XSLT and XQuery

September 15, 2016

Improving stylesheets through the use of advanced features

Priscilla Walmsley, Datypic, Inc.

Class Outline

Introduction

This session

What we'll do in this session:

- Talk about specific capabilities of XSLT and where they are best applied
- Improve some XSLT together, discussing the pros and cons of different designs

Class objectives:

- Learn something you didn't know about XSLT's capabilities
- Use that knowledge to improve some of your existing XSLT
- Become inspired and undaunted by the prospect of refactoring your XSLT code

Why is XSLT often messy?

Much of the XSLT I see in my work is not optimal, usually because it was:

- Developed by people whose primary expertise is in other languages.
- Built up piecemeal over time.
- Developed under time constraints.
- Generated from a tool.
- Developed in XSLT 1.0 and never revised to take advantage of 2.0 (or 3.0) features.
- Copied from other existing XSLT that was itself messy.

Why "improve" XSLT?

("It's messy" is not usually a good enough reason.)

- Poor performance.
- Poor maintainability; small corrections lead to new bugs.
- Outdated output (e.g. old-school HTML that is not working well on all browsers/devices).
- Significant changes to the XSLT provide an opportunity for redesign.

XSLT requirements

An XSLT stylesheet *must* be:

- Correct
  - Obviously you need your XSLT to create correct output.
  - Using good XSLT techniques can make your code easier to debug and test, and therefore more likely to be correct.
- Robust
  - The XSLT needs to handle *all* possible input, not just the most common cases.
- Efficient
  - A correct and robust XSLT is useless if it is running too slow to meet user expectations.
  - Much more on performance tomorrow...

XSLT design goals

Ideally, an XSLT stylesheet should also be:

- Clear
  - Well-documented, succinct code is easier to debug, and much easier to maintain.
- Modular
  - Code that is repetitive is harder to understand and maintain.
  - Modular code that is broken into discrete, reusable components is much easier to test.
- Current
  - Using up-to-date features of XSLT can make your code more succinct and often perform better.
  - Using up-to-date output tags (e.g. HTML5) can improve readability of your output across browsers/devices.

Small Syntactic Conveniences

Concatenating strings

You can use the `string-join` function instead of looping through strings.

It is also possible to use `xsl:value-of` on a sequence of multiple items and specify a separator (default is a space).

Harder way

```
<xsl:for-each select="//name">
  <xsl:value-of select="."/>
  <xsl:if test="position() != last()">,  </xsl:if>
</xsl:for-each>
```

Easier way

```
<xsl:value-of select="string-join(//name,', ')"/>
```

Easier way #2

```
<xsl:value-of select="//name" separator=", "/>
```

Comparing with a sequence

The comparison operators return true if it is true for any pair of items in either operand

Harder way

```
<xsl:if test="dept = 'ACC' or dept = 'WMN' or dept = 'MEN'">
    ...
</xsl:if>
```

Easier way

```
<xsl:if test="dept = ('ACC','WMN','MEN')">
    ...
</xsl:if>
```

Conditional expressions

The XPath 2.0 conditional expression (`if-then-else`) is a more compact alternative to `xsl:choose`.

Hard way

```
<xsl:variable name="heading">
   <xsl:choose>
       <xsl:when test="desc">
           <xsl:value-of select="substring(desc,1,32)"/>
      </xsl:when>
      <xsl:otherwise>
           <xsl:value-of select="name"/>
      </xsl:otherwise>
   </xsl:choose>
</xsl:variable>
```

Easier way

```
<xsl:variable name="heading"
select="if (desc) then substring(desc,1,32) else name"/>
```

Using XPath steps instead of `xsl:for-each`

The last step in a path can return an atomic value, not just a node.

