# JMUnit 1.2

Java Micro Unit

Brunno Silva
Carl Meijer

August 2008

## 1 Unit-Tests and Test-Driven Development

A *unit test* is a test that exercises only one piece of functionality within a piece of production code. Since unit tests execute so little code, they run fast which encourages frequent testing. Also, when a unit test fails it is usually fairly straight forward to locate the error since only a few lines of code were exercised by the test.

Unit-testing is frequently associated with Test-Driven Development (TDD). In TDD a developer will write a test that specifies some behavior that they expect to see. Only once the test has been written will they write the production code that should pass the test. Paul Hammill [1] refers to TDD as "test twice, code once" since the development of a piece of code involves the following three stages:

- Write a test, run it and see it fail.
- Write the code that exhibits the expected behaviour.
- Run the test again and watch it succeed.

The unit tests should run within a framework that can collect all the tests that need to be run and report the results in a concise and easy to understand manner. The most popular test framework for Java is JUnit created by Kent Beck and Erich Gamma. The success of JUnit has spawned a family of test frameworks for other languages that are known collectively as xUnit. JUnit relies on the reflection API to perform its magic; unfortunately reflection isn't supported in Java ME. JMUnit is a member of the xUnit family suitable for testing Java ME applications.

## 2 Using JMUnit

Figure 1 shows the core classes of JMUnit; in practice you will almost certainly extend the `TestCase` when creating your tests. Since `TestCase` extends `MIDlet` your tests can be run directly in your IDE or on a device. As an aside, this is a significant difference from JUnit where `TestCase`s are run by a `TestRunner` class that uses introspection to determine the tests that need to be run. Since Java ME doesn't support reflection the JUnit `TestRunner` idiom is poorly suited for mobile testing.

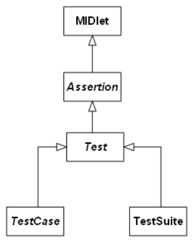


Figure 1: Core JMUnit classes

## 2.1 Assertions

The `Assertion` class provides the basic functionality of the test framework: an ability to check values and to throw an exception if an actual value differs from the expected value. The methods of the Assertion class are static.

An example of a method of the `Assertion` class is `assertEquals(int expected, int actual)`. This method will throw an `AssertionFailedException` if `expected != actual`. There are similar methods for checking whether `boolean`s, `byte`s, `short`s, `char`s, `long`s or `Object`s are equal. If you've looked at the JMUnit distribution, you'll have seen that there are two libraries; the first is for CLDC 1.0 and the second is for CLDC 1.1. The CLDC 1.1 distribution includes assertions for checking that `float`s and `double`s are equal.

The `Assertion` class also has convenience methods for checking whether a boolean result is `true` or `false` (`assertTrue` and `assertFalse`), whether an `Object` is `null` (`assertNull` and `assertNotNull`) and whether `Objects` are identical (`assertSame` and `assertNotSame`).

There are two variants of each `assert` method. The one variant takes a `String` message as a parameter and the second doesn't. If an assertion fails and the first variant is used, the message will be set in the `AssertionFailedException`.

Finally there are two `fail` static methods (one with a message parameter and one without) that can be used to throw an `AssertionFailedException`. Typically these are used when a method under test is expected to throw an `Exception` but failed to.

The JMUnit JavaDocs contain more information. A good and inexpensive source of information is Kent Beck's pocket guide to JUnit [2] which also includes concise information about writing idiomatic unit tests.

The rest of this section will look at writing tests; the samples can be found in the `examples` directory in the JMUnit distribution.

## 2.2 Writing a TestCase

To write a test case you will need to extend the `jmunit.framework.cldc1X.TestCase` class where "X" is either 0 or 1 depending on whether you are targeting CLDC 1.0 or 1.1. The `TestCase` class does not provide a default constructor so your test needs to implement a constructor. Your test will also need to implement the abstract `test(int testNumber)` method. Listing 1 shows a minimal `TestCase`.

