Testing Techniques

Contents

- Essence
- Terminology
- Classification
  - Unit, System …
  - BlackBox, WhiteBox
- Debugging

Objective Explained

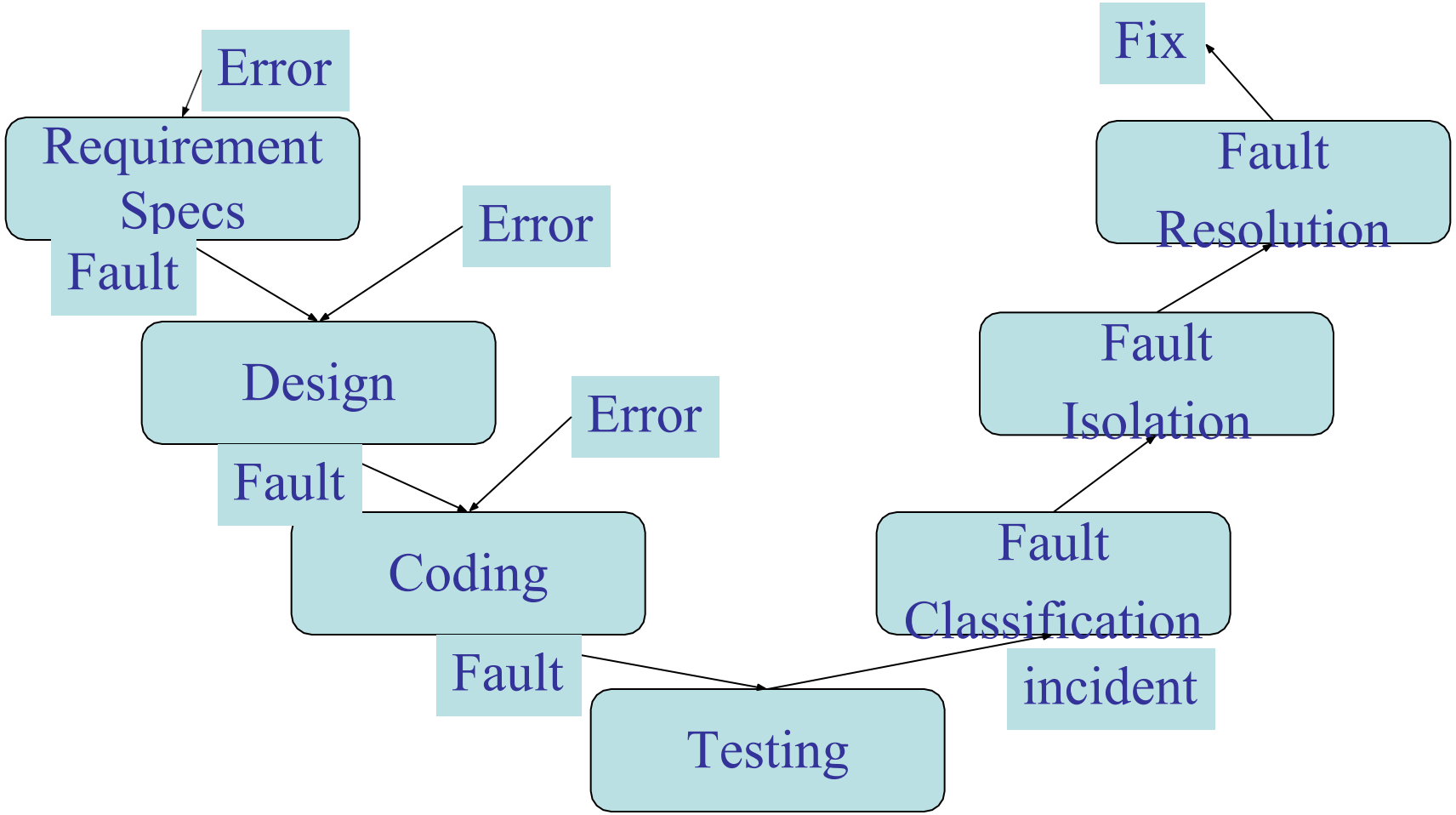
- Testing is the process of executing a program with

  the intent of finding errors

- Testing is obviously concerned with errors, faults, failures and incidents. A test is the act of exercising software with test cases with an objective of

- Finding failure

A Testing Life Cycle



Terminology

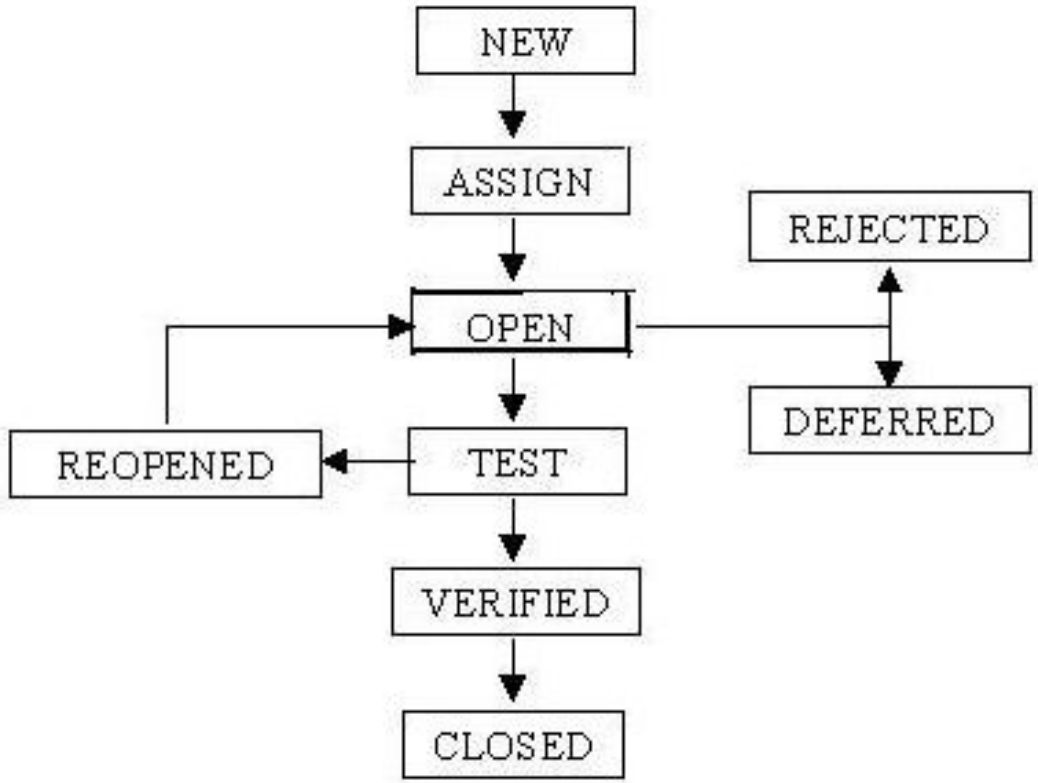
- Error
  - Represents mistakes made by people
- Fault
  - Is result of error. May be categorized as
    - Fault of Commission – we enter something into representation that is incorrect

Cont…

- Failure
  - Occurs when fault executes.
- Incident
  - Behavior of fault. An incident is the symptom(s) associated with a failure that alerts user to the occurrence of a failure
- Test case
  - Associated with program behavior. It carries set of input and list of expected output


Software Testing Life Cycle (STLC)
Requirements/Design Review
Test Planning
Test Designing
Test Environment Setup
Test Execution
Test Reporting

BUG LIFE CYCLE



Test cases and Test suites

- Test case is a triplet [I, S, O] where
  - I is input data
  - S is state of system at which data will be input
  - O is the expected output
- Test suite is set of all test cases
- Test cases are not randomly selected. Instead even they need to be designed.

Terms

- Verification

Process of determining whether output of one phase of development conforms to its previous phase.

