Let XSLT do things the way it was supposed to work.

A little while ago I was asked to look at some XSLT for someone who had done some changes to a template they didn't originally create, but who now couldn't figure out what the hell was going on. You know, the usual stuff.

So I told the person to send me the stuff, and I'll have a look and see what I can do to help. The files appeared; one input XML file and one input XSLT template, and one HTML file as an example of what it should look like when working properly. The XML file was basically a repository of various guides that has been made, and the XSLT was for creating an index of these guides, sorted by either subject or format, and grouped alphabetically.

This article will deal with a number of things, from badly created schemas to Muenchian grouping, giving some advice along the way to some real examples. Yes, I will inflict you with my opinionated advice on things such as filtering, XSLT best-practice and sorting and grouping. The example codes used are real data, but with the disclaimer that its somewhat old, the author doesn't work here anymore, some content is scrambled, and I have seen lots of good code here as well. Ahem.

I skimmed through the XML and knew already then that the XSLT – in the virtue of having to deal with this XML file - would be a real charmer indeed. Let's start with a snippet from the very top of the page;

```
<GUIDES>
   <HEADING>Guides</HEADING>
   <GUIDEDESCRIPTION> [snipped] </GUIDEDESCRIPTION>
   <CONTENTS>Contents</CONTENTS>
   <TOPICLIST>
      <TOPICNAME>Aged</TOPICNAME>
      <BOOKMARK>aged</BOOKMARK>
   </TOPICLIST>
   <TOPICLIST>
      <TOPICNAME>Asia</TOPICNAME>
      <BOOKMARK>asia</BOOKMARK>
   </TOPICLIST>
   <TOPICLIST>
      <TOPICNAME>Australia</TOPICNAME>
      <BOOKMARK>aust</BOOKMARK>
   </TOPICLIST>
   <TOPICLIST>
      <TOPICNAME>Australian Studies</TOPICNAME>
      <BOOKMARK>elec</BOOKMARK>
   </TOPICLIST>
   <TOPICLIST>
      <TOPICNAME>Biography</TOPICNAME>
      <BOOKMARK>biog</BOOKMARK>
   </TOPICLIST>
```

Ignoring the fetish for all capitals for all your markup needs, you have to ask why there are so many topic lists all having apparently one entry each? Of course what they're trying to portray is something more like this;

```
<TOPICLIST>
   <TOPIC>
      <NAME>Aged</NAME>
      <BOOKMARK>aged</BOOKMARK>
   </TOPIC>
   <TOPIC>
      <NAME>Australia</NAME>
      <BOOKMARK>aust</BOOKMARK>
   </TOPIC>
</TOPICLIST>
```

And so forth. Basically we're having a semantic problem right there; the XML uses language to portray lists, while in reality they are all one list. Confusing. Let's look a bit further down our XML file, after about 100 <TOPICLIST> items;

```
<TOPICLIST>
   <TOPICNAME>Miscellaneous</TOPICNAME>
   <BOOKMARK>misc</BOOKMARK>
</TOPICLIST>
<FORMATLIST>
   <FORMATNAME>Bibliographies</FORMATNAME>
   <BOOKMARK>bib</BOOKMARK>
</FORMATLIST>
<FORMATLIST>
   <FORMATNAME>Collection Guides</FORMATNAME>
   <BOOKMARK>collection</BOOKMARK>
```

```
</FORMATLIST>
```

All of a sudden the <TOPICLIST> turns into <FORMATLIST>, and note how the name of the item in question also changes from <TOPICNAME> to <FORMATNAME>. It boggles the mind why people feel that the semantics of your XML structure must be given in your element names when the structure is, um, quite apparently there for all to see. And why are we mixing up two lists like this? Maybe there is some reason for this that will be apparent once we dig into the XSLT?

