XML Transformation using XSL and XSLT

Geoff Adams, FGM Inc
6 July 2006

Why use XSL?

Converting an XML document into some other format is frequently a useful thing to do. Perhaps:

- Someone publishes some data in a different XML markup than you use.

- You have changed your XML markup, such as with a new revision of your software. You want to upgrade your old XML documents to the new format.

- You want to transform your XML documents into some other format, such as HTML.

A brief note about XML

XML is used to mark up information or data, known as content. The best uses give semantic context to each piece of data being marked up. How that information is used is then entirely up to the application that uses it.

Imagine you have some content in an XML format, and you want to publish it in some way, such as on a web site, or in a PDF file, or on a poster. You want to take the information and present it an a useful and pleasing way. To publish in HTML, you would specify page titles, headings, and paragraphs. To publish as a PDF or a poster, you'd lay out the content on the page, specify fonts, sizes, and relative positions of the text, and so on. All of these operations can be thought of as adding style to the content in order to generate the presentation of the content.

XSLT can be used to apply this stylistic information to the content. That's why XSL stylesheets are called stylesheets.

What is XSL / XSLT?

XSLT is XML Stylesheet Language Transformation, the operation that uses an XML Stylesheet Language (XSL) stylesheet to transform an input XML document into an output document.

An XSL stylesheet is a collection of template rules that describe what output to generate when the transformer encounters certain elements in the input document. That makes it a rule-based language, like Make or Prolog. In fact, XSL is a Turing Complete language. That means that it is possible to implement Turing Machines in XSLT, and it is widely believed that Turing machines are powerful enough to perform any calculation that can be performed by a modern computer program.

The structure of XML

For the purposes of XSLT, XML documents are considered to be a tree of nodes. There are seven types of nodes in the XSL model:

- the root
- elements
- text
- attributes
- namespaces
- processing instructions
- comments

The root node is the node that contains all the other nodes. This is separate from the root element, since comments and processing instructions may appear outside of the root element. The root node is the parent of the root element, and represents the document as a whole.

All well-formed XML documents consist of a single root element, which itself contains any collection of other elements, attributes, text, and so on. This means that the root node contains exactly one child element node (but possibly other, non-element nodes).

It's important to think of the XML document as containing elements, not start and end tags. Each element may have attributes, which are just attribute nodes within the element node. Any child elements or text that are within the element are element nodes or text nodes within the element. And so on.

Let's jump right in

So, let's take a look at a real, working XSL stylesheet.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <html>
      <p>Hello, world</p>
    </html>
  </xsl:template>

</xsl:stylesheet>
```
Figure 1: hello-world.xsl

First, you'll notice that the stylesheet begins with an XML declaration. That's because XSL stylesheets are, themselves, XML documents. And, as a valid XML document, our stylesheet contains a single root element, the `xsl:stylesheet` element, in the required namespace. If you're not familiar with XML namespaces, then suffice it to say that all XSL elements will start with the "`xsl:`" prefix. (You could pick a different prefix, but let's stick with the convention, eh?)

The fact that XSL stylesheets are themselves marked up in XML is both a benefit and a hinderance. It's useful because it means we can transform them and generate them as the result of transforming other XML documents. It's annoying, though, because it tends to make the language fairly hard to read.

The meat of this particular stylesheet is a single template rule. This rule matches the root node ("/") in the source document and outputs a valid (if simplistic) XHTML document. (We'll get to how that works shortly.) Your first program in any language is supposed to be something like "Hello, world," right? This is "Hello, world" in XSL.

So, how do we perform XSL Transformation on an XML document using this stylesheet? There are several ways. You might configure your web server to transform XML documents before sending the results to the client. Or you might do this as an embedded step within your application. But the easiest way for us right now is to use a stand-alone XSLT implementation, such as Xalan or the SAXON or xsltproc command-line tools. I'll use xsltproc, since it's fast and complete, doesn't care what version of Java you have, and comes with Mac OS X.

