White Box Testing

Objectives

- To focus on white box testing
- To distinguish between different types of white box testing
- To place special emphasis on
  - Code reviews / inspections
  - Methods of code coverage and relationship to quality
  - Code complexity

The slide is Basically explaining that for each phase in the development, There's a testing kind that should happen during it

ex: code → unit testing

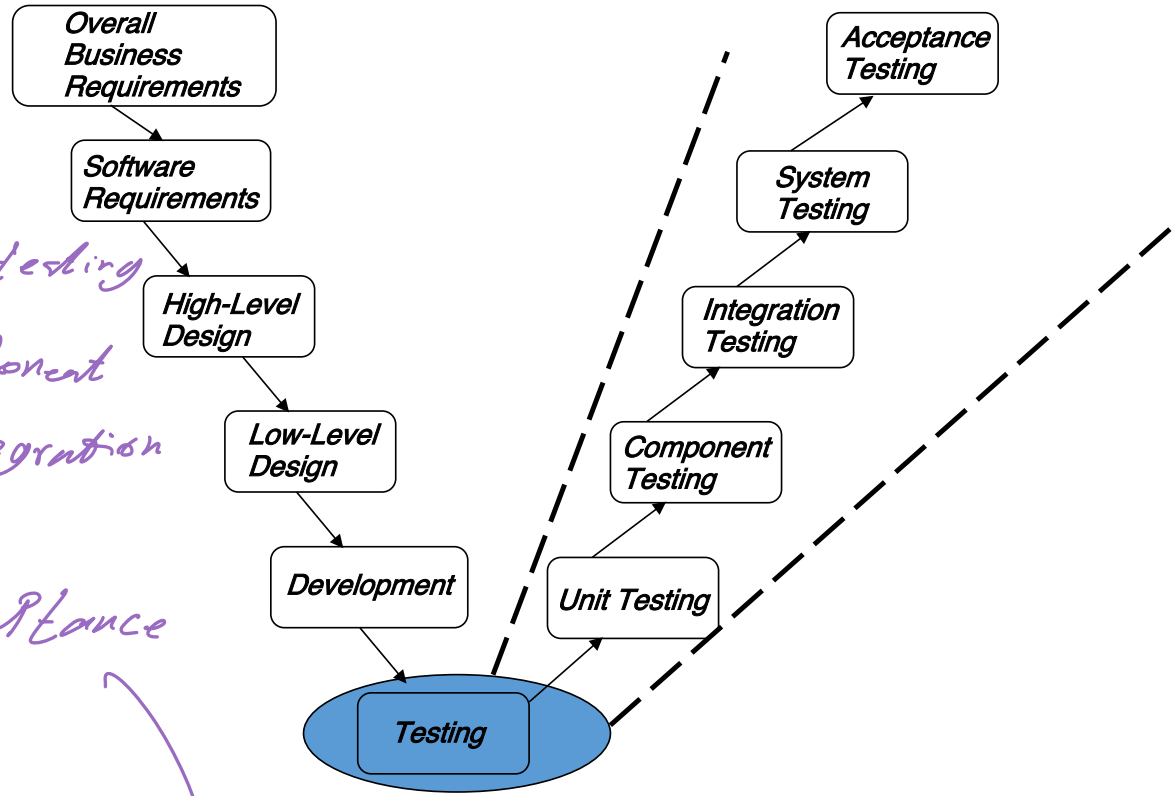
low-level design → component

high level design → integration

SW reqs → system

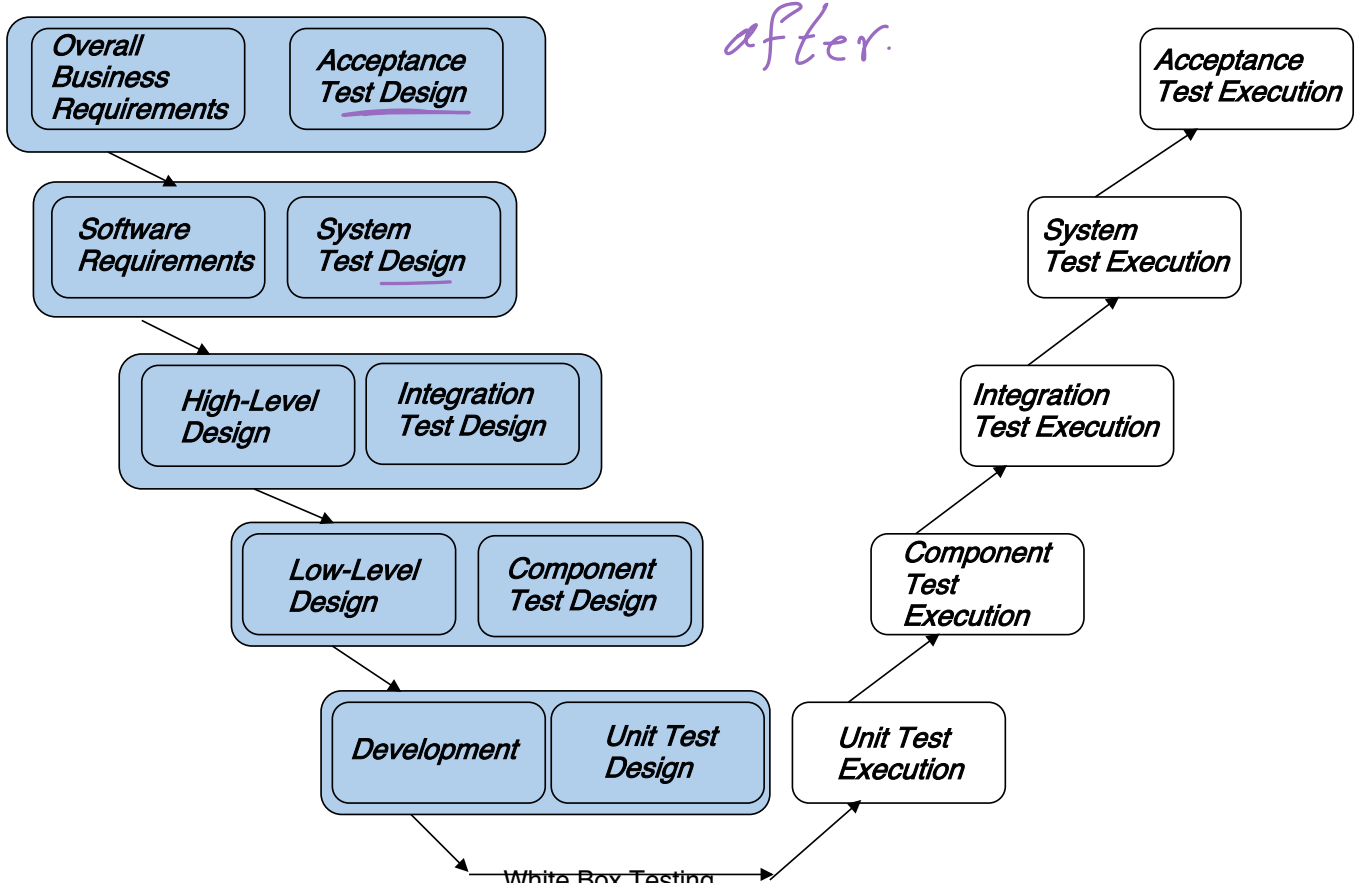
Business reqs → acceptance

"Bending the Waterfall"



V Model

design the appropriate test for the phase during, execute it after.



