Real-world Case Studies: Getting XML From Adobe InDesign Layouts for Repurposing

TABLE OF CONTENTS

1 Executive Summary

2 Case Study #1: How ***T+L Golf*** Magazine Creates HTML with InDesign

3 Mapping Styles to Tags

4 Organizing the Elements

7 The XSLT Template

8 Creating XHTML with InDesign and GoLive

10 Case Study #2: How ***Texas Lawyer*** Generates Complete XML Files with Structure and Attributes

10 Better Results with Styles to Tags

11 Grouping Elements of Inline Articles

12 Grouping a Range of Inline Elements with the Structure Pane

12 Using Attributes to Record Article Metadata

13 Validating XML Before Exporting

16 Exporting Many Articles

16 The Final Transformation

16 Conclusion

17 Notes

17 InCopy Notes and InDesign CS

18 Appendices A1-D2: Scripting Samples

Executive Summary

This white paper presents two real-world case studies of publishers who are using the powerful XML features in Adobe® InDesign® CS to repurpose content. Because XML is an open standard, it can be understood by a wide range of applications, making it easy to move content from InDesign layouts directly to HTML for websites or to large database-driven systems that control websites.

The first case study explores some of the techniques that American Express Publishing employs to repurpose content from InDesign CS layouts of its bimonthly *T+L Golf* to HTML. The second case study focuses on how a weekly newspaper, *Texas Lawyer*, creates XML files from InDesign CS layouts for its powerful content management system (CMS) to produce website content and generate XML for syndication. Both publishers use XML to maintain critical formatting information such as bold, italic, headline tagging, body text formats, and contextual information such as the publication's name, date, and author. (XML is flexible enough to store other types of information as well, including copyright specifics, licensing terms, keywords, and categorization information.)

InDesign is renowned for the quality of its composition and layout engine and for its extensive XML and scripting capabilities, which enable it to support the complex needs of today's publishers. While this paper focuses on what two customers have accomplished with InDesign CS, much of this experience is relevant for InDesign CS2 users as well. For a more detailed overview of all of the new and existing XML capabilities in InDesign CS2, see *Adobe InDesign and XML: A Technical Reference* on the Adobe website at www.adobe.com/products/indesign/crossmedia.html.

Companies that produce magazines, newspapers, and books are realizing increasingly that text content appearing in their publications has value beyond that publication. These publishers take articles, photographs, and other content that originally appeared in their printed publications and publish it on websites. Sometimes they license the content to other companies that may then reprint the articles or publish it on other websites. These websites may charge a small fee to read the articles, or they may make the articles available on a subscription basis. Some publishers offer content for free.

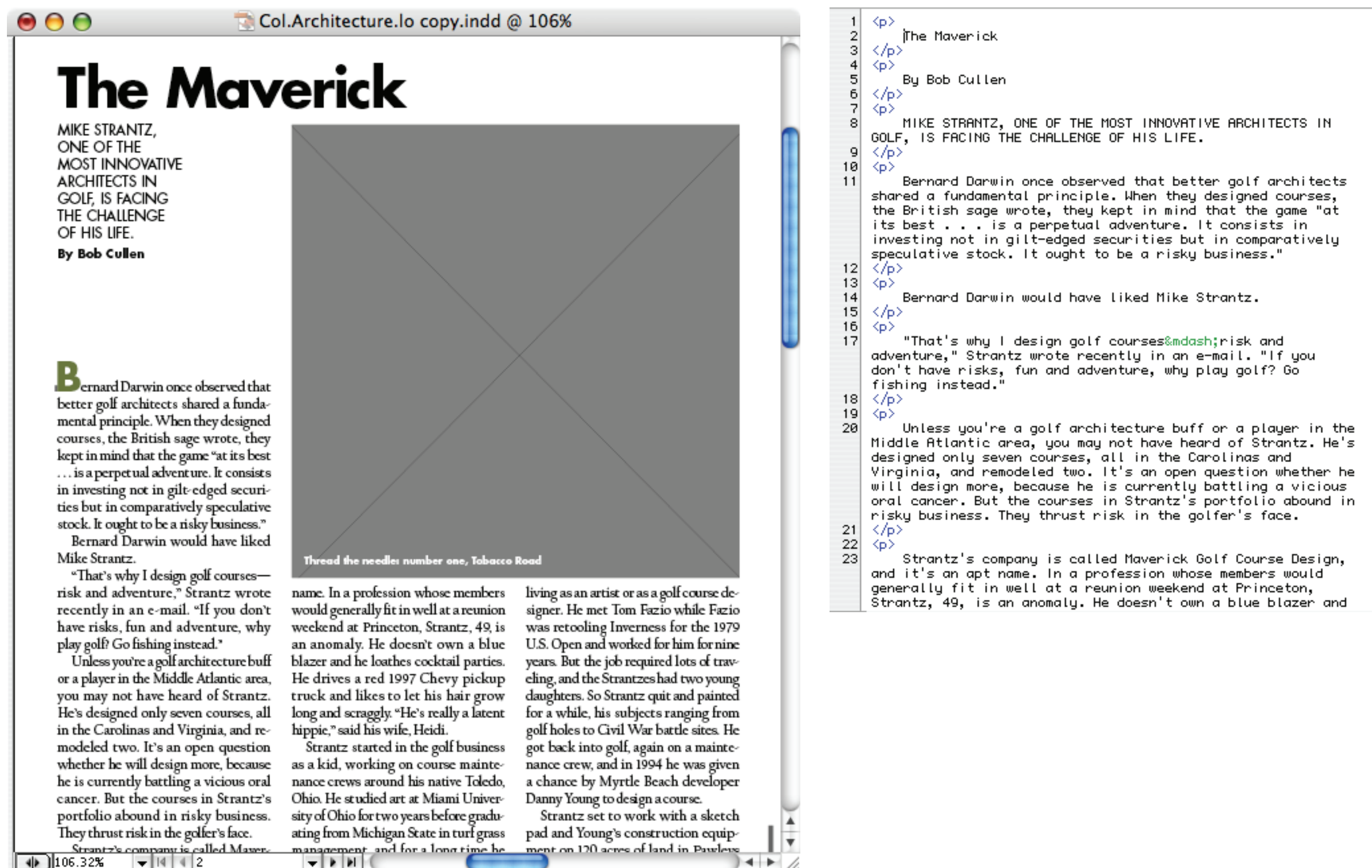
In all of these cases, the publisher strives to find the most efficient way to extract text from a page and convert it to a format that can be easily distributed and reused. Determining which format to use to store the content has been challenging, but Extensible Markup Language (XML) has become a popular format because its broad support makes it easy to publish the same content in many different formats. In the world of publishing, XML is used to separate form from content. Form is applied to content by transforming the XML into another format. A number of technologies exist for transforming XML, but a new standard known as eXtensible Stylesheet Language for Transformations (XSLT) is gaining acceptance. XSLT is used to format XML content in different ways as required by publishers. Often, XSLT is used to transform XML files into HTML, but it can just as easily be used to generate other types of text formats This paper assumes a basic familiarity with XML and HTML concepts. For primers on XML, visit www.w3schools.com/xml. For HTML information, visit webmonkey.wired.com/webmonkey/authoring.

Adobe

Case Study #1: How *T+L Golf* Magazine Creates HTML with InDesign

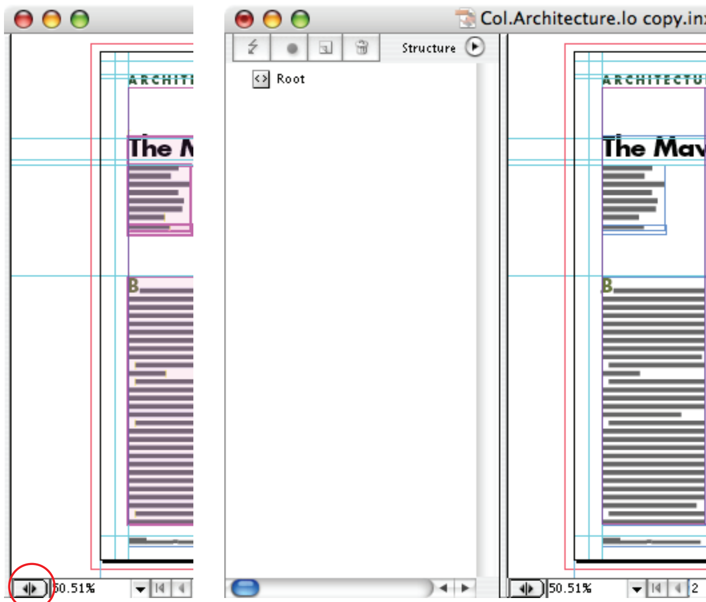
The website for *T+L Golf* magazine provides readers with an online edition of the current issue. A sophisticated content management system (CMS) uses a database to hold the articles for each issue. The database automatically creates HTML that displays selected articles in the context of the entire web page, which includes navigational elements and banners. This CMS makes it easy for the website editors at *T+L Golf* to make articles available for online viewing. To add an article to the website, a web editor first uses the XML features in InDesign to create an XML file that includes common HTML tags. After InDesign generates the XML, the files are further processed with XSLT to insert additional tags, and to translate high-end typographic characters to more web-friendly, lower ASCII characters. Finally, the XML files are submitted to the CMS.

