XSLT – Efficient Programming Techniques

Author: Prathit Bondre

With the growing popularity of XML as a medium to interact with different systems, more and more organizations are turning to XML to solve their interoperability issues. Also with architects trying to achieve a clear separation between display and business logic, XSLT is gaining importance. XSL, in itself, is an XML document tree (conforming to a specific DTD) that is applied to a data tree (XML document) to produce an output tree (HTML, WML etc).

This article presents a list of the best practices to be followed when writing XSL style sheets. This article can be used as a guide to a better way of achieving the right results in XSL. It is meant for developers who are familiar with the basics of XSL but need a roadmap to an efficient way of programming in XSL since performance is always a concern with XML based systems. The information in this article is based on my own extensive reading on XML and XSL. The list of best practices has been compiled from different sources to provide a comprehensive document that will grow as more good practices are discovered. If you have some best practices that you follow which are not listed below, drop me a mail at pbondre@gr.com.

1. Include external files the right way:

There are three use cases for including external files in your xsl:
1. You have additional HTML files that you want to include in the document you're producing.
If you have an HTML file that you want to include in your output, in exactly the form you want to include it in your output, probably the simplest way to get this into your output is to simply include it as an external parsed entity in the stylesheet. This involves declaring and referencing the entity within your stylesheet:

```
---- header.html ----
<table>
  <tr>
    <td><a href="/">Home</a></td>
    <td><a href="/movies/">Movies</a></td>
    <td><a href="/shop/">Shop</a></td>
  </tr>
</table>
----
```

```
---- data.xsl ----
<?xml version="1.0"?>
<!DOCTYPE xsl:stylesheet [
<!-- declares header.html as an external parsed entity
-->
<!ENTITY header SYSTEM "header.html">
]>
<xsl:stylesheet version="1.0"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
    <head><title>People</title></head>
    <body>
      <!-- includes header.html directly -->
      &header;
      <xsl:apply-templates />
    </body>
  </html>
</xsl:template>

</xsl:stylesheet>
----
```


2. You have additional XML files that you want to transform and include in the document you're producing.

If you have a xml file that you want to include in your output, you need to use the document() function to access this information and you need to have templates in your stylesheet to transform it and include it in your output:

```
---- header.xml ----
<menu>
  <item href="/">Home</item>
  <item href="/movies/">Movies</item>
  <item href="/shop/">Shop</item>
</menu>
----
```

```
---- data.xsl ----
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
  <html>
    <head><title>People</title></head>
    <body>
      <!-- applies templates to the information contained in
header.xml -->
      <xsl:apply-templates select="document('header.xml')"
/>
      <!-- applies templates to the input file -->
      <xsl:apply-templates />
    </body>
  </html>
</xsl:template>

<!-- transforms the XML in header.xml into the table we
want -->
<xsl:template match="menu">
  <table>
    <tr>
      <xsl:for-each select="item">
        <td><a href="{@href}"><xsl:value-of select="."
/></a></tr>
      </xsl:for-each>
    </tr>
  </table>
</xsl:template>

</xsl:stylesheet>
----
```

3. You have additional XSLT files that you want to use to transform your input:

If you have an input XML document that includes some information that you want as well as the data for the rest of the page, you want to import or include this stylesheet so that the templates that are defined within it are processed as if they were part of your general stylesheet. **Whether you want to use xsl:import or xsl:include depends on whether you want to override (some of) the templates that are defined in the imported stylesheet: if you do, then use xsl:import, otherwise, use xsl:include.**

```
---- data.xml ----
<?xml version="1.0"?>
<doc>
<menu>
  <item
href="/">Home</item>
  <item
href="/movies/">Movies</item
```

```
---- header.xsl ----
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"

xmlns:xsl="http://www.w3.org/1999/X
SL/Transform">

<xsl:template match="menu">
```

```
---- data.xsl ----
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"

xmlns:xsl="http://www.w3.org/1999/X
SL/Transform">

<!-- includes the templates from the
```

```
>
  <item
href="/shop/">Shop</item>
</menu>
<people>
  <person age="50"
name="larry"/>
  <person age="50"
name="larry"/>
</people>
</doc>
----
```

```
  <table>
    <tr>
      <xsl:for-each select="item">
        <td><a
href="{@href}"><xsl:value-of
select="." /></a></tr>
      </xsl:for-each>
    </tr>
  </table>
</xsl:template>

</xsl:stylesheet>
----
```

```
header.xsl stylesheet -->
<xsl:include href="header.xsl" />

<xsl:template match="/">
  <html>

<head><title>People</title></head>
    <body>
      <!-- applies templates to the menu
definition to create the header
          - the templates come from
header.xsl -->
      <xsl:apply-templates
select="doc/menu" />
      <!-- applies templates to the data
to create the rest of the document
-->
      <xsl:apply-templates
select="doc/people" />
    </body>
  </html>
</xsl:template>

...

</xsl:stylesheet>
```

You should also have a stylesheet that contains templates to transform this header information into the output that you want:

1.Use XSL Design Patterns.