Principles of the V Model

Should i memorize these?

- Do an early test design.
- Involve the right people in the design of the right type of tests.
- Make the tests reflect closely the steps on the left-hand side of the V.
- Unit testing
  - White box testing (focus of this chapter)
  - Black box testing (focus of the next chapter)

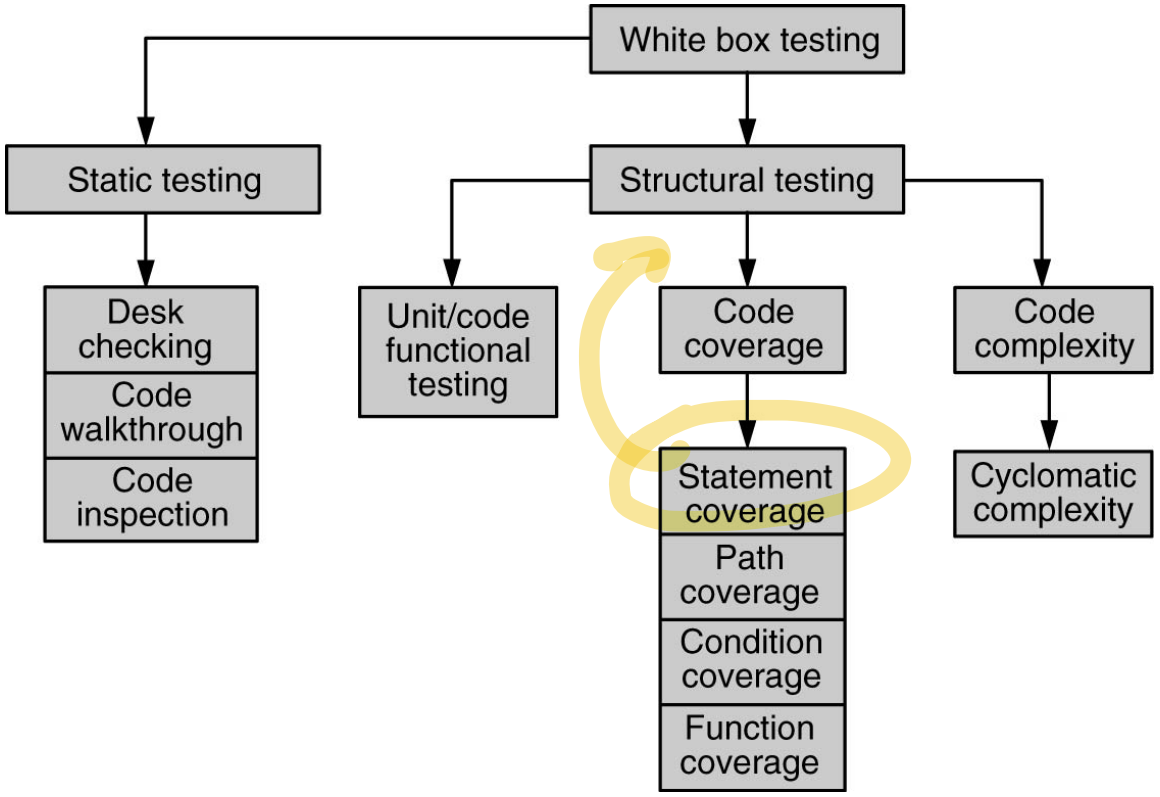
White Box Testing

- Requires access to code
- Looks at the program code and performs testing by mapping program code to functionality

Why White Box Testing?

- The program code truly represents what the program actually does and not just what it is intended to do!
- It minimizes delay between defect injection and defect detection (i.e., does not postpone detection to be done by someone else).
- Can catch “obvious” programming errors that do not necessarily map to common user scenarios (e.g., divide by zero).

Types of White Box Testing



Static Testing

- Involves only the source code and not the executables or binaries
- Does not involve executing the program on a machine but rather humans going through it or the usage of specialized tools
- Some of the things tested by static testing:
  - Whether the code works according to the functional requirement
  - Whether the code has been written in accordance with the design developed earlier in the project life cycle
  - Whether the code follows all applicable standards
  - Whether the code for any functionality has been missed out
  - Whether the code handles errors properly

Static Testing by Humans

- Humans can find errors that computers can’t.
- Multiple humans can provide multiple perspectives.
- A human evaluation can compare the code against the specifications more thoroughly.
- It can detect multiple defects at one go.
- It minimizes the delay in identification of the problems.
- It reduces downstream, inline pressure.

Static Testing Types

- Different types
  - Desk checking of the code
  - Code walkthrough
  - Code review
  - Code inspection
- “QA vs QC” argument
- Increasing the involvement of people
- More variety of perspectives
- Increasing formalism
- Increasing the likelihood of identifying more complex defects

Desk Checking

- Author informally checks the code against the specifications and corrects defects found.
- No structured method or formalism is required to ensure completeness.
- No log or checklist is maintained.
- It relies only on the thoroughness of the author.
- It may suffice for “obvious” programming errors but may not be effective for incomplete / misunderstood requirements.

Advantages and Disadvantages of Desk Checking

- Advantages
  - The programmer knows the code and programming language well and hence is best suited to read the program
  - Less scheduling and logistics overheads
  - Reduces delay in defect detection and correction
- Disadvantages
  - Person dependent, not scalable or reproducible
  - Tunnel visioning of developers
  - Developers prefer writing new code and don’t like testing

Code Walkthrough

- Group oriented (as against desk checking)
- Brings in multiple perspectives
- Less formal than inspection

Formal Inspection / Fagan Inspection

- Group-oriented activity
- Highly formal and structured
- Has specific roles
- Requires thorough preparation

Roles in a formal inspection

- Author
  - Author of the work product
  - Makes available the required material to the reviewers
  - Fixes defects that are reported
- Moderator
  - Controls the meeting(s)
- Inspectors (reviewers)
  - Prepare by reading the required documents
  - Take part in the meeting(s) and report defects
- Scribe
  - Takes down notes during the meeting
  - Assigned in advance by turns
  - Can participate to review to the extent possible
  - Documents the minutes and circulates them to participants

Process in a Fagan Inspection

- Typical documents circulated:
  - Program code
  - Design / program specifications
  - SRS (if needed)
  - Any applicable standards (e.g., coding standards)
  - Any necessary checklists (e.g., code review checklist)

Meetings in a Fagan Inspection

- **Preliminary meeting (optional)**
  - Author explains his / her perspective
  - Makes available the necessary documents
  - Highlights concern areas, if any, for which review comments are sought
- **Defect Logging Meeting**
  - All come prepared!
  - Moderator goes through the code sequentially
  - Each reviewer comes up with comments
  - Comments / defects categorized as “defect” / “observation,” “major” / “minor,” “systemic” / “mis-execution”
  - Scribe documents all the findings and circulates them
- **Follow-up meeting (optional)**
  - Author fixes defects
  - If required, a follow-up meeting is called to verify completeness of fixes

Advantages and Disadvantages of Fagan Inspection

Why do we Fagan inspection?

