XMLUnit Java User's Guide

COLLABORATORS

	TITLE : XMLUnit Java User's Guide	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Tim Bacon and Stefan Bodewig	February 7, 2013	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
1.0	January 2003	Documentation for XMLUnit Java 1.0	Tim Bacon
1.1	April 2007	Documentation for XMLUnit Java 1.1	
1.2	June 2008	Documentation for XMLUnit Java 1.2	
1.3	September 2009	Documentation for XMLUnit Java 1.3	
1.4	February 2013	Documentation for XMLUnit Java 1.4	

Contents

List of Tables

1 A Tour of XMLUnit

This first section contains a tour through XMLUnit's features, the next sections will cover them in more detail.

Note that it has a strong focus on using the `XMLTestCase` class which is one option to use XMLUnit, but not the only one. XMLUnit's features can be fully used without any dependency on JUnit at all.

1.1 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 `assertXMLEqual()` 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.

1.2 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 classes implementing the `XpathEngine` interface)
- 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).

1.3 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.

1.4 Configuring XMLUnit

There are many Java XML parsers available, and XMLUnit should work with any JAXP compliant parser library, such as Xerces-J [2] from the Apache Software Foundation. To use the XSLT and XPath features of XMLUnit a Trax (the XSLT portion of JAXP) compliant transformation engine is required, such as Xalan-J[3], from the Apache Software Foundation. To configure XMLUnit to use a specific parser and transformation engine set three System properties before any tests are run, e.g.

[1] For more information on JUnit see http://www.junit.org
[2] http://xerces.apache.org/
[3] http://xalan.apache.org/

Example 1.1 Configuring JAXP via System Properties

```
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");
```

You may want to read Section 2.4.1 for more details - in particular if you are using Java 1.4 or later.

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

Example 1.2 Configuring JAXP via XMLUnit class

```
XMLUnit.setControlParser("org.apache.xerces.jaxp.DocumentBuilderFactoryImpl");
XMLUnit.setTestParser("org.apache.xerces.jaxp.DocumentBuilderFactoryImpl");
XMLUnit.setSAXParserFactory("org.apache.xerces.jaxp.SAXParserFactoryImpl");
XMLUnit.setTransformerFactory("org.apache.xalan.processor.TransformerFactoryImpl");
```

1.5 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:

Example 1.3 A simple comparison test

```
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 `assertXMLEqual()` 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:

Example 1.4 Creating a `Diff` instance

```
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:

Example 1.5 Using `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[4] 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`.

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

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`:

Example 1.6 Using 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 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 nth child `<abc>` test element to the nth 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`:

Example 1.7 Using 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);
}
```

1.6 Comparing XML Transformations

XMLUnit can test XSLT 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:

Example 1.8 Testing the Result of a Transformation

```
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 XSLT transformation programmatically if required, for example as below:

Example 1.9 Using `Transform` programmatically

```
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());
}
```

1.7 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.

Example 1.10 Validating Against a DTD

```
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());
}
```

Starting with XMLUnit 1.1, the `Validator` class can also validate against one or more XML Schema definitions. See Section 4.1.2 for details.

XMLUnit 1.2 introduces a new `Validator` class that relies on JAXP 1.3's `javax.xml.validation` package. This Validator can validate against W3C XML Schema, but may support different Schema languages like RELAX NG if your JAXP implementation supports it. See Section 4.4 for details.

1.8 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:

Example 1.11 Using XPath Tests

```
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);
    assertXpathNotExists("//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 `Node`s. The methods in the previous test `assertXpathExists`, `assertXpathNotExists`, `assertXpathsEqual`, and `assertXpathsNotEqual` use these `NodeList`s. 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[5]:

Example 1.12 Testing XPath Values

```
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>"
        + "<jvm current='yes' latest='yes'>1.4.x</jvm></javaflavours>";
    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:

Example 1.13 Working with non well-formed HTML

```
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 =
```

[5] Each of the `assertXpath...()` methods uses an implementation of the `XpathEngine` interface to evaluate an XPath expression.

```
        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.

1.9 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:

Example 1.14 Using `CountingNodeTester`

```
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:

Example 1.15 Using a Custom `NodeTester`

```
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]
```

2 Using XMLUnit

2.1 Requirements

XMLUnit requires a JAXP compliant XML parser virtually everywhere. Several features of XMLUnit also require a JAXP compliant XSLT transformer. If it is available, a JAXP compliant XPath engine will be used for XPath tests.

To build XMLUnit at least JAXP 1.2 is required, this is the version provided by the Java class library in JDK 1.4. The JAXP 1.3 (i.e. Java5 and above) XPath engine can only be built when JAXP 1.3 is available.

As long as you don't require support for XML Namespaces or XML Schema, any JAXP 1.1 compliant implementations should work at runtime. For namespace and schema support you will need a parser that complies to JAXP 1.2 and supports the required feature. The XML parser shipping with JDK 1.4 (a version of Apache Crimson) for example is compliant to JAXP 1.2 but doesn't support Schema validation.

XMLUnit is supposed to build and run on any Java version after 1.3 (at least no new hard JDK 1.4 dependencies have been added in XMLUnit 1.1), but it has only been tested on JDK 1.4.2 and above.

To build XMLUnit JUnit 3.x (only tested with JUnit 3.8.x) is required. It is not required at runtime unless you intend to use the `XMLTestCase` or `XMLAssert` classes.

2.2 Basic Usage

XMLUnit consists of a few classes all living in the `org.custommonkey.xmlunit` package. You can use these classes directly from your code, no matter whether you are writing a unit test or want to use XMLUnit's features for any other purpose.

This section provides a few hints of where to start if you want to use a certain feature of XMLUnit, more details can be found in the more specific sections later in this document.

2.2.1 Comparing Pieces of XML

Heart and soul of XMLUnit's comparison engine is `DifferenceEngine` but most of the time you will use it indirectly via the `Diff` class.

You can influence the engine by providing (custom) implementations for various interfaces and by setting a couple of options on the `XMLUnit` class.

More information is available in Section 3.

2.2.2 Validating

All validation happens in the `Validator` class. The default is to validate against a DTD, but XML Schema validation can be enabled by an option (see `Validator.useXMLSchema`).

Several options of the `XMLUnit` class affect validation.

More information is available in Section 4.

2.2.3 XSLT Transformations

The `Transform` class provides an easy to use layer on top of JAXP's transformations. An instance of this class is initialized with the source document and a stylesheet and the result of the transformation can be retrieved as a `String` or DOM `Document`.

The output of `Transform` can be used as input to comparisons, validations, XPath tests and so on. There is no detailed sections on transformations since they are really only a different way to create input for the rest of XMLUnit's machinery. Examples can be found in Section 1.6.

It is possible to provide a custom `javax.xml.transform.URIResolver` via the `XMLUnit.setURIResolver` method.

You can access the underlying XSLT transformer via `XMLUnit.getTransformerFactory`.

2.2.4 XPath Engine

The central piece of XMLUnit's XPath support is the `XpathEngine` interface. Currently two implementations of the interface exist, `SimpleXpathEngine` and `org.custommonkey.xmlunit.jaxp13.Jaxp13XpathEngine`.

`SimpleXpathEngine` is a very basic implementation that uses your XSLT transformer under the covers. This also means it will expose you to the bugs found in your transformer like the transformer claiming a stylesheet couldn't be compiled for very basic XPath expressions. This has been reported to be the case for JDK 1.5.

`org.custommonkey.xmlunit.jaxp13.Jaxp13XpathEngine` uses JAXP 1.3's `javax.xml.xpath` package and seems to work more reliable, stable and performant than `SimpleXpathEngine`.

You use the `XMLUnit.newXpathEngine` method to obtain an instance of the `XpathEngine`. As of XMLUnit 1.1 this will try to use JAXP 1.3 if it is available and fall back to `SimpleXpathEngine`.

Instances of `XpathEngine` can return the results of XPath queries either as DOM `NodeList` or plain `Strings`.

More information is available in Section 5.

2.2.5 DOM Tree Walking

To test pieces of XML by traversing the DOM tree you use the `NodeTester` class. Each DOM `Node` will be passed to a `NodeTester` implementation you provide. The `AbstractNodeTester` class is provided as a NullObject Pattern base class for implementations of your own.

More information is available in Section 6.

2.3 Using XMLUnit With JUnit 3.x

Initially XMLUnit was tightly coupled to JUnit and the recommended approach was to write unit tests by inheriting from the `XMLTestCase` class. `XMLTestCase` provides a pretty long list of `assert...` methods that may simplify your interaction with XMLUnit's internals in many common cases.

The `XMLAssert` class provides the same set of `assert...`s as static methods. Use `XMLAssert` instead of `XMLTestCase` for your unit tests if you can't or don't want to inherit from `XMLTestCase`.

All power of XMLUnit is available whether you use `XMLTestCase` and/or `XMLAssert` or the underlying API directly. If you are using JUnit 3.x then using the specific classes may prove to be more convenient.

2.4 Common Configuration Options

2.4.1 JAXP

If you are using a JDK 1.4 or later, your Java class library already contains the required XML parsers and XSLT transformers. Still you may want to use a different parser/transformer than the one of your JDK - in particular since the versions shipping with some JDKs are known to contain serious bugs.

