LSU

# Software Systems Development

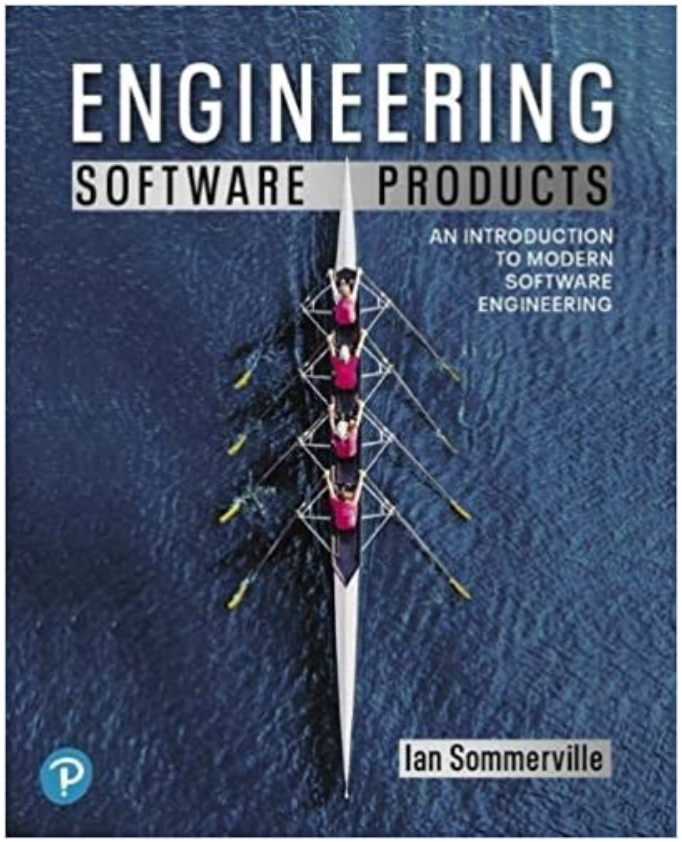
Nash Mahmoud

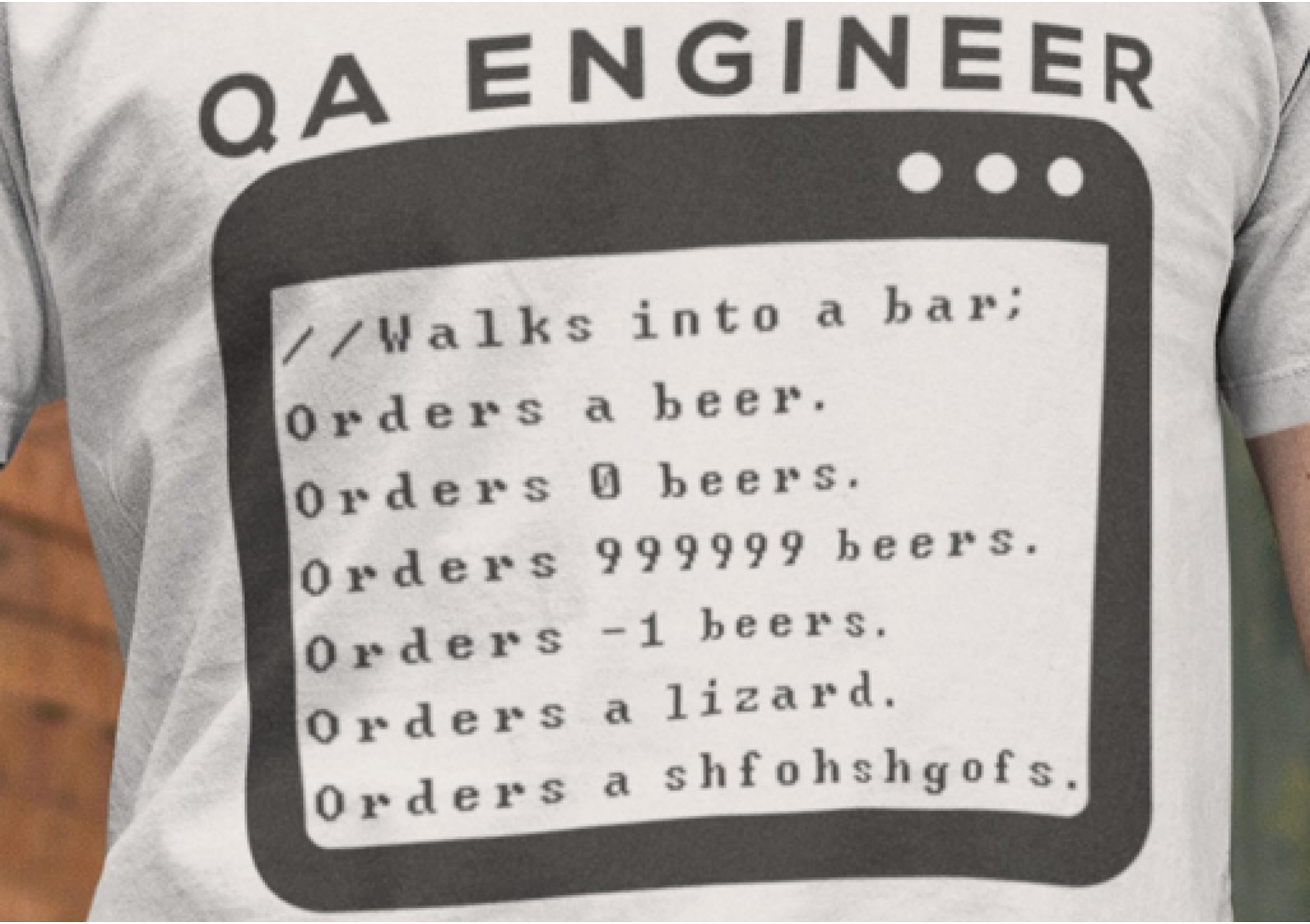
Associate Professor

Louisiana State University

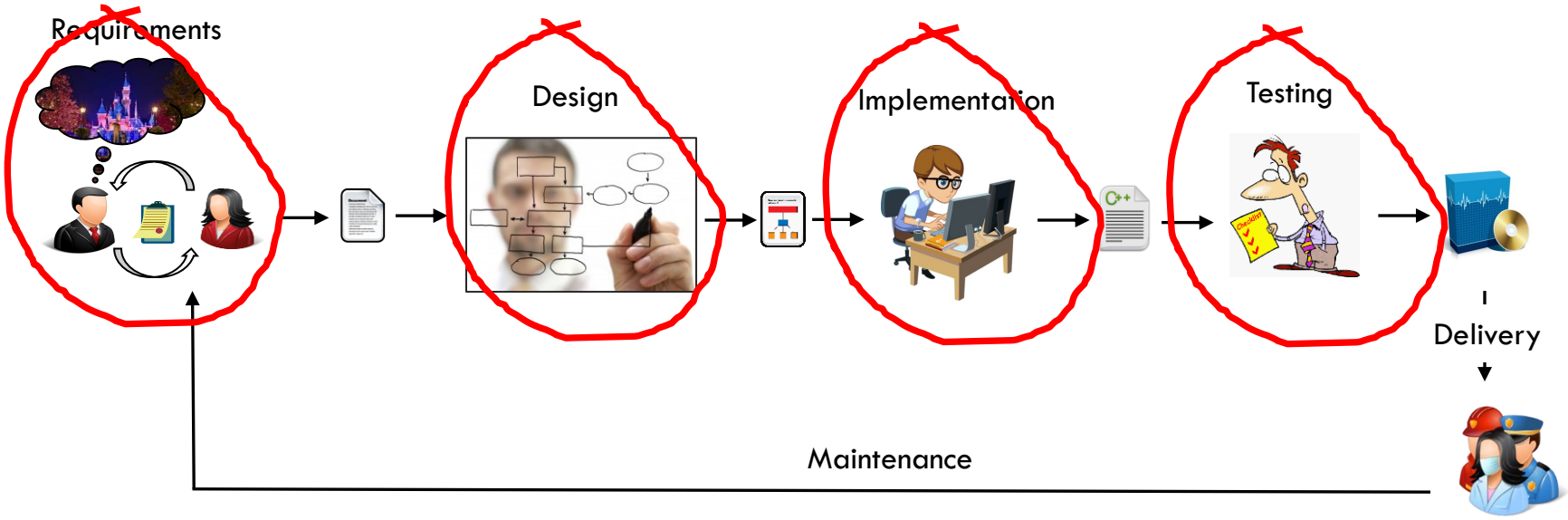
# CHAPTER VII

Testing the System



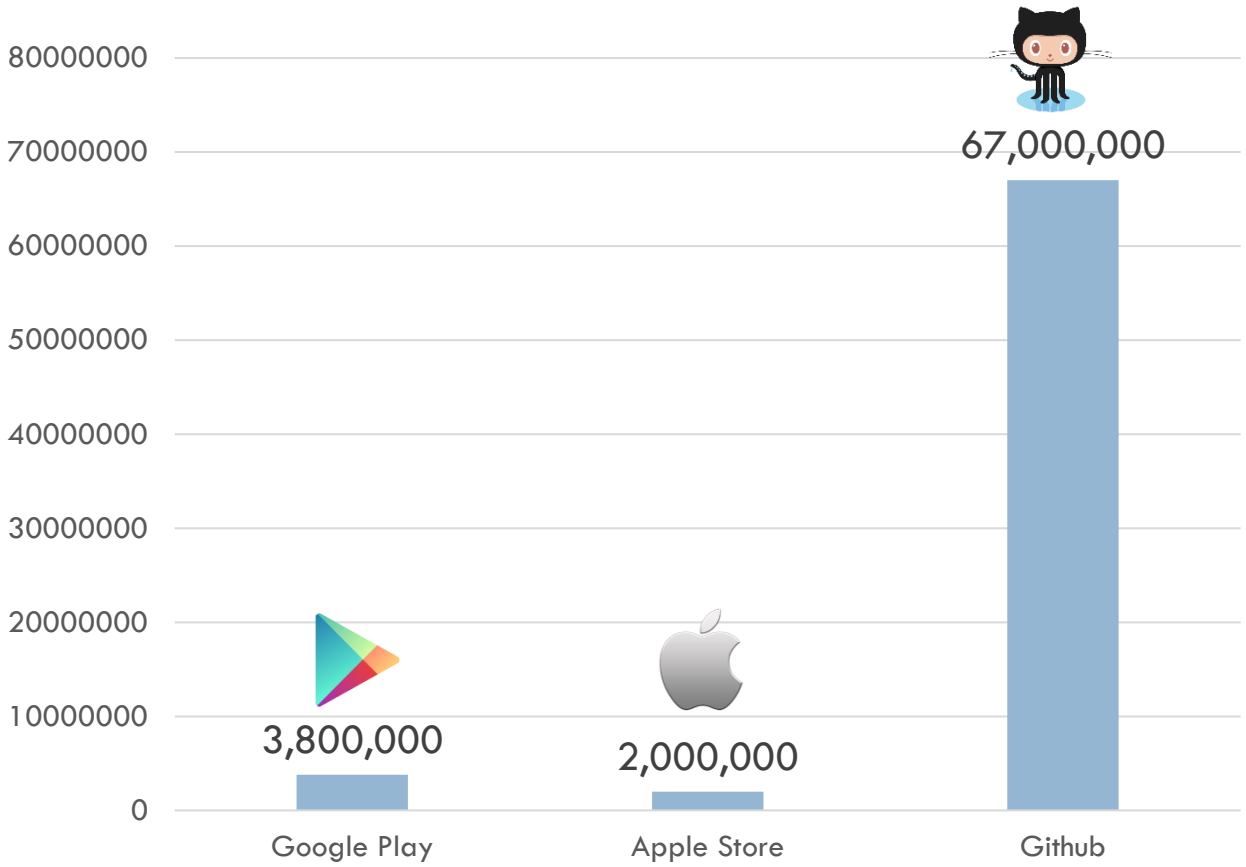

QA ENGINEER
//Walks into a bar;
Orders a beer.
Orders 0 beers.
Orders 999999 beers.
Orders -1 beers.
Orders a lizard.
Orders a shfohshgofs.

# Software life cycle: main activities



# Applications and Bugs





## LO$$E$ FROM SOFTWARE FAILURES (USD)

# 1,715,430,778,504

ONETRILLIONSEVENHUNDREDFIFTEENBILLIONFOURHUNDREDTHIRTYMILLIONSEVENHUNDREDSEVENTY-EIGHTTHOUSANDFIVEHUNDREDFOUR

## COST OF A SOFTWARE BUG

| $100 | $1,500 | $10,000 |
| --- | --- | --- |
| If found in the **Gathering Requirements** stage | If found in **QA Testing** stage | If found in **Production** |

# Bugs are Expensive




## 96% of Users do not report bugs - they just leave

Most users do not report bugs and just leave your product, instead

# Competent Programmer Hypothesis

*Programmers write programs that are almost perfect. Program faults are syntactically small and can be corrected with a few keystrokes.*

# Software Testing

- Types of faults and how to classify them
- The purpose of testing
- Unit testing
- Integration testing
- System testing
- Test automation
- Test Driven Development

Error → Fault → Failure

# Error, Fault, and Failure

- Error: A Human generated mistake
- Fault: An encoding of the error
- Failure: A departure from the system's required behaviour

# Objective of Testing

- Discover faults
- A test is successful only when a fault is discovered

# Types of Faults

- Algorithmic fault: occurs when a component's algorithm or logic does not produce proper output
  - Example: Some input cause a "divide by zero" or "infinite loop" error
  - Syntax error, missing initialization, incorrect branching (IF/ELSE)!

# Algorithmic Faults

U.S Space mission to Venus failed due to a period instead of comma in a Fortran do loop

DO 5  K=1, 3:

Interpreted as assignment:

DO5K = 1.3:

# Computation and Precision Fault

- Computation and precision fault
  - Example: Mistakenly combining integers and floats in one formula

*For six unfortunate patients in 1986 and 1987, the **Therac**-25 did the unthinkable: it exposed them to massive overdoses of **radiation**, killing four and leaving two others with lifelong injuries.*

# Precision fault

## Zillow, facing big losses, quits flipping houses and will lay off a quarter of its staff.

The real estate website had been relying on its algorithm that estimates home values to buy and resell homes. That part of its business lost about $420 million in three months.

# Capacity Faults

- Capacity or boundary faults
  - System's performance not acceptable when certain limits are reached (DOS Attacks)

# Performance Faults

- Performance faults
  - System does not perform at the required speed (response time)

# Types of Faults

- Documentation fault
  - Documentation doesn't match what the program does
  - Example: user manual says the system accepts coins of all denominations, the system only accepts quarters.

