Software Testing

An overview

Introduction & Fundamentals

What is Software Testing?

Why testing is necessary?

Who does the testing?

What has to be tested?

When is testing done?

How often to test?

MOST COMMON SOFTWARE PROBLEMS

- Incorrect calculation
- Incorrect data edits & ineffective data edits
- Incorrect matching and merging of data
- Data searches that yields incorrect results
- Incorrect processing of data relationship
- Incorrect coding / implementation of business rules
- Inadequate software performance

- Confusing or misleading data
- Software usability by end users & Obsolete Software
- Inconsistent processing
- Unreliable results or performance
- Inadequate support of business needs
- Incorrect or inadequate interfaces with other systems
- Inadequate performance and security controls
- Incorrect file handling

OBJECTIVES OF TESTING

- Executing a program with the intent of finding an *error*.
- To check if the system meets the requirements and be executed successfully in the Intended environment.
- To check if the system is “ Fit for purpose”.
- To check if the system does what it is expected to do.

TERMINOLOGIES

1. Error
2. Fault
3. Failure
4. Mistake
5. Test Case

ERROR, FAULT AND FAILURE

- **ERROR:** Refers to the difference between the actual output of a program and the expected output.

- **FAULT:** Condition that causes a system to fail in performing its required functions.

- **FAILURE:** Inability of a system to perform a required function according to its specification.

ERROR, FAULT AND FAILURE

- Whenever there is a failure, there is a fault in the system but vice-versa may not be true. That is, sometimes there is a fault in the software but failure is not observed.

- Fault is just like an infection in the body. Whenever there is fever there is an infection, but sometimes body has infection but fever is not observed.

EXAMPLE OF ERROR, FAULT, FAILURE

- The program was supposed to add two numbers but it certainly did not add 5 and 3. 5 + 3 should be 8, but the result is 2. For now we have detected a ***failure.***

- We notice at line number 10, there is a '-' sign present instead of '+' sign. So the **fault** in the program is the '-' sign. Line 10 has the fault which caused the program to deviate from the functionality.

- Error is the mistake I made by typing '-' instead of '+' sign.

```
1       #include<stdio.h>
2
3       int main ()
4       {
5       int value1, value2, ans;
6
7       value1 = 5;
8       value2 = 3;
9
10      ans = value1 - value2;
11
12      printf("The addition of
     5 + 3 = %d.", ans);
13
14      return 0;
15      }
```

MISTAKE, TEST CASE

- **Mistake:** Any programmer action that later shows up as an incorrect result during program execution.

- **Test Case:** It is a triplet[I,S,R] where

  I is the data input to the program under test,

  S is the state of program at which the data is to be input,

  R is the result expected to be produced by the program.

  The state of the program is called **execution mode.**

Example: [input: "abc", state: edit, result: abc is displayed]

- **Test suite:** Set of all test cases with which a given software product is tested.

OBJECTIVES OF TEST CASE

- A good test case is one that has a probability of finding an as yet undiscovered error.
- A good test is not redundant.
- A good test should be “best of breed”.
- A good test should neither be too simple nor too complex.

Design of Test Cases

- Exhaustive testing of any non-trivial system is impractical:
  - input data domain is extremely large.
- Design an optimal test suite:
  - of reasonable size
  - to uncover as many errors as possible.

Design of Test Cases

- If test cases are selected randomly:
  - many test cases do not contribute to the significance of the test suite,
  - do not detect errors not already detected by other test cases in the suite.
- The number of test cases in a randomly selected test suite:
  - not an indication of the effectiveness of the testing.

Design of Test Cases

- Testing a system using a large number of randomly selected test cases:
  - does not mean that many errors in the system will be uncovered.
- Consider an example:
  - finding the maximum of two integers x and y.

Design of Test Cases

- If (x>y) max = x

  else max = x;
- The code has a simple error:
- test suite {(x=3,y=2);(x=2,y=3)} can detect the error,
- a larger test suite {(x=3,y=2);(x=4,y=3);(x=5,y=1)} does not detect the error.

Design of Test Cases

- Systematic approaches are required to design an optimal test suite:
  - each test case in the suite should detect different errors.

Design of Test Cases

- Two main approaches to design test cases:
  - Black-box approach
  - White-box (or glass-box) approach

OBJECTIVE OF A SOFTWARE TESTER

- Find bugs as early as possible and make sure they get fixed.
- To understand the application well.
- Study the functionality in detail to find where the bugs are likely to occur.
- Study the code to ensure that each and every line of code is tested.
- Create test cases in such a way that testing is done to uncover the hidden bugs and also ensure that the software is usable and reliable

VERIFICATION & VALIDATION

SNO.	VERIFICATION	VALIDATION
1	Carried out during development process.	Carried out towards the end of the development process.
2	It is the process of determining whether the output of one phase of software development conforms to that of its previous phase	It is the process of determining whether a fully developed system conforms to its requirements specification.
3	It does not require execution of the software.	It requires execution of the software.
4	It is concerned with phase containment of errors.	To make the final product error free.
5	Techniques used are: review, simulation, formal verification and testing	Techniques are based on product testing.

VERIFICATION & VALIDATION

SNO	VERIFICATION	VALIDATION
6	Low level activity	High level activity
7	Am I building the product right?	Am I building the right product?

Testing Levels

- Unit testing
- Integration testing
- System testing
- Acceptance testing

Unit testing

- The most ‘micro’ scale of testing.
- Tests done on particular functions or code modules.
- Requires knowledge of the internal program design and code.
- Done by Programmers (not by testers).

Unit testing

Objectives	• To test the function of a program or unit of code such as a program or module • To test internal logic • To verify internal design • To test path & conditions coverage • To test exception conditions & error handling
When	• After modules are coded and reviewed
Input	• Internal Application Design • Master Test Plan • Unit Test Plan
Output	• Unit Test Report

Who	•Developer
Methods	•White Box testing techniques •Test Coverage techniques
Tools	•Debug •Re-structure •Code Analyzers •Path/statement coverage tools

Incremental integration testing

- Continuous testing of an application as and when a new functionality is added.
- Application’s functionality aspects are required to be independent enough to work separately before completion of development.
- Done by programmers or testers.

Integration Testing

- Testing of combined parts of an application to determine their functional correctness.

- ‘Parts’ can be
  - code modules
  - individual applications
  - client/server applications on a network.

Types of Integration Testing

»Big Bang testing

»Top Down Integration testing

»Bottom Up Integration testing

Integration testing

Objectives	• To technically verify proper interfacing between modules, and within sub-systems
When	• After modules are unit tested
Input	• Internal & External Application Design • Master Test Plan • Integration Test Plan
Output	• Integration Test report

Who	•Developers
Methods	•White and Black Box techniques •Problem / Configuration Management
Tools	•Debug •Re-structure •Code Analyzers

Objectives	• To verify that the system components perform control functions • To perform inter-system test • To demonstrate that the system performs both functionally and operationally as specified • To perform appropriate types of tests relating to Transaction Flow, Installation, Reliability, Regression etc.
When	• After Integration Testing
Input	• Detailed Requirements & External Application Design • Master Test Plan • System Test Plan
Output	• System Test Report

Who	•Development Team and Users
Methods	•Problem / Configuration Management
Tools	•Recommended set of tools

Systems Integration Testing

Objectives	• To test the co-existence of products and applications that are required to perform together in the production-like operational environment (hardware, software, network) • To ensure that the system functions together with all the components of its environment as a total system • To ensure that the system releases can be deployed in the current environment
When	• After system testing • Often performed outside of project life-cycle
Input	• Test Strategy • Master Test Plan • Systems Integration Test Plan
Output	• Systems Integration Test report

Who	• System Testers
Methods	• White and Black Box techniques • Problem / Configuration Management
Tools	• Recommended set of tools

Acceptance Testing

Objectives	• To verify that the system meets the user requirements
When	• After System Testing
Input	• Business Needs & Detailed Requirements • Master Test Plan • User Acceptance Test Plan
Output	• User Acceptance Test report

Who	Users / End Users
Methods	•Black Box techniques •Problem / Configuration Management
Tools	Compare, keystroke capture & playback, regression testing

TESTING METHODOLOGIES AND TYPES

TESTING METHODOLOGIES

BLACK BOX TESTING

WHITE BOX TESTING

PERFORMANCE TESTING

DIFFERENCE BETWEEN BLACK BOX AND WHITE BOX TESTING

SNO	BLACK BOX TESTING	WHITE BOX TESTING
1	Black box testing also known as functional test or external testing.	White box testing also known as structural test or interior testing or clear box testing or glass box testing
2	No knowledge of internal design or code required.	Knowledge of the internal program design and code required.
3	Tests are based on requirements and functionality	Tests are based on coverage of code statements, branches, paths, conditions.
4	This type of testing is carried out by testers.	This type of testing is carried out by software developer.
5	Implementation Knowledge is not required to carry out Black Box Testing.	Implementation Knowledge is required to carry out Black Box Testing.