The CMS requires a very basic HTML-like format in which paragraphs are delineated by <p> tags and bold and italic text is marked by <b> and <i>, respectively.



This illustration shows how an article appears in InDesign (on the left) and its XML counterpart (on the right).

The web editor processes each InDesign file for the magazine by grouping article components, then converting the text to XML. After opening an InDesign layout file, the editor displays the XML Structure pane by clicking the Splitter button in the lower left corner of the window or by Choosing View > Structure > Show Structure.

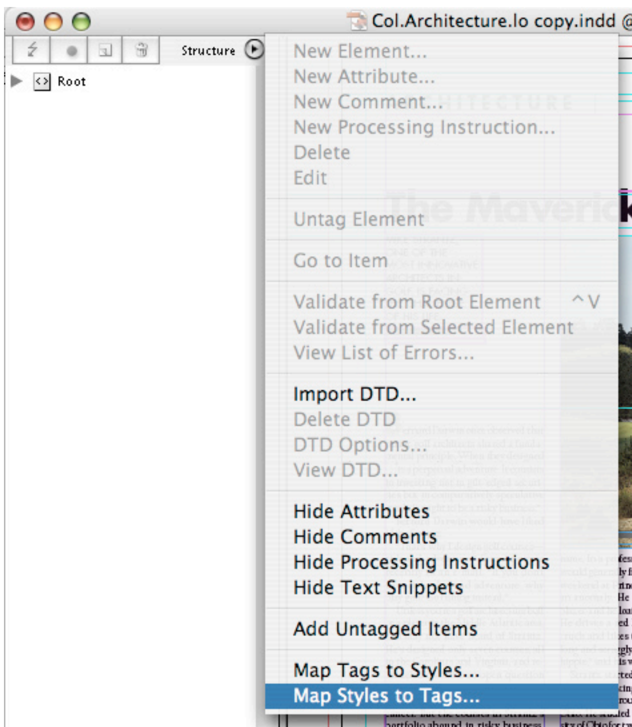


You can display the XML Structure pane by clicking the Splitter button in the lower left corner of the window or by Choosing View>Structure>Show Structure.

The Structure pane shows a tree of XML elements. If no elements have been created, the tree is empty except for the default Root element. The designers and editors at *T+L Golf* will add XML elements for the paragraph and character styles they use to format their text. First, they'll use paragraph styles in InDesign to create the <p> tags. Next, they'll use specific character styles to create the <b> and <i> tags for bold and italic text.

Mapping Styles to Tags

To map the styles to tags in InDesign, the website editor chooses Map Styles to Tags from the pop-up menu on the Structure pane or the pop-up menu on the Tags palette.



You'll find Map Styles to Tags on the pop-up menu in the Structure pane.

To assign a shortcut key for Untag

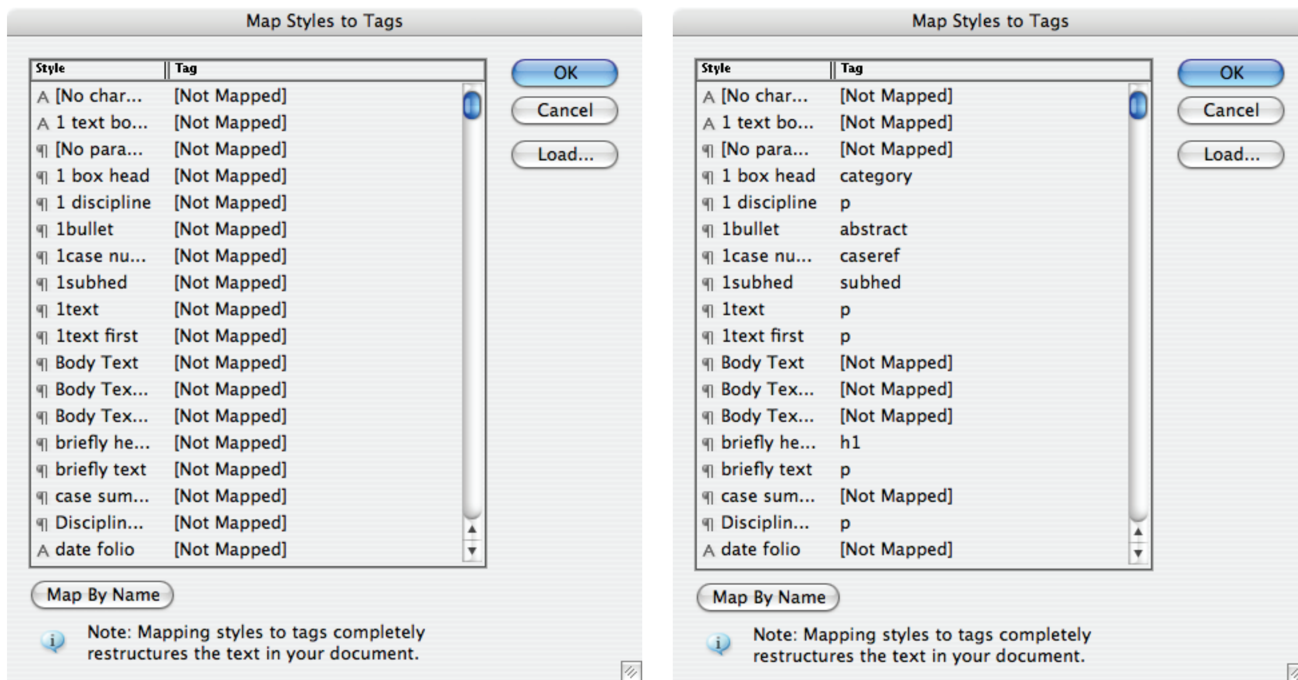
1. Choose Edit > Keyboard Shortcuts.
2. Choose Product Area > Palette Menus.
3. Choose Commands > Tags: Untag.
4. Set Context to XML Selection.
5. Type a shortcut key in the New Shortcut box (for example, Shift+Command+U). Since the context is XML Selection, this shortcut is less likely to conflict with other shortcut key assignments.
6. Click Assign, then click OK.

The Map Styles to Tags dialog box shows a list of character and paragraph styles and a pop-up menu that specifies which element to use for which style. The web editor finds each character style used to format bold or emphasized text and chooses the <b> tag for each. The editor then does the same for the character styles used to format italic text, choosing the <i> tag for each. Text that is both bold and italic, would ideally be tagged with both the <b> and <i> tags, but for now, it is tagged with a <bi> tag. Later on the web editor converts the <bi> elements to <i> elements nested inside a <b> element using XSLT.

```
<b><i>bold italic text here</i></b>
```

All of the paragraph stylesheets will be mapped to <p> tags.

T+L Golf uses many styles, and specifying these mappings can be time consuming. To save time these settings can be loaded from a file that specifies the appropriate mappings. To load mappings for Styles to Tags, the web editor clicks the Load button and chooses the file that contains the mappings.



At left: before style mappings are loaded. At right: Styles to Tags Map is loaded.

Clicking OK completes the style-mapping process. The Structure pane now shows a number of Story elements nested below the Root element. Each Story element contains the <p> elements, and inside some of the <p> elements are <b> and <i> elements.

Note: *All style and tag definitions from the source document will be added to the document. To avoid adding extraneous style and tag definitions, load style maps from a document that contains matching styles and tags.*

Organizing the Elements

InDesign requires that an element tag is assigned to each text frame (or chain of text frames) before its text can be tagged. Since none of the text frames had element tags assigned to them, InDesign automatically assigned a Story tag. It's tempting to interpret Story as meaning an article complete with headline, deck, captions, body text, and so on. In cases where body text follows headlines in a linked frame, it is possible to get an entire article contained in this single tag. However, in most cases, an article will consist of two or more text flows for headline, body text, and other elements.

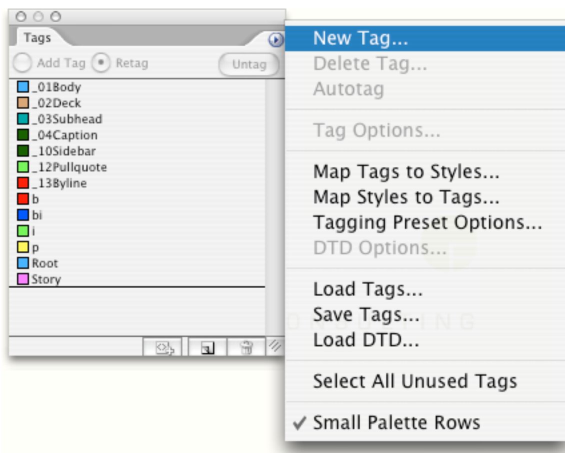
Note: *InDesign CS2 allows users to change the name of the default XML tag. Renaming the Story tag to TextFlow or Component may be more clear.*

The web editor must now group the Story elements by article. These groups will make it possible to export a single XML file for each article. To group XML elements, the editor creates a new element called Group, then uses the Structure pane to drag the Article elements to the Group element.

