SWE 637 Software Testing
Activities, week 2

Dr. Brittany Johnson-Matthews
(Dr. B for short)

https://go.gmu.edu/SWE637

Adapted from slides by Jeff Offutt and Bob Kurtz

Class Activity #2

Consider exercises 5 and 7 in Chapter 1 (p. 13-17)

a) what is the fault?

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.

d) if possible, identify a test case that results in an error, but not a failure, and identify any initial error state.

e) if possible, identify a test case that causes a failure

Class Activity #2 (ex. 5, part 1)

```
/**
 * Find last index of element
 *
 * @param x array to search
 * @param y value to look for
 * @return last index of y in x; -1 if absent
 * @throws NullPointerException if x is null
 */
public static int findLast (int[] x, int y) {
    for (int i=x.length-1; i > 0; i--) {
        if (x[i] == y) {
            return i;
        }
    }
    return -1;
}
// test: x = [2, 3, 5]; y = 2; Expected = 0
```

The fault:

The loop terminates early at `i=1` (code should be `i>=0`)

A test case that does not execute the fault:

A `null` value for `x` will terminate with an exception before reaching the fault

A test case that does not result in an error:

`x`=[0,1,2], `y`=2 (or any case where *y* appears after the first element of *x* or if `x` is empty) executes the fault but does not cause an error

An error that does not result in a failure:

If `y` is not in `x`, then the final value of `i` is an error but not a failure

A failure:

`y` is the first element in `x`

Class Activity #2 (ex. 5, part 2)

```
/**
 * Find last index of zero
 *
 * @param x array to search
 *
 * @return last index of 0 in x; -1 if absent
 * @throws NullPointerException if x is null
 */
public static int lastZero (int[] x) {
    for (int i=0; i < x.length; i++) {
        if (x[i] == 0) {
            return i;
        }
    }
    return -1;
}
// test: x = [0, 1, 0]; Expected = 2
```

a) what is the fault?

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.

d) if possible, identify a test case that results in an error, but not a failure, and identify any initial error state.

e) if possible, identify a test case that causes a failure

Class Activity #2 (ex. 5, part 2)

```
/**
 * Find last index of zero
 *
 * @param x array to search
 *
 * @return last index of 0 in x; -1 if absent
 * @throws NullPointerException if x is null
 */
public static int lastZero (int[] x) {
    for (int i=0; i < x.length; i++) {
        if (x[i] == 0) {
            return i;
        }
    }
    return -1;
}
// test: x = [0, 1, 0]; Expected = 2
```

The fault:

The loop returns the first index of zero, it should count down

A test case that does not execute the fault:

None- all inputs execute the initialization and evaluation parts of the loop

A test case that does not result in an error:

`x` is `null` or has a length of 0 (arguably, if `x` has length of 1, final value of `i` will be incorrect)

An error that does not result in a failure:

Any value for `x` in which zero appears no more than once will have an error state but no failure

A failure:

Zero appears twice in `x`

Class Activity #2 (ex. 5, part 3)

```
/**
 * Count positive elements
 * Note: zero is not considered positive
 *
 * @param x array to search
 * @return count of positive elements in x
 * @throws NullPointerException if x is null
 */
public static int countPositive (int[] x) {
    int count = 0;
    for (int i=0; i < x.length; i++) {
        if (x[i] >= 0) {
            count++;
        }
    }
    return count;
}

// test: x = [-4, 2, 0, 2]; Expected = 2
```

a) what is the fault?