# To sum up …

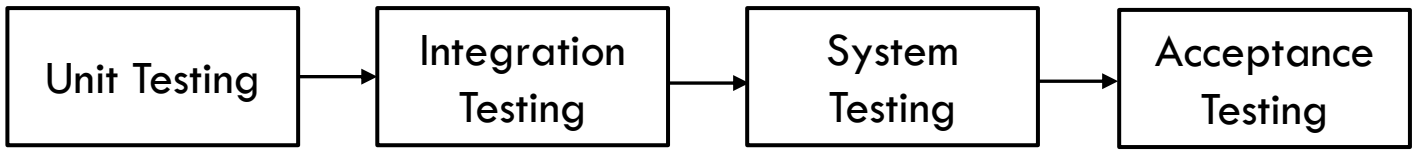
- Types of faults
  - Algorithmic fault
  - Computation and precision fault
  - Capacity or boundary faults
  - Performance faults
  - Documentation fault

# Testing

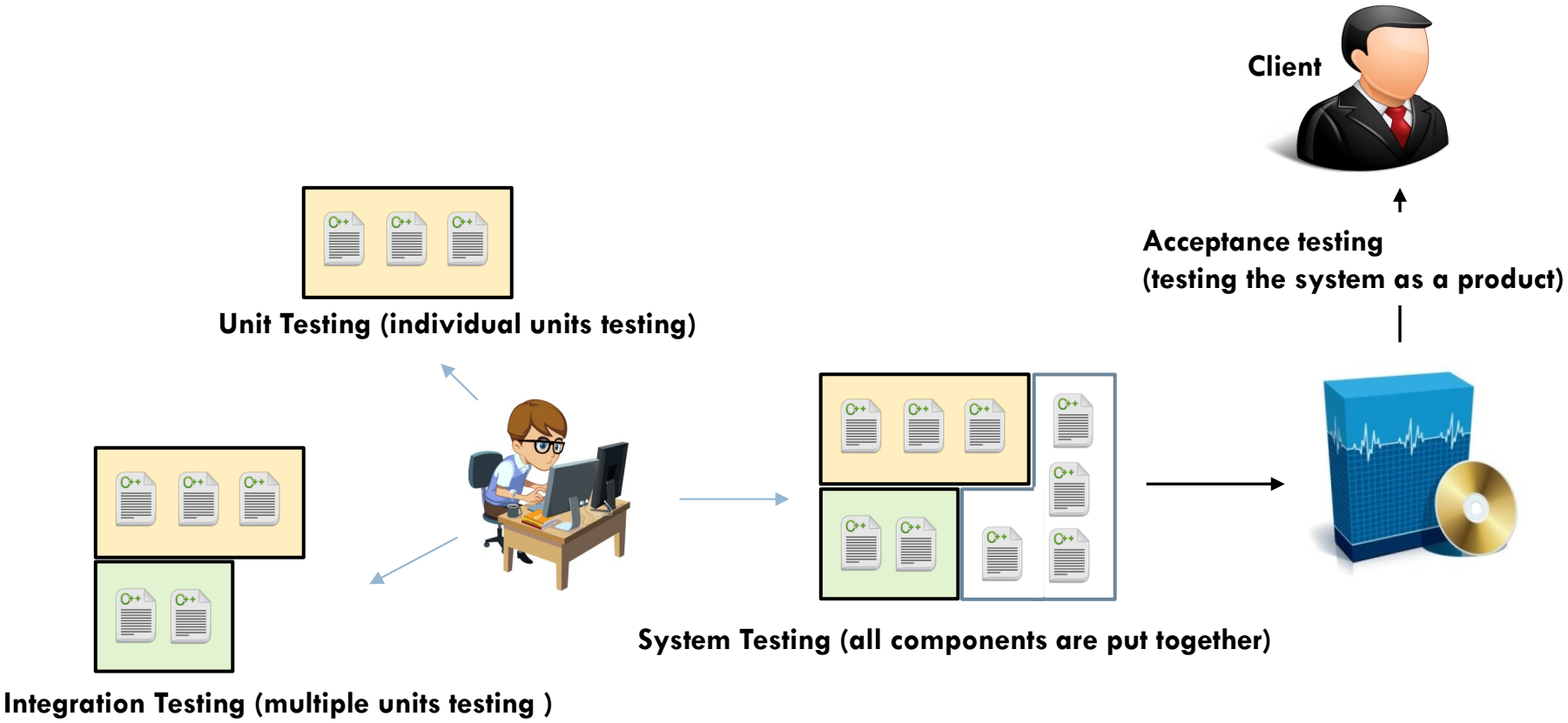
# Types of Testing

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

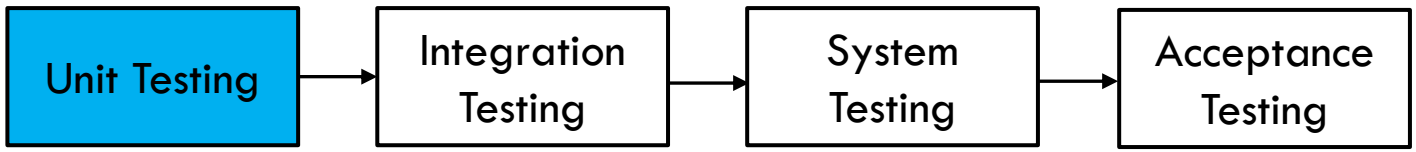
# Types of Testing



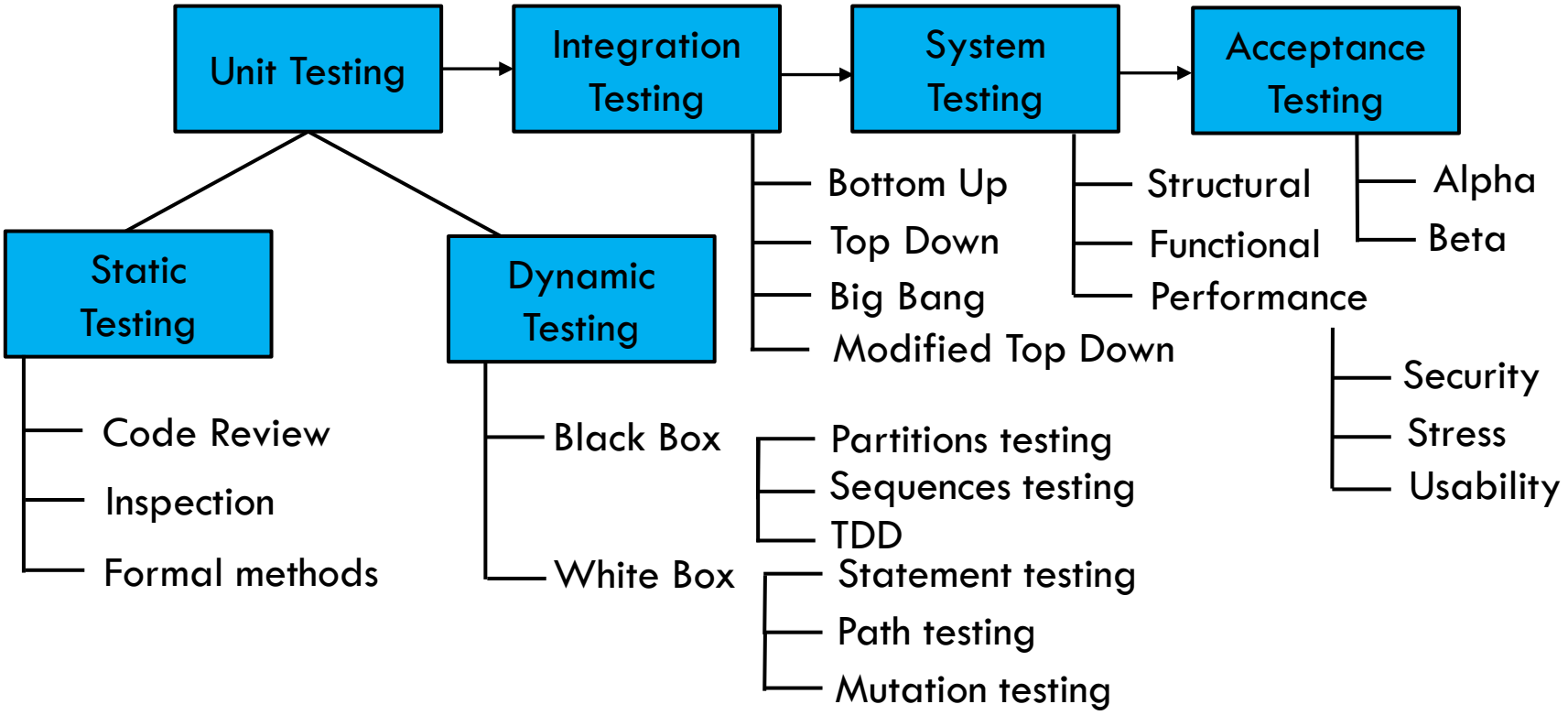
# Testing: Unit, Integration, System



# Types of Testing



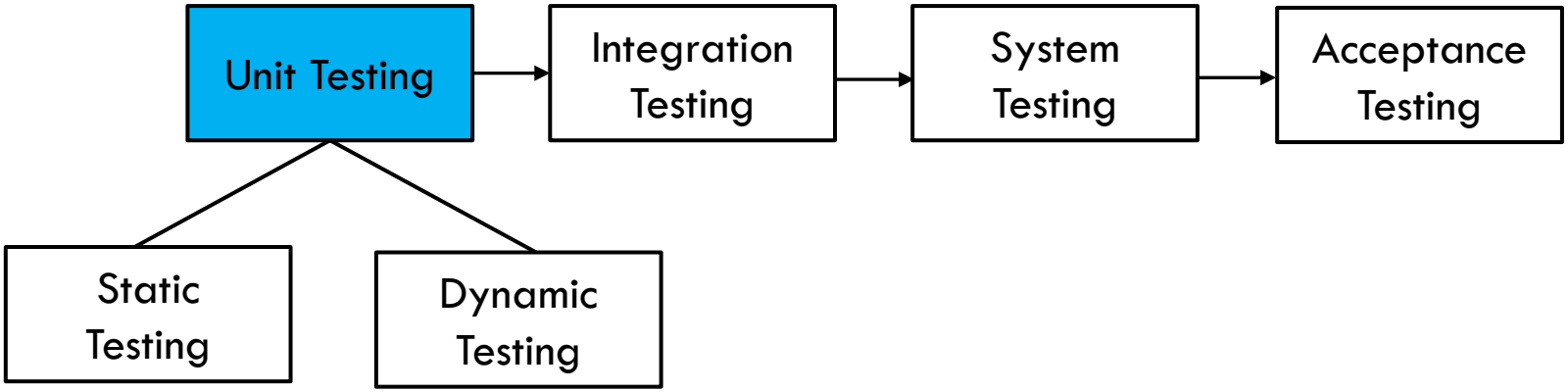
# Types of Testing



# Unit Testing

- Each program component is tested on its own, isolated from the other components in the system.
- Carried out by developers
- Goal: Confirm that the component is correctly coded
- Methods: Static and dynamic

# Unit Testing: Static vs. Dynamic



# Unit Testing: Static Testing

- Code Reviews, at compile time:
  - Code Walkthrough
  - Code Inspection
  - Formal Methods

# Walkthrough

- Informal
- Present your code and documentation to the review team (other developers)
- Team comments on code correctness

# Unit testing: Code reviews: Walkthrough

Focus should be on the code, not the coder

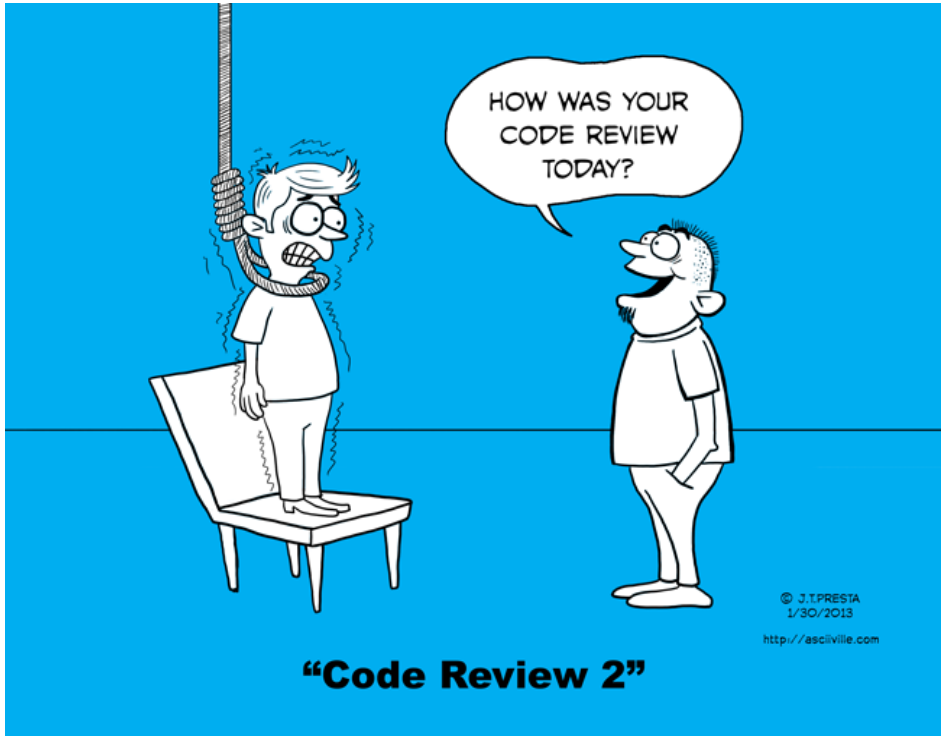
# Unit testing: Code reviews: Inspection

- Formal
- The review team checks the code and documentation against a prepared list of concerns

# Unit testing: Code reviews: Inspection

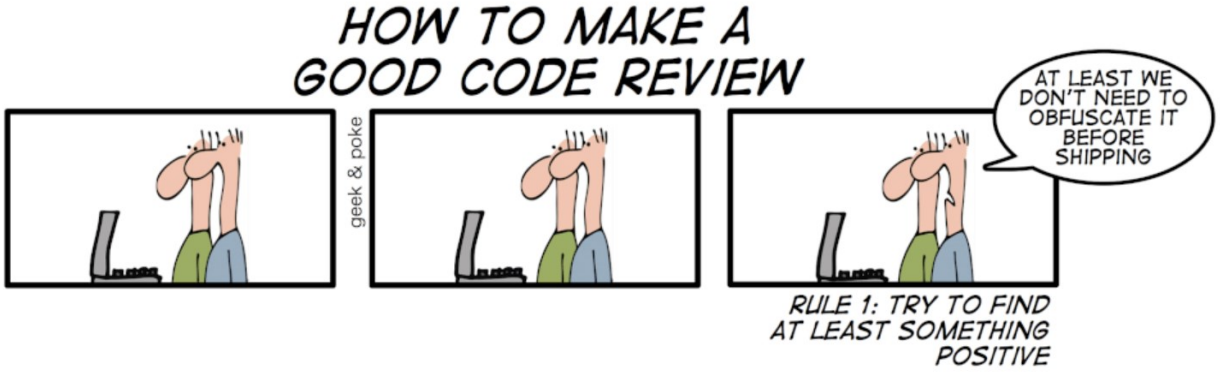
- **inspection**: A more formalized code review with:
  - roles (moderator, author, reviewer, scribe, etc.)
  - several reviewers looking at the same piece of code
  - a specific checklist of kinds of flaws to look for
    - possibly focusing on flaws that have been seen previously
    - possibly focusing on high-risk areas such as security
  - specific expected outcomes (e.g. report, list of defects)

