Software Testing, Quality Assurance and Maintenance Winter 2026

Lecture 3 — January 12, 2026

Patrick Lam *version 2*

Assertions

We build tests using assertions. In JUnit, there are three basic built-in choices:

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

(There are others too, but let's start with these.)

`assertTrue` is more flexible, since you can write anything with a boolean value. However, it can give hard-to-diagnose error messages—you need try harder when using it if you want good tests.

Using Assertions. Why use assertions? Assertions are good for:

- checking all the things that should be true (more = better);
- serving 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.
- allowing failure diagnosis (include assertion messages!)

There are alternatives to using assertions. For instance, one can also do external result verification:

- write output to files; and
- use diff (or your own custom diff) to compare expected and actual output.

The twist is that the expected result is then not visible when looking at test's source code. (What's a good workaround?)

Verifying Behaviour. The key is to observe actions (calls) of the SUT. Some options:

- procedural behaviour verification (the challenge in that case: recording and verifying behaviour); or
- expected behaviour specification (capturing the outbound calls of the SUT).

Representation invariants

We said that assertions are good for checking all the things that should be true. In particular, they can help with checking the integrity of complex data structures; in such a case, we can aim to write out all the things that should be true of the data structure. Let's look at examples from the Fuzzing chapter of the Fuzzing Book[1]. The example isn't really a complex data structure, but it does have some expected properties. Your typical CS341 data structure is going to have more complex properties, but they are harder to illustrate.

```
1  airport_codes: dict[str, str] = {
2      "YVR": "Vancouver",
3      "JFK": "New York-JFK",
4      "CDG": "Paris-Charles de Gaulle",
5      "CAI": "Cairo",
6      "LED": "St. Petersburg",
7      "PEK": "Beijing",
8      "HND": "Tokyo-Haneda",
9      "AKL": "Auckland"
10 }   # plus many more
```

We can introduce a *representation invariant* for this `dict`:

```
1  def code_repOK(code: str) -> bool:
2      assert len(code) == 3, "Airport code must have three characters:
           " + repr(code)
3      for c in code:
4          assert c.isalpha(), "Non-letter in airport code: " + repr(
               code)
5          assert c.isupper(), "Lowercase letter in airport code: " +
               repr(code)
6      return True
```

[1] https://www.fuzzingbook.org/html/Fuzzer.html

and we can check it:

```
assert code_repOK("SEA")
```

which quietly does not fail. You can also invoke `code_repOK` on all of the codes in `airport_codes`, and in fact, you can make a function to do so:

```
def airport_codes_repOK():
  for code in airport_codes:
      assert code_repOK(code)
  return True
```

If this is an abstract data type, you would usually use a function to add to it:

```
def add_new_airport(code: str, city: str) -> None:
    assert code_repOK(code)
    assert airport_codes_repOK()
    airport_codes[code] = city
    assert airport_codes_repOK()
```

which checks the data structure before-and-after as well as the input. Might be slow.

Note that this checks properties that are *specific to your data structure*. I call them domain-specific properties. It's not just "program doesn't crash", it's something that encodes information about how your program is supposed to behave. It's a matter of taste to encode the right things.

Here's a collection of checks for a red-black tree:

```
class RedBlackTree:
  def repOK(self):
      assert self.rootHasNoParent()
      assert self.rootIsBlack()
      assert self.rootNodesHaveOnlyBlackChildren()
      assert self.treeIsAcyclic()
      assert self.parentsAreConsistent()
      return True

  def rootIsBlack(self):
      if self.parent is None:
          assert self.color == BLACK
          return True
```

You might (automatically) run these checks every time you add or remove from the red-black tree, though an acylicity check might be expensive and best to disable in production. This kind of check does work well with automatically-generated inputs from fuzzing, which we'll discuss in a few weeks.

Test Doubles

Mock objects are a particular kind of test double. We need test doubles because objects collaborate with other objects, but we only want to test one object at a time. Meszaros categorizes test doubles as follows:

- dummy objects: these are not actually test doubles; they don't do anything, but just take up space in parameter lists. Are like `null`, but get past nullness checks in code.
- fake objects: have actual behaviour (which is correct), but somehow unsuitable for use in production; typical example is an in-memory database.
- stubs: produce canned answers in response to interactions from the class under test.
- mocks: like stubs, also produce canned answers. Difference: mock objects also check that the class under test makes the appropriate calls.
- spies: usually wraps the real object (instead of the mock, which stubs it), and records interactions for later verification.

