Introductory Programming, IMM, DTU
Test of Object-oriented Software

An Addendum to the Lecture Note[Ses98] by Sestoft

Anne Haxthausen

July 15, 2005

Abstract

The goal of this note is to give some *inspiration* rather than a recipe for how you can test object-oriented software. A more complete coverage of this topic is beyond the scope of this course.

Contents

1 Testing a collection of classes **1**

2 Testing a single class **2**
2.1 Test cases for applicative methods 2
2.2 Test cases for imperative methods 2
2.2.1 Approach one 3
2.2.2 Approach two 3
2.3 Designing test cases 4
2.4 Test program 5

3 Example **5**
3.1 Test cases 5
3.2 Test program 6
3.3 Results of running the test program 7
3.4 Conclusion 7
3.5 Exercise 7

A `Account.java` **8**

B `TestAccount1.java` **9**

C `TestAccount.java` **10**

1 Testing a collection of classes

Many strategies for doing this exist. For this course we suggest that you use a *bottom-up* strategy: First test those classes that do not depend on other classes. Continue testing classes that only depend on classes that have already been tested.

2 Testing a single class

To systematically test a class you should systematically test each of its constructors and methods.

To systematically test a constructor or a method you should first systematically design *test cases* according to a certain principle (functional or structural), and then implement and execute a test program that *performs* and possibly also *evaluates* the test cases.

In the following sections we will investigate these topics further.

2.1 Test cases for applicative methods

If the method under test is applicative, i.e. the method is static or the class under test does not contain field variables, each *test case* for the method consists of a pair of input and expected output. In [Ses98] it is shown how test cases can be presented in tables. To *perform* and *evaluate* such a test case you should invoke the method under test with the specified input and then compare the output with expected output.

2.2 Test cases for imperative methods

However, if the class under test contains field variables and the method under test is not static, the situation is more complicated as:

- for an object instance of the class, the effect of a method invocation may not only depend on the input (actual parameters), but also on the current state[1] of the object and the class, 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.

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`.
EndExample

In the following discussion, a constructor can just be considered as a special case of a method that does not depend on the state, but only on input, and that does not return a value, but only initializes the state.

[1] The (current) *state* of an object is the current contents of the (non-static) field variables of the object, and the (current) *state* of a class is the current contents of the static field variables.

2.2.1 Approach one

The above observations leads to the following suggestion:

- A *test case* for a method should not only specify input (if any) and expected output (if any[2] output), but also
  - a *pre state* (i.e. the state in which the method invocation should take place), and,
  - an *expected post state* (i.e. the state expected just after the method invocation has taken place).
- To *perform and evaluate a test case*, you should invoke the method with the specified input in the specified pre state, and then not only compare the output with expected output, but also compare the actual post state with the expected post state.

The format in [Ses98] for a *test case table* for a method can be generalized by adding columns for pre state and expected post state. Hence, the format becomes:

test case id	pre state	input	expected output	expected post state

The input column should consist of as many sub-columns as there are formal parameters, and each of the two state columns should consist of as many sub-columns as there are fields. For methods that do not take input and/or return any output the corresponding columns should be removed.

As an example, a test case table for testing the `increase` method of the `COUNTER` class could look like this:

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

Here is a test case table for the `COUNTER` constructor:

test case id	expected post state (`counter`)
case 0	0

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 tables. Instead one may choose immediately to write the test cases in the test program.

2.2.2 Approach two

The first approach does not always suffice:

- If the test program (that should perform the test) should be written in a class different from the class under test, it can't access the private state components (fields like `counter`) directly.
- In a functional test, we do not know anything about which fields exist.

[2] A method having return type `void` does not return any value.

Under these circumstances, in the test cases and test case tables you can't explicitly refer to fields. Instead you can use *query methods*, i.e. public methods that return information about the state.

As an example, for the `COUNTER` class, you can use `getCounter()` and the test case table of the `increase` method could look like:

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

Here test case 1 states that in a state for which `getCounter()` gives `0`, an invocation `increase(3)` is expected to return the value `3` and lead to a new state for which `getCounter()` gives `3`.

Instead of using a "get-method" for each state component you may like to use single query method like `toString()` that returns a value representing the whole state.

Observe that in this approach a test case specifies a test of not just a single method in isolation, but rather a test of the method in combination with one or more appropriate query methods.

Note that this approach for writing tables only works for query methods taking no arguments. If they take arguments you could for instance use one of the other approaches mentioned in next paragraph.

Other approaches The ideas presented above may be further generalized. Here we list some hints about that, but it is outside the scope of this course to go into details.

Instead of specifying the pre state and expected post state for a *single* method invocation, it may be for a *sequence* of method invocations. Then inputs for each of the invoked methods should be specified.

Instead of specifying a specific input and a specific pre state (or specific output of query methods in the pre state) one may specify a *pre condition*, i.e. a predicate that these items should satisfy. For instance, that an integer input value is greater than `0`.

