XSLT and XQuery

September 14, 2023

Developing High-Quality XSLT Stylesheets

Priscilla Walmsley, Datypic, Inc.

Class Outline

Introduction

Why Is XSLT Often "Less than Optimal"?

Much of the XSLT I see in my work is messy, usually because it was:

- developed by people whose primary expertise is in other languages or other fields
- 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

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.

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.

XSLT Design Goals (Optional)

It may optionally be an objective to make your XSLT:

- Customizable
  - If you are developing XSLTs that are intended to be used in various environments, or for various output devices, you should take steps to make your code easier to customize.
- Interoperable
  - Some XSLTs need to be able to run by multiple processors or versions, e.g.:
    - Xalan and Saxon
    - Saxon-HE and Saxon-EE
    - Saxon 10 and Saxon 11
  - Some XSLTs need to be able to handle both validated and non-validated input.

Clarity

Improving Clarity

- Indent your code!
- Use good naming practices:
  - Especially for names of functions, named templates, parameters, variables and stylesheet files, but also for modes, keys, decimal formats, outputs, attribute sets, etc.
  - Use descriptive names, avoid `string` or `arg1`.
  - Use consistent names and order for parameters across functions and named templates.
  - Use consistent naming standards for upper/lower/camel-case, and/or word separator characters.
- Document your code. But remember, documentation doesn't take the place of writing clear code.

Cleaning Up

You can start with basic refactoring of your XSLT code. Obvious areas to clean up:

- Unused code (functions, templates, variables, etc.)
- Unnecessarily verbose code
- Repetitive code

Why?

- Helps meet the goal of clarity.
- Often improves correctness by exposing previously unrecognized bugs.
- Often also improves performance.

This topic was covered in detail in previous years' courses, please see http://www.datypic.com/services/xslt/refactoring.pdf

Exercise 1: Improving Clarity.

Modularity

Modularization Techniques

- Smaller "push"-style template rules
- Shared template rules
- Named templates and user-defined functions
- Attribute sets
- Consolidating modes
- Included and imported stylesheets
- `xsl:apply-imports` and `xsl:next-match`

General Style: Pull Approach

- The stylesheet "pulls" the information from the input document using instructions.
- Also known as "stylesheet-driven" or "program-driven".
- Uses hardcoded paths to extract data from specific locations in the source document.
- Dependent upon a predictable structure of the input file.
- "Get the `books` element, now do *x* for each its `book` children, now do *y* for each `book`'s `title` child, etc."

```
<xsl:template match="books">
  <table>
   <xsl:for-each select="book">
    <tr>
      <td><xsl:value-of select="title"/></td>
      <td><xsl:value-of select="author/person/last"/></td>
    </tr>
   </xsl:for-each>
  </table>
</xsl:template>
```

General Style: Push Approach

- Template rules specify what to do when you encounter an `x` element.
- This is also known as "event-driven" or "input-driven".
- Used when the structure of the input file is *not* known, or is changeable, or is highly recursive.
- "Every time I happen across a `book` element, put in a table row. Every time I happen across a `title` element, put in a table cell."

```
<xsl:template match="books">
 <table><xsl:apply-templates/></table>
</xsl:template>
<xsl:template match="book">
 <tr><xsl:apply-templates/></tr>
</xsl:template>
<xsl:template match="title|last">
 <td><xsl:value-of select="."/></td>
</xsl:template>
```

The push approach is essential for mixed content. For more predictably structured content, it also has some benefits:

- It breaks the code into discrete units that are easier to understand. You can easily see, for example, that a `book` becomes a table row.
- It eliminates repetitiveness when an element can appear in more than one place.
- Small units are easier to customize than a large monolithic template rule.

Exercise 2: Pull to push.

Shared Templates

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>
```

User-Defined Functions

```
<xsl:template match="person-name">
  <xsl:value-of select="my:format-name(firstname,
 lastname)"/>
</xsl:template>
<!-- ... -->
<xsl:function name="my:format-name"
      as="xs:string">                                ← return type
  <xsl:param name="firstname" as="xs:string"/>       ← parameters
  <xsl:param name="lastname" as="xs:string"/>        ← parameters
  <xsl:sequence select="concat($lastname,', ',
$firstname)"/>
</xsl:function>
```

Named Templates vs. Functions

- The syntax to call a function is usually much more compact.
- Functions can be called from places where only a simple XPath expression or pattern is allowed, e.g.:
  - the `match` attribute of `xsl:template`

    ```
    <xsl:template match="*[my:is-heading(.)]">...</xsl:template>
    ```

  - the `select` attribute of `xsl:sort`

    ```
    <xsl:sort select="my:title-sort-key(.)"/>
    ```

  - the `select` or `group-by` attributes of `xsl:for-each-group`
- Named templates take the context from where they are called, and functions do not.
- Parameters to functions cannot be optional or have default values.

Multiple Signatures

- For a good balance between ease of use and flexibility, consider multiple signatures that allow parameters to be optionally specified.
- Be consistent in the order and meaning of parameters.
- It may be appropriate for the *n*-parameter version to call the *n+1*-parameter version.

