Using XSLT in Adobe FrameMaker 7.2

TABLE OF CONTENTS

1 Structured content, XML, and XSLT
2 XSLT in a nutshell
6 How XSLT fits into the Structured FrameMaker environment
7 Use cases for FrameMaker and XSLT
7 Tutorials
28 Summary
28 Recommended next steps

The combination of Adobe® FrameMaker®, XML, and XSLT provides a powerful and flexible platform for authoring and publishing a wide variety of content, from manuals or maintenance documentation to research reports. This white paper focuses on the XSLT features of FrameMaker 7.2, which extend the core XML authoring capabilities of FrameMaker with greater interoperability and publishing power than ever before.

This document complements existing documentation on the XML features of FrameMaker. You can find an introduction to XML and FrameMaker at www.adobe.com/products/framemaker/pdfs/FM_XML.pdf, and an explanation of how to migrate unstructured FrameMaker content to structured FrameMaker at www.adobe.com/products/framemaker/pdfs/migrationguide.pdf.

Structured content, XML, and XSLT

The XSLT features of FrameMaker are only available if you are using structured FrameMaker. With the release of FrameMaker 7.0, Adobe combined what had been two separate applications into one. Now, both structured and unstructured authoring modes are provided in one package, and changing between those modes is a user Preference setting.

How XSLT extends the capabilities of FrameMaker

XSLT extends the capabilities of FrameMaker in several ways, mainly by enhancing preprocessing and postprocessing during the authoring and publishing process, including:

- Providing general transformation of XML on import and export.
- Replacing or augmenting FrameMaker Developer Kit (FDK) import/export clients.
- Replacing or augmenting read/write rules.

In general, XSLT comes in handy when you need to restructure the XML you are authoring in FrameMaker. Typical examples would include preprocessing source XML into a structure ideal for authoring and publishing with FrameMaker, or postprocessing to publish in a specific presentation form such as VoiceXML or XHTML.

To understand how XSLT can be used in FrameMaker, you need to know some basics of XML, XSLT, and structured FrameMaker. We'll cover this in the following four pages; however, if you're already familiar with these concepts, you can jump directly to the section "How XSLT fits into the Structured FrameMaker environment" on page 6.

Why structure your content?

Structuring content has many benefits:

- It allows content to be tagged according to its meaning (for example, a warning can be tagged as *warning* and a note can be tagged as *note*), and content can be hierarchically structured (books contain chapters; chapters contain sections; sections contain paragraphs, lists, and tables, and so on).
- It allows you to embed metadata into the content (an element may have certain *attributes*). For example, a list may be *ordered* or *unordered*; a procedure might be *easy*, *medium*, or *hard*; a feature description might be for the *North American* version or for the *European* version.

Adobe

- It allows the separation of content from formatting, and formatting may be applied automatically using a template. For example, the same XML source content can be laid out for print or output as HTML, depending on the template used.
- It facilitates the reuse of content. Document components authored in FrameMaker can be stored in a content management system and reused across multiple documents; a change in one component can ripple through a whole body of documents.
- It enables collaborative authoring. By using FrameMaker and a content management system, multiple authors can maintain document components independently.

The combination of these factors can lead to significant cost and time savings. For example, in localization, the reuse of components means less content needs to be translated for each new publication. Also, content can be automatically formatted (for multiple media types, if need be) using templates once it has been translated.

Hierarchical structure and embedded metadata can be used to automatically filter content for personalization. Once filtered, that content can be automatically formatted using templates.

XML-based structured authoring may demand significant up-front setup, but can pay for itself many times over for organizations with substantial information publishing needs.

Why use XML to represent your structured content?

XML is an open standard defined by the World Wide Web Consortium (W3C) for creating special-purpose markup languages. The XML specification is available at www.w3.org/TR/REC-xml/.

While it is possible to store structured content in formats other than XML, there are an ever-increasing number of advantages to using XML:

- The standards-based nature of XML means that a wide range of software tools can perform processes on documents, as long as they conform to XML standards.
- XML offers a validation mechanism (testing the conformance of documents against schemas or DTDs) that helps maintain compliant structure.
- Industry-standard XML schemas let organizations communicate more effectively.

Since attaining Recommendation status from the W3C in 1998, XML has permeated document- and data-centric software. Applications such as FrameMaker continue to evolve in order to offer state-of-the-art XML authoring capability.

Why use XSLT to process XML?

In the course of exchanging XML content between systems, the need to restructure or transform source XML content arises from time to time. In publishing to the Web, for example, source XML content may be transformed to a presentation form such as XHTML, for rendition in a browser. XSLT, another specification defined by the W3C, offers a language for such transformation. FrameMaker 7.2 supports XSLT.

XSLT in a nutshell

According to the XSLT 1.0 specification (at www.w3.org/TR/xslt), XSLT is a "language for transforming XML documents into other XML documents." XSLT is very useful for various tasks, for example, filtering documents, translating grammars, merging documents, sorting elements, or restructuring documents.

In this paper, "XSLT" is used to represent not only the language, but also an instance of a transformation.

One of the original applications for XSLT was to apply formatting to XML documents. XSLT programs or scripts are still called *stylesheets* to this day, even when they are designed for purposes other than formatting.

XSLT transformation occurs when an XSLT stylesheet and an input XML file are passed to an XSLT processor (also know as a *parser*). The XSLT processor applies the XSLT stylesheet to the input, and outputs the result.

An XSLT stylesheet is itself an XML document and is made up of template elements that define rules for processing nodes of the source input tree.

Template elements are fragments of code that specify which nodes they process and how to output the result of processing those nodes. This style of programming is known as *pattern matching* and is quite different from the procedural programming model that most programmers are familiar with. Languages that follow the pattern matching model often use *recursion*—a function that calls itself as a way to process a sequence of related elements.

A simple example

While you won't learn XSLT from this paper alone, a simple example can show the basic concept. See the end of this paper for resources related to learning XSLT.

Here's the input:

```
<?xml version="1.0"?>
<document>
  <section>
    <paragraph>Some text.</paragraph>
    <comment>Some comment.</comment>
  </section>
</document>
```

Here's an XSLT that outputs a version of the XML with shorter element names and the *comment* element filtered out:

```
<?xml version="1.0"?>
<xsl:transform version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="document">
    <doc>
      <xsl:apply-templates/>
    </doc>
  </xsl:template>
  <xsl:template match="section">
    <sect>
      <xsl:apply-templates/>
    </sect>
  </xsl:template>
  <xsl:template match="paragraph">
    <para>
      <xsl:apply-templates/>
    </para>
  </xsl:template>
  <!-- filter out the comment: -->
  <xsl:template match="comment"/>
</xsl:transform>
```

Without additional specification, a default template is applied that passes all text and attribute nodes to the output tree.

The XSLT contains four *template* elements. Each has a *match* attribute that specifies the element to which the template applies. For example, the *template* element `<xsl:template match="document">` will process the *document* element of the source XML instance.

Note: *The clause match="document" utilizes a language known as XPath, which we will discuss briefly in the next section.*

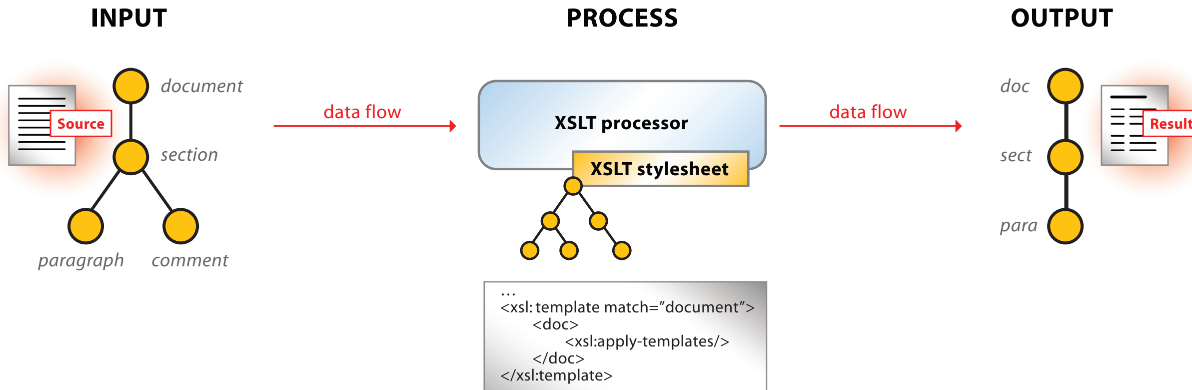
When this script is run, the XSLT processor applies the treatment that matches the *document* element. The processor outputs the start tag of the *doc* element and then calls `<xsl:apply-templates/>` to process the children of the *document* element. Once that's completed, the template outputs the end tag of the *doc* element. This effectively replaces the *document* element with the shorter *doc* element name. The next two *template* elements in the script are invoked by matching against the other elements in the source XML, which produces a similar result.

The last *template* element is different. It processes the matching *comment* element and outputs nothing, effectively causing the *comment* element to be omitted from the output.

Here's the result of applying the XSLT to the input document:

```
<?xml version="1.0"?>
<doc>
<sect>
    <para>Some text.</para>
  </sect>
</doc>
```

The XSLT processor plus stylesheet can be thought of as a unit through which the data flows, as in the following illustration.



The XSLT processor plus stylesheet can be thought of as a unit through which the data flows.

