Software Testing

Software Testing

- What is Testing?

Many people understand many definitions of testing **:**

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

These definitions are incorrect.

Software Testing

A more appropriate definition is:

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

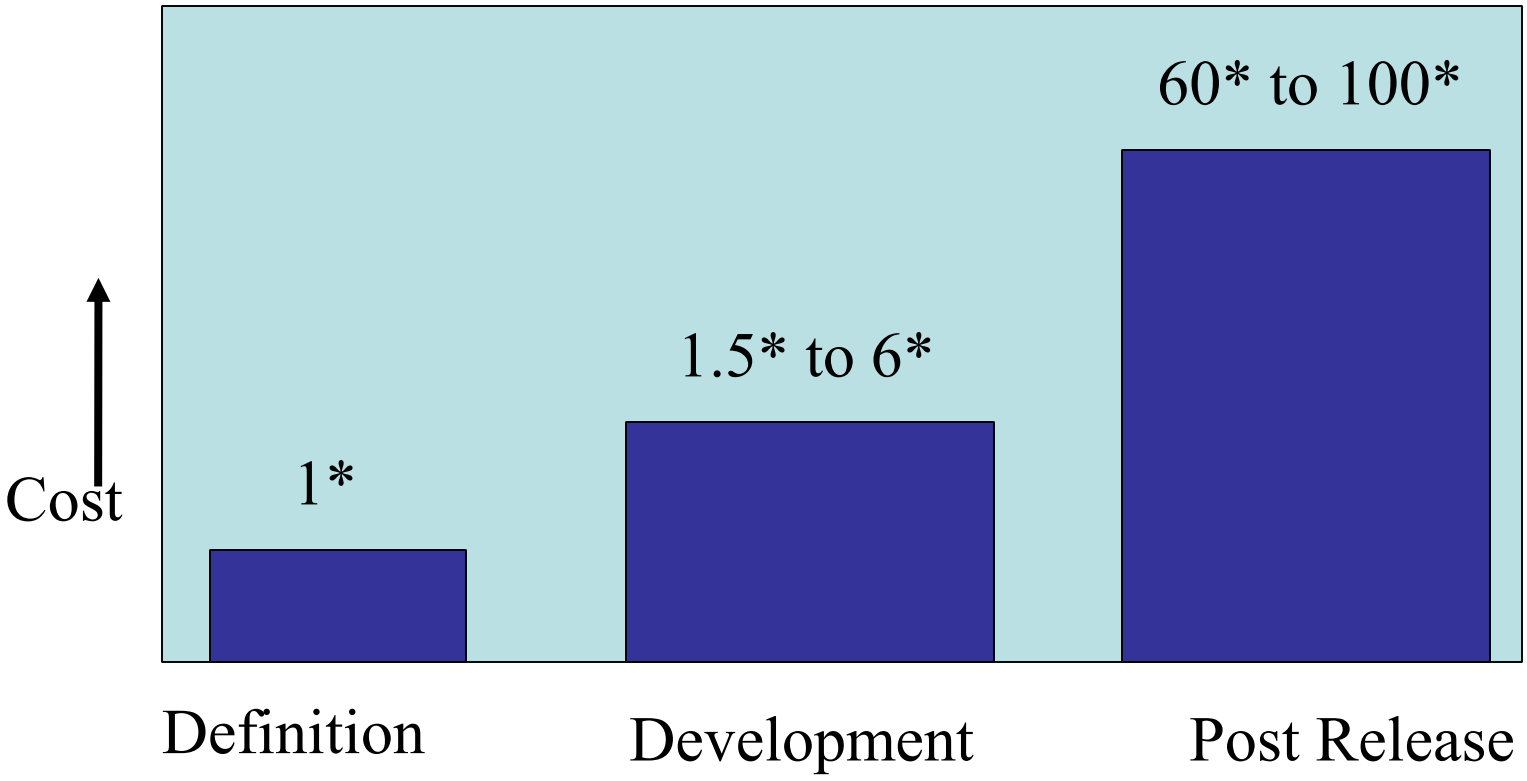
Software Testing

- Why should We Test ?

Although software testing is itself an expensive activity, yet launching of software without testing may lead to cost potentially much higher than that of testing, specially in systems where human safety is involved.

In the software life cycle the earlier the errors are discovered and removed, the lower is the cost of their removal.

Cost to fix faults



Software Testing

- Who should Do the Testing ?
  - Testing requires the developers to find errors from their software.
  - It is difficult for software developer to point out errors from own creations.
  - Many organisations have made a distinction between development and testing phase by making different people responsible for each phase.

- What should We Test ?

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

Some Terminologies

➢ Error, Mistake, Bug, Fault and Failure

People make **errors**. A good synonym is **mistake**. This may be a syntax error or misunderstanding of specifications. Sometimes, there are logical errors.

When developers make mistakes while coding, we call these mistakes "**bugs**".

A **fault** is the representation of an error, where representation is the mode of expression, such as narrative text, data flow diagrams, ER diagrams, source code etc. Defect is a good synonym for fault.

A **failure** occurs when a fault executes. A particular fault may cause different failures, depending on how it has been exercised.

➢ Test, Test Case and Test Suite

Test and **Test case** terms are used interchangeably. In practice, both are same and are treated as synonyms. Test case describes an input description and an expected output description.

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

Fig. 2: Test case template

The set of test cases is called a **test suite**. Hence any combination of test cases may generate a test suite.

Software Testing

- Verification and Validation

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

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 .

Testing= Verification+Validation

Software Testing

➢ Alpha, Beta and Acceptance Testing

The term **Acceptance Testing** is used when the software is developed for a specific customer. A series of tests are conducted to enable the customer to validate all requirements. These tests are conducted by the end user / customer and may range from adhoc tests to well planned systematic series of tests.

The terms alpha and beta testing are used when the software is developed as a product for anonymous customers.

Alpha Tests are conducted at the developer's site by some potential customers. These tests are conducted in a controlled environment. Alpha testing may be started when formal testing process is near completion.

Beta Tests are conducted by the customers / end users at their sites. Unlike alpha testing, developer is not present here. Beta testing is conducted in a real environment that cannot be controlled by the developer.

Regression Testing

Test modified versions of a previously validated system. Usually done by testers. The goal is to assure that changes to the system have not introduced errors (caused the system to regress).

The primary issue is how to choose an effective regression test suite from existing, previously run test cases.