Hard way

```
<xsl:for-each select=".//name">
  <xsl:value-of select="substring(.,1,32)"/>
</xsl:for-each>
```

Easier way

```
<xsl:value-of select=".//name/substring(.,1,32)"/>
```

Using range expressions

<*integer*> to <*integer*> syntax is useful for iterating a specific number of times

Hard way

recursive templates that add a count on each call, or iterating through fake nodes

Easier way

```
<xsl:for-each select="1 to 5">
   ...
</xsl:for-each>
```

For expressions

- Simplified version of XQuery FLWOR expressions
- only one `for` clause, no `let` or `where`
- More compact alternative to `xsl:for-each`

Hard way

```
<xsl:for-each select=".//product/name">
  <xsl:variable name="name-trunc" select="substring(.,1,9)"/>
  <xsl:value-of select="replace($name-trunc,'x','y')"/>
</xsl:for-each>
```

Easier way

```
<xsl:value-of select="for $name-trunc in .//product/substring(name,1,9)
                      return replace($name-trunc,'x','y')"/>
```

Quantified expressions

- To determine whether some or all items in a sequence meet a criteria
- Evaluates to a boolean value
- Use `some` or `every` with `satisfies`

```
some $dept in //product/@dept
satisfies ($dept = "ACC")
```

```
every $dept in //product/@dept
satisfies ($dept = "ACC")
```

The `except` operator

- Everything in the first sequence except what's in the second sequence
- Evaluates to a boolean value

Hard way

```
<xsl:apply-templates select="child1"/>
  <div>-------</div>
  <xsl:apply-templates select="child2"/>
  <xsl:apply-templates select="child3"/>
  <xsl:apply-templates select="child4"/>
```

Easier way

```
<xsl:apply-templates select="child1"/>
  <div>-------</div>
  <xsl:apply-templates select="* except child1"/>
```

Regular Expression Matching

Recognizing textual patterns

- XSLT 2.0 has excellent regular expression support
- `xsl:analyze-string` element splits string into matching and non-matching parts, based on a regex
  - `xsl:matching-substring` child specifies what to do with matching parts
  - `xsl:non-matching-substring` child specifies what to do with non-matching parts

`xsl:analyze-string` Example

```
can be reached at 231/555-1212 or...
```

⇓

```
<xsl:function name="my:markUpPhone">
  <xsl:param name="theText"/>
  <xsl:analyze-string select="$theText" regex="[0-9]{{3}}/[0-9]{{3}}-[0-9]{{4}}">
    <xsl:matching-substring>
      <phone>
        <xsl:value-of select="."/>
      </phone>
    </xsl:matching-substring>
    <xsl:non-matching-substring>
      <xsl:value-of select="."/>
    </xsl:non-matching-substring>
  </xsl:analyze-string>
</xsl:function>
```

⇓

```
can be reached at <phone>231/555-1212</phone> or...
```

The `regex-group` function

```
can be reached at 231/555-1212 or...
```

⇓

```
<xsl:function name="my:markUpPhone">
  <xsl:param name="theText"/>
  <xsl:analyze-string select="$theText" regex="([0-9]{{3}})/([0-9]{{3}}-[0-9]{{4}})">
    <xsl:matching-substring>
      <phone>
        <areaCode><xsl:value-of select="regex-group(1)"/></areaCode>
        <number><xsl:value-of select="regex-group(2)"/></number>
      </phone>
    </xsl:matching-substring>
    <xsl:non-matching-substring>
      <xsl:value-of select="."/>
    </xsl:non-matching-substring>
  </xsl:analyze-string>
</xsl:function>
```

⇓

```
can be reached at <phone><areaCode>231</areaCode><number>555-1212</number></phone> or...
```

The `matches` function

- whether a string matches a regular expression
- uses the XML Schema regex syntax (similar to Perl)
- optional third "flags" argument allows for interpretation of regular expression
  - case sensitivity
  - multi-line mode
  - whitespace sensitivity

```
matches("query", "^qu")
```
⇒
```
true
```

The `tokenize` function

- delimiter specified as a regular expression
- returns a sequence of strings

```
tokenize("a b c", "\s")
```
⇒
```
("a", "b", "c")
```

```
tokenize("2006-12-25T12:15:00", "[\-T:]")
```
⇒
```
("2006","12","25","12","15","00")
```

The `replace` function

Arguments are:

- the string to be manipulated
- the regular expression
- the replacement string

```
replace("query", "r", "as")
```
⇒
```
queasy
```

```
replace("query", "qu", "quack")
```
⇒
```
quackery
```

```
replace("query", "[ry]", "l")
```
⇒
```
quell
```

```
replace("query", "[ry]+", "l")
```
⇒
```
quel
```

Replacing with subexpressions

Use `$1`, `$2`, etc. in replacement string to insert strings that matched parenthesized subexpressions

```
replace("Chap 2...Chap 3...Chap 4...",
 "Chap (\d)", "Sec $1.0")
```
⇒
```
Sec 2.0...Sec 3.0...Sec 4.0...
```

```
replace("2006-10-18", "\d{2}(\d{2})-(\d{2})-
(\d{2})", "$2/$3/$1")
```
⇒
```
10/18/06
```

Exercise 2: Pattern Matching.

