

# **Software Testing, Quality Assurance & Maintenance—Lecture 3**

Patrick Lam  
University of Waterloo

January 12, 2026

# Kinds of Assertions

Three built-in choices:

- 1 `assertTrue(aBooleanExpression)`
- 2 `assertEquals(expected, actual)`
- 3 `assertEquals(expected, actual, tolerance)`

note: `assertTrue` can give  
hard-to-diagnose error messages  
(must try harder when using).

# Using Assertions

Assertions are good:

- to check all things that should be true  
(more = better)
- to serve as documentation:  
when system in state  $S_1$ ,  
and I do  $X$ ,  
assert that the result should be  $R$ , and  
that system should be in  $S_2$ .
- to allow failure diagnosis  
(include assertion messages!)

# Not Using Assertions

Can also do external result verification:

Write output to files, diff (or custom diff) expected and actual output.

Twist: expected result then not visible when looking at test.

(What's a good workaround?)

# Verifying Behaviour

Observe actions (calls) of the SUT.

- procedural behaviour verification; or,  
(challenge: recording & verifying  
behaviour)
- via expected behaviour specification.  
(also captures outbound calls of SUT)

# Part I

## Mock Objects

A black and white line drawing illustration. On the left, a Gryphon is shown from the chest up, with its large, feathered wings spread. In the center, a young girl (Alice) is sitting, looking towards the right. On the right, a Mock Turtle is standing, facing Alice. The Mock Turtle has a long neck, a small head with a single eye, and a large, patterned shell. The background consists of simple horizontal lines representing a landscape or sky.

John Tenniel's original (1865) illustration for Lewis Carroll's "Alice in Wonderland". Alice sitting between Gryphon and Mock turtle.

# Things like mocks

- **dummy objects**: don't do anything; are like `null` or `None`, but pass nullness checks.
- **fake objects**: have actual (correct) behaviour, but unsuitable for production, e.g. in-memory db.
- **stubs**: produce canned answers.
- **mocks**: produce canned answers, but also check the interaction protocol.

## Mock Objects: Email Sender

```
// state or behaviour verification?  
public class MailServiceFake  
    implements MailService {  
    private List<Message> messages =  
        new ArrayList<Message>();  
  
    public void send (Message msg) {  
        messages.add(msg);  
    }  
    public int numberSent() {  
        return messages.size();  
    }  
}
```



# Mock Objects: Behaviour Verification

```
// jMock syntax
class OrderInteractionTester... {
    public void testOrderSendsMailIfUnfilled() {
        Order order = new Order(TALISKER, 51);
        Mock warehouse = mock(Warehouse.class); // (1)
        Mock mailer = mock(MailService.class);
        order.setMailer((MailService) mailer.proxy());

        mailer.expects(once()).method("send"); // (2)
        warehouse.expects(once()).method("hasInventory")
            .withAnyArguments()
            .will(returnValue(false));

        order.fill((Warehouse) warehouse.proxy());
    }
}
```

# Mock Objects: Document Management

```
// EasyMock syntax
@RunWith(EasyMockRunner.class)
public class ExampleTest {
    @TestSubject
    private ClassUnderTest classUnderTest =
        new ClassUnderTest();

    @Mock // creates a mock object
    private Collaborator mock;

    @Test
    public void testRemoveNonExistingDocument() {
        replay(mock);
        classUnderTest.removeDocument
            ("Does not exist");
    }
}
```

# Mock Objects: Using a Mock

```
@Test
public void testAddDocument() {
    // ** recording phase **
    // expect document addition
    mock.documentAdded("Document");
    // expect to be asked to vote for document removal,
    expect(mock.voteForRemoval("Document"))
        .andReturn((byte) 42);
    // expect document removal
    mock.documentRemoved("Document");
    replay(mock);
    // ** replaying phase **
    // we expect the recorded actions to happen
    classUnderTest.addDocument("New Document",
        new byte[0]);
    // check that the behaviour actually happened:
    verify(mock);
}
```

## Part II

# **Flakiness: Good for croissants, bad for tests**

(thanks Pixabay for the picture)

# Dealing with Flakiness

There are mitigations:

- label known-flaky tests and rerun them
- ignore or remove flaky tests

They're not great.

