

Software Testing (CO4)

- Testing is a set of activities that can be planned in advance and conducted systematically. For this reason a template for software testing- a set of steps into which we can place specific test case design techniques and testing method should be defined for the software process.
- Testing have the following generic characteristics.
 - To perform effective testing, a software team should conduct effective formal technical reviews. By doing this many errors will be eliminated before testing commence.

Objective of Testing

- The testing objective is to test the code whereby there is a high probability of discovering all errors.
- This objective also demonstrates that the software functions are working according to software requirements specifications (SRS) with regard to functionality, features, facilities and performance.
- Testing objectives are
 - Testing is a process of executing a program with the intent of finding an error.
 - A good test case is one that has a high probability of finding an as yet undiscovered error

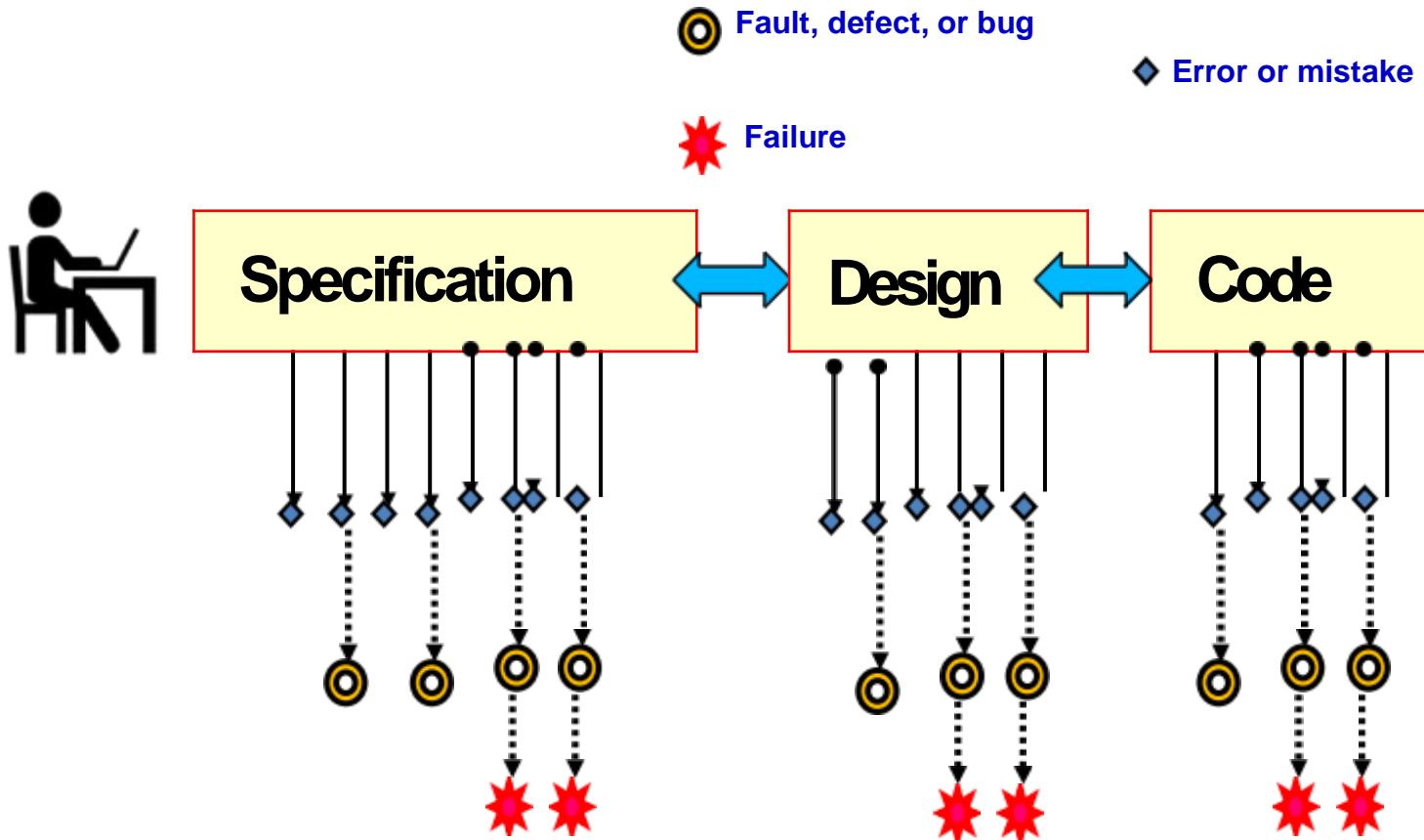
Some Terminologies

- Programming is human effort-intensive:
 - Therefore, inherently error prone.
- IEEE std 1044, 1993 defined errors and faults as synonyms :
- IEEE Revision of std 1044 in 2010 introduced finer distinctions:
 - To support more expressive communications, it distinguished between Errors and Faults

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.

Some Terminologies



A Few Error Facts

- Even experienced programmers make many errors:
 - Avg. 50 bugs per 1000 lines of source code
- Extensively tested software contains:
 - About 1 bug per 1000 lines of source code.
- Bug distribution:
 - 60% spec/design, 40% implementation.



Bug Source



How to Reduce Bugs?

- Review
- **Testing**
- Formal specification and verification
- Use of development process

How to Test?

- Check if the program behaved as expected.
- Input test data to the program.
- Observe the output:

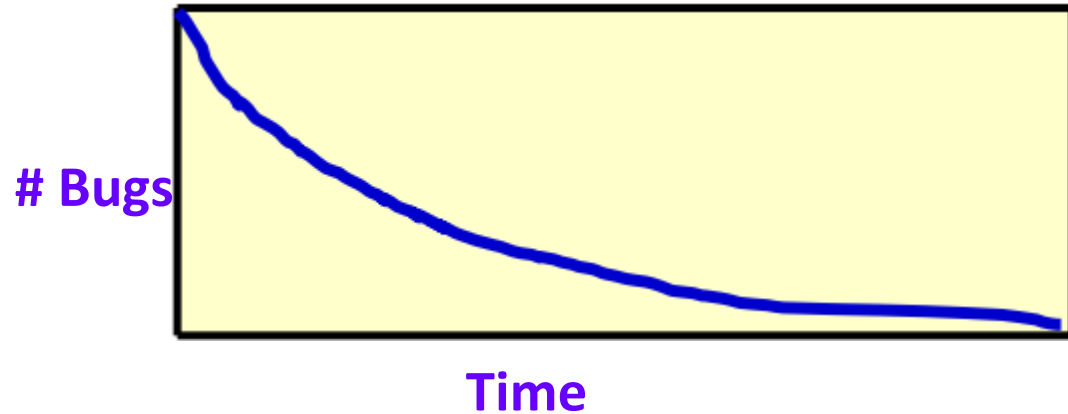
- Consumes the largest effort among all development activities:
 - Largest manpower among all roles
 - Implies more job opportunities
- About 50% development effort
 - But 10% of development time?
 - How?

Testing is getting more complex and sophisticated every year.

- Larger and more complex programs
- Newer programming paradigms
- Newer testing techniques
- Test automation

Test How Long?

One way:



- Another way:
 - Seed bugs... run test cases
 - See if all (or most) are getting detected

Test, Test Case and Test Suite (CO4)

- **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:

Verification and Validation

- **Verification:** Software should confirm to its.(are we building the product right?)
 - It is the process of determining whether the output of one phase of software development conforms to that of its previous phase. Thus verification is concerned with phase containment of errors
- **Validation:** Software should do what the user really require.(are we building the right product?)
 - It is the process of determining whether a fully developed system conforms to its requirements specification. the aim of validation is that the final product be error free.

Testing= Verification + Validation

Verification and Validation

- Verification is the process of determining:
 - Whether output of one phase of development conforms to its previous phase.
- Validation is the process of determining:
 - Whether a fully developed system conforms to its SRS document..

Verification and Validation

- Verification is the process of determining:
 - Whether output of one phase of development conforms to its previous phase.
- Validation is the process of determining:
 - Whether a fully developed system conforms to its SRS document..
- Verification is concerned with phase containment of errors:
 - Whereas, the aim of validation is that the final product is error free.

Verification and Validation

Verification and Validation Techniques

- Review
 - Simulation
 - Unit testing
 - Integration testing
- System testing

Verification and Validation

Verification	Validation
Are you building it right?	Have you built the right thing?
Checks whether an artifact conforms to its previous artifact.	Checks the final product against the specification.
Done by developers.	Done by Testers.
Static and dynamic activities: reviews, unit testing.	Dynamic activities: Execute software and check against requirements.

4 Testing Levels

- Software tested at 4 levels:

- Unit testing

- Integration testing

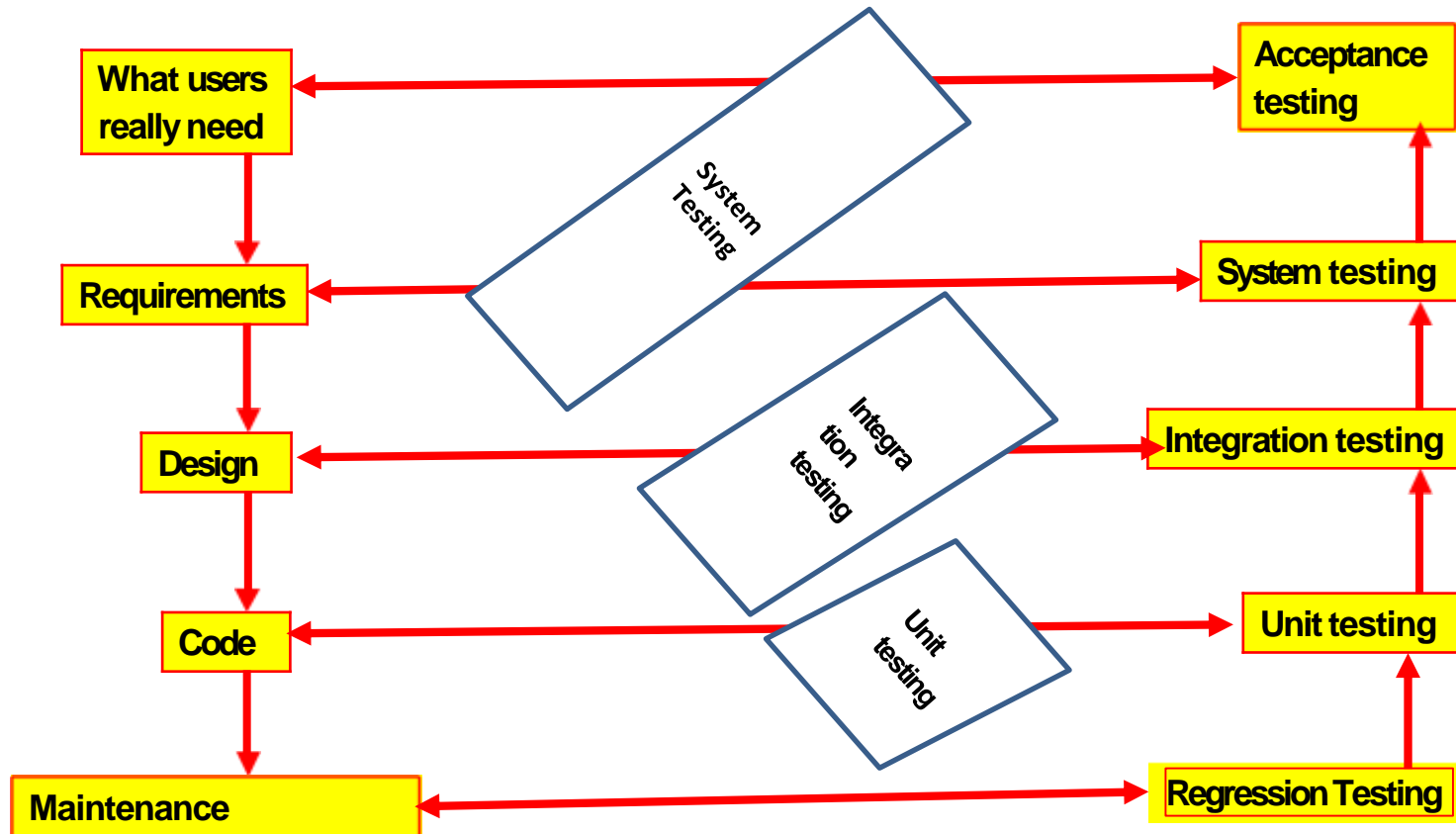
- System testing

- Regression testing

Levels of Testing

- **Unit testing**
 - Test each module (unit, or component) independently
 - **Mostly done by developers of the modules**
- **Integration and system testing**
 - Test the system as a whole
 - **Often done by separate testing or QA team**
- **Acceptance testing**
 - **Validation of system functions by the customer**

Levels of Testing



Overview of Activities During System and Integration Testing

- Test Suite Design
- Run test cases
- Check results to detect failures.
- Prepare failure list

Tester

- Debug to locate errors
- Correct errors.

Developer

Unit testing

- During unit testing, functions (or modules) are tested in isolation:
 - What if all modules were to be tested together (i.e. system testing)?
 - It would become difficult to determine which module has the error.

Integration Testing

- After modules of a system have been coded and unit tested:
 - Modules are integrated in steps according to an integration plan
 - The partially integrated system is tested at each integration step.

Integration and System Testing

- **Integration test evaluates a group of functions or classes:**
 - Identifies interface compatibility, unexpected parameter values or state interactions, and run-time exceptions
 - **System test tests working of the entire system**
- **Smoke test:**
 - System test performed daily or several times a week after every build.

Types of System Testing

- Based on types test:
 - **Functionality test**
 - **Performance test**
- Based on who performs testing:
 - **Alpha**
 - **Beta**
 - **Acceptance test**

Performance test

- Determines whether a system or subsystem meets its non-functional requirements:
 - **Response times**
 - **Throughput**
 - **Usability**
 - **Stress**
 - **Recovery**
 - **Configuration, etc.**

62

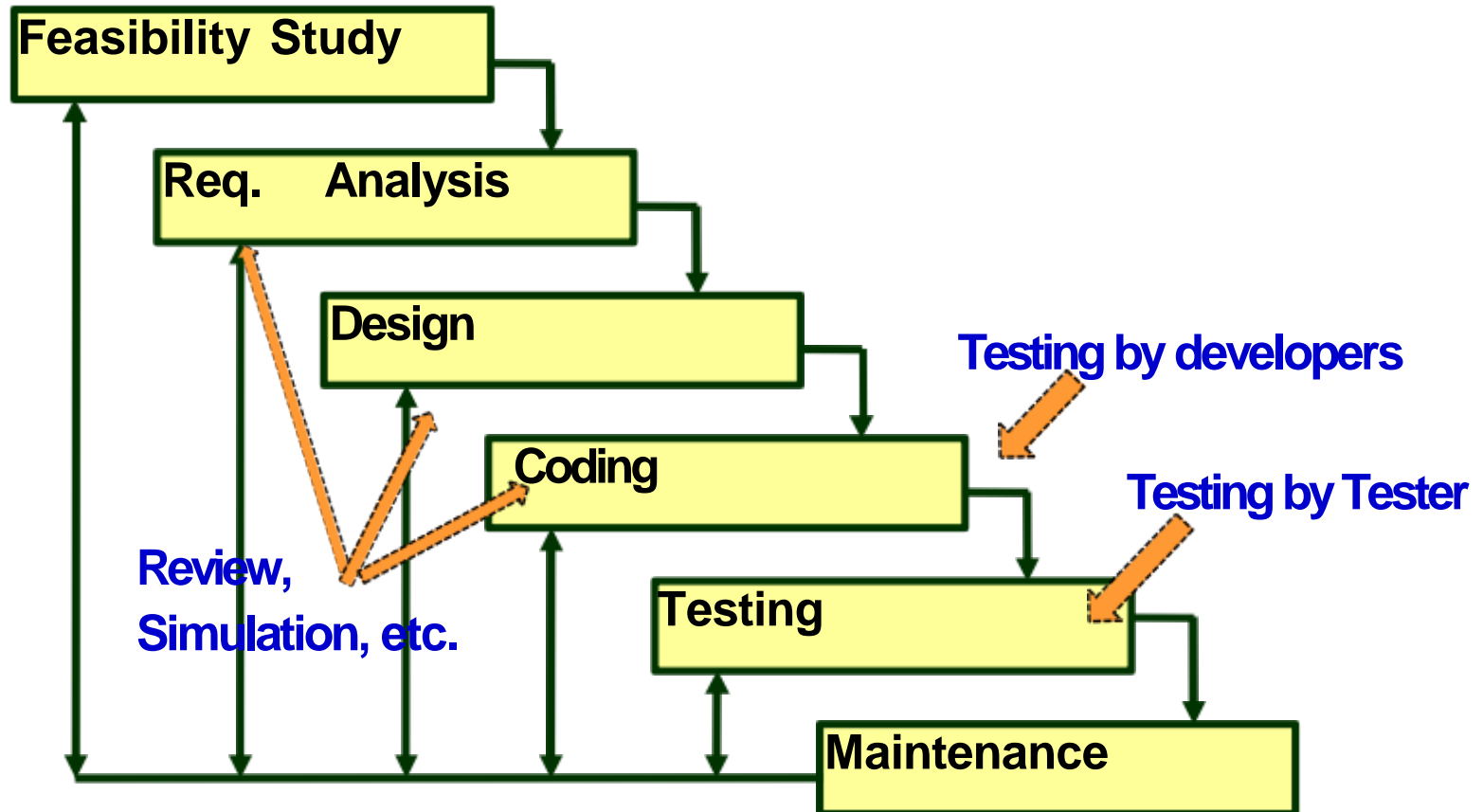
User Acceptance Testing

- User determines whether the system fulfills his requirements
 - **Accepts or rejects delivered system based on the test results.**

Who Tests Software?

- **Programmers:**
 - Unit testing
 - Test their own or other's programmer's code
- **Users:**
 - Usability and acceptance testing
 - Volunteers are frequently used to test beta versions
- **Test team:**
 - All types of testing except unit and acceptance
 - Develop test plans and strategy

64



Test Cases

- Each test case typically tries to establish correct working of some functionality:
 - Executes (covers) some program elements.
 - For certain restricted types of faults, fault-based testing can be used.

Test data versus test cases

- **Test data:**

- Inputs used to test the system

- **Test cases:**

- Inputs to test the system,
- State of the software, and
- The predicted outputs from the inputs

Test Cases and Test Suites

- A **test case** is a triplet [I,S,O]
 - I is the data to be input to the system,
 - S is the state of the system at which the data will be input,
 - O is the expected output of the system.
- Test a software using a set of carefully designed test cases:
 - The set of all test cases is called the **test suite**.

What are Negative Test Cases?

- **Purpose:**
 - Helps to ensure that the application gracefully handles invalid and unexpected user inputs and the application does not crash.
- **Example:**
 - If user types letter in a numeric field, it should not crash but politely display the message: **“incorrect data type, please enter a number...”**

Test Execution Example: Return Book

Test case [I,S,O]

- 1. Set the program in the required state:** Book record created, member record created, Book issued
- 2. Give the defined input:** Select renew book option and request renew for a further 2 week period.
- 3. Observe the output:**
Compare it to the expected output.

Sample: Recording of Test Case & Results

Test Case number

Test Case author

Test purpose Pre-condition:

Test inputs:

Expected outputs (if any):

Post-condition:

Test Execution history:

Test execution date

Person executing Test

Test execution result (s) : Pass/Fail

If failed : Failure information and fix status

Test Team- Human Resources

- **Test Planning:** Experienced people
- **Test scenario and test case design:** Experienced and test qualified people
- **Test execution:** semi-experienced to inexperienced
- **Test result analysis:** experienced people
- **Test tool support:** experienced people
- May include external people:
 - **Users**
 - **Industry experts**

Why Design of Test Cases?

- Exhaustive testing of any non-trivial system is impractical: –Input data domain is extremely large.
- Design an **optimal test suite**, meaning: –Of reasonable size, and
–Uncovers as many errors as possible.

Design of Test Cases

- If test cases are selected randomly:
 - Many test cases would not contribute to the significance of the test suite,
 - Would only detect errors that are already detected by other test cases in the suite.
- Therefore, the number of test cases in a randomly selected test suite:
 - Does not indicate the effectiveness of testing
 - Testing a system using a large number of randomly selected test cases:
 - Does not mean that most errors in the system will be uncovered.

Example

Find the maximum of two integers x and y .

- The code has a simple programming error:

- If $(x > y)$ $\text{max} = x$;

$\text{else max} = x$; // **should be $\text{max} = y$;**

- Test suite $\{(x=3, y=2); (x=2, y=3)\}$ can detect the bug,
- A larger test suite $\{(x=3, y=2); (x=4, y=3); (x=5, y=1)\}$ does not detect the bug.

