Chapter 8

Testing

Declaration

- These slides are made for UIT, BU students only. I am not holding any copy write of it as I had collected these study materials from different books and websites etc. I have not mentioned those to avoid complexity.

What is Testing?

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

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.

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 organizations 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 256x256. 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.

What should We Test ?

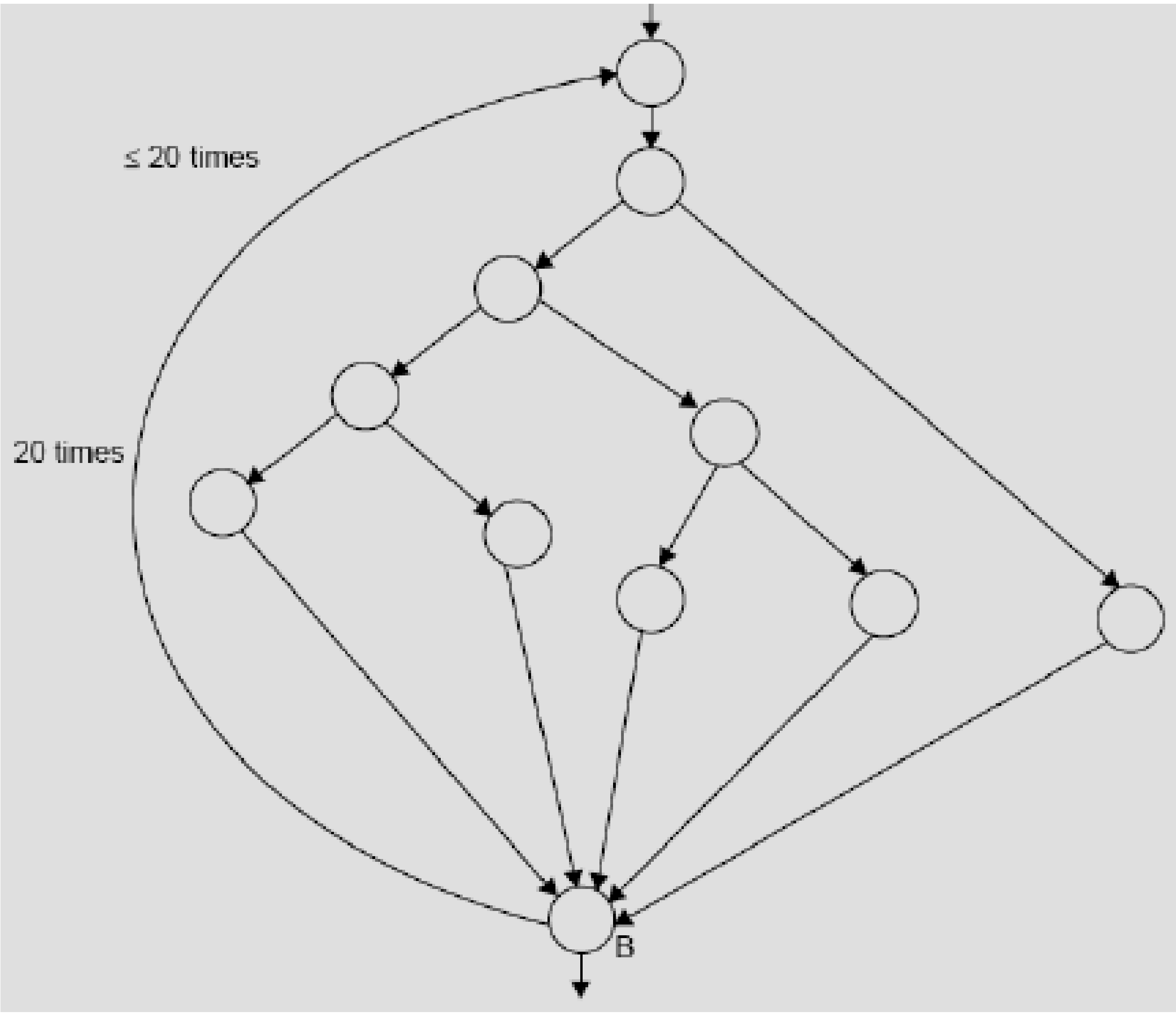


Fig. 1: Control flow graph

What should We Test ?

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

Some Terminologies

- 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.

Some Terminologies

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

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

Some Terminologies

- **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**

Levels of 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.

Functional Testing

- **FUNCTIONAL TESTING** is a type of software testing whereby the system is tested against the functional requirements/ specifications.
- Functions (or features) are tested by feeding them input and examining the output. Functional testing ensures that the requirements are properly satisfied by the application. This type of testing is not concerned with how processing occurs, but rather, with the results of processing. It simulates actual system usage but does not make any system structure assumptions.

Functional Testing

- During functional testing, Black Box Testing technique is used in which the internal logic of the system being tested is not known to the tester.

