In unit testing, generally, classes are treated as unit, and functional and structural testing techniques are equally applicable. Verification techniques such as peer reviews,

inspections and walkthroughs are easily applicable and may find a good number of faults. State-based testing, path testing, class testing, boundary value analysis, equivalence class and decision table-based testing techniques are also applicable.

9.9.2 Integration Testing

The objective of integration testing is to test the various combinations of different units and to check that they are working together properly. We do not have hierarchical control structure in object-oriented systems. Hence, conventional integration testing techniques such as top down, bottom up and sandwitch integration may not be applicable in their true sense. The meaning of integration testing is basically interclass testing. There are three ways to carry out interclass testing. The most popular interclass testing is thread-based testing. In the thread-based testing, we integrate classes that are required to respond to an input given to the system. When the input is given to the software, one or more classes are needed for execution, and such classes make a thread. There may be many such threads depending on the inputs. The expected output of every thread is calculated and is compared with the actual output. This technique is simple and easy to implement.

The second technique is the use case-based testing. We test every basic and alternative paths of a use case. A path may require one or more classes for execution. Every use case scenario (path) is tested and due to involvement of many classes, interclass issues are automatically tested. The third technique is the cluster testing where classes are combined to show one collaboration. In all approaches, classes are combined on the basic of logic and then executed to know the outcome. The most popular and simple technique is the thread-based testing.

9.9.3 System Testing

The system testing is performed after the unit testing and integration testing. The complete software within its expected environment is tested. We define a system as a combination of software, hardware and other associated parts which work together to provide the desired functionality. All functional testing techniques are effectively applicable. Structural testing techniques may also be used technically but they are not very common due to the large size of the software. Verification techniques are normally used for reviewing the source code and documents. We test the functional requirements of the software under stated conditions. This is the only level where non-functional requirements such as stress, load, reliability, usability and performance are tested. A good number of testing tools are available to test the functional and non-functional requirements.

We may like to ensure a reasonable level of correctness of the software before delivering it to the customer. Whenever the source code is modified to remove an error, an impact analysis of this modification is performed. If any error is not possible to remove due to lack of time or is technically not possible in the present design, the best way is to document the error as a limitation of the system. We would like to test every stated functionality of the software, keeping in mind, the customer's expectations. After completion of the system testing, the software is ready for customers.

9.9.4 Acceptance Testing

The acceptance testing is carried out by the customer(s) or their authorized persons for the purpose of accepting the software. The place of testing may be the developer's site or the customer's site depending on the mutual agreement. In practice, generally, it is carried out at the customer's site. Customers may like to test the software as per their expectations. They may do it in an ad hoc way or in a well-planned systematic way in order to establish the confidence about the correctness of the software.

When a software product is developed for anonymous customers (in case of operating systems, compilers, CASE tools, etc.), potential customers are identified to use the software as per their expectations. If they use the software at the developer's site under the supervision of the developers, it is known as *alpha testing*. Another approach is to distribute the software to potential customers and ask them to use at their site in a free and independent environment and this is known as *beta testing*. The purpose of the acceptance testing is to test the software with an intention to accept it after getting reasonable confidence about its usage and correctness.

9.10 Software Testing Tools

Software testing tools are available for various applications. They help us to design and execute test cases, analyse the program complexity, identify the non-coverage area of the source code and ascertain the performance of the software. There are numerous similar applications during testing which may be improved using a testing tool. The whole process of testing a software may be automated and carried out without human involvement. Software tools are also very effective for repeated testing where similar data set is to be given again and again. Many non-functional requirements such as performance under load, efficiency, reliability and extreme stress conditions are also tested using software testing tools. Broadly, these tools may be partitioned into three categories—static, dynamic and process management. Most of the tools may fall into any one of the categories and have specified scope with predefined applications.