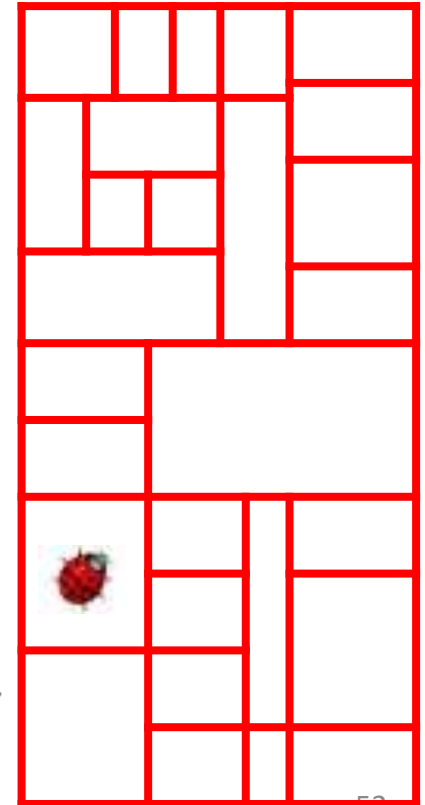
Test Plan

- Before testing activities start, a test plan is developed.
- The test plan documents the following:
 - Features to be tested
 - Features not to be tested
 - Test strategy
 - Test suspension criteria
 - stopping criteria
 - Test effort
 - Test schedule

Unit Testing

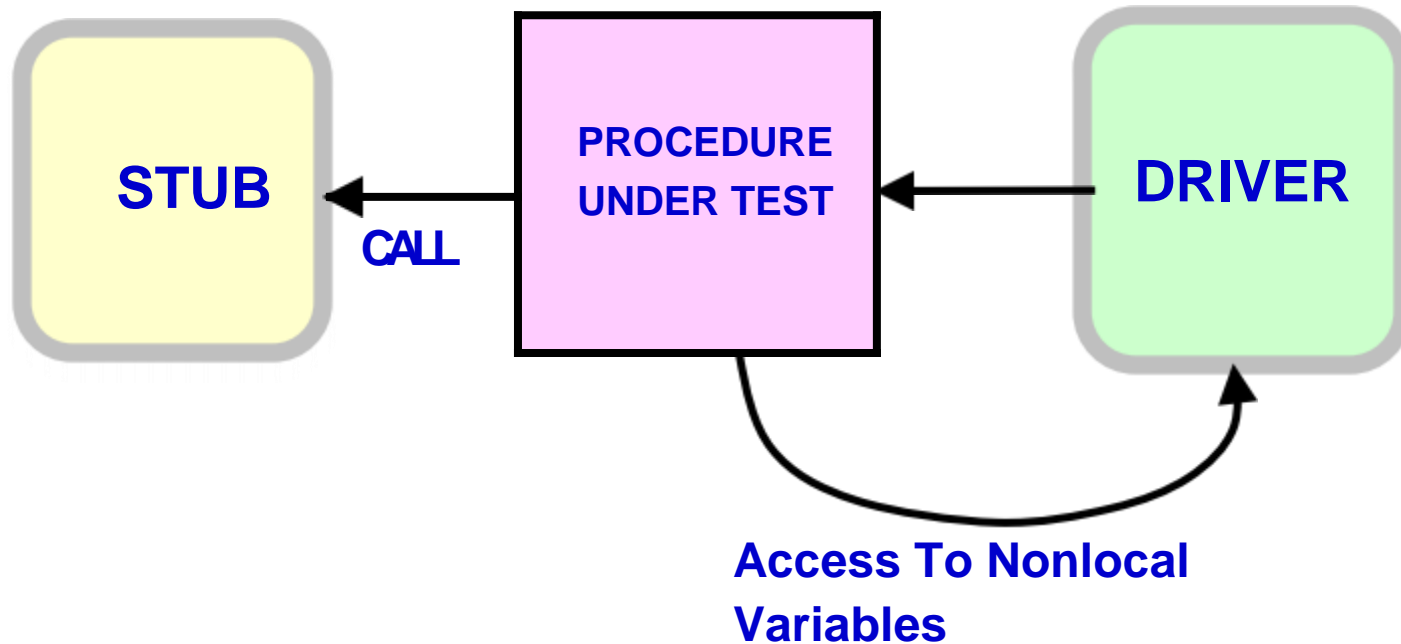
When and Why of Unit Testing?

- Unit testing carried out:
 - After coding of a unit is complete and it compiles successfully.
- Unit testing reduces debugging effort substantially.
- Without unit test:
 - Errors become difficult to track down.
 - Debugging cost increases substantially...



- Testing of individual methods, modules, classes, or components in isolation:
 - Carried out before integrating with other parts of the software being developed.
- Following support required for Unit testing:
 - Driver
 - Simulates the behavior of a function that calls and supplies necessary data to the function being tested.
 - Stub
 - Simulates the behavior of a function that has not yet been written.

Unit Testing



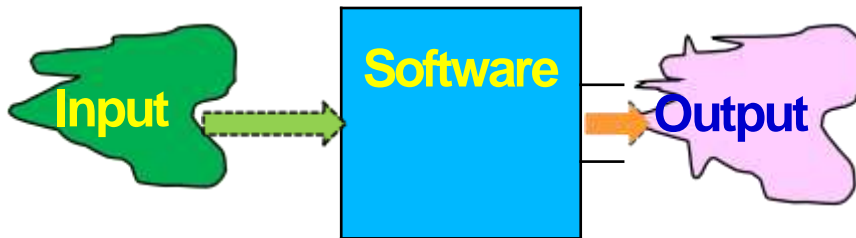
Design of Unit Test Cases

- There are essentially three main approaches to design test cases:
 - Black-box approach
 - White-box (or glass-box) approach
 - Grey-box approach

13

Black-Box Testing

- Test cases are designed using only **functional specification** of the software:
 - Without any knowledge of the internal structure of the software.



- Black-box testing is also known as **functional testing**.

What is Hard about BB Testing

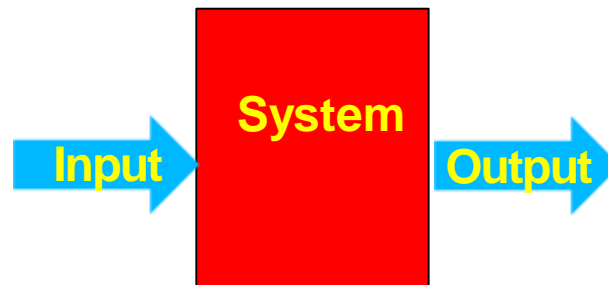
- Data domain is large
- A function may take multiple parameters:
 - We need to consider the combinations of the values of the different parameters

White-box Testing

- To design test cases:
 - Knowledge of internal structure of software necessary.
 - White-box testing is also called structural testing.

Black Box Testing

- Software considered as a black box:
 - Test data derived from the specification
 - **No knowledge of code necessary**
- Also known as:
 - Data-driven or
 - Input/output driven testing
- The goal is to achieve the thoroughness of exhaustive input testing:
 - With much less effort!!!!



Black-Box Testing

- Boundary Values analysis
- Robustness Testing
- Worst case Testing
- Equivalence Class Testing
- Decision Table Based Testing

Equivalence Class Partitioning


–Identify scenarios

–Examine the input data.

–Examine output

- Few guidelines for determining the equivalence classes can be given...

Guidelines to Identify Equivalence Classes

- If an input is a range, one valid and two invalid equivalence classes are defined. Example: 1 to 100 
- If an input is a set, one valid and one invalid equivalence classes are defined. Example: {a,b,c}
- If an input is a Boolean value, one valid and one invalid class are defined.

Example:

- **Area code:** input value defined between 10000 and 90000--- **range**
- **Password:** string of six characters ---**set**

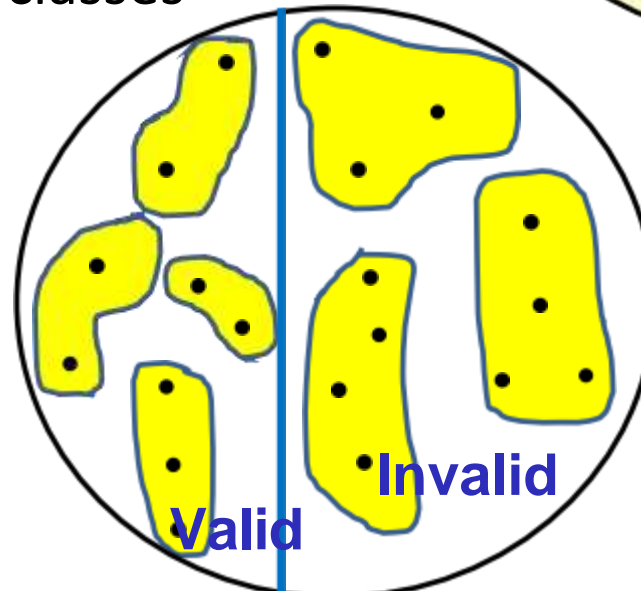
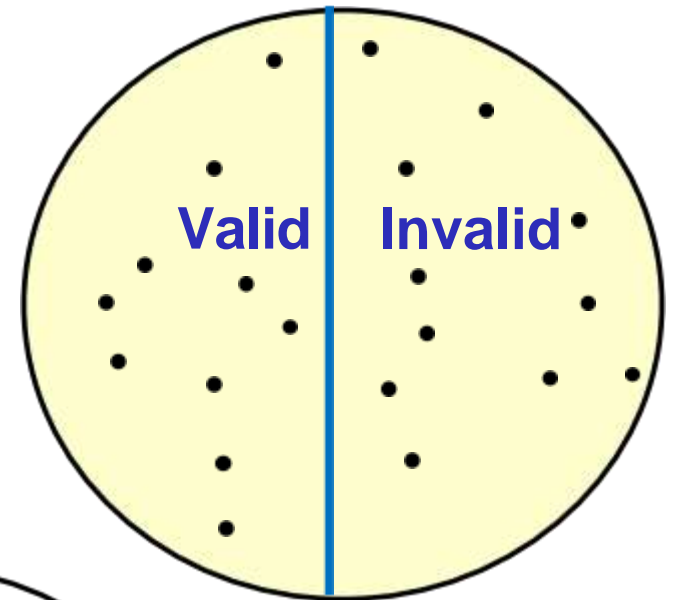
Example

- Given three sides, determine the type of the triangle:
 - Isosceles
 - Scalene
 - Equilateral, etc.

- Hint: scenarios correspond to output in this case.

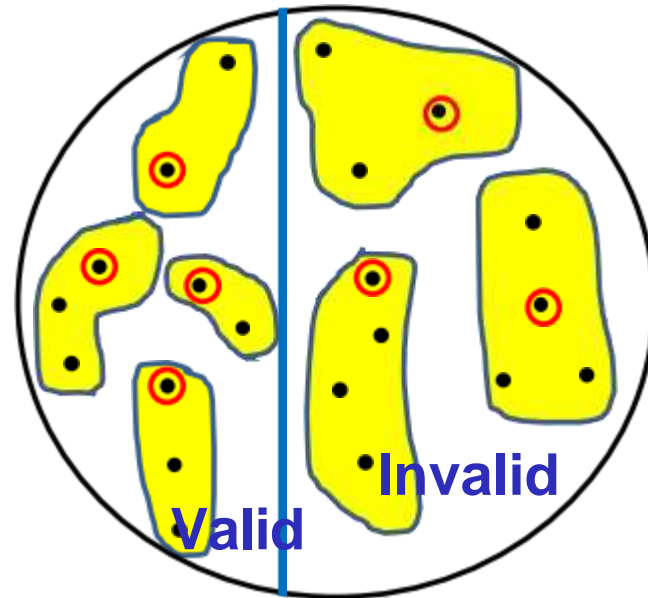
Equivalence Partitioning

- First-level partitioning:
 - Valid vs. Invalid test cases
- Further partition valid and invalid test cases into equivalence classes



Equivalence Partitioning

- Create a test case using at least one value from each equivalence class



Special Value Testing

- What are special values?
 - The tester has reasons to believe that execution with certain values may expose bugs:
 - General risk:** Example-- Boundary value testing
 - Special risk:** Example-- Leap year not considered

Boundary Value Analysis

- Some typical programming errors occur:

- At boundaries of equivalence classes

- Might be purely due to psychological factors.**



- Programmers often commit mistakes in the:

- Special processing at the boundaries of equivalence classes.**

Boundary Value Analysis

- Programmers may improperly use $<$ instead of $<=$
- Boundary value analysis:
 - **Select test cases at the boundaries of different equivalence classes.**

Boundary Value Analysis: Guidelines

- If an input is a range, bounded by values a and b:
 - Test cases should be designed with value a and b, just above and below a and b.
- **Example 1:** Integer D with input range [-3, 10],
 - test values: -3, 10, 11, -2, 0
- **Example 2:** Input in the range: [3,102]
 - test values: 3, 102, -1, 200, 5

Boundary Value Testing Example

- Process employment applications based on a person's age.

