Using Perl to Read and Write OpenOffice Documents

By Tom Anderson t@tomacorp.com

OpenOffice is an open source suite of programs that can be used as a replacement for Microsoft Word, Excel, and PowerPoint. While OpenOffice can read and write Microsoft-compatible doc, xls, and ppt files, its native format is XML. Perl programs can take advantage of this XML format to directly read and write OpenOffice files.

This paper explains the basic organization and structure of OpenOffice XML. OpenOffice files are a zipped collection of XML files. Unzipping the file reveals its XML content. A Perl module handles this task and keeps the content organized.

General purpose Perl XML modules can read and write OpenOffice files. This presentation demonstrates these Perl XML modules in example applications.

An example application is a web service that generates PNG web banners. This example uses the OpenOffice Draw application. It automatically creates bitmap banners for the web. These banners include attractive fonts and images.

Another example program is a spelling and style checker for OpenOffice. The program evaluates the readability and spelling of an OpenOffice document.

* * *

OpenOffice is a suite of programs that is an alternative to Microsoft Office. OpenOffice can read and write Microsoft Office files, and also translate these files in its own native format. Since this native OpenOffice format is XML, the files are easily to manipulate with perl. These perl programs can overcome limitations of Microsoft Office, making previously difficult problems easy and impossible problems possible.

This paper briefly introduces the OpenOffice file format, and describes three example applications that take advantage of the XML format. Some of the code is included in the paper, and the rest is available electronically.

The OpenOffice File Format

OpenOffice and stores its files in XML. If you looked at a file to see the XML, you would be disappointed by result:

```
PK###Ќ?23&??/mimetypeapplication/vnd.oasis.opendocument.presentationPK###
Ќ?2#Configurations2/PK###Ќ?2##???#?#-
Pictures/100000000000030B0000018AC9EDC20C.png?PNG#IHDR####t#-?
#PLTE??????????????????@ ` ? ? ? ?\
```

This is because the XML is stored in several files, and these are zipped together to save space. Run unzip on them. This works even though the file doesn't have a '.zip' extension.

```
unzip mypaper.sxw
```

The document text is stored in the file content.xml. This file is also hard to read because the XML has no extra whitespace, it is one long line! Run content.xml through an XML pretty-printer to fix it up. I use xml_pp, which comes with the CPAN module XML::Twig.

```
xml_pp content.xml > pcontent.xml
```

This creates easy to read XML in pcontent.xml. I use gvim to view the XML because it provides syntax highlighting.

Here is a snippet of XML from an OpenOffice Impress presentation that was imported from PowerPoint:

```
<presentation:notes draw:style-name="dp4">
  <draw:page-thumbnail draw:layer="layout" draw:page-number="10"
              svg:height="9.684cm" svg:width="12.911cm"
              svg:x="3.074cm" svg:y="1.936cm"/>
  <draw:frame draw:layer="layout" draw:text-style-name="P2"
              presentation:class="notes" presentation:style-name="pr6"
              presentation:user-transformed="true"
              svg:height="11.624cm" svg:width="13.97cm"
              svg:x="2.54cm" svg:y="12.263cm">
    <draw:text-box>
      <text:p text:style-name="P4">
        The open prototype doesn’t have the right airflow, either.
        You need to do a real environmental test to get accurate
        results.
      </text:p>
    </draw:text-box>
  </draw:frame>
</presentation:notes>
```

Application #1: Extracting Speaker Notes from PowerPoint

I wanted to extract the speaker notes from a PowerPoint presentation, so I imported the ppt file as an OpenOffice Impress presentation, and saved the it and unzipped it as above. Here is a complete program that extracts the speaker notes from an Impress presentation and renders them in HTML format:

```
use Encode;
use XML::Twig;

my $html_notes = '<meta http-equiv="Content-type" content="text/html;'.
                 ' charset=utf-8">';
my $in_notes   = 0;

my $t= XML::Twig->new(
         twig_roots => {'draw:page-thumbnail' => \&page,
                        'text:p'              => \&notes },
 start_tag_handlers => {'presentation:notes'  => sub {$in_notes = 1;} },
   end_tag_handlers => {'presentation:notes'  => sub {$in_notes = 0;} }
                      );

$t->parsefile('pcontent.xml');
print encode('UTF-8', $html_notes);

sub page {
  my ($t, $elt)= @_;
  $html_notes .= '<H3>Slide '.$elt->att('draw:page-number')."</H3>\n";
}

sub notes {
  my ($t, $elt)= @_;
  return 0 if not $in_notes;
  $html_notes .= $elt->text."<BR>\n";
}
```

This program uses XML::Twig by Michel Rodriguez. Other parser modules will also work.