9.10.1 Static Testing Tools

Static testing tools analyse the program without executing it. They may calculate program complexity and also identify those portions of the program which are hard to test and maintain. These tools may find a good number of faults prior to the execution of the program. The identified faults may range from logical faults to syntax faults and may include non-declaration of a variable, double declaration of a variable, divide by zero issue, unspecified inputs, etc. Some tools may also examine the implementation of good programming guidelines and practices and highlight the violations, if any, to improve the quality of the program. Many tools which calculate the metrics are static analysis tools. Some of the popular tools are CMTJava (Complexity Measures Tool for Java), Jenssoft's Code Companion, Sun Microsystems' JavaPureCheck, ParaSoft's Jtest, Rational Purify, CMT++ (Complexity Measures Tool for C++), ParaSoft CodeWizard, ObjectSoftware's ObjectDetail, Software Research's STATIC (Syntac and Semantic Analysis Tool), Eastern System's TestBed, McCabe QA, etc.

9.10.2 Dynamic Testing Tools

Dynamic testing tools execute the programs for specified inputs. The observed output is compared with the expected output and if they are different, the program is considered in a failure condition. These tools also analyse the programs and the reasons of such failure may also be found. Dynamic testing tools are also very effective to test the non-functional requirements such as performance, reliability, efficiency and portability.

Performance Testing Tools

These tools are used to test the performance of the software under stress and load. The performance testing is also called stress and load testing. Popular tools are Mercury Interactive's Load Runner, Apache JMeter, Rational's Performance Tester, Compuware's QALOAD, Auto Tester's Autocontroller, Quest Software's Benchmark Factory, Sun Microsystems' Java Load, Minq Software's PureLoad, Rational Software's Test Studio, etc. These tools generate heavy load on the system to test under extreme conditions.

Functional/Regression Testing Tools

These tools are used to test the functionality of the software. They may generate test cases and execute them without human involvement. In regression testing, the software is retested after modifications. Most of the tools are common for both groups. Some of the popular tools are Junit, Test Manager, Rational's Robot, Mercury Interactive's Win Runner, Compuware's QA Centre, Segue Software's Silktest, AutoTester for Windows, Qronus Interactive's TestRunner, Automated QA's AQtest, Rational's Visual Test, etc.

Coverage Analysis Tools

These tools are used to provide an idea about the level of coverage of the program. They also indicate the effectiveness of the test cases. They may also highlight the untested portion of the program which may help us to design special test cases for the coverage of that portion of the program. Some popular source code coverage analysis tools are Rational's Pure Coverage, Quality Checked Software's Cantata++, CentreLine Software's QC/coverage, Vision Soft's Vision Soft, Plum Hall's SQS, etc.

Some popular test coverage analysis tools are Software Research's TCAT for Java, IBM's Visual Test Coverage, Bulseye Testing Technology's C-Cover, Testwell's CTC++, Testing Foundation's GCT, Parasoft's TCA, Software Research's TCAT C/C++, and McCabe's Visual Testing Tool Set.

9.10.3 Process Management Tools

The focus of process management tools is to improve the testing processes. They may help us to allocate resources, prepare test plan and keep track of the status of testing. Some of the popular tools are IBM Rational Test Manager, Mercury Interactive's Test Director, Segue Software's Silk Plan Pro, Compuware's QA Director, etc. Some configuration management tools are IBM Rational Software clear DDTs Bugzilla, Samba's Jitterbug. A few test management tools are

Vector Software's Vector CAST, Auto Tester's Auto Advisor, Silver Mark's Test Mentor, Test Master's TMS and TOOTSIE.

A software reliability measurement tool is used to estimate the reliability of the software. It may also calculate the time needed to achieve an objective failure intensity. A popular tool is SoftRel's WhenToStop. Software testing tools not only reduce the testing effort but also make testing a pleasant discipline. Moreover, some non-functional requirements (such as performance and efficiency) cannot be tested without using a software testing tool.

