Docx4j output conversion - design

*By Alberto Zerolo, Jason Harrop*

This document gives a brief overview of how docx4j converts a docx to PDF or HTML. Conversion of pptx to svg in html is similar.

The intended audience is developers working to improve docx4j's output capability by improving support for existing features, or implementing support for additional features of the docx file format.

# Introduction

Docx4j supports output to PDF and XHTML.

The PDF output is done via XSL FO. An XSL FO processor (for example, Apache FOP) is then used to convert the XSL FO to PDF. (In the past, docx4j has supported other methods of producing PDF output (for example, via iText), but that code is no longer maintained.)

Historically XSLT was used to transform to XSL FO and XHTML. In both cases, the XSLT used Xalan Java extension functions to call back into docx4j's Java code to do a lot of the work.

More recently, docx4j has also supported generating XSLFO and XHTML, without using Xalan/XSLT. This approach has several benefits:

* It works on Android, where the Xalan approach is problematic
* It is faster
* It has no dependency on Xalan

It is envisaged that once the non XSLT code reaches feature parity with the XSLT approach, that the XSLT approach will be retired.

However at present, both approaches are maintained.

So we have:

|  |  |  |
| --- | --- | --- |
|  | via XSLT | nonXSLT |
| PDF |  |  |
| XHTML |  |  |

For all of these, the common workflow can be found in <https://github.com/plutext/docx4j/blob/master/src/main/java/org/docx4j/convert/out/common/AbstractExporter.java>

convert.out.common.AbstractExporter.export performs the following steps:

* **preprocess**(conversionSettings);  
    
  AbstractWmlExporter does this (for WordprocessingMLPackage only), by calling Preprocess.process.  
    
  That method executes features, for example:

**if** (features.contains(*PP\_COMMON\_COMBINE\_FIELDS*)) {

FieldsCombiner.*process*(ret);

}

**if** (features.contains(*PP\_COMMON\_MOVE\_BOOKMARKS*)) {

BookmarkMover.*process*(ret);

}

**if** (features.contains(*PP\_COMMON\_MOVE\_PAGEBREAK*)) {

PageBreak.*process*(ret);

}

**if** (features.contains(*PP\_PDF\_COVERPAGE\_MOVE\_SECTPR*)) {

CoverPageSectPrMover.*process*(ret);

}

**if** (features.contains(*PP\_COMMON\_CONTAINERIZATION*)) {

Containerization.*process*(ret);

}

**if** (features.contains(*PP\_PDF\_APACHEFOP\_DISABLE\_PAGEBREAK\_FIRST\_PARAGRAPH*)) {

FopWorkaroundDisablePageBreakOnFirstParagraph.*process*(ret);

}

**if** (features.contains(*PP\_PDF\_APACHEFOP\_DISABLE\_PAGEBREAK\_LIST\_ITEM*)) {

FopWorkaroundReplacePageBreakInEachList.*process*(ret);

}

**if** (features.contains(*PP\_COMMON\_TABLE\_PARAGRAPH\_STYLE\_FIX*)) {

ParagraphStylesInTableFix.*process*(ret);

}

* **createWrappers**(conversionSettings, preprocessedPackage);  
    
  This creates Section containers to facilitate hierarchical processing better that sectPr point tags.  
    
  AbstractWmlExporter does this (for WordprocessingMLPackage only),   
  by calling CreateWrappers.process,   
  which in turn calls ConversionSectionWrapperFactory class.
* **createContext**(conversionSettings, preprocessedPackage, sectionWrappers);  
    
  AbstractFOExporter creates an FOConversionContext  
    
  AbstractHTMLExporter3 creates an HTMLConversionContext
* **createIntermediateOutputStream**(outputStream);  
    
  AbstractFOExporter just creates a ByteArrayOutputStream, which will ultimately contain FO
* **process**(conversionSettings, conversionContext, intermediateOutputStream);

**public** **abstract** **class** AbstractWmlExporter<CS **..** **extends** AbstractExporter<CS, CC, WordprocessingMLPackage> contains the process method:

**exporterDelegate.process**(conversionSettings, conversionContext, outputStream)

AbstractWmlExporter gets its exporterDelegate object via its constructor which takes a AbstractExporterDelegate object. That declares a single method *process, which does the core conversion****.*** The method is abstract though; the class is extended by:

* + **AbstractXSLTExporterDelegate (contains the process method** which performs the XSLT transform), and is extended by WmlXsltExporterDelegate, which is in turn a field of:
    - FOExporterXslt
    - HTMLExporterXslt
  + **AbstractVisitorExporterDelegate (non XSLT) , which contains the process method**,  
    and is extended by:
    - FOExporter**Visitor**Delegate
    - HTMLExporter**Visitor**Delegate

Those invoke superconstructor with a FO|HTMLExporterVisitorGenerator

* **postprocess**(conversionSettings, conversionContext, intermediateOutputStream, outputStream);   
    
  AbstractFOExporter calls the FO renderer to perform the render step here (ie FO to PDF or whatever). FO renderer (FORenderer is an interface, which FOP or other renderer sits behind).