0-16	Do not hire
16-18	May hire on part time basis
18-55	May hire full time
55-99	Do not hire

- Notice the problem at the boundaries.
 - Age "16" is included in two different equivalence classes (as are 18 and 55).

Boundary Value Testing: Code Example

- If (applicantAge >= 0 && applicantAge <=16) hireStatus="NO";
- If (applicantAge >= 16 && applicantAge <=18) hireStatus="PART";
- If (applicantAge >= 18 && applicantAge <=55) hireStatus="FULL";
- If (applicantAge >= 55 && applicantAge <=99) hireStatus="NO";

Boundary Value Testing Example (cont)

- Corrected boundaries:

0–15

Don't hire

16–17

Can hire on a part-time basis only

18–54

Can hire as full-time employees

55–99

Don't hire

- What about ages -3 and 101?
- The requirements do not specify how these values should be treated.

Boundary Value Testing Example (cont)

- The code to implement the corrected rules is:

If (applicantAge >= 0 && applicantAge <=15)

hireStatus="NO";

If (applicantAge >= 16 && applicantAge <=17)

hireStatus="PART";

If (applicantAge >= 18 && applicantAge <=54)

hireStatus="FULL";

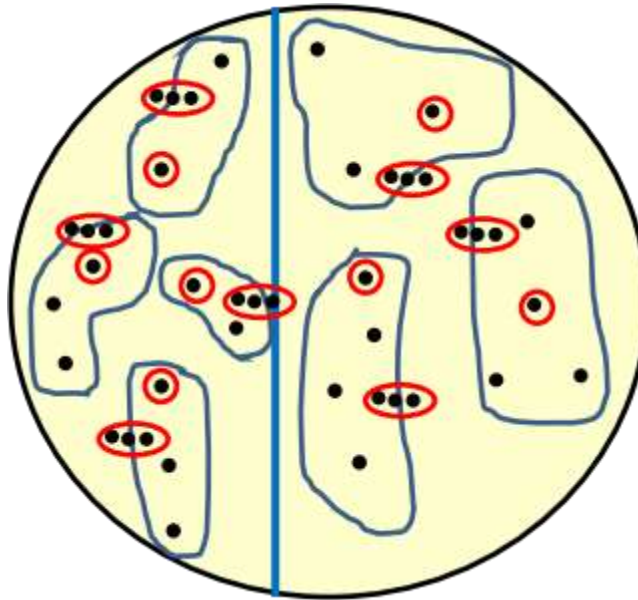
If (applicantAge >= 55 && applicantAge <=99)

hireStatus="NO";

- Special values on or near the boundaries in this example are {-1, 0, 1}, {14, 15, 16}, {17, 18, 19}, {54, 55, 56}, and {98, 99, 100}.

Boundary Value Analysis

- Create test cases to test boundaries of equivalence classes



Example 1

- For a function that computes the square root of an integer in the range of 1 and 5000:
 - Test cases must include the values:
 $\{0, 1, 2, 4999, 5000, 5001\}$.



Example 2

- Consider a program that reads the “age” of employees and computes the average age.

input (ages) → Program → output: average age

Assume valid age is 1 to 150



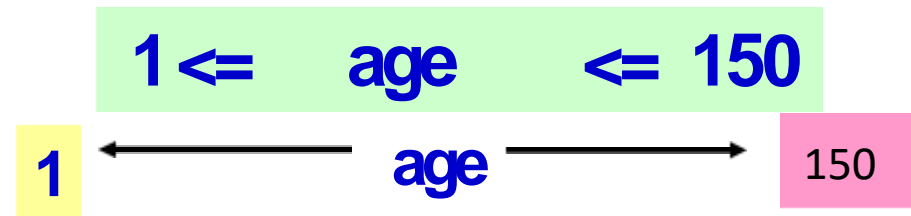
- How would you test this?
 - How many test cases would you generate?
 - What type of test data would you input to test this program?

Boundaries of the inputs

The “basic” boundary value testing would include 5 test cases:

1. - at minimum boundary
2. - immediately above minimum
3. - between minimum and maximum (nominal)
4. - immediately below maximum
5. - at maximum boundary

predict-longevity(age)



Boundary Value Testing Example (cont)

How many test cases for the example ?

- Test input values? :

- | | | |
|-----|----------------------|-------------------------------------|
| 1 | at the minimum | <code>predict-longevity(age)</code> |
| 2 | at one above minimum | |
| 45 | at middle | |
| 149 | at one below maximum | |
| 150 | at maximum | |

Boundary Value Testing Example (cont)

How many test cases for the example ?

- Test input values? :

- 1 at the minimum predict-longevity(age)
- 2 at one above minimum
- 45 at middle
- 149 at one below maximum
- 150 at maximum

65

Decision table based testing

- Applicable to requirements involving conditional actions. • This is represented as a decision table:

–Conditions = inputs

–Actions = outputs

–Rules =test cases

- Assume independence of inputs

- Example

–If c1 AND c2 OR c3 then A1

	Rule1	Rule2	Rule3	Rule4
Condition1	Yes	Yes	No	No
Condition2	Yes	X	No	X
Condition3	No	Yes	No	X
Condition4	No	Yes	No	Yes
Action1	Yes	Yes	No	No
Action2	No	No	Yes	No
Action3	No	No	No	Yes

Example

Conditions											
C1: $a < b+c?$	F	T	T	T	T	T	T	T	T	T	T
C2: $b < a+c?$	-	F	T	T	T	T	T	T	T	T	T
C3: $c < a+b?$	-	-	F	T	T	T	T	T	T	T	T
C4: $a=b?$	-	-	-	T	T	T	T	F	F	F	F
C5: $a=c?$	-	-	-	T	T	F	F	T	T	F	F
C6: $b=c?$	-	-	-	T	F	T	F	T	F	T	F
Actions											
A1: Not a Triangle	X	X	X								
A2: Scalene											X
A3: Isosceles							X		X	X	
A4: Equilateral				X							
A5: Impossible					X	X		X			

Example (cont)

Case ID	a	b	c	Expected Output
DT1	4	1	2	Not a Triangle
DT2	1	4	2	Not a Triangle
DT3	1	2	4	Not a Triangle
DT4	5	5	5	Equilateral
DT5	?	?	?	Impossible
DT6	?	?	?	Impossible
DT7	2	2	3	Isosceles
DT8	?	?	?	Impossible
DT9	2	3	2	Isosceles
DT10	3	2	2	Isosceles
DT11	3	4	5	Scalene

White Box Testing(Structural Testing)

- It is a complementary approach to functional testing. It permits us to examine the internal structure of the program.
- Also called **Structural Testing**

1. Path Testing
2. Cyclomatic Complexity
3. Graph Matrices
4. Data Flow Testing
5. Mutation Testing

Why Both BB and WB Testing?

Black-box

- Impossible to write a test case for every possible set of inputs and outputs
- Some code parts may not be reachable
- **Does not tell if extra functionality has been implemented.**

White-box

- Does not address the question of whether a program matches the specification
- Does not tell if all functionalities have been implemented
- **Does not uncover any missing program logic**

Path Testing

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.

Step 1:

Generate Control flow graph of a program.

Step 2:

Draw a Decision to Decision path (**DD-path**) graph from the control flow graph.

it is a directed graph. node are sequences of statements and edges represent control flow between them.

DD path is used to find **Independent path**.

Independent path: it is used to ensure

1. Every statement in the program has been executed at least once.
2. Every branch has been exercised for true and false condition.

Path Testing

Example

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

Example

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

Fig. : Code of triangle classification problem

Example

Solution :

Flow graph of triangle problem is:

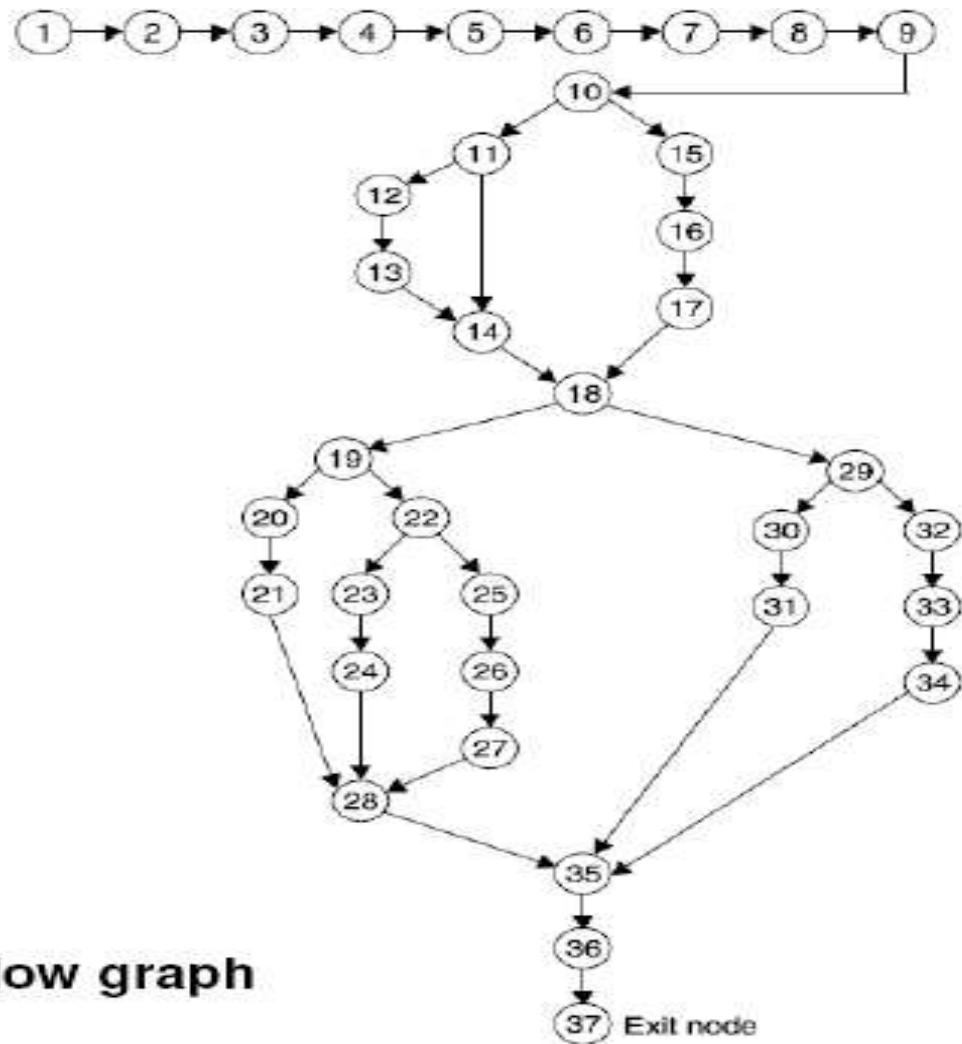


Fig. : Program flow graph

Example

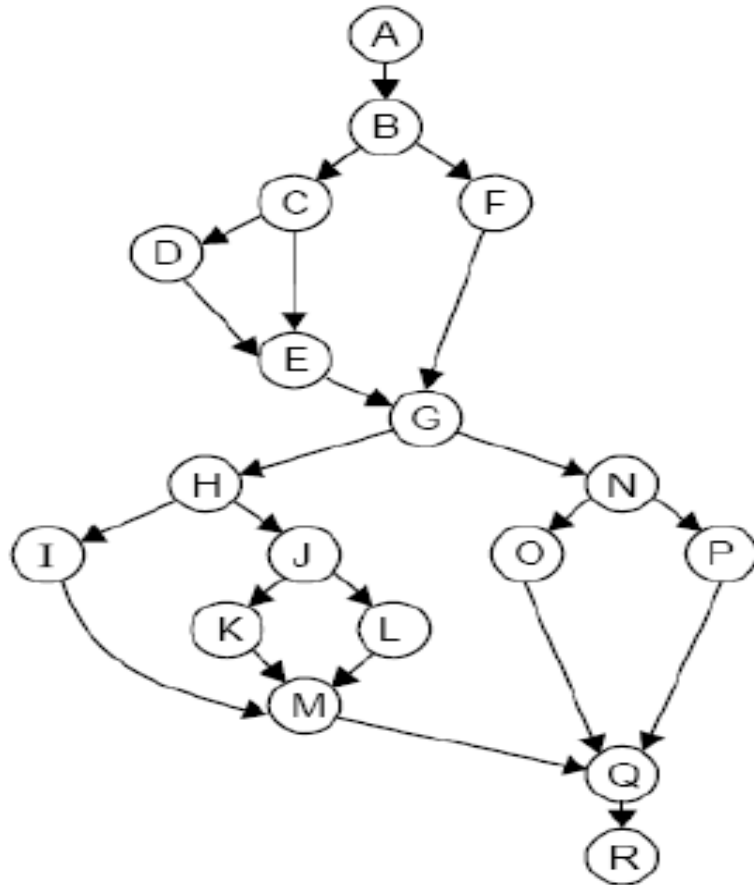
The mapping table for DD path graph is:

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

Fig. DD Path graph

Example

DD Path graph is given in Fig.



Independent paths are:

- (i) ABFGNPQR
- (ii) ABFGNOQR
- (iii) ABCEGNPQR
- (iv) ABCDEGNOQR
- (v) ABFGHIMQR
- (vi) ABFGHJKMQR
- (vii) ABFGHJMQR

Fig.

DD Path graph

Path Testing?

Path Testing?

Alpha, Beta and Acceptance Testing (CO4)

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.

Alpha, Beta and Acceptance Testing (CO4)

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.

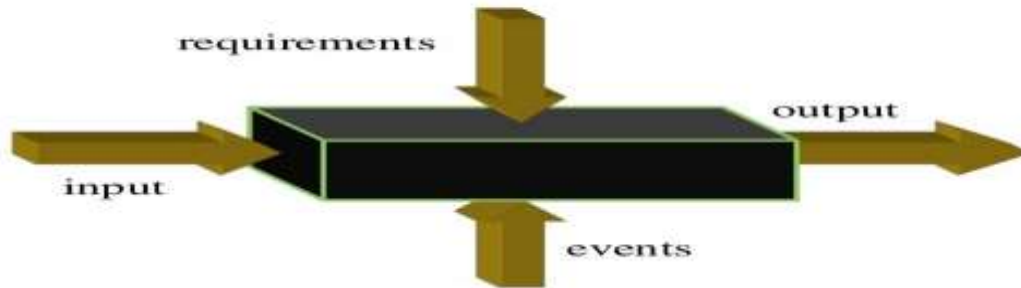
The terms **alpha** and **beta testing** are used when the software is developed as a product for anonymous customers.

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.

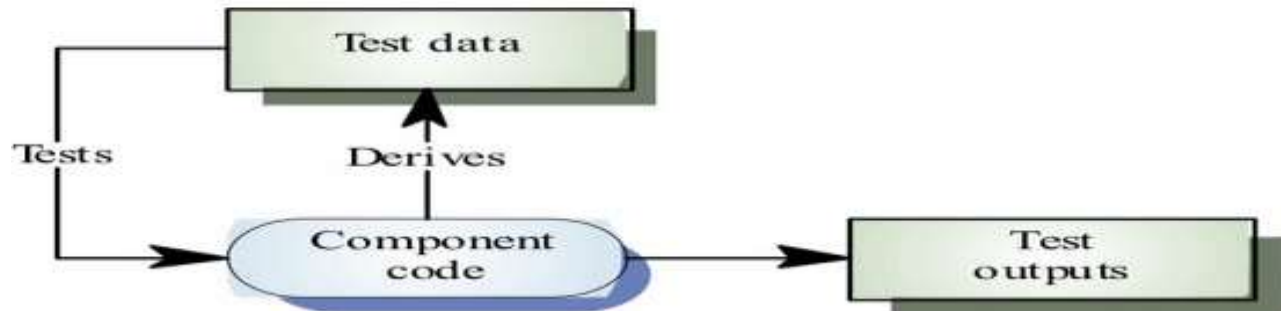
Beta Tests are conducted by the customers / end users at their sites. Unlike alpha testing, developer is not present here. Beta testing is conducted in a real environment that cannot be controlled by the developer.

Testing Methodology

Black box testing

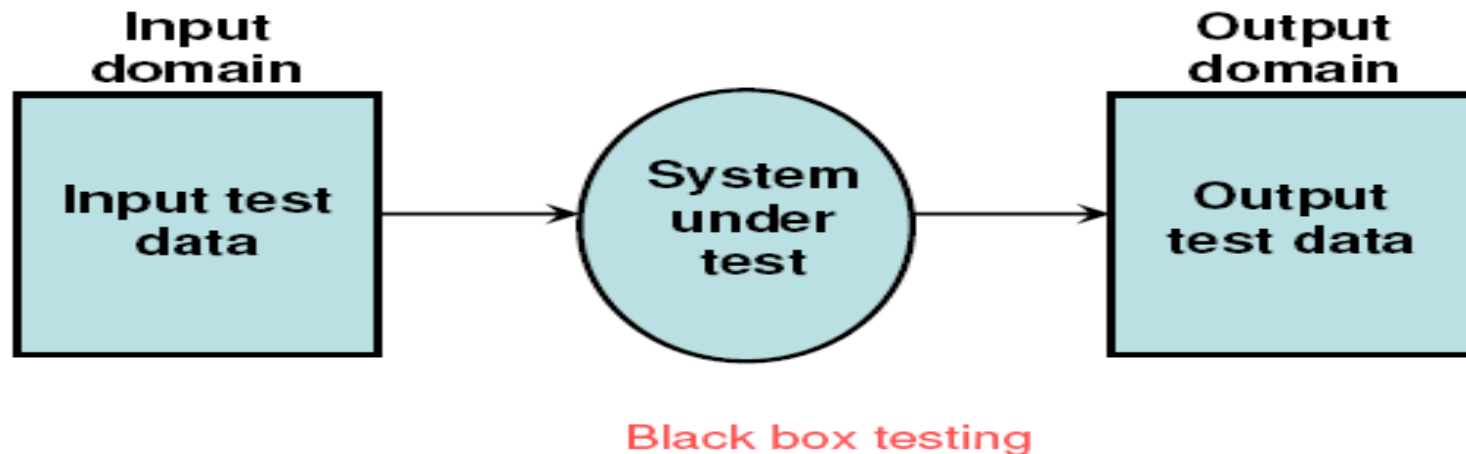


White box testing



Black Box Testing(CO4)

- Test cases are designed from an examination of the input/output values only and no knowledge of design, or code is required.
- Also called **Functional Testing**



Functional Testing

The following are the main approaches to designing black box test cases.

1. Boundary Values analysis
2. Robustness Testing
3. Worst case Testing
4. Equivalence Class Testing
5. Decision Table Based Testing

Boundary Value Analysis

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

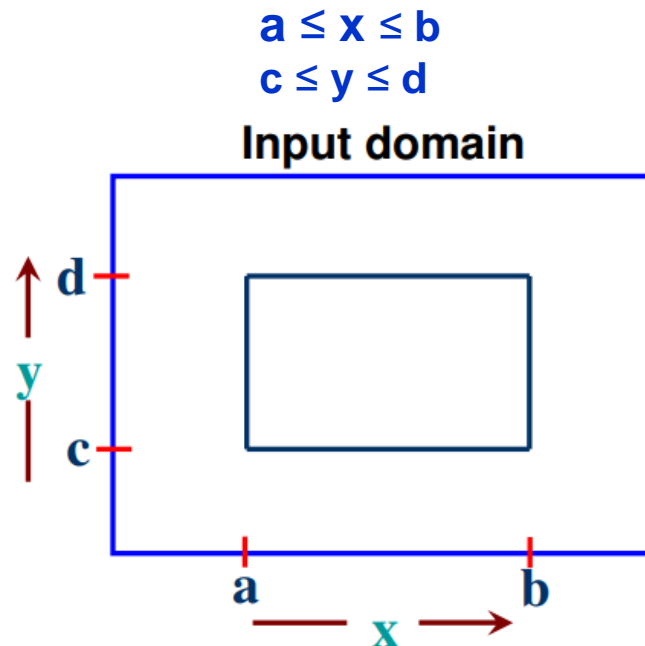


Fig. : Input domain for program having two input variables

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

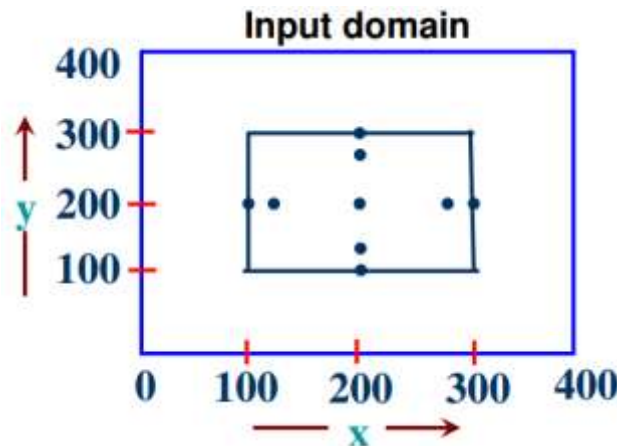


Fig. : Input domain of two variables x and y with boundaries [100,300] each

Example

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.

