Software Testing, Quality Assurance & Maintenance—Lecture 3

Patrick Lam
University of Waterloo

January 13, 2025

Plan

More on testing.

Then, fuzzing.

Later this week, shifting gears:
operational semantics!

Part I

When to stop?
Idea 1: Coverage

How many tests?

Do you have enough test? How do you know?

Test all inputs?

State-of-the-industry: code coverage—

statement coverage, branch coverage.

Side note: white-box and black-box

When you write tests:

White-box testing: you can look at the code;

Black-box testing: you can't look at the code.

Control-Flow Graphs

Mostly people use lines of source code to evaluate coverage, but then your coverage depends on newlines.

We are sometimes going to be more precise and use control-flow graphs.

CFG nodes and edges

Nodes: a node represents 0 or more statements;

Edges: an edge (s_1, s_2) indicates that s_1 may be followed by s_2 in an execution.

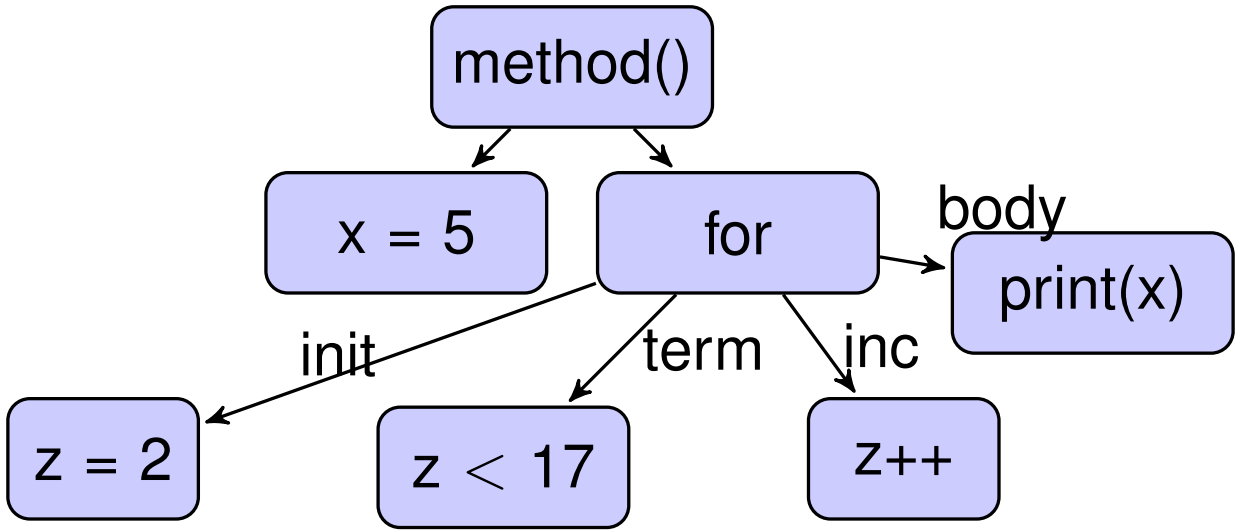
Example: code for CFG

```
x = 5;
for (z = 2; z < 17; z++)
   print(x);
```