DIFFERENCE BETWEEN BLACK BOX AND WHITE BOX TESTING

SNO	BLACK BOX TESTING	WHITE BOX TESTING
6	Testing is applicable on higher levels of testing like System Testing, Acceptance testing.	Testing is applicable on lower level of testing like Unit Testing, Integration testing.
7	The main aim of this testing to check on what functionality is performing by the system under test.	The main aim of White Box testing to check on how System is performing.
8	Black Box testing can be started based on Requirement Specifications documents.	White Box testing can be started based on Detail Design documents.
9	Programming Knowledge is not required to carry out Black Box Testing.	Programming Knowledge is required to carry out White Box Testing.

PERFORMANCE TESTING

- Carried out to check whether the system meets the non-functional requirements identified in the SRS document.

- All performance tests can be considered as black-box tests.

PERFORMANCE TESTING

- There are several types of performance testing.
  - Stress testing
  - Volume testing
  - Configuration testing
  - Compatibility testing
  - Regression testing
  - Recovery testing
  - Maintenance testing
  - Documentation testing
  - Usability testing

STRESS TESTING

- Stress testing is also known as endurance testing.

- Stress testing evaluates system performance when it is stressed for short periods of time.

- For example, if the non-functional requirement specification states that the response time should not be more than 20 secs per transaction when 60 concurrent users are working, then during the stress testing the response time is checked with 60 users working simultaneously.

VOLUME TESTING

- To check whether the data structures (arrays, queues, stacks, etc.) have been designed to successfully handle extraordinary situations.

- For example, a compiler might be tested to check whether the symbol table overflows when a very large program is compiled.

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.
- For example, if the system needs to communicate with a large database system to retrieve information, compatibility testing is required to test the speed and accuracy of data retrieval.

CONFIGURATION TESTING

- This is used to analyze system behavior in various hardware and software configurations specified in the requirements.

- Systems are built in variable configurations for different users.

REGRESSION TESTING

- To fix some bugs or enhance functionality, performance, etc.

- Regression testing is the practice of running an old test suite after each change to the system or after each bug fix to ensure that no new bug has been introduced due to the change or the bug fix.

- If only a few statements are changed, then the entire test suite need not be run - only those test cases that test the functions that are likely to be affected by the change need to be run.

RECOVERY TESTING

- Tests the response of the system to the presence of faults, or loss of power, devices, services, data, etc.
- For example, the printer can be disconnected to check if the system hangs or the power may be shut down to check the extent of data loss and corruption.

MAINTENANCE 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.

DOCUMENTATION TESTING

- It is checked that the required user manual, maintenance manuals, and technical manuals exist and are consistent.

USABILITY TESTING

- Concerns checking the user interface to see if it meets all user requirements concerning the user interface.

- During usability testing, the display screens, report formats, and other aspects relating to the user interface requirements are tested.

- **Alpha Testing:** Refers to the system testing carried out by the test team within the developing organization.

- **Beta testing:** Performed by a select group of friendly customers.

- **Acceptance Testing:** Performed by the customer to determine whether he should accept the delivery of the system.

BLACK BOX TESTING TECHNIQUE

- Based on requirements and functionality
- Not based on any knowledge of internal design or code

- This type of testing is carried out by testers.
- Implementation Knowledge is not required to carry out Black Box Testing.

BLACK BOX TESTING TECHNIQUE

Other names for black box testing:

- Specification testing
- Behavioral testing
- Functional testing
- Input/output driven testing

BLACK BOX TESTING TECHNIQUE

Black box testing attempts to uncover the following:

- Incorrect functions
- Data structure errors
- Missing functions
- Performance errors
- Initialization and termination errors
- External database access errors

BLACK BOX TESTING TECHNIQUE

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

- Equivalence class partitioning.
- Boundary value analysis

EQUIVALENCE CLASS PARTITIONING

- The domain of input values to a program is partitioned into a set of equivalence classes.

- This partitioning is done such that the behavior of the program is similar for every input data belonging to the same equivalence class.

EQUIVALENCE CLASS PARTITIONING

- Main idea behind defining the equivalence classes is

  - testing the code with any one value belonging to an equivalence class is as good as testing the software with any other value belonging to that equivalence class.

EQUIVALENCE CLASS PARTITIONING

Guidelines for designing the equivalence classes:

1. If the input data values to a system can be **specified by a range of values**, then one valid and two invalid equivalence classes should be defined.

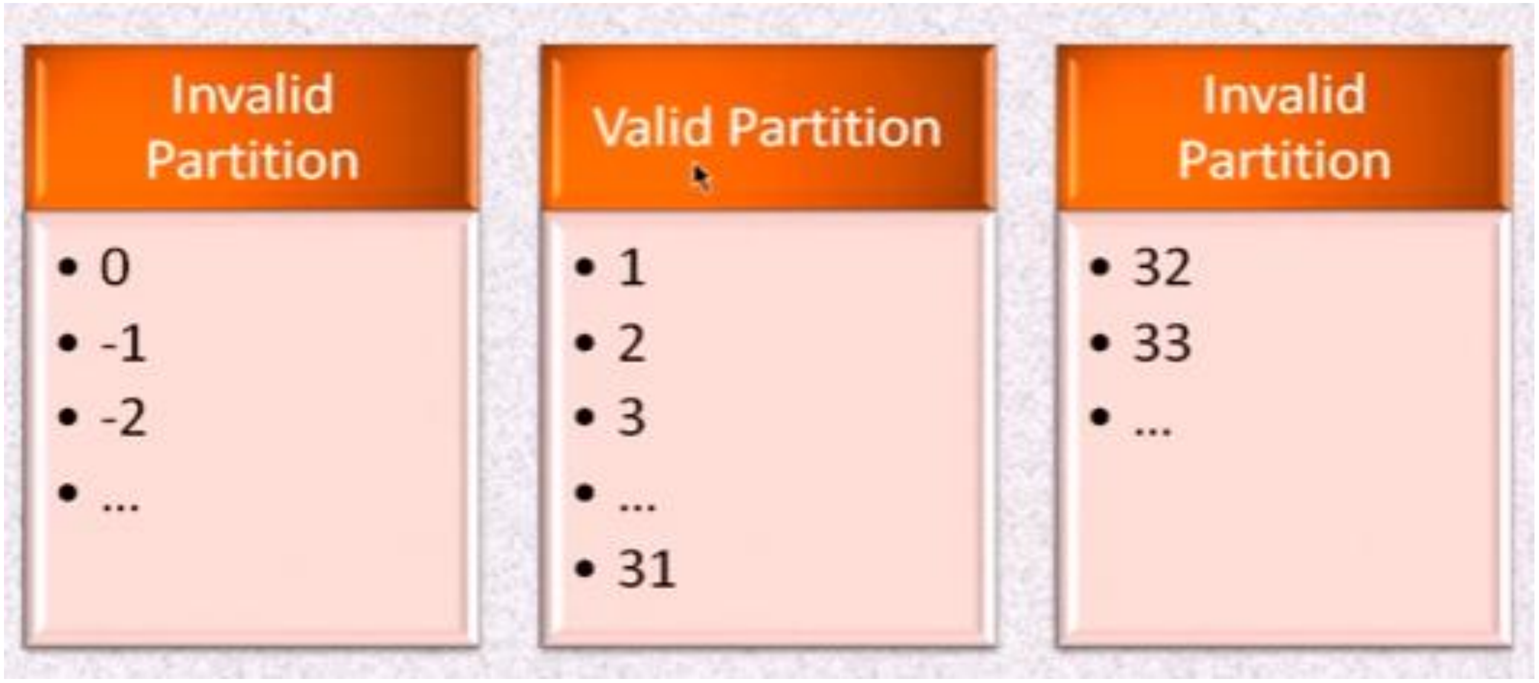
2. If the input data assumes values from a **set of discrete members of some domain,** then one equivalence class for valid input values and another equivalence class for invalid input values should be defined.

EXAMPLE OF EQUIVALENCE CLASS PARTITIONING

Example 1: User want to enter a date field in the range of 1 to 31.

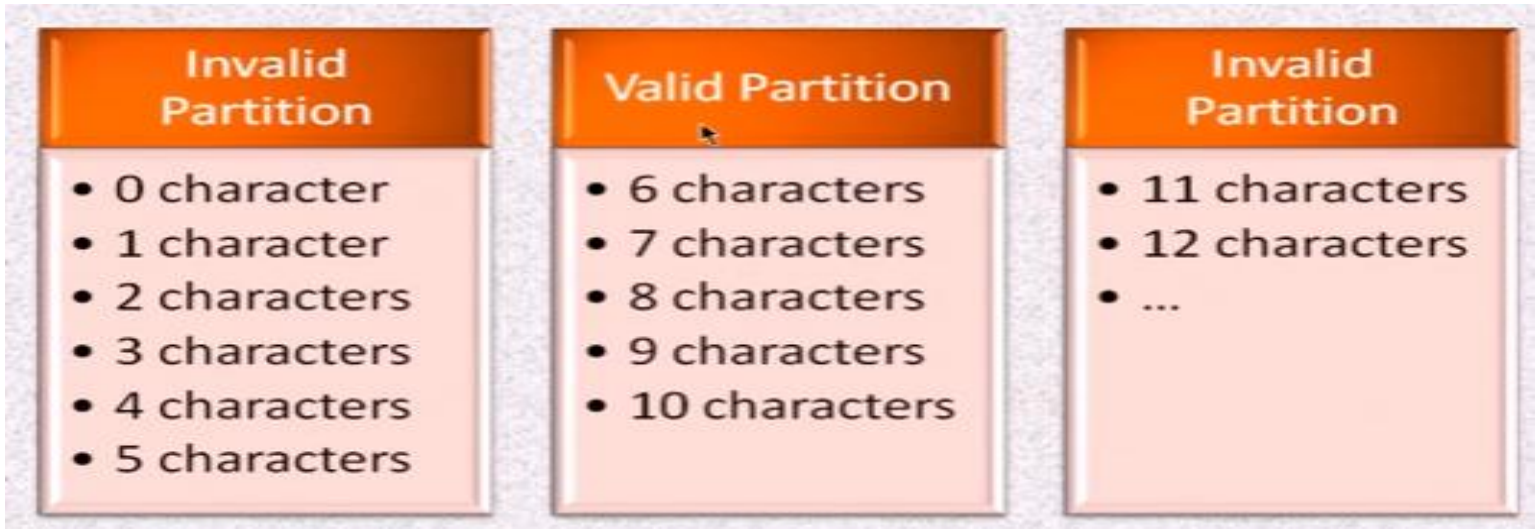
There are three equivalence classes: The set of negative integers, the set of integers in the range of 1 and 31, and the integers larger than 31. The test cases must include representatives for each of the three equivalence classes and a possible test set can be: {-1,10,37}.