As described in Section 1.4 there are two main approaches to choose the XML parser of XSLT transformer: System properties and setters in the `XMLUnit` class.

If you use system properties you have the advantage that your choice affects the whole JAXP system, whether it is used inside of XMLUnit or not. If you are using JDK 1.4 or later you may also want to review the Endorsed Standards Override Mechanism to use a different parser/transformer than the one shipping with your JDK.

The second option - using the `XMLUnit` class - allows you to use different parsers for control and test documents, it even allows you to use different parsers for different test cases, if you really want to stretch it that far. It may also work for JDK 1.4 and above, even if you don't override the endorsed standards libraries.

You can access the underlying JAXP parser by `XMLUnit.newControlParser`, `XMLUnit.newTestParser`, `XMLUnit.getControlDocumentBuilderFactory`, `XMLUnit.getTestDocumentBuilderFactory` and `XMLUnit.getSAXParserFactory` (used by `Validator`). Note that all these methods return factories or parsers that are namespace aware.

The various `build...` methods in `XMLUnit` provide convenience layers for building DOM `Documents` using the configured parsers.

You can also set the class name for the `XPathFactory` to use when using JAXP 1.3 by passing the class name to `XMLUnit.setXPathFactory`.

2.4.2 EntityResolver

You can provide a custom `org.xml.sax.EntityResolver` for the control and test parsers via `XMLUnit.setControlEntityResolver` and `XMLUnit.setTestEntityResolver`. `Validator` uses the resolver set via `setControlEntityResolver` as well.

2.4.3 Element Content Whitespace

Element content whitespace - also known as ignorable whitespace - is whitespace contained in elements whose content model doesn't allow text content. I.e. the newline and space characters between `<foo>` and `<bar>` in the following example could belong into this category.

```
<foo>
  <bar/></foo>
```

Using `XMLUnit.setIgnoreWhitespace` it is possible to make the test and control parser ignore this kind of whitespace.

Note that setting this property to `true` usually doesn't have any effect since it only works on validating parsers and XMLUnit doesn't enable validation by default. It does have an effect when comparing pieces of XML, though, since the same flag is used for a different purpose as well in that case. See Section 3.8.1 for more details.

2.4.4 XSLT Stylesheet Version

Some features of XMLUnit use XSLT stylesheets under the covers, in particular XSLT will be used to strip element content whitespace or comments as well as by `SimpleXpathEngine`. These stylesheets only require a XSLT transformer that supports XSLT 1.0 and will say so in the `stylesheet` element.

If your XSLT transformer supports XSLT 2.0 or newer it may[6] issue a warning for these stylesheets which can be annoying. You can use `XMLUnit.setXSLTVersion` to make XMLUnit change the version attribute to a different value. Note that XMLUnit hasn't been tested with a value other than `"1.0"`.

2.5 Providing Input to XMLUnit

Most methods in XMLUnit that expect a piece of XML as input provide several overloads that obtain their input from different sources. The most common options are:

- A DOM `Document`.

  Here you have all control over the document's creation. Such a `Document` could as well be the result of an XSLT transformation via the `Transform` class.

- A SAX `InputSource`.

  This is the most generic way since `InputSource` allows you to read from arbitrary `InputStreams` or `Readers`. Use an `InputStream` wrapped by an `InputSource` if you want the XML parser to pick up the proper encoding from the XML declaration.

- A `String`.

  Here a DOM `Document` is built from the input `String` using the JAXP parser specified for control or test documents - depending on whether the input is a control or test piece of XML.

  Note that using a `String` assumes that your XML has already been converted from its XML encoding to a Java `String` upfront.

- A `Reader`.

  Here a DOM `Document` is built from the input `Reader` using the JAXP parser specified for control or test documents - depending on whether the input is a control or test piece of XML.

  Note that using a `Reader` is a bad choice if your XML encoding is different from your platform's default encoding since Java's IO system won't read your XML declaration. It is a good practice to use one of the other overloads rather than the `Reader` version to ensure encoding has been dealt with properly.

3 Comparing Pieces of XML

3.1 The Difference Engine

At the center of XMLUnit's support for comparisons is the `DifferenceEngine` class. In practice you rarely deal with it directly but rather use it via instances of `Diff` or `DetailedDiff` classes (see Section 3.5).

The `DifferenceEngine` walks two trees of DOM `Nodes`, the control and the test tree, and compares the nodes. Whenever it detects a difference, it sends a message to a configured `DifferenceListener` (see Section 3.3) and asks a `ComparisonController` (see Section 3.2) whether the current comparison should be halted.

In some cases the order of elements in two pieces of XML may not be significant. If this is true, the `DifferenceEngine` needs help to determine which `Elements` to compare. This is the job of an `ElementQualifier` (see Section 3.4).

The types of differences `DifferenceEngine` can detect are enumerated in the `DifferenceConstants` interface and represented by instances of the `Difference` class.

[6] The W3C recommendation says it SHOULD.

A `Difference` can be recoverable; recoverable `Differences` make the `Diff` class consider two pieces of XML similar while non-recoverable `Differences` render the two pieces different.

The types of `Differences` that are currently detected are listed in Table 1 to Table 4 (the first two columns refer to the `DifferenceConstants` class).

Note that some of the differences listed may be ignored by the `DifferenceEngine` if certain configuration options have been specified. See Section 3.8 for details.

`DifferenceEngine` passes differences found around as instances of the `Difference` class. In addition to the type of of difference this class also holds information on the nodes that have been found to be different. The nodes are described by `NodeDetail` instances that encapsulate the DOM `Node` instance as well as the XPath expression that locates the `Node` inside the given piece of XML. `NodeDetail` also contains a "value" that provides more information on the actual values that have been found to be different, the concrete interpretation depends on the type of difference as can be seen in Table 5.

As said in the first paragraph you won't deal with `DifferenceEngine` directly in most cases. In cases where `Diff` or `DetailedDiff` don't provide what you need you'd create an instance of `DifferenceEngine` passing a `ComparisonController` in the constructor and invoke `compare` with your DOM trees to compare as well as a `DifferenceListener` and `ElementQualifier`. The listener will be called on any differences while the `control` method is executing.

Example 3.1 Using `DifferenceEngine` Directly

```
class MyDifferenceListener implements DifferenceListener {
    private boolean calledFlag = false;
    public boolean called() { return calledFlag; }

    public int differenceFound(Difference difference) {
        calledFlag = true;
        return RETURN_ACCEPT_DIFFERENCE;
    }

    public void skippedComparison(Node control, Node test) {
    }
}

DifferenceEngine engine = new DifferenceEngine(myComparisonController);
MyDifferenceListener listener = new MyDifferenceListener();
engine.compare(controlNode, testNode, listener,
               myElementQualifier);
System.err.println("There have been "
                   + (listener.called() ? "" : "no ")
                   + "differences.");
```

3.2 ComparisonController

The `ComparisonController`'s job is to decide whether a comparison should be halted after a difference has been found. Its interface is:

```
    /**
     * Determine whether a Difference that the listener has been notified of
     *  should halt further XML comparison. Default behaviour for a Diff
     *  instance is to halt if the Difference is not recoverable.
     * @see Difference#isRecoverable
     * @param afterDifference the last Difference passed to <code>differenceFound</code>
     * @return true to halt further comparison, false otherwise
     */
    boolean haltComparison(Difference afterDifference);
```

[7] Note that the order of attributes is not significant in XML, different parsers may return attributes in a different order even if parsing the same XML document. There is an option to turn this check off - see Section 3.8 - but it is on by default for backwards compatibility reasons

[8] In order for this difference to be detected the parser must have been in validating mode when the piece of XML was parsed and the DTD or XML Schema must have been available.

ID	Constant	recoverable	Description
HAS_DOCTYPE_DECLARATION_ID	HAS_DOCTYPE_DECLARATION	true	One piece of XML has a DOCTYPE declaration while the other one has not.
DOCTYPE_NAME_ID	DOCTYPE_NAME	false	Both pieces of XML contain a DOCTYPE declaration but the declarations specify different names for the root element.
DOCTYPE_PUBLIC_ID_ID	DOCTYPE_PUBLIC_ID	false	Both pieces of XML contain a DOCTYPE declaration but the declarations specify different PUBLIC identifiers.
DOCTYPE_SYSTEM_ID_ID	DOCTYPE_SYSTEM_ID	true	Both pieces of XML contain a DOCTYPE declaration but the declarations specify different SYSTEM identifiers.
NODE_TYPE_ID	NODE_TYPE	false	The test piece of XML contains a different type of node than was expected. This type of difference will also occur if either the root control or test `Node` is `null` while the other is not.
NAMESPACE_PREFIX_ID	NAMESPACE_PREFIX	true	Two nodes use different prefixes for the same XML Namespace URI in the two pieces of XML.
NAMESPACE_URI_ID	NAMESPACE_URI	false	Two nodes in the two pieces of XML share the same local name but use different XML Namespace URIs.
SCHEMA_LOCATION_ID	SCHEMA_LOCATION	true	Two nodes have different values for the `schemaLocation` attribute of the XMLSchema-Instance namespace. The attribute could be present on only one of the two nodes.
NO_NAMESPACE_SCHEMA_LOCATION_ID	NO_NAMESPACE_SCHEMA_LOCATION	true	Two nodes have different values for the `noNamespaceSchemaLocation` attribute of the XMLSchema-Instance namespace. The attribute could be present on only one of the two nodes.