```
<xsl:function name="functx:substring-before-match" as="xs:string?">
  <xsl:param name="arg" as="xs:string?"/>
  <xsl:param name="pattern" as="xs:string"/>
  <xsl:sequence select="functx:substring-before-match($arg,$pattern,'')"/>
</xsl:function>

<xsl:function name="functx:substring-before-match" as="xs:string?">
  <xsl:param name="arg" as="xs:string?"/>
  <xsl:param name="pattern" as="xs:string"/>
  <xsl:param name="flags" as="xs:string"/>
  <xsl:sequence select="tokenize($arg,$pattern,$flags)[1]"/>
</xsl:function>
```

Exercise 3: Shared Templates and Functions.

Attribute Sets

Attribute sets can help organize style information.

They are especially useful when generating something like XSL-FO, which requires very repetitive attributes.

```
<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.

They are often used 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, as a TOC entry -->
</xsl:template>
<xsl:template match="section" mode="mainBody">
  <!-- display the section itself -->
</xsl:template>
```

Consolidating Modes

Multiple modes can be specified for a template rule.

```
<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

Reuse of entire stylesheets can be accomplished with `xsl:include` or `xsl:import`.

`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 to `xsl:include`, but when template rules 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>
```

Template Rule Priority

When multiple patterns match (regardless of `xsl:import`):

- More specific patterns have higher priority.
- You can use a `priority` attribute to manually set priority.
- If two have the same priority, it either raises an error or the last one in the stylesheet is used.

```
<xsl:template match="firstname" priority="5">...</ ←   5
xsl:template>
<xsl:template match="name/firstname">...</          ←   0.5
xsl:template>
<xsl:template match="firstname[. = 'John']">...</   ←   0.5
xsl:template>
<xsl:template match="firstname">...</xsl:template> ←   0
<xsl:template match="*">...</xsl:template>         ←   -0.5
```

`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 template rules always have priority over all imported ones.

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"                   ←  4th
 priority="500">...
  <xsl:template match="name/firstname">...          ←  5th
  <xsl:template match="*">...                       ←  6th
</xsl:stylesheet>
```

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

- Used to invoke an overridden or less specific template rule
  - applies imported template rules 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`
  - only looks at imported template rules
- `xsl:next-match`
  - looks at all template rules 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>
```

Output:

```
<a name="xx"/>
<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>
```

Output:

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

Exercise 4: Imports and Includes.

Robustness

Improving Robustness

- Consider what will happen for each expression (or function) if cardinalities are different from expected (inserting the `as` attributes often forces you to think about this):
  - What if a value is the empty sequence? Should the expression return the empty sequence, or a default value, or raise an error?
  - What if a value is more than one item? For example, the XSLT is expecting only one `author` per `book`, but a `book` in the input has multiple `author`s.
- Consider what will happen if atomic values are different from expected:
  - numbers: do negative values work?
  - strings: does a zero-length string work?
  - dates/times: do values with or without time zones work?
  - names: do names with or without namespaces work?
  - `xs:anyAtomicType`: will it really work on values of *any* type?

Adding Types

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: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

Parameter Types in Named Templates and Functions

Be liberal in the arguments you accept.

- Specify a more general parameter type for maximum flexibility.
  - However, you can take this too far -- you should not use `item()*` for every parameter!
- Allow a variety of cardinalities (including the empty sequence).
  - Most built-in functions accept the empty sequence for the "main" argument but do not accept it for arguments that control how the function operates.
  - For example, the `substring` function accepts the empty sequence for the first argument and returns a zero-length string in that case.
  - This makes it easier on the code calling the function.

Exercise 5: Adding types. For a detailed explanation of exercise 5, please see my articles on adding types to XSLT stylesheets at http://datypic.com/services/xslt/xslt-article2.html.

Respecting Narrative Content

Many existing XSLT stylesheets treat mixed content poorly without even knowing it!

- Use `xsl:strip-space` sparingly. It can remove significant whitespace.
- Don't indent output. It can introduce unwanted whitespace.
- Don't use `text()` on a mixed content element. It will ignore content in child elements.
- Don't use `<xsl:value-of select="."/>` on a mixed content element unless that's what you really mean to do. It will ignore any child tags that might be, for example, applying styles.
- Use a push-style stylesheet rather than pull-style to handle variations in content.

Exercise 6: Respecting narrative content.

Generating the Most Robust Output

- Use current tags, e.g. for HTML, stick to HTML5 or HTML4 Strict.
- Don't abuse HTML elements.
  - Using empty `p` elements for vertical space.
  - Using `br` elements to create "paragraphs".
  - Using `hr` for borders.
  - Using `blockquote` for indenting.
- Don't generate unnecessarily large output. Remove:
  - Excessive whitespace because of indented output.
  - Excessive whitespace copied from input documents (for non `strip-space` elements).
  - Excessive whitespace because `xsl:text` was not used to limit excess whitespace from the XSLT.
  - Excessive whitespace at the beginning/end of HTML block elements, or just before block elements.
  - Unnecessary `div` elements (e.g. with no `class` or `id` attribute, and only containing another `div`).
  - Style information (especially when repetitive) that should be in a CSS class.
  - Class attributes that refer to CSS classes that don't exist.
- Never, ever use `disable-output-escaping` or character maps to create elements.

Exercise 7: Improving HTML output.

Error Handling

Test for error conditions before evaluating expression, including:

- Use `castable as` to determine if, for example, something can be converted to a number before you attempt to perform arithmetic on it.
- Use the `number` function to convert a value that may or may not be a valid number.
- Use `doc-available` and `unparsed-text-available` to test for the existence of a file before you attempt to open it.
- In 3.0, use `xsl:try` and `xsl:catch` to recover gracefully from error conditions.

Use the `error` function to provide better feedback on errors:

```
<xsl:function name="functx:mmddyyyy-to-date" as="xs:date?">
  <xsl:param name="dateString" as="xs:string?"/>
  <xsl:sequence select="
   if (empty($dateString))
   then ()
   else if (not(matches($dateString,
                        '^\D*(\d{2})\D*(\d{2})\D*(\d{4})\D*$')))
   then error(xs:QName('functx:Invalid_Date_Format'),
                  concat('Value ',$dateString,' does not match MMDDYYY'))
   else xs:date(replace($dateString,
                        '^\D*(\d{2})\D*(\d{2})\D*(\d{4})\D*$',
                        '$3-$1-$2'))
 "/>
</xsl:function>
```

Other Improvements

Improving Performance

Benchmarking is the best way to find out why your XSLT is running slowly, but some general suggestions are:

- Use keys.
- Avoid the "//" operator.
- Avoiding reevaluating the same expression (create a variable).
- Avoid counting items to see if the exist, e.g. use `exists($x)` instead of `count($x) > 0`.
- Break down a complex problem into multiple passes.
- Decrease the size of the XSLT (if the size decrease is dramatic).
- Decrease the size of the output (if the size decrease is dramatic).

Improving Customizability

- Many, smaller templates are easier to customize via `xsl:import` than huge monolithic ones.
- Global variables should be used in stylesheets for customizable values that can be overridden on import.
- Stylesheet parameters should be used for values that need to be provided at runtime.
- `xsl:next-match` is a great way to make some customizations without having to repeat a lot of code.
- For HTML output, all styling should be moved to CSS for easier customization.
- Samples should be provided for developers who wish to customize the XSLTs.

Graceful Versioning

- Consider backward compatibility when making changes, especially to stylesheets that may be imported/included by others.
  - Avoid changing a function to:
    - be more strict in what it accepts
    - be less strict in what it returns
    - change the behavior in unexpected ways
  - Consider instead:
    - a different function name
    - a different arity for the same function name
    - a different namespace?
- Document changes carefully.

Improving Interoperability

If interoperability is a goal:

- Avoid processor-specific extension functions, or
- Use `function-available` and `element-available` to test support for extension functions.
- Avoid advanced features that are not available in all versions of a processor that need to be supported.
- Handle both validated and unvalidated input.
  - Cast atomic values to the expected type.
  - Normalize whitespace if appropriate.
  - Do not assume default/fixed attributes are present.

Thank you

Questions? Let's discuss.

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

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