EXAMPLE OF EQUIVALENCE CLASS PARTITIONING



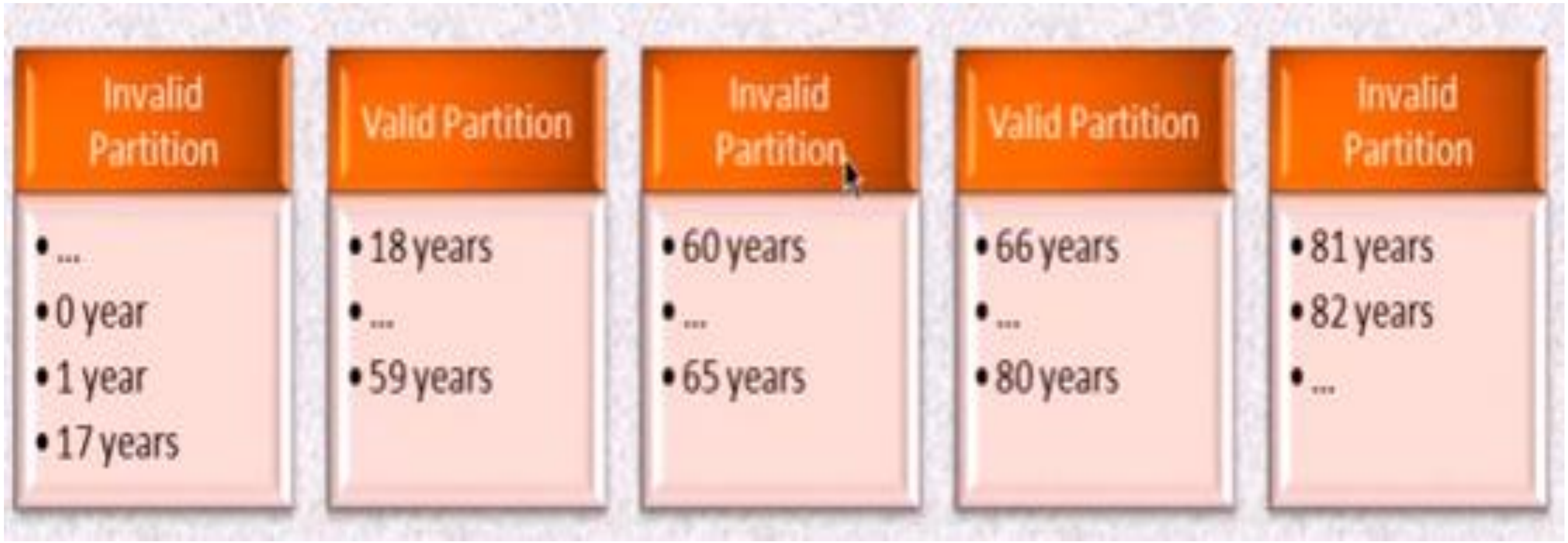
Invalid Partition	Valid Partition	Invalid Partition
• 0	• 1	• 32
• -1	• 2	• 33
• -2	• 3	• …
• …	• …	
	• 31	

Example 2: Suppose you have very important tool at office, accepts valid User Name and Password field to work on that tool, and accepts minimum 6 characters and maximum 10 characters. There are three equivalence classes: Valid range 6-10, Invalid range 5 or less than 5 and Invalid range 11 or more than 11. The test cases must include representatives for each of the three equivalence classes and a possible test set can be:{3,7,17}.



Invalid Partition	Valid Partition	Invalid Partition
• 0 character	• 6 characters	• 11 characters
• 1 character	• 7 characters	• 12 characters
• 2 characters	• 8 characters	• ...
• 3 characters	• 9 characters	
• 4 characters	• 10 characters	
• 5 characters		

Example 3: Suppose a system wants to enter age between 18-80 years except 60-65 years. There will be 5 equivalence classes. Three invalid class and two valid class.



Invalid Partition	Valid Partition	Invalid Partition	Valid Partition	Invalid Partition
• ...	• 18 years	• 60 years	• 66 years	• 81 years
• 0 year	• ...	• ...	• ...	• 82 years
• 1 year	• 59 years	• 65 years	• 80 years	• ...
• 17 years				

BOUNDARY VALUE TESTING

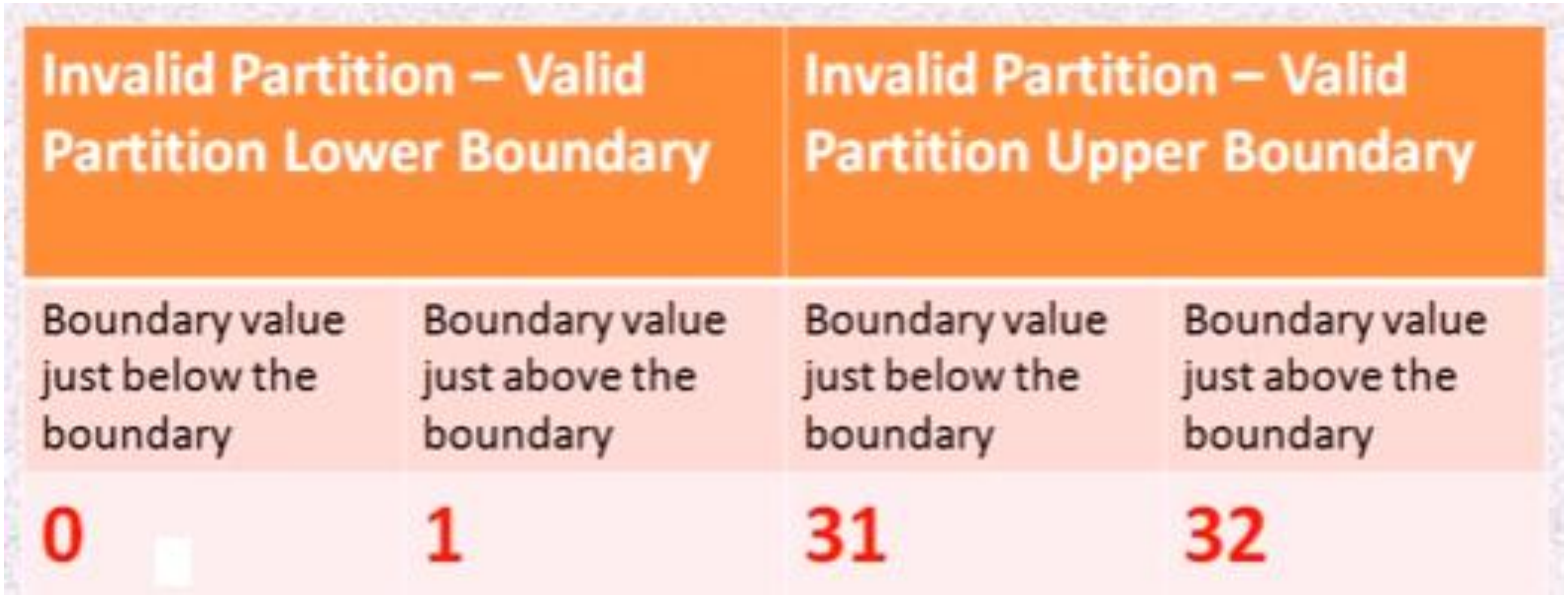
- To identify errors at extreme ends or boundaries rather than finding those exist in centre of input domain.

- These extreme ends like Start- End, Lower-Upper, Maximum-Minimum, Just Inside-Just Outside values are called **boundary values** and the testing is called **boundary testing**.

