XML indexing

Michele Combs

The author introduces XML indexing, and explains some of the techniques used in preparing XML indexes.

Increasingly, publishers are creating and editing their documents in XML to make document production simpler and more consistent. Surprisingly, a search of the ASI's Indexer Locator finds only 14 who list XML among their indexing services. Knowing how to do XML indexing can expand your customer base and give you a skill set that is intensely relevant to today's electronic book production environment. We will start with some definitions, walk through what is involved, and then go over some special considerations related to peculiarities of XML.

What is XML?

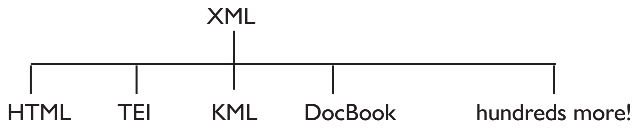
XML stands for eXtended Markup Language. An XML document is a plain text[1] file which has been 'marked up' using a set of elements (also called tags) to show its structure – where paragraphs begin and end, where chapters begin and end, where a list of items begins and ends, and so on. The most popularly known example is HyperText Markup Language, or HTML. For example, here is a piece of HTML with the markup highlighted in grey:

```
<p>This paragraph has a <a href="http://
cnn.com">link to CNN</a> in it.</p>
```

Note that all the elements come in pairs: a start tag to mark where a structural element begins (`<p>`) and an end tag to mark where it ends (`</p>`). Note also that the linking element `<a>` has an attribute called `href`, which is used to specify the target of the link; in this case, the attribute is given the value '`http://cnn.com`'. Elements and attributes are the basic building blocks of any XML document.

The purpose of applying markup is to separate a document's content from its structure; this makes all sorts of things possible. For example, one can generate a table of contents by finding and extracting all chapter titles, or search for a word occurring only in a specific tag, or output two differently formatted versions of a document from a single source file. Think about all the web pages you have seen: they look wildly different, but they are all encoded in HTML; because they all have the same underlying structure and are marked up with the same tags – `<p>`, `<h1>`, `<a>`, `<img>`, and so on – web browsers have no problem reading them all, regardless of how they may look to the human eye. (The differences in appearances are controlled by 'style sheets,' a subject beyond the scope of this article.)

XML is merely a standard for how to define a set of tags; it does not itself define or contain any particular set of tags. So when you sit down to mark up, or encode, a document, you have to decide what tags to use. While it is possible to simply invent your own tags – you might decide to use `<chap>` for chapters, or `<pg>` for paragraphs of text – in most cases it is better (and easier) to choose an existing set of tags suitable for the type of document you want to encode. A *document type definition (DTD)* or *schema* defines a specific set of tags and attributes, along with the rules for how they can be used. There are literally hundreds of XML DTDs and schemas out there for all sorts of purposes. Web pages use HTML. Google Earth stores its data using KML. Books and other narrative documents are often encoded using TEI or DocBook. There are XML DTDs for geospatial data, medical data, financial data — there is even one for comic books called CBML!



Each of these DTDs defines a different set of tags, because they are designed for different kinds of data. HTML, for example, has no tag for marking where chapters begin and end, because web pages simply are not that long and do not have that kind of structure. TEI and DocBook, on the other hand, were designed for encoding long narrative texts, so they both provide a method of marking chapters: DocBook has a `<chapter>` element while TEI uses `<div type="chapter">`.

The first line of an XML file consists of the *doctype declaration*, stating what DTD was used to create the file. A file is said to validate against a DTD if it complies with all of the tag names and usage rules laid out in the DTD.

What is XML indexing?

XML indexing is a form of embedded indexing in which tags are inserted into an XML document to mark the occurrences of indexable terms or topics. The client's publishing process automatically generates an index from these embedded index elements. Fortunately, because this automated process handles all layout and formatting (font style and size, run-in or indented, word-by-word or letter-by-letter alphabetization, etc.), it is not necessary (or in fact possible) to concern yourself with these issues.

