Unit Testing Methods

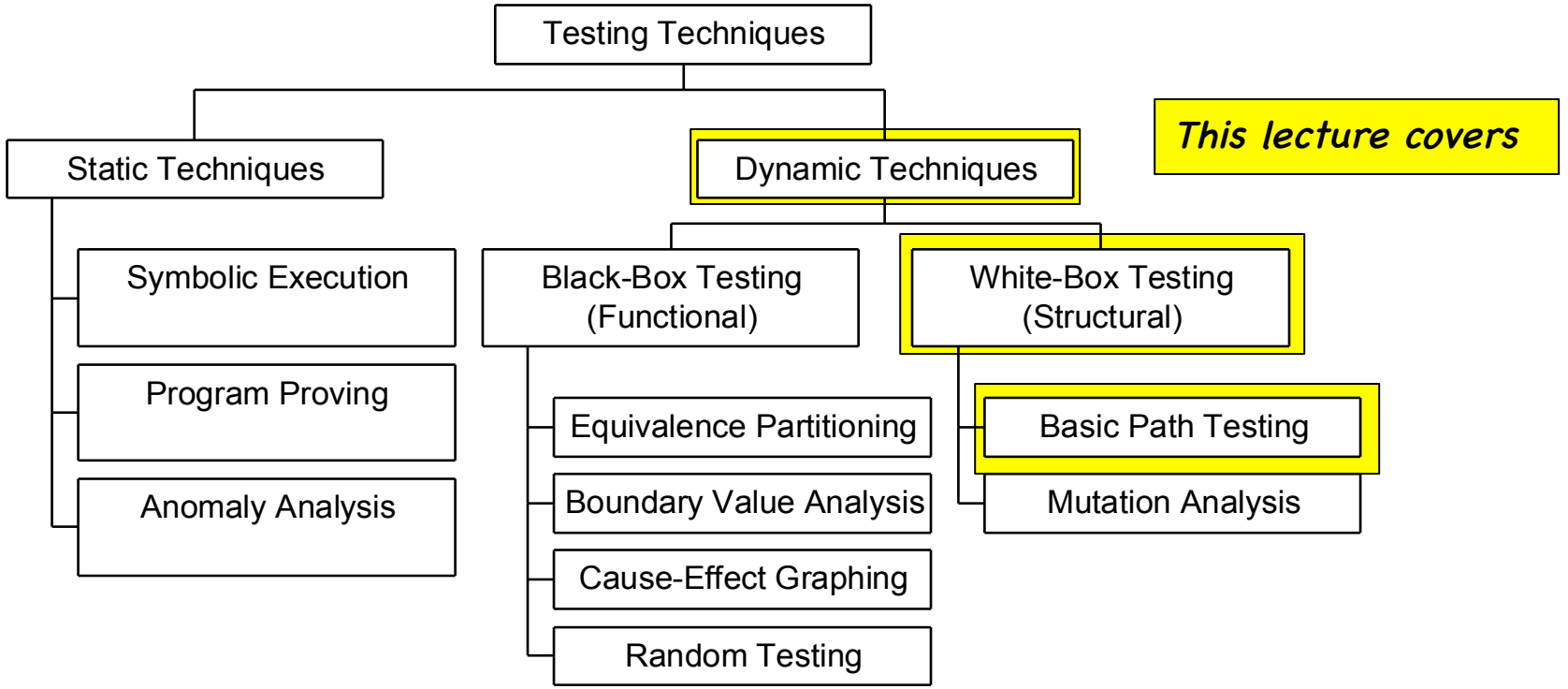
White-box Testing

Code Coverage

Unit Testing Techniques

Unit Testing checks that an individual program unit (subprogram, object class, package, module) behaves correctly.