BOUNDARY VALUE TESTING

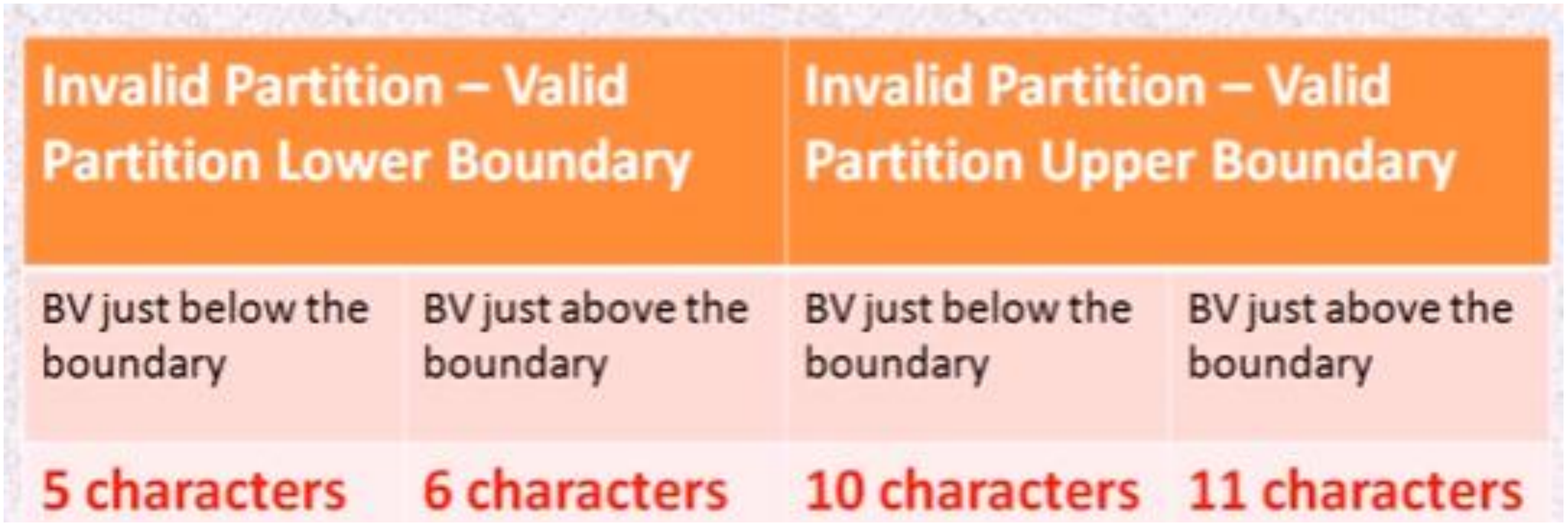
- In Boundary Testing, Equivalence Class Partitioning plays a good role.
- Boundary Testing comes after the Equivalence Class Partitioning.
- Test cases are selected at the edges of the equivalence classes.
- Boundary value analysis is most common when checking a range of numbers.

Example 1: User want to enter a date field in the range of 1 to 31.



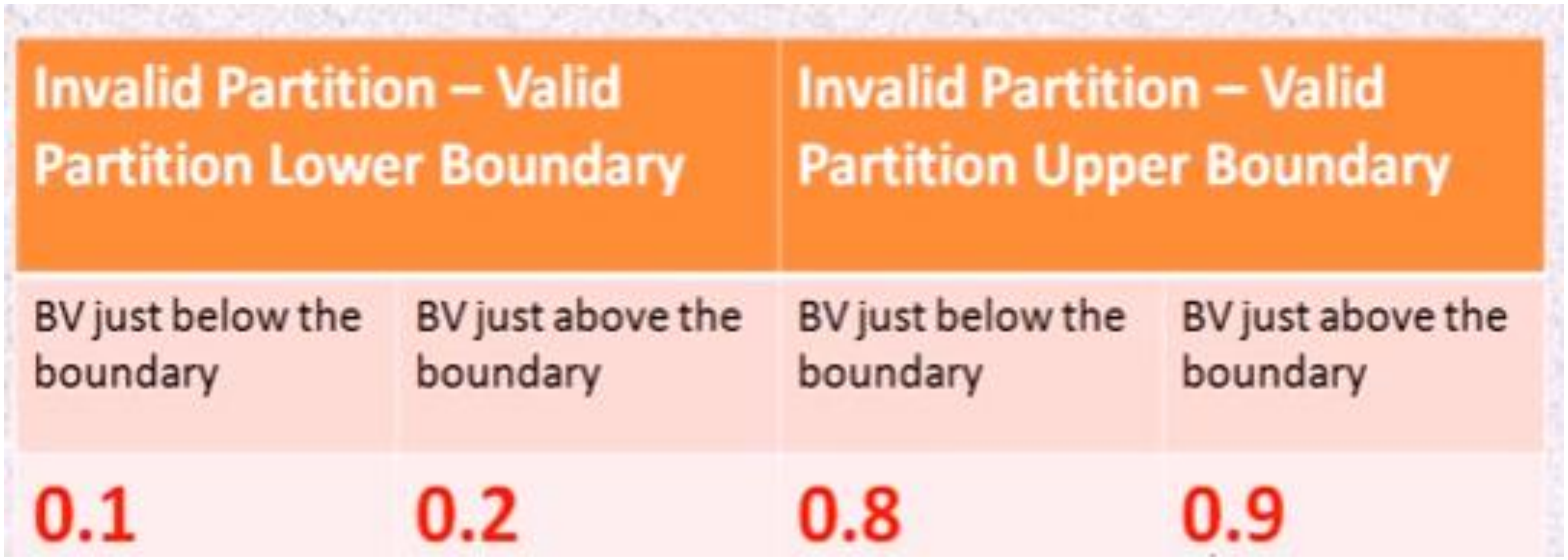
Invalid Partition – Valid Partition Lower Boundary		Invalid Partition – Valid Partition Upper Boundary	
Boundary value just below the boundary	Boundary value just above the boundary	Boundary value just below the boundary	Boundary value just above the boundary
0	1	31	32

Example 2: Suppose you have very important tool at office, accepts valid User Name and Password field to work on that tool, and accepts minimum 6 characters and maximum 10 characters.



Invalid Partition – Valid Partition Lower Boundary		Invalid Partition – Valid Partition Upper Boundary	
BV just below the boundary	BV just above the boundary	BV just below the boundary	BV just above the boundary
5 characters	6 characters	10 characters	11 characters

Example 3: User wants to enter a floating point number in the range of 0.2 to 0.8



Invalid Partition – Valid Partition Lower Boundary		Invalid Partition – Valid Partition Upper Boundary	
BV just below the boundary	BV just above the boundary	BV just below the boundary	BV just above the boundary
0.1	0.2	0.8	0.9

Example of equivalence partitioning

- For example, a savings account in a bank has a different rate of interest depending on the balance in the account. For example, 3% rate of interest is given if the balance in the account is in the range of $0 to $100, 5% rate of interest is given if the balance in the account is in the range of $100 to $1000, and 7% rate of interest is given if the balance in the account is $1000 and above.
- How many partitions will be done?

White box testing / Structural testing

- Based on knowledge of internal logic of an application's code
- Based on coverage of code statements, branches, paths, conditions
- Tests are logic driven

White-Box Testing

- There exist several popular white-box testing methodologies:
  - Statement coverage
  - branch coverage
  - path coverage
  - condition coverage
  - mutation testing
  - data flow-based testing

Statement Coverage

- Statement coverage methodology:
  - design test cases so that
    - every statement in a program is executed at least once.

Statement Coverage

- The principal idea:
  - unless a statement is executed,
  - we have no way of knowing if an error exists in that statement.

Statement Coverage

- Executing some statement once and observing that

  - it behaves properly for that input value

  - no guarantee that it will behave correctly for all input values.

Statement Coverage

- **Test requirements:** Statements in the program
- **Coverage measure:**

$$\frac{\textbf{Number of statements executed}}{\textbf{Total number of statements}}$$

Example

```
1. Printsum(int a, int b){
2. Int result =a+b;
3. If (result>0)
4. Printcol("red",result);
5. Elseif (result<0)
6. Printcol("blue", result);
[else do nothing]
7. }
```

- {(a=3,b=9),(a=-5, b=-8)}

Example

```
• int f1(int x, int y){
• 1 while (x != y){
• 2    if (x>y) then
• 3        x=x-y;
• 4    else y=y-x;
• 5 }
• 6 return x;        }
```

Branch Coverage

- Also known as **Edge testing.**
- Test cases are designed such that:
  - different branch conditions
    - given true and false values in turn.

Branch Coverage

- Branch testing guarantees statement coverage:
  - a stronger testing compared to the statement coverage-based testing.

