

FM+SGML Information Design

Author: Dan Emory,
Dan Emory & Associates

Although the FM+SGML Developer's Guide provides all the details needed to produce an Element Definition Document (EDD), it offers little guidance about information design methods. Nor does the Developer's Guide or any other Adobe-provided documentation discuss the complementary roles that the EDD and its templates play in implementing sound information designs. This paper is an attempt to remedy those deficiencies.

This paper (a structured document created with FM+SGML) focuses on the case where there is no existing SGML Document Type Definition (DTD), which provides the greatest opportunity for optimizing information design. However, many of the same techniques can be applied when there is a preexisting DTD. But a preexisting DTD often imposes a severe constraint on the design optimization methods discussed here.

1 Overview

*Note that the subject of this paper is **Information Design**, not document design..*

Marcus Carr defines where we're going

I correspond frequently with Marcus Carr of Allette Systems (Australia), who has broad experience with SGML and XML. In a recent email posted on a discussion list to which we both subscribe, he wrote:

"...but what if the data also needs to be communicated to a database, not just another person? It's unreasonable to expect the database to understand n+ word processors...The only solution is to provide a common target for all applications that produce data, and build middleware to facilitate communication amongst them..."

"As a technical communicator, you surely feel you are already a creator of information, but the goal that you're used to (making people understand) is being supplemented with other potentially more important imperatives. You will soon have to consider not only how to make information clear to a user, but also how to make it clear to whatever other applications may wish to make use of your data..."

"...I'm unashamedly biased—if it was up to me, everyone would produce SGML data and we would dynamically bolt together fragments in response to specific queries..."

The New Paradigm creates new requirements for information Design.

The new paradigm described above will be implemented by object-oriented database repositories which directly store parsed SGML/XML and entities. Queries directed at this database will retrieve documents, or portions thereof, and middleware will dynamically assemble the retrieved components for delivery in the manner prescribed by the human user or the using application. The next five sections describe the elements of FM+SGML information design that I believe are essential to realizing the full potential of the new paradigm. Section 7 describes Extensible Markup Language (XML), which will be used to implement the new paradigm

2 Universality

Ideally, an enterprise would utilize a single EDD/DTD for all of its mission-critical document types: Everything from large multi-chapter manuals to standalone articles, specifications, proposals, training materials, and even memos, could be delivered either as printed documents, or as documents for on-line viewing in a variety of formats to meet different customer and departmental requirements.

HTML has proven the advantages of universality

HTML's huge success, despite its many shortcomings, demonstrates the overarching advantage of universality. The most salient feature of HTML is a simple, flexible structure in which the structural components are *document object types* not *information types*. Contrast this with the typical reductionistic SGML DTD, which prescribes a rigid structure of elements whose names describe information types rather than document objects.

The reductionistic approach to structure and element naming destroys universality

The reductionistic approach is doomed from the outset. Information content has many facets that cannot possibly be described in a single (usually cryptic) element name. This approach usually leads to the following undesirable results:

1. The DTD/EDD is specific to a particular document type, thus its universality is destroyed.
2. The DTD/EDD is volatile because it is susceptible to change every time there is a shift in technology or processes. These shifts often create new information types, and make others obsolete, requiring the addition of new elements and the deletion of others. The resulting volatility invalidates legacy documents prepared to the earlier version of the DTD/EDD.
3. Either the element names are so generic that they fail to supply useful information about the content,

OR

The number of elements expands to describe each possible combination of information facets, making the DTD so unwieldy, reductionistic, volatile, and arcane that no one can use it.



Note: A reductionistic dimension to information design is revived in [Section 5, Extensibility of the Modular Structure](#).

Attributes should define information types

The obvious way out of the reductionistic dead-end is to use attributes, rather than element names, to describe the information content of each element. A set of generic attributes, easily adaptable and extensible for different organizational requirements, can provide the multiple facets needed to adequately describe information content.

Element names should describe document object types

Authors who create documents with word processors and desktop publishing systems share a common way of thinking about document objects (e.g., headings, paragraphs, strings, lists, steps, notes, cautions, warnings, graphics, equations, tables). When an author looks at an element catalog listing all the valid elements that can be inserted at some point in a document structure, s(he)'s not thinking about information content, s(he)'s thinking about document objects. Authors need to know what kind of document object will be created by each element so they can choose the appropriate one.

The best of both possible worlds

Requiring authors to choose among elements whose names describe information types rather than document objects is like telling a traveller he must first reach his destination before he is permitted to choose his mode of travel. It's counterintuitive and counterproductive.