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 $a=0$

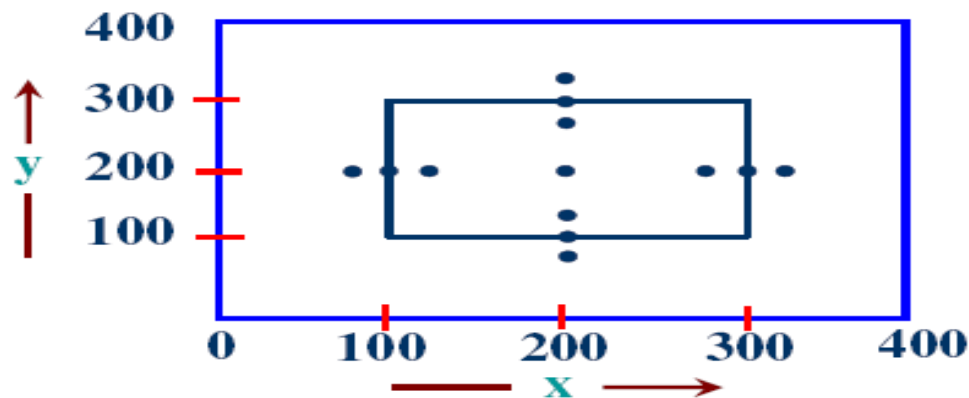
Example

The boundary value test cases are :

Test Case	<i>a</i>	<i>b</i>	<i>c</i>	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

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



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.

Equivalence Class Testing

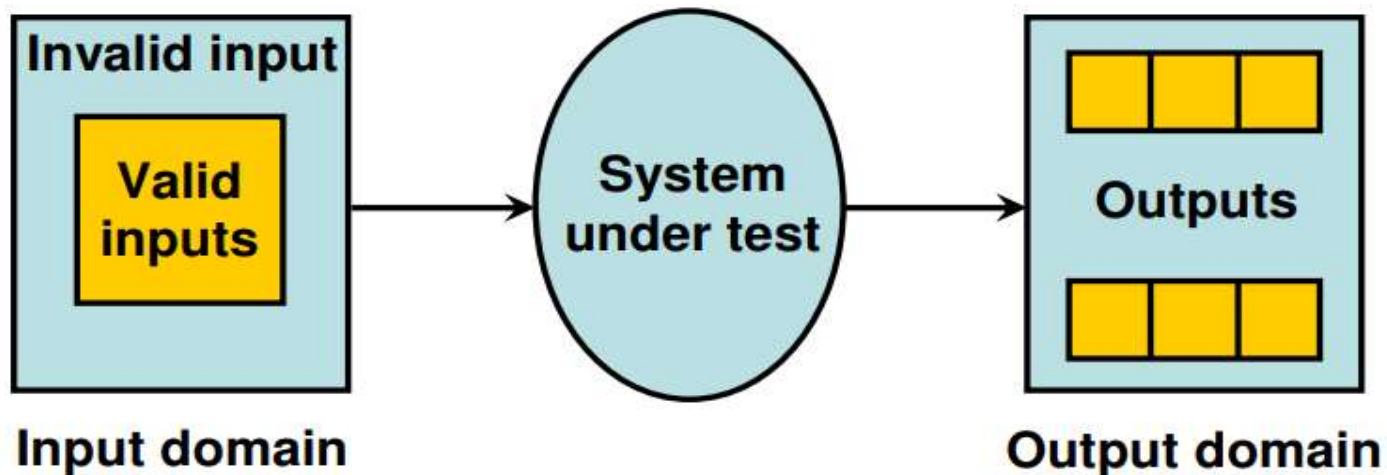
In this method, input domain of a program is partitioned into a finite number of equivalence classes such that one can reasonably assume, but not be absolutely sure, that the test of a representative value of each class is equivalent to a test of any other value

Two steps are required

- ❖ The equivalence classes are identified by taking each input condition and partitioning it into valid and invalid classes
- ❖ 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



Path Testing

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.

Step 1:

Generate Control flow graph of a program.

Step 2:

Draw a Decision to Decision path (**DD-path**) graph from the control flow graph.

it is a directed graph. node are sequences of statements and edges represent control flow between them.

DD path is used to find **Independent path**.

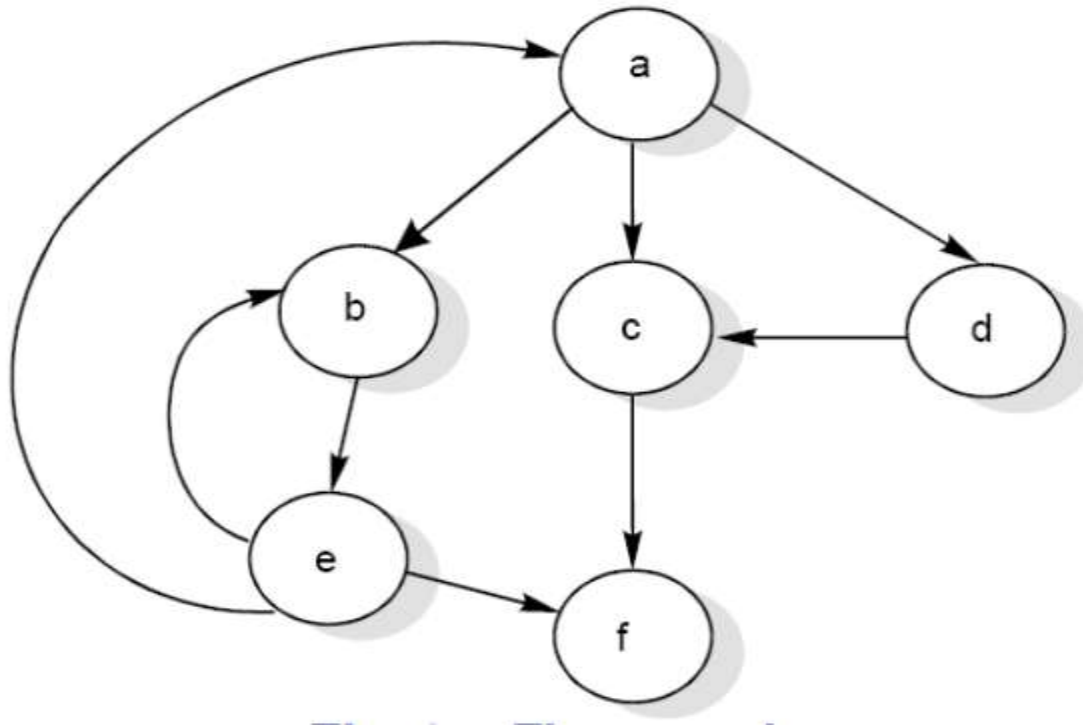
Independent path: it is used to ensure

1. Every statement in the program has been executed at least once.
2. Every branch has been exercised for true and false condition.

Cyclomatic Complexity

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

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



Cyclomatic Complexity

The value of cyclomatic complexity can be calculated as :

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

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

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

Path 1 : a c f

Path 2 : a b e f

Path 3 : a d c f

Path 4 : a b e a c f or a b e a b e f

Path 5 : a b e b e f

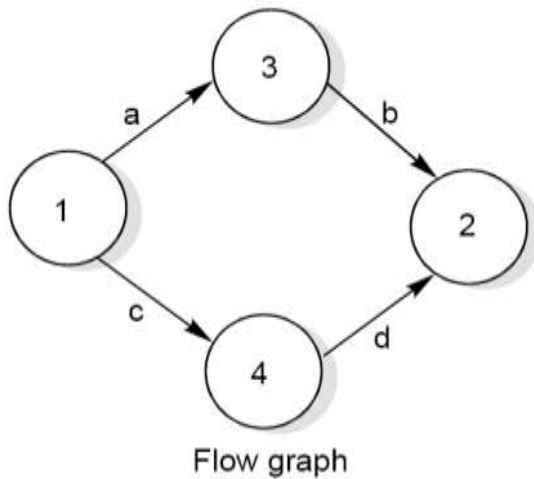
Cyclomatic Complexity

Several properties of cyclomatic complexity are stated below:

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

Graph Matrices

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



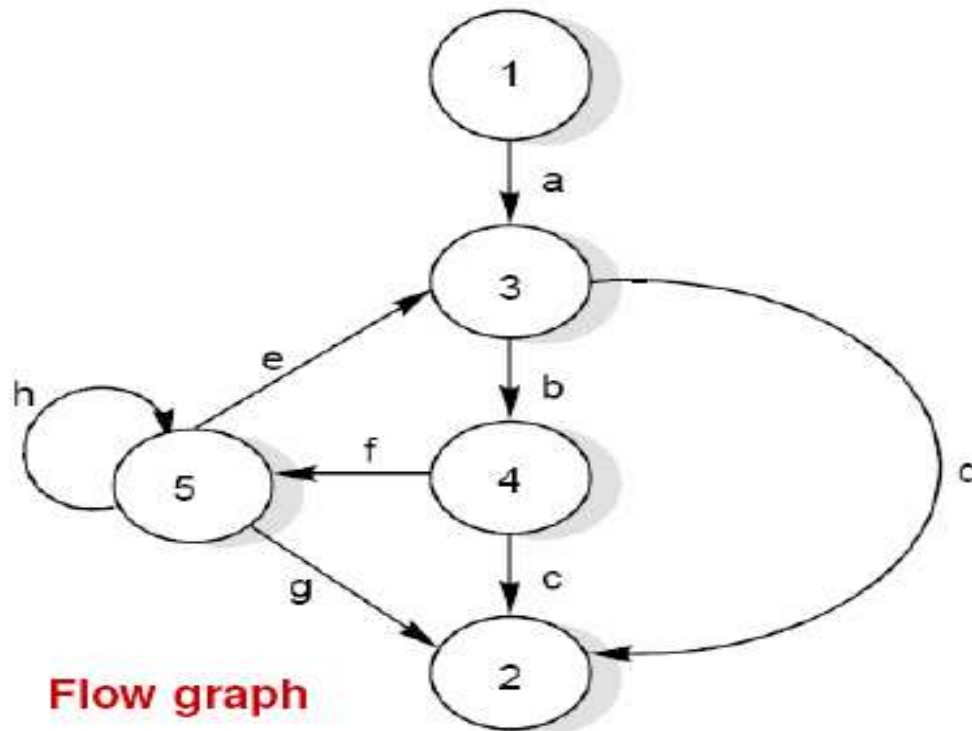
	1	2	3	4
1			a	c
2				
3		b		
4		d		

Graph Matrix

Example

Example

Consider the flow graph shown in the Fig. 26 and draw the graph & connection matrices.



Example

Solution

The graph & connection matrices are given below :

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

Graph Matrix (A)

	1	2	3	4	5
1			1		
2					
3		1		1	
4		1			1
5		1	1		1

Connection Matrix

Connections

$$1 - 1 = 0$$

$$2 - 1 = 1$$

$$2 - 1 = 1$$

$$3 - 1 = 2$$

$$4 + 1 = 5$$

Data Flow Testing

Data flow testing is another form of structural testing. It has nothing to do with data flow diagrams.