```
xsltproc hello-world.xsl defects.xml
```

That command instructs xsltproc to perform XSL Transformation using our stylesheet on some random xml file we have lying around. It doesn't really matter what's in the XML file, since our simplistic stylesheet doesn't actually do anything with the information in it, anyway.

It produces this output:

```
<html><p>Hello, world</p></html>
```

Now go back and look at the XSL stylesheet. Notice that it contains a mix of elements in the "xsl" namespace, and some elements with no namespace prefix. The xsl: elements are XSLT instructions, and any other elements are just literal markup to be copied into the output document.

What happened to our line breaks and indentation? Well, let's not worry about that for now. There are ways to control formatting, but they would just make our simple stylesheet more complicated.

So, what happened to all the information in the XML file? The one rule in our stylesheet matched the root node, supplied some output, and contained no instructions to look further into the source XML document's tree for further matches against the templates in the stylesheet. So the rest of the source XML document (other than the existence of its root node) was ignored.

The xsl:apply-templates element

To look further into the input document, we need to tell the transformer to match against the children of the root node. If you think of general tree-parsing algorithms, you'll think of recursion, and that's exactly what you do in XSL.

First, let's take a closer look at the defects.xml file that we just used a minute ago.

```
<?xml version="1.0"?>
<DefectList>
  <Defect number="5">
    <Description>When I bend my arm like that, it hurts</Description>
    <Found>2006-01-15</Found>
    <Fix fixedby="Wise Doctor" date="2006-01-15">
      <Note>Don't bend your arm like that.</Note>
    </Fix>
  </Defect>
  <Defect number="18">
    <Description>Typo on page 3 of the User Manual</Description>
    <Found>2006-03-12</Found>
  </Defect>
  <Defect number="17">
    <Description>Page 3 of the User Manual is missing</Description>
    <Fix fixedby="Frank" date="2006-03-11">
      <Note>Yeah, my bad. I added it back in.</Note>
    </Fix>
    <Found>2006-03-05</Found>
  </Defect>
  <Defect number="20">
    <Description>The calculator function computes 2+2=5</Description>
    <Found>2006-04-01</Found>
    <Comment>On my computer, I get 2+2=4.73333. Weird.</Comment>
  </Defect>
  <Defect number="9">
    <Description>Program crashes when I sneeze</Description>
    <Found>2006-02-05</Found>
    <Comment>Are you kicking the power cord when you sneeze?</Comment>
  </Defect>
</DefectList>
```
Figure 2: defects.xml

So, clearly there's some information in there we could use. Let's modify our stylesheet a little:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <html>
      <xsl:apply-templates />
    </html>
  </xsl:template>

  <xsl:template match="DefectList">
    <body>
      <ul>
        <xsl:apply-templates />
      </ul>
    </body>
  </xsl:template>

  <xsl:template match="Defect">
    <li>A defect</li>
  </xsl:template>

</xsl:stylesheet>
```

Figure 3: defect-summary1.xsl

Notice that the first template, instead of outputting a simple paragraph element inside the html element, applies further templates inside the html element. So, here's what happens when we invoke the stylesheet on our defect list XML document:

```
      xsltproc defect-summary1.xsl defects.xml
```

First, the processor compares the root node against all the template rules in the stylesheet. The first rule matches, so the processor outputs the <html> tag. Then, the `xsl:apply-templates` element causes the transformer to process the child nodes of the root element of the input document, looking for matches for them.

The first child of the root node is the DefectList element, which matches the second template rule, so that rule is invoked. That rule outputs the <body> start tag and a <ul> tag within it. It then uses `xsl:apply-templates`, which causes the transformer to process the child nodes of the DefectList element. You may recognize this now as a depth-first parse of the source XML tree.

The first child of the DefectList element is a Defect element. That matches the third rule. The third rule outputs an <li> element containing the text "A defect," and then ends.

The second, third, and fourth child elements of the DefectList are more Defect elements, so the third rule is matched for them, as well. Three more <li> elements containing "A defect" are written out.

There are no more child elements of the DefectList element, so the DefectList template outputs </ul> and </body> tags, and then finishes.