- Static Testing - testing a unit without executing the unit code
- Dynamic Testing - testing a unit by executing a program unit using test data



Program testing can be used to show the presence of bugs, but never to show their absence [Dijkstra]

Inquiry

Recall Black-box testing:

1. When a test fails, do you know what the bug is?
2. You constructed equivalence partitions and boundary cases – is that enough to know whether any faults exist in the code?
3. What is a software *failure?* A software *fault?* A software *error?*

Reliability Terminology

Failure

- Incorrect or unexpected output
- Symptom of a fault

Fault

- Invalid execution state
- Symptom of an error
- May or may not produce a failure

Error

- Defect or anomaly in source code
- Commonly referred to as a “bug”
- May or may not produce a fault

Defects may be injected at any time in the lifecycle

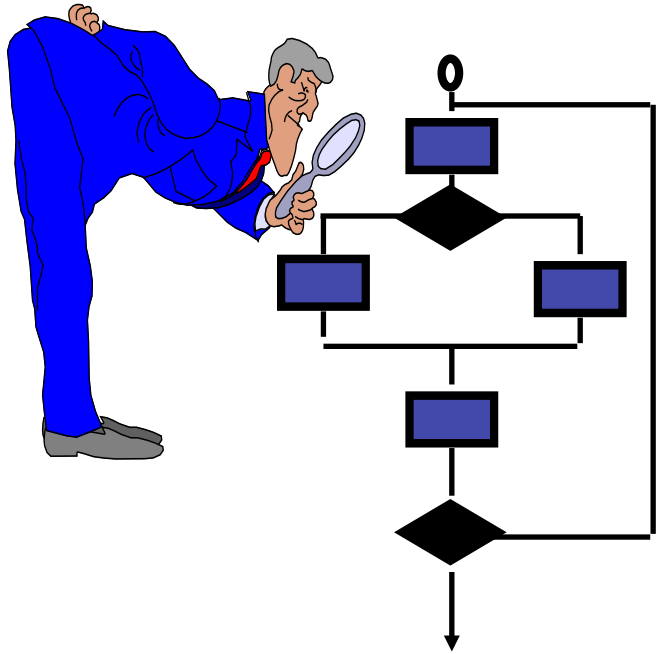
Recall Watts Humphrey (father of PSP):

(paraphrase) *A defect is anything that necessitates a change in the code*

Structural (White-Box) Testing

Test cases designed, selected, and run based on the structure of the source code

- Scale: tests the nitty-gritty (line-by-line)
- Drawbacks: need access to the source



... our goal is to ensure that all statements and conditions have been executed at least once ...

Structural (White-Box) Testing

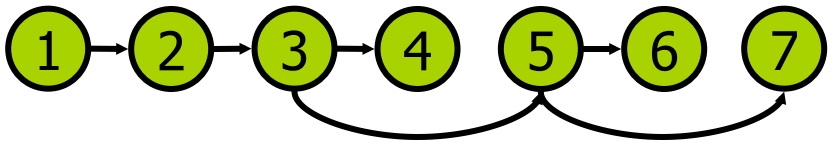
Use source code to derive test cases

- Build a graph model of the system
  - Control flow
  - Data flow

Choose test cases that guarantee types of coverage

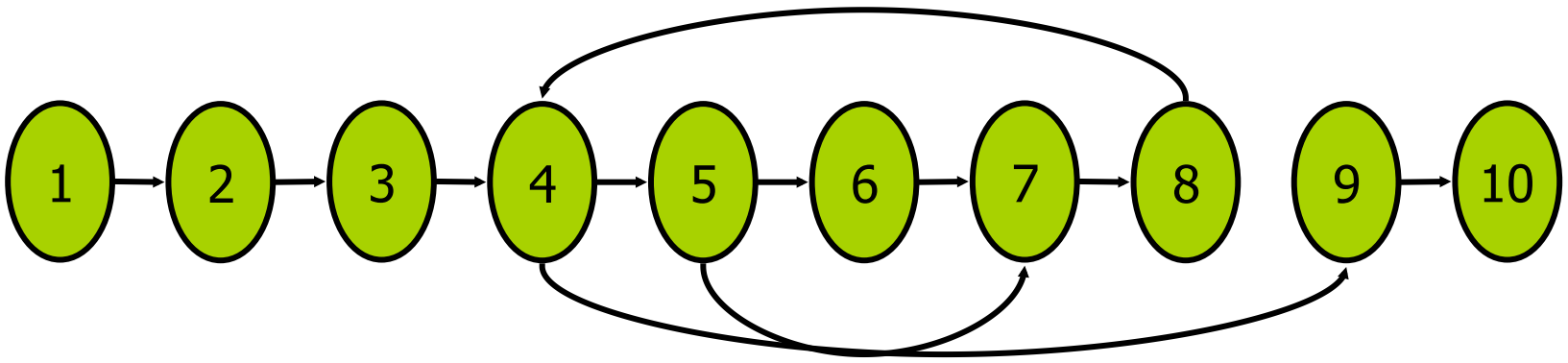
- Node coverage
- Edge coverage
  - Loop coverage
  - Condition coverage
- Path coverage

```
1  Node getSecondElement() {
2      Node head = getHead();
3      if (head == null)
4          return null;
5      if (head.next == null)
6          return null;
7      return head.next.node;
8  }
```