- i. Statements where variables receive values.
- ii. Statements where these values are used or referenced.

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

- i. A variable is defined but not used/ referenced.
- ii. A variable is used but never defined.
- iii. A variable is defined twice before it is used.

Data Flow Testing

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

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

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

Data Flow Testing

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

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

Mutation testing

- In this, software is first tested:
 - Using an initial test suite designed using white-box strategies we already discussed.
- After the initial testing is complete,
 - Mutation testing is taken up.
- **The idea behind mutation testing:**
 - Make a few arbitrary small changes to a program at a time.**

Mutation testing

Main Idea

- Insert faults into a program:
 - Check whether the test suite is able to detect these.
 - This either validates or invalidates the test suite.

Mutation Testing Terminology

- Each time the program is changed:
 - It is called a **mutated program**
 - The change is called a **mutant**.

Mutation Testing

- A mutated program:
 - Tested against the full test suite of the program.
- If there exists at least one test case in the test suite for which:
 - A mutant gives an incorrect result,
 - Then the mutant is said to be **dead**.
- If a mutant remains alive ---even after all test cases have exhausted,
 - The test suite is enhanced to kill the mutant.
- The process of generation and killing of mutants:
 - Can be automated by predefining a set of primitive changes that can be applied to the program.

Mutation Testing

Example primitive changes to a program:

- Deleting a statement
- Altering an arithmetic operator,
- Changing the value of a constant,
- Changing a data type, etc.

- **Deletion of a statement**
- Boolean:
 - Replacement of a statement with another
eg. == and >=, < and <=
 - Replacement of boolean expressions with true or false eg. a || b
with true
- **Replacement of arithmetic operator**
eg. * and +, / and -
- **Replacement of a variable** (ensuring same scope/type)

Mutation testing

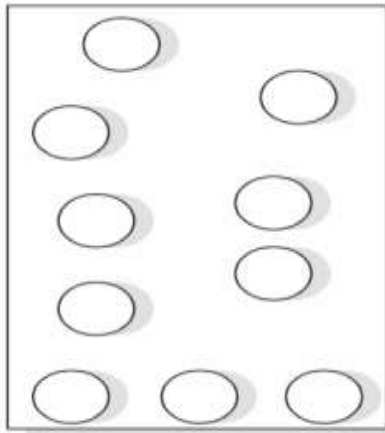
Disadvantage:

- it is computationally very expensive, due to a large number of possible mutants can be generated.
- it is not suitable for manual testing. Mutation testing should be used in conjunction of some testing tool which would run all the test cases automatically. Version

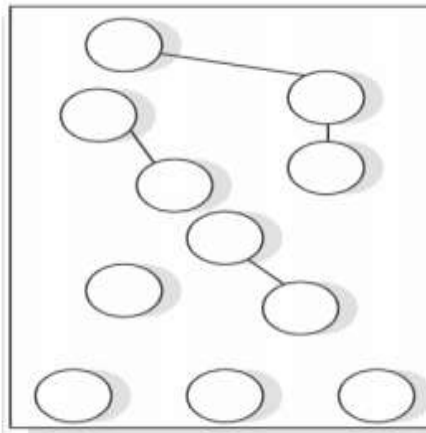
Level Of Testing

There are 3 levels of testing:

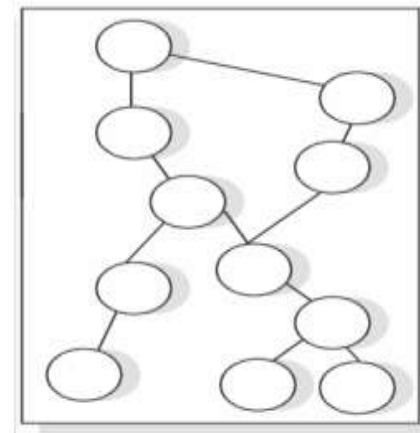
- i. Unit Testing
- ii. Integration Testing
- iii. System Testing



UNIT TESTING



INTEGRATION TESTING



SYSTEM TESTING

Unit Testing (CO4)

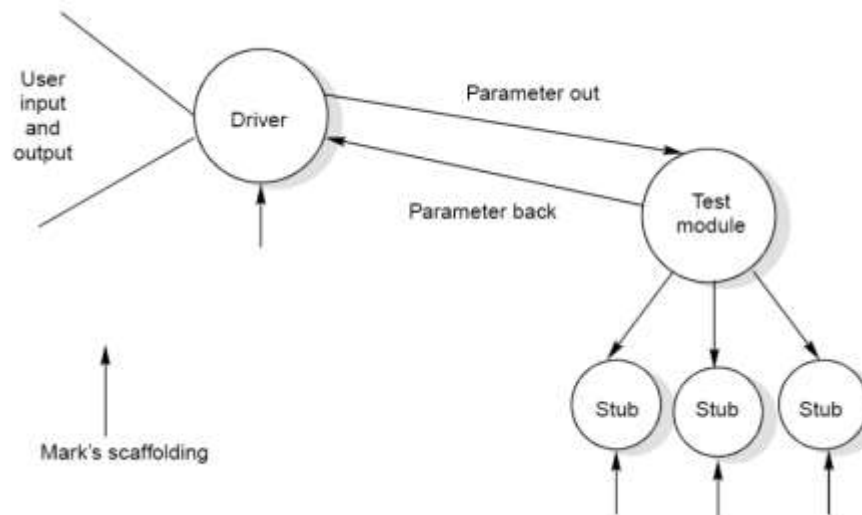
- It is also called module testing.
- Follows a white box testing(logic of program).
- Done by developer.
- It is the testing of different units (or modules) of a system in isolation.
- steps are needed in order to be able to test the module:
 - The procedures belonging to other modules that the module under test calls.
 - Nonlocal data structures that the module accesses.
 - A procedure to call the functions of the module under test with appropriate parameters.

Stub and Driver

Stubs and drivers are designed to provide the complete environment for a module.

Stub: it is a dummy procedure that has the same I/O parameters as the given procedure but has a highly simplified behavior.

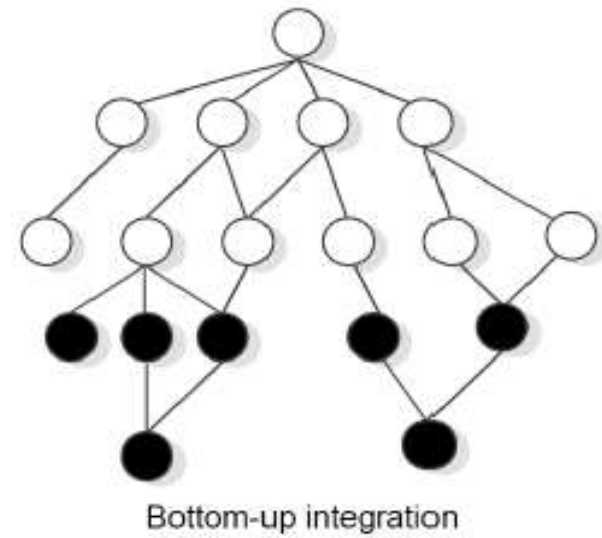
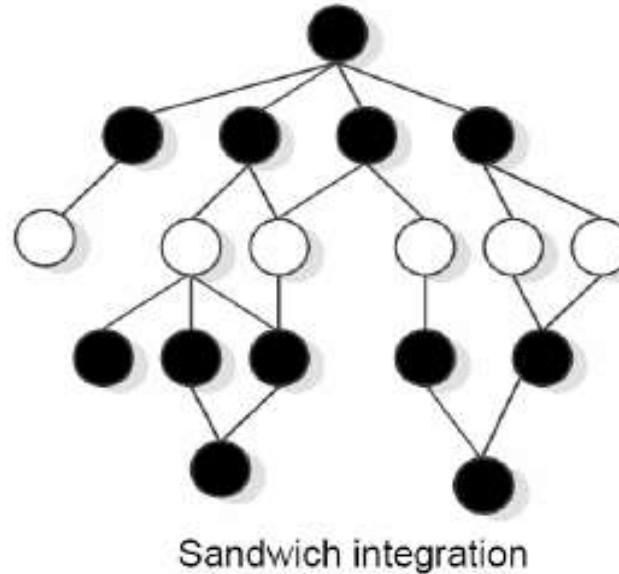
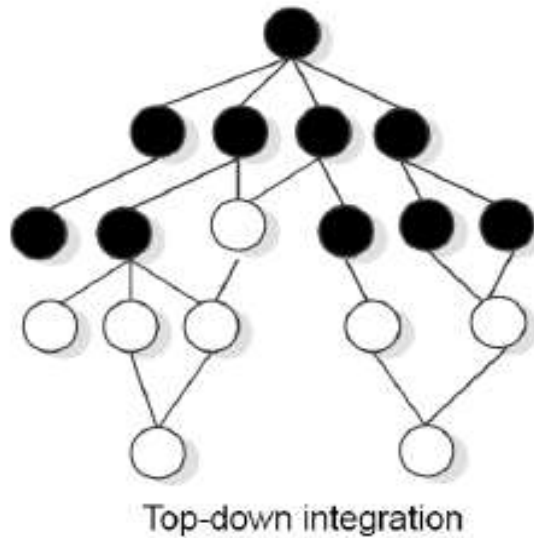
Driver : it contain the nonlocal data structures accessed by the module under test, and would also have the code to call the different functions of the module with appropriate parameter values.