When a client requests a quote for XML indexing, your first question should be 'What DTD are you using?' so you know whether you will be working with a familiar tag set or a totally new one. The most commonly used DTD in publishing is DocBook, so for the remainder of this article I will use those tags in my examples. If your publisher is using a different DTD the tag names will be different but the principles will be the same.[2]

What makes it work?

Index entries in DocBook are encoded using the `<indexterm>` element and its five child elements `<primary>`, `<secondary>`, `<tertiary>`, `<see>` and `<see also>`. These elements and their attributes provide for all standard index features, including main entries, subentries, alternate alphabetization order, single page locators and page ranges, and *see* and *see also* references. They are summarized below.

`<indexterm>` element: wrapper element for an index entry of any type

- `class` attribute: set to 'startofrange' to mark the beginning of a range; set to 'endofrange' to mark the end of a range
- `id` attribute: if class is set to startofrange, the id attribute must be set
- `startref` attribute: if class is set to endofrange, the startref attribute must be set
- All other attributes are not used.

`<primary>` element: use for main entry

- `sortas` attribute: set to desired string to force a term to sort other than as written; for example, `<primary sortas="NET">.NET</primary>`
- All other attributes are not used.

`<secondary>` element: use for subentry

- `sortas` attribute: set to desired string to force a term to sort other than as written; for example, `<secondary sortas="women">of women</secondary>`
- All other attributes are not used.

`<tertiary>` element: use for sub-subentry

- `sortas` attribute: set to desired string to force a term to sort other than as written; for example, `<tertiary sortas="children">in children</tertiary>`
- All other attributes are not used.

`<see>` element: used with `<primary>` or `<secondary>` for '*see*' references

- No attributes are used.

`<seealso>` element: used with `<primary>` or `<secondary>` for '*see also*' references

- No attributes are used.

The following examples show how these indexing elements are used to create the basic index structures.[3] Your client should also be able to provide samples of indexed files on request.

Main headings consists of an `<indexterm>` and a `<primary>`:

```
<indexterm>
<primary>women</primary>
</indexterm>
```

Subheadings consist of an `<indexterm>`, a `<primary>`, and a `<secondary>`:

```
<indexterm>
<primary>women</primary>
<secondary>voting rights</primary>
</indexterm>
```

Sub-subheadings consist of an `<indexterm>`, a `<primary>`, a `<secondary>`, and a `<tertiary>`*:*

```
<indexterm>
<primary>women</primary>
<secondary>voting rights</primary>
<tertiary>Saudi Arabia</tertiary>
</indexterm>
```

Forced sorting: in cases where the first letters or words in a term should be ignored when sorting, you can use the `sortas` attribute to specify how a term should be sorted:

```
<indexterm>
<primary>mushrooms</primary>
<secondary sortas="medicinal">for
medicinal purposes</primary>
</indexterm>
```

If the discussion of a topic is brief, only one `<indexterm>` element is needed:

```
<indexterm>
<primary>mushrooms</primary>
</indexterm>
```

A longer discussion that spans several paragraphs requires two `<indexterm>` elements, one to mark where the discussion begins and the second to mark where it ends. The first has its class attribute set to `startofrange` and the second has its class attribute set to `endofrange`.

The first must also have its id attribute set to some string (more on IDs below) and the second must have its startref attribute set to the identical string. Note that the second `<indexterm>` element is empty; it is there only to show where the discussion of the topic ends. For example:

```
<indexterm class="startofrange"
id="ch0406"><primary>potatoes</primary>
</indexterm>
. . . [loooong discussion of potatoes
over several paragraphs] . . .
<indexterm class="endofrange" startref=
"ch0406"></indexterm>
```

Since the above entries will have page numbers associated with them in the final book, they need to be placed in the text exactly where the indexable item occurs, or where it begins and ends (more on this in 'Placement of index elements' below). *See* and *see also* references, however, do not have associated page numbers and so can be placed anywhere in the document.

