summer school

XSLT and XQuery

September 21, 2017

Refactoring XSLT

Priscilla Walmsley, Datypic, Inc.

Class Outline

Introduction 2

Obligatory Wikipedia Quote 3

Code refactoring is the process of restructuring existing computer code - changing the factoring - without changing its external behavior. Refactoring improves nonfunctional attributes of the software. Advantages include improved *code readability* and *reduced complexity*; these can improve *source code maintainability* and create a more expressive internal architecture or object model to *improve extensibility*.

Typically, refactoring applies a series of standardised basic *micro-refactorings*, each of which is (usually) a tiny change in a computer program's source code that either preserves the behaviour of the software, or at least does not modify its conformance to functional requirements. Many development environments provide automated support for performing the mechanical aspects of these basic refactorings. If done extremely well, code refactoring may also *resolve hidden, dormant, or undiscovered computer bugs* or vulnerabilities in the system by simplifying the underlying logic and eliminating unnecessary levels of complexity. If done poorly it may fail the requirement that external functionality not be changed, and/or introduce new bugs.

--Code refactoring. (2015, July 30). In *Wikipedia, The Free Encyclopedia*. Retrieved 23:17, September 16, 2015, from https://en.wikipedia.org/w/index.php?title=Code_refactoring&oldid=673771880.

Why Is XSLT Often Messy? 4

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.

Why Refactor XSLT? 5

("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.

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 9.7 and Saxon 9.8
  - Some XSLTs need to be able to handle both validated and non-validated input.

Cleaning Up

Cleaning Up

What to clean up:

- Unused code
- Unnecessarily verbose code
- Repetitive code

Why?

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

Unused Code

- Functions and named templates that are never called.
- Variables and parameters that are never used because:
  - They are never referenced.
  - A different variable with the same name is created in scope.
- Template rules that are never used because:
  - There is a template rule with a higher priority that will always be used.
  - There is a template rule with the same priority that will always be used (usually a warning at runtime).
  - `xsl:apply-templates` is never used for that element in the input document.
  - `xsl:apply-templates` is never used with that mode.
  - It is always overridden by an import.
  - The element/pattern will never exist in the input document.
- `xsl:when` elements that are ignored because:
  - They are redundant with a previous `xsl:when`.
  - Their condition will never return true.
- Entire stylesheets that are never used but are imported/included anyway.
- Blocks of commented-out code.

Exercise 1: Removing unused code.

Unnecessarily Verbose Code

- Unnecessary use of `xsl:element` or `xsl:attribute`.
- `xsl:choose` with:
  - Only one `xsl:when` and no `xsl:otherwise` (can be an `xsl:if` instead).
  - Empty `xsl:otherwise` that can be deleted.
  - `xsl:otherwise` that contains only an `xsl:choose` (`xsl:otherwise/xsl:choose` can be skipped).
  - Simple conditions and values that could be more succinctly expressed in an XPath `if-then-else`.
- Templates that just perform the default behavior (`xsl:apply-templates` for elements)
- Variables with simple expressions that are used only once.
- Variables and parameters that contain an `xsl:value-of` instead of using a `select` attribute.
- Unnecessary `xsl:if` around `xsl:apply-templates` or `xsl:for-each` or `xsl:value-of`.
- Unnecessary `xsl:text` when it is the only child of its parent.
- Consecutive `xsl:text` elements that can be merged.
- XPaths that:
  - start with `./`
  - contain unnecessary parentheses
  - use `child::` or non-abbreviated axes

Exercise 2: Making code more concise.

Repetitive Code

- `xsl:choose` where every `xsl:when` contains similar content.
- XPath expressions that appear in many places (should be variables).
- Large blocks of code that are repeated (should be functions or named templates).
- Templates with identical/similar content that could be merged.
- Templates/functions that are identical across multiple XSLT stylesheets (should use include or import).

Exercise 3: Consolidating repetitive code.

Improving Code Quality

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. 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.
- 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 4: Pull to push.

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

Exercise 5: Adding types.

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

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 Best Possible 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.

Other Improvements

Improving Performance

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

- Using keys.
- Avoiding reevaluating the same expression (create a variable).
- Breaking down a complex problem into multiple passes.
- Decreasing the size of the XSLT (if the size decrease is dramatic).
- Decreasing the size of the output (if the size decrease is dramatic).

More on performance tomorrow from Michael Kay...

Improving Clarity

- Indent your code!!!!
- Practice good naming practices:
  - Naming practices should apply especially to functions, named templates, parameters, variables and stylesheet files, but also to modes, keys, decimal formats, outputs, attribute sets, etc.
  - Use descriptive names, avoid `string` or `arg1`, `var2`.
  - 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, ideally with documentation for each template, function, variable and parameter.

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?
- 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.

Improving Modularity

- Reducing repetition increases modularity.
- Create compact, reusable functions that are liberal in what they accept as parameters, and conservative in what they return.
- Use `xsl:copy` or `xsl:element` to make generic templates reusable.
- Move common components into separate XSLT stylesheets and reuse them via `xsl:include` or `xsl:import`.

Making Your XSLT More Current

- Understand the new features in the version of XSLT you are using, and take advantage of them, for example:
  - Muenchian grouping in 1.0 should become `xsl:for-each-group` in 2.0.
  - Simple `xsl:for-each` instructions in 1.0 could become more concise XPath 2.0 `for` expressions.
- (When upgrading from 1.0 to 2.0, make sure you have addressed all backward compatibility issues.)
- Do not use deprecated features like `disable-output-escaping` and the `namespace::` axis.
- Generate current output (especially in the case of HTML).

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.
- All styling should be moved to CSS for easier customization.
- Samples should be provided for developers who wish to customize the XSLTs.

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/refactoring.pdf