Encode, UTF-8, and the HTML header are needed to correctly render the special characters in my speaker notes. Even though I didn't type them in on purpose, PowerPoint takes liberties with quotes and other characters. It 'intelligently' translates typing into Unicode text, which is not seven-bit ASCII. I learned about “the mysterious world of character sets, encodings, Unicode, all that stuff,” from Joel Spolsky's web site. His article The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!) is available at http://www.joelonsoftware.com/articles/Unicode.html.

Complete documentation of the OpenOffice file format is available online. Instead of reading the documentation, I can usually get away with reading the XML itself. I find the tags containing the text that I need, and then use a subroutine to grab this content.

Twig handlers are subroutines that are called when an XML tag is parsed. There are different kinds of handlers. I used three types, twig_root, start_tag_handlers, and end_tag_handlers. The twig_roots calls cause XML::Twig to only build a data structure. The data structure only has the content inside the specified tags; the rest of the content is skimmed over. While it's skimming, it will still deal with the start_tag_handlers and the end tag handlers. In this program, the start tag and end tag handlers flip a flag

to say whether or not the parser is reading inside the notes section.

The speaker notes are text paragraphs that are in the presentation:notes tag, as shown in the example XML snippet.

The text paragraphs are the stuff that is inside the text:p tags. All the stuff inside the text:p tags is parsed. The start and end tag handlers keep track of whether or not the text is in the speaker notes. The text:p handler checks the state of the speaker note flag to decide whether or not to print the text.

Callbacks in XML::Twig get two arguments, a reference to the whole XML::Twig object and a reference to the current element. I called methods on the current element reference to get the text between the tags with `$elt->text()`. The only element attribute that I needed is the page number, which I access with the call `$elt->att('draw:page-number')`.

XML::Twig has over 500 methods that deal with XML text, entities, and tags. I use only a few of them. Mostly I use the methods shown in this example, especially `att()` and `text()`.

To create the input file `pcontent.xml`, open the PowerPoint ppt file in OpenOffice and save it in native Impress format. I unzipped the Impress document, and ran the enclosed content.xml file through a pretty printer. I saved it as pcontent.xml. This can be done in Perl with the module Archive::Zip by Ned Konz. It extracts files from a zip archive.

```
sub get_string_from_zip {
  my ($zip_member_fn, $zip_archive_fn)= @_;
  my $zip = Archive::Zip->new();
  if ($zip->read($zip_archive_fn) == AZ_OK) {
    return $zip->contents($zip_member_fn);
  } else {
    die "Can't find member $zip_member_fn in archive $zip_archive_fn" ;
  }
}
```

I also added slightly better HTML generation and command line argument handling to complete the program. The HTML generation routine uses the CGI module to create the HTML tags.

```
sub render_html {
  my ($html, $fn)= @_;
  my $encapsulated_html= html(
    head('<meta http-equiv="Content-Type" '.
         'content="text/html; charset=utf-8" />'."\n".
    title("Speaker Notes Extracted from $fn")."\n").
    body($html)
  );
  return encode('UTF-8', $encapsulated_html);
}
```

The complete program, with a few enhancements, is available at http://tomacorp.com/perl/oo/get_speaker_notes.html.

Application #2: Generating Web Banners

A banner across the top of a web site is a good for usability. The banner provides a hint to inform users what to expect. I wanted to create banners with slight variations for different sections of a large dynamic web site. I didn't want to create all the banners by hand, partly because we might change our mind on the colors, fonts, logo, corporate name, etc. I needed a batch process to make a few hundred banners.

OpenOffice can read and write a large variety of data formats, including native OpenOffice, PDF, and Microsoft. The OpenOffice Draw program can create various bitmap formats. To create the banners in a bitmap format, I needed to loop through a list of banners and:

- Create an OpenOffice document that contains the banner content.
- Open the OpenOffice document with OpenOffice
- Export the banner bitmap file from OpenOffice
- Close the OpenOffice document.

