

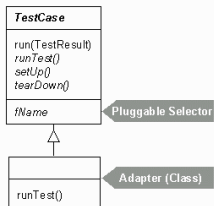
The right B.I.C.E.P.S.

Pieter van den Hombergh

13 september 2005

- 1 Intro
- 2 Simplified architecture of JUnit
- 3 Right Bicep(s)
 - Right
 - Boundary Conditions
 - Inverse relation
 - Cross check
 - Error conditions, Exceptions
 - Performance

- 1 What is the purpose of setup and tearDown?
- 2 How can you compose test suites



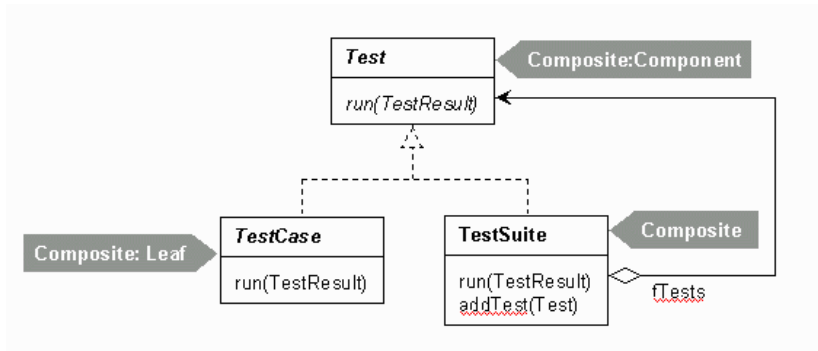
For more details on the architecture of junit have a look at <http://junit.sourceforge.net/doc/cookstour/cookstour.htm>

Testcase implements a *template method*

Testcase implements the run method, required by the Test interface as a template method.

```
public void run () {  
    setUp ();  
    runTest ();  
    tearDown ();  
}
```

Testsuite is part of a composite pattern



See module OOAD. By the way. Although JUnit is a smallish package, it is already a nice exercise to explain the design patterns used.

Right Biceps



Are we testing the **Right** B.I.C.E.P.s.?

Testing without a strategy is futile. The next few sheets will help you remember a simple strategy.

Right BIPCEPs

Do you testing the Right BICEPs?
Right Are the results right?



Right BIPCEPs

Do you testing the Right BICEPs?

Right Are the results **right**?

B Are all the **Boundary** conditions correct?



Right BIPCEPs

Do you testing the Right BICEPs?

- Right Are the results **right**?
- B Are all the **Boundary** conditions correct?
- I Can you test the **Inverse** relationship?



Right BIPCEPs

Do you testing the Right BICEPs?

- R**ight Are the results **right**?
- B** Are all the **Boundary** conditions correct?
- I** Can you test the **Inverse** relationship?
- C** Can you **Cross-check** using other methods/means?



Right BIPCEPs

Do you testing the Right BICEPs?

- Right Are the results **right**?
- B Are all the **Boundary** conditions correct?
- I Can you test the **Inverse** relationship?
- C Can you **Cross-check** using other methods/means?
- E Can you force **Error conditions** to happen?



Right BIPCEPs

Do you testing the Right BICEPs?

- Right Are the results **right**?
- B Are all the **Boundary** conditions correct?
- I Can you test the **Inverse** relationship?
- C Can you **Cross-check** using other methods/means?
- E Can you force **Error conditions** to happen?
- P Are **Performance** conditions within bounds?



Right BIPCEPs

Do you testing the Right BICEPs?

- Right Are the results **right**?
- B Are all the **Boundary** conditions correct?
- I Can you test the **Inverse** relationship?
- C Can you **Cross-check** using other methods/means?
- E Can you force **Error conditions** to happen?
- P Are **Performance** conditions within bounds?
- s Always go for the whole **set**.



If it ran correctly, how would I know?

If there is a way to collect the input of a class/method and the correct result, then put them in a file and read them from there. Something along the line:

```
# This file contains test data for a max finding
# algorithm. Hashes (#) introduce comment
# format: <expected> <value>+
# all space separated.
9 1 4 7 -1 9
1 1
-1 -2 -4 -1
```

Your test method could read this data, create the test array and then invoke the method under test.

Some boundary conditions

- Totally bogus or inconsistent values, such as filenames.
- Badly formatted data like '1970-23-12' when 1970-12-23 is required.
- Empty or missing values.
- Validity like age in years between 0 and e.g. 115.
- Duplicates in list that should not have any.
- Ordered list that aren't.
- Things that arrive out of order.