# Inspection

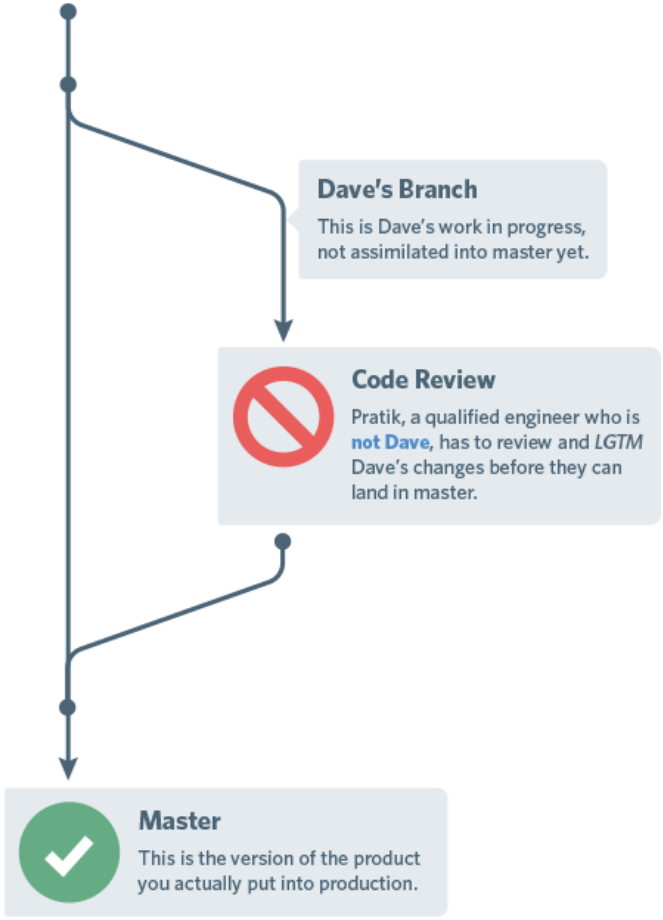



https://www.youtube.com/watch?v=ocMraYgqHvg

# Inspection


HOW TO MAKE A GOOD CODE REVIEW
geek & poke
AT LEAST WE DON'T NEED TO OBFUSCATE IT BEFORE SHIPPING
RULE 1: TRY TO FIND AT LEAST SOMETHING POSITIVE

# Code Review at Google



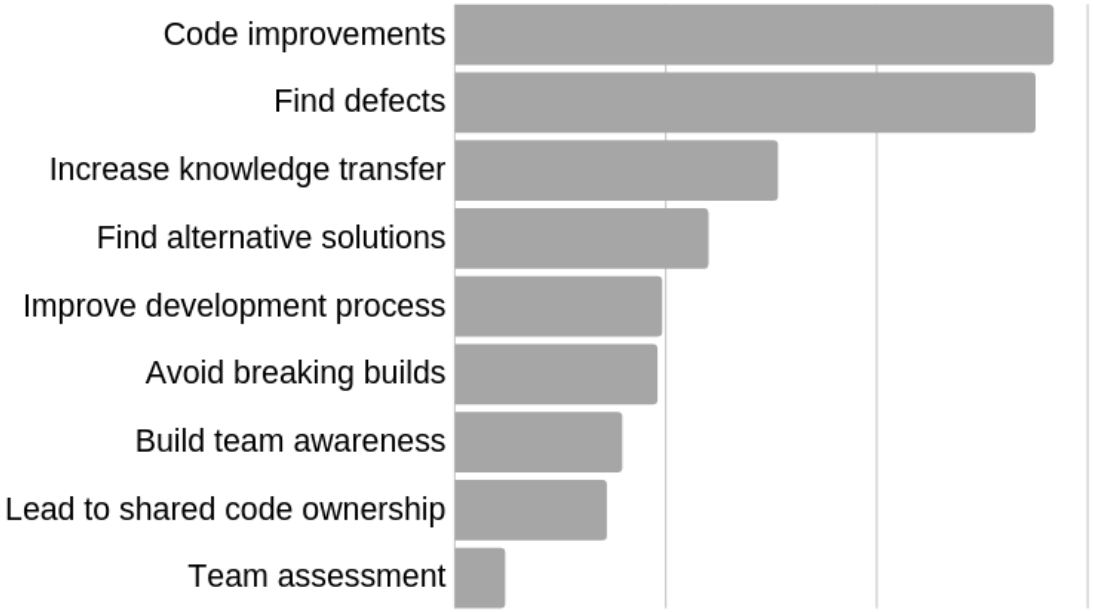

FULLSTORY CULTURE • 8 MIN READ

## What We Learned from Google: Code Reviews Aren't Just for Catching Bugs

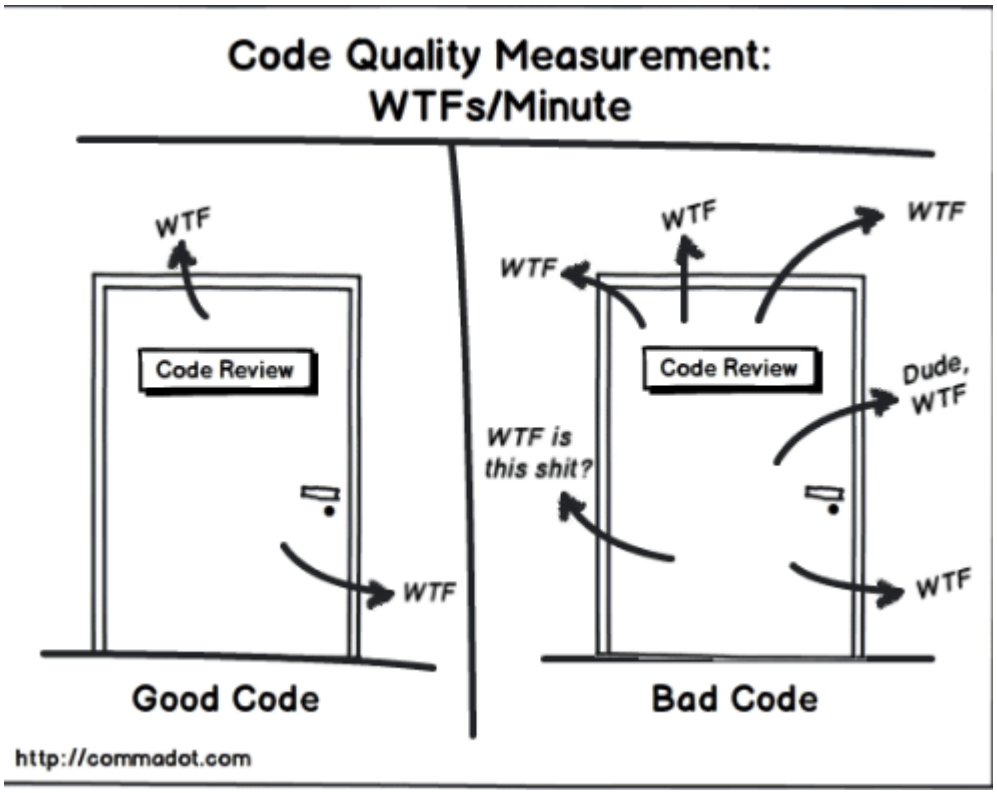
Bruce Johnson posted on April 19, 2016

# Code Review at Microsoft

At Microsoft, code reviewing is a highly adopted engineering practice and perceived as a great best practice. CLICK TO TWEET



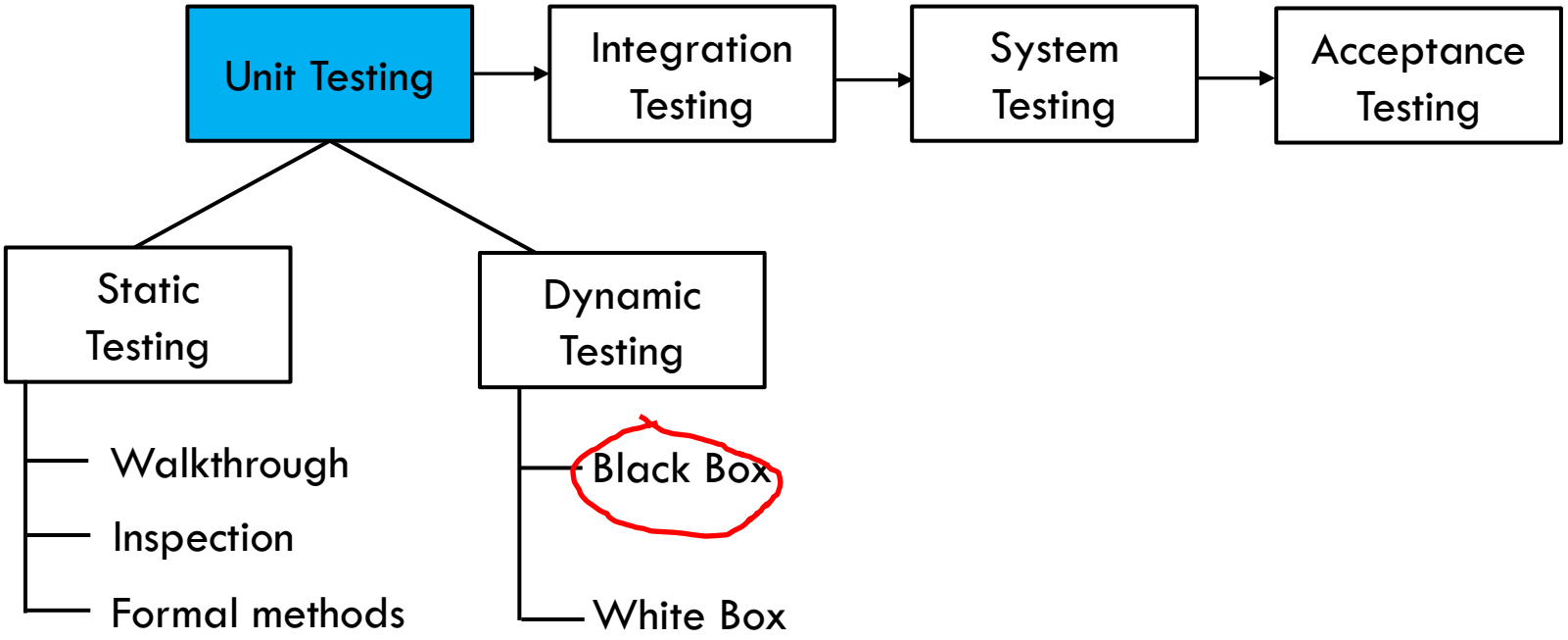
# Code Reviews



# Formal Proof Techniques

- A more structured way to establish code correctness
- Code correctness is formulated as theorem proofing
- Extremely complicated, often takes years.
- Necessary in safety-critical systems

# Unit Testing

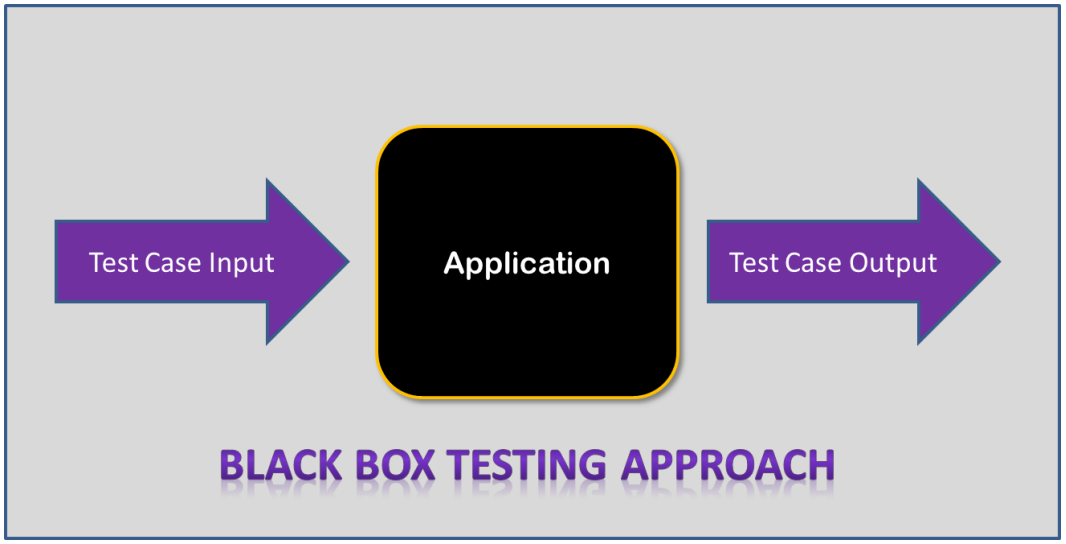


# Unit Testing: Dynamic Testing

- Testing the program at runtime
  - Black box testing (closed)
  - White box testing (clear)

# Black Box Testing

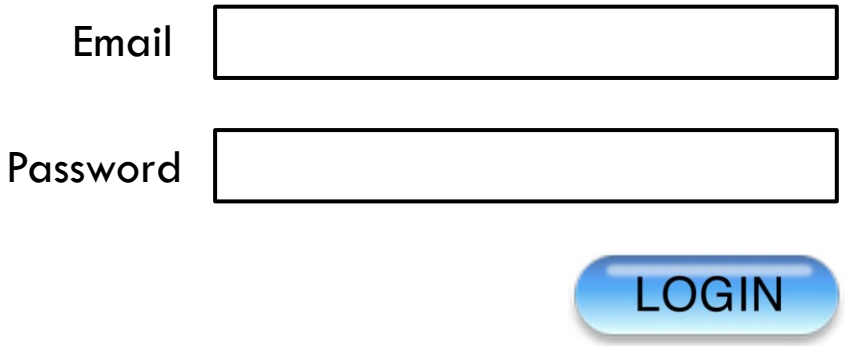
- Test the functionality of the component
- External behavior description



# Black Box Testing

- Based on GUI (input and output).
- Tester designs test cases with inputs and the corresponding expected outputs.