Example

```
1   float homeworkAverage(float[] scores) {
2     float min = 99999;
3     float total = 0;
4     for (int i = 0 ; i < scores.length ; i++) {
5       if (scores[i] < min)
6         min = scores[i];
7       total += scores[i];
8     }
9     total = total – min;
10    return total / (scores.length – 1);
11  }
```



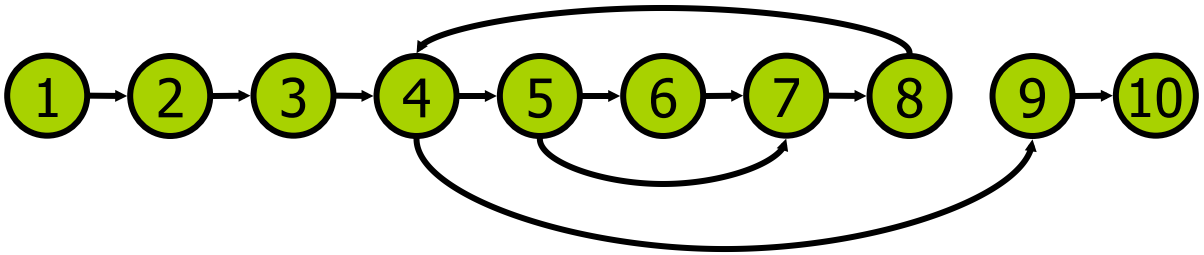
Node Coverage

Select test cases such that every node in the graph is visited

- Also called statement coverage
  - Guarantees that every statement in the source code is executed at least once

Selects minimal number of test cases

Test case: { <1,2,3,4,5,6,7,8,4,9,10> }



Note that a test case is a set {} each element is a sequence <>

Edge Coverage

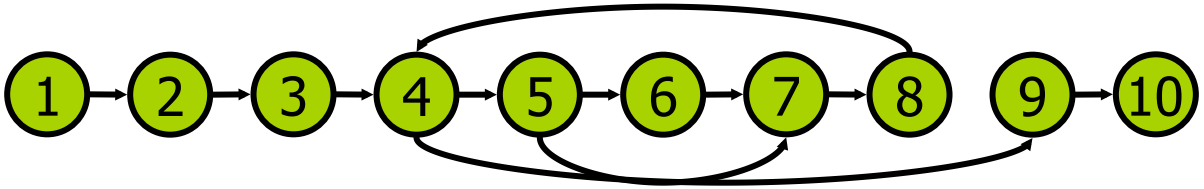
Select test cases such that every edge is visited

- Also called branch coverage
  - Guarantees every branch in the code is executed at least once

More thorough than node coverage

- More likely to reveal logical errors

Test case: { <1,2,3,4,5,6,7,8,4,5,7,8,4,9,10> }



Variations on Edge Coverage

- Loop coverage: ensure loop executes 0, 1, and n times
- Condition coverage: Check all components of a compound conditional
  - e.g. if (x < 100 && x > 50 && !done) – how many tests needed?

