MC/DC Testing

Recap: Condition Testing

- **Simple or (basic) Condition Testing:**
  - Test cases make each atomic condition to have both T and F values
  - Example: `if (a>10 && b<50)`
  - **The following test inputs would achieve basic condition coverage**
  - **a=15, b=30**
  - **a=5, b=60**
- Does basic condition coverage subsume decision coverage?

Condition Testing

- **Condition/decision coverage:**
  - Each atomic condition made to assume both T and F values
  - Decisions are also made to get T an F values
- **Multiple condition coverage (MCC):**
  - Atomic conditions made to assume all possible combinations of truth values

MCC

- Test cases make Conditions to assume all possible combinations of truth values.
- Consider: **if (a || b && c) then …**

Test	a	b	c
(1)	T	T	T
(2)	T	T	F
(3)	T	F	T
(4)	T	F	F
(5)	F	T	T
(6)	T	T	F
(7)	F	F	T
(8)	F	F	F

Exponential in the number of basic conditions

Shortcomings of Condition Testing

- **Redundancy of test cases:** Condition evaluation could be compiler-dependent:
  - Short circuit evaluation of conditions
- **Coverage may be Unachievable:** Possible dependencies among variables:
  - Example: **((chr==`A´)||(chr==`E´))** can not both be true at the same time

Short-circuit Evaluation

- **if(a>30 && b<50)…**
- If a>30 is FALSE compiler need not evaluate (b<50)

- Similarly, **if(a>30 || b<50)…**
- If a>30 is TRUE compiler need not evaluate (b<50)

Multiple Condition Coverage

- Consider a Boolean expression having n components:
  - For condition coverage we require 2^n test cases.
  - Therefore practical only if n (the number of component conditions) is small (two or three).

Compound conditions: Exponential complexity

(((a || b) && c) || d) && e

Test Case	a	b	c	d	e
(1)	T	—	T	—	T
(2)	F	T	T	—	T
(3)	T	—	F	T	T
(4)	F	T	F	T	T
(5)	F	F	—	T	T
(6)	T	—	T	—	F
(7)	F	T	T	—	F
(8)	T	—	F	T	F
(9)	F	T	F	T	F
(10)	F	F	—	T	F
(11)	T	—	F	F	—
(12)	F	T	F	F	—
(13)	F	F	—	F	—

$2^5=32$

- Short-circuit evaluation often reduces number of test cases to a more manageable number, but not always…

Modified Condition/Decision Coverage (MC/DC)

- **Motivation:** Effectively test important combinations of conditions, without exponential blowup to test suite size:
  - **"Important" combinations means:** Each basic condition shown to independently affect the outcome of each decision

- **Requires:**

  If((A==0) $\vee$ (B>5) $\wedge$ (C<100)) ….

  - For each basic condition c, two test cases
  - Values of all evaluated conditions except c remain the same
  - Compound condition as a whole evaluates to true for one and false for the other

Test Coverage Criteria

Condition/Decision Coverage

- Condition: true, false.
- Decision: true, false.

Multiple Condition coverage (MCC)

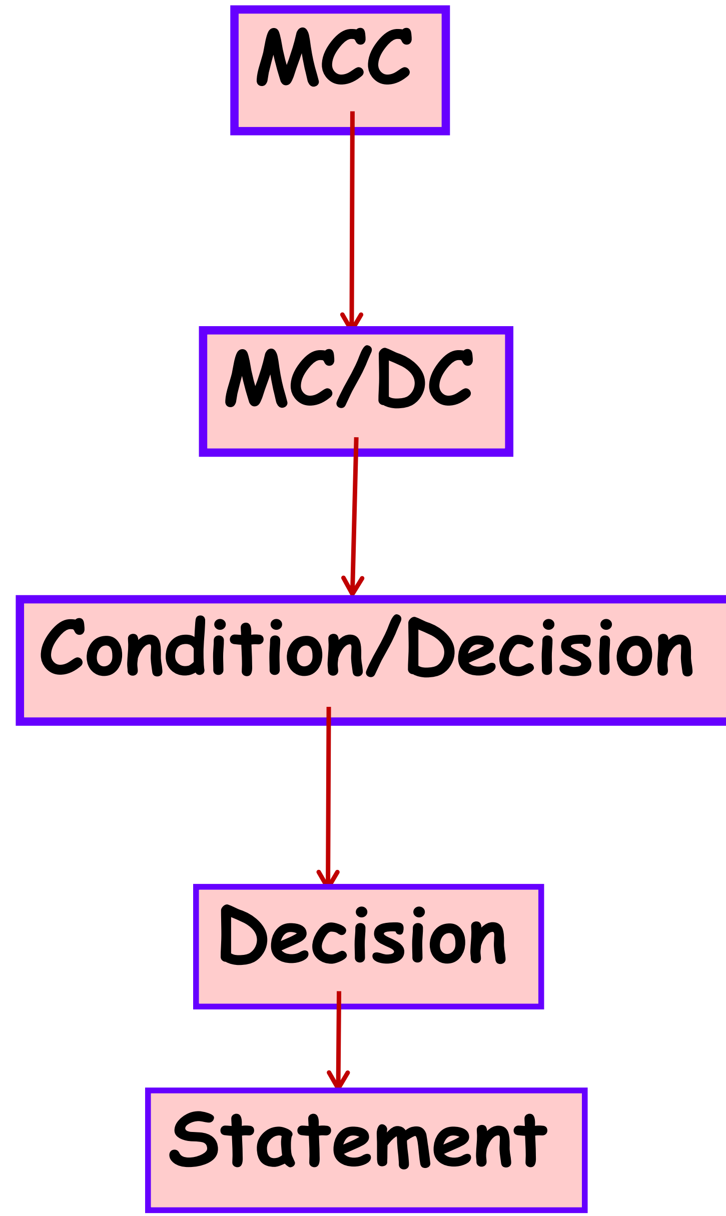
- all possible combinations of condition outcomes in a decision
- for a decision with **n** conditions

  2^n test cases are required

Modified Condition/Decision coverage (MC/DC)

- Bug-detection effectiveness almost similar to MCC
- Number of test cases linear in the number of basic conditions.

Subsumption hierarchy



What is MC/DC?

- MC/DC stands for **Modified Condition / Decision Coverage**
- It is a condition coverage technique
  - Condition: Atomic conditions in expression.
  - Decision: Controls the program flow.
- **Main idea:** Each condition must be shown to independently affect the outcome of a decision.
  - The outcome of a decision changes as a result of changing a single condition.

Three Requirements in MC/DC

Requirement 1:

- Every decision in a program must take T/F values.

Requirement 2:

- Every condition in each decision must take T/F values.

Requirement 3:

- Each condition in a decision should independently affect the decision's outcome.

MC/DC Requirement 1

- The decision is made to take both T/F values.