The first template then outputs the </html> tag, and we're done. The output ends up looking like this:

```
<html><body><ul>
  <li>A defect</li>
  <li>A defect</li>
  <li>A defect</li>
  <li>A defect</li>
  <li>A defect</li>
</ul></body></html>
```

Well, that's not too exciting. There's no real content there. So, let's modify the third template rule using the `select` attribute of `xsl:apply-templates`. This attribute allows us to select particular nodes to apply templates to, rather than simply selecting all children of the current node.

```
  <xsl:template match="Defect">
    <li><xsl:apply-templates select="Description" /></li>
  </xsl:template>
```

Figure 4: Excerpt from defect-summary2.xsl

Now, for each Defect element we match, we'll output an <li> tag, the text of the Description element, and a </li> tag. The output will now look like this:

```
<html><body><ul>
  <li>When I bend my arm like that, it hurts</li>
  <li>Typo on page 3 of the User Manual</li>
  <li>Page 3 of the User Manual is missing</li>
  <li>The calculator function computes 2+2=5</li>
  <li>Program crashes when I sneeze</li>
</ul></body></html>
```

Isn't that useful? We now have a simple web page that presents the descriptions of all the defects listed in the XML file.

There are two important questions that you might now have. First, why did applying-templates on the Description element output its text contents? That happens because there are default template rules that match everything and output the textual content of the elements matched. We could also have used another XSLT instruction, the `xsl:value-of` element, in that last rule:

```
  <xsl:template match="Defect">
    <li><xsl:value-of select="Description" /></li>
  </xsl:template>
```

That makes things a bit more clear. The `xsl:value-of` element computes the value of an element, which is always a string. The value of an element is just the textual content (parsed character data) of the element and its descendant elements. The values of other node types are similarly sensible; for attributes, the value is the normalized value of the attribute, for comments, the value is the text of the comment between the <!-- and -->, etc.

The second question is about what sorts of things can we select using the `select` attribute. If we can select child elements, can we select other elements? In fact, we can. We can also select other types of nodes, using any XPath expression.

XPath

XPath expressions allow you to identify a particular element, a set of elements, a bit of text, or other parts of an XML document. XPath expressions are defined in a separate specification, so they can be used in other contexts. There are Java libraries that allow you to use XPaths to select content within DOM trees, for instance.

XPath is a very powerful expression language. In fact, it's so rich, that I'll just describe a subset of it here.

XPath expressions start with a context node. In the case of one of our template rules, the context node is the node which the template matched and is currently processing. For instance, in the "Defect" template rule from above,

```
<xsl:template match="Defect">
    <li><xsl:value-of select="Description" /></li>
</xsl:template>
```

the context node is a particular Defect element:

```
<Defect number="5">
  <Description>When I bend my arm like that, it hurts</Description>
  <Found>2006-01-15</Found>
  <Fix fixedby="Wise Doctor" date="2006-01-15">
    <Note>Don't bend your arm like that.</Note>
  </Fix>
</Defect>
```

Within this rule, we select using the XPath expression "Description." This is one of the simplest forms of XPath expression; it selects elements named "Description" within the context node. We happen to know that there is only one Description element, so this XPath expression returns a node-set containing a single node, which is a Description element.

XPath expressions can return node sets, strings, numbers, and some other types of data. But in most cases, these types are automatically converted to an appropriate type depending on how you use them, so we'll ignore this detail for now.

Hierarchy

We can use the hierarchy operator "/" in our expressions to select child nodes of child nodes:

```
<xsl:value-of select="Fix/Note" />
```

This will select the Note element nodes within a Fix element (within the context node, which in this case is a Defect element) and compute its value.

The "//" operator selects any matching descendant nodes, not just immediate children. For instance, we can select all the Note elements within the context node (abbreviated "."), regardless of how many levels of the hierarchy we need to traverse to reach them.

```
<xsl:value-of select=".//Note" />
```

We can also select attributes, such as the "number" attribute of the Defect using "@":

```
<xsl:value-of select="@number" />
```

We can put these together, of course, such as this expression to compute the value of the "date" attribute of the Fix element:

```
      <xsl:value-of select="Fix/@date" />
```

We don't have to start with the context node, by the way. If the XPath expression starts with a "/", then it starts searching from the root node of the source document. For instance, we can select all the Fix nodes within the entire document like this:

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

And we can also use the asterisk to match a node of any name. For instance, we can get the value of any attribute of our "Fix" child element like this:

```
      <xsl:value-of select="Fix/@*" />
```

The asterisk also works for selecting elements, of course. However, since these XPath expressions return a node set, and the set may contain more than one node, we won't get what we might expect, here. `xsl:value-of` only takes the value of the first node in the set, so it will only return the value of the first attribute of the Fix element that it comes across. And attributes don't have any defined order.

