EXERCISES
Foundations of Software Testing 2023-2024

Lecture 1: What is testing?

1. Consider the following faulty program with a test case that results in failure. Show this by

```
public int findLast (int[] x, int y)
{
//Effects: If x==null throw NullPointerException
//   else return the index of the last element
//   in x that equals y.
//   If no such element exists, return -1
   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
```

calculating the actual output.

(a) Identify 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 state.
(d) If possible identify a test case that results in an error, but not a failure. Hint: Don't forget about the program counter PC.
(e) For the given test case, identify the first error state. Be sure to describe the complete state.
(f) Fix the fault and verify that the given test now produces the expected output.

findLast() Solution
Actual output: -1

(a) The for-loop should include the 0 index:
for (int i=x.length-1; i >= 0; i--)
(b) A null value for x will result in a NullPointerException before the loop test is evaluated. Hence no execution of the fault.
Input: x = null; y = 3
Expected Output: NullPointerException
Actual Output: NullPointerException
(c) For any input where y appears in the second or later position, there is no error. Also, if x is empty, there is no error.
Input: x = [2, 3, 5]; y = 3;
Expected Output: 1
Actual Output: 1
(d) For an input where y is not in x, the missing path (i.e. an execution that should run the entire loop but does not take the final one) is an error, but there is no failure.
Input: x = [2, 3, 5]; y = 7;

Expected Output: -1
Actual Output: -1

(e) Note that the key aspect of the error state is that the PC is outside the loop (following the false evaluation of the 0>0 test. In a correct program, the PC should be at the if-test, with index i==0.

Input: x = [2, 3, 5]; y = 2;
Expected Output: 0
Actual Output: -1

First Error State: x = [2, 3, 5], y = 2; i = 0 (or undefined or 1, depending on the compiler);
PC = just before return -1;;

(f) See (a)

2. Consider the following faulty program with a test case that results in failure. Show this by calculating the actual output.

```
public static int lastZero (int[] x)
{
//Effects: if x==null throw NullPointerException
//   else return the index of the LAST 0 in x.
//   Return -1 if 0 does not occur in 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) Identify 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 state.
(d) If possible identify a test case that results in an error, but not a failure. Hint: Don't forget about the program counter PC.
(e) For the given test case, identify the first error state. Be sure to describe the complete state.
(f) Fix the fault and verify that the given test now produces the expected output.

lastZero() Solution

Actual output: 0

(a) The for-loop should search high to low:
for (int **i=x.length-1; i >= 0; i--**)

(b) All inputs execute the fault – even the null input.

(c) If the loop is not executed at all, there is no error. If the loop is executed only once, high-to-low and low-to-high evaluation are the same. Hence there is no error for length 0 or length 1 inputs.

Input: x = [3]
Expected Output: -1
Actual Output: -1

(d) There is an error anytime the loop is executed more than once, since the values of index i ascend instead of descend.

Input: x = [1, 0, 3]
Expected Output: 1

Actual Output: 1

(e) The first error state is when index i has the value 0 when it should have a value at the end of the array, namely x.length-1. Hence, the first error state is encountered immediately after the assignment to i in the for-statement if there is more than one value in x.

Input: x = [0, 1, 0]
Expected Output: 2
Actual Output: 0

First Error State: x = [0, 1, 0], i = 0, PC = just after i= 0;

(f) See (a)

3. Consider the following faulty program with a test case that results in failure. Show this by calculating the actual output.

```
public int countPositive (int[] x)
{
//Effects: If x==null throw NullPointerException
//   else return the number of
//        positive elements in 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) Identify 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 state.
(d) If possible identify a test case that results in an error, but not a failure. Hint: Don't forget about the program counter PC.
(e) For the given test case, identify the first error state. Be sure to describe the complete state.
(f) Fix the fault and verify that the given test now produces the expected output.

countPositive() Solution

Actual output: 3

(a) The test in the conditional should be:

if (x[i] > 0)

(b) x must be either null or empty. All other inputs result in the fault being executed. We give the empty case here.

Input: x = []
Expected Output: 0
Actual Output: 0

(c) Any nonempty x without a 0 entry works fine.

Input: x = [1, 2, 3]
Expected Output: 3
Actual Output: 3

(d) For this particular program, every input that results in error also results in failure. The reason is that error states are not repairable by subsequent processing. If there is a 0 in x, all subsequent states (after processing the 0) will be error states no matter what else is in x.