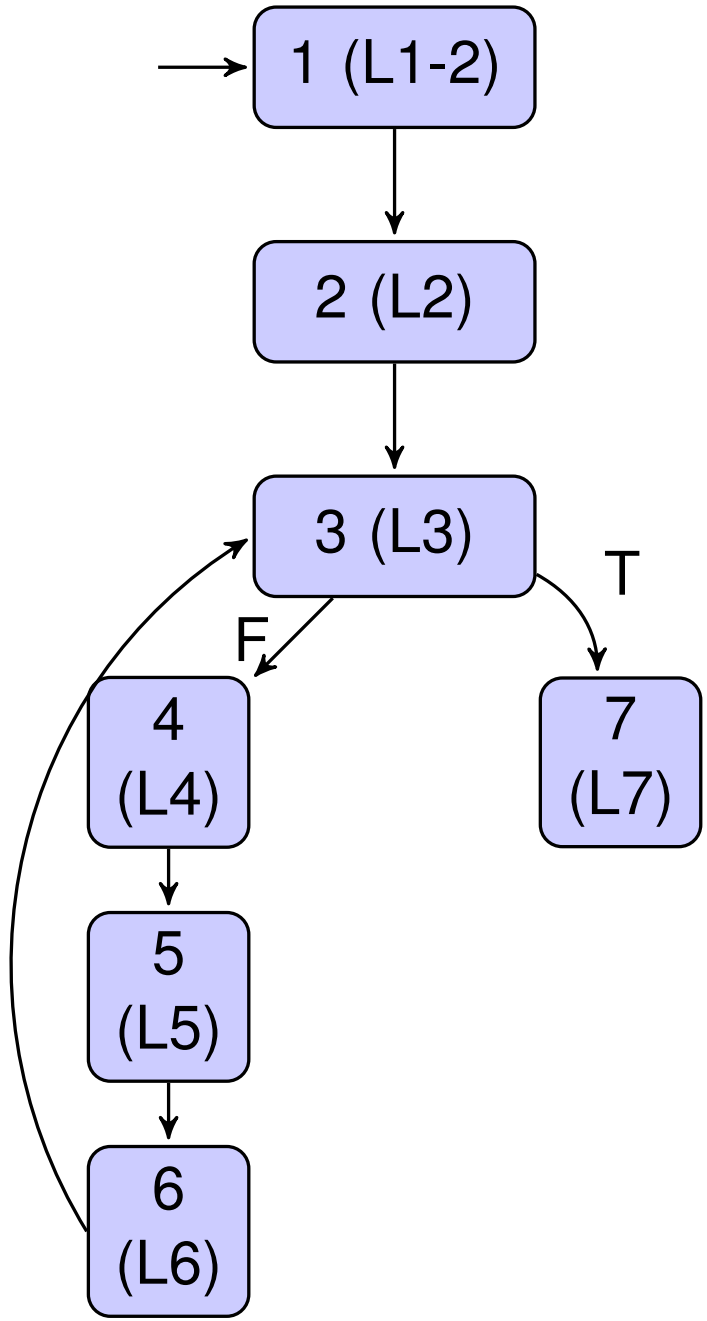
Reminder: how we compile

- lexing:
  stream of characters $\rightarrow$ stream of tokens (if, while, strings)
- parsing:
  stream of tokens $\rightarrow$ concrete syntax tree
- construction of Abstract Syntax Tree (AST):
  cleans up the concrete syntax tree
- conversion to Control Flow Graph:
  AST $\rightarrow$ CFG
- optimizations
  CFG $\rightarrow$ CFG
- convert to bytecode/machine code
  CFG $\rightarrow$ bytecode/machine code

Abstract Syntax Tree for Example



To a Control-Flow Graph

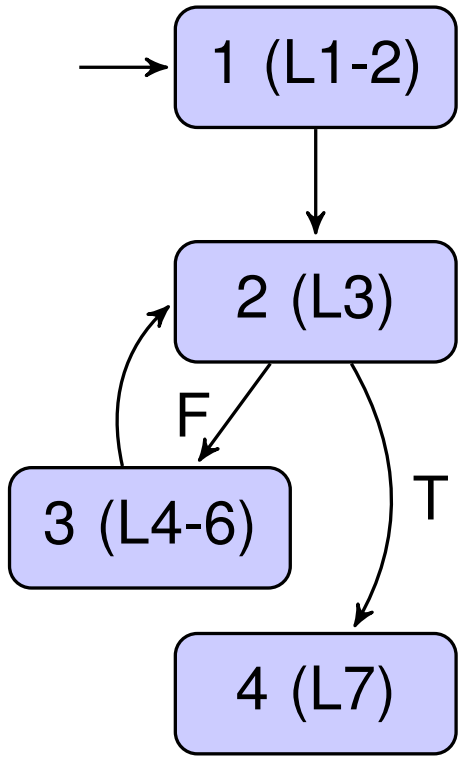


Low-level Code

```
     x = 5
     z = 2
q0:  if (z < 17) goto q1
     z = z + 1
     print (x)
     goto q0
q1:  nop
```

Basic Blocks

Group together statements which always execute together (in sequential programs):



Basic Block Definition

A basic block is a sequence of instructions in the control-flow graph that has one entry point and one exit point.