Table 1: Document level `Differences` detected by `DifferenceEngine`

ID	Constant	recoverable	Description
ELEMENT_TAG_NAME_ID	ELEMENT_TAG_NAME	false	The two pieces of XML contain elements with different tag names.
ELEMENT_NUM_ATTRIBUTES_ID	ELEMENT_NUM_ATTRIBUTES	false	The two pieces of XML contain a common element, but the number of attributes on the element is different.
HAS_CHILD_NODES_ID	HAS_CHILD_NODES	false	An element in one piece of XML has child nodes while the corresponding one in the other has not.
CHILD_NODELIST_LENGTH_ID	CHILD_NODELIST_LENGTH	false	Two elements in the two pieces of XML differ by their number of child nodes.
CHILD_NODELIST_SEQUENCE_ID	CHILD_NODELIST_SEQUENCE	true	Two elements in the two pieces of XML contain the same child nodes but in a different order.
CHILD_NODE_NOT_FOUND_ID	CHILD_NODE_NOT_FOUND	false	A child node in one piece of XML couldn't be matched against any other node of the other piece.
ATTR_SEQUENCE_ID	ATTR_SEQUENCE	true	The attributes on an element appear in different order[7] in the two pieces of XML.

Table 2: Element level Differences detected by DifferenceEngine

ID	Constant	recoverable	Description
ATTR_VALUE_EXPLICITLY_SPECIFIED_ID	ATTR_VALUE_EXPLICITLY_SPECIFIED	true	An attribute that has a default value according to the content model of the element in question has been specified explicitly in one piece of XML but not in the other.[8]
ATTR_NAME_NOT_FOUND_ID	ATTR_NAME_NOT_FOUND	false	One piece of XML contains an attribute on an element that is missing in the other.
ATTR_VALUE_ID	ATTR_VALUE	false	The value of an element's attribute is different in the two pieces of XML.

Table 3: Attribute level Differences detected by DifferenceEngine

ID	Constant	recoverable	Description
COMMENT_VALUE_ID	COMMENT_VALUE	false	The content of two comments is different in the two pieces of XML.
PROCESSING_INSTRUCTION_TARGET_ID	PROCESSING_INSTRUCTION_TARGET	false	The target of two processing instructions is different in the two pieces of XML.
PROCESSING_INSTRUCTION_DATA_ID	PROCESSING_INSTRUCTION_DATA	false	The data of two processing instructions is different in the two pieces of XML.
CDATA_VALUE_ID	CDATA_VALUE	false	The content of two CDATA sections is different in the two pieces of XML.
TEXT_VALUE_ID	TEXT_VALUE	false	The value of two texts is different in the two pieces of XML.

Table 4: Other `Differences` detected by `DifferenceEngine`

Difference.getId()	NodeDetail.getValue()
HAS_DOCTYPE_DECLARATION_ID	"not null" if the document has a DOCTYPE declaration, "null" otherwise.
DOCTYPE_NAME_ID	The name of the root element.
DOCTYPE_PUBLIC_ID	The PUBLIC identifier.
DOCTYPE_SYSTEM_ID	The SYSTEM identifier.
NODE_TYPE_ID	If one node was absent: "not null" if the node exists, "null" otherwise. If the node types differ the value will be a string-ified version of `org.w3c.dom.Node.getNodeType()`.
NAMESPACE_PREFIX_ID	The Namespace prefix.
NAMESPACE_URI_ID	The Namespace URI.
SCHEMA_LOCATION_ID	The attribute's value or "[attribute absent]" if it has not been specified.
NO_NAMESPACE_SCHEMA_LOCATION_ID	The attribute's value or "[attribute absent]" if it has not been specified.
ELEMENT_TAG_NAME_ID	The tag name with any Namespace information stripped.
ELEMENT_NUM_ATTRIBUTES_ID	The number of attributes present turned into a `String`.
HAS_CHILD_NODES_ID	"true" if the element has child nodes, "false" otherwise.
CHILD_NODELIST_LENGTH_ID	The number of child nodes present turned into a `String`.
CHILD_NODELIST_SEQUENCE_ID	The sequence number of this child node turned into a `String`.
CHILD_NODE_NOT_FOUND_ID	The name of the unmatched node or `null`.
ATTR_SEQUENCE_ID	The attribute's name.
ATTR_VALUE_EXPLICITLY_SPECIFIED_ID	"true" if the attribute has been specified, "false" otherwise.
ATTR_NAME_NOT_FOUND_ID	The attribute's name or `null`.
ATTR_VALUE_ID	The attribute's value.
COMMENT_VALUE_ID	The actual comment.
PROCESSING_INSTRUCTION_TARGET_ID	The processing instruction's target.
PROCESSING_INSTRUCTION_DATA_ID	The processing instruction's data.
CDATA_VALUE_ID	The content of the CDATA section.
TEXT_VALUE_ID	The actual text.

Table 5: Contents of `NodeDetail.getValue()` for `Differences`

Whenever a difference has been detected by the `DifferenceEngine` the `haltComparison` method will be called immediately after the `DifferenceListener` has been informed of the difference. This is true no matter what type of `Difference` has been found or which value the `DifferenceListener` has returned.

The only implementations of `ComparisonController` that ship with XMLUnit are `Diff` and `DetailedDiff`, see Section 3.5 for details about them.

A `ComparisonController` that halted the comparison on any non-recoverable difference could be implemented as:

Example 3.2 A Simple `ComparisonController`

```
public class HaltOnNonRecoverable implements ComparisonController {
    public boolean haltComparison(Difference afterDifference) {
        return !afterDifference.isRecoverable();
    }
}
```

3.3 `DifferenceListener`

`DifferenceListener` contains two callback methods that are invoked by the `DifferenceEngine` when differences are detected:

```
    /**
     * Receive notification that 2 nodes are different.
     * @param difference a Difference instance as defined in {@link
     * DifferenceConstants DifferenceConstants} describing the cause
     * of the difference and containing the detail of the nodes that
     * differ
     * @return int one of the RETURN_... constants describing how this
     * difference was interpreted
     */
    int differenceFound(Difference difference);

    /**
     * Receive notification that a comparison between 2 nodes has been skipped
     *  because the node types are not comparable by the DifferenceEngine
     * @param control the control node being compared
     * @param test the test node being compared
     * @see DifferenceEngine
     */
    void skippedComparison(Node control, Node test);
```

`differenceFound` is invoked by `DifferenceEngine` as soon as a difference has been detected. The return value of that method is completely ignored by `DifferenceEngine`, it becomes important when used together with `Diff`, though (see Section 3.5). The return value should be one of the four constants defined in the the `DifferenceListener` interface:

```
    /**
     * Standard return value for the <code>differenceFound</code> method.
     * Indicates that the <code>Difference</code> is interpreted as defined
     * in {@link DifferenceConstants DifferenceConstants}.
     */
    int RETURN_ACCEPT_DIFFERENCE;
    /**
     * Override return value for the <code>differenceFound</code> method.
     * Indicates that the nodes identified as being different should be
     * interpreted as being identical.
     */
    int RETURN_IGNORE_DIFFERENCE_NODES_IDENTICAL;
    /**
     * Override return value for the <code>differenceFound</code> method.
```

```
     * Indicates that the nodes identified as being different should be
     * interpreted as being similar.
     */
    int RETURN_IGNORE_DIFFERENCE_NODES_SIMILAR;
    /**
     * Override return value for the <code>differenceFound</code> method.
     * Indicates that the nodes identified as being similar should be
     * interpreted as being different.
     */
    int RETURN_UPGRADE_DIFFERENCE_NODES_DIFFERENT = 3;
```

The `skippedComparison` method is invoked if the `DifferenceEngine` encounters two `Nodes` it cannot compare. Before invoking `skippedComparison` `DifferenceEngine` will have invoked `differenceFound` with a `Difference` of type `NODE_TYPE`.

A custom `DifferenceListener` that ignored any DOCTYPE related differences could be written as:

Example 3.3 A `DifferenceListener` that Ignores DOCTYPE Differences

```
public class IgnoreDoctype implements DifferenceListener {
    private static final int[] IGNORE = new int[] {
        DifferenceConstants.HAS_DOCTYPE_DECLARATION_ID,
        DifferenceConstants.DOCTYPE_NAME_ID,
        DifferenceConstants.DOCTYPE_PUBLIC_ID_ID,
        DifferenceConstants.DOCTYPE_SYSTEM_ID_ID
    };

    static {
        Arrays.sort(IGNORE);
    }

    public int differenceFound(Difference difference) {
        return Arrays.binarySearch(IGNORE, difference.getId()) >= 0
            ? RETURN_IGNORE_DIFFERENCE_NODES_IDENTICAL
            : RETURN_ACCEPT_DIFFERENCE;
    }

    public void skippedComparison(Node control, Node test) {
    }
}
```

Apart from `Diff` and `DetailedDiff` XMLUnit ships with an additional implementation of `DifferenceListener`.

3.3.1 IgnoreTextAndAttributeValuesDifferenceListener

`IgnoreTextAndAttributeValuesDifferenceListener` doesn't do anything in `skippedComparison`. It "downgrades" `Differences` of type `ATTR_VALUE`, `ATTR_VALUE_EXPLICITLY_SPECIFIED` and `TEXT_VALUE` to recoverable differences.