- Advantages
  - Thorough, when prepared well
  - Brings in multiple perspectives
  - Has been found to be very effective
- Disadvantages
  - Logistically difficult
  - Time consuming
  - May not be possible to exhaustively go through the entire code

Combinations of various methods

- Prioritizing various programs
  - High priority / important codes subject to formal inspection, medium ones for walkthrough and low ones for desk checking
- Deciding between 100% coverage and partial coverage

Static Analysis Tools

Help identify / are used:

- Whether there are unreachable codes (usage of goto statements sometimes creates this situation; there could be other reasons too)
- Variables declared but not used
- Mismatch in definition and assignment of values to variables
- Illegal or error-prone type-casting of variables
- Use of non-portable or architecture-dependent programming constructs
- Memory allocated but not having corresponding statements for freeing up memory
- For calculation of cyclomatic complexity
- As an extension of compilers (lint, compiler flag driven checking…)

Use of a Code Review Checklist

- “Never leave home without it!”
- The code review checklist aids in organizational learning by indicating what to look for.
- It should be kept current as we learn.

Structural Testing

- Done by running the executable on the machine
- Entails running the product against some predefined test cases and comparing the results against the expected results
- Designing test cases and test data to ***exercise*** certain portions of code
- Types of structural testing
  - Unit / code functional testing
  - Code coverage testing
  - Code complexity testing

Unit / Code Functional Testing

- Initial quick checks by developer
- Removes “obvious” errors
- Done before more expensive checks
- Building “debug versions”
- Running through an IDE
- “Debugging” or “testing”? (*Who cares?!*)

Code Coverage Testing

- Exercises different parts of code
- Maps parts of code to required functionality
- Finds out percentage of code covered by “instrumentation”
- Instrumentation rebuilds the product, linking it with libraries
- Instrumented code can monitor what parts of code is covered
- Can help identify critical portions of code, executed often

Types of Code Coverage

- Statement coverage
- Path coverage
- Condition coverage
- Function coverage

Programming Constructs

- Sequential instructions
- Two-way decision statements (if – then – else)
- Multi-way decision statements (switch statements)
- Loops, like while/do, for, repeat/until, etc

Code Basically.

"Programs are always written in sequential formats"

Testing sequential instructions

- Generate test data to make the program enter the sequential block, to make it go through the entire block (Is this valid?)
  - Multiple entry points, in non-structured programming
- Statement coverage as a metric:
  - # of statements exercised / Total # of statements

Testing IF THEN ELSE

- Have data to test the THEN part
- Have data to test the ELSE part
- Relevance of statement coverage?

```
Total = 0; /* set total to zero */
  if (code == "M") {
       stmt1;
       stmt2;
       Stmt3;
       stmt4;
       Stmt5;
       stmt6;
       Stmt7;

  }
  else percent = value/Total*100; /* divide by zero */
```

Testing a Loop

- There should be test cases that:
  - Skip the loop completely
  - Exercise the loop between once and the maximum number of times
  - Try covering the loop, around the “boundary” of n

Exhaustive coverage of all statements in a program is impossible for all practical purposes

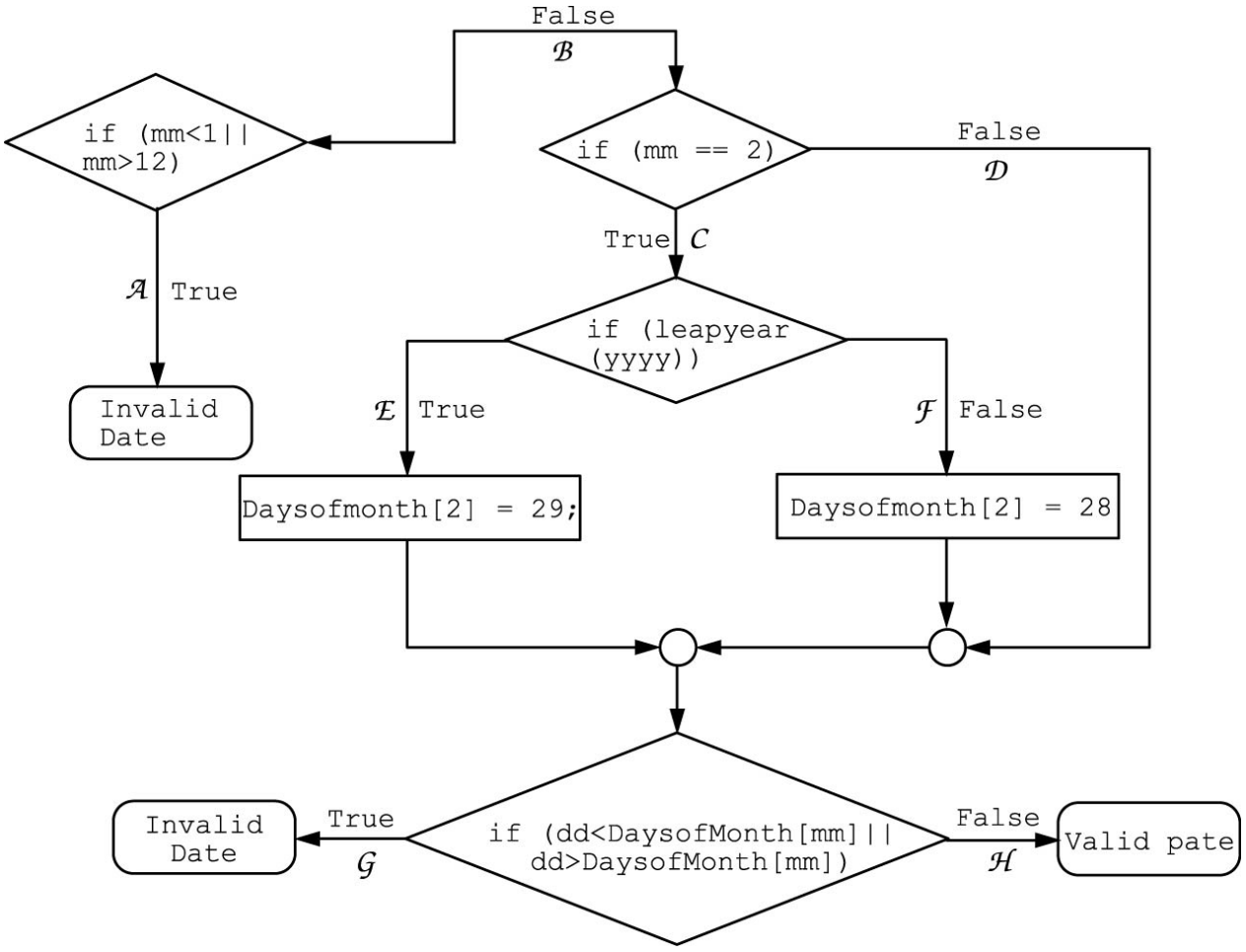
Statement Coverage

- Even if we were to achieve a very high level of statement coverage, it does not mean that the program is defect-free.

```
Total = 0;
If (code ==“M”) {
        stmt1;
        stmt2;
        stmt3;
        stmt4;
        stmt5;
        stmt6;
        stmt7;
}
Else percent = value / Total * 100; // divide by zero
```