The following will result in 'fungi, *see* mushrooms':

```
<indexterm>
<primary>fungi</primary>
<see>mushrooms</see>
</indexterm>
```

The following will result in ‘mushrooms, *see also* toadstools’:

```
<indexterm>
<primary>mushrooms</primary>
<seealso>toadstools</seealso>
</indexterm>
```

Examples in context

Figure 1 shows a small section of a DocBook XML file with index elements inserted. Index-specific elements are highlighted in grey. Other elements shown, such as `<xref>` and `<fig>`, are unrelated to the indexing process.
The above would result in something like this when the files are eventually processed and page numbers generated:

basidiospores, 8
. . .
mushrooms
identifying, 8-9
. . .
spore prints, 8, 9

How to get these tags into the XML document

There are two approaches to indexing an XML book. The first is to work directly in the XML files, creating and inserting index elements as you work your way through the

```
<section>
  <title>Identification<indexterm class="startofrange" id="ch02 003">
<primary>mushrooms</primary><secondary>identifying</secondary>        </indexterm>
</title>
  <para>Identifying mushrooms requires a basic understanding of their macroscopic
structure. Most are Basidiomycetes and gilled. Their spores, called <indexterm
id="ch02004"><primary>basidiospores</primary></indexterm><firstterm>basidiospore
s</firstterm>, are produced on the gills and fall in a fine rain of powder from
under the caps as a result. At the microscopic level the basidiospores are shot
off basidia and then fall between the gills in the dead air space. As a result,
for most mushrooms, if the cap is cut off and placed gill-side-down overnight, a
powdery impression reflecting the shape of the gills (or pores, or spines, etc.) is
formed (when the fruit body is sporulating). The color of the powdery print, called
a <indexterm id="ch02005"><primary>spore prints</primary></indexterm>spore print, is
used to help classify mushrooms and can help to identify them. Spore print colors,
as shown in <xref linkend="fig _ sporeprint"/>, include white (most common), brown,
black, purple-brown, pink, yellow, and cream, but almost never blue, green, or red.
</ para>
  <figure d="fig _ sporeprint">  <title>Spore print colors<indexterm  id = "ch020
05f"><primary>spore prints</primary></ indexterm></title> <media   object>   <image
object> <imagedata fileref ="figures/sporeprints.png"/> </image object> </mediaobject>
</figure>
  <para>While modern identification of mushrooms is quickly becoming molecular,
the standard methods for identification are still used by most and have developed
into a fine art harking back to medieval times and the Victorian era, combined with
microscopic examination. The presence of juices upon breaking, bruising reactions,
odors, tastes, shades of color, habitat, habit, and season are all considered by
both amateur and professional mycologists. Tasting and smelling mushrooms carries
its own hazards because of poisons and allergens. Chemical tests are also used for
some genera.</para>
  <para>In general, identification to genus can often be accomplished in the field
using a local mushroom guide. Identification to species, however, requires more
effort; one must remember that a mushroom develops from a button stage into a mature
structure, and only the latter can provide certain characteristics needed for the
identification of the species. However, over-mature specimens lose features and
cease producing spores. Many novices have mistaken humid water marks on paper for
white spore prints, or discolored paper from oozing liquids on lamella edges for
colored  <indexterm  id="ch02006"><primary>spore  prints</primary></indexterm>spore
prints.<indexterm class="endofrange" startref="ch02003"></indexterm></para>
</section>
```

Figure 1 Section of a DocBook XML file with index elements

book; the other is to work from a PDF or paper copy of the final book, create the index the traditional way using your preferred indexing software, and then at the end transfer the index entries into the XML files.