Verification is concerned with phase containment of errors

- Validation

Process of determining whether a fully developed system conforms to its SRS document

Validation is concerned about the final product to be error free

Classification of Test

- There are two levels of classification
  - One distinguishes at granularity level
    - Unit level
    - System level
    - Integration level
  - Other classification (mostly for unit level) is based on methodologies
    - Black box (Functional) Testing
    - White box (Structural) Testing

Test methodologies

- Functional (Black box) inspects specified behavior
- Structural (White box) inspects programmed behavior

Exhaustive Testing is Hard

```
int max(int x, int y)
{
  if (x > y)
  return x;
  else
    return x;
}
```

- Number of possible test cases (assuming 32 bit integers)
  - $2^{32} \times 2^{32} = 2^{64}$
- Do bigger test sets help?
  - Test set
    {(x=3,y=2), (x=2,y=3)}
    will detect the error
  - Test set
    {(x=3,y=2),(x=4,y=3),(x=5,y=1)}
    will not detect the error although it has more test cases
- The power of the test set is not determined by the number of test cases
- But, if $T_1 \supseteq T_2$, then T_2 will detect every fault detected by T_1

When to use what

- Few set of guidelines available
- A logical approach could be
  - Prepare functional test cases as part of specification.

    However they could be used only after unit and/or system is available.
  - Preparation of Structural test cases could be part of implementation/code phase.
  - Unit, Integration and System testing are performed in order.

Black box testing

- Equivalence class partitioning
- Boundary value analysis
- Decision Table based testing
- Cause Effect Graph

Equivalence Class Partitioning

- Input values to a program are partitioned into equivalence classes.
- Partitioning is done such that:
  - program behaves in similar ways to every input value belonging to an equivalence class.

Why define equivalence classes?

- Test the code with just one representative value from each equivalence class:
  - as good as testing using any other values from the equivalence classes.

Equivalence Class Partitioning

- How do you determine the equivalence classes?
  - examine the input data.
  - few general guidelines for determining the equivalence classes can be given

Equivalence Class Partitioning

- If the input data to the program is specified by a range of values:
  - e.g. numbers between 1 to 5000.
  - one valid and two invalid equivalence classes are defined.

Equivalence Class Partitioning

- If input is an enumerated set of values:
  - e.g. {a,b,c}
  - one equivalence class for valid input values
  - another equivalence class for invalid input values should be defined.

Example

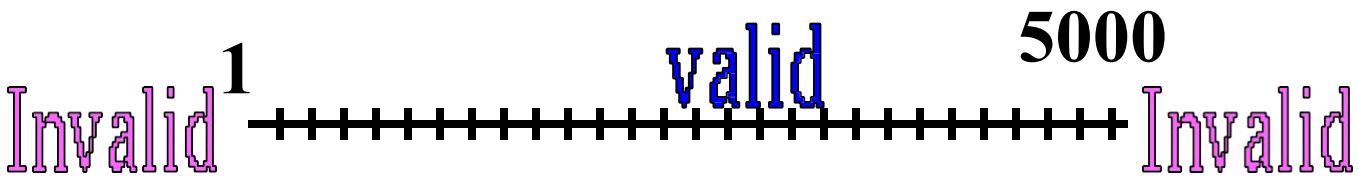
- A program reads an input value in the range of 1 and 5000:
  - computes the square root of the input number

Example (cont.)

- There are three equivalence classes:
  - the set of negative integers,
  - set of integers in the range of 1 and 5000,
  - integers larger than 5000.

Example (cont.)

- The test suite must include:
  - representatives from each of the three equivalence classes:
  - a possible test suite can be: {-5,500,6000}.



Boundary Value Analysis

- Some typical programming errors occur:
  - at boundaries of equivalence classes
  - might be purely due to psychological factors.
- Programmers often fail to see:
  - special processing required 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.

Example

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

Boundary value Analysis

- 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 te

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 $a=0$

The boundary value test cases are :

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

Equivalence Class Testing

- 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.

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

O_1={<a,b,c>:Not a quadratic equation if a = 0}
O_1={<a,b,c>:Real roots if $(b^2-4ac)>0$}
O_1={<a,b,c>:Imaginary roots if $(b^2-4ac)<0$}
O_1={<a,b,c>:Equal roots if $(b^2-4ac)=0$}`

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

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

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

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

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

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

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

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

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

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

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

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

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

Test Case	a	b	c	Expected output
1	0	50	50	Not a quadratic equation
2	-1	50	50	Invalid input
3	50	50	50	Imaginary Roots
4	101	50	50	invalid input
5	50	50	50	Imaginary Roots
6	50	-1	50	invalid input
7	50	101	50	invalid input
8	50	50	50	Imaginary Roots
9	50	50	-1	invalid input
10	50	50	101	invalid input

Here test cases 5 and 8 are redundant test cases. If we choose any value other than nominal, we may not have redundant test cases. Hence total test cases are 10+4=14 for this problem.

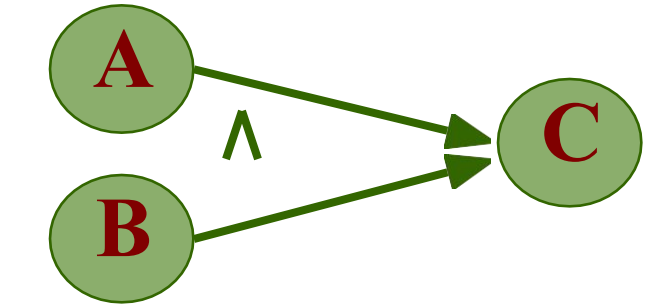
Cause and Effect Graphs

- A “Cause” stands for a distinct input condition that fetches about an internal change in the system.
- An “Effect” represents an output condition, a system state that results from a combination of causes.

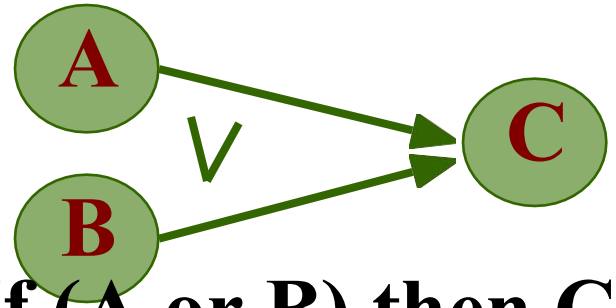
Cause and Effect Graphs

- Convert the graph to a decision table:
  - each column of the decision table corresponds to a test case for functional testing.

