A Tour of XMLUnit

What is XMLUnit?

XMLUnit enables JUnit-style assertions to be made about the content and structure of XML[1]. It is an open source project hosted at http://xmlunit.sourceforge.net that grew out of a need to test a system that generated and received custom XML messages. The problem that we faced was how to verify that the system generated the correct message from a known set of inputs. Obviously we could use a DTD or a schema to validate the message output, but this approach wouldn't allow us to distinguish between valid XML with correct content (e.g. element <foo>bar</foo>) and valid XML with incorrect content (e.g. element <foo>baz</foo>). What we really wanted was an *assertXMLEquals()* method, so we could compare the message that we expected the system to generate and the message that the system actually generated. And that was the beginning of XMLUnit.

Quick tour

XMLUnit provides a single JUnit extension class, *XMLTestCase*, and a set of supporting classes that allow assertions to be made about:

- The differences between two pieces of XML (via *Diff* and *DetailedDiff* classes)
- The validity of a piece of XML (via *Validator* class)
- The outcome of transforming a piece of XML using XSLT (via *Transform* class)
- The evaluation of an XPath expression on a piece of XML (via *SimpleXpathEngine* class)
- Individual nodes in a piece of XML that are exposed by DOM Traversal (via *NodeTest* class)

XMLUnit can also treat HTML content, even badly-formed HTML, as valid XML to allow these assertions to be made about web pages (via the *HTMLDocumentBuilder* class).

Glossary

As with many projects some words in XMLUnit have particular meanings so here is a quick overview. A *piece* of XML is a DOM Document, a String containing marked-up content, or a Source or Reader that allows access to marked-up content within some resource. XMLUnit compares the expected *control* XML to some actual *test* XML. The comparison can reveal that two pieces of XML are *identical*, *similar* or *different*. The unit of measurement used by the comparison is a *difference*, and *differences* can be either *recoverable* or *unrecoverable*. Two pieces of XML are *identical* if there are no *differences* between them, *similar* if there are only *recoverable differences* between them, and *different* if there are any *unrecoverable differences* between them.

Configuring XMLUnit

There are many Java XML parsers available, and XMLUnit should work with any JAXP compliant parser library, such as Xerces from the Apache Jakarta project. To use the XSL and XPath features of XMLUnit a Trax compliant transformation engine is required, such as Xalan, from the Apache Jakarta project. To configure XMLUnit to use your parser and transformation engine set three System properties before any tests are run, e.g.

```
System.setProperty("javax.xml.parsers.DocumentBuilderFactory",
"org.apache.xerces.jaxp.DocumentBuilderFactoryImpl");
System.setProperty("javax.xml.parsers.SAXParserFactory",
"org.apache.xerces.jaxp.SAXParserFactoryImpl");
System.setProperty("javax.xml.transform.TransformerFactory",
"org.apache.xalan.processor.TransformerFactoryImpl");
```

Alternatively there are static methods on the XMLUnit class that can be called directly. The advantage of this approach is that you can specify a different parser class for control and test XML and change the

[1] For more information about JUnit see http://www.junit.org.

current parser class at any time in your tests, should you need to make assertions about the compatibility of different parsers.

```
XMLUnit.setControlParser("org.apache.xerces.jaxp.DocumentBuilderFactoryI
mpl");
XMLUnit.setTestParser("org.apache.xerces.jaxp.DocumentBuilderFactoryImpl
");
XMLUnit.setSAXParserFactory("org.apache.xerces.jaxp.SAXParserFactoryImpl
");
XMLUnit.setTransformerFactory("org.apache.xalan.processor.TransformerFac
toryImpl");
```

Writing XML comparison tests

Let's say we have two pieces of XML that we wish to compare and assert that they are equal. We could write a simple test class like this:

```
public class MyXMLTestCase extends XMLTestCase {
    public MyXMLTestCase(String name) {
        super(name);
    }
    public void testForEquality() throws Exception {
        String myControlXML = "<msg><uuid>0x00435A8C</uuid></msg>";
        String myTestXML = "<msg><localId>2376</localId></msg>";
        assertXMLEqual("Comparing test xml to control xml",
            myControlXML, myTestXML);
    }
}
```