Listing 1

```
package com.foo;

import jmunit.framework.cldc10.TestCase;

public class FooTest extends TestCase {
    public FooTest() {
        super(0, "FooTest");
    }

    public void test(int testNumber) throws Throwable {
        switch (testNumber) {
        }
    }
}
```

The `TestCase` constructor takes two arguments: the number of tests to be run and the name of the test. A more realistic `TestCase` would have more than zero tests as in listing 2 in which we have three tests.

Listing 2

```
package com.foo;

import jmunit.framework.cldc10.TestCase;

public class FooTest extends TestCase {

    public FooTest() {
        super(3, "FooTest");
    }

    public void test(int testNumber) throws Throwable {
        switch (testNumber) {
        case 0:
            testIsEmpty();
            break;

        case 1:
            testAdd(3, 5, 8);
            break;

        case 2:
            testAdd(2, 1, 3);
            break;

        default:
            break;
        }
    }

    public void testIsEmpty() {
        Foo foo = new Foo();
        assertTrue(foo.isEmpty());
        foo.add(new Object());
        assertFalse(foo.isEmpty());
    }

    public void testAdd(int addend, int augend, int expected) {
        Foo foo = new Foo();
        int actual = foo.add(addend, augend);
        assertEquals("testAdd", expected, actual);
    }
}
```

The `TestCase` in listing 2 has two test methods but the one test method is invoked twice, so the `TestCase` constructor is invoked with the argument 3 not 2. Note that this method of writing parameterized tests is different from JUnit; JUnit 3.8 doesn't support parameterized tests while JUnit 4 uses annotations to specify test parameters.

Both test methods in listing 2 need to create an instance of `Foo`. A common pattern is to create a *test fixture* to create the objects that you will be testing against; these objects can include the object under test (OUT) as well as collaborators of the OUT. In JMUnit, like JUnit, the common creation code resides in the `setUp` method of the `TestCase`. The `setUp` method is called before a test runs. Listing 3 shows a refactoring of listing 2 where we create the OUT in the `setUp` method. Obviously the benefits of using a `setUp()` are greater when you have more test methods and when the OUT collaborates with other objects that also need to be created.

Listing 3

```
package com.foo;
```

```
import jmunit.framework.cldc10.TestCase;

public class FooTest extends TestCase {

    // The OUT.
    private Foo foo;

    public void setUp() {
        this.foo = new Foo();
    }

    public FooTest() {
        super(3, "FooTest");
    }

    public void test(int testNumber) throws Throwable {
        switch (testNumber) {
        case 0:
            testIsEmpty();
            break;

        case 1:
            testAdd(3, 5, 8);
            break;

        case 2:
            testAdd(2, 1, 3);
            break;

        default:
            break;
        }
    }

    public void testIsEmpty() {
        assertTrue(foo.isEmpty());
        foo.add(new Object());
        assertFalse(foo.isEmpty());
    }

    public void testAdd(int addend, int augend, int expected) {
        int actual = foo.add(addend, augend);
        assertEquals("testAdd", expected, actual);
    }
}
```

Finally, JMUnit also supports a `tearDown` method that is called after a test has completed. This can be used to free resources allocated during a test. For example, one might ensure that all network connections opened in a test are closed or that any record stores created are deleted. Our `TestCase` doesn't need a `tearDown` method.

## 2.3 TestSuite

Java ME applications are very small by comparison with Java SE and EE applications and much of their code is difficult to test (for example, UI components), so they usually have fewer tests. However even for Java ME applications you will usually need more than one `TestCase`. Running each `TestCase` individually is tedious; the solution is to create a `TestSuite`. Referring back to figure 1, `TestSuite` also extends `MIDlet` and so can also be executed. A `TestSuite` contains instances of `Test`. When a `TestSuite` is run, all the `Test`s are run. Notice that since both `TestCase` and `TestSuite` extend `Test`, `TestSuite`s can contain `TestCase`s and other `TestSuite`s.

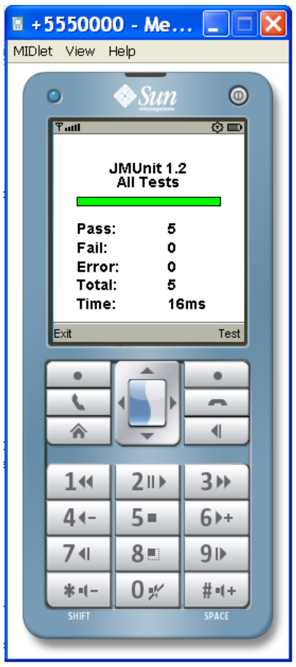
Listing 4 shows how simple it is to create a `TestSuite`: simply invoke the `add(Test test)` method inside the constructor of the `TestSuite`.