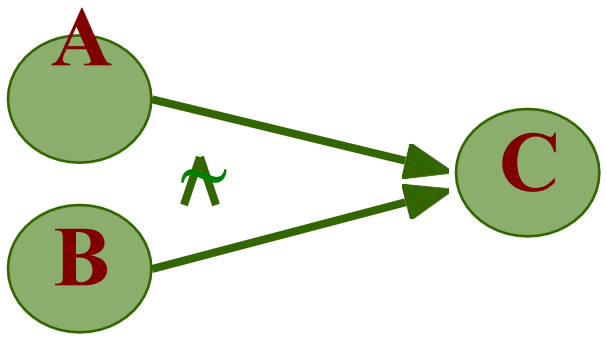
Drawing Cause-Effect Graphs



Drawing Cause-Effect Graphs

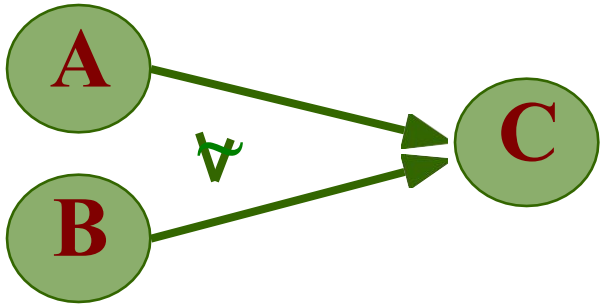


If (A or B) then C



If (not(A and B)) then C

Drawing Cause-Effect Graphs



If (not (A or B))then C

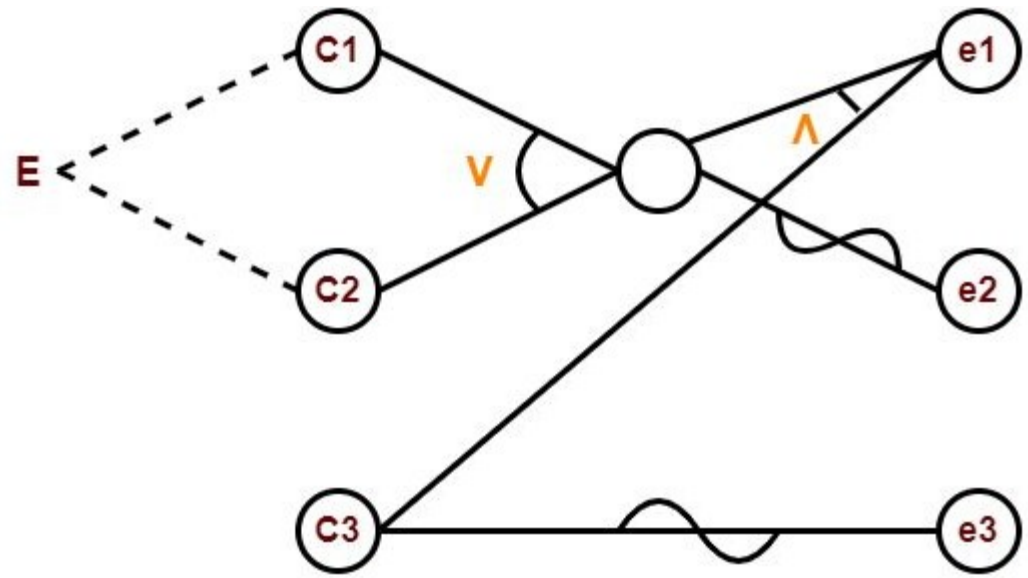
If (not A) then B

EXAMPLE

- Design test cases for the following problem-
- If the character of the first column is ‘A’ or ‘B’ and the second column is a number, then the file is considered updated. If the first character is erroneous, then message x should be printed. If the second column is not a number, then message y should be printed.

- Identify and describe the input conditions (causes) and actions (effect).
- The causes represented by letter “C” are as follows-
- C1 : The character in column 1 is ‘A’
- C2 : The character in column 1 is ‘B’
- C3 : The character in column 2 is a number

- The effects represented by letter “e” are as follows-
- e1 : File update is made
- e2 : Message x is printed


E
C1
C2
C3
V
Λ
e1
e2
e3

Decision Table

- Two dimensional mapping of *condition* against *actions* to be performed
  - Conditions evaluate to Boolean
  - Action corresponds to expected activity
- They can be derived from Cause Effect graph too
  - Map cause as condition
  - Map effect as action

- The decision table works on input conditions and actions.
- That is to say, we create a Table in which the top rows are input conditions, and in the same vein, the bottom rows are resulting actions. Similarly, the columns correspond to unique combinations of these conditions.

- ***Common Notations for Decision Table***

- ***For Conditions***
  - *Y means the condition is True. We can depict it as **T** Or **1.***
  - *N indicates the condition is False. We can also refer to it as **F** Or **0.***
  - *– It means the condition doesn't matter. We can represent it as **N/A.***

- ***For Actions***
  - ***X** means action should occur. We can also refer to it as **Y** Or **T** Or **1.***
  - *Blank means action should not happen. We can also represent it as **N** Or **F** Or **0.***

Writing Decision Table Based On Cause And Effect graph

First, write down the Causes and Effects in a single column shown below

Actions
C1
C2
C3
E1
E2
E3

The Key is the same. Go from bottom to top which means traverse from Effect to Cause.

Start with Effect E1. For E1 to be true, the condition is (C1 ∨ C2) ∧ C3.
Here we are representing True as **1** and False as **0**

First, put Effect E1 as True in the next column as

Actions	
C1	
C2	
C3	
E1	1
E2	
E3	

Now for E1 to be “1” (true), we have the below two conditions –
C1 AND C3 will be true
C2 AND C3 will be true

Actions		
C1	1	
C2		1
C3	1	1
E1	1	1
E2		
E3		

For E2 to be True, either C1 or C2 has to be False shown as,

Actions				
C1	1		0	
C2		1		0
C3	1	1	0	1
E1	1	1		
E2			1	1
E3				

For E3 to be true, C3 should be false.

Actions						
C1	1		0		1	
C2		1		0		1
C3	1	1	0	1		
E1	1	1				
E2			1	1		
E3					1	1

Actions	TC1	TC2	TC3	TC4	TC5	TC6
C1	1	0	0	0	1	0
C2	0	1	0	0	0	1
C3	1	1	0	1	0	0
E1	1	1	0	0	0	0
E2	0	0	1	1	0	0
E3	0	0	0	0	1	1

Writing Test Cases From The Decision Table

Below is a sample test case for Test Case 1 (TC1) and Test Case 2 (TC2).

TC ID	TC Name	Description	Steps	Expected result
TC1	TC1_FileUpdate Scenario1	Validate that system updates the file when first character is A and second character is a digit.	1. Open the application. 2. Enter first character as "A" 3. Enter second character as a digit	File is updated.
TC2	TC2_FileUpdate Scenario2	Validate that system updates the file when first character is B and second character is a digit.	1. Open the application. 2. Enter first character as "B" 3. Enter second character as a digit	File is updated.

- *Consider a Banking application that will ask the user to fill a personal loan application online. Based on the inputs, the application will display real-time whether the loan will get approval, rejection, or requires a visit to the branch for further documentation and discussion. Let's assume that the loan amount is 5L. In addition to this, we won't change it to reduce the complexity of this scenario.*

- *Accordingly, the application has the following business rules:*

- *If you are Salaried and your Monthly Salary is greater than or equal to 75k, then your loan will be approved.*

- *If you are Salaried and your Monthly Salary is between 25k and 75k, then you will need to visit the branch for further discussion.*

- *If you are Salaried and your Monthly Salary is less than 25k, then your loan will be rejected.*

STEPS

- ***Step 1 – Identify all possible Conditions***
- Consequently, the possible conditions are as below:
- *First, whether the person is Salaried or not.*
- *Second, Monthly Salary of the applicant.*
-
- ***Step 2 – Identify the corresponding actions that may occur in the system***
- Therefore, the possible actions are:
- *Should the loan be approved (Possible values Y or N)*
- *Should the loan be rejected (Possible values Y or N)*
- *Should the applicant be called for further documentation and discussion (Possible values Y or N)*

- ***Step 3 – Generate All possible Combinations of Conditions***
- Each of these combinations forms a column of the decision table. Firstly, let's see how many variations are possible for each condition:
- ***Condition 1*** *– Whether the person is salaried or not – 2 Variations: Y or N*
- ***Condition 2*** *– Monthly Salary of the Applicant – 3 Variations : <25k , 25k – 75k and >75k*
- Subsequently, based on this our total combinations will come up as 2*3 = 6
- ***1st Combination*** *: Salaried = Yes , Monthly Salary < 25k*
- ***2nd Combination:*** *Salaried = Yes, Monthly Salary 25k – 75k*
- ***3rd Combination:*** *Salaried = Yes, Monthly Salary >75k*
- ***4th Combination*** *: Salaried = No , Monthly Salary < 25k*
- ***5th Combination:*** *Salaried = No, Monthly Salary 25k – 75k*
- ***6th Combination:*** *Salaried = No, Monthly Salary >75k*

- **Step 4 – Identify Actions based on the combination of conditions**
- Subsequently, the next step will be identifying actions for each combination based on the business logic as defined for the system.
- ***1st Combination*** *– Action = Loan Rejected*
- ***2nd Combination*** *– Action = Further Documentation & Visit*
- ***3rd Combination*** *– Action = Loan Approved*
- ***4th Combination*** *– Action = Loan Rejected*
- ***5th Combination*** *– Action = Loan Rejected*
- ***6th Combination*** *– Action = Further Documentation & Visit*

CONDITIONS	Combination # 1	Combination # 2	Combination # 3	Combination # 4	Combination # 5	Combination # 6
Salaried?	Y	Y	Y	N	N	N
Monthly Income < 25k	Y	NA	NA	Y	NA	NA
Monthly Income 25k – 75k	NA	Y	NA	NA	Y	NA
Monthly Income >75k	NA	NA	Y	NA	NA	Y
ACTIONS						
Loan Approved	NA	NA	Y	NA	NA	NA
Loan Rejected	Y	NA	NA	Y	Y	NA
Further Documentation & Visit	NA	Y	NA	NA	NA	Y