The InDesign to Tags mapping feature is comprehensive and has created Story elements for text boxes on master pages and text frames on the pasteboard. Even empty text frames are tagged. Since the text in these frames will not be included in any articles, the web editor goes through the Structure pane and untags these story elements first. Double-clicking the Story element tells InDesign to show the associated text frame. If the text appears to be on the pasteboard, or on a master page, the web editor selects the element, then untags it by choosing Untag Element from the pop-up menu on the Structure pane. The editor also untags folio boxes and other extraneous text that will not be included in the article.

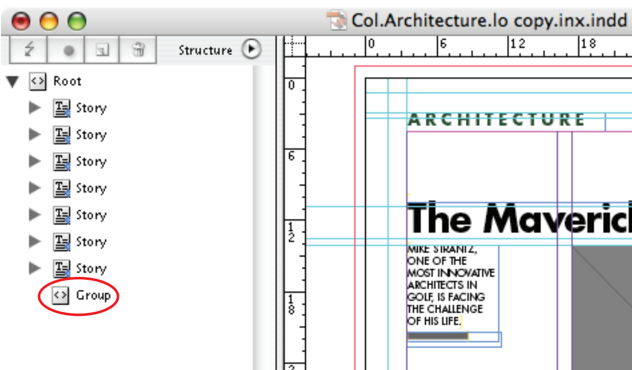
On longer multipage documents, the combined total of text frames on master pages, text frames on pasteboards, and text frames with no text can quickly add up. In these cases, it is helpful to use a script to quickly untag them. See Appendix A for AppleScript and JavaScript scripts for untagging boxes.

After the web editor runs the script, the Structure pane shows only elements that will be included in articles, making the grouping process much easier. To group elements, the web editor must first open the Tags palette by choosing Window > Tags. If the Group tag isn't listed in the Tags palette, it can be created easily by choosing New Tag from the pop-up menu. The editor then specifies the name "Group" and selects a color to distinguish it from other tags.



To create a new tag, select New Tag from the pop-up menu in the Tags palette.

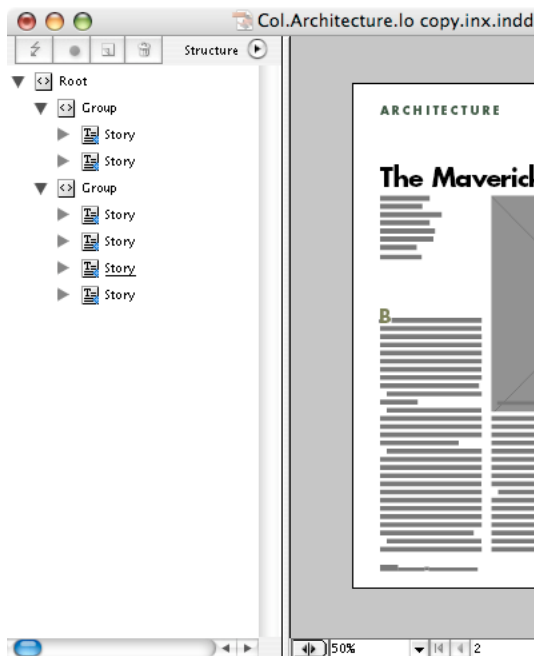
The Group tag now appears in the Tags palette list. The editor then creates a new Group element in the document's XML structure by selecting the Root element, and then choosing New Element from the pop-up menu in the Structure pane. InDesign shows a dialog asking which tag to apply to this new element. The editor chooses the Group tag from the pop-up list and a new Group tag appears.



Newly added elements appear at the end of the list.

To add Story elements, the editor clicks the element and drags it to the Group tag. Multiple Story elements can be added to the group by Ctrl-clicking (Windows®) or Command-clicking (Mac® OS) each element, and then dragging the selected elements to the Group tag.

When InDesign documents contain many Story elements for many articles, it can be a challenge to keep track of which Story elements correspond to which blocks of text on a page. To make this process easier, choose Show Text Snippets from the pop-up menu on the Structure pane. When this option is selected, the first 32 characters of each tagged text element appears on the Structure pane. When text or a text box is selected on a layout, the Structure pane underlines the corresponding element. Double-clicking an element displays the tagged object or text on the layout.



Story elements grouped in the Structure pane.

Each group represents a separate article, so the web editor exports each one to a separate XML file. To export, the editor selects the element that will be the top element in the exported file and then chooses File > Export. The export dialog appears. The editor chooses XML as the format and specifies a name for the XML file. Clicking Save opens the Export XML options dialog. The web editor views the General tab and checks the Export From Selected Element option, then specifies a UTF-8 encoding. Because the editor will not be including any images from the article, image options should remain unchecked on the Images tab. Finally, the editor checks Export to generate an XML file for the article, which follows this format:

```
<Group>
   <Story>
      <p>headline</p>
   </Story>
   <Story>
      <p>body text</p>
      <p>body text</p>
      <p>body text</p>
   </Story>
</Group>
```

We now have a nicely formatted XML file that holds all of the article's content. The web editor should next convert this XML file into something that the magazine's content management system can accept. The CMS needs an HTML fragment that can be incorporated into a complete HTML page. The HTML fragment contains a subset of the HTML tags defined in XML file—in this case, the <p>, <b>, and <i> tags—so the Group and Story tags must be removed. Because *T+L Golf's* website doesn't support Unicode, characters in the HTML fragment must be encoded in the ISO-8859-1 format (rather than the UTF-8 format generated by InDesign). Many of the high-ASCII characters and symbols supported by ISO-8859-1 are further converted to lower ASCII values or HTML entities.

The web editor can easily remove the Group and Story tags using a standard text editing application or Adobe GoLive®, but changing UTF-8 to ISO-8859-1 encoding is a challenge because some text editing applications don't support the UTF-8 format. Once the text is converted to ISO-8859-1, however, the text editing application can be scripted to change high-ASCII characters to lower ASCII values or HTML entities. The web editor *could* automate this conversion process using the right text editor, but since the file is well-formed XML, a much more powerful tool is available.

Around 1999, the XML community began working on a specification for transforming XML documents from one format to another. Two W3C standards resulted: XSLT and XPath. Together, these standards provide an easy way to transform XML into plain text, HTML, or other forms of XML. XSLT and XPath technologies can be used in a number of ways. In this scenario, we provided our web editor with an XSLT template file that will be used by an XSLT transformation program to convert the InDesign XML files into the desired HTML fragments. To use the XSLT file, the web editor drags it and a group of XML files onto an AppleScript droplet. This droplet transforms each XML file into an HTML file using the XSLT file and saves each resulting file using the name of the XML file plus an .HTML extension.

Note: *To use AppleScript to transform XML with XSLT, you'll need a scripting addition from Late Night Software called XSLT Tools. This added functionality is provided by a scripting addition from Late Night Software called XSLT Tools (http://latenightsw.com/freeware/XSLTTools/index.html). The XSLT Tools Scripting Addition includes the Apache Software Foundation's Xerces-C 2.3.0 XML parser and Xalan-C 1.6 XSLT processor to process XSLT.*

The XSLT Template

The details of the XSLT specification are beyond the scope of this document, but certain parts of the XSLT template are worth noting. In the example below, line 5 indicates that the output method is HTML and the encoding used is ISO-8859-1. Lines 12-14 and 16-18 handle the Group and Story elements by removing the <Group> and <Story> tags while maintaining their contents. On lines 20-30, tags and content are preserved for <p>, <b>, and <i>. On line 32, the <bi> elements are converted to an <i> element enclosed by a <b> element. Finally, on line 40, characters in text nodes are mapped from high-ASCII characters to low-ASCII characters, and HTML entities with the XPath translate function. Note that many of the entities are not specified here. The XSLT processor generates many of these entities automatically because HTML is specified as the output method (as shown in line 5).

```
1  <?xml version='1.0' encoding='iso-8859-1'?>

2

3  <xsl:stylesheet version='1.0' xmlns:xsl='http://www.w3.org/1999/XSL/
        Transform'>

4

5  <xsl:output method='HTML' version='1.0' encoding='iso-8859-1' indent='no'/>

6

7  <xsl:template match="/">

8     <xsl:apply-templates />

9  </xsl:template>

10

11

12 <xsl:template match="Group">

13    <xsl:apply-templates />

14 </xsl:template>

15

16 <xsl:template match="Story">

17    <xsl:apply-templates />

18 </xsl:template>

19

20 <xsl:template match="p">

21 <p><xsl:apply-templates /></p>

22 </xsl:template>

23

24 <xsl:template match="b">
```