Using document objects for element names and attributes to describe information content provides the best of both possible worlds. Major benefits of this approach include:

Benefits Derived from a Universal DTD/EDD

- **Information Reusability:** Since all document types utilize the same DTD/EDD, information "packets" extracted from any document can be reused in any other structured document, simply by copying and pasting them.
- **Reduced Training Costs**
- **Reduced Operating Costs**
- **Less Volatility:** Since the set of document object types is relatively stable, DTD/EDD volatility can be greatly reduced when element names describe document objects.

3 Structural Enrichment

A high-end DTP for unstructured documents resembles a Lego set, where the immutable set of building blocks is a relatively small group of basic document object types. In FrameMaker, a template, consisting of a set of catalogs, adds value by defining all of the different document object subtypes needed by authors. Using these catalogs, authors can string pre-formatted document objects together in any sequence or combination allowed by the DTP.

Fundamentally, the purpose of a DTD/EDD should be to impose order and consistency on the document structure, without limiting the range of options needed by authors to present complex information. We need an orderly way to *enrich the structure when required*.

An example of structural enrichment

Take for example, a list of numbered steps for a procedure:

Step 1. Some steps may require multiple levels of autonumbered substeps:

- a. Substep a
 1. First sub-substep.
 2. A second sub-substep.
- b. Substep b.

Step 2. A step may have multiple paragraphs, where only the first paragraph is numbered, as demonstrated below:

The succeeding paragraphs are indented to the same level as the text in the numbered paragraph so as to make clear it is part of the step.

Step 3. In some cases, a step needs to be preceded or followed by a note, caution, or warning which is an integral part of the step:



Note: This is a note that precedes (and pertains to) Step 4. The `Alert` container element prescribes the note's structure and format. This note is actually part of the `Step` element containing the text of Step 4.

Step 4. In other cases, a step may include a graphic, table, or other object which is inserted below the step's text, as demonstrated below:

By the Way...

This is a 1-row, 1-column table containing special instructions or explanations pertaining to a procedural step.

Methods for achieving structural enrichment

The DTD/EDD structure rules for the example shown above should allow all of the indicated possibilities, and more. There are two methods for accomplishing this:

1. List all of the possible object types shown above as inclusions in the `Steps` container (a numbered list), whose general rule is:

`Step, Step+`

This is the easiest solution, but it is also the least desirable one, because it clutters up the element catalog with those inclusions, which is confusing to authors because the inclusions will be indicated as valid anywhere in the `Steps` container, even though they would all be invalid everywhere except at the end of a `Step` paragraph.

2. Include in the structure rule for each `Step` element one or more optional "**structure enricher**" container elements. Let's call one of them `#Body`, where the `#` prefix indicates that it is optional, and the other `Alert`, which creates a note, caution, or warning before the text of the `Step` element. The `#Body` element is an "attachment spine" for miscellaneous document objects. The General Rule for the `Step` element specifies that the `#Body` container can optionally be inserted at the end of the text in any numbered `Step` container element, and that an `Alert` container element can be optionally inserted before the text of the step. That is, the general rule for element `Step` is:

`Alert?, <TEXT>, #Body?`

and the general rule for the `#Body` element might specify:

`(Figure | Next_Level | Table | Equation | More_Text) *`

Where:

- `Figure` produces a captioned or uncaptioned graphic
- `Next_Level` produces substeps under a numbered step, or sub-substeps under a substep.
- `Table` produces a table of any selected type
- `Equation` produces a captioned or uncaptioned equation
- `More_Text` produces additional unnumbered text paragraphs under a numbered step or substep paragraph.

Unlike inclusions, the element catalog will indicate that the #Body container is valid only when the text cursor is placed at the end of a Step paragraph, and that the Alert container is valid only before the text.

If the author needs one or more of the object types provided by #Body, its insertion at the end of a Step paragraph causes the element catalog to list all of the object types included in the #Body container's general rule.

This method has the added advantage of "encapsulating" all of those added objects as part of the Step element to which they pertain. Consequently, when such a Step element is moved or reused, *all of its attached objects go with it.*

[Figure 1](#) shows the structure view of the implementation described in item 2 above, as it would be applied to the four-step example presented earlier in this section.