Modularity and Reuse 25

Modularization techniques 26

- Shared templates and functions
- Attribute sets
- Consolidating modes
- Included and imported stylesheets
- `xsl:apply-imports` and `xsl:next-match`

Shared templates 27

Similar template rules can often be consolidated.

- `xsl:copy` is useful in templates that match many element names

```
<xsl:template match="b|i|u|br">
  <xsl:copy>
   <xsl:apply-templates/>
  </xsl:copy>
</xsl:template>
```

Named templates are also useful for modularity

```
<xsl:template match="person-name">
  <xsl:call-template name="format-name"/>
</xsl:template>
<!-- ... -->
<xsl:template name="format-name">
  <xsl:value-of select="concat(lastname,', ',firstname)"/>
</xsl:template>
```

Shared functions 28

function definition

```
<xsl:function name="my:truncate"
     as="xs:string">                                  ←  return type
  <xsl:param name="string" as="xs:string"/>           ←  parameters
  <xsl:param name="length" as="xs:integer"/>          ←  parameters
  <xsl:sequence select="substring($string,1,
$length)"/>
</xsl:function>
```

function call

```
<xsl:value-of select="my:truncate('abcde', 3)"/>      ←  returns "abc"
```

Attribute sets

Attribute sets can help organize style information

```
<xsl:template match="h2">
 <fo:block xsl:use-attribute-sets="heading" font-size="18pt">
   <xsl:apply-templates/>
 </fo:block>
</xsl:template>
<xsl:template match="h3">
 <fo:block xsl:use-attribute-sets="heading" font-size="16pt">
   <xsl:apply-templates/>
 </fo:block>
</xsl:template>
<xsl:attribute-set name="heading">
  <xsl:attribute name="background-color">#FFFF99</xsl:attribute>
  <xsl:attribute name="margin-bottom">12px</xsl:attribute>
  <xsl:attribute name="keep-with-next">always</xsl:attribute>
  <xsl:attribute name="padding-before">24pt</xsl:attribute>
</xsl:attribute-set>
```

Modes

- Modes are used to process the same element different ways at different times
- Useful for making multiple passes at a document

```
<xsl:template match="document">
  <xsl:apply-templates mode="toc"/>
  <xsl:apply-templates mode="mainBody"/>
</xsl:template>
...
<xsl:template match="section" mode="toc">
  <!-- display the name of the section, possibly indented -->
</xsl:template>
<xsl:template match="section" mode="mainBody">
  <!-- display the section itself -->
</xsl:template>
```

Consolidating modes

Multiple modes can be specified for a template

```
<xsl:template match="section" mode="toc mainBody #default">
  <!-- do something -->
  <xsl:apply-templates mode="#current"/>
</xsl:template>
```

`#all` keyword can be used to match all modes

```
<xsl:template match="section" mode="#all">
  <!-- do something -->
</xsl:template>
```

Including and importing other stylesheets

Allows for modular stylesheet design

`xsl:include`

- Just like cutting and pasting - resulting stylesheet is a union of all the included XSLTs
- No duplicate global variables, named templates, functions allowed

```
<xsl:stylesheet version="1.0" xmlns:xsl=
  "http://www.w3.org/1999/XSL/Transform">
  <xsl:include href="transform2.xsl"/>
  <!-- ... -->
</xsl:stylesheet>
```

`xsl:import`

- Similar, but when templates conflict, the importing stylesheet has priority
- Duplicates *are* allowed (and overridden)
- Useful for customizing large and complex XSLTs, but also generally useful to increase code reuse more flexibly

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:import href="transform2.xsl"/>
  <!-- ... -->
</xsl:stylesheet>
```

`xsl:import` conflict resolution

If there are multiple named templates, functions or global variables with the same name:

- Importing stylesheet always has precedence over imported stylesheet
- Import order is significant: later imports have precedence over earlier ones

If there are multiple "match" templates:

- *All* importing templates always have priority over all imported ones

Importing example

t1.xsl