You can also use ".." to select the parent of the context node, as you'd expect.

Predicates

For the rest of these examples, I'll skip the `xsl:value-of` element, and just concentrate on the XPath expression itself, which is the string value for the `select` attribute in the examples above.

So far, this has all been exciting, but sometimes it is necessary to select particular elements, not just all the elements of a given name. We can do this using predicates. Predicates appear in square brackets ("[" and "]") after a step in the node path. Predicates allow you to select particular matched nodes based on properties of and tests on the node. For instance, we can select the third Defect element that appears in the source document:

```
      //Defect[3]
```

It is important to consider operator precedence here, as with expressions in many languages. For instance, note the difference between the following expressions:

`/DefectList/Defect/Fix[1]`	Selects the first Fix element of every Defect in the DefectList, since the [1] predicate is applied to the last step in the path, the Fix element. In our example, this would return two Fix elements.
`(/DefectList/Defect/Fix)[1]`	Selects the first Fix element in a Defect in the DefectList. This will only return one Fix element.

We can select the second Fix element in the third Defect element, as well:

```
/DefectList/Defect[3]/Fix[2]
```

There are all kinds of useful things we can put in predicates. We can test for the existence of child nodes, simply by naming them. So, this expression selects any Defect elements that have a Fix element as a child:

```
/DefectList/Defect[Fix]
```

And this expression, with two predicates, selects the second Defect with a Fix:

```
/DefectList/Defect[Fix][2]
```

We can test values of nodes, too, using such tests as "=", "!=", ">", and "<", and so on. This expression selects all Defect elements within the context node that were fixed by "Frank":

```
Defect[Fix/@fixedby = 'Frank']
```

This, of course, can be tricky. You need to know exactly what values to expect in the source document. For this reason, it's often better to test on the values of attributes, rather than values of elements, especially since elements often have lots of ignorable whitespace that won't be ignored, while attribute values are normalized (including removing leading and trailing white space).

There are also a number of functions you can use in predicates, such as position() and last(). So, you can select the last Defect in the DefectList:

```
Defect[last()]
```

Although, since the Defects aren't necessarily ordered in the source document, that may not mean much. It might be more interesting to count them:

```
count(Defect)
```

This would return the number of Defect elements within the context node. How about how many Defect elements that don't have a Fix child element?

```
count(Defect[not(Fix)])
```

We can also do math, since XSL supports numbers. For instance, we can select every other Defect in the DefectList:

```
/DefectList/Defect[position() mod 2 = 1]
```

More tricks

In addition to restricting the selected nodes using predicates, you can expand the selection using the pipe character ("|"), known as the "or" operator. Think of it as performing a union operation. For instance, this expression selects all Fix and Comment child elements of the context element:

```
Fix | Comment
```

Similarly, this expression selects all Defect elements that have either a Fix or a Comment child element:

```
Defect[Fix | Comment]
```

You can also do normal boolean operations in predicates, such as:

```
Defect[Fix/@fixedby='Frank' or Fix/@fixedby='Gertrude']
```

There are also some functions that match nodes based on their type. These are `node()`, `text()`, `comment()`, and `processing-instrcution()`. So, we can match on Defects that have XML comments within them:

```
Defect[comment()]
```

Of these, `text()` is used most often. It can be used, for instance, to refer explicitly to the text content within an element when that element may contain both text and other elements as children. This is subtle, and best illustrated by an example. If we have this XML fragment:

```
<Title><Article>The</Article> Crucible</Title>
```

We could select just the text " Crucible" (omitting the article) using this XPath expression:

```
Title/text()
```

We can even get rid of that leading white space using a text-processing function:

```
normalize-space(Title/text())
```

