Avoid common XSLT mistakes
Trade in bad habits for great code

Jirka Kosek

December 19, 2008

In the course of my teaching and consulting work, I've seen plenty of badly designed and poorly written XSLT code. Many of these bad habits are repeated over and over and can cause critical flaws in XSLT code. In this article, get a feel for the typical problems that come up in stylesheets and how to remedy them.

Writing code to handle XML transformations in XSLT is much easier than in any other commonly used programming language. But the XSLT language has such a different syntax and processing model from classical programming languages that it takes time to grasp all of XSLT's subtle nuances.

This article is in no way meant as an extensive and complex XSLT tutorial. Instead, it starts with explanation of topics that pose the biggest difficulties for inexperienced XML and XSLT developers. Later, it moves to topics related to the overall design of stylesheets and their performance.

Frequently used acronyms
- HTML: Hypertext Markup Language
- SOAP: Simple Object Access Protocol
- W3C: World Wide Web Consortium
- XHTML: Extensible Hypertext Markup Language
- XML: Extensible Markup Language
- XSLT: Extensible Stylesheet Language Transformation

Working with namespaces
Although it's increasingly rare to see XML documents without namespaces, there still seems to be some confusion related to their proper use in different technologies. Many documents use prefixes to denote elements in a namespace, and this explicit notation of namespaces doesn't typically lead to confusion. The example in Listing 1 shows a simple SOAP message that uses two namespaces—one for the SOAP envelope and one for the actual payload.

Listing 1. XML document with namespaces

```xml
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Body>
    <p:itinerary
        xmlns:p="http://travelcompany.example.org/reservation/travel">
```
As elements in the source document have prefixes, it's clear that they belong to a namespace. No one will have problems processing such a document in XSLT. It is sufficient to duplicate namespace declarations from the source document in the stylesheet. Although you can use arbitrary prefixes, it's usually more convenient to use the same prefixes as in typical input documents, as in Listing 2.

### Listing 2. Stylesheet that accesses information in a namespaced document

```xml
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
    xmlns:env="http://www.w3.org/2003/05/soap-envelope"
    xmlns:p="http://travelcompany.example.org/reservation/travel">

  <xsl:template match="/">
    Departure location:
  </xsl:template>

</xsl:stylesheet>
```

As you can see, this code declares namespace prefixes `env` and `p` on the root element `xsl:stylesheet`. Such declarations are then inherited by all elements in the stylesheet so you can use them in any embedded XPath expression. Also note that in XPath expressions, you must prefix all elements with the appropriate namespace prefix. If you forget to mention a prefix in any step, your expression will return nothing—an error for which it's difficult to track the cause.

Documents that use namespaces are typically the cause of trouble when the use of namespaces is not apparent at first blush. If you have a lot of elements in one namespace, you can define this namespace as a default using the `xmlns` attribute. Elements from the default namespace do not use prefixes; therefore, it's easy to miss that they're actually in a namespace. Imagine that you have to transform the XHTML document in Listing 3.
Avoid common XSLT mistakes

Listing 3. XHTML document using a default namespace

```xml
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Example XHTML document</title>
  </head>
  <body>
    <p>Sample content</p>
  </body>
</html>
```

It might be that you simply glanced over `xmlns="http://www.w3.org/1999/xhtml"`, or it might be that this default namespace declaration is preceded by a dozen other attributes and you simply didn't see what was in column 167—even on your widescreen display. It is quite natural to write XPath expressions like `/html/head/title`, but such expressions return an empty node set, because the input document contains no elements like `title`. All elements in the input document belong to the `http://www.w3.org/1999/xhtml` namespace, and this must be reflected in the XPath expressions.

To access namespaced elements in XPath, you must define a prefix for their namespace. For example, if you want to access a title in the sample XHTML document, you have to define a prefix for the XHTML namespace, then use this prefix in all XPath steps, as the example stylesheet in Listing 4 shows.

Listing 4. The transformation must use namespace prefixes even for input documents that use a default namespace

```xml
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0"
    xmlns:h="http://www.w3.org/1999/xhtml">
  <xsl:template match="/">
    Title of document:
    <xsl:value-of select="/h:html/h:head/h:title"/>
  </xsl:template>
</xsl:stylesheet>
```

Again, you have to be very careful about prefixes in XPath expressions. One missing prefix, and you'll get the wrong result.