XML, episode 1 : Very real Menace

Before we dig into those existential questions, let's cut to the real meat of our XML, the data in which our XML was conceptually designed to handle;

```
<!-- The following records begin with "O" -->
<RECORDS>
   <BIBREC>
      <TITLE>Oral History Sites in Australia and Overseas</TITLE>
      <DESCRIPTION>[snip] </DESCRIPTION>
      <URL>../../oh/links.html</URL>
      <SUBJECT>Oral History</SUBJECT>
      <FORMAT>Internet Guide</FORMAT>
      <CATEGORY>Multi-Disciplinary</CATEGORY>
   </BIBREC>
</RECORDS>
<!-- The following records begin with "P" -->
<RECORDS>
   <BIBREC>
      <TITLE>Find Pacific Materials</TITLE>
      <DESCRIPTION>[snip] </DESCRIPTION>
      <URL>../../find/pacific.html</URL>
      <SUBJECT>Pacific</SUBJECT>
      <SUBJECT>Manuscripts</SUBJECT>
      <FORMAT>Find...</FORMAT>
      <CATEGORY>Multi-Disciplinary</CATEGORY>
   </BIBREC>
   <BIBREC>
      <TITLE>Pacific Internet Resources</TITLE>
      <DESCRIPTION>[snip] </DESCRIPTION>
      <URL>../../oz/pacsites.html</URL>
      <SUBJECT>Pacific</SUBJECT>
      <FORMAT>Internet Guide</FORMAT>
      <CATEGORY>Multi-Disciplinary</CATEGORY>
   </BIBREC>
<RECORDS>
```

There is about 300 of these records, all grouped by a first letter. The first thing to note here is of course the general structure of the thing;

```
<!--  Records starting  with  some letter  -->
<RECORDS>
   <BIBREC><DATA /></BIBREC>
   <BIBREC><DATA /></BIBREC>
</RECORDS>
```

There is a fun little thing to notice here; can you spot in the records starting with 'P' the record that doesn't start with 'P'? Now some may say that that record should be sorted and displayed in the 'P' section because it is a granular of the “Pacific' keyword, which is a big assumption, especially since the markup doesn't indicate any such thing. In fact, 'P' in the context of the XML is what group it is to be sorted by, not title. I know this from lots of research talking to the stakeholders. There is no way to know that by looking at the XML nor the result page, not even by the comments in the XML. In other words, this XML does not tell us – through semantics nor structure – why the data is structured the way it is.

Further, if you're going to all the trouble of sorting your records by something, wouldn't it help things if you gave an indication anywhere in the XML data itself about this wonderful fact and not just in the remarks? I don't know, I would certainly feel that helpful, because if you don't, then how on earth can you act upon that given data with XSLT when XSLT can't get it? How are we supposed to write XSLT to handle this? Ok, let's have a tiny look at the original XSLT.

XSLT, episode 2 : Attack of the cloning

First of all, the template has two matches; 'ROOT', and going from that, the match 'GUIDES'. Some framework HTML is created in 'ROOT' with a <xsl:apply-templates /> in it, and all the content is generated in the 'GUIDES' rule, <GUIDES> of course being the root element of our XML file. Let's have a look at one of the hundreds of similar sections of that rule;

```
<!-- Begin code to insert all records with the subject that equals "Asia" -->
<xsl:for-each select="RECORDS/BIBREC [SUBJECT='Asia']">
   <p><a>
      <xsl:attribute name="href"><xsl:value-of select="URL"/></xsl:attribute>
      <xsl:value-of select="TITLE"/>
   </a></p>
   <p><xsl:value-of select="DESCRIPTION"/></p>
</xsl:for-each>
<!-- end of code -->
```

So basically there is a loop for every darn BIBREC with the conditional put on its SUBJECT sub-element. What do we have, then? A 1107 lines long active and hand-maintained piece of XSLT. And remember those <TOPICLIST> and <FORMATLIST> elements in the beginning of the XML file I didn't quite understand what did? At the top of this XSLT we've got some code to generate a list of all our topics;

```
<xsl:for-each select="TOPICLIST">
   <li><a class="toc">
      <xsl:attribute name="href">#<xsl:value-of select="BOOKMARK"/></xsl:attribute>
      <xsl:value-of select="TOPICNAME"/>
   </a></li>
</xsl:for-each>
```

Isn't it funny how some people want to do things the hard way? All of the following code;

```
<a class="toc">
   <xsl:attribute name="href">#<xsl:value-of select="BOOKMARK"/></xsl:attribute>
   <xsl:value-of select="TOPICNAME"/>
</a>
```

Could be written as;

```
<a class="toc" href="#{BOOKMARK}"><xsl:value-of select="TOPICNAME"/></a>
```

No wonder people complain about XSLT being a mess when people write such messy code. Anyway, a slight digression from the real matter, and the crux of what I want to talk about in this article;

The list of the topics and the topics themselves are both maintained in the one XML file. But where are the <FORMATLIST> items used? Ah, well you see, there is actually two XSLT files; one for listing the records by topic and one for listing them by format. So, the end result as of now is that we have one XML file that is maintained by hand clocking in at 1837 lines, and two XSLT templates maintained by hand at 1107 and 418 lines of code respectively. Every time you add or change any record in the XML file, you need to update both XSLT templates with data specific rules. Every time. The mind boggles.