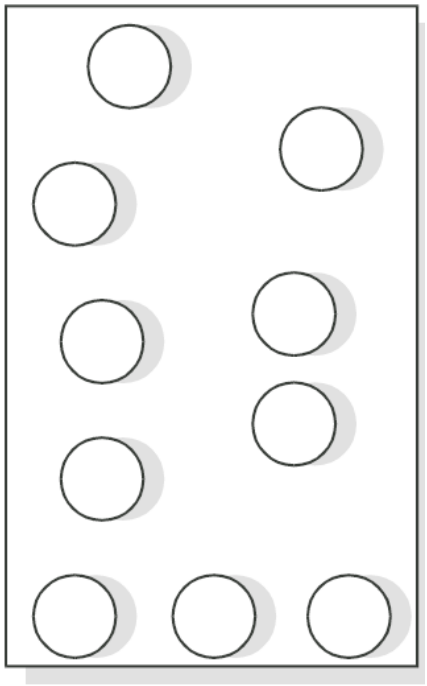
Levels of Testing

There are 3 levels of testing:

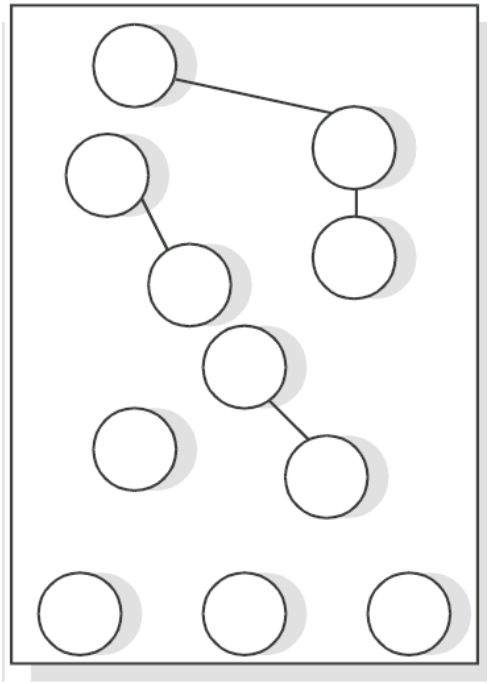
i. Unit Testing

ii. Integration Testing

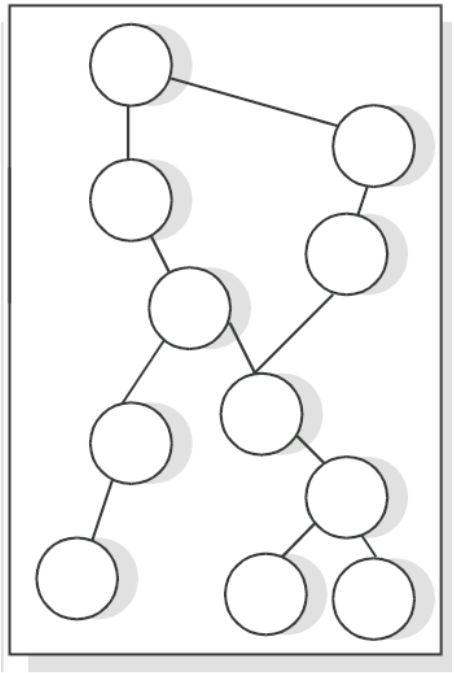
iii. System Testing



UNIT TESTING



INTEGRATION TESTING



SYSTEM TESTING

Unit Testing

There are number of reasons in support of unit testing than testing the entire product.

1. The size of a single module is small enough that we can locate an error fairly easily.

2. The module is small enough that we can attempt to test it in some demonstrably exhaustive fashion.

3. Confusing interactions of multiple errors in widely different parts of the software are eliminated.

Software Testing

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

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

This overhead code, called scaffolding represents effort that is import for testing, but does not appear in the delivered product as shown in Fig. 29.

Software Testing

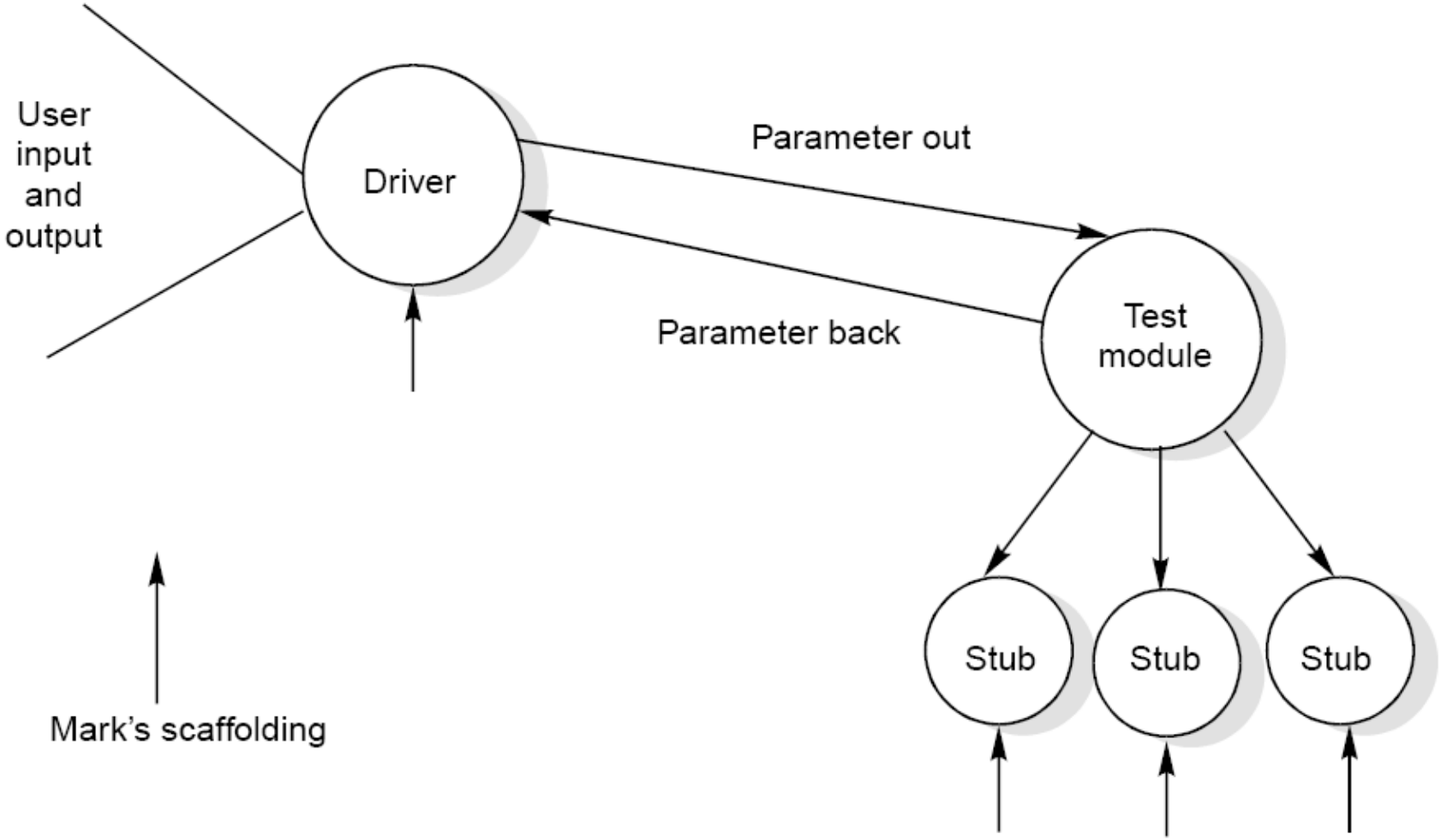
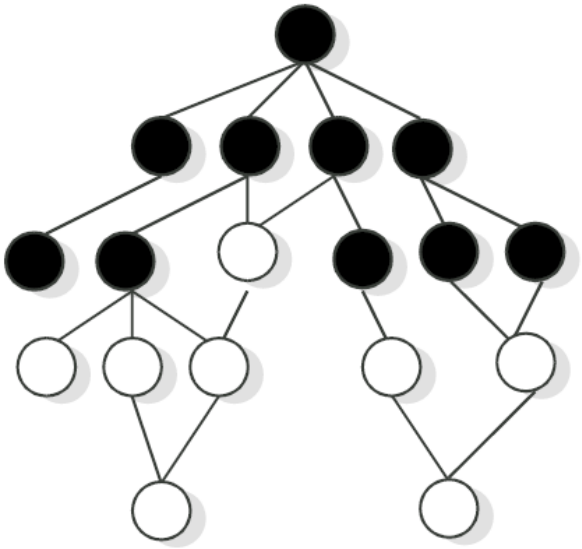