Working directly in XML

Because XML files are plain text, they can be edited using any plain text editor such as NotePad++, NoteTab, TextEdit or emacs.[4] However, XML indexing in plain text can be laborious and time-consuming since XML is harder to read than regular text. In addition, most plain text editors will not validate the document, nor will they prevent you from introducing encoding errors such as putting a tag in an invalid location or forgetting an end tag. Specialized XML editing software can make all this much easier. A full discussion of XML editing applications is beyond the scope of this paper, but they include XMetaL, oXygen, XMLSpy and XMLMind, and most come in Windows and Mac flavors. A very good plugin for Notepad++ called 'XML Tools' is also available (www.sourceforge.net/projects/npp-plugins/files). The difference between plain text and XML authoring software can be seen in Figures 2 and 3.

Note that in the XML authoring software each element is treated visually as a unified item rather than as just typed strings of characters, and elements can be inserted with the click of a button. In Figure 3, an <indexterm> element (highlighted in the text and in the 'Insert element' menu) has just been inserted with the click of a button. In addition, the software can validate the file and remind you if you have left out required elements. Most XML authoring software allows you to create macros that greatly speed up the editing process. For example, you could create a macro that copies whatever text is currently selected, backspaces, inserts <indexterm> and <primary>, and pastes the copied text into the <primary>, thus creating a correctly tagged index entry with a single keystroke.

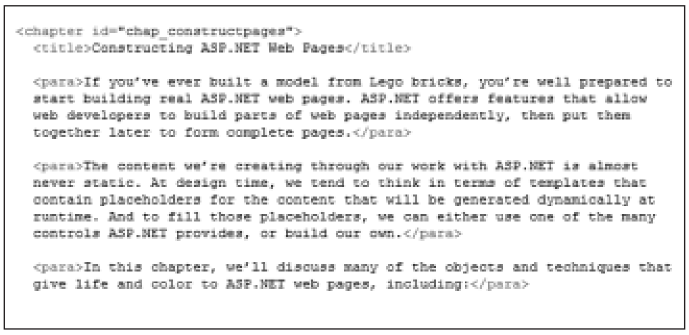
At this point you might be asking, 'But if the index entries are scattered across one or more XML files, how will I proof it?' This is one of the more difficult challenges of creating the index directly in the XML files. There are open-source style sheets that will render DocBook XML into HTML, complete with the index (see for example www.sourceforge.net/ projects/docbook/files/epub3; although it's called EPUB3 it also includes style sheets for HTML5). Setting these up on your own requires some knowledge of XML transformations and can be challenging for the beginner, but this is another area where specialized XML authoring software can be a great help: oXygen for example can apply a style sheet to an XML file and render it into HTML with the click of a button. Some indexers, myself among them, have written custom scripts to periodically generate the book and the index, but this requires additional knowledge of XSL and is not really an option for a beginner.

The traditional approach, plus copy/paste

In addition to the problem of proofing, neither a text editor nor XML authoring software provides the same functionality as dedicated indexing software like Sky or Cindex. Cross-reference checking, filtering, flipping, grouping, autocomplete, and so on are unavailable, and it is difficult if not impossible to proof an index when the <indexterm>s are scattered across one or more XML files. For this reason, some XML indexers choose to work from a PDF or print version of the book (the client can probably provide this on request) and create their index the traditional way, using dedicated indexing software. When the index is finished and has been proofed and corrected, the entries must be transferred to their corresponding locations in the XML file. This solves the problem of proofing as well as allowing the indexer to take advantage of the error-checking features mentioned above.

Transferring entries to the XML file may be done by creating the desired elements (e.g. <indexterm><primary></primary></indexterm>) in XML and then copying in the text of the index entry. Also, both CIndex and Sky offer 'tagged text' output. Although at the time of writing, neither has sufficient flexibility to output fully tagged DocBook <indexterm> structures, you could use this to output most of the tagging and then use wildcard search-and-replace in Word to add the remainder. Then you would simply copy each <indexterm> structure into its appropriate location in the XML file. In either case, it is also necessary to manually add id and class attributes where necessary, set any necessary sortas attributes, and manually insert <indexentry class="endofrange">s. Again, this editing of the XML can be done in a plain text editor or using XML authoring software, as discussed above.