Similarly, one may use *post conditions* instead of specifying a specific output and a specific expected post state.

One may also wish to let a test case specify that two sequences of method invocations results in the same returned value and state. This idea is well suited for abstract data types / abstract data structures[3] that have been axiomatically described.

2.3 Designing test cases

In a *functional test*, the test cases must cover "typical" as well as "extreme" input and pre states.

In a *structural test*, there must be enough test cases to make sure that all parts of the code have been executed in a certain way (according to a chosen level of coverage). In this course you should use the coverage criteria explained in [Ses98].

Besides presenting test cases in test case tables, Sestoft [Ses98] suggests additional tables that describe what the test cases are testing.

[3]You will learn more about what this means in later courses.

For a structural test he uses a table that map each possible choice of flow through the code (e.g. the choice of going through the else branch of an if statement, because the if-condition is false) to a test case.

choice	test case id

An alternative, also appealing format (with the same purpose) can be found in appendix C of [SM02].

For a functional test one can use a table that characterizes each test case by a certain property of the input and/or pre state.

2.4 Test program

For each class `C` to be tested, a test program (a main method) executing the test cases designed for that class should be made. The main method could be placed in the class it-self or in a separate test class, say `TestC`.

The test program should perform and possibly evaluate each test case. A typical algorithm for test cases as described in first and second approach above is as follows:

1. invoke the method under test with the specified input in the specified pre state
2. treat the results, i.e. the returned output and observations of the new state (either values of fields or values returned by query methods) of the invocation

What does it mean to treat the results? In a simple test program it may just be to print the results on the screen, so that one can manually compare with expected results. However, a more advanced test program may investigate whether the results are as expected and write some test report.

Various tools for assisting the creation of test programs exist, e.g. JUnit for Java.

3 Example

Here we give an example of a structural test for the `Account` class shown in appendix A.

3.1 Test cases

Except for the `deposit` method, all methods of this class have bodies containing no "choices" (branches or loops). So for each of these methods we should just ensure that there is at least one test case making a call of that method. The `deposit` method has a "choice" determined by the value of a simple logical expression `(amount >= 0)`, so we need two test cases for deposit: one in which the expression is true and one in which it is false.

Here is an survey of in which test cases the various methods and choices for these are testet:

method/constructor	choice	test case id(s)
`Account`	n/a	test0
`getBalance`	n/a	test[0-3]
`getNumber`	n/a	test[0-3]
`toString`	n/a	test1
`deposit`	`(amount >= 0)` true	test2
`deposit`	`(amount >= 0)` false	test3

Note that no choice depends on the internal state.

The test cases can be characterised as follows wrt. expected changes of the state:

test0 Initial state created by constructor

test1 No change caused by toString

test2 Proper change by valid deposit

test3 No change with a negative deposit

Test cases for `Account` in combination with `getBalance` and `getNumber`:

test case id	input		expected post state	
	`account`	`initial`	`getNumber()`	`getBalance()`
test0	436019	1000.00	436019	1000.00

Test cases for `toString` in combination with `getBalance` and `getNumber`:

test case id	pre state		expected output	expected post state	
	`getNumber()`	`getBalance()`		`getNumber()`	`getBalance()`
test1	436019	1000.00	"436019 $1,000.00"	436019	1000.00

Test cases for `deposit` in combination with `getBalance` and `getNumber`:

test case id	pre state		input	expected output	expected post state	
	`getNumber()`	`getBalance()`			`getNumber()`	`getBalance()`
test2	436019	1000.00	5000.00	6000.00	436019	6000.00
test3	436019	6000.00	-2.50	6000.00	436019	6000.00

Note how the test cases are designed so that the pre state of test*i* is equal to the expected post state of test*i-1*, for $i = 1, \ldots 3$.

3.2 Test program

A test program is shown in appendix B. It executes each of the test cases and automatically evaluates whether actual results are as expected. A test report is written to the screen. The report informs for each of the cases whether it passed or failed.

Another test program behaving in the same way, but having the property that each test case can be excuted by a single method call, is shown in appendix C.

The latter program uses an auxiliary method named `checkState`. The parameters of `checkState` have meanings as follows

obj A reference to the object being tested

id An identification of a test case

res Tells if an observed value is as expeced

expNumber An expected result of obj.getNumber()

expBalance An expected result of obj.getBalance()

Note that the lines that invoke `checkState` have been written so that they look like a table almost similar to those of section 3.1. There is no equivalent to the pre state columns as the pre state of test*i* is equal to the expected post state of test*i-1*.

As mentioned earlier, if the state space is large it may be too cumbersome to write up tables as we did for this example. Instead one may choose immediately to write the test cases in the test program in such a table like form.

3.3 Results of running the test program

Here is the output from the test program:

```
test0 passed the test.

test1 passed the test.

test2 passed the test.

test3 passed the test.
```

3.4 Conclusion

The structural test of `Account` did not reveal any errors.

