
Using XML in a Multi-Format Publishing Environment

XML has been developed primarily as a means of information storage and interchange. However, it can also be used very effectively for real world publishing. It is particularly suitable as a means of maintaining material which is to be published in a variety of output formats (such as print, CD, Web).

This paper illustrates techniques which can be used to simplify markup design and document maintenance, and facilitate the further processing of the stored information, both for publishing and for utilities such as table and index extraction.

Introduction

Turn-Key has been developing publishing software for over 25 years, so we look to XML primarily as a means of document markup in real world production environments. Because of this, our approach may be viewed as novel by some, unorthodox by others. Nevertheless, we have found that the techniques described below are highly effective in producing flexible and useful markup with very low design and implementation overheads.

The techniques are grouped together into sections covering particular areas. Those which simply clarify our design philosophy are called *tips*, while those which run contrary to classical SGML wisdom are known as *heresies*.

Tip 1: Develop a Consistent Markup Design Philosophy

All our DTDs follow a few simple rules:

- tags fall into one of three classes: level, content, or effect;
- every document consists of a series of nested levels;
- every level is introduced by a unique tag;
- every level has the same basic structure;
- content of the same general type is always introduced by the same tag, regardless of level or output format;
- similar structures in different publications are always tagged in the same manner;
- tagging is kept as simple and general as possible, consistent with required functionality.

When the above rules are followed, the design of a new DTD is greatly simplified. This is partly because the general structure is already mapped out, but also because it is possible to reuse element models from other publications.

Levels

Level tags range from `<document>` and `<chapter>` through `<section>` down to `<item4>`, which is a fourth level indented list item. Naturally, the choice of level tags depends on the nature of

the publication, but this scheme can apply equally well to books, catalogs, printed tables and indexes etc. In SGML-speak, a level tag introduces element context, and never contains text directly.

Each level has the metamodel

(label?, desc?, OTHER*, CONTENT)

where:

- `<label>` serves to order the level amongst others of its type. A chapter or section number is a label, as is the **(a)** or **(vii)** of a list item.
- `<desc>` is the description of the level, such as the text of a heading. However, even list items can have `<desc>`s, which often manifest as a brief segment of bold or italic text at the beginning of the item.
- **OTHER** refers to any additional tags carrying information relating to the level as a whole. In printed works, these might include a telltale to be printed above the text on each page.
- **CONTENT** is typically either **SUBLEV+** or **p+**. That is, the content of a level is either a group of sublevels, or some real textual material (which might itself contain sublevels).

Two typical level definitions might be as follows:

- `<!ELEMENT chapter (label, desc, telltale?, intro, section+) >`
- `<!ELEMENT item1 (label?, desc?, p+) >`

Content and Effects

There are three main content elements: `<label>`, `<desc>` and `<p>`. The last introduces a general content unit at the current level. Its name comes from the fact that it typically (though not universally) manifests as a paragraph in print. We reserve the `<para>` tag for those documents which actually contain a paragraph level. Content tags represent a change from element context to mixed context.

There are a number of other possible content tags. The `<telltale>` mentioned above is one. We also use the `<cell>` tag within tables in this manner. However, the `<p>` tag is preferred except where some specific functionality is implied.

The final class of tags are the effects. These can range from the trivial (such as `<sub>` for subscripts), to the highly sophisticated (such as `<quote>` which can change fonts, infer quote marks of varying types, and invoke hyperlinks in electronic products).

Heresy 1: The DTD is Used to Support the Markup, Not to Enforce It!

The DTD has been SGML's great claim to fame. It is presented as a means of ensuring the structural integrity of the document. Unfortunately, this was never quite true and is even less applicable with the rise of XML.

HTML has shown us that a strict DTD is not necessary to provide a functional markup, while XML has the option of doing away with the DTD altogether. In fact we are not advocating the abolition of the DTD just yet, though that time may come.

We use the DTD to ensure that valid tags are used in a reasonable fashion, but that does not imply that a document is correct. In particular the generic content tag `<p>` can contain a host of sublevels, as can a `<quote>`. It is quite valid to place an `<item1>` within a `<p>` within an `<item3>`, but

this is generally an example of bad markup. It may be fairly stated that the price we pay for a simple, flexible markup regime is a rather loose DTD specification.

This is caused by two main weaknesses in DTD functionality. Firstly, there is no facility for context sensitive content models. Thus, I can not say that a `<p>` can only contain an `<item3>` when the `<p>` is itself within an `<item2>`. One can of course define `<item1.p>`, `<item2.p>` etc, but this lowers the level of tag abstraction, simplicity, and reusability.