There is a handy tool called DEXembed (www.editorium.com/dexembed.htm) which automates many of these steps if you are indexing in DocBook. DEXembed employs Micro-



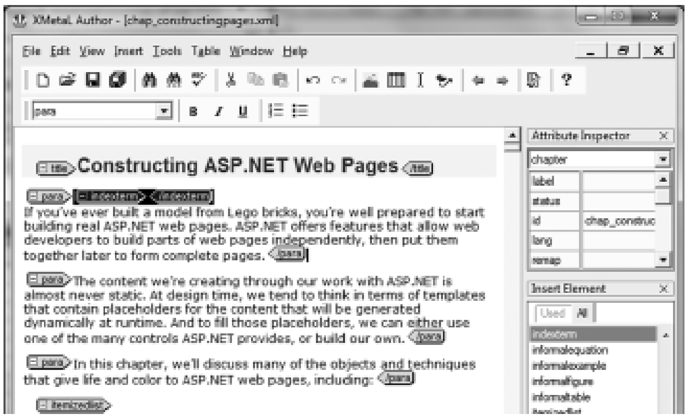
```
<chapter id="chap_constructpages">
  <title>Constructing ASP.NET Web Pages</title>

  <para>If you've ever built a model from Lego bricks, you're well prepared to
  start building real ASP.NET web pages. ASP.NET offers features that allow
  web developers to build parts of web pages independently, then put them
  together later to form complete pages.</para>

  <para>The content we're creating through our work with ASP.NET is almost
  never static. At design time, we tend to think in terms of templates that
  contain placeholders for the content that will be generated dynamically at
  runtime. And to fill those placeholders, we can either use one of the many
  controls ASP.NET provides, or build our own.</para>

  <para>In this chapter, we'll discuss many of the objects and techniques that
  give life and color to ASP.NET web pages, including:</para>
```

Figure 2 Plain text editor



Constructing ASP.NET Web Pages

If you've ever built a model from Lego bricks, you're well prepared to start building real ASP.NET web pages. ASP.NET offers features that allow web developers to build parts of web pages independently, then put them together later to form complete pages.

The content we're creating through our work with ASP.NET is almost never static. At design time, we tend to think in terms of templates that contain placeholders for the content that will be generated dynamically at runtime. And to fill those placeholders, we can either use one of the many controls ASP.NET provides, or build our own.

In this chapter, we'll discuss many of the objects and techniques that give life and color to ASP.NET web pages, including:

Figure 3 XML authoring software

soft Word in conjunction with some customized scripting to 1) temporarily add paragraph numbers to a DocBook XML file, which you then use as locators in creating your index, and then 2) embed the finished index entries into the XML file, complete with correct XML encoding. There are a number of steps involved that must be done with care: see Section 4.13 of the DEXembed user guide for more information. DEXembed does assume that you have some sort of XML authoring software for part of the process.

Validating the XML files

XML files delivered to your client must validate against their specified DTD, without exception. XML authoring software can be a great help here, since validation is done automatically each time you save a file, and can also be done as desired with the click of a button. In addition, the software will prevent you from introducing most encoding errors as you work. If you edit the files in a plain text editor, you will have to take the extra step of validating the files yourself. The easiest way to do this is to use the World Wide Web Consortium's (W3C) online validator at `www.validator.w3.org`. You can use either the 'Validate by File Upload' option or the 'Validate by Direct Input' method.