```
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:import href="t2.xsl"/>
<xsl:template match="firstname" priority="5">...   ←  1st
<xsl:template match="name/firstname">...          ←  2nd
<xsl:template match="*">...                       ←  3rd
</xsl:stylesheet>
```

imports

t2.xsl

```
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="firstname" priority="500">... ←  4th
<xsl:template match="name/firstname">...          ←  5th
<xsl:template match="*">...                       ←  6th
</xsl:stylesheet>
```

`xsl:apply-imports` and `xsl:next-match`

- Used to call an overridden template
  - applies imported templates to the *current* node (not the children)
- Often used to create new preceding or wrapping elements and then process the elements normally
- `xsl:apply-imports` (available in 1.0)
  - only looks at imported templates
- `xsl:next-match` (2.0 only)
  - looks at all templates of lower precedence

`xsl:apply-imports`

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="example">
    <pre><xsl:value-of select="."/></pre>
  </xsl:template>
</xsl:stylesheet>
```

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:import href="t2.xsl"/>
  <xsl:template match="example">
    <a name="xx"/>
    <div style="border: solid red">
      <xsl:apply-imports/>
    </div>
  </xsl:template>
</xsl:stylesheet>
```

```
<div style="border: solid red">
  <pre>...</pre>
</div>
```

xsl:next-match

```
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="example">
    <example-wrap>
      <xsl:next-match/>
    </example-wrap>
  </xsl:template>

  <xsl:template match="*">
    <xsl:copy>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

```
<example-wrap>
  <example>...</example>
</example-wrap>
```

Exercise 3: Modularity and reuse.

Using Types

Specifying types with the `as` attribute

You can use the `as` attribute to indicate the required sequence type of an expression, or the return type of a function or template.

Benefits:

- Significantly helps with debugging.
- Improves error messages when the wrong values are passed.
- Serves as documentation of what is expected/handled.
- Minimizes the differences between validated and unvalidated input.

You can use an `as` attribute on:

- `xsl:variable` or `xsl:param` to indicate the type of that variable or parameter.
- `xsl:template` or `xsl:function` to indicate the return type of that template or function.
- `xsl:sequence` to indicate the type of the sequence.
- `xsl:with-param` to indicate the type of a value passed to a template.

```
<xsl:function name="my:name2ndDigit" as="xs:string?">
  <xsl:param name="theName" as="element()?"/>
  <xsl:value-of select="substring($theName/firstname,2,1)"/>
</xsl:function>
```

For more detailed instructions for adding types, see my article entitled *Using types and schemas to improve your XSLT 2.0 stylesheets* at http://datypic.com/services/xslt/xslt-article2.html

Sequence type syntax

Common XML Schema data types

Data type name	Description	Example(s)
`xs:string`	Any text string	abc, this is a string
`xs:integer`	An integer of any size	1, 2
`xs:decimal`	A decimal number	1.2, 5.0
`xs:double`	A double-precision floating point number	1.2, 5.0
`xs:date`	A date, in `YYYY-MM-DD` format	2009-12-25
`xs:time`	A time, in `HH:MM:SS` format	12:05:04
`xs:boolean`	A true/false value	true, false
`xs:anyAtomicType`	A value of any of the simple types	a string, 123, false, 2009-12-25

Sequence types representing XML nodes

Sequence type	Description
`element()`	Any element
`element(book)`	Any element named `book`
`attribute()`	Any attribute
`attribute(isbn)`	Any attribute named `isbn`
`text()`	Any text node
`node()`	A node of any kind (element, attribute, text node, etc.)
`item()`	Either a node of any kind or an atomic value of any kind (e.g. a string, integer, etc.)

Using occurrence indicators

Occurrence indicator	Description
`*`	Zero to many
`?`	Zero to one
`+`	One to many
(no occurrence indicator)	One and only one

Exercise 4: Adding types.