This means if instances of `IgnoreTextAndAttributeValuesDifferenceListener` are used together with `Diff` then two pieces of XML will be considered similar if they have the same basic structure. They are not considered identical, though.

Note that the list of ignored differences doesn't cover all textual differences. You should configure XMLUnit to ignore comments and whitespace and to consider CDATA sections and text nodes to be the same (see Section 3.8) in order to cover `COMMENT_VALUE` and `CDATA_VALUE` as well.

3.4 ElementQualifier

When `DifferenceEngine` encounters a list of DOM `Elements` as children of another `Element` it will ask the configured `ElementQualifier` which `Element` of the control piece of XML should be compared to which of the test piece. Its contract

is:

```
    /**
     * Determine whether two elements are comparable
     * @param control an Element from the control XML NodeList
     * @param test an Element from the test XML NodeList
     * @return true if the elements are comparable, false otherwise
     */
    boolean qualifyForComparison(Element control, Element test);
```

For any given `Element` in the control piece of XML `DifferenceEngine` will cycle through the corresponding list of `Elements` in the test piece of XML until `qualifyForComparison` has returned `true` or the test document is exhausted.

When using `DifferenceEngine` or `Diff` it is completely legal to set the `ElementQualifier` to `null`. In this case any kind of `Node` is compared to the test `Node` that appears at the same position in the sequence.

Example 3.4 Example Nodes for `ElementQualifier` (the comments are not part of the example)

```
<!-- control piece of XML -->
<parent>
  <child1/>                       <!-- control node 1 -->
  <child2/>                       <!-- control node 2 -->
  <child2 foo="bar">xyzzy</child2> <!-- control node 3 -->
  <child2 foo="baz"/>             <!-- control node 4 -->
</parent>

<!-- test piece of XML -->
<parent>
  <child2 foo="baz"/>             <!-- test node 1 -->
  <child1/>                       <!-- test node 2 -->
  <child2>xyzzy</child2>          <!-- test node 3 -->
  <child2 foo="bar"/>             <!-- test node 4 -->
</parent>
```

Taking Example 3.4 without any `ElementQualifier` `DifferenceEngine` will compare control node `n` to test node `n` for `n` between 1 and 4. In many cases this is exactly what is desired, but sometimes `<a><b/><c/></a>` should be similar to `<a><c/><b/></a>` because the order of elements doesn't matter - this is when you'd use a different `ElementQualifier`. XMLUnit ships with several implementations.

3.4.1 `ElementNameQualifier`

Only `Elements` with the same name - and Namespace URI if present - qualify.

In Example 3.4 this means control node 1 will be compared to test node 2. Then control node 2 will be compared to test node 3 because `DifferenceEngine` will start to search for the matching test `Element` at the second test node, the same sequence number the control node is at. Control node 3 is compared to test node 3 as well and control node 4 to test node 4.

3.4.2 `ElementNameAndAttributeQualifier`

Only `Elements` with the same name - and Namespace URI if present - as well as the same values for all attributes given in `ElementNameAndAttributeQualifier`'s constructor qualify.

Let's say `"foo"` has been passed to `ElementNameAndAttributeQualifier`'s constructor when looking at Example 3.4. This again means control node 1 will be compared to test node 2 since they do have the same name and no value at all for attribute `"foo"`. Then control node 2 will be compared to test node 3 - again, no value for `"foo"`. Control node 3 is compared to test node 4 as they have the same value `"bar"`. Finally control node 4 is compared to test node 1; here `DifferenceEngine` searches from the beginning of the test node list after test node 4 didn't match.

There are three constructors in `ElementNameAndAttributeQualifier`. The no-arg constructor creates an instance that compares all attributes while the others will compare a single attribute or a given subset of all attributes.

3.4.3 ElementNameAndTextQualifier

Only Elements with the same name - and Namespace URI if present - as well as the same text content nested into them qualify.

In Example 3.4 this means control node 1 will be compared to test node 2 since they both don't have any nested text at all. Then control node 2 will be compared to test node 4. Control node 3 is compared to test node 3 since they have the same nested text and control node 4 to test node 4.

3.4.4 org.custommonkey.xmlunit.examples.RecursiveElementNameAndTextQualifier

All ElementQualifiers seen so far only looked at the Elements themselves and not at the structure nested into them at a deeper level. A frequent user question has been which ElementQualifier should be used if the pieces of XML in Example 3.5 should be considered similar.

Example 3.5 Example for RecursiveElementNameAndTextQualifier (the comments are not part of the example)

```
<!-- control -->
<table>
  <tr>            <!-- control row 1 -->
    <td>foo</td>
  </tr>
  <tr>            <!-- control row 2 -->
    <td>bar</td>
  </tr>
</table>

<!-- test -->
<table>
  <tr>            <!-- test row 1 -->
    <td>bar</td>
  </tr>
  <tr>            <!-- test row 2 -->
    <td>foo</td>
  </tr>
</table>
```

At first glance ElementNameAndTextQualifier should work but it doesn't. When DifferenceEngine processed the children of table it would compare control row 1 to test row 1 since both tr elements have the same name and both have no textual content at all.

What is needed in this case is an ElementQualifier that looks at the element's name, as well as the name of the first child element and the text nested into that first child element. This is what RecursiveElementNameAndTextQualifier does.

RecursiveElementNameAndTextQualifier ignores whitespace between the elements leading up to the nested text.

3.4.5 org.custommonkey.xmlunit.examples.MultiLevelElementNameAndTextQualifier

MultiLevelElementNameAndTextQualifier has in a way been the predecessor of Section 3.4.4. It also matches element names and those of nested child elements until it finds matches, but unlike RecursiveElementNameAndTextQualifier, you must tell MultiLevelElementNameAndTextQualifier at which nesting level it should expect the nested text.

MultiLevelElementNameAndTextQualifier's constructor expects a single argument which is the nesting level of the expected text. If you use an argument of 1, MultiLevelElementNameAndTextQualifier is identical to ElementNameAndTextQualifier. In Example 3.5 a value of 2 would be needed.

By default MultiLevelElementNameAndTextQualifier will not ignore whitespace between the elements leading up to the nested text. If your piece of XML contains this sort of whitespace (like Example 3.5 which contains a newline and several space characters between <tr> and <td>) you can either instruct XMLUnit to ignore whitespace completely (see Section 3.8.1)

or use the two-arg constructor of `MultiLevelElementNameAndTextQualifier` introduced with XMLUnit 1.2 and set the `ignoreEmptyTexts` argument to true.

In general `RecursiveElementNameAndTextQualifier` requires less knowledge upfront and its whitespace-handling is more intuitive.

3.5 `Diff` and `DetailedDiff`

`Diff` and `DetailedDiff` provide simplified access to `DifferenceEngine` by implementing the `ComparisonController` and `DifferenceListener` interfaces themselves. They cover the two most common use cases for comparing two pieces of XML: checking whether the pieces are different (this is what `Diff` does) and finding all differences between them (this is what `DetailedDiff` does).

`DetailedDiff` is a subclass of `Diff` and can only be constructed by creating a `Diff` instance first.

The major difference between them is their implementation of the `ComparisonController` interface: `DetailedDiff` will never stop the comparison since it wants to collect all differences. `Diff` in turn will halt the comparison as soon as the first `Difference` is found that is not recoverable. In addition `DetailedDiff` collects all `Differences` in a list and provides access to it.

By default `Diff` will consider two pieces of XML as identical if no differences have been found at all, similar if all differences that have been found have been recoverable (see Table 1 to Table 4) and different as soon as any non-recoverable difference has been found.

It is possible to specify a `DifferenceListener` to `Diff` using the `overrideDifferenceListener` method. In this case each `Difference` will be evaluated by the passed in `DifferenceListener`. By returning `RETURN_IGNORE_DIFFERENCE_NODES_IDENTICAL` the custom listener can make `Diff` ignore the difference completely. Likewise any `Difference` for which the custom listener returns `RETURN_IGNORE_DIFFERENCE_NODES_SIMILAR` will be treated as if the `Difference` was recoverable.

There are several overloads of the `Diff` constructor that allow you to specify your piece of XML in many ways. There are overloads that accept additional `DifferenceEngine` and `ElementQualifier` arguments. Passing in a `DifferenceEngine` of your own is the only way to use a `ComparisonController` other than `Diff`.

Note that `Diff` and `DetailedDiff` use `ElementNameQualifier` as their default `ElementQualifier`. This is different from `DifferenceEngine` which defaults to no `ElementQualifier` at all.

To use a custom `ElementQualifier` you can also use the `overrideElementQualifier` method. Use this with an argument of `null` to unset the default `ElementQualifier` as well.

To compare two pieces of XML you'd create a `Diff` instance from those two pieces and invoke `identical` to check that there have been no differences at all and `similar` to check that any difference, if any, has been recoverable. If the pieces are identical they are also similar. Likewise if they are not similar they can't be identical either.

Example 3.6 Comparing Two Pieces of XML Using `Diff`

```
Diff d = new Diff("<a><b/><c/></a>", "<a><c/><b/></a>");
assertFalse(d.identical()); // CHILD_NODELIST_SEQUENCE Difference
assertTrue(d.similar());
```

The result of the comparison is cached in `Diff`, repeated invocations of `identical` or `similar` will not reevaluate the pieces of XML.

