


<oo> → <dh> Digital humanities

Maintained by: David J. Birnbaum (djbitt@gmail.com) 

Last modified: 2018-03-12T15:40:38+0000

XSLT assignment #2

The input text

For this assignment we'll be using an XML file originally prepared by a member of this class in spring 2012. We've modified it for use as an XSLT exercise, and the new version is available at <http://dh.obdudon.org/skyrim.xml>. You should right-click on this link, download the file, and open it in <Oxygen/>. We'll be using it for subsequent XSLT assignments, so keep a copy when you're done with this one. You don't need the Relax NG schema, but if you'd like to look at it, it's available at <http://dh.obdudon.org/skyrim.rnc>.

Because this document (unlike our version of *Hamlet*) is not in a namespace, you should not add the `@xpath-default-namespace` attribute.

Overview of the assignment

For this assignment you want to write an XSLT stylesheet that will transform the XML input document into an HTML document that consists entirely of tables of characters and factions. You can see the desired output at <http://dh.obdudon.org/skyrim-02.xhtml>.

Analysis of the task

The information that you want to output is all found near the top of the input document, inside the `<cast>` element. For the moment we're going to ignore everything else, including the `<body>`. You want to generate one HTML table to contain information that you'll extract from the `<character>` elements and a second table to contain information that you'll extract from the `<faction>` elements. Your XSLT, then, should proceed along the following lines:

1. In a template rule for the document node (`<xsl:template match="/">`), create the HTML superstructure. Between the start and end `<body>` tags that you'll be creating you should insert the main tags for two HTML tables, one for characters and one for factions. Between the start and end `<table>` tags for each table, you should then insert `<xsl:apply-template>` elements that select the nodes you want to process to build those tables. Be careful, though; there may be `<character>` or `<faction>` elements elsewhere in the document, such as inside `<paragraph>` elements, and for the tables you're producing you want only the ones that are inside the `<cast>` element.
2. In XSLT, processing something normally happens in two parts. You normally have an `<xsl:apply-templates>` element that tells the system what elements (or other nodes) you want to process, and you then have an `<xsl:template>` element that tells the system exactly how you want to process those elements, that is, what you want to do with them. If you find the image helpful, you

can think of this as a situation where the `<xsl:apply-templates>` elements *throw some nodes out into space and say “would someone please process these?”* and the various `<xsl:template>` elements sit around watching nodes fly by, and when they match something, they *grab it and process it*. In this case, then, your `<xsl:apply-template>` elements inside the template rule for the document node will tell the system that you want to process `<character>` and `<faction>` elements, at which point the template rule for the document node will have done its work by announcing what needs to be done. That work actually gets done by other `<xsl:template>` rules, the ones that you’ve written that match the `<character>` and `<faction>` elements.

3. In the `<xsl:template>` rules for `<character>` or `<faction>` elements you’ll need to output something for each one. That is, for each `<character>` or `<faction>` element, you’ll need to output a line in your table.
4. The values you use to populate your table cells come from attributes. Remember that attributes are on the *attribute axis*, which you can address by prefixing the name of the attribute with an “at” sign. For example, open `skyrim.xml` in `<oXygen/>`, go into the XPath browser box in the upper left, and search for `//cast/character/@id`, and you’ll retrieve all of the `@id` attribute values associated with `<character>` elements. You won’t use this exact XPath in your assignment, but it can serve to remind you how you address an attribute in XPath.

What goes where

You’ll want to create three template rules, one for the document node (`/`), one for each type of element you need to process.

In the template for the document node you create the HTML superstructure, as well as the wrappers for the tables (that is, the `<table>` elements themselves). Then, inside the `<table>` elements you’ll create the header rows, with the labels for the columns in each table, and after the header rows you’ll need to use an `<xsl:apply-templates>` element and select the data you want to use to populate that table. For the character table, the data will be the characters, and for the faction table, it will be the factions.

The actual rows in the tables for each of the characters and factions should be created in the template rules that you’ll write for those elements. Note that the stuff you do just once per table (create the table and the header row) should be in the template rule for the document node, but the stuff that has to happen repeatedly, once for each row of data, should be in the template rule for those elements (`<character>` and `<faction>`). That’s the XSLT way to ensure that you create a new row for every `<character>` or `<faction>` element in the input.

How to develop an XSLT stylesheet

The problem with trying to code everything at once and then trying to run it is that if it doesn’t work, it can sometimes be hard to find just where the error is. When we develop an XSLT stylesheet, we begin by writing a template rule for the document node (`/`), and inside that we do basic housekeeping (e.g., HTML superstructure) and include whatever `<xsl:apply-templates>` elements we need. Initially for the templates that get called by those `<xsl:apply-templates>` elements we put in a simplified placeholder, something that will produce output that may not be what we want eventually, but that will let us confirm that our templates are being called. Once the basic framework is in place (we’re calling the right templates in the right places), we then start fine-tuning the individual template rules, replacing the placeholder code with code that produces the results we actually want. The technical term for this type of placeholder is *stub*.

In this case, in the template rules for characters and faction we might start by just outputting some plain text. That won’t be valid in HTML, but it will tell you whether the templates are being called when you

want. Once that's working, you can expand it by creating real HTML rows and cells and filling each one with fixed text of some sort. That's now valid HTML, but it isn't the real content. Once you've determined that the cells are being created in the right place, you can then replace that fixed text with XSLT code that retrieves the information you actually want in your table. It may be tempting to write all of the code at once, but typically it won't all be correct the first time, and you'll save time in the long run by proceeding one step at a time. That isn't just for beginners; it's normal professional practice, and it's what we do in our own development, as well.