One can ponder about how anyone came to such a setup, but I won't go into too much detail here. But these are real examples that's out there, and I see them all the time. For some context, a Google-search for “xslt tutorial” gives about 5,770 specific hits. Searches for “DTD tutorial” and “Schema tutorial” gave 610 and 757 hits respectively. That's a ratio of normative 4.2:1 which to me really implies why we're in this sorry mess, and I'll let off some steam now before we get to an alternative version of the XML and XSLT we've just seen.

XML, episode III : Vindictive schemes

First off, creating schemas - be it a DTD, an XML Schema or RELAX-NG – is fun. I like it. It makes me think of my data in interesting ways. I love to make my schemas as semantically rich as possible without losing sight of simplicity and elegance. If I create a schema that does the job but doesn't feel elegant, I scrap it and start over. Let me exemplify;

```
<topics>
  <topic>
     <topic-name />
     <topic-id>[n]</topic-id>
  </topic>
</topics>
```

This is a typical schema structure and naming of something called a topic. But I would not be happy with this at all.

First of all, we've got structure embedded in the naming of the elements, which is a big no-no in my world because you may want to reuse elements, restructure them and otherwise mangle the schema to fit more data than what your puny test-cases imply. A <name> element is contextualized by its place in the structure, not by its name. Many things can have a name, but only topics can have a <topic-name>. We want to reuse <name> in as many places as we can, because it gives us more semantically rich markup, so that at some later stage we can ask for any element with a <name> sub-element instead of group many name variants together, clogging down the system.

Second, the use of plurals to indicate groups and lists can quickly become visually difficult to differentiate between, especially with lots of in and out traffic between them. <Topics> <topic> <topics> <topic> <topic> </topics> </topics> </topic> Of course, good editors and validating tools help out here, but we're kind of assuming that no person will ever look at our XML for clues to what we're doing.

Thirdly, an element is used for something that is a direct PCDATA reference point to the element, in this case a unique identifier. I prefer these attached as attributes instead, cutting down on the ambiguity. Here's what I would prefer;

```
<topic-list>
  <topic id="[n]">
     <name />
  </topic>
</topic-list>
```

Here we got a clear indication of what the root level is (it is a list of topics, not just a bunch of topics), that each topic has a unique id (and not as many id's you like), and it has a name (and not a topic-name, which may or may not be the same as a name). All this semantically rich data without even filling it with data. That's how I like it. And it looks tidier too.

The next issue at this point is to talk a bit about what exactly is in the schema. In our XML file dissected a the beginning of the article we had presentation data mixed with actually data, where the data also relied on the order in which your XML elements came. Mixing presentation and data is something that should be avoided in all aspects of your development, including the schema level. Also given the declarative nature of XSLT, relying on the order of elements in your XML is a big no because order isn't guaranteed. Both of these are stumbling blocks in the land of not just XSLT processing, but any XML related development. Clear separations and no ambiguous assumptions about the processing of your XML is essential to happy development.

Let's get back to our XSLT problem. I fiddled with the original code a few minutes, and decided that not only did I want to rewrite the XSLT from scratch, but also change the XML file in question as well. Since these were both hand-maintained, there were no problems with doing it this way.

XML, episode IV : A new hope

First, I changed the XML;

```
<guides>
   <heading>Guides</heading>
   <description>[snip]</description>
```

Gone are the all-capital elements. This isn't a major issue, but it can be significant if your XML in some way needs to resemble classes and instances of things. In Java, a class might be 'SomeBeaver' while an instance might be 'dFormBeaver'. You never know. Next;

```
<!-- Our subjects; choose a different label if the presentation needs to look different -->
<index-presentation>
   <subject name="Maps">Maps</subject>
   <subject name="Manuscripts">Manuscripts</subject>
   <format name="Bibliography">Bibliographies</format>
   <format name="Collection Guide">Collection Guides</format>
   <format name="Discover Guide">Discover Guides</format>
</index-presentation>
```

Ok, ok, I admit it; I could have put these presentation related items in a separate XML file, but I decided that clearly marked and semantically prepped up, it would not be too evil to have them at the top of the XML file. So I'm a pragmatist. Sue me.