Let's look at an example that performs a more useful transformation. Suppose we have a "howto" document marked up with some ad-hoc XML language:

```
<?xml version="1.0" encoding="UTF-8"?>
<howto>
  <name>How to turn off the light</name>
  <steps>
    <step>Find the switch.</step>
    <step>Flick it.</step>
  </steps>
</howto>
```

Here is the XSLT to transform the XML to XHTML (some lines broken to fit):

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns="http://www.w3.org/1999/xhtml">
  <xsl:output indent="yes"
  doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"
  doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"/>
  <xsl:template match="/howto">
    <html>
      <head>
        <title>
          <xsl:value-of select="name"/>
        </title>
      </head>
      <body>
        <xsl:apply-templates/>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="name">
    <h1>
      <xsl:apply-templates/>
    </h1>
```

```
  </xsl:template>
  <xsl:template match="steps">
    <ol>
      <xsl:apply-templates/>
    </ol>
  </xsl:template>
  <xsl:template match="step">
    <li>
      <xsl:apply-templates/>
    </li>
  </xsl:template>
</xsl:transform>
```

Here's the resulting XHTML output (some lines broken to fit):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>How to turn off the light</title>
</head>
<body>
  <h1>How to turn off the light</h1>
  <ol>
    <li>Find the switch.</li>
    <li>Flick it.</li>
  </ol>
</body>
</html>
```

In this example, you'll see how the XSLT contains HTML markup (passed through to the output as-is), and *value-of* and *apply-templates* elements are used to insert data from the XML source into the output. An alternative approach would be to use an instruction such as `<xsl:element>` to directly create XML markup.

XPath

XPath is a major component of XSLT. It is often used for addressing objects in the input XML document (nodes in the input tree). For example, if a template should be applied to each *para* element inside a *section* element, the first line of that template could look like this:

```
<xsl:template match="para[ancestor::section]">
```

The XPath in the *match* attribute says that the template should be applied to all nodes that match this pattern. XPath treats an XML document as a tree of nodes. An XML file can be treated very much like a directory within a file system, with a single slash *(/)* representing the root element. An XPath expression can be provided as either a path from the root of an XML document, starting with /, or by using a relative expression from the current context node.

For more detail about XPath syntax and usage, please see the XPath specification at www.w3.org/TR/xpath.

How XSLT fits into the Structured FrameMaker environment

FrameMaker provides two points at which XSLT transformations can be applied to source XML content: on import (preprocessing) and on export (postprocessing). To understand how to specify such XSLT processing, it is useful to briefly consider the structured FrameMaker environment.

FrameMaker environment overview

Working with a source XML document in FrameMaker requires a set of associated files:

- A DTD or schema defining document structure.
- XSLT stylesheets (optional).
- Read/write rules, specifying changes to the rules FrameMaker uses when mapping between the structured content expressed in XML and the structured FrameMaker document model (including tables, cross-references, markers, graphics, and other familiar document objects).
- FDK import and export clients (optional). When custom clients are not specified, FrameMaker uses the default import/export client.
- A structured template, including an element definition document (EDD) that specifies the elements in your structure and rules for how they are formatted in FrameMaker.
- A structured application, listing all associated files related to the XML application, and setting configuration parameters.

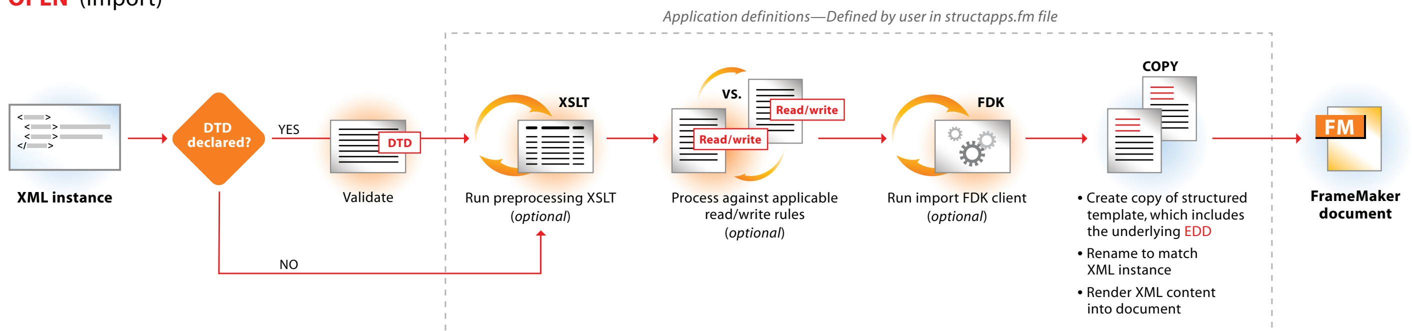
In combination, and sometimes in conjunction, these files define how FrameMaker imports and exports XML.

Dataflow within a structured application

XSLT transformations are applied either on import or export in FrameMaker. The XSLT transformations can be specified in either the source XML file or in the structured application.

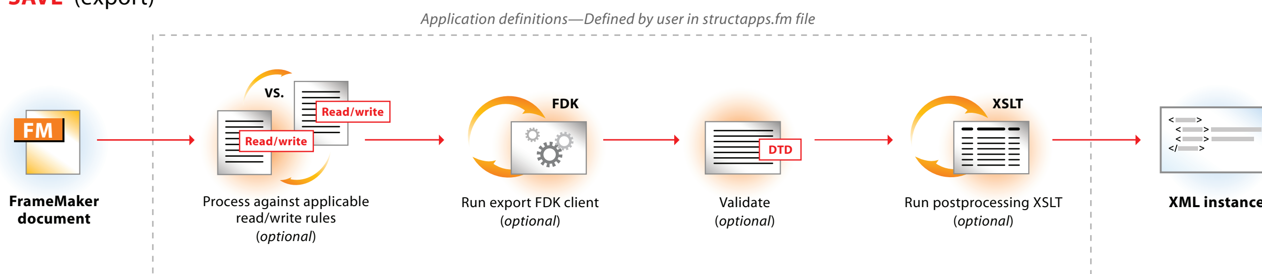
If XSLT transformations are specified on import, they are performed prior to the application of read/write rules. On export, such transformations are performed subsequent to the application of read/write rules.

OPEN (import)



Data flow through FrameMaker application components as an XML instance is opened (imported)

SAVE (export)



Data flow through FrameMaker application components as an XML document is saved (exported)

Specifying XSLT files for import and export

The XSLT used to preprocess an XML document on import can be specified in one of two ways:

- In the source XML itself, via a Processing Instruction (PI). PIs can reference either XSLT embedded in the XML document or an external XSLT stylesheet.
- In the application definitions of the structured application.

For complete details, please see the *Structure Application Developer's Guide*. The tutorial in this paper provides examples of specifying XSLT for import and export.

Use cases for FrameMaker and XSLT

There are many use cases where XSLT might be of value, and the number of situations in which XSLT is used in FrameMaker authoring is bound to expand as XML systems proliferate and new use cases are discovered. Here are four examples:

- **Mapping between an optimized authoring environment and the stored structure**

  One primary use case is that of maintaining the structure of source XML (for purposes of data exchange, consistency, open-source tool integration, etc.) while providing an optimized structure within the authoring environment. In this case, XSLT can preprocess and postprocess source XML to enable the authoring structure to differ from the structure of the XML file saved to disk.

- **Filtering and transforming data**

  Another common use case of XSLT is in filtering or transforming data. The processing power of XSLT can enable users to:

  ➤ Filter imported XML to exclude extraneous content.

  ➤ Sort items in a table or glossary.

  ➤ Transform output streams to feed other systems.

  There are almost unlimited potential uses of XSLT as a utility. In some cases, the processing capabilities of XSLT overlap with features built into FrameMaker.

- **Automating publishing workflows**

  Probably the most common use of XSLT is for publishing to the web. Here, XSLT can transform source XML to a variety of output formats ranging from presentation formats such as XHTML, HTML, or SVG, to semantic formats such as RSS.

  XSLT also plays a role in aggregating content for publishing. In workflows centered around FrameMaker as a page-composition engine, for example, content from multiple structures can be transformed with XSLT into structures required for formatted output.

- **Integrating with content management systems—assembling and splitting (chunking)**

  When authoring XML in a collaborative environment, it is common practice to divide XML content into chunks and work with, for example, individual sections of a document one at a time. In such cases, XSLT can chunk a document into its components, and reassemble the components back into a single document.

