Software Testing, Quality Assurance and Maintenance Winter 2025

Lecture 3 — January 13, 2025

Patrick Lam *version 1*

We're going to continue talking about testing for now. This week, I also plan to talk about fuzzing. Then we'll shift gears—we are going to talk about operational semantics, which is the invisible foundation for the proofs that you saw in SE 212. Making the semantics visible makes the proofs doable by machine.

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), then that would clearly be enough. This doesn't work. Even if your program is not timing-sensitive. There are just too many inputs. That's exponential growth for you.

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. But that's a bit ambiguous. We use control-flow graphs instead.

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.

The fundamental graph for source code is the *Control-Flow Graph* (CFG), which originates from compilers.

- CFG nodes: a node represents zero or more statements;
- CFG edges: an edge (s_1, s_2) indicates that s_1 may be followed by s_2 in an execution.

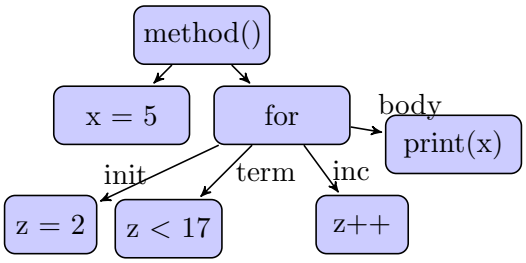
Example. Consider the following code.

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

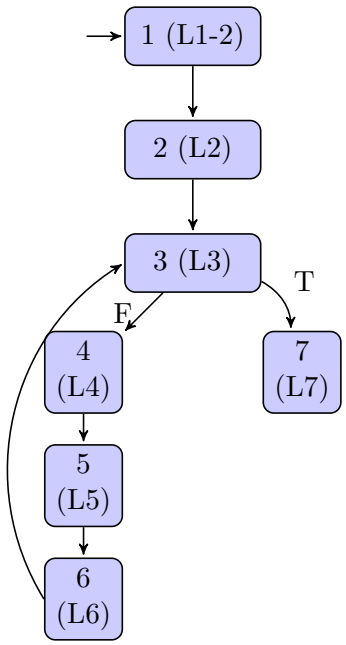
Recall the steps in compilation:

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

The Abstract Syntax Tree corresponding to the example code might look like this:



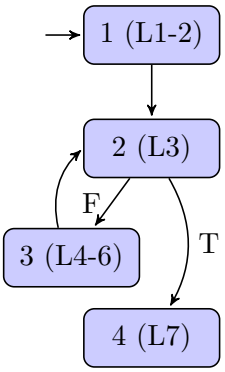
From ASTs to CFGs. We can convert the Abstract Syntax Tree into the following Control Flow Graph.



From CFG to low-level code. And we can convert the CFG into the following low-level code:

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

Basic Blocks. We can simplify a CFG by grouping together statements which always execute together (in sequential programs):



We use the following definition:

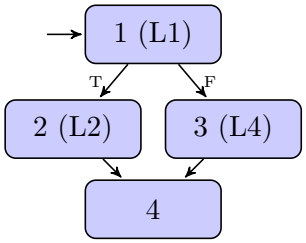
Definition 1 *A basic block is a sequence of instructions in the control-flow graph that has one entry point and one exit point.*

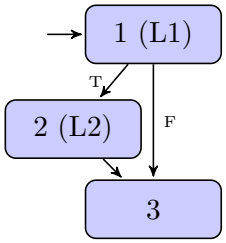
We are usually interested in forming maximal basic blocks. Note that a basic block may have multiple successors. However, there may not be any jumps into the middle of a basic block (which is why statement `l0` has its own basic block.)

Constructing CFGs. This is a mechanical process which I don't want to talk about much.

if statements: One can put the conditions (and hence uses) on the control-flow edges, rather than in the `if` node. I prefer putting the condition in the node.

```
1  if (z < 17)
2    print (x);
3  else
4    print (y);
```



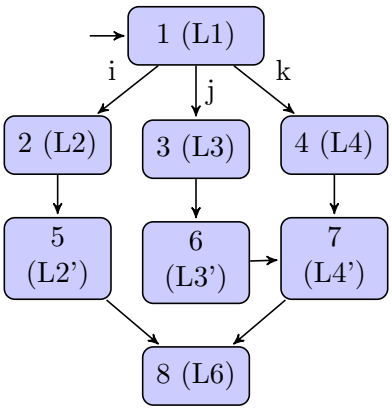



```
1  if (z < 17)
2    print(x);
```

Short-circuit `if` evaluation is more complicated; I recommend working it out yourself.