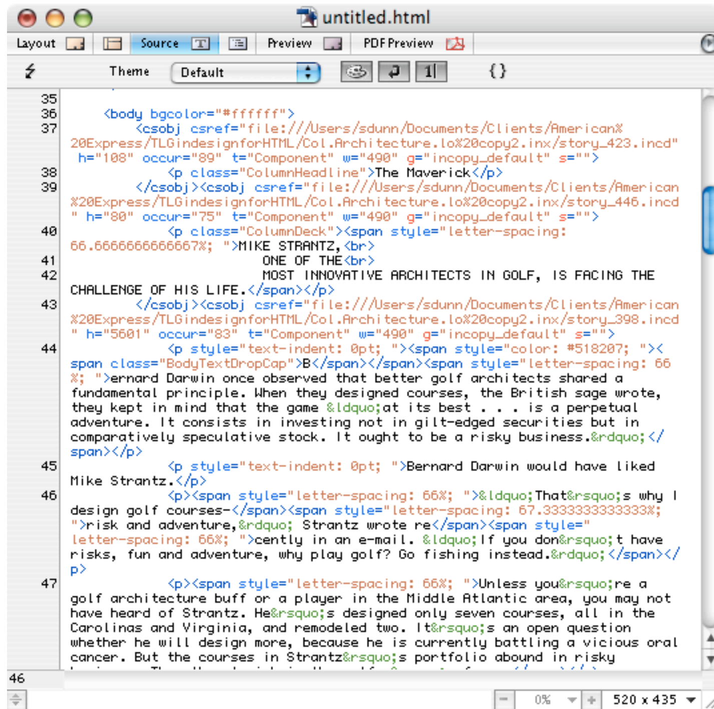
```
25 <b><xsl:apply-templates /></b>
26 </xsl:template>
27
28 <xsl:template match=”i”>
29 <i><xsl:apply-templates /></i>
30 </xsl:template>
31
32 <xsl:template match=”bi”>
33 <b><i><xsl:apply-templates /></i></b>
34 </xsl:template>
35
36 <xsl:template match=”text()”>
37    <xsl:variable name=”apos”>’</xsl:variable>
38    <xsl:variable name=”quot”>”</xsl:variable>
39
40    <xsl:value-of select=”translate(.,
41    ‘&#8232;&#305;`&#733;&#8260;-&#8224;&#8226;&#8482;&#8217;&#402;&#8230;&#3
         38;&#339;&#8211;&#8220;&#8221;&#8216;&#376;&#8249;&#8250;&#8225;&#8218
         ;&#8222;&#8240;&#710;&#732;&#8364;&#8800;&#8734;&#8804;&#8805;&#8706;&
         #8721;&#8719;&#960;&#8747;&#170;&#186;&#937;&#8730;&#8776;&#8710;&#9674
         ;&#183;&#728;&#729;&#730;&#184;&#731;&#711;&#63743;&#9632;&#8233;’
42
43 ,
44
45       concat(‘ i’,$apos,$quot,’/&#45;&#134;&#149;&#153;’,$apos, ‘&#131;&#133;
         &#140;&#156;&#150;’,$quot,$quot,$apos,’&#159;&#139;&#155;&#135;&#130;&
         #132;&#137;&#136;&#152; ‘))”
46        />
47
48 </xsl:template>
49 </xsl:stylesheet>
```

Creating XHTML with InDesign and GoLive

T+L Golf had some fairly specific requirements for its HTML that mandated the two-step XML generation and transformation approach described in the first case study. This approach could be adopted by anyone wanting to create very basic HTML fragments. However, if your requirements are less stringent, or if formatting must be retained with CSS, it is easier to use InDesign together with GoLive to create HTML files. In the previous scenario, *T+L Golf's* web editors generated simple HTML files for each article. To do this with GoLive, you can simply package the InDesign file for GoLive. GoLive uses the package (a collection of files in a single folder) to understand what's happening on the InDesign page. In GoLive CS, open the toc.HTML file, and a window appears showing a preview of the InDesign page. To view the packaged InDesign file in GoLive CS, open the toc.HTML file. In GoLive CS2, open the package.idpk file. Now create a new HTML page for the first article. If XML content is desired, make a new XHTML page (in GoLive CS, choose File > New Special > XHTML; in GoLive CS2, choose File > New, then click Pages and make sure that the XHTML option is checked). Drag text boxes from the InDesign preview onto the new XHTML page. When boxes are dragged to the page, their contents are automatically encoded into HTML fragments. Most of the text formatting is preserved using span tags with CSS style attributes. Unicode and high-ASCII characters are translated into HTML entities; <p></p> tags enclose each paragraph, and forced returns are translated to <br>.

Note: *If your document requires strict XHTML adherence (and it will if you want to work with it as an XML file or transform it with XSLT), the <br> tags must be manually changed to <br />. In GoLive CS2, however, changing the document type to HTML and then back to XHTML changes the <br> tags to <br /> tags.*

Looking at the source view in GoLive, we see a well-formed XML file that adheres to XHTML standards. Since the file is also XML, it could be further transformed to a format that adheres to the structure required by a CMS.



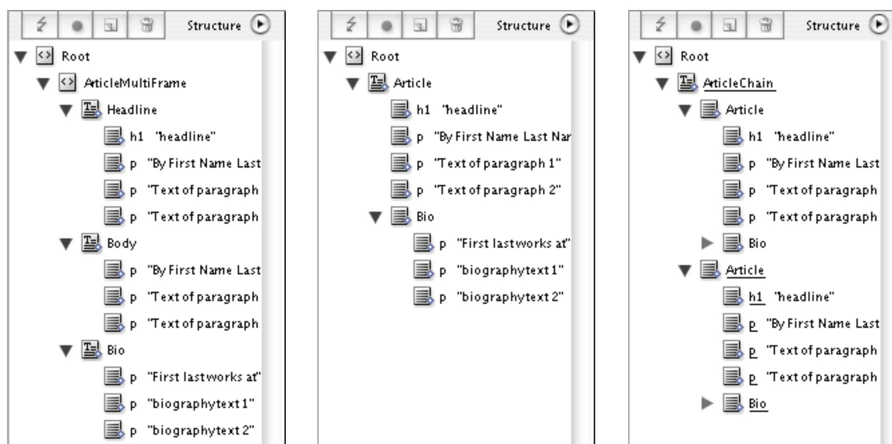
A portion of the XML file created from the InDesign package.

Case Study #2: How *Texas Lawyer* Generates Complete XML Files with Structure and Attributes

At *T+L Golf*, headlines, bylines, and other article elements were tagged as the web editor added HTML fragments to the content management system (CMS). Other publications require a more comprehensive approach. In these situations, the XML file must identify the headlines, bylines, body text, and so on as separate elements. Additionally, the publisher may need information in the XML file that identifies the article type, category, section, unique subject matter, page number, section, and other data.

Texas Lawyer, a weekly newspaper in Dallas, sends its articles to a CMS, where they are converted into various formats for websites and syndication. This CMS requires XML that notes the article's headline, byline, body text, subhead, section, author biography, and page number. The articles in one section of the newspaper contain information about specific court cases. For these articles, additional tags are used to note the category, abstract, and specific case reference.

The article text can be structured in InDesign in one of three ways. Most articles will have text in multiple unlinked text frames. For example, in a multiframe article, the headline might appear in one frame while the body text appears in a second frame, and a third frame might contain a biography of the article's author. A single-frame article contains the complete article in a single text flow. Finally, a third type, the chained article, coexists with many other articles in one long text flow. Each of these formats has its own XML structure.

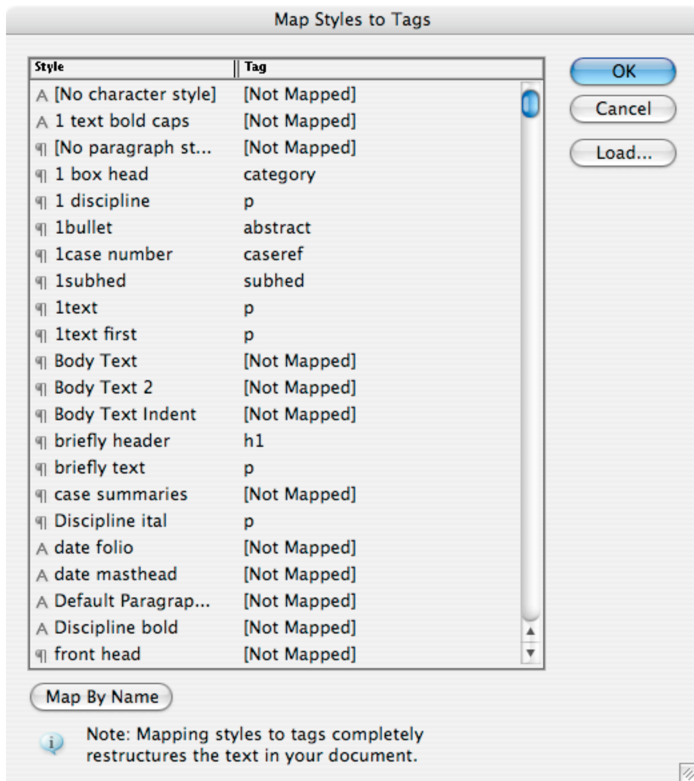


Element groupings in the Structure pane for multiframe (left), single frame (middle), and article chain (right) articles.

These three basic structures can accommodate most article types using the InDesign XML Structure pane. To create these structures with InDesign, the web editor at *Texas Lawyer* builds on the techniques outlined in the previous examples and add steps for marking article components and attributes. They also employ a new technique for grouping article components for articles that run sequentially in one long text flow.