Tutorials

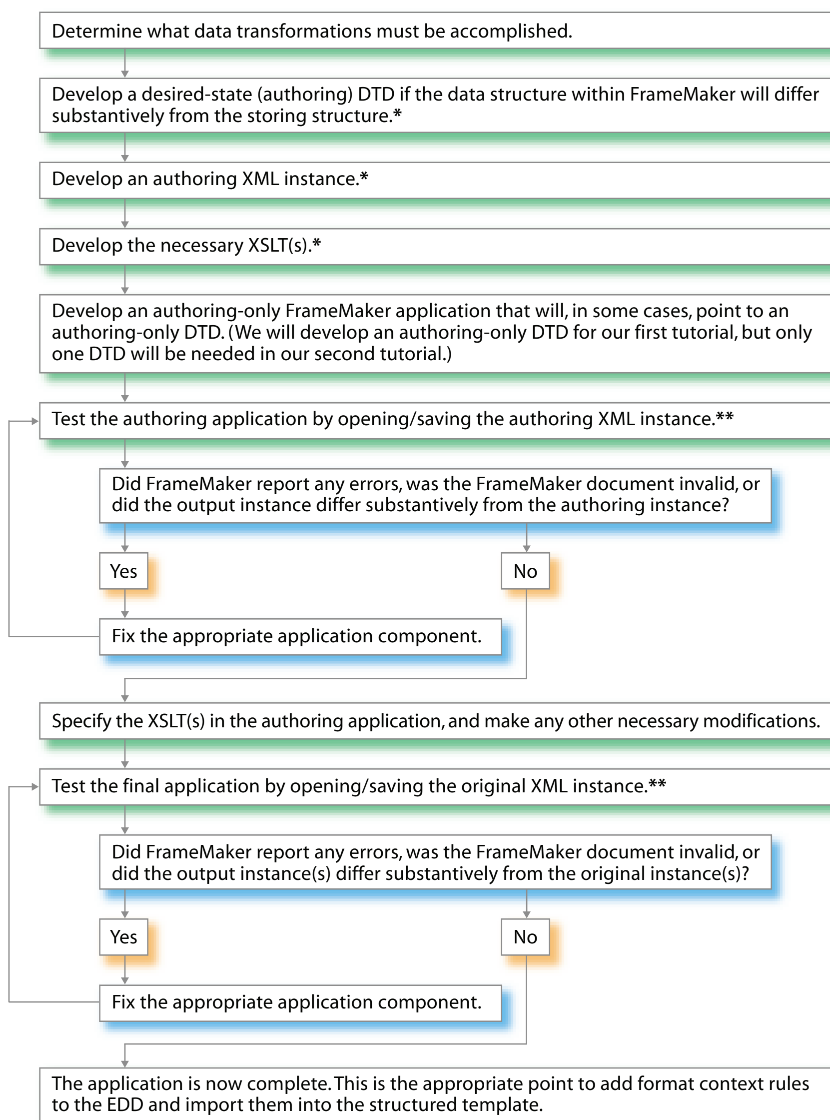
The remainder of this document comprises two step-by-step tutorials that show how to use XSLT with FrameMaker 7.2 in two common use cases. The first use case is the simpler one: it deals with a common scenario where the optimum model for authoring structured content is different from the way structured content is stored and exchanged. The second use case is common when working with content reuse and a content management system; it deals with a scenario where content is stored and managed in discrete components, but it is authored and published as a single publication.

These tutorials explain each step in detail. Familiarity with XML, DTDs, XSLT, and how structured FrameMaker works is not a requirement, but it would be helpful.

Of course, the number of steps involved in creating a FrameMaker application increases when you include XSLT transformations, but to avoid any big surprises, we will take this step by step. Although there is more than one way to develop our FrameMaker applications, for both clarity and accuracy we'll be using the following general process:

- **Examine a case that requires the inclusion of XSLTs in our process.** We will look at how altering our data structure can benefit FrameMaker authoring, and we will determine what kind of transformation must be accomplished by the XSLTs during Open and Save.
- **Develop and test a preliminary *authoring-only* FrameMaker application and XML instance.** We will develop and test our authoring environment. To avoid the complications that can arise by having too many variables during testing, we will initially limit our testing to include only our authoring environment without any preprocessing or postprocessing data transformations. In this development phase, we will use an action/test/fix/repeat process.
- **Add the XSLTs to our FrameMaker application, and test Open and Save using our final application**. As we did in testing our authoring-only application, we will use the action/test/fix/repeat process to test our final application.

The following represents the process we'll use in our application development for both tutorials:



* *Can be created using a text editor, an XML editor, or the text-processing capabilities of FrameMaker.*

** *When testing, protect your original or development files by saving the test instance(s) to a different folder or by using Save As and modifying the file name.*

Process for developing an XSLT-extended FrameMaker application

Tutorial 1: Elements from/to attributes (simple case)

The goal of this tutorial is to walk you through the steps required to create and iteratively test a FrameMaker application that includes XSLT processing to transform the structure of an XML instance on import and transform it back on export. First, we will examine the attribute-value-to-element-content transformation necessary (in our scenario) for a given XML instance and DTD. Next, we will step through what is required to create the appropriate initial test files and then develop our FrameMaker application. A two-tiered approach will be taken in our FrameMaker application development, which will enable us to make sure our basic application is working before we add in the transformations.

Note: *Reading through the steps of the tutorial will provide a basic understanding of the process; however, to actually perform the steps, you must download the xslt_companion.zip file located at www.adobe.com/products/framemaker/downloads/xslt_companion.zip. Unzipping the archive creates an xslt_companion folder with two subfolders: optimize and integrate. Use the optimize subfolder for this tutorial.*

Suppose we have data containing information about a U.K. release of a Beatles album. The data is in the form of an XML instance. The structure of this XML instance is defined as follows: the title of each song is contained in an attribute (*title*) within an element (*song*). For example:

```
<song title="I Saw Her Standing There">
```

Under this structure, authors in the FrameMaker environment would be able to modify a song title only in Structure View, and not directly in the WYSIWYG document window. To rectify this situation, we, as developers, can provide this added flexibility to authors by converting *title* from an attribute to an element that we will call *title-optimized*. The desired result is a Structure View consisting of *song* and *title-optimized* elements as in the following:

Structure View representation of *song* element with *title-optimized* child element

In an XML instance, this structure would be represented as:

```
<song><title-optimized>I Saw Her Standing There</title-optimized>
```

To allow for this convenience in the FrameMaker authoring environment, we use an XSLT to transform the attribute *title* in our XML instance to the element *title-optimized* when the XML instance is opened in FrameMaker. Then, in order to preserve the original structure of the XML document, we transform the *title-optimized* element back to the *title* attribute when the file is saved to XML from FrameMaker. As we proceed, we'll refer to the former structure, the one outside of our FrameMaker environment, as our *storing* structure. The optimized structure we're planning to use within FrameMaker will be referred to as our *authoring* structure.

Note: *For the purposes of this tutorial, we will assume that a DTD and a conforming XML instance already exist for the data.*

The DTD for our storing structure, storing.dtd, is defined as follows:

Copies of the files described here are available in the companion download at www.adobe.com/products/framemaker/downloads/xslt_companion.zip.

```
<!--DTD for optimize. Typically invoked by
      <!DOCTYPE  album-remarks  SYSTEM  "storing.dtd">
-->
<!ELEMENT album-remarks (album-title, artist+, remark?, citation*, song+) >
<!ELEMENT album-title (#PCDATA) >
<!ELEMENT artist (#PCDATA) >
<!ELEMENT remark (#PCDATA) >
<!ELEMENT citation (#PCDATA) >
<!ATTLIST citation cite-title CDATA #REQUIRED >
<!ELEMENT song (non-band-authorship?, initial-release+, remark?, citation*) >
<!ATTLIST song title CDATA #REQUIRED >
<!ELEMENT non-band-authorship (#PCDATA) >
<!ELEMENT initial-release (#PCDATA) >
<!ATTLIST initial-release country CDATA #IMPLIED >
```

Having identified the necessary transformation, we next create an optimized DTD that corresponds to the data structure we would like to use for FrameMaker authoring. This DTD will be used only to test our FrameMaker application prior to the incorporation of the transformation XSLTs. The purpose of this test is to ensure that we can accurately round-trip the XML data.

Before you begin: Reading through the steps of the tutorial will provide a basic understanding of the process; however, to actually perform the steps, you must download the xslt_companion.zip file located at www.adobe.com/products/framemaker/downloads/xslt_companion.zip. Unzipping the archive creates an *xslt_companion* folder with two subfolders. Use the *optimize* subfolder for this tutorial.

1 **Develop an optimized DTD for authoring in FrameMaker.**
This is easily accomplished by modifying a copy of storing.dtd in a text or XML editor and saving it in the same folder as authoring.dtd. We need only modify the DOCTYPE line and the definition for *song*, and then replace the attribute *title* line with a new definition for an element we are calling *title-optimized*.

   a Using a text or XML editor (or the text editing capabilities of FrameMaker), open storing.dtd from the *optimize* folder.

   b Save the file as `authoring.dtd` in that same folder, but don't close the file yet. (If you used the FrameMaker text editing capabilities, make sure you save the file as *Text Only.*)

   c Find and change the appropriate lines, as indicated:

A completed version of authoring.dtd is available in the *finals* subfolder of the *optimize* folder. If this version is used, be sure to copy it or save it to the *optimize* folder.

The optimize folder is available as part of an xslt companion file set, which can be downloaded from www.adobe.com/products/framemaker/downloads/xslt_companion.zip.