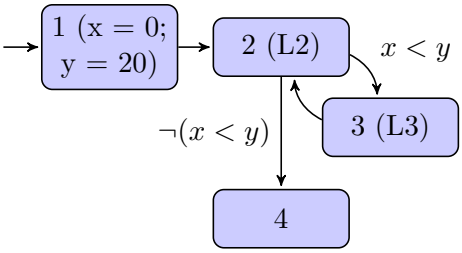
case / switch statements:

```
1  switch (n) {
2    case 'I': ...; break;
3    case 'J': ...; // fall thru
4    case 'K': ...; break;
5  }
6  // ...
```



while statements:

```
1  x = 0; y =
       20;
2  while (x < y)
       {
3    x ++; y --;
4  }
```



Note that arbitrarily complicated structures may occur inside the loop body.

for statements:

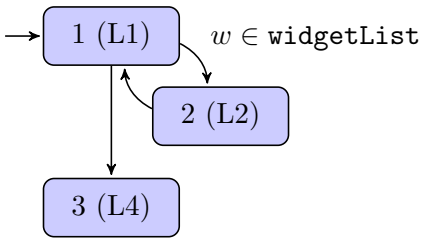
```
1  for (int i = 0; i < 57; i++) {
2    if (i % 3 == 0) {
3      print (i);
4    }
5  }
```

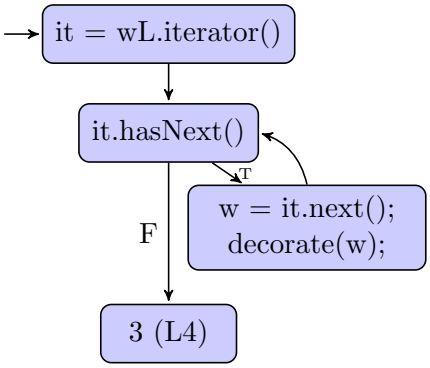
(an exercise for the reader;
we saw one earlier!)

This example uses Java's enhanced for loops, which iterates over all of the elements in the `widgetList`:

```
1  for (Widget w : widgetList) {
2    decorate(w);
3  }
```

Either the simplified one or the more useful one on the right are OK:





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. For this purpose, the program is expressed in terms of a set of CFGs (one per method).

Instrument the program to count whether each statement (control-flow node) is executed or not. Also instrument it to count whether each branch (control-flow edge) is taken or not.

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.

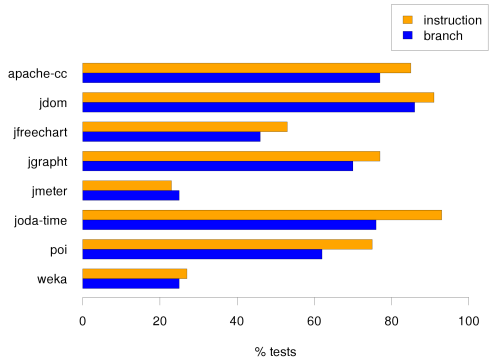
[Should give an example]

Infeasible Test Requirements; How Much Coverage, Anyway?

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.

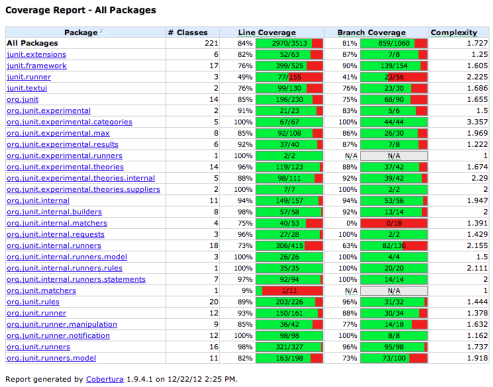
There is essentially no quantitative analysis of input space coverage; for most purposes, input spaces are essentially infinite.

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%	2570/3513	81%	859/1060	1.727
junit.extensions	6	82%	32/63	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.

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.

Fuzzing

Consider the following JavaScript code[1].

```
1 function test() {
2     var f = function g() {
3         if (this != 10) f();
4     };
5     var a = f();
6 }
7 test();
```

Turns out that it can crash WebKit (`https://bugs.webkit.org/show_bug.cgi?id=116853`). Plus, it was automatically generated by the Fuzzinator tool, based on a grammar for JavaScript.

Fuzzing is the modern-day implementation of the input space based grammar testing that we talked about in last time. While the fundamental concepts were in the earlier lecture, we will see how those concepts actually work in practice. Fuzzing effectively finds software bugs, especially security-based bugs (caused, for instance, by a lack of sufficient input validation.)

Origin Story. It starts with line noise. In 1988, Prof. Barton Miller was using a 1200-baud dialup modem to communicate with a UNIX system on a dark and stormy night. He found that the random characters inserted by the noisy line would cause his UNIX utilities to crash. He then challenged graduate students in his Advanced Operating Systems class to write a fuzzer—a program which would generate (unstructured ASCII) random inputs for other programs. The result: the students observed that 25%-33% of UNIX utilties crashed on random inputs[2].

(That was not the earliest known example of fuzz testing. Apple implemented "The Monkey" in 1983[3] to generate random events for MacPaint and MacWrite. It found lots of bugs. The limiting factor was that eventually the monkey would hit the Quit command. The solution was to introduce a system flag, "MonkeyLives", and have MacPaint and MacWrite ignore the quit command if MonkeyLives was true.)