Branch Coverage

- **Test requirements:** Branches in the program

- **Coverage measure:**

$$\frac{\textbf{Number of branches executed}}{\textbf{Total number of branches}}$$

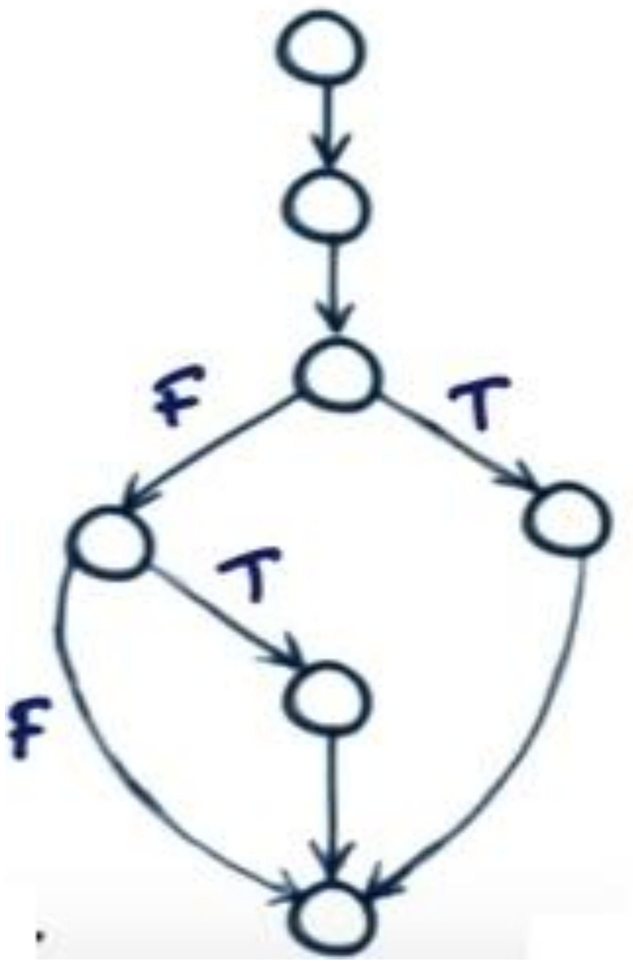
- **Branches are the outgoing edges from the decision point.**

Example

1. Printsum(int a, int b){
2. Int result =a+b;
3. If (result>0)
4. Printcol(“red”,result);
5. Elseif (result<0)
6. Printcol(“blue”, result);

[else do nothing]

7. }

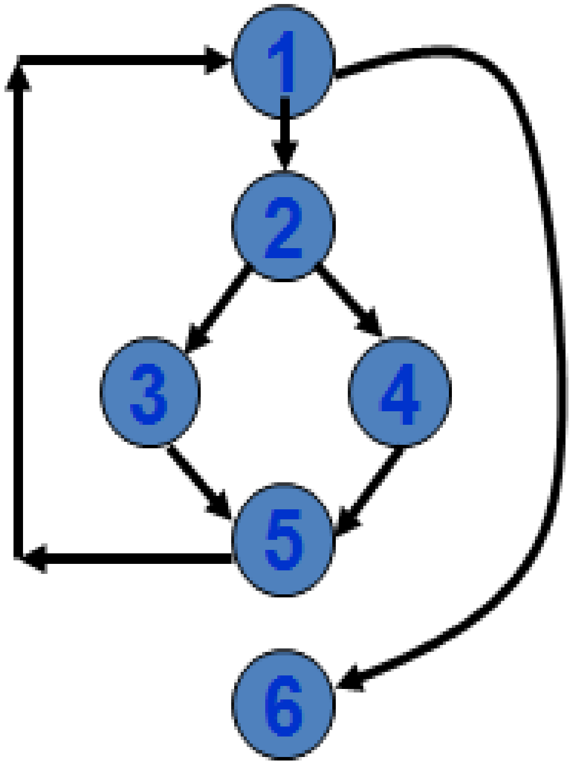


- {(a=3,b=9),(a=-5, b=-8), (a=0, b=0)}

Example

```
int f1(int x,int y){
1 while (x != y){
2    if (x>y) then
3          x=x-y;
4    else y=y-x;
5 }
6 return x;       }
```

- Test cases for branch coverage can be:
- {(x=3,y=3),(x=3,y=2),(x=3,y=4)}



Condition Coverage

- Test cases are designed such that:
  - each component of a composite conditional expression
    - given both true and false values.

Example

- Consider the conditional expression
  - ((c1.and.c2).or.c3):
- Each of c1, c2, and c3 are exercised at least once,
  - i.e. given true and false values.

Branch testing

- Condition testing
  - stronger testing than branch testing:
- Branch testing
  - stronger than statement coverage testing.

Condition coverage

- Consider a boolean expression having n components:
  - for condition coverage we require 2^n test cases.

Condition coverage

- **Test requirements:** Individual conditions in the program

- **Coverage measure:**

$$\frac{\textbf{Number of conditions that are both T and F}}{\textbf{Total number of conditions}}$$

Path Coverage

- Design test cases such that:
  - all linearly independent paths in the program are executed at least once.

Linearly independent paths

- Defined in terms of
  - control flow graph (CFG) of a program.

Path coverage-based testing

- To understand the path coverage-based testing:
  - we need to learn how to draw control flow graph of a program.

Control flow graph (CFG)

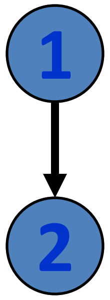
- A control flow graph (CFG) describes:
  - the sequence in which different instructions of a program get executed.
  - the way control flows through the program.

How to draw Control flow graph?

- Number all the statements of a program.
- Numbered statements:
  - represent nodes of the control flow graph.
  - An edge from one node to another node exists if the execution of the statement representing the first node can result in the transfer of control to the other node.

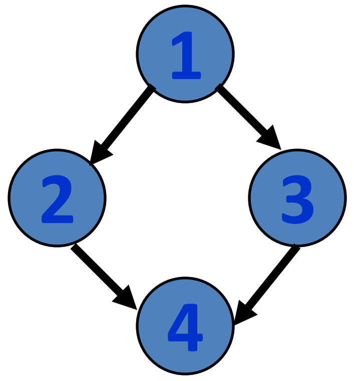
How to draw Control flow graph?

- Sequence:
  - 1 a=5;
  - 2 b=a*b-1;


$$\begin{tikzcd}
1 \arrow[d] \\
2
\end{tikzcd}$$

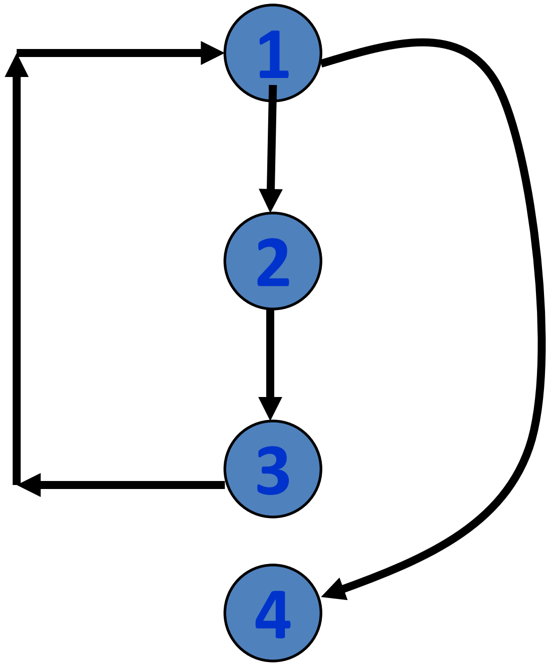
How to draw Control flow graph?

- Selection:
  - 1 if(a>b) then
  - 2 c=3;
  - 3 else c=5;
  - 4 c=c*c;



How to draw Control flow graph?

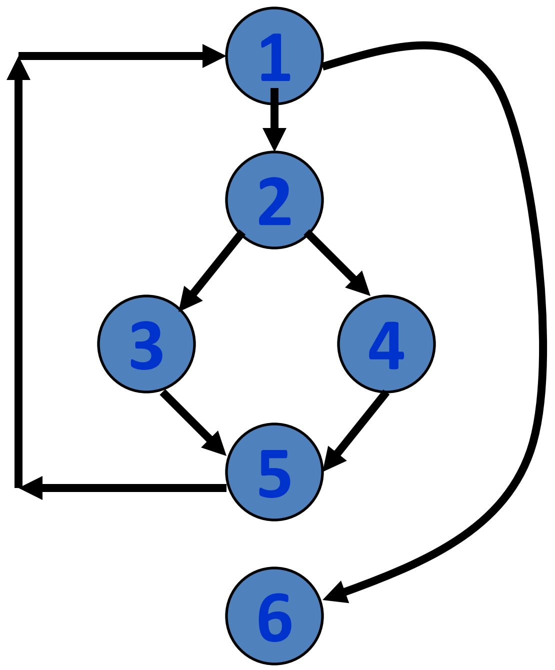
- Iteration:
  - 1 while(a>b){
  - 2 b=b*a;
  - 3 b=b-1;}
  - 4 c=b+d;



Example

```
• int f1(int x,int y){
• 1 while (x != y){
• 2    if (x>y) then
• 3        x=x-y;
• 4    else y=y-x;
• 5 }
• 6 return x;       }
```

Example Control Flow Graph



$$\begin{tikzcd}
 & 1 \arrow[d] \arrow[dddd, bend left=60] & \\
 & 2 \arrow[dl] \arrow[dr] & \\
3 \arrow[dr] & & 4 \arrow[dl] \\
 & 5 \arrow[uuu, bend left=90] & \\
 & 6 &
\end{tikzcd}$$

PATH

- A path through a program:
  - a node and edge sequence from the starting node to a terminal node of the control flow graph.
  - There may be several terminal nodes for program.

LINEARLY INDEPENDENT PATH

- Any path through the program:
  - introducing at least one new node:
    - that is not included in any other independent paths.

- If a path has one new node

  - that compared to all other linearly independent paths, then path is also linearly independent.

INDEPENDENT PATH

- It is straight forward:
  - to identify linearly independent paths of simple programs.
- For complicated programs:
  - it is not so easy to determine the number of independent paths.