Find:
```
<!DOCTYPE  album-remarks  SYSTEM  "storing.dtd">
```
Change it to:
```
<!DOCTYPE  album-remarks  SYSTEM  "authoring.dtd">
```

Find:
```
<!ELEMENT song (non-band-authorship?, initial-release+, remark?, citation*) >
```
Change it to:
```
<!ELEMENT song (title-optimized, non-band-authorship?, initial-release+,
remark?, citation*) >
```

Find:
```
<!ATTLIST song title CDATA #REQUIRED >
```
Change it to:
```
<!ELEMENT title-optimized (#PCDATA) >
```

When completed, authoring.dtd should look like this:

```
<!--DTD for optimize. Typically invoked by
      <!DOCTYPE  album-remarks  SYSTEM  "authoring.dtd">
-->
<!ELEMENT album-remarks (album-title, artist+, remark?, citation*, song+) >
<!ELEMENT album-title (#PCDATA) >
<!ELEMENT artist (#PCDATA) >
<!ELEMENT remark (#PCDATA) >
<!ELEMENT citation (#PCDATA) >
<!ATTLIST citation cite-title CDATA #REQUIRED >
<!ELEMENT song (title-optimized, non-band-authorship?, initial-release+, remark?,
citation*) >
<!ELEMENT title-optimized (#PCDATA) >
<!ELEMENT non-band-authorship (#PCDATA) >
<!ELEMENT initial-release (#PCDATA) >
<!ATTLIST initial-release country CDATA #IMPLIED >
```

2 **Develop an XML test instance that conforms to our authoring structure.**
We will do this by copying our storing structure version, modifying it, and saving it with a new file name.

   a Using a text or XML editor (or the text editing capabilities of FrameMaker), open AlbumRemarks.xml in the optimize folder

b Save the file as AlbumRemarksAuthoring.xml, but don't close it yet. (If you used the FrameMaker text editing capabilities, make sure you save the file as *Text Only.*)

c Find and change all three *song* occurrences.

Find:
```
<song title="I Saw Her Standing There">
```
Change it to:
```
<song><title-optimized>I Saw Her Standing There</title-optimized>
```

Find:
```
<song title="Misery">
```
Change it to:
```
<song><title-optimized>Misery</title-optimized>
```

Find:
```
<song title="Anna (Go to Him)">
```
Change it to:
```
<song><title-optimized>Anna (Go to Him)</title-optimized>
```

d Since this is our authoring test instance, replace the file name of the declared DTD with the file name of our authoring DTD.

Find:
```
<!DOCTYPE album-remarks SYSTEM "storing.dtd">
```
Change it to:
```
<!DOCTYPE album-remarks SYSTEM "authoring.dtd">
```

e Save and close the file.

3 Develop the XSLTs.

Since the focus of this tutorial is using XSLT with FrameMaker, we'll not attempt to create an XSLT from scratch. Instead, we'll look at the previously developed XSLTs designed to transform a storing XML instance to an authoring XML instance and then back again. Our preprocessing XSLT, attributes_to_elements.xsl, is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output doctype-system="authoring.dtd"/>
  <xsl:template match="node()|@*">
    <xsl:copy>
      <xsl:apply-templates select="node()|@*"/>
    </xsl:copy>
  </xsl:template>
  <xsl:template match="song[@title]">
    <song>
      <xsl:apply-templates select="@*[not(local-name()='title')]"/>
      <title-optimized>
        <xsl:apply-templates select="@title"/>
      </title-optimized>
        <xsl:apply-templates select="node()"/>
    </song>
  </xsl:template>
  <xsl:template match="song/@title">
    <xsl:value-of select="."/>
  </xsl:template>
</xsl:transform>
```

The lines of particular interest are those containing a file name, such as authoring.dtd, and those containing *song*, *title*, and *title-optimized*.

Our postprocessing XSLT, elements_to_attributes.xsl, is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output doctype-system="storing.dtd"/>
  <xsl:template match="node()|@*">
    <xsl:copy>
      <xsl:apply-templates select="node()|@*"/>
    </xsl:copy>
  </xsl:template>
  <xsl:template match="song[child::title-optimized]">
    <song title="{title-optimized}">
      <xsl:apply-templates select="node()|@*"/>
    </song>
  </xsl:template>
  <xsl:template match="title-optimized[parent::song]"/>
</xsl:transform>
```

Our next step is to create the first of several components needed for a FrameMaker application: an EDD that matches our authoring structure.

4 **Create an EDD** (part of both our authoring-only and final applications).
From the optimized DTD, we will create our EDD. FrameMaker has a built-in function to accomplish this task.

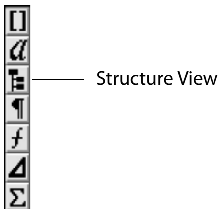
a Choose File > Structure Tools > Open DTD

b Open authoring.dtd in the *optimize* subfolder of the *xslt_companion* folder.

c From the Use Structured Application dialog box, choose <No Application>, and click Continue.

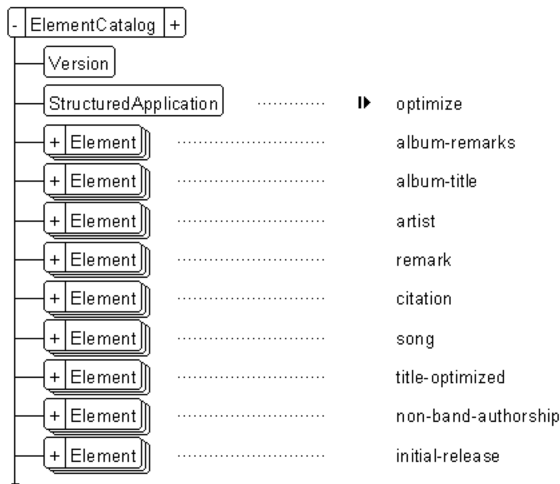
d From the Select Type dialog box, click XML, and in the FrameMaker dialog box, click OK.

e Open the Structure View (click on the third icon from the top at the upper right side of your document window).



f At the top of our new EDD, click within (to the right of) the *StructuredApplication* element, and type `optimize`. This label specifies that our structure is associated with a structured application titled *optimize*. (Later you'll see some formatting information that was added to this EDD.)

With all the elements collapsed, the EDD should now look like this in the Structure View:



Structure View representation of the *optimize* EDD

g Save the Untitled EDD as `authoringEDD.fm` in the same folder as your DTDs, and leave it open but minimized.

5 Create a structured template file (part of both our authoring-only and final applications).

a Choose File > New > Document.

b In the New dialog box, click Portrait.

c Choose File > Import > Element Definitions.

d From the pop-up menu, choose authoringEDD.fm, click Import, and then in the FrameMaker dialog, click OK.

e Save the Untitled template file as `authoringTmpl.fm` in your *optimize* folder. This will be used as the FrameMaker authoring template, which now consists of all the structure imported from the EDD, as well as our *title-optimized* element. After our application is working properly, this is the template into which you can place any master page design features, reference page items, table catalog formats, and so on.

f Close authoringTmpl.fm.

g Close authoringEDD.fm.

6 Create a read/write rules file (part of our authoring-only application).

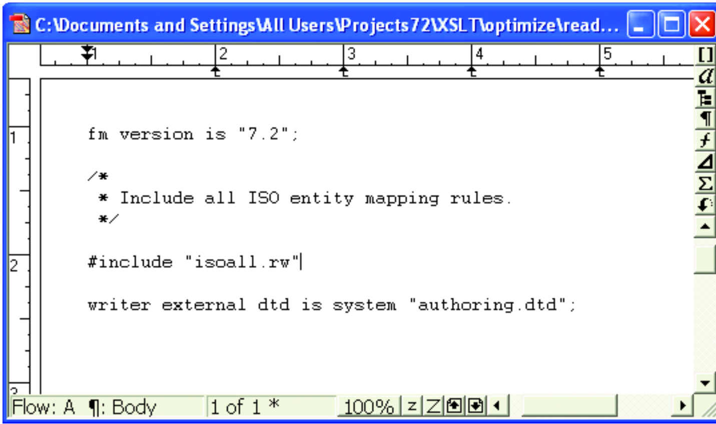
a Choose File > Structure Tools > New Read/Write Rules.
A new, Untitled read/write rules file opens with some default instructions already present.

b Below the default lines, type this line:

```
writer external dtd is system "authoring.dtd";
```

This instruction will be used for our initial open/save round-trip testing. After that, it will be unnecessary, and we will delete it.

After typing the previous line, the read/write rules file should look like this:



Document window view of authoring read/write rules after adding the new rule

c Save the file as `readwriterules.fm` in your *optimize* folder, and then close it.

7 Add the application to the Application Definitions document.
All available SGML, XML, and XHTML applications are listed in the structapps.fm file.

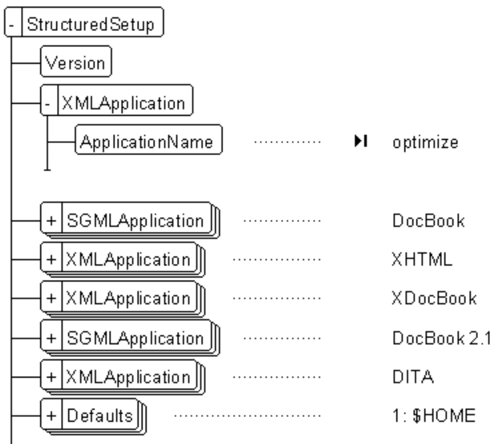
Note: *It is a good—and safe—practice to make a copy of your original structapps.fm file and store it in a safe place before making any modifications.*

We now have all the components required for our authoring-only application. In this step we will create the framework that FrameMaker needs to recognize the application and all its components.

a Choose File > Structure Tools > Edit Application Definitions.

b Put your insertion point immediately below the *Version* element, and from the Element Catalog, select *XMLApplication*, and then click Insert.

c Click within the *ApplicationName* element, and type `optimize`. The Application Definitions file should now look like this in the Structure View:



Structure View representation of the *optimize* application after naming the application

d Just below the *ApplicationName* element and within the *XMLApplication* element, insert a *DOCTYPE* element.

e Within the *DOCTYPE* element, type `album-remarks`.

f Below the *DOCTYPE* element and within the *XMLApplication* element, insert a *DTD* element.