One easy way to create an OpenOffice document is from another document that acts as a template. Use regular expressions or your favorite template module to substitute the part of the document that varies.

The other tasks, opening the document and exporting the content, are commonly accomplished by running the OpenOffice user interface. How would I do this automatically?

Much of the functionality of the OpenOffice user interface is available to an API. There are a variety of ways to call this API. Since my needs are simple, I used the simplest approach that I thought could possibly work. I wrote a short BASIC subroutine to open, export, and close the files. I could call the subroutine from the OpenOffice command line.

```
Sub ConvertDrawToGIF( cFile )
   cURL = ConvertToURL( cFile )
   oDoc = StarDesktop.loadComponentFromURL( cURL, "_blank", 0, Array(_
            MakePropertyValue( "Hidden", True ),_
            ))
   cFile = Left( cFile, Len( cFile ) - 4 ) + ".gif"
   cURL = ConvertToURL( cFile )
   oDoc.storeToURL( cURL, Array(_
               MakePropertyValue( "FilterName", "draw_gif_Export" ),_
            )
   oDoc.close( True )
End Sub
```

I learned this approach from Danny Brewer, who has written extensively on OpenOffice BASIC programming. I started from one of his examples that is also quite useful by itself. It is a program that will open a wide variety of input formats, and create an PDF of the input. See http://www.oooforum.org/forum/viewtopic.phtml?p=62219#62219 for details. The OpenOffice community provides friendly support for new users of BASIC macros. I was able to find examples that were close to what I needed. The website http://www.ooforum.org/ has many examples.

To debug my code and see the results of different approaches, I used the MsgBox() function. It creates a popup window where I could write out variable values.

One quirk of the bitmap export function is that it creates a bitmap that is the size of the paper. If the

content is selected, the bitmap is the size of the selected objects. I wasn't able to figure out how to select the content in BASIC and take advantage of the reduced bitmap size. Instead, I redefined the 'User' page size to be the right size for my banner. This page size would work for a short, wide strip of paper, but I only use it to make a banner with the correct size.

The BASIC macro can be stored either in the document or with the OpenOffice application. When stored with the document, the code can be found in an XML file that is included in the document's zip archive. I stored the macro in the OpenOffice application instead of in the document.

I ran the command line conversion process with a batch file called `convert_draw_to_gif.bat`:

```
"c:\program files\OpenOffice.org1.1.4\program\soffice" -invisible
"macro:///Standard.Module1.ConvertDrawToGIF(%1)"
```

When I ran this batch file from the command line, I needed to use the full path to the sxd file.

```
convert_draw_to_gif c:\es\OOO\OOt.sxd
```

Next I wrote a Perl sub that is equivalent to the batch file. It uses backticks to launch OpenOffice.

```
#!/usr/bin/perl

my $fn='c:/es/paper/paper/OOO/PGBANNER/shapes3D.sxd';
create_gif_from_sxd($fn);

sub create_gif_from_sxd
{
  my ($fn)= @_;
  my $soffice= 'c:/program files/OpenOffice.org1.1.4/program/soffice';
  my $macro  = "macro:///Standard.Module1.ConvertDrawToGIF($fn)";
  my $status = `"$soffice" -invisible "$macro"`;
  return $status;
}
```

You can wrap this code in a web service, or use it in a process to create a bunch of banners. Example banners and the complete code to generate them is available at http://tomacorp.com/perl/oo/banners.html.

Application #3: A Spell and Style Checker in Perl

Spell checkers and style checkers are great tools for improving my writing. But when I check an article like this one, the spell checker insists that there are spelling errors in my perl code. By creating my own spell checker, I can fix this problem.

There are a variety of spell-checkers to choose from, and I wanted one that is multiplatform, easy to install, and easy to customize. After looking around, I didn't find a spell checker that I liked, but I did find the collection of word lists at http://wordlist.sourceforge.net/. I chose SCOWL (Spell Checker Oriented Word Lists) for my spell checker.

The procedure for spell checking is simple:

- Read the entire dictionary and load the words into the keys of a big hash.
- Loop through the lists of words in the document, and check if they are in the dictionary.
- Generate a report of the words that are not in the dictionary.