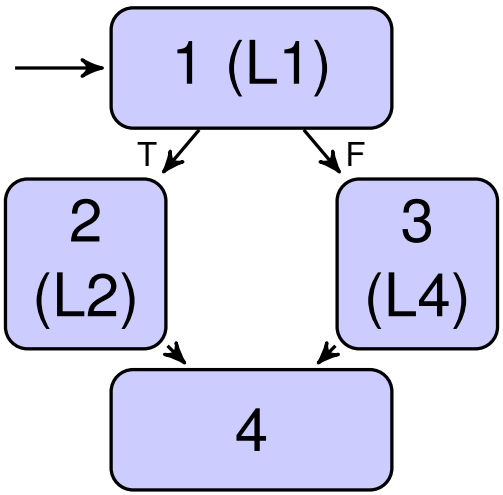
Usually want maximal basic blocks.

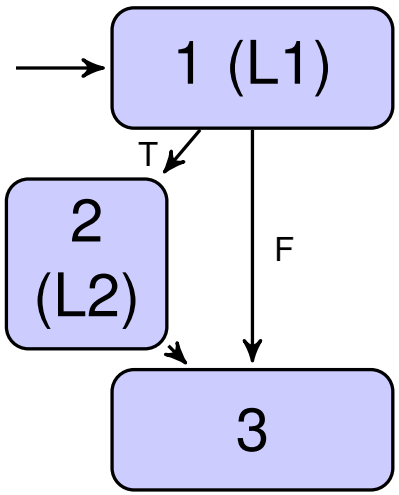
May have multiple successors.

No jumps into the middle of a basic block.

Constructing Basic Blocks: if

```
if (z < 17)
  print (x);
else
  print (y);
```



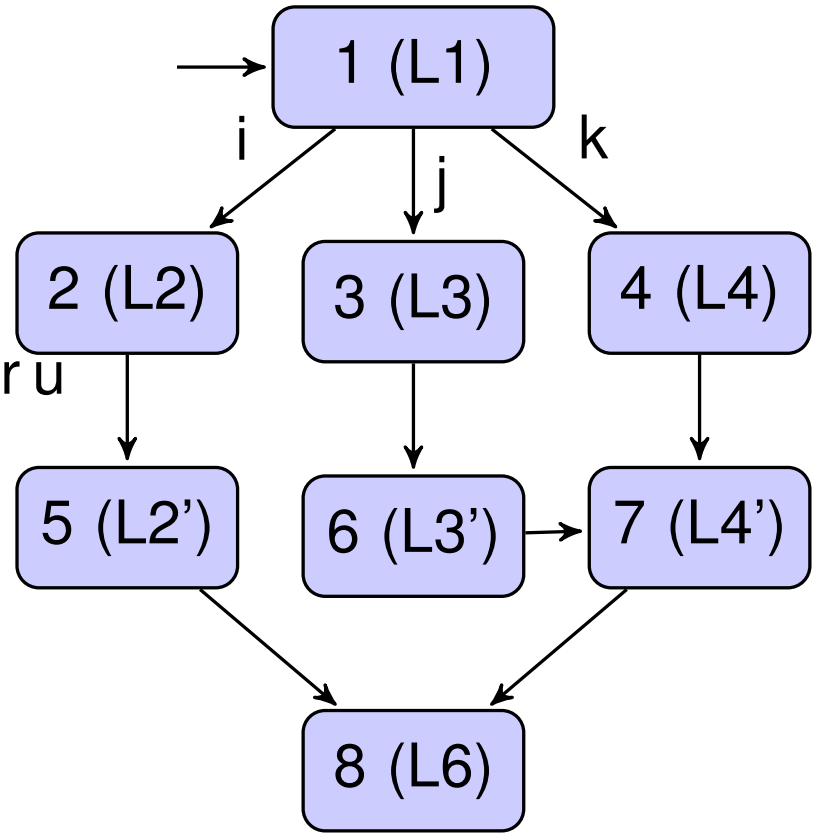



```
if (z < 17)
  print(x);
```

Not talking about short-circuit evaluation.