If ((a>10) && ((b<50) && (c==0))) then

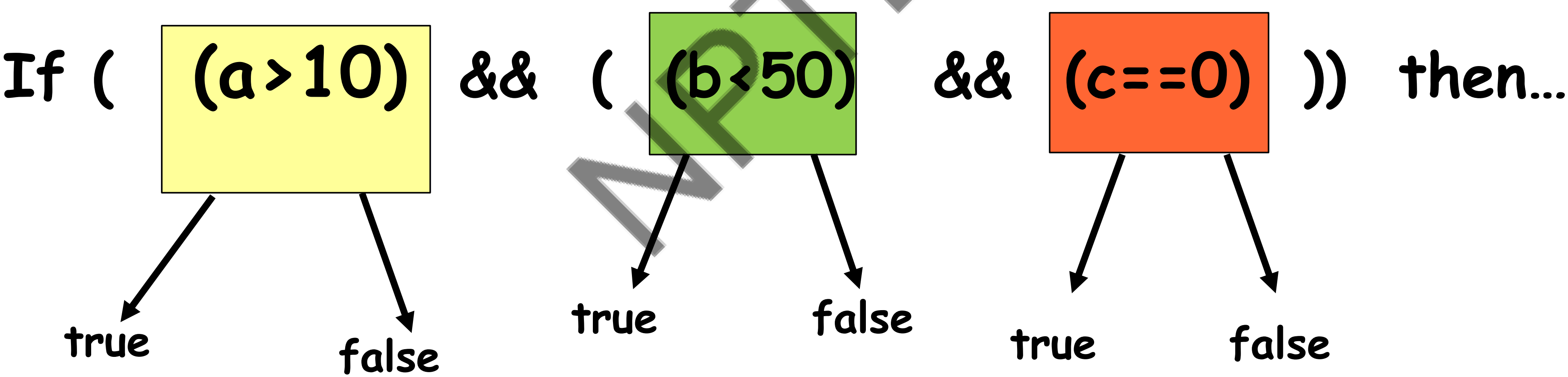
true

false

- This is as in Branch coverage.

MC/DC Requirement 2

- Test cases make every condition in the decision to evaluate to both T and F at least once.



MC/DC Requirement 3

- Every condition in the decision independently affects the decision's outcome.

If ((a>10) && ((b<50) || (c==0))) then…

true false true false

If ((a>50) && (b<50) || (c==0))) then…

true true false false

If ((a>50) && ((b<50) || (c==0))) then…

true false true false

MC/DC: Another Example

- N+1 test cases required for N basic conditions
- **Example:**

((((a>10 || b<50) && c==0) || d<5) && e==10)

Test Case	a>10	b<50	c==0	d<5	e==10	outcome
(1)	true	false	true	false	true	true
(2)	false	true	true	false	true	true
(3)	true	false	false	true	true	true
(6)	true	false	true	false	false	false
(11)	true	false	false	false	true	false
(13)	false	false	true	false	true	false

- Underlined values independently affect the output of the decision

Creating MC/DC test cases

- Create truth table for conditions.
- Extend the truth table to represent test case pair that lead to show the independence influence of each condition.

Example : If (A and B) then . . .

Test Case Number	A	B	Decision	Test case pair for A	Test case pair for B
1	T	T	T	3	2
2	T	F	F		1
3	F	T	F	1	
4	F	F	F		

- Show independence of A :
  - Take 1 + 3
- Show independence of B :
  - Take 1 + 2
- Resulting test cases are
  - 1 + 2 + 3

Another Example

If($(A \vee B) \wedge C$) ....

	A	B	C	Result	A	B	C	MC/DC
1	1	1	1	1			*	*
2	1	1	0	0			*	*
3	1	0	1	1	*			*
4	0	1	1	1		*		*
5	1	0	0	0				
6	0	1	0	0				
7	0	0	1	0	*	*		*
8	0	0	0	0				

Minimal Set Example

If (A and (B or C)) then…

TC#	ABC	Result	A	B	C
1	TTT	T	5		
2	TTF	T	6	4	
3	TFT	T	7		4
4	TFF	F		2	3
5	FTT	F	1		
6	FTF	F	2		
7	FFT	F	3		
8	FFF	F			

We want to determine the MINIMAL set of test cases

Here:

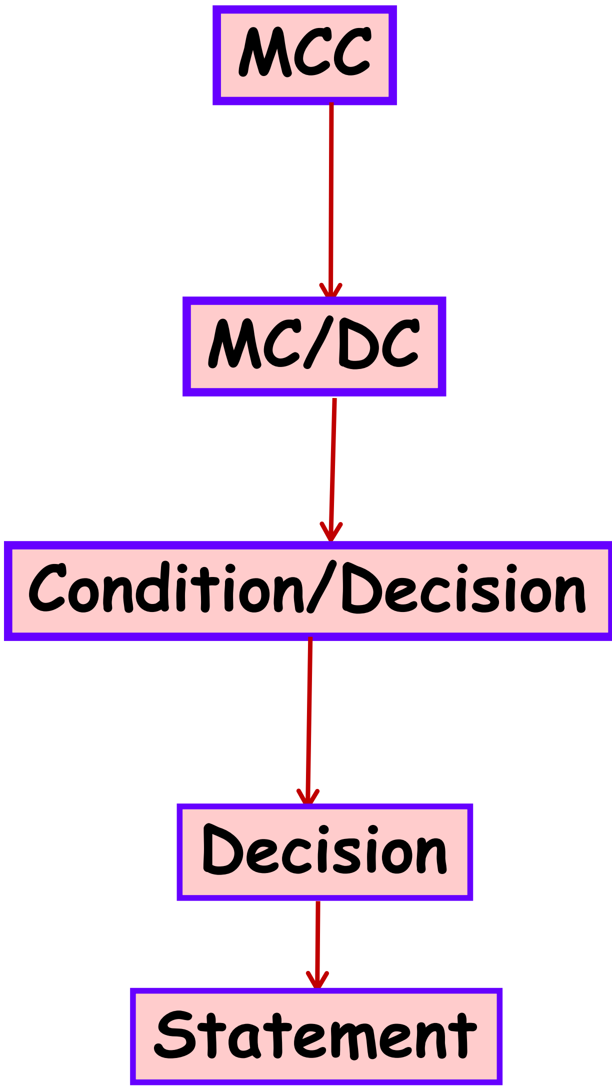
- {2,3,4,6}
- {2,3,4,7}

Non-minimal set is:

- {1,2,3,4,5}

Critique

- MC/DC criterion is stronger than condition/decision coverage criterion,
  - but the number of test cases to achieve the MC/DC criterions still linear in the number of conditions n in the decisions.



MC/DC: Summary

- MC/DC essentially is :
  - basic condition coverage (C)
  - branch coverage (DC)
  - plus one additional condition (M):
    every condition must *independently affect* the decision's output
- It is subsumed by MCC and subsumes all other criteria discussed so far
  - stronger than statement and branch coverage
- ***A good balance of thoroughness and test size and therefore widely used…***

Path Testing

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.

Path Coverage-Based Testing

- To understand the path coverage-based testing:
  - We need to learn how to draw control flow graph of a program.
- A control flow graph (CFG) describes:
  - The sequence in which different instructions of a program get executed.
  - The way control flows through the program.