CYCLOMATIC COMPLEXITY

- McCabe's metric provides:
  - a quantitative measure of testing difficulty and the ultimate reliability
- Intuitively,
  - number of bounded areas increases with the number of decision nodes and loops.

APPLICATION OF CYCLOMATIC COMPLEXITY

- Relationship exists between:
  - McCabe's metric
  - the number of errors existing in the code,
  - the time required to find and correct the errors.

CYCLOMATIC COMPLEXITY

- From maintenance perspective,
  - limit cyclomatic complexity
    - of modules to some reasonable value.

  - Good software development organizations:
    - restrict cyclomatic complexity of functions to a maximum of ten or so.

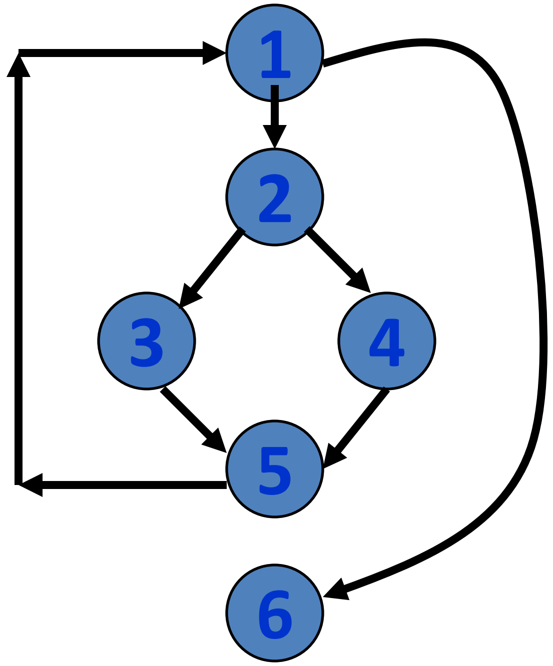
MCCABE'S CYCLOMATIC METRIC

- Defines an upper bound:
  - for the number of linearly independent paths of a program
- Provides a practical way of determining:
  - the maximum number of linearly independent paths in a program.

MCCABE'S CYCLOMATIC METRIC

- Given a control flow graph G, cyclomatic complexity V(G):
  - V(G)= E-N+2
    - N is the number of nodes in G
    - E is the number of edges in G

EXAMPLE CONTROL FLOW GRAPH



Example

- Cyclomatic complexity = 7-6+2 = 3.

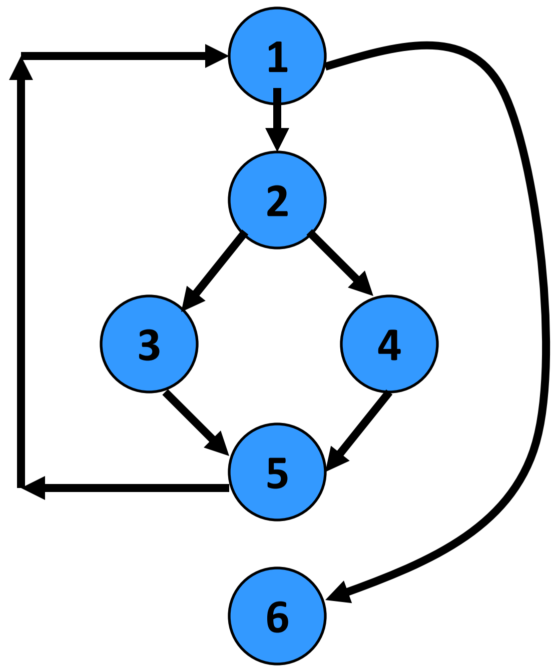
CYCLOMATIC COMPLEXITY

- Another way of computing cyclomatic complexity:
  - inspect control flow graph
  - determine number of bounded areas in the graph
- V(G) = Total number of bounded areas + 1

BOUNDED AREA

- Any region enclosed by a nodes and edge sequence.
- Cannot be used for non-structured programs.
- The number of bounded areas
  - increases with the number of decision paths and loops.

EXAMPLE CONTROL FLOW GRAPH



EXAMPLE

- From a visual examination of the CFG:
  - the number of bounded areas is 2.
  - cyclomatic complexity = 2+1=3.

CYCLOMATIC COMPLEXITY

- Another way of computing cyclomatic complexity:
  - computing the number of decision statements of the program.
  - N is the number of decision statement of a program,
  - McCabe’s metric is equal to N+1.

PATH TESTING

- The tester proposes:
  - an initial set of test data using his experience and judgement.

PATH TESTING

- A dynamic program analyzer is used:
  - to indicate which parts of the program have been tested
  - the output of the dynamic analysis
    - used to guide the tester in selecting additional test cases.

DERIVATION OF TEST CASES

- Let us discuss the steps:
  - to derive path coverage-based test cases of a program.

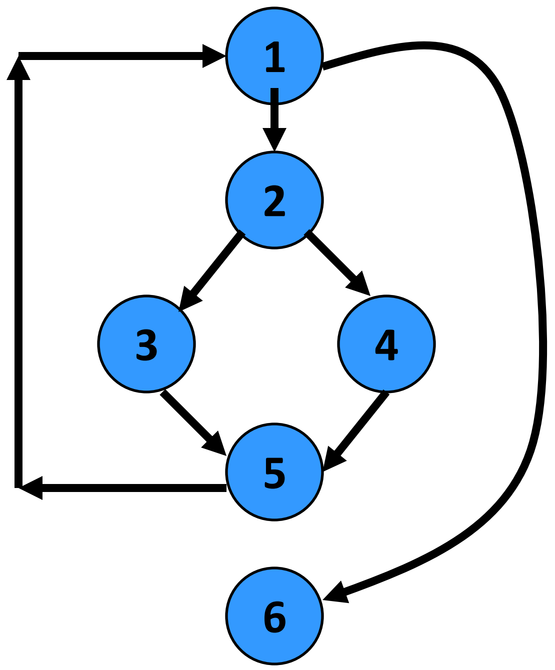
DERIVATION OF TEST CASES

- Draw control flow graph.
- Determine V(G).
- Determine the set of linearly independent paths.
- Prepare test cases:
  - to force execution along each path.

Example

```
int f1(int x,int y){
1 while (x != y){
2    if (x>y) then
3         x=x-y;
4    else y=y-x;
5 }
6 return x;        }
```

Example Control Flow Diagram



DERIVATION OF TEST CASES

- V(G)= E-N+2

  7 - 6 + 2 = 3

- Number of independent paths: 3
  - 1,6 test case (x=1, y=1)
  - 1,2,3,5,1,6 test case(x=2, y=1)
  - 1,2,4,5,1,6 test case(x=1, y=2)

DATA FLOW-BASED TESTING

- Selects test paths of a program

  - according to the locations of the -

  - definitions and uses of different variables in a program.

DATA FLOW-BASED TESTING

- For a statement numbered S, let

  DEF(S) = {X/statement S contains a definition of X}, and

  USES(S) = {X/statement S contains a use of X}
- For the statement **S:a=b+c;, DEF(S) = {a}. USES(S) = {b,c}.**

- The definition of variable X at statement S is said to be live at statement S1, if there exists a path from statement S to statement S1 which does not contain any definition of X.

EXAMPLE OF DATA FLOW BASED TESTING

```
1  X(){
2  int a=5; /* Defines variable a */
3  While(c>5) {
4  if (d<50)
5  b=a*a; /*Uses variable a */
6  a=a-1; /* Defines variable a */
…
7  }
8  print(a); } /*Uses variable a */
```

DATA FLOW-BASED TESTING

- Data flow testing strategies are useful for

  - selecting test paths of a program containing

  - nested if and loop statements.

MUTATION TESTING

- It is a **fault based testing techniques.**
- The idea behind mutation testing is to make few arbitrary changes to a program at a time.
- Each time the program is changed, it is called as a **mutated program** and the change effected is called as a **mutant**.
- A mutated program is tested against the full test suite of the program.

