Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 10 — January 23, 2019

Patrick Lam *version 1*

REST APIs

REST (Representational State Transfer) is an architectural style widely used in web applications. Architectural styles are SE 464 material, but relevant to Assignment 1, so we'll talk about them briefly here. [I wouldn't ask questions about REST itself on exams, but one could expect questions about testing REST systems.]

Many applications in the world these days are web apps, and as you know, these apps usually have a frontend and a backend. Assignment 1 has you testing both the frontend and the backend of my app. We'll talk about the backend here. For our purposes, it provides an API which allows the client to store data.

Key features of REST:

- client-server;
- stateless; and
- provides uniform interfaces.

Client-server is easy. The client (in our case, the front-end JavaScript code) sends requests to a server. The server is responsible for storing the data. This enables the client to change somewhat independently of the server. For instance, the server might decide to change what database it uses behind the scenes. The client doesn't need to know about that—separation of concerns.

Statelessness helps systems scale better. What this means is that the server isn't responsible for remembering anything about client identities. The server receives requests and processes them independently. The usual concern is about authentication; requests need to carry authentication data with them (usually provided by a separate service). An analogy: because SE is a small program, I try to remember state in my head about each of you between interactions—at least your name. Larger programs need students to provide the necessary state (student ID numbers) with each request; the advisors would then look you up in a database. You can see how my approach doesn't scale to larger sets of students.

Uniform interfaces include resources and verbs. Resources are what the REST service is modifying, e.g. cat pictures. Each resource has a Uniform Resource Identifier. In our se465-flashcards app, it's just an ID number. But it could be more complicated, e.g. `/store/1/employee/2`. Verbs are what to do with the resources. HTTP defines standard operations, including `POST`, `GET`, and `DELETE`.

Further reading:

- Brian Mosigisi. "A Quick Understanding of REST".
  `https://scotch.io/bar-talk/a-quick-understanding-of-rest`

- Wikipedia is an OK reference here: `https://en.wikipedia.org/wiki/Representational_state_transfer`

Testing the REST API

Sending REST requests. At the lowest level, we need to be able to send REST requests. It's easy to send GET requests in the browser by just navigating to a page. You can't quite send other requests with your bare hands. But you can use the browser developer tools in Firefox and Chrome. The syntax can be a bit unwieldy; stackoverflow provides the following example[1]:

```
fetch('https://jsonplaceholder.typicode.com/posts', {
  method: 'POST',
  body: JSON.stringify({
    title: 'foo', body: 'bar', userId: 1
  }),
  headers: {
    'Content-type': 'application/json; charset=UTF-8'
  }
})
.then(res => res.json())
.then(console.log)
```

Or, you can use command-line tools like `cURL` and `wget`. The Firefox dev tools will allow you to copy a request as cURL and run at the command line, possibly after editing.

But really, you want to script this. REST Assured lets you write Java code like this[2]:

```
    @Test public void makeSureThatGoogleIsUp() {
        given().when().get("http://www.google.com").then().statusCode(200);
    }
```

and run it as a JUnit test.

To check the result, one can compare the status code to what's expected. 200 OK is usually a good HTTP code, which APIs should return when everything went well. And there's also a result, which one can reason about:

```
    @Test public void verifyNameStructured() {
        given().when().get("/garage").then().body("name",equalTo("Acme garage"));
    }
```

You can also construct a POST request using the REST Assured API.

Testing Strategy. We discussed testing at the tactical level above, which is all you have to do for Assignment 1. The broader view from this course also includes test suite construction strategies, and we can talk about various kinds of coverage. You could, for instance, require coverage of the complete API. Better yet, you could require API coverage and also require that the tests cover the specified behaviour of the API.

[1] `https://stackoverflow.com/questions/14248296/making-http-requests-using-chrome-developer-tools`

[2] `https://semaphoreci.com/community/tutorials/testing-rest-endpoints-using-rest-assured`

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 11 — January 30, 2019

Patrick Lam *version 0*

Review: Statements, branches and beyond

We talked about statement (length-0 paths) and branch coverage (length-1 paths) last time. In lecture, we reviewed the elements needed to satisfy these coverage criteria: given a graph and a criterion, we get a set of Test Requirements TR. We execute each test t in the test set T on the system under test, giving a set of test paths $p \in P$. The criterion is satisfied if there exists at least one test path $p \in P$ that satisfies each of the test requirements tr $\in$ TR.