# Black Box Testing: Example




Specs:

User with valid username and password shall be able to login

User with invalid username and password shall receive error message

The system locks after 3 attempts, see admin to reactivate

# Generating Test Cases

Test with valid email/password → login

Test with valid email/invalid password → error

Test with invalid email/valid password → error

Test with invalid email/invalid password → error

Test with invalid “email/password” 1 time → error

Test with invalid “email/password” 2 times → error

Test with invalid “email/password” 3 times → lock

# Test Cases Table

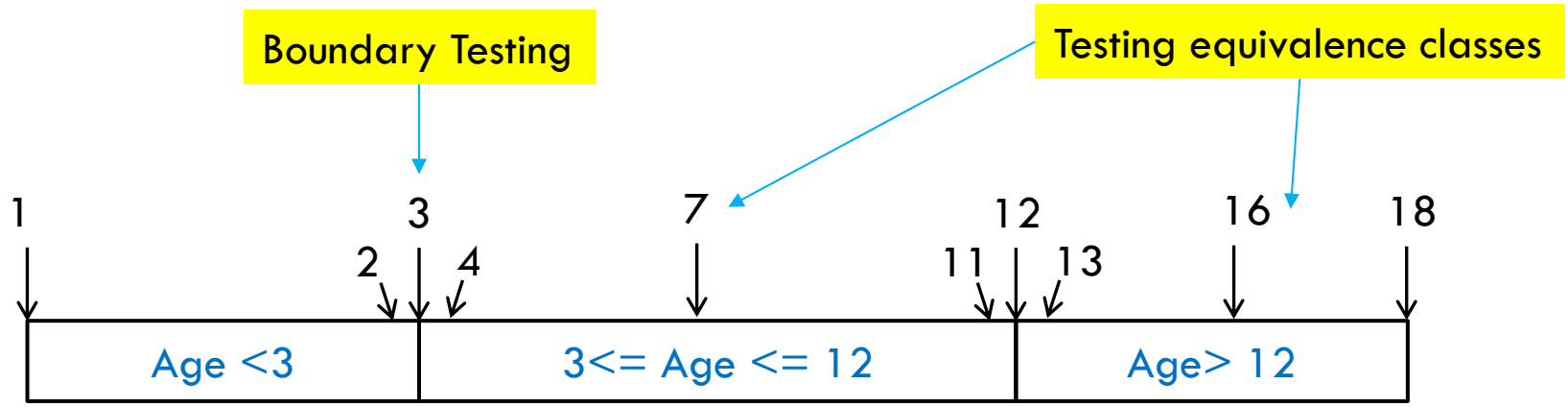
| Test case | Expected Output |
| --- | --- |
| valid Email/ valid password | Login |
| invalid Email/ valid password | Error message |
| valid Email/ invalid password | Error message |
| invalid Email/ invalid password | Error message |
| invalid “Email/ password” 1 time | Error message |
| invalid “Email/ password” 2 times | Error message |
| invalid “Email/ password” 3 times | Error message/lock |

# Black Box Testing

How many test cases do we need?

# Black Box – Partition Testing

- A system accepts the child's in between the age [1 - 18] and outputs required set of vaccines as follows: Children under the age of 3 years old get vaccine group A, children between 3 and 12 get group B, children over 12 get group C.



# Black Box – Partition Testing

| Test Cases | |
|---|---|
| Input: Age | Output: Vaccine |
| 1 | A |
| 2 | A |
| 3 | B |
| 4 | B |
| 7 | B |
| 11 | B |
| 12 | B |
| 13 | C |
| 16 | C |
| 18 | C |

# Black Box – Partition Testing

- Divides the input domain of a program into classes of data from which test cases are derived
- An equivalence class represents a set of valid or invalid states for input conditions
- It is always good to test on boundaries

# Equivalence Partitioning

- **Equivalence partitioning:** a software testing technique that divides the input data of a software unit into partitions of equivalent data from which test cases can be derived.

- **Boundary Value Analysis (BVA):** selecting test cases that would identify errors at the boundaries of equivalence partitions.

# Equivalence Partitioning

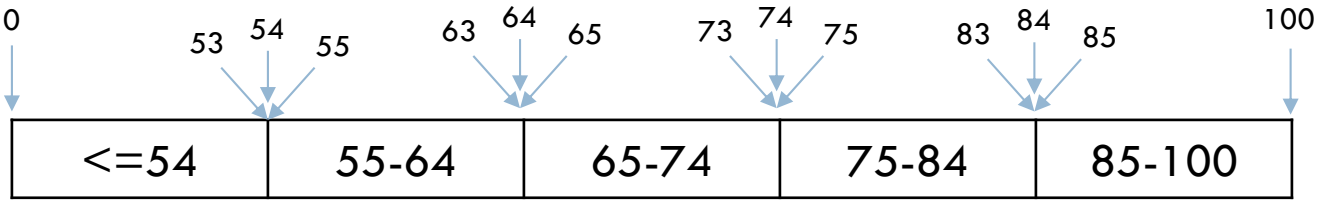
- The method `getLetterGrade` takes the input grade as a numeric value [0-100] and returns the equivalent letter grade based on the following scale.

| Numeric Grade (g) | Letter Grade |
|---|---|
| 84 < g <= 100 | A |
| 74 < g <= 84 | B |
| 64 < g <= 74 | C |
| 54 < g <= 64 | D |
| <=54 | F |

Generate test cases based on Boundary Value Analysis (BVA) to test this method.

# Equivalence Partitioning

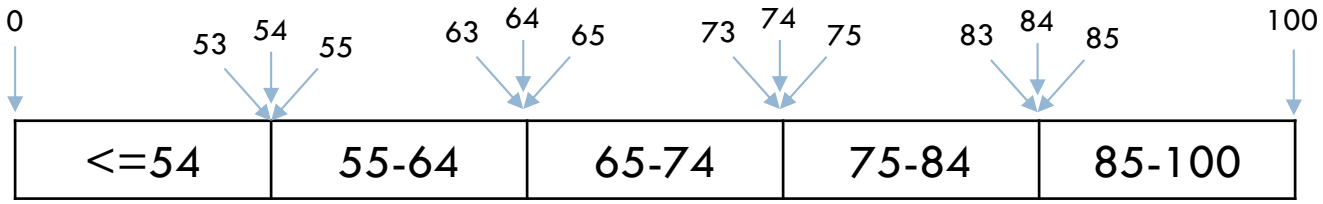
| Numeric Grade | Letter Grade |
|---|---|
| 84 < g <= 100 | A |
| 74 < g <= 84 | B |
| 64 < g <= 74 | C |
| 54 < g <= 64 | D |
| <=54 | F |



# Equivalence Partitioning

| Numeric Grade | Letter Grade |
|---|---|
| 84 < g <= 100 | A |
| 74 < g <= 84 | B |
| 64 < g <= 74 | C |
| 54 < g <= 64 | D |
| <=54 | F |

| Numeric Grade | Expected outcome |
|---|---|
| 0 | F |
| 53 | F |
| 54 | F |
| 55 | D |
| 63 | D |
| 64 | D |
| 65 | C |
| 73 | C |
| 74 | C |
| 75 | B |
| 83 | B |
| 84 | B |
| 85 | A |
| 100 | A |



# Black Box Testing: Sequences

```
int searchSequence(int[] array, int target)
```

Target 99

| Sequence | 10 | 15 | 17 | 99 | 0 | -3 | 14 | -2 | 141 | -30 |
|---|---|---|---|---|---|---|---|---|---|---|

Input:
1- A target integer
2- A sequence of integers of length *n*

Output:
1- Target found: return index
2- Target not in sequence: return -1

# Black Box Testing: Sequences

- Specifications:

1. The program takes a sequence of integers and a key element
2. The program searches the sequence for the key element
3. If the key element is found, the program returns the position of that element in the sequence
4. Else the program returns -1
5. Precondition: the sequence has at least one integer

# Black Box Testing: Sequences

- Possible equivalent classes (partitions)?
  - Target is/is not in sequence
  - Sequence length
  - Target location in sequence

# Black Box Testing: Sequences

| Sequence | Element |
|---|---|
| Single Value | In sequence |
| Single Value | Not in sequence |
| More than 1 value | First element in sequence |
| More than 1 value | Last element in sequence |
| More than 1 value | Middle element in sequence |
| More than 1 value | Not in sequence |

# Black Box Testing: Sequences

| Sequence | Element | Outcome |
|---|---|---|
| 17 | 17 | 1 |
| 15 | 5 | -1 |
| 17, 29, 21, 23 | 17 | 1 |
| 41, 18, 21, 23, 29, 41, 38 | 38 | 7 |
| 17, 18, 21, 23, 29, 41, 38, 12, 3 | 29 | 5 |
| 21, 23, 29, 33, 38 | 25 | -1 |

Program passes the test if it generates the expected outcome of each case

# Black Box Testing: Sequences

- General guidelines for testing sequences:
- Always test on a single element sequence
- Always test on the edges
- A middle point is always good
- Use different size sequences in different tests
- Make sure to generate all possible outcomes (example: found, not found)

# Sequences Testing: `getMedian`

- The method `getMedian` takes an integer array as input and returns the median (middle value) in the array. Suggest 5 equivalence classes with sample test cases to test this method.

```
double getMedian(int[] a)
```

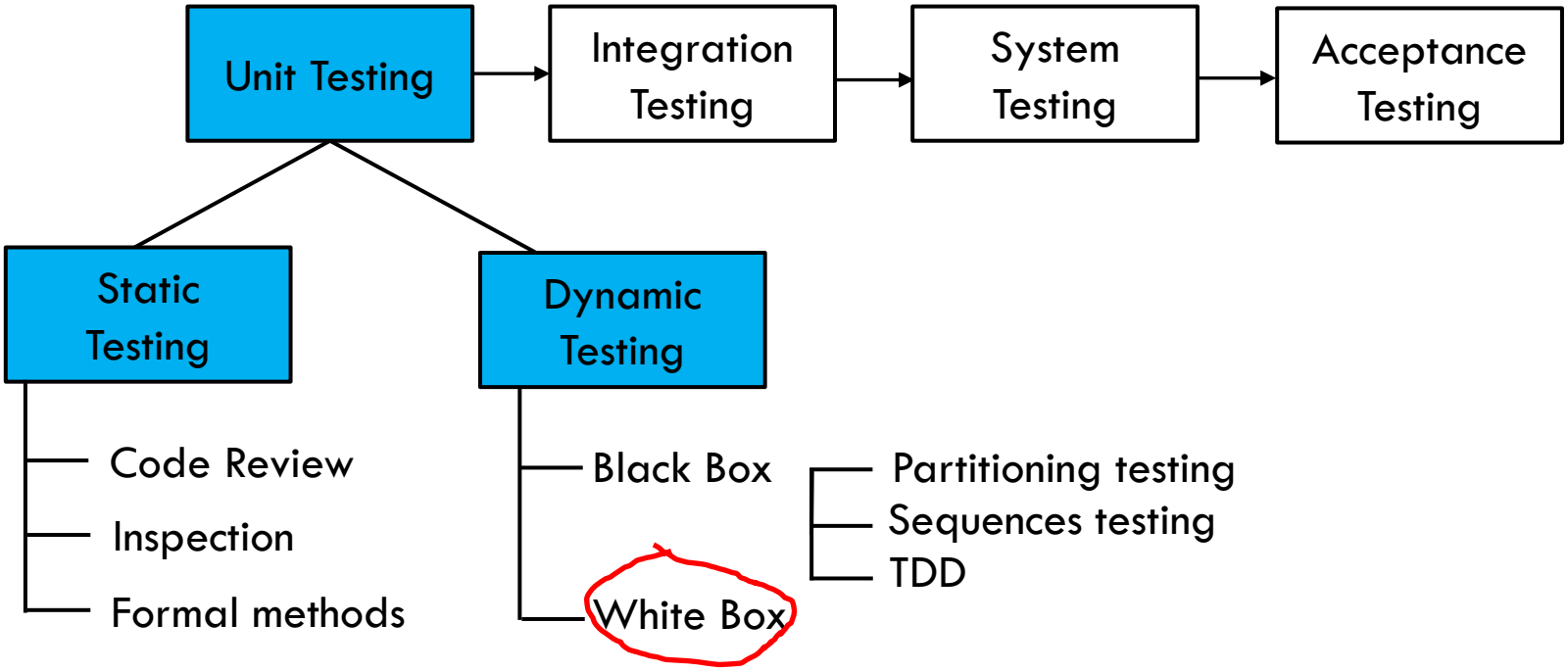
| Input | 10 | 15 | 17 | 99 | 0 | -3 | 14 | -2 | 141 | -30 |
|---|---|---|---|---|---|---|---|---|---|---|

Output: 12.0

# Sequences Testing: `getMedian`

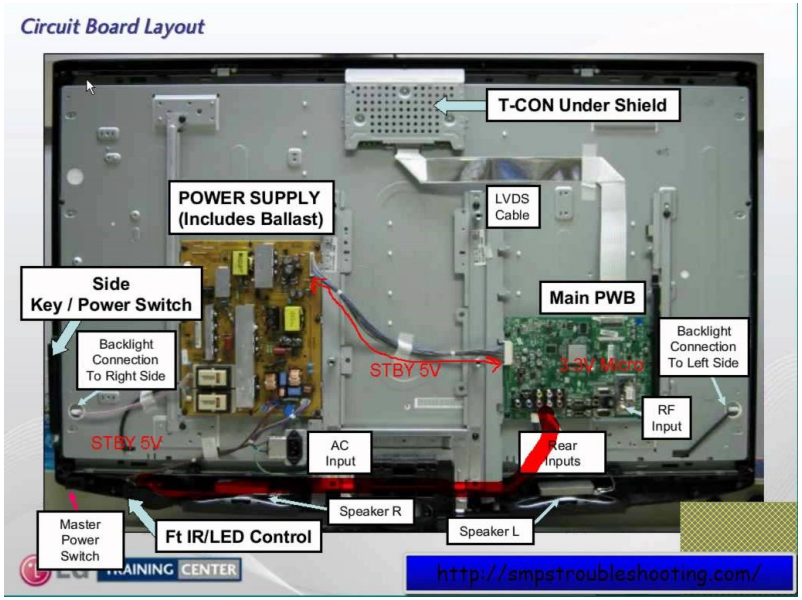
| Equivalence class | Sample test case | Expected outcome |
|---|---|---|
| Single element | [3] | 3 |
| # Elements > 1, odd | [3, 5, 4] | 4 |
| Even number of elements | [6, -3, 4, 11] | 5 |
| Same element | [-7, -7, -7, -7] | -7 |
| Sorted array | [-3, 3, 4, 5, 9] | 4 |