Unfortunately, XSLT version 1.0 has no concept similar to a default namespace; therefore, you must repeat namespace prefixes again and again. This problem was rectified in XSLT version 2.0, where you can specify a default namespace that applies to un-prefixed elements in an XPath expression. In XSLT 2.0, you can simplify the previous stylesheet as in Listing 5.
Avoid common XSLT mistakes

Listing 5. Declaration of a XPath default namespace in XSLT 2.0

```xml
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0"
    xpath-default-namespace="http://www.w3.org/1999/xhtml">
  <xsl:template match="/">
    Title of document:
    <xsl:value-of select="/html/head/title"/>
  </xsl:template>
</xsl:stylesheet>
```

Improper use of node test text()

Most stylesheets contain dozens of simple templates that are responsible for processing leaf elements in input documents. For example, you store a price inside an element:

```
<price>124.95</price>
```

and you want to output it as a new paragraph in HTML with the currency and a label added:

```
<p>Price: 124.95 USD</p>
```

In many stylesheets I have seen, templates that handle this functionality can fail miserably. The reason is the use of the `text()` node test inside the template body, which in 99 percent of cases leads to broken code. What's wrong with the following template?

```
<xsl:template match="price">
  <p>Price: <xsl:value-of select="text()"/> USD</p>
</xsl:template>
```

The XPath expression inside the `xsl:value-of` instruction is shorthand for the expression `child::text()`. This expression selects all text nodes between the children of the `<price>` element. Typically, there's only one such node, and everything works as expected. But imagine that you put a comment or processing instruction in the middle of the `<price>` element:

```
<price>12<!-- I'm a comment. I should be ignored. -->4.95</price>
```

The expression now returns two text nodes: 12 and 4.95. But the semantics of `xsl:value-of` is such that it returns only the first node of the node set. In this case, you'll get the wrong output:

```
<p>Price: 12 USD</p>
```

Because `xsl:value-of` expects a single node, you must use it with an expressions that returns a single node. In many situations, a reference to the current node (.) is the right approach. The correct form of the example template above, then, is:

```
<xsl:template match="price">
  <p>Price: <xsl:value-of select="."/> USD</p>
</xsl:template>
```
The current node (.) now returns the whole <price> element. The xsl:value-of instruction automatically returns the string value of a node that is a concatenation of all text node descendants. Such an approach guarantees that you will always get the whole content of an element regardless of included comments, processing instructions, or sub-elements.

In XSLT 2.0, the semantics of the xsl:value-of instruction is changed, and it returns a string value of all passed nodes—not just of the first one. But it's still better to reference the element for which content should be returned to its text nodes. This way, code won't break when new sub-elements are added to provide more granular markup.

**Don't lose the context node**

Each template (xsl:template) or iteration (xsl:for-each) is instantiated with a current node. All relative XPath expressions are evaluated starting from this current node. If you start an XPath expression with /, the expression won't be evaluated against the current node; instead, the evaluation will start at the document root node. The result of such expressions will always be the same, and it won't be related to the current node.

Imagine that you want to process the simple invoice in **Listing 6**.

**Listing 6. Sample invoice**

```xml
<invoice>
  <item>
    <description>Pilsner Beer</description>
    <qty>6</qty>
    <unitPrice>1.60</unitPrice>
  </item>
  <item>
    <description>Sausage</description>
    <qty>3</qty>
    <unitPrice>0.59</unitPrice>
  </item>
  <item>
    <description>Portable Barbecue</description>
    <qty>1</qty>
    <unitPrice>23.99</unitPrice>
  </item>
  <item>
    <description>Charcoal</description>
    <qty>2</qty>
    <unitPrice>1.19</unitPrice>
  </item>
</invoice>
```

If you forgot to write expressions relative to the current node, you can easily end up with the wrong stylesheet, as in **Listing 7**.
Avoid common XSLT mistakes

Listing 7. Example of a bad stylesheet that loses context

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="/">
    <html>
      <head>
        <title>Invoice</title>
      </head>
      <body>
        <table>
          <xsl:for-each select="/invoice/item">
            <tr>
              <td><xsl:value-of select="/invoice/item/description"></td>
              <td><xsl:value-of select="/invoice/item/qty"></td>
              <td><xsl:value-of select="/invoice/item/unitPrice"></td>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

The expression `/invoice/item` in `xsl:for-each` correctly selects all items in the invoice. But expressions inside `xsl:for-each` are wrong, as they start with `/`, which means that they're absolute. Such expressions always return a description, the quantity, and price of the first item (remember from the previous section that `xsl:value-of` returns only the first node from a node set), because an absolute expression does not depend on the current node, which corresponds to the currently processed item.

To easily fix this problem, use a relative expression inside `xsl:for-each`, as in Listing 8.

