

CSSE2003

Software Engineering Studio

Lecture 5 Testing

Supplementary Materials Sample Solutions

Semester 2, 2009

School of Information Technology and Electrical Engineering
The University of Queensland

What follows are (of necessity, given the nature of the exercises) samples only.

Exercise 1

One simple strategy for an initial test suite design that nicely applies to this case is to partition the test cases according to the values of the output. A decision to use this process involves a commitment to four sets test cases. The precondition requires that all three sides are greater than zero, and hence we only use test cases that satisfy this requirement.

Test cases for output EQUILATERAL (i.e., all three sides equal). We use two test cases: one on the boundary (least possible lengths of sides) and the other not.

Input: (1, 1, 1) Input: (42, 42, 42)

Test cases for output ISOSCELES. There are multiple ways in which 2 sides can be equal: a and b, b and c, or c and a. It is worth trying each of these, first on the boundary (with the least possible length sides) and then not.

Input: (1, 2, 2) Input: (2, 1, 2) Input: (2, 2, 1)
Input: (42, 42, 20) Input: (20, 42, 42) Input: (42, 20, 42)

Test cases for output SCALENE. We want to differentiate scalene triangles from invalid triangles, as well as than from isosceles or equilateral. To be a valid triangle the sum of any two sides must be greater than the third side. We choose our test cases so that the largest side is only just less than the sum of the other two (24 just less than 10+15). As there are three argument positions for the largest side it makes sense to have three test cases where the largest side is in the third, second and first positions. We use one set on the boundary (the least possible lengths of sides for a scalene triangle are 2, 3 and 4) and one not.

Input (2, 3, 4) Input: (2, 4, 3) Input: (4, 2, 3)
Input: (10, 15, 24) Input: (10, 24, 15) Input: (24, 10, 15)

Test cases for output INVALID. This complements the cases for scalene in that there are three choices for the side that is not less than the sum of the other two sides:

Input: (10, 15, 25) Input: (10, 25, 15) Input: (25, 10, 15)

The value 25 is chosen to be just not less than (i.e., equal to) 10+15. We can also try a value strictly greater than the sum of the other two sides:

Input: (10, 15, 26) Input: (10, 26, 15) Input: (26, 10, 15)

For each test case for invalid triangles and the last three test cases for scalene triangles, another test case can be generated by swapping the positions of parameters 10 and 15.

Exercise 2

The code under test is

```
/*@ requires 0 < a && 0 < b && 0 < c */
int tri(int a, int b, int c) {
    int r = INVALID;
    if (a < b+c && b < a+c && c < a+b) {
        if (a == b && a == c) {
            r = EQUILATERAL;
        } else if (a == b || a == c || b == c) {
            r = ISOSCELES;
        } else {
            r = SCALENE;
        }
    }
    return r;
}
```

When generating test cases the expected output value corresponding to the inputs must be determined by the specification and not by looking at what the code produces.

- (a) When generating glass box test cases one approach is to consider paths through the code. For this program there is one path corresponding to the condition of the outer “if” being false (output INVALID) and three paths if it is true, corresponding to the three branches of the inner “if” statement (giving outputs EQUILATERAL, ISOSCELES and SCALENE, respectively). In this case the four paths correspond to the four outputs (as considered in Exercise 1). The condition in the outer “if” contains three conjuncts, and hence we can introduce separate test cases focusing on the three separate conjuncts. Each test case will make one of the conjuncts false. In general one might also consider test cases that make multiple conjuncts false. However, for this example at most one can be false, for example, if $a \geq b+c$ then this implies $a-c \geq b$, but $a+c > a-c$ because c is positive and hence $a+c > b$. Similarly, $a+b > c$. When considering a condition like $a < b+c$ being false, it is worth considering both the cases $a = b+c$ (on the boundary) and $a > b+c$. Combining these give the same last six test cases from Exercise 1. That covers the case when the condition in the outer “if” is false.

To differentiate the paths in the inner “if” we need to consider a set of test cases, all of which satisfy the condition in the outer “if”. We need a test case for the first branch with all three sides equal (as for the EQUILATERAL case in Exercise 1). The first condition in the inner “if” has two conjuncts. Hence we can consider test cases that make each of the conjuncts false, and others that make both false. The choice of test cases is determined by considering the second condition in the inner “if”. First we note that if any two of the disjuncts in the ISOSCELES test are true, then all three sides are equal (i.e., EQUILATERAL) and hence we only need to consider test cases where just one disjunct is true (giving the test cases for ISOSCELES in Exercise 1) and test cases where none of the disjuncts are true, i.e., all three sides are different. The choices for the SCALENE path can consider the boundary cases for the conjuncts in the outer “if” as discussed above.