Listing 4

```
package com.foo;

import jmunit.framework.cldc10.TestSuite;

public class Suite extends TestSuite {
    public Suite() {
        super("All␣Tests");
        add(new FooTest());
        add(new BarTest());
    }
}
```



```
JMUnit 1.2
All Tests

Pass:    5
Fail:    0
Error:   0
Total:   5
Time:    16ms
```

Figure 2: JMUnit MIDlet

## 2.4 Running the Tests

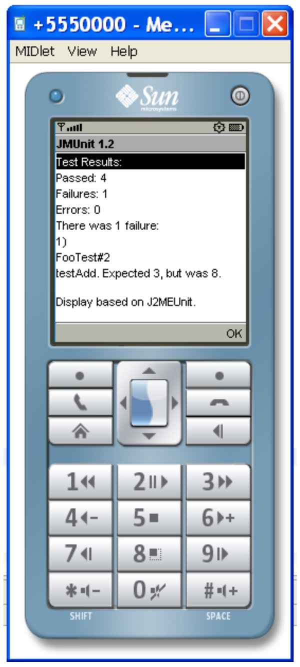
If you run the `TestSuite` of listing 4 in the WTK emulator, you'll see something similar to figure 2. The MIDlet has two buttons, "Exit" and "Test". Pressing the "Test" button runs all the tests. In this case all tests passed. If tests fail, stack traces will be dumped to the console and the test bar will change from green to red. Once the tests have finished, any errors will also be displayed on the device screen as shown in figure 3. The errors shown on the device are not as detailed as those printed to the console. Notice that an error specifies the test case name (FooTest) and the test number (2).

## 3 Testing on Real Devices

The strength of JMUnit is that it allows your unit tests to be run on real phones. You could use JUnit to write your unit tests and run them within your IDE but that wouldn't give the same degree of confidence in your software as running the tests on a real handset. Also JMUnit 1.2 introduces some performance monitoring methods (described later) to check that your code performs adequately on a real device.

### 3.1 Device Problems

As shown in figure 1, both `TestCase` and `TestSuite` extend `MIDlet`. If you instantiate a `TestSuite` you will create more than one instance of `MIDlet`: the `TestSuite` MIDlet and one or more `TestCase` MIDlets. Java ME has a security constraint that a MIDlet may only create another MIDlet in its constructor (which is why the `TestCase`s are added to the `TestSuite` in the constructor in listing 4). Unfortunately many real devices regard any attempt to create a MIDlet within a MIDlet as a `SecurityException`. In particular devices from



```
JMUnit 1.2
Test Results:
Passed: 4
Failures: 1
Errors: 0
There was 1 failure:
1)
FooTest#2
testAdd. Expected 3, but was 8.

Display based on J2MEUnit.
                                OK
```

Figure 3: JMUnit Errors

Nokia and Motorola will not run `TestSuites`, Sony-Ericssons appear to be perfectly happy with `TestSuites`.

What this means is:

**Run *only* `TestCases` on real devices, *not* `TestSuites`.**

## 3.2 Performance Monitoring

Java ME has two significant limitations compared to Java SE:

- Heap space. MSA devices should provide at least 1 MB of heap space while JTWI compliant devices need to provide 256 kB of heap space. This is substantially less than is available on the Java SE platform.

- Speed. Java ME applications run considerably more slowly on low-end devices than Java SE applications or applications running within the WTK emulator.

The speed issue can be particularly noticeable when accessing the record store; see `http://www.poqit.com/midp/bench/` for more details on how slow CRUD operations on a `RecordStore` can be; Motorola devices appear to be particularly bad so (and are, consequently, recommended for real world testing).

