Systematic Software Test (II)

Testing OO Software

Anne Haxthausen

Informatics and Mathematical Modelling
Technical University of Denmark

Overview

- **Goal:** To give some *inspiration* for how you can *design* tests for object-oriented software.
- For the *implementation* of tests we recommend tools like JUnit.
- In these foils we discuss how to transfer the theory from Sestoft's note to OO software.

Testing a collection of classes

Make a test for each class.

Testing a single class `C`

Test each of the constructors and methods:

1. design *test cases*
2. execute the test cases
3. evaluate the results

Step 2 (and possibly also step 3) should be done by a *test program*. You have to design and implement such a program or use existing tools like JUnit. In this course you have to do the latter.

Testing a method `m` of a class having no fields

Test case = `input` + `expected output`
Success criteria: `m(input)` is equal to `expected output`

A systematic (functional or structural) test of the methods can be planned (using tables) and implemented as explained in the notes by Peter Sestoft and the other collection of overheads about test.

Testing a method of a class having fields

However, if the class under test contains field variables the situation is more complicated as:

- the effect of a method invocation may not only depend on the input (actual parameters), but also on the state in which it is invoked, and,
- the effect of a method invocation may not only be to return a value (if any at all), but also to change the state.

Test case = `pre state + input + expected output + expected post state`

Success criteria:
`m(input)` is equal to `expected output` &&
`post state` is equal to `expected post state`

Definition: a *state* is a particular contents of the field variables.

Example

```
class Counter {
  private int counter;

  public Counter() { counter = 0; }

  public int getCounter() { return counter; }

  public int increase(int amount) {
    counter = counter + amount;
    return counter;
  }
}
```

In the state where `counter` is 0:
`increase(3)` will return 3, and change the state to one in which `counter` is 3.
In the state where `counter` is 2:
`increase(3)` will return 5, and change the state to one in which `counter` is 5.

Test case tables

The ideas from the notes by Sestoft can be generalized by adding two extra columns in the test case tables:

- pre state
- expected post state

test case id	pre state	input	expected output	expected post state
...				

How can we express the pre and post states?

Approach one: Refer explicitly to the field using their names.

Test case table for `increase` method in `COUNTER`:

test case id	pre state (`counter`)	input (`amount`)	expected output	expected post state (`counter`)
case 1	0	3	3	3
case 2	3	4	7	7

Implementation of test case 1:

```
Counter c = new Counter(); //now c.counter == 0
assertEquals(c.increase(3), 3);
assertEquals(c.counter, 3);
```


How can we express the pre and post states?

Approach one does not always suffice:

- If the test case is implemented in a class different from the class under test, we can't access the private state components (fields like `counter`) directly.
- In a functional test, we do not know anything about which fields exist.

In the first case there is a workaround using the Java Reflection API, see:

`http://www.onjava.com/pub/a/onjava/2003/11/12/reflection.html`

How can we express the pre and post states?

Approach 2:
Use *query methods* returning informations about the state.

Test case table for `increase` method in `COUNTER`:

test case id	pre state (`getCounter()`)	input (`amount`)	expected output	expected post state (`getCounter()`)
case 1	0	3	3	3
case 2	3	4	7	7

Implementation of test case 1:

```
Counter c = new Counter(); //now c.getCounter() == 0

assertEquals(c.increase(3), 3);
assertEquals(c.getCounter(), 3);
```


Designing test cases

In a *functional test*, the test cases must cover "typical" as well as "extreme" input and pre states. Use cases is a good base for making the test cases.

In a *structural test*, there must be enough test cases to make sure that all parts of the code have been executed in the way described in the note by Sestoft.

Besides making test case tables as shown above you should also make additional *survey tables* that give a survey of what is tested in which test cases.

If the state space is large (either because there are many fields or because the data structures of the fields are large) it may be too cumbersome to write up the test case tables.