Overall we have ended up with the same set of test cases as for Exercise 1, but we got there via a different path.

- (b) Here we add test cases where the length of one or more of the sides is zero or negative. The output should be INVALID. We can consider test cases with non-positive lengths that also satisfy the condition in the outer “if”. If c is non-positive, then from the first two conjuncts in the outer “if” we have $a < b$ and $b < a$, which can’t both hold together. Hence the condition in the outer “if” should be false if any of the sides is non-positive. The additional tests we add are tests with each of a , b and c zero and each of a , b and c negative (and possibly combinations of these).
- (c) For comparison of the two glass-box test suites: in the case where the *tri()* method precondition is weakened, we are asking a different question, viz. ‘what kind of output should *tri()* generate with input (a,b,c) satisfying the condition to be an equilateral triangle’, vs. ‘what kind of output should *tri()* generate with input (a,b,c) satisfying the condition to be an equilateral triangle and the

precondition'. The former question imposes a weaker condition on input (a,b,c) and so admits a larger set of concrete test cases (i.e., substitutions of definite constants for arbitrary constants), and so places a larger burden of correctness on method *tri()* (since the method must now operate correctly in more cases) and so places a larger burden on the tester of method *tri()* (since they must now sample a larger space of possible inputs). For example, input (0,0,0) is not a possible input for *tri()* in (a), but it is a possible input for *tri()* in (b).

Exercise 3

There are two parameters that vary here: the value to be searched for, x , and the non-descending array, a . I'll note straight away that we should consider the case in which the array is empty, and perhaps the case where it is length one for separate testing. Two cases we should distinguish are whether x is to be found in the array or not.

(a) Let us start with the case where x is in the array. The result is the position of the last occurrence of x in a . We consider different lengths of arrays.

(i) One simple special case is if x is the only element in the array:

Input: (3, {3}) Output: 0

(ii) For a longer array, cases we should consider are if x is the first element of the array, the last element of the array, somewhere in the middle of a .

Input: (2, {2, 5, 9}) Output: 0

Input: (5, {2, 5, 9}) Output: 1

Input: (9, {2, 5, 9}) Output: 2

(iii) In addition, we can consider similar cases if there are multiple occurrences of x in the array to make sure the position returned is that of the last occurrence:

Input: (2, {2, 2, 5, 9}) Output: 1

Input: (5, {2, 5, 5, 9}) Output: 2

Input: (9, {2, 5, 9, 9}) Output: 3

We could also consider tests with more than two occurrences of x .

(b) For the case when x is not in the array, we also have to test that the position returned is appropriate. We consider different lengths of arrays.

(i) A special case is an empty array – there is no way x can be in that. If we insert x into an empty array it will end up at position 0. Hence the output in this case should be -1.

Input: (3, {}) Output: -1

For this test case the output value wasn't very well defined by the description in the exercise; this is a common problem with natural language specifications.

(ii) For an array of length one, one can test searching for a value less than that in the array and a value greater than it:

Input: (1, {3}) Output: -1

Input: (4, {3}) Output: 0

(iii) There are special cases when x is less than (greater than) all the elements in the array. If x precedes all the elements in the array, the output should be the position after which it should be inserted; as it will need to end up at position 0, the output should be -1.

Input: (1, {2, 5, 9}) Output: -1

Input: (6, {2, 5, 9}) Output: 1

Input: (10, {2, 5, 9}) Output: 2

Exercise 4

For this exercise we can consider the structure of the program. Because there is a loop there isn't a fixed set of paths that can be followed (as with the "if" statement from Exercise 2). In this case the loop guard consists of a conjunction of two conditions. The guard can be false if either of these conditions is false, and hence we need to consider both these ways of the loop terminating. In addition, the second conjunct $x < ar[i]$ can be false if either $x == ar[i]$ or $x > ar[i]$.

However, if the first conjunct is false, the second conjunct is not evaluated and hence we don't need to consider the second conjunct in this case (which is fortunate because the second conjunct is not defined due to an array index out of range). An interesting case to consider is if the loop is never entered because the guard is initially false (either of the conjuncts could be false and the second conjunct can be false for the two reasons outlined above). We'll start with these cases.