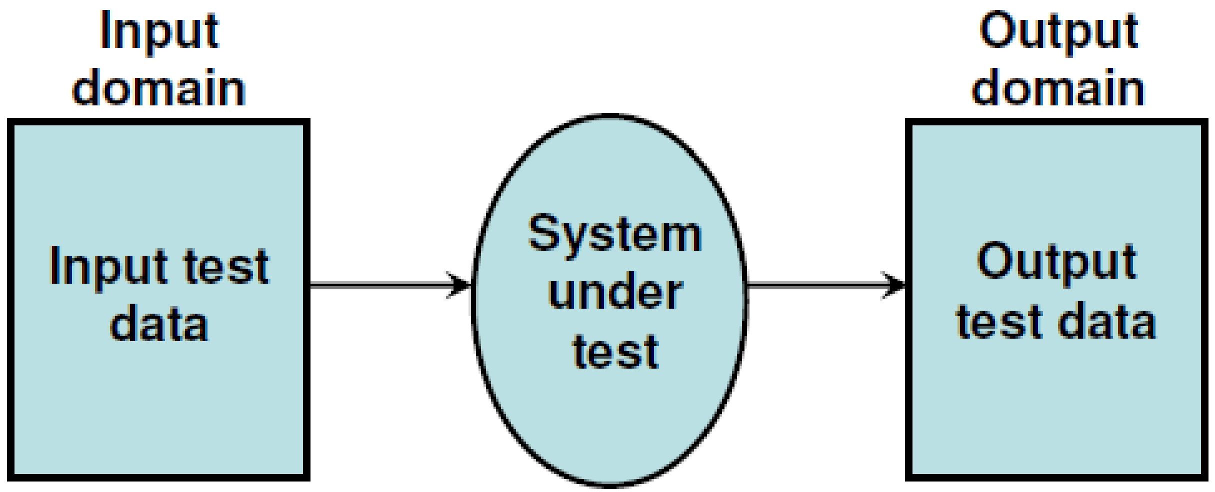


Fig. 3: Black box testing

Boundary Value Analysis

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

$$a \le x \le b$$

$$c \le y \le d$$

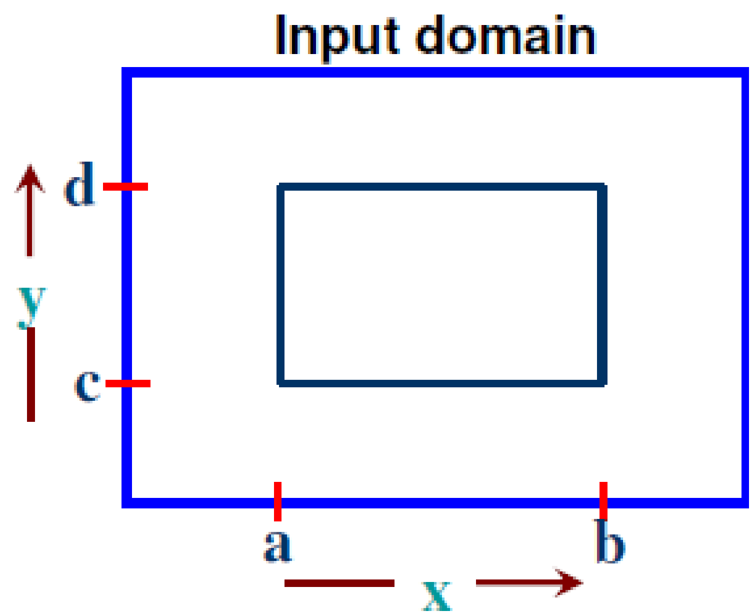


Fig.4: Input domain for program having two input variables

Boundary Value Analysis

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.

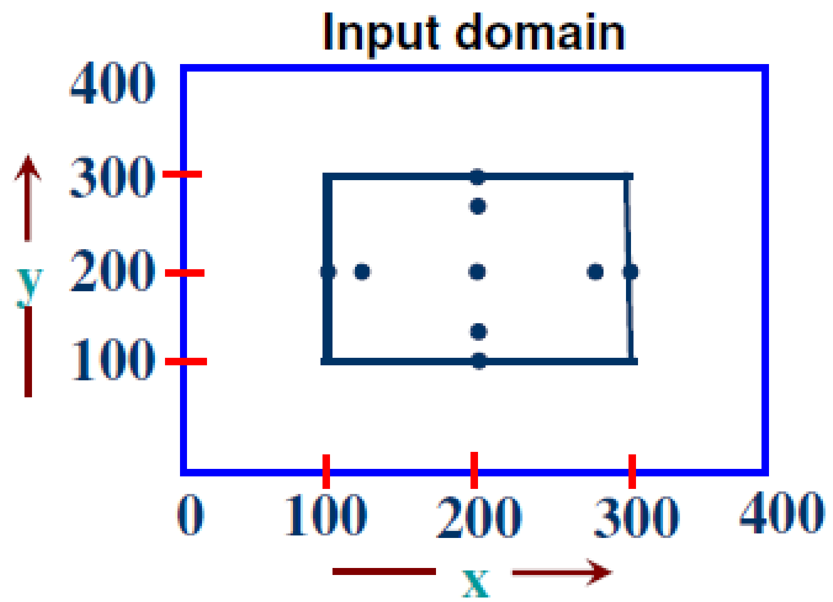


Fig. 5: Input domain of two variables x and y with boundaries [100,300] each

Boundary Value Analysis

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.

Boundary Value Analysis

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 $a=0$

Boundary Value Analysis

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

Boundary Value Analysis

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.

Boundary Value Analysis

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.

Boundary Value Analysis

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

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 and shown in Fig. 6 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)

Robustness Testing

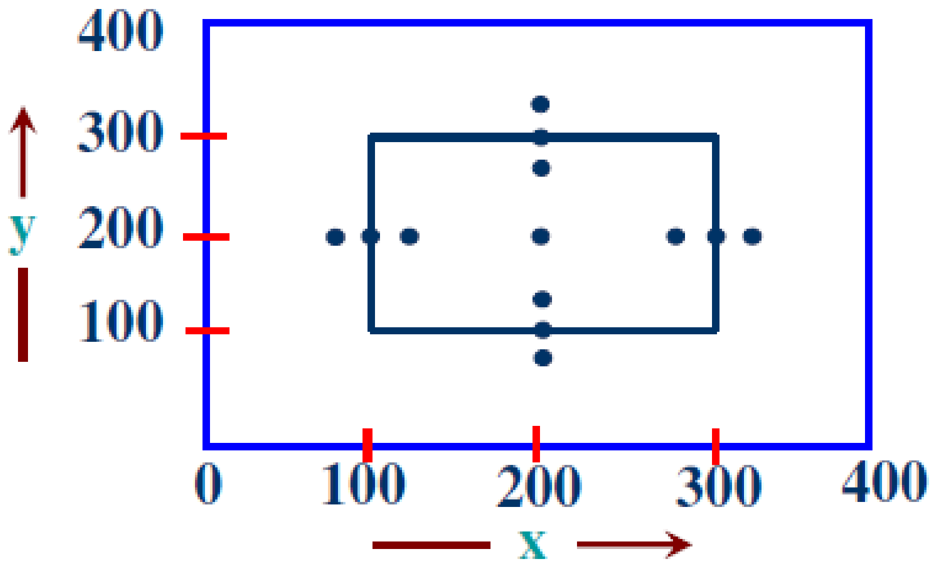


Fig. 8.6: Robustness test cases for two variables x and y with range [100,300] each

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.