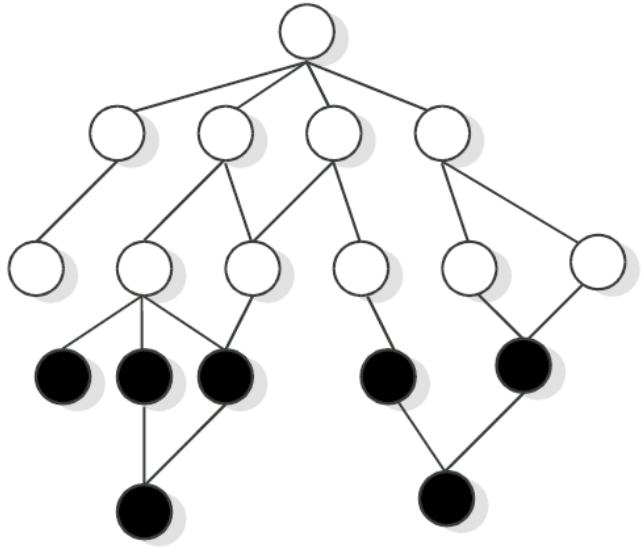
Fig. 29 : Scaffolding required testing a program unit (module)

Integration Testing

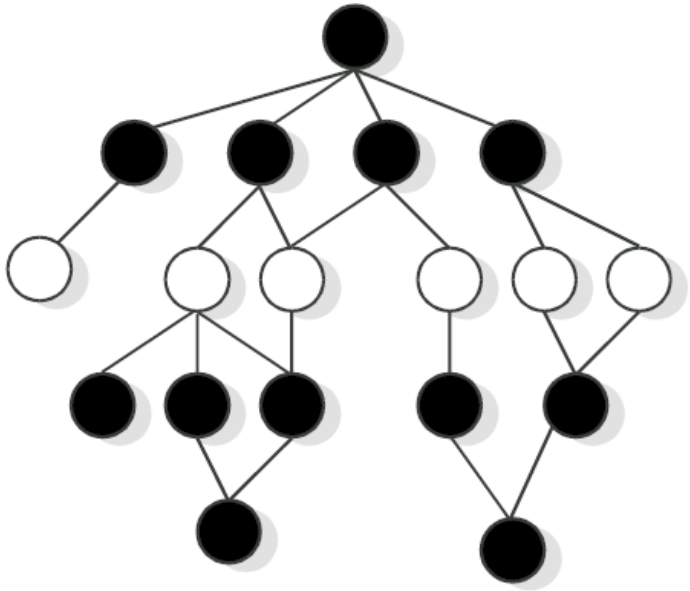
The purpose of unit testing is to determine that each independent module is correctly implemented. This gives little chance to determine that the interface between modules is also correct, and for this reason integration testing must be performed. One specific target of integration testing is the interface: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



Top-down integration



Bottom-up integration



Sandwich integration

Fig. 30 : Three different integration approaches

System Testing

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

Petschenik gives some guidelines for choosing test cases during system testing.

Software Testing

During system testing, we should evaluate a number of attributes of the software that are vital to the user and are listed in Fig. 31. These represent the operational correctness of the product and may be part of the software specifications.

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

Fig. 31 : Attributes of software to be tested during system testing

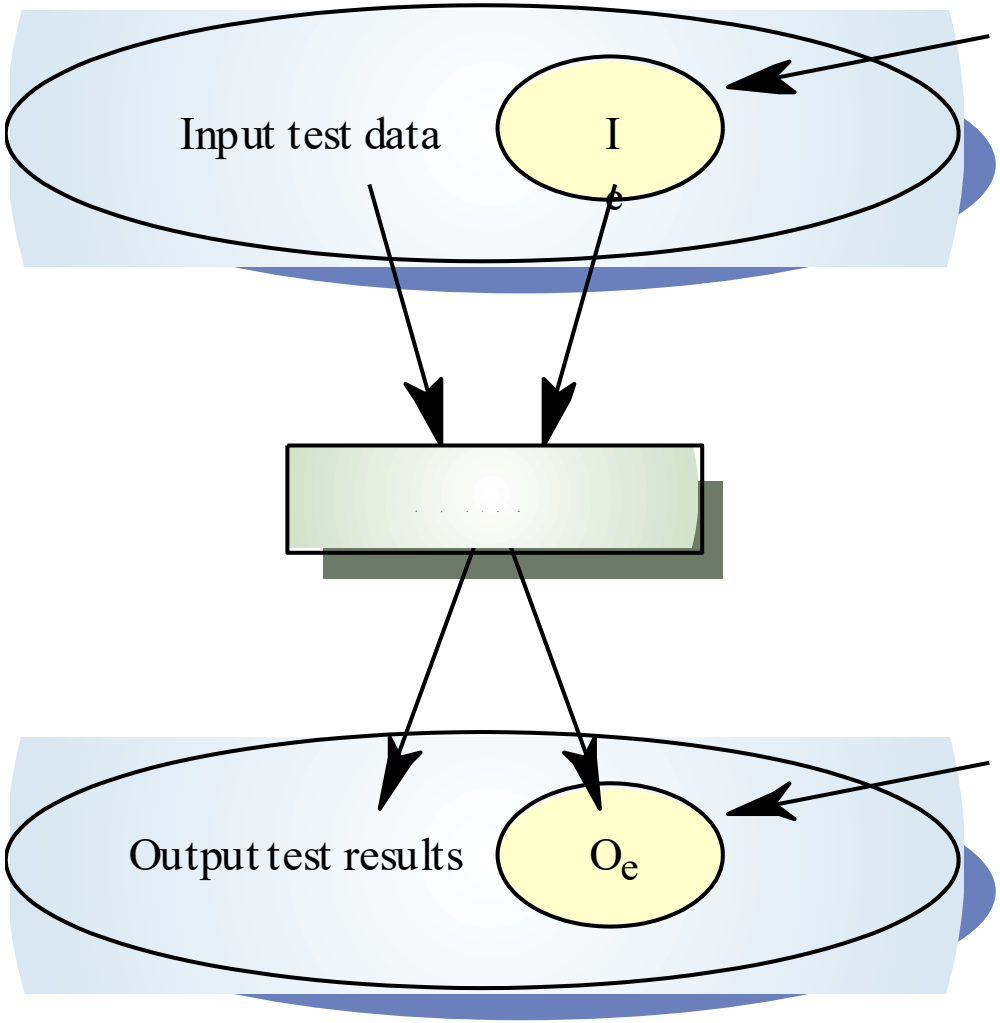
Methods of testing

- Test to specification:
  - Black box,
  - Data driven
  - Functional testing
  - Code is ignored: only use specification document to develop test cases
- Test to code:
  - Glass box/White box
  - Logic driven testing
  - Ignore specification and only examine the code.