Input: (3, {}) Output: -1

Input: (9, {2, 5, 9}) Output: 2

Input: (10, {2, 5, 9}) Output: 2

These three cases also occur in the previous exercise. We can include cases where the loop terminates after just one iteration, again considering the three reasons for termination:

Input: (1, {3}) Output: -1

Input: (5, {2, 5, 9}) Output: 1

Input: (6, {2, 5, 9}) Output: 1

These cases also occur in the previous exercise. The above cover the cases of zero or one iteration. For the array containing three elements we can have up to three iterations. Hence we can consider the following cases, for two iterations we consider two causes of termination, and for the case of three iterations there is only one.

Input: (2, {2, 5, 9}) Output: 0

Input: (3, {2, 5, 9}) Output: 0

Input: (1, {2, 5, 9}) Output: -1

Of these the second test case wasn't one from the previous exercise. In looking at the code the case of duplicate elements in the array didn't immediately spring to mind, but test cases for that and perhaps for other length arrays are further possibilities.

Exercise 5

The code for this exercise is quite different to the code for the previous exercise, although they have the same specification. In the previous exercise the length of the array wasn't a significant factor, except for the special case when the array is empty or perhaps of length one. Because the code for this version uses binary search, the length of the array is more significant. There are issues such as whether the array is of odd or even length. Similarly, whereas for the previous question any position other than possibly the first or last is treated the same, for the binary search this is not the case (consider an error like having an assignment " $lo = mid+1$ " rather than " $lo = mid$ "). Hence for this code we should consider arrays of different lengths (that are long enough to require a number of iterations) and exhaustive searching for values at each possible position within the array including between all positions in the array as well as before the first and after the last.

Input: (1, {2, 4, 6, 8, 10, 12, 14, 16}) Output: -1

Input: (2, {2, 4, 6, 8, 10, 12, 14, 16}) Output: 0

Input: (3, {2, 4, 6, 8, 10, 12, 14, 16}) Output: 0

Input: (4, {2, 4, 6, 8, 10, 12, 14, 16}) Output: 1

Input: (5, {2, 4, 6, 8, 10, 12, 14, 16}) Output: 1

Input: (6, {2, 4, 6, 8, 10, 12, 14, 16}) Output: 2

Input: (7, {2, 4, 6, 8, 10, 12, 14, 16})	Output: 2
Input: (8, {2, 4, 6, 8, 10, 12, 14, 16})	Output: 3
Input: (9, {2, 4, 6, 8, 10, 12, 14, 16})	Output: 3
Input: (10, {2, 4, 6, 8, 10, 12, 14, 16})	Output: 4
Input: (11, {2, 4, 6, 8, 10, 12, 14, 16})	Output: 4
Input: (12, {2, 4, 6, 8, 10, 12, 14, 16})	Output: 5
Input: (13, {2, 4, 6, 8, 10, 12, 14, 16})	Output: 5
Input: (14, {2, 4, 6, 8, 10, 12, 14, 16})	Output: 6
Input: (15, {2, 4, 6, 8, 10, 12, 14, 16})	Output: 6
Input: (16, {2, 4, 6, 8, 10, 12, 14, 16})	Output: 7
Input: (17, {2, 4, 6, 8, 10, 12, 14, 16})	Output: 7

Similar sets of test cases for a different length array would be useful (not shown here) and the (possibly special) cases of an array of length zero and of length one:

Input: (3, {})	Output: -1
Input: (1, {2})	Output: -1
Input: (2, {2})	Output: 0
Input: (3, {2})	Output: 0

In addition, we can devise test cases with arrays that contain multiple occurrences of the value being searched for. We need to construct arrays such that the value being searched for is duplicated. To be thorough we need different arrays (of length 8, say) with the duplicated value in different places so that the test cases return each possible position (from 1 to 7). These test cases are not shown here.