Path Coverage

Path coverage

- Select test cases such that every path is traversed
- Loops are a problem
  - Consider example on earlier slide – what is scores.length?
    - Suppose it was 5, how many test cases would you need?

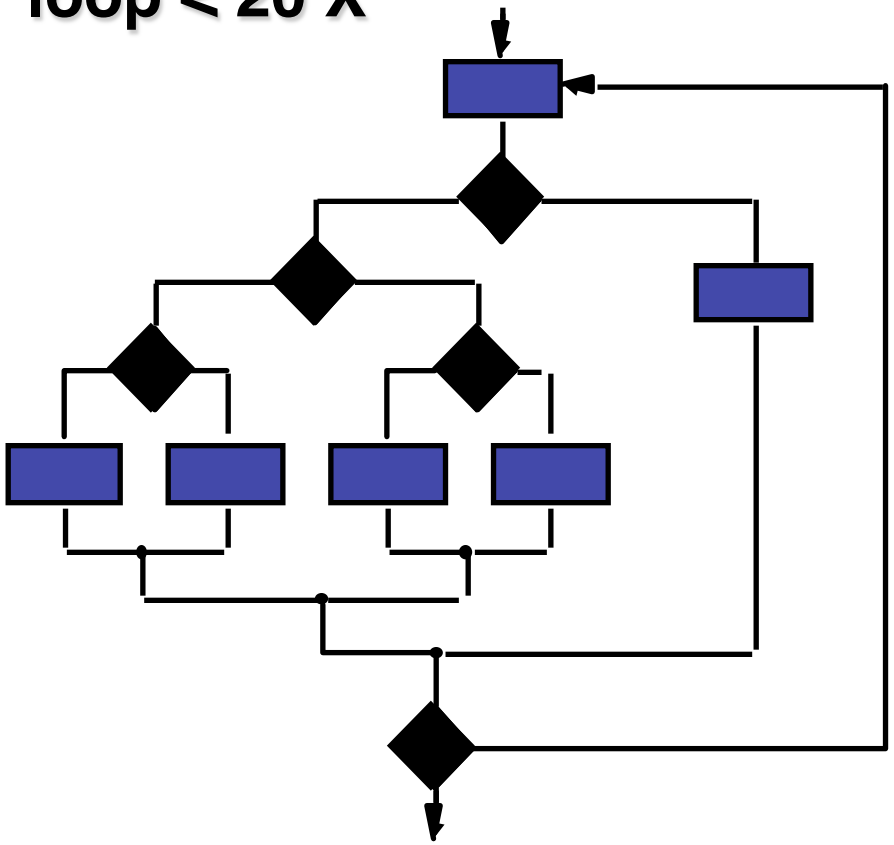
Why are paths important?

- Because previous instructions may cause side effects to runtime state that require you to verify later instructions in light of those side effects

