

# Software Development (cs2500)

## Lecture 17: Fixing Bugs with JUnit Tests

M. R. C. van Dongen

October 30, 2013

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Assertions</b>	<b>1</b>
<b>3</b>	<b>What is Unit Testing?</b>	<b>2</b>
<b>4</b>	<b>Why Unit Testing?</b>	<b>2</b>
<b>5</b>	<b>Concrete Unit Tests</b>	<b>3</b>
5.1	Are the Results RIGHT? . . . . .	3
5.2	Boundary Conditions . . . . .	4
5.3	Inverse Relationships . . . . .	4
5.4	Cross-checking . . . . .	5
<b>6</b>	<b>First JUnit Tests</b>	<b>6</b>
<b>7</b>	<b>JUnit Assertions</b>	<b>8</b>
<b>8</b>	<b>Test Wrappers</b>	<b>8</b>
<b>9</b>	<b>Class Wrappers</b>	<b>10</b>
<b>10</b>	<b>Tests with Timeout</b>	<b>11</b>
<b>11</b>	<b>Acknowledgements</b>	<b>12</b>
<b>12</b>	<b>For Friday</b>	<b>12</b>

## 1 Introduction

This lecture is about *unit testing* with JUnit4, which is a Java framework that allows you to write simple tests to test your methods in isolation. The material from this lecture about JUnit4 is not

examinable for the written examination. However, you may have to do some JUnit4 testing for some of the assignments. This lecture is mainly based on [Ferguson Smart 2008, Chapter 10], [Hunt, and Thomas 2007], and [Sommerville 2007, Chapter 23]. More information about JUnit testing may be found on [http://en.wikipedia.org/wiki/Unit\\_test](http://en.wikipedia.org/wiki/Unit_test).

## 2 Assertions

Before studying unit testing with Java's JUnit4 framework, it is useful to study Java's *assertion* mechanism, which is another way to locate errors in the logic of your programs. You add assertions to your program using Java's `assert` method: Each assertion should state a condition that is true.

```
public class TestAssert {  
    public static void main( String[] args ) {  
        int value = 1;  
        assert( value == 1 );  
        value = 2; // Simulate error.  
        assert( value == 1 );  
        System.out.println( "value = " + value );  
    }  
}
```

Java

With assertions you compile your sources as per usual. When you run your program you enable assertion checking if you add the `-ea` flag. This forces java to check the assertions and throw an error for a failed assertions.

```
$ java -ea TestAssert  
Exception in thread "main" java.lang.AssertionError  
    at TestAssert.main(TestAssert.java:6)  
$
```

Unix Session

By default, assertion checking is off, so running java without `-ea` “turns” assertion checking off.

```
$ java TestAssert  
value = 2  
$
```

Unix Session

The following are some advantages and disadvantages of assertions:

- Assertions are an easy mechanism to integrate tests and code.
- It is easy to turn tests on and off.
- Assertions in programs provide a form of documentation: they state conditions that should be true.
- Assertions provide an easy mechanism for *integrated* testing (testing as part of the whole application).
- However, assertions do not allow you to test your individual methods in isolation.

### 3 What is Unit Testing?

A *unit* (component) is the smallest testable part of an application. A *unit test* is a test that verifies the individual units of your application are working properly. Unit testing is also known as *component testing*. Many companies have special *test engineers* that test software written by software developers. Unit tests are written by the software developers themselves.

The goal of unit testing is to show that the individual units are correct. Unit testing is done right from the early stage of development.

### 4 Why Unit Testing?

Unit testing has the following advantages.

**Defect testing:** Unit testing is a *defect testing process*: its goal is to expose faults in the components. Before you read on, make sure you understand this. Writing tests that succeed regardless of errors doesn't help: the tests *must* be designed to break your code.

**Provides confidence:** Tests eliminate uncertainty about the units.

**Automates testing:** Unit tests can be automated.

**Regression testing:** It supports *regression testing*, which is a form of testing which insists on (1) repeating all previous tests, (2) repeating all new tests, and on (3) making sure all tests pass.

**Robustness to changes:** Tests remain valid after changes to the code of the unit.

**Simplifies integration:** Bugs are eliminated at an early stage. This saves much time. For example, chasing bugs when you're integrating several classes is easier if each of the classes are bug free. If the classes aren't bug free then it may be difficult to determine what is causing unexpected behaviour. The trade-off is not "test now" versus "test later" but "linear increase in testing and coding effort now" versus "exponential testing effort later."

**Documentation:** Unit tests provide a form of documentation of the unit API.

**Contract:** Each test is a written contract that the unit must satisfy (this is related to the previous item).

**Drives design:** Writing a unit test before writing the unit may drive the design of the unit. For example, the test documents (some) of the intended behaviour of the unit, which may be used in addition to a formal specification of method behaviour.