`DetailedDiff` provides only a single constructor that expects a `Diff` as argument. Don't use `DetailedDiff` if all you need to know is whether two pieces of XML are identical/similar - use `Diff` directly since its short-cut `ComparisonController` implementation will save time in this case.

Example 3.7 Finding All Differences Using `DetailedDiff`

```
Diff d = new Diff("<a><b/><c/></a>", "<a><c/><b/></a>");
DetailedDiff dd = new DetailedDiff(d);
dd.overrideElementQualifier(null);
assertFalse(dd.similar());
List l = dd.getAllDifferences();
assertEquals(2, l.size()); // expected <b/> but was <c/> and vice versa
```

3.6 `MatchTracker`

Sometimes you might be interested in any sort of comparison result and want to get notified of successful matches as well. Maybe you want to provide feedback on the amount of differences and similarities between two documents, for example.

The interface `MatchTracker` can be implemented to get notified on each and every successful match, note that there may be a lot more comparisons going on than you might expect and that your callback gets notified a lot.

Example 3.8 The `MatchTracker` interface

```
package org.custommonkey.xmlunit;

/**
 * Listener for callbacks from a {@link DifferenceEngine#compare
 * DifferenceEngine comparison} that is notified on each and every
 * comparision that resulted in a match.
 */
public interface MatchTracker {
    /**
     * Receive notification that 2 match.
     * @param match a Difference instance as defined in {@link
     * DifferenceConstants DifferenceConstants} describing the test
     * that matched and containing the detail of the nodes that have
     * been compared
     */
    void matchFound(Difference difference);
}
```

Despite its name the `Difference` instance passed into the `matchFound` method really describes a match and not a difference. You can expect that the `getValue` method on both the control and the test `NodeDetail` will be equal.

`DifferenceEngine` provides a constructor overload that allows you to pass in a `MatchTracker` instance and also provides a `setMatchTracker` method. `Diff` and `DetailedDiff` provide `overrideMatchTracker` methods that fill the same purpose.

Note that your `MatchTracker` won't receive any callbacks once the configured `ComparisonController` has decided that `DifferenceEngine` should halt the comparison.

3.7 JUnit 3.x Convenience Methods

`XMLAssert` and `XMLTestCase` contain quite a few overloads of methods for comparing two pieces of XML.

The method's names use the word `Equal` to mean the same as `similar` in the `Diff` class (or throughout this guide). So `assertXMLEqual` will assert that only recoverable differences have been encountered where `assertXMLNotEqual` asserts that some differences have been non-recoverable. `assertXMLIdentical` asserts that there haven't been any differences at all while `assertXMLNotIdentical` asserts that there have been differences (recoverable or not).

Most of the overloads of `assertXMLEqual` just provide different means to specify the pieces of XML as `Strings`, `InputSources`, `Readers`[9] or `Documents`. For each method there is a version that takes an additional `err` argument which is used to create the message if the assertion fails.

If you don't need any control over the `ElementQualifier` or `DifferenceListener` used by `Diff` these methods will save some boilerplate code. If `CONTROL` and `TEST` are pieces of XML represented as one of the supported inputs then

```
Diff d = new Diff(CONTROL, TEST);
assertTrue("expected pieces to be similar, " + d.toString(),
           d.similar());
```

and

```
assertXMLEqual("expected pieces to be similar", CONTROL, TEST);
```

are equivalent.

If you need more control over the `Diff` instance there is a version of `assertXMLEqual` (and `assertXMLIdentical`) that accepts a `Diff` instance as its argument as well as a `boolean` indicating whether you expect the `Diff` to be `similar` (`identical`) or not.

`XMLTestCase` contains a couple of `compareXML` methods that really are only shortcuts to `Diff`'s constructors.

There is no way to use `DifferenceEngine` or `DetailedDiff` directly via the convenience methods.

3.8 Configuration Options

Unless you are using `Document` or `DOMSource` overrides when specifying your pieces of XML, XMLUnit will use the configured XML parsers (see Section 2.4.1) and `EntityResolvers` (see Section 2.4.2). There are configuration options to use different settings for the control and test pieces of XML.

In addition some of the other configuration settings may lead to XMLUnit using the configured XSLT transformer (see Section 2.4.1) under the covers.

3.8.1 Whitespace Handling

Two different configuration options affect how XMLUnit treats whitespace in comparisons:

- Element Content Whitespace (see Section 2.4.3)

  If XMLUnit has been configured to ignore element content whitespace it will trim any text nodes found by the parser. This means that there won't appear to be any textual content in element `<foo>` for the following example. If you don't set `XMLUnit.setIgnoreWhitespace` there would be textual content consisting of a new line character.

  ```
  <foo>
  </foo>
  ```

  At the same time the following two `<foo>` elements will be considered identical if the option has been enabled, though.

  ```
  <foo>bar</foo>
  <foo> bar </foo>
  ```

  When this option is set to `true`, `Diff` will use the XSLT transformer under the covers.

- "Normalizing" Whitespace If you set `XMLUnit.setNormalizeWhitespace` to true then XMLUnit will replace any kind of whitespace found in character content with a SPACE character and collapse consecutive whitespace characters to a single SPACE. It will also trim the resulting character content on both ends.

  The following two `<foo>` elements will be considered identical if the option has been set:

[9] See Section 2.5 for some advice on choosing your input format.

```
<foo>bar baz</foo>
<foo> bar
            baz</foo>
```

Note that this is not related to "normalizing" the document as a whole (see Section 3.8.2).

3.8.2 "Normalizing" `Documents`

"Normalize" in this context corresponds to the `normalize` method in DOM's `Document` class. It is the process of merging adjacent `Text` nodes and is not related to "normalizing whitespace" as described in the previous section.

Usually you don't need to care about this option since the XML parser is required to normalize the `Document` when creating it. The only reason you may want to change the option via `XMLUnit.setNormalize` is that your `Document` instances have not been created by an XML parser but rather been put together in memory using the DOM API directly.

3.8.3 Ignoring Comments

Using `XMLUnit.setIgnoreComments` you can make XMLUnit's difference engine ignore comments completely.

When this option is set to `true`, `Diff` will use the XSLT transformer under the covers.

3.8.4 Treating CDATA Sections and Text Nodes Alike

It is not always necessary to know whether a text has been put into a CDATA section or not. Using `XMLUnit.setIgnoreDiffBetweenTextAndCDATA` you can make XMLUnit consider the following two pieces of XML identical:

```
<foo>&lt;bar&gt;</foo>
```

```
<foo><![CDATA[<bar>]]></foo>
```

3.8.5 Entity Reference Expansion

Normally the XML parser will expand character references to their Unicode equivalents but for more complex entity definitions the parser may expand them or not. Using `XMLUnit.setExpandEntityReferences` you can control the parser's setting.

3.8.6 Comparison of Unmatched Elements

When XMLUnit cannot match a control Element to a test Element (the configured ElementQualifier - see Section 3.4 - doesn't return true for any of the test Elements) it will try to compare it against the first unmatched test Element (if there is one). Starting with XMLUnit 1.3 one can use `XMLUnit.setCompareUnmatched` to disable this behavior and generate `CHILD_NODE_NOT_FOUND` differences instead.

If the control document is

```
<root>
  <a/>
</root>
```

and the test document is

```
<root>
  <b/>
</root>
```

the default setting will create a single `ELEMENT_TAG_NAME` Difference ("expected a but found b"). Setting `XMLUnit.setCompareUnmatched` to false will create two Differences of type `CHILD_NODE_NOT_FOUND` (one for "a" and one for "b") instead.

4 Validating XML Documents

4.1 The `Validator` Class

The `Validator` class encapsulates XMLUnit's validation support. It will use the `SAXParser` configured in XMLUnit (see Section 2.4.1).

The piece of XML to validate is specified in the constructor. The constructors using more than a single argument are only relevant if you want to validate against a DTD and need to provide the location of the DTD itself - for details see the next section.

By default, `Validator` will validate against a DTD, but it is possible to validate against a (or multiple) Schema(s) as well. Schema validation requires an XML parser that supports it, of course.

4.1.1 DTD Validation

Validating against a DTD is straight forward if the piece of XML contains a `DOCTYPE` declaration with a `SYSTEM` identifier that can be resolved at validation time. Simply create a `Validator` object using one of the single argument constructors.

Example 4.1 Validating Against the DTD Defined in `DOCTYPE`

```
InputSource is = new InputSource(new FileInputStream(myXmlDocument));
Validator v = new Validator(is);
boolean isValid = v.isValid();
```

If the piece of XML doesn't contain any `DOCTYPE` declaration at all or it contains a `DOCTYPE` but you want to validate against a different DTD, you'd use one of the three argument versions of `Validator`'s constructors. In this case the `publicId` argument becomes the `PUBLIC` and `systemId` the `SYSTEM` identifier of the `DOCTYPE` that is implicitly added to the piece of XML. Any existing `DOCTYPE` will be removed. The `systemId` should be a URL that can be resolved by your parser.