Takes a long time to re-run tests.

# Causes of Flakiness

Luo et al studied 201 fixes to flaky tests in open-source projects. Fixable causes:

- improper waits for asynchronous responses;
- concurrency; and
- test order dependency.

# Asynchronous waits

Typically: a `sleep()` call which didn't wait long enough.

Best practice:

- use some sort of `wait()` call to wait for the result, instead of hardcoding a sleep time.

# Concurrency

The usual (see CS343 for more):

- data races;
- atomicity violations;
- deadlocks.



## Test order dependency

- Test A expects test B to have already executed  
(and to have left a side effect, e.g. a file).


Especially common in the transition from Java 6 to Java 7, because the test execution order changed.

Solution: remove the dependency.

# Plan

More on testing (when to stop?).

Then, mutation analysis (are your tests actually good?).



## Part III

### **When to stop?** **Idea 1: Coverage**

## How many tests?

Do you have enough tests? 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.

It is possible to be more precise and use **control-flow graphs**, but we won't, this time.

# 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	Missing
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?

# Collecting Coverage Information



How to collect coverage info? Different in different languages.

- C: recompile with instrumentation
- Java: virtual machine can collect
- Python: provides hooks

# Tracking Python Coverage in Python

Use `sys.settrace(f)` to register `f` as a tracing function.  
You can find this tracing function in `code/L03/tracing.py`:

```
def traceit(frame: FrameType, event: str, arg: Any)
    -> Optional[Callable]:
    if event == 'line':
        global coverage
        function_name = frame.f_code.co_name
        lineno = frame.f_lineno
        coverage.append(lineno)

    return traceit
```

It records the lines that have been executed in variable `coverage`.

## Example: using traceit

There is also a function `cgi_decode()` from The Fuzzing Book which I've included in the repo.

We can activate tracing as follows:

```
def cgi_decode_traced(s: str) -> None:  
    global coverage  
    coverage = []  
    sys.settrace(traceit)  # tracing on  
    cgi_decode(s)  
    sys.settrace(None)    # tracing off
```

and if we call it and print out the coverage, we get something like:

```
>>> cgi_decode_traced("a+b")  
>>> print(coverage)  
[12, 13, 12, 13, 12, 13, 12, 13, 12, 13, 12, 14, 12, 14, 12]
```

# Python trick: using with

Cleaner:

```
with Coverage() as cov:  
    function_to_be_traced()  
c = cov.coverage()
```

Also, Python introspection can retrieve source code and pretty-print it, highlighting non-covered lines.

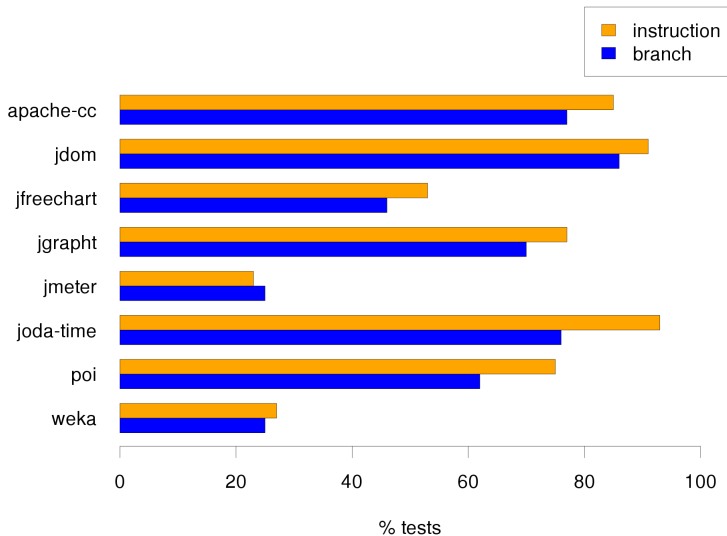
This allows you to write tests to cover these lines.

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

```
 *
```

```
 * @throws if setup fails (which will disable {@code
```

```
 */
```

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



# 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!