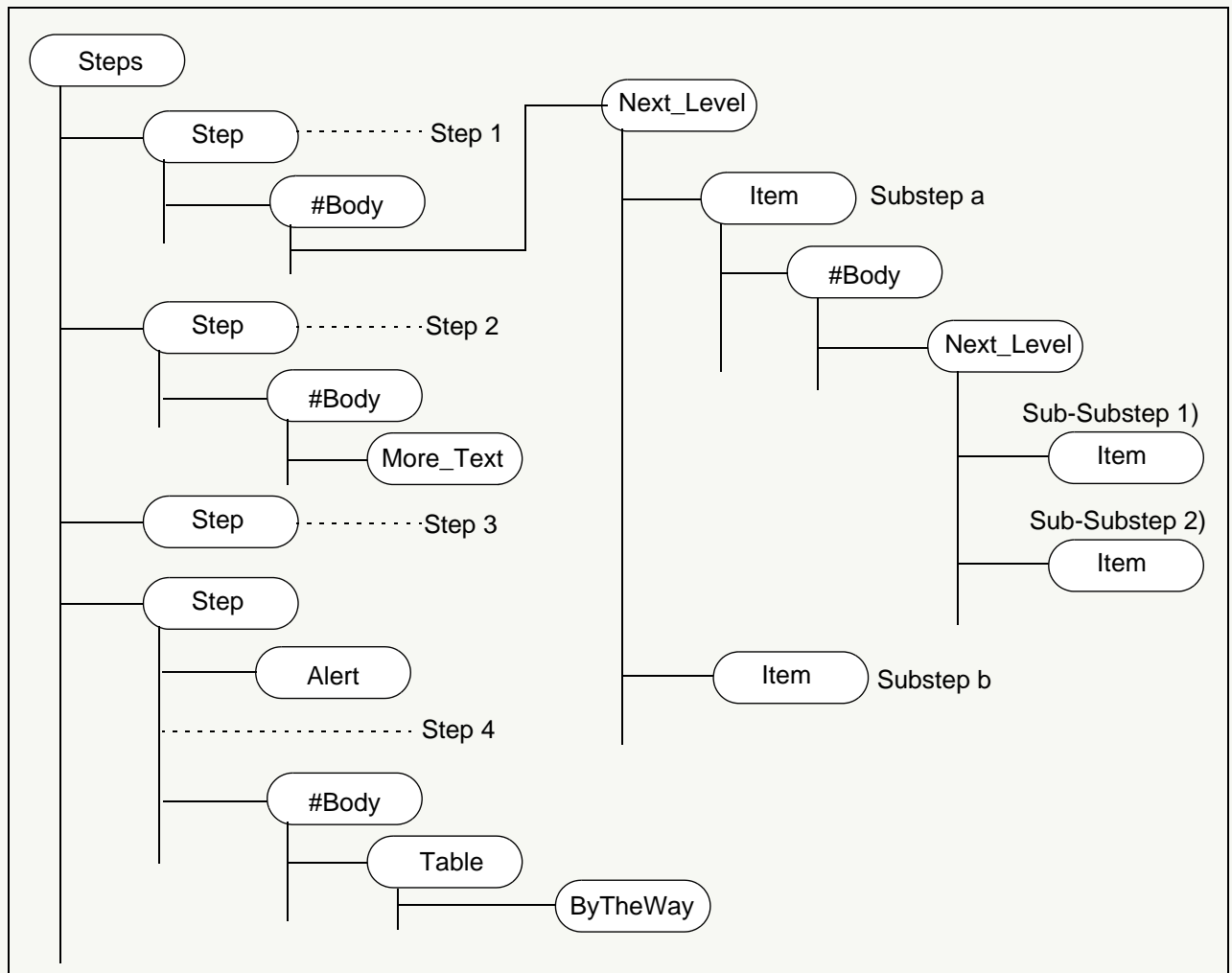


Figure 1. Structural Enrichment Example

4 Encapsulation

*Encapsulation provides an effective way of managing, reusing, retrieving, and delivering packets of information below the document or file level. "Wrapper" elements that provide the means of encapsulation should offer **metadata enrichment** (i.e., additional attributes) that enhance the ability to manage, reuse, and retrieve the encapsulated information.*

Moreover, encapsulation wrappers can greatly facilitate collaborative authoring, where two or more individuals work simultaneously and without conflict on different parts of the same document .

4.1 Uses of Encapsulation Wrappers

Creating Text Insets

If an encapsulation wrapper is valid at the highest level, it can be used to create text insets of the following types:

- Create libraries of repetitively used boilerplate material, such as standardized notes cautions and warnings, contract clauses, etc. By creating each such boilerplate packet in a separately named text flow within a single FrameMaker file, an entire library of boilerplate material can be contained within a single FrameMaker "fragment" file.
- In the same manner described above, each author in a collaborative authoring environment can create his/her individual contributions to a writing project in separately named text flows within a single FrameMaker fragment file. When used in this manner, the master document files that use the individual contributions essentially become "skeletons".