The *assertXMLEqual* test will pass if the control and test XML are either similar or identical. Obviously in this case the pieces of XML are different and the test will fail. The failure message indicates both what the difference is and the Xpath locations of the nodes that were being compared:

```
Comparing test xml to control xml
[different] Expected element tag name 'uuid' but was 'localId' -
comparing <uuid...> at /msg[1]/uuid[1] to <localId...> at
/msg[1]/localId[1]
```

When comparing pieces of XML, the *XMLTestCase* actually creates an instance of the *Diff* class. The *Diff* class stores the result of an XML comparison and makes it available through the methods *similar()* and *identical()*. The *assertXMLEquals()* method tests the value of *Diff.similar()* and the *assertXMLIdentical()* method tests the value of *Diff.identical()*.
It is easy to create a *Diff* instance directly without using the *XMLTestCase* class as below:

```
    public void testXMLIdentical()throws Exception {
        String myControlXML =
            "<struct><int>3</int><boolean>false</boolean></struct>";
        String myTestXML =
            "<struct><boolean>false</boolean><int>3</int></struct>";
        Diff myDiff = new Diff(myControlXML, myTestXML);
        assertTrue("XML similar " + myDiff.toString(),
            myDiff.similar());
        assertTrue("XML identical " + myDiff.toString(),
            myDiff.identical());
    }
```

This test fails as two pieces of XML are similar but not identical if their nodes occur in a different sequence. The failure message reported by JUnit from the call to *myDiff.toString()* looks like this:

```
[not identical] Expected sequence of child nodes '0' but was '1' -
comparing <int...> at /struct[1]/int[1] to <int...> at /struct[1]/int[1]
```

For efficiency reasons a *Diff* stops the comparison process as soon as the first difference is found. To get all the differences between two pieces of XML an instance of the *DetailedDiff* class, a subclass of *Diff*, is required. Note that a *DetailedDiff* is constructed using an existing *Diff* instance.

Consider this test that uses a *DetailedDiff*:

```
    public void testAllDifferences() throws Exception {
        String myControlXML = "<news><item id=\"1\">War</item>"
        + "<item id=\"2\">Plague</item>"
        + "<item id=\"3\">Famine</item></news>";
        String myTestXML = "<news><item id=\"1\">Peace</item>"
        + "<item id=\"2\">Health</item>"
        + "<item id=\"3\">Plenty</item></news>";
        DetailedDiff myDiff = new DetailedDiff(
            new Diff(myControlXML, myTestXML));
        List allDifferences = myDiff.getAllDifferences();
        assertEquals(myDiff.toString(), 2, allDifferences.size());
    }
```

This test fails with the message below as each of the 3 news items differs between the control and test XML:

```
[different] Expected text value 'War' but was 'Peace' - comparing <item
...>War</item> at /news[1]/item[1]/text()[1] to <item ...>Peace</item>
at /news[1]/item[1]/text()[1]
[different] Expected text value 'Plague' but was 'Health' - comparing
<item ...>Plague</item> at /news[1]/item[2]/text()[1] to <item
...>Health</item> at /news[1]/item[2]/text()[1]
[different] Expected text value 'Famine' but was 'Plenty' - comparing
<item ...>Famine</item> at /news[1]/item[3]/text()[1] to <item
...>Plenty</item> at /news[1]/item[3]/text()[1]
expected <2> but was <3>
```

The List returned from the *getAllDifferences()* method contains *Difference* instances. These instances describe both the type[2] of difference found between a control node and test node and the *NodeDetail* of those nodes (including the XPath location of each node). *Difference* instances are passed at runtime in notification events to a registered *DifferenceListener*, an interface whose default implementation is provided by the *Diff* class.