Shorter reference about test doubles: `martinfowler.com/articles/mocksArentStubs.html`

Mock Objects

Before we talk about mock objects, let's look at a stub. Imagine that you have a service that sends out emails. You don't actually want to send out emails while you're testing. So here's a class that pretends to send out emails.

```
1 public class MailServiceStub implements MailService {
2   private List<Message> messages = new ArrayList<Message>();
3   public void send (Message msg) {
4     messages.add(msg);
5   }
6   public int numberSent() {
7     return messages.size();
8   }
9 }
```

This stub permits *behavioural verification*, as seen in the following assert in a test:

```
1   assertEquals(1, mailer.numberSent());
```

This is more like behavioural verification because it's checking that interactions that have happened. Technically, it's checking the state that records the interaction, but I'd still say behavioural rather than state. One could also check the recipients, contents of messages, etc.

jMock example. Instead of state verification, we can also do behaviour verification. This is jMock syntax.

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

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

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

The calls to `mock()` create mock objects which have the appropriate type. If you are using the objects as simple dummy objects, calling `mock()` and `proxy()` is enough. Note that we have a real `Order` object but we're giving it the fake proxy objects, as created by the `Mock`'s `proxy()` methods.

We also specify the expected behaviour of the `mailer` and the `warehouse`. The test case is saying that the mailer ought to have `send()` called on it once, and that the warehouse ought to have `hasInventory()` called; that method should return `false()`.

EasyMock example. Different mock object libraries have different syntax. Here's another example, this time for EasyMock.

```
@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");
  }
}
```

Here we are testing the `ClassUnderTest` and creating a mock object of `Collaborator` type. EasyMock 2.3 reads the `@Mock` annotation and automatically fills in a mock object of the appropriate type. In our test case, we call `replay(mock)` to indicate that we are no longer recording expectations, but are instead starting the test case itself. In the above code, there are currently no expectations.

Let's add some expectations.

```
1  @Test
2  public void testAddDocument() {
3    // ** recording phase **
4    // expect document addition
5    mock.documentAdded("Document");
6    // expect to be asked to vote for document removal, and vote for it
7    expect(mock.voteForRemoval("Document"))
8                .andReturn((byte) 42);
9    // expect document removal
10   mock.documentRemoved("Document");
11   replay(mock);
12   // ** replaying phase ** we expect the recorded actions to happen
13   classUnderTest.addDocument("New Document", new byte[0]);
14   // check that the behaviour actually happened:
15   verify(mock);
16 }
```

Here we record the fact that the mock should be called with `documentAdded` and a parameter "New Document". We also record that the mock's `voteForRemoval` method should be called, and when that happens, it should return value 42. Finally, we add a call `verify()` to let EasyMock know that we're done and that it can go ahead and check that the expected behaviour actually happened.

Flaky Tests

The second test engineering topic I want to talk about today is flaky tests. Flaky tests are those that sometimes fail (nondeterministically). Flakiness is not something you want in your test cases. (I have heard one defense of a flaky test: it lets you know that the system has the potential to actually work.) In general, flaky tests don't play well with the expectation that your test suite passes 100%.

Reference:
Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, Darko Marinov. "An Empirical Analysis of Flaky Tests". In Proceedings of Foundations of Software Engineering '14.

Dealing with flaky tests. Companies with large test suites have found mitigations for the flaky test suite problem. One can label known-flaky tests as flaky and automatically re-run them to see if they eventually pass. One can also ignore or remove flaky tests. But this is unsatisfactory: it takes a long time to re-run failing tests.

Causes of flakiness. Luo et al studied 201 fixes to flaky tests in open-source projects. They found that the three most common causes of fixable flaky tests were:

1. improper waits for asynchronous responses;
2. concurrency; and
3. test order dependency.

The problem that caused flakiness for asynchronous waits was that there was typically a `sleep()`

call which didn't wait long enough for the action (perhaps a network call) to finish. The best practice is to use some sort of `wait()` call to wait for the result instead of hardcoding a sleep time.

Concurrency problems were what one might expect. The problem could either be in the system under test or in the test itself. Problems included data races, atomicity violations, and deadlocks; the solutions were the proper use of concurrency primitives (e.g. locks) as seen in CS343.

Test order dependency problems arose when some tests expected other tests to have already executed (and left a side effect like a file in the filesystem). They came up especially in the transition from Java 6 to Java 7 because that transition changed the (not-guaranteed) test execution order. The solution is to remove the dependency.

Not unit tests

Many interesting things happen when different units interact. Integration tests verify end-to-end functionality. They're slower, flakier, and harder both to write and use. Focus on unit tests while developing code. But you eventually need integration tests to make sure the user sees the right thing.

When to stop? Idea 1: Coverage

So, in the testing space, we write a bunch of tests and hope it's good enough. For most of us, it's never that fun to write tests. But, you want to do a good job, so you should write some number of tests. What is that number?

If you could test your function or program on every single input (and if you had an oracle to tell you if the output was correct—more on that later), then that would clearly be enough. But, this doesn't work. Even if your program is not timing-sensitive. There are just too many inputs. Exponential growth strikes again.