White-Box Testing

- Statement coverage
- Branch coverage
- Path coverage
- Condition coverage
- Mutation testing
- Data flow-based testing

Coverage Metrics

- ***Statement coverage***: all statements in the programs should be executed at least once
- ***Branch coverage***: all branches in the program should be executed at least once
- ***Path coverage***: all execution paths in the program should be executed at lest once

Statement Coverage

- Choose a test set T such that by executing program P for each test case in T, each basic statement of P is executed at least once
- Executing a statement once and observing that it behaves correctly is not a guarantee for correctness, but it is an heuristic
  - this goes for all testing efforts since in general checking correctness is undecidable

```
bool isEqual(int x, int y)
{
  if (x = y)
    z := false;
  else
    z := false;
  return z;
}
```

```
int max(int x, int y)
{
  if (x > y)
    return x;
  else
    return x;
}
```

Statement Coverage

```
areTheyPositive(int x, int y)
{
  if (x >= 0)
    print("x is positive");
  else
    print("x is negative");
  if (y >= 0)
    print("y is positive");
  else
     print("y is negative");
}
```

Following test set will give us statement coverage:

$T = \{(x=12,y=5), (x=-1,y=35), (x=115,y=-13),(x=-91,y=-2)\}$

There are smaller test cases which will give us statement coverage too:

$T_2 = \{(x=12,y=-5), (x=-1,y=35)\}$

There is a difference between these two test sets though

Control flow graph (CFG)

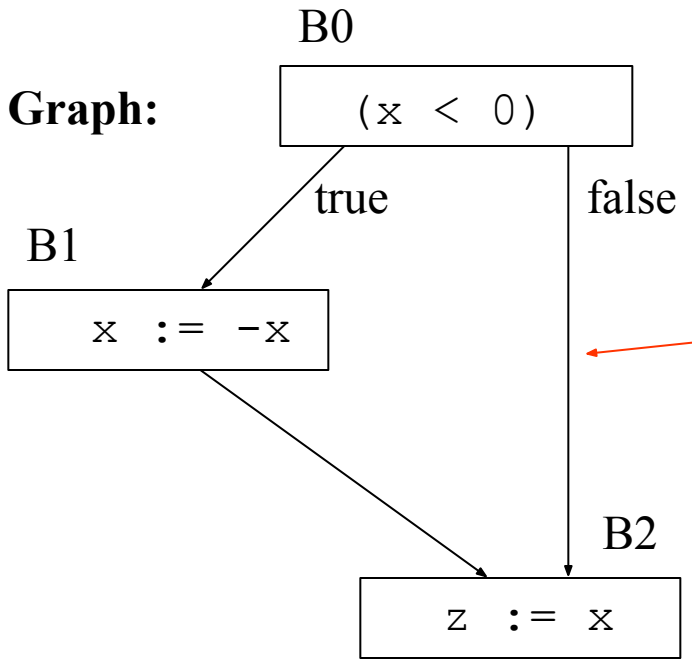
- A control flow graph (CFG) describes:
  - the sequence in which different instructions of a program get executed.
  - the way control flows through the program.

Statement vs. Branch Coverage

```
assignAbsolute(int x)
{
  if (x < 0)
    x := -x;
  z := x;
}
```

Consider this program segment, the test set T = {x=−1} will give statement coverage, however not branch coverage

Control Flow Graph:



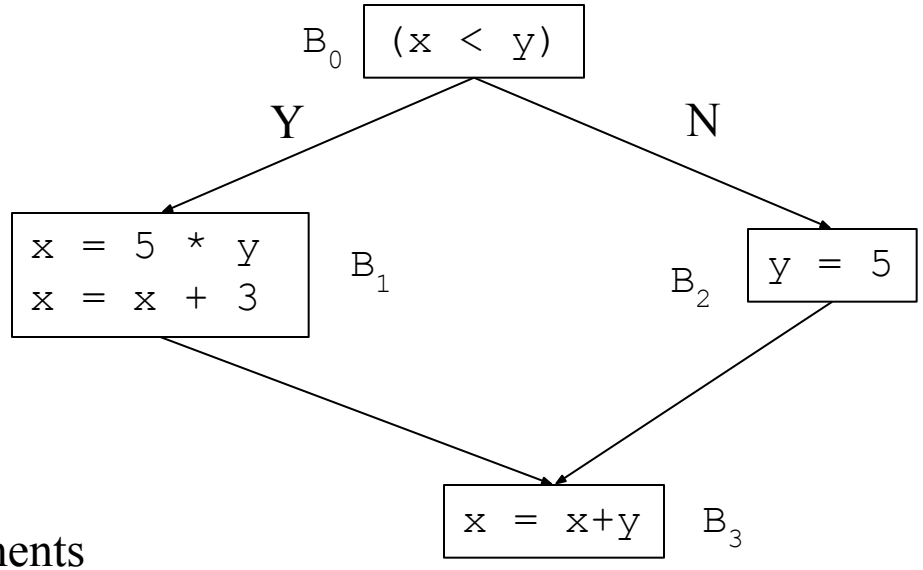
Test set {x=−1} does not execute this edge, hence, it does not give branch coverage

CFG

- Nodes in the control flow graph are basic blocks
  - A ***basic block*** is a sequence of statements always entered at the beginning of the block and exited at the end
- Edges in the control flow graph represent the control flow

Control Flow Graphs (CFGs)

```
if (x < y) {
  x = 5 * y;
  x = x + 3;
}
else
  y = 5;
x = x+y;
```

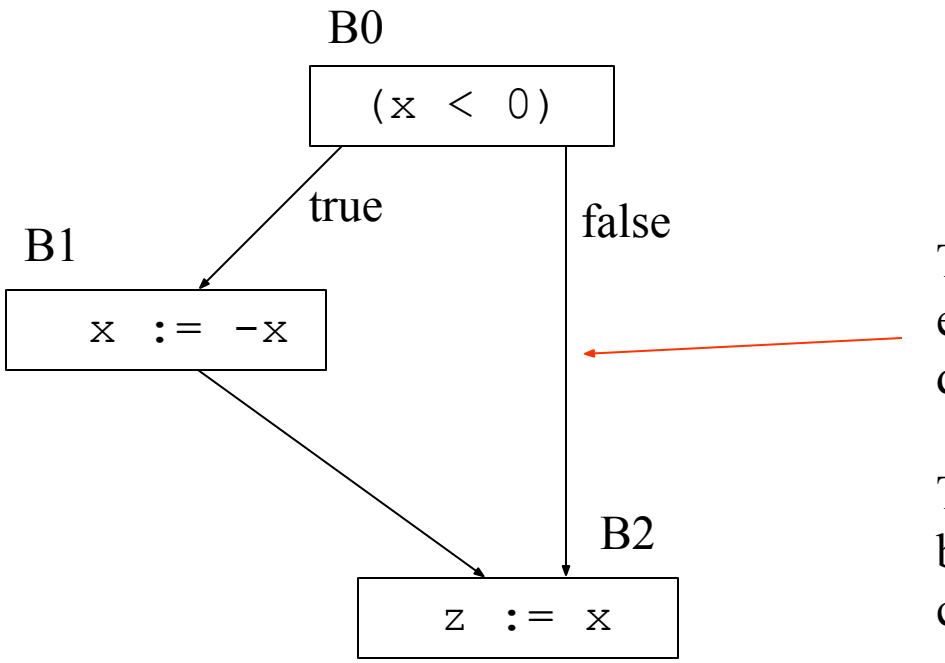


- Each block has a sequence of statements
- No jump from or to the middle of the block
- Once a block starts executing, it will execute till the end

Branch Coverage

- Construct the control flow graph

- Select a test set *T* such that by executing program *P* for each test case *d* in *T*, each edge of *P*'s control flow graph is traversed at least once



Test set {x=−1} does not execute this edge, hence, it does not give branch coverage

