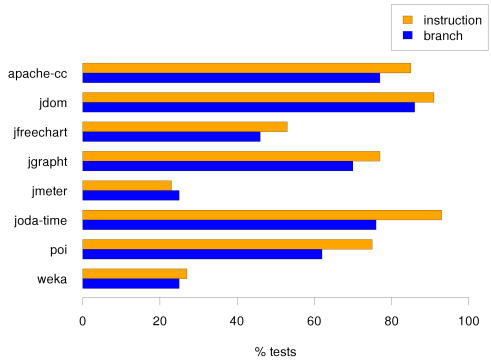
Software Testing, Quality Assurance and Maintenance Winter 2017

Lecture 13 — February 1, 2017

Patrick Lam *version 0*

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.

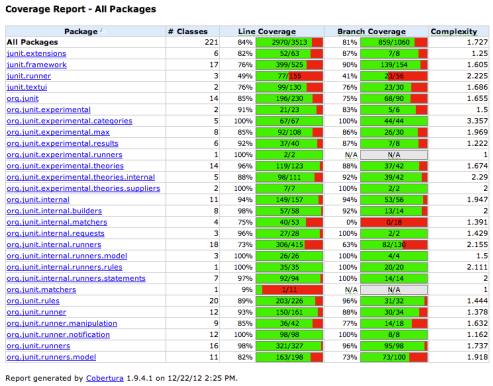
There is essentially no quantitative analysis of input space coverage; for most purposes, input spaces are essentially infinite.

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%	28/68	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.

Stats. Overall instruction (statement) coverage for JUnit 4.11 is about 85%; there are 13,000 lines of code and 15,000 lines ot 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.