Black-box testing

- An approach to testing where the program is considered as a ‘black-box’
- The program test cases are based on the system specification
- Test planning can begin early in the software process

Black-box testing



Functional Testing (Black Box Testing)

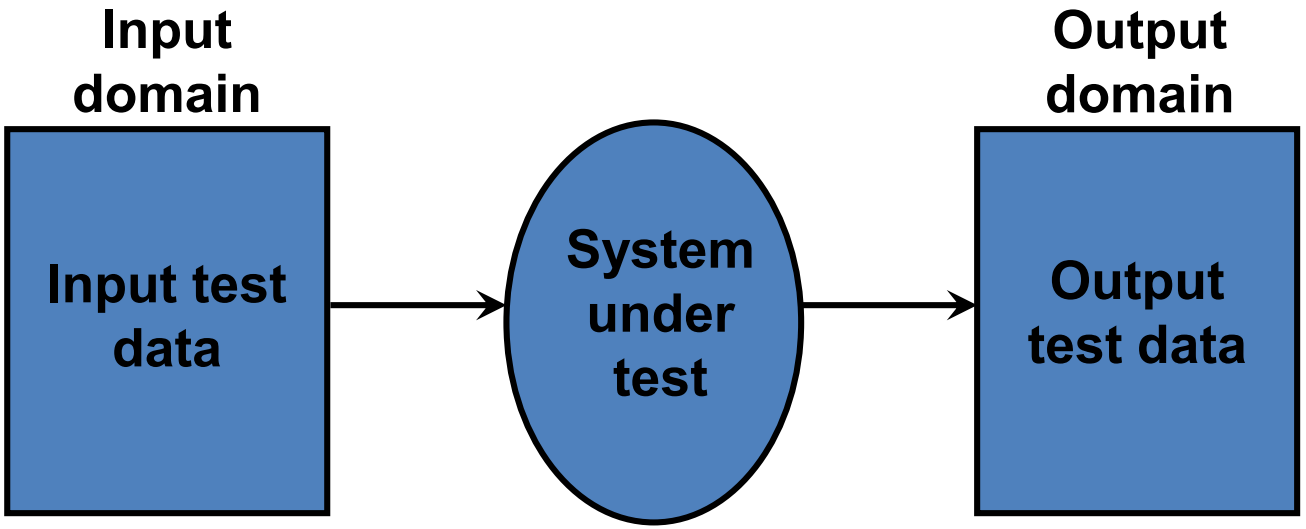


Fig. 3: Black box testing

Boundary Value Analysis

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

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

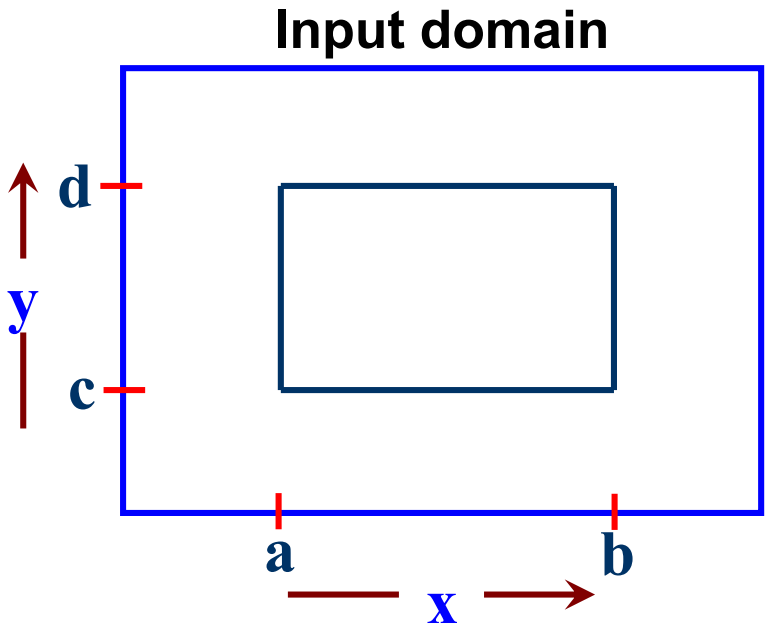


Fig.4: Input domain for program having two input variables

The boundary value analysis test cases for our program with two inputs variables (x and y) that may have any value from 100 to 300 are: (200,100), (200,101), (200,200), (200,299), (200,300), (100,200), (101,200), (299,200) and (300,200). This input domain is shown in Fig. 8.5. Each dot represent a test case and inner rectangle is the domain of legitimate inputs. Thus, for a program of n variables, boundary value analysis yield **4n + 1** test cases.

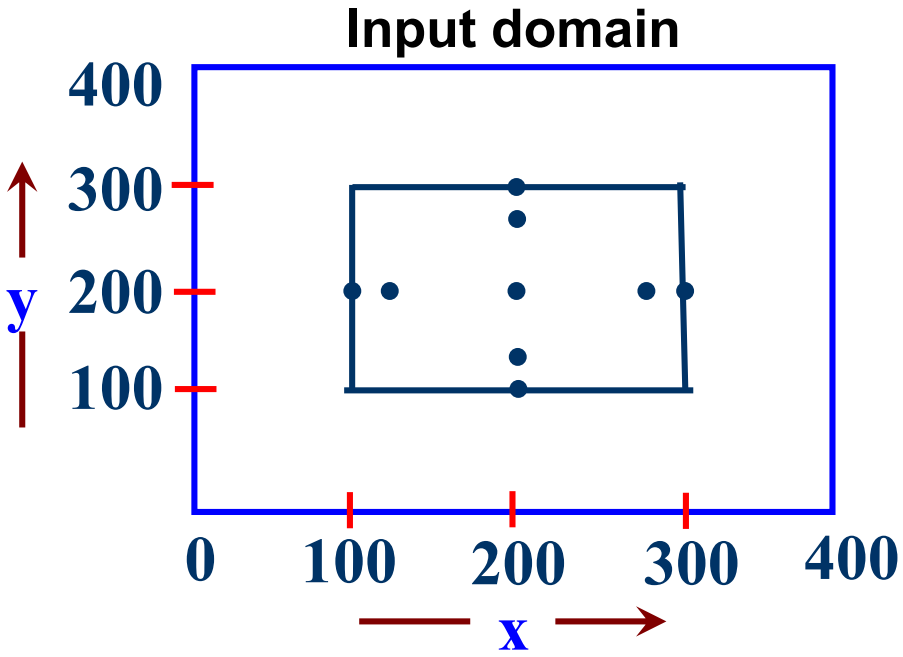


Fig. 5: Input domain of two variables x and y with boundaries [100,300] each

Example- 8.I

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

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

Design the boundary value test cases.