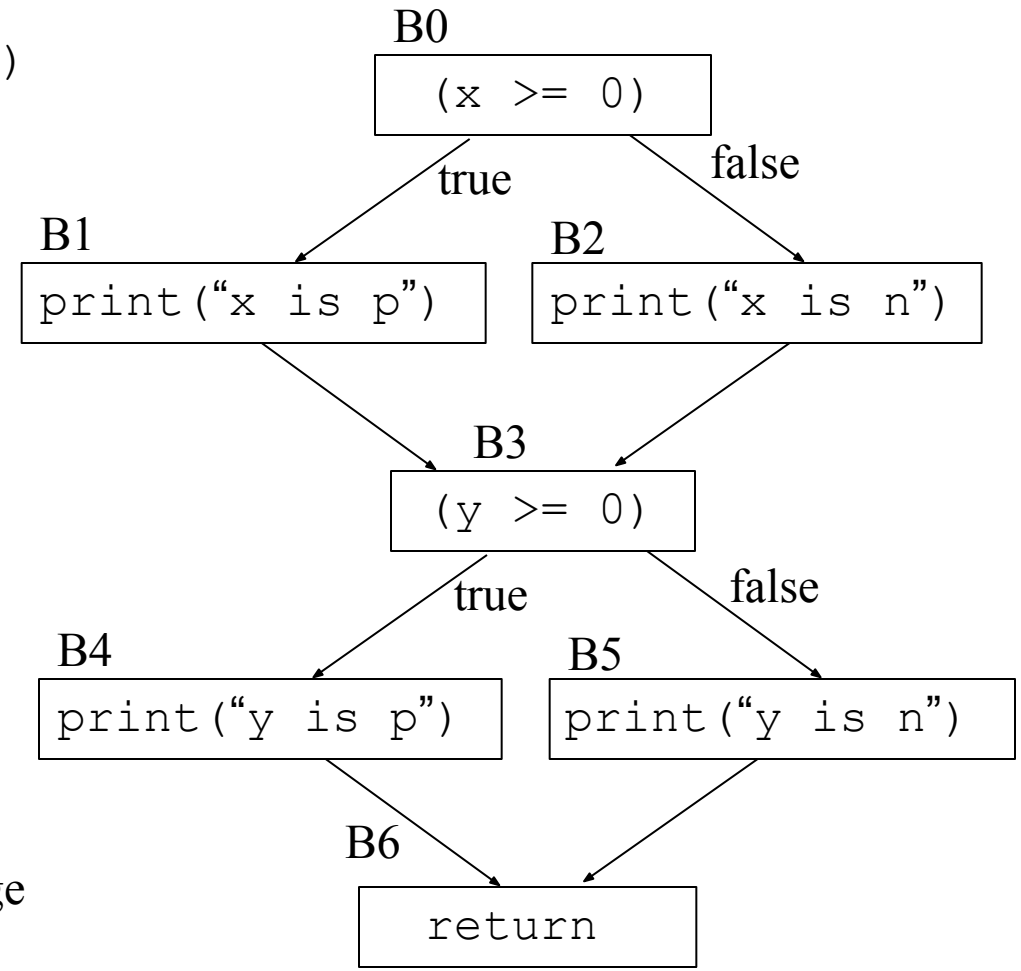
Test set {x= −1, x=2}gives both statement and branch coverage

Path Coverage

- Select a test set *T* such that by executing program *P* for each test case *d* in *T*, all paths leading from the initial to the final node of P's control flow graph are traversed

Path Coverage

```
areTheyPositive(int x, int y)
{
  if (x >= 0)
     print("x is positive");
  else
    print("x is negative");
  if (y >= 0)
    print("y is positive");
  else
     print("y is negative");
}
```



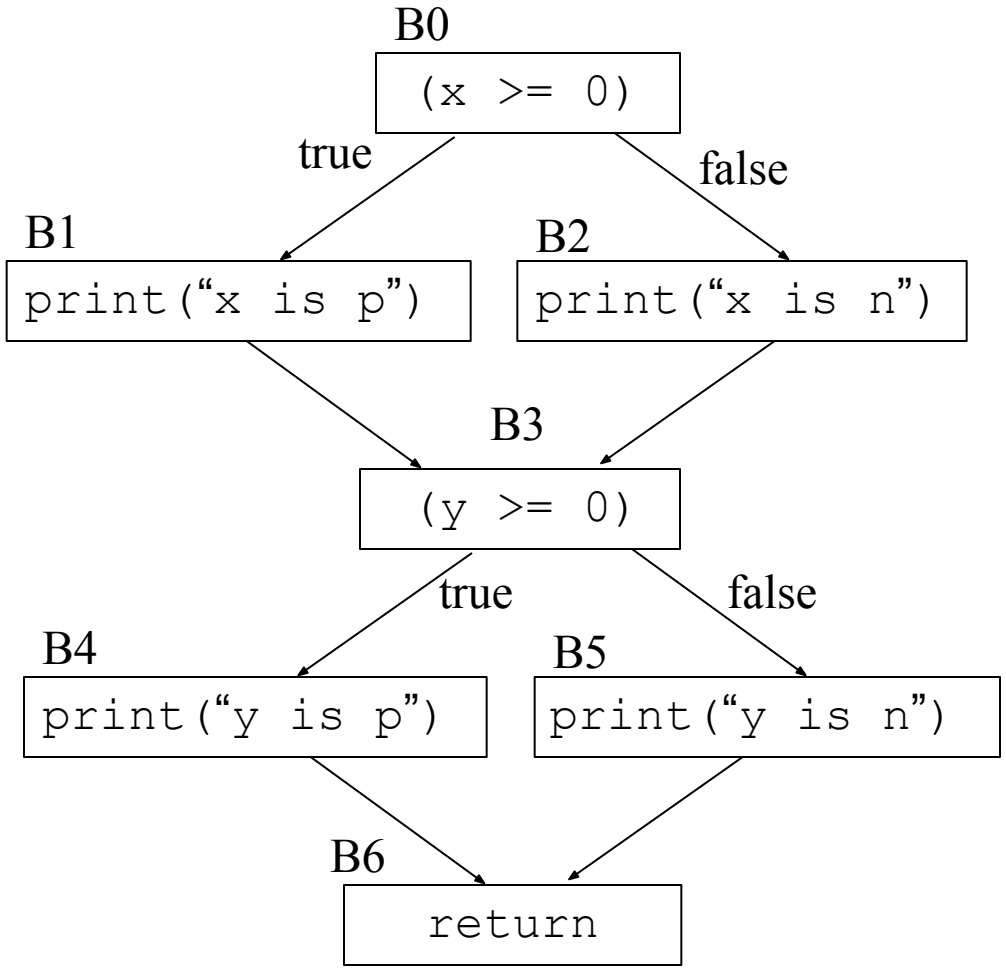
Test set:
T_2 = {(x=12,y= − 5), (x= −1,y=35)}
gives both branch and statement coverage but it does not give path coverage

Set of all execution paths: {(B0,B1,B3,B4,B6), (B0,B1,B3,B5,B6), (B0,B2,B3,B4,B6), (B0,B2,B3,B5,B6)}
Test set T_2 executes only paths: (B0,B1,B3,B5,B6) and (B0,B2,B3,B4,B6)

Path Coverage

```
areTheyPositive(int x, int y)
{
  if (x >= 0)
     print("x is positive");
  else
    print("x is negative");
  if (y >= 0)
    print("y is positive");
  else
     print("y is negative");
}
```

Test set:
T_1 = {(x=12,y=5), (x= −1,y=35), (x=115,y=−13),(x=−91,y= −2)}
gives both branch, statement and path coverage



Path Coverage

- Number of paths is exponential in the number of conditional branches
  - testing cost may be expensive
- Note that every path in the control flow graphs may not be executable
  - It is possible that there are paths which will never be executed due to dependencies between branch conditions
- In the presence of cycles in the control flow graph (for example loops) we need to clarify what we mean by path coverage

Path Coverage

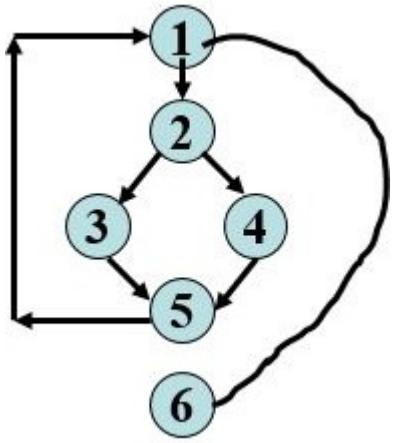
- Design test cases such that:
  - all linearly independent paths in the program are executed at least once.
- Defined in terms of
  - control flow graph (CFG) of a program.