Worst-case testing

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	--		

Worst-case testing

Example - 8.4

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

Robustness Testing

Solution

Robust test cases are 6n+1. Hence, in 3 variable input cases total number of test cases are 19 as given on next slide:

Robustness Testing

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

Worst-case testing

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

Test Case	a	b	c	Expected output
1	0	0	0	Not Quadratic
2	0	0	1	Not Quadratic
3	0	0	50	Not Quadratic
4	0	0	99	Not Quadratic
5	0	0	100	Not Quadratic
6	0	1	0	Not Quadratic
7	0	1	1	Not Quadratic
8	0	1	50	Not Quadratic
9	0	1	99	Not Quadratic
10	0	1	100	Not Quadratic
11	0	50	0	Not Quadratic
12	0	50	1	Not Quadratic
13	0	50	50	Not Quadratic
14	0	50	99	Not Quadratic

(Contd.)...

Worst-case testing

Test Case	A	b	c	Expected output
15	0	50	100	Not Quadratic
16	0	99	0	Not Quadratic
17	0	99	1	Not Quadratic
18	0	99	50	Not Quadratic
19	0	99	99	Not Quadratic
20	0	99	100	Not Quadratic
21	0	100	0	Not Quadratic
22	0	100	1	Not Quadratic
23	0	100	50	Not Quadratic
24	0	100	99	Not Quadratic
25	0	100	100	Not Quadratic
26	1	0	0	Equal Roots
27	1	0	1	Imaginary
28	1	0	50	Imaginary
29	1	0	99	Imaginary
30	1	0	100	Imaginary
31	1	1	0	Real Roots

(Contd.)...

Worst-case testing

Test Case	A	b	C	Expected output
32	1	1	1	Imaginary
33	1	1	50	Imaginary
34	1	1	99	Imaginary
35	1	1	100	Imaginary
36	1	50	0	Real Roots
37	1	50	1	Real Roots
38	1	50	50	Real Roots
39	1	50	99	Real Roots
40	1	50	100	Real Roots
41	1	99	0	Real Roots
42	1	99	1	Real Roots
43	1	99	50	Real Roots
44	1	99	99	Real Roots
45	1	99	100	Real Roots
46	1	100	0	Real Roots
47	1	100	1	Real Roots
48	1	100	50	Real Roots

(Contd.)...

Worst-case testing

Test Case	A	b	C	Expected output
49	1	100	99	Real Roots
50	1	100	100	Real Roots
51	50	0	0	Equal Roots
52	50	0	1	Imaginary
53	50	0	50	Imaginary
54	50	0	99	Imaginary
55	50	0	100	Imaginary
56	50	1	0	Real Roots
57	50	1	1	Imaginary
58	50	1	50	Imaginary
59	50	1	99	Imaginary
60	50	1	100	Imaginary
61	50	50	0	Real Roots
62	50	50	1	Real Roots
63	50	50	50	Imaginary
64	50	50	99	Imaginary
65	50	50	100	Imaginary

(Contd.)..

Worst-case testing

Test Case	A	b	C	Expected output
66	50	99	0	Real Roots
67	50	99	1	Real Roots
68	50	99	50	Imaginary
69	50	99	99	Imaginary
70	50	99	100	Imaginary
71	50	100	0	Real Roots
72	50	100	1	Real Roots
73	50	100	50	Equal Roots
74	50	100	99	Imaginary
75	50	100	100	Imaginary
76	99	0	0	Equal Roots
77	99	0	1	Imaginary
78	99	0	50	Imaginary
79	99	0	99	Imaginary
80	99	0	100	Imaginary
81	99	1	0	Real Roots
82	99	1	1	Imaginary

(Contd.)...

Worst-case testing

Test Case	A	b	C	Expected output
83	99	1	50	Imaginary
84	99	1	99	Imaginary
85	99	1	100	Imaginary
86	99	50	0	Real Roots
87	99	50	1	Real Roots
88	99	50	50	Imaginary
89	99	50	99	Imaginary
90	99	50	100	Imaginary
91	99	99	0	Real Roots
92	99	99	1	Real Roots
93	99	99	50	Imaginary Roots
94	99	99	99	Imaginary
95	99	99	100	Imaginary
96	99	100	0	Real Roots
97	99	100	1	Real Roots
98	99	100	50	Imaginary
99	99	100	99	Imaginary
100	99	100	100	Imaginary

Software Engineering (3rd ed.), By K.K Aggarwal & Yogesh Singh, Copyright © New Age International Publishers, 2007

(Contd.)...

Worst-case testing

Test Case	A	b	C	Expected output
119	100	99	99	Imaginary
120	100	99	100	Imaginary
121	100	100	0	Real Roots
122	100	100	1	Real Roots
123	100	100	50	Imaginary
124	100	100	99	Imaginary
125	100	100	100	Imaginary

Equivalence Class Testing

- In this method, input domain of a program is partitioned into a finite number of equivalence classes such that one can reasonably assume, but not be absolutely sure, that the test of a representative value of each class is equivalent to a test of any other value.
- **Two steps are required to implementing this method:**

  1. The equivalence classes are identified by taking each input condition and partitioning it into valid and invalid classes. For example, if an input condition specifies a range of values from 1 to 999, we identify one valid equivalence class [1<item<999]; and two invalid equivalence classes [item<1] and [item>999].

