Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 16 — February 25, 2019

Patrick Lam *version 1*

Recall that the goals of mutation testing are:

1. mimic (and hence test for) typical mistakes;
2. encode knowledge about specific kinds of effective tests in practice, e.g. statement coverage ($\Delta 4$ from Lecture 15), checking for 0 values ($\Delta 6$).

Reiterating the process for using mutation testing:

- *Goal:* kill mutants
- *Desired Side Effect:* good tests which kill the mutants.

These tests will help find faults (we hope). We find these tests by intuition and analysis.

Mutation Operators

We'll define a number of mutation operators, although precise definitions are specific to a language of interest. Typical mutation operators will encode typical programmer mistakes, e.g. by changing relational operators or variable references; or common testing heuristics, e.g. fail on zero. Some mutation operators are better than others.

You can find a more exhaustive list of mutation operators in the PIT documentation:

`http://pitest.org/quickstart/mutators/`

How many (intraprocedural) mutation operators can you invent for the following code?

```
int mutationTest(int a, b) {
  int x = 3 * a, y;
  if (m > n) {
    y = -n;
  }
  else if (!(a > -b)) {
    x = a * b;
  }
  return x;
}
```

Integration Mutation. We can go beyond mutating method bodies by also mutating interfaces between methods, e.g.

- change calling method by changing actual parameter values;
- change calling method by changing callee; or
- change callee by changing inputs and outputs.

```
class M {
  int f, g;

  void c(int x) {
    foo (x, g);
    bar (3, x);
  }

  int foo(int a, int b) {
    return a + b * f;
  }

  int bar(int a, int b) {
    return a * b;
  }
}
```

[Absolute value insertion, operator replacement, scalar variable replacement, statement replacement with crash statements. . .]

Mutation for OO Programs. One can also use some operators specific to object-oriented programs. Most obviously, one can modify the object on which field accesses and method calls occur.

```
class A {
  public int x;
  Object f;
  Square s;

  void m() {
    int x;
    f = new Object();
    this.x = 5;
  }
}

class B extends A {
  int x;
}
```

Exercise. Come up with a test case to kill each of these types of mutants.

- **ABS**: Absolute Value Insertion
  `x = 3 * a` $\Longrightarrow$ `x = 3 * abs(a)`, `x = 3 * -abs(a)`, `x = 3 * failOnZero(a);`
- **ROR**: Relational Operator Replacement
  `if (m > n)` $\Longrightarrow$ `if (m >= n)`, `if (m < n)`, `if (m <= n)`, `if (m == n)`, `if (m != n)`, `if (false)`, `if (true)`
- **UOD**: Unary Operator Deletion
  `if (!(a > -b))` $\Longrightarrow$ `if (a > -b)`, `if (!(a > b))`

Summary of Syntax-Based Testing.

	Program-based	Input Space/Fuzzing
Grammar	Programming language	Input languages / XML
Summary	Mutates programs / tests integration	Input space testing
Use Ground String?	Yes (compare outputs)	Sometimes
Use Valid Strings Only?	Yes (mutants must compile)	No
Tests	Mutants are not tests	Mutants are tests
Killing	Generate tests by killing	Not applicable

Notes:

- Program-based testing has notion of strong and weak mutants; applied exhaustively, program-based testing could subsume many other techniques.
- Sometimes we mutate the grammar, not strings, and get tests from the mutated grammar.

Tool support. PIT Mutation testing tool: `http://pitest.org`. Mutates your program, reruns your test suite, tells you how it went. You need to distinguish equivalent vs. not-killed.

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 17 — February 27, 2019

Patrick Lam *version 1*

We've talked about mutation testing in the past few lectures. I thought I'd summarize some recent research out of Waterloo, in collaboration with the University of Washington and the University of Sheffield, about: (1) the effectiveness of mutation testing; and (2) what coverage gets you in terms of test suite effectiveness.

Is Mutation Testing Any Good?

We've talked about mutation testing as a metric for evaluating test suites and making sure that test suites exercise the system under test sufficiently. The problem with metrics is that they can be gamed, or that they might measure not quite the right thing. When using metrics, it's critical to keep in mind what the right thing is. In this case, the right thing is the fault detection power of a test suite.

Some researchers set out to determine just that. They carried out a study, using realistic code, where they isolated a number of bugs, and evaluated whether or not there exists a correlation between real fault detection and mutant detection.

Summary. The answer is **yes**: test suites that kill more mutants are also better at finding real bugs. The researchers also investigated when mutation testing fell short—they enumerated types of bugs that mutation testing, as currently practiced, would not detect.

Methodology. The authors used 5 open-source projects. They isolated a total of 357 reproducible faults in these projects using the projects' bug reporting systems and source control repositories. They they generated 230,000 mutants using the Major mutation framework and investigated the ability of both developer-written test suites and automatically-generated test suites (EvoSuite, Randoop, JCrasher) to detect the 357 faults.

For each fault, the authors started with a developer-written test suite T_{bug} that did not detect the fault. Then, using the source repository, they extracted a developer-written test that detects the fault. Call this suite T_{fix}. Does T_{fix} detect more mutants than T_{bug}? If so, then we can conclude that the mutant behaves like a bug.

Results. The authors found that Major-generated mutation tests could detect 73% of the faults. In other words, for 73% of faults, some mutant will be killed by a test that also detects the fault. Increasing mutation coverage thus also increases the likelihood of finding faults.

The analogous numbers for branch coverage and statement coverage are, respectively, 50% and 40%. Specifically: the 357 tests that find faults only increase branch coverage 50% of the time, and they only increase statement coverage 40% of the time. So: improving your test suite often doesn't get rewarded with a better statement coverage score, and half the time doesn't result in a better branch coverage score. Conversely, improving statement coverage doesn't help find more bugs because you're already reaching the fault, but you aren't sensitive to the erroneous state.

The authors also looked at the 27% of remaining faults that are not found by mutants. For 10% of these, better mutation operators could have helped. The remaining 17% were not suitable for mutation testing: they were fixed by e.g. algorithmic improvements or code deletion.

Reference. René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. "Are Mutants a Valid Substitute for Real Faults in Software Testing?" In *Foundations of Software Engineering* 2014. pp654–665. `http://www.linozemtseva.com/research/2014/fse/mutant_validity/`

What Does (Graph) Coverage Buy You?

We've talked about graph coverage, notably statement coverage (node coverage) and branch coverage (edge coverage). They're popular because they are easy to compute. But, are they any good? Reid Holmes (a former Waterloo CS prof) and his student Laura Inozemtseva set out to answer that question.

Answer. Coverage does not correlate with high quality when it comes to test suites. Specifically: test suites that are larger are better because they are larger, not because they have higher coverage.

Methodology. The authors picked 5 large programs and created test suites for these programs by taking random subsets of the developer-written test suites. They measured coverage and they measured effectiveness (defined as % mutants detected; we've seen that detecting mutants is good, above).

Result. In more technical terms: after controlling for suite size, coverage is not strongly correlated with effectiveness.

Furthermore, stronger coverage (e.g. branch vs statement, logic vs branch) doesn't buy you better test suites.

Discussion. So why are we making you learn about coverage? Well, it's what's out there, so you should know about it. But be aware of its limitations.

Plus: if you are not covering some program element, then you obviously get no information about the behaviour of that element. Low coverage is bad. But high coverage is not necessarily good.

Reference. Laura Inozemtseva and Reid Holmes. "Coverage is Not Strongly Correlated with Test Suite Effectiveness." In *International Conference on Software Engineering* 2014. pp435–445. `http://www.linozemtseva.com/research/2014/icse/coverage/`

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 18 — February 27, 2019

Patrick Lam *version 1*

Engineering Test Suites

We are going to move onto the second part of the course. In the first part of the course, you learned ways to make sure that your test suites are exhaustive enough. In this part of the course, you will learn about making your test suites better as engineered artifacts.

Why tests? Let's start by talking about what test suites can do for you (as a developer).

Reference: Kat Busch. "A beginner's guide to automated testing."
`https://hackernoon.com/treat-yourself-e55a7c522f71`

Anecdote: "TODO: write tests." We're all busy, right? Surely tests are less important than writing actual code to solve problems. And it can be hard to set up the test.

When you're writing code, you clearly want to know that it can at least work. So you may have a development setup. Kat Busch describes writing a Java server to interface with an Android app. Her development setup involved manual testing:

- set up test server on dev machine;
- install app on test phone;
- manually create a test case on test phone.

She writes: "Obviously I didn't test very many code paths because it was just so tedious."

Since Dropbox (her employer at the time) used code review, she actually had to write tests to pass code review, even if it was annoying to do so.

"Lo and behold, I soon needed to fix a small bug." But, of course, it's easy to introduce even more bugs when fixing something. Fortunately, she had some tests.

> I ran the tests. Within a few seconds, I knew that everything still worked! Not just a single code path (as in a manual test), but all code paths for which I'd written tests! It was magical. It was so much faster than my manual testing. And I knew I didn't forget to test any edge cases, since they were all still covered in the automated tests.

Not writing tests is incurring technical debt. You'll pay for it later, when you have to maintain the code. Having tests allows you to move faster later, without worrying about breaking your code.