In testing this code we could add an assertion to explicitly check that the loop invariant holds. First we define a function to evaluate the invariant.

```
static boolean search2Inv( int x, int[] ar, int lo, int hi) {
    return (-1 <= lo && lo < hi && hi <= ar.length) &&
        (lo == -1 || ar[lo] <= x) &&
        (hi == ar.length || x < ar[hi])
}
```

Assertions would be inserted just before the loop (the invariant should hold initially) and at the end of the body of the loop (the invariant should be re-established by the body of the loop).

```
if ( debug ) { assert( search2Inv( x, ar, lo, hi ) ); }
```

Exercise 6

A better design might be to raise an exception if there is insufficient room on the flight. Note that there is an error in the program below in that `flight.currentPassengers / flight.maxPassengers` uses integer division and hence will typically evaluate to zero. Hopefully our testing would find that error.

```
/*@ requires flight != null && 0 < numPassengers
 * ensures if numPassengers can fit on the flight then \result is the cost
 * of all their flights. If the flight is more than 80% full they must pay
 * the full cost, otherwise there is a 10% discount. If numPassengers
 * won't fit on the flight then \result is -1. */
double flightCost( Flight flight, int numPassengers ) {
```

```

if ( flight.currentPassengers + numPassengers > flight.maxPassengers ) {
    return -1;
}
if ( flight.currentPassengers / flight.maxPassengers > 0.8 ) {
    return flight.fullCost * numPassengers;
} else {
    return 0.9 * flight.fullCost * numPassengers;
}
}
}

```

To test `flightCost` we need to set up a number of flights: one more than 80% full and the other less than or equal to. Then we can test for different numbers of passengers including cases where the passengers will just fit on the flight and where they (just) won't fit on the flight (flights f1 and f2 below). We need to add a constructor to `flight` to create flights. We'll use the following flights throughout the tests.

	from	to	date	maxPassengers	currentPassengers	fullCost
f1	"Brisbane"	"Sydney"	14/11/2005	20	16	10
f2	"Brisbane"	"Melbourne"	14/11/2005	20	17	10
f3	"Brisbane"	"Cairns"	14/11/2005	20	16	10
f4	"Brisbane"	"Adelaide"	14/11/2005	20	17	10
f5	"Brisbane"	"Adelaide"	14/11/2005	20	16	10

Then our tests are

```

Input: ( f1, 3 )      Output: 27
Input: ( f1, 4 )      Output: 36
Input: ( f1, 5 )      Output: -1
Input: ( f2, 2 )      Output: 20
Input: ( f2, 3 )      Output: 30
Input: ( f2, 4 )      Output: -1

```

The method `flightCost` includes a comparison of values of type `double`. Because such values may not be accurately represented in binary floating point representation, it is important to test values on the boundary where

```
(double) flight.currentPassengers / (double) flight.maxPassengers == 0.8
```

to pick up errors caused by this inaccuracy. The program should really allow for a small margin of error in the comparison.

To test `find` we need to set up a `FlightDataBase` with a number of flights. As we don't have an implementation of `FlightDataBase`, we can write a simple one for testing that uses a Java `List`. We use a `List` rather than, say, a `HashMap`, so that we can control the order of the elements in the structure and hence the order of the iteration through the collection. This ensures that we can test that the loop within `find` that uses the iterator doesn't miss, say, the first or the last elements in the iteration.

```

class FlightDatabase implements Iterable {
    private List<Flight> flights = new ArrayList<Flight>( 5 );
    /* appends a flight to the vector of flights */
    public void add(Flight f) {
        flights.add(f);
    }
}

```

```

    /* returns an iterator to go through the flights in vector order */
    public java.util.Iterator iterator() {
        return flights.iterator();
    }
}

```

We test searching for flights that exist both with a number of passengers that will fit and a number that won't. In the case of the Brisbane-Adelaide flights f4 will fit 3 passengers, but f5 can fit 4 passengers.

Input: (4, "Brisbane", "Sydney", 14/11/2005)	Output: f1
Input: (5, "Brisbane", "Sydney", 14/11/2005)	Output: null
Input: (3, "Brisbane", "Melbourne", 14/11/2005)	Output: f2
Input: (4, "Brisbane", "Melbourne", 14/11/2005)	Output: null
Input: (3, "Brisbane", "Adelaide", 14/11/2005)	Output: f4
Input: (4, "Brisbane", "Adelaide", 14/11/2005)	Output: f5
Input: (5, "Brisbane", "Adelaide", 14/11/2005)	Output: null

To test that flights aren't incorrectly matched, we add tests that match all but one of the conditions and only require a place for 1 passenger.

Input: (1, "Sydney", "Melbourne", 14/11/2005)	Output: null
Input: (1, "Brisbane", "Perth", 14/11/2005)	Output: null
Input: (1, "Brisbane", "Sydney", 15/11/2005)	Output: null
Input: (1, "Brisbane", "Perth", 15/11/2005)	Output: null
Input: (1, "Sydney", "Perth", 14/11/2005)	Output: null
Input: (1, "Sydney", "Adelaide", 15/11/2005)	Output: null