Better Results with Styles to Tags

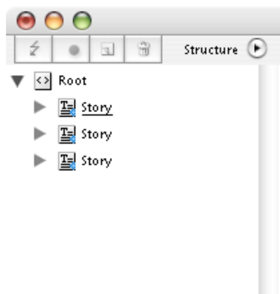
In the earlier Styles to Tags conversion example, the web editor opted to map every paragraph style to a tag. This ensured that all text was tagged, even if it was formatted with the wrong paragraph style, or without a paragraph style at all. Unfortunately, it produced a number of unwanted elements for items on master pages, folios, and so on. At *Texas Lawyer*, designers carefully use styles to format all text for articles, and they use different styles to format text that will not be included in article text. When they map styles to tags, they can selectively apply tags to content only.



Unnecessary styles are excluded from the Styles to Tags conversion process when Not Mapped is selected.

Keeping the document clean with no stray text frames on pasteboards and no empty text frames helps ensure that empty or otherwise unwanted elements do not appear on the XML tree in the Structure pane.

When *Texas Lawyer* runs the Styles to Tags conversion, a short, clean list of Story elements appears below the Root element.



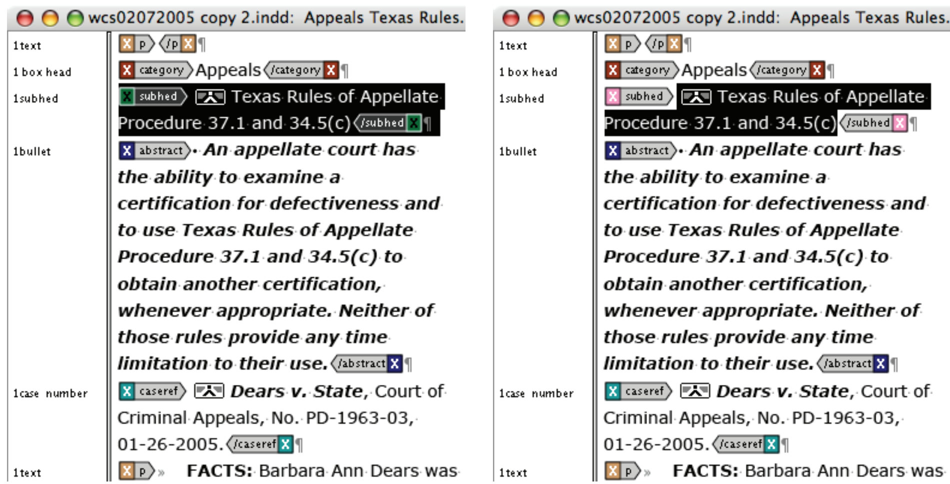
The XML tree resulting from the Styles to Tags conversion.

Grouping Elements of Inline Articles

The Case Summaries section of *Texas Lawyer* has only three Story elements that contain content to be exported. However, the second Story element contains 19 individual Case Summary articles that appear in 10 different categories. To export individual XML files for these articles, the headline, summary, case reference, and body text elements must be grouped. Section, category, type, and page attributes will be applied to the Group element.

Grouping elements for inline articles is different from grouping multiple frames. With multiple frames, the elements were dragged to a new Group element using the Structure pane. Doing that with inline elements will change the text in the document.

To group the inline elements without changing the text, carefully select the text of the elements to be marked up and apply the Group tag. Notice that the Add Tag and Retag buttons are grayed out when text is selected. To make sure that a tag is not added in the wrong spot, use the Edit In Story Editor feature (Edit > Edit in Story Editor) to select text and its enclosing element markers before applying the tag. This view of the text clearly shows where elements start and end and makes it easy to see when a complete element is selected.



The XML file open in Story Editor. On the left, an entire element is selected; on the right, just the content of the element is selected.

Grouping a Range of Inline Elements with the Structure Pane

By itself, the Structure pane cannot group inline elements without changing the text. But since the Structure pane provides a neat hierarchy of the XML elements, it can make it easier to select the ranges of elements to be grouped. Once a range of elements is selected, a script can be used to nest them in a group. See Appendix B for the script source code to group a range of inline elements.

Using Attributes to Record Article Metadata

The CMS that will ultimately receive the XML files needs to know things about the article that are not in the article text, such as the publication name, section, and issue date. This type of information is commonly referred to as metadata. In some cases, XML formats store metadata in the text node of an XML element. In the following example, the publication name is stored as a text node in the Publication element.

```
<Article>
   <Publication>Texas Lawyer</Publication>
   <Headline>Headline</Headline>
   <p>Article Text</p>
   <p>Article Text</p>
</Article>
```

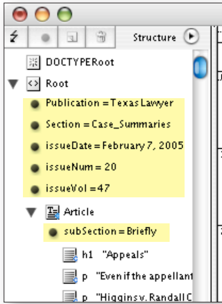
For InDesign to produce XML like this, the publication name must either appear in the article, or in an unplaced element. Since the publication name is unlikely to be repeated in each article, the content must be stored in an unplaced element or in an attribute. Unplaced elements can be created by using a temporary text frame to hold the text elements, then deleting the text frame once the text elements have been created. The text elements can later be edited by placing them in a temporary text frame. Unplaced elements can easily be copied and pasted in other places in the XML tree.

In this example, the publication name does not appear as content in the article, so it is classified as metadata. The XML conventions at *Texas Lawyer* call for metadata—and consequently the publication name—to be stored as an attribute of the Article element as follows:

```
<Article publication=”Texas Lawyer”>
   <Headline>Headline</Headline>
   <p>Article Text</p>
   <p>Article Text</p>
</Article>
```

The publication name and section are fixed for each section template of the publication, so these values are stored as attributes of the Root element. The volume, issue number, and issue date are also stored as Root attributes, and they must be updated for each issue.

The section templates have placeholder boxes for their subsections. The subsections have attributes assigned to them as well.



Attributes of the Root and Article elements appear in the Structure pane.

One section of the publication contains a chain of articles organized by category. These articles run one after another in a single text flow. A category headline appears once to start the category, and articles belonging to that category appear sequentially beneath it. Even though the category text appears only once, each of the exported articles needs this category metadata. The metadata is recorded as an attribute of each headline for each article in the category.

Manually adding an attribute to an element is a simple matter of selecting the element, clicking Add An Attribute at the top of the Structure pane, and entering the name and value. Since *Texas Lawyer* uses five attributes for about 50 articles per issue, about 250 attributes must be assigned. Fortunately, a script can be used to automate the process. The script in Appendix C finds the Publication metadata stored as attributes in the Root element and then assigns those attributes to each article. It then loops through each element in the article chain where articles are categorized. When a category name is found, it is stored in a variable, and when an article beneath it is found, a category attribute is added to the Article tag using the category name stored in the variable. In a similar fashion, the subsection attribute for the article chain is also assigned to each article.

Validating XML Before Exporting

XML is very flexible, and with InDesign, users can assign XML tags to any text they like. They can also customize XML tag names, attributes, and so fourth. Content management systems require a strict XML structure, and so the XML produced by InDesign must be correctly formatted with the XML elements placed in the right location and order, and have the correct element and attribute names. The rules for a specific XML format can be contained in a Document Type Definition (DTD). Quite simply, a DTD is a file that defines the elements and data structure that can be contained in an XML document. A DTD can specify that a <p> element must be a sibling of another element like <Article>. It also specifies a proper name for all elements in the XML document.

InDesign can use a DTD to verify that its XML is properly formatted. This process is called validation. When InDesign validates an XML structure, it compares its XML to the imported DTD. If it finds a discrepancy, InDesign reports discrepancies and provides suggestions for resolving them.

Texas Lawyer uses a DTD to validate its XML before exporting the articles. This ensures that the XSLT transformation runs smoothly, and that the CMS receives the required XML.

In many cases, the CMS's DTD cannot be used. Content management systems usually have a DTD that defines the single XML structure for a single article. InDesign XML has the ability to structure XML for multiple articles. These articles may have varying structures, with some articles consisting of multiple frames and others appearing in a chain of articles within a single text flow. These more advanced and variable structures are not included in the CMS's DTD, so a custom DTD is needed.