**Separates testing:** Unit tests allow testing of your methods in *isolation* from other classes. This allows more thorough tests.

## 5 Concrete Unit Tests

This section provides some concrete suggestions for unit tests. Your “**RIGHT-BICEP**” may be the key to successful unit testing.

**RIGHT results:** Are the results right?

**Boundary conditions:** Are the boundary conditions correct?

**Inverse relationships:** Can you check inverse relationships?

**Cross-checking:** Can you cross-checks results using different means?

**Error conditions:** Can you force error-conditions to happen?

**Performance:** Are the performance characteristics within bound?

### 5.1 Are the Results RIGHT?

These are simple tests to verify if the results are correct. Given the input these tests should be able to decide what would be the right result.

- Given numbers 2, 4, and 1: which is the larger?
- Given numbers 45, and 2345: which is the larger?
- Given number 1, what is the absolute value?
- Given number -4, what is the absolute value?

### 5.2 Boundary Conditions

Many bugs are caused by range or boundary errors:

**Range:** This class of errors are related to the allowed values in “a general way”. For example:

- Wrong file extension.
- Bogus input: surname is cw4vr@:.
- Missing values in array declarations.
- Unreasonable input: a 10000 years old person.
- Ordered lists aren’t sorted.
- Negative/positive numbers.
- Sequencing errors.

**Boundary conditions:** These errors are caused by exceptional values or forgotten values at the start or end of a range.

- Fence-post problems.
- Other off-by one errors.
- Empty lists.
- Division by zero.

When checking boundary conditions, it helps to check if they are **CORRECT**:

**Conformance:** Does the value conform to an expected format?

**Ordering:** Are the values correctly ordered? Should the result be independent of the ordering?

**Range:** Is the value within reasonable minimum and maximum values?

**Reference:** Are there references to objects in other classes? If yes, are the conditions for referencing them right?

**Existence:** Is the value non-zero, is it present in a set, ...? Is the string non-empty? Is the reference non-null?

**Cardinality:** Is the number of things correct? Are there off-by-one errors?

**Time:** Are things happening at the right time, in the right order?

## 5.3 Inverse Relationships

Some results can be cross-checked using *inverse relationships*. For example:

- $x = \sqrt{x^2}$ , provided  $x$  is non-negative.
- $x = -(-x)$ .
- $x = (2x)/2$ .

Inverse relationships are ideal for creating unit tests. For example, if you have to implement a method that computes  $f(x)$ , where  $f(\cdot)$  is some given function. Furthermore, let's assume that you know how to compute an inverse of  $f^{-1}(x)$  for certain allowed values  $x$ . Given this background you can assert that  $x = f^{-1}(f(x))$  should hold, for one or several allowed  $x$ . Furthermore, you can assert that  $x = f(f^{-1}(x))$  for allowed  $x$ .

Inverse relationships are also excellent relationships for tests with pseudo-random data in for-loop.

Testing other relationships may also provide confidence. For example:

- There may be recurrence relations:  $f_{n+2} = f_{n+1} + f_n$ ,  $m! = (m-1)! \times m$ , .... Such recurrence relations are ideal. For example, let's assume you've implemented a method `fibonacci` that satisfies the first recurrence relations. You can simply plug in your "method calls" into the recurrence relation.

```
for (int n = 0; n <= MAX_N; n++) {  
    final int fib2 = fibonacci( n + 2 );  
    final int fib1 = fibonacci( n + 1 );  
    final int fib0 = fibonacci( n + 0 );  
    assertTrue( fib2 == fib1 + fib0 );  
}
```

Java

- Solutions may be known given certain boundary conditions. For example:

- ★ You may know the answer if the input is even.

```
for (int n = -MAX_N; n <= MAX_N; n++) {
    final int newAnswer = myNewMethod( 2 * n );
    final int evenAnswer = myAnswerForEvenNumbers( 2 * n );
    assertTrue( newAnswer == evenAnswer );
}
```

Java

- ★ You may know the answer if the input is a power of 2.
- ★ ...
- ....

## 5.4 Cross-checking

Can you cross-check results using other means?

- Do you know a less efficient way to compute the result? For example,
  - ★ For example, you have to compute  $f(n) = \sum_{i=0}^n i$ .
  - ★ You implement a clever way to compute it:  $f(n) = n(n+1)/2$ .
  - ★ You can check the result using a simple for-loop.
- Can you check against a previous release? This may be considered an instance of the previous item if your previous release is correct(?).
- Is there a database with test-cases?