Path Coverage

- Statement Coverage may not indicate “true coverage”
- Path Coverage provides better representation
- More detailed example of path coverage in next slide
- Path coverage = “# of paths exercised / total # of paths in the program”


False
B
if (mm<1||
mm>12)
if (mm == 2)
False
D
True C
A True
if (leapyear
(yyyy))
Invalid
Date
E True
F False
Daysofmonth[2] = 29;
Daysofmonth[2] = 28
Invalid
Date
True
G
if (dd<DaysofMonth[mm]||
dd>DaysofMonth[mm])
False
H
Valid pate

Condition Coverage

- Further refinement of path coverage
- Makes sure each constituent condition in a boolean expression is covered
- Protects against compiler optimizations
- Condition coverage = # of conditions covered / number of conditions in the program

Function Coverage

- Finds how many functions are covered by test cases
- Higher level of abstraction; hence possible to achieve 100% coverage
- More “logical” than the other types of coverage
- Can be derived from “RTM”
- Easy to prioritize as they are based on requirements
- Leads more naturally to black box tests
- Can also be a natural predecessor to performance testing

Other Uses of Code Coverage Methods

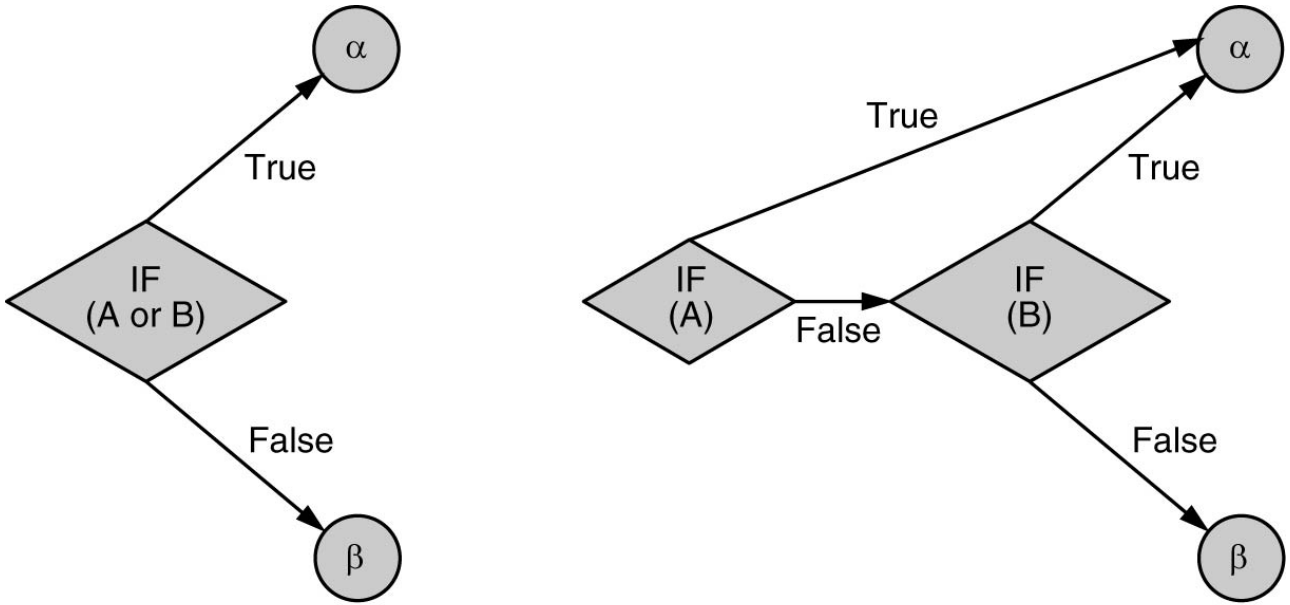
- Performance analysis and optimization
- Resource usage analysis
- Checking of critical sections
- Identifying memory leaks
- Dynamically generated code (e.g., dynamic generation of URLs and security checking)

Code Complexity Testing

- Which of the paths are independent? (If paths are not independent, we may be able to minimize the number of tests.)
- Is there an upper bound on the number of tests to be executed to ensure that all the statements have been executed at least once?
- *Cyclomatic complexity* provides answers to some of these questions.
- It provides an indication of how many “independent paths” are there in a program

Steps in Determining Cyclomatic Complexity

- Start with a conventional flow chart.
- Identify the “decision points.”
- Convert “complex predicates” to “simple predicates” (see next slide).

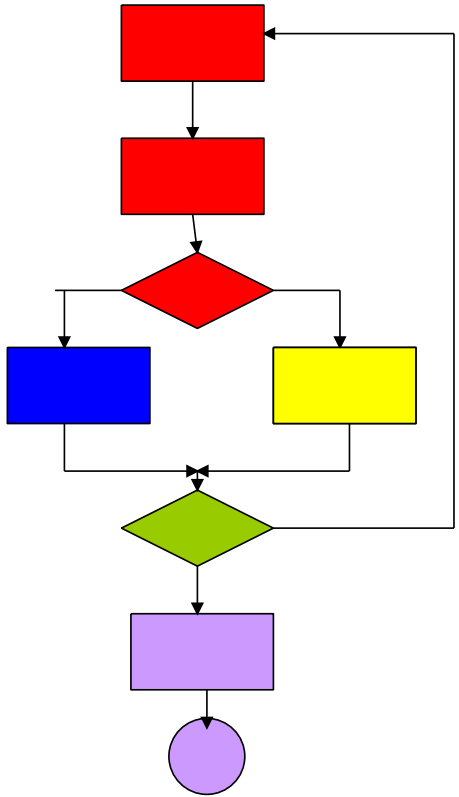


(a) A predicate with a Boolean OR (b) An equivalent set of simple predicates

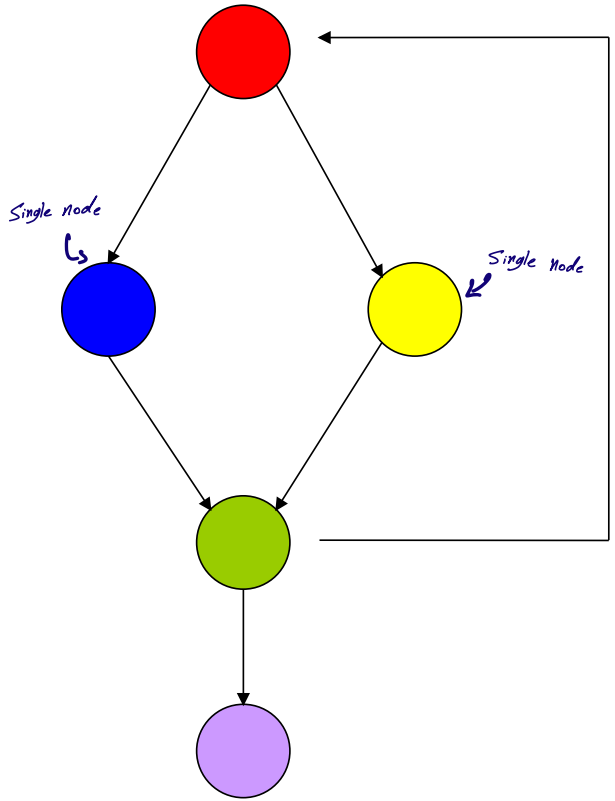
Steps in Determining Cyclomatic Complexity