Solution

Quadratic equation will be of type:

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

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

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

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

Equation is not quadratic if $a=0$

Software Testing

The boundary value test cases are :

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

Example – 8.2

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

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

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

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

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

Solution

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

With single fault assumption theory, 4n+1 test cases can be designed and which are equal to 13.

Software Testing

The boundary value test cases are:

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

Software Testing

Software Testing

Robustness testing

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

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

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

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

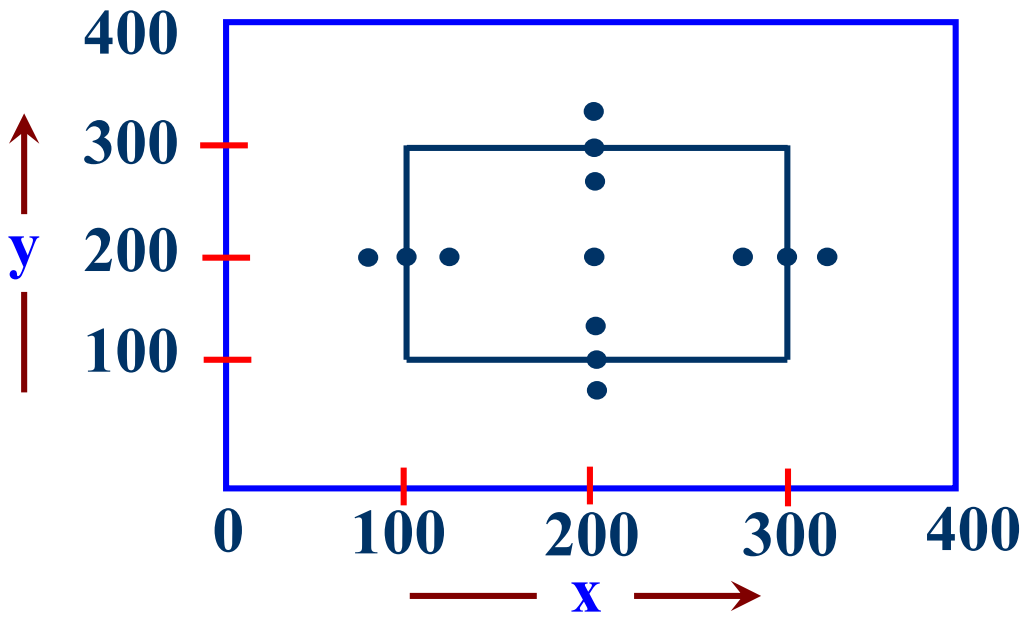


Fig. 8.6: Robustness test cases for two variables x and y with range [100,300] each

Worst-case testing

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

Table 1: Worst cases test inputs for two variables example

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

Structural Testing

A complementary approach to functional testing is called structural / white box testing. It permits us to examine the internal structure of the program.

Path Testing

Path testing is the name given to a group of test techniques based on judiciously selecting a set of test paths through the program. If the set of paths is properly chosen, then it means that we have achieved some measure of test thoroughness.

This type of testing involves:

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

White-box Testing

Methods based on the internal structure of code:

- Statement coverage
- Branch coverage
- Path coverage
- Data-flow coverage

White-box Testing

White-box methods can be used for

- Test case selection or generation.
- Test case adequacy assessment.

In practice, the most common use of white-box methods is as adequacy criteria after tests have been generated by some other method.

Flow Graph

The control flow of a program can be analysed using a graphical representation known as flow graph. The flow graph is a directed graph in which nodes are either entire statements or fragments of a statement, and edges represents flow of control.

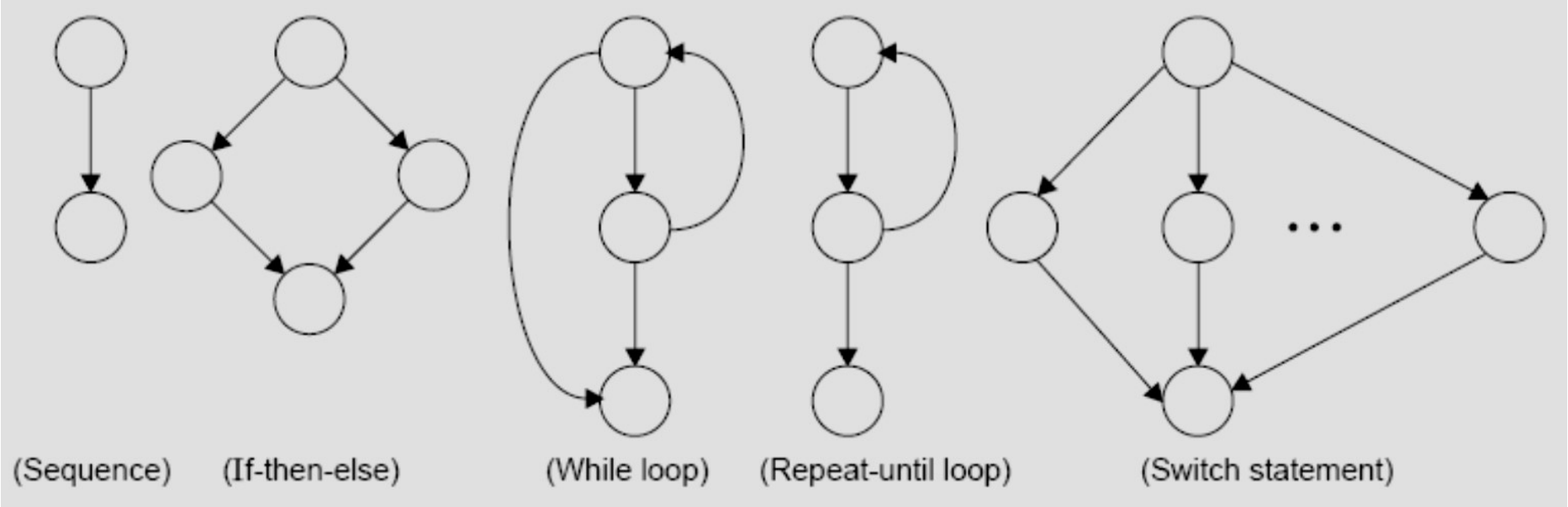


Fig. 14: The basic construct of the flow graph

Software Testing

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

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

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

(Contd.)...

```
12          if (day >= 1 && day <= 31) {
13              validDate = 1;
14          }
15          else {
16            validDate = 0;
17          }
18        }
19        else if (month == 2) {
20          int rVal=0;
21          if (year%4 == 0) {
22            rVal=1;
23            if ((year%100)==0 && (year % 400) !=0) {
24              rVal=0;
25            }
26          }
27          if (rVal ==1 && (day >=1 && day <=29) ) {
28            validDate = 1;
29          }
30          else if (day >=1 && day <= 28 ) {
31            validDate = 1;
32          }
```

(Contd.)...