Example 4.2 Validating a Piece of XML that doesn't Contain a `DOCTYPE`

```
InputSource is = new InputSource(new FileInputStream(myXmlDocument));
Validator v = new Validator(is,
                            (new File(myDTD)).toURI().toURL().toString(),
                            myPublicId);
boolean isValid = v.isValid();
```

If the piece of XML already has the correct `DOCTYPE` declaration but the declaration either doesn't specify a `SYSTEM` identifier at all or you want the `SYSTEM` identifier to resolve to a different location you have two options:

- Use one of the two argument constructors and specify the alternative URL as `systemId`.

Example 4.3 Validating Against a Local DTD

```
InputSource is = new InputSource(new FileInputStream(myXmlDocument));
Validator v = new Validator(is,
                            (new File(myDTD)).toURI().toURL().toString());
boolean isValid = v.isValid();
```

- Use a custom `EntityResolver` via `XMLUnit.setControlEntityResolver` together with one of the single argument constructor overloads of Validator.

  This approach would allow you to use an OASIS catalog[10] in conjunction with the Apache XML Resolver library[11] to resolve the DTD location as well as the location of any other entity in your piece of XML, for example.

[10] http://www.oasis-open.org/committees/download.php/14809/xml-catalogs.html
[11] http://xml.apache.org/commons/components/resolver/index.html

Example 4.4 Validating Against a DTD Using Apache's XML Resolver and an XML Catalog

```
InputSource is = new InputSource(new FileInputStream(myXmlDocument));
XMLUnit.setControlEntityResolver(new CatalogResolver());
Validator v = new Validator(is);
boolean isValid = v.isValid();
```

```
#CatalogManager.properties

verbosity=1
relative-catalogs=yes
catalogs=/some/path/to/catalog
prefer=public
static-catalog=yes
catalog-class-name=org.apache.xml.resolver.Resolver
```

```
<!-- catalog file -->

<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">
  <public publicId="-//Some//DTD V 1.1//EN"
          uri="mydtd.dtd"/>
</catalog>
```

4.1.2 XML Schema Validation

In order to validate against the XML Schema language Schema validation has to be enabled via the `useXMLSchema` method of `Validator`.

By default the parser will try to resolve the location of Schema definition files via a `schemaLocation` attribute if it is present in the piece of XML or it will try to open the Schema's URI as an URL and read from it.

The `setJAXP12SchemaSource` method of `Validator` allows you to override this behavior as long as the parser supports the `http://java.sun.com/xml/jaxp/properties/schemaSource` property in the way described in "JAXP 1.2 Approved CHANGES"[12].

`setJAXP12SchemaSource`'s argument can be one of

- A `String` which contains an URI.
- An `InputStream` the Schema can be read from.
- An `InputSource` the Schema can be read from.
- A `File` the Schema can be read from.
- An array containing any of the above.

If the property has been set using a `String`, the `Validator` class will provide its `systemId` as specified in the constructor when asked to resolve it. You must only use the single argument constructors if you want to avoid this behavior. If no `systemId` has been specified, the configured `EntityResolver` may still be used.

Example 4.5 Validating Against a Local XML Schema

```
InputSource is = new InputSource(new FileInputStream(myXmlDocument));
Validator v = new Validator(is);
v.useXMLSchema(true);
v.setJAXP12SchemaSource(new File(myXmlSchemaFile));
boolean isValid = v.isValid();
```

[12] http://java.sun.com/webservices/jaxp/change-requests-11.html

4.2 JUnit 3.x Convenience Methods

Both `XMLAssert` and `XMLTestCase` provide an `assertXMLValid(Validator)` method that will fail if `Validator`'s `isValid` method returns `false`.

In addition several overloads of the `assertXMLValid` method are provided that directly correspond to similar overloads of `Validator`'s constructor. These overloads don't support XML Schema validation at all.

`Validator` itself provides an `assertIsValid` method that will throw an `AssertionFailedError` if validation fails.

Neither method provides any control over the message of the `AssertionFailedError` in case of a failure.

4.3 Configuration Options

- `Validator` uses a SAX parser created by the configured SAX parser factory (see Section 2.4.1).
- It will use the "control" `EntityResolver` if one has been specified (see Section 2.4.2).
- The location of a DTD can be specified via `Validator`'s `systemId` constructor argument or a custom EntityResolver (see Section 4.1.1).
- XML Schema validation is enabled via `Validator.useXMLSchema(true)`.
- The location(s) of XML Schema document(s) can be specified via `Validator.setJAXP12SchemaSource` (see Section 4.1.2).

4.4 JAXP 1.3 Validation

JAXP 1.3 - shipping with Java5 or better and available as a separate product for earlier Java VMs - introduces a new package `javax.xml.validation` designed for validations of snippets of XML against different schema languages. Any compliant implementation must support the W3C XML Schema language, but other languages like RELAX NG or Schematron may be supported as well.

The class `org.custommonkey.xmlunit.jaxp13.Validator` can be used to validate a piece of XML against a schema definition but also to validate the schema definition itself. By default `Validator` will assume your definition uses the W3C XML Schema language, but it provides a constructor that can be used to specify a different language via an URL supported by the `SchemaFactory` class. Alternatively you can specify the schema factory itself.

The schema definition itself can be given via `Source` elements, just like the pieces of XML to validate are specified as `Source` as well.

The following example uses `org.custommonkey.xmlunit.jaxp13.Validator` to perform the same type of validation shown in Example 4.5.

Example 4.6 Validating Against a Local XML Schema

```
Validator v = new Validator();
v.addSchemaSource(new StreamSource(new File(myXmlSchemaFile)));
StreamSource is = new StreamSource(new File(myXmlDocument));
boolean isValid = v.isInstanceValid(is);
```

Validating a schema definition is shown in the next example.

Example 4.7 Validating an XML Schema Definition

```
Validator v = new Validator();
v.addSchemaSource(new StreamSource(new File(myXmlSchemaFile)));
boolean isValid = v.isSchemaValid();
```

There is no explicit JUnit 3 support for `org.custommonkey.xmlunit.jaxp13.Validator`.

5 XPath Tests

5.1 XPath Engines

Central to XMLUnit's XPath support is the `XpathEngine` interface which consists of only three methods:

```
/**
 * Execute the specified xpath syntax <code>select</code> expression
 * on the specified document and return the list of nodes (could have
 * length zero) that match
 * @param select
 * @param document
 * @return list of matching nodes
 */
NodeList getMatchingNodes(String select, Document document)
    throws XpathException;

/**
 * Evaluate the result of executing the specified XPath syntax
 * <code>select</code> expression on the specified document
 * @param select
 * @param document
 * @return evaluated result
 */
String evaluate(String select, Document document)
    throws XpathException;

/**
 * Establish a namespace context.
 */
void setNamespaceContext(NamespaceContext ctx);
```

The first two methods expect an XPath expression that selects content from the DOM document that is the second argument. The result of the selection can be either a DOM `NodeList` or a `String`. The later form tries to flatten the result, the value is said to be "String-ified".

The third method is part of XMLUnit's support for XML Namespaces in XPath expressions. See Section 5.2 for more details.

There are two implementations of the interface, `org.custommonkey.xmlunit.SimpleXpathEngine` and `org.custommonkey.xmlunit.jaxp13.Jaxp13XpathEngine`. The first implementation is the only one available in XMLUnit 1.0 and uses the configured JAXP XSLT transformer. The second is new to XMLUnit 1.1 and only available if JAXP 1.3 or later is supported, which should be the case for Java 5 and later.

`XpathException` is an `Exception` that will be thrown for invalid XPath expressions or other problems with the underlying XPath engine. It will typically wrap a `javax.xml.xpath.XPathExpressionException` in the `Jaxp13XpathEngine` case or a `javax.xml.transform.TransformerException` when `SimpleXpathEngine` is used.

The `XMLUnit.newXpathEngine` method will first try to create an instance of `Jaxp13XpathEngine` and fall back to `SimpleXpathEngine` if JAXP 1.3 is not supported.

One example of using the XPath support is included inside it `org.custommonkey.xmlunit.examples` package. It asserts that the string-ified form of an XPath selection matches a regular expression. The code needed for this is:

Example 5.1 Matching an XPath Selection Against a Regular Expression

```
XpathEngine engine = XMLUnit.newXpathEngine();
String value = engine.evaluate(xpath, doc);
Assert.assertTrue(message, value.matches(regex));
```

5.2 Using XML Namespaces in XPath Selectors

Starting with XMLUnit 1.1 XML Namespaces are supported for XPath queries.

The `NamespaceContext` interface provides a mapping from prefix to namespace URI and is used by the XPath engine. XPath selections then use the mapping's prefixes where needed. Note that a prefix used in the document under test and a prefix given as part of the `NamespaceContext` are not related at all; the context only applies to the XPath expression, the prefix used in the document is ignored completely.

Right now XMLUnit provides only a single implementation of the `NamespaceContext` interface: `SimpleNamespaceContext`. This implementation expects a `java.util.Map` as its constructor argument. The `Map` must contain (`String`) prefixes as keys and (`String`) namespace URIs as values.

Note there is nothing like a default namespace in XPath selectors. If you are using namespaces in your XPath, all namespaces need a prefix (of length greater than zero). This is independent of the prefixes used in your document.

The following example is taken from XMLUnit's own tests. It demonstrates that the namespace prefix of the document under test is irrelevant and shows how to set up the namespace context.