There are also a number of axes that can be used when selecting elements. For instance, there are `self`, `attribute`, `ancestor`, `child`, `descendant`, `following`, `following-sibling`, and several others. These axes are used by following the axis name with a double-colon "::" and then the name of something you'd like to match on that axis. For instance, we can use the `child` axis to select a child node of the context node, like this:

```
child::Comment
```

However, there is an abbreviated syntax, which in fact the above examples have been using. In abbreviated syntax, that expression is simply `Comment`. Here is the list of axis abbreviations:

`.`	`self::node()`
`..`	`parent::node()`
`name`	`child::name`

@name	attribute::name
//	/descendant-or-self::node()/

Well, that's a really brief introduction to XPath. There's a lot more to it, and XSL stylesheet documents benefit greatly from its power.

Some more XSL instructions

Well, al that XPath was fun, but it's time to get back to XSLT. So far, we've learned about `xsl:stylesheets` and `xsl:template` rules and about two instructions that can go in the template bodies, `xsl:apply-templates` and `xsl:value-of`. But there are several more.

For instance, we don't have to apply templates if we want to do something with a bunch of nodes within the context node. We can use `xsl:for-each` to do something with each member of a node set. For instance:

```
<xsl:template match="DefectList">
  <body>
    <ul>
      <xsl:for-each select="Defect">
        <li><xsl:value-of select="Description" /></li>
      </xsl:for-each>
    </ul>
  </body>
</xsl:template>
```

Note that it's important that the stylesheet remain well-formed XML. So, the literal elements (the elements such as <li> that appear in the template bodies) must be nested correctly according to the XML rules.

This new version of the template has the same end effect as the previous one, in which we called `xsl:apply-templates` to process all the Defect child elements, together with the template rule that matched Defect elements and emitted an "li" element containing the value of the Description child element.

Also notice that within the body of the `xsl:for-each` element, the context node has changed to the particular Defect node that we're processing each time through the "loop." This way, the XPath expression "Description" refers to the Description element that's a child of the context node, which is a Defect element. That's just what we'd expect, right?

Why would we use `xsl:for-each`? In this case, perhaps we already had a template rule matching Defect, but it did something else. Or perhaps we just don't want to be bothered writing a whole template rule just to output one little "li" element. But, of course, sometimes the distinction is very handy.

One very handy thing we can do with that `xsl:for-each` instruction is sort the elements from the source document before we process them:

```
<xsl:template match="DefectList">
  <body>
```

```
      <ul>
        <xsl:for-each select="Defect">
          <xsl:sort select="@number" data-type="number" order="descending" />
          <li><xsl:value-of select="Description" /></li>
        </xsl:for-each>
      </ul>
    </body>
  </xsl:template>
```

Now we know that the defects are going to appear in the output sorted by descending order of their Defect/@number attribute. Note that I specified that the data-type is "number"; the default is "text," and I think we all know the trouble that can lead to when sorting numbers. You can also apply sort to `xsl:apply-templates`, by the way.

So, now we can recurse, and we can loop. This really is looking like a programming language, isn't it? What else might we want in a programming language? Why, variables, of course.

In XSLT, you can define a variable using the `xsl:variable` instruction. However, variables behave a little like constants in XSLT, since you can only define them once within any given scope, and their value is set when it's defined. However, a variable's value may be derived from content in the source document, so they can vary from one invocation of the stylesheet to the next, or from one invocation of a template rule to the next.

Here is a fairly pointless use of a variable in the Defect template:

```
  <xsl:template match="Defect">
    <xsl:variable name="descr">
      <xsl:value-of select="Description" />
    </xsl:variable>
    <li><xsl:value-of select="$descr" /></li>
  </xsl:template>
```

Note that the XPath expression "$descr" refers to the variable. That's part of XPath that I left out, above. This template rule puts the string value of the Description element (its text content) into the variable. About the only thing we can do with that is put the value somewhere, such as in the "li" element. Notice the difference in this version:

```
  <xsl:template match="Defect">
    <xsl:variable name="descr" select="Description"/>
    <li><xsl:apply-templates select="$descr" /></li>
  </xsl:template>
```

In this version, the variable value is a node set containing all the Description elements that are children of the Defect element. (There's just one, looking at our source XML document.) Given a node set, we can apply templates on those nodes. This has exactly the same end effect, but the mechanism is a bit different.

