

Software Testing, Quality Assurance & Maintenance—Lecture 3

Patrick Lam
University of Waterloo

January 9/11, 2019

Plan

More examples on faults, errors and failures:

- numZero example (again);
- assignment 1-like exercise for `findLast`;
- testing line intersection algorithm

```
public static numZero(int[] x) {  
    int count = 0;  
    for (int i = 1; i < x.length; i++) {  
        if (x[i] == 0) count++;  
    }  
    return count;  
}
```

```
static public int findLast(int[] x, int y) {  
    for (int i = x.length - 1; i > 0 ; i--) {  
        if (x[i] == y) {  
            return i;  
        }  
    }  
    return -1;  
}
```

@Test

```
public void testFindLast() {  
    int[] x = new int[] {2, 3, 5};  
    assertEquals(0, FindLast.findLast(x, 2));  
}
```

Exercise: Faults

Read the faulty program `findLast`, which includes a test case exhibiting a failure.

b) trick question, $x = \text{null}$, $y = 3$

Answer the following questions:

- (a) Identify the fault, and fix it. $i > 0$, should be $i \geq 0$
- (b) If possible, identify a test case that does not execute the fault. $x = [1, 2]$, $y = 2$ \leftarrow wrong because still executes $i > 0$
- (c) If possible, identify a test case that executes the fault, but does not result in an error state. $x = [1, 2]$, $y = 2$. executes $i > 0$
- (d) If possible, identify a test case that results in an error, but not a failure. (Hint: PC) $x = [0]$, $y = 1$ or $x = [0, 2]$, $y = 1$
- (e) For the given test case, identify the first error state. Be sure to describe the complete state.

$x = [0, 2]$, $y = 1$, $i = 0$, PC = $i > 0$

PC never reaches inside the loop when $i = 0$

```

class LineSegment:
    def __init__(self, x1, x2):
        self.x1 = x1; self.x2 = x2;

    def intersect(a, b):
        return (a.x1 < b.x2) & (a.x2 > b.x1);

def test_aAbB(self):
    a = LineSegment(0, 2)
    b = LineSegment(3, 7)
    self.assertFalse(intersect(a, b))
    self.assertFalse(intersect(b, a))
    -> satisfies statement coverage (trivially) and branch coverage

```

Other ways of generating tests:

- be sure to cover all outputs (or at least all classes of outputs)
- generate cases randomly
 - need to know expected answer
 - hard to hit known interesting points in the problem space
- cover all values of logical sub-expressions
- cover all interesting combinations of input classes (input space coverage)

Sketch of proof of correctness of intersect

Rename inputs:

a	A	b	B
a.x1	a.x2	b.x1	b.x2

assume all points distinct (then check the assumption!)

write cases

a = A	a = b	a = B
b = a	b = A	b = B

assume $a < b$ (and then swap them)

assume $a < A$ and $b < B$ (constructor should reject violating lines)

Have reduced input space to permutations:

aAbB

abAB

abBA

aAbB: ...

abAB: ...

abBA: ...

test_equality:

(0,2) and (2,4) \Rightarrow true

(2, 2) and (3,4) \Rightarrow true

(0,2) and (0,4) \Rightarrow true

but then have to add \geq and \leq in LineSegment intersect definition