MUTATION TESTING

- 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 been exhausted,
  - **The test suite is enhanced to kill the mutant.**

MUTATION TESTING

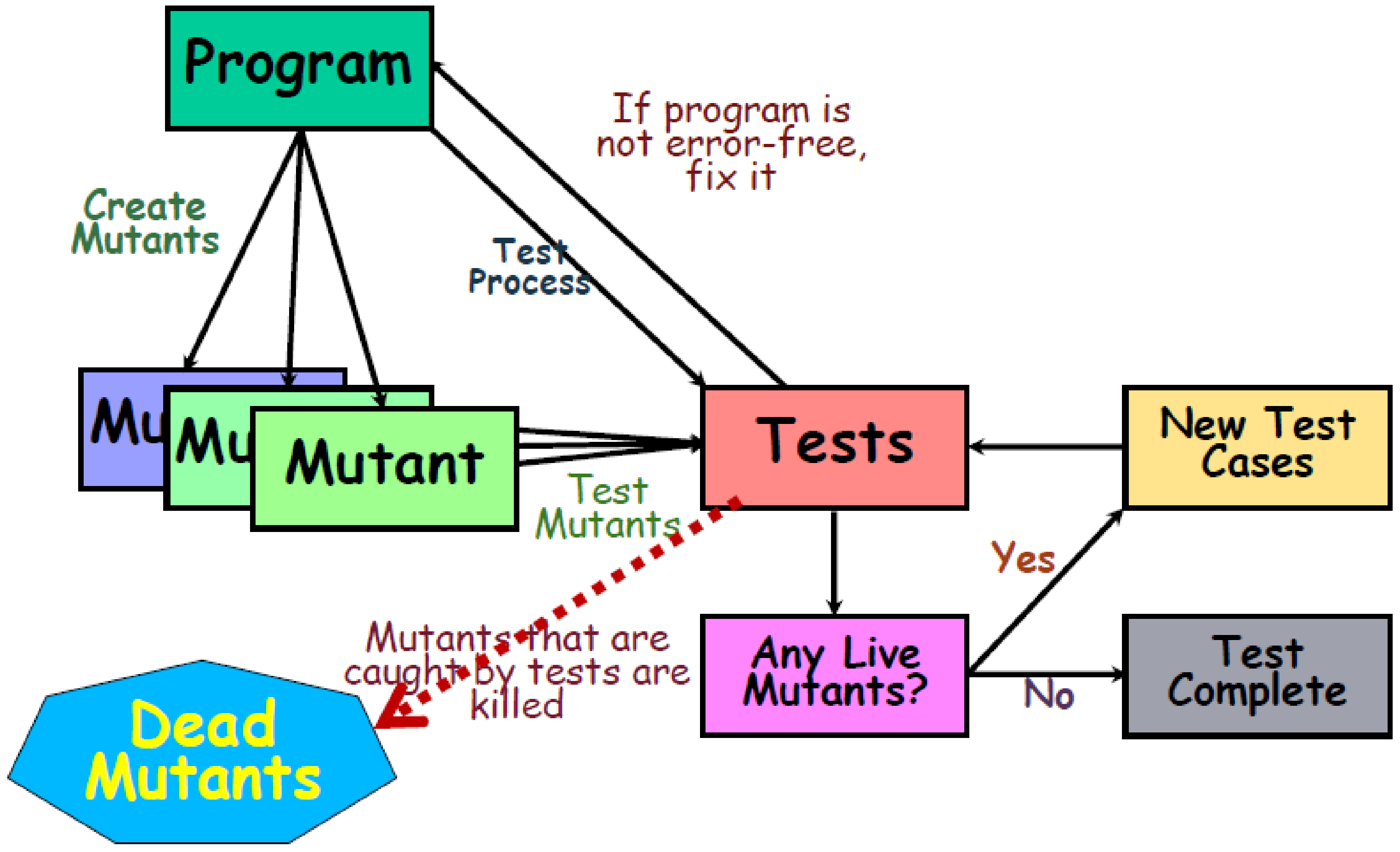
- The process of generation and killing of mutants:
  - **Can be automated by predefining a set of primitive changes that can be applied to the program.**

MUTATION TESTING

Example primitive changes to a program:

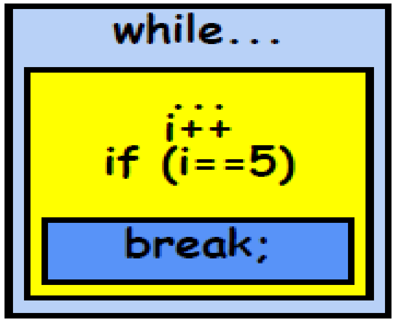
- Deleting a statement
- Altering an arithmetic operator,
- Changing the value of a constant,
- Changing a data type, etc.

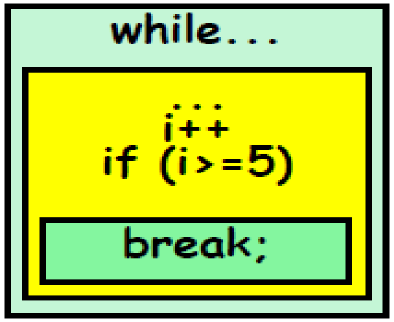
MUTATION TESTING PROCESS



Equivalent Mutants

- There may be surviving mutants that cannot be killed,
  - These are called Equivalent Mutants
- Although syntactically different:
  - These mutants are indistinguishable through testing.
- Therefore have to be checked 'by hand'





Advantages of Mutation Testing

1) It can show the ambiguities in code.

2) It leads to move reliable product.

3) A comprehensive testing can be done.

DISADVANTAGES OF MUTATION TESTING

- Equivalent mutants
- Computationally very expensive.
  - A large number of possible mutants can be generated.
- Certain types of faults are very difficult to inject.
  - Only simple syntactic faults introduced

UNIT TESTING

- Modules required to provide the necessary environment is usually not available until they too have been unit tested.

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

UNIT TESTING

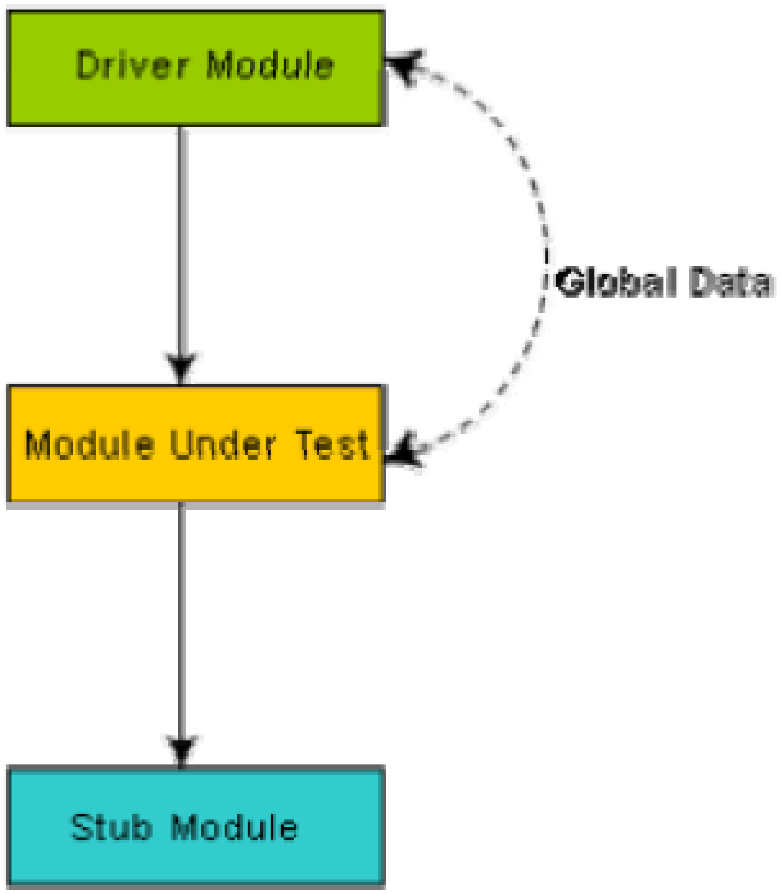
- **Stub procedure** is a dummy procedure that has the same I/O parameters as the given procedure but has a highly simplified behavior.

- For example, a stub procedure may produce the expected behavior using a simple table lookup mechanism.

UNIT TESTING

- **Driver module** contain
  - Nonlocal data structures accessed by the module
  - and have the code to call the different functions of the module with appropriate parameter values.

UNIT TESTING



INTEGRATION TESTING

- Objective of integration testing is

  - to test the module interfaces,

  - i.e. there are no errors in the parameter passing, when one module invokes another module.

- During integration testing, different modules of a system are integrated in a planned manner using an integration plan.

INTEGRATION TESTING

- An important factor that guides the integration plan is
  - the **module dependency graph**.

- The structure chart or module dependency graph denotes the order in which different modules call each other.

INTEGRATION TESTING

Four types of integration testing approaches:

- Big bang approach
- Top-down approach
- Bottom-up approach
- Mixed-approach

BIG BANG INTEGRATION TESTING

- Big bang approach is the simplest integration testing approach:
  - All the modules are simply put together and tested.
  - This technique is used only for very small systems.