Secondly, the DTD cannot validate element content, for example by ensuring that text inside a `<date>..</date>` pair is actually a valid date.

At Turn-Key we have been considering using XSL for this purpose. The draft specification for XSL suggests the possibility of designing a validating stylesheet which can supplement (or even replace) the DTD, by testing the document instance against a number of context (and content) sensitive rules. Such issues are however outside the scope of this presentation.

Tip 2: Handy Hints for Generalized Markup

The most important rule in marking for multiple output formats is that the markup must be sufficient to drive all of the required conversions without the need for further human intervention. If this rule is not followed, there is a recurrent cost every time the markup is processed, which almost always proves unsustainable in the long run.

The following are a few guidelines to follow in your markup. They may seem simple enough, but we hardly ever see a document style where at least one of these rules is not broken.

- Pitch your markup at the highest reasonable level of abstraction. Our use of the `<desc>` tag is one example. This tag indicates the name or title of a level, but the output format can vary greatly between levels. Unfortunately, we often see `<chapter.title>`, `<example.title>`, `<section.head>`, and even `<highlight style="ital">` all used for the same concept at different levels in a single document (the last used at the item level). Keeping the number of distinct element names to a minimum makes the markup easier to master, reduces errors, and increases opportunities for reusing text fragments.
- Keep the structure of the various levels consistent. This seems obvious, but is often not implemented in practice. For example, when `<label>` and `<desc>` both occur in a level the `<label>` always comes first, even if the description precedes the label in the output. This allows different output formats to have different styles in this regard. Once again, a consistent structure allows material from one publication to be easily slotted into another, without the need for tag modification.
- Don't introduce unnecessary constructs into your tagging. One common example of this problem occurs when a paragraph contains a list of items. Some DTDs require that the paragraph be closed before the list can be opened, and that a continuation paragraph be started when the list is complete. Aside from introducing superfluous tagging, the structure actually runs counter to the structure of the document, which clearly indicates that the list forms part of the paragraph. A more common technique is to wrap the items in a `<list>..</list>` pair. This is certainly a preferable alternative, but there is rarely a strong case to be made for including the wrapper, as it is extremely rare for two lists to appear together without intervening text.
- Never key material in ALL CAPS. We very often find headings and other major entries keyed in caps because that's the way they appear in the book. Problems arise when you try to use these headings in (say) a Table of Contents. The only text which should ever be keyed in caps is that which can never appear in any other way. For example **Them v Us in the US** is

correctly capitalized. Always remember, to capitalize mixed case is trivial, but to convert the other way is both difficult and unreliable.

- Never key real text as attributes. Anything that is (or may be) printed must be marked as real tagged text. For example, I saw one DTD that had page telltales keyed as attributes since they lay outside the normal page area. This proved adequate until a telltale arose which had an effect (such as a subscript) which could not easily be accommodated within an attribute value.
- Try to key footnotes in situ where possible. That is, the `<footnote>..</footnote>` group should be attached to the word which triggers the footnote. This allows the note to appear at the bottom of the page, the end of the section, or (in electronic output) as a popup or hyperlink.
- Finally, try to keep your audience in mind when designing tag names. If your client's editors refer to lines at the top of the page as running heads, or cues, rather than telltales, then respect that convention in the markup. Yes, material should be reusable across publications, but this does not imply that Client B must use the same tagging conventions as Client A.

Remember that, in general, printed output makes significantly greater demands on the markup than CD or HTML formats. Some of the items to be specified include: continuation lines; telltales and running heads; page number gapping; vertical justification; placement and numbering of footnotes; multi-page tables; rotated text; production of title pages, contents, indexes etc. If you can get your print formatting under control, there should be very little to add to accommodate other output media, since most items to be tagged require (or have the potential for) typesetting effects.

Heresy 2: Too Much Tagging is as Bad as Too Little!

Many practitioners of SGML regard extensive tagging as a sign of good markup. However, each additional tag carries with it a significant cost.

First, each tag has to be placed and maintained, and is a small but continuing drain on resources. If the data is managed by a repository which handles the tagging, this cost is reduced. However, if the data is to be accessed via an editor such as Adept, the tags add bulk and complexity to the screen view. Personnel such as authors and editors operate best when the level of tagging is kept to a minimum.

Second and perhaps even more important is that the smaller the number and variety of tags, the greater the likelihood that text can be freely exchanged between publications. Accordingly, we take pains to eliminate as much tagging, and particularly style related tagging, as possible from our documents. This will be discussed further in the following section.