  2. Generate the test cases using the equivalence classes identified in the previous step. This is performed by writing test cases covering all the valid equivalence classes. Then a test case is written for each invalid equivalence class so that no test contains more than one invalid class. This is to ensure that no two invalid classes mask each other.

Equivalence Class Testing

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

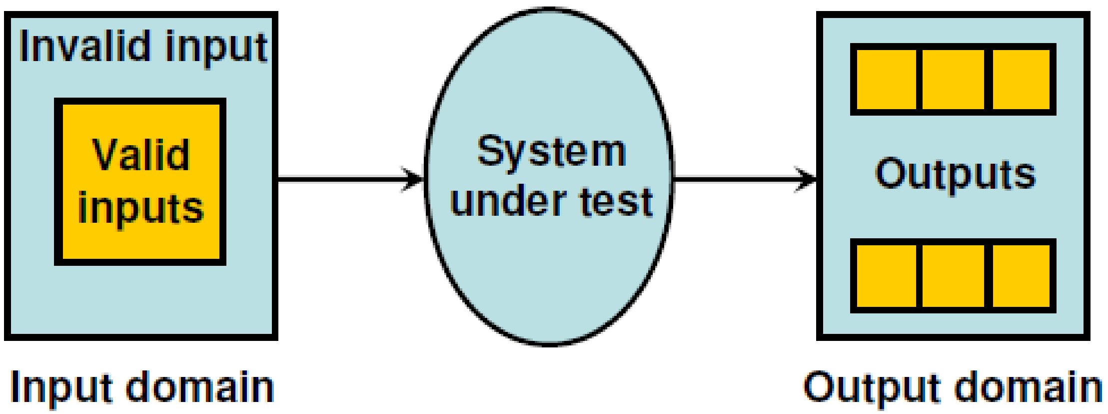


Fig. 7: Equivalence partitioning

Equivalence Class Testing

Example 8.7

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

Equivalence Class Testing

Solution

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

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

$O_1=\{<a,b,c>:\text{Real roots if } (b^2-4ac)>0\}$

$O_1=\{<a,b,c>:\text{Imaginary roots if } (b^2-4ac)<0\}$

$O_1=\{<a,b,c>:\text{Equal roots if } (b^2-4ac)=0\}$`

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

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

Equivalence Class Testing

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

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

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

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

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

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

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

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

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

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

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

- Hence total test cases are10+4=14 for this problem.

Structural Testing or white box 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.

Structural Testing or white box testing

- **Flow Graph:** The control flow of a program can be analyzed 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.

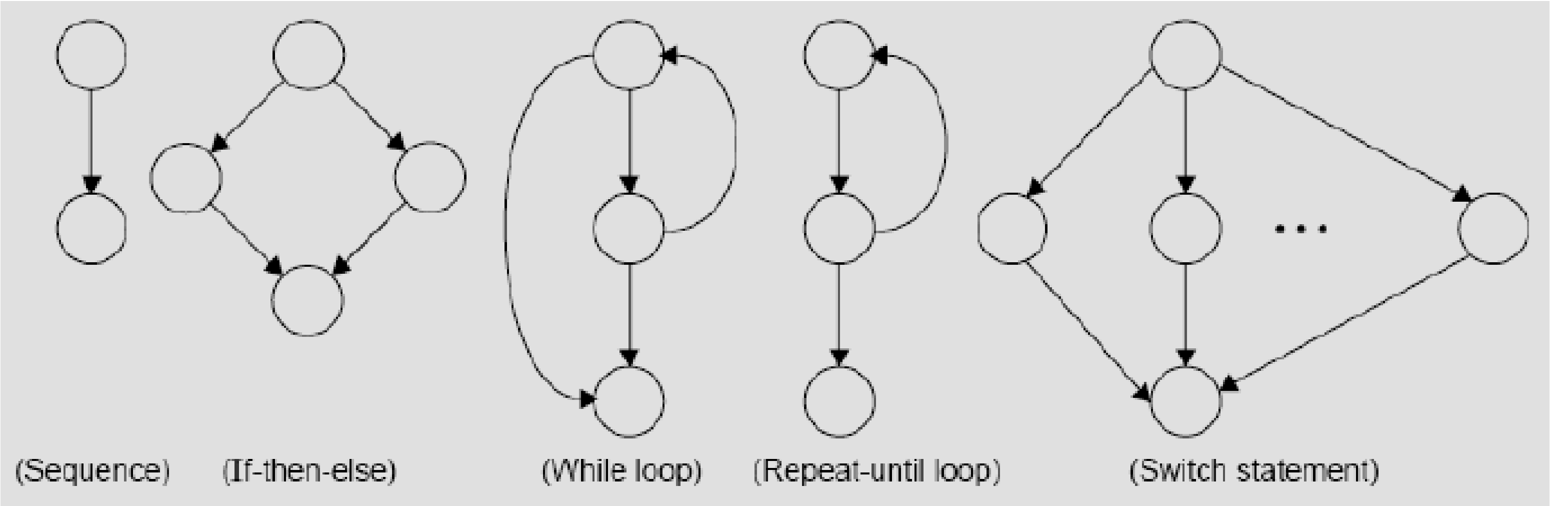


Fig. 14: The basic construct of the flow graph

Structural Testing or white box 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>

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

(Contd.)...