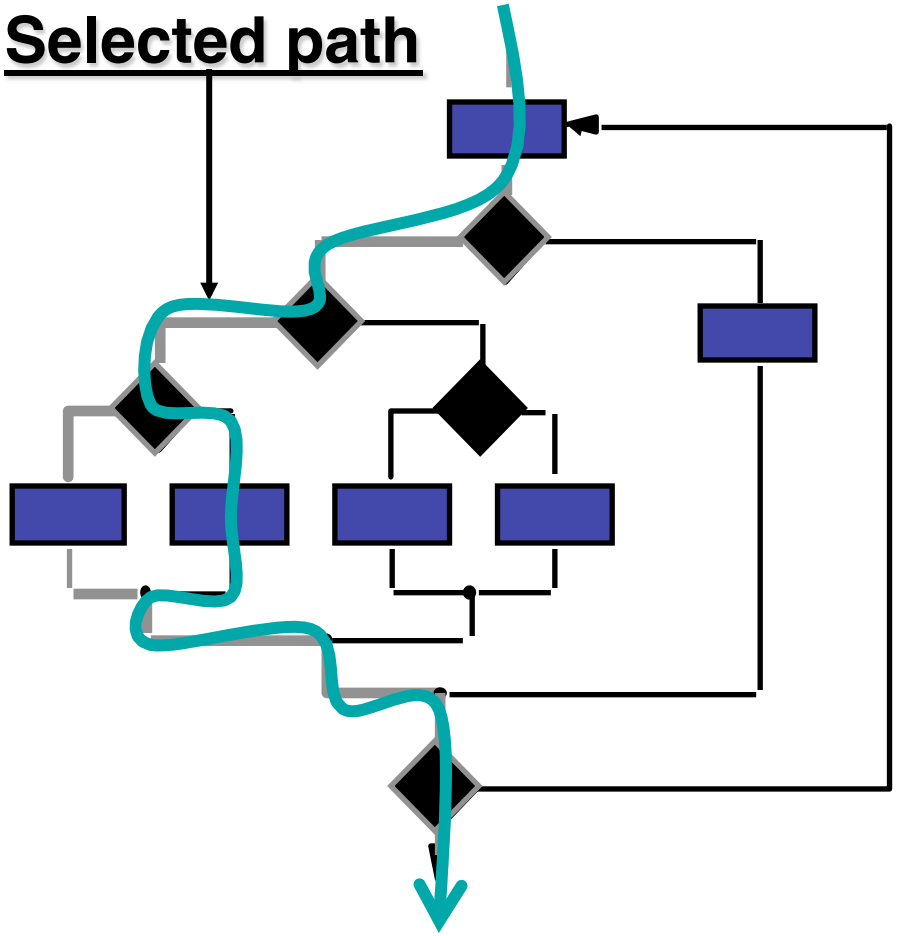
Most thorough……but is it feasible?

Exhaustive vs. Selective Testing

loop < 20 X



Selected path



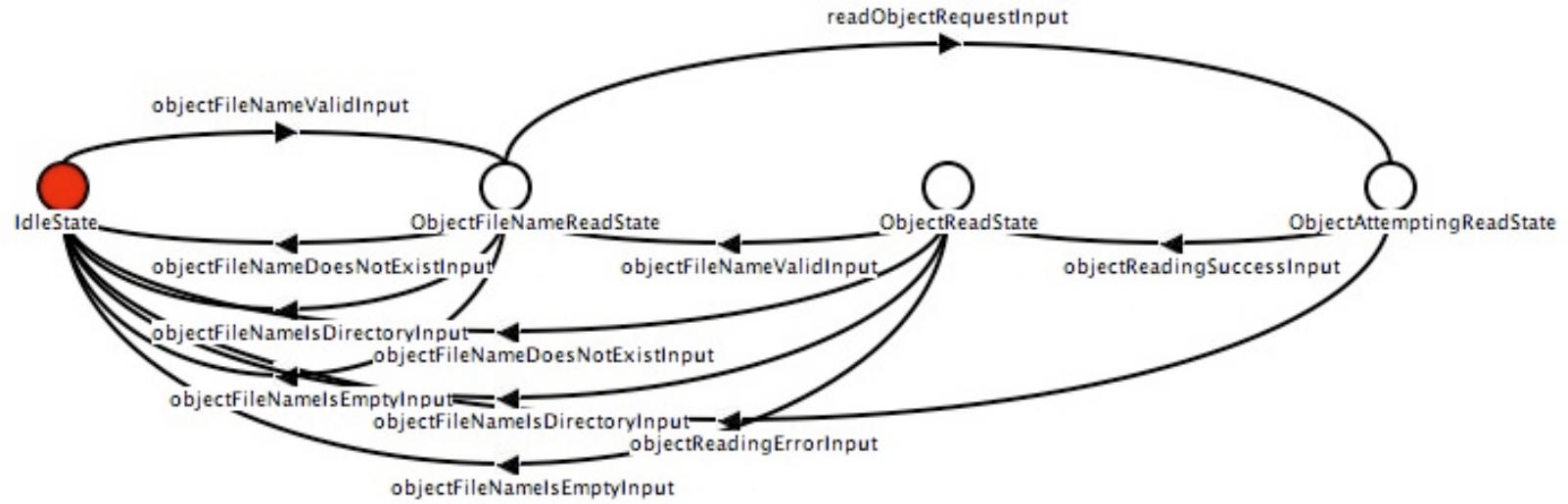
There are 10^{14} possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!