## 6 First JUnit Tests

This section demonstrates how to implement simple unit tests using Java's JUnit4 framework. As part of this section we shall develop some unit tests for the following method, which contains some deliberate errors. Throughout, the names of the identifiers and coding style have been kept simple to keep the examples short.

```
public class Largest {
    public static int largest( int[] ints ) {
        int max = Integer.MAX_VALUE;
        for (int i = 0; i < ints.length - 1; i++) {
            if (ints[ i ] > max) {
                max = ints[ i ];
            }
        }
        return max;
    }
}
```

Java

The following is a class called `TestLargest` that defines a unit test for the method `largest`. The `@Test` annotation indicates that the method following it is a method that should be called in isolation as a unit test. As you can see from the example, you may specify more than one test. The static import statement is usually frowned upon. However, here it is used to avoid having to

fully qualify all references to the class `Assert`. In short, it allows you to write `assertEquals( ... )` instead of `Assert.assertEquals( ... )`.

```
import static org.junit.Assert.*;
import org.junit.Test;

public class TestLargest {
    @Test
    public void orderTest( ) {
        final int[] ints = new int[] {7,6,8,9};
        assertEquals( 9, Largest.largest( ints ) );
    }

    @Test
    public void success( ) {
        // Simulate successful test.
    }
}
```

Before we can run the test we have to adjust and export the java CLASSPATH.

```
$ CLASSPATH=${CLASSPATH}:/usr/share/java/junit4.jar:.
$ export CLASSPATH
```

It is probably a good idea to add these two statements to your `.bashrc` file. That way you don't have to carry them out by hand each time you want to run a JUnit test.

Now that the CLASSPATH is set up, you can apply the test. Notice that this is done in isolation, i.e. without the need for a main. This is one of the great advantages of the JUnit framework: you don't have to write test drivers classes with test driver `main( )` methods in them any more.

```
$ javac Largest.java TestLargest.java
$ java org.junit.runner.JUnitCore TestLargest
...
Time: 0.007
JUnit version 4.3.1
.E
Time: 0.009
There was 1 failure:
1) orderTest(TestLargest)
java.lang.AssertionError: expected:<9> but was:<2147483647>
...
FAILURES!!!
Tests run: 2, Failures: 1
$
```

It looks as if we've just found a bug. On closer inspection we notice that the assignment 'int max = Integer.MAX\_VALUE' should have been 'int max = Integer.MIN\_VALUE'.

```
$ javac Largest.java TestLargest.java
$ java org.junit.runner.JUnitCore TestLargest
...
Time: 0.007
JUnit version 4.3.1
.E.
Time: 0.012
There was 1 failure:
1) orderTest(TestLargest)
java.lang.AssertionError: expected:<9> but was:<8>
...
FAILURES!!!
Tests run: 2, Failures: 1
$
```

Oops, it looks like we still haven't got rid of all bugs yet.

This time, finding the bug takes slightly more time. To locate the bug, it helps noticing that the method `largest` failed to find the maximum 9, which is the *last* member of the array. Interestingly, the method did seem to be able to locate the maximum of the numbers before the 9. Errors like this are usually caused by off-by-one errors.

One more look at the method `largest` and we quickly see that the termination condition for the for statement is wrong: it should have been 'i <= ints.length + 1' or (better) 'i < ints.length'.

After fixing this embarrassing error we once more run our unit tests.

```
$ javac Largest.java TestLargest.java
$ java org.junit.runner.JUnitCore TestLargest
...
JUnit version 4.3.1
..
Time: 0.011

OK (2 tests)
$
```

Whew! Looks like we're out of trouble. But can we really be sure?

## 7 JUnit Assertions

The following are some *assertions* which are provided by JUnit4.

- `assertEquals( expected, actual )` compares expected and actual and fails if the two are not equal. Possible types for the arguments are boolean, int, short, .... For object types, the method compares for deep equality.
- `assertEquals( expected, actual, tolerance )` compares expected and actual and aborts if they are not within the given tolerance. This method is used for comparing floating



point types.

- `assertNull( object )` asserts that `object == null`.
- `assertNotNull( object )` asserts that `object != null`.
- `assertSame( expected, actual )` asserts that `object` and `actual` are aliases (shallowly equal).
- `assertNotSame( expected, actual )` asserts that `object` and `actual` are not aliases.
- `assertTrue( condition )` asserts that `condition` is true.
- `assertFalse( condition )` asserts that `condition` is false.
- `fail( )` fails immediately.

