# Com S 417
# Software Testing

Fall 2017 – Week 3, Lecture 6

# Quiz

1. SUT
2. basic block
3. black box
4. integration testing
5. equivalence class
6. stress testing
7. load testing
8. @Before
9. coverage
10. static tests

a. A static test tool.
b. Demonstrates that the SUT can a handle a specific level of load.
c. Runs once per class.
d. Evaluate software without executing it.
e. Runs once per test.
f. A measure of test set quality.
g. Software under test.
h. Attempts to load the SUT until it breaks.
i. (I) Measures the long-term probability of SUT failure.
j. Noise injected in a test.
k. A code segment that executes sequentially
l. (EL) Tests the interoperation of modules
m. Tests designed from requirements only.
n. A subset of an input domain where all members are expected to trigger the same behavior.

# Homework #1 Debrief

**Problem 1**

- 2.9 Consider an application App that takes two inputs name and age, where name is a nonempty string containing at most 20 alphabetic characters and age is an integer that must satisfy the constraint $0 \leq age \leq 120$.
- The App is required to display an error message if the input value provided for age is out of range. The application truncates any name that is more than 20-characters in length and generates an error message if an empty string is supplied for name.
- Partition the input domain using (a) unidimensional partitioning Construct a test-data test for App using the equivalence classes derived in (a).

From Mathur, chapter 2 exercises.

# Homework #1 Debrief

**Problem 2**

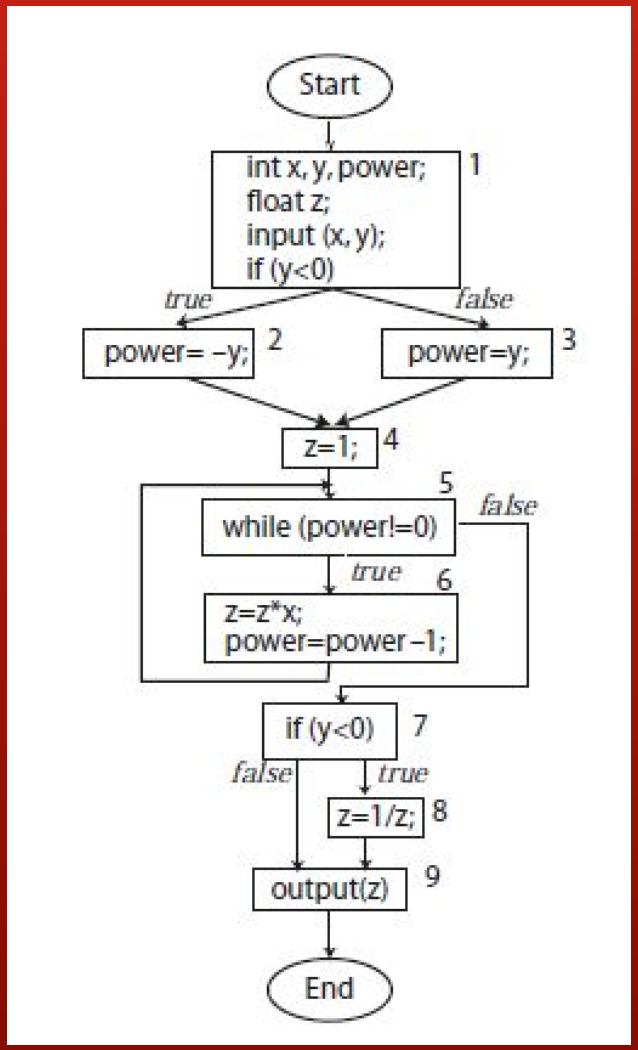
- 2.11 An application takes two inputs x and y where $x \leq y$ and $-5 \leq y \leq 4$. (a) Partition the input domain using unidimensional partitioning. Derive test sets based on the partitions created in (a).