# Headers/Footers generally, and LayoutMasterSetBuilder

See /docs/headers\_footers.docx – read that first!

# Word versus XSL FO

If there are different even and odd headers/footers you can do it the fo-way (calling them even/odd) or the winword-way (calling them even/default). But there is a small problem if the LayoutMasterSetBuilder outputs the stuff with the fo-naming but the xslt uses the winword-naming (the "odd" part is commented out). I have changed everything to use the winword-conventions.

If there are even/default headers but only default footers, then the HeaderFooterPolicy will now let the even footer reference the default footer (and the other way around). In Word headers and footers are independent but in the fo structure it is an even/odd page, that includes headers and footers. If the HeaderFooterPolicy wouldn't act this way, then on the even page the footer (or header) would be missing.

# Different count of sections

if the count of the sections in the document model and the xslt/non-xslt are different("continuous"-sections), then the section index will access the wrong section.

For this reason I made a subclass of the SectionWrapper (ConversionSectionWrapper) that corresponds to the sections in the xslt (i.e. without continuous sections). These sections are now used throughout the conversion process. Obviously any settings of the continous SectPr are lost. As a nice side effect, the handling of sections in the XSLFOExporterNonXSLT is simplified.

# Concepts

## Static entry points

With “static entry points” I’m referring to the static methods that get called from the XSL documents for a transformation. These are the places where the transformation calls back into the Java code.

* **Design principle:** *All entry points should be within a class in the convert.out-packages. If it is a common entry point, i.e. it get’s called from more than one XSLT then it goes into the Converter class.*
* The first parameter is always the ConversionContext.

I didn’t move the entry points of the WordXmlPictureE10 and WordXmlPictureE20 because I expect them to be replaced with ModelConverter (all the code is there, it only needs to be rewrapped).

Rationale:

* The XSLT depend on these entry points, by moving them into the convert.out-packages they are nearer to the place they get used.
* This reduces the clutter in those classes that represent the model.
* By having the ConversionContext as a parameter it would increase the dependencies in those model classes and this can be avoided by keeping the methods out of the model classes.
* These functions are “technical” functions for the output, again there is no need to have them in the model.

## ConversionContext and ConversionSettings

* The ConversionContext is a private class in the conversion process.
* The ConversionSettings are a class the user passes to the conversion process.

**Design principle:** *the user is the owner of ConversionSettings and therefore the conversion process shouldn’t change this instance.*

For this reason some of the information of the ConversionSettings is copied into the ConversionContext and then changed there, and this is also the reason why the ConversionContext has a method getXSLTParameters.

## Data in ConversionContext and TransformState

The Map with the TransformState(s) and the ConversionContext are two ways of passing data around. The advantage I see with the ConversionContext, is that it makes explicit, what data it has. In the IDE you can check anytime the references to a getter, you can’t do that with a map value.

**Design principle:** *if the data is private data of a ModelConverter, i.e. it’s a 1:1 relation, it goes is done as a TransformState. If the data get’s used in several ModelConverter or anywhere else it goes into the ConversionContext.*

**Design principle:** *There is only one instance of the Writer, it should never keep a reference to a WordprocessingMLPackage, as it would cause issues in a multithreading environment. Instead it should use the WMLPackage, which gets passed as part of the ConversionContext.*

Apart from build, the ModelConverter has now two new methods: getID and createTransformState. Having a getID ensures that the Writer is managed under it’s own type (w:sym, w:tbl,…) and createTransformState ensures that it gets the TransformState it expects.