Special considerations

ID attributes

Technically, the `id` attribute is necessary only for `<indexterm>`s with `class="startofrange"` but it is a good habit to give all `<indexterm>`s an ID. The ID can be generated using any scheme you wish as long as each one is unique across the entire book. Values for the `id` attribute must consist only of alphanumeric characters, dashes or underscores, and must begin with an alphabetic character. Examples of valid IDs are 'ch02-004', 'c2632' and 'fig_32'. Examples of invalid IDs are '624f' (starts with a number), 'ch5 003' (contains a space) and '65&f3' (the ampersand is not acceptable).

Placement of index elements

Index entries and their associated elements are always added to the XML document. The `<indexterm>` element should never be wrapped around text already in the document since, in the final product, `<indexterm>` and its child elements are suppressed in the text flow and only appear in the index. The `<indexterm>` element should always be placed as closely as possible to the occurrence of the term or concept in the text, according to the following guidelines.

When indexing a single occurrence of a word, phrase, name or concept, the `<indexterm>` element should be placed just before the occurrence in the text.

When indexing a longer discussion of a word, phrase, name or concept, you must decide whether to use a single `<indexterm>`, which will always generate a single page locator (e.g. 25), or two `<indexterm>`s, which may or may not generate a page range (e.g. 25 or 25–26), depending on how page breaks fall in the final book. (Fortunately, you don't need to worry about which it will be since the automated production process will take care of that.) Since an XML document has no physical pages, determining whether to use a single locator is less straightforward than when indexing from paper. In general, use the following rules of thumb:

- If discussion of a topic encompasses a single full paragraph, use a single `<indexterm>` element placed just inside the opening para tag.
- If discussion of a topic encompasses portions of consecutive paragraphs – for example, the discussion begins halfway through paragraph 2, encompasses all of paragraph 3, and ends halfway through paragraph 4 — place the `<indexterm class="startofrange">` in the text just before the start of the discussion in paragraph 2 and the matching `<indexterm class="endofrange">` in the text just after the end of the discussion in paragraph 4.
- If discussion of a topic encompasses several entire paragraphs, place the `<indexterm class ="startofrange">` just inside the opening para tag of the first associated paragraph and the matching `<indexterm class="endofrange">` just inside the closing para tag of the last associated paragraph. The `<indexterm>`s should not be placed outside the first or last para element, since when the automated processing is performed later, it will be unclear whether the indexterm is associated with the para that it precedes or the one it follows.
- If discussion of a topic encompasses more than one paragraph and begins at a new chapter, section, or subsection, place the `<indexterm class="startofrange">` just inside the title element for the chapter, section, or subsection.

These are guidelines only. Your publisher may have different requirements. Note that if you use DexEmbed, it might or might not place the index entries where you want them, so you might need to move them slightly. This is another question you will want to ask your client up front.

Figures and tables

You will want to ask your client if figures and tables need to be indexed specifically regardless of where they appear in the file, since it is impossible to tell from the XML where a figure or table will fall in the final print edition. A figure might appear in print exactly where it appears in the XML, or it might 'float' to a later page. Place the `<indexterm>` element just inside the figures or table's `<title>` element. When preparing traditional print indexes, figure and table locators are sometimes followed by 'f' or 't' or similar, but nothing like that is necessary in an XML index. If that is desired, it will be automatically added by the production process.

Special characters and symbols

Clients in need of XML indexing are often publishers of technical and computer books, so you might encounter symbols or special characters. Depending on what software you are using to edit the XML, the character in question might appear correctly rendered, or it might appear as a *character entity*, an alphanumeric string. For example, the Greek omega (Ω) is represented as `&omega;`. In addition, the ampersand has a special meaning in XML, so an ampersand in the text is always represented as `&amp;`. Regardless of how a special character looks in your editor, always use whatever is given in the XML file for your index term; simply copy the string in question and paste it inside the `<primary>` or `<secondary>` element:

```
<p><indexterm><primary>omega (&omega;)
</primary></indexterm> <indexterm><primary>
&omega; (omega)</primary></indexterm>The
Greek letter &omega; is used to repre-
sent...</p>

<p><indexterm><primary>Pratt &amp;
Whitney</primary></indexterm>The
company's name is Pratt &amp;Whitney.
</p>
```

White space

Avoid introducing white space or line breaks when adding your index elements. In most cases extra white space is ignored during processing, but there is a small chance that it will affect page breaks in the final output.