I'll say this again later, but for real programs, 80% coverage is usually good enough; but also consider what is not tested. Also, Assignment 1 Question 1 should point out to you that it's possible to have 100% statement coverage but not actually test anything, if you write test cases that don't have asserts.

We could extend to paths of length 2 and beyond, but soon that gets us to Complete Path Coverage (CPC), which requires an infinite number of test requirements.

Criterion 1 Complete Path Coverage. *(CPC)* TR *contains all paths in G.*

Note that CPC is impossible to achieve for graphs with loops.

[note: conceptually, the L14 notes should be here]

Testing State Behaviour of Software via FSMs

We can also model the behaviour of software using a finite-state machine. Such models are higher-level than the control-flow graphs that we've seen to date. They instead capture the design of the software. There is generally no obvious mapping between a design-level FSM and the code.

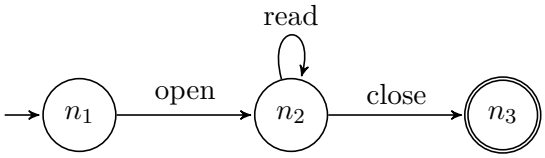
We propose the use of graph coverage criteria to test with FSMs.

- nodes: software states (e.g. sets of values for key variables);
- edges: transitions between software states, i.e. something changes in the environment or someone enters a command.

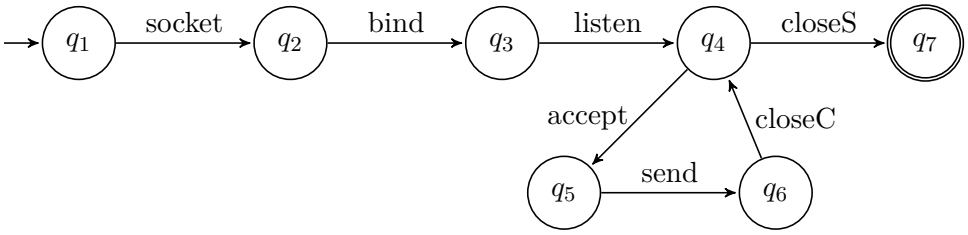
The FSM enables exploration of the software system's state space. A software state consists of values for (possibly abstract) program variables, while a transition represents a change to these program variables. Often transitions are guarded by preconditions and postconditions; the preconditions must hold for the FSM to take the corresponding transition, and the postconditions are guaranteed to hold after the FSM has taken the transition.

- node coverage: visiting every FSM state = state coverage;
- edge coverage: visiting every FSM transition = transition coverage;
- edge-pair coverage (extension of edge coverage to paths of length at most 2): actually useful for FSMS; transition-pair, two-trip coverage.

Examples. The next few graphs represent finite state machines rather than control-flow graphs. Our motivation will be to set up criteria that visit round trips in cyclic graphs.



or perhaps



The next criteria are mostly not for CFGs.

Definition 1 *A* round trip *path is a path of nonzero length with no internal cycles that starts and ends at the same node.*

Criterion 2 Simple Round Trip Coverage. *(SRTC)* TR *contains at least one round-trip path for each reachable node in G that begins and ends a round-trip path.*

Criterion 3 Complete Round Trip Coverage. *(CRTC)* TR *contains all round-trip paths for each reachable node in G.*

Exercise. Create a Finite State Machine for some system that you're familiar with.

Deriving Finite-State Machines

You might have to test software which doesn't come with a handy FSM. Deriving an FSM aids your understanding of the software. (You might be finding yourself re-deriving the same FSM as the software evolves; design information tends to become stale.)

We'll see some tools—iComment and Daikon—for obtaining sequencing constraints from comments/documentation and from the code.

Control-Flow Graphs. Does not really give FSMs.

- nodes aren't really states; they just abstract the program counter;
- inessential nondeterminism due e.g. to method calls;
- can only build these when you have an implementation;
- tend to be large and unwieldy.

Software Structure. Better than CFGs.

- subjective (which is not necessarily bad);
- requires lots of effort;
- requires detailed design information and knowledge of system.

Modelling State. This approach is more mechanical: once you've chosen relevant state variables and abstracted them, you need not think much.