How Fuzzing Works. Two kinds of fuzzing: *mutation-based* and *generation-based*. Mutation-based testing starts with existing test cases and randomly modifies them to explore new behaviours. Generation-based testing starts with a grammar and generates inputs that match the grammar.

One detail that I didn't mention last time was the bug detection part. Back then, I just talked about generating interesting inputs. In fuzzing, you feed these inputs to the program and find crashes, or assertion failures, or you run the program under a dynamic analysis tool such as Valgrind and observe runtime errors.

[1] `http://webkit.sed.hu/blog/20130710/fuzzinator-mutation-and-generation-based-browser-fuzzer`
[2] `http://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html`
[3] `http://www.folklore.org/StoryView.py?story=Monkey_Lives.txt`

The Simplest Thing That Could Possibly Work. Consider generation-based testing for HTML5. The simplest grammar—actually a regular expression—that could possibly work[4] is `.*`, where `.` is "any character" and `*` means "0 or more". Indeed, that grammar found the following WebKit assertion failure: `https://bugs.webkit.org/show_bug.cgi?id=132179`.

The process is as described previously. Take the regular expression and generate random strings from it. Feed them to the browser and see what happens. Find an assertion failure/crash.

More sophisticated fuzzing. Let's say that we're trying to generate C programs. One could propose the following hierarchy of inputs[5]:

1. sequence of ASCII characters;
2. sequence of words, separators, and white space (gets past the lexer);
3. syntactically correct C program (gets past the parser);
4. type-correct C program (gets past the type checker);
5. statically conforming C program (starts to exercise optimizations);
6. dynamically conforming C program;
7. model conforming C program.

Each of these levels contains a subset of the inputs from previous levels. However, as the level increases, we are more likely to find interesting bugs that reveal functionality specific to the system (rather than simply input validation issues).

While the example is specific to C, the concept applies to all generational fuzzing tools. Of course, the system under test shouldn't ever crash on random ASCII characters. But it's hard to find the really interesting cases without incorporating knowledge about correct syntax for inputs (or, as in the Apple case, excluding the "quit" command). Increasing the level should also increase code coverage.

John Regehr discusses this issue at greater length[6] and concludes that generational fuzzing tools should operate at all levels.

Mutation-based fuzzing. In mutation-based fuzzing, you develop a tool that randomly modifies existing inputs. You could do this totally randomly by flipping bytes in the input, or you could parse the input and then change some of the nonterminals. If you flip bytes, you also need to update any applicable checksums if you want to see anything interesting (similar to level 3 above).

Here's a description of a mutation-based fuzzing workflow by the author of Fuzzinator.

> More than a year ago, when I started fuzzing, I was mostly focusing on mutation-based fuzzer technologies since they were easy to build and pretty effective. Having a nice error-prone test

[4] `http://trevorjim.com/a-grammar-for-html5/`

[5] `http://www.cs.dartmouth.edu/~mckeeman/references/DifferentialTestingForSoftware.pdf`

[6] `blog.regehr.org/archives/1039`

> suite (e.g. LayoutTests) was the warrant for fresh new bugs. At least for a while. As expected, the test generator based on the knowledge extracted from a finite set of test cases reached the edge of its possibilities after some time and didn't generate essentially new test cases anymore. At this point, a fuzzer girl can reload her gun with new input test sets and will probably find new bugs. This works a few times but she will soon find herself in a loop testing the common code paths and running into the same bugs again and again.[7]

Fuzzing Summary. Fuzzing is a useful technique for finding interesting test cases. It works best at interfaces between components. Advantages: it runs automatically and really works. Disadvantages: without significant work, it won't find sophisticated domain-specific issues.

Related: Chaos Monkey

Instead of thinking about bogus inputs, consider instead what happens in a distributed system when some instances (components) randomly fail (because of bogus inputs, or for other reasons). Ideally, the system would smoothly continue, perhaps with some graceful degradation until the instance can come back online. Since failures are inevitable, it's best that they occur when engineers are around to diagnose them and prevent unintended consequences of failures.

Netflix has implemented this in the form of the Chaos Monkey[8] and its relatives. The Chaos Monkey operates at instance level, while Chaos Gorilla disables an Availability Zone, and Chaos Kong knocks out an entire Amazon region. These tools, and others, form the Netflix Simian Army[9].

Jeff Atwood (co-founder of StackOverflow) writes about experiences with a Chaos Monkey-like system[10]. Why inflict such a system on yourself? "Sometimes you don't get a choice; the Chaos Monkey chooses you." In his words, software engineering benefits of the Chaos Monkey included:

- "Where we had one server performing an essential function, we switched to two."
- "If we didn't have a sensible fallback for something, we created one."
- "We removed dependencies all over the place, paring down to the absolute minimum we required to run."
- "We implemented workarounds to stay running at all times, even when services we previously considered essential were suddenly no longer available."

[7] `http://webkit.sed.hu/blog/20141023/fuzzinator-reloaded`
[8] `http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html`
[9] `http://techblog.netflix.com/2011/07/netflix-simian-army.html`
[10] `http://blog.codinghorror.com/working-with-the-chaos-monkey/`