1a. Use the Kaysian method for set intersection, difference and symmetric difference

The only set operation provided in XSLT is the Union -- and it can be specified using the XPath and XSLT union operator "|".It is possible to express the intersection of two node-sets in pure XPath. This technique was discovered by Michael Kay and is known as the Kaysian method.

```
<xsl:variable name="intersection" select="$ns1[count(.|ns2)=count(ns2)]"/>

<xsl:variable name="set-difference" select="$ns1[count(.|ns2)!=count(ns2)]"/>
```

Example:

```
 <?xml version="1.0"?>
 <xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
              xmlns:data="crane"
              version="1.0">
 <xsl:output method="text"/>
 <data:data>            <!--data source for testing purposes-->
   item>1</item><item>2</item><item>3</item>
  <item>4</item><item>5</item><item>6</item>
</data:data>
<xsl:template match="/">                    <!--root rule-->
```

```
----Output----
Intersection: 2
Intersection: 3
Intersection: 4
Difference: 1
Difference: 5
Difference: 6
```

```
  <xsl:variable name="ns1" select="//item[position()>1]"/>
  <xsl:variable name="ns2" select="//item[position()&lt;5]"/>
  <xsl:for-each select="$ns1[count(.|$ns2)=count($ns2)]">
    Intersection: <xsl:value-of select="."/>
  </xsl:for-each>
  <xsl:for-each select="(   $ns1[count(.|$ns2)!=count($ns2)]
                          | $ns2[count(.|$ns1)!=count($ns1)] )">
    Difference: <xsl:value-of select="."/>
  </xsl:for-each>
</xsl:template>
 </xsl:stylesheet>
```

1b. Use Wendel Piez method of non-recursive looping

The Wendel Piez method demonstrates a way to avoid XSLT recursion when implementing loops.

Example:

```
--Source--

<Tag ID="1">
 <Value>4</Value>
 </Tag> <Tag ID="2">
 <Value>2</Value>
</Tag>
```

Required Output

```
<TABLE>
 <TR ID="1">
 <TD> </TD>
 <TD> </TD>
 <TD> </TD>
 <TD> </TD>
 </TR>
</TABLE>
<TABLE>
 <TR ID="2">
 <TD> </TD>
 <TD> </TD>
 </TR>
</TABLE>
```

In other words, I want to create a set of new nodes, the count of which is based upon a *value* contained in the document. Below I present a small generalization which is independent of the number of nodes in the XML source document and uses the number of nodes in the stylesheet instead:

```
<xsl:template match="TAG">
 <TABLE>
 <TR ID="@ID">
 <xsl:for-each select="(document('')//*)[position() &lt;= Value]">
 <TD> </TD>
 </xsl:for-each>
 </TR>
 </TABLE>
</xsl:template>
```

This uses the capacity of the stylesheet for element nodes only. This capacity will be considerably increased if we test for more types of nodes like this:

```
<xsl:for-each select="($st//node()| $st//@* | $st//namespace::*) [position() &lt;= Value]">
```

where $st has been defined as document('') -- that is the root node of the stylesheet.

1c. Oliver Becker's method of conditional selection

Xpath's ablility to select a node-set based on complex conditions is very powerful. However it lacks the capabilities for specifying a string as opposed to a nodeset. Often you have to use a verbose multi-line xsl:choose construct just to specify that "in case1 use string1, in case2 use string2, ..., in caseN use stringN"?

In all such cases we feel the need of a technique, which would allow us to specify in a single XPath expression a string, which depends on condition(s).

Here's how to do it:
We want an XPath expression, which returns a string when some given condition is true, and returns the empty string if this same condition is false.We can think of "true" as "1" and of "false" as "0".But how to fit "1" to any string? Which string - handling function can we use? substring() seems quite convenient. And here's the trick: we can use substring() with only two arguments : substring(str,nOffset) will return the (remainder of the) string str starting at offset nOffset.

In particular:
substring(str,1) returns the whole string
substring(str, nVeryLargeNumber) will return the empty string, if nVeryLargeNumber is guaranteed to be greater than any possible string length.

So, the expression we might use would be:

```
concat(
substring(str1,exp(Condition)),
substring(str2,exp(not(Condition))
      )
```

and we want exp(Condition) to be 1 if Condition is true, and exp(Condition) to be Infinity if Condition is false.