Structural Testing or white box testing

```
        if (day >= 1 && day <= 31) {
            validDate = 1;
        }
        else {
          validDate = 0;
        }
      }
      else if (month == 2) {
        int rVal=0;
        if (year%4 == 0) {
          rVal=1;
          if ((year%100)==0 && (year % 400) !=0) {
            rVal=0;
          }
        }
        if (rVal ==1 && (day >=1 && day <=29) ) {
          validDate = 1;
        }
        else if (day >=1 && day <= 28 ) {
          validDate = 1;
        }
```

(Contd.)...

Structural Testing or white box testing

```
        else {
          validDate = 0;
        }
      }
      else if ((month >= 1 && month <= 12) && (day >= 1 && day <= 30)) {
        validDate = 1;
      }
      else {
        validDate = 0;
      }
    }
    /*Prev Date Calculation*/
    if (validDate) {
      if (day == 1) {
        if (month == 1) {
          year--;
          day=31;
          month=12;
        }
        else if (month == 3) {
          int rVal=0;
```

(Contd.)...

Structural Testing or white box testing

```
          if (year%4 == 0) {
            rVal=1;
            if ((year%100)==0 && (year % 400) !=0) {
          rVal=0;
        }
          }
          if (rVal ==1) {
            day=29;
            month--;
          }
          else {
            day=28;
            month--;
          }
        }
        else if (month == 2 || month == 4 || month == 6 || month == 9 ||
        month == 11) {
          day = 31;
          month--;
```

(Contd.)...

Structural Testing or white box testing

```
            }
            else {
              day=30;
              month--;
            }
          }
          else {
            day--;
          }
          printf("The next date is: %d-%d-%d",day,month,year);
        }
        else {
          printf("The entered date ( %d-%d-%d ) is invalid",day,month, year);
        }
        getche ();
        return 1;
    }
```

Fig. 15: Program for previous date problem

Structural Testing or white box testing

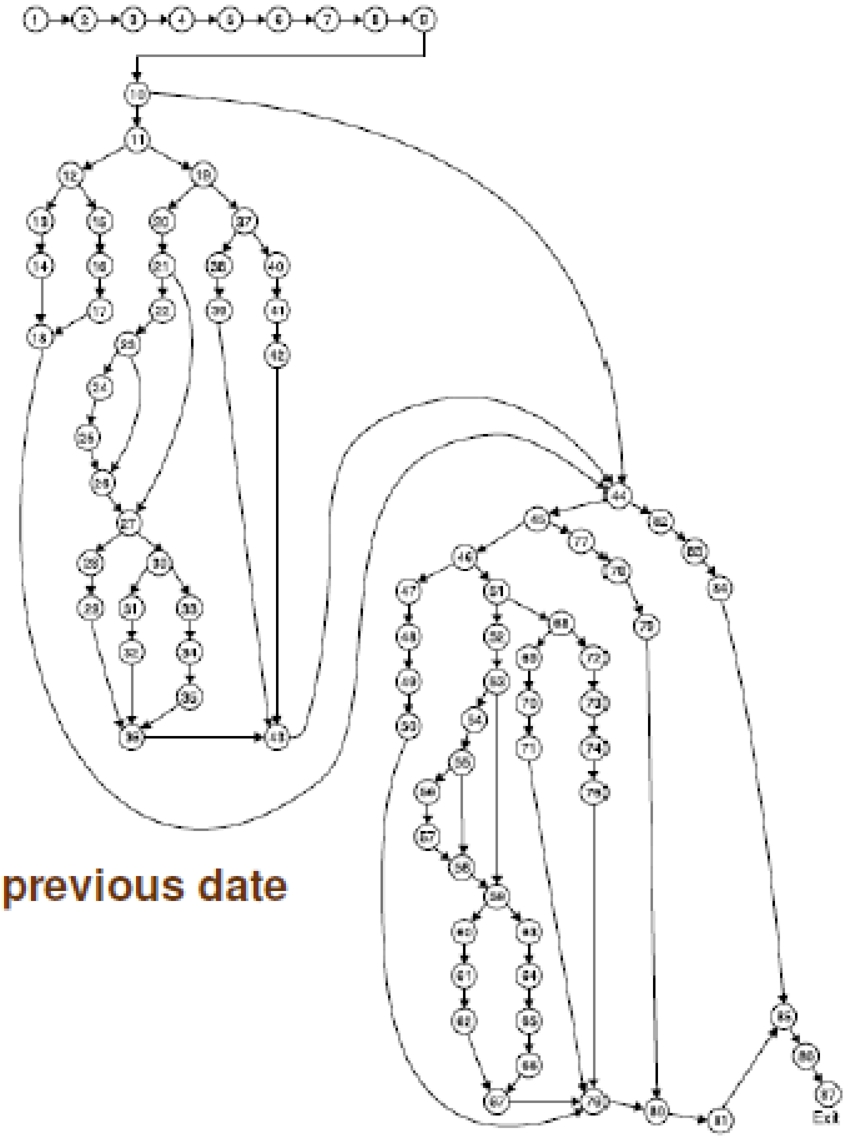


Fig. 16: Flow graph of previous date problem

Structural Testing or white box testing

Cyclomatic Complexity

McCabe's cyclomatic metric $V(G) = e - n + 2P$.

For example, a flow graph shown in in Fig. 21 with entry node '*a*' and exit node '*f*'.