How to draw Control flow graph?

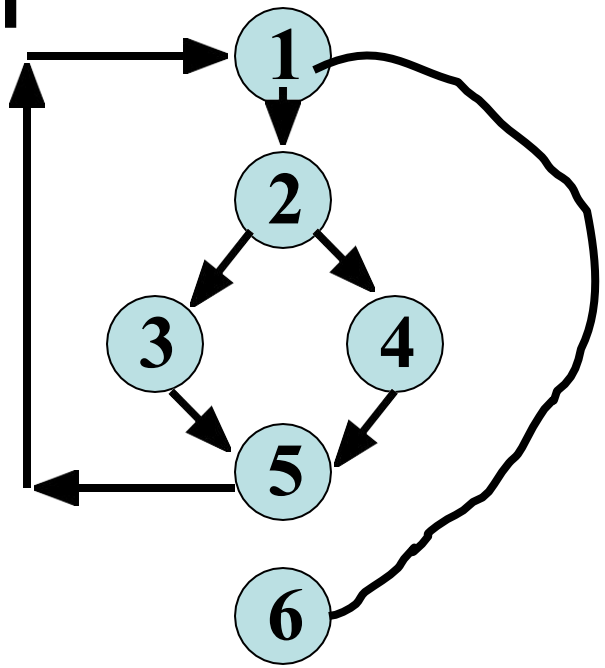
- Number all the statements of a program.
- Numbered statements:
  - represent nodes of the control flow graph.
- An edge from one node to another node exists:
  - if execution of the statement representing the first node can result in transfer of control to the other node.

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



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.

McCabe's cyclomatic metric

- 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

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

Cyclomatic complexity

- Another way of computing cyclomatic complexity:
  - determine number of bounded areas in the graph
    - Any region enclosed by a nodes and edge sequence.
- V(G) = Total number of bounded areas + 1

Example

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

Cyclomatic complexity

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

Cyclomatic complexity

- The cyclomatic complexity of a program provides:
  - a lower bound on the number of test cases to be designed
  - to guarantee coverage of all linearly independent paths.

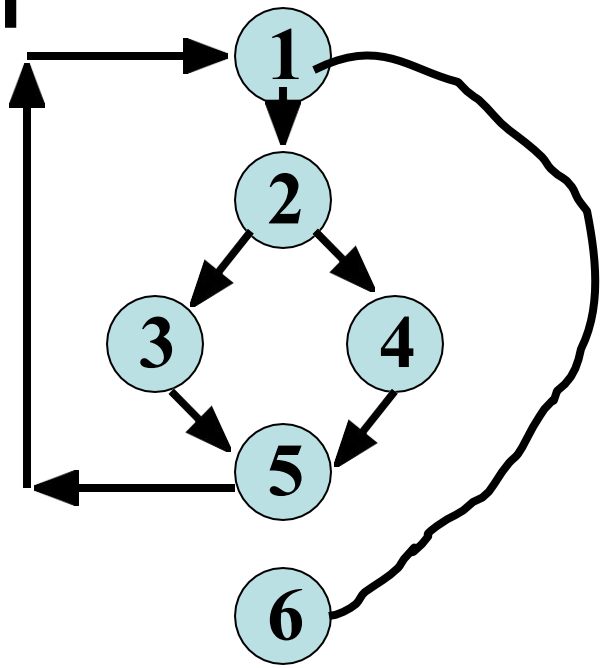
Cyclomatic complexity

- Defines the number of independent paths in a program.
- Provides a lower bound:
  - for the number of test cases for path coverage.
- only gives an indication of the minimum number of test cases required.

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 Control Flow Graph



Derivation of Test Cases

- Number of independent paths: 3
  - 1, 6 test case (x=1, y=1)
  - 1, 2, 3, 5, 1, 6 test case(x=11, y=2)
  - 1, 2, 4, 5, 1, 6 test case(x=2, y=31)

An interesting 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

- Cyclomatic complexity of a program:
  - also indicates the psychological complexity of a program.
  - difficulty level of understanding the program.

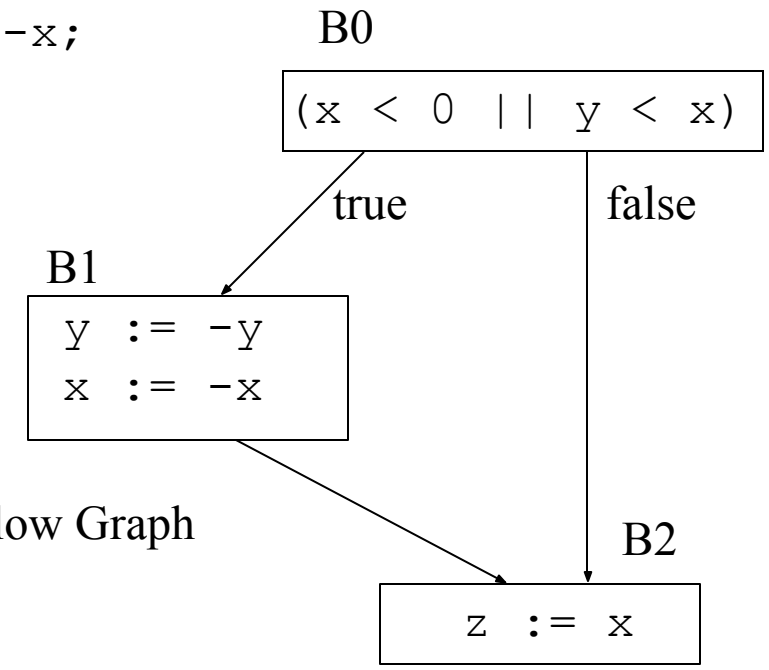
Condition Coverage

- In the branch coverage we make sure that we execute every branch at least once
  - For conditional branches, this means that, we execute the TRUE branch at least once and the FALSE branch at least once
- Conditions for conditional branches can be compound boolean expressions
  - A compound boolean expression consists of a combination of boolean terms combined with logical connectives AND, OR, and NOT
- Condition coverage:
  - Select a test set *T* such that by executing program *P* for each test case *d* in *T*, **(1)** each edge of *P*'s control flow graph is traversed at least once **and (2)** each boolean term that appears in a branch condition takes the value TRUE at least once and the value FALSE at least once
- Condition coverage is a refinement of branch coverage (part (1) is same as the branch coverage)

Condition Coverage

```
something(int x)
{
  if (x < 0 ||  y < x)
  {
    y := -y;
    x := -x;
  }
  z := x;
}
```

T = {(x=−1, y=1), (x=1, y=1)} will achieve statement, branch and path coverage, however T will not achieve condition coverage because the boolean term `(y < x)` never evaluates to true. This test set satisfies part (1) but does not satisfy part (2).



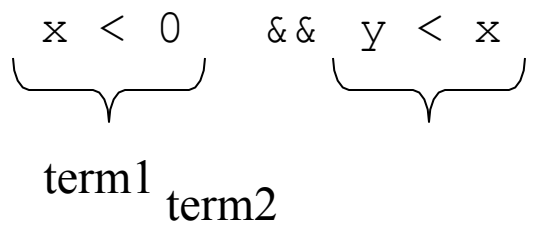
Control Flow Graph

T = {(x=−1, y=1), (x=1, y=0)} will not achieve condition coverage either. This test set satisfies part (2) but does not satisfy part (1). It does not achieve branch coverage since both test cases take the true branch, and, hence, it does not achieve condition coverage by definition.

T = {(x=−1, y=−2), {(x=1, y=1)} achieves condition coverage.