Similar to setting and using variables is passing parameters to a template rule. That might look like this:

```
  <xsl:template match="DefectList">
    <body>
      <ul>
        <xsl:apply-templates>
          <xsl:sort select="@number" data-type="number" order="descending" />
          <xsl:with-param name="url">http://ourserver/defects</xsl:with-param>
        </xsl:apply-templates>
      </ul>
    </body>
  </xsl:template>

  <xsl:template match="Defect">
    <xsl:param name="url"/>
    <li><a href="{$url}/{@number}.html">
      <xsl:apply-templates select="Description" /></a>
    </li>
  </xsl:template>
```
Figure 5: Excerpt from defect-summary3.xsl

The output of the stylesheet will now be something like this:

```
<html><body><ul>
<li><a href="http://ourserver/defects/20.html">The calculator function computes 2
+2=5</a></li>
<li><a href="http://ourserver/defects/18.html">Typo on page 3 of the User
Manual</a></li>
<li><a href="http://ourserver/defects/17.html">Page 3 of the User Manual is
missing</a></li>
<li><a href="http://ourserver/defects/5.html">When I bend my arm like that, it
hurts</a></li>
</ul></body></html>
```

Now, we have a useful page containing links to the defects themselves. Notice the use of curly braces in the href attribute. They allow us to use XPath expressions directly in literal elements' attribute values. How convenient. In this case, the href attribute is composed of the value of the "url" parameter, which we use just like a variable reference, a literal slash character, the value of the Defect element's number attribute, and finally the literal text ".html".

Tests

One more thing that you definitely want in a programming language is conditional evaluation (execution). We have that in XSL using `xsl:if` and `xsl:choose`. `xsl:if` is very simple. If the test passes, the template body within the `xsl:if` element will be evaluated.

```
  <xsl:template match="DefectList">
    <body>
      <ul>
        <xsl:for-each select="Defect">
          <xsl:if test="Fix">
            <li><xsl:value-of select="Description" /></li>
          </xsl:if>
        </xsl:for-each>
      </ul>
    </body>
  </xsl:template>
```

This template rule will only generate output for Defect elements that have a Fix element as a child. There is no "else" or "else if" construct. For that, we use `xsl:choose`.

```
  <xsl:template match="DefectList">
    <body>
      <ul>
        <xsl:for-each select="Defect">
          <xsl:choose>
            <xsl:when test="Fix">
              <li style="text-decoration: line-through">
                <xsl:value-of select="Description" /></li>
            </xsl:when>
            <xsl:otherwise>
              <li><xsl:value-of select="Description" /></li>
            </xsl:otherwise>
          </xsl:choose>
        </xsl:for-each>
      </ul>
    </body>
  </xsl:template>
```

This will put a line through any defects that are already fixed, and print unfixed defects normally. Seems kinda useful. Note that you can have as many `xsl:when` elements as you like inside an `xsl:choose`. You may have one `xsl:otherwise`, but you don't have to. `xsl:choose` works sort of like a switch/case statement, with `xsl:otherwise` being equivalent to default. But since you can have interesting tests for each `xsl:when`, it also works where you'd use if, else if, and else.

Putting it all together

There are a few more XSL instructions, but this is enough to get you started. In fact, using just this knowledge, let's write a stylesheet that will build two tables, the first of as-yet unfixed defects, sorted in ascending order, followed by the fixed defects, in descending order. We'll include useful headers, and include dates entered and dates fixed where appropriate. We'll include links to each defect's detailed description (generated elsewhere), and we'll include a summary at the top of the page including the number of open and fixed defects, as well as the date of the oldest unfixed defect.

