

Unit Testing in XSLT 2.0

Norman Walsh

Sun Microsystems, Inc.

Table of Contents

Using unit testing to improve stylesheet quality and documentation.

- **Background**
- **Motivation**
- **Survey of the framework**
- **Demo**

Background

- **The DocBook XSL (1.0) stylesheets for HTML and FO consist of roughly:**
 - > **2,500 match templates**
 - > **800 named templates**
 - > **250 modes**
 - > **115 modules**
 - > **62,000 lines**
 - > **485 parameters**
- **Development is a collaborative effort**
- **The document test suite contains 200 or so test documents**

Integration testing

- **200 test documents isn't sufficient**
- **Elements occur in many contexts**
- **Markup and stylesheet parameters interact**
- **Comparing documents is slow and tedious**
- **Comparing documents often yields false negatives**
- **Would 2,000 tests be sufficient? 20,000?**
- **Would they be practical?**

Can we do better?

- **Motivated by development of XSLT 2.0 stylesheets for DocBook**
- **Based on experience from development of XSLT 1.0 stylesheets for DocBook**
- **Attempt to build a testing framework**
- **Not a survey of available frameworks**

Why do you test?

- **Many modern software development paradigms stress early and frequent testing**
- **To make sure your code works**
- **To ease the burden of maintenance**
- **To make collaboration easier**

How can you test?

- **Integration testing: run the stylesheet over a suite of test documents and see what happens. Problems:**
 - > Comparison is difficult
 - > Coverage is incomplete
 - > Coverage is too coarse
- **Unit testing: test individual modules, in the XSLT case, individual templates and functions.**

What can you test?

In XSLT 2.0, we want to be able to test:

- **Individual named templates**
- **Individual user-defined functions**
- **Match templates in specific contexts**

A first attempt

Consider this function:

```
<xsl:function name="f:basename" as="xs:string">
  <xsl:param name="filename" as="xs:string"/>
  <xsl:value-of select="tokenize($filename, '/') [last]" />
</xsl:function>
```

And this test:

```
<xsl:if test="f:basename('/path/to/my/file.ext')
  = 'file.ext'">PASS</xsl:if>
```

Prints “PASS” (or nothing)

A first attempt (Continued)

But what about the tests that failed?

A second attempt

```
<xsl:if test="f:basename('/path/to/my/file.ext')  
            != 'file.ext'">FAIL</xsl:if>
```

Prints “FAIL” (or nothing)

A third attempt

```
<xsl:if test="f:basename( '/path/to/my/file.ext' )  
    != 'file.ext' ">f:basename( '/path/to/my/file.ext' '
```

Prints “f:basename('/path/to/my/file.ext') FAIL” (or nothing)

A fourth attempt

```
<xsl:choose>
  <xsl:when test="f:basename('/path/to/my/file.ext')
    <xsl:text>f:basename('/path/to/my/file.ext') PA
  </xsl:when>
  <xsl:otherwise>
    <xsl:text>f:basename('/path/to/my/file.ext') FA
  </xsl:otherwise>
</xsl:choose>
```

Ok. Now imagine several hundred tests like that, or several thousand.

What's wrong?

The tests are:

- Hard to write
- Hard to read
- Are more complex to write if the arguments or results are nodes or atomic values other than strings
- Are even more complex if they need to test in a specific context (e.g. a footnote inside a title).

Challenges

- **Make the tests easy to write**
- **Make the tests easy to read**
- **Support atomic values and nodes as input and output**
- **Support tests that require a specific context**

Meeting the challenges

- **Keep tests (and documentation) close to the actual code**
 - > **Not quite literate programming**
- **Minimize the amount of boilerplate in the tests**
- **Work within the constraints imposed by XSLT**

The test harness

```
<xsl:stylesheet>
```

```
<doc:function name="f:function-name">  
  ...documentation...  
</doc:function>
```

```
<u:unittests name="f:function-name">  
  ...tests...  
</u:unittests>
```

```
<xsl:function name="f:function-name">  
  ...code...  
</xsl:function>
```