Any separately named text flow from a fragment file can then be added as a text inset to any document that uses the same EDD, as follows:

1. Insert an `Inset_Wrapper` element whose structure rules include the encapsulation wrapper used in the fragment files. The `Inset_Wrapper` element has attributes that identify the name of the source fragment file, the source fragment within that file and any other needed amplifying information (e.g., the author's name).
2. Once the `Inset_Wrapper` element is inserted, any selected text inset from a fragment file can be imported by reference as a child of `Inset_Wrapper`.

After these steps are taken, all text insets in a document are automatically updated whenever their source fragment is updated in its fragment file.

Encapsulation of information packets within documents

Minimum Deliverable Units (MDUs)

Candidates for encapsulation include:

An MDU would typically be a single subject (e.g., a section) within a document. The section and all its subsections constitutes an information packet suitable for independent delivery for on-line viewing or other purposes. Delivery of MDUs stored in a database could result, for example, from the outcome of a search for some combination of attribute values in MDU encapsulation wrappers. Only the encapsulated MDUs which satisfy the search criteria are delivered. This delivery approach allows users to find and analyze relevant information much more rapidly than if an entire document were delivered for each database "hit".

Minimum Reusable Units (MRUs)

An MRU is a group of contiguous elements that forms a reusable information packet. Delivery of MRUs could result, for example, from the outcome of a database search for some combination of attribute values in MRU encapsulation wrappers. Such queries would allow authors to rapidly determine which MRUs, if any, match the requirements for for a specific reuse.

4.2 How Encapsulation Wrappers are Implemented

Structure Rules

Encapsulation wrappers are special attachment spines that are distinguished from “ordinary” attachment spines by their different purpose. The structure rule for an encapsulation wrapper is nearly identical to the structure rule for the “ordinary” attachment spine that contains it. Consequently, any subset of contiguous elements attached to an “ordinary” attachment spine can be encapsulated.

But these same encapsulation wrappers are also valid at the highest level so that they can be used to create text insets in separate text flows of a fragment file. [Figure 2](#) shows an implementation.

Metadata Enrichment

Attributes provide the way for encapsulation wrappers to provide details about the information content, as well as information needed to properly manage and reuse the data. Here is a representative list of the attributes which might be included in encapsulation wrappers:

- ResourceID: An Identifier (e.g., a Universal Resource Identifier) that uniquely identifies the resource represented by the MDU or MRU.
- ProjectID: Identifies the project under which the encapsulated data was created.
- WorkOrder: Identifies the work order that authorized the creation of the encapsulated data.
- Subject: The subject(s) of the encapsulated data.
- Description: More detail about the content of the resource.
- Purpose: Describes the purpose of the information.
- InfoType and InfoSubtype: These attributes Identify the information type and sub-type.
- UsedIn: Identifies the document for which the encapsulated data was originally created.
- Effectivity: Indicates product release numbers, model numbers, etc. for which the encapsulated data is valid.
- ECOs: Identifies Engineering Change Orders that have impacted the content of the encapsulated data.
- Author: Identifies the author of the encapsulated data.
- Owner: Identifies the owner (e.g., a department or project) that has control over the content of the encapsulated data.
- SecurityClass: The security classification of the encapsulated data.
- Keywords: Lists the applicable keywords/phrases. Allows keyword searches to be conducted on attribute values rather than text, guaranteeing that all hits are relevant.
- Correlations: One or more attributes identify design documents, spec-

ifications, regulations, etc. that influenced the information content. For example, an OSHA regulation may prescribe how a caution or warning is written. If the regulation changes, an attribute search for the regulation number yields all cautions or warnings that might be affected by the change.