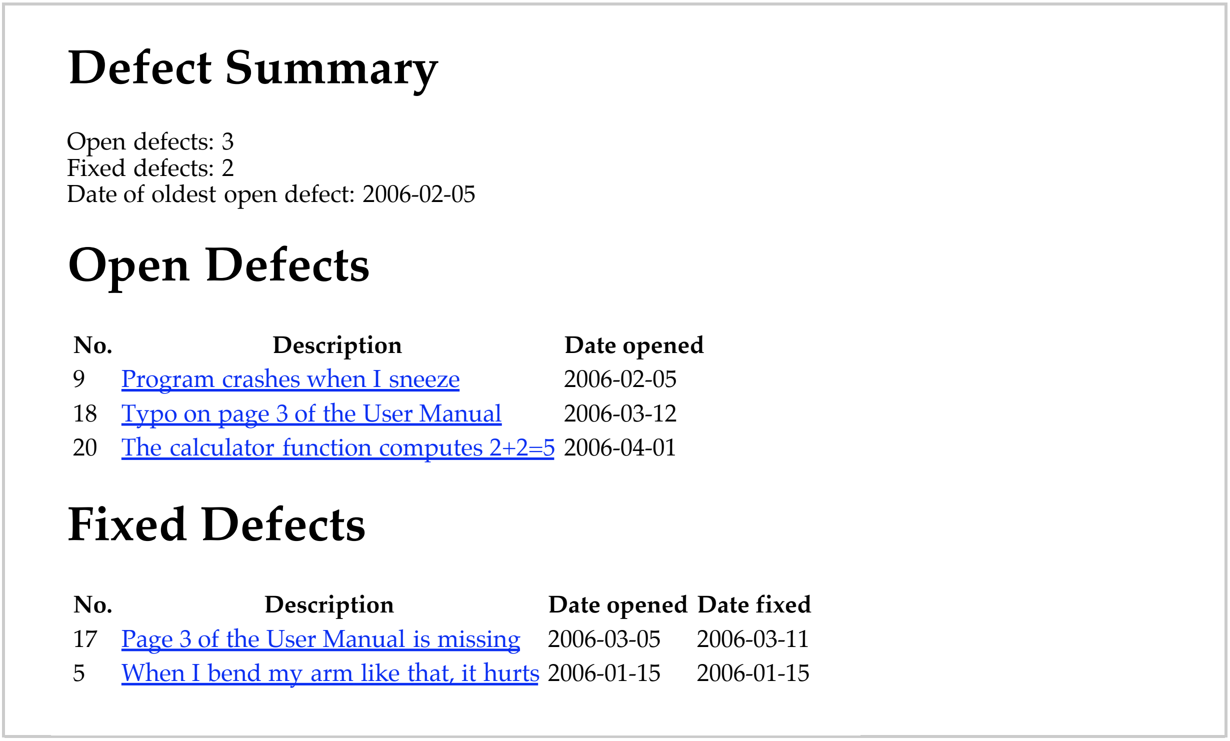
That should be pretty easy, right? And I promise, I'll only use the features I've already described. You might want to pause here and try it for yourself.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <html>
      <xsl:apply-templates/>
    </html>
```

```
</xsl:template>

<xsl:template match="DefectList">
  <body>
    <h1>Defect Summary</h1>
    <p>
    Open defects: <xsl:value-of select="count(Defect[not(Fix)])" />
    <br/>
    Fixed defects: <xsl:value-of select="count(Defect[Fix])" />
    <br/>
    Date of oldest open defect:
    <xsl:for-each select="Defect[not(Fix)]">
      <xsl:sort select="Found"/>
      <xsl:if test="position() = 1">
        <xsl:value-of select="Found" />
      </xsl:if>
    </xsl:for-each>
    </p>

    <xsl:variable name="base-url">http://ourserver/defects</xsl:variable>

    <!-- Open defects -->
    <h1>Open Defects</h1>
    <table>
      <tr><th>No.</th><th>Description</th><th>Date opened</th></tr>
      <xsl:apply-templates select="Defect[not(Fix)]">
        <xsl:sort select="@number" data-type="number" />
        <xsl:with-param name="url" select="$base-url" />
      </xsl:apply-templates>
    </table>