Figure 4 shows the `PerformanceMeasurement` interface and that it's implemented by the `TimedMeasurement` and `MemoryMeasurement` classes. The `PerformanceMeasurement` interface exposes two methods: a `startMeasurement()` method and an `endMeasurement()` method. A `TestCase` invokes `startMeasurement()` immediately *after* calling the `setUp()` method. The `endMeasurement()` method is called immediately *before* the `tearDown()` method is invoked. This ensures that the test fixture code does not affect the measurement.

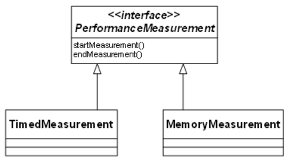


Figure 4: Performance measurement classes

A `PerformanceMeasurement` object is registered with a `TestCase` using the `addPerformanceMeasurement()` method. You can, of course, implement classes that measure performance with respect to metrics other than heap usage or speed. A `PerformanceMeasurement` class should throw an (unchecked) exception if the object under test performed unsatisfactorily.

JMUnit's performance monitoring was influenced by the JUnitPerf extensions for JUnit (`http://clarkware.com/software/JUnitPerf.html`).

### 3.2.1 Monitoring Speed

Listing 5 illustrates the use of the `TimedMeasurement` class. We run three tests each of which take about 100 milliseconds. However we expect that the second test will take 50 milliseconds or less. Consequently when we run the `TestCase` we see one failure:

```
SpeedTest#1
jmunit.framework.cldc10.AssertionFailedException: Test took too long: 109 ms.
        at jmunit.framework.cldc10.Assertion.fail(+8)
        at jmunit.framework.cldc10.Assertion.assertTrue(+8)
        at jmunit.framework.cldc10.TimedMeasurement.endMeasurement(+56)
        at jmunit.framework.cldc10.TestCase.endPerformanceMeasurements(+43)
        at jmunit.framework.cldc10.TestCase.run(+63)
        at jmunit.framework.cldc10.Test.test(+19)
        at jmunit.framework.cldc10.GuiListener$1.run(+10)
```

Listing 5

```java
package com.foo;

import jmunit.framework.cldc10.TestCase;
import jmunit.framework.cldc10.TimedMeasurement;

public class SpeedTest extends TestCase {

    public void tearDown() {
        removeAllPerformanceMeasurements();
    }

    public SpeedTest() {
        super(3, "SpeedTest");
    }

    public void testNoSpeedRequirements() throws InterruptedException {
        Thread.sleep(100);
    }

    public void testSpeedRequirements() throws InterruptedException {
        addPerformanceMeasurement(new TimedMeasurement(50, 0));
        Thread.sleep(100);
    }

    public void test(int testNum) throws Throwable {
        switch (testNum) {
        case 0:
        case 2:
            testNoSpeedRequirements();
```

```
                break;

            case 1:
                testSpeedRequirements();
                break;

            default:
                fail("No␣such␣test.");
            }
        }
    }
```

The constructor of the `TimedMeasurement` class takes two arguments: a maximum running time and a clock resolution. There is also a constructor which takes only a single argument in which case the clock resolution is assumed to be 40 milliseconds. Notice how we use the `tearDown` method to ensure that the performance measurement is removed after all tests.

Common places to add performance measurements are in the constructor of your `TestCase` or, like in listing 5, in a particular test method.

### 3.2.2 Monitoring Heap Usage

Listing 6 shows how to monitor heap usage. In this `TestCase` we add the performance monitoring in the constructor; we expect that no test will allocate more than 80 bytes on the heap. The “`false`” argument passed in the constructor indicates that the garbage collector must not be run before checking memory usage.

Listing 6

```
package com.foo;

import jmunit.framework.cldc10.MemoryMeasurement;
import jmunit.framework.cldc10.TestCase;

public class MemoryTest extends TestCase {

    public void test100Bytes() {
        byte[] b = new byte[100];
    }

    public void test40Bytes() {
        byte[] b = new byte[40];
    }

    public void test60Bytes() {
        byte[] b = new byte[60];
    }

    public MemoryTest() {
        super(3, "MemoryTest");
        addPerformanceMeasurement(new MemoryMeasurement(80, false));
    }

    public void test(int testNum) throws Throwable {
        switch (testNum) {
        case 0:
            test100Bytes();
            break;

        case 1:
            test40Bytes();
            break;

        case 2:
            test60Bytes();
            break;

        default:
            fail("No␣such␣test.");
        }
    }
}
```