Writing tests is like eating your vegetables. It'll enable your code to go big and strong.

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 19 — March 4, 2019

Patrick Lam *version 1*

Another reason for tests: avoiding regressions. Code often lives in a web of dependencies; often, code isn't right or wrong on its own, but rather in terms of how it's used. "If you haven't written tests, then there's no reliable way for other coders to know that their commit has impacted yours." The biggest codebases we deal in your courses are tens of thousands of lines, but industrial codebases are millions of lines. Even though you may have worked with them on co-op, you were only there for a short time, not years.

> If your code is still in the codebase a year (or five) after you've committed it and there are no tests for it, bugs will creep in and nobody will notice for a long time.

She continues with an anecdote about a user-facing feature broken due to a seemingly-unrelated change. Test cases can help prevent this kind of brokenness.

> **If it matters that the code works you should write a test for it.** There is no other way you can guarantee it will work.

Test design principles

Now that I've sold you on why you should have tests, let's talk about how your tests should be structured.

Some more references:

- Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code.*
- Kent Beck. *Test Driven Development: By Example.*
- Roy Osherove. *The Art of Unit Testing: with examples in C#.*

Once again, I'm summarizing the article by Kat Busch, "A beginner's guide to automated testing." `https://hackernoon.com/treat-yourself-e55a7c522f71`

Many small tests, not one big test. My Assignment 1 Question 1 solution had 14 test cases. Each test case verified a different part of the behaviour. Out of 232 solutions, only 21 had fewer than 5 test cases. So most of you intuitively know this already. But why is this good?

- easier to deal with failures: you get a specific failed test case, ideally with a descriptive name. This is easier to debug.
- easier to understand what's being tested: again, because you used descriptive names, it's easier for future you, or someone else, to understand whether they need to add a new test case or not. Example: `test_integer_addition`, `test_integer_subtraction`, etc.

Make it easy to add new tests. I think that was the case for you on A1Q1 eventually, based on my `testEmpty` method. You just had to emulate that test case: create inputs; create a `FormattedCommandAlias`; and assert against the result. Using @Before and @After setup and teardown methods can help too (but was overkill for our case). "You should always make it easy for people to do the right thing."

Unit versus integration tests. Unit tests are more low-level and focus on one particular "class, module, or function". They should execute quickly. Sometimes you need to create fake inputs (or mocks) for unit tests; we'll talk about that too. You should generally not use an entire real input for a unit test.

As we discussed before, 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 hting.

"TODO: write tests." Should you write the tests first? Test-Driven Development says, well, tests first, and only write as much code as needed to pass the tests that you write. A middle ground might be good: write some code, write some tests, repeat.

> You'll find that writing tests as you go makes your interfaces better and makes your code more testable. If you find yourself writing something hard to test, you'll notice it early on when there's still time to improve the design.

Flaky tests are terrible. Although your A1Q1 tests should have been deterministic, not all tests are deterministic. Some tests fail a small percentage of the time.

- timeouts can fail when something takes surprisingly long;
- iterators can return items in random order;
- random number generators are, well, random.

Try really hard to make your tests not flaky. There are ways of dealing with them, but they're not good.

Looking inside the system under test. Even in A1Q1 I had to expose some of the internal state, namely the command that actually got executed by the `commandSender`. As much as possible, try to avoid testing internal state, but rather only what is externally visible. This makes your tests more resilient to rewrites, and also focuses the test on what actually matters.

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 20 — March 4, 2019

Patrick Lam *version 1*

We'll now shift to the topic of bug-finding. This is generally useful to know about, and is relevant to your course project.

A bug has got to be a violation of some specification. This might be a language-level specification (e.g. don't dereference null pointers), or it may be specific to an API that you are using.

For instance, you may have a method:

```
// callers must acquire the lock
static int reset_hardware(...) { ... }
```

and then a caller in the Linux kernel:

```
// linux/drivers/scsi/in2000.c:
static int in2000_bus_reset(...)
{
  ...
  // no lock acquisition => a bug!
  reset_hardware(...);
  ...
}
```

Examples of specifications:

- `len < 100`;
- `x = 16*y + 4*z + 3`;
- `fclose()` must be called after `fopen()`;
- and example from above: callers of `reset_hardware()` must acquire lock.

How do you get specifications? Well, programming languages come with some specifications applicable to all programs in that language. Or, you can have developers write specifications (an uphill battle). Or, you can automatically infer specifications from source code, execution traces, comments, etc. This includes static tools like Coverity, iComment, PR-Miner, etc. It also includes dynamic tools like Daikon, DIDUCE, etc.

More on Coverity. Coverity is a commercial product which can find many bugs in large (millions of lines) programs; it is therefore a leading company in building bug detection tools. We'll talk a bit more about Coverity in the future. Clients (900+) include organizations such as Blackberry, Yahoo, Mozilla, MySQL, McAfee, ECI Telecom, Samsung, Siemens, Synopsys, NetApp, Akamai, etc. These include domains including EDA, storage, security, networking, government (NASA, JPL), embedded systems, business applications, operating systems, and open source software. We have access to Coverity for this course, but there's also a free trial:

`http://softwareintegrity.coverity.com/FreeTrialWebSite.html`

Mistaken beliefs. Coverity reported encounters with a number of mistaken beliefs about how languages work:

"No, the loop will go through once!"

```
for (i = 1; i < 0; i++) {
  // ... this code is dead ...
}
```

"No, `&&` is 'or'!"

```
void *foo(void *p, void *q) {
  if (!p && !q)
    return 0;
}
```

"No, ANSI lets you write 1 past end of the array!"

```
unsigned p[4]; p[4] = 1;
```

("We'll just have to agree to disagree.")

Goal. Coverity aims to find as many serious bugs as possible. The problem is: what's the definition of a bug? Is there objective truth? If there is, Coverity doesn't know it.

- Contradictions: It attempts to find lies by cross-examining; contradictions indicate errors.
- Deviance: It assumes programs are mostly-correct and tries to infer correct behaviour from that assumption. If 1 person does X, then maybe it's right, or maybe that was just a coincidence. But if 1000 people do X and 1 person does Y, the 1 person is probably wrong.

Crucially: a contradiction constitutes an error, even without knowing the correct belief.

MUST-beliefs versus MAY-beliefs. We differentiate between MUST-beliefs (related to contradictions) and MAY-beliefs (related to deviance).

MUST-beliefs are inferred from acts that imply beliefs about the code. For instance:

```
x = *p / z; // MUST: p not null
            // MUST: z != 0
unlock(l); // MUST: l acquired
x++; // MUST: x not protected by l
```

MAY-beliefs, on the other hand, could be coincidental.

A();	A();	A();	A();	
// ...	// ...	// ...	// ...	// MAY: A() and B() are paired.
B();	B();	B();	B();	

We can check them as if they're MUST-beliefs and then rank errors by belief confidence (more on that later).

MUST-belief examples. Let's look first at a couple of MUST-beliefs about null pointers.

- If I write `*p` in a C program, I'm stating a MUST-belief that `p` had better not be `NULL`.
- If I write the check `p == NULL`, I'm implying two MUST-beliefs: 1) POST: `p` is `NULL` on true path, not-`NULL` on false path; 2) PRE: `p` was unknown before the check.

We can cross-check these for three different error types. I leave finding the actual error to you:

- check-then-use (79 errors, 26 false positives):

```
/* linux 2.4.1: drivers/isdn/svmb1/capidrv.c */
if (!card)
  printk(KERN_ERR, "capidrv-%d: ...", card->contrnr...);
```

- use-then-check (102 errors, 4 false positives):

```
/* linux 2.4.7: drivers/char/mxser.c */
struct mxser_struct *info = tty->driver_data;
unsigned flags;
if (!tty || !info->xmit_buf)
  return 0;
```

- contradictions/redundant checks (24 errors, 10 false positives:)

```
/* linux 2.4.7/drivers/video/tdfxfb.c */
fb_info.regbase_virt = ioremap_nocache(...);
if (!fb_info.regbase_virt)
  return -ENXIO;
fb_info.bufbase_virt = ioremap_nocache(...);
/* [META: meant fb_info.bufbase_virt!] */
if (!fb_info.regbase_virt) {
  iounmap(fb_info.regbase_virt);
```

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 21 — March 6, 2019

Patrick Lam *version 2*

Recall that we've been discussing beliefs. Here are a couple of beliefs that are worthwhile to check. (examples courtesy Dawson Engler.)

Redundancy Checking. 1) Code ought to do something. So, when you have code that doesn't do anything, that's suspicious. Look for identity operations, e.g.

$$\texttt{x} = \texttt{x},\ \texttt{1} * \texttt{y},\ \texttt{x\&x},\ \texttt{x|x}.$$

Or, a longer example:

```
/* 2.4.5-ac8/net/appletalk/aarp.c */
da.s_node = sa.s_node;
da.s_net = da.s_net;
```

Also, look for unread writes:

```
for (entry=priv->lec_arp_tables[i];
     entry != NULL; entry=next) {
  next = entry->next; // never read!
  ...
}
```

Redundancy suggests conceptual confusion.