Tip: *You can create a user variable (Special > Variable > Create Variable) to use as the path to your optimize folder for the DTD, Template, ReadWriteRules, and other Application Definitions elements that require a path. To produce a backward slash "\" when a variable is inserted, that character must be defined in the variable using two backslashes "\\". The first one informs FrameMaker that it should take the following character literally.*

g Within the *DTD* element, type the full path for `authoring.dtd`.

h Below the *DTD* element and within the *XMLApplication* element, insert a *Template* element.

i Within the *Template* element, type the full path for `authoringTmpl.fm`.

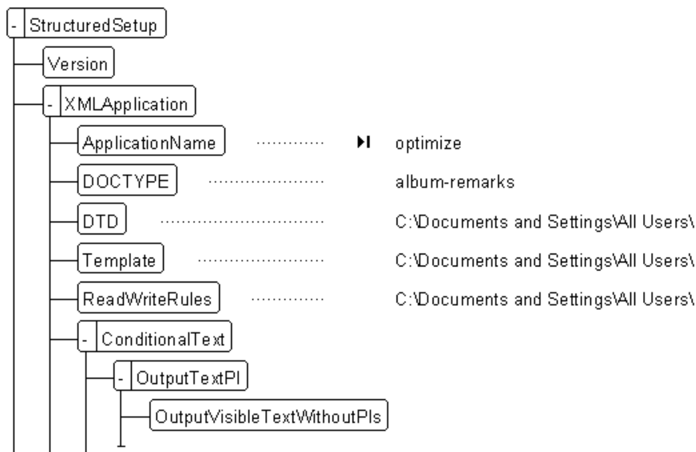
j Below the *Template* element and within the *XMLApplication* element, insert a *ReadWriteRules* element.

k Within the *ReadWriteRules* element, type the full path for `readwriterules.fm`.

l Below the *ReadWriteRules* element and within the *XMLApplication* element, insert a *ConditionalText* element.

m Within the *ConditionalText* element, add the child element *OutputTextPI* and grandchild element *OutputVisibleTextWithoutPIs*.

The Application Definitions file should now look like this:



Structure View representation of the *optimize* application when ready for authoring test

n Save the file, but don't close it yet.

o Choose File > Structure Tools > Read Application Definitions.

Note: *FrameMaker reads this file when it starts. Whenever the file is edited, FrameMaker must be directed to read it again.*

p Close the file.

8 Test the authoring-only application.

To test our authoring-only application, we open and save our XML test instance. Since we have not yet specified an XSLT in our Application Definitions, the Open and Save will be performed without any transformation. For the sake of accurate troubleshooting, we will divide this process into two parts and first perform only the document open function.

Open the XML instance as follows:

a From FrameMaker, choose File > Open.

b Open AlbumRemarksAuthoring.xml.

c If the instance opens without any errors, proceed to the validate step. If any problems occur, note the nature of the problem, close the file without saving, make the necessary corrections, and retry.

Tip: *If FrameMaker is unable to locate a file or application component during import or export, check first for typographical errors in the file name or variable in your application definitions.*

Once the AlbumRemarksAuthoring.xml file opens without errors, make sure that it is valid within FrameMaker in our authoring structure. At this point, the structural integrity of the data is our focus, so don't be concerned with the formatting. We'll say something about that a little later.

d Choose Element > Validate.

e Make sure that Scope: in the Element Validation dialog box is set to Entire Document, and click Start Validating.

If the document is valid, you will receive a "Document is Valid" message. If you do, then proceed to the save test; otherwise, note the nature of the problem, close the file without saving, make the necessary corrections, and retry by reopening AlbumRemarksAuthoring.xml. When the instance opens and validates without any errors, proceed to the save test.

In the second half of our round-trip test, we make sure to avoid any potential corruption to our original XML instance by using Save As (instead of Save) and giving our test output a modified file name.

Save the XML instance as follows:

f Choose File > Save As, type `AlbumRemarksAuthoring_o.xml` as our test output file name, and then click Save.

g If the FrameMaker document saves without any errors, close it, and then proceed to the inspection step. If any problems occur, note the nature of the problem, close the file without saving, make the necessary corrections, and retry.

h Inspect the new output file, and compare it to AlbumRemarksAuthoring.xml. When comparing, be aware that extra comment lines, a difference in indention, or a change in the location of most line breaks are of no concern.

9 Specify the XSLTs in the Application Definitions file.
Having successfully tested our authoring-only application, we will now specify our two XSLTs in the Application Definitions file.

a Choose File > Structure Tools > Edit Application Definitions.

b Below the *ReadWriteRules* element, insert a *Stylesheets* element.

c Within the *Stylesheets* element, insert an *XSLTPreferences* element.

d Within the *XSLTPreferences* element, insert a *PreProcessing* element.

e Within the *PreProcessing* element, insert a *Stylesheet* element.

f Within the *Stylesheet* element, type the full path to `attributes_to_elements.xsl`.

g As a sibling to the *PreProcessing* element, insert a *PostProcessing* element.

h Within the *PostProcessing* element, insert a *Stylesheet* element.

i Within the *Stylesheet* element, type the full path to `elements_to_attributes.xsl`.

j Choose File > Save, but don't close the file yet.

k Choose File > Structure Tools > Read Application Definitions.

l Close the file.

The DTD declaration supplied by the one entry we made in our read/write rules file will be stripped out and replaced by our postprocessing XSLT. Because the entry was only required for our initial round-trip testing, we can now delete it or comment it out.

m Choose File > Open.

n Open readwriterules.fm in the *optimize* folder.

o Delete the following line:

```
writer external dtd is system "authoring.dtd";
```

p Save and close the read/write rules file.

With these changes, our final application is now ready to be tested.

10 Test the final application.
Our XML instance that uses the storing structure should open in FrameMaker with the *title* attributes transformed to *title-optimize* elements. As previously noted, this allows authors to modify a song title in the WYSIWYG document window as well as in the Structure View. On Save or Save As, the *title-optimize* elements will be transformed back to *title* attributes to preserve the storing XML structure.

Open the XML instance as follows:

a Choose File > Open.

b Open AlbumRemarks.xml.

c If the instance opens without error, proceed to the validate test. If any problems occur, note the nature of the problem, close the file without saving, make the necessary corrections, and retry.

Tip: *If FrameMaker is unable to locate a file or application component during import or export, check first for typographical errors in the file name or variable in your application definitions.*

When the instance opens without errors, validate it in FrameMaker.

d Choose Element > Validate.

e Make sure that Scope: in the Element Validation dialog box is set to Entire Document, and click Start Validating.

If the document is valid, you will receive a "Document is Valid" message. If you do, then proceed to the save test; otherwise, note the nature of the problem, close the file without saving, make the necessary corrections, and retry by reopening AlbumRemarks.xml.

When the instance opens without errors and is valid in FrameMaker, we are ready to save our document back out to XML. Because we are still in a test mode, we will use Save As (instead of Save), and give our test output a modified file name.

Save the XML instance as follows:

f Choose File > Save As, type `AlbumRemarks_o.xml` as our test output file name, and then click Save.

g If the FrameMaker document saves without any errors, close it, and then proceed to the inspection step. If any problems occur, note the nature of the problem, close the file without saving, make the necessary corrections, and retry.

h Inspect the new output file, and compare it to the original. As before, a difference in line breaks or indention should be considered insignificant.

Using the EDD to format your document.

The focus of this paper is using XSLT with FrameMaker; however, it is instructive to see how an EDD that includes some rudimentary formatting can change the look of your data. We will leave the full development of the format context rules to you as an exercise, but present the final result through the following steps:

The EDD used in this section is available in the *finals* subfolder of the *optimize* folder. It is available as part of the xslt companion file set, which can be downloaded from www.adobe.com/products/framemaker/downloads/xslt_companion.zip.

1 Choose File > Open.

2 Open authoringEDD_Arial.fm in the *finals* subfolder of the *optimize* folder.

3 Choose File > Open.

4 Open the AlbumRemarks.xml instance.

5 With AlbumRemarks.xml the active document, choose File > Import > Element Definitions.

6 From the pop-up menu, choose the authoringEDD_Arial.fm file we opened a moment ago.

7 Click Import, and in the FrameMaker dialog box, click OK.

   Notice the change to the formats in AlbumRemarks.xml. Glance through the format context rules in this EDD to see how this formatting was accomplished. A variety of formatting methods are included to provide you with a glimpse at some of the techniques used to create highly formatted documents. Some of the bolded and colorized labels that appear in your document are enabled by EDD-driven Prefix rules, and Suffix rules. Even more can be accomplished by combining format context rules with additions/changes to the structured template's character, table, and cross-reference formats, master pages, and reference pages.