Next, all records converted into a similar but richer format, and not grouped by anything;

```
<record>
```

```
      <title>Aerial Photographs</title>
      <description>[snip]</description>
      <url>../../map/aerialphoto.html</url>
      <subject>Maps</subject>
      <format>Collection Guide</format>
   </record>
```

First, let me point out the link between "/guides/index-presentation/format/@name 'CollectionGuide'" and the "/guides/record/format 'CollectionGuide'"; every record has one or more <subject> and <format> elements which is an indicator of what group that record belongs to and should be sorted by. Remember that we had two XSLT templates, one for each of these groupings. One of the key points to the rewriting was to combine them so that ordering is automatic.

Next, none of the records are sorted in the XML in any way. They can be entered willy-nilly, and hence we don't care if the maintainer of the XML puts them in some kind of order or not. That is the second point to the rewrite; we want the sorting to be automatic.

Here is a reminder of one of the <topiclist> items;

```
<TOPICLIST>
   <TOPICNAME>Miscellaneous</TOPICNAME>
   <BOOKMARK>misc</BOOKMARK>
</TOPICLIST>
```

Notice the <BOOKMARK> element, now gone in the new XML schema; it was used to create anchors within the index page to the group it belongs to. Since we want to do this all automagically, it is removed and will be replaced with some filtering to make it HTML friendly instead. That means that to add a new group, you just ... um, add a new group to the records you feel fit and the system deals with it.

A quick check tells us that our XML file has gone from 1837 lines to 890 lines. That's a good start, considering the same data has been retained. But let's see how we can deal with the XSLT.

First, the very parameter that will drive the template;

```
<!-- Parameter in; 'sort' is either 'subjects' or 'formats'   -->
<xsl:param name="sort" select="'subjects'" />
```

The parameter 'sort' is, as the XML comment says, either 'subjects' or 'formats', which more or less speak for itself, and the default value is 'subjects'. That means that we can now apply templates based on this parameter;

```
<xsl:choose>
   <xsl:when test="$sort='subjects'">
      <xsl:call-template name="html.create.bookmarks">
         <xsl:with-param name="sort.by" select="$group.subjects" />
      </xsl:call-template>
   </xsl:when>
   <xsl:when test="$sort='formats'">
      <xsl:call-template name="html.create.bookmarks">
         <xsl:with-param name="sort.by" select="$group.formats" />
      </xsl:call-template>
   </xsl:when>
</xsl:choose>
```

We call the template 'html.create.bookmarks' with either the parameter '$group.subjects' or '$group.formats'. Let's have a closer look at how we set '$group.subjects' up at the beginning of our template, because within lies the secret to it all!

The dread pirate Muenchian!

Since we're dealing with grouping, we need to take a deep breath, and invoke our skills in the Muenchian Method. This fact alone is possibly one of the biggest reasons why so much XSLT is unnecessary complex. Here's the crunch; learn Muenchian Method grouping Right Now(TM)! Let's look at how this is done;

```
<xsl:key name="group.subjects.key" match="/guides/record/subject" use="." />
<xsl:variable name="group.subjects"
   select="/guides/record/subject[generate-id(.)=generate-id(key('group.subjects.key', .))]" />
```

Look at this code very carefully. First we create a key-set that holds all <subjects> of all our records. That means all of them, including duplicates, but to do proper grouping we need to get rid of all the duplicates. The way to do that is to invoke the black magic of Muenchian grouping (named so after Steve Muench, who came upon it) in our variable 'group.subjects', using the function 'generate-id()';

We take our list of subjects, including the duplicates, and we select the same node set again with the conditional that the element's unique id (as provided with the function 'generate-id()') in the node set is the same as the one found for that node in the key set. The explanation is that generate-id() gives us the unique id of the element in the node tree. Every node, every little scrap of info has an id that we can get, and we use this id marker to make sure that all duplicates are taken out of the final result.

Let's create a sequence of imaginative subjects; 'ABCCDB', and let's assign some unique id's to them: 'A(1) B(2) C(3) C(4) D(5) B(6)'. To create a new list, we look up each of these elements in our before created key table, and compare the id of them, like this; “For element A, is the current id '1' the same as the id for the element found in our key table?“ In fact, we're literary asking if the current id is the same as the first instance of that same element's id in the table. Is the element we're looking at the same as the first element of the keys? If it is, it is in our result; we want this element to be part of our result. If not, meaning it is not in the first position – hence meaning that it is already included in our result earlier – then it's not what we want in our result. Makes sense, doesn't it?