Are the boundaries C.O.R.R.E.C.T.?

C Conformance. Is the format Ok?

Are the boundaries C.O.R.R.E.C.T.?

- C Conformance. Is the format Ok?
- O Ordering. Is the set (un)ordered as appropriate e.g. as dictated by invariant.

Are the boundaries C.O.R.R.E.C.T.?

- C Conformance. Is the format Ok?
- O Ordering. Is the set (un)ordered as appropriate e.g. as dictated by invariant.
- R Is the value within acceptable Range?

Are the boundaries C.O.R.R.E.C.T.?

- C Conformance. Is the format Ok?
- O Ordering. Is the set (un)ordered as appropriate e.g. as dictated by invariant.
- R Is the value within acceptable Range?
- R Does the code Reference anything external not under its direct control?

Are the boundaries C.O.R.R.E.C.T.?

- C Conformance. Is the format Ok?
- O Ordering. Is the set (un)ordered as appropriate e.g. as dictated by invariant.
- R Is the value within acceptable Range?
- R Does the code Reference anything external not under its direct control?
- E Does the value Exist?

Are the boundaries C.O.R.R.E.C.T.?

- C Conformance. Is the format Ok?
- O Ordering. Is the set (un)ordered as appropriate e.g. as dictated by invariant.
- R Is the value within acceptable Range?
- R Does the code Reference anything external not under its direct control?
- E Does the value Exist?
- C Cardinality or countability; Are there enough values?

Are the boundaries C.O.R.R.E.C.T.?

- C Conformance. Is the format Ok?
- O Ordering. Is the set (un)ordered as appropriate e.g. as dictated by invariant.
- R Is the value within acceptable Range?
- R Does the code Reference anything external not under its direct control?
- E Does the value Exist?
- C Cardinality or countability; Are there enough values?
- T Tim(e)ing). Is everything happening in the right order?

To test e.g. a square root algorithm, use the inverse relation:

```
public void testSquareRootUsingInverse () {  
    double x= mySquareRoot (4.0);  
    assertEquals ( " Square_root_does"+  
        " _not_match_inverse" ,  
        x*x ,  
        0.0001 );  
}
```

Be carefull not t derive the inverse test from the code under test.
A flaw in the code mught mask an error.

B.I.Cross check.E.P.s.

If e.g. you are devising a square root for an embedded system without float capability, you could test it (*in the host environment*) using the standard implementation, like so:

```
public void testSquareRootUsingStd () {  
    double number=3830900.0;  
    double root1= mySquareRoot(number);  
    double root2= Math.sqrt(number);  
    assertEquals( root2 , root1 , 0.0001 );  
}
```

B.I.C.Errors.P.s.

The world is harsh, out there. . .

Can your module survive

- Out of memory. Do not wipe the disk as well. . .
- Running out of disk space.
- Strange things in time keeping (like a clock that has been set back or summer and winter time).
- Network failure.
- High system load.
- Limited color availability in user interface.
- Very high or very low screen resolution.

B.I.C.E.P Performance.s.

A normal unit test should test for the functional requirements. But if the non-functional requirement performance or speeds depend on the size of the data, you should test that too.

Example: a spam filter that tests against “naughty” or black-listed mail servers. Tests your code with different lists sizes. Typically a small list for build testing, but maybe also with a longer list e.g. once a day.

B.I.C.E.P Performance.s.

```
import junit.framework.TestCase;
class BlackListTest extends TestCase {
    String naughtyServer;
    Timer timer;
    public void setup() {
        Timer= new Timer();
        naughtyServer="smtp.xxxxx.com";
    }
    public void tearDown(){
        timer = null;
    }
    public void testBlackListFilterSmall(){
        BlackListFilter f =
            new BlackListFilter( small_list );
        assertTrue( timeBlackListFilter(f, timer) < 0.1 );
    }
    public void testBlackListFilterHuge(){
        BlackListFilter f =
            new BlackListFilter( small_list );
        assertTrue( timeBlackListFilter(f, timer) < 3.0 );
    }
    private double timeBlackListFilter( BlackListFilter f, Timer timer ) {
        // note the use of a method to do the timing
        timer.start();
        f.check();
        timer.end();
        return timer.elapsedTime();
    }
}
```