8 Without saving, close the files.

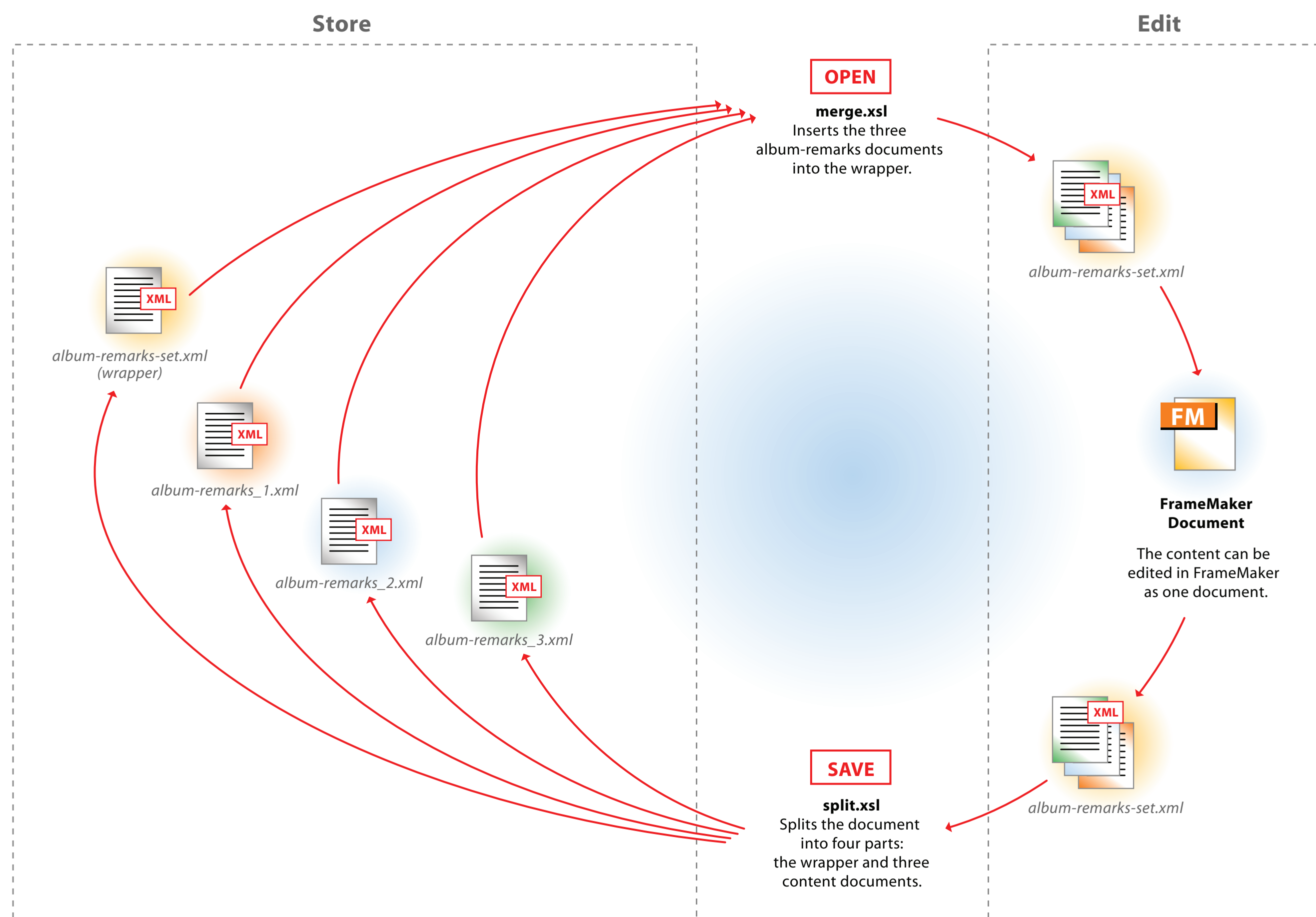
In the real world, you would want the formatting specified in your EDD to automatically appear each time you opened AlbumRemarks.xml. To accomplish this, import this EDD's element definitions into the structured template file, and then save it.

Tutorial 2: Assembling on import and splitting on export (complex case)

The goal of this tutorial is to walk you through the steps required to create and iteratively test a FrameMaker application that includes XSLT processing to assemble an XML instance on import and split it on export. Suppose we have XML data that must be maintained outside of FrameMaker in discrete chunks, perhaps by a content management system or a learning management system. We want to assemble these chunks into a single document, edit the document as a whole in FrameMaker, paginate as necessary for publication, and then write the information back out into its discrete chunks. This merging and splitting can be accomplished from within our FrameMaker application by using the new FrameMaker XSLT capabilities, which enable more granularity in merging and splitting than has ever been available in FrameMaker before.

Note: *Reading through the steps of the tutorial will provide a basic understanding of the process; however, to actually perform the steps, you must download the xslt_companion.zip file located at www.adobe.com/products/framemaker/downloads/xslt_companion.zip. Unzipping the archive creates an xslt_companion folder with two subfolders: optimize and integrate. Use the integrate subfolder for this tutorial.*

In our example, the discrete chunks that we have chosen to maintain are album-remarks XML instances. We want to combine these chunks into an *album-remarks-set*, modify them in FrameMaker, and appropriately split the document back out into *album-remarks*.



Data flow when opening or saving the album-remarks-set.xml file using XSLT preprocessing and postprocessing in a completed application

The outer structure of each *album-remarks* XML instance is shown in the following code snippet:

```
<album-remarks>
  <album-title>Please Please Me</album-title>
  .
  .
  .
</album-remarks>
```

To accomplish our task, three of these XML instances will be merged by an XSLT into a wrapping instance, album-remarks-set.xml, prior to import into FrameMaker. The wrapping instance looks like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE album-remarks-set SYSTEM "integrate.dtd">
<album-remarks-set>
  <set-title>The Beatles—first four UK albums</set-title>
  <set-title-sub>Remarks, impressions, and musings (based on U.K. albums and song
order from half-speed mastered, vinyl stereo versions)</set-title-sub>
  <include-xml path="album-remarks_1.xml"/>
  <include-xml path="album-remarks_2.xml"/>
  <include-xml path="album-remarks_3.xml"/>
</album-remarks-set>
```

Each *include-xml* line specifies a file to be merged into the single FrameMaker document; however, this is not the *xsl:include* element. Instead, we have created an element in our own DTD called *include-xml*, and we'll define this element to contain the path of the file we want to include.

Note: *For the purposes of this tutorial, we will assume that a DTD encompassing the merged instance, as well as the conforming, discrete XML instances, already exist for the data.*

The following DTD, integrate.dtd, defines the structure necessary for both our split XML instances and our merged instance:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--DTD for album-remarks-set. Typically invoked by
      <!DOCTYPE  album-remarks-set  SYSTEM "integrate.dtd">
-->
<!ELEMENT album-remarks-set
                     (set-title, set-title-sub?, (album-remarks* |
                     include-xml*)) >
<!ELEMENT include-xml  EMPTY >
<!ATTLIST include-xml path   CDATA     #REQUIRED >
<!ELEMENT set-title  (#PCDATA) >
<!ELEMENT set-title-sub (#PCDATA) >
<!ELEMENT album-remarks (album-title, remark?, citation*, song+) >
<!ELEMENT album-title (#PCDATA) >
<!ELEMENT citation   (#PCDATA) >
<!ATTLIST citation   cite-title  CDATA     #REQUIRED >
<!ELEMENT remark     (#PCDATA) >
<!ELEMENT song       (title, non-band-authorship?, initial-release+,
                              personnel, remark?, citation*) >
<!ELEMENT title      (#PCDATA) >
<!ELEMENT non-band-authorship
                     (#PCDATA) >
<!ELEMENT initial-release
                     (#PCDATA) >
<!ATTLIST initial-release country   CDATA     #IMPLIED >
<!ELEMENT personnel  (musician+) >
<!ELEMENT musician   (name, contribution) >
<!ELEMENT name       (#PCDATA) >
<!ELEMENT contribution (#PCDATA) >
```

Because the structure of our data is the same inside and outside of FrameMaker, we will not need to develop a separate authoring DTD.

Before you begin: Reading through the steps of the tutorial will provide a basic understanding of the process; however, to actually perform the steps, you must download the xslt_companion.zip file located at www.adobe.com/products/framemaker/downloads/xslt_companion.zip. Unzipping the archive creates an *xslt_companion* folder with two subfolders. Use the *integrate* subfolder for this tutorial.

1 **Develop an authoring-only XML instance**.
We will develop an XML test instance that conforms to our merged structure by first copying our wrapper instance, then copying into it each of the include files it specifies, and finally saving it with a new file name.

A completed version of album-remarks-set-merged.xml is available in the *finals* subfolder of the *integrate* folder. If this version is used, be sure to copy or save it to the *integrate* folder.

The *integrate* folder is available as part of an xslt companion file set which can be downloaded from www.adobe.com/products/framemaker/downloads/ xslt_companion.zip.

a Using a text or XML editor (or the text editing capabilities of FrameMaker), open album-remarks-set.xml.

b Save the file as `album-remarks-set-merged.xml`, but don't close it yet. (If you used the FrameMaker text editing capabilities, make sure you save the file as *Text Only.*)

c Open our first chunk, album-remarks_1.xml.

d Exclude the top two lines, and copy everything from the line that begins `<album-remarks>` down to and including `</album-remarks>`.

e In our album-remarks-set-merged.xml file, select the line:
`<include-xml path="album-remarks_1.xml"/>`

f Paste over this line with the content you copied from album-remarks_1.xml.

g Close album-remarks_1.xml without saving.

h Using this same approach, copy all but the first two lines of album-remarks_2.xml and album-remarks_3.xml over the appropriate include lines of album-remarks-set_merged.xml.

i When all three *album-remarks* files have been pasted into album-remarks-set-merged.xml, save and close it.

2 **Develop the XSLTs.**
Since the focus of this tutorial is using XSLT with FrameMaker, we'll not attempt to create an XSLT from scratch. Instead, we'll look at the previously developed XSLTs designed to merge our discrete instances on import and split them out again on export.