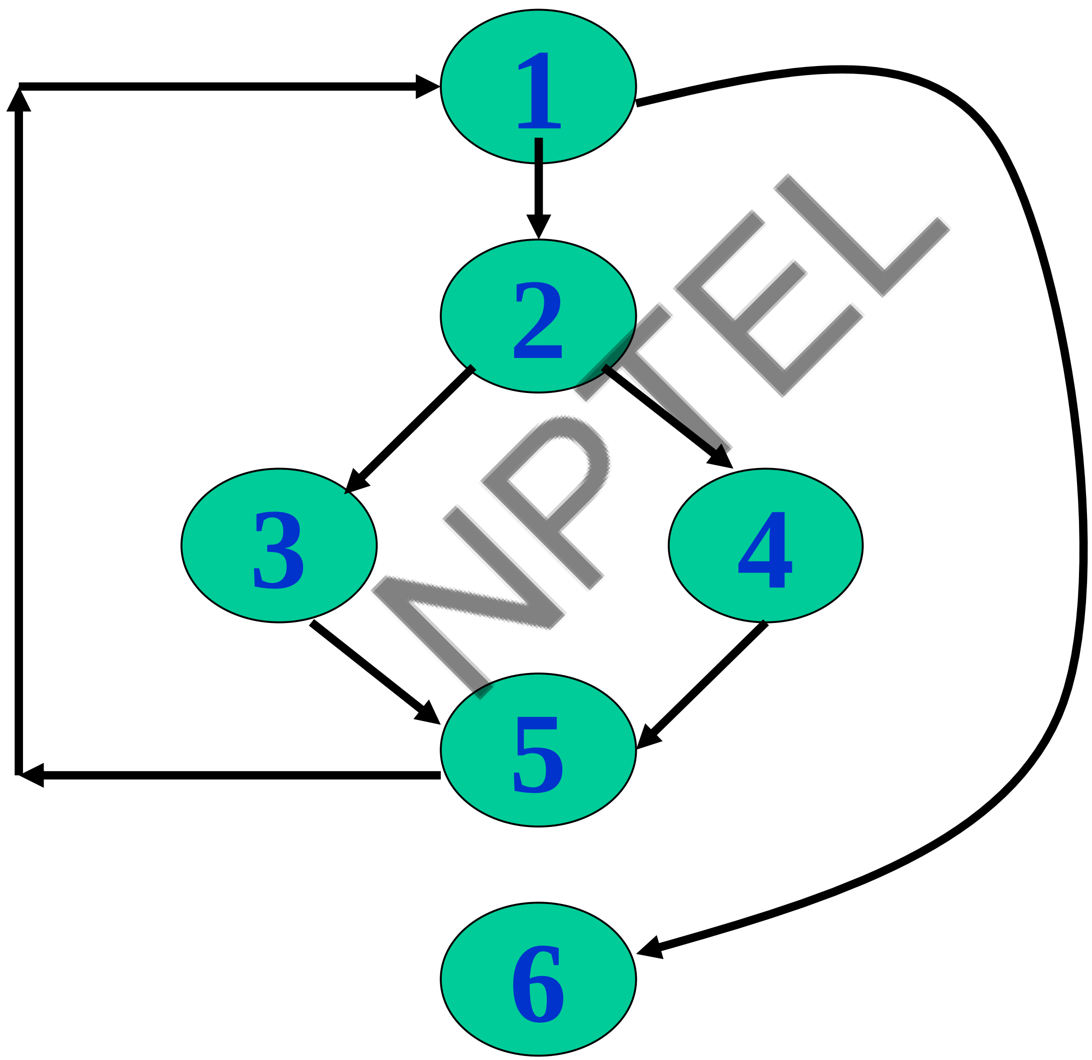
How to Draw Control Flow Graph?