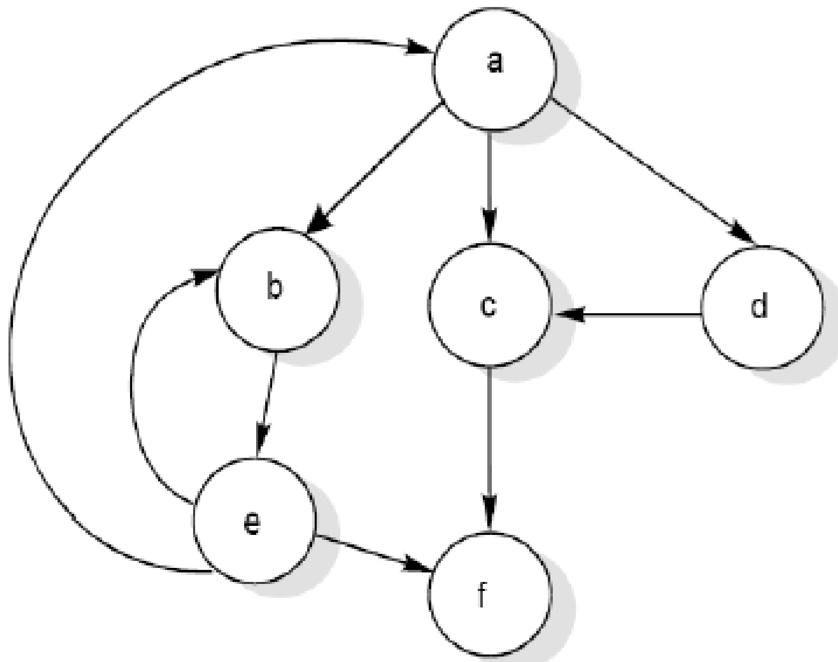


Fig. 21: Flow graph

Structural Testing or white box testing

- *where,*
  e = the number of edges in the control flow graph
  n = the number of nodes in the control flow graph
  P = the number of connected components

The value of cyclomatic complexity can be calculated as :

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

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

Structural Testing or white box testing

The role of P in the complexity calculation $V(G)=e-n+2P$ is required to be understood correctly. We define a flow graph with unique entry and exit nodes, all nodes reachable from the entry, and exit reachable from all nodes. This definition would result in all flow graphs having only one connected component. One could, however, imagine a main program M and two called subroutines A and B having a flow graph shown in Fig. 22.