Software Testing

```
33          else {
34            validDate = 0;
35          }
36        }
37        else if ((month >= 1 && month <= 12) && (day >= 1 && day <= 30)) {
38          validDate = 1;
39        }
40        else {
41          validDate = 0;
42        }
43      }
        /*Prev Date Calculation*/
44      if (validDate) {
45        if (day == 1) {
46          if (month == 1) {
47            year--;
48            day=31;
49            month=12;
50          }
51          else if (month == 3) {
52            int rVal=0;
```

(Contd.)...

Software Testing

```
53            if (year%4 == 0) {
54              rVal=1;
55              if ((year%100)==0 && (year % 400) !=0) {
56            rVal=0;
57          }
58            }
59            if (rVal ==1) {
60              day=29;
61              month--;
62            }
63            else {
64              day=28;
65              month--;
66            }
67          }
68          else if (month == 2 || month == 4 || month == 6 || month == 9 ||
            month == 11) {
69            day = 31;
70            month--;
```

(Contd.)...

```
71          }
72          else {
73            day=30;
74            month--;
75          }
76        }
77        else {
78          day--;
79        }
80        printf("The next date is: %d-%d-%d",day,month,year);
81      }
82      else {
83        printf("The entered date ( %d-%d-%d ) is invalid",day,month, year);
84      }
85      getche ();
86      return 1;
87  }
```

Fig. 15: Program for previous date problem

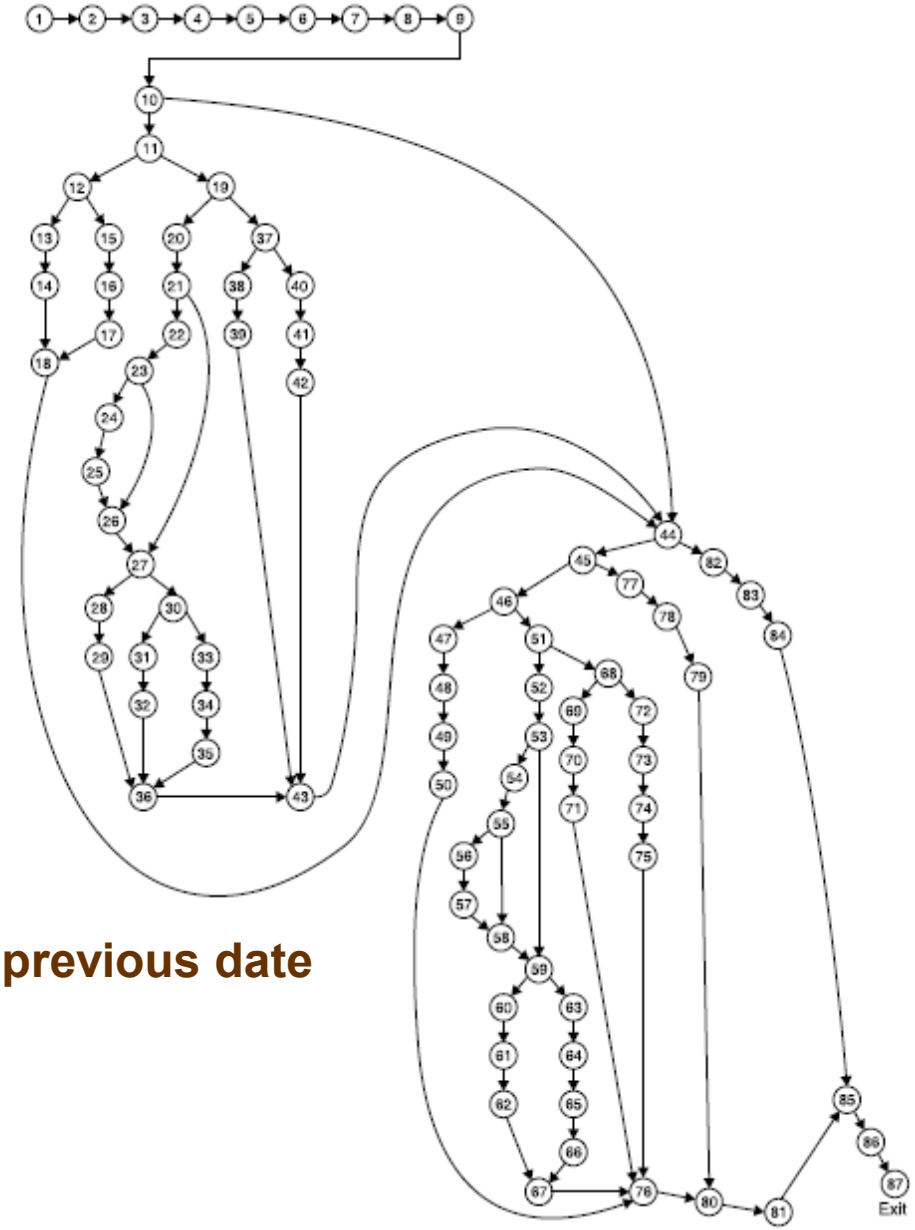


Fig. 16: Flow graph of previous date problem

DD Path Graph

Table 7: Mapping of flow graph nodes and DD path nodes

Flow graph nodes	DD Path graph corresponding node	Remarks
1 to 9	n_1	There is a sequential flow from node 1 to 9
10	n_2	Decision node, if true go to 11 else go to 44
11	n_3	Decision node, if true go to 12 else go to 19
12	n_4	Decision node, if true go to 13 else go to 15
13,14	n_5	Sequential nodes and are combined to form new node n_5
15,16,17	n_6	Sequential nodes
18	n_7	Edges from node 14 to 17 are terminated here
19	n_8	Decision node, if true go to 20 else go to 37
20	n_9	Intermediate node with one input edge and one output edge
21	n_{10}	Decision node, if true go to 22 else go to 27
22	n_{11}	Intermediate node
23	n_{12}	Decision node, if true go to 24 else go to 26

Cont….

Flow graph nodes	DD Path graph corresponding node	Remarks
24,25	n_{13}	Sequential nodes
26	n_{14}	Two edges from node 25 & 23 are terminated here
27	n_{15}	Two edges from node 26 & 21 are terminated here. Also a decision node
28,29	n_{16}	Sequential nodes
30	n_{17}	Decision node, if true go to 31 else go to 33
31,32	n_{18}	Sequential nodes
33,34,35	n_{19}	Sequential nodes
36	n_{20}	Three edge from node 29,32 and 35 are terminated here
37	n_{21}	Decision node, if true go to 38 else go to 40
38,39	n_{22}	Sequential nodes
40,41,42	n_{23}	Sequential nodes
43	n_{24}	Three edge from node 36,39 and 42 are terminated here

Cont….