Conclusion

While the preceding is probably not sufficient to launch you on a career in XML indexing, it should be enough to get you started experimenting. XML authoring software can be had for less than $100, and there is a great deal of information about DocBook available online. XML indexing is not for everyone, but be bold and try your hand at it – you might find that it clicks with you!

Acknowledgement

This is a slightly edited version of an article that first appeared in *Key Words*, vol. 19, no. 4. It is reprinted with kind permission of the editor.

Notes

1. 'Plain text' means that the document contains no formatting: no font face or size information, no underlining, no bold, nothing but the plain text. A document created in NotePad, for example, is plain text.
2. See www.docbook.org/ for complete information on DocBook.
3. An excellent and thorough discussion of the five indexing elements is available on the DocBook website at www.docbook.org/tdg5/en/html/pt02.html. The full text of *DocBook 5: the definitive guide* is also available online at www.docbook.org/tdg5/en/html.
4. XML files should not be edited or resaved with more sophisticated word processing software such as Microsoft Word, as this is likely to introduce extraneous data which will render the XML files invalid.

Michele Combs has almost 20 years of experience with XML, XSL, and data conversion. She is presently lead archivist at Syracuse University's Special Collections Research Center, where she works with XML on a daily basis. She is also a freelance indexer, editor, and writer. Email: mrrothen@syr.edu

Reviews: breaking the page

Breaking the page: transforming books and the reading experience. Peter Meyers. OReilly Media: 5 December 2011 (preview edition). ASIN: B006IIXBO0. A. Kindle edition.

Freelance journalist, electronic textbook publisher, and editor Peter Meyers calls himself a 'digital book architect' and this is his latest O'Reilly's Tools of Change for Publishing Conference ebook project. This review is of the three-chapter preview edition available free at the O'Reilly website and at Amazon.com.

What makes this book particularly prescient is that Meyers is actually writing about book navigation, electronically and in print, and rather than focusing on the technology per se he articulates the problems associated with the technological transition from print to digital. He sees the issue as being a conceptual one, compounded in many cases by a lack of imaginative design. As an example of the difficulties in envisioning the true potentials of any new invention, he mentions the Horsey Horseless Automobile, in which the head and neck affixed to the front of a 19th-century automobile was meant to convey the ease and familiarity of the horse-drawn carriage upon the newly-invented car. This clumsy and useless addition was later dropped in favor of improvements to the car as an automobile, having its own standards and features as a means of transportation apart from the horse.

Meyers does not use the cinema as a similar example, but it should be remembered that silent film, too, went through a period of being yoked unnecessarily to both books and to the proscenium stage, until the scene cuts, dissolves, montages, and the organic use of metadata (think of the difference between the use of opening titles in D.W. Griffith's *The birth of a nation* and Ridley Scott's *Alien*) allowed the cinema to evolve its own language and gain respectability as an art form in its own right.

So it is with ebooks: those that merely strive to recreate the printed page are hobbled by unintended inconveniences – their design detracts from, rather than adds to, the promised experience, causing frustration and confusion. Before one can design an ebook that works well for the reader, one must remember that printed books were also a technology, and not one meant to recreate the scroll or the clay tablet. Printed books are not more 'natural' than are ebooks, but have their own arbitrary and cumbersome navigational elements due to their limitations, which certainly do not need to be migrated to the ebook. Meyers successfully makes the case that ebooks should not merely replicate in an electronic format the particular features of the printed book that are, like the horse bust affixed to a car, vestigial organs, but that designers should give the ebook the ability to deliver a seamless reading experience.

Specifically, Meyers argues that users are not content to scroll

continued on page 64