So far, we've talked about MUST-beliefs; violations are clearly wrong (in some sense). Let's examine MAY beliefs next. For such beliefs, we need more evidence to convict the program.

Process for verifying MAY beliefs. We proceed as follows:

1. Record every successful MAY-belief check as "check".
2. Record every unsucessful belief check as "error".
3. Rank errors based on "check" : "error" ratio.

Most likely errors occur when "check" is large, "error" small.

Example. One example of a belief is use-after-free:

```
free(p);
print(*p);
```

That particular case is a MUST-belief. However, other resources are freed by custom (undocumented) free functions. It's hard to get a list of what is a free function and what isn't. So, let's derive them behaviourally.

Inferring beliefs: finding custom free functions. The key idea is: if pointer `p` is not used after calling `foo(p)`, then derive a MAY belief that `foo(p)` frees `p`.

OK, so which functions are free functions? Well, just assume all functions free all arguments:

- emit "check" at every call site;
- emit "error" at every use.

(in reality, filter functions with suggestive names).

Putting that into practice, we might observe:

```
foo(p)      foo(p)      foo(p)      bar(p)      bar(p)      bar(p)
*p = x;     *p = x;     *p = x;     p = 0;      p = 0;      *p = x;
```

We would then rank `bar`'s error first. Plausible results might be: 23 free errors, 11 false positives.

Inferring beliefs: finding routines that may return `NULL`. The situation: we want to know which routines may return `NULL`. Can we use static analysis to find out?

- sadly, this is difficult to know statically ("`return p->next;`"?) and,
- we get false positives: some functions return `NULL` under special cases only.

Instead, let's observe what the programmer does. Again, rank errors based on checks vs non-checks. As a first approximation, assume **all** functions can return `NULL`.

- if pointer checked before use: emit "check";
- if pointer used before check: emit "error".

This time, we might observe:

```
p = bar(...);    p = bar(...);        p = bar(...);        p = bar(...);
*p = x;          if (!p) return;      if (!p) return;      if (!p) return;
                 *p = x;              *p = x;              *p = x;
```

Again, sort errors based on the "check":"error" ratio.

Plausible results: 152 free errors, 16 false positives.

General statistical technique

When we write "a(); . . . b();", we mean a MAY-belief that a() is followed by b(). We don't actually know that this is a valid belief. It's a hypothesis, and we'll try it out. Algorithm:

- assume every `a–b` is a valid pair;
- emit "check" for each path with "a()" and then "b()";
- emit "error" for each path with "a()" and no "b()".

(actually, prefilter functions that look paired).

Consider:

foo(p, . . .);	foo(p, . . .);	foo(p, . . .);
bar(p, . . .); // check	bar(p, . . .); // check	// error: foo, no bar!

This applies to the course project as well.

```
void scope1() {
 A(); B(); C(); D();
}

void scope2() {
 A(); C(); D();
}

void scope3() {
 A(); B();
}

void scope4() {
 B(); D(); scope1();
}

void scope5() {
 B(); D(); A();
}

void scope6() {
 B(); D();
}
```

"A() and B() must be paired":
either A() then B() or B() then A().

Support = # times a pair of functions appears together.
support({A,B})=3

Confidence({A,B},{A}) =
support({A,B})/support({A}) = 3/4

Sample output for support threshold 3, confidence threshold 65% (intra-procedural analysis):

- bug:A in scope2, pair: (A B), support: 3, confidence: 75.00%
- bug:A in scope3, pair: (A D), support: 3, confidence: 75.00%
- bug:B in scope3, pair: (B D), support: 4, confidence: 80.00%
- bug:D in scope2, pair: (B D), support: 4, confidence: 80.00%

The point is to find examples like the one from `cmpci.c` where there's a `lock_kernel()` call, but, on an exceptional path, no `unlock_kernel()` call.

Summary: Belief Analysis. We don't know what the right spec is. So, look for contradictions.

- MUST-beliefs: contradictions = errors!
- MAY-beliefs: pretend they're MUST, rank by confidence.

(A key assumption behind this belief analysis technique: most of the code is correct.)

Further references. Dawson R. Engler, David Yu Chen, Seth Hallem, Andy Chou and Benjamin Chelf. "Bugs as Deviant Behaviors: A general approach to inferring errors in systems code". In SOSP '01.

Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. "Checking system rules using system-specific, programmer-written compiler extensions". In OSDI '00 (best paper). `www.stanford.edu/~engler/mc-osdi.pdf`

Junfeng Yang, Can Sar and Dawson Engler. "eXplode: a Lightweight, General system for Finding Serious Storage System Errors". In OSDI'06. `www.stanford.edu/~engler/explode-osdi06.pdf`

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 22 — March 11, 2019

Patrick Lam *version 1*

Regression Testing

Regression testing refers to any software testing that uncovers errors by retesting the modified program (Wikipedia). This form of testing often refers to comprehensive sets of test cases to detect regressions:

- of bug fixes that a developer has proposed.
- of related and unrelated other features that have been added.

Regression testing usually refers to system level (integration level) testing that runs the entire process.

Attributes of Regression Tests

Regression tests usually have the following attributes:

- **Automated**: no real reason to have manual regression tests.
- **Appropriately Sized**: too small and bugs will be missed. Too large and they will take a long time to run. Optimally, we want to run tests continuously.
- **Up-to-date**: ensure that tests are valid for the version of program being tested.

Automating Regression Tests

Regression tests often have a low yield in terms of finding bugs (and are boring to run). Automation is key.

Input

If the input is from a file, regression tests are easy to run (but should still be automatically triggered on a regular interval). There may still be a problem with validating output. We can also create special mocks that can take input from a file or other sources (e.g. scripting engines).

For UIs, the standard approach is to capture and replay events. This approach can be fragile! For example, tests may fail based on window placement or whitespace. For web applications, there is capture and replay for HTTP using Selenium. Mozilla has a project named Marionette[1] that is used to test Firefox and Thunderbird; it is like Selenium but also works on Chrome elements.

[1] `https://developer.mozilla.org/en-US/docs/Mozilla/QA/Marionette`

Output

Verifying output can be hard!! Problems can arise from issues such as resolution, whitespace, window placement etc.

Mozilla Case Study[2]

The case study presents an approach to testing Gecko based applications. Gecko is the layout engine for Mozilla applications like Firefox and Thunderbird.

In the past, a frame tree with coordinates for all UI elements was created and manual testers performed tests. Not optimal! The new approach was to capture screenshots after test cases and compare them with the expected screenshot. There were a few problems with the approach due to nondeterminism:

- Animated images: no way to ensure tests would function with animated images.
- Font Hinting: the same character would appear slightly differently after each run of the application.
- Other bugs: resolution problems, minor changes in layout etc.

The problem was "really hellish" and the partial solution was to enable logging in the application. The logs would essentially be compared with expected logs. This became very ugly given 1300 or so test cases; distributing across different computers helped.

Some Notes

- Sometimes it is a good idea to have 1 test case per bug fix. This can get unwieldy over time if redundant test cases are not removed.
- One could have tiered level of tests, which are progressively more detailed; run the high-level suite often and more detailed tests as needed.
- There is some interest in test case prioritization but little practical application since it is unclear how to implement it.
- Expect low yields in terms of finding bugs so automation is key!
- Ensure regression tests are up-to-date. This can be a pain but is necessary since when software changes, test suits break or become incomplete. When a new software version comes out, try the old suite.
  - If there are no failures, new tests may need to be added to test new functionality.
  - If there are some failures, determine if software or test is broken. Tests can depend on inessential features of the output (e.g. order or text etc.).
- Try to get rid of irrelevant tests over time. For example, if you have a bunch of tests that are related to the same bug, keep only 1 test for the bug.

[2] `http://robert.ocallahan.org/2005/03/visual-regression-tests_04.html`

Industrial Best Practices

- **Unit Tests:** Each class has an associated unit test. If you change a class, you must also modify the unit test.
- **Code Reviews:** Each branch in the central version control system has owners. In order to commit code to that branch, you must have your code reviewed and approved by one of the owners. This ensures code quality.
- **Continuous Builds:** There is often a machine that continuously checks out and tests the latest code. All unit and regression tests are run and the status is made public to the team. The status contains information about the whether the code was built successfully, whether all unit and regression tests passed, and a list of the last few commits that were made to the branch. This ensures developers try to submit good code since if you break something, everyone knows about it :)
- **One-button Deploy:** If all tests have passed, one should be able to deploy to production with one command.
- **Back Button:** Systems should be designed so that it's possible to roll back changes.

Software Testing, Quality Assurance and Maintenance Winter 2019

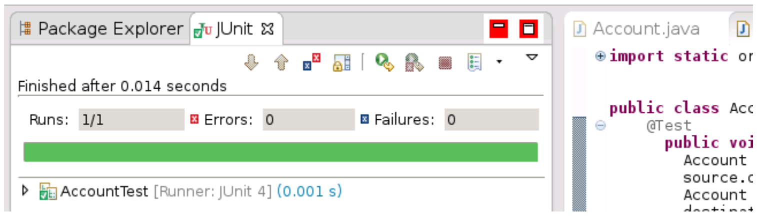
Lecture 23 — March 13, 2019

Patrick Lam *version 1*

We will see techniques for improving test design today, particularly with respect to verifying results.