While creating a DTD is beyond the scope of this document, consider the following approach. Start out by creating an InDesign XML tree that includes all of the possible structures that must be validated. Our example has Article elements that might exist as a direct descendant of the Root element, or as a descendant of an Article Chain element. Articles themselves may contain headlines, bylines, body text, bios, subheads, and abstracts. Articles may also consist of elements that group headlines and bylines into a single element. A basic XML structure for *Texas Lawyer* with sample content appears as follows:

```
<?xml version=”1.0” encoding=”UTF-8” standalone=”yes”?>
<Root Publication=”Texas Lawyer” Section=”Case_Summaries”
      issueDate=”February 7, 2005” issueNum=”20” issueVol=”47”>
  <Article Publication=”Texas Lawyer” Section=”Case_Summaries”
      issueDate=”February 7, 2005” issueNum=”20” issueVol=”47”>
    <h1>Appeals</h1>
      <p>paragraph text</p>
      <p>paragraph text</p>
    <h1>Civil Practice</h1>
      <p>paragraph text</p>
  </Article>
  <ArticleChain>
    <category>Appeals</category>
    <Article Category=”Appeals” Publication=”Texas Lawyer”
      Section=”Case_Summaries” issueDate=”February 7, 2005”
      issueNum=”20” issueVol=”47”>
        <h1>Headline here</h1>
        <abstract>abstract text here</abstract>
        <caseref>Case reference here</caseref>
        <p>paragraph text</p>
        <p>paragraph text</p>
        <subhed>subhed text here</subhed>
        <p>paragraph text</p>
    </Article>
    <category>Criminal Practice</category>
    <Article page=”00” Category=”Criminal Practice” Publication=”Texas
      Lawyer” Section=”Case_Summaries” issueDate=”February 7, 2005”
      issueNum=”20” issueVol=”47”>
        <h1>Headline here</h1>
        <abstract>abstract text here</abstract>
        <caseref>Case reference here</caseref>
        <p>paragraph text</p>
        <p>paragraph text</p>
        <subhed>subhed text here</subhed>
        <p>paragraph text</p>
    </Article>
    <Article page=”00” Category=”Criminal Practice” Publication=”Texas
      Lawyer” Section=”Case_Summaries” issueDate=”February 7, 2005”
      issueNum=”20” issueVol=”47”>
        <h1>Headline here</h1>
        <abstract>abstract text here</abstract>
        <caseref>Case reference here</caseref>
        <p>paragraph text</p>
        <p>paragraph text</p>
        <subhed>subhed text here</subhed>
        <p>paragraph text</p>
    </Article>
  </ArticleChain>
  <ArticleMultiBox page=”00” Publication=”Texas Lawyer” Section=”Main”
      issueDate=”February 7, 2005” issueNum=”20” issueVol=”47”>
      <component>
        <h1>Headline here</h1>
```

```
            <deck>Headline here</deck>
            <byline>Headline here</byline>
            <p>paragraph text</p>
            <p>paragraph text</p>
         </component>
         <bio>
            <p>bio text here</p>
            <p>bio text here</p>
         </bio>
   </Article>
</Root>
```

The following DTD defines the XML structure for the sample above. This DTD contains element type declarations for each entity and attribute list declarations for entities with attributes. Each element has restrictions on what it may or may not contain. For example, the Article Chain element may contain only Article elements and Category elements. Elements also have rules about their attributes. Theses attributes in turn have their own rules about the values they may contain. On line 12 of the sample below, the attribute list declaration for the Article element calls for a number of attributes, including Publication and Section. Line 3 specifies that the Publication attribute must always be "Texas Lawyer." Line 4 specifies that the "Section" attribute is optional and that it may be either "Main" or "Case_Summaries." For an introduction to document type declarations, see http://www.w3schools.com/dtd/.

```
1  <!ELEMENT Root  (Article | ArticleChain | ArticleMultiBox )*>
2  <!ATTLIST Root
3   Publication  CDATA #FIXED "Texas Lawyer"
4   Section (Case_Summaries | Main) #IMPLIED
5   issueDate CDATA #REQUIRED
6   issueNum CDATA #IMPLIED
7   issueVol CDATA #IMPLIED
8   >
9
10 <!ELEMENT ArticleChain (category | Article)*>
11 <!ELEMENT Article (( b | i | h1 | subhed| p )* | (subhed, abstract,
       caseref, (p  | caseref )*))>
12 <!ATTLIST Article
13  Publication  CDATA #FIXED "Texas Lawyer"
14  Section (Case_Summaries | Main) #IMPLIED
15  issueDate CDATA #REQUIRED
16  issueNum CDATA #IMPLIED
17  issueVol CDATA #IMPLIED
18  Category CDATA #IMPLIED
19  subSection (FullSummaries|Briefly|Discipline) #REQUIRED
20  >
21
22 <!ELEMENT ArticleMultiBox (Component+)>
23
24 <!ATTLIST ArticleMultiBox
25  Publication  CDATA #FIXED "Texas Lawyer"
26  Section (CaseSummaries | Main) #IMPLIED
27  issueDate CDATA #REQUIRED
28  issueNum CDATA #IMPLIED
29  issueVol CDATA #IMPLIED
30  Category CDATA #IMPLIED
31  subSection (FullSummaries|Briefly|Discipline) #REQUIRED
32  >
33
```

```
34 <!ELEMENT Component ( b | i | h1 | subhed| p| (subhed, abstract,
       caseref, (p  | caseref )*))>
35
36 <!ELEMENT h1 (#PCDATA | b | i |p)*>
37
38 <!ELEMENT subhed (#PCDATA | b | i |p)*>
39
40 <!ELEMENT abstract (#PCDATA | b | i |p)*>
41
42 <!ELEMENT byline (#PCDATA | b | i |p)*>
43
44 <!ELEMENT caseref (#PCDATA | b | i |p)*>
45
46 <!ELEMENT category (#PCDATA | b | i |p)*>
47
48 <!ELEMENT p (#PCDATA | b | i)*>
49 <!ATTLIST p
50  href CDATA #IMPLIED
51  >
52
53 <!ELEMENT b (#PCDATA)>
54
55 <!ELEMENT i (#PCDATA)>
```

When this DTD is imported into the *Texas Lawyer* document, the document can be validated using the Validate button on the Structure pane. Doing this points to an error in the Article Chain—an empty <p> element at the beginning of the text. InDesign indicates that, according to the DTD, the <ArticleChain> may contain only <Category> and <Article> elements. InDesign also provides options for solving this problem. Choosing to untag the element while preserving content is a handy option that does not affect the text. The XML tag is simply removed from the empty paragraph. Clicking the Validate button again shows that there are no errors in the document and that the XML is ready to be exported.

Tip: *To speed up the process of resolving XML discrepancies, assign keyboard shortcuts to the Validate button, and to the Show Next/Previous error functions (Edit > Keyboard Shortcuts). You can also Ctrl-click (Windows) or Command-click (Mac OS) a suggestion to apply the suggestion and perform the validation.*

Exporting Many Articles

Each issue of *Texas Lawyer* has so many articles that a script was written to automate the process. The script in Appendix D looks for Article elements in the Structure pane, exporting each one it finds to a folder specified by the user.

The Final Transformation

In the earlier example we looked at a simple XSL transformation that would put the finishing touches on an XML file before sending it to the CMS. At *Texas Lawyer*, these transformations occur inside the CMS, so no further transformation is necessary here.

Conclusion

Both of these case studies demonstrate a broad range of the powerful XML features in InDesign. These features combined with its advanced scripting architecture give InDesign seemingly limitless potential for addressing the XML needs of today's publishers.

Notes

InCopy Notes and InDesign CS

T+L Golf uses Adobe InCopy® to allow editors and designers to collaborate on development of the magazine pages. InCopy adds a feature to InDesign that allows users to insert nonprinting notes into text. With InDesign CS, these notes can create an undesired effect on the Styles to Tags mapping process. During Styles to Tags, when InDesign CS sees a note, it ends the element just before the note starts and begins a new element just after the note ends. Therefore paragraphs with notes are split into multiple <p> elements. (This issue does not affect InDesign CS2.) To keep each paragraph contained in a single <p> element, the web editors delete every note in the document before invoking Styles to Tags. Choosing Notes > Remove Notes from the Story menu removes notes for a single text flow. Repeating the Remove Notes step for each text flow will remove all notes from the document. To save time, the web editor can run a script to remove all the notes from the document.

AppleScript:

```
tell document 1 of application "InDesign CS" to delete notes of stories
```

JavaScript:

```
app.activeDocument.stories.everyItem().notes.everyItem().remove();
```

Appendices A1-D2: Scripting Samples

These appendices include sample scripts written in AppleScript and JavaScript for untagging boxes, grouping inline elements, assigning attributes, and exporting multiples XML files. InDesign provides extensive support for AppleScript, JavaScript, and VBScript, which greatly enhances the process of repurposing content using XML.

Appendix A1 - AppleScript for Untagging Boxes

```
on run
   my untagmasterboxes()
   my unTagPasteBoardBoxes()
   tell application "InDesign CS" to ¬
      my UnTagEmpties(every XML element of XML element 1 of document 1)
end

on UnTagMasterBoxes()
   tell application "InDesign CS"
      tell document 1
         repeat 5 times
            tell every group to ungroup
         end repeat
         set AllMasterSpreads to every master spread

         repeat with masterSpread in AllMasterSpreads
            if (count of every text frame of masterSpread) > 0 then
               repeat with p in parent story of every text frame of
                                             masterSpread
                  if associated XML elements of p … {nothing} then ¬
                     untag item 1 of  (associated XML elements of p)
               end repeat
            end if
         end repeat
      end tell
   end tell
end UnTagMasterBoxes

on unTagPasteBoardBoxes()
   tell application "InDesign CS"
      tell document 1
         repeat with C from 1 to count of every story
            set ElementsOfStoryC to associated XML elements of story C
            if ElementsOfStoryC ≠ nothing then
               if class of parent of every text frame of story C does not
         contain page then
                  untag item 1 of associated XML elements of story C
               end if
            end if
         end repeat
      end tell
   end tell
end unTagPasteBoardBoxes

on UnTagEmpties(theElementList)
   repeat with theElement in theElementList
      tell application "InDesign CS"
         if (count of every XML element of theElement) = 0 then
            set TheContents to contents of properties of theElement
            if TheContents = "" then untag theElement
         else if (count of every XML element of theElement) > 0 then
            my UnTagEmpties(every XML element of theElement)
            if (count of every XML element of theElement) = 0 then
               set TheContents to contents of properties of theElement
```

```
          if TheContents = "" then
            untag theElement
          end if
        end if
      end if
    end tell
  end repeat
end UnTagEmpties
```