- **Number all statements of a program.**
- Numbered statements:
  - Represent nodes of control flow graph.
- Draw an edge from one node to another node:
  - **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){
 while (x != y){
     if (x>y) then
         x=x-y;
     else y=y-x;
 }
 return x;          }
```

Example Control Flow Graph

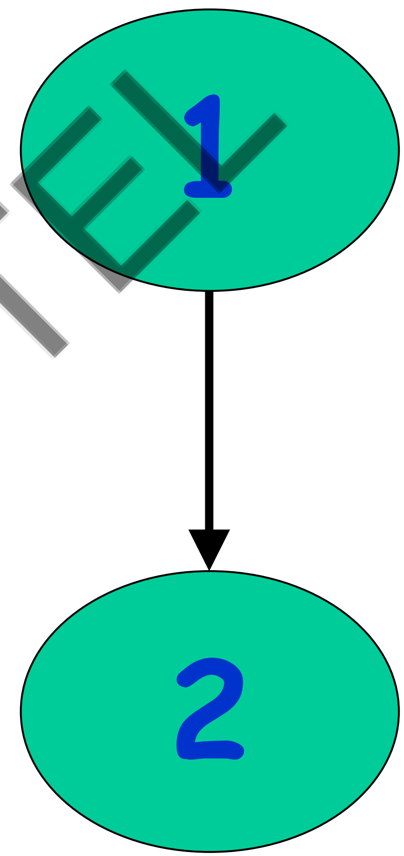


How to Draw Control flow Graph?

- Every program is composed of:
  - **Sequence**
  - **Selection**
  - **Iteration**
- If we know how to draw CFG corresponding these basic statements:
  - We can draw CFG for any program.

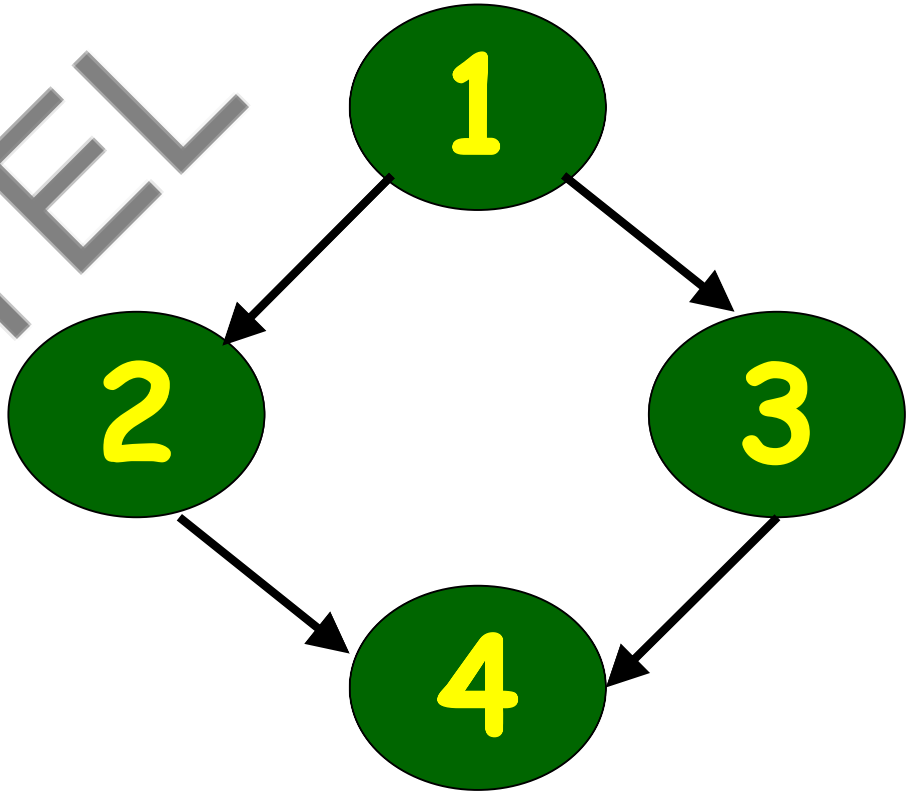
How to Draw Control flow Graph?

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



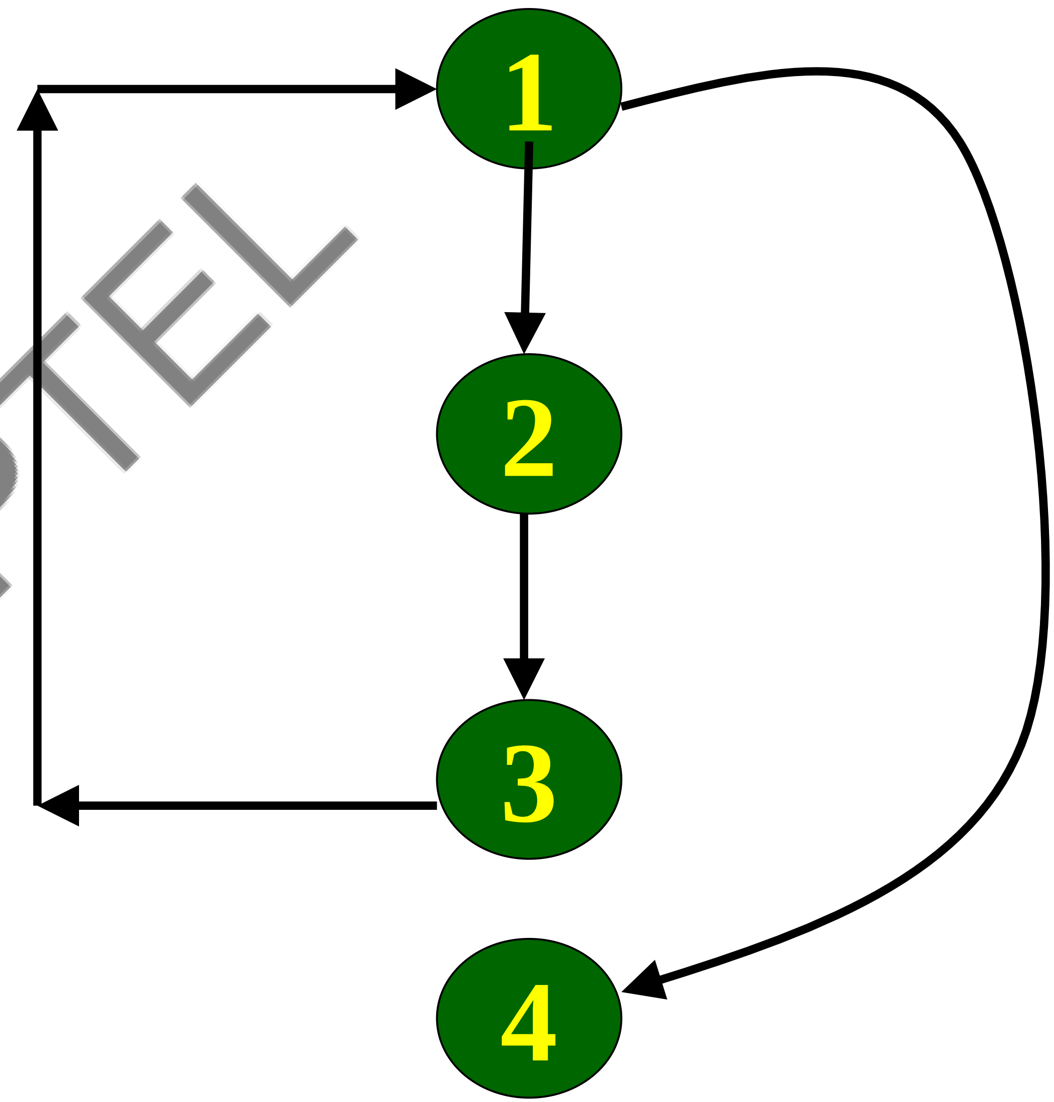
How to Draw Control Flow Graph?

- **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;



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.

All Path Criterion

- In the presence of loops, the number paths can become extremely large:
  - This makes all path testing impractical

Linearly Independent Paths

- A path is said to be a linear combination of paths p1, …, pn
  - If there are integers a1, …, an such that p =Σaipi (ai could be negative, zero, or positive)

- A set of paths is linearly independent if no path in the set is a linear combination of any other paths in the set

- A linearly independent path is any path through the program ("complete path") that introduces at least one new edge that is not included in any other linearly independent paths.

Linearly Independent Path

- Any path through the program that:
  - Introduces at least one new edge:
    - Not included in any other independent paths.