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.

d) if possible, identify a test case that results in an error, but not a failure, and identify any initial error state.

e) if possible, identify a test case that causes a failure

Class Activity #2 (ex. 5, part 3)

```
/**
 * Count positive elements
 * Note: zero is not considered positive
 *
 * @param x array to search
 * @return count of positive elements in x
 * @throws NullPointerException if x is null
 */
public static int countPositive (int[] x) {
    int count = 0;
    for (int i=0; i < x.length; i++) {
        if (x[i] >= 0) {
            count++;
        }
    }
    return count;
}

// test: x = [-4, 2, 0, 2]; Expected = 2
```

The fault:

Algorithm counts zeros as positive

A test case that does not execute the fault:

If `x` is `null` or has a length of zero, the fault is not executed

A test case that does not result in an error:

Any values for `x` which does not contain zero will execute the fault but not have an error

An error that does not result in a failure:

None – every input that causes an error results in a failure

A failure:

Any value of `x` that includes zero

Class Activity #2 (ex. 5, part 4)

```
/**
 * Count odd or positive elements
 *
 * @param x array to search
 * @return count of odd/pos elements in x
 * @throws NullPointerException if x is null
 */
public static int oddOrPos (int[] x) {
    int count = 0;
    for (int i=0; i < x.length; i++) {
        if (x[i]%2 == 1 || x[i] > 0) {
            count++;
        }
    }
    return count;
}

// test: x = [-3, -2, 0, 1, 4]; Expected = 3
```

a) what is the fault?

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.

d) if possible, identify a test case that results in an error, but not a failure, and identify any initial error state.

e) if possible, identify a test case that causes a failure

Class Activity #2 (ex. 5, part 4)

```
/**
 * Count odd or positive elements
 *
 * @param x array to search
 * @return count of odd/pos elements in x
 * @throws NullPointerException if x is null
 */
public static int oddOrPos (int[] x) {
    int count = 0;
    for (int i=0; i < x.length; i++) {
        if (x[i]%2 == 1 || x[i] > 0) {
            count++;
        }
    }
    return count;
}

// test: x = [-3, -2, 0, 1, 4]; Expected = 3
```

The fault:

Algorithm does not count negative odd numbers (Java's mod operator takes the sign of the quotient, so –3%2=-1)

A test case that does not execute the fault:

If `x` is `null` or has a length of zero, the fault is not executed

A test case that does not result in an error:

Any values for `x` which contain only non-negative and/or even negative numbers will execute the fault but not the cause of the error

An error that does not result in a failure:

None – every input causes an error results in a failure

A failure:

Any value of `x` that includes a negative odd number

Class Activity #2 (ex. 7, part 2)

```
public class BigDecimalTest {
  BigDecimal x = new BigDecimal ("1.0");
  BigDecimal y = new BigDecimal ("1.00");
  // Fact: !x.equals(y), but x.compareTo(y)==0

  Set <BigDecimal> tree = new TreeSet<BigDecimal>();
  Set <BigDecimal> hash;

  @Before public void setUp() {
     x = new BigDecimal("1.0");
     y = new BigDecimal("1.00");
     // Fact:  !x.equals(y), but x.compareTo(y) == 0

     tree = new TreeSet <BigDecimal> ();
     hash = new HashSet <BigDecimal> ();
  }

  // this test fails!
  @Test public void inconsistentSets() {
     tree.add(x); tree.add(y);
     // TreeSet uses compareTo(), so tree now has 1 element

     hash.add(x);
     hash.add(y);
     // HashSet uses equals(), so hash now has 2 elements

     assertEquals(tree, hash);
     // hence the above assertion cannot possibly be true
  }
}
```

Incorrectly named in textbook

a) what is the fault?

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.

d) if possible, identify a test case that results in an error, but not a failure, and identify any initial error state.

e) if possible, identify a test case that causes a failure

Exercise 7 answers + Discussion

The fault:

BigDecimal's `equals()` requires instances to be the same in scale and value, so that "1.0" ≠ "1.00"; `compareTo()` only requires instances to be the same in value, so that "1.0" == "1.00". If we assume `compareTo()` is correct and want to change `equals()`, that implies that `hashCode()` is also incorrect.

A test case that does not execute the fault:

Any code that does not call `equals()` or `hashCode()`, including methods of HashSet, will not reach the fault.

A test case that does not result in an error:

Tests of HashSet using only different values or the same values with the same scale reach the fault but do not result in an error.

An error that does not result in a failure:

None - tests of HashSet using the same values but different scales reach the fault, cause an error, and result in a failure.