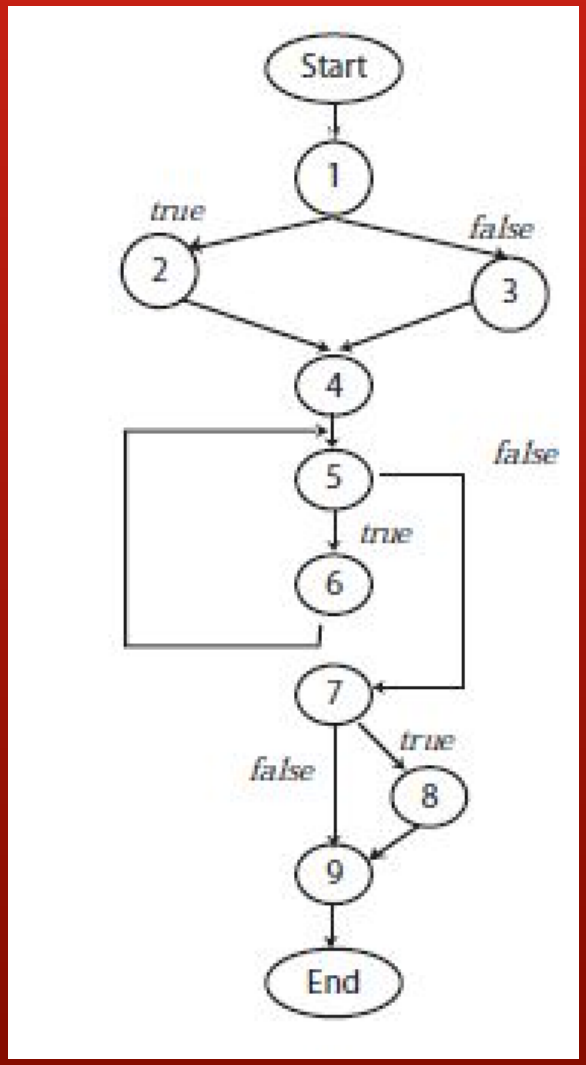
From Mathur, chapter 2 exercises.

# Model-Driven Testing

Selecting tests based on abstractions drawn from the code.

# Reducing detail to coverage related specifics.




Figures from: Mathur

# CFG Details

**Basic Block:** a section of code which will always execute sequentially.

- If any statement in the block executes, then all statements in the block will execute.

Nodes represent basic blocks.

Arcs represent non-sequential control structures; i.e., transitions between basic blocks

- if, switch, while, etc.

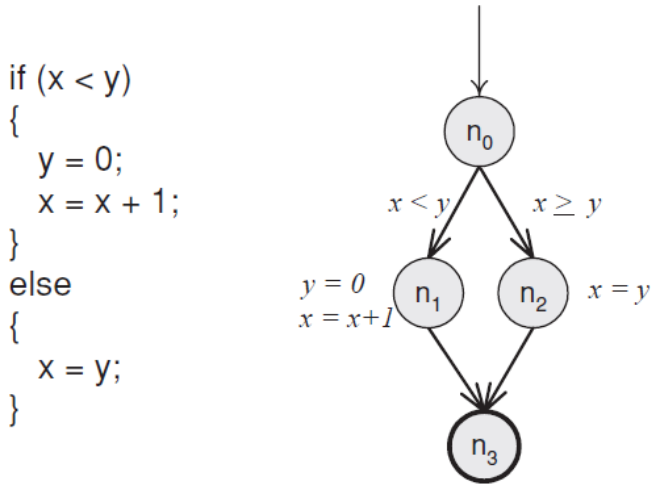



**Figure 2.16.** CFG fragment for the if-else structure.

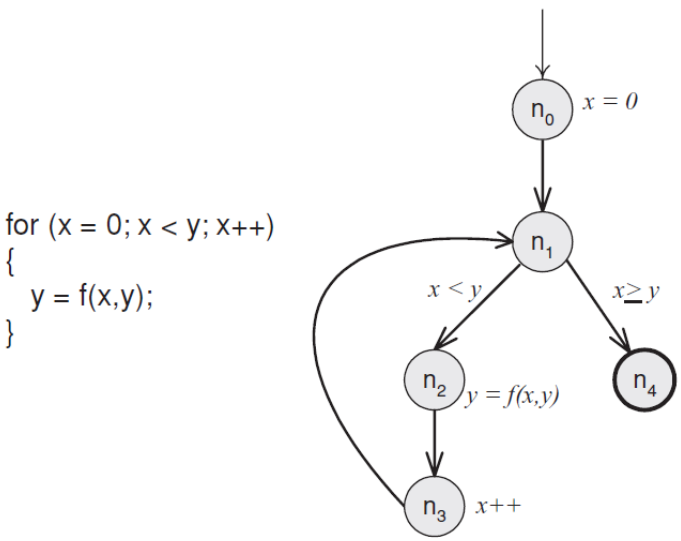



**Figure 2.19.** CFG fragment for the for loop structure.

Figures from Ammann and Offutt

# Notes on Creating CFGs

- Beginning and end of basic blocks is based on execution patterns:
  - if a line/action (x) can be executed directly after any line or action other than the “lexically preceding” line or action, then that line (x) must begin a new basic block.
  - if more than one line/action can (at different times) directly follow the execution of a particular line/action (x), then x must end it’s containing basic block.
- some actions (such as for loop increments) will be mapped to where they execute, instead of where they appear lexically, e.g., the x++ on slide 5

# Graph coverage with Flow Graphs

How do the graphs help us select tests?