3.5 Exercise

1. How can you improve the test program to give better reports?
2. Suggest how test case 3 and the corresponding part of the test program should be changed, if the `deposit` method was changed to throw an Exception for negative amount input?

Acknowledgements The author would like to thank Hans Henrik Løvengreen, Jørgen Steensgaard-Madsen and Jens Thyge Kristensen for valuable comments and discussions.

References

[Ses98] Peter Sestoft. Systematic software test. Technical report, Department of Mathematics and Physics, Royal Veterinary and Agricultural University, Denmark, 1998.

[SM02] Jørgen Steensgaard-Madsen. Documentation of software systems. Technical report, Technical University of Denmark, October 9, 2002.

A Account.java

```
//********************************************************************
//  Account.java
//  Author: Lewis/Loftus
//  Shortened version by Anne Haxthausen, 11/11-04
//
//  Represents a bank account with basic services such as deposit.
//********************************************************************

import java.text.NumberFormat;

public class Account {
    private NumberFormat fmt = NumberFormat.getCurrencyInstance();

    private long number;    //account number
    private double balance; //current balance of the account

   //-----------------------------------------------------------------
   //  Sets up the account by defining its account number,
   //  and initial balance.
   //-----------------------------------------------------------------
   public Account (long account, double initial) {
      number = account;
      balance = initial;
   }

   //-----------------------------------------------------------------
   //  Validates the transaction, then deposits the specified amount
   //  into the account. Returns the new balance.
   //-----------------------------------------------------------------
   public double deposit (double amount) {
      if (amount >= 0)  // deposit value is not negative
        balance = balance + amount;

      return balance;
   }

   //-----------------------------------------------------------------
   //  Returns the account number.
   //-----------------------------------------------------------------
   public long getNumber () {
      return number;
   }

   //-----------------------------------------------------------------
   //  Returns the balance.
   //-----------------------------------------------------------------
   public double getBalance () {
      return balance;
   }


   //-----------------------------------------------------------------
   //  Returns a one-line description of the account as a string.
   //-----------------------------------------------------------------
   public String toString () {
      return (number + "␣" + fmt.format(balance));
   }
}
```

B TestAccount1.java

```
//*********************************************************************
//  TestAccount1java
//  Author: Anne Haxthausen, 11/11-04
//
//  A test driver for the Account class.
//*********************************************************************
public class TestAccount1 {
  public static void main (String[] args) {
    boolean ok; //aux. variable indicating whether the current test passed


    //test0:
    Account a = new Account(436019, 1000.00);
    ok =  (a.getNumber() == 436019) && (a.getBalance() == 1000.00);
    report("test0", ok);

    //test1:
    ok =  (a.toString()).equals("436019 $1,000.00") &&
          (a.getNumber() == 436019) && (a.getBalance() == 1000.00);
    report("test1", ok);

    //test2:
    ok =  (a.deposit(5000.00) == 6000.00) &&
          (a.getNumber() == 436019) && (a.getBalance() == 6000.00);
    report("test2", ok);

    //test3:
    ok = (a.deposit(-2.50) == 6000.00) &&
         (a.getNumber() == 436019) && (a.getBalance() == 6000.00);
    report("test3", ok);

  }

  //Reports whether the test having test case id testCaseId passed or failed
  private static void report(String testCaseId, boolean ok) {
    if (ok)
      System.out.println(testCaseId + " passed the test.\n");
    else
      System.out.println(testCaseId + " failed the test.\n");
  }
}
```

C TestAccount.java

```
//*********************************************************************
//  TestAccount.java
//  Author: Anne Haxthausen and Jørgen Steensgaard-Madsen, 11/11-04
//
//  A test driver for the Account class.
//*********************************************************************
public class TestAccount {
  public static void main (String[] args) {

    Account a = new Account(436019, 1000.00);  // object to test

    /* test cases test0-3: */

    checkState(a, "test0", true,                                436019, 1000.00);
    checkState(a, "test1", (a.toString()).equals
                              ("436019␣$1,000.00"),             436019, 1000.00);
    checkState(a, "test2", a.deposit(5000.00) == 6000.00,       436019, 6000.00);
    checkState(a, "test3", a.deposit(  -2.50) == 6000.00,       436019, 6000.00);

  }

  // Checks that getNumber() and getBalance() of obj return expected values
  // and that an auxiliary result, res, is true. Reports accordingly.

  private static void

      checkState(Account obj,         // the object being tested
                 String id,           // test case identification
                 boolean res,         // validation of observed result
                 long expNumber,      // expected result of getNumber()
                 double expBalance    // expected result of getBalance()
                )
  {
    if (res
        && obj.getNumber() == expNumber
        && obj.getBalance() == expBalance
       )
        System.out.println(id + "␣passed␣the␣test.\n");
    else
        System.out.println(id + "␣failed␣the␣test.\n");
  }

}
```