So, first we set up a lookup table of all subject elements, and then we pick out only the ones that has that first position id. Now we have a list of non-duplicated subjects. We're now free to write code that sort anything by subject. Let's see how that is done;

```
<xsl:call-template name="html.create.records">
   <xsl:with-param name="sort.by" select="$group.subjects" />
</xsl:call-template>
```

Here we invoke the template 'html.create.records' with the parameter 'sort.by' which IS our non-duplicated subject list. This, as with 'html.create.bookmarks', can be switched alternated depending on the '$sort' parameter. So let's have a look at what we find in the invoked template 'html.create.records';

```
<xsl:template name="html.create.records">
   <xsl:param name="sort.by" />
   <xsl:for-each select="$sort.by">
      <xsl:sort select="." />
```

For each of the elements in our input - here meaning for each of the non-duplicated subjects - sorted by the current node's content (meaning the name of the subject), do the following;

```
<xsl:variable name="this" select="." />
<xsl:variable name="records"
   select="//record[format = $this or subject = $this]" />
```

We create one variable with reference to the current subject node because we'll reference it later on, and another variable that contains all records found in our XML that has a sub-element named 'subject' or 'format' with content matching the content of the current subject node. That means all records with a subject or format the same as our list of subjects.

Let's look at creating an anchor to this group automatically;

```
<!-- Using translate() here to convert spaces to underlines in names and links -->
<a><xsl:attribute name="name">
   <xsl:value-of select="translate($this, ' ', '_')" />
</xsl:attribute></a>
```

See, it isn't so hard; we use the 'translate()' function to translate all spaces to underlines. We can add more rules here as other characters need translations, but for the intent of this XSLT, this was sufficient. We put all our bookmarks in the left column, and all our grouped records in the right column. Let's start off by getting the name of the group. This can be either one of two; either just the name of the subject itself (since we're demostrating this through the subjects list), or taken from the XML file if it has a different name for it in the <index-presentation> table. Earlier, at the top of our XSLT, we created;

```
<xsl:key name="naming.key" match="/guides/index-presentation/*" use="@name" />
```

We then can use this key set to look up names if they exist as so;

```
<xsl:variable name="label">
   <xsl:choose>
      <xsl:when test="string(key( 'naming.key', $this ))">
         <xsl:value-of select="key( 'naming.key', $this )" />
      </xsl:when>
      <xsl:otherwise><xsl:value-of select="$this" /></xsl:otherwise>
   </xsl:choose>
</xsl:variable>
```

```
<xsl:value-of select="$label" />
```

Basically, if the subjects (or formats) name is found in the 'naming.key' table, use the new name from the table. Otherwise, just use the subject name. One might wonder why I did this, and the answer is to provide an alternative display name for the subject. Very often you'll find that the subject is 'boat' when you want to display a more flamboyant title like 'The Boats of the Blue Pacific'. Options are good.

Now that we have the subject group sorted (pardon the pun), we want to list all the records within this group. This is easy now with this reminder from our XSLT above;

```
<xsl:variable name="records"
   select="//record[format = $this or subject = $this]" />
```

So we simply loop through them;

```
<xsl:for-each select="$records">
   <p class='record-title'><a>
      <xsl:attribute name="href"><xsl:value-of select="url" /></xsl:attribute>
      <xsl:value-of select="title" />
   </a></p>
   <p class='record-description'><xsl:value-of select="description" /></p>
</xsl:for-each>
```

And then wrap it all up;

```
   </xsl:for-each>
</xsl:template>
```

What have we learned?

I hope that this little article has given you some fraction of real-world hands-on examples of problems you might find and how to solve them in XSLT in a more true-to-the-XSLT-spirit way. I know this only scratches the surface, and alternatives to what is presented here also exist. I've however tried to focus on the real issues, giving examples of solutions that are somewhat easy to understand.

I hope I've given some good example to follow when you encounter XML and XSLT files like this. First of all, make sure that the XSLT doesn't need to be altered if your XML adds data; we want our XSLT templates to be totally data driven. Secondly, learn Muenchian and don't be afraid to use it. Third, don't let poor schemas get in the way of good XSLT. Fourth, separate your data from the presentation as much as possible. And fifth, size does matter; the smaller the better.

There is a sixth lesson learned, but since it involves princesses, pirates and pain, I've omitted it for now, in your best interest.

Regards,

The Dread pirate Alex