You can think of each basic block or node as an “equivalence class” of lines partitioned from the “domain” of the SUT’s executable code.

- Any input that gets you to one of the lines in the block, automatically exercises all of the other lines.
- You can easily focus on inputs that affect the conditions controlling flow.

What rule would you use to select high-coverage tests?

We will describe our coverage criteria in terms of sets of paths with certain properties.

# Path-based selection criteria

- When using graph models, we use paths through the graph to select test items.
- Paths are specified as a sequence of nodes.
- How would you decide which paths to activate?

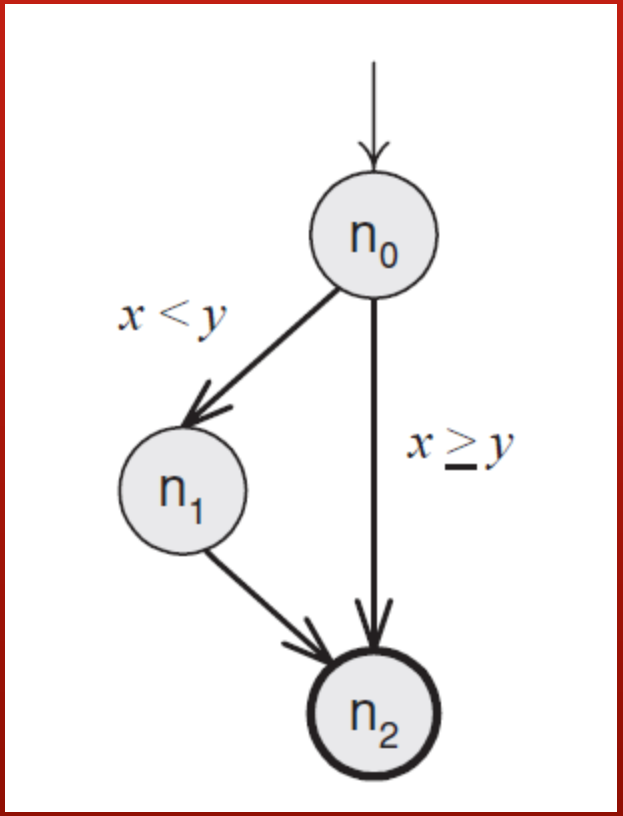
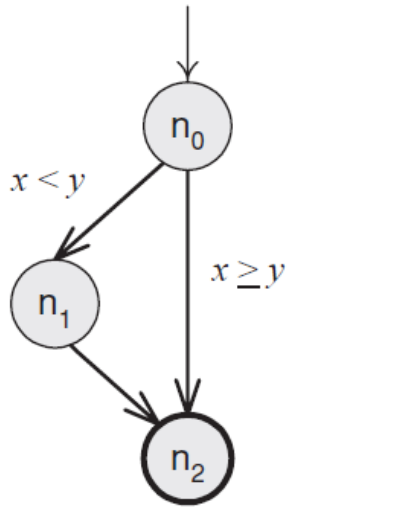



Figure from Amman & Offutt, p 34

# *Node Coverage*

We select test sets that correspond to certain paths which, combined, visit all reachable nodes in the graph.

- simple
- corresponds to 100% branch coverage.




# *Edge Coverage*

We select test sets that traverse all reachable edges in the graph.

- Augmentation of node coverage
- corresponds to a simple form of conditional coverage

$T_1 = \{ t_1 \}$
$T_1$ satisfies node coverage on the graph

$T_2 = \{ t_1 , t_2 \}$
$T_2$ satisfies edge coverage on the graph

Figures from Ammann & Offutt, p. 34

# Path Definition

We can define paths as a sequence of nodes or as a sequence of edges. Mathur starts with edges and then switches to nodes as a “shorthand.”

For a path to be “feasible” (think reachable), there must be at least one test case which causes the path to be traversed.

Why is (start, 1, 2, 4, 5, 7, 9, end) from slide 5 infeasible?

# Example

# Edge-Pair Coverage

Definition:

- The test set contains all reachable paths of length 2 (measured by edge count).

  See Ammann & Offnut (1st Ed) Section 2.4 for graph coverage criteria. (pp 32-39). Available on-line through the library.

Equivalence?

Advantages?

# Questions

Reading, Last Lecture, Lab?

# Reading assignment

https://dev.to/danlebrero/the-tragedy-of-100-code-coverage

- Be sure to read and think about exchange with commenter Mauro Bringolf

- http://blog.cleancoder.com/uncle-bob/2017/03/03/TDD-Harms-Architecture.html
- and watch video at https://www.youtube.com/watch?v=AoIfc5NwRks