Since the first test creates a 100 byte array, it is not surprising that the test fails:

```
MemoryTest#0
```

```
jmunit.framework.cldc10.AssertionFailedException: Test used too much memory: 116 bytes.
        at jmunit.framework.cldc10.Assertion.fail(+8)
        at jmunit.framework.cldc10.Assertion.assertTrue(+8)
        at jmunit.framework.cldc10.MemoryMeasurement.endMeasurement(+63)
        at jmunit.framework.cldc10.TestCase.endPerformanceMeasurements(+43)
        at jmunit.framework.cldc10.TestCase.run(+63)
        at jmunit.framework.cldc10.TestSuite.run(+30)
        at jmunit.framework.cldc10.Test.test(+19)
        at jmunit.framework.cldc10.GuiListener$1.run(+10)
```

The `MemoryMeasurement` classes uses the `Runtime` class's `freeMemory()` method to determine the amount of heap space used. Some care is needed when running memory measurement tests in the emulator. If a class under test instantiates some other class which has not yet been loaded, the emulator will load the class into memory and the amount of heap space can be reduced by several kilobytes. Since JavaME does not use `ClassLoader` classes for dynamically loading classes into RAM this appears to be less of a problem on actual devices; i.e. tests may pass on a Java ME device but fail in the emulator.

## 4 JMUnit and Ant

The Ant build tool has excellent support for JUnit since running tests is seen as integral before packaging and releasing software. Ant is widely used by Java ME developers since mobile device incompatibilities mean that several different builds of the same software may be needed; manually building all releases is both tedious and error-prone. Antenna is widely used with Java ME projects because of the preprocessor support it adds to Java and because of its support for packaging, preverifying, obfuscating and signing MIDlets.

In this section we show how JMUnit can be added to your build environment. There are two approaches:

- Running your JMUnit tests with a third-party Java ME for SE library like the microemulator (`http://microemu.org`) or ME4SE (part of the kObjects project).
- Launching the WTK emulator from within your build.

The advantage of the first solution is that the tests run without the WTK emulator (headlessly). The disadvantage is that there are certain problems when trying to test persistence classes (that use the `RecordStore`) or UI components.

Two tasks, `jmunit` and `testlistener`, are in the `jmunit_anttasks-1.2.1.jar` file in the `dist` directory of this distribution. The `build.xml` file in the `examples` directory illustrates their use. The JMUnit Ant tasks generate XML reports that can be parsed by the `junitreport` task to generate reports (e.g. an HTML report that can be emailed to recipients as part of a continuous build process).

### 4.1 The Jmunit Task

The `jmunit` task relies on a third-party library like the microemulator. It also depends on the JDOM API for generating XML and on JMUnit. There are two derivatives of the `jmunit` task; one for CLDC 1.0 and the other for CLDC 1.1.

The following line in the build script shows how to add the (CLDC 1.0) `jmunit` task to your build script (you could substitute the ME4SE library for the microemulator):

```
<taskdef name="jmunit" classname="jmunit.anttask.cldc10.Jmunit"
        classpath="../lib/microemulator.jar:../lib/jdom.jar:
../dist/jmunit_anttasks-1.2.1.jar:../dist/jmunit4cldc10-1.2.1.jar" />
```

The following lines show how to invoke the task. The task supports a proper subset of the `junit` attributes and elements. If you're familiar with the `junit` task you should be able to get the `jmunit` task working.

```
<jmunit haltonerror="false" haltonfailure="false" failureproperty="testfailure">
        <formatter type="xml" />
        <classpath>
                <path path="classes" />
        </classpath>
        <test name="com.foo.BarTest" todir="test_results_jmunit" />
        <test name="com.foo.FooTest" todir="test_results_jmunit" />
</jmunit>
```

The above script runs the `BarTest` and `FooTest` `TestCase`s and writes the results to the `test_results_jmunit` directory.

The only type currently supported by the `formatter` is XML; specifying any other type will cause a `BuildException` to be thrown.