Grouping

Grouping

- xsl:for-each-group element allows you to iterate through groups
  - select attribute identifies items to group
  - grouping attribute specifies the grouping key

Grouping attribute	Meaning
group-by	groups all items with the same key value together
group-adjacent	groups adjacent items with the same key value together
group-starting-with	creates a group of items starting with the specified element
group-ending-with	creates a group of items ending with the specified element

- Two functions can be used within for-each-group:
  - current-group() returns members of current group
  - current-grouping-key() returns the current grouping key

Grouping by value

```
<catalog>
  <product dept="WMN">...</product>
  <product dept="ACC">...</product>
  <product dept="ACC">...</product>
  <product dept="MEN">...</product>
</catalog>
```

⇓

```
<xsl:template match="/">
  <RESULTS>
    <xsl:for-each-group select="catalog/product" group-by="@dept">
      <xsl:sort select="current-grouping-key()"/>
      <DEPT name="{current-grouping-key()}" prodCount="{count(current-group())}"/>
    </xsl:for-each-group>
  </RESULTS>
</xsl:template>
```

⇓

```
<RESULTS>
  <DEPT name="ACC" prodCount="2"/>
  <DEPT name="MEN" prodCount="1"/>
  <DEPT name="WMN" prodCount="1"/>
</RESULTS>
```

Grouping by starting element

```
<body>
  <h1>Chapter 1</h1>
  <h2>Section 1.1</h2>
  <p>In this section...</p>
  <p>More text</p>
  <h2>Section 1.2</h2>
  <p>In this section...</p>
</body>
```

⇓

```
<xsl:template match="/">
  <xsl:for-each-group select="body/*" group-starting-with="h1">
    <section level="1">
      <xsl:for-each-group select="current-group()" group-starting-with="h2">
        <xsl:choose>
          <xsl:when test="current-group()[self::h2]">
            <section level="2">
              <xsl:apply-templates select="current-group()"/>
            </section>
          </xsl:when>
          <xsl:otherwise>
            <xsl:apply-templates select="current-group()"/>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:for-each-group>
    </section>
  </xsl:for-each-group>
</xsl:template>
```

⇓

```
<section level="1">
  <heading>Chapter 1</heading>
  <section level="2">
    <heading>Section 1.1</heading>
    <p>In this section...</p>
    <p>More text</p>
  </section>
  <section level="2">
    <heading>Section 1.2</heading>
    <p>In this section...</p>
  </section>
</section>
```

Grouping adjacent items

```
<body>
  <p>The following...:</p>
  <p>1. Open the file.</p>
  <p>2. Change it to...</p>
  <p>3. Save the file.</p>
  <p>As you can see...</p>
</body>
```

⇓

```
<xsl:template match="body">
  <body>
    <xsl:for-each-group select="*" group-adjacent="my:is-a-list-item(.)">
      <xsl:choose>
        <xsl:when test="current-grouping-key() = true()">
          <ul>
            <xsl:for-each select="current-group()">
              <li><xsl:apply-templates/></li>
            </xsl:for-each>
          </ul>
        </xsl:when>
        <xsl:otherwise>
          <xsl:copy-of select="."/>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:for-each-group>
  </body>
</xsl:template>
<xsl:template match="text()">
  <xsl:choose>
    <xsl:when test="my:is-a-list-item(parent::p)
                    and my:is-a-list-item(.)
                    and (. is parent::p/node()[1])">
      <xsl:value-of select="replace(.,'^\s*\d+\.\s*','')"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:copy-of select="."/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
<xsl:function name="my:is-a-list-item">
  <xsl:param name="node"/>
  <xsl:sequence select="matches($node,'^\s*\d+\.')"/>
</xsl:function>
```

⇓

```
<body>
  <p>The following...:</p>
  <ol>
    <li>Open the file.</li>
    <li>Change it to...</li>
    <li>Save the file.</li>
  </ol>
  <p>As you can see...</p>
</body>
```

Exercise 5: Grouping.

Conclusions

Using advanced features of XSLT can:

- simplify your XSLT code
- make it easier to maintain
- make it easier to debug
- make it perform faster
- allow it to be more flexible
- produce better output

Questions? Let's discuss.

...and you can contact me at `pwalmsley@datypic.com`

Slides are at http://www.datypic.com/services/xslt/improving.pdf.

You might also be interested in my Refactoring XSLT talk from last year at http://www.datypic.com/services/xslt/refactoring.pdf.