Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 33 — April 3, 2019

Patrick Lam *version 1*

One more dynamic tool: Randoop

Key Idea: "Writing tests is a difficult and time-consuming activity, and yet it is a crucial part of good software engineering. Randoop automatically generates unit tests for Java classes."

Randoop generates random sequence of method calls, looking for object contract violations.

To use it, simply point it at a program & let it run.

Randoop discards bad method sequences (e.g. illegal argument exceptions). It remembers method sequences that create complex objects, and sequences that result in object contract violations.

`code.google.com/p/randoop/`

Here is an example generated by Randoop:

```
public static void test1() {
    LinkedList list = new LinkedList();
    Object o1 = new Object();
    list.addFirst(o1);

    TreeSet t1 = new TreeSet(list);
    Set s1 = Collections.synchronizedSet(t1);

    // violated in the Java standard library!
    Assert.assertTrue(s1.equals(s1));
  }
```

Course Summary

Many of the topics in this course are fairly straightforward. I hope that seeing them all in one place can help you make connections between the different topics.

Introduction

We started by talking about *faults*, *errors*, and *failures*. We also discussed *static* versus *dynamic* approaches, something which recurred throughout the course.

Defining Test Suites

Before defining test suites, I thought it was important for everyone to understand *exploratory testing*. We then moved on to *statement* and *branch* coverage, which require you to understand *control-flow graphs.* Alternatively, you might have a *Finite State Machine* and want to build test suites to cover round-trips in your FSM.

Grammar-based approaches are also important, particularly *fuzzing* for security-based properties. (Don't forget to try out the american fuzzy lop tool). We can also generate inputs from a grammar.

Mutation testing is probably the most difficult concept in the course. Recall that it's indirect: you're trying to make your test suite better by making sure that it can actually detect defects in the code.

We also looked at research which empirically evaluated best-case coverage of well-tested code (JUnit, can reach 93%; 80% is usual benchmark); which evaluated the usefulness of mutation testing (it actually works); and which evaluated the usefulness of coverage (not very, as a goal in itself).

Engineering Test Suites

We then moved on to discuss how to engineer test suites as artifacts. There's a lot more that I would have liked to talk about, like ensuring testability and test smells. But here's what we did discuss.

First, we talked about why you need good tests—it enables you to fearlessly modify your code without worrying about breaking it ("eat your vegetables!") We then talked about some *test design principles.* Moving on to more concrete points, we saw how Selenium let you write tests for webapps. *Regression testing* is also a key use for test suites; they should be fast and automated.

Tests themselves should be *self-checking.* They might verify either *state* or *behaviour* (using *mock objects*). They should be hooked up to a *continuous integration* system and should not be *flaky.*

Tools

A fundamental distinction is between *dynamic* and *static* approaches. Dynamic approaches have perfect information about a limited set of runs; static approaches have approximations which are valid for all runs.

We talked about bug-finding tools somewhat out of sequence to enable you to work on your project. The fundamental idea behind Coverity is to find contradictions and suspicious usage patterns. Your project does the same, but at a simpler level.

On to real tools, the first technique I talked about was still not a tool: *code review.* Along the same lines, *reporting bugs* is also important to talk about, but not strictly speaking a tool either.

We finally continued with real tools: *PMD* and *FindBugs*, which statically detect suspicious code patterns in Java source code and bytecode respectively. We also saw how to use PMD to run queries on your own codebases (using XPath expressions). `jshint` is another tool in the same spirit, but it detects sketchiness in JavaScript (like undefined variables, which you can't even use in sane languages). *Facebook Infer* uses more powerful static analysis to find memory leaks and null pointer dereferences, among others. All of these tools work on significant codebases.

Finally, on the dynamic tool side, we talked about *valgrind* and *Address Sanitizer*, which detect memory errors at runtime by instrumenting the code.