Our preprocessing XSLT, merge.xsl, is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output doctype-system="integrate.dtd"/>
  <!-- copy everything ... -->
  <xsl:template match="node()|@*">
    <xsl:copy>
      <xsl:apply-templates select="node()|@*"/>
    </xsl:copy>
  </xsl:template>
  <!-- ... except: -->
  <xsl:template match="include-xml">
    <!-- replace the element with the referenced XML document -->
    <xsl:copy-of select="document(@path)"/>
  </xsl:template>
</xsl:transform>
```

The first line of particular interest is the following:

```
<xsl:template match="include-xml">
```

This instruction looks for an *include-xml* element. The next line in view is the following:

```
<xsl:copy-of select="document(@path)"/>
```

This line of code takes the value of the *path* attribute in the matched *include-xml* element, and replaces the entire element with the contents of the file referenced in that path. You might recognize this behavior as being very much like resolving an external entity.

Note: *In this example, we don't preserve the file name of the included component. Instead, in our postprocessing XSLT, we regenerate the file name when we save out the files. An alternative, though more complex, approach might save the file name in the merged (authoring) version of the document so that it could be user-specified instead of auto-generated.*

Our postprocessing XSLT, split.xsl, is the following (some lines broken to fit):

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xalan="http://xml.apache.org/xalan"
extension-element-prefixes="xalan">
<!--
MIGHT OVERWITE FILES
-->
  <xsl:output doctype-system="integrate.dtd"/>
  <!-- copy everything ... -->
  <xsl:template match="node()|@*">
    <xsl:copy>
      <xsl:apply-templates select="node()|@*"/>
    </xsl:copy>
  </xsl:template>
  <!-- ... except: -->
  <xsl:template match="album-remarks">
    <xsl:variable name="position"
    select="count(preceding-sibling::album-remarks)+1"/>
    <xsl:variable name="file"
    select="concat('album-remarks_',$position,'.xml')"/>
    <include-xml path="{$file}"/>
    <xalan:write select="$file">
      <xsl:copy-of select="."/>
    </xalan:write>
  </xsl:template>
</xsl:transform>
```

This XSLT includes a template that comes into play each time our *album-remarks* element is found. A variable named *position* is set by counting the number of previous *album-remarks* sibling elements and incrementing that number by 1. A file name variable (*file*) is then built by concatenating the string `album-remarks_`, the *position* variable (in our case, this would be 1, 2, or 3), and the string `.xml`.

The file name variable (*file*) is used in two ways: first, it is set within the `<include-xml path="{$file}"/>` line of code for output to our wrapping *album-remarks-set* file, and then it is used to name a file into which, through the xalan:write extension, the *album-remarks* element (and all its content) is written out.

Our next step is to create the first of several components needed for a FrameMaker application: an EDD that matches the merged structure. From there, we will create our structured template and read/write rules file and add them to our Application Definitions. As in our last example, we will first set up and test our application without the XSL transformations. The purpose is to ensure that we can accurately round-trip the XML data before adding the XSLTs.

3 **Create an EDD** (part of both our authoring-only and final applications).
We will use the built-in FrameMaker function to create an EDD from an existing DTD.

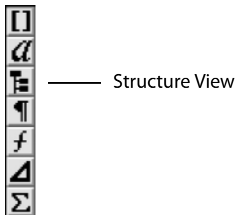
  a Choose File > Structure Tools > Open DTD.

  b Open integrate.dtd in the *integrate* folder.

  c From the Use Structured Application dialog box, choose <No Application>, and click Continue.

d From the Select Type dialog box, click XML, and from the FrameMaker dialog box, click OK.

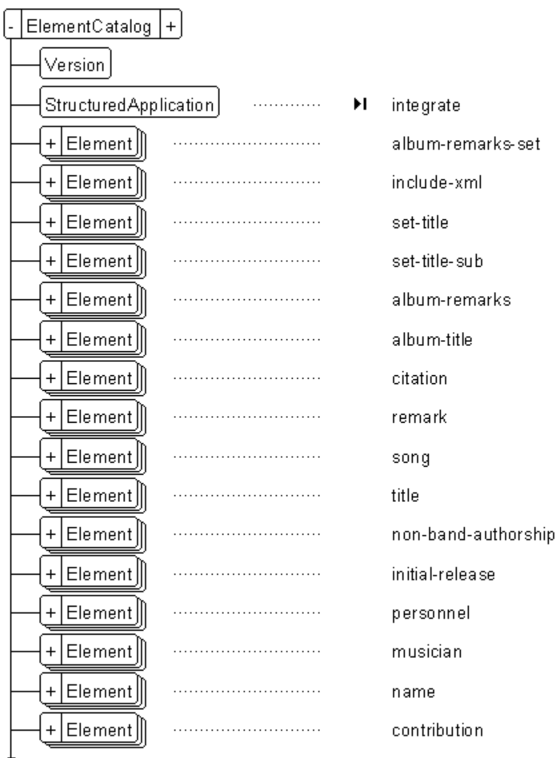
e Open the Structure View (click on the third icon from the top at the upper right side of your document window).



The Structure View icon

f At the top of the new EDD, click within (to the right of) the *StructuredApplication* element, and type `integrate`. This label specifies that our structure is associated with a structured application titled *integrate*. (Later you'll see some formatting information that was added to this EDD.)

With the elements collapsed, the EDD should now look like this in the Structure View:



Structure View representation of the *integrate* EDD

g Save the Untitled EDD as `integrateEDD.fm` in the same folder as your DTDs, and leave it open but minimized.

4 Create a structured template (part of both our authoring-only and final applications).

a Choose File > New > Document.

b In the New dialog box, click Portrait.

c Choose File > Import > Element Definitions.

d From the pop-up menu, choose integrateEDD.fm, click Import, and then in the FrameMaker dialog, click OK.

e Save the Untitled template file as `integrateTmpl.fm` in your *integrate* folder. This will be used as the FrameMaker template, which now consists of all the structure imported from the EDD. After our application is working properly, this is the template into which you can place any master page design features, reference page items, table catalog formats, and so on.

f Close integrateTmpl.fm.

g Close integrateEDD.fm.

5 **Create read/write rules** (part of both our authoring-only and final applications).

a Choose File > Structure Tools > New Read/Write Rules.
A new, Untitled read/write rules file opens with some default instructions already present.

b Below the default lines, type this line:
`writer external dtd is system "integrate.dtd";`
This instruction will be used for our initial open/save round-trip testing. After that, it will be unnecessary, and we will delete it.

After typing the previous line, the read/write rules file should look like this:

```
fm version is "7.2";

/*
 * Include all ISO entity mapping rules.
 */

#include "isoall.rw"

writer external dtd is system "integrate.dtd";
```

Document window view of the authoring read/write rules after adding the new rule

c Save the file as `readwriterules.fm` in your *integrate* folder.

d Close the readwriterules.fm file.

6 **Add your application to the Application Definitions document.**
All available SGML, XML, and XHTML applications are listed in the structapps.fm file.

Note: *It is a good—and safe—practice to make a copy of your original structapps.fm file and store it in a safe place before making any modifications.*

a Choose File > Structure Tools > Edit Application Definitions.

b Put your insertion point immediately below the *Version* element, and from the Element Catalog, select *XMLApplication*, and then click Insert.

c Click within the *ApplicationName* element, and type `integrate`.

The top portion of the Application Definitions file should now look like this in the Structure View:

Top portion of the *integrate* application after typing the application name

d Just below the *ApplicationName* element and within the *XMLApplication* element, insert a *DOCTYPE* element.

e Within the *DOCTYPE* element, type `album-remarks-set`.

f Below the *DOCTYPE* element and within the *XMLApplication* element, insert a *DTD* element.

Tip: *You can create a user variable (Special > Variable > Create Variable) to use as the path to your integrate folder for the DTD, Template, ReadWriteRules, and other Application Definitions elements that require a path. To produce a backward slash "\" when a variable is inserted, that character must be defined in the variable using two backslashes "\\". The first one informs FrameMaker that it should take the following character literally.*

g Within the *DTD* element, type the full path for `integrate.dtd`.

h Below the *DTD* element and within the *XMLApplication* element, insert a *Template* element.

i Within the *Template* element, type the full path for `integrateTmpl.fm`.

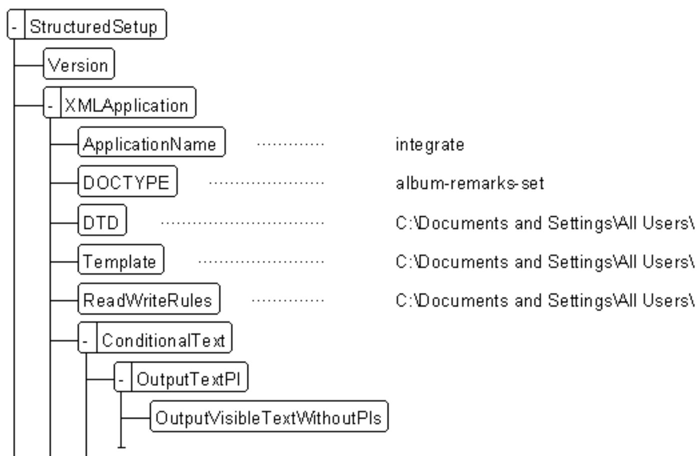
j Below the *Template* element and within the *XMLApplication* element, insert a *ReadWriteRules* element.

k Within the *ReadWriteRules* element, type the full path for `readwriterules.fm`.

l Below the *ReadWriteRules* element and within the *XMLApplication* element, insert a *ConditionalText* element.

m Within the *ConditionalText* element, add the child element *OutputTextPI* and grandchild element *OutputVisibleTextWithoutPIs*.