Reference: Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. [Highly recommended. Available in the DC library.]

Goal. Well-designed tests are *self-checking*. That means that if the test runs with no errors and no failures (and hence produces a green bar in your IDE), we know that the test was successful.



Finished after 0.014 seconds

Runs: 1/1 Errors: 0 Failures: 0

AccountTest [Runner: JUnit 4] (0.001 s)

```
import static or

public class Acc
    @Test
    public voi
        Account
        source.c
        Account
```

Writing self-checking tests means that the tests automatically report the status of the code. This enables a "keep the bar green" coding style. Implications: 1) you can worry less about introducing bugs (but still take ordinary care); and 2) the tests help document your system's specs.

How to write self-checking tests. One might think:

> "Isn't it just calling asserts?"

Sadly, no. That's not enough.

Two questions about actually deploying asserts:

- Q: what for?
  A: check method call results
- Q: where?
  A: usually after calling SUT (System Under Test)

Counter example. Here's some example code.

```java
public class Counter {
    int count;

    public int getCount() { return count; }
    public void addToCount(int n) { count += n; }
}
```

We can test it with the following JUnit test.

```java
// java -cp /usr/share/java/junit4.jar:. org.junit.runner.JUnitCore CounterTest
import static org.junit.Assert.*;
import org.junit.Test;

public class CounterTest {
  @org.junit.Test
  public void add10() {
      Counter c = new Counter();
      c.addToCount(10);
      // after calling SUT, read off results
      assertEquals("value", 10, c.getCount());
  }
}
```

What kind of test is this? Let's consider two kinds of tests: state-based tests vs. behaviour-based tests.

- **State:** e.g. object field values. Verify by calling accessor methods.
- **Behaviour:** which calls SUT makes. Verify by inserting observation points, monitoring interactions.

Does Counter Test verify state or behaviour?

Flight example. Here's more example code.

```java
// Meszaros, p. 471
// not self-checking
public void testRemoveFlightLogging_NSC() {
 // setup:
 FlightDto expectedFlightDto=createRegisteredFlight();
 FlightMgmtFacade=new FlightMgmtFacadeImpl();
 // exercise:
 facade.removeFlight(expectedFlightDto.getFlightNo());
 // verify:
 // have not found a way to verify the outcome yet
 //  Log contains record of Flight removal
}
```

Implementing State Verification

We can verify state:

```
// Meszaros, p. 471
// extended state specification
public void testRemoveFlightLogging_NSC() {
 // setup:
 FlightDto expectedFlightDto=createRegisteredFlight();
 FlightMgmtFacade=new FlightMgmtFacadeImpl();
 // exercise:
 facade.removeFlight(expectedFlightDto.getFlightNo());
 // verify:
 assertFalse("flight still exists after removed",
             facade.flightExists(expectedFlightDto, getFlightNo()));
}
```

Note that we are exercising the SUT, verifying state, and checking return values.

In state-based tests, we inspect only outputs, and only call methods from SUT. We do not instrument the SUT. We do not check interactions.

You have two options for verifying state:

1. procedural (bunch of asserts); or,
2. via expected objects (stay tuned).

Returning to the flight example:

- We do check that the flight got removed.
- We don't check that the removal got logged.
- Hard to check state and observe logging.
- Solution: Spy on SUT behaviour.

Implementing Procedural Behaviour Verification

Or, we can implement behaviour verification. This is one way to do so, behaviourally:

```java
// Meszaros, p. 472
// procedural behaviour verification
public void testRemoveFlightLogging_PBV() {
 // fixture setup:
 FlightDto expectedFlightDto=createRegisteredFlight();
 FlightMgmtFacade=new FlightMgmtFacadeImpl();
 // test double setup:
 AuditLogSpy logSpy = new AuditLogSpy();
 facade.setAuditLog(logSpy);
 // exercise:
 facade.removeFlight(expectedFlightDto.getFlightNo());
 // verify:
 assertEquals("number of calls",
              1, logSpy.getNumberOfCalls());
 // ...
 assertEquals("detail",
              expectedFlightDto.getFlightNumber(),
              logSpy.getDetail());
}
```

As an alternative, we can use a mock object framework (e.g. JMock) to define expected behaviour.

Idea. Observe calls to the logger, make sure right calls happen.

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 for doing this:

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

How to Improve Your Tests

Next, we'll talk about some techniques for improving your test cases. Some tests are just better designed than others, making them easier to maintain and to understand. Applying these techniques will help.

Reducing Test Code Duplication. Copy-pasting is common when writing tests. This results in duplicate code in test cases, which has some undesirable side effects (bloat, unnecessary asserts). We'll talk about some techniques to mitigate duplication:

- Expected Objects
- Custom Assertions
- Verification Methods

Let's start with an example. One might expect many test methods like this one.

```
// Meszaros, p115
public void testInvoice_addLineItem7() {
  LineItem expItem = new LineItem(...);
  inv.addItemQuantity(product, QUANTITY);
  List lineItems = inv.getLineItems();
  LineItem actual = (LineItem) lineItems.get(0);
  assertEquals(expItem.getInv(), actual.getInv());
  assertEquals(expItem.getProd(), actual.getProd());
  assertEquals(expItem.getQuantity(),
               actual.getQuantity());
}
```

Using an Expected Object. We can compare objects instead:

```
// Meszaros, p115
public void testInvoice_addLineItem8() {
  LineItem expItem = new LineItem(...);
  inv.addItemQuantity(product, QUANTITY);
  List lineItems = inv.getLineItems();
  LineItem actual = (LineItem) lineItems.get(0);
  assertEquals("Item", expItem, actual);
}
```

What we need:

- a way to create the Expected Object;
- a suitable `equals()` method.

Here are some potential barriers:

- we might need a special `equals()` method,
  e.g. to compare subset of fields; or,
- we may only have an `equals()` that checks identity; or,
- we can't create the desired expected object.

Some solutions:

- we can create a custom assertion; or,
- we can provide special `equals()` on expected object.

Custom Assertions Example. Consider the following code:

```
// Meszaros, p116
static void assertLineItemsEqual(String msg, LineItem exp, LineItem act) {
  assertEquals(msg+" Inv", expItem.getInv(),
               actual.getInv());
  assertEquals(msg+" Prod", expItem.getProd(),
               actual.getProd());
  assertEquals(msg+" Qty", expItem.getQuantity(),
               actual.getQuantity());
}
```

Tips:

- Pick a good, declarative name.
- Create the custom assertion by refactoring, using usual techniques.

Benefits of Custom Assertions. Writing custom assertions can help with your test design. They:

- hide irrelevant detail;
- label actions with a good name (names are super important); and
- are themselves testable;

Variant: Outcome-describing Verification Method. Instead of using a custom assertion, you might use a verification method, like this one:

```java
// Meszaros, p117
static void assertInvoiceContainsOnlyThisLineItem(Invoice inv, LineItem exp) {
  List lineItems = inv.getLineItems();
  assertEquals("number of items", lineItems.size(), 1);
  LineItem actual = (LineItem)lineItems.get(0);
  assertLineItemsEqual("", expItem, actual);
}
```

Note that the verification method also interacts with SUT, but may have arbitrary parameters.

Going Further: Parameterized, Data-Driven Tests. While we're at it, there might be entire tests that differ only in input data.

Concrete tests invoke parametrized tests.

Other Best Practices for Tests

Avoid logic in tests.

The root problem is that tests are pretty much untestable. If you including ifs and loops in tests, you're asking for trouble. How are you going to make sure they're right?

Conditionals. For example,

```
// BAD
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
  // ...
} else {
  fail("Invoice should have exactly 1 line item");
}
```

Instead, do this:

```
// GOOD
List lineItems = invoice.getLineItems();
// (guard assertion:)
assertEquals("number of items", lineItems.size(), 1);
// ... proceed as before
```

The guard keeps you out of trouble.

Loops. Don't put loops directly in tests. Use a well-named, testable Test Utility Method instead.

Summary

In this lecture, we saw practical techniques for writing tests. This included techniques for result verification (using state verification and behaviour verification), as well as techniques for improving your tests by reducing duplication and by simplifying your tests.

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 24 — March 18, 2019

Patrick Lam *version 2*

We'll continue discussing how to engineer your test suites today. In particular, we'll see more details on how to make behaviour verification actually happen (using mock objects); we'll discuss the bane of flaky tests; and we'll talk about continuous integration.

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.

```
public class MailServiceStub implements MailService {
  private List<Message> messages = new ArrayList<Message>();
  public void send (Message msg) {
    messages.add(msg);
  }
  public int numberSent() {
    return messages.size();
  }
}
```

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

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

This is state verification because it's checking the contents of memory (which should reflect interactions that have happened in the past). 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.

```
@Test
public void testAddDocument() {
  // ** recording phase **
  // expect document addition
  mock.documentAdded("Document");
  // expect to be asked to vote for document removal, and vote for it
  expect(mock.voteForRemoval("Document"))
             .andReturn((byte) 42);
  // expect document removal
  mock.documentRemoved("Document");
  replay(mock);
  // ** replaying phase ** we expect the recorded actions to happen
  classUnderTest.addDocument("New Document", new byte[0]);
  // check that the behaviour actually happened:
  verify(mock);
}
```

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 your Operating Systems course.