- Start with a conventional flow chart.
- Identify the "decision points." = Predicate.
- Convert "complex predicates" to "simple predicates".
- If there are loops, convert the loop conditions to simple predicates.
- Combine all the sequential statements into a single node in the flow graph.
- When a set of sequential statements are followed by a simple predicate, combine all the sequential statements and the predicate check into one node and have two edges emanating out of this node. Nodes with two emanating edges are called predicate nodes.
- Make sure that all edges terminate in some node, adding a node to represent all the statements at the end of the program.

Conventional Flow Chart



Flow Diagram for Calculating Complexity

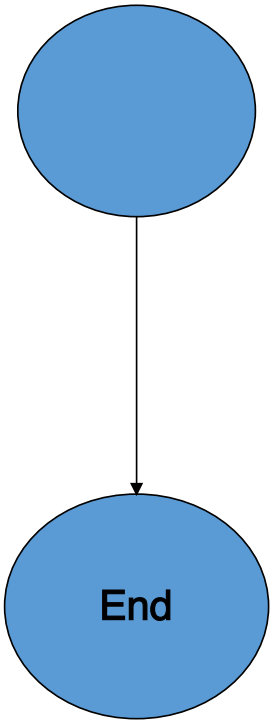


Formal and Intuitive Interpretation of Cyclomatic Complexity

- Formal definitions
  - CC = # of predicate nodes + 1
  - CC = E – N + 2 (E = edges, N = nodes)

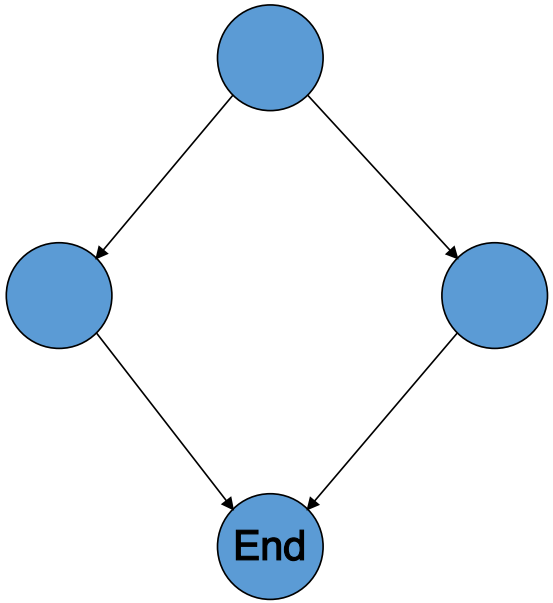
- Intuitive definitions
  - When a condition is introduced, adds a new path
  - See the next two slides

Case (i): A hypothetical program with no decision node



- *# of independent paths = 1*
- *# of nodes, N = 2;*
- *# of edges, E = 1;*
- *Cyclomatic complexity = E – N + 2 = 1*
- *# of predicate nodes, P = 0;*
- *Cyclomatic complexity = P + 1 = 1*

Case (ii): Adding one decision node



- *# of independent paths = 2*
- *# of nodes, N = 4;*
- *# of edges, E = 4;*
- *Cyclomatic complexity = E – N + 2 = 2*
- *# of predicate nodes, P = 1;*
- *Cyclomatic complexity = P + 1 = 2*

Independent Paths and Basis Sets

- An independent path is a path in the flow graph that has not been traversed before in other paths.
- A set of independent paths that cover all the edges is called a basis set.
- Find the basis set and write the test cases to execute all the paths in the basis set.

Meaning and Interpretation of Cyclomatic Complexity

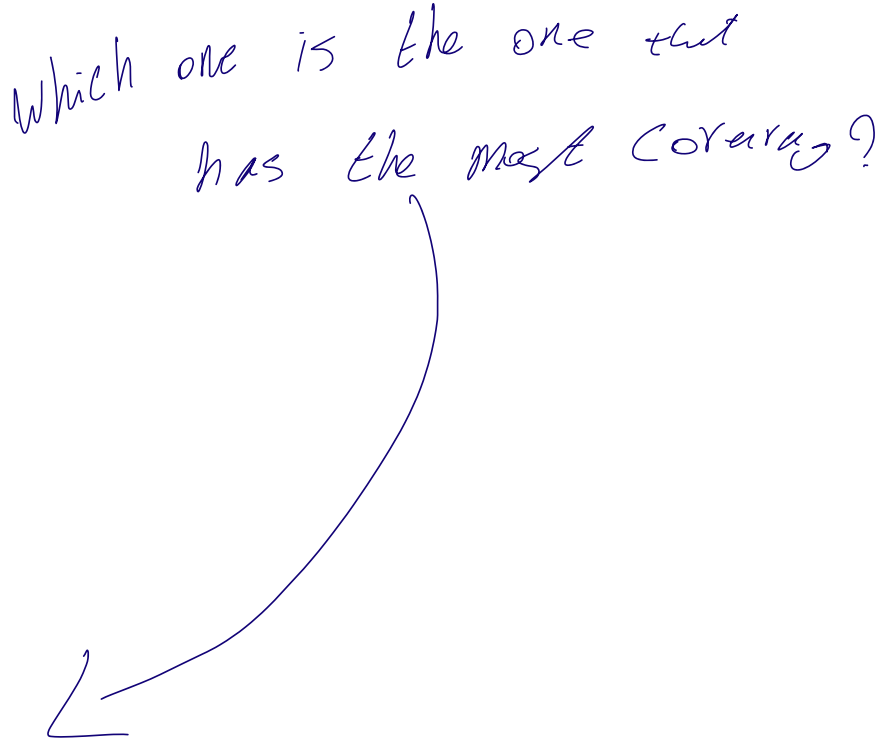
Complexity	What it means
1-10	Well-written code, testability is high, cost / effort to maintain is low
10-20	Moderately complex, testability is medium, cost / effort to maintain is medium
20-40	Very complex, testability is low, cost / effort to maintain is high
>40	Not testable, any amount of money / effort to maintain may not be enough

Challenges in White Box Testing

- Developers “not finding time for testing”
- They develop blind spots for their own defects
- Fully covered code may not correspond to realistic scenarios

White-Box Testing Techniques

- Statement coverage
- Decision coverage
- Condition coverage
- Decision-condition coverage
- Multiple-condition coverage



White-Box Testing

- It is concerned with the degree to which test cases exercise or cover the logic of source code of the program

- The ultimate white-box testing is to execute every path in the program—an unrealistic goal with a program with loops.
- If we back away from executing every path, a reasonable goal is to execute every statement at least once.