Listing 8. Use of relative XPath expressions inside the iteration body

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="/">
    <html>
      <head>
        <title>Invoice</title>
      </head>
      <body>
        <table>
          <xsl:for-each select="/invoice/item">
            <tr>
              <td><xsl:value-of select="description"></td>
              <td><xsl:value-of select="qty"></td>
              <td><xsl:value-of select="unitPrice"></td>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Avoid broken links in non-Microsoft browsers

XSLT is good at automating common tasks. One such boring and laborious task is preparing a table of contents. With XSLT, you can generate such a table automatically. You simply generate
anchors, then links pointing back to them. In HTML, you create an anchor simply by putting a unique identifier inside the \texttt{id} attribute:

\begin{verbatim}
<div id="label">…</div>
\end{verbatim}

When you construct a link back to this anchor, add \texttt{label} after the fragment identifier (\#) to indicate that this is a link to a particular place inside the document:

\begin{verbatim}
<a href="#label">link to …</a>
\end{verbatim}

A real stylesheet typically produces labels and links by using the \texttt{generate-id()} function or a real identifier provided in the input document.

The problem with this linking task is actually not in XSLT itself but in some "too clever" Web browsers. I've seen many stylesheets in which a fragment identifier (\#) was added to the anchor by mistake. The output of the stylesheet was then tested only in Windows® Internet Explorer®. Unfortunately, Internet Explorer can recover from many errors in HTML code, so there's no problem with links from the user perspective. But if you try the same page in such browsers as Mozilla Firefox or Opera, the links are broken, because these browsers can't recover from the excessive \#.

To avoid other similar problems, the best you can do is test your stylesheet-generated output in multiple browsers.

**Simplify stylesheets by changing the context node**

If you process business documents or data-oriented XML, it's common not to rely extensively on a template mechanism but rather just cherry-pick the required content and assemble it to the desired form in one large template. Imagine that you want to process the invoice in Listing 9.

**Listing 9. Invoice with a complex structure**

\begin{verbatim}
<Invoice>
  <ID>IN 2003/00645</ID>
  <IssueDate>2003-02-25</IssueDate>
  <TaxPointDate>2003-02-25</TaxPointDate>
  <OrderReference>
    <BuyersID>S03-034257</BuyersID>
    <SellersID>SW/F1/50156</SellersID>
    <IssueDate>2003-02-03</IssueDate>
  </OrderReference>
  <BuyerParty>
    <Party>
      <Name>Jerry Builder plc</Name>
      <Address>
        <StreetName>Marsh Lane</StreetName>
        <CityName>Nowhere</CityName>
        <PostalZone>NR18 4XX</PostalZone>
        <CountrySubentity>Norfolk</CountrySubentity>
      </Address>
      <Contact>Eva Brick</Contact>
    </Party>
  </BuyerParty>
  
...</Invoice>
\end{verbatim}
A typical stylesheet for processing this document (see Listing 10) will contain a lot of repeated paths in XPath expressions, because a good deal of information is in the same part of the input XML tree.

Listing 10. This naive stylesheet uses a lot of repeated XPath code

```xml
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="/">
    <html>
      <head>
        <title>Invoice #<xsl:value-of select="/Invoice/ID"/></title>
      </head>
      <body>
        <h1>Invoice #<xsl:value-of select="/Invoice/ID"/>
          issued on <xsl:value-of select="/Invoice/IssueDate"/></h1>
        <div>
          <h2>Buyer:</h2>
          <p><b><xsl:value-of select="/Invoice/BuyerParty/Party/Name"/></b></p>
          <p>Address:<br/>
            <xsl:value-of select="/Invoice/BuyerParty/Party/Address/StreetName"/><br/>
            <xsl:value-of select="/Invoice/BuyerParty/Party/Address/CityName"/><br/>
            <xsl:value-of select="/Invoice/BuyerParty/Party/Address/PostalZone"/></p>
          <p>Contact person: <xsl:value-of select="/Invoice/BuyerParty/Party/Contact"/></p>
        </div>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Those repetitions in XPath expression are tedious—you have to repeat them again and again. They can also prove a future burden. Any changes to the structure of the input document create more places in which you have to adjust the expression. You can simplify the stylesheet by factoring out a common part of the expressions. You do this by using instructions that change the current node—`xsl:template` and `xsl:for-each`. The stylesheet in Listing 11 contains significantly less repeated information.

Listing 11. stylesheet with common XPath paths factored out