Multiple Condition Coverage

- Multiple Condition Coverage requires that all possible combination of truth assignments for the boolean terms in each branch condition should happen at least once
- For example for the previous example we had:



- Test set {(x=−1, y=−2), (x=1, y=1)}, achieves condition coverage:
  - test case (x=−1, y=−2) makes term1=true, term2=true, and the whole expression evaluates to true (i.e., we take the true branch)
  - test case (x=1, y=1) makes term1=false, term2=false, and the whole expression evaluates to false (i.e., we take the false branch)
- However, test set {(x=−1, y= −2), (x=1, y=1)} does not achieve multiple condition coverage since we did not observe the following truth assignments
  - term1=true, term2=false
  - term1=false, term2=true

Path testing

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

Data Flow-Based Testing

Data Flow Testing is a type of structural testing. It is a method that is used to find the test paths of a program according to the locations of definitions and uses of variables in the program. It has nothing to do with data flow diagrams.

It is concerned with:

Statements where variables receive values,

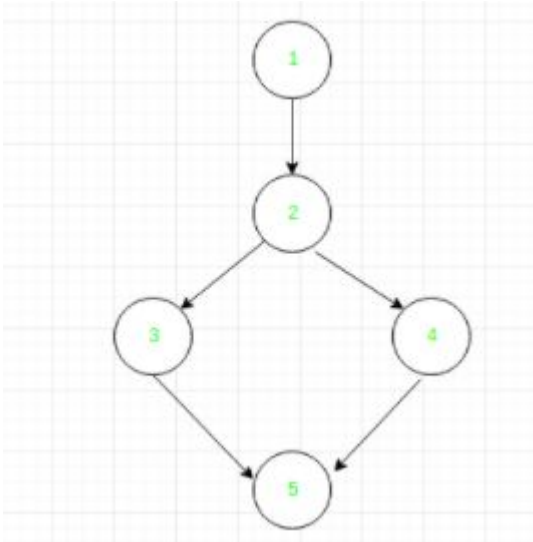
Statements where these values are used or referenced.

Data Flow-Based Testing

$$DEF(S) = \{X \mid \text{statement } S \text{ contains the definition of } X\}$$

$$USE(S) = \{X \mid \text{statement } S \text{ contains the use of } X\}$$

```
1. read x, y;
2. if(x>y)
3. a = x+1
else
4. a = y-1
5. print a;
```



Variable	Defined at node	Used at node
x	1	2, 3
y	1	2, 4
a	3, 4	5

Mutation Testing

- The software is first tested:
  - using an initial testing method based on 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

- 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.

Mutation Testing

- If a mutant remains alive:
  - even after all test cases have been 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

- The primitive changes can be:
  - altering an arithmetic operator,
  - changing the value of a constant,
  - changing a data type, etc.

Mutation Testing

- A major disadvantage of mutation testing:
  - computationally very expensive,
  - a large number of possible mutants can be generated.

Black Box Testing	*Grey Box Testing*	*White Box Testing*
The Internal Workings of an application are not required to be known	*Somewhat knowledge of the internal workings are known*	*Tester has full knowledge of the Internal workings of the application*
Performed by end users and also by testers and developers	*Performed by end users and also by testers and developers*	*Normally done by testers and developers*
Testing is based on external expectations. Internal behavior of the application is unknown	*Testing is done on the basis of high level database diagrams and data flow diagrams*	*Internal workings are fully known and the tester can design test data accordingly*
Also known as closed box testing, data driven testing and functional testing	*Another term for grey box testing is translucent testing as the tester has limited knowledge of the insides of the application*	*Also known as clear box testing, structural testing or code based testing*

Types of Testing

- Unit (Module) testing
  - testing of a single module in an isolated environment

- Integration testing
  - testing parts of the system by combining the modules

- System testing
  - testing of the system as a whole after the integration phase

- Acceptance testing
  - testing the system as a whole to find out if it satisfies the requirements specifications

Unit Testing

- Involves testing a single isolated module

- Note that unit testing allows us to isolate the errors to a single module
  - we know that if we find an error during unit testing it is in the module we are testing

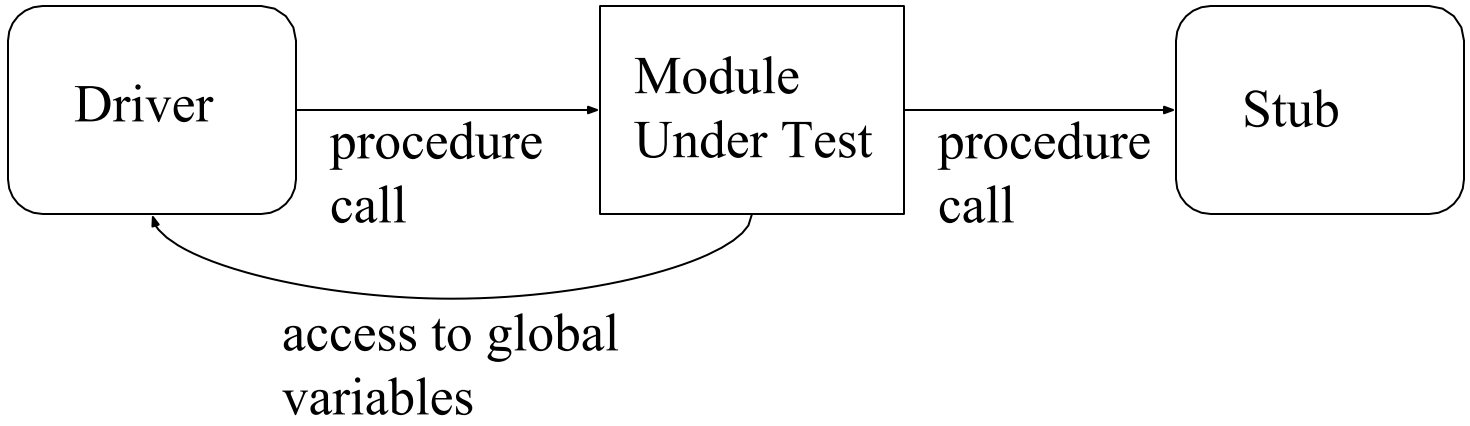
- Modules in a program are not isolated, they interact with each other. Possible interactions:
  - calling procedures in other modules
  - receiving procedure calls from other modules
  - sharing variables

- For unit testing we need to isolate the module we want to test, we do this using two things

Drivers and Stubs

- **Driver:** A program that calls the interface procedures of the module being tested and reports the results
  - A driver simulates a module that calls the module currently being tested
- **Stub:** A program that has the same interface as a module that is being used by the module being tested, but is simpler.
  - A stub simulates a module called by the module currently being tested
  - Mock objects: Create an object that mimics only

Drivers and Stubs



- Driver and Stub should have the same interface as the modules they replace

- Driver and Stub should be simpler than the modules they replace

Integration Testing

- Integration testing: Integrated collection of modules tested as a group or partial system