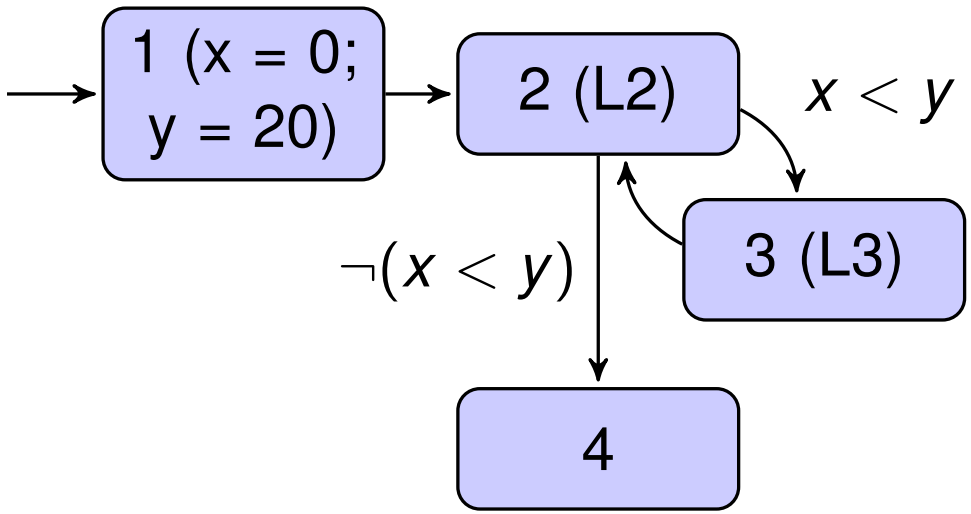
Constructing Basic Blocks: case/switch

```
switch (n) {
  case 'I': ...; break;
  case 'J': ...; //fallthru
  case 'K': ...; break;
}
// ...
```



Constructing Basic Blocks: while

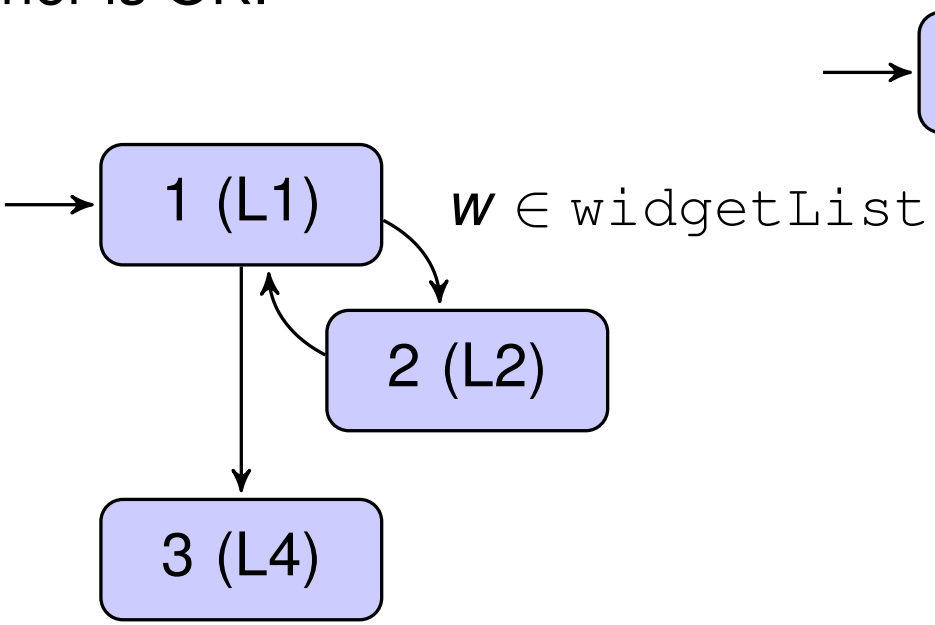
```
x = 0; y = 20;
while (x < y) {
  x ++; y --;
}
```