However it is possible to override this default behaviour by implementing the interface in your own class. The *IgnoreTextAndAttributeValuesDifferenceListener* class is an example of how to implement a custom *DifferenceListener*. It allows an XML comparison to be made that ignores differences in the values of text and attribute nodes, for example when comparing a skeleton or outline piece of XML to some generated XML.
The following test illustrates the use of a custom *DifferenceListener*:

```
    public void testCompareToSkeletonXML() throws Exception {
      String myControlXML = "<location><street-address>22 any
street</street-address><postcode>XY00 99Z</postcode></location>";
      String myTestXML = "<location><street-address>20 east
cheap</street-address><postcode>EC3M 1EB</postcode></location>";
      DifferenceListener myDifferenceListener =
        new IgnoreTextAndAttributeValuesDifferenceListener();
      Diff myDiff = new Diff(myControlXML, myTestXML);
      myDiff.overrideDifferenceListener(myDifferenceListener);
      assertTrue("test XML matches control skeleton XML",
        myDiff.similar());
    }
```

The *DifferenceEngine* class generates the events that are passed to a *DifferenceListener* implementation as two pieces of XML are compared. Using recursion it navigates through the nodes in the control XML DOM, and determines which node in the test XML DOM qualifies for comparison to the current control

[2] A full set of prototype *Difference* instances - one for each type of difference - is defined using final static fields in the *DifferenceConstants* class.

node. The qualifying test node will match the control node's node type, as well as the node name and namespace (if defined for the control node).

However when the control node is an Element, it is less straightforward to determine which test Element qualifies for comparison as the parent node may contain repeated child Elements with the same name and namespace. So for Element nodes, an instance of the *ElementQualifier* interface is used determine whether a given test Element node qualifies for comparison with a control Element node. This separates the decision about whether two Elements should be compared from the decision about whether those two Elements are considered similar. By default an *ElementNameQualifier* class is used that compares the n^{th} child <*abc*> test element to the n^{th} child <*abc*> control element, i.e. the sequence of the child elements in the test XML is important. However this default behaviour can be overridden using an *ElementNameAndTextQualifier* or *ElementNameAndAttributesQualifier*.

The test below demonstrates the use of a custom *ElementQualifier*:

```
public void testRepeatedChildElements() throws Exception {
      String myControlXML = "<suite>"
        + "<test status=\"pass\">FirstTestCase</test>"
        + "<test status=\"pass\">SecondTestCase</test></suite>";
      String myTestXML = "<suite>"
        + "<test status=\"pass\">SecondTestCase</test>"
        + "<test status=\"pass\">FirstTestCase</test></suite>";

      assertXMLNotEqual("Repeated child elements in different sequence
order are not equal by default",
            myControlXML, myTestXML);

      Diff myDiff = new Diff(myControlXML, myTestXML);
      myDiff.overrideElementQualifier(
          new ElementNameAndTextQualifier());
      assertXMLEqual("But they are equal when an ElementQualifier
controls which test element is compared with each control element",
            myDiff, true);
    }
```

Comparing XML Transformations

XMLUnit can test XSL transformations at a high level using the *Transform* class that wraps an *javax.xml.transform.Transformer* instance. Knowing the input XML, input stylesheet and expected output XML we can assert that the output of the transformation matches the expected output as follows:

```
    public void testXSLTransformation() throws Exception {
      String myInputXML = "...";
      File myStylesheetFile = new File("...");
      Transform myTransform =
          new Transform(myInputXML, myStylesheetFile);
      String myExpectedOutputXML = "...";
      Diff myDiff = new Diff(myExpectedOutputXML, myTransform);
      assertTrue("XSL transformation worked as expected",
myDiff.similar());
    }
```

The *getResultString()* and *getResultDocument()* methods of the *Transform* class can be used to access the result of the XSL transformation programmatically if required, for example as below:

```
public void testAnotherXSLTransformation() throws Exception {
      File myInputXMLFile = new File("...");
      File myStylesheetFile = new File("...");
      Transform myTransform = new Transform(
          new StreamSource(myInputXMLFile),
          new StreamSource(myStylesheetFile));
```

```
        Document myExpectedOutputXML =
            XMLUnit.buildDocument(XMLUnit.getControlParser(),
            new FileReader("..."));
        Diff myDiff = new Diff(myExpectedOutputXML,
            myTransform.getResultDocument());
        assertTrue("XSL transformation worked as expected",
            myDiff.similar());
    }
```