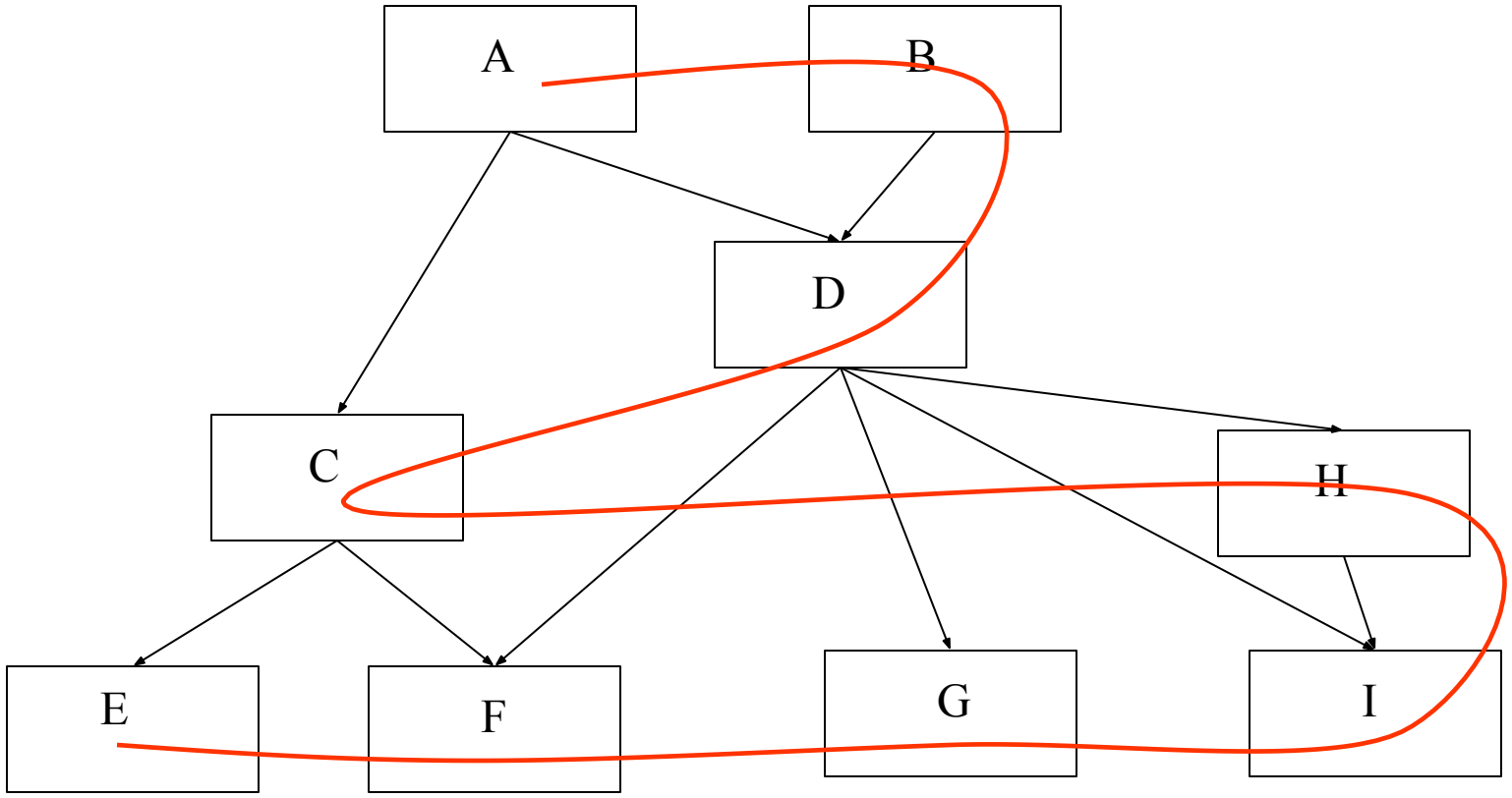
- Integration plan specifies the order in which to combine modules into partial systems

- Different approaches to integration testing
  - Bottom-up
  - Top-down

Bottom-Up Integration

- Only terminal modules (i.e., the modules that do not call other modules) are tested in isolation
- Modules at lower levels are tested using the previously tested higher 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

Bottom-up Integration



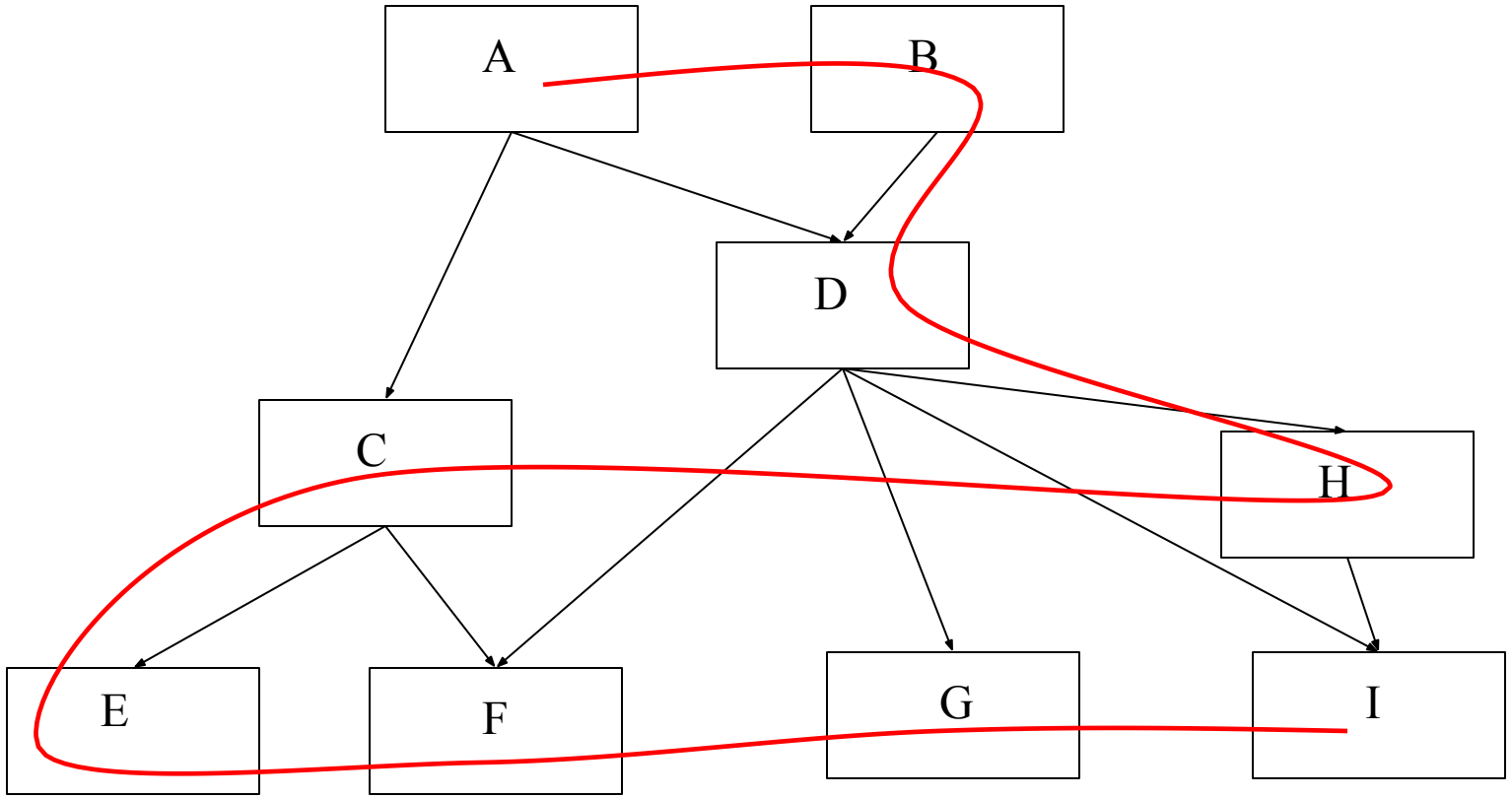
Top-down Integration

- 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 Approaches to Integration

- 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

- System testing involves:
  - validating a fully developed system against its requirements.
- There are three main kinds of system testing:
  - Alpha Testing
  - Beta Testing
  - Acceptance Testing

Regression testing

- During the maintenance phase, when a change is made to the program, the test cases that have been saved are used to do **regression testing**
  - figuring out if a change made to the program introduced any faults
- Regression testing is crucial during maintenance
  - It is a good idea to automate regression testing so that all test cases are run after each modification to the software
- When you find a bug in your program you should write a test case that exhibits the bug
  - Then using regression testing you can make sure

Alpha Testing
Done In A Lab Type Environment

Beta Testing
Includes A Limited Group Of Users Outside Of The Company

Twin Aims Of Beta Testing
To Get Customer Feedback And Inspiration To Do A Sanity Test

Phased versus Incremental Integration Testing

- Integration can be incremental or phased.
- In incremental integration testing,
  - only one new module is added to the partial system each time.

Performance Testing

- Addresses non-functional requirements.
  - May sometimes involve testing hardware and software together.
  - There are several categories of performance testing.

Stress testing

- Evaluates system performance
  - when stressed for short periods of time.
- Stress testing
  - also known as endurance testing.
- Stress tests are black box tests:
  - designed to impose a range of abnormal and even illegal input conditions
  - so as to stress the capabilities of the software.