All methods take an optional first `String` argument, which is output in case of failure:

```
assertEquals( "Should be 3 1/3", 3.33, 10.0/3.0, 0.01 );
```

Java

## 8 Test Wrappers

Many tests require access to some resource that has to be set up (initialised) prior to the test and torn down after the test. For example, the test may assume that it has a connection to a database. For the test to work, the database connection has to be set up before the test and closed after the test.

To support this kind of functionality, the JUnit4 framework provides the following annotations:

**@Before:** This annotation indicates the method that initialises the test.

**@After:** This annotation indicates the method that tears down the test.

Figure 1 depicts an example. Note that this example is not about implementing meaningful unit tests but about showing the per-test set up and tear down mechanism.

```
$ javac TestBeforeAfter.java
$ java org.junit.runner.JUnitCore TestBeforeAfter
JUnit version 4.3.1
.Initialising
Tearing down. Value is: 4
.Initialising
Tearing down. Value is: 5

Time: 0.04

OK (2 tests)
$
```

Unix Session

Notice that the output clearly shows that each test is individually set up and torn down.

## 9 Class Wrappers

As you may have guessed, per-class setting up and tearing down is also supported. This mechanism is useful for setting up and tearing down “expensive” resources that take much time to initialise

---

```
import org.junit.Before;
import org.junit.After;
import org.junit.Test;

public class TestBeforeAfter {
    private int value;

    @Before
    public void initialise( ) {
        System.out.println( "Initialising" );
        value = 1;
    }

    @After
    public void tearDown( ) {
        System.out.println( "Tearing down. Value is: " + value );
    }

    @Test
    public void test1( ) {
        value += 3;
    }

    @Test
    public void test2( ) {
        value += 4;
    }
}
```

---

Figure 1: Setting up and tearing down JUnit tests.

---

and/or tear down.

To support this kind of functionality, the JUnit4 framework provides the following annotations:

**@BeforeClass:** This annotation indicates the *class* method that initialises the test.

**@AfterClass:** This annotation indicates the *class* method that tears down the test.

The following is an example. Again, note that this example is not about implementing meaningful unit tests but about showing the per-class set up and tear down mechanism.

```
import org.junit.BeforeClass;
import org.junit.AfterClass;
import org.junit.Test;

public class TestBeforeAfterClass {
    private static int value;

    @BeforeClass
    public static void initialiseClass( ) { value = 1; }

    @Test public void test1( ) { value += 3; }
    @Test public void test2( ) { value += 4; }

    @AfterClass
    public static void tearClassDown( ) {
        System.out.println( "Tearing down. Value is: " + value );
    }
}
```

```
$ javac TestBeforeAfterClass.java
$ java org.junit.runner.JUnitCore TestBeforeAfterClass

JUnit version 4.3.1
..Tearing down. Value is: 8

Time: 0.012

OK (2 tests)

$
```

## 10 Tests with Timeout

Many programs critically depend on time. For example, user interfaces should be responsive. Other computations cannot take forever. Testing with a maximum computation makes sense.

- Catches errors due to slow response time;

- May catch infinite loops;
- ....

The JUnit4 framework allows you to specify a maximum computation time as part of the `@Test` annotation. Specifically, the notation `@Test(timeout=<ms>)` annotates a method as a test method with a maximum computation time of '`<ms>`' milliseconds. A test that takes more than its allowed specified timeout is considered a failed test.

The following demonstrates how timeouts work.

```
import org.junit.Test;

public class TestTimeout {
    @Test(timeout=10)
    public void failure( ) {
        for (int index = 0; ; ) ;
    }

    @Test(timeout=1)
    public void success( ) {
    }
}
```

```
$ javac TestTimeout.java
$ java org.junit.runner.JUnitCore TestTimeout
JUnit version 4.3.1
.E.
Time: 0.04
There was 1 failure:
1) failure(TestTimeout)
java.lang.Exception: test timed out after 10 milliseconds
:
FAILURES!!!
Tests run: 2, Failures: 1
$
```

## 11 Acknowledgements

This lecture is based on [Furguson Smart 2008, Chapter 10], [Hunt, and Thomas 2007], and [Somerville 2007, Chapter 23]. Further information about JUnit testing may be found at [http://en.wikipedia.org/wiki/Unit\\_test](http://en.wikipedia.org/wiki/Unit_test).

## References

Furguson Smart, John [2008]. *Java Power Tools*. O'Reilly. ISBN: 978-0-596-52793-8.

Hunt, Andrew, and David Thomas [2007]. *Pragmatic Unit Testing In Java with JUnit*. The Pragmatic Programmers. ISBN: 978-09745140-1-7.  
Sommerville, Ian [2007]. *Software Engineering*. Eight Edition. Addison Wesley. ISBN: 978-0-321-31379-9.

## **12 For Friday**

Study the lecture notes and study Chapter 6.