You can also remove impossible states from such an FSM, for instance by using domain knowledge.

Specifications. These are similar to building FSMs based on software structure. Generally cleaner and easier to understand. Should resemble UML statecharts.

General Discussion. Advantages of FSMs:

- enable creation of tests before implementation;
- easier to analyze an FSM than the code.

Disadvantages:

- abstract models are not necessarily exhaustive;
- subjective (so they could be poorly done);
- FSM may not match the implementation.

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 12 — February 4, 2019

Patrick Lam *version 1*

Syntax-Based Testing

We are going to completely switch gears now. We will see two applications of context-free grammars:

1. input-space grammars: create inputs (both valid and invalid)
2. program-based grammars: modify programs (mutation testing)

Mutation testing. The basic idea behind mutation testing is to improve your test suites by creating modified programs (*mutants*) which force your test suites to include test cases which verify certain specific behaviours of the original programs by killing the mutants. We'll talk about this in more detail in a week or two.

Generating Inputs: Regular Expressions and Grammars

Consider the following Perl regular expression for Visa numbers:

```
^4[0-9]{12}(?:[0-9]{3})?$
```

Idea: generate "valid" tests from regexps and invalid tests by mutating the grammar/regexp. (Why did I put valid in quotes? What is the fundamental limitation of regexps?)

Instead, we can use grammars to generate inputs (including sequences of input events).

Typical grammar fragment:

$$
\begin{aligned}
\text{mult\_exp} &= \text{unary\_exp} \mid \text{mult\_exp } \texttt{STAR} \text{ unary\_arith\_exp} \mid \text{mult\_exp } \texttt{DIV} \text{ unary\_arith\_exp;} \\
\text{unary\_exp} &= \text{quant\_exp} \mid \text{unary\_exp } \texttt{DOT INT} \mid \text{unary\_exp up;} \\
&\vdots \\
\text{start} &= \text{header}^{?}\text{declaration}^{*}
\end{aligned}
$$

Using Grammars. Two ways you can use input grammars for software testing and maintenance:

- recognizer: can include them in a program to validate inputs;
- generator: can create program inputs for testing.

Generators start with the start production and replace nonterminals with their right-hand sides to get (eventually) strings belonging to the input languages. Typically you would generate inputs until you have enough, either randomly sampling from the input space or systematically generating inputs of larger and larger sizes.

Another Grammar.

```
roll    = action*
action  = dep | deb
dep     = "deposit" account amount
deb     = "debit" account amount
account = digit { 3 }
amount  = "$" digit+ "." digit { 2 }
digit   = ["0" - "9"]
```

Examples of valid strings.

Note: creating a grammar for a system that doesn't have one, but should, is a useful QA exercise. Using this grammar at runtime to validate inputs can improve software reliability, although it makes tests generated from the grammar less useful.

Some Grammar Mutation Operators.

- Nonterminal Replacement; e.g.
  `dep = "deposit" account amount` $\Longrightarrow$ `dep = "deposit" amount amount`

  (Use your judgement to replace nonterminals with similar nonterminals.)
- Terminal Replacement; e.g.
  `amount = "$" digit+ "." digit { 2 }` $\Longrightarrow$ `amount = "$" digit+ "$" digit { 2 }`

- Terminal and Nonterminal Deletion; e.g.
  `dep = "deposit" account amount` $\Longrightarrow$ `dep = "deposit" amount`

- Terminal and Nonterminal Duplication; e.g.
  `dep = "deposit" account amount` $\Longrightarrow$ `dep = "deposit" account account amount`

Using grammar mutation operators.

1. mutate grammar, generate (invalid) inputs; or,
2. use correct grammar, but mis-derive a rule once—gives "closer" inputs (since you only miss once.)

Why test invalid inputs? Bill Sempf (@sempf), on Twitter: "QA Engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 999999999 beers. Orders a lizard. Orders -1 beers. Orders a sfdeljknesv."

If you're lucky, your program accepts strings (or events) described by a regexp or a grammar. But you might not use a parser or regexp engine. Generating using the regexp or grammar helps detect deviations, both right now and in the future.

What we'll do is to mutate the grammars and generate test strings from the mutated grammars.

