Technical Manual
for TestScript-Parser Version 1.0.0

Klaus Hammermüller
klaus@ifs.tuwien.ac.at

September 1, 1999

Abstract

This paper is the technical documentation of the *"TestScript-Parser"* generating unit-test-statements for Java from an XML-file. This tool is used in the *Asgaard-framework* built on pure Java components. The focus is to have an automated test-environment which guarantee the consistence of the dayly built.

Contents

1 Overview

1.1 What is a Test-Script?

A *Test-script* is a collection of *test-statements*. Each statement is a method-call of an Object or Class. There are tree aspects monitored:

- Is an Exception thrown, the test of this method fails, the Exception is written to the display and a log-file;
- The result of an method is stored and can be accessed due future test-statements and for examination by the test-environment;
- The execution-time of each native-method call is recorded which may help meeting time-constraints.

1.1.1 Why testing?

Our experience about software without testing is one sentence: "It does not work." But also testing is not enough. To find an bug nearby it's creation you have to run a whole set of test as often as possible and therefore you need automated support. The aim is to run the tests each day and have a guaranteed running software every evening.

The Idea is from Ronald E Jeffries, and his white paper about "Extreme-Programming" 1997, jeffries@ic.net http://ic.net/ jeffries, cite:

¿¿ *Unit Tests*

Each class must have unit tests. Every classs unit tests must score 100%. We use Kent Becks public domain testing framework, augmented with a GUI that runs all the tests and shows the percent correct. At this writing, there are over 1300 unit tests, and they all run at 100 percent.

We recommend that unit tests be written before the class is written. This is a good way to focus attention on what the class is really about. In any case, a class isnt done until its unit tests are in the test suite.

When classes are released, all unit tests must be running at 100 percent. All. That is, if the changes you make break the unit tests for some other class, the problem must be resolved before you release.

On the contrary, what if Im not the one who broke the test, or what if someone is using my class incorrectly?

- Since the unit tests ran at 100 percent before you released, you did break the tests.
- If someone is using your class incorrectly, why did the tests run at 100 percent when they released? Something you changed has caused the problem, because you know the tests ran at 100% before you started to release your code.
- Even if the class is used incorrectly, you need to resolve the problem before you release. Partner with the user of your class if need be.

To emphasize the point: all the unit tests must run at 100 percent, all the time.¡¡

1.1.2 Why using a script?

- Without separating each method-call the detailed monitoring and recording would not be possible;
- Nested Exceptions are impossible and side-effects minimized;
- The measure of execution-time needs an defined test-environment like this one;
- In the end the script is easy to create and maintain and does not slow down compilation times during coding.

1.2 Limitations

These test-scripts can not:

- Implement functional tests. A functional test defines a set of input data plus a corresponding valid set output data. This test-scripts define only a unit-level test;
- Prevent from unusable test scenarios and -cases.

A special case of functional tests are implemented by Robert Kosara to do regression tests on dynamic web-pages. These are a separate package. Some classic test-environment for "input-file : call : output-file" has not been implemented yet.

2 Running Test-Scripts

To run a test-script you have to run the following Steps:

1. Install the *TestScript-Parser* (once);
2. Write and maintain a test-script with an XML-editor;
3. Running *TestScript-Parser* to create Java-code;
4. Compile the test-script-class with javac;
5. Run the test-script-class.

The last three steps may be concated in some makefile. The generated test-script-class needs only the packages which are with the `test.jar` archive. For more information about the details of this code have a look at `asgaard.test.Test.java` and it's javadoc - documentation.

3 Install and Run the TestScript-Parser

3.1 Distribution

The Distribution is under GNU public license Version 2 `http://www.gnu.org/copyleft/gpl.htm`, there is absolutely NO WARRANTY on this software.

A complete distribution contains on one compressed zip-archive `test.zip` containing the following parts:

- `test.jar` containing the packages `asgaard.utils.log` `asgaard.utils.test` `asgaard.utils.encode` `asgaard.utils.mail`;
- `xml4j.jar` IBM's XML-parser;
- SimpleText flatfile JDBC driver (included in `test.jar`);
- all sources and documentation to the listed packages above.

To run this classes you need a JDK 1.1.6 or later.