We express exp(Condition) as:

1 div Condition because a boolean expression is first converted to a number (true -> 1, false -> 0), we get exactly:

```
exp(true)  = 1
exp(false) = Infinity.
```

To summarise:
The XPath expression returning Str1 if a condition Cond is true and returning Str2 if this same condition Cond is false -- this is:

```
concat(
substring(Str1,1 div Cond),
substring(Str2,1 div not(Cond))
      )
```

This was first used by (Oliver?) Becker and is being quoted as the method of Becker.

Example:
I want to have a template, which generates the text: "My department" when it is passed a parameter "IT" and to generate the text "Some other department" if the value of the parameter is not "IT".

Of course, no xsl:if or xsl:when -s are allowed.

Here's the code, and when applied to any xml source document, it generates:
IT:
  My department

Finance:
  Some other department

Example stylesheet:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
            xmlns:data="crane"
            version="1.0">
<xsl:output method="text"/>
  <xsl:template match="/">
                    IT:
                    <xsl:call-template name="whoIs">
                      <xsl:with-param name="department" select="'IT'" />
                    </xsl:call-template>
                    <br/>
                    Finance:
                    <xsl:call-template name="whoIs">
                      <xsl:with-param name="department" select="'Finance'" />
                    </xsl:call-template>
  </xsl:template>
  <xsl:template name="whoIs">
    <xsl:param name="department" select="someDepartment" />
          <br/>
    <xsl:value-of
    select="concat(substring('My department',
                       1 div ($department = 'IT')),
              substring('Some other department',
                      1 div not(($department = 'IT'))
                    ))" /><br/>
  </xsl:template>
</xsl:stylesheet>
```

1d. ***Use the Muenchian method for grouping.***

Grouping is often inefficiently implemented in XSL. A common situation in which this issue arises is when you are getting XML output (ungrouped) from a database which needs to be grouped by XSL. The database usually gives you results that are structured according to the records in the database. For example let us consider an employee table, which returns the following xml:

<data> <employee no="1"> <name>Prathit Bondre</name>	**-- Required Output --** **Finance** Adheet Bondre

<department>IT</department> </employee> <employee no="2"> <name>Adheet Bondre</name> <department>Finance</department> </employee> <employee no="3"> <name>Sinan Edil</name> <department>IT</department> </employee> <employee no="4"> <name>Jeremy King</name> <department>Finance</department> </employee> </data>	Jeremy King **IT** Prathit Bondre Sinan Edil

The problem is how to turn this flat input into a number of lists grouped by department to give the required output shown above.

There are two steps in getting to a solution:

- Identifying what the departments are.
- Getting all the employees that have the same department.

Identifying what the departments are involves identifying one employee with each department within the XML, which may as well be the first one that appears in . One way to find these is to get those employees that do not have a department that is the same as a department of any previous employee.

```
employee[not(department = preceding-sibling::employee/department)]
```

Once these employees have been identified, it's easy to find out their departments, and to gather together all the employees that have the same deaprtment:

```
<xsl:apply-templates select="/data/employee[department = current()/department]" />
```

The trouble with this method is that it involves two XPaths that take a lot of processing for big XML sources. Searching through all the preceding siblings with the 'preceding-siblings' axis takes a long time if you're near the end of the records. Similarly, getting all the contacts with a certain department involves looking at every single employee each time. This makes it very inefficient.

The Muenchian Method is a method developed by Steve Muench for performing these functions in a more efficient way using keys. Keys work by assigning a key value to a node and giving you easy access to that node through the key value. If there are lots of nodes that have the same key value, then all those nodes are retrieved when you use that key value. Effectively this means that if you want to group a set of nodes according to a particular property of the node, then you can use keys to group them together.

In the example above, we want to group the employees according to their department, so we create a key that assigns each employee a key value that is the department given in the record. The nodes that we want to group should be matched by the pattern in the 'match' attribute. The key value that we want to use is the one that's given by the 'use' attribute:

```
<xsl:key name="employees-by-deparment" match="employee" use="department" />
```

Once this key is defined, if we know a department, we can quickly access all the employees that have that department.

For example:
key(‘employees-by-department’, ‘IT’) will give all the records that have the department of ‘IT’.