Example 5.2 Using Namespaces in XPath Tests

```
String testDoc = "<t:test xmlns:t=\"urn:foo\"><t:bar/></t:test>";
Document d = XMLUnit.buildControlDocument(testDoc);
HashMap m = new HashMap();
m.put("foo", "urn:foo");

NamespaceContext ctx = new SimpleNamespaceContext(m);
XpathEngine engine = XMLUnit.newXpathEngine();
engine.setNamespaceContext(ctx);

NodeList l = engine.getMatchingNodes("//foo:bar", d);
assertEquals(1, l.getLength());
assertEquals(Node.ELEMENT_NODE, l.item(0).getNodeType());
```

It is possible to set a global `NamespaceContext`, see Section 5.4 for details.

5.3 JUnit 3.x Convenience Methods

`XMLTestCase` and `XMLAssert` provide several overloads for the following common types of assertions:

- Two XPath expression should return the same DOM `NodeList` as result: `assertXpathsEqual`. There are methods that use two different expressions on the same document and others that compare expressions selecting from two different documents.

  The `NodeLists` are wrapped into a surrogate root XML element and the resulting DOM `Documents` are compared using `Diff.similar()`.
- The opposite of the above, the expressions should yield different results: `assertXpathsNotEqual`.
- Two XPath expression should return the same "String-ified" result: `assertXpathValuesEqual`. There are methods that use two different expressions on the same document and others that compare expressions selecting from two different documents.
- The opposite of the above, the expressions should yield different results: `assertXpathValuesNotEqual`.
- The XPath expression should return an expected value when "String-ified": `assertXpathEvaluatesTo`.
- The `NodeList` selected by an XPath expression is not empty: `assertXpathExists`.
- The `NodeList` selected by an XPath expression is empty: `assertXpathNotExists`.

Neither method provides any control over the message of the `AssertionFailedError` in case of a failure.

5.4 Configuration Options

When using `XpathEngine` directly you are responsible for creating the DOM document yourself. If you use the convenience methods of `XMLTestCase` or `XMLAssert` you have several options to specify the input; XMLUnit will use the control or test parser that has been configured (see Section 2.4.1) to create a DOM document from the given piece of XML in that case - using the configured `EntityResolver`(s) (see Section 2.4.2) if any.

If JAXP 1.3 is not available, `SimpleXpathEngine` will use the configured JAXP XSLT transformer (see Section 2.4.1) under the covers.

When using JAXP 1.3 you can chose the actual `XPathFactory` implementation using `XMLUnit.setXPathFactory`.

It is possible to establish a global `NamespaceContext` with the help of the `XMLUnit.setXpathNamespaceContext` method. Any `XpathEngine` created by `XMLUnit.newXpathEngine` will automatically use the given context. Note that the JUnit 3 convenience methods use `XMLUnit.newXpathEngine` implicitly and will thus use the configured `NamespaceContext`.

6 DOM Tree Walking

Sometimes it is easier to test a piece of XML's validity by traversing the whole document node by node and test each node individually. Maybe there is no control XML to validate against or the expected value of an element's content has to be calculated. There may be several reasons.

XMLUnit supports this approach of testing via the `NodeTest` class. In order to use it, you need a DOM implementation that generates `Document` instances that implement the optional `org.w3c.traversal.DocumentTraversal` interface, which is not part of JAXP's standardized DOM support.

6.1 `DocumentTraversal`

As of the release of XMLUnit 1.1 the `Document` instances created by most parsers implement `DocumentTraversal`, this includes but is not limited to Apache Xerces, the parser shipping with Sun's JDK 5 and later or GNU JAXP. One notable exception is Apache Crimson, which also means the parser shipping with Sun's JDK 1.4 does not support traversal; you need to specify a different parser when using JDK 1.4 (see Section 2.4.1).

You can test whether your XML parser supports `DocumentTraversal` by invoking `org.w3c.dom.DOMImplementation`'s `hasFeature` method with the feature `"Traversal"`.

6.2 `NodeTest`

The `NodeTest` is instantiated with a piece of XML to traverse. It offers two `performTest` methods:

```
/**
 * Does this NodeTest pass using the specified NodeTester instance?
 * @param tester
 * @param singleNodeType note <code>Node.ATTRIBUTE_NODE</code> is not
 *  exposed by the DocumentTraversal node iterator unless the root node
 *  is itself an attribute - so a NodeTester that needs to test attributes
 *  should obtain those attributes from <code>Node.ELEMENT_NODE</code>
 *  nodes
 * @exception NodeTestException if test fails
 */
public void performTest(NodeTester tester, short singleNodeType);

/**
 * Does this NodeTest pass using the specified NodeTester instance?
 * @param tester
 * @param nodeTypes note <code>Node.ATTRIBUTE_NODE</code> is not
 *  exposed by the DocumentTraversal node iterator unless the root node
```

```
     *  is itself an attribute - so a NodeTester that needs to test attributes
     *  should obtain those attributes from <code>Node.ELEMENT_NODE</code>
     *  nodes instead
     * @exception NodeTestException if test fails
     */
    public void performTest(NodeTester tester, short[] nodeTypes);
```

`NodeTester` is the class testing each node and is described in the next section.

The second argument limits the tests on DOM `Nodes` of (a) specific type(s). `Node` types are specified via the `static` fields of the `Node` class. Any `Node` of a type not specified as the second argument to `performTest` will be ignored.

Unfortunately XML attributes are not exposed as `Nodes` during traversal. If you need access to attributes you must add `Node.ELEMENT_NODE` to the second argument of `performTest` and access the attributes from their parent `Element`.

Example 6.1 Accessing Attributes in a `NodeTest`

```
    ...
    NodeTest nt = new NodeTest(myXML);
    NodeTester tester = new MyNodeTester();
    nt.performTest(tester, Node.ELEMENT_NODE);
    ...

class MyNodeTester implements NodeTester {
    public void testNode(Node aNode, NodeTest test) {
        Element anElement = (Element) aNode;
        Attr attributeToTest = anElement.getAttributeNode(ATTRIBUTE_NAME);
        ...
    }
    ...
}
```

Any entities that appear as part of the `Document` are expanded before the traversal starts.

6.3 NodeTester

Implementations of the `NodeTester` interface are responsible for the actual test:

```
    /**
     * Validate a single Node
     * @param aNode
     * @param forTest
     * @exception NodeTestException if the node fails the test
     */
    void testNode(Node aNode, NodeTest forTest) throws NodeTestException ;

    /**
     * Validate that the Nodes passed one-by-one to the <code>testNode</code>
     * method were all the Nodes expected.
     * @param forTest
     * @exception NodeTestException if this instance was expecting more nodes
     */
    void noMoreNodes(NodeTest forTest) throws NodeTestException ;
```

`NodeTest` invokes `testNode` for each `Node` as soon as it is reached on the traversal. This means `NodeTester` "sees" the `Nodes` in the same order they appear within the tree.

`noMoreNodes` is invoked when the traversal is finished. It will also be invoked if the tree didn't contain any matched `Nodes` at all.

Implementations of `NodeTester` are expected to throw a `NodeTestException` if the current not doesn't match the test's expectations or more nodes have been expected when `noMoreNodes` is called.

XMLUnit ships with two implementations of `NodeTest` that are described in the following to sections.

6.3.1 AbstractNodeTester

`AbstractNodeTester` implements `testNode` by testing the passed in `Node` for its type and delegating to one of the more specific `test...` Methods it adds. By default the new `test...` methods all throw a `NodeTestException` because of an unexpected `Node`.

It further implements `noMoreNodes` with an empty method - i.e. it does nothing.

If you are only testing for specific types of `Node` it may be more convenient to subclass `AbstractNodeTester`. For example Example 6.1 could be re-written as:

Example 6.2 Accessing Attributes in a `NodeTest` - `AbstractNodeTester` version

```
...
NodeTest nt = new NodeTest(myXML);
NodeTester tester = new AbstractNodeTester() {
    public void testElement(Element element) throws NodeTestException {
        Attr attributeToTest = element.getAttributeNode(ATTRIBUTE_NAME);
        ...
    }
};
nt.performTest(tester, Node.ELEMENT_NODE);
...
```

Note that even though `AbstractNodeTester` contains a `testAttribute` method it will never be called by default and you still need to access attributes via their parent elements.

Note also that the root of the test is the document's root element, so any `Node`s preceding the document's root `Element` won't be visited either. For this reason the `testDocumentType`, `testEntity` and `testNotation` methods are probably never called either.

Finally, all entity references have been expanded before the traversal started. `EntityReference`s will have been replaced by their replacement text if it is available, which means `testEntityReference` will not be called for them either. Instead the replacement text will show up as (part of) a `Text` node or as `Element` node, depending on the entity's definition.

6.3.2 CountingNodeTester

`org.custommonkey.xmlunit.examples.CountingNodeTester` is a simple example `NodeTester` that asserts that a given number of `Node`s have been traversed. It will throw a `NodeTestException` when `noMoreNodes` is called before the expected number of `Node`s has been visited or the actual number of nodes exceeded the expected count.

6.4 JUnit 3.x Convenience Methods

`XMLAssert` and `XMLTestCase` contain overloads of `assertNodeTestPasses` methods.

The most general form of it expects you to create a `NodeTest` instance yourself and lets you specify whether you expect the test to fail or to pass.