3.1.1 IBM's XML-parser

The classes of this parser which is free for not commercial products without any warranty are included in the `xml4j.jar` archive. Sources, documentation and license is freely available under `http://www.alphaworks.ibm.com/tech/xml4j`.

3.1.2 SimpleText

This is an ASCII flatfile JDBC driver which is used to generate the log-file.

```
Copyright:   (C) 1997 THOUGHT Inc.  All rights reserved.
Copyright:   (C) 1996 Karl Moss.  All rights reserved.
             You may study, use, modify and distribute this example
             for any purpose, provided that this copyright notice
             appears in all copies.  This example is provided WITHOUT
             WARRANTY either expressed or implied.
```

3.1.3 Remark

The mail-capabilities to inform automatically about failures of test-runs have been deactivated and would insist the packages `mail.jar` and `activaion.jar` from SUN's free 1.1 Mail-Distribution.

3.2 Installation

All you need to run the *TestScript-Parser* is with the zip-archive, especially the two `.jar` archives.

3.3 Command-Line Parameter

To start the *TestScript-Parser* type

```
java -cp test.jar;xml4j.jar asgaard.utils.test.TestScriptParser <xml-file>
```

Command line parameter is only the `<xml-file>` which defines an test-script.
Remark: To ensure your XML-file is valid (if you don't use an validating editor) you can use a tree-viewer of your test-script which is with the xml-parser:

```
java -cp swing.jar;xml4j.jar;xml4jSamples.jar ui.TreeViewer <xml-file>
```

3.4 Test the TestScript-Parser

To test the *TestScript-Parser* type

```
java -cp test.jar;xml4j.jar asgaard.utils.test.TestScriptParser
asgaard/utils/test/TestScriptParser.xml
```

All instructions and information about the test are described in the head of the `TestScriptParser.xml` file.

3.5 Output

The Output of the *TestScript-Parser* is written to `System.out`, errors are written to `System.err`, all details are documented in the `.testlog/log.dbf` file. More details about the logging-mechanism is with the Documentation of the `asgaard.utils.log` package (see `log.html`).

```
TEST starting at Wed Sep 01 13:33:35 CEST 1999
JDBC-Driver SimpleText (mod.) Copyright (c) 1996 Karl Moss, 1997 THOUGHT Inc.
New Logfile created: .testlog\log.sdf
TEST LOG: .testlog MAILTO: klaus@ifs.tuwien.ac.at
TEST#1 TST TEST:newString#1 SUCCEED [0ms] (java.lang.String.)
TEST#2 TST TEST:newString#2 SUCCEED [0ms] (java.lang.String.)
TEST#3 CMP TEST:compareString 1/2 SUCCEED (EQUAL==EQUAL)
TEST LOG ERROR: statement not recorded.
TEST#4 TST TEST:compStr#1 SUCCEED [0ms] (java.lang.String.compareTo)
```

```
TEST#5 TST TEST:newVector#1 SUCCEED [0ms] (java.util.Vector.)
TEST#6 TST TEST:addVector#1 SUCCEED [0ms] (java.util.Vector.addElement)
TEST#7 TST TEST:addVector#2 SUCCEED [0ms] (java.util.Vector.addElement)
TEST#8 TST TEST:getVector#1 SUCCEED [0ms] (java.util.Vector.firstElement)
TEST#9 CMP TEST:compareString 1/1 SUCCEED (EQUAL==EQUAL)
TEST#10 ARR TEST:getArray#1 SUCCEED
TEST#11 CMP TEST:compareItem SUCCEED (B==B)
TEST SUCCEED: TEST 0/11 tests failed
```

This is the report of the successful self-test of the `Test` class itself.

4 Writing Test-Scripts

To write an test-script you have to use XML. You can edit an XML-file wich every ASCII-editor or you use one of the free XML-editors (have a look at `http://www.ibm.com/developer/xml/` where is a large commented list of available software).