Independent path

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

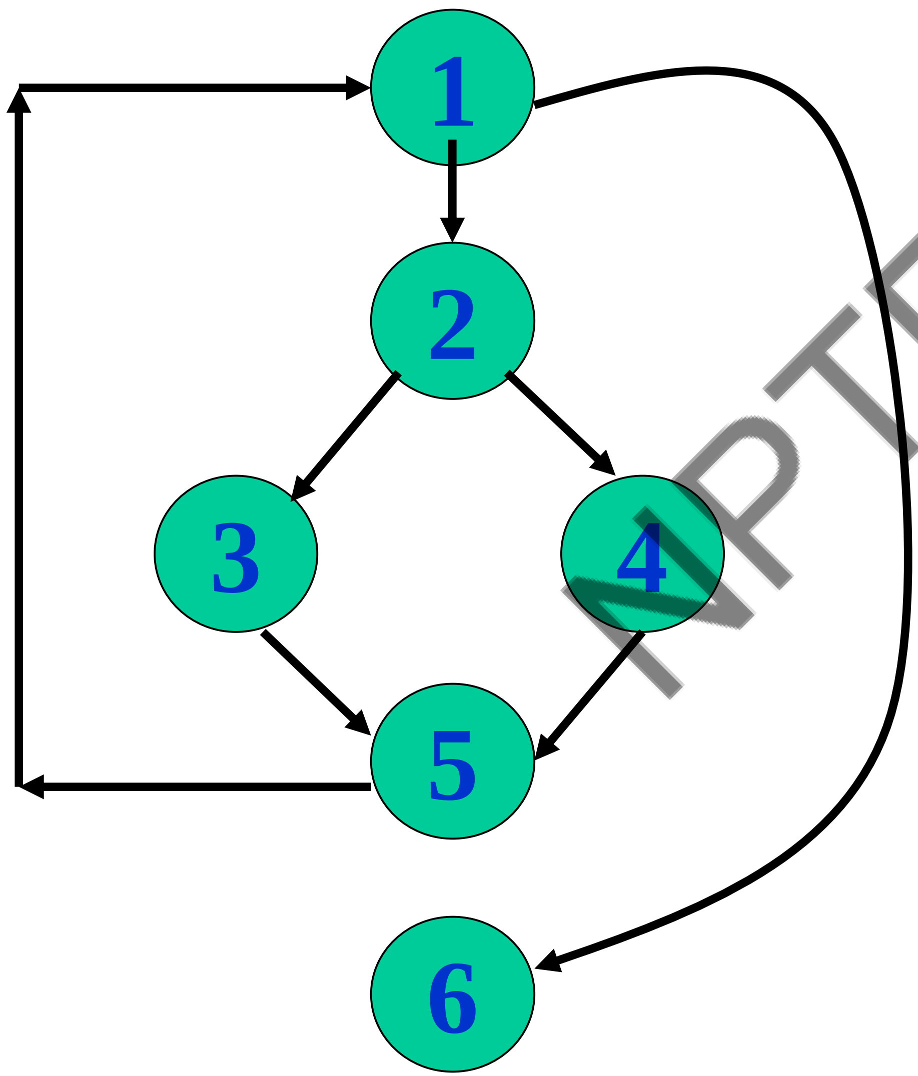
McCabe's Cyclomatic Metric

- An upper bound:
  - For the number of linearly independent paths of a program
- Provides a practical way of determining:
  - The maximum number of test cases required for basis path testing.

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

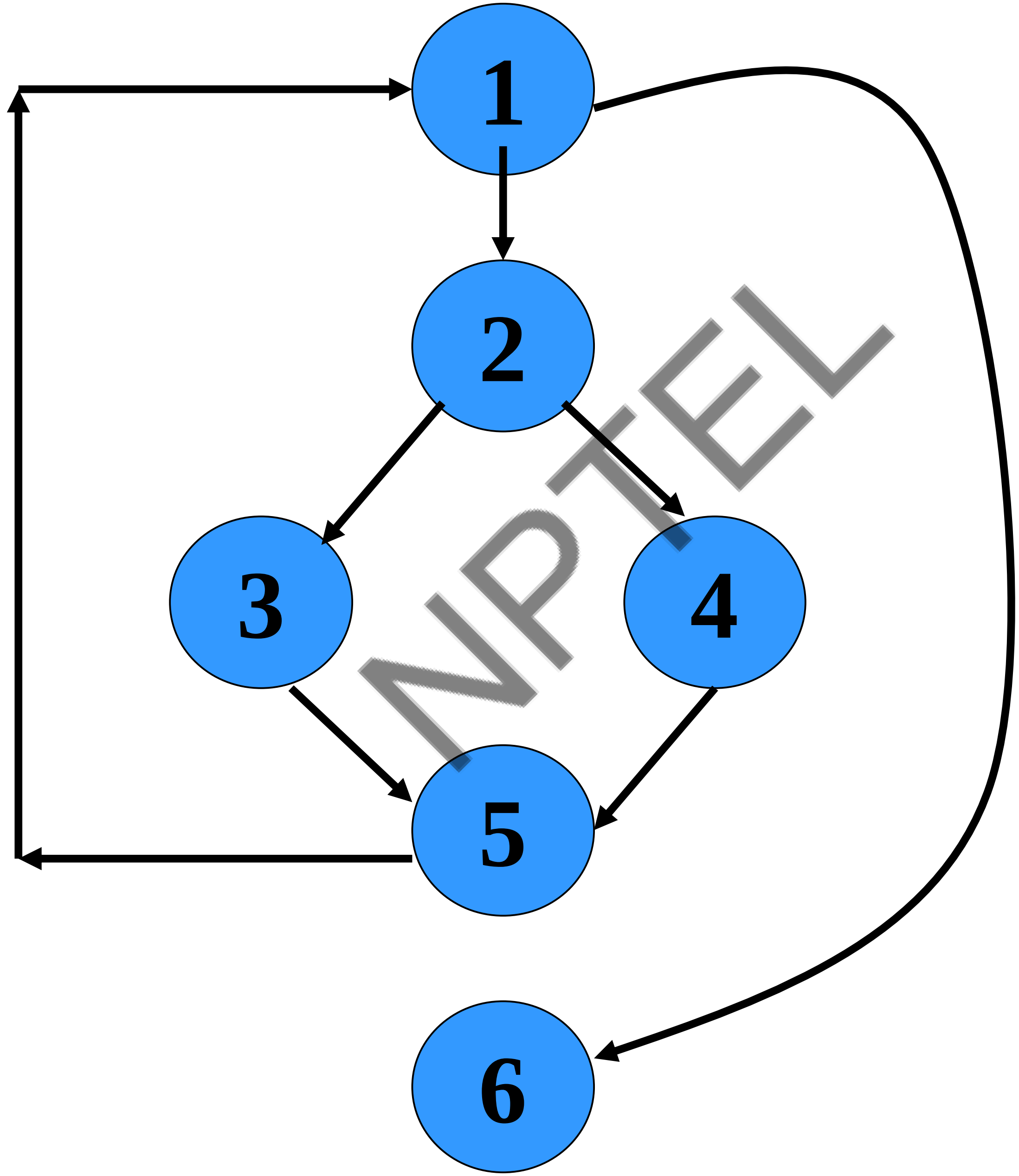


Cyclomatic complexity = 7-6+2 = 3.

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
  - Any region enclosed by a nodes and edge sequence.

Example Control Flow Graph



Example

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

Cyclomatic Complexity

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

Cyclomatic Complexity

- The first method of computing V(G) is amenable to automation:
  - You can write a program which determines the number of nodes and edges of a graph
  - Applies the formula to find V(G).

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

- Knowing the number of test cases required:
  - Does not make it any easier to derive the test cases,
  - Only gives an indication of the minimum number of test cases required.

Practical Path Testing

- The tester proposes initial set of test data :
  - Using his experience and judgment.
- A dynamic program analyzer used:
  - Measures which parts of the program have been tested
  - Result used to determine when to stop testing.

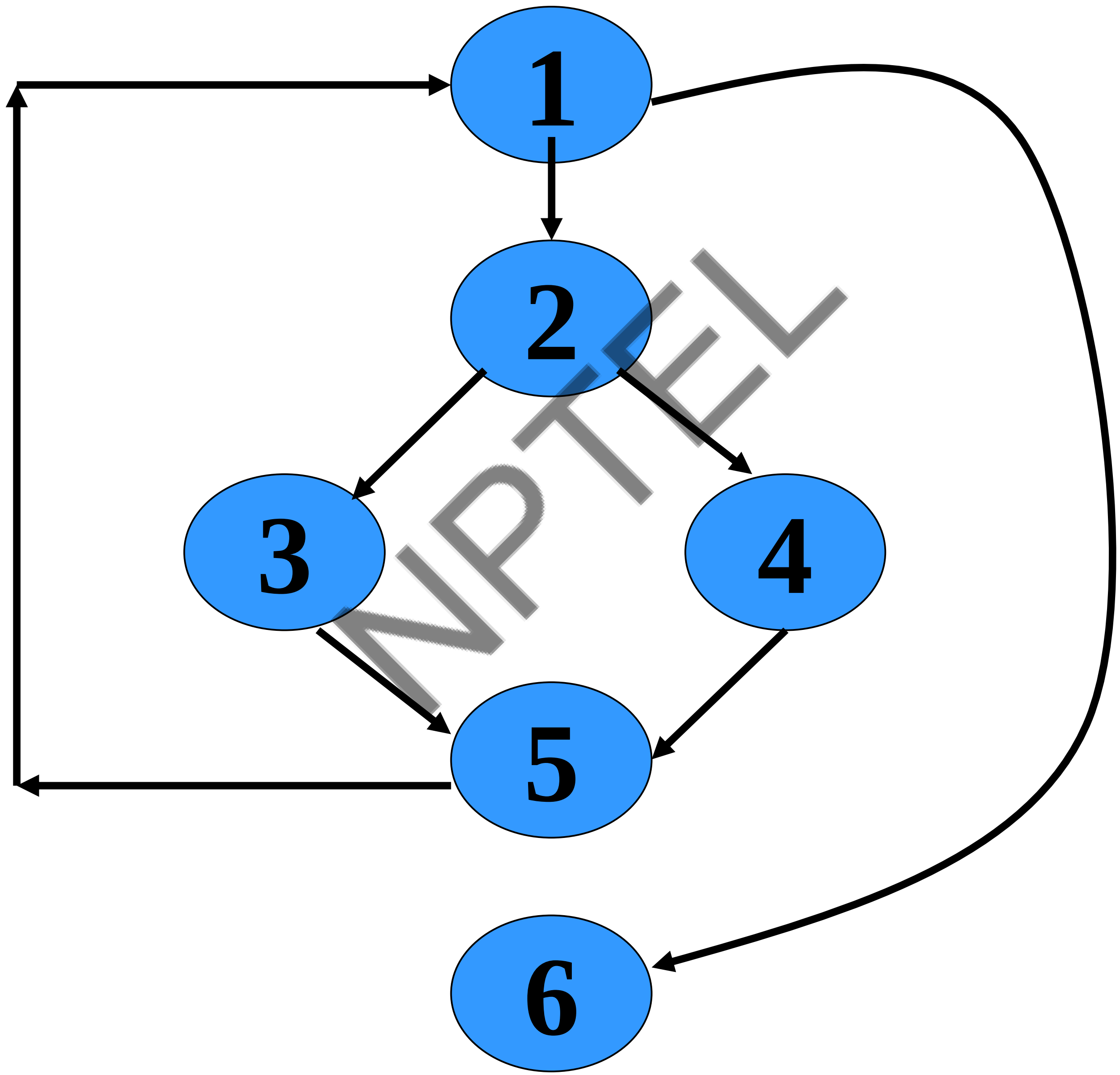
Derivation of Test Cases

- Draw control flow graph.
- Determine V(G).
- Determine the set of linearly independent paths.
- Prepare test cases:
  - Force execution along each path.
  - Not practical for larger programs.

Example