Tip 3: Inferred Effects

One method used for reducing the tag density and complexity in our documents is to use inferred effects. These are usually stylistic effects, such as font changes. However, it is possible to infer complex structures, such as multiple hyperlinks.

This is probably best illustrated by an example. Let us consider a simple legislative reference, such as:

```
<l.ref>Sections 25 to 27AA of the <legn>Crimes Act  
1975</legn></l.ref>
```

The rule for using these tags is: apply a `<l.ref>` tag to the entire reference, and a `<legn>` to the Act name, including year and jurisdiction (if any).

Now, in order to print this simple reference, two effects need to be inferred. Firstly, our print style dictates that section numbers such as **27AA** should be printed with the **AA** as small caps. Secondly, the act name should be in italic but not the year. We cannot simply exclude the year from the `<legn>` tag, because the year is actually a part of the name and we need to extract it when building our table of Acts.

In addition, we wish to publish this reference on a CD infobase, but with hyperlinks from the section numbers to the actual act sections. These links must of course precisely target the corresponding sections.

To achieve all this using classical markup techniques, we'd need something like:

```
<l.ref>Sections <secref ref="CRIM75/25">25</secref>
to <secref ref="CRIM75/27AA">27<small>AA</small></secref>
of the <legn>Crimes Act <year>1975</year></legn></l.ref>
```

This is perfectly correct XML, but far more difficult to read and to enter accurately.

To infer the required effects in print, we use tag conversion applets. These are currently implemented as plug-ins to our typesetting system. They could equally well be implemented as ECMAScript (JavaScript) routines in an XSLT transform. The first plug-in effectively swallows the entire `<l.ref>..</l.ref>` block and emits a transformed equivalent. In this case the only required effect is to reduce caps after numbers to small caps. The second plug-in handles the `<legn>` block and italicizes the Act name, but not the year.

Because the plug-ins deal with (normally) small quantities of data and simple transformations, they are very easy to write and reliable to run. A similar but somewhat more complex routine is used by the CD builder to infer the appropriate hyperlinks.

Using the plug-ins reduces the required markup by two thirds even in this simple example. The savings in bulk and complexity over a 100K page volume set are substantial. Do the plug-ins ever infer effects incorrectly? Yes, but very rarely. Experience shows us that the error rate caused by the use of a properly designed set of plug-ins is more than offset by the reduction in human error due to the simplified tagging. Of course, when a plug-in goes wrong, the results can be dramatic, but the programmers are usually informed of their mistake rather promptly!

Inferred effects also allow us to further isolate the essential structure of the data from the details of presentation. As previously intimated, one advantage is that the markup becomes more widely applicable, so that material from one publication can be used in another without the need for tag modification. It may of course be necessary to invoke customized plug-ins to produce the precise effects required by a particular output style.

Heresy 3: The PI Can Be Your Friend <? :=) ?>

It is said that the processing instruction is introduced by a question mark as a constant reminder that their use is dubious at best. However, the much maligned PI can prove very valuable if used with discretion.

In particular, PIs can mark an alternate tag hierarchy to that given by the "proper" tags. One example is the marking of page boundaries. This can be valuable for legacy material where references are by page number. It is also useful for looseleaf services, where the location of the leaf boundaries must be accurately marked.

SGML allowed empty marker tags to be defined as inclusions in higher level elements. This option is not available in XML. To add such a marker into the structural hierarchy would be extremely difficult, and would render the DTD almost unreadable.

The fact that the DTD is unable to verify such PIs is largely irrelevant, as the DTD by its very nature cannot be relied upon to strictly enforce even normal tag usage.

The moral here is that you can and should use all the tools that XML provides to maximize the effectiveness of your markup.

Conclusion:

The techniques outlined above have proven highly effective in practice. We have a number of examples of products being converted from various proprietary formats to fully functional XML with a one-off overhead of about 30% on normal production costs and timings. When properly designed and implemented, the XML production environment eliminates most of the problems associated with suspect data and inconsistent tagging. In addition, the new tagging style is generally neater, and easier to learn and maintain. The conversion costs are therefore quickly recovered.

The abstraction of the markup gives a further edge in that whole ranges of products (for example all books and looseleaf services) can be handled by a single DTD. Occasionally specialist services, such as catalogs, require a different structure. However, even then there is a 80-90% commonality in the coding of the two DTDs.

Obviously, the use of highly abstract context sensitive markup depends on typesetting and other conversion software which can handle the required mappings. Fortunately the proposed XSL standard has all the capabilities needed to perform this task.

The lesson here is that, with suitable planning and design, converting even large and complex publishing environments to XML can be both relatively painless and highly profitable.