For all the details you may have a look in the commented `asgaard/utils/test/TestScript.dtd` document definition file. For a quick start, here are some examples:

4.1 Script-Heading

All the Files have to start the same way:

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no"?>
<!DOCTYPE TestScript [<!ENTITY % UnitTest SYSTEM "asgaard/utils/test/UnitTest.dtd">
%UnitTest;
<!ELEMENT TestScript (UnitTest)> ]>
```

All necessary definitions are embedded into the document this way (and may be mixed with other definitions too, but this is out of our focus.

Remark: Please replace `asgaard/utils/test/UnitTest.dtd` by the path to the dtd you are using.

4.2 Script-Body with one Test-Statement

After the heading there smalles possible test-script looks like

```
<TestScript>
  <UnitTest package="some.package" name="SimpleScript">
      <UnitTestScript name="SIMPLETESTSCRIPT">
          <construct test="newString1" class="java.lang.String">
              <param>
                  <object class="java.lang.String">"Just some String"</object>
              </param>
          </construct>
      </UnitTestScript>
  </UnitTest>
</TestScript>
```

What happens here? A Test-Statement with the name `"newString1"` which carries `"Just some String"` as content is created using the `String(java.lang.String)` constructor. This test-statement is placed in a test-script named `SIMPLETESTSCRIPT` which is implemented in the class `SimpleScript` in the `some.package` package.

4.3 Some Simple Examples

For this Example you need a simple Class "Person":

```
public class Person {
    public void Person() {
        super();
    }
    public void setName(String name) {
        ...
    }
    public String getName() {
        ...
    }
    public void flush() {
        ...
    }
```

Now we want to test these three methods:

```
<TestScript>
  <UnitTest package="some.package" name="SimpleScript2">
      <UnitTestScript name="SIMPLETESTSCRIPT2">
          <construct test="newPerson1" class="asgaard.lang.Person">
          </construct>
          <method test="modPerson1" name="setName">
              <instance>
                  <reference test="newPerson1" cast="asgaard.lang.Person"/>
              </instance>
              <param>
                  <object class="java.lang.String">"Birgit Auer"</object>
              </param>
          </method>
          <method test="savPerson1" name="flush">
              <instance>
                  <reference test="newPerson1" cast="asgaard.lang.Person"/>
              </instance>
          </method>
```

First we create a `Person` instance with an parameter-less constructor. Then we invoke the `setName` method and "save" the Changes with the `flush` method. In those both method-calls we do not create new instances of `Person` but use the one we have created in the first test-statement by referencing it using the `reference` tag. To set the parameter of the `setName` method we instanciate an object as parameter-value using the `object` tag.

Of cause, the `setName` can only work if the constructor did well, otherwise we would get an `NullPointerException`. But to evaluate if the `setName` has worked well we need another mechanism:

```
          <method test="getName1" name="getName">
              <instance>
                  <reference test="newPerson1" cast="asgaard.lang.Person"/>
              </instance>
          </method>
          <compare test="compareName1">
                  <reference test="getName1" cast="java.lang.String"/>
```

```
                <object class="java.lang.String">"Birgit Auer"</object>
            </compare>
        </UnitTestScript>
    </UnitTest>
</TestScript>
```

First we invoke the `getName` method and compare the result of this test-statement with the name `getName1` to an String carrying the name which should be hold by the `newPerson1` object.

Remark: The `object` tag holds *native Java-code* so you need to add `"` to Strings.

4.4 Advanced Examples

At some point you may come to the point where you get an array as result of an method-call an want to get e.g. th first item:

```
<construct test="newArray1" class="java.lang.Object[]">
    <param>
        <object class="java.lang.Boolean">false</object>
        <object class="java.lang.Boolean">true</object>
    </param>
</construct>
<item test="firstItem1" index="0">
    <reference test="newArray1" cast="java.lang.Object[]"/>
</item>
```

This dummy-example creates an array with two entries in the first test-statement. The `item` tag in the second statement returns the Object with the requested index, where it is accessible for former examination.

Remark: The capabilities of the `constructor` tag are restricted to create an `Object[]` array, because to invoke an Array - constructor is not straight forward coding.

To enhance the Java-code of the generated test-class use the `javacode` tag:

```
<javaCode>
    <main><![CDATA[
        System.out.println("Hello, this is skid.Test, checking the asgaard.skid package.");
        System.out.println("Usage: java asgaard.skid.TestSkid <host>");
        // ...
        // Some remote connection is established here
        // ...
        app.testIt(app);
    ]]></main>
</javaCode>
```

For more information about the details of this code have a look at the sources of the `asgaard.test` package and it's javadoc - documentation. And, of course hava a look at the `TestScript.dtd`.