Appendix A2 - JavaScript for Untagging Boxes

```
//DESCRIPTION: Untagger

// Method to determine if object is in array
Object.prototype.isInArray = function(myArray){
   for (var i=0; myArray.length > i; i++) {
      if(myArray[i] == this){
         return true;
      }
   }
   return false;
}

unTagMasterBoxes();
unTagPasteBoardBoxes();
unTagEmpties(app.activeDocument.xmlElements[0].xmlElements);

// +++++++ Functions Start Here ++++++++++++++++++++++

function unTagEmpties(theElementList) {
   // Changing the order of the tests simplifies this function
   var TELlen = theElementList.length;
   for (var j = TELlen - 1; j >= 0; j--) {
      if (theElementList[j].xmlElements.length > 0) {
         unTagEmpties(theElementList[j].xmlElements);
      }
      if (theElementList[j].xmlElements.length == 0) {
         var TheContents = theElementList[j].contents; // properties.
         contents? Isn't that the same?
         if (TheContents == "") { theElementList[j].untag();}
      }
   }
}

function unTagPasteBoardBoxes() {
   // Note: this function, as written, will also untag any stories that are
         inline or anchored
   var myDoc = app.activeDocument;
   var myStories = myDoc.stories;
   var Slen = myStories.length;
   for (var j = 0; Slen > j; j++) {
      var myTFsParents = myStories[j].textFrames.everyItem().parent;
      if (myTFsParents.constructor.name != "Array") { myTFsParents = [myTFs-
         Parents] }
      var TFPlen = myTFsParents.length;
      var noPage = true;
      for (var k = 0; TFPlen > k; k++) {
         if (myTFsParents[k].constructor.name.isInArray(["Page","Character"]))
         {
            noPage = false;
            break;
         }
      }
      if (noPage) {
         unTagStory(myStories[j]);
```

```
      }
    }
  }

function unTagMasterBoxes() {
  var myDoc = app.activeDocument;
  while (true) {
    var myGroups = myDoc.groups;
    if (myGroups.length == 0) { break } // continue until there are no
      groups
    myDoc.groups.everyItem().ungroup();
  }
  var AllMasterSpreads = myDoc.masterSpreads;
  var MSlen = AllMasterSpreads.length;
  for (var j=0; MSlen > j; j++) {
    var myTFs = AllMasterSpreads[j].textFrames;
    var TFlen = myTFs.length;
    for (var k=0; TFlen > k; k++) {
      unTagStory( myTFs[k].parentStory)
    }
  }
}

function unTagStory (theStory) {
  try { // Story might not have an associated XML element
    theStory.associatedXMLElements.untag();
  } catch (e) { }
}
```

Appendix B1 - AppleScript for Grouping Inline Elements

```
tell application "InDesign CS"
  set s to selection
  if class of item 1 of s is XML element then
    set p to parent of item 1 of s
    try
      if (id of parent story of p) … (id of parent story of item 1 of s)
         then ¬
         set goodSelection to false
    on error
      set goodSelection to false
    end try
    if not goodSelection then
      beep
      display dialog "This script may only be used to group a range of
                      elements in a single story"
    end if

    set l to index of item 1 of s
    set h to l
    repeat with theElement in s
      if index of theElement < l then set l to (index of theElement) as
              integer
      if index of theElement > h then set h to (index of theElement) as
              integer
    end repeat

    set RangeStart to (story offset of XML element l of p) + 1

    if h < (count of every XML element of p) then
      set RangeEnd to (story offset of XML element (h + 1) of p)
    else
      set RangeEnd to length of parent story of item 1 of s
    end if

    set gTag to make new XML element at p with properties {markup
                    tag:"Group"}
    make new XML attribute at gTag with properties {name:"findme",
                    value:"1"}
    markup (text from character (RangeStart) to character (RangeEnd) of
                    (parent story of (item 1 of s))) using gTag
    set gTagNew to every XML attribute of every XML element of p whose
                    name = "findme"
    set gTagNew to parent of item 1 of gTagNew
    delete XML attribute "findme" of gTagNew
    select gTagNew
  else
    beep
    display dialog "One or more XML elements must be selected."
  end if
end tell
```

Appendix B2 - JavaScript for Grouping Inline Elements

```
//DESCRIPTION: Group selected XML elements

var theMsg = "One or more XML elements must be selected"
if ((app.documents.length !=0) && (app.selection.length > 0)) {
   var s = app.selection;
   if (s[0].constructor.name != "XMLElement") {
      errorExit(theMsg)
   }
   // deviates from AppleScript; once we find a bad one, who cares about the
        rest?
   theMsg = "This script may only be used to group a range of elements in a
        single story";
   for (var C=0; s.length > C; C++) {
      var p = s[C].parent;
      try {t
         if (p.parentStory != s[0].parentStory) {
            errorExit(theMsg)
         }
      } catch (e) {
         errorExit(theMsg);
      }
   }
   // once we get here, the selection is good.
   var I = s[0].index;
   var h = I;
   for (var j=0; s.length > j; j++) {
      I = Math.min(s[j].index,I);
      h = Math.max(s[j].index,h);
   }

   var RangeStart = p.xmlElements.item(I).storyOffset; // "+1" not needed
        because JS counts from zero
   var RangeEnd; // declares local variable without setting it
   if (h < p.xmlElements.length) {
      RangeEnd = p.xmlElements.item(h + 1).storyOffset;
   } else {
      RangeEnd = s[0].parentStory.length;
   }

   var gTag = p.xmlElements.add({markupTag:"Group"});
   //gTag.xmlAttributes.add({name:"findme", value:"1"});
   s[0].parentStory.texts[0].characters.itemByRange(RangeStart,RangeEnd).
        markup(gTag);
   app.select(p.xmlElements[I]);
} else {
   errorExit(theMsg);
}
// +++++++ Functions Start Here ++++++++++++++++++++++

function errorExit(message) {
   if (app.version != 3) { beep() } // CS2 includes beep() function.
   if (arguments.length > 0) {
      alert(message);
   }
   exit(); // CS exits with a beep; CS2 exits silently.
}
```

Appendix C1 - AppleScript for Assigning Attributes

```
tell document 1 of application "Adobe InDesign CS2"

   set Publication to my getAttribute(XML element 1, "Publication")
   set SectionNm to my getAttribute(XML element 1, "Section")

   set issueDate to my GetLevel3Contents("issueDate")
   set issueVolume to my GetLevel3Contents("issueVol")
   set issueNumber to my GetLevel3Contents("issueNum")

   set Storyelements to every XML element of XML element 1
   repeat with theElement in Storyelements
      set Category to ""
      set subSection to my getAttribute(theElement, "subSection")
      set thePage to my getPage(theElement)

      if name of (markup tag of every XML element of theElement) contains
         "Article" then
         try
            set subSection to (every XML attribute of theElement whose name
         = "subhed") as string
         end try
         set subElements to every XML element of theElement
         repeat with theSubElement in subElements
            tell theSubElement
               if "category" = (name of markup tag) then
                  set Category to (text of theSubElement) as text
               else if "Article" = (name of markup tag) then
                  set thePage to my getPage(theSubElement)
                  my SetAttribute(theSubElement, "Category", Category)
                  my SetAttribute(theSubElement, "subSection", subSection)
                  my SetAttribute(theSubElement, "Publication", Publication)
                  my SetAttribute(theSubElement, "Section", SectionNm)
                  my SetAttribute(theSubElement, "issueDate", issueDate)
                  my SetAttribute(theSubElement, "issueVol", issueVolume)
                  my SetAttribute(theSubElement, "issueNum", issueNumber)
                  my SetAttribute(theSubElement, "page", thePage)
               end if
            end tell
         end repeat
      else if name of (markup tag of theElement) = "Article" then
         my SetAttribute(theElement, "Category", Category)
         my SetAttribute(theElement, "subSection", subSection)
         my SetAttribute(theElement, "Publication", Publication)
         my SetAttribute(theElement, "Section", SectionNm)
         my SetAttribute(theElement, "issueDate", issueDate)
         my SetAttribute(theElement, "issueVol", issueVolume)
         my SetAttribute(theElement, "issueNum", issueNumber)
         my SetAttribute(theElement, "page", thePage)
      end if
   end repeat
end tell

on getAttribute(theElement, theAttributeName)
   tell document 1 of application "Adobe InDesign CS2"
```

```
    if (count of every XML attribute of theElement) > 0 and (name of
      every XML attribute of theElement) contains theAttributeName then
      return value of XML attribute theAttributeName of theElement
    else
      return ""
    end if
  end tell
end getAttribute


on SetAttribute(theElement, AttributeName, AttributeValue)
  tell application "Adobe InDesign CS2"
    tell theElement
      if AttributeValue ≠ "" then
        if every XML attribute = {} or (name of every XML attribute)
      does not contain AttributeName then
          make new XML attribute with properties {name:AttributeName as
      string, value:AttributeValue as string}
        else
          set value of XML attribute AttributeName to (AttributeValue as
      string)
        end if
      end if
    end tell
  end tell
end SetAttribute

on GetLevel3Contents(elementName)
  tell document 1 of application "Adobe InDesign CS2"
    set matchingElements to (every XML element of every XML element of
      XML element 1 whose name of markup tag = elementName)
    if matchingElements ≠ {} then
      return (text of item 1 of matchingElements) as text
    else
      return ""
    end if
  end tell
end GetLevel3Contents

on getPage(theElement)
  tell document 1 of application "Adobe InDesign CS2"
    set TextFrameLengths to length of text of every text frame of parent
      story of theElement
    set ElementPosition to story offset of theElement
    set TextFrameSubTotal to 0
    repeat with c from 1 to count of TextFrameLengths
      set TextFrameSubTotal to TextFrameSubTotal + (item c of
      TextFrameLengths)
      if TextFrameSubTotal ≥ ElementPosition then exit repeat
    end repeat
    try
      return name of parent of text frame c of parent story of
      theElement
    on error
      return "error"
    end try
  end tell
end getPage
```