Some notes:

- Can generate strings still in the grammar even after mutation.
- Recall that we aren't talking about semantic checks.
- Some programs accept only a subset of a specified larger language, e.g. Blogger HTML comments. Then testing intersection is useful.

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 13 — February 4/6, 2019

Patrick Lam *version 1*

Fuzzing

Consider the following JavaScript code[1].

```
function test() {
    var f = function g() {
        if (this != 10) f();
    };
    var a = f();
}
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.

[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`

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.

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.

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

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

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

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 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/`

Software Testing, Quality Assurance and Maintenance Winter 2019

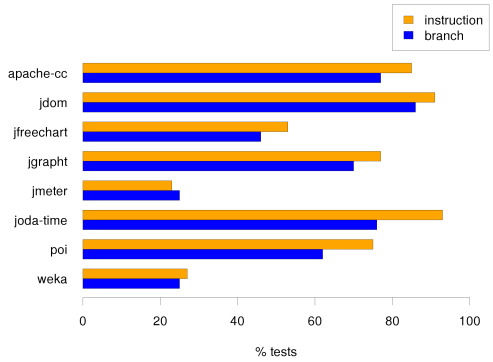
Lecture 14 — February 6, 2019

Patrick Lam *version 1*

[this makes more sense after the summary in L11]

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.

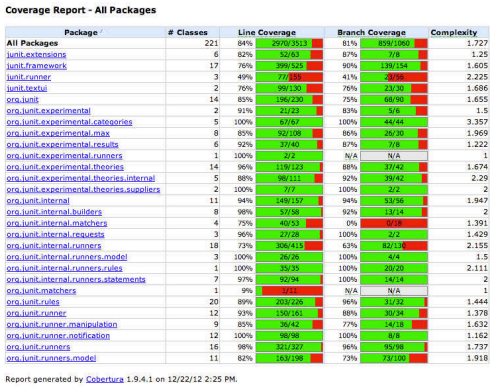
There is essentially no quantitative analysis of input space coverage; for most purposes, input spaces are essentially infinite.

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%	77/156	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%	2/21	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.

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 15 — February 13, 2019

Patrick Lam *version 1.1*

Mutation Testing

The second major way to use grammars in testing is mutation testing. Let's start with an example. Here is a program, along with some mutations to the program, which are derived using the grammar.

```
// original
int min(int a, int b) {
  int minVal;
  minVal = a;

  if (b < a) {



    minVal = b;