# Previously on CSC 4330




GLOBALLY MINDED
SKILLED COMMUNICATOR
PROBLEM SOLVER
GLOBALLY MINDED
SKILLED COMMUNICATOR
PROBLEM SOLVER

# White box testing

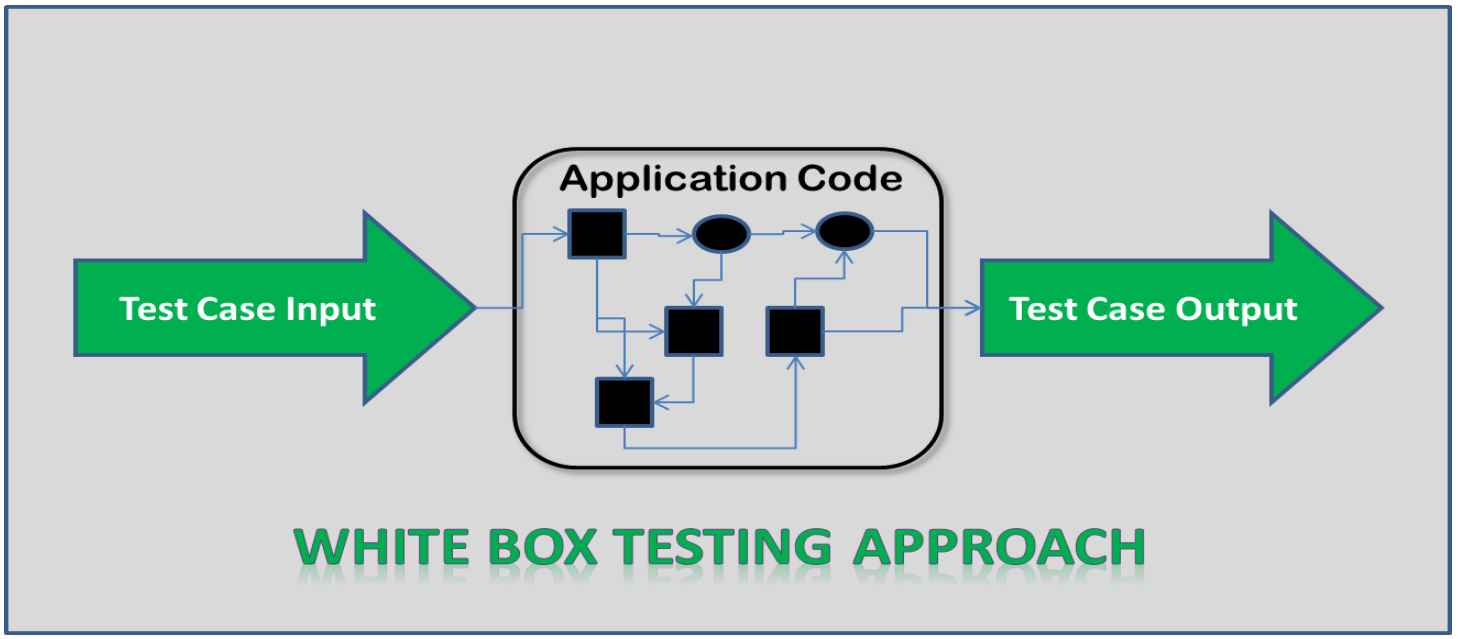

**Black box**

**White box**

# White Box Testing

- The tester has a knowledge of the internals of a system
- Develop test cases that will examine the implementation of the code including:
  - Statement coverage
  - Path coverage

# White Box Testing

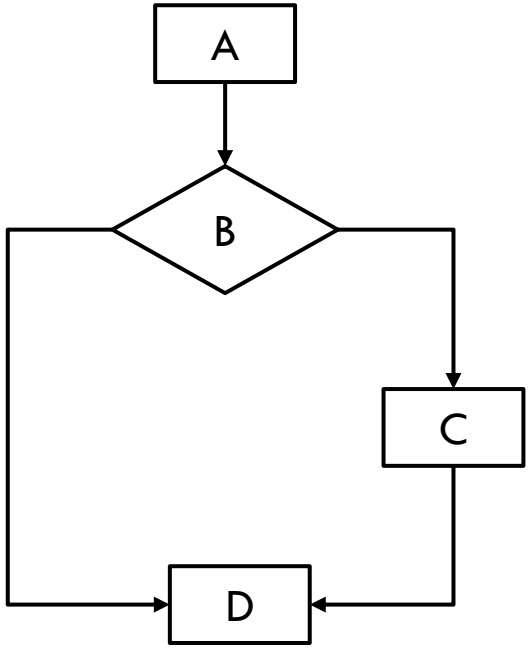


# Path Testing

- This methodology uses the graphical representation of the source code
- Thus it is very much “control flow”
- Analyzes the number of paths that exist in the system
- Develop a test case for each path

# Path Testing

```
A.  INPUT NUM

B.  IF NUM > 0
C.      PRINT "+"

D.  PRINT "END"
```

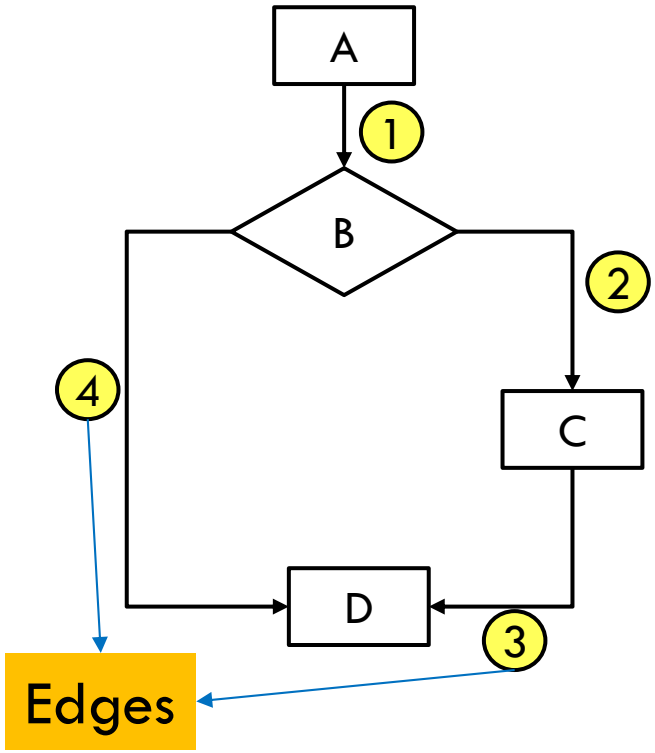



How Many Paths?

# How Many Paths?

Each edge connects two statements

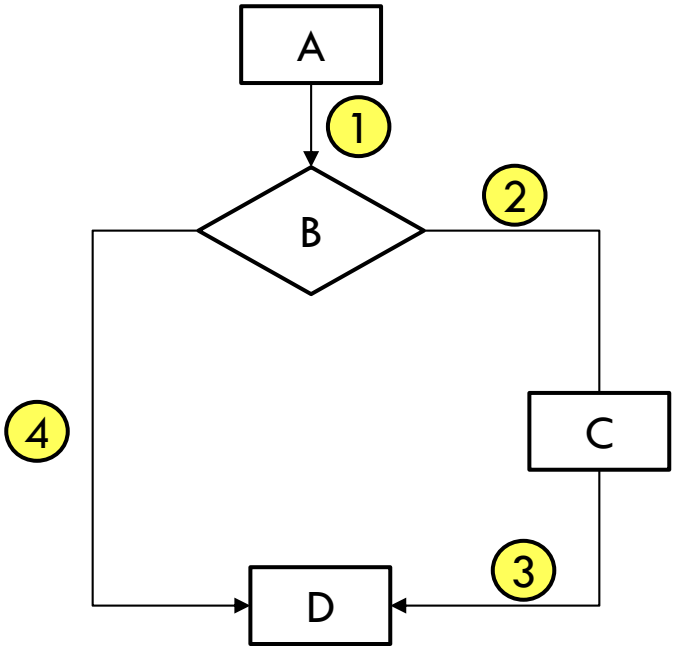
A test case is needed for each unique path in the system



# How Many Paths?

Path1 = A,B,C,D

Path2 = A,B,D

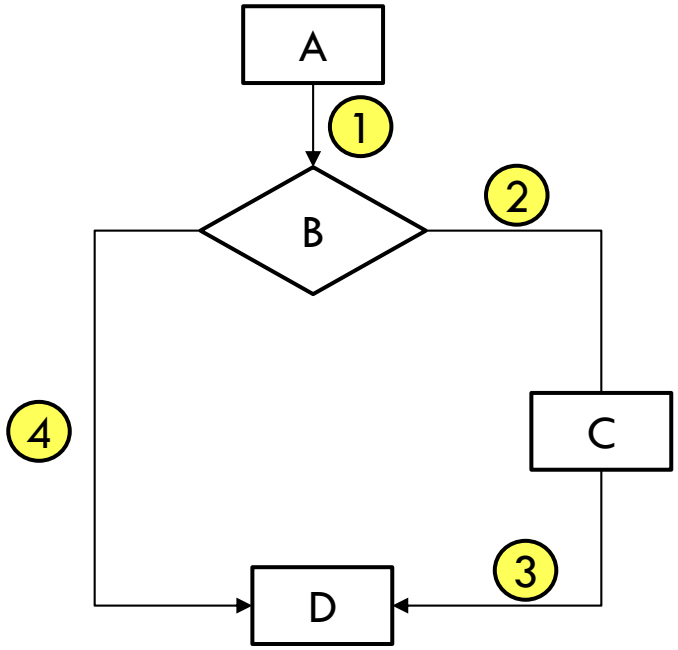


# How Many Paths?

$Path_1$ = A,B,C,D

$Path_2$ = A,B,D

```
A. INPUT NUM

B. IF NUM > 0
C.     PRINT "+"

D. PRINT "END"
```

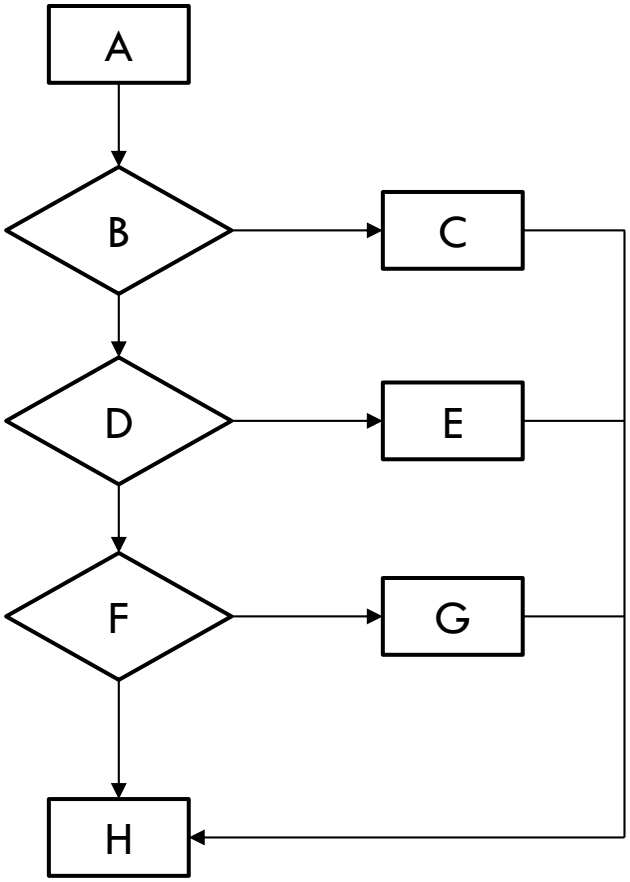



$TC_1$ : NUM = 1 ➔ executes $Path_1$

$TC_2$ : NUM = -1 ➔ executes $Path_2$

# Path Testing: Ex-2

- We need test cases to execute each path



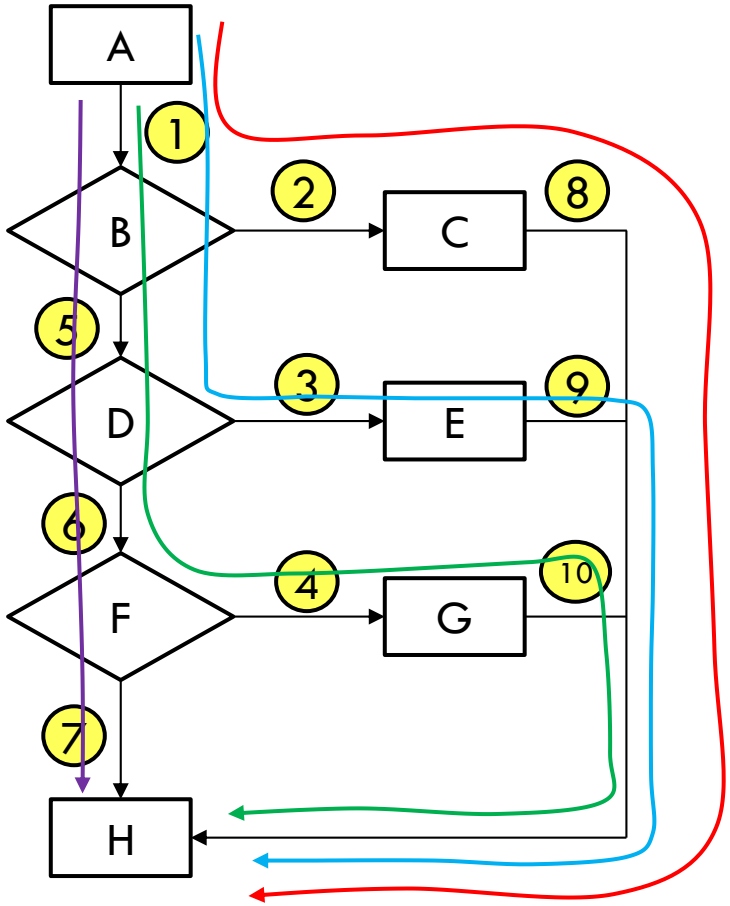
# Path Testing: Ex-2

$Path_1$ = ABCH

$Path_2$ = ABDEH

$Path_3$ = ABDFGH

$Path_4$ = ABDFH




Number of test cases = number of paths

# How Many Unique Paths Exist?

- Cyclomatic complexity: a software metric used to calculate the number of independent paths in a program

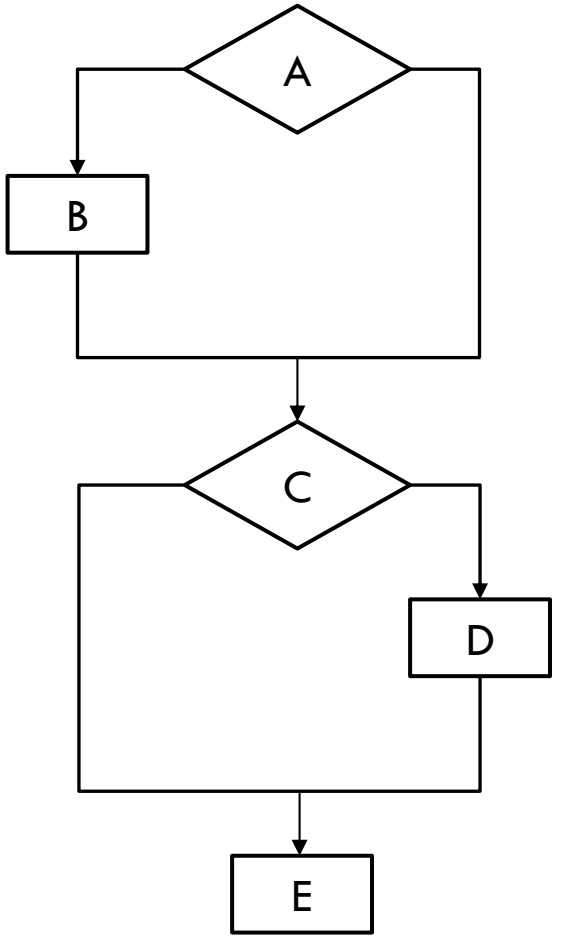
Cyclomatic complexity = Number of binary conditions +1

- A binary condition is a conditional statement (<, <=, >, =>, ==, !=)