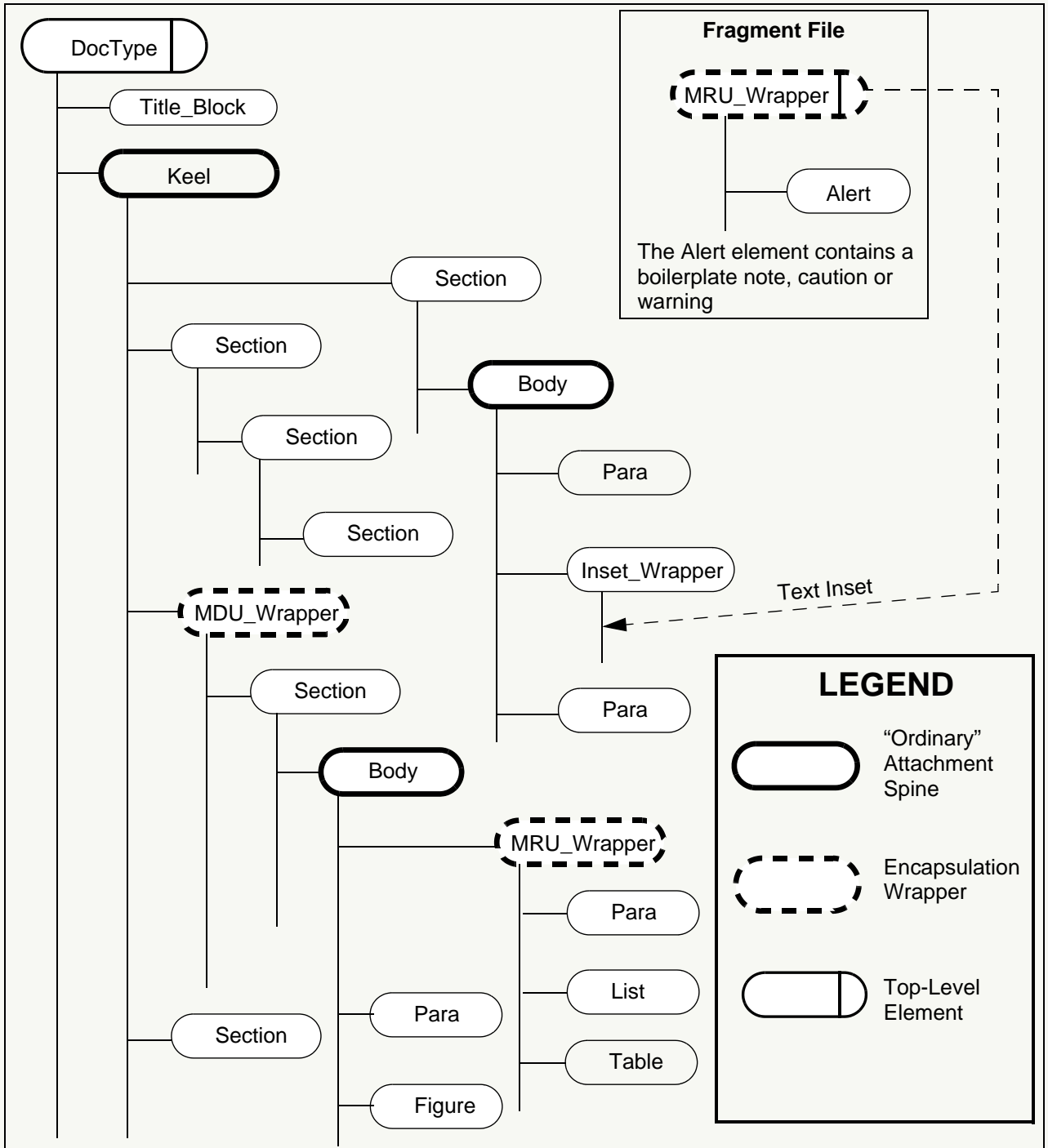


Figure 2. An Encapsulation Wrapper Implementation

4.3 Low-Level Encapsulation

The information packets (hereafter called **chunks**) within each titled section should also be titled and encapsulated.

An example of a multilevel chunk structure (Level 1)

Level 2 Chunk

Level 3 Chunk

Level 4 Chunk

This is an example of a multilevel chunk structure. At each level but the last, the title appears in the sidehead, and all of the document objects (text, graphics, tables, etc.) that are part of the chunk appear in the normal text column. Each chunk element has an optional #Body attachment spine for attaching such document objects.

Formatting differentiates a second-level chunk from a first-level one.

Indenture and formatting differentiate a third-level chunk from a second-level chunk.

Formatting differentiates a fourth-level chunk from a third-level chunk.

Level 5 Chunk: The fifth-level chunk appears in the normal text column, with the title paragraph runin with the text paragraph that follows.

Chunk structures provide a simple way to title and encapsulate information packets below the section level

[Figure 3](#) is the structure view of the 5-level structure shown above. Notice that the `Chunk_Level1` element encapsulates the entire structure. It should also be observed that chunk structures are natural candidates to become MRUs, in which case their metadata can be enriched by wrapping them in an `MRU_Wrapper`.