Short of that, one metric that people use in industry is the notion of code coverage. In particular, statement coverage and branch coverage. There are other notions of coverage, but they are not widely used and I don't think they actually tell you anything useful.

You could evaluate statement coverage and branch coverage based on program source code and lines of code. When we talk about the real world, we'll do that. The easiest notation to reason about is an intermediate representation with a control-flow graph, but we don't actually need that in this version of this course. We will assume that each non-blank, non-comment line of code corresponds to one program statement.

Aside: white-box and black-box testing. I don't think this is really a big deal these days, but there is the term *white-box* testing, which means that you can look at the source code when you write tests, and *black-box* testing, where you can't.

Statement & Branch Coverage

I used to give a more formal definition, but it boils down to this. You have a test suite and a program.

Instrument the program to count whether each statement is executed or not. Also instrument it to count whether each branch is taken or not.

Statement coverage is the fraction of statements that are executed by the test suite. *Branch coverage* is the fraction of branches that are executed.

Let's look at an example, which is also available in the `code/L03` directory of the pdfs repository. Here is some code.

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

And here's a test suite.

```
1  # from parent directory, run:
2  #   > python3 -m unittest L03.test_suite
3  # or:
4  #   > python3-coverage run -m unittest L03.test_suite
5  #   > python3-coverage run --branch -m unittest L03.test_suite
6  #   > python3-coverage report -m
7  #   > python3-coverage html
8
9  import unittest
10
11 from .foo import Foo
12
13 class CoverageTests(unittest.TestCase):
14     def test_one(self):
15         f = Foo()
16         f.m(1, 2)
17
18     def test_two(self):
19         f = Foo()
20         f.m(1, -2)
21
22     def test_three(self):
```

```
23          f = Foo()
24          f.m(-1, 2)
```

The textual report has much of the important information, though the HTML report is easier to navigate:

```
Name                                                                Stmts   Miss Branch BrPart  Cover   Missing
---------------------------------------------------------------------------------------------------------------
/usr/lib/python3/dist-packages/_distutils_hack/__init__.py            101     96     38      0     6%   2-101, 111-239
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%
```

One can add missing test cases to make sure that all the lines are visited. One can also observe that, even with 100% branch coverage, one is missing an important behaviour: what if `b` is 0?

Tracking Python Coverage in Python

I said "instrument the program". Different languages have different support for this. In C, you need to compile the program with instrumentation. In Java, the virtual machine can collect information as the program executes (more useful if there is debug information). Python also provides hooks[2] so that you can collect information about program execution and implement your own *dynamic program analyses*.

Specifically, you can use the Python function `sys.settrace(f)` to register `f()` as a *tracing function* which gets called once per line executed. It gets passed information about the line which is being executed and the current context (e.g. contents of variables and the call stack.)

Here is an example tracing function, which I've put in the repo at `code/L03/tracing.py`.

```
1  from types import FrameType, TracebackType
2  coverage = []
3
4  def traceit(frame: FrameType, event: str, arg: Any) -> Optional[
       Callable]:
5      """Trace program execution. To be passed to sys.settrace()."""
6      if event == 'line':
7          global coverage
8          function_name = frame.f_code.co_name
9          lineno = frame.f_lineno
10         coverage.append(lineno)
11
12     return traceit
```

This records the line number and function name to be executed in the global `coverage` list. More generally, the `frame` parameter includes information about the function name, line number, and local variables and arguments. The `event` parameter tells you that Python is about to execute a new `"line"`, `"call"`, or perform some other event. Finally, `arg` gives you additional information about the `event` when appropriate, e.g. the value being returned for a `"return"` event.

[2] We are following `https://www.fuzzingbook.org/html/Coverage.html` here.

I've included a function `cgi_decode()` from the *Fuzzing Book* in `tracing.py`, and we can use it to test out tracing:

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

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

```
1 >>> cgi_decode_traced("a+b")
2 >>> print(coverage)
3 [12, 13, 12, 13, 12, 13, 12, 13, 12, 13, 12, 14, 12, 14, 12, 14, 12,
    14, 12, 14, 12, 15, 12, 15, 12, 15, 12, 15, 12, 15, 12, 15, 12,
   16, 12, 16, 12, 19, 20, 21, 22, 23, 25, 34, 35, 21, 22, 23, 24,
   35, 21, 22, 23, 25, 34, 35, 21, 36]
```

The *Fuzzing Book* continues with using Python introspection to retrieve the source code and pretty-print it again, highlighting which lines have and, importantly, have not been covered. This allows the user to write tests to ensure better statement coverage.