Review Questions

1. What is software testing? Discuss issues, limitations, practices and future of software testing.
2. "Testing is not related to only one phase of software development life cycle". Comment on the correctness of this statement.
3. Differentiate between verification and validation. Which one is more important and why?
4. Which is the most popular verification technique? Explain with suitable examples.
5. Write short notes on the following verification techniques:
   (a) Peer reviews
   (b) Walkthroughs
   (c) Inspections
6. Design an SRS document verification checklist. Highlight some important issues which such a checklist must address.
7. What is a checklist? Discuss its significance and role in software testing.
8. Differentiate between walkthroughs and inspections. List the advantages and disadvantages of both techniques.
9. Design checklists for OOA document and OOD document. Both checklists should cover important issues of both documents.
10. Establish the relationship between verification, validation and testing along with suitable examples.
11. What is functional testing? Discuss any one technique with the help of an example.
12. What is boundary value analysis? What are its extensions? List the advantages and limitations of each extension.
13. Consider a program that determines the previous date and the next date. Its inputs are a triple of day, month and year with its values in the following range:

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

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

    $1947 \leq \text{year} \leq 2011$

    The possible outputs are "next date", "previous date" or "invalid input". Design boundary value analysis test cases, robust test cases, worst test cases and robust worst test cases.

14. Consider a program that calculates median of three numbers. Its input is a triple of positive integers (say a, b and c) and values are from the interval [1, 1000]. Generate boundary value and robust test cases.
15. Describe the equivalence class testing technique. How is it different from boundary value analysis?
16. What is decision table-based testing? What is the role of a rule count? Discuss the concept with an example.
17. Consider a program to sort a list in ascending order. Write test cases using equivalence class testing and decision table-based testing.
18. What are the limitations of boundary value analysis? Highlight the situations in which it is not effective.
19. Consider a program that counts the number of digits in a number. The range of input is given as [1, 1000]. Design the boundary value analysis and equivalence class test cases.
20. What is structural testing? How is it different from functional testing?
21. What is path testing? How can we make it more effective and useful?
22. Show that a very high level of statement coverage does not guarantee a defect-free program. Give an example to explain the concept.
23. What is mutation testing? What is the significance of mutation score? Why are higher-order mutants not preferred?
24. Why is mutation testing becoming popular? List some significant advantages and limitations.
25. What is a program graph? How can it be used in path testing?
26. Define the following:
    (a) Program graph
    (b) DD path graph
    (c) Mapping table
    (d) Independent paths
27. What is class testing? Why should unspecified behaviour not be implemented?
28. What is state-based testing? What are alpha and omega states? List the advantages and limitations of the state-based testing.
29. What are levels of testing? Which level is most popular and why?
30. Explain the concept of flattening of classes. How can it help in unit testing?
31. Discuss the importance of software testing tools. List some static, dynamic and process management tools.
32. Differentiate between static and dynamic testing tools. Which one is more effective and why?
33. "Some non-functional requirements cannot be tested without using a software testing tools". Comment on this statement and justify with an example.

10.3.3 Poor Documentation and Manuals

Software maintenance activities can be performed effectively with the help of proper documentation and operating procedure manuals. They help in understanding the software and also help to identify the reasons of software failures. The reasons may be present in SRS document, SDD document or in the source code itself. After the identification of reason(s), a corrective action is required to make the software operational. In the case of addition of new functionality, documents provide foundations for design of interfaces and ways to proper sharing of data amongst new and old classes.

10.3.4 Inadequate Budgetary Provisions

Maintenance activities are performed under time and resource constraints. Generally, adequate budgetary provisions are not made for such activities. Software costing models are not giving due importance to maintenance activities. Maintenance effort cannot be imagined and estimated at the time of software costing. We may ask additional cost for new functionality but many times customers are not willing to pay requested cost happily. The conflict between the customer and the company may make the maintenance phase more challenging. The problems may be minimized if adequate budgetary provisions are made at the time of costing of the software product.

10.3.5 Emergency Fixing of Bugs

There is always an emergency to restore the operations of the software after its failure. The bugs are identified and corrective actions are performed. We may like to retest the software in order to ensure that corrections are rightly implemented and have not affected other parts of the software. This retesting (which is also called regression testing) is generally performed in emergency due to serious shortage of time. In reality, modified portion of the software is given with inadequate testing. This emergency fixing of bugs is also known as *patching*. Patching deteriorates the quality and structure of the software and may make the software more vulnerable. Regression testing should be performed effectively to maintain the expected standards of the software. If we do not do so, we are playing with the quality of the software. This practice of fixing bugs under emergency conditions resulting in inadequate testing is very common, which makes the maintenance phase much more challenging and difficult.