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.

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 25 — March 20, 2019

Patrick Lam *version 2*

More on Flaky Tests

Google also has a blog post about their experience with flaky tests. It provides a different perspective on flaky tests.

`https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html`

Here are some useful facts from that post:

- Rate of flakiness: 1.5% of all test runs. Over time, their flaky test insertion rate is about the same as the flaky test removal rate. The number of *tests* which are occasionally flaky is 16%.
- Code can only be submitted after it passes tests. There are also pre-release test suites which must succeed before the project can be released. This is presumably after correcting for flakiness.
- The vast majority (84%) of post-submit test failures in continuous integration are associated with flaky tests.
- The flaky test rate means that if you have 1000 pre-release tests, you should expect 15 failing tests which require manual investigation. If you ignore them, you might well be missing an actual problem.
- Mitigation: in addition to what I mentioned last time (automatic re-running), Google also allows re-running only flaky tests. Known flaky tests get automatically re-run 3 times before reporting a fail. This slows down notification of test failures, too. Also, tests that are seen to be flaky are automatically quarantined (not run every time; bug reported).

Case Study: Mocks/Stubs

Next, we'll summarize another Google case study, this time about mocks/stubs.

`https://testing.googleblog.com/2016/11/what-test-engineers-do-at-google.html`

This blog entry describes the work of a Google Test Engineer in improving test infrastructure.

Situation. Legacy system. Flaky tests: large and brittle. End-to-end tests were difficult to introduce fakes into. System used many external dependencies.

First false start. Splitting the end-to-end tests. Didn't work: would have needed refactoring for entire legacy system, not just the part the author's team was working on.

Second false start. Mock services that were not actually required. Not viable: dependencies changed often. Imposed test maintenance cost.

Actual solution. Replace the client code (which calls depended-on services) by unit tests of calls to RPC stubs. Stub implemented with mock objects. Then, in another test, send the data to the actual service (i.e. test the tests).

The benefits are much faster tests (10× speedup: from 30 minutes to 3 minutes) which are not at all flaky and which can run on developer machines.

Continuous Integration

This is not a complicated concept. Literally, continuous integration requires you to use a single shared master branch with your development teams. That is integration because you're merging your changes into master, and continuous because you're doing it all the time.

Why CI Is Awesome. Before CI, people could be stuck integrating changes for months (or longer!) after each team finishes developing their change. This is not good. Instead, your software always stays in a working state.

Necessary CI Practices. You can't just do CI, though. It's not new, but there was a time before CI, because the infrastructure didn't exist yet. You need *continuous builds* and *test automation* to make CI work (and, of course, a source control repository). You can then use CI to do Continous Deployment. Here's how CI works.

1. You clone the repo (which works).
2. You make your changes.
3. You commit and push your changes (often!)
4. A machine pulls the changes, compiles them, and runs automated tests.
5. Everyone knows whether your changes passed tests or not.

Continuous Deployment is a minor variant to CI where the production machines also pull changes as soon as the tests pass, and deploy them.

Key Details. To make Continuous Integration work, you'll probably have to follow these practices as well:

- Fix broken builds immediately! (Commits shouldn't even be accepted if they don't compile; test cases should start immediately and fixing them is a high priority task).
- Keep the build fast (minutes): parallelize the build and tests. Deploy tiered tests: fast tests run first, then slower, more detailed tests.
- Test in a production-like environment. Virtual machines are helpful here.

References about Continuous Integration.
Bullet points from Gitlab: `about.gitlab.com/2015/02/03/7-reasons-why-you-should-be-using-ci/`

Mid-length article from Atlassian:
`www.atlassian.com/agile/continuous-integration`

Longer article by Martin Fowler:
`martinfowler.com/articles/continuousIntegration.html`

Serverless CI:
`medium.com/@hichaelmart/lambci-4c3e29d6599b`

Airbnb Testing Infrastructure

Let's change it up here and talk about Airbnb instead.

`http://nerds.airbnb.com/testing-at-airbnb/`

This describes how the author worked with colleagues to introduce a (developer-based) testing culture at Airbnb.

Running tests locally. As I mentioned earlier, running tests in prod-like environments helps a lot. Airbnb did so using Vagrant boxes ("ready to run tests out of the box.") Airbnb runs on Ruby on Rails and they use Zeus to allow developers to start up the Rails environment super quickly.

Continuous Integration. They also use Solano for Continuous Integration (as described in the L24 notes). Tests run at Airbnb (not in the public cloud), in parallel for improved throughput.

Furthermore, their Github displays the build/test status of every pull request, and presumably developers only merge pull requests that pass the tests.

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 26 — March 25, 2019

Patrick Lam *version 1*

Code Review

Code review is a powerful tool for improving code quality. Today's lecture is based on the following references:

- course reading on code review (main source):
  `http://web.mit.edu/6.031/www/sp17/classes/04-code-review/`
- how code review works in an MIT course on software construction:
  `http://web.mit.edu/6.031/www/sp17/general/code-review.html`
- Fog Creek code review checklist:
  `https://blog.fogcreek.com/increase-defect-detection-with-our-code-review-checklist-example/`

Our course includes code review as part of Assignment 3. The MIT offering is more comprehensive and requires students to respond to code reviews as well.

Code review is a communication-intensive activity. A reviewer needs to 1) read someone else's code and 2) communicate suggestions to the author of that code. We sometimes think that communicating with the computer is our primary goal when programming, but communicating with other people is at least as important over the long run.

Purpose of code review. The communication inherent in code review aims to improve both the code itself as well as the author of the code. Good code review can give timely information to developers about the context in which their code operates, particularly the project and best-practices uses of the language.

We'll continue with a list of items that you are inspecting when you do a code review.

Formatting. Consistency in formatting helps avoid preventable errors. Positioning of { }s isn't something that necessarily has one right answer. Spaces are probably better than tabs. But the most important thing is to be self-consistent with yourself and within your project.

Code smell example 1.

The following code has a number of bad smells:

```java
public static int dayOfYear(int month, int dayOfMonth, int year) {
    if (month == 2) {
        dayOfMonth += 31;
    } else if (month == 3) {
        dayOfMonth += 59;
    } else if (month == 4) {
        dayOfMonth += 90;
    } else if (month == 5) {
        dayOfMonth += 31 + 28 + 31 + 30;
    } else if (month == 6) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31;
    } else if (month == 7) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;
    }
    // ... through month == 12
    return dayOfMonth;
}
```

Let's go through some of the bad smells.

- **Don't Repeat Yourself**. A few weeks ago in Lecture 23, we talked about code cloning and how it's not always bad. Sometimes it is bad, as in the code above. The usual reason for it being bad is that fixes in one place may remain unfixed in the other place. (Recall: it wasn't always bad when used for forking and templating). For instance, if February actually had 30 days, you'd need to change a lot of code.

- **Fail Fast.** In the language of 6.031, we mean that a defect should be caught closest to when it's written. Static checks, as performed in compilers, catch defects earlier than dynamic checks, which catch defects earlier than letting wrong values percolate in the program state. In this particular example, there are no checks ensuring that a user had not permuted `month` and `dayOfMonth`.

- **Avoid Magic Numbers.** The above code is full of magic numbers. Particularly magical numbers include the `59` and `90` examples, as well as the month lengths and the month numbers. Instead, use names like `FEBRUARY` etc. Enums are a good way to encode months, and days-of-months should be in an array. The `59` should really be `31 + 28`, or better yet, `MONTH_LENGTH[JANUARY] + MONTH_LENGTH[FEBRUARY]`.

- **One Purpose Per Variable.** The specific variable that's being re-used in the above example is `dayOfMonth`, but this also applies to variables that you might use in your method. Use different variables for different purposes. They don't cost anything. Best to make method parameters `final` and hence non-modifiable.

Comments and code documentation

Code should, ideally, be self-documenting, with good names for classes, methods, and variables. Methods should come with specifications in the form of Javadoc comments, e.g.

```java
/**
 * Compute the hailstone sequence.
 * See http://en.wikipedia.org/wiki/Collatz_conjecture#Statement_of_the_problem
 * @param n starting number of sequence; requires n > 0.
 * @return the hailstone sequence starting at n and ending with 1.
 *         For example, hailstone(3)=[3,10,5,16,8,4,2,1].
 */
public static List<Integer> hailstoneSequence(int n) {
    ...
}
```

Note how this comment describes what the method does, in one sentence, provides context, and then describes the parameters and return values.

Also, when you incorporate code from other sources, cite the sources. For instance, in the A2 `index.html` file:

```
    // adapted from Eli Bendersky's Lexer: http://eli.thegreenplace.net/2013/07/16/hand
        -written-lexer-in-javascript-compared-to-the-regex-based-ones
    // public domain according to author
    // modifications by Patrick Lam
```

This helps, for instance, when the source is later updated, and is the right thing to do in terms of IP (assuming, of course, that your use of the software is allowed by its license).

Don't write comments that don't contribute to code understanding. If it's blatantly obvious from the code, it shouldn't be a comment. Such comments can mislead and hence do more harm than good.