Statement Coverage

```
void foo(int a, int b, int x)
{
if (a>1 && b==0) {
   x=x/a;
}

if (a==2 || x>1) {
   x=x+1;
}
}
```

How many paths in this program?

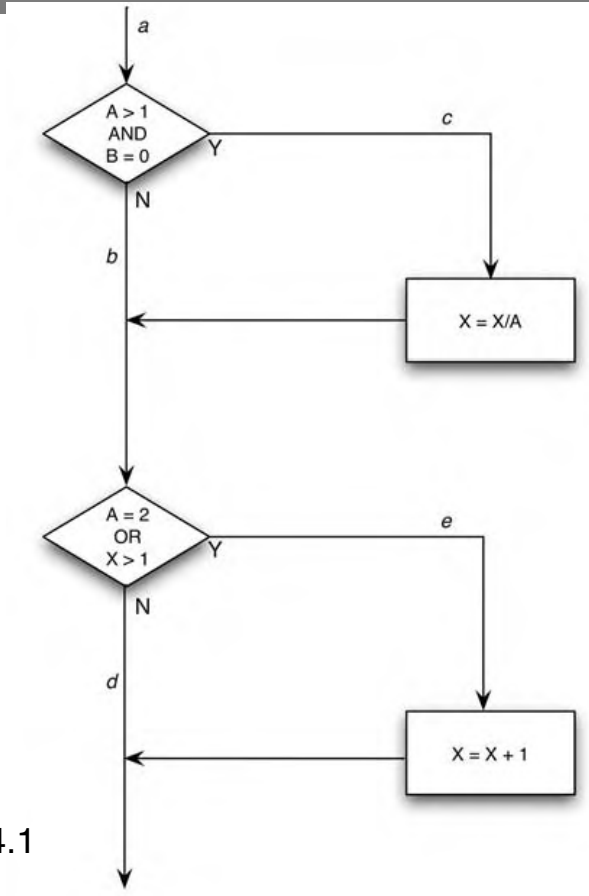
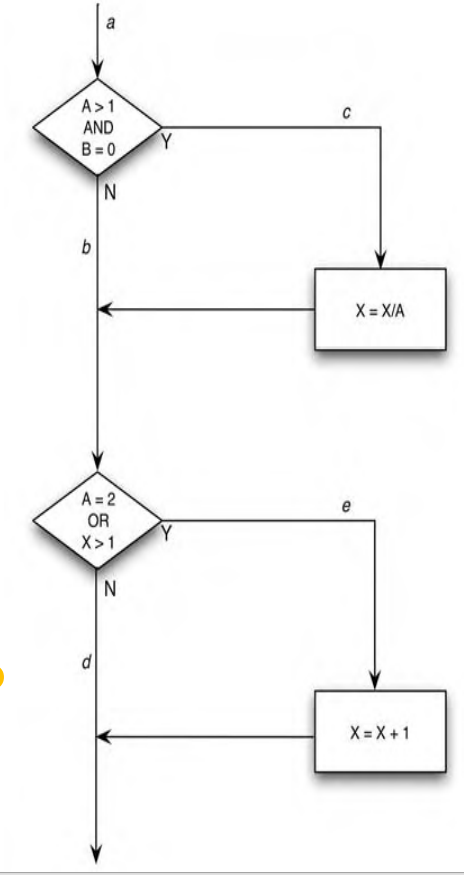


Figure 4.1

All possible Paths

Path	Input	Output
ace	A=2, B=0, X=4	X=3
abd	A=0, B=0, X=0	X=0
acd	A=3, B=0, X=3	X=1
abe	A=2, B=1, X=1	X=2



Which paths provide the most coverage?

Statement Coverage

- A set of test cases that traverses every statement at least once.
- Path ace provides statement coverage.
- The test case consists of (A=2, B=0, X=4) for input and the expected output is (X=3).
  - i.e., execute every statement at least once.
- This criterion is poor.
- Some undetected errors.
  1. Maybe, the first condition should be an OR rather than an AND
  2. Maybe, the second condition should be X>0 rather than X>1
  3. The path *abd* (A=1, B=1, X=1) leaves X unchanged (if it is an error)

Decision Coverage

or Branch Coverage

- A stronger logic-coverage criterion than the statement coverage.
- We must write enough test cases to cover each decision with both the true and false outcomes at least once.
- In case of multiple choices in a decision, we must write test cases to cover all choices at least once
  - Switch statement in C and Java
- Some programming languages may have multiple entry points for a module or program (PL/1)

Decision Coverage (Cont'd)

- A more precise definition for the decision-coverage testing is
  - to exercise every possible outcome of all decisions at least once and that each point of entry is invoked at least once.

Decision Coverage vs. Statement coverage

- Decision coverage provides statement coverage in most circumstances
  - every statement must be executed if every branch direction is executed.

Decision Coverage vs. Statement coverage

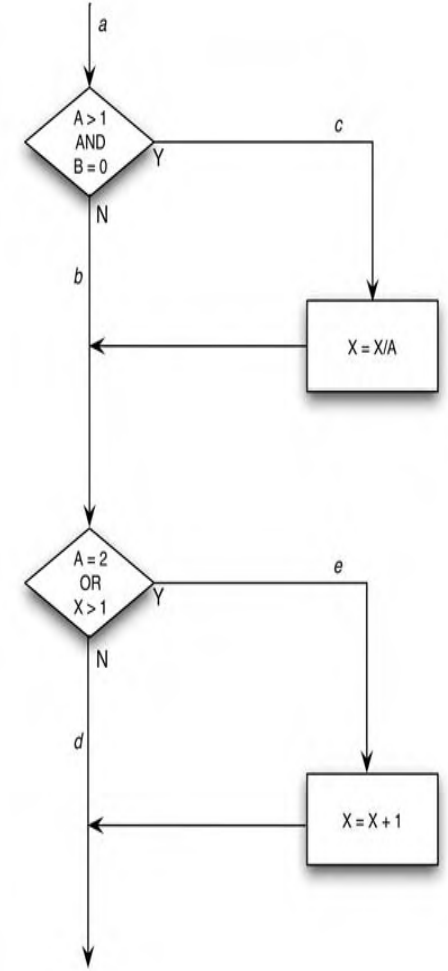
- Exceptions
  - Programs with no decisions.
  - Programs or subroutines with multiple entry points:
    - A given statement might be executed only if the program is entered at a particular entry point.

- So a stronger definition of the decision coverage
  - The decision-coverage testing requires that each outcome of each decision is exercised at least once, that each statement is exercised at least once, and that each entry point is invoked at least once.