The hard part was realizing that perl would be fast enough and that the memory requirements would be acceptable to read the whole dictionary, which contains 687,101 words. It takes about 12 seconds to load the list into a hash on an old slow computer, and under 2 seconds on a reasonably fast machine.

The SCOWL word list is split into files based on the rarity of word usage. I used this rarity information as the value of the words in the hash. Here is a small subset of the resulting data structure:

```
my $dict = {
  'or'   => 'english-words.10',
  'Perl' => 'special-hacker.50',
  'ort'  => 'english-words.70',
  'supercalifragilisticexpialidocious' => 'english-words.70',
}
```

Common words have a lower number at the end of the file name. This enabled me to create a report which flags rare words. This helps when an uncommon word is a misspelling of a common word, such as 'ort' for 'or'.

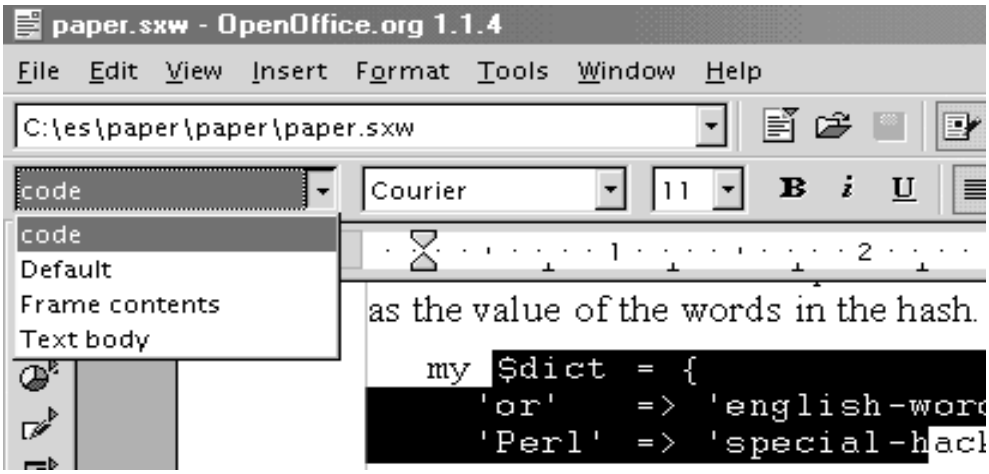

Illustration 1 Using a Paragraph Format in OpenOffice

When I check the spelling of a paper such as this one, the spell checker flags errors in the perl code. To fix this, I made the spell checker recognize the paragraph style, so that it won't check code. I defined a

paragraph style called 'Text Body' for my writing. I made more paragraph styles for the perl code, the OpenOffice BASIC code, and other non-English languages. Illustration 1 shows the paragraph format tools in OpenOffice.

A modification to the XML parser separates the code from the English text.

```
sub text_out {
  my ($t, $elt)= @_;
  if ($elt->att('text:style-name') eq 'Text body') {
    $t->{_spellcheck_text} .= $elt->text."\n";
  }
  elsif ($elt->att('text:style-name') eq 'code') {
    $t->{_code_text} .= $elt->text."\n";
  }
}

sub get_text_content {
  my $self= shift;
  return $self->{_spellcheck_text};
}

sub get_code_content {
  my $self= shift;
  return $self->{_code_text};
}
```

The syntax checking routine runs `perl -c` on the extracted code. The `-c` option checks the syntax of the perl code and exits without running the main body of the perl code. Don't run untrusted code, though, because some of the code is run. To learn about the `-c` option, I read the perl documentation at http:://perldoc.perl.org.

If there are syntax errors, `perl -c` prints them to `STDERR`. I used the method of capturing `STDERR` using `IPC::Open3` from an external program as described in the Perl FAQ, also available at http://perldoc.perl.org.

Kim Ryan's module Lingua::EN::Fathom provides a assessment of writing that measures English writing and evaluates various metrics. It calculates the readability statistics Fog, Flesch, and Kincaid. I called this module to produce these statistics.

I also created a module to flag long sentences. These tend to find their way into my writing, and are often harder to read than I intended.

The complete style and spell checker is available at http://tomacorp.com/perl/oo/style_and_spell.html. I encourage you to modify it for your own needs, and to create your own OpenOffice tools with Perl.