Validation Tests

XML parsers that validate a piece of XML against a DTD are common, however they rely on a DTD reference being present in the XML, and they can only validate against a single DTD. When writing a system that exchanges XML messages with third parties there are times when you would like to validate the XML against a DTD that is not available to the recipient of the message and so cannot be referenced in the message itself. XMLUnit provides a *Validator* class for this purpose.

```
    public void testValidation() throws Exception {
        XMLUnit.getTestDocumentBuilderFactory().setValidating(true);
        // As the document is parsed it is validated against its
referenced DTD
        Document myTestDocument = XMLUnit.buildTestDocument("...");
        String mySystemId = "...";
        String myDTDUrl = new File("...").toURL().toExternalForm();
        Validator myValidator = new Validator(
            myTestDocument, mySystemId, myDTDUrl);
        assertTrue("test document validates against unreferenced DTD",
            myValidator.isValid());
    }
```

Xpath Tests

One of the strengths of XML is the ability to programmatically extract specific parts of a document using XPath expressions. The XMLTestCase class offers a number of XPath related assertion methods, as demonstrated in this test:

```
    public void testXPaths() throws Exception {
        String mySolarSystemXML = "<solar-system>"
            + "<planet name='Earth' position='3' supportsLife='yes'/>"
            + "<planet name='Venus' position='4'/></solar-system>";
        assertXpathExists("//planet[@name='Earth']", mySolarSystemXML);
        assertNotXpathExists("//star[@name='alpha centauri']",
            mySolarSystemXML);
        assertXpathsEqual("//planet[@name='Earth']", "
            "//planet[@position='3']", mySolarSystemXML);
        assertXpathsNotEqual("//planet[@name='Venus']", "
            "//planet[@supportsLife='yes']", mySolarSystemXML);
    }
```

When an XPath expression is evaluated against a piece of XML a *NodeList* is created that contains the matching Nodes. The methods in the previous test – *assertXPathExists*, *assertNotXPathExists*, *assertXPathsEqual*, and *assertXPathsNotEqual* – use these *NodeLists*. However, the contents of a *NodeList* can be flattened (or String-ified) to a single value, and XMLUnit also allows assertions to be made about this single value, as in this test[3]:

```
    public void testXPathValues() throws Exception {
        String myJavaFlavours = "<java-flavours>"
            + "<jvm current='some platforms'>1.1.x</jvm>"
            + "<jvm current='no'>1.2.x</jvm>"
            + "<jvm current='yes'>1.3.x</jvm>"
```

[3] Each of the *assertXpath…()* methods uses the *SimpleXpathEngine* class to evaluate an Xpath expression.

```
          + "<jvm current='yes' latest='yes'>1.4.x</jvm></java-
flavours>";
      assertXpathEvaluatesTo("2", "count(//jvm[@current='yes'])", "
          myJavaFlavours);
      assertXpathValuesEqual("//jvm[4]/@latest", "//jvm[4]/@current", "
          myJavaFlavours);
      assertXpathValuesNotEqual("//jvm[2]/@current", "
          "//jvm[3]/@current", myJavaFlavours);
    }
```

Xpaths are especially useful where a document is made up largely of known, unchanging content with only a small amount of changing content created by the system. One of the main areas where constant 'boilerplate' markup is combined with system generated markup is of course in web applications. The power of XPath expressions can make testing web page output quite trivial, and XMLUnit supplies a means of converting even very badly formed HTML into XML to aid this approach to testing.