# Unique Paths ≠ All Possible Paths

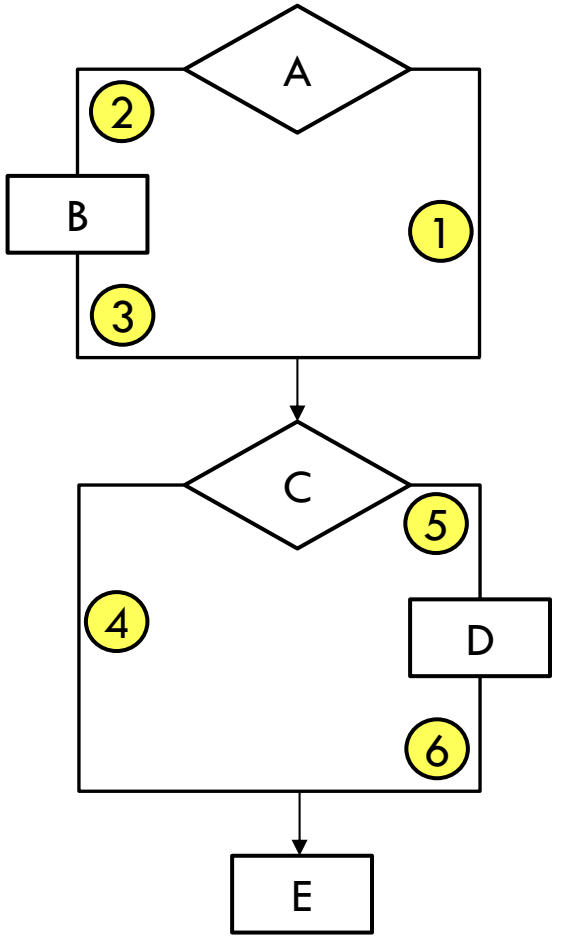
- $Path_1$ = ACDE
- $Path_2$ = ACE
- $Path_3$ = ABCDE
- $Path_4$ = ABCE

- $Path_4$ does not introduce any new edges. Thus can be eliminated from the set of paths to test



# Cyclomatic Complexity

- Two binary conditions (A and C)
- Cyclomatic complexity V(G)
- V(G) = 2 + 1 = 3



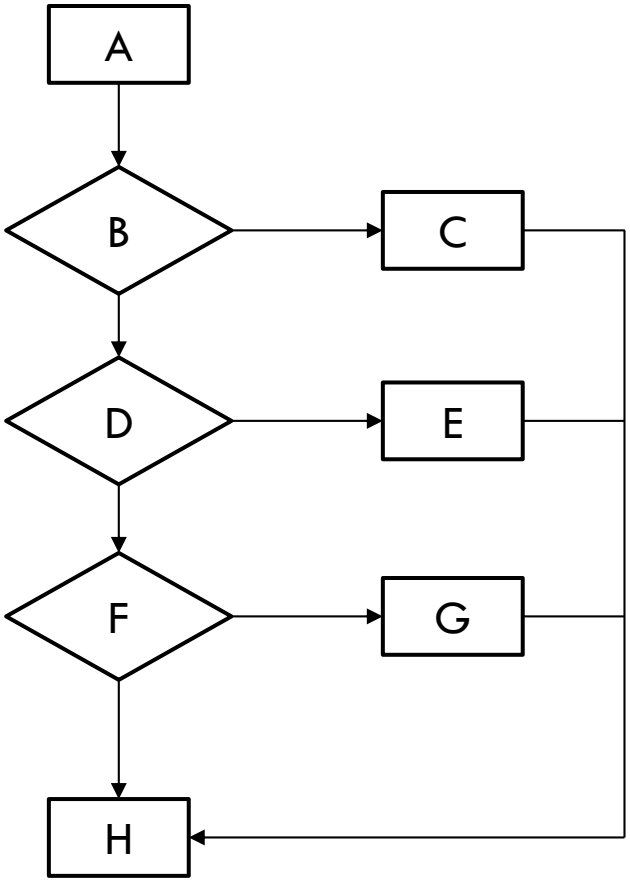
# How To Find Basic Paths?

- Select a valid path through the system that goes through the largest possible number of conditions
- Flip each condition once, each flip creates a new path. Continue until all the decisions have been flipped.

# Path Testing: Ex-2

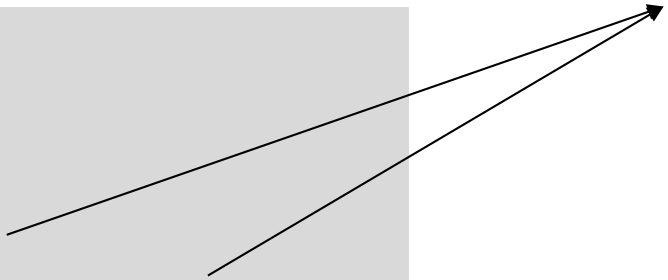
- Select P1 = A,B,D,F,H
- Flip B → A,B,C,H
- Flip D → A,B,D,E,H
- Flip F → A,B,D,F,G,H

- No more conditions to flip



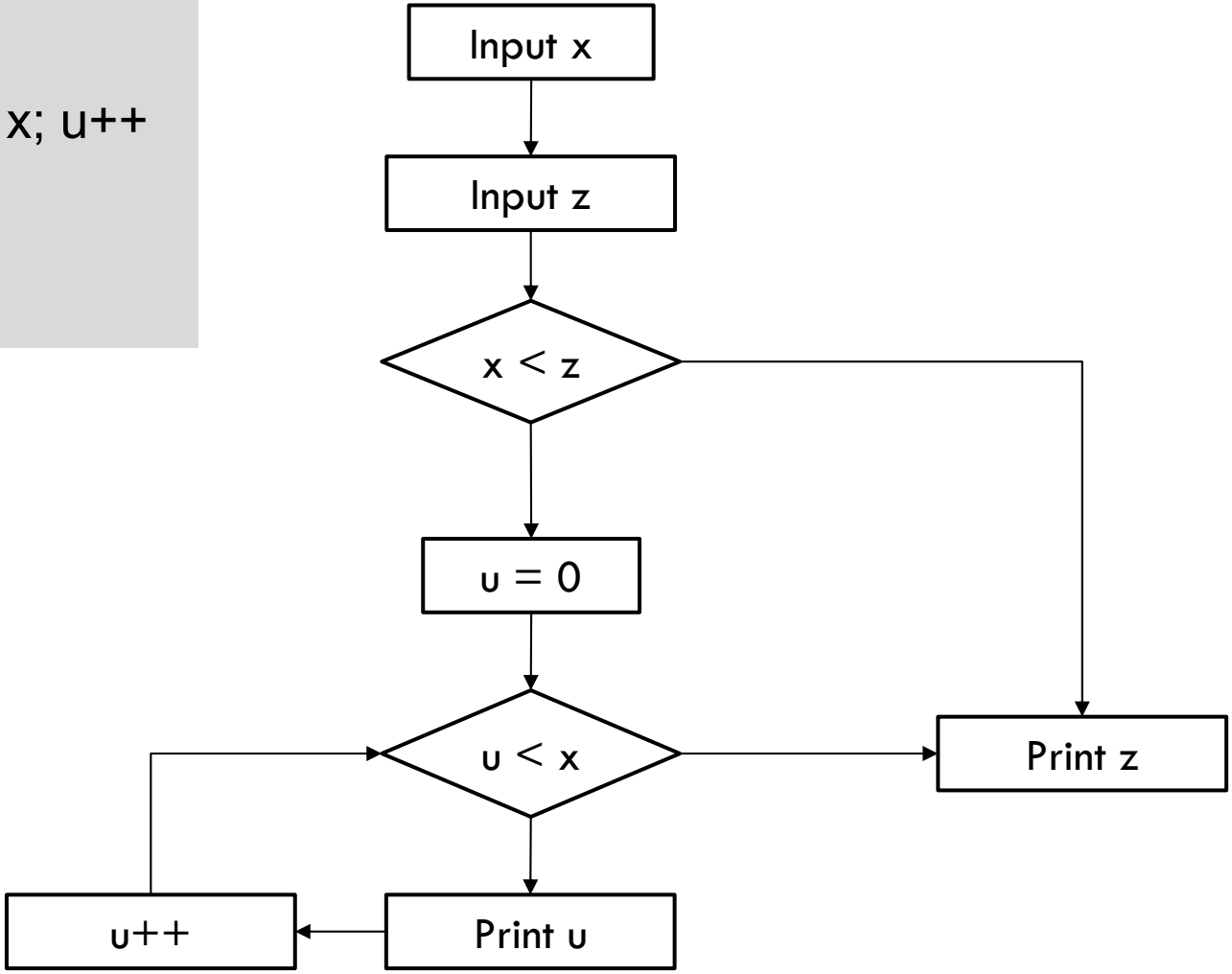
# White Box – Path Testing

```
A.  INPUT x
B.  INPUT z

C.  IF  x < z
D.      FOR  u=0, u < x; u++
E.          PRINT u

F.  PRINT z
```



Two conditions
V(G)= 2 + 1 = 3

```
A.  INPUT x
B.  INPUT z

C.  IF  x < z
D.      FOR  u=0, u < x; u++
E.          PRINT u

F.  PRINT z
```

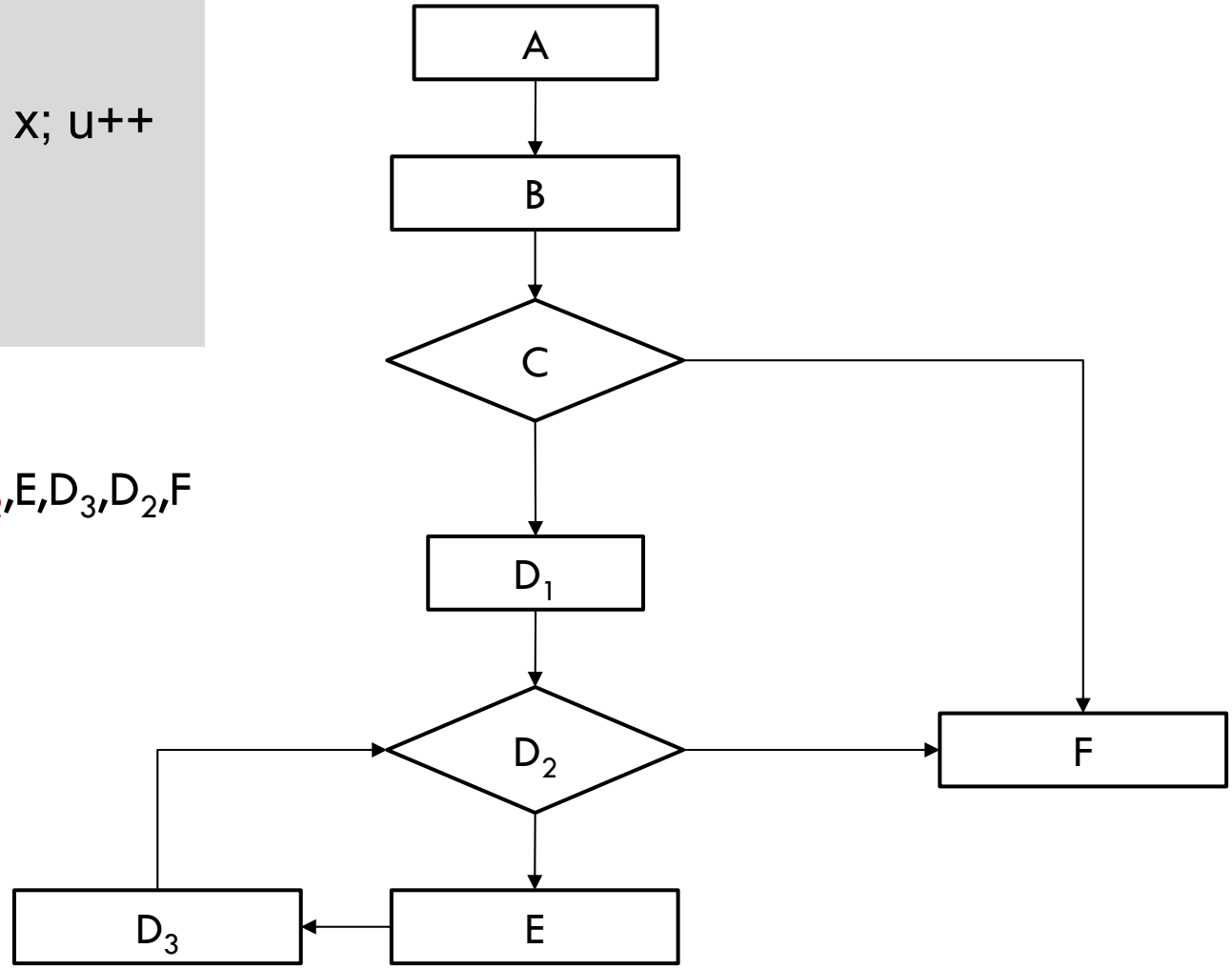


```
A. INPUT x
B. INPUT z

C. IF  x < z
D.      FOR  u=0, u < x; u++
E.          PRINT u

F.  PRINT z
```