10.4 Maintenance of Object-Oriented Software

One of the reasons of the popularity of object-oriented paradigm is that such software products are easy to maintain and help to reduce maintenance effort. The fundamental entity is an object which is an independent unit of the software. Implementation details of an object are not visible to the outside world due to information hiding. An operation of an object is carried out after receiving a message which is the only allowed way of communication. When we make a change in an object, it will not have any impact outside that object due to information hiding. With this concept, we are able to minimize the scope of a change, which further makes it easy to maintain such a software product. A change will not affect the other portions of the source code

which are outside the modified object. Physical and conceptual independence of objects may help to identify that portions of the source code which are to be modified to achieve a specific maintenance purpose.

Inheritance and dynamic binding are two important characteristics of object-oriented software development and are commonly used in practice. With all their great advantages (like software reuse), these characteristics complicate the operations and also reduce the understandability of the program. Poor understanding may further make the maintenance tasks difficult and challenging.

Inheritance promotes the distributed class descriptions. A complete description of a class can be obtained by viewing not only the class but also its superclasses. Classes are generally described at different places in the program and there is no single place where a developer can turn to get a complete description of a class. This dimension reduces the understandability and a developer has to spend a substantial amount of time to search through various class descriptions to find the desired information. It is also required to study all the parts in sufficient detail to understand the program. In some cases, maintenance effort may not reduce significantly as compared to procedural software development. For example, consider the class hierarchy given in Figure 10.2. The student class inherits the person class, the examination class inherits the student class and the result class inherits the examination class. In order to understand the result class, a developer in the maintenance phase will have to study all the classes of the inheritance tree.

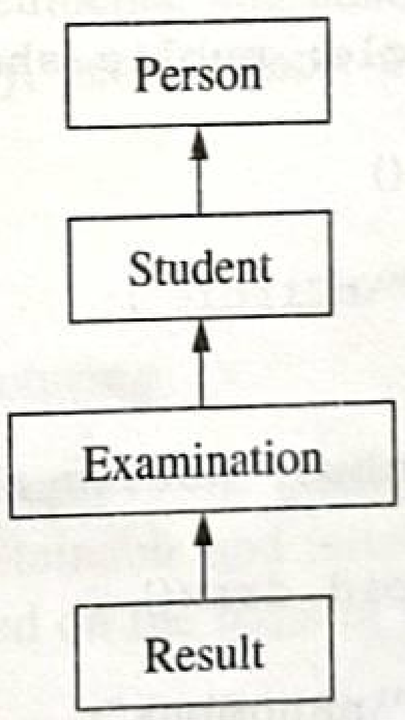

Figure 10.2 Inheritance hierarchy.

Dynamic binding provides many facilities (like flexibility) in object-oriented software development, but it may make it difficult to trace the dependencies. The difficulty in tracing has an adverse effect on the understandability of the program and on the calculation for the severity of impact of any change in the program. Dynamic binding can be achieved through run-time polymorphism. In run-time polymorphism, the objects of the polymorphic class change and respond in a different manner at run-time for the same message. Run-time polymorphism can be a problem during the maintenance phase. Consider the example shown in Figure 10.3. The base class is the shapes class and has four subclasses—line, square, circle and rhombus.

In the base class shapes, draw is a virtual function. Thus, all the four subclasses must implement their own draw function. With the absence of virtual functions, all the outputs, no matter the object of the shapes class contains reference to any of the subclasses, would print

```
.....
class shapes
{:
public:
    virtual void draw()
{
    cout<<"Point";
}
};
class Line: public shapes
{
void draw()
{
    cout<< "\nLine";
}
};
class Square: public shapes
{
void draw()
{
    cout<< "\nSquare";
}
};
class Circle: public shapes
{
void draw()
{
    cout<<"\nCircle";
}
};
class Rhombus: public shapes
{
virtual void draw()
{
    cout<<"\nRhombus";
}
};
.....
```

Figure 10.3 Example to demonstrate run-time polymorphism.