```
int f1(int x,int y){
while (x != y){
    if (x>y) then
        x=x-y;
    else y=y-x;
 }
return x;          }
```

Example Control Flow Diagram



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=1, y=2)
  - 1,2,4,5,1,6 test case(x=2, y=1)

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.

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.

Dataflow and Mutation Testing

White Box Testing: Quiz

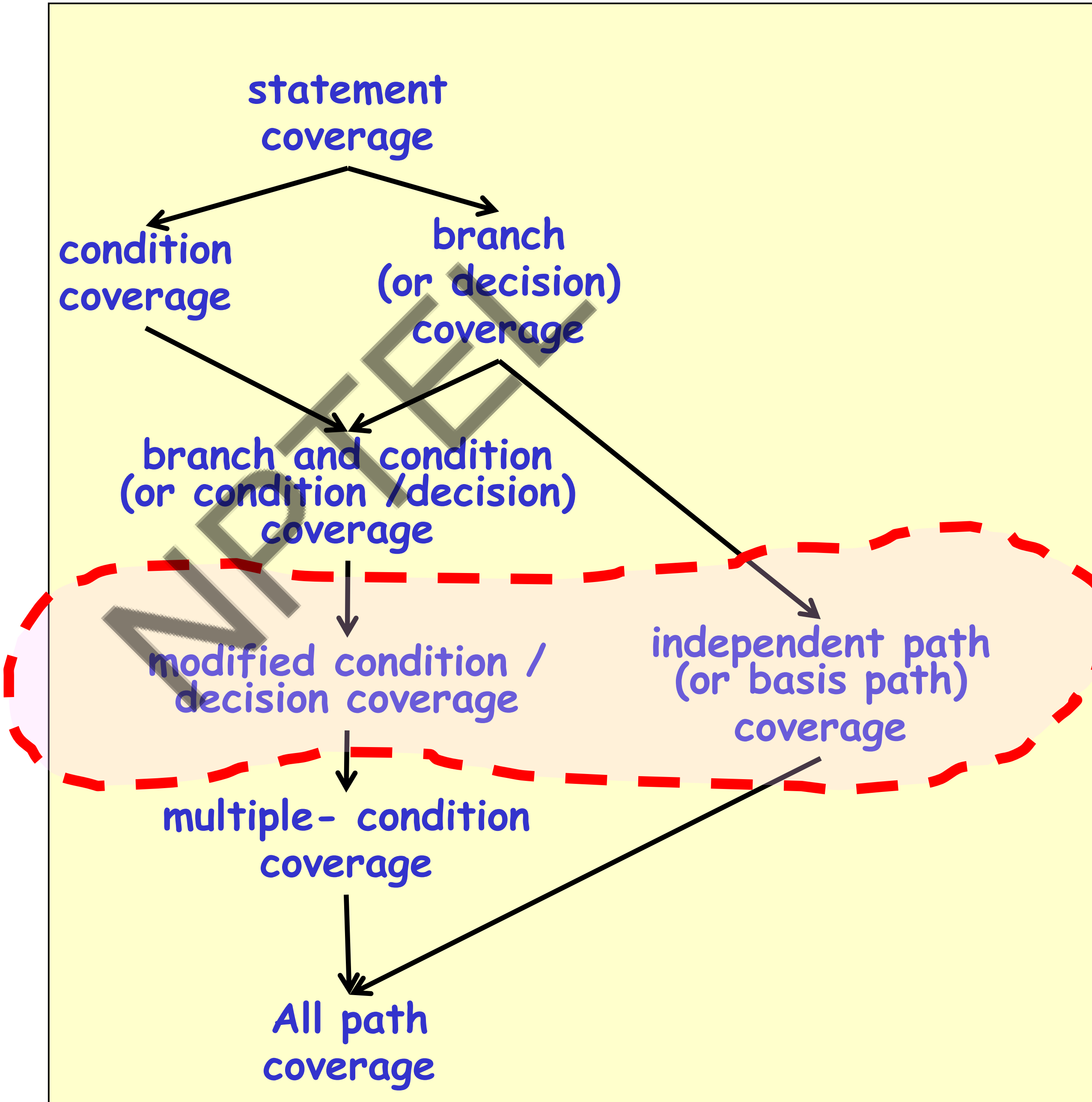
1. What do you mean by coverage-based testing?
2. What are the different types of coverage based testing?
3. How is a specific coverage-based testing carried out?
4. What do you understand by fault-based testing?
5. Give an example of fault-based testing?

White-Box Testing : Recap

weakest

Practically important coverage techniques

strongest



Data flow Testing

Data Flow-Based Testing

- Selects test paths of a program:
  - According to the locations of
    - Definitions and uses of different variables in a program.