BIG BANG INTEGRATION TESTING

Main problems with this approach:

–If an error is found:

•It is very difficult to localize the error

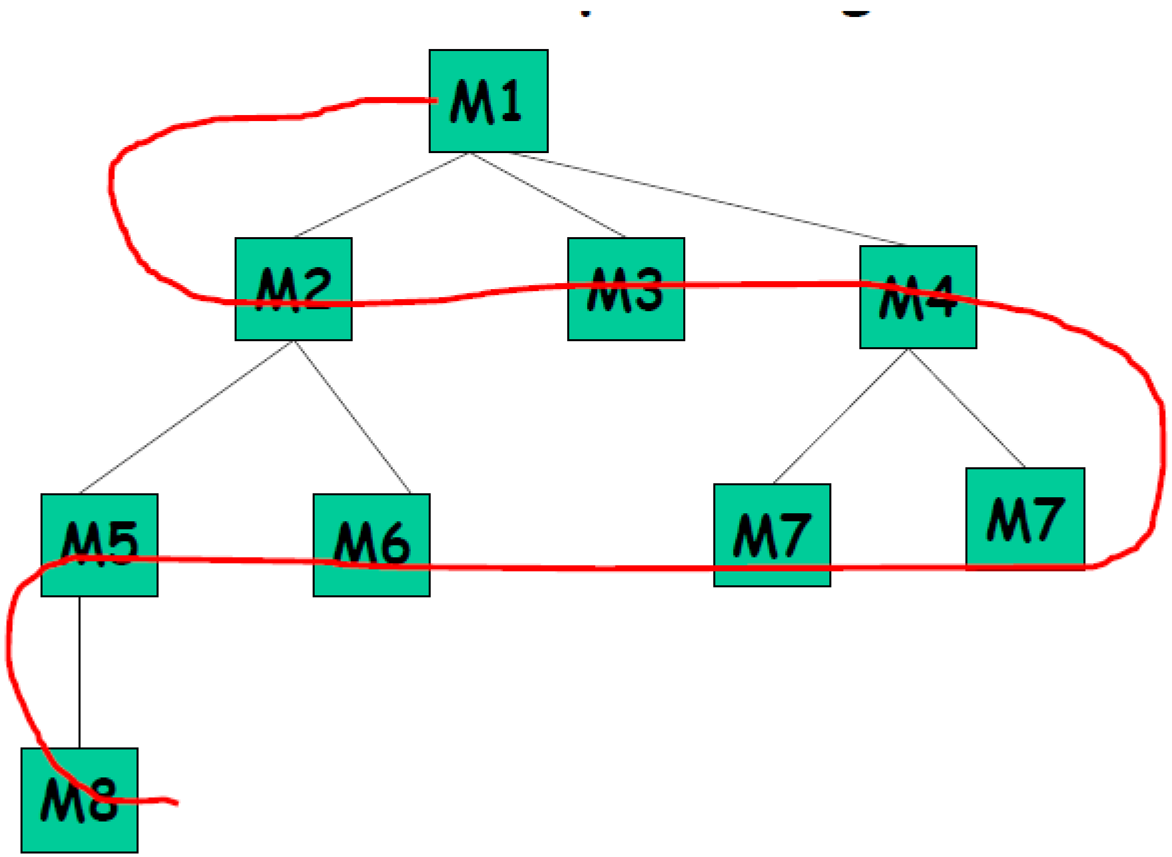
•The error may potentially belong to any of the modules being integrated.

–Debugging becomes very expensive.

BOTTOM-UP INTEGRATION TESTING

- Integrate and test the bottom level modules first.
- Disadvantages of bottom-up testing:
  - Drivers have to be written.
  - Test engineers cannot observe system level functions from a partly integrated system.

BOTTOM-UP INTEGRATION TESTING

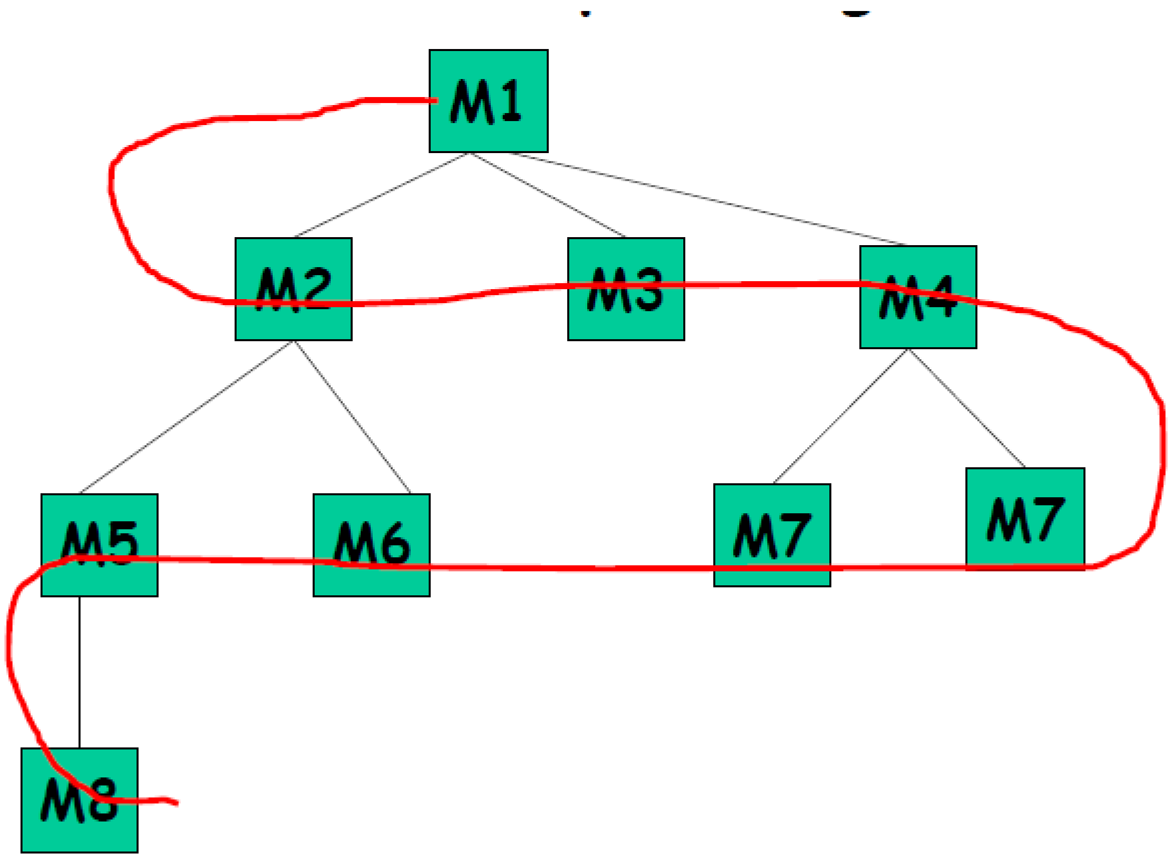


TOP-DOWN INTEGRATION TESTING

- Top-down integration testing starts with the top module:
  – and one or two subordinate modules

- After the top-level 'skeleton’ has been tested:
  – Immediate subordinate modules of the 'skeleton’ are combined with it and tested.

TOP-DOWN INTEGRATION TESTING



ADVANTAGES OF TOP-DOWN INTEGRATION TESTING

– Test cases designed to test the integration of some module are reused after integrating other modules at lower level.

– Advantageous if major flaws occur toward the top of the program.

DISADVANTAGES OF TOP-DOWN INTEGRATION TESTING

– It may not be possible to observe meaningful system functions because of an absence of lower level modules which handle I/O.

– Stub design become increasingly difficult when stubs lie far away from the level of the module.

MIXED INTEGRATION TESTING

- Mixed (or sandwiched) integration testing:

–Uses both top-down and
bottom-up testing approaches.

–Requires less stubs and drivers

MIXED INTEGRATION TESTING

- In top-down approach:
  - Integration testing waits till all top-level modules are coded and unit tested.

- In bottom-up approach:
  - Testing can start only after bottom level modules are ready.

REGRESSION TESTING

- Regression testing is testing done to check that a system update does not cause new errors or re-introduce errors that have been corrected earlier.

Need for Regression Testing

- Any system during use undergoes frequent code changes.
  - Corrective, Adaptive, and Perfective changes.

- Regression testing needed after every change:
  - Ensures unchanged features continue to work fine.

Automated Regression Testing

- Test cases that have already run once:
  - – May have to be run again and again after each change
  - – Test cases may be executed hundreds of time
  - – Automation very important.
- Fortunately, capture and replay type tools appear almost a perfect fit:
  - – However, test cases may fail for reasons such as date or time change
  - – Also test cases may need to be maintained after code change

ERROR SEEDING

- Seeds the code with some known errors.

- The number of these seeded errors detected in the course of the standard testing procedure is determined.

ERROR SEEDING

- These values in conjunction with the number of unseeded errors detected can be used to predict:
    - The number of errors remaining in the product.
    - The effectiveness of the testing strategy.

ERROR SEEDING

- Let N be the total number of errors in the system
- Let n of errors be found by testing.
- Let S be the total number of seeded errors
- Let s of these errors be found during testing.

$n/N = s/S$

or

$N = S \times n/s$

ERROR SEEDING

- Defects still remaining after testing

$$\mathbf{N - n = n \times (S - s)/s}$$

- Error seeding works satisfactorily only if the kind of seeded errors matches closely with the kind of defects that actually exist.

ERROR SEEDING

- A defect-seeding program inserts 81 defects into an application. Inspections and testing found 5,832 defects. Of these, 72 were seeded defects. How many errors or defects are predicted to remain in the application?

  523
  640
  648
  729

ERROR SEEDING

- Correct Answer: 729

  To determine the total number of defects, use the following formula:
  72(discovered seeded defects)
  81(total seeded defects)=5832(total discovered defects)
  X(total defects)
  Then, solve for X:

  72X=5832(81)

DIFFERENCE BETWEEN ERROR SEEDING AND MUTATION TESTING

	Error Seeding	Mutation Testing
1	No mutants are present here.	Mutants are developed for testing.
2	Here source code is tested within itself.	Here mutants are combined, compared for testing to find error introduced.
3	Errors are introduced directly.	Special techniques are used to introduce errors.
4	Test cases which detect errors are used for testing.	Here, test cases which kill mutants are used for testing.
5	It is less efficient error testing technique.	It is more efficient than error seeding.
6	It requires less time.	It is more time consuming.
7	It is economical to perform.	It is expensive to perform.
8	It is better method for bigger problems.	It is a better method for small size programs.