'points' because all the calls to draw will refer to the function draw in the base class. However, with the use of virtual functions, the object of the base class shape will call the corresponding object which it refers to. The consequence of the run-time polymorphism will be shown in the maintenance phase. The object of the shapes class is declared as follows:

```
Shapes *obj;
obj → draw( );
```

If the testing person has to test obj → draw( ), then he/she has to understand what would be the output if the object obj consists of reference to any of the four subclasses—line, square, circle

and rhombus. The trace through the source code is the only solution to understand the run-time polymorphism either by debugging or manually.

In the inheritance hierarchy, if there is a change in any of the base classes, all the classes dependent on the base class will have to be analysed and retested. Hence, although inheritance and dynamic binding have many advantages during software development, they also make the life of the developer difficult in the software maintenance phase due to poor understanding of the program and complicated dependencies. Error handling mechanism helps in constructing a robust system and reduces the cost to failure. In object-oriented languages such as C++ and Java, this error handling mechanism is called *exception handling*. Exception handling is the method of building a system to detect and recover from exceptional conditions. This mechanism may be helpful in detecting the exceptions in the maintenance phase.

10.5 Software Rejuvenation

It is not easy to maintain a software product for a long period of time due to continuous changes in the source code and associated documents. All changes should be done in a systematic way and at all desired places of the software. Associated documents should also be modified in order to make them relevant and meaningful after the changes. Software rejuvenation addresses all those activities which are carried out to enhance the quality of the existing system. This may include modifications in the documents, upgradations of software interfaces, restructuring of source code to improve efficiency, performance, usefulness, etc. Some of the activities of software rejuvenation are given as:

(i) Reverse engineering
(ii) Software re-engineering
(iii) Redocumentation and restructuring

The purpose of such activities is to improve the quality of the software system and preserve its value. The output may be a more sustainable and reliable software system and in some cases, a new software system will be produced on the basis of the existing requirements.

10.5.1 Reverse Engineering

There are situations in practice when the only documentation available for maintenance is the source code. These situations are really very difficult from the perspective of maintaining such software product. One of the solutions is to understand the source code thoroughly and prepare a design document and also a specification document. This process of creating the design document from the source code and the specification document from the design document is called *reverse engineering*. Hence, the journey of creating documents from source code to specifications (in parts or complete) is called reverse engineering and is shown in Figure 10.4.

In software development life cycle, we start from the specification phase and go through the analysis phase, design phase and implementation phase. This journey of software development life cycle is called *forward engineering*.

The change control board (CCB) is responsible for approving and monitoring the changes. All the changes must be carefully reviewed by this board before approval. After the changes have been made, these changes must be notified so that they are included in the software library. The software library is a repository where the entire products produced during the software development life cycle are kept. The librarian is responsible for keeping all the current issues of each software component. Once the changes have been made, implemented and deployed, then these changes are officially notified using software change notice. The software change notice informs the librarian of a change. The format of the software change notice is given in Table 10.1.

Table 10.1 Format of software change notice

Module in which change is made:	
Identifier of item in which change is made#:	
Change approved by:	
Date and time of the change approval:	
Change implemented by:	
Date and time of the change implementation:	
Signature:	

The changes may include defect corrections, quality improvement or any enhancement in the software product.

10.7.3 Configuration Accounting

The responsibility of configuration accounting is to keep track of all the activities including changes and any action taken that affects the configuration of the product. Configuration accounting is responsible for keeping all the data corresponding to the change. The parts affected by the change must also be tracked and monitored so that all the identifiers of the affected components can be updated after the changes have been made. The multiple instances of any software item may be present, and configuration accounting should ensure that the correct instance is called each time.