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 27 — March 25, 2019

Patrick Lam *version 1.1*

Reporting Bugs

Goal of reporting bugs:

> Get bugs fixed. Which bugs? The most important ones—the right ones.

How?[1][2]

Properties of Important Bugs

- Bug is sufficiently general to affect many users (easily reproducible).
- Bug has severe consequences (crashes, dataloss).
- Bug is new to most recent version.
- Bug has security implications.

Reproducibility

Developers can't fix problems they can't observe.

- Be maximally specific in describing steps to reproduce.
- Write the steps down as soon as possible, before you forget.
- Try to find a minimal testcase that demonstrates the problem.

[1] `http://blog.cleverelephant.ca/2010/03/how-to-get-your-bug-fixed.html`
[2] `http://blog.threepress.org/2009/11/17/how-to-get-your-bug-fixed/`

Anatomy of a Bug Report

Bug report formats are fairly standard. Here are some fields in a Bugzilla report.

Standard bookkeeping fields:

- Bug id: automatically generated number
- Reporter: usually an email address
- Product, Version, Component: helps direct the bug to the right developers
- Platform, OS: e.g. PC/Linux, or Mac/Mac OS X 10.5.
- Severity: based on consequences; examples: blocker, critical, normal, trivial, enhancement
- Assign to: person responsible for bug
- CC: list of people who track bug changes
- Keywords: e.g. crash, intl, patch, security
- Depends on/blocks: relationships between bugs
- URL: (especially applicable for browsers)
- Attachments: e.g. test cases/input files which exhibit the bug

Summary. Perhaps the most critical field: a one-line recap of the bug. Enables searching for and judgement of the bug.

Examples (some good, some bad):

- "RPM 4 installer crashes if launched on Red Hat 6.2 (RPM 3) system"
- "Back button does not work"
- "When memory cache disabled, no-store pages not displayed at all"
- "History and bookmarks completely inoperable"
- "Can't install"

Description. Should be a complete description of the bug, including:

- Overview: expanded summary, e.g. "Drag-selecting any page crashes Mac builds in NSGetFactory".
- Steps to Reproduce (key!): minimized easy-to-follow steps to trigger the bug; 1) try to reproduce the bug based on the steps you report; 2) try to describe the steps to that anyone (like the developer) can reproduce the bug.
  Be specific: instead of "save the document", "File > Save, select foo from dropdown, ...".
- Actual Results: what you see when you perform the steps to reproduce. e.g. "Application crashes. Stack trace included."
- Expected Results: what you think is correct
- Build Date and Platform: on development software, helps find the bug; include additional builds and platforms the bug might apply to.

Lifecycle-related fields. Some fields summarize the current state of the bug.

- Comments: either by the assigned developer, original reporter, or interested bystanders. Often people give additional information ("also happens on latest build") or help diagnose the problem.
- Status: e.g. UNCONFIRMED, NEW, REOPENED, VERIFIED
- Priority: for internal use by development team.

Properties of Good Bug Reports

- Reported in the database.
- Simple: one bug per report.
- Understandable, minimal, and generalizable.
- Reproducible.
- Non-judgemental. "The developers are all morons".
- Not a duplicate.

Bug Triage

Some of the fields we've seen help in identifying the most important bugs (which ones?). Consider this approach for assigning a single number to a bug which summarizes its importants ("user pain"):

`http://lostgarden.com/2008/05/improving-bug-triage-with-user-pain.html`

The main attributes are type, likelihood, and priority, all evaluated on anchored scales on which staff are calibrated.

Research: What Makes a Good Bug Report?

Betternburg et al performed a survey asking experienced developers to rate bug reports and to identify important information in them [BJS+08], published in Foundations of Software Engineering 2008.

I recommend consulting the full paper. But the executive summary is that developers rated steps to reproduce (83%), stack traces (57%) and test cases (51%) as most useful. Furthermore, by examining data from Apache projects, Eclipse, and Mozilla, they found that bug reports with stack traces get fixed sooner, and that bug reports that are easier to read have lower lifetimes. Code samples also help increase the chance that a bug report gets fixed.

References

[BJS+08] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of the 16th International Symposium on Foundations of Software Engineering*, November 2008.

[Kan] Cem Kaner. Bug advocacy. Retrieved from: `http://www.kaner.com/pdfs/bugadvoc.pdf`. Retrieved March 12, 2015.

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 28 — March 25/27, 2019

Patrick Lam *version 1*

Additional note about "fail fast" from last week: often, software reads configuration in the initialization phase and then acts on it later. Example: a monitoring system calls "dial" executable to report an issue. It's better if the monitoring system reports that its path for "dial" doesn't work upon first load, rather than after there is an issue. Because if "dial" isn't there, you have the original issue, but you're probably not looking at the system, so the system can't tell you that there's that issue. Plus "dial" not being there is another issue.

Static code analysis: PMD

We'll dive into static code analysis by talking about one particular code analysis tool, PMD[1]. Last week, we said that it was better to use tools to flag style issues. PMD is one way to do so.

PMD out of the box: built-in rulesets

The easiest way to use PMD is in an IDE with its built-in rulesets. It has rulesets for languages from C++ to Scala, including Java. For Java there are a number of rulesets, which group related rules. Let's look at a few examples of rules.

- SimplifyConditional: [design ruleset] detect redundant null checks

```
class Foo {
  void bar(Object x) {
    if (x != null && x instanceof Bar) {
      // just drop the "x != null" check
    }
  }
}
```

  Note that this code is not wrong. It's just redundant.

- UseCollectionIsEmpty: [design ruleset] better to use `c.isEmpty()` rather than `c.size() == 0`

```
class Foo {
    void good() {
        List foo = getList();
        if (foo.isEmpty()) { /* blah */ }
    }

    void bad() {
        List foo = getList();
```

[1]`pmd.github.io`

```
        if (foo.size() == 0) { /* blah */ }
    }
}
```

Again, it's not wrong to call `size()` and see if the result is 0. It's just more idiomatic, and sometimes more efficient, to check `isEmpty()`.

- MisplacedNullCheck: [basic ruleset] don't check nullness after relying on non-nullness

```
public class Foo {
    void bar() { if (a.equals(baz) || a == null) {}     }
}
```

The check `a == null` is never going to succeed, because `a.equals()` would throw a `NullPointerException` instead. So if `a` can ever be null, there is a fault.

- UseNotifyAllInsteadOfNotify: [design ruleset] most of the time, `notifyAll()` is the right call to use, not `notify()`. Unless you know what you're doing, using `notify()` is going to result in a bunch of stuck threads, which is a bug.

Find more about the above rules at `https://pmd.github.io/pmd-5.5.4/pmd-java/rules/java/design.html`. The pages on the PMD site are also useful for your assignment as sample code.

I've included examples from the design and basic rulesets. There's a ruleset specifically for JUnit, rule sets detecting empty or otherwise useless code, naming conventions, and much more.

Linking back to last week's material: PMD and tools like it can tell you about things that may be wrong, or that are certainly wrong. However, they cannot tell you about how important that wrongness is. We still need (experienced!) human judgment to know that.

Writing your own PMD rules

Assignment 3 Question 1 asks you to write your own PMD rule. So we'll talk about how to write PMD rules. The intellectual core of a PMD rule is a query on the Abstract Syntax Tree (AST). You can use either Java or XPath to describe this query. XPath is cleaner, in that it's a declarative query language.

Here are some links about how to make rule sets and the boilerplate you need for rules:

- `https://pmd.github.io/latest/customizing/howtomakearuleset.html`
- `https://pmd.github.io/latest/customizing/howtowritearule.html`

We'll be focussing on what goes into the rule itself, as per `https://pmd.github.io/latest/customizing/xpathruletutorial.html`. The tutorial skips a lot of detail about how to actually use XPath. So let's start with that.

XPath. Let's start from the fundamentals. You write *selectors* to find nodes. We'll look at a simple XML document. XPath also applies to web programming (in particular the Document Object Model) and also to the Java code we'll be analyzing. Source: `https://www.w3schools.com/xml/xpath_syntax.asp`; specification: `https://www.w3.org/TR/xpath/`.

Here is an XML file:

```xml
<?xml version="1.0" encoding="UTF-8"?>

<bookstore>
 <book>
  <title lang="fr">Harry Potter</title>
  <price>29.99</price>
 </book>
 <book>
  <title lang="en">Learning XML</title>
  <price>39.95</price>
 </book>
</bookstore>
```

You can observe the tree structure of the file. And you can play along with either `https://www.w3schools.com/xml/tryit.asp?filename=try_xpath_select_cdnodes` (which is hardcoded to XML similar to the above) or else `http://www.freeformatter.com/xpath-tester.html`.

Consider XPath expression `//price`. The result is the set of price nodes with data `29.99`, `39.95`. So, expression `//price` selects nodes with name `price`; the `//` means any descendants (including self) of the context node (= root node, here)—we asked for all descendants of the root named `price`. And, `count(//price)` counts the number of `price` elements in the tree.

We can also specify an exact path through the tree, say with `/bookstore/book[1]/title`. This starts at the root, visits the `bookstore` element, then its first `book` child, then returns the title. If we omitted `[1]`, then we'd get all of the titles.

