The University of Melbourne
SWEN90006: Software & Security Testing

Tutorial 1 Solutions

NOTE You are expected to prepare for this tutorial by sketching answers to the tasks and questions before attending the tutorial.

What is Testing Really?

> "*It works! Trust me, I've tested the code thoroughly!*"

. . . and with those bold words many software projects have been made bankrupt.

The aim of this tutorial is to start to get an understanding of our major themes; that is, of the factors that make up quality and how testing effects them. While we have not yet looked at testing formally yet you will be asked in this tutorial to test a small program fragment with the purpose of starting to see where the difficulties of testing lay in practice.

As a first step, let's discuss what testing is, and why we would want to perform testing on our software.

Discussion

(i) Recall that we do not ever show that the program does what is intended, or that it meets its specification, when testing. What we do, in essence, is to try to find the cases where it does not meet its specification.

Put another way, we can never guarantee that a function is *correct*, we just seek assurance that it is not incorrect.

(ii) The first point implies that the better we are at covering the input domains (but this is not easy) and the better we are at finding errors, the closer we get to being confident that the program is correct, at least with respect to the specifications that we are testing against. But explicitly trying to find cases that do meet their specification does not improve the quality of the software. It is only when we find a can we fix it, and thus improve quality.

(iii) Even small programs are typically complex enough that even with existing testing techniques we can never be sure that we have found every last problem with the program. Testing needs to be complemented with other processes, and other assurance methods in order to be more confident that the program is correct and exhibits *quality*.

Important Terminology

Before beginning the exercises, consider the following four definitions, the first three of which are defined in Chapter 1 of the lecture notes:

Fault: An incorrect step, process, or data definition in a computer program. Faults are the source of failures – a fault in the program triggers a failure under the right circumstances.

Failure: A deviation between the observed behaviour of a program, or a system, from its specification.

Error: An incorrect *internal* state that is the result of some fault. An error may not result in a failure – it is possible that an internal state is incorrect but that it does not affect the output.

Failures and errors are the result of faults – a fault in a program can trigger a failure and/or an error under the right circumstances. In normal language, software faults are usually referred to as "bugs", but the term "bug" is ambiguous and can mean to faults, failures, or errors; as such, as will avoid this term.

The Programs

The following are taken from the exercises in Chapter 1 of *Introduction to Software Testing* by Offutt and Ammann. For each program, the test case below the program results in a failure that is caused by a fault in the program.

```
/**
 * 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
 */
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
//Input: x = [2, 3, 5]; y = 2
//Expected output = 0
```

```
/**
 * If x == null throw NullPointerException.
 * Else return the index of the LAST 0 in x.
 * Return -1 if 0 does not occur in x.
 *
 */
public static int lastZero (int[] x)
{
   for (int i = 0; i < x.length; i++)
   {
      if (x[i] == 0)
      {
         return i;
      }
   }
   return -1;
}

//Test
//Input: x = [0, 1, 0]
//Expected output = 2
```

```
/**
 * If x == null throw NullPointerException.
 * Else return the number of positive
 * elements in x.
 */
public int countPositive (int[] x)
{
   int count = 0;
   for (int i = 0; i < x.length; i++)
   {
      if (x[i] >= 0)
      {
         count++;
      }
   }
   return count;
}

//Test
//Input: x = [-4, 2, 0, 2]
//Expected output = 2
```

```
/**
 * If x == null throw NullPointerException.
 * Else return the number of elements in x
 * that are either odd or positive (or both)
 */
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
//Input: x = [-3, -2, 0, 1, 4]
//Expected output = 3
```

Your tasks

For each of the programs, perform the following tasks:

1. Specify the input domain and valid input domain of the function.
2. Identify the fault.
3. If possible, identify a test case that does not execute the faulty statement.
4. If possible, identify a test case that executes the faulty statement, but does not result in an failure.
5. If possible identify a test case that forces the program into an error (that is, the first step that a program deviations from its intended behaviour), but does not produce a failure.
6. Fix the fault and verify that the given test now produces the expected output.

Solutions

`findLast` program

1. The input domain is $int[] \times int$. That is, a set of pairs, in which the first element is an array of integers and the second element is an integer.

   Assuming that an invalid input is one that throws an exception, the valid input domain is

   $\{(x, y) : int[] \times int \mid x \neq null\}$.

2. `i > 0` should be `i >= 0`.
3. `x == null`.
4. Any case with a last element, where the element is not in position 0.

5. Any case in which the element is not in the list. When the loop executes, the state is in error, but will not be revealed.
6. When `i == 0`. The loop should execute, but does not.

`lastZero` program

1. The input domain is $int[]$.
   The valid input domain is $\{x : int[] \mid x \neq null\}$
2. Loop counter should go from last to first, not first to last.
3. All inputs execute the fault.
4. An empty array, or any array with 1 element.
5. Any list with 0 or 1 occurrences of the element.
6. When `i == 0`, but should point to the final element in the array.

`countPos` program

1. The input domain is $int[]$.
   The valid input domain is $\{x : int[] \mid x \neq null\}$
2. `x[i] >= 0` should be `x[i] > 0`.
3. `x` is null or empty.
4. Any list with no occurrence of 0.
5. None?
6. When `x[i] == 0`.

`oddOrPos` program

1. The input domain is $int[]$.
   The valid input domain is $\{x : int[] \mid x \neq null\}$
2. The program does not count negative odd negative numbers because `x[i]%2` would be `-1` for any negative number. Using `x[i]%2 == -1` would work. We do not need to check `x[i]%2 == 1` because any positive odd number is counted by `x[i]>0`.
3. `x` is null or empty.
4. Any list that does not contain negative numbers.
5. None?
6. After the first increment of `count` when `i == 0`.

Important discussion points

The point of this exercise is three-fold:

1. To illustrate the difference between testing and debugging. In SWEN90006, we cover testing, but not debugging.
2. To clarify the terminology of faults, failures, and errors. Faults and failures are quite often confused, and importantly, the term "bugs" is used to mean both. In SWEN90006, the term "bugs" is disallowed due to its ambiguity.
3. To illustrate an important point about coverage-based testing (which we will cover later in the subject): just executing every statement/branch etc. is a *necessary but not sufficient* measure of a good test suite. It is not only possible to execute a statement but not trigger an error, but to trigger an error that does NOT result in a failure.

   These cases are known as *coincidental correctness*, and are exceedingly common in real-world software.