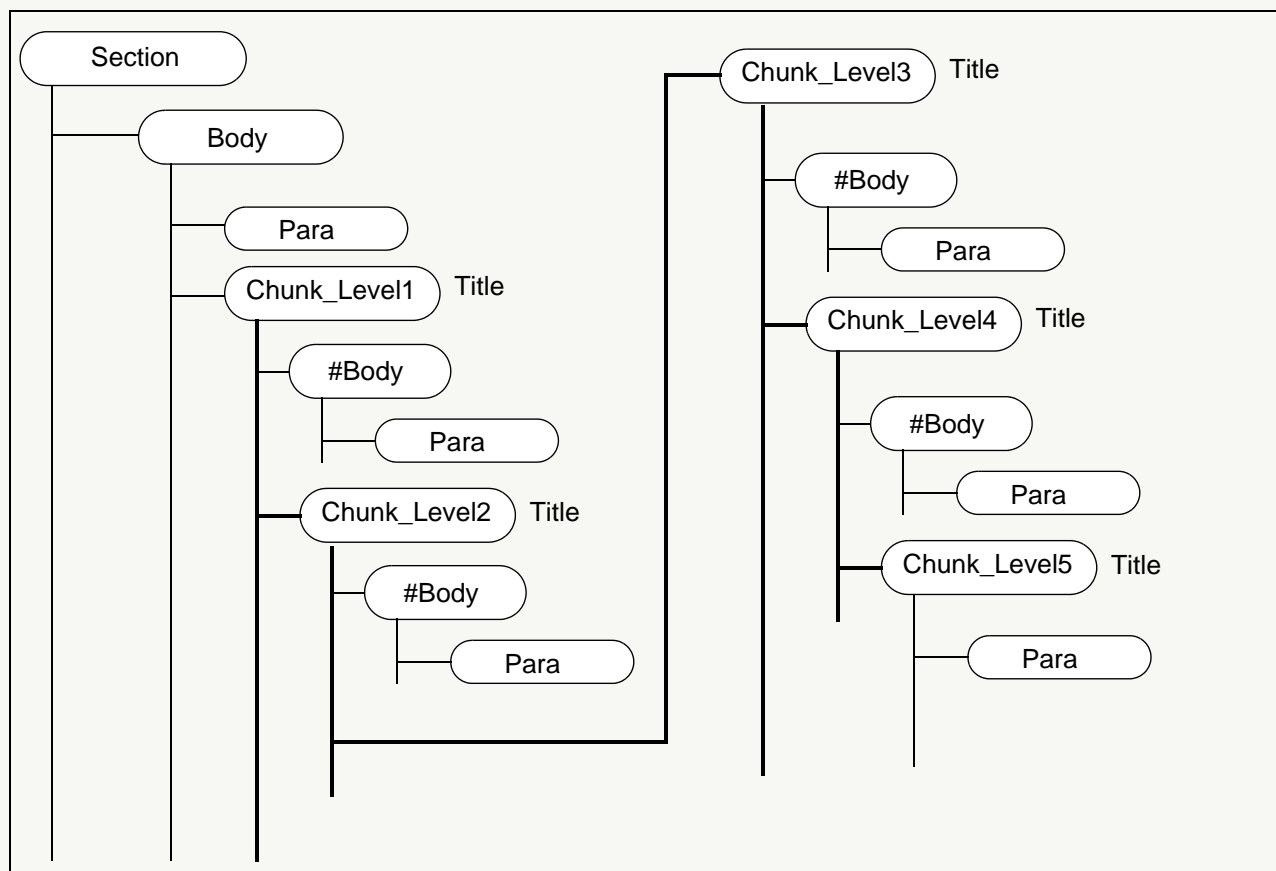


Figure 3. Chunk Structure Implementation

5 Extensibility of the Modular Structure

Having demolished the reductionistic approach to DTD/EDD design in Section 2, I now resurrect it.

The Modular Design Concepts Presented Thus Far

The modular design presented in Sections 3 and 4 includes:

1. Three levels of “Ordinary” attachment spines, including:
 - a. `keel`, which is the attachment spine for first-level section heads and MDU encapsulation wrappers.
 - b. `Body`, which is the mandatory attachment spine, or “sub-keel”, for chunk structures, miscellaneous document objects, and MRU encapsulation wrappers under section heads.
 - c. `#Body`, which is the optional “structural enrichment” attachment spine for attaching miscellaneous document objects and MRU wrappers under chunk structures, items, paragraphs, and other text container elements.
2. Encapsulation wrappers, which include:
 - a. Wrappers that are valid at the highest level, which are used to create text insets in fragment file text flows.
 - b. Wrappers for Minimum Deliverable Units (MDUs), which can be attached to the `keel` attachment spine, or to first- and second-level section heads.
 - c. Wrappers for Minimum Reusable Units (MRUs), which can be attached to the `Body` and `#Body` attachment spines.
3. Titled Chunks, attached to the `Body` attachment spine, which serve as encapsulators of low-level structure.

[Figure 4](#) presents a generalized structure view showing how documents are constructed from these modular components.