We can also select all elements that satisfy some condition, e.g. `/bookstore/book[price>35]/title` selects the titles of books with price greater than 35.

Note the `lang` attribute. We can select elements with a certain value for `lang`: `//title[@lang="fr"]`.

In general, square brackets can contain predicates. We've seen pretty simple ones, but you can put arbitrary tree queries, e.g. `//price[../title[text()="Learning XML"]]`.

You can also combine predicates with `and`, `or`, etc. e.g. `//title[../price < 35 or @lang="en"]`. `*` works as you might expect.

The double-slash `//` includes descendants and self. If you want descendants excluding self, write e.g. `descendant::book` as part of your expression. For the above example, there's no difference, but you can see it here:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
 <book><book><title lang="fr">Harry Potter</title></book></book>
 <book><title lang="en">Learning XML</title></book>
</bookstore>
```

UPDATED: Try `//book[descendant::book]` versus `//book[//book]`. I used `descendant::` in my solution, but you may be able to avoid it.

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 29 — March 25/27, 2019

Patrick Lam *version 1*

Using XPath to write PMD Rules

Last time, we saw how some notions on how to use XPath in general. Today, we'll talk about XPath and PMD.

Here's a rule ensuring that while statements must not contain directly contain other statements:

`//WhileStatement[not(Statement/Block)]`

This rule matches `WhileStatement`s whose `Statement` child does not contain a `Block`.

See `https://pmd.github.io/latest/customizing/xpathruletutorial.html` for more examples. Let's consider this example, detecting definitions of `Logger` variables.

```
public class a {
   Logger log1 = null;
   Logger log2 = null;
   int b;

   void myMethod() {
      Logger log = null;
      int a;
   }
   class c {
      Logger a;
      Logger b;
   }
}
```

First, we can detect `Logger` variables. Note the use of `@Image`. We are matching `VariableDeclarator`s whose type is `Logger`:

```
//VariableDeclarator[../Type/ReferenceType/ClassOrInterfaceType[@Image='Logger']]
```

But we really want class definitions which contain more than one variable of type `Logger`.

```
//ClassOrInterfaceDeclaration
    [count(.//VariableDeclarator
      [../Type/ReferenceType/ClassOrInterfaceType[@Image='Logger']])>1]
```

You can use the PMD Designer to debug your rules, as I'll demo in class. The PMD Designer allows you to browse the AST and try out your queries as you build them.

Hints for Assignment 3. I think I've talked about everything you need to use, except for the `starts-with` function that you can use for predicates. Everything else you can find from the PMD Designer. Recall that you're matching test methods that do *not* have a call to `mockCommandSender.getLastMessage` in an assert.

The Landscape of Testing and Static Analysis Tools

Here's a survey of your options:

- manual testing;
- running a JUnit test suite, manually generated;
- running automatically-generated tests;
- running static analysis tools.

We'll examine several points on this continuum. Some examples:

- Coverity: a static analysis tool used by 900+ companies, including BlackBerry, Mozilla, etc.
- Microsoft requires Windows device drivers to pass their Static Driver Verifier for certification.

Using Linters

We will also talk about linters in this lecture, based on Jamie Wong's blog post `jamie-wong.com/2015/02/02/linters-as-invariants/`.

First there was C. In statically-typed languages, like C,

```c
#include <stdio.h>

int main() {
  printf("%d\n", num);
  return 0;
}
```

the compiler saves you from yourself. The guaranteed invariant:

"if code compiles, all symbols resolve."

Less-nice languages. OK, so you try to run that in JavaScript and it crashes right away. Invariant?

"if code runs, all symbols resolve?"

But what about this:

```javascript
function main(x) {
  if (x) {
    console.log("Yay");
  } else {
    console.log(num);
  }
}

main(true);
```

Nope! The above invariant doesn't work.

OK, what about this invariant:

"if code runs without crashing, all symbols referenced in the code path executed resolve?"

Nope!

```javascript
function main() {
  try {
    console.log(num);
  } catch (err) {
    console.log("nothing to see here");
  }
}

main();
```

So, when you're working in JavaScript and maintaining old code, you always have to deduce:

- is this variable defined?
- is this variable always defined?
- do I need to load a script to define that variable?

We have computers. They're powerful. Why is this the developer's problem?!

Solution: Linters.

```javascript
1  //jshint undef:true, devel:true
2
3  function main(x) {
4    "use strict";
5    if (x) {
6      console.log("Yay");
7    } else {
8      console.log(num);
9    }
10 }
11
12 main(true);
```

Now:

```
$ jshint lintee.js
lintee.js: line 8, col 17, 'num' is not defined.

1 error
```

Invariant:

"If code passes JSHint, all top-level symbols resolve."

Strengthening the Invariant. Can we do better? How about adding a pre-commit hook?

"If code is checked-in and commit hook ran,
all top-level symbols resolve."

Of course, sometimes the commit hook didn't run. Better yet:

- Block deploys on test failures.

Better invariant.

"If code is deployed,
all top-level symbols resolve."

Even better yet. It is hard to tell whether code is deployed or not. Use git feature branches, merge when deployed.

"If code is in master,
all top-level symbols resolve."

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 30 — April 1, 2019

Patrick Lam *version 1*

We've seen two tools so far: PMD and jshint. These tools both operate on Abstract Syntax Trees and ensure relatively shallow program properties. Out of the box, PMD guarantees generic properties (but we also talked about writing our own checkers).

jshint, PMD, and other tools (like FindBugs, below) enforce generic rules that all programs should satisfy. You can read a comparison of different tools in this paper:

`www.cs.umd.edu/~jfoster/papers/issre04.pdf`

FindBugs. This tool is somewhat deeper than PMD. FindBugs is an open-source static *bytecode* analyzer for Java out of the University of Maryland. A key difference is that it performs static analysis at Java bytecode level rather than AST level. It's therefore harder to write FindBugs rules.

`findbugs.sourceforge.net`

FindBugs finds bug patterns like:

- off-by-one;
- null pointer dereference;
- ignored `read()` return value;
- ignored return value (immutable classes);
- uninitialized read in constructor;
- and more…

Such patterns are typically easier to evaluate at bytecode level because the variability of the AST has been compiled away.

False positives. FindBugs, like all static analysis tools, gives some false positives. (The course project has a question about false positives.) In general, they occur because the analysis tool is not powerful enough. Because of the halting problem, there can be no all-powerful tool.

Consider this case:

```
try { socket.close(); }
catch (Exception ignore) {}

try { reader.close(); }
catch (Exception ignore) {}
```

FindBugs, of course, declares "this method might ignore an exception", but it's fine in this case, since there's nothing that the program needs to do about failed close actions.

Here are some techniques to help avoid false positives:

`patricklam.ca/papers/14.msr.saa.pdf`

Beyond hard-coded rules

Inferring specifications: Coverity Static Analyzer. This industrial-strength tool (statically) identifies bugs in C/C++, Java, and C# codebases. It claims to scale to "hundreds of users, thousands of defects, and millions of lines of code in a single analysis." It does so by inferring must-beliefs and may-beliefs from the code base, as we've discussed in Lecture 22. Coverity does a lot of work to keep the false positive rate low. Your project reproduces some of the key technology behind Coverity.

Coverity is a commercial product which can find many bugs in large (millions of lines) programs; it is therefore a leading company in building bug detection tools. Clients (900+) include organizations such as Blackberry, Yahoo, Mozilla, MySQL, McAfee, ECI Telecom, Samsung, Siemens, Synopsys, NetApp, Akamai, etc. These include domains including EDA, storage, security, networking, government (NASA, JPL), embedded systems, business applications, operating systems, and open source software. We have access to Coverity for this course, but there's also a free trial:

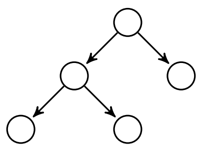
`http://softwareintegrity.coverity.com/FreeTrialWebSite.html`

Developer-provided specifications

A number of research-quality tools aim to enforce developer-provided specifications. Developers should be aware of how their software works. Specification languages enable developers to express this knowledge in a machine-readable way and tools can enforce that the software satisfies its specification. We'll discuss some specification-based tools below.

Korat (University of Illinois). Key Idea: Generate Java objects from a representation invariant specification written as a Java method.

For instance, here's a binary tree:



One characteristic of a binary tree:

- left & right pointers don't refer to same node.

We can express that characteristic in Java as follows:

```java
boolean repOk() {
  if (root == null) return size == 0;                // empty tree has size 0
  Set visited = new HashSet(); visited.add(root);
  List workList = new LinkedList(); workList.add(root);
  while (!workList.isEmpty()) {
    Node current = (Node)workList.removeFirst();
    if (current.left != null) {
      if (!visited.add(current.left)) return false; // acyclicity
      workList.add(current.left);
    }
    if (current.right != null) {
      if (!visited.add(current.right)) return false; // acyclicity
      workList.add(current.right);
    }
  }
  if (visited.size() != size) return false;          // consistency of size
  return true;
}
```

Korat then generates all distinct ("non-isomorphic") trees, up to a given size (say 3). It uses these trees as inputs for testing the `add()` method of the tree (or for any other methods.)

`korat.sourceforge.net/index.html`

Lecture 31 — April 3, 2019

Patrick Lam *version 1*

Facebook Infer: Deeper, but still generic, properties

We'll see one more tool, the Facebook Infer static analyzer tool. This tool is particularly important because it is both open-source and designed to work on industrial-sized codebases in a number of languages including C, Objective-C, C++, and Java (including Android code). Facebook uses Infer on its code.

Infer's open-source nature enables others to write their own analyses, both generic and specific to particular libraries. Infer was designed to be easily extensible and applicable to multiple languages.

Try Infer in-browser (Java) here:

`https://codeboard.io/projects/11587?view=2.1-21.0-22.0`

You can find some talks about Infer here:

`https://code.facebook.com/projects/449931035189038/infer/`

Infer has linters (like PMD and FindBugs) but also more sophisticated interprocedural analyses which use separation logic (beyond the scope of this course).

Infer Eradicate: detecting null pointer dereferences. The goal of this subtool is to prevent null pointer exceptions. It does so by forcing treating Java references as not-null by default—that is, the tool ensures that the developer may not put `null` into an unannotated reference. It turns out, however, that it's quite difficult to program without `null`. So developers can mark a reference as `@Nullable`. For instance, Infer Eradicate complains that `s` might be null at the call to `length()`.