The *HTMLDocumentBuilder* class uses the Swing HTML parser to convert marked-up content to Sax events. The *TolerantSaxDocumentBuilder* class handles the Sax events to build up a DOM document in a tolerant fashion i.e. without mandating that opened elements are closed. (In a purely XML world this class would have no purpose as there are plenty of Sax event handlers that can build DOM documents from well formed content). The test below illustrates how the use of these classes:

```
    public void testXpathsInHTML() throws Exception {
      String someBadlyFormedHTML = "<html><title>Ugh</title>"
          + "<body><h1>Heading<ul>"
          + "<li id='1'>Item One<li id='2'>Item Two";
      TolerantSaxDocumentBuilder tolerantSaxDocumentBuilder = "
          new TolerantSaxDocumentBuilder(XMLUnit.getTestParser());
      HTMLDocumentBuilder htmlDocumentBuilder = "
          new HTMLDocumentBuilder(tolerantSaxDocumentBuilder);
      Document wellFormedDocument = "
          htmlDocumentBuilder.parse(someBadlyFormedHTML);
      assertXpathEvaluatesTo("Item One", "/html/body//li[@id='1']", "
          wellFormedDocument);
    }
```

One of the key points about using Xpaths with HTML content is that extracting values in tests requires the values to be identifiable. (This is just another way of saying that testing HTML is easier when it is written to be testable.) In the previous example *id* attributes were used to identify the list item values that needed to be testable, however *class* attributes or span and *div* tags can also be used to identify specific content for testing.

Testing by Tree Walking

The DOM specification allows a *Document* to optionally implement the *DocumentTraversal* interface. This interface allows an application to iterate over the Nodes contained in a Document, or to 'walk the DOM tree'. The XMLUnit *NodeTest* class and *NodeTester* interface make use of *DocumentTraversal* to expose individual Nodes in tests: the former handles the mechanics of iteration, and the latter allows custom test strategies to be implemented. A sample test strategy is supplied by the *CountingNodeTester* class that counts the nodes presented to it and compares the actual count to an expected count. The test below illustrates its use:

```
    public void testCountingNodeTester() throws Exception {
      String testXML = "<fibonacci><val>1</val><val>2</val><val>3</val>"
            + "<val>5</val><val>9</val></fibonacci>";
      CountingNodeTester countingNodeTester = new CountingNodeTester(4);
      assertNodeTestPasses(testXML, countingNodeTester, Node.TEXT_NODE);
    }
```

This test fails as there are 5 text nodes, and JUnit supplies the following message:

```
Expected node test to pass, but it failed! Counted 5 node(s) but
expected 4
```

Note that if your DOM implementation does not support the *DocumentTraversal* interface then XMLUnit will throw an *IllegalArgumentException* informing you that you cannot use the *NodeTest* or *NodeTester* classes. Unfortunately even if your DOM implementation does support *DocumentTraversal*, attributes are not exposed by iteration: however they can be examined from the Element node that contains them.

While the previous test could have been easily performed using XPath, there are times when Node iteration is more powerful. In general, this is true when there are programmatic relationships between nodes that can be more easily tested iteratively. The following test uses a custom *NodeTester* class to illustrate the potential:

```
public void testCustomNodeTester() throws Exception {
  String testXML = "<fibonacci><val>1</val><val>2</val><val>3</val>"
        + "<val>5</val><val>9</val></fibonacci>";
  NodeTest nodeTest = new NodeTest(testXML);
  assertNodeTestPasses(nodeTest, new FibonacciNodeTester(),
        new short[] {Node.TEXT_NODE, Node.ELEMENT_NODE}, true);
}

private class FibonacciNodeTester extends AbstractNodeTester {
  private int nextVal = 1, lastVal = 1, priorVal = 0;
  public void testText(Text text) throws NodeTestException {
        int val = Integer.parseInt(text.getData());
        if (nextVal != val) {
          throw new NodeTestException("Incorrect value", text);
        }
        nextVal = val + lastVal;
        priorVal = lastVal;
        lastVal = val;
  }
  public void testElement(Element element)
  throws NodeTestException {
        String name = element.getLocalName();
        if ("fibonacci".equals(name) || "val".equals(name)) {
              return;
        }
        throw new NodeTestException("Unexpected element", element);
  }
  public void noMoreNodes(NodeTest nodeTest)
  throws NodeTestException {
  }
}
```

The test fails because the XML contains the wrong value for the last number in the sequence:

```
Expected node test to pass, but it failed! Incorrect value [#text: 9]
```