So instead we select paths based on test purpose, heuristics, or from basis paths (more to come…)

Unit Testing: OO Perspective

See your black-box testing notes for the OO Perspective

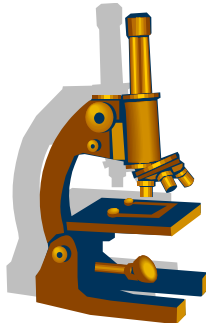
- One approach considers the statechart of the object, and develops unit tests based on the valid set of states an object goes through
- We can consider this white-box in that what we want to exercise all valid sequences of states the object can go through in (ab)normal operation.
- Coverage in this sense becomes
  - "did the test visit all the states of the statechart?" (Node coverage)
  - "did it visit all of the transitions?" (Edge coverage)
  - "did it visit all of the possible sequences?" (Path coverage)
  - "did it attempt invalid transitions?" (Negative testing)



Summary: Unit Testing

White-box

- Structural evaluation
- Coverage difficult – set a target!



Black-box

- Treats implementation of function as “unknown”
- Test for valid outputs given a range of inputs
- Science based on domain/range of the inputs

Unit-level testing process

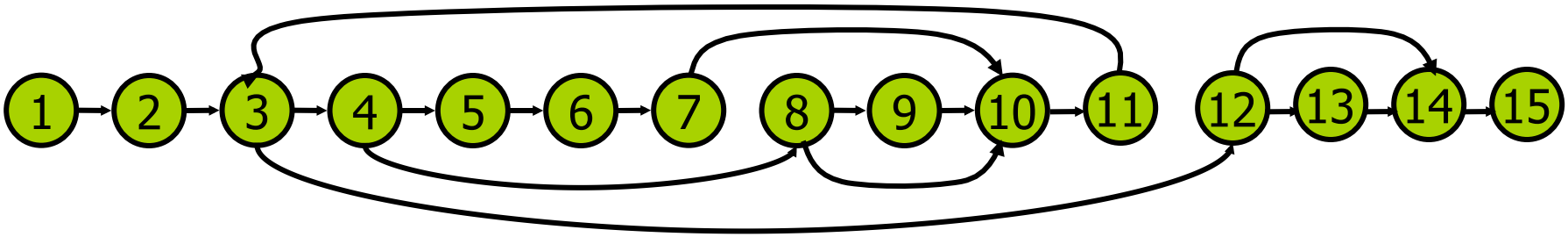
- Unit testing now considered a very agile way of coding.
  - Automate Automate Automate
- TDD a great way to ensure you verify & validate as you go.
- Many developers struggle to write complete unit tests.
- Many developers struggle to maintain their unit tests.

Another Example (for practice)

```
1  float findAverage(float[] scores) {
2    float total = 0, min = MAX_FLOAT, min2 = MAX_FLOAT;
3    for (int i = 0 ; i < scores.length ; i++) {
4      if (scores[i] < min)   {
5        min2 = min;
6        min = scores[i];
7      }
8      else if (scores[i] < min2)
9        min2 = scores[i];
10     total += scores[i];
11   }
12   if (min != MAX_FLOAT && min2 != MAX_FLOAT)
13     output(min, min2);
14   return total / scores.length;
15 }
```

First, draw the control flow graph for this code

Another example: control-flow graph



Next, determine node and edge coverage test cases

Another Example: Coverage

Node coverage:

Test case: {<1,2,3,4,5,6,7,10,11,3,4,8,9,10,11,3,12,13,14,15>}

Edge Coverage:

Test cases: {<1,2,3,4,5,6,7,10,11,3,12,14,15>,
<1,2,3,4,8,9,10,11,3,4,8,10,11,3,12,13,14,15>}