```java
class C {
  int getLength(@Nullable String s) {
    return s.length(); // Infer complains!
  }
}
```

Infer can use its flow-sensitive static analysis to guarantee that there are no null dereferences here:

```java
class C {
  int getLength(@Nullable String s) {
    if (s != null) { return s.length(); } else { return -1; }
    // Infer is happy.
  }
}
```

Eradicate checks a generic property—`null` may never be dereferenced—but uses annotations to help with the analysis.

Although the above examples are intraprocedural, Infer also knows enough to catch interprocedural errors, like this one:

```
struct Person {
  int age; // ... etc.
};
int get_age(struct Person *who) {
  return who->age;
}
int null_pointer_interproc() {
  struct Person *joe = 0;
  return get_age(joe);
}
```

Infer Analyzer: Leaks. Infer also statically detects both resource leaks in C and Java as well as memory leaks in languages like C and C++. Resources include files and sockets—entities that should generally be closed after they have been opened. For instance, Infer will complain about the following code:

```
void resource_leak_bug() {
    FILE *fp;
    fp=fopen("test.txt", "r"); // file opened and not closed.
}
```

That example is fairly straightforward for both dynamic and static approaches to find. But consider also this code:

```
  public static void foo () throws IOException {
    FileOutputStream fos = new FileOutputStream(new File("whatever.txt"));
    fos.write(7);   //DOH! What if exception?
    fos.close();
  }
```

It could happen that `fos.write()` might throw an exception (which is caught in a caller), thus leaving `fos` open. Such a leak is difficult to catch dynamically. On the other hand, static analyzers generally give you all possible leaks, including zillions that never actually happen.

By the way, you can fix the above bug in Java 7 by using try-with-resources:

```
  public static void foo() throws IOException {
    try (
      FileOutputStream fos = new FileOutputStream(new File("everwhat.txt"))
    ) {
      fos.write(7);
    }
  }
```

Note that try-without-resources works when the lifetime of the resource can be statically bounded to a lexical scope (in the above example, the single `write`) statement. Otherwise you might consider using `try`/`finally`.

Infer and tainting. Infer also supposedly contains an analysis to check for some security and privacy issues, although details are scarce. The general idea is to use a taint-based analysis. Such analyses label certain values as unsafe (e.g. data read from an untrusted source) or secret (e.g. preferences data). The analysis complains when tainted values may reach sensitive functions or the outside world.

Static versus Dynamic Analysis

We saw that Infer could detect memory and resource leaks statically. This is a good time to compare static and dynamic analyses; `valgrind` detects memory leaks dynamically. We'll do the comparison for JML and `jmlc` versus `ESC/Java` as well.

Recall that a *dynamic* analysis monitors program behaviour at runtime, whlie a *static* analysis reasons about the program text. We talked about this back in Lecture 4, but we're ready to provide more technical details now that we know more about testing and static analysis.

- dynamically: You have complete information about program state on observed executions.
- statically: You have partial information about all executions.

So how does that work out on specific examples?

Virtual method call resolution. The question here is: given a virtual method call like `m.foo()`, where the actual runtime type of `m` could vary (due to subclassing), which `foo()` method actually gets called?

Dynamically, it is trivial to answer this question. (How?)

Statically, this is quite difficult. The problem is that one needs to know the type of `m`. The easiest answer is by using Class Hierarchy Analysis: using the declared type of `m`, compute the allowed types using the class hierarchy (i.e. subclasses of the declared type). Rapid Type Analysis gives a better answer—it limits the answer to all types that are instantiated somewhere in the program. But that requires an approximation of the reachable code, which in turn requires the answer to the very question we're trying to answer. There are even more accurate algorithms which propagate type constraints through the program.

Checking data structures for cycles. Last time, we saw code to ensure that the tree was actually acyclic. That was fairly straightforward. Here again, static checks are difficult; they require the analysis to verify that the code maintains the acyclicity invariant through all possible executions.

Unreachable code. Not everything is easier to verify dynamically than statically. Consider the question of whether a line of code is reachable or not. This is relatively easy to approximate statically (that's what dead code elimination does), but quite hard to check dynamically, since it depends heavily on finding appropriate program inputs.

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 32 — April 3, 2019

Patrick Lam *version 1*

Dynamic Analyses

We'll continue talking about generic properties, but this time we'll talk about dynamic verification of these properties.

Memory errors

Let's start by talking about leaks and other memory errors. We saw static detection of leaks with Facebook Infer. Valgrind's Memcheck tool and Clang's Address Sanitizer detect memory errors dynamically.

This webpage compares different tools:

`https://github.com/google/sanitizers/wiki/AddressSanitizerComparisonOfMemoryTools`

Valgrind's Memcheck detects the following errors:

- Illegal reads/writes: Memcheck complains about accesses to memory that the program should not be accessing (e.g. not returned from a `malloc`, or below the stack pointer).
- Reads of uninitialized variables: Memcheck tells you about reads of memory that has never been initialized, if the program prints out the resulting values.
- Illegal/wrong frees: Of course, you're not allowed to free memory that you didn't get from a memory allocation function, so Memcheck tells you. It also tells you when you use `free()` on something you got from `new[]`.
- Overlapping source/destination for memory moves.
- Fishy argument values: You probably don't mean to request either -3 bytes or more than 2^{63} bytes.
- Memory leaks: Memcheck tells you when you have memory that didn't get freed but that you no longer have any pointers to.

You will pay a significant performance penalty when using Valgrind; Memcheck typically comes with a 10×–50× slowdown. This is usable for bug diagnosis but obviously not for production.

Here is more information on what Valgrind can do. It can do more than just Memcheck as well.

`http://maintainablecode.logdown.com/posts/245425-valgrind-is-not-a-leak-checker`

The AddressSanitizer tool, supported by both clang and gcc, also detects memory errors. These errors are similar to those that Valgrind can detect. The technology is different, but the goals are similar. Because it uses different implementation technology, it runs much more quickly than Valgrind, with a reported typical slowdown of 2×.

Find some information about AddressSanitizer here:

`http://btorpey.github.io/blog/2014/03/27/using-clangs-address-sanitizer/`

AddressSanitizer finds the following errors:

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Use-after-return
- Use-after-scope
- Double-free, invalid free
- Memory leaks (experimental)

Note that Valgrind may return false positives, while AddressSanitizer has a design goal of returning no false positives. In fact, ASan aborts the program whenever it encounters a memory problem. You're supposed to fix it before proceeding. Of course, ASan might miss some problems.

Implementation Techniques. Valgrind and ASan use different techniques, hence yield different results. Valgrind emulates a CPU and checks, at every memory access, whether that access is legitimate or not. ASan, on the other hand, rewrites relevant memory accesses at compile-time to call a checking library. It turns out that calling a library is cheaper than emulating the CPU.

Conceptually, ASan maintains shadow memory— metadata about where the program is allowed to access (or not). Then, it replaces the `malloc()` and `free()` calls with its own versions; these versions update shadow memory and indicate that allocated memory is OK to access, unallocated memory not OK. Finally, every memory access in the program gets replaced with a checked access:

```
if (IsPoisoned(address)) {
  ReportError(address, kAccessSize, kIsWrite);
}
*address = ...;  // or: ... = *address;
```

This is not so different from what Java does to check array accesses, for instance, but applies much more generally, to every memory access in the program.

More details about ASan:

`https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm`

Both of these tools are open-source. The source code is the ultimate description of the tools.

Race detectors

Valgrind also has a race detection tool, Helgrind. This tool dynamically detects memory that is concurrently accessed by two threads. Such accesses are fine as long as they are controlled by a lock. At runtime, Helgrind knows which locks are held. If the program does not hold enough locks, then Helgrind flags a memory error.

```
acquire(lock1);          acquire(lock2);
read(x);                 write(x); // different lock!
release(lock1);          release(lock2);
```

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:

```java
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.