```xml
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="Invoice">
    <html>
      <head>
        <title>Invoice #<xsl:value-of select="ID"/></title>
      </head>
      <body>
        <h1>Invoice #<xsl:value-of select="ID"/>
          issued on <xsl:value-of select="IssueDate"/></h1>
        <div>
          <h2>Buyer:</h2>
          <p>
            <b><xsl:value-of select="/Invoice/BuyerParty/Party/Name"/></b>
          </p>
          ...  
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```
<xsl:for-each select="BuyerParty/Party">
  <p>
    <b><xsl:value-of select="Name"/></b>
  </p>
  <xsl:for-each select="Address">
    <p>Address:<br/>
    <xsl:value-of select="StreetName"/><br/>
    <xsl:value-of select="CityName"/><br/>
    <xsl:value-of select="PostalZone"/>
    </p>
  </xsl:for-each>
  <p>Contact person: <xsl:value-of select="Contact"/></p>
</xsl:for-each>
...
</div>
</body>
</html>
</xsl:template>
</xsl:stylesheet>

I've changed the match on the template from / to Invoice so that I don't have to repeat this root element name at the start of each XPath expression. Inside the template, I used xsl:for-each to temporally change the current node to buyer (BuyerParty/Party) and inside it once again to address (Address). It might seem strange to use xsl:for-each for non-repeating elements, but there's nothing wrong with it: The body of the iteration will be invoked only once but with a changed current node, which will save a lot of repeated typing.

## Processing mixed content

Mixed content is typically present in document-oriented XML. *Mixed content* is structure in which an element contains as children both elements and text nodes. A typical example of mixed content is a paragraph that contains text with additional markup, like emphasis or links:

```
```

It is important to process mixed content in document order; otherwise, you can get completely mangled output, with a changed order of sentence parts. The most natural way to process mixed content is by calling xsl:apply-templates on the element with mixed content or on all of its children. Subsequent templates can then handle embedded markup such as emphasis and links.

I've seen many stylesheets that use a "cherry-picked" approach for mixed content handling. This approach is well suited to documents with regular structure, but mixed content typically varies in its internal structure and is difficult to handle correctly this way. So, whenever you see mixed content, try to forget about simple xsl:value-of and xsl:for-each and move your interest to templates.
Ineffectiveness in your stylesheets

If you write small transformations operating on rather small datasets—for example, a view layer in a Web application—you’re probably not very concerned about performance of transformation itself, as this process is typically fractional to the rest of processing. But when an XSLT stylesheet performs complex operations or works on a large input document, it’s time to start thinking about the performance impact of constructs used in the stylesheet.

In general, it’s difficult to make any judgments solely from XSLT code, as it depends on the particular XSLT implementation—whether it can handle some code well and possibly speed it up by using some sort of optimization.

Regardless, some things are good to skip in real stylesheets. If you want to save the planet, use the descendant axis (\//) very carefully. When you use \\/, the XSLT processor has to inspect the whole tree (or subtree) in its full depth. In larger documents, this can be a very expensive operation. It is wise to write more specific expressions that explicitly specify where to look for nodes. For example, to get a buyer’s address, it’s better to write /Invoice/BuyerParty/Party/Address instead of //BuyerParty//Address or even //Address. The first variant is much faster, because only a fraction of the nodes have to be inspected during evaluation. Such targeted expressions are also less likely to be affected by the document structure evolution, where new elements with the same name but a different meaning can be added into different contexts in the input document.

Another trick when you do a lot of lookups, define a lookup key using xsl:key, then use the key() function to perform the lookup.

You can make plenty of other optimizations, but their impact depends on the XSLT processor you use.

XSLT 1.0 or 2.0?

Which XSLT version you use depends on several factors, but generally, I recommend using XSLT 2.0. The latest version of the language contains many new instructions and functions that can greatly simplify many tasks—shorter and straightforward code is always easier to maintain. Moreover, in XSLT 2.0, you can write schema-aware stylesheets, which use a schema to validate both input and output documents. Schema-aware stylesheets can use information contained in a schema to automatically detect some types of errors and mistakes in your stylesheets.

Conclusion

This article covered some areas that tend to be more challenging in XSLT. I hope that now you have better understanding of some XSLT features and that you will be able to write better XSLT stylesheets.
Related topics

- What kind of language is XSLT?
- Planning to upgrade XSLT 1.0 to 2.0, Part 3
- Search the xsl-list /at/ lists.mulberrytech.com Archives
- XML technical library on developerWorks

© Copyright IBM Corporation 2008
Trademark
(www.ibm.com/developerworks/ibm/trademarks/)