Select P1 = A,B,C,$D_1$,$D_2$,E,$D_3$,$D_2$,F

Flip C ➔ A,B,C,F

Flip $D_2$ ➔ A,B,C,$D_1$,$D_2$,F

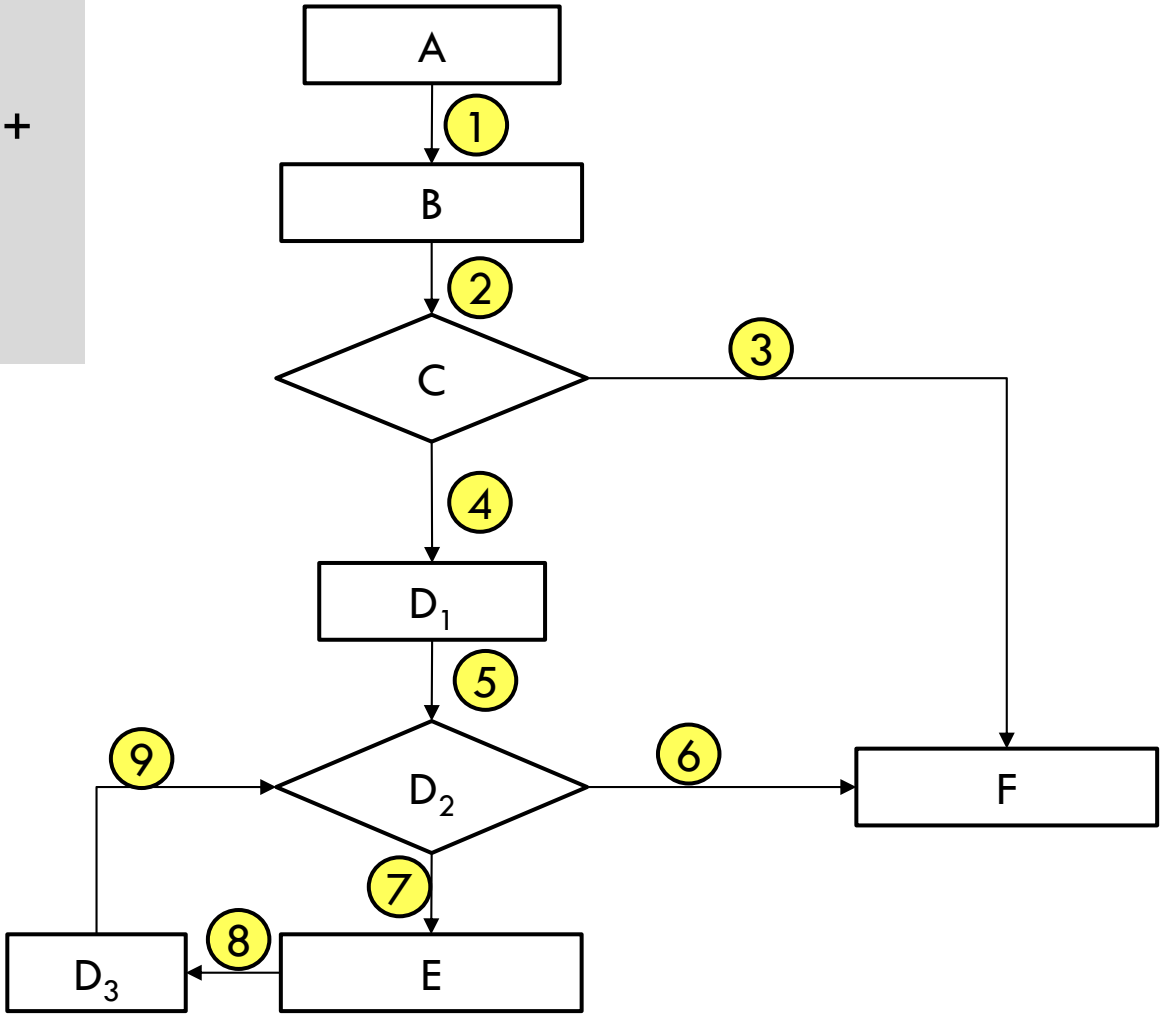


```
A. INPUT x
B. INPUT z

C. IF  x < z
D.      FOR  u=0, u < x; u++
E.          PRINT u

F.  PRINT z
```

1 paths is only required to achieve full statement coverage ($P_1$)

$P_1$ = 1,2,4,5,7,8,9,6
$P_2$ = 1,2,3
$P_3$ = 1,2,4,5,6

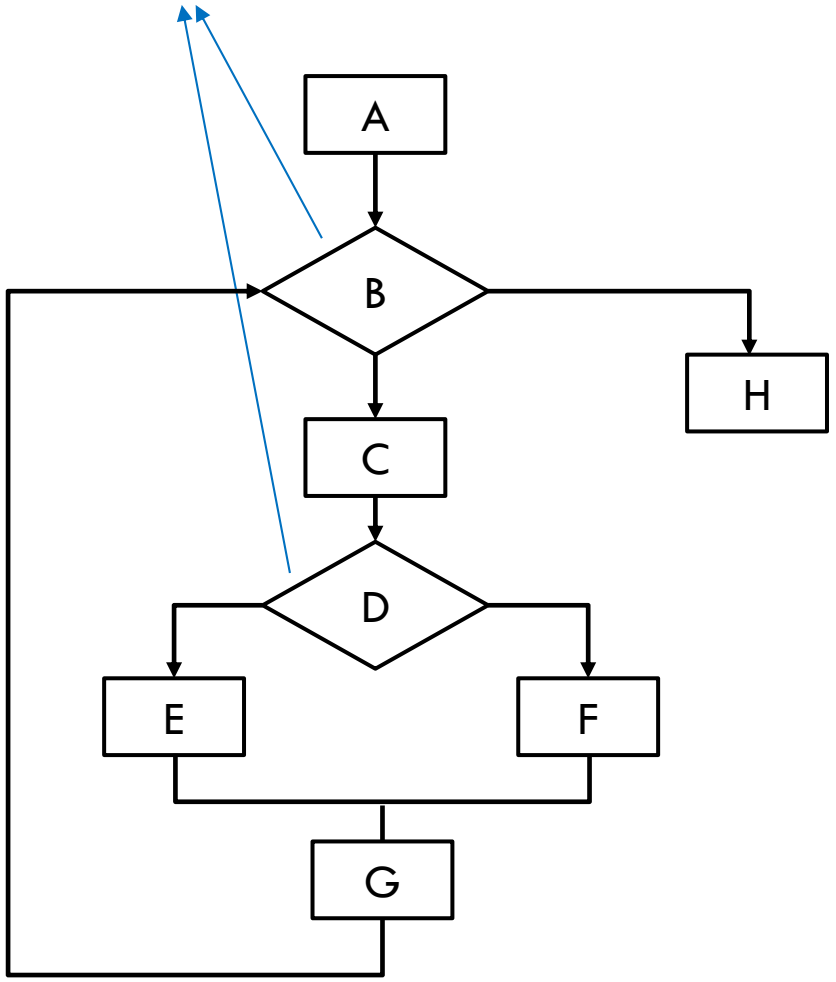
$P_1 = TC_1$ (x = 1, z =2)
$P_2 = TC_2$ (x = 2, z =1)
$P_3 = TC_3$ (x = 0, z =1)



# Example

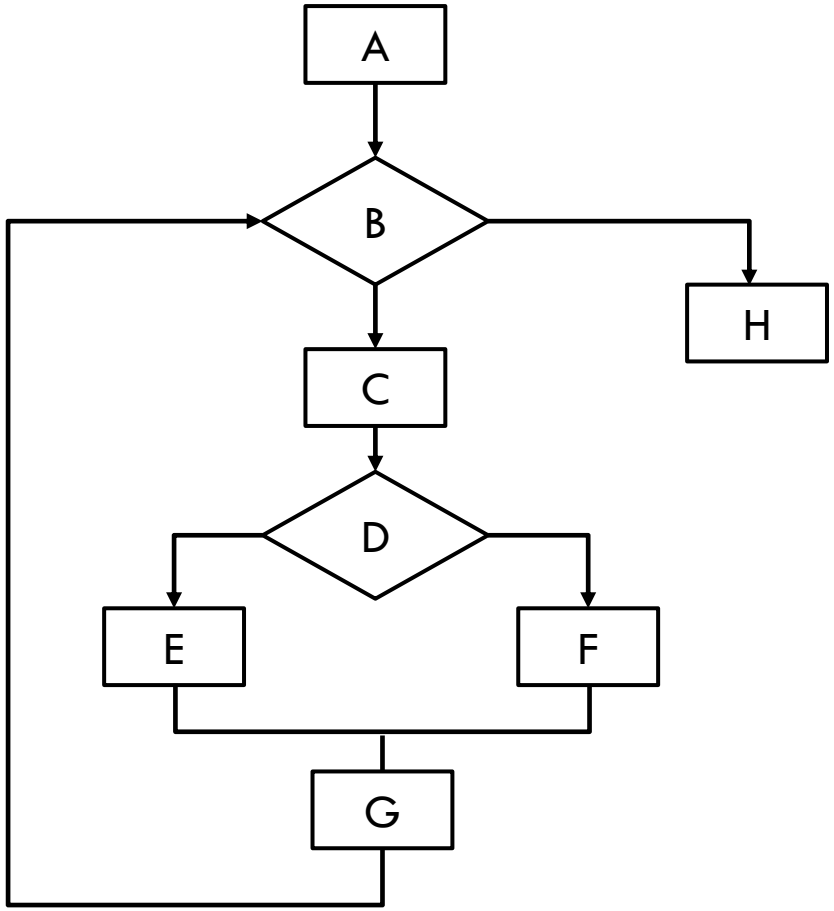
```
A.  INPUT NUM

B.  WHILE NUM < 10
    {
C.   PRINT NUM

D.   IF( NUM % 2 == 0)
E.           { PRINT EVEN }
     ELSE
F.           { PRINT ODD }

G.   NUM++;
    }

H. PRINT END
```

V(G) = 2 + 1 = 3



# Example

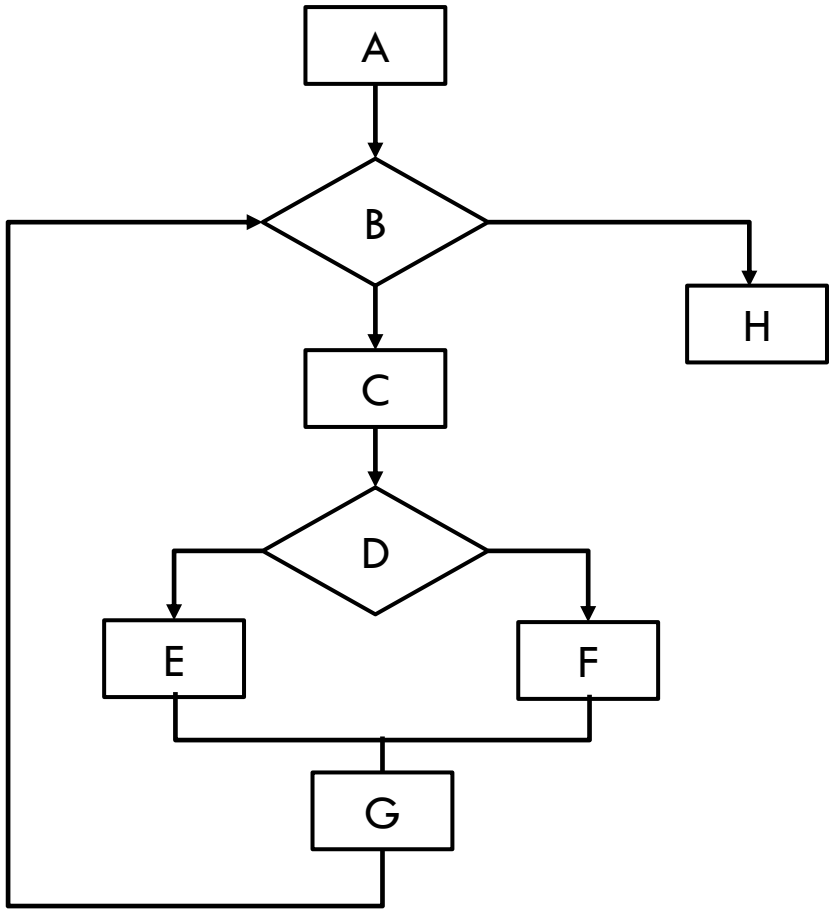
Select P1 = A,B,C,D,F,G,B,H
Flip B ➔ A,B,H
Flip D ➔ A,B,C,D,E,G,B,H



# Example

$Path_1$ = A,B,C,D,F,G,B,H
$Path_2$ = A,B,H
$Path_3$ = A,B,C,D,E,G,B,H

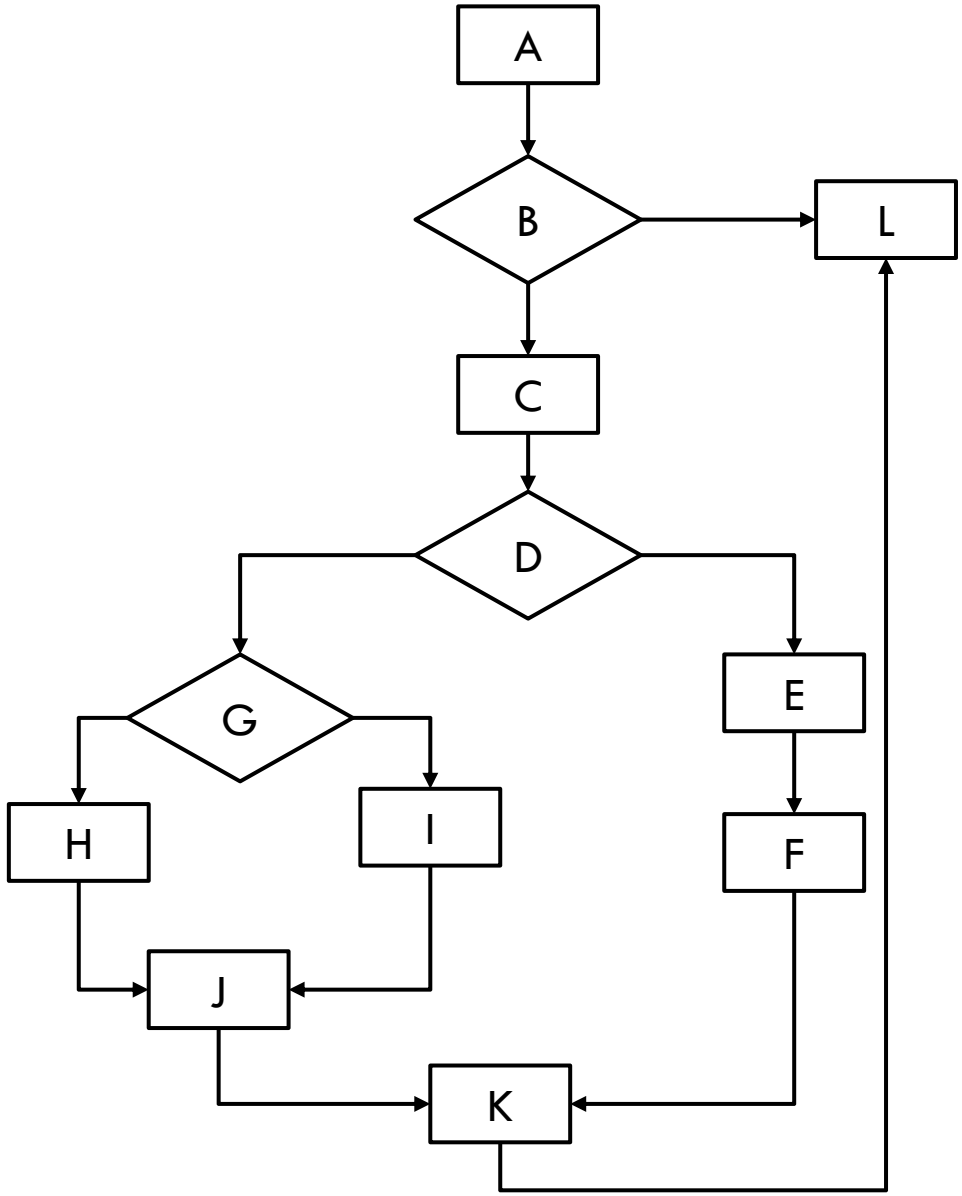
```
A.  INPUT NUM

B.  WHILE NUM < 10
    {
C.   PRINT NUM

D.   IF( NUM % 2 == 0)
E.          { PRINT EVEN }
     ELSE
F.          { PRINT ODD }

G.   NUM++;
    }

H. PRINT END
```




$TC_1$ → NUM = 9, covers $P_1$
$TC_2$ → NUM = 11, covers $P_2$
$TC_3$ → NUM = 8, covers $P_3$