10.8 Regression Testing

When the software is developed, development testing is carried out in order to increase the confidence in the correctness and reliability of the software. The development testing involves construction of test plans, development of test design and procedure, and design and execution of test cases that produce test results. When the software is modified in the maintenance phase, it requires retesting to assure its continued operation. This process of retesting is known as *regression testing*. Regression testing is defined as the process of testing the modified parts of the software and ensures that no new faults have occurred in the previously developed source code due to these modifications. Thus, regression testing not only tests the modified part of the

software, but also checks the parts affected by the change. Hence, regression testing is done to ensure that:

1. There are no faults in the other parts of the software due to modifications.
2. Correctness of the software.
3. Quality and reliability of the software.
4. Confidence in the modified parts.

The existing test suite may be used in regression testing. However, if any new portions are added in the source code, then an additional regression test case must be developed and an updated test suite should be prepared which can be used in any future modification. The difference found from the expected results must be addressed and the required corrective action must be made.

Regression testing is not only a part of maintenance activity, it may also be performed during the later stages of software development cycle. This later phase begins after the completion of the construction of test plans and test suites and start of initial testing of the software. During this phase, we test the source code with the aim to correct faults and hence is similar to the maintenance activity. The differences between development and regression testing are given in Table 10.2.

Table 10.2 Development testing versus regression testing

S. No.	Development testing	Regression testing
1	Test plans are prepared and test suites are constructed.	The test suites including the test cases are already available. Some additional test cases may be added.
2	The complete software is tested.	Only the modified parts of the software and the other portion of the software affected by these modifications are tested.
3	It is carried out once in the entire lifetime of the software.	It may occur many times during the lifetime of the software.
4	Budget and time are allocated separately.	Generally, no separate budget and time are allocated.
5	It is performed under the release date pressure	It is performed under greater pressure and time constraints.

The process of regression testing is very costly and involves the following steps:

1. Analysis of failure report
2. Debugging of the source code
3. Identification of fault(s) in the source code
4. Source code modification (new and old programs will be different)
5. Selection of test cases from existing test suite to ensure the correctness of modification(s)
6. Addition of new test cases, if required
7. Performing retesting to ensure correctness using selected test cases and new test cases, if any

Regression testing may be performed effectively within specified time, if existing test cases are prioritized according to some criteria. Many techniques and algorithms are available in literature

with proclaimed effectiveness and efficiency for prioritizing the test cases. However, applications of such techniques are limited to specific projects under stated conditions. Some software tools are also available which may assist and help during the process of regression testing.

Review Questions

1. What is software maintenance? Explain the different categories of maintenance.
2. What are the challenges involved in maintaining an object-oriented system? Explain with the help of an example.
3. Discuss the various problems during maintenance of object-oriented software? Describe some solutions to these problems.
4. Explain the significance of maintenance. Which category of maintenance consumes maximum effort and why?
5. What are the challenges of software maintenance?
6. Inheritance and polymorphism are two major constructs of object-oriented software. How do these characteristics increase complications in the maintenance phase of software development? How can these complications be reduced?
7. What is software rejuvenation? What are the activities involved in software rejuvenation?
8. What is regression testing? How is it different than development testing?
9. What is reverse engineering? List the tools for reverse engineering.
10. What is re-engineering? Explain the process of re-engineering.
11. Annual charge traffic in a software system is 30% per year. The initial development cost was ₹ 25 lacs. Total lifetime for the software is 12 years. What is the total cost of the software system?
12. How can maintenance effort be computed using Belady–Lehman's model?
13. The development effort of a software project is 2000 person-months. The value of K for the project is 0.8. The complexity of the source code is 10. Find the maintenance effort if
    (i) the maintenance team has very good understanding of the project ($d = 0.6$).
    (ii) the maintenance team has poor understanding of the project ($d = 0.4$).
14. Describe Boehm's model of software maintenance. What is the role of effort adjustment factor?
15. What is configuration management? Explain the various activities of configuration management.
16. What is the role of change control board in configuration management?
17. Explain the typical change management procedure with the help of a block diagram.
18. (a) What are the various activities followed in configuration management? Explain them in detail.
    (b) What is the use of software libraries?