```
X(){
 int a=5; /* Defines variable a */
       ....
 While(c>5) {
     if (d<50)
           b=a*a; /*Uses variable a */
           a=a-1; /* Defines variable a */
...
     }
 print(a); } /*Uses variable a */
```

Data Flow-Based Testing

- For a statement numbered S,
  - DEF(S) = {X/statement S contains a definition of X}
  - USES(S)= {X/statement S contains a use of X}
  - Example: **1: a=b;** DEF(1)={a}, USES(1)={b}.
  - Example: **2: a=a+b;** DEF(1)={a}, USES(1)={a,b}.

Data Flow-Based Testing

- A variable X is said to be live at statement S1, if
  - X is defined at a statement S:
  - There exists a path from S to S1 not containing any definition of X.

DU Chain Example

```
X(){
  int a=5; /* Defines variable a */
  While(c>5) {
    if (d<50)
        b=a*a; /*Uses variable a */
        a=a-1; /* Defines variable a */
  }
 print(a); } /*Uses variable a */
```

Definition-use chain (DU chain)

- [X,S,S1],
  - S and S1 are statement numbers,
  - X in DEF(S)
  - X in USES(S1), and
  - the definition of X in the statement S is live at statement S1.

Data Flow-Based Testing

- One simple data flow testing strategy:
  - **Every DU chain in a program be covered at least once.**
- Data flow testing strategies:
  - Useful for selecting test paths of a program containing nested if and loop statements.

Data Flow-Based Testing

```
X(){
 B1;      /* Defines variable a */
 While(C1) {
    if (C2)
          if(C4) B4; /*Uses variable a */
       else B5;
          else  if (C3) B2;
       else B3;     }
 B6 }
```

Data Flow-Based Testing

- [a,1,5]: a DU chain.
- Assume:
  - DEF(X) = {B1, B2, B3, B4, B5}
  - USES(X) = {B2, B3, B4, B5, B6}
  - There are 25 DU chains.
- However only 5 paths are needed to cover these chains.

Mutation Testing

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.**

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.**

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

- Example primitive changes to a program:
  - Deleting a statement
  - Altering an arithmetic operator,
  - Changing the value of a constant,
  - Changing a data type, etc.

Traditional Mutation Operators

- 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

  eg. * and +, / and -
- Replacement of a variable (ensuring same scope/type)

Underlying Hypotheses

- Mutation testing is based on the following two hypotheses:
  - **The Competent Programmer Hypothesis**
  - **The Coupling Effect**

Both of these were proposed by DeMillo *et al.*,1978

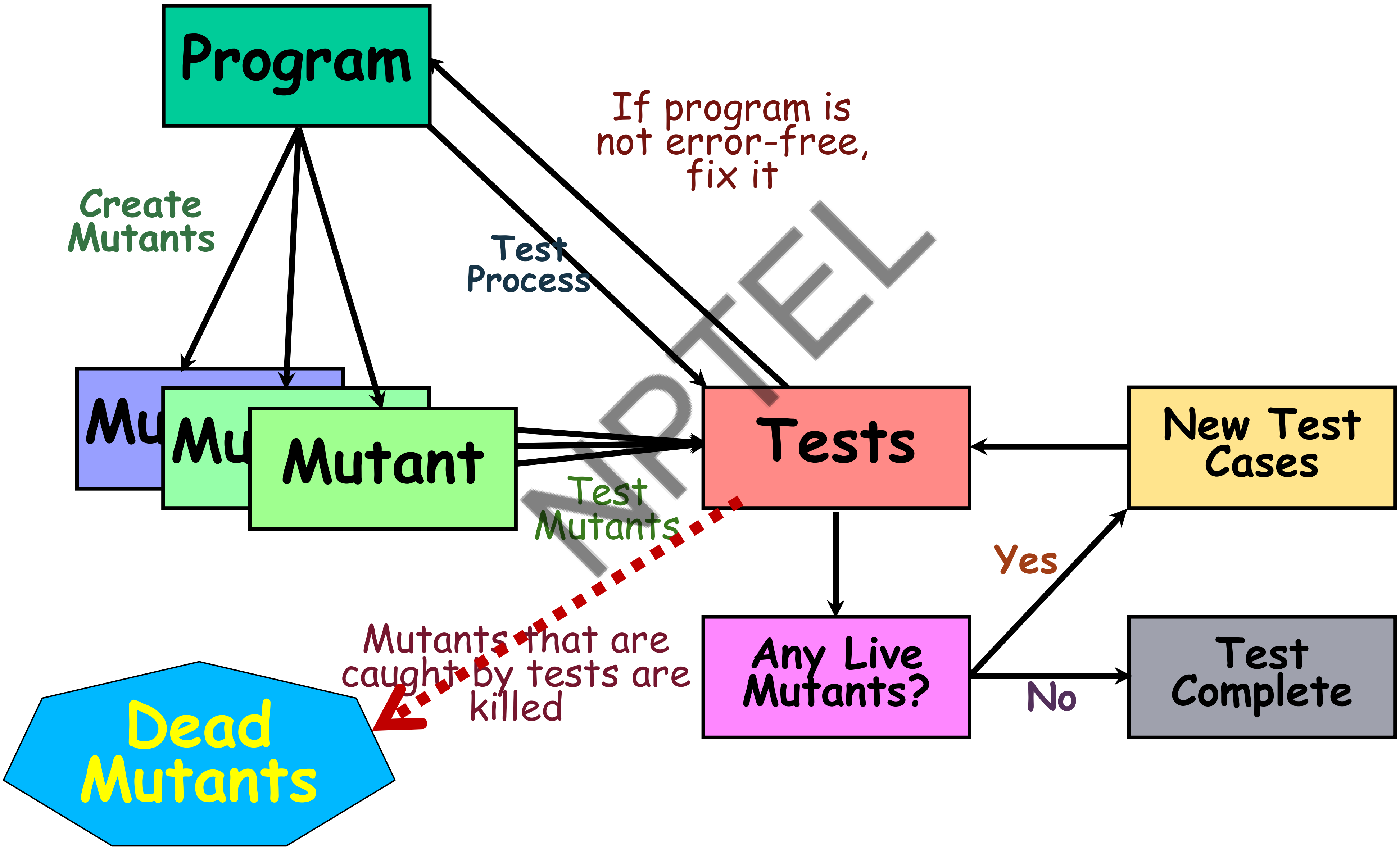
The Competent Programmer Hypothesis

- Programmers create programs that are close to being correct:
  - Differ from the correct program by some simple errors.

The Coupling Effect

- **Complex errors are caused due to several simple errors.**
- It therefore suffices to check for the presence of the simple errors

The Mutation 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'