The Application Definitions file should now look like this:



Structure View representation of the *integrate* application when ready for authoring test

n Save the file, but don't close it yet.

o Choose File > Structure Tools > Read Application Definitions.

Note: *FrameMaker reads this file when it launches. Whenever the file is edited, FrameMaker must be directed to read it again.*

p Close the file.

7 Test the authoring application.

To test our application, we first open and save our XML test instance. Since we have not yet specified an XSLT in our Application Definitions, the Open and Save will be performed without any transformation. For the sake of accurate troubleshooting, we will divide this process into two parts and first perform only the document open function.

Open the XML instance as follows:

a From FrameMaker, choose File > Open.

b Open album-remarks-set-merged.xml.

c If the instance opens without any errors, proceed to the validate step. If any problems occur, note the nature of the problem, close the file without saving, make the necessary corrections, and retry.

Tip: *If FrameMaker is unable to locate a file or application component during import or export, check first for typographical errors in the file name or variable in your application definitions.*

Once the album-remarks-set-merged.xml file opens without errors, make sure that it is valid within FrameMaker. At this point, the structural integrity of the data is our focus, so don't be concerned with the formatting. We'll say something about that a little later.

d Choose Element > Validate.

e Make sure that Scope: in the Element Validation dialog box is set to Entire Document, and click Start Validating.

If the document is valid, you will receive a "Document is Valid" message. If you do, then proceed to the save test; otherwise, note the nature of the problem, close the file without saving, make the necessary corrections, and retry by reopening album-remarks-set-merged.xml. When the instance opens and validates without any errors, proceed to the save test.

In the second half of our round-trip test, we make sure to avoid any potential corruption to our original XML instance by using Save As (instead of Save) and giving our test output a modified file name.

Save the XML instance as follows:

f Choose File > Save As, type `album-remarks-set-merged_o.xml` as our test output file name, and then click Save.

g If the FrameMaker document saves without any errors, close it, and then proceed to the inspection step. If any problems occur, note the nature of the problem, close the file without saving, make the necessary corrections, and retry.

h Inspect the new output file, and compare it to album-remarks-set-merged.xml. As before, a difference in the location of line breaks or indention is not a concern.

8 Specify the XSLTs in the Application Definitions file.
Having successfully tested our authoring application, we will now specify our two XSLTs in the Application Definitions file.

a Choose File > Structure Tools > Edit Application Definitions.

b Below the *ReadWriteRules* element, insert a *Stylesheets* element.

c Within the *Stylesheets* element, insert an *XSLTPreferences* element.

d Within the *XSLTPreferences* element, insert a *PreProcessing* element.

e Within the *PreProcessing* element, insert a *Stylesheet* element.

f Within the *Stylesheet* element, type the full path to `merge.xsl`.

g As a sibling to the *PreProcessing* element, insert a *PostProcessing* element.

h Within the *PostProcessing* element, insert a *Stylesheet* element.

i Within the *Stylesheet* element, type the full path to `split.xsl`.

j Choose File > Save, but don't close the file yet.

k Choose File > Structure Tools > Read Application Definitions.

l Close the file.

The DTD declaration supplied by the one entry we made in our read/write rules file will be stripped out and replaced by our postprocessing XSLT. Because it was only required for our initial round-trip testing, we can now delete it or comment it out.

m Choose File > Open.

n Open readwriterules.fm in your *integrate* folder.

o Delete the following line:

```
writer external dtd is system "integrate.dtd";
```

p Save and close readwriterules.fm.

With these changes, our final application is now ready to be tested.

9 Test the final application.

With our XSLTs specified in the Application Definitions file, we can now test our completed application. Our wrapping XML instance should open in FrameMaker with the separate *album-remarks* files already merged. On Save or Save As, the wrapping structure will be saved as an album-remarks-set.xml file, and the *album-remarks* elements will be split off and written to separate files.

Since we are still in test mode, we should take the precaution of copying our original XML instances to another folder. This way, if our first attempt is not successful, we will not have overwritten our only set of source files. Copy the following files to another folder:

album-remarks-set.xml
album-remarks_1.xml
album-remarks_2.xml
album-remarks_3.xml

Open the XML instance as follows:

a Choose File > Open.

b Open the original album-remarks-set.xml.

c If the instance opens without error, proceed to the validate step. If any problems occur, note the nature of the problem, close the file without saving, make the necessary corrections, and retry.

Tip: *If FrameMaker is unable to locate a file or application component during import or export, check first for typographical errors in the file name or variable in your application definitions*.

Once the album-remarks-set.xml file opens without errors, make sure that it is valid within FrameMaker.

d Choose Element > Validate.

e Make sure that Scope: in the Element Validation dialog box is set to Entire Document, and click Start Validating.

If the document is valid, you will receive a "Document is Valid" message. If you do, then proceed to the save test; otherwise, note the nature of the problem, close the file without saving, make the necessary corrections, and retry by reopening album-remarks-set.xml.

When the instance opens without errors and is valid in FrameMaker, we are ready to perform the save portion of our application test. Because we took some precautions to archive our data files, we can use Save instead of Save As for this test.

Save the XML instance as follows:

f Choose File > Save.

g If the FrameMaker document saves without any errors, close it, and then proceed to the inspection step. If any problems occur, note the nature of the problem, close the file without saving, make the necessary corrections, and retry.

h Inspect the new output files, and compare them to the originals. As before, a difference in line breaks should be considered insignificant.

Using the EDD to format your document.

The focus of this paper is using XSLT with FrameMaker; however, it is instructive to see how an EDD that includes some rudimentary formatting can change the look of your data. We will leave development of the format context rules to you as an exercise, but present the final result through the following steps:

The EDD used in this section is available in the *finals* subfolder of the *integrate* folder. It is available as part of the xslt companion file set, which can be downloaded from www.adobe.com/products/framemaker/downloads/xslt_companion.zip.

1. Choose File > Open.
2. Open integrateEDD_Arial.fm in the *finals* subfolder of the *integrate* folder.
3. Choose File > Open.
4. Open the album-remarks-set.xml instance.
5. With album-remarks-set.xml the active document, choose File > Import > Element Definitions.
6. From the pop-up menu, choose the integrateEDD_Arial.fm file we opened a moment ago.
7. Click Import, and in the FrameMaker dialog box, click OK.

   Notice the change to the formats in our album-remarks-set.xml instance. Check the format context rules in the authoring EDD to see how this was accomplished. A variety of formatting methods are included to provide you with a glimpse at some of the techniques used to create highly formatted documents. Some of the bolded and colorized labels that appear in your document are enabled by EDD-driven autonumbering, Prefix rules, Suffix rules, and attribute evaluation. Even more can be accomplished by combining format context rules with additions/changes to the structured template's character, table, and cross-reference formats, master pages, and reference pages.
8. Without saving, close the files.

In the real world, you would want the formatting specified in your EDD to automatically appear each time you opened album-remarks-set.xml. To accomplish this, import this EDD's element definitions into the structured template file, and then save it.

Summary

In this paper, we've seen the benefits of structuring content in XML and the utility of XSLT as a means of transforming XML from one structure to another. We've provided a brief introduction to XSLT on its own, and we've surveyed how XSLT fits into the structured FrameMaker environment, considering common use cases, as well as the nuts of bolts of implementation. The two tutorials offer step-by-step examples of using XSLT with FrameMaker, which should provide a foundation for further exploration.

Organizations managing their content in XML will find a wide range of applications for XSLT in their XML authoring and publishing workflows. Effective use of the XSLT features of FrameMaker depends on understanding structured FrameMaker applications, as well as knowledge of XSLT itself. A number of resources offer further information on both subjects.

Recommended next steps

If you are already well versed in the development of FrameMaker applications and understand the functioning of XSLT, this paper has described what you needed to know: how XSLT is used in the FrameMaker environment. If you would, however, like more information on some of the topics touched upon in this paper, glance through some of the resources provided in this section.

Find out more about creating FrameMaker applications

The *Structure Application Developer's Guide* provides fundamental help on working with XML and FrameMaker, including core techniques for working with XSLT. It is available in the OnlineManuals folder of your FrameMaker application folder.

Find out more about XSLT

There are a variety of resources on the web for learning XSLT, starting with the W3C specifications and XSLT tutorials available at www.w3.org/Style/XSL/.

Useful resources

- XSLT Specification—www.w3.org/TR/xslt
- XSL Frequently Asked Questions—www.dpawson.co.uk/xsl/index.html
- XSL Listserv—www.mulberrytech.com/xsl/xsl-list/index.html
- XSLT Tutorials—www.jenitennison.com/xslt/index.html
- Xerces, an open source XML parser: http://xml.apache.org/xerces-c/
- Xalan, an open source XSLT processor—http://xml.apache.org/xalan-c/
- Java version of the Xalan XSLT processor—http://xml.apache.org/xalan-j/

Adobe Systems Incorporated • 345 Park Avenue, San Jose, CA 95110-2704 USA • *www.adobe.com*

Adobe, the Adobe logo, and FrameMaker are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. SVG is a trademark of the World Wide Web Consortium; marks of the W3C are registered and held by its host institutions MIT, INRIA and Keio. All other trademarks are the property of their respective owners.

© 2006 Adobe Systems Incorporated. All rights reserved.