The other two overloads create a `NodeTest` instance from either `String` or a SAX `InputSource` and are specialized for the case where you are only interested in a single `Node` type and expect the test to pass.

Neither method provides any control over the message of the `AssertionFailedError` in case of a failure.

6.5 Configuration Options

The only configurable option for `NodeTest` is the XML parser used if the piece of XML is not specified as a `Document` or `DocumentTraversal`. `NodeTest` will use the "control" parser that has been configured - see Section 2.4.1 for details.

It will also use the `EntityResolver` configured for the control parser if one has been set - see Section 2.4.2.

A Changes

A.1 Changes from XMLUnit 1.0 to 1.1

XMLUnit 1.1's main focus was to add two features that have been asked for repeatedly:

- Support for XML Namespaces in XPath processing
- Support for XML Schema validation.

In addition some JAXP features that have been added after the release of XMLUnit 1.0 are now supported - most notably XPath support - and all reported bugs and feature requests have been addressed.

A.1.1 Breaking Changes

- `XMLTestCase` is now abstract. You probably have never created instances of this class without subclassing it, but if you did, your code will now break. You will most likely want to look at the `XMLAssert` class.
- All methods that have been deprecated in XMLUnit 1.0 have been removed.
- All methods that had been declared to throw `TransformerConfigurationException` or `ParserConfigurationException` now no longer declare it. Exceptions of these types cannot be recovered from anyway, so XMLUnit will now wrap them in a `org.custommonkey.xmlunit.exceptions.ConfigurationException` which is an unchecked exception.

  This change doesn't have a big impact on your tests, but if you tried to catch these exceptions they will now bypass your catch blocks.
- A new type of `Difference` (`CHILD_NODE_NOT_FOUND_ID`) has been added. It will be raised for the excess children if the control element has more child nodes than the test element - or vice versa.

  Prior to XMLUnit 1.1 a `Difference` of either `ELEMENT_TAG_NAME_ID` or `NODE_TYPE_ID` would have been raised if the control element had more children. The excess children were compared to the very first child node of the test element. Excess children of the test element were not reported at all.
- The `schemaLocation` and `noNamespaceSchemaLocation` attributes of the XMLSchema-Instance Namespace are now treated in a different way from "normal" attributes. They will be flagged as new kinds of `Difference` that is recoverable.

  This means that two pieces of XML that were different in XMLUnit 1.0 because they differed in one of the two attributes will be similar in XMLUnit 1.1.
- When comparing two elements that differ on attributes the comparison is now symmetric.

  In XMLUnit 1.0 if an attribute was present on the test but not the control element this wasn't flagged as a `Difference`; in XMLUnit 1.1 it is.

  In most practical cases this doesn't cause any problems since the two elements either have a different number of attributes or there are attributes in the control element that are missing in the test element - so the pieces of XML have been flagged as different before as well. If you are using `DetailedDiff` this change may lead to more detected `Differences`, though.

A.1.2 New Features

- XMLUnit 1.0 shipped with rudimentary support for XML Schema validation (it worked with Apache Xerces-J but no other parsers). XMLUnit 1.1 supports Schema validation for any JAXP compliant XML parser (that supports Schema itself). You can also tell XMLUnit where to look for the XML Schema definitions. See Section 4.1.2 for details.
- XPath support has undergone significant changes, see Section 5 for more details. In particular XMLUnit will now use `javax.xml.xpath` if it is available (which also helps to avoid the buggy XSLTC version that is the default transformer engine in Java 5) and supports XML namespaces.
- Several new configuration options have been added, see Section 3.8.
  - Treat CDATA sections and Texts alike. Issue 1262148.
  - Ignore differences in Text whitespace. Issue 754812.
  - Ignore comments completely. Issue 707255.
  - Ignore the order of attributes.
- It is now possible to provide a custom `org.xml.sax.EntityResolver` for control and test parsers.
- It is now possible to provide a custom `javax.xml.transform.URIResolver` for transformations.
- New overloads have been added that allow `org.xml.sax.InputSource` to be used as a "piece of XML" in many classes.
- `Validator` will now use the custom `EntityResolver`configured for the "control" parser as a fallback.
- A new package `org.custommonkey.xmlunit.examples` has been added that showcases some of XMLUnit's abilities. It currently contains two classes:
  1. `MultiLevelElementNameAndTextQualifier` see Section 3.4.5 for a description.
  2. `XPathRegexAssert` that provides a JUnit 3.x like `assertXPathMatches` method to verify that the string-ified value of an XPath match matches a given regular expression (requires JDK 1.4 or above).

A.1.3 Important Bug Fixes

- `ElementNameAndAttributeQualifier` would throw an `NullPointerException` if the control piece of XML contained attributes that were missing in the test piece of XML. Issue 952920.
- `XMLTestCase.assertXMLNotEqual(String, Reader, Reader)` delegated to `assertXMLEqual` instead of `assertXMLNotEqual` internally, negating the assertion's logic. Issue 956372.
- `XMLTestCase.assertXMLIdentical(Diff, boolean)` delegated to `assertXMLEqual`, weakening the assertion.
- Under certain circumstances the reported XPath expressions for nodes that showed differences were wrong. XMLUnit could lose the root element or erroneously append an extra attribute name. Issues 1047364 and 1027863.
- `TolerantSaxParser`'s logic in `characters` was broken and could cause `StringIndexOutOfBoundsExceptions`. Issue 1150234.

A.2 Changes from XMLUnit 1.1 to 1.2

A.2.1 Breaking Changes

- If XMLUnit detects that it cannot match a certain node (i.e. it encounters a `CHILD_NODE_NOT_FOUND` kind of difference) the XPath for the "missing" node will be null. It used to be some random XPath of a different node.
- `XMLUnit.setIgnoreDiffBetweenTextAndCDATA` now also sets `DocumentBuilderFactory.setCoalescing`. This has been done so that whitespace differences can be resolved according to the corresponding flags even in the presence of CDATA sections. Issue 1903923.
- Two protected methods in `SimpleXPathEngine` (which you shouldn't extend anyway) have added XpathException to their throws list.

A.2.2 New Features

- The `SAXParserFactory` used by `Validator` can now be configured completely. Issue 1903928.
- A new class `org.custommonkey.xmlunit.jaxp13.Validator` can be used to validate schema definitions and schema instances using the `javax.xml.validation` package of JAXP 1.3. Depending on your JAXP implementation this may allow you to validate documents against schema definitions written in RELAX NG or other schema languages in addition to W3C XML Schema. See Section 4.4 for details.
- `DifferenceListener` can now "upgrade" recoverable differences to non-recoverable by returning `RETURN_UPGRADE_DIFFERENCE_NODES_DIFFERENT` in the `differenceFound` method. Issue 1854284.
- A new callback interface `MatchTracker` is now notified on successful matches of Nodes. For more details see Section 3.6. Issue 1860491.
- It is now possible to have more control over whether the parser expand entity references or not by using `XMLUnit.setExpandEntityReferences`, see Section 3.8.5. Issue 1877458.
- New examples have been added:
  - `RecursiveElementNameAndTextQualifier` - a more flexible `ElementQualifier` that fills the same need as `MultiLevelElementNameAndTextQualifier` See Section 3.4.4 for more details.
  - `CaseInsensitiveDifferenceListener` a - `DifferenceListener` that ignores case when comparing texts.
  - `FloatingPointTolerantDifferenceListener` a - `DifferenceListener` that tries to parse texts as floating point numbers and compares them using a configurable tolerance.

A.2.3 Important Bug Fixes

- If XMLUnit couldn't match nodes (i.e. it encountered a `CHILD_NODE_NOT_FOUND` kind of difference), the XPath expressions of the node details have been random. Issue 1860681.

A.3 Changes from XMLUnit 1.2 to 1.3

A.3.1 Breaking Changes

A.3.2 New Features

- If XMLUnit doesn't find a matching Element for a control Element, it will match it against the first unmatched test Element (if there is one) instead of creating a `CHILD_NODE_NOT_FOUND` Difference. There now is a new configuration option `compareUnmatched` in the `XMLUnit` class that can be used to turn off this behavior - as a result two `CHILD_NODE_NOT_FOUND` Differences (one for the unmatched control Element and one for an unmatched test Element) will be created instead of a single Difference comparing the two likely unrelated nodes. See Section 3.8.6. Issue 2758280.

A.3.3 Important Bug Fixes

- If XMLUnit couldn't match attributes (i.e. it encountered a `ATTR_NAME_NOT_FOUND_ID` kind of difference), the XPath expressions of the node details have been random. Issue 2386807.
- In some cases XMLUnit matched test nodes to multiple control nodes and then created a "missing child" difference for remaining test nodes even though they would have been valid targets for control node matches as well. Issue 2807167.

A.4 Changes from XMLUnit 1.3 to 1.4

A.4.1 Breaking Changes

A.4.2 New Features

- xsi:type attributes now have their value interpreted as a QName and will compare as identical if their namespace URI and local names match even if they use different prefixes. Issue 3602981

A.4.3 Important Bug Fixes

- `XMLTestCase`'s and `XMLAssert`'s `assertXpathsEqual` methods threw an exception when at least one XPath matched an attribute. Issue 377768.
- `FloatingPointTolerantDifferenceListener` expected numbers to differ by less than the given tolerance rather than "less or equal" than as the docs said. Issue 3593368