The XML report generated can be parsed by the `junitreport` task as specified in the following lines:

```
<junitreport todir="test_results_jmunit">
        <fileset dir="test_results_jmunit">
                <include name="TEST-*.xml" />
        </fileset>
        <report format="noframes" todir="test_results_jmunit" />
</junitreport>
```

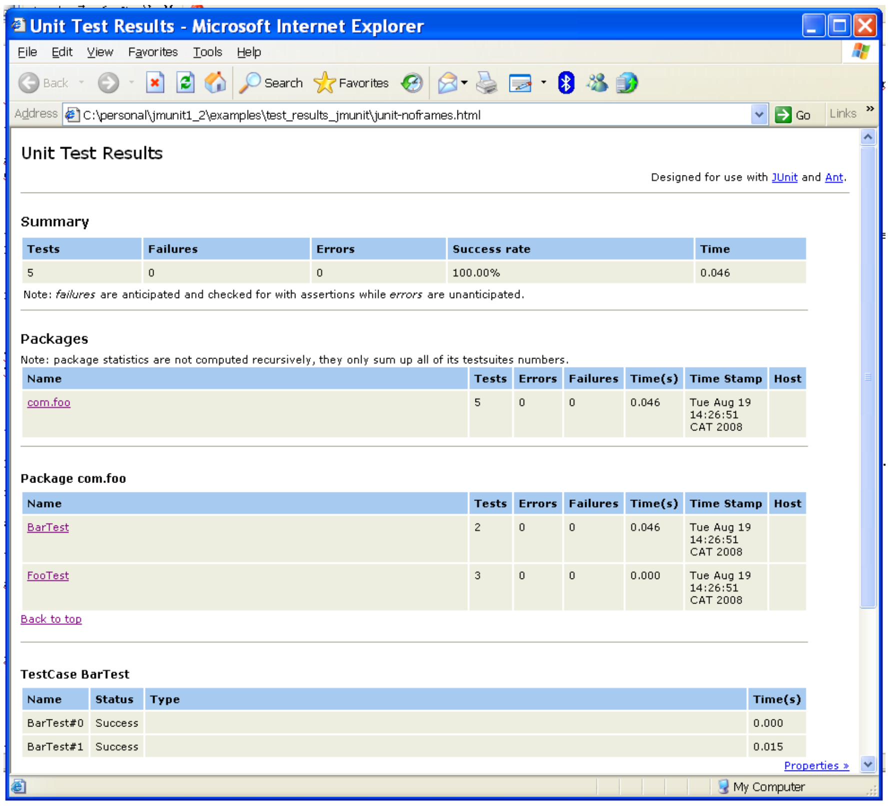
Figure 5 shows the test results. The results are attractively presented (if you ignore the fact that the browser is Internet Explorer 6).

## 4.2 The TestListener Task

The `jmunit` task is fairly straightforward since it is similar to the `junit` task. Unfortunately the microemulator tends to throw `NullPointerException`s when you invoke a `RecordStore` if a MIDlet isn't running. Similar problems exist for some UI components. Consequently tests may fail with a `NullPointerException` error. An alternative to using the `jmunit` task is to use the WTK emulator and the `testlistener` task. The `testlistener` task was originally part of the Hammock J2ME mock object framework. Using the `testlistener` task is more complicated than using the `jmunit` task and involves the following steps:

- Create an instance of the `TestRunner` class that specifies the test to be run.

- Package the test into a MIDlet (e.g. using Antenna).


Unit Test Results - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address C:\personal\jmunit1_2\examples\test_results_jmunit\junit-noframes.html
Go Links
Properties »
My Computer

Unit Test Results

Designed for use with JUnit and Ant.