The main advantages of a modular structure are its extensibility and flexibility

Suppose, for example, the DTD/EDD defines many different document types, and each such type requires various types of MDU/MRU-type wrappers whose names describe doctype-specific information content (e.g., the names of the major structural components of an automobile or aircraft). This could be easily done, as follows:

1. Expand the structure rules for the `keel`, `Body`, and `#Body` attachment spine elements to include the names of the additional MDU- and MRU-type wrappers that can be attached to them.
2. In the structure rule of the top-level element for each doctype, insert an exclusions line that lists those MDU- and MRU-type wrappers that are not applicable to that doctype.

The structure rules for each doctype-specific encapsulation wrapper could define structures and elements that are unique to that doctype.

If this approach were taken, information interchange and universality could be preserved by adding a “Generic” doctype whose top-level element does not have any exclusions, thus all MDU- and MRU-type wrappers and their children would be allowed in the “Generic” doctype.

It’s also possible to define doctypes without section heads. The structure rule would omit the `keel` element, replacing it with a `Body` element, which allows titled chunks to be substituted for section heads.

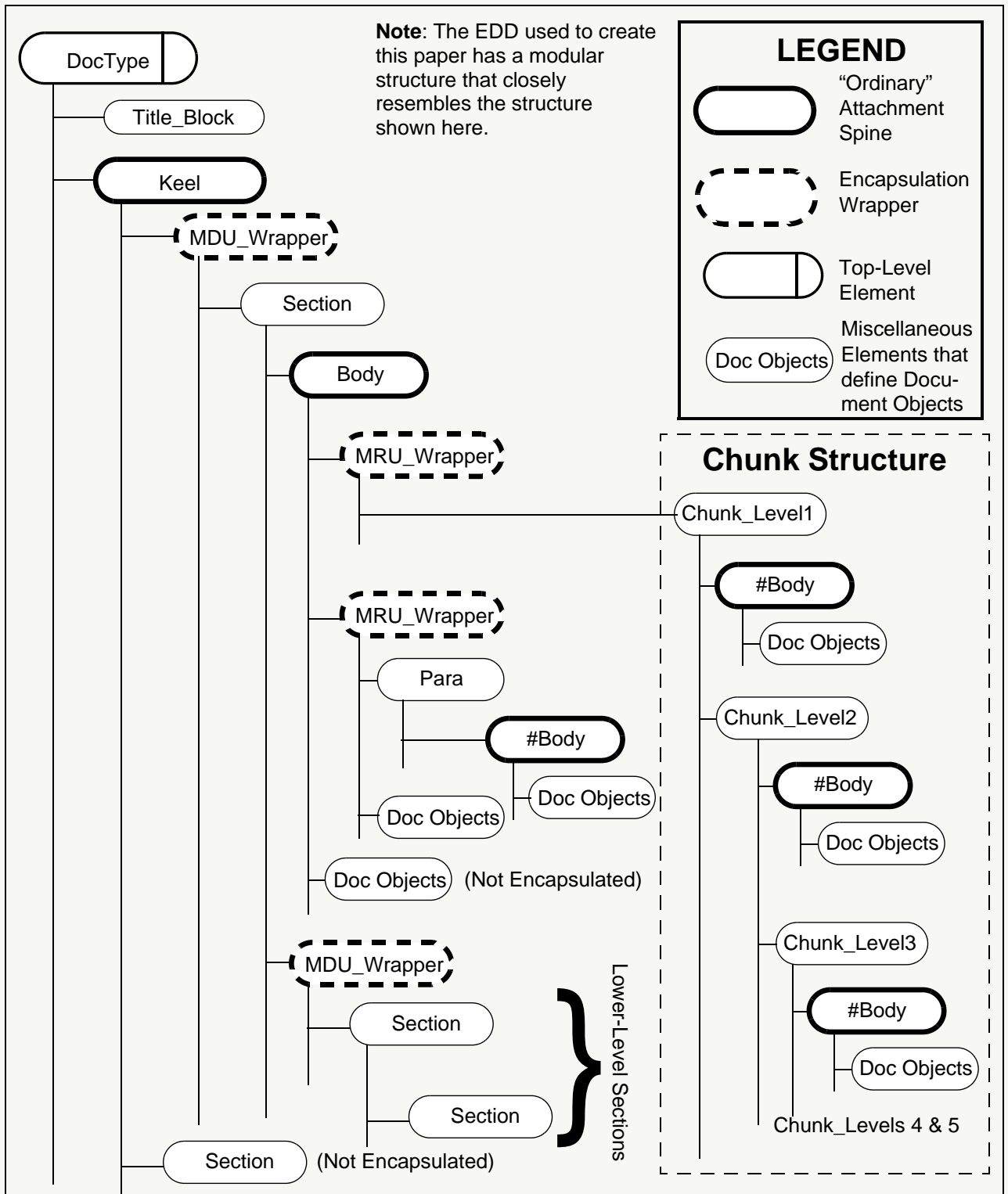


Figure 4. The Modular Document Structure

6 Using FM+SGML's Unique Feature Set

FM+SGML's feature set makes it well-suited for authoring structured documents, and as a print engine for outputting them to paper or PDF.