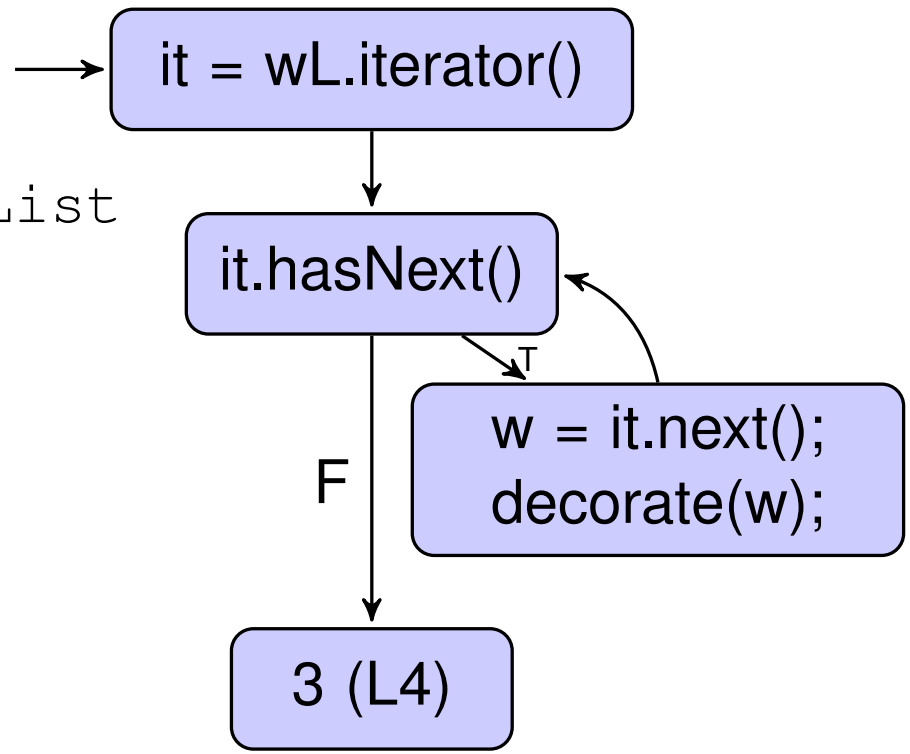


Constructing Basic Blocks: for

```
for (Widget w : widgetList) {
  decorate(w);
}
```

Either is OK:





Back to Statement and Branch Coverage

Given a test suite and a program,
instrument the program to:

- count whether each statement (CFG node) is executed;
- count whether each branch (CFG edge) is taken.

Statement coverage is the fraction of statements (nodes) that are executed by the test suite.

Branch coverage is the fraction of branches (edges) that are executed.

Example Code

```
class Foo:
    def m(self, a, b):
        if a < 0 and b < 0:
            return 4
        elif a < 0 and b > 0:
            return 3
        elif a > 0 and b < 0:
            return 2
        elif a >= 0 and b >= 0:
            return a/b
        raise Exception("I didn't think things through
```

Example Test Suite

```
import unittest

from .foo import Foo

class CoverageTests(unittest.TestCase):
    def test_one(self):
        f = Foo()
        f.m(1, 2)

    def test_two(self):
        f = Foo()
        f.m(1, -2)

    def test_three(self):
        f = Foo()
        f.m(-1, 2)
```

Coverage Report

```
Name                    Stmts   Miss  Branch  BrPart  Cover   Missi
-------------------------------------------------------------------
l03/foo.py                 11      2       8       2    79%   4, 11
l03/test_suite.py          12      0       0       0   100%
-------------------------------------------------------------------
TOTAL                     124     98      46       2    21%
```

HTML report also available.