## org.docx4j.convert.out.AbstractConversionContext

* Root of the different ConversionContext
* Contains references to the OpcPackage, ImageHandler and MessageWriter

## org.docx4j.convert.out.AbstractMessageWriter

* Root of the different MessageWriter
* The corresponding MessageWriter implementation adds it’s dialect (html, fo)
* One instance per output type (html, fo)
* The implementations are done as anonymous classes in the corresponding ConversionContext

## org.docx4j.convert.out.AbstractModelRegistry

* Root for the different ModelRegistry
* The corresponding ModelRegistry implementation register their ModelConverter (html, fo)
* One instance per output type (html, fo)
* The implementations are done as anonymous classes in the corresponding ConversionContext

## org.docx4j.convert.out.AbstractSymbolWriter

* Superclass for the (fo)SymbolWriter and (html)SymbolWriter.
* Implementation of getID, createTransformState and SymbolModelTransformState
* Question: Why not remove the SymbolModelTransformState if it doesn’t get used? If somebody needs an example he can look at the TableModelTransformState.

## org.docx4j.convert.out.AbstractTableWriter

* Implementation of the new interface
* Use of ConversionContext
* Moved TableModelTransformState here

## org.docx4j.convert.out.AbstractWmlConversionContext

* ConversionContext to process WordprocessingMLPackage.
* Adds methods for the ModelRegistry and TransformStates
* Implements the PartTracker, footnote and endnote numbering

## org.docx4j.convert.out.Converter

* Moved the management of the Models to the ModelRegistry
* Moved common static functions, that get called from the xslt transformations into this class.

## org.docx4j.convert.out.html.AbstractHtmlExporter

* Use of ConversionContext
* Moved common entry points to Converter (resolveHref)
* Moved common functions (NonXSLT, SvgExporter…) to HtmlCssHelper

## org.docx4j.convert.out.html.docx2xhtmlNG2.xslt

* Replace wmlPackage, modelStates, imageHandler with ConversionContext
* Question: Where are the parameters fontMapper, fontFamilyStack, conditionalComments used? Remove if unused?

## org.docx4j.convert.out.html.HTMLConversionContext

* ConversionContext for HTML output
* Adds the FontMapper (where does it get used?)

## org.docx4j.convert.out.html.HTMLConversionContextNonXSLT

* ConversionContext for HTML NonXSLT output
* Adds the StyleTree to the default HTMLConversionContext

## org.docx4j.convert.out.html.HtmlCssHelper

* Common HTML CSS Functions that get used from the AbstractHtmlExporter, HtmlExporterNG2, HtmlExporterNonXSLT, SvgExporter

## org.docx4j.convert.out.html.HtmlExporterNG2

* Use ConversionContext
* Delegate to HtmlCssHelper
* Moved Footnote/Endnote numbering to AbstractWmlConversionContext, entry points for XSLT in Converter.
* message and notImplemented are replaced with the MessageWriter in the ConversionContext.

## org.docx4j.convert.out.html.HtmlExporterNonXSLT

* Use ConversionContext
* Delegate to HtmlCssHelper
* Use the toNode of the ModelRegistry to generate table
* message and notImplemented are replaced with the MessageWriter in the ConversionContext.

## org.docx4j.convert.out.ModelConverter

* ModelConverter is now an interface
* *ModelConverter should not keep a reference to the WMLPackage*
* getID and createTransformState are used in the ModelRegistry to ensure a consistent state ID – ModelConverter –TransformState
* Common toNode for XSLT/NonXSLT uses

## org.docx4j.convert.out.pdf.viaXSLFO.Conversion

* Use ConversionContext
* message and notImplemented are replaced with the MessageWriter in the ConversionContext.

## org.docx4j.convert.out.pdf.viaXSLFO.docx2fo.xslt

* Replace wmlPackage, modelStates, imageHandler with ConversionContext
* Question: Where are the parameters substituterInstance, fontFamilyStack, docxWikiMenu, docID used? Remove if unused?

## org.docx4j.convert.out.pdf.viaXSLFO.InField

* Static entry points for XSLT are now in the Converter class
* InField is now used in the AbstractWmlConversionContext, it isn’t any longer a TransformState

## org.docx4j.convert.out.XSLFO.XSLFOExporterNonXSLT

* Use ConversionContext
* Use the toNode of the ModelRegistry to generate table
* Question: Why not put XSLFOExporterNonXSLT into the convert.out.pdf.viaXSLFO-package and call it PdfExporterNonXSLT? It’s probably easier to add later functionality than to move and rename it once it get’s used.

## org.docx4j.model.Model

* Changed the signature of build to accept a JAXB instead of a Node and a as content a Node instead of a NodeList. This way it can be used the same way from XSLT and NonXSLT.