  }
  return minVal;
}
```

```
// with mutants
int min(int a, int b) {
  int minVal;
  minVal = a;
  minVal = b;                 // Δ 1
  if (b < a) {
  if (b > a) {                // Δ 2
  if (b < minVal) {           // Δ 3
    minVal = b;
    BOMB();                   // Δ 4
    minVal = a;               // Δ 5
    minVal = failOnZero(b); // Δ 6
  }
  return minVal;
}
```

Conceptually we've shown 6 programs, but we display them together for convenience. You'll find code in `live-coding/minval.c`.

We're generating mutants m for the original program m_0.

Definition 1 *Test case t* kills *m if running t on m gives different output than running t on m_0.*

We use these mutants to evaluate test suites. Here's a simple test suite. I'm abstractly representing a JUnit test case by a tuple; assume that it calls `min()` and asserts on the return value. You can fill out this table.

	Δ 1	Δ 2	Δ 3	Δ 4	Δ 5	Δ 6
$\langle a = 0, b = 1, \exp = 0 \rangle$	kill		–			
$\langle a = 1, b = 0, \exp = 0 \rangle$	–		–			
$\langle a = 1, b = 1, \exp = 1 \rangle$			–			
$\langle a = 1, b = 349, \exp = 1 \rangle$			–			

Note that, for instance, Δ 3 is not killable; if you look at the modification, you can see that it is equivalent to the original.

The idea is to use mutation testing to evaluate test suite quality/improve test suites. Good test suites ought to be effective at killing mutants.

General Concepts

Mutation testing relies on two hypotheses, summarized from [DPHG+18].

The *Competent Programmer Hypothesis* posits that programmers usually are almost right. There may be "subtle, low-level faults". Mutation testing introduces faults that are similar to such faults. (We can think of exceptions to this hypothesis—if the code isn't tested, for instance; or, if the code was written to the wrong requirements.)

The *Coupling Effect Hypothesis* posits that complex faults are the result of simple faults combining; hence, detecting all simple faults will detect many complex faults.

If we accept these hypotheses, then test suites that are good at ensuring program quality are also good at killing mutants.

Mutation is hard to apply by hand, and automation is complicated. The testing community generally considers mutation to be a "gold standard" that serves as a benchmark against which to compare other testing criteria against. For example, consider a test suite T which ensures statement coverage. What can mutation testing say about how good T is?

Mutation testing proceeds as follows.

1. *Generate mutants:* apply mutation operators to the program to get a set of mutants M.
2. *Execute mutants:* execute the test suite on each mutant and collect suite pass/fail results.
3. *Classify:* interpret the results as either killing each mutant or not; a failed test suite execution implies a killed mutant.

Although you could generate mutants by hand, typically you would tend to use a tool which parses the input program, applies a mutation operator, and then unparses back to source code, which is then recompiled.

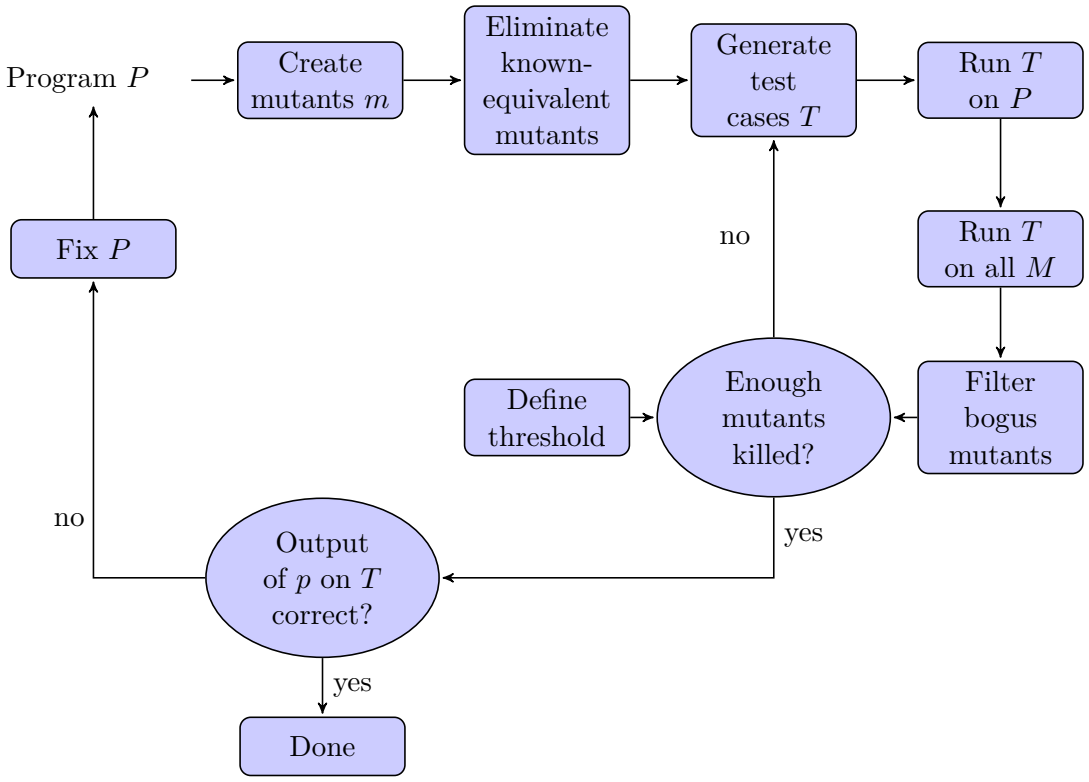
Executing the mutants can be computationally expensive, since you have to run the entire test suite on each of the mutants. This is a good time to use all the compute infrastructure available to you.

Generating and executing are computationally expensive, but classifying is worse, because it requires manual analysis. In particular, a not-killed result could be due to an equivalent mutant (like Δ 3 above); compilers can help, but the problem is fundamentally undecidable. Alternately, not-killed could be because the test suite isn't good enough. It's up to you to distinguish these cases. On the next page, "bogus mutants" denotes equivalent, stillborn and trivial mutants.

You would normally want to then craft new test cases to kill the non-equivalent mutants that you found.

Testing Programs with Mutation

Here's a picture that illustrates a variant of the above workflow.



Generating Mutants

Now let's see how to generate mutants; this ties in to the grammar-based material I talked about last time. For mutation testing, strings will always be programs.

Definition 2 *Ground string: a (valid) string belonging to the language of the grammar (i.e. a programming language grammar).*

Definition 3 *Mutation Operator: a rule that specifies syntactic variations of strings generated from a grammar.*

Definition 4 *Mutant: the result of one application of a mutation operator to a ground string.*

The workflow is to parse the ground string (original program), apply a mutation operator, and then unparse.

It is generally difficult to find good mutation operators. One example of a bad mutation operator might be to change all boolean expressions to "true". Fortunately, the research shows that you don't need many mutation operators—the right 5 will do fine.

Some points:

- How many mutation operators should you apply to get mutants? *One.*
- Should you apply every mutation operator everywhere it might apply? *Too much work; choose randomly.*

Killing Mutants. We can also define a mutation score, which is the percentage of mutants killed.

To use mutation testing for generating test cases, one would measure the effectiveness of a test suite (the mutation score), and keep adding tests until reaching a desired mutation score.

So far we've talked about requiring differences in the *output* for mutants. We call such mutants **strong mutants**. We can relax this by only requiring changes in the *state*, which we'll call **weak mutants**.

In other words,

- *strong mutation*: fault must be *reachable*, *infect* state, and **propagate** to output.
- *weak mutation*: a fault which kills a mutant need only be *reachable* and *infect state*.

Supposedly, experiments show that weak and strong mutation require almost the same number of tests to satisfy them.

Let's consider mutant $\Delta 1$ from above, i.e. we change `minVal = a` to `minVal = b`. In this case:

- reachability: unavoidable;
- infection: need $b \neq a$;
- propagation: wrong `minVal` needs to return to the caller; that is, we can't execute the body of the `if` statement, so we need $b > a$.

A test case for strong mutation is therefore $a = 5, b = 7$ (return value = $\sqcup$, expected $\sqcup$), and for weak mutation $a = 7, b = 5$ (return value = $\sqcup$, expected $\sqcup$).

Now consider mutant $\Delta 3$, which replaces `b < a` with `b < minVal`. This mutant is an equivalent mutant, since `a = minVal`. (The infection condition boils down to "false".)

Equivalence testing is, in its full generality, undecidable, but we can always estimate.

Program Based Grammars

The usual way to use mutation testing for generating test cases is by generating mutants by modifying programs according to the language grammar, using mutation operators.

Mutants are *valid programs* (not tests) which ought to behave differently from the ground string.

Mutation testing looks for tests which distinguish mutants from originals.

Example. Given the ground string `x = a + b`, we might create mutants `x = a - b`, `x = a * b`, etc. A possible original on the left and a mutant on the right:

```
int foo(int x, int y) { // original
  if (x > 5) return x + y;
  else return x;
}
```

```
int foo(int x, int y) { // mutant
  if (x > 5) return x - y;
  else return x;
}
```

In this example, the test case $\langle 6, 2\rangle$ will kill the mutant, since it returns 8 for the original and 4 for the mutant, while the case $\langle 6, 0\rangle$ will not kill the mutant, since it returns 6 in both cases.

Once we find a test case that kills a mutant, we can forget the mutant and keep the test case. The mutant is then *dead*.

The other thing that can happen when you are running a test case on a mutant is that it loops indefinitely. You'd want to use a timeout when running testcases, and then you have a timeout failure, which you can presumably use to distinguish the mutant from the original.

Uninteresting Mutants. Three kinds of mutants are uninteresting:

- *stillborn*: such mutants cannot compile (or immediately crash);
- *trivial*: killed by almost any test case;
- *equivalent*: indistinguishable from original program.

The usual application of program-based mutation is to individual statements in unit-level (per-method) testing.

References

[DPHG+18] Pedro Delgado-Pérez, Ibrahim Habli, Steve Gregory, Rob Alexander, John Clark, and Inmaculada Medina-Bulo. Evaluation of mutation testing in a nuclear industry case study. In *IEEE Transactions on Reliability*, pages 1406–1419, 2018.