Summary

| Tests | Failures | Errors | Success rate | Time |
|---|---|---|---|---|
| 5 | 0 | 0 | 100.00% | 0.046 |

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

Packages

Note: package statistics are not computed recursively, they only sum up all of its testsuites numbers.

| Name | Tests | Errors | Failures | Time(s) | Time Stamp | Host |
|---|---|---|---|---|---|---|
| com.foo | 5 | 0 | 0 | 0.046 | Tue Aug 19 14:26:51 CAT 2008 | |

Package com.foo

| Name | Tests | Errors | Failures | Time(s) | Time Stamp | Host |
|---|---|---|---|---|---|---|
| BarTest | 2 | 0 | 0 | 0.046 | Tue Aug 19 14:26:51 CAT 2008 | |
| FooTest | 3 | 0 | 0 | 0.000 | Tue Aug 19 14:26:51 CAT 2008 | |

Back to top

TestCase BarTest

| Name | Status | Type | Time(s) |
|---|---|---|---|
| BarTest#0 | Success | | 0.000 |
| BarTest#1 | Success | | 0.015 |

Figure 5: Junitreport Test Results

- Instantiate the `testlistener` task.
- Run the MIDlet inside the WTK emulator.

Figure 1 showed that `TestCase` and `TestSuite` both extend the `Test` class. In fact there is a third class, `TestRunner`, that also extends `Test`. Both `TestCase` and `TestSuite` require that a user presses the “Test” button to start running tests; the `TestRunner` class runs tests immediately when launched and closes when the tests have finished running. Additionally, the `TestRunner` class writes the test results to standard output as XML. Listing 7 shows how to extend the `TestRunner` class (the magic constant 3000 indicates that the test will wait 3 seconds before closing down the MIDlet).

Listing 7

```
package com.foo;

import jmunit.framework.cldc10.Test;
import jmunit.framework.cldc10.TestRunner;

public class SuiteRunner extends TestRunner {

    private Test nestedTest;

    public SuiteRunner() {
        super(3000);
        this.nestedTest = new Suite();
    }

    protected Test getNestedTest() {
        return this.nestedTest;
    }

}
```

The `testlistener` task is added with the following `taskdef`

```
<taskdef name="testlistener" classname="jmunit.anttask.TestListener"
         classpath="../dist/jmunit_anttasks-1.2.1.jar" />
```

The following lines use Antenna to package the `TestRunner` into a MIDlet

```
<wtkjad jadfile="dist/alltests.jad" jarfile="dist/alltests.jar"
        name="AllTests" vendor="CAM" version="1.0.0">
        <midlet name="AllTests" class="com.foo.SuiteRunner" />
</wtkjad>
<wtkpackage jarfile="dist/alltests.jar" jadfile="dist/alltests.jad"
        preverify="true" obfuscate="false">
        <libclasspath>
                <path path="../dist/jmunit4cldc10-1.2.1.jar" />
        </libclasspath>
        <fileset dir="classes"/>
</wtkpackage>
```

The lines below start and stop the `testlistener` task and run the tests within the emulator. Note that you must have set the `wtk.home` property in the `ant-properties.xml` file to point to your installation of the wireless toolkit. There is also an assumption that you're using the Windows executable of the WTK.

```
<testlistener haltonerror="false" haltonfailure="false" failureproperty="testfailure"
        todir="test_results_testlistener" run="true" />
<exec executable="${wtk.home}/bin/emulator.exe">
        <arg line="-Xdescriptor:dist/alltests.jad com.foo.SuiteRunner" />
</exec>
<testlistener run="false" />
```

The `testlistener` task monitors the text written by the `TestRunner` to standard output and directs the text to a file 'TEST-AllTests.xml'.

## 5 Acknowledgements

JMUnit relies on several open source projects and the distribution includes binaries from several projects. The list below includes where the source code may be obtained and the license governing the software (all licenses may be found in the `licenses` directory):

- JDOM, `http://jdom.org/`; Apache license with the acknowledgment clause removed: `jdom-license.txt`.
- The microemulator, `http://microemu.org/`; LGPL license, version 2.1.
- Antenna, `http://sourceforge.net/projects/antenna`; LGPL license, version 3.
- j2meunit, `http://sourceforge.net/projects/j2meunit`; common public license, version 2.0.
- Hammock, `http://sourceforge.net/projects/hammockmocks`; Apache license, version 2.0.

## References

[1] Paul Hammill, *Unit Testing Frameworks*, O'Reilly, Sebastopol, California, 2004.

[2] Kent Beck, *JUnit Pocket Guide*, O'Reilly, Sebastopol, California, 2004.