```
<xsl:apply-templates select="key(‘employees-by-department’, department)" />
```

The first thing that we needed to do, though, was identify what the departments were, which involved identifying the first employee within the XML that had a particular department. We can use keys again here. We know that a employee will be part of list of nodes that is given when we use the key on its department: the question is whether it will be the first in that list (which is arranged in document order) or further down? We're only interested in the records that are first in the list.
Finding out whether a employee is first in the list returned by the key involves comparing the employee node with the node that is first in the list returned by the key. This technique can also be used for getting distinct elements in the XML file.There are a couple of generic methods of testing whether two nodes are identical:

1. compare the unique identifiers generated for the two nodes (using generate-id()):
2. employee[generate-id() =
3. generate-id(key('employees-by-department', department)[1])]
4. see whether a node set made up of the two nodes has one or two nodes in it - nodes can't be repeated in a node set, so if there's only one node in it, then they must be the same node:
5. employee[count(. | key('employees-by-department', department)[1]) = 1]

Once you've identified the groups, you can sort them in whatever order you like. Similarly, you can sort the nodes within the group however you want. Here is a template, then, that creates the output that we specified from the XML we were given from the database:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
        <xsl:output method = "html" encoding="Windows-1252" />

        <xsl:key name = "employees-by-department"  match ="employee" use = "department" />

        <xsl:template match="data">
                <html>
                        <head></head>
                        <body>

                        <xsl:for-each select = "employee[count(.|key('employees-by-
department',department)[1])=1]">
                                <xsl:sort select="department" />
                                <b><u><xsl:value-of select="department" /></u></b><br/>
                                <xsl:for-each select="key('employees-by-
department',department)">
                                        <xsl:sort select="name"/>
                                        <xsl:value-of select="name" /><br/>
                                </xsl:for-each>

                        </xsl:for-each>

                        </body>
                </html>
        </xsl:template></xsl:stylesheet>
```

The Muenchian Method is usually the best method to use for grouping nodes together from the XML source to your output because it doesn't involve trawling through large numbers of nodes, and it's therefore more efficient. It's especially beneficial where you have a flat output from a database, for example, that you need to structure into some kind of hierarchy. It can be applied in any situation where you are grouping nodes according to a property of the node that is retrievable through an XPath.
The downside is that the Muenchian Method will only work with a XSLT Processor that supports keys. In addition, using keys can be quite memory intensive, because all the nodes and their key values have to be kept in memory. Finally, it can be quite complicated to use keys where the nodes that you want to group are spread across different source documents.

2. Usage of XSL:IMPORT

Use <xsl:import> to import common, general-purpose rules into a stylesheet designed to handle the specific transformation. If you can help it, don't xsl:import any more xsl than you need.

3. Using static HTML

For any "static" html portions of the page (such as headers, footers, nav bars), it's definitely more efficient to store the snippets as external xml files & copy them to the output tree using xsl:copy-of and the document() function, rather than using a named template and xsl:import.

4. Understand the difference between call and apply templates.

Call-template, unlike apply-templates, doesn't change the context node. Also, a select attribute is only meaningful on apply-templates, not on call-template.

5. Code reuse and refactoring.

The problem with using one template with many conditionals is that the code gets nasty and unreadable and unmaintainable very quickly. The problems with many templates are that you often replicate code. The happy medium is to use many templates and when you need to replicate code, use calls to named templates, sometimes with parameters if there are slight variations that need to be accounted for. Named templates provide the equivalent of subroutines or private methods.

Example:
Say you want to process 'item' elements, and you want to have one template for when item's 'type' attribute is 'Book', one template for when it's 'CD', and one template for all other 'items:

```
<xsl:template match="item[@type='Book']"/>
<xsl:template match=" item [@type='CD']"/>
<xsl:template match="item"/>
```

And these are in addition to the built-in template that matches "*" (any element). The templates with the greater degree of specificity will have higher priority for matching.

6. Automate XSL documentation.

Programmers usually hate documentation and hence usually don't do justice to writing it. Javadocs in java provided great relief to the programmer community by providing a way to auto generate the documentation. There is a similar tool that was written for XSL called xsldoc. It is available for free download at: www.xsldoc.org
This will provide an automated, standard and reliable way to generate documentation about your XSL files and since it is command line based it could also be made a part of your build process.

7. Don't reinvent the wheel by using the XSLT library.

XSLT library is an open source repository of XSL templates that have been written as tested. The library has a lot of templates for string manipulation, date handling, node processing , etc that can be effectively used in your xsl files. So save yourself some time by resing this library. The library is available at http://xsltsl.sourceforge.net

8. Decrease the size of your HTML documents

Decrease the size of your HTML documents by using the indent="no" in the <xsl:output> tag. The attribute tells the XSLT processor not to indent the HTML document , which typically results in smaller HTML files that download faster.
Eg. <xsl:output method="html" indent="no" />

References :

http://www.vbxml.com
http://www.ibm.com