(e) Input: x = [-4, 2, 0, 2]

Expected Output: 2

Actual Output: 3

First Error State: x = [-4, 2, 0, 2], i = 2, count = 1;

PC = immediately before the count++ statement. (taking the branch to the count++ statement could be considered erroneous.)

(f) See (a)

Lecture 2: Input space partitioning

1. Answer the following questions for the method search() below:

```
public static int search (List list, Object element)
// Effects: if list or element is null throw NullPointerException
// else if element is in the list, return an index of element in the list;
// else return -1
// for example, search ([3,3,1], 3) = either 0 or 1 search ([1,7,5], 2) = -1
```

Base your answer on the following characteristic partitioning:

Characteristic: Location of element in list

Block 1: element is first entry in list

Block 2: element is last entry in list

Block 3: element is in some position other than first or last

a) The characteristic fails the disjointness property. Give an example that illustrates this.

b) The characteristic fails the completeness property. Give an example that illustrates this.

c) Supply one or more new partitions that capture the intent of the above characteristic but do not suffer from completeness or disjointness problems.

Solution:

a) Lots of examples can be found. For example list = [3, 4, 3]; element = 3.

b) The problem is that e may not be in the list: list = [5, 3]; e = 4.

c) The easiest approach is to separate the characteristics into two partitions:

- element is first in the list: true, false
- element is last in the list: true, false

1. Derive input space partitioning tests for the `GenericStack class` with the following method signatures:

```
public GenericStack ();
public void Push (Object X);
public Object Pop ();
public boolean IsEmt ();
```

Assume the usual semantics for the stack. Define 4 characteristics of inputs, partition each characteristic into blocks, and give values for the blocks

Solution

Note that there are 4 testable units here (the constructor and the three methods), but that there is substantial overlap between the characteristics relevant for each one. For the three methods, the implicit parameter is the state of the GenericStack. The only explicit input is the Object x parameter in Push().

The constructor has neither inputs nor implicit parameters.

Typical characteristics for the implicit state are

- Whether the stack is empty.
    - true (Value stack = [])
    - false (Values stack = [”cat”], [”cat”, ”hat”])
- The size of the stack.
    - 0 (Value stack = [])
    - 1 (Possible values stack = [”cat”], [null])
    - more than 1 (Possible values stack = [”cat”, ”hat”], [”cat”, null], [”cat”, ”hat”, ”ox”])
- Whether the stack contains null entries
    - true (Possible values stack = [null], [null, ”cat”, null])
    - false (Possible values stack = [”cat”, ”hat”], [”cat”, ”hat”, ”ox”])

A typical characteristic for Object x is

- Whether x is null.
    - true (Value x = null)
    - false (Possible values x = ”cat”, ”hat”, ””)

The following is a characteristic that involves the combination of Object x and the stack state:

- Does Object x appear in the stack?
    - true (Possible values: (null, [null, ”cat”, null]), (”cat”, [”cat”, ”hat”]))
    - false (Possible values: (null, [”cat”]), (”cat”, [”hat”, ”ox”]))

3. Consider the following values for three partitions of three input parameters, each partition with fours bloks

C1	2	1	0	-1
C2	2	1	0	-1
C3	2	1	0	-1

a) Enumerate all tests to satisfy the Pair-Wise (PWC) criterion.
b) Enumerate all tests to satisfy the Multiple Base Choice (MBCC) given the two base tests (2, 2, 2) and (1, 2, 2)

Solution:

a) Here a possible solution

(2, 2, 2) (2, 1, 1) (2, 0, 0) (2,−1,−1)
(1, 2, 1) (1, 1, 2) (1, 0,−1) (1,−1, 0)
(0, 2, 0) (0, 1,−1) (0, 0, 2) (0,−1, 1)
(−1, 2,−1) (−1, 1, 0) (−1, 0, 1) (−1,−1, 2)

b) There are 20 tests, with the base choices listed first:

(2, 2, 2)
(2, 1, 2) (2, 0, 2) (2,−1, 2)
(2, 2, 1) (2, 2, 0) (2, 2,−1)
(2, 2, 2) (0, 2, 2) (−1, 2, 2)

(1, 2, 2)
(1, 2, 2) (0, 2, 2) (−1, 2, 2)
(1, 2, 1) (1, 2, 0) (1, 2,−1)
(1, 1, 2) (1, 0, 2) (1,−1, 2),