There is one more thing. The *Fuzzing Book* also talks about the usage of `with` to turn tracing on and off, and storing the result to a specified variable:

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

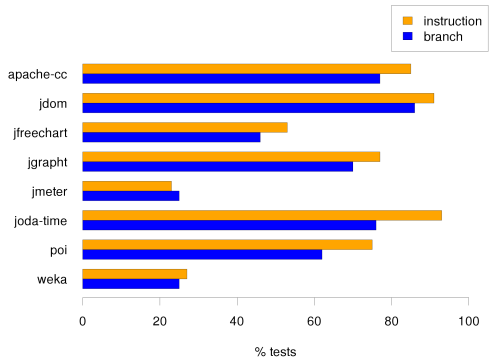
Here, the `Coverage` class does a bit more magic to provide the desired functionality, but we won't talk about it. Basically, we can print out either `cov` (which has a useful pretty-printing function) or the result of its `coverage()` method, which is a set of method/line pairs.

Infeasible Test Requirements; How Much Coverage, Anyway?

About "better statement coverage". For toy programs, we can reach 100% statement and branch coverage. For real programs, this is still theoretically possible, but becomes impractical.

We'll wrap up our unit on defining test suites by exploring the question "How much is enough?" We'll discuss coverage first and then mutation testing as ways of answering this question.

First, we can look at actual test suites and see how much coverage they achieve. I collected this data a few years ago, measured with the EclEmma tool.



We can see that the coverage varies between 20% and 95% on actual open-source projects. I investigated further and found that while Weka has low test coverage, it instead uses scientific peer review for QA: its features come from published articles. Common practice in industry is that about 80% coverage (doesn't matter which kind) is good enough.

Let's look at a more specific case study, JUnit. The rest of this lecture is based on a blog post by Arie van Deursen:

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

Although you might think of JUnit as something that just magically exists in the world, it is a software artifact too. JUnit is written by developers who obviously really care about testing. Let's see what they do.

Here's the Cobertura report for JUnit:

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%	77/157	41%	18/44	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%	3/33	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.

Stats. Overall instruction (statement) coverage for JUnit 4.11 is about 85%; there are 13,000 lines of code and 15,000 lines of test code. (It's not that unusual for there to be more tests than code.) This is consistent with the industry average.

Deprecated code? Sometimes library authors decide that some functionality was not a good idea after all. In that case they might *deprecate* some methods or classes, signalling that these APIs will disappear in the future.

In JUnit, deprecated and older code has lower coverage levels. Its 13 deprecated classes have only 65% instruction coverage. Ignoring deprecated code, JUnit achieves 93% instruction coverage. Furthermore, newer code in package `org.junit.*` has 90% instruction coverage, while older code in `junit.*` has 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.)

Untested class. The blog post points out one class that was completely untested, which is unusual for JUnit. It turns out that the code came with tests, but that the tests never got run because they were never added to any test suites. Furthermore, these tests also failed, perhaps because no one had ever tried them. The continuous integration infrastructure did not detect this change. (More on CI later.)

What else? Arie van Deursen characterizes the remaining 6% as "the usual suspects". In JUnit's case, there was no method with more than 2 to 3 uncovered lines. Here's what he found.

Too simple to test. Sometimes it doesn't make sense to test a method, because it's not really doing anything. For instance:

```
1  public static void assumeFalse(boolean b) {
2    assumeTrue(!b);
3  }
```

or just getters or `toString()` methods (which can still be wrong).

The empty method is also too simple to test; one might write such a method to allow it to be overridden in subclasses:

```
1  /**
2  * Override to set up your specific external resource.
3  *
4  * @throws if setup fails (which will disable {@code after}
5  */
6  protected void before() throws Throwable {
7    // do nothing
8  }
```

Dead by design. Sometimes a method really should never be called, for instance a constructor on a class that should never be instantiated:

```
1  /**
2  * Protect constructor since it is a static only class
3  */
4  protected Assert() { }
```

A related case is code that should never be executed:

```
1  catch (InitializationError e) {
2    throw new RuntimeException(
```

```
3       "Bug in saff's brain: " +
4       "Suite constructor, called as above, should always complete");
5     }
```

Similarly, switch statements may have unreachable default cases. Or other unreachable code. Sometimes the code is just highly unlikely to happen:

```
1     try {
2       ...
3     } catch (InitializationError e) {
4       return new ErrorReportingRunner(null, e); // uncovered
5     }
```

Conclusions. We explored empirically the instruction coverage of JUnit, which is written by people who really care about testing. Don't forget that coverage doesn't actually guarantee, by itself, that your code is well-exercised; what is in the tests matters too. For non-deprecated code, they achieved 93% instruction coverage, and so it really is possible to have no more than 2-3 untested lines of code per method. It's probably OK to have lower coverage for deprecated code. Beware when you are adding a class and check that you are also testing it.