Appendix C2 - JavaScript for Assigning Attributes

```
//DESCRIPTION: Translation of Set Pub Attribute AppleScript

Object.prototype.isInArray = function(myArray){
   if (myArray.constructor.name != "Array") {
      myArray = [myArray];
   }
   for (var i=0; myArray.length > i; i++) {
      if(myArray[i] == this){
         return true;
      }
   }
   return false;
}

myDoc = app.activeDocument // Global variable
var Publication = getAttribute(myDoc.xmlElements[0], "Publication");
var SectionNm = getAttribute(myDoc.xmlElements[0], "Section");

var issueDate = GetLevel3Contents("issueDate");
var issueVolume = GetLevel3Contents("issueVol");
var issueNumber = GetLevel3Contents("issueNum");

var Storyelements = myDoc.xmlElements[0].xmlElements;
var Slim = Storyelements.length;
for (var j = 0; Slim > j; j++) {
   var theElement = Storyelements[j];

   var Category = "";
   var subSection = getAttribute(theElement, "subSection");
   var thePage = getPage(theElement);

   var theTagNames = [];
   try {
      var theTags = theElement.xmlElements.everyItem().markupTag;
      var Tlim = theTags.length;
      for (var n = 0; Tlim > n; n++) {
         theTagNames.push(theTags[n].name);
      }
   } catch (e) {}
   if ("Article".isInArray(theTagNames)) {
      try {
         subSection = theElement.xmlAttributes.item("subhed").value; //
         Different from AS
      } catch (e) {}
      var subElements = theElement.xmlElements;
      var tLim = subElements.length;
      for (var k = 0; tLim > k; k++) {
         theSubElement = subElements[k];
         if ("category" == theSubElement.markupTag.name) {
            Category = theSubElement.texts[0].contents;
         } else if ("Article" == theSubElement.markupTag.name) {
            setAttribute(theSubElement,"Category", Category);
            setAttribute(theSubElement,"subSection",subSection);
            setAttribute(theSubElement,"Publication",Publication);
```

```
          setAttribute(theSubElement,"Section",SectionNm);
          setAttribute(theSubElement,"issueDate",issueDate);
          setAttribute(theSubElement,"issueVol",issueVolume);
          setAttribute(theSubElement,"issueNum",issueNumber);
          setAttribute(theSubElement,"page",thePage);
        }
      }
    } else if (theElement.markupTag.name == "Article") {
      setAttribute(theElement,"Category", Category);
      setAttribute(theElement,"subSection",subSection);
      setAttribute(theElement,"Publication",Publication);
      setAttribute(theElement,"Section",SectionNm);
      setAttribute(theElement,"issueDate",issueDate);
      setAttribute(theElement,"issueVol",issueVolume);
      setAttribute(theElement,"issueNum",issueNumber);
      setAttribute(theElement,"page",thePage);
    }
}

// +++++++ Functions Start Here ++++++++++++++++++++++

function getAttribute(theElement, theAttributeName) {
    var lim = theElement.xmlAttributes.length;
    for (var j = 0; lim > j; j++) {
      if (theElement.xmlAttributes[j].name == theAttributeName) {
        return theElement.xmlAttributes.item(theAttributeName).value;
      }
    }
    return "";
}

function setAttribute(theElement, AttributeName, AttributeValue) {
    if (AttributeValue != "") {
      if ((theElement.xmlAttributes.length == 0) ||
         (!AttributeName.isInArray(theElement.xmlAttributes.everyItem().
        name))) {
        theElement.xmlAttributes.add(AttributeName,String
        (AttributeValue));
      } else {
        theElement.xmlAttributes.item(AttributeName).value =
        String(AttributeValue);
      }
    }
}

function GetLevel3Contents(elementName) {
    var L2elements = myDoc.xmlElements[0].xmlElements;
    var lim = L2elements.length;
    for (var j = 0; lim > j; j++) {
      var L2lim = L2elements[j].xmlElements.length;
      for (var k=0; L2lim > k; k++) {
        if (L2elements[j].xmlElements[k].markupTag.name == elementName) {
          return L2elements[j].xmlElements[k].texts[0].contents;
        }
      }
    }
```

```
  return "";
}

function getPage(theElement) {
  // The AppleScript for this seems convoluted. Perhaps I'm missing
      something
  var ElementPosition = theElement.storyOffset;
  try {
    if (app.version == 3) { return theElement.parentStory.character
      s[ElementPosition].parentTextFrame.parent.name }
    return theElement.parentStory.characters[ElementPosition].
      parentTextFrames[0].parent.name;
  } catch (e) {
    return "error" //overset
  }
}
t
function errorExit(message) {
  if (app.version != 3) { beep() } // CS2 includes beep() function.
  if (arguments.length > 0) {
    alert(message);
  }
  exit(); // CS exits with a beep; CS2 exits silently.
}
```

FOR MORE INFORMATION

For a comprehensive overview of Adobe InDesign CS2, please visit http://www.adobe.com/products/InDesign/.

Appendix D1 - AppleScript for Exporting Multiple XML Files

```
set XMLOutputFolder to choose folder with prompt "Select the location for
       exporting the XML files”
set C to 0

tell document 1 of application "InDesign CS”
   set ElementsToExport to every XML element of XML element 1 whose name
        of markup tag = "Article”
   set ElementsToExport to ElementsToExport & (every XML element of XML
        element 1 whose name of markup tag = "ArticleMultiBox”)
   set ElementsToExport to ElementsToExport & (every XML element of every
        XML element of XML element 1 whose name of markup tag = "Article”)

   repeat with theElementToExport in ElementsToExport
     set C to C + 1
     set C3 to (characters -3 thru -1 of ("000” & C)) as string
     select theElementToExport
     set export from selected of XML export preferences to true
     export theElementToExport format XML to ((XMLOutputFolder as string)
       & "Article” & C3 & ".xml”) without showing options
   end repeat
end tell
```

Appendix D2 - JavaScript for Exporting Multiple XML Files

```
//DESCRIPTION: Export XML

// Add method to test is an object is a member of the specified array
Object.prototype.isInArray = function(myArray){
   for (var i=0; myArray.length > i; i++) {
      if(myArray[i] == this){
         return true;
      }
   }
   return false;
}

XMLOutputFolder = Folder.selectDialog("Select the location for ex-
         porting the XML files"); // Global variable
C = 0; //Global variable
myDoc = app.activeDocument;
findEm(myDoc.xmlElements[0]);
errorExit(String(C) + " XML articles exported");

// +++++++ Functions Start Here ++++++++++++++++++++++

function findEm(theElement) {
   // Recursive function that searches whole XML element tree
   if (theElement.markupTag.name.isInArray(["Article","ArticleMultiBox"
         ])) {
      exportIt(theElement);
   }
   var theElements = theElement.xmlElements;
   var myElen = theElements.length;
   for (var i = 0; myElen > i; i++ ) {
      findEm(theElements[i]); // Recursive call to troll lower-level
         elements
   }
 }

function exportIt(myElement) {
   // This code is just about a straight translation of what was in
         the AppleScript repeat loop.
   C++; // Increment global variable C
   C3 = ("000" + String(C));
   C3 = C3.substring(C3.length - 3);
   myFile = File(XMLOutputFolder.fsName + "/Article" + C3 + ".xml");
   app.xmlExportPreferences.exportFromSelected = true; // This could
         be done once only at top level
   mySel = myElement.select();
   myElement.exportFile("XML",myFile,false);
}

function errorExit(message) {
   if (app.version != 3) { beep() } // CS2 includes beep() function.
   if (arguments.length > 0) {
      alert(message);
   }
   exit(); // CS exits with a beep; CS2 exits silently.
}
```

Author Scott Dunn is a founding partner at Flux Consulting • *www.fluxconsulting.com.*

JavaScripts translated from AppleScript by Dave Saunders of PDS Associates • *www.pdsassoc.com.*

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

Adobe, the Adobe logo, Acrobat, GoLive, InCopy, and InDesign are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. Mac is a trademarks of Apple Computer, Inc., registered in the United States and other countries. Windows is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries. All other trademarks are the property of their respective owners.

© 2005 Adobe Systems Incorporated. All rights reserved.
Printed in the USA. 7/05