Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 3 — January 7/9, 2019

Patrick Lam *version 0*

Exercise: findLast. Here's a faulty program.

```
1     static public int findLast(int[] x, int y) {
2         /* bug: loop test should be i >= 0 */
3         for (int i = x.length - 1; i > 0; i--) {
4             if (x[i] == y) {
5                 return i;
6             }
7         }
8         return -1;
9     }
```

You might expect to be asked a question like this:

(a) Identify the fault, and fix it.

(b) If possible, identify a test case that does not execute the fault.

(c) If possible, identify a test case that executes the fault, but does not result in an error state.

(d) If possible, identify a test case that results in an error, but not a failure. (Hint: program counter)

(e) For the given test case, identify the first error state. Be sure to describe the complete state.

I asked you to work on that question in class. Here are some answers:

(a) The loop condition must include index 0: `i >= 0`.

(b) This is a bit of a trick question. To avoid the loop condition, you must not enter the loop body. The only way to do that is to pass in `x = null`. You should also state, for instance, `y = 3`. The expected output is a `NullPointerException`, which is also the actual output.

(c) Inputs where `y` appears in the second or later position execute the fault but do not result in the error state; nor do inputs where `x` is an empty array. (There may be other inputs as well.) So, a concrete input: `x = {2, 3, 5}; y = 3`. Expected output = actual output = 1.

(d) One error input that does not lead to a failure is where `y` does not occur in `x`. That results in an incorrect PC after the final executed iteration of the loop.

(e) After running line 6 with `i = 1`, the decrement occurs, followed by the evaluation of `i > 0`, causing the PC to exit the loop (statement 8) instead of returning to statement 4. The faulty state is `x = {2, 3, 5}; y = 3; i = 0; PC = 8`, while correct state would be `PC = 4`.

Someone asked about distinguishing errors from failures. These questions were about failures at the method level, and so a wrong return value would be a failure when we're asking about methods. In the context of a bigger program, that return value might not be visible to the user, and so it might not constitute a failure, just an error.

Line Intersections

We then talked about different ways of validating a test suite. Consider the following Python code, found by Michael Thiessen on stackoverflow (`http://stackoverflow.com/questions/306316/determine-if-two-rectangles-overlap-each-other`).

```
1 class LineSegment:
2     def __init__(self, x1, x2):
3         self.x1 = x1; self.x2 = x2;
4
5 def intersect(a, b):
6     return (a.x1 < b.x2) & (a.x2 > b.x1);
```

We could construct test suites that:

- execute every statement in `intersect` (statement coverage). Well, that's not very useful; any old test case will do that. There are no branches, so what we'll call edge coverage doesn't help either.
- feed random inputs to `intersect`; unfortunately, interesting behaviours are not random, so it won't help much in general.
- check all outputs of `intersect` (i.e. a test case with lines that intersect and one with lines that don't intersect): we're getting somewhere—that will certify that the method works in some cases, but it's easy to think of situations that we missed.
- check different values of clauses `a.x1 < b.x2` and `a.x2 > b.x1` (logic coverage)—better than the above coverage criteria, but still misses interesting behaviours;
- analyze possible inputs and cover all interesting combinations of inputs (input space coverage)—can create an exhaustive test suite, if your analysis is sound.

Let's try to prove correctness of `intersect`. There's an old saying about testing—supposedly, it can only find the presence of bugs, not their absence. This is not completely true, especially if you have reliable software that automatically constructs an exhaustive test suite. But that is beyond the state of the practice in 2015, for the most part.

Inputs to `intersect`. There are essentially four inputs to this function. Rename them $aAbB$, for `a.x1`, `a.x2`, etc.

- Let's first assume that all points are distinct. We should make a note to ourselves to check violations of this, as well: we may have $a = A, a = b, a = B$, and symmetrically for B: $b = a, b = A, b = B$.
- For the purpose of the analysis, let's assume that $a < b$; when constructing test cases, we can swap a and b around. That's why there are duplicate assert statements below.
- Without loss of generality, we can assume that $a < A$ and $b < B$. (We ought to update the constructor if we want to make that assumption.)

With these assumptions, we have to test the three possible permutations $aAbB$, $abAB$, and $abBA$. It is simple to construct test cases for these permutations, using Python's unittest framework:

```
1       def test_aAbB(self):
2         a = LineSegment(0,2)
3         b = LineSegment(3,7)
4         self.assertFalse(intersect(a,b))
5         self.assertFalse(intersect(b,a))
6
7     def test_abAB(self):
8         a = LineSegment(0,4)
9         b = LineSegment(3,7)
10        self.assertTrue(intersect(a,b))
11        self.assertTrue(intersect(b,a))
12
13    def test_abBA(self):
14        a = LineSegment(0,4)
15        b = LineSegment(1,2)
16        self.assertTrue(intersect(a,b))
17        self.assertTrue(intersect(b,a))
```

Those test cases pass. However, if you construct test cases for equality (as I've committed to the repository), you see that the given `intersect` function fails on line segments that intersect only at a point. Replacing `<` with `<=` and `>` with `>=` fixes the code.