```
while...
    ...
    i++
    if (i==5)
        break;
```

```
while...
    ...
    i++
    if (i>=5)
        break;
```

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

Quiz 1

- Identify one advantage and one disadvantage of the mutation test technique.

Quiz 1: Solution

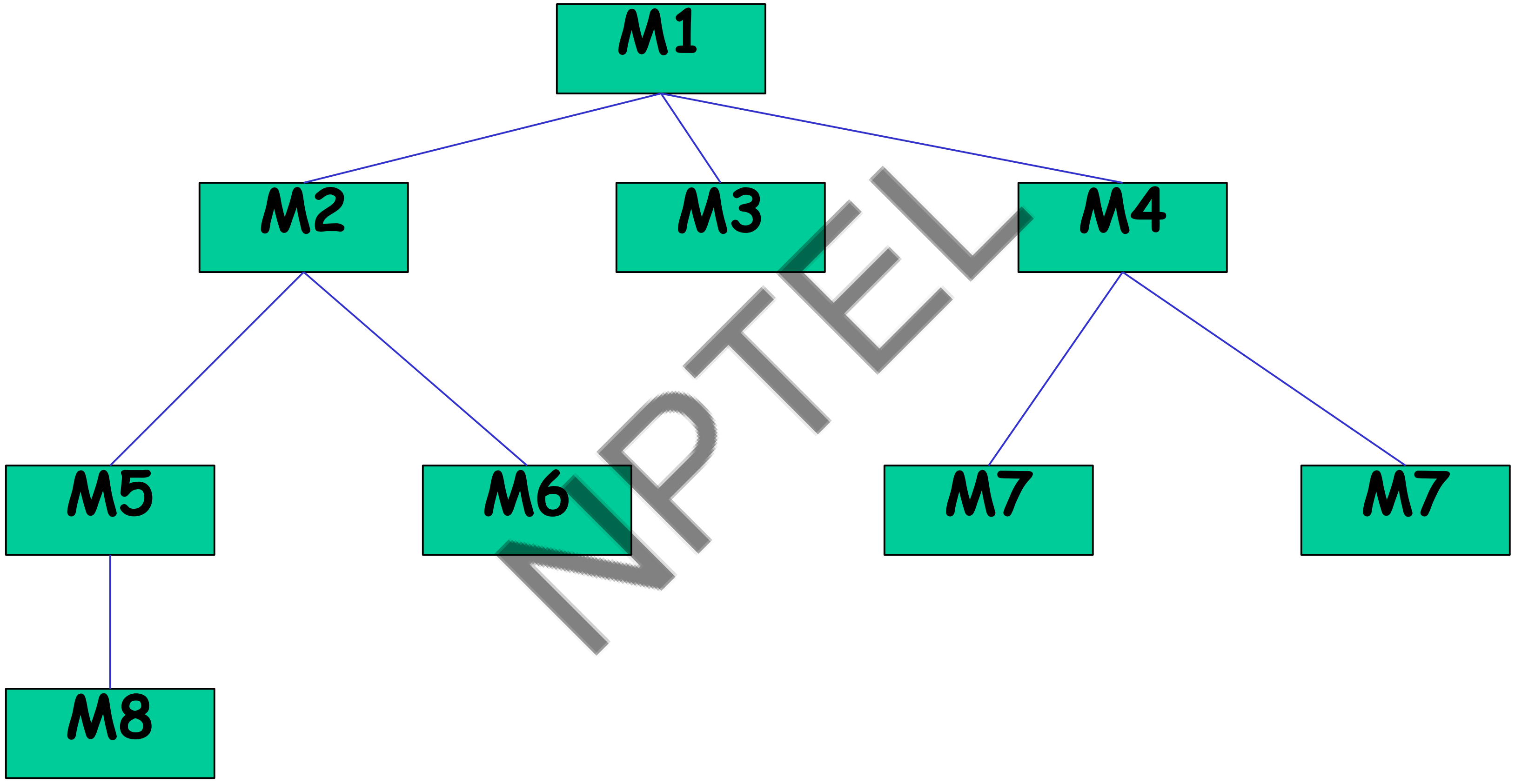
- Identify two advantages and two disadvantages of the mutation test technique.
- **Adv**:
  - Can be automated
  - Helps effectively strengthen black box and coverage-based test suite
- **Disadv:**
  - Equivalent mutants

Integration Testing

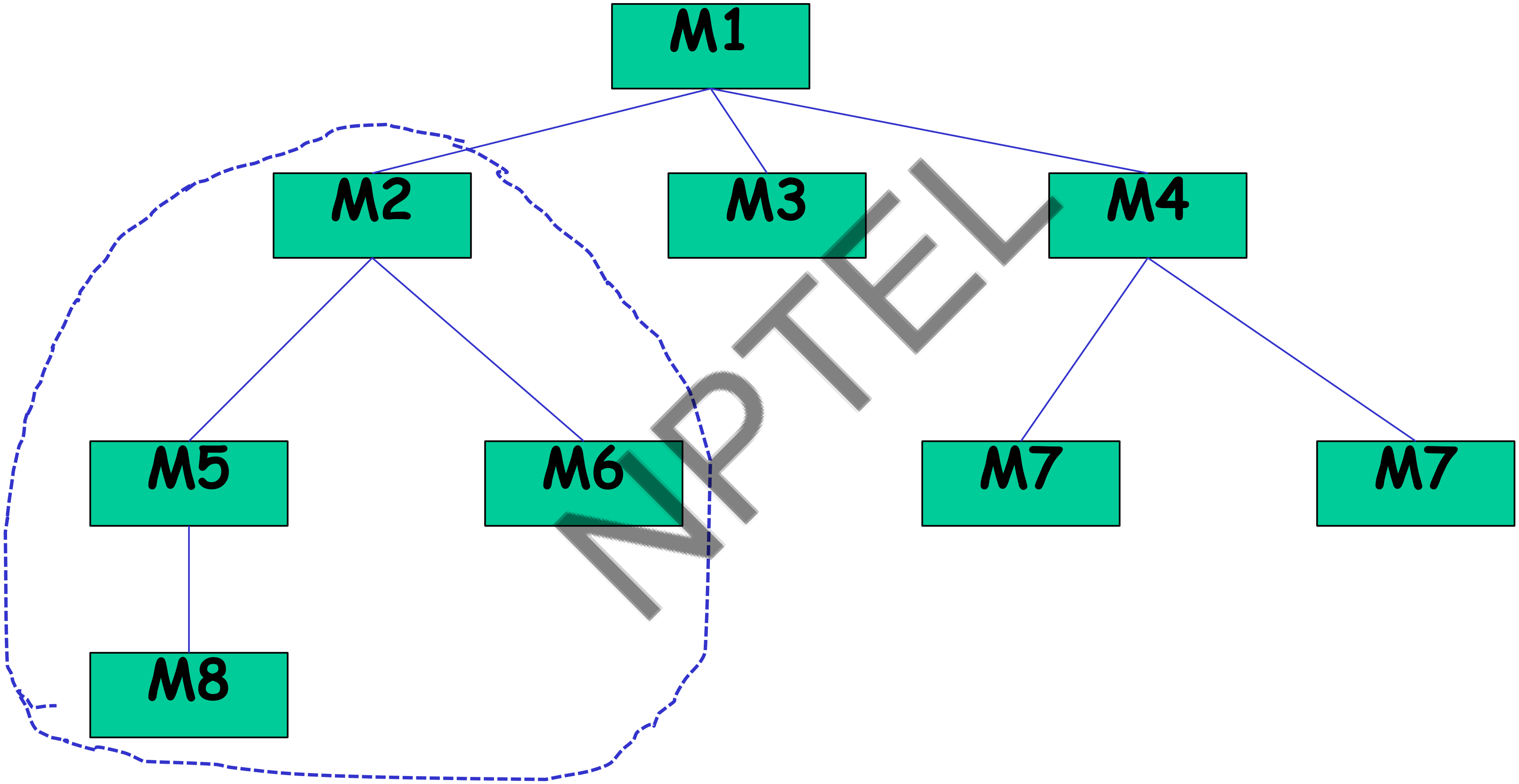
Integration Testing Approaches

- Develop the integration plan by examining the structure chart :
  - big bang approach
  - top-down approach
  - bottom-up approach
  - mixed approach

Example Structured Design



Example Structured Design



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.
- A disadvantage of bottom-up testing:
  - When the system is made up of a large number of small subsystems.
  - This extreme case corresponds to the big bang approach.

Bottom-up testing