On Coverage

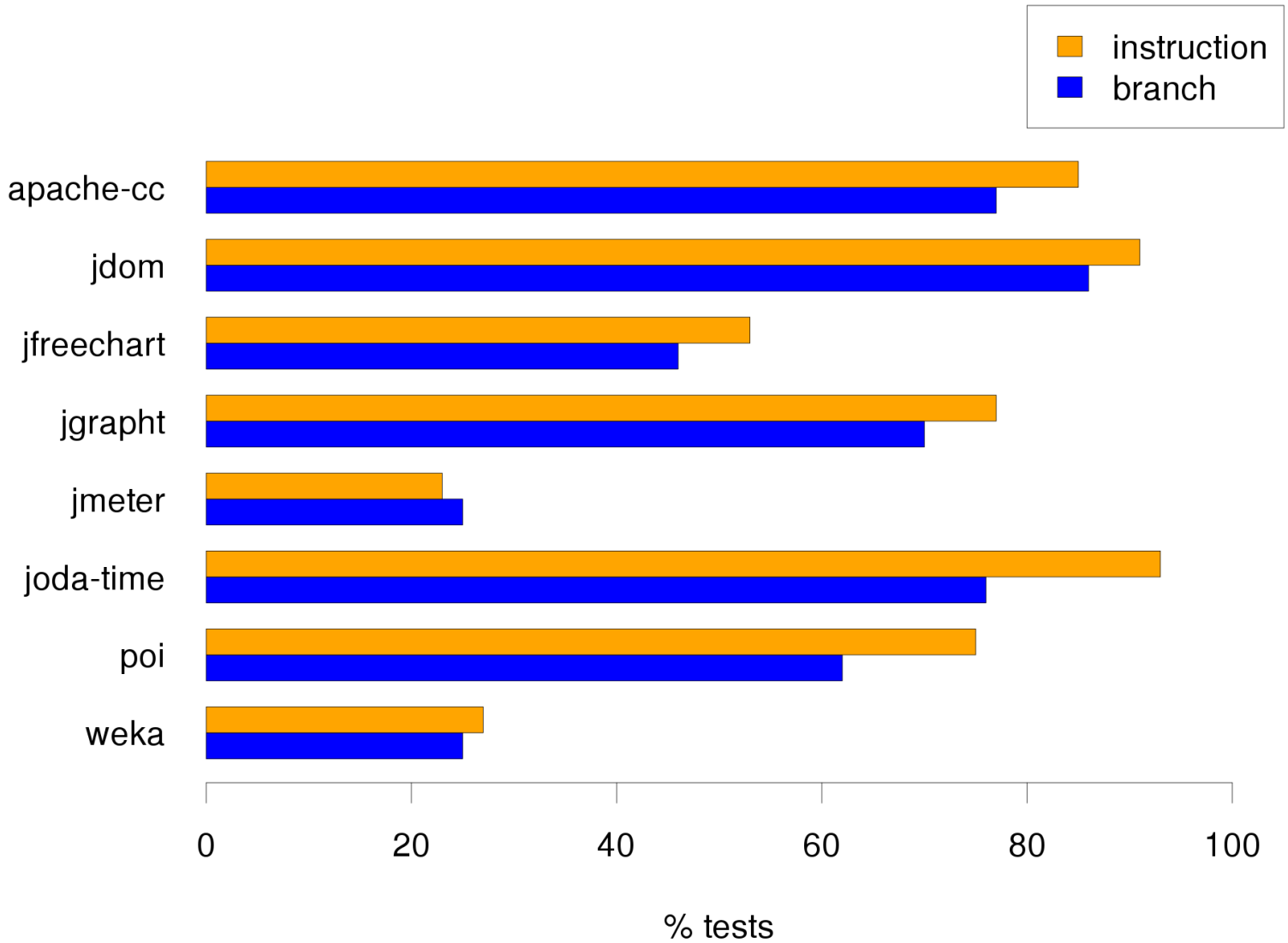
Can add missing test cases to visit all lines.

Even with 100% branch coverage,
one is missing an important behaviour: what if `b` is 0?

Infeasible Test Requirements

Infeasible to reach 100% coverage on real programs.
How much is enough, and why is there a gap?

