

CSSE2003

Software Engineering Studio

Lecture 5 Testing

Supplementary Materials

Semester 2, 2009

School of Information Technology and Electrical Engineering
The University of Queensland

Goal:

1. Practise creating test data.

Exercise 1

Method *tri()* takes three integer arguments representing the lengths of the sides of a triangle and returns an integer classifying the triangle or an error. The classifications are equilateral, isosceles, scalene and invalid (a combination of arguments that can not represent a triangle of any sort). Equilateral takes precedence over isosceles, which takes precedence over scalene.

```
final int EQUILATERAL = 0; // All 3 sides equal
final int ISOSCELES = 1;   // Any 2 sides equal
final int SCALENE = 2;    // Valid triangle
final int INVALID = 3;    // Cannot form a valid triangle

/*@ requires 0 < a && 0 < b && 0 < c */
int tri(int a, int b, int c)
```

For the purpose of this exercise, a test case is a pair that represents a possible method input and the method output that is expected to result from this input. For example, a test case for *tri()* might be

Input: (2,2,2) Output: EQUILATERAL

A test case is developed in the black-box style when the test designer is given a description of the purpose of code under test (as above) but is not given the code itself.

Develop a black-box test suite for method *tri()*. The actual suite that you develop is of little interest in this exercise. The interest is in identifying the process that you use to develop the suite. You are expected to devise and follow a systematic process. Once you've done this, write a paragraph describing the process you followed and justifying each test case in your test suite by showing how the process gave rise to it.

Exercise 2

Method *tri()* takes three integer arguments representing the lengths of the sides of a triangle and returns an integer encoding the kind of triangle or an error.

A test case is developed in the glass-box style when the test designer is given access to the code under test as well as a description of its purpose. 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;
}

```

- (a) Develop a glass-box test suite for method *tri()*. As before the interest is in identifying the process that you use to develop the suite. Write a paragraph describing the process you followed and justifying each test case in your test suite by showing how the process gave rise to it. Compare your test suites in this exercise and in Exercise 1. Compare the processes you followed. Describe how they differ.
- (b) Develop a glass-box test suite for method *tri()* with this change
- ```

/*@ requires true */
int tri(int a, int b, int c) { As above }

```
- (c) Compare your two glass-box test suites. Describe how they differ.

### Exercise 3

Develop a black-box test suite for the method *search()* specified as follows.

```

/*@ requires Array ar is non-descending
 @ ensures \result is an integer i such that x is less than all the
 @ elements in ar after position i, and x is greater than or
 @ equal to all the elements in ar before or equal to position i.
 @*/
static int search(int x, int [] ar)

```

In other words, if *x* is in the array *ar*, then the result, *i*, is the position of the last occurrence of *x* in *ar*, and if *x* is not in *ar*, the result is the index after which *x* would be inserted in *ar* in order to maintain the non-descending ordering of *ar*.

### Exercise 4

Consider the following code to implement the specification in Exercise 3. What changes/additions to the test cases would you consider to perform glass-box testing for this code?

```

/*@ requires Array ar is non-descending */
static int search1(int x, int [] ar){
 int i = ar.length-1;

 while (i != -1 && x < ar[i]) {
 i = i - 1;
 }
 return i;
}

```

## Exercise 5

Develop a glass-box test suite for the method *search2()* that also implements the specification in Exercise 3.

```
/*@ requires Array ar is non-descending */
static int search2(int x, int [] ar){
 int lo = -1;
 int hi = ar.length;

 /* Invariant: (lo == -1 || ar[lo] <= x) &&
 (hi == ar.length || x < ar[hi]) &&
 (-1 <= lo < hi <= ar.length) */
 while (lo+1 != hi) {
 int mid = (lo+hi)/2;
 if (ar[mid] <= x) {
 lo = mid;
 } else {
 hi = mid;
 }
 }
 return lo;
}
```

## Exercise 6

Consider the following code for calculating the cost of a flight.

```
class Flight {
 int flightNum;
 String from, to;
 java.util.Date date;
 int maxPassengers; // non-negative
 int currentPassengers; // non-negative
 double fullCost; // non-negative
 int distance; // non-negative

 public Flight(int flightNum, String from, String to,
 java.util.Date date, int maxPassengers,
 int currentPassengers, double fullCost, int distance){
 ...
 }
}

class FlightDatabase {
 public java.util.Iterator iterator() { ... }
 public Flight findFlight(int flightNumber) { ... }
}

class Flights {
 FlightDatabase flights;

 /*@ requires 0 < numPassengers */
 /*@ ensures \result is some flight that that matches from, to, and date
 * and has enough room for the passengers. If there is no such flight,
 * \result is null. */

 Flight find(int numPassengers, String from, String to, Date date) {
 Iterator it = flights.iterator();
 while (it.hasNext()) {
 Flight f = (Flight) it.next();
 if (from == f.from && to == f.to && date == f.date &&
 f.currentPassengers + numPassengers <= maxPassengers) {

```

```

 return f;
 }
}
return null;
}

/*@ 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;
 }
}
}
}

```

Devise suitable glass box test cases to test methods `find` and `flightCost`. The class `FlightDatabase` is incomplete; it is intended that it will be implemented using a database system, but it hasn't been developed yet. Hence to develop test cases for these methods you will need to develop a stub for `FlightDatabase`, i.e., an implementation of it that is sufficient for the purposes of testing only.