Flow graph nodes	DD Path graph corresponding node	Remarks
44	n_{25}	Decision node, if true go to 45 else go to 82. Three edges from 18,43 & 10 are also terminated here.
45	n_{26}	Decision node, if true go to 46 else go to 77
46	n_{27}	Decision node, if true go to 47 else go to 51
47,48,49,50	n_{28}	Sequential nodes
51	n_{29}	Decision node, if true go to 52 else go to 68
52	n_{30}	Intermediate node with one input edge & one output ege
53	n_{31}	Decision node, if true go to 54 else go to 59
54	n_{32}	Intermediate node
55	n_{33}	Decision node, if true go to 56 else go to 58
56,57	n_{34}	Sequential nodes
58	n_{35}	Two edge from node 57 and 55 are terminated here
59	n_{36}	Decision node, if true go to 60 else go to 63. Two edge from nodes 58 and 53 are terminated.

Cont….

Flow graph nodes	DD Path graph corresponding node	Remarks
60,61,62	n_{37}	Sequential nodes
63,64,65,66	n_{38}	Sequential nodes
67	n_{39}	Two edge from node 62 and 66 are terminated here
68	n_{40}	Decision node, if true go to 69 else go to 72
69,70,71	n_{41}	Sequential nodes
72,73,74,75	n_{42}	Sequential nodes
76	n_{43}	Four edges from nodes 50, 67, 71 and 75 are terminated here.
77,78,79	n_{44}	Sequential nodes
80	n_{45}	Two edges from nodes 76 & 79 are terminated here
81	n_{46}	Intermediate node
82,83,84	n_{47}	Sequential nodes
85	n_{48}	Two edges from nodes 81 and 84 are terminated here
86,87	n_{49}	Sequential nodes with exit node

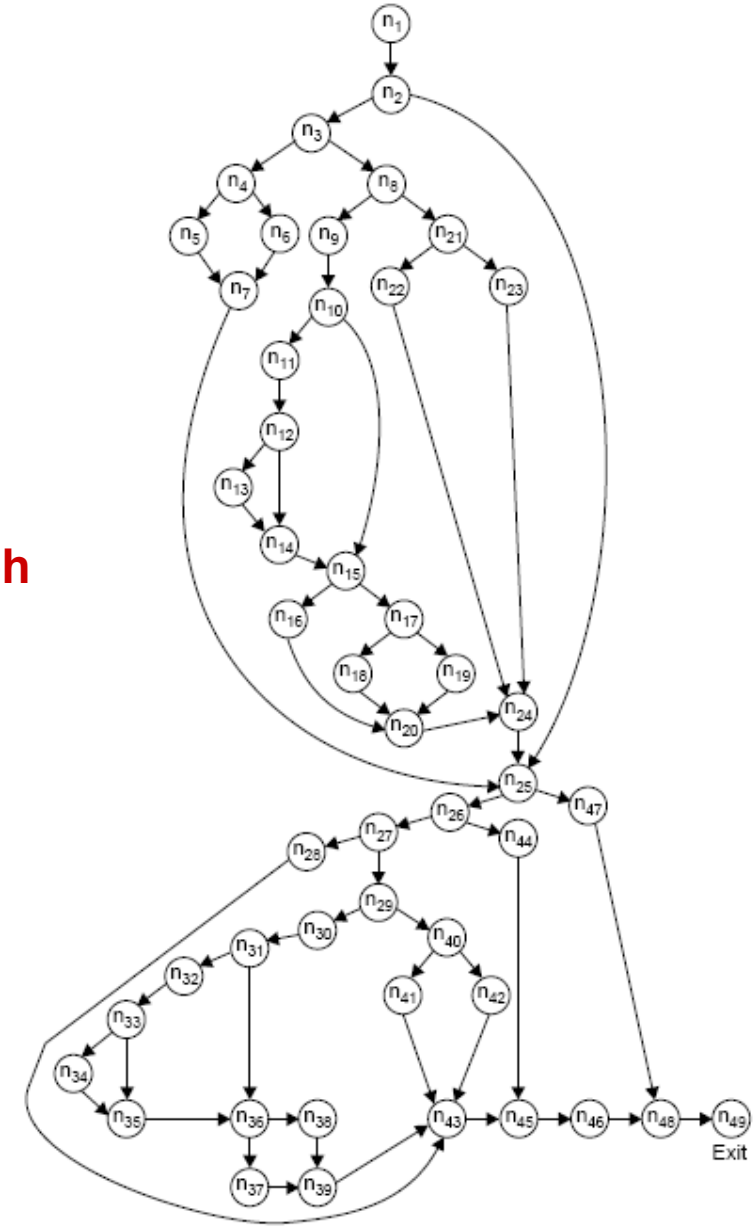


Fig. 17: DD path graph of previous date problem

Example 8.13

Consider the problem for the determination of the nature of roots of a quadratic equation. Its input a triple of positive integers (say a,b,c) and value may be from interval [0,100].

The program is given in fig. 19. The output may have one of the following words:

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

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

Software Testing

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

Cont…

```
21        }
22        else if ( d > 0 ) {
23          D=sqrt(d);
24          printf("The roots are real and are r1 = %f and r2 = %f\n",
                    (-b-D)/(2* a), (-b+D)/(2* a));
25        }
26        else {
27          D=sqrt(-d)/(2*a);
28          printf("The roots are imaginary and are r1 = (%f,%f) and
                    r2 = (%f,%f)\n", -b/(2.0*a),D,-b/(2.0*a),-D);
29        }
30      }
31      else if (validInput == -1) {
32        printf("The vlaues do not constitute a Quadratic equation.");
33      }
34      else {
35        printf("The inputs belong to invalid range.");
36      }
37      getche();
38      return 1;
39  }
```