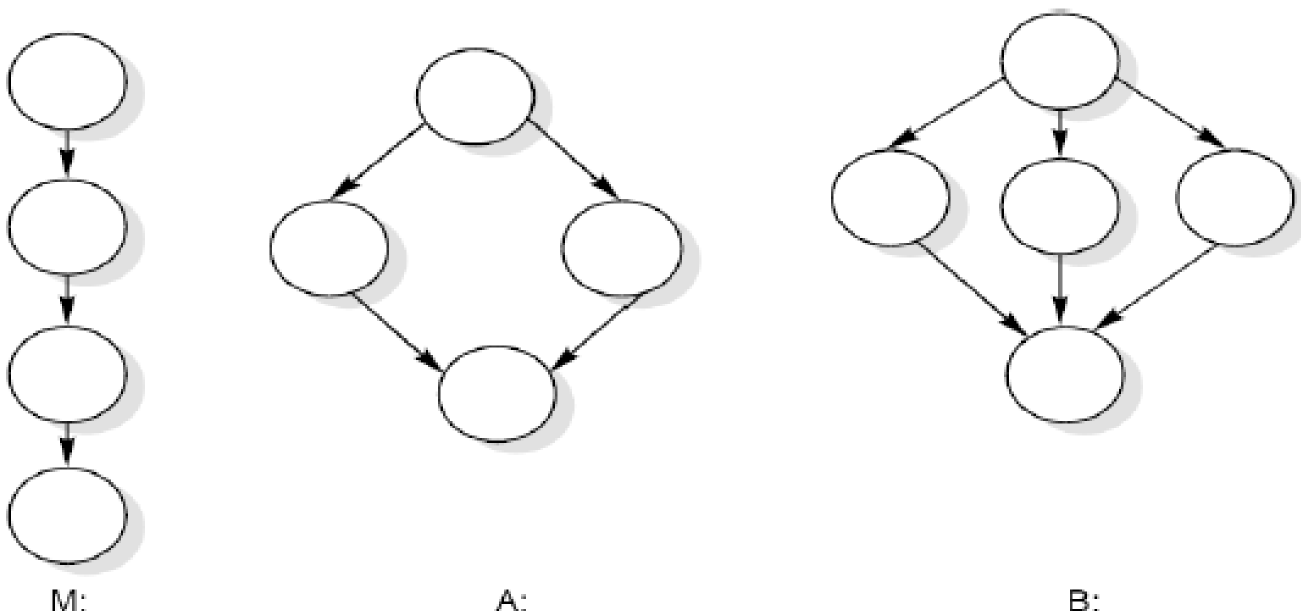


Fig. 22

Structural Testing or white box testing

Let us denote the total graph above with 3 connected components as

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

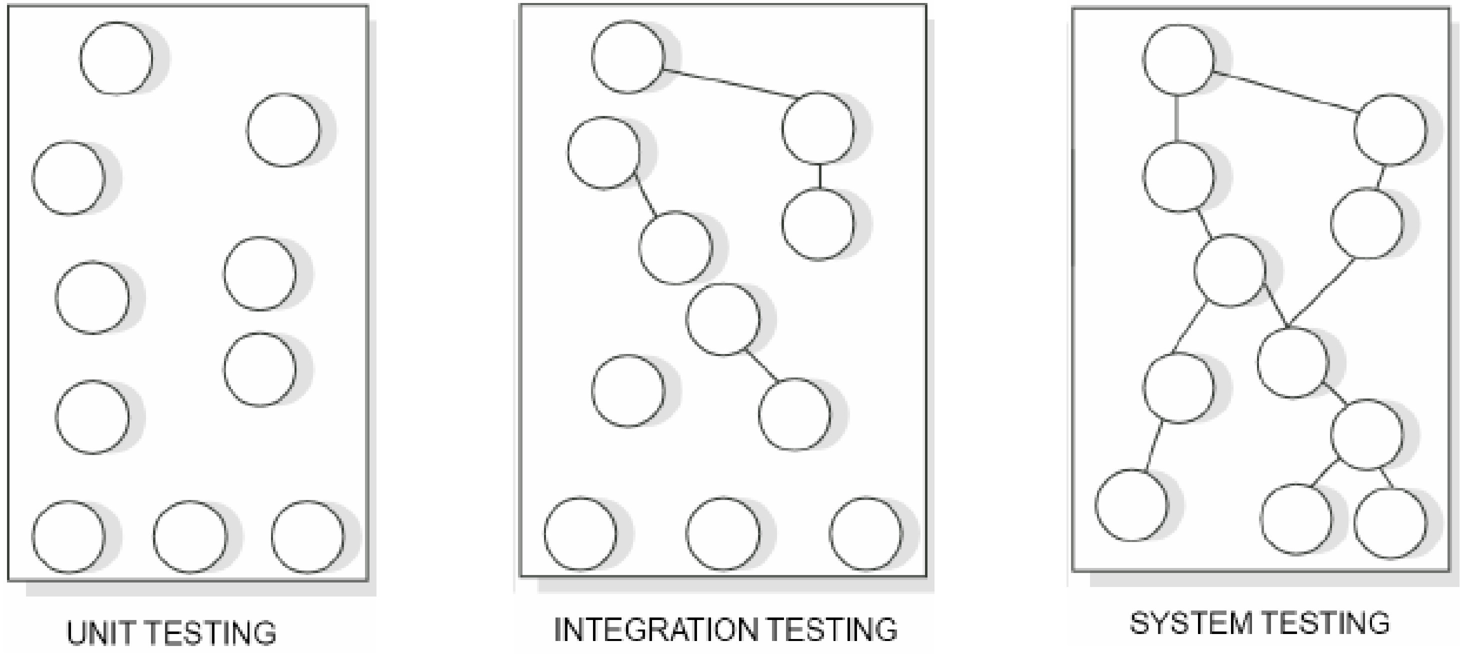
$$= 13\text{-}13\text{+}2\text{*}3$$

$$= 6$$

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

Levels of Testing

- There are 3 levels of testing:
  - i. Unit Testing
  - ii. Integration Testing
  - iii. System Testing



Unit Testing

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

Unit Testing

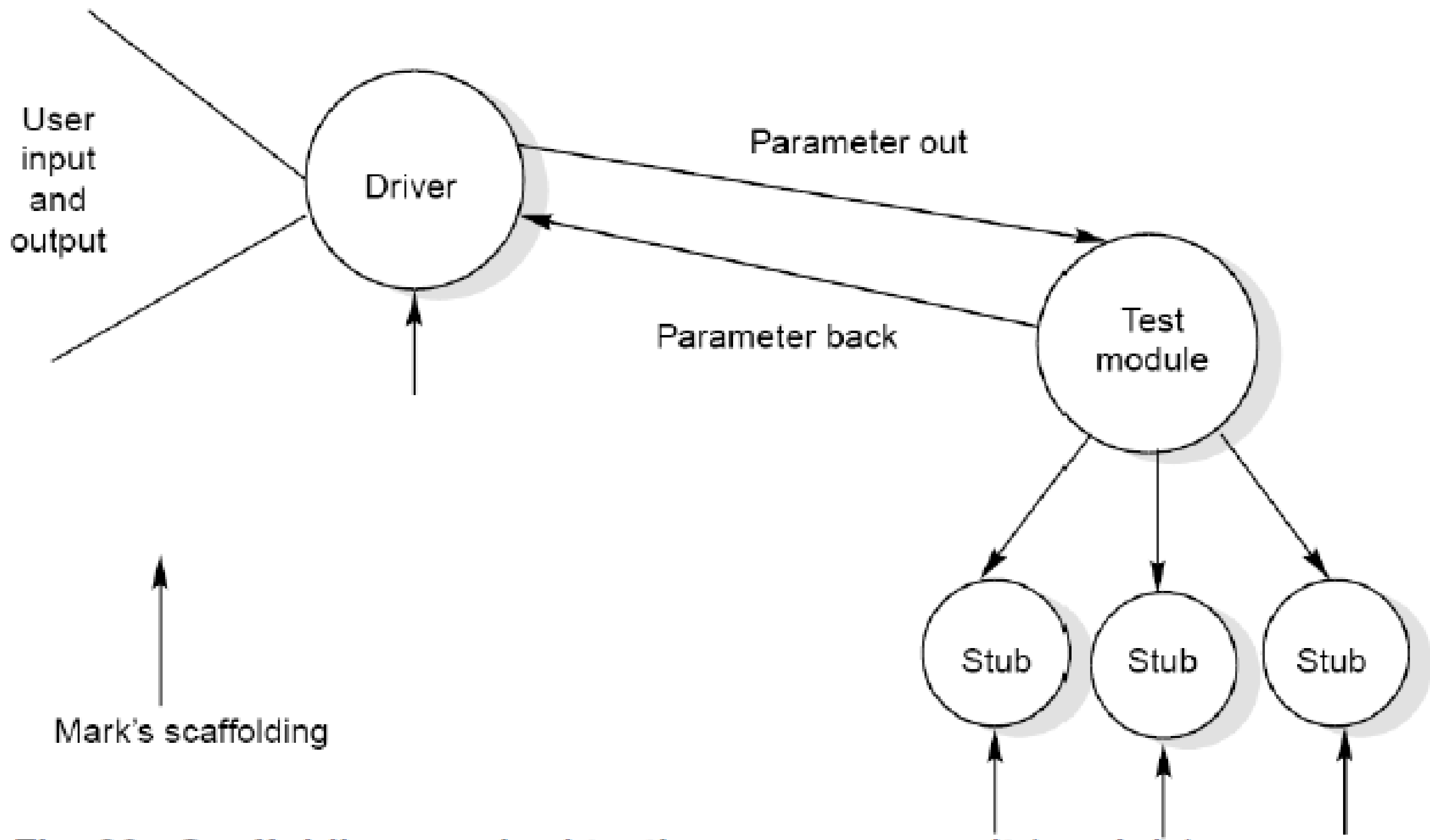


Fig. 29 : Scaffolding required testing a program unit (module)

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.