2 paths are only required to achieve full statement coverage (1,3)

```
A.  INPUT NUM
B.  If NUM > 10
    {
C.    PRINT NUM
D.    IF( NUM % 2 == 0)
          {
E.         PRINT “Div2”
F.         NUM = NUM +2
          }
      ELSE
          {
G.         IF (NUM %3 ==0)
H.             PRINT Div3
           ELSE
I.             NUM = NUM +4
J.         NUM++
           }
K.     PRINT “END”
    }
L.  PRINT “EXIT”;
```



Select P1 = A,B,C,D,G,H,J,K,L

Flip B ➔ A,B,L

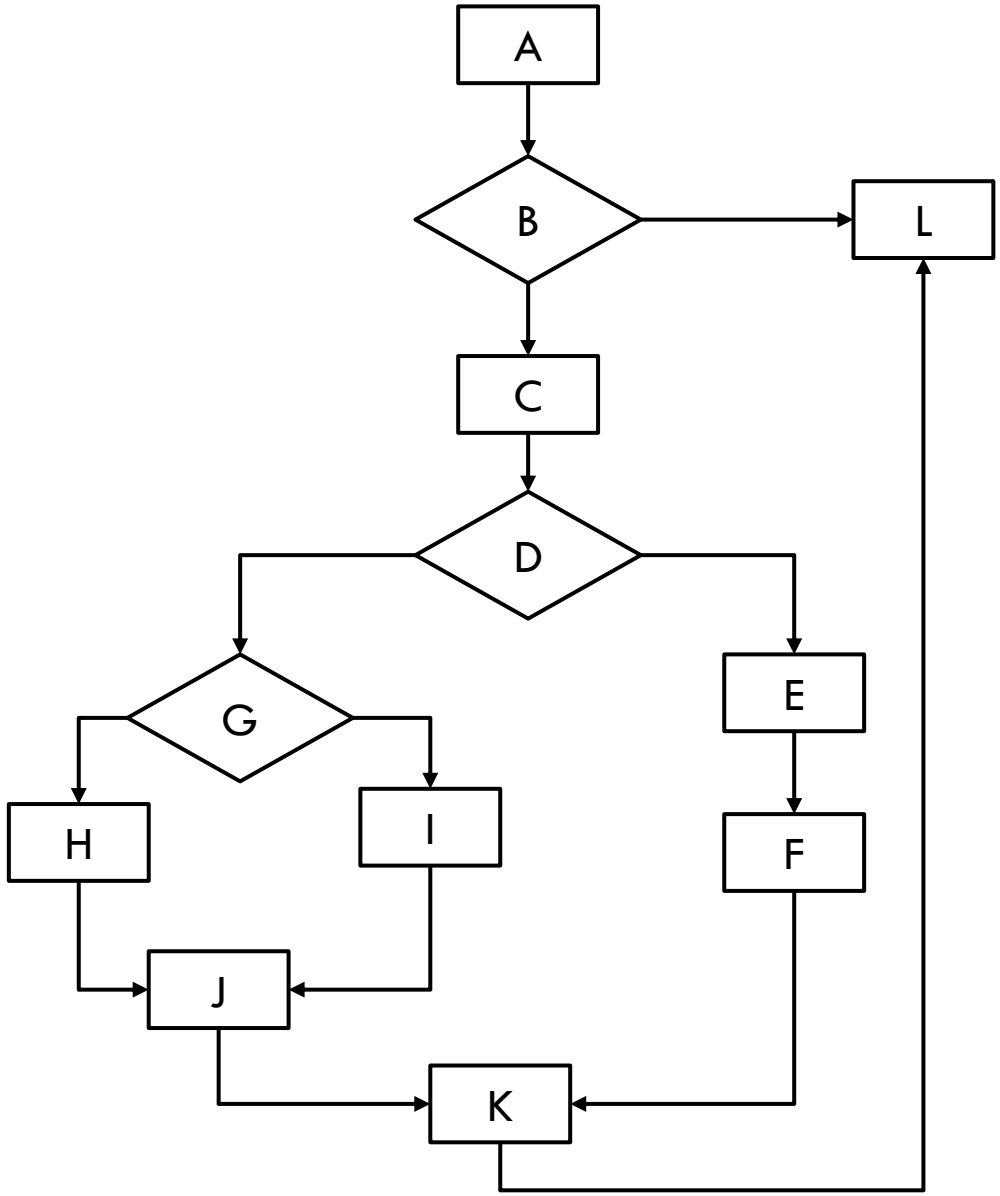
Flip D ➔ A,B,C,D,E,F,K,L

Flip G ➔ A,B,C,D,G,I,J,K,L

V(G) = 3 + 1 = 4

$P_1$ = ABCDGHJKL
$P_2$ = ABL
$P_3$ = ABCDEFKL
$P_4$ = ABCDGIJKL

3 paths are only required to achieve full statement coverage (1,3,4)



# Statement Testing – Example

$$\text{Statement coverage} = \frac{Number\ of\ Executed\ Statements}{Number\ of\ Statements} \times 100\%$$

```
1.  IF a > b
2.      Print "a > b"
3.      IF a > 10
4.          Print "a > 10"
5.  ELSE Print "a =< b"
```

a = 20, b = 10 ➔ executes: 1, 2,3,4
Coverage = $4/5$ = 80%

a = 10, b = 20 ➔ executes: 1,5
Coverage = $2/5$ = 40%

# Statement Coverage: How To?

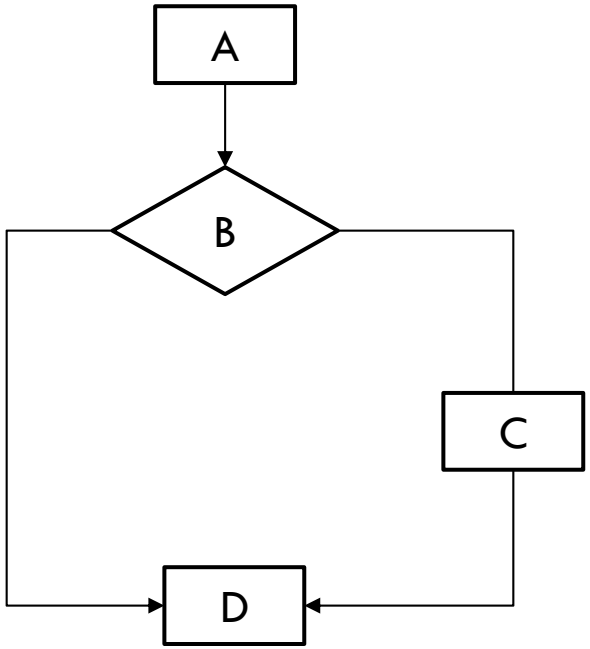
- Find all the paths
- Pick the minimum number of paths that will include all the statements

# Statement Testing

- This program has 4 statements: A, B, C, D
- V(G) = 1 + 1 = 2

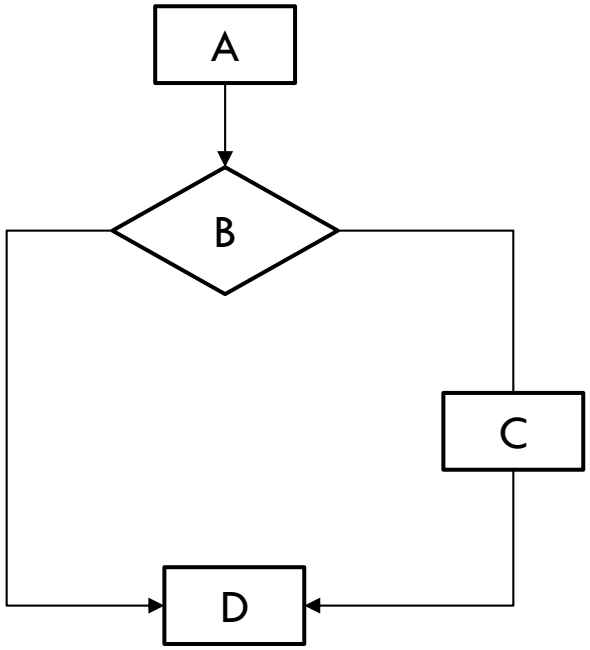
- The program has 2 paths:
  - $P_1$ covers A,B,C,D
  - $P_2$ covers A,B,D

- We only need $P_1$ to achieve 100% statement coverage



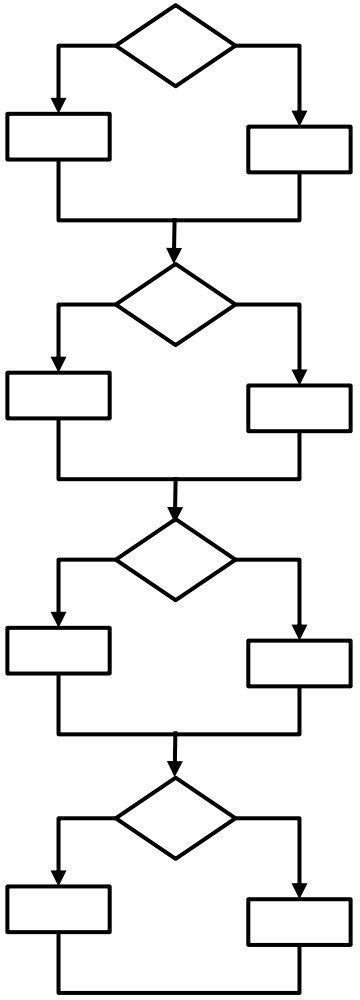
# Statement Testing

- Path coverage is more reliable than statement coverage!
- You might execute all the statements, but not all the paths!
- Executing all the paths will ensure that you executed all the statements too!



# Sequential IF Structures

- Number of unique paths $2^n$
- 24 = 16 logical paths !
- V(G) = 4 + 1 = 5
- 5 are good enough for path coverage
- Only two paths are needed to achieve full statement coverage



# Mutation Testing

# JUnit

- **JUnit** is a unit testing framework for the Java programming language.

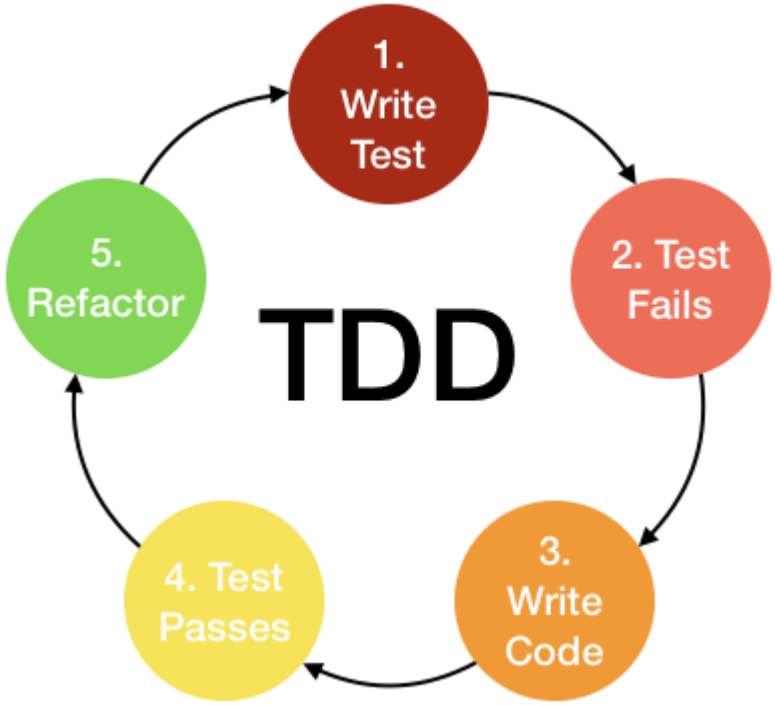
- Please see the code on Moodle.

# Test Driven Development

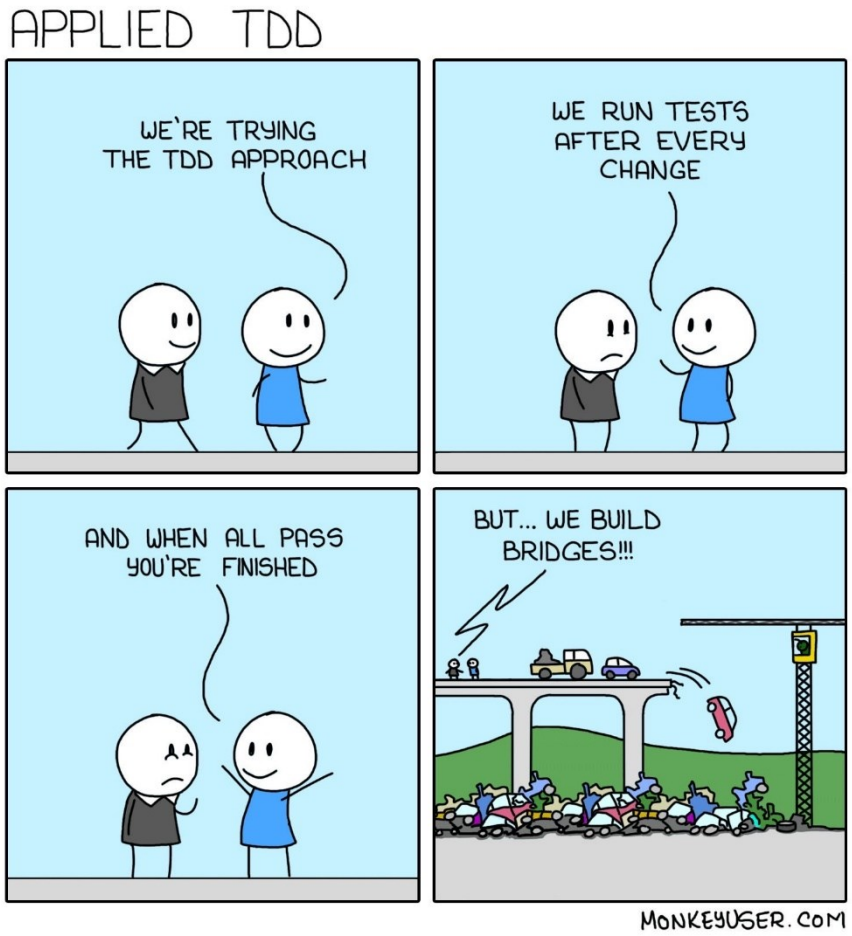
- **Test-driven development** (**TDD**) is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the software is improved so that the tests pass.

# TDD

- 1. Add a test
- 2. Run all tests and see if the new test fails
- 3. Write the code
- 4. Run tests
- 5. Refactor code
- 6. Repeat



# TDD


APPLIED TDD
WE'RE TRYING THE TDD APPROACH
WE RUN TESTS AFTER EVERY CHANGE
AND WHEN ALL PASS YOU'RE FINISHED
BUT... WE BUILD BRIDGES!!!
MONKEYUSER.COM

# Unit Testing at Google

- 150 million test executions per day
- 99% of all test executions pass
- Almost all testing is automated
- No time for Quality Assurance

# Unit Testing at Google

*"You are 97% likely to cause a breakage because you are editing a Java source file modified by 15 other developers in the last 30 days."*

Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming google-scale continuous testing. In Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP '17).

# Previously on CSC 4330