Fig. 19: Code of quadratic equation problem

Solution

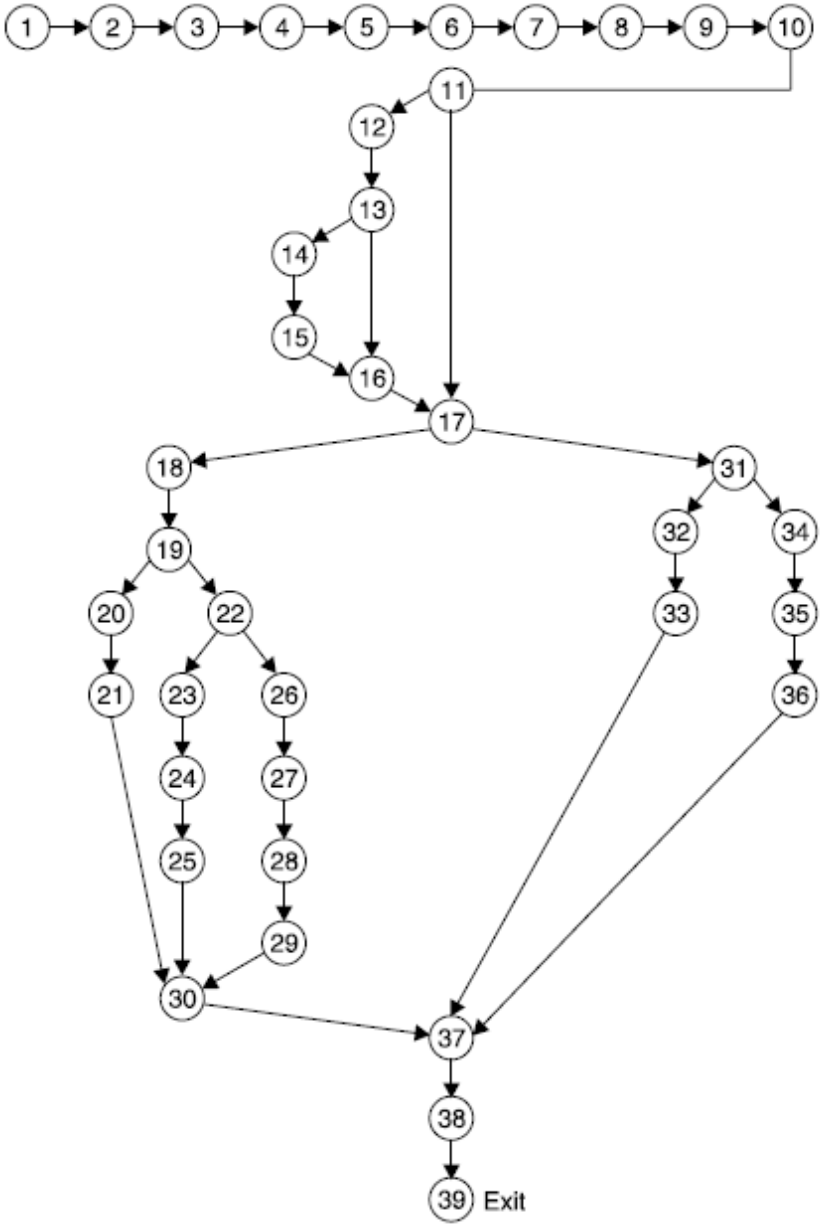


Fig. 19 (a) : Program flow graph

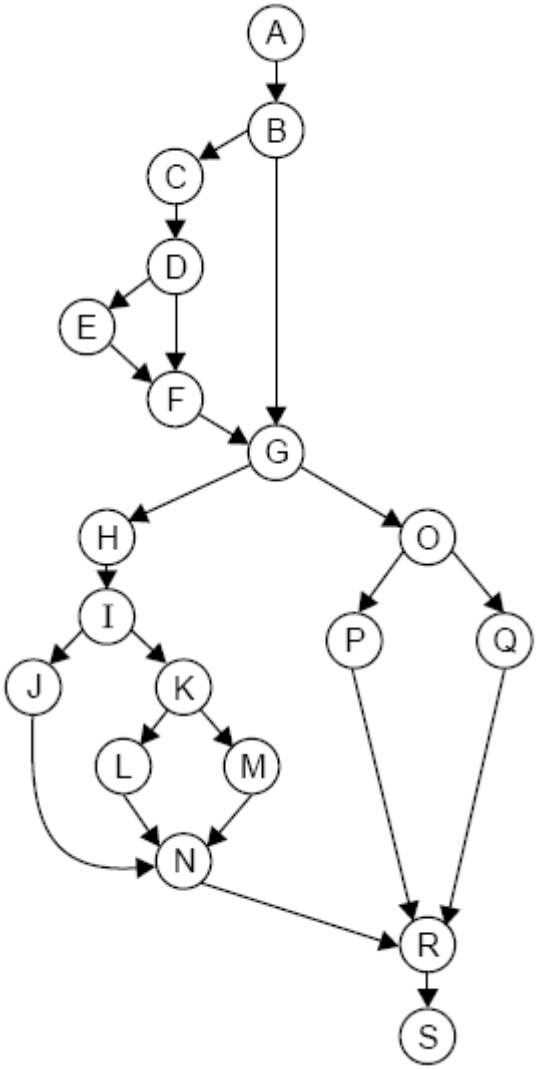


Fig. 19 (b) : DD Path graph

The mapping table for DD path graph is:

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

Cont….

Flow graph nodes	DD Path graph corresponding node	Remarks
26,27,28,29	M	Sequential nodes
30	N	Three edges are combined
31	O	Decision node
32,33	P	Sequential node
34,35,36	Q	Sequential node
37	R	Three edges are combined here
38,39	S	Sequential nodes with exit node

Software Testing

Software Testing

Software Testing

Software Testing

Software Testing

Software Testing

Software Testing

Static and dynamic verification

- *Software inspections and walkthroughs* - Concerned with analysis of the static system representation to discover problems *(*static verification)
- *Software testing* - Concerned with exercising and observing product behaviour (dynamic verification)
  - The system is executed with test data and its operational behaviour is observed

Formal Technical Reviews

Peer Reviews

- Industries' best-practice for detecting software defects
- Can be differentiated into:
  - Walk Through
  - Software Inspections

Walkthroughs

- Informal examination of a product (document)
- Made up of:
  - developers
  - client
  - next phase developers
  - Software Quality Assurance group leader
- Produces:
  - list of items not understood
  - list of items thought to be incorrect

Software inspections

- Involve people examining the source representation with the aim of discovering anomalies and defects
- Do not require execution of a system so may be used before implementation
- May be applied to any representation of the system (requirements, design, test data, etc.)
- Very effective technique for discovering errors

Inspection success

- Many different defects may be discovered in a single inspection. In testing, one defect may mask another so several executions are required
- The reuse domain and programming knowledge so reviewers are likely to have seen the types of error that commonly arise

Inspections and testing

- Inspections and testing are complementary and not opposing verification techniques.
- Both should be used during the V & V process.
- Inspections can check conformance with a specification but not conformance with the customer's real requirements.
- Inspections cannot check non-functional characteristics such as performance, usability, etc.

Inspection procedure

- System overview presented to inspection team
- Code and associated documents are distributed to inspection team in advance
- Inspection takes place and discovered errors are noted
- Modifications are made to repair discovered errors
- Re-inspection may or may not be required

Inspection teams

- Made up of at least 4 members
  - Author of the code being inspected
  - Inspector who finds errors, omissions and inconsistencies
  - Reader who reads the code to the team
  - Moderator who chairs the meeting and notes discovered errors
- Other roles are Scribe and Chief moderator

Inspection rate

- 500 statements/hour during overview
- 125 source statement/hour during individual preparation
- 90-125 statements/hour can be inspected
- Inspection is therefore an expensive process
- Inspecting 5000 lines costs about 40 man/hours effort (@ $50/hr = $2000!!!)

More Realities of Software Testing

- Exhaustive testing is not possible.
- Testing is creative and difficult.
- A major objective of testing is failure prevention.
- Testing must be planned.
- Testing should be done by people who are independent of the developers.