Some Real Coverage Data



Case Study: JUnit (4.11) the Artifact

https://avandeursen.com/2012/12/21/line-coverage-lessons-from-junit/

JUnit Measurements

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	221	84%	2970/3513	81%	859/1060	1.727
junit.extensions	6	82%	52/63	87%	7/8	1.25
junit.framework	17	76%	399/525	90%	139/154	1.605
junit.runner	3	49%	77/155	41%	23/56	2.225
junit.textui	2	76%	99/130	76%	23/30	1.686
org.junit	14	85%	196/230	75%	68/90	1.655
org.junit.experimental	2	91%	21/23	83%	5/6	1.5
org.junit.experimental.categories	5	100%	67/67	100%	44/44	3.357
org.junit.experimental.max	8	85%	92/108	86%	26/30	1.969
org.junit.experimental.results	6	92%	37/40	87%	7/8	1.222
org.junit.experimental.runners	1	100%	2/2	N/A	N/A	1
org.junit.experimental.theories	14	96%	119/123	88%	37/42	1.674
org.junit.experimental.theories.internal	5	88%	98/111	92%	39/42	2.29
org.junit.experimental.theories.suppliers	2	100%	7/7	100%	2/2	2
org.junit.internal	11	94%	149/157	94%	53/56	1.947
org.junit.internal.builders	8	98%	57/58	92%	13/14	2
org.junit.internal.matchers	4	75%	40/53	0%	0/18	1.391
org.junit.internal.requests	3	96%	27/28	100%	2/2	1.429
org.junit.internal.runners	18	73%	306/415	63%	82/130	2.155
org.junit.internal.runners.model	3	100%	26/26	100%	4/4	1.5
org.junit.internal.runners.rules	1	100%	35/35	100%	20/20	2.111
org.junit.internal.runners.statements	7	97%	92/94	100%	14/14	2
org.junit.matchers	1	9%	1/11	N/A	N/A	1
org.junit.rules	20	89%	203/226	96%	31/32	1.444
org.junit.runner	12	93%	150/161	88%	30/34	1.378
org.junit.runner.manipulation	9	85%	36/42	77%	14/18	1.632
org.junit.runner.notification	12	100%	98/98	100%	8/8	1.162
org.junit.runners	16	98%	321/327	96%	95/98	1.737
org.junit.runners.model	11	82%	163/198	73%	73/100	1.918

Report generated by Cobertura 1.9.4.1 on 12/22/12 2:25 PM.

JUnit Stats

Overall instruction coverage: 85%.

13,000 lines of code, 15,000 lines of test.

Consistent with industry average.

What's not covered? Deprecation

- deprecated code: 65% instruction coverage
- nondeprecated code: 93% instruction coverage

- newer code (in `org.junit.*`): 90% instruction coverage
- older code (in `junit.*`): 70% instruction coverage

(Why is this? Perhaps the coverage decreased over time for the deprecated code, since no one is really maintaining it anymore, and failing test cases just get removed.)

A Whole Untested Class

Blogpost author found one class that was completely untested!

There were tests.
But the tests never got run, because they were never added to CI.

They also failed when run. (You don't run it, it doesn't work.)

The Usual Suspects 1: Too Simple to Test

```
public static void assumeFalse(boolean b) {
  assumeTrue(!b);
}

/**
* Override to set up your specific external reso
*
* @throws if setup fails (which will disable {@c
*/
protected void before() throws Throwable {
  // do nothing
}
```

The Usual Suspects 2: Dead by Design

```
/**
* Protect constructor since it is a static only
*/
protected Assert() { }

// should never be executed:
catch (InitializationError e) {
  throw new RuntimeException(
  "Bug in saff's brain: " +
  "Suite constructor, called as above, should alw
}

// unreachable
try {
  ...
} catch (InitializationError e) {
  return new ErrorReportingRunner(null, e); // u
}
```

Thoughts on JUnit Coverage

JUnit: written by people who care about testing.

Non-deprecated code: 93% instruction coverage,
i.e. $\leq$ 2–3 untested lines of code per method.

Probably OK to have lower coverage for deprecated code.

Don't forget that what is in the tests matters too!