Integration Testing

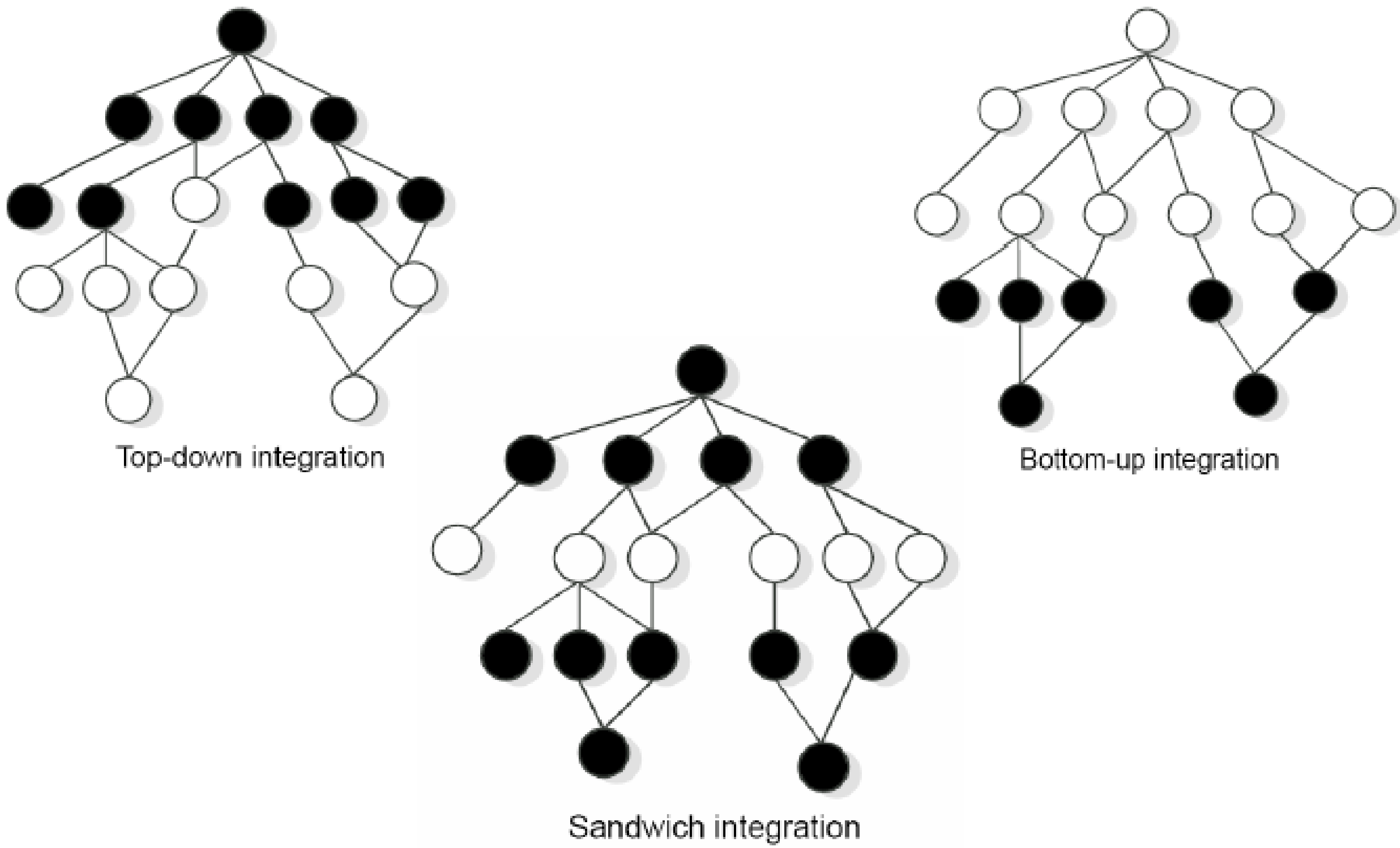


Fig. 30 : Three different integration approaches

System Testing

- Of the three levels of testing, the system level is closest 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.

System 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

Real-time testing

- **Real-time testing** is the process of testing real-time computer systems.
- Software testing is performed to detect and help correct bugs (errors) in computer software. Testing involves ensuring not only that the software is error-free but that it provides the required functionality to the user. Static and conventional methods of testing can detect bugs, but such techniques may not ensure correct results in real time software systems. Real-time software systems have strict timing constraints and have a deterministic behavior. These systems have to schedule their tasks such that the timing constraints imposed on them are met. Conventional static way of analysis is not adequate to deal with such timing constraints, hence additional real-time testing is important.

Real-time testing

- Test case design for real time testing can be proposed in four steps
- *Task testing:* In the very first step, each task is tested individually with conventional static testing. This testing is performed only to discover the errors in logic or syntax of the program. Order of the events doesn't matter as task testing doesn't deal with timing constraints and time properties of events.
- *Behavioral testing:* Using the system models designed with the help of automated testing tools, it is possible to simulate behavior of real time system and impact of concurrent external events on its behavior.

Real-time testing

- *Intertask testing*: Once the testing with the individual task is done, then task is supposed to be error free in coding and behavioral area. Time-related constraints are tested with intertask testing. To reveal the errors in communication, asynchronous tasks are tested with variable data rates and different payloads.
- *System testing :* In this testing, software and hardware are integrated and full range of system tests are conducted to discover errors, if any, during software and hardware interfacing.

Typical Test Case Parameters

- Test Case ID
- Test Scenario
- Test Case Description
- Test Steps
- Prerequisite
- Test Data
- Expected Result
- Test Parameters
- Actual Result
- Environment Information
- Comments

End of Chapter 8

Questions?