Decision Coverage

- There are two sets of test cases that satisfy the criterion:

Path	Input	Output
ace	A=2, B=0, X=4	X=3
abd	A=1, B=1, X=1	X=1
acd	A=3, B=0, X=3	X=1
abe	A=2, B=1, X=1	X=2



Decision Coverage

Problems!

- Only 50% chance of catching the "X must be changed" error (if it is stated and described in the expected outcomes of the test cases)
  - Only the path abd will catch this error and we have only a 50% chance of selecting the set of test cases that contains abd.

- If the second decision were in error (if it should have said X<1 instead of X>1),
  - the mistake would not be detected by the two test cases in Yellow.

Therefore, we need a better coverage criterion.

Condition Coverage

- This testing requires that we write enough test cases so that each condition in each decision takes on all possible values at least once and that each point of entry is invoked at least once

- A stronger logic-coverage test than the decision coverage

Condition Coverage

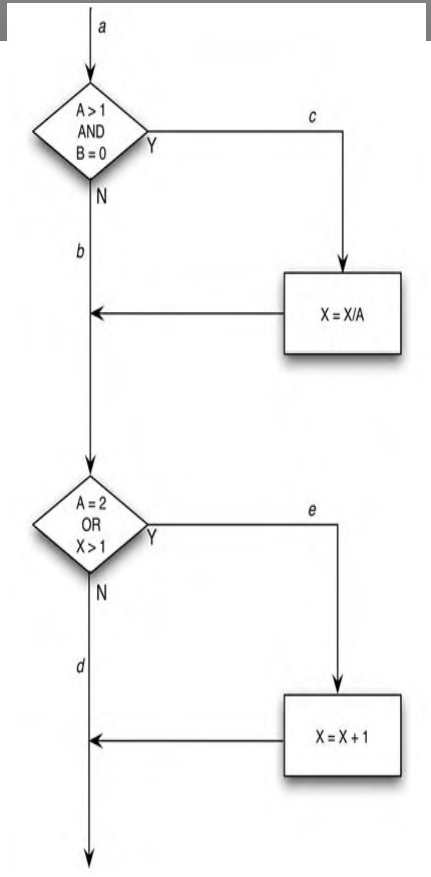
Exercise:- for and while loops

DO K=0 to 50 WHILE (J+K<QUEST)

- Conditions
  - 1: K<=50
  - 2: J+K<QUEST
- Test cases:
  1. K<=50,
  2. K>50,
  3. J+K<QUEST,
  4. J+K>=QUEST.

Condition Coverage (Cont'd)

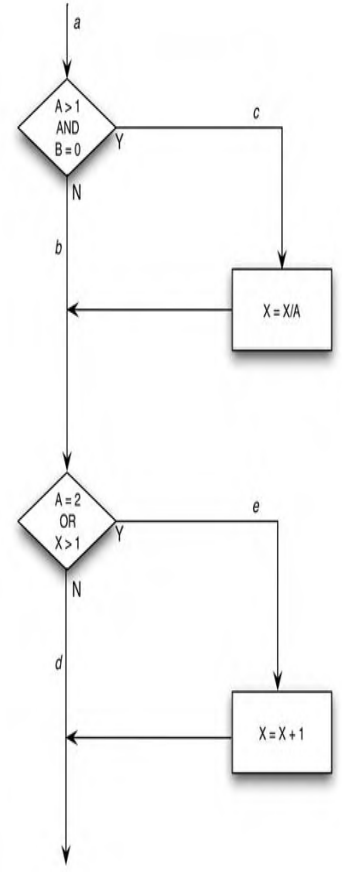
- Four conditions
  - A > 1
  - B = 0 (B == 0 in C and Java)
  - A = 2 (A == 2 in C and Java)
  - X > 1



Condition Coverage-Test Cases

- Condition coverage test cases must cover the following:
  - A > 1, A <= 1
  - B = 0, B <>0 (B != 0 in C and Java)
  - A = 2, A <> 2
  - X > 1, X <= 1
- When we pick a value for a variable, the value may satisfy more than one condition requirements

A	B	X	
2	0	4	(ace)
1	1	1	(abd)

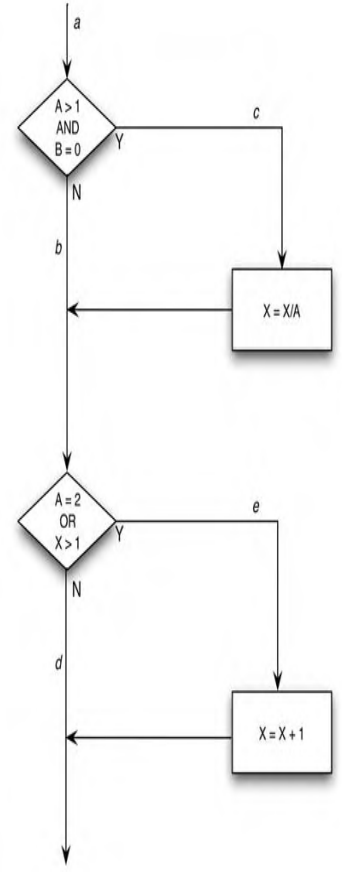


Condition Coverage-All decisions

- Condition coverage test cases must cover the following:
  - A > 1, A <= 1,B = 0, B <>0 (B != 0 in C and Java)
  - A = 2, A <> 2, X > 1, X <= 1
- The following table covers all decisions, but how many paths?

Path	Input	Output
ace	A=2, B=0, X=4	X=3
abd	A=1, B=1, X=1	X=1
acd	A=3, B=0, X=3	X=1
abe	A=2, B=1, X=1	X=2

Again, decision coverage does not cover all conditions as shown in this example.

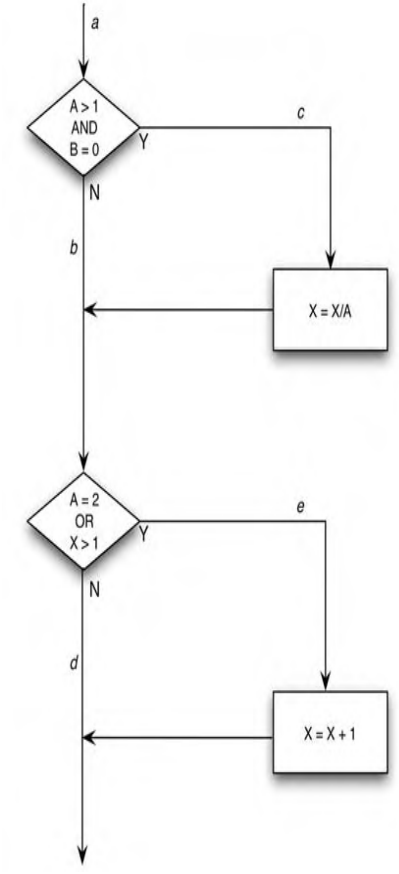


Condition Coverage-All decisions

- Another set of condition-coverage test cases

A	B	X	
2	4	4	(abe)
0	0	1	(abd)