Integration Testing (CO4)

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

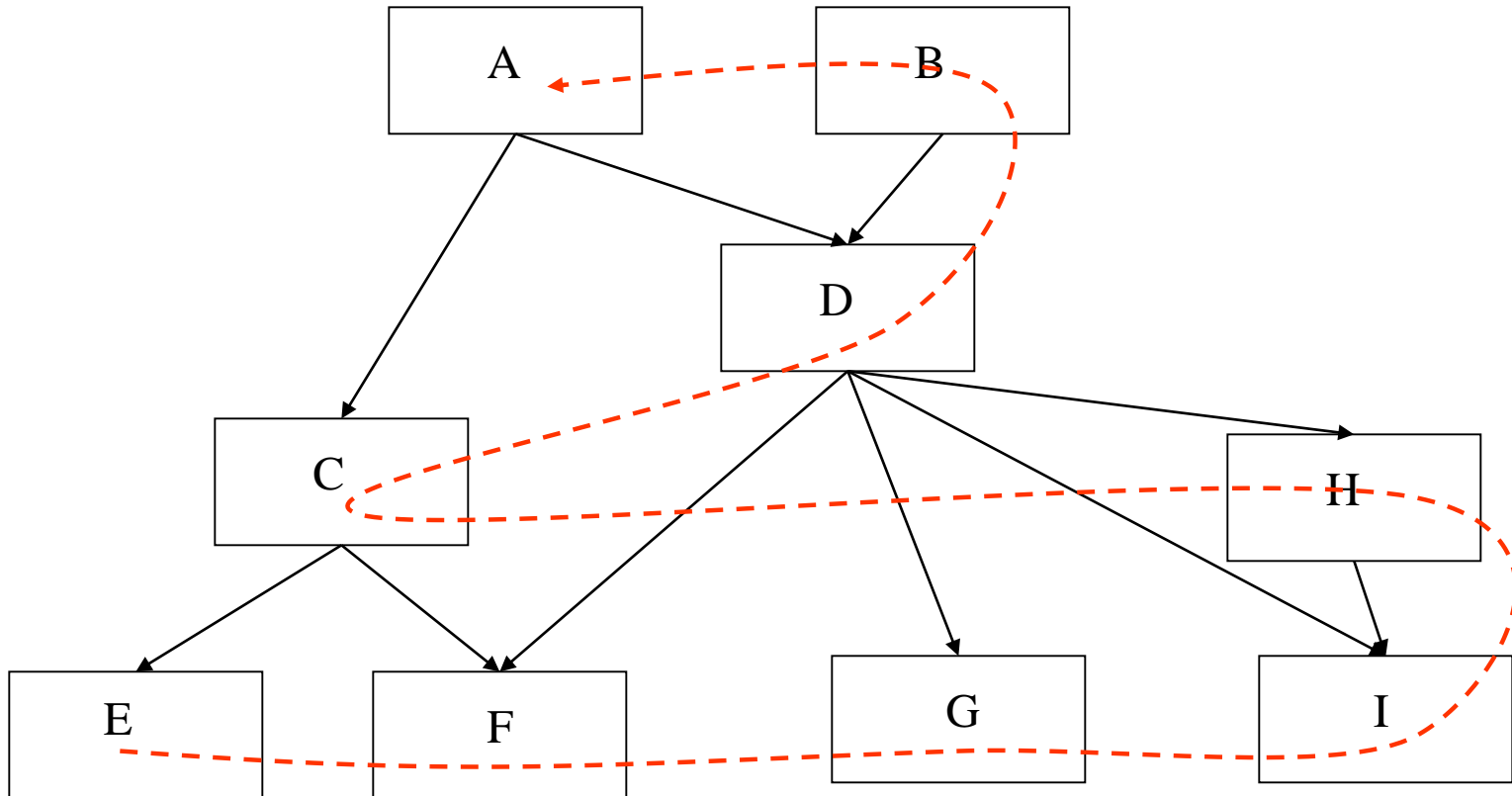


Three different integration approaches

Bottom-Up Integration

- Only terminal modules (i.e., the modules that do not call other modules) are tested in isolation
- Modules at higher level are tested using the previously tested lower level modules
- Non-terminal modules are not tested in isolation
- Requires a module **driver** for each module to feed the test case input to the interface of the module being tested
 - However, **stubs are not needed** since we are starting with the terminal modules and use already tested modules when testing modules in the lower levels

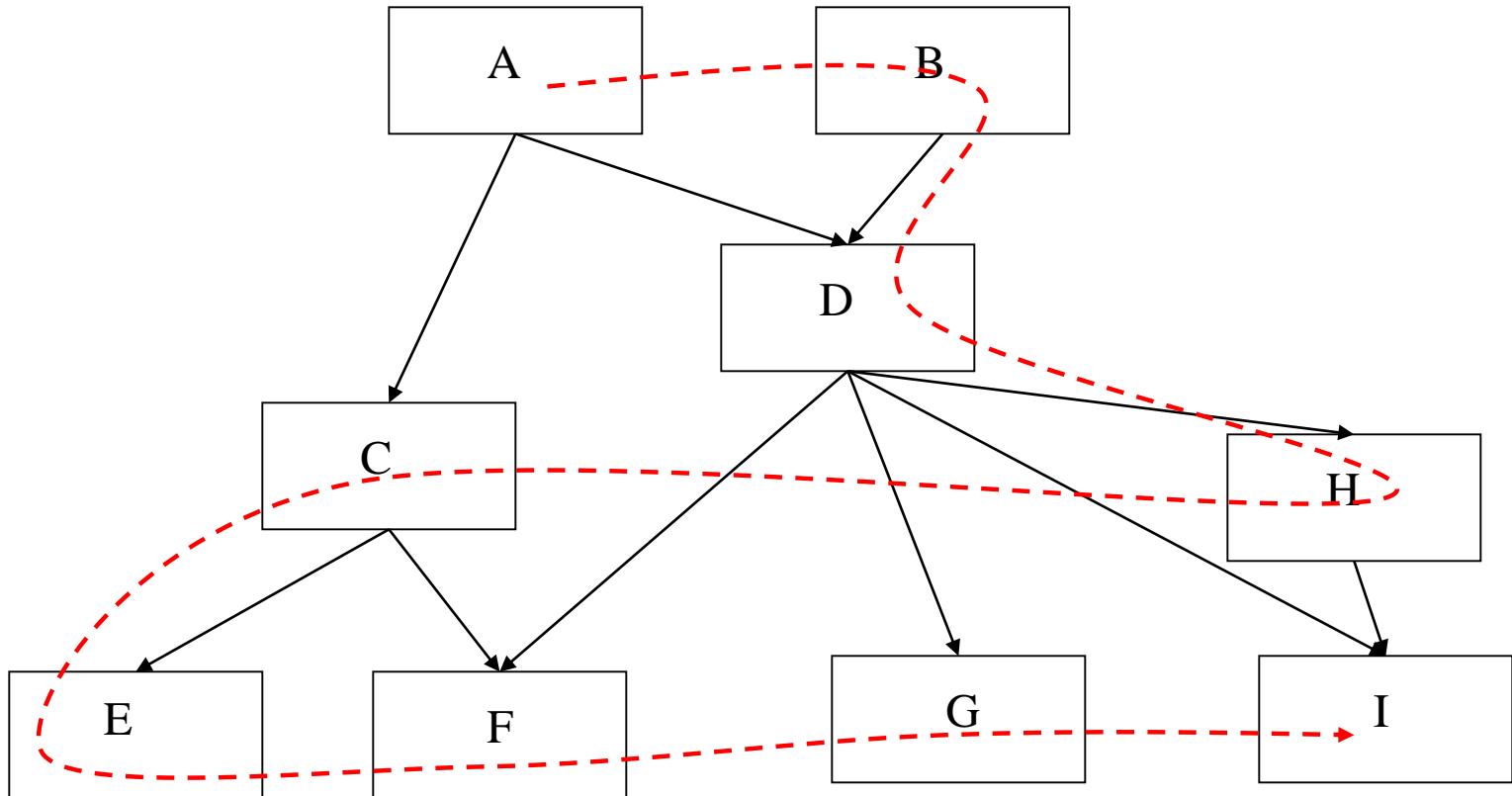
Bottom-Up Integration



Top-down Integration

- Only modules tested in isolation are the modules which are at the highest level
- After a module is tested, the modules directly called by that module are merged with the already tested module and the combination is tested
- Requires **stub** modules to simulate the functions of the missing modules that may be called
 - However, **drivers are not needed** since we are starting with the modules which is not used by any other module and use already tested modules when testing modules in the higher levels

Top-down Integration



Other Integration Approaches

- **Sandwich Integration**
 - Compromise between bottom-up and top-down testing
 - Simultaneously begin bottom-up and top-down testing and meet at a predetermined point in the middle
- **Big Bang Integration**
 - Every module is unit tested in isolation
 - After all of the modules are tested they are all integrated together at once and tested
 - **No driver or stub is needed**
 - However, in this approach, it may be hard to isolate the bugs!

System Testing (CO4)

- **Alpha Testing:** Alpha testing refers to the system testing carried out by the test team within the developing organization.
- **Beta testing:** Beta testing is the system testing performed by a select group of friendly customers.
- **Acceptance Testing:** Acceptance testing is the system testing performed by the customer to determine whether he should accept the delivery of the system.

Performance testing : Performance testing is carried out to check whether the system needs the nonfunctional requirements identified in the SRS document. All performance tests can be considered as **black-box tests**.

- Stress testing
- Volume testing
- Configuration testing
- Compatibility testing
- Regression testing
- Recovery testing
- Maintenance testing
- Documentation testing
- Usability testing

Stress testing : Stress tests are black box tests which are designed to impose a range of abnormal and even illegal input conditions so as to stress the capabilities of the software. **For example**, suppose an operating system is supposed to support 15 multi programmed jobs, the system is stressed by attempting to run 15 or more jobs simultaneously. A real-time system might be tested to determine the effect of simultaneous arrival of several high-priority interrupts.

- **Volume testing:** It is especially important to check whether the data structures (arrays, queues, stacks, etc.) have been designed to successfully extraordinary situations. **For example**, a compiler might be tested to check whether the symbol table overflows when a very large program is compiled

System Testing

- **Configuration testing:** This is used to analyze system behavior in various hardware and software configurations specified in the requirements.
- **Compatibility testing :** This type of testing is required when the system interfaces with other types of systems. Compatibility aims to check whether the interface functions perform as required.

System Testing

- **Regression testing:** This type of testing is required when the system being tested is an upgradation of an already existing system to fix some bugs or enhance functionality, performance, etc.
- **Recovery testing :** Recovery testing tests the response of the system to the presence of faults, or loss of power, devices, services, data, etc.
- **Maintenance testing :** This testing addresses the diagnostic programs, and other procedures that are required to be developed to help maintenance of the system. It is verified that the artifacts exist and they perform properly.

System Testing

- **Documentation testing:** It is checked that the required user manual, maintenance manuals, and technical manuals exist and are consistent.
- **Usability testing:** Usability testing concerns checking the user interface to see if it meets all user requirements concerning the user interface.

Formal Technical Review(Code review)

- Perform on module.
- It is carried out after the module is successfully compiled and all the syntax errors have been eliminated.
- It is extremely cost-effective strategies for reduction in coding errors and to produce high quality code.
- Normally, **two types** of reviews are carried out on the code of a module.
 - 1 Code inspection
 - 2 Code walk Through

Code Walk Through (CO4)

1. The main objectives of the it is to discover the algorithmic and logical errors in the code.
2. It is an informal code analyses technique.
3. It is done after a module has been coded, successfully compiled and all syntax errors eliminated.
4. A few members of the development team are given the code few days before the walk through meeting to read and understand code.
5. Each member selects some test cases and simulates execution and tracing of the code by hand.
6. The members note down their findings to discuss these in a walk through meeting where the coder of the module is present.
7. After meeting several guidelines have evolved over the years for making this useful analysis technique more effective.

Code Walk Through

Some of these guidelines are the following.

- The team performing code walk through should not be either too big or too small.
- Discussion should focus on discovery of errors and not on how to fix the discovered errors.
- In order to cooperation and to avoid the feeling among engineers that they are being evaluated in the code walk through meeting, managers should not attend the walk through meetings.

Code inspection (CO4)

Code is examined for the presence of certain kinds of errors, in contrast to the hand simulation of code execution done in code walk through.

List of some classical programming errors which can be checked during code inspection:

- Use of uninitialized variables.
- Jumps into loops.
- Non terminating loops.
- Incompatible assignments.
- Array indices out of bounds.
- Improper storage allocation and deallocation