    <!-- Fixed defects -->
    <h1>Fixed Defects</h1>
    <table>
      <tr><th>No.</th><th>Description</th><th>Date opened</th>
        <th>Date fixed</th></tr>
      <xsl:apply-templates select="Defect[Fix]">
        <xsl:sort select="@number" data-type="number" order="descending" />
        <xsl:with-param name="url" select="$base-url" />
      </xsl:apply-templates>
    </table>
  </body>
</xsl:template>

<xsl:template match="Defect">
  <xsl:param name="url"/>
  <tr>
    <td><xsl:value-of select="@number" /></td>
    <td><a href="{$url}/{@number}.html">
    <xsl:apply-templates select="Description" /></a></td>
    <td><xsl:value-of select="Found" /></td>
    <xsl:if test="Fix">
      <td><xsl:value-of select="Fix/@date" /></td>
    </xsl:if>
  </tr>
</xsl:template>
```

```
</xsl:stylesheet>
```
Figure 6: defect-summary4.xsl

The result is a web page that looks like this:



Defect Summary

Open defects: 3
Fixed defects: 2
Date of oldest open defect: 2006-02-05

Open Defects

No.	Description	Date opened
9	Program crashes when I sneeze	2006-02-05
18	Typo on page 3 of the User Manual	2006-03-12
20	The calculator function computes 2+2=5	2006-04-01

Fixed Defects

No.	Description	Date opened	Date fixed
17	Page 3 of the User Manual is missing	2006-03-05	2006-03-11
5	When I bend my arm like that, it hurts	2006-01-15	2006-01-15

A final project

So far, we've only generated HTML output. But I claimed that XSLT was useful for transforming from one XML format to another, or even generating other output. We can certainly do that.

For instance, let's generate an excerpt of the defects list. Our manager wants a list of only the open defects, sorted in ascending order by defect number. That's easy:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="DefectList">
    <DefectList>
      <xsl:for-each select="Defect[not(Fix)]">
        <xsl:sort select="@number" data-type="number" />
        <xsl:text>
  </xsl:text>
        <xsl:copy-of select="." />
      </xsl:for-each>
    </DefectList>
  </xsl:template>
```

```
</xsl:stylesheet>
```
Figure 7: excerpt-open-defects1.xsl

Ah, but I cheated. I used `xsl:copy-of`, which I didn't tell you about. But it does what you think. I also used `xsl:text` to add some white space, since the formatting came out funny without it. Well, I did tell you there was more to XSL than I've told you about.

One last thing. Our manager wanted that excerpt in a simplified XML format of his own devising. He wants the XML to look like this:

```
<?xml version="1.0"?>
<DefectList>
  <Defect>
    <Number>18</Number>
    <DateFound>2006-03-12</DateFound>
    <Description>Typo on page 3 of the User Manual</Description>
  </Defect>
  ... etc ..
<DefectList>
```

That's easily accomplished (modulo formatting issues) with this stylesheet:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="DefectList">
    <DefectList>
      <xsl:for-each select="Defect[not(Fix)]">
        <xsl:sort select="@number" data-type="number" />
        <Defect>
          <Number><xsl:value-of select="@number" /></Number>
          <DateFound><xsl:value-of select="Found" /></DateFound>
          <Description><xsl:value-of select="Description" /></Description>
        </Defect>
      </xsl:for-each>
    </DefectList>
  </xsl:template>

</xsl:stylesheet>
```
Figure 8: excerpt-open-defects2.xsl

Summary

We've now seen how easy it is to start using XSLT. It's very easy to transform an XML source document into HTML, other forms of XML, or even just extract data into the same XML format. You can also output arbitrary text, and although I haven't shown you how, this means that you can transform XML documents into essentially any other format you can imagine.

Finally, I've mentioned that XSL is a Turing-complete language. For a proof, take a look at <http://www.unidex.com/turing/utm.htm>, where you'll find an XSL stylesheet that implements a Universal Turing Machine which is in turn programmed using TMML, the Turing Machine Markup Language. Clever and entertaining. If you're a computer science geek.

References

http://www.cafeconleche.org/books/bible2/chapters/ch17.html
http://en.wikipedia.org/wiki/Turing-complete
http://www.unidex.com/turing/utm.htm