The test harness (Continued)

```
<xsl:stylesheet>
```

Focus of this talk

```
<u:unittests function="f:node-id">
  <u:param name="persistent.generated.ids" select="(
    <u:test>
      <u:param><db:anchor id='id' /></u:param>
      <u:result>'id'</u:result>
    </u:test>
  </u:unittests>
```

The unittests element

The `unittests` element:

- Identifies the template or function being tested
- Groups a collection of related tests
- Specifies the value of any top-level parameters related to those tests

```
<u:unittests function="...">  
  <u:param>...</u:param>  
  <u:test>...</u:test>  
  <u:test>...</u:test>  
</u:unittests>
```

The test element

The test element:

- Describes an individual test
- Specifies arguments to the artifact being tested
- May specify variables
- Specifies the expected result

```
<u:test>  
  <u:param>...</u:param>  
  <u:result>...</u:result>  
</u:test>
```

Test parameters and variables

- The `u:variable` and `u:param` elements are semantic clones of the `xsl:param` element
- Each must have a `name` attribute
- The value of a variable or parameter is specified with either a `select` attribute or as content
- The `as` attribute can be used to specify the type

Why variables?

To select part of a structure and pass it as an argument:

```
<u:test>
  <u:variable name="mydoc">
    <db:book>
      <db:title>Some Title</db:title>
      <db:chapter>
        <db:title>Some Chapter Title</db:title>
        <db:para>My para.</db:para>
      </db:chapter>
    </db:book>
  </u:variable>
  <u:param select="$mydoc//db:para[1]" />
```

Why variables? (Continued)

```
<u:result>'R.1.2.2'</u:result>  
</u:test>
```

Test results

- Results are expressed with `result`
- Always expressed in element content
- Literal strings must be quoted

Establishing context

- **Parameters and variables are insufficient for testing named templates**
- **Named templates often expect to be called in a particular context**
- **The `context` element in a `test` specifies the context.**

Context example

```
<u:test>
  <u:context>
    <db:varname xml:id="varfoo">someVarName</db:varname>
  </u:context>
  <u:result>
    <span xmlns="http://www.w3.org/1999/xhtml"
      class="varname" id="varfoo">someVarName</span>
  </u:result>
</u:test>
```

How does context work?

1. The context is stored in a variable with an automatically generated, unique name.
2. Instead of calling the named template directly, `xs1:apply-templates` is performed on that variable in an automatically generated, unique mode.
3. Finally, a top-level template is generated which matches `node()` in the appropriate mode. The call to the named template is placed inside that template.

The practical effect of this three-step process is to change the context node to the specified context and call the named template in that context.

Testing match patterns

- **Establishing a context is a necessary step in testing a match pattern, but it is not a sufficient step.**
- **Simply calling apply-templates can not guarantee that the correct template will be evaluated because the stylesheet processor will select the template based on the match pattern, priority, mode, and input precedence.**
- **To overcome this obstacle, the framework creates a copy of the template that should be tested and gives it an automatically generated, unique name. It can then call the template directly using that name.**

Testing match patterns

This methodology has drawbacks:

- The framework selects a template based on the match pattern, mode, and priority. Sometimes that's insufficient, in which case the framework simply reports “indeterminate results”.
- The framework's exact, textual match is more restrictive than the real processor which would see no distinction between `priority="1"` and `priority="01"`.
- It is possible to force a template to be applied in a context that it can never match in the real stylesheet.

Using the test harness

- **Running the tests and constructing the report is a mostly-automated process**
- **In order to allow tests to specify different top-level stylesheet parameters, it is often necessary to transform the collection of unit tests into several stylesheets**
- **After the stylesheets have been written, each must be run**
- **Each stylesheet produces a fragment of XHTML that describes the test results**
- **These fragments are then assembled to produce the final report**

Running the tests

- Transform the stylesheet with `writetests.xml`
- Run each `testn.xml` stylesheet
 - > The input is irrelevant
- Assemble the results with `assembletests.xml`
- view the result in your browser of choice

Demo

- **In which norm tempts the fates...and Murphy.**

Getting the bits

- **It's all checked into the DocBook project at SourceForge:**
 - > `http://sourceforge.net/projects/docbook`
- **It's not in any distribution yet, but it's in CVS:**
 - > `/xsl2/tools/`