  - These test cases covers all conditions
  - However, The path c and the statement on path c are never executed by this set of test cases.
  - So, condition coverage does not guarantee decision coverage, thus no guarantee for path or statement coverage
  - Also try
    - A=1, B=0, X=3
    - A=2, B=1, X=1



Condition Coverage (Cont'd)

Solution! Decision/Condition Coverage

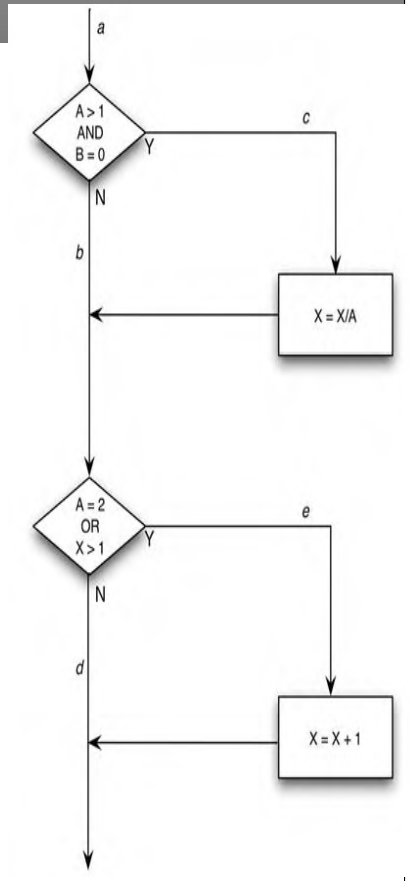
- Select test cases such that
  - Every condition outcome, every decision outcome, and every point of entry are exercised at least once.

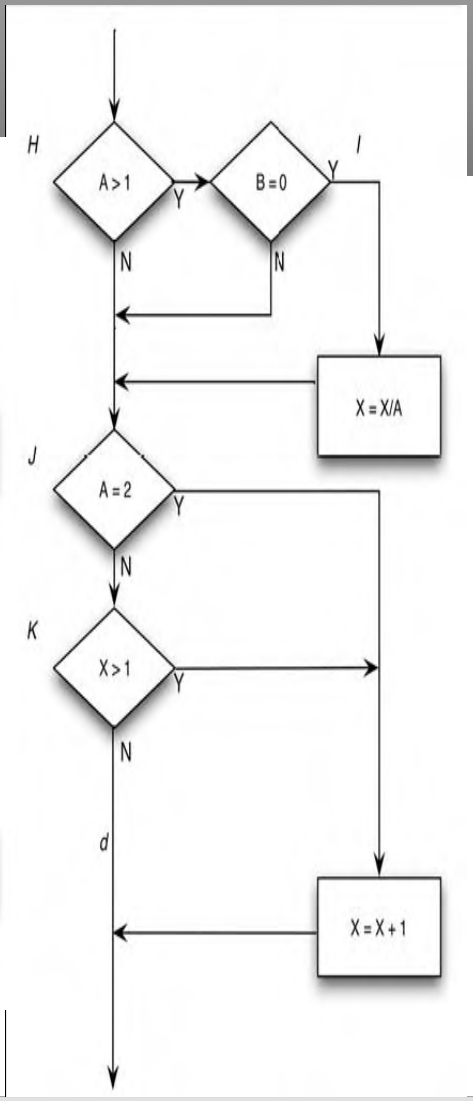
Try the in RED
do they cover all

Path	Input	Output
ace	A=2, B=0, X=4	X=3
abd	A=1, B=1, X=1	X=1
acd	A=3, B=0, X=3	X=1
abe	A=2, B=1, X=1	X=2

The two test cases (ace, abd) that We thought would provide condition coverage, but actually do not.

1. The false outcome of condition I (B==0) is missed
2. The true outcome of condition K (X>1) is missed





Multiple Condition Coverage

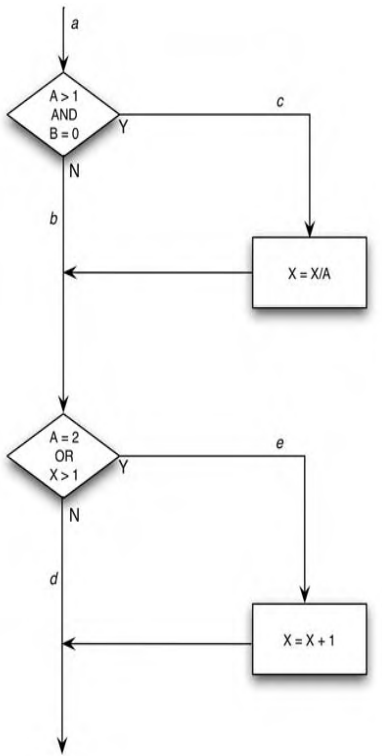
- We write sufficient test cases so that all possible combinations of condition outcomes in each decision and all points of entry are exercised at least once

- The multiple condition coverage satisfies condition-, decision-, and decision/condition-coverage criteria

Multiple Condition Coverage (Cont'd)

- In Figure the following, we need to write test cases that cover the following outcome combinations:

1. A> 1 and B=0
2. A > 1 and B<>0
3. A<=1 and B=0
4. A<=1 and B<>0
5. A=2 and X>1
6. A=2 and X<=1
7. A<>2 and X>1
8. A<>2 and X<=1



Multiple Condition Coverage (Cont'd)

- The following test cases cover all eight possibilities:

  A=2, B=0, X=4 Covers 1, 5
  A=2, B=1, X=1 Covers 2, 6
  A=1, B=0, X=2 Covers 3, 7
  A=1, B=1, X=1 Covers 4, 8

  - Multiple-condition coverage does not guarantee path coverage
    - The above test cases miss path acd
  - The fact that there are four test cases and four paths in the program is a pure coincidence.
  - There is no relationship between the number of paths and the number of test cases

Example

The four situations to be tested are:

1. I<=TABSIZE and NOTFOUND is true.

2. I<=TABSIZE and NOTFOUND is false (finding the entry before hitting the end of the table).

3. I>TABSIZE and NOTFOUND is true (hitting the end of the table without finding the entry).

4. I>TABSIZE and NOTFOUND is false (the entry is the last one in the table).

Example

- You would need eight test cases for the following decision:

```
if(x==y && length(z)==0 && FLAG)
        j=1;
else
        i=1;
```

x==y	length	FLAG	X	Y	Length
F	F	F	1	3	1
F	F	T	1	3	1
F	T	F	1	3	0
F	T	T	1	3	0
T	F	F	1	1	1
T	T	F	1	1	0
T	F	T	1	1	1
T	T	T	1	1	0

Summary

- For programs containing only one condition per decision, a minimum test criterion is a sufficient number of test cases to
  1. evoke all outcomes of each decision at least once and
  2. Invoke each point of entry (such as entry point or ON-unit) at least once, to ensure that all statements are executed at least once.
- For programs containing decisions having multiple conditions, the minimum criterion is a sufficient number of test cases to
  1. evoke all possible combinations of condition outcomes in each decision, and
  2. all points of entry to the program, at least once.
- (The word “possible” is inserted because some combinations may be found to be impossible to create.)