6.1 Advantages of FM+SGML as an Authoring and Formatting Tool

The authoring tool must provide authors with a WYSIWYG view

This statement simply affirms the basic premise underlying all high-end DTPs, namely that authors cannot effectively design how information is presented unless the on-screen editing view is an accurate representation of how the information will appear when it is read by the end user.

Authoring and formatting features of FM+SGML

- Its capability, through templates and EDD format rules, to format structured documents is unmatched
- Its WYSIWYG editing view assures that authors can effectively utilize those formatting capabilities
- Its interactive structure view serves as a powerful editing tool, as well as an aid to the analysis and management of structure
- Its element catalog shows authors which elements are valid at any given insertion point.
- Its built-in validation capability detects the following types of anomalies:
 - Invalid elements
 - Missing elements that are required
 - Invalid attributes
 - Invalid values in attributes
 - Missing values for required attributes.

FM+SGML stops at each anomaly, highlights it in the structure view, and describes the defect

- Built-in Find/Change capability for searching on/changing, element tagnames, attribute names, and attribute values
- FM+SGML eliminates most requirements for format-related processing instructions (PIs) in SGML document instances.

Graphic Conversion

Graphics in FM+SGML-created structured documents can be exported to SGML as entities in almost any read/write rules-specified graphic format.

Text Insets

FM+SGML's text inset feature provides an innovative way to facilitate information reuse and collaborative authoring. Each structured text inset is encapsulated by an encapsulation wrapper element in a separate text flow within a fragment file.

6.2 FM+SGML's Formatting Capabilities

Formatting nuances help readers to comprehend complex information, and to rapidly scan through documents in search of the particular information they seek. Format variations are likely to be needed for different document types, different departmental standards within the enterprise, and different modes of information delivery.

The printed or PDF version of a document usually has the most demanding formatting requirements

It follows, therefore, that the combination of the EDD and its companion template(s) must be capable of satisfying the formatting requirements for high-quality printed or PDF output.

When this goal is achieved, it is reasonable to assume that the information design will also be adaptable to the formatting requirements for alternative (and less demanding) information delivery modes.

6.2.1 The Formatting Role of the EDD

The EDD's format rules define the format tagnames that will be included in the template's catalogs

The EDD prescribes how document objects are formatted.

When element definitions are imported into a template, the import action creates the EDD-specified tagnames in the template's catalogs. The affected catalogs are:

- The paragraph catalog, which contains the base paragraph format tags specified in the format rules of container elements.
- The character catalog, which contains the character format tags specified in the format rules of text range elements.
- The table catalog, which contains the table format tags specified in the format rules for table elements.
- The cross-reference format catalog, which contains the cross-reference tags specified in the format rules for cross-reference elements.

The EDD defines Format changes that are applied to the base paragraph tags in the template

These format changes are specified in All Context, Context, Level, Prefix, and Suffix rules of individual container elements.

6.2.2 The Formatting Role of FM+SGML Templates

1. The template specifies the formatting details of the tags added to its catalogs when the EDD is imported into it

An FM+SGML template is used to create new structured documents, or to modify the formatting of an existing structured document. Many templates may be created for the same EDD, each having format variations for different document types and/or different information delivery modes.

When these tags are created in the catalogs by the Import Element Definitions action, they are plain-vanilla. Formatting details must now be added by the template designer to fulfill the intended purpose of each tag.

2. The template has nearly total control over many formatting aspects

Among the aspects that are under near-total control of the template are:

- **Page Layouts**, including running headers and footers
- **Character formats**. In this case, the EDD format rules for text ranges specify "Use Character format", which creates the character tags in the character catalog when the element definitions are imported into the template. The template designer has control over the formatting details for each such character tag.
- **System and User Variable Definitions**

- **Color Definitions**
- **Reference Frames** on reference pages
- **Conditional Text Settings**
- **Math Definitions.**

3. The extent to which a template can influence paragraph formatting is determined by how the EDD's format rules are written

The Three EDD design methods described below demonstrate how the EDD's format rules can affect the extent to which templates influence paragraph formatting.

3.1 The "Top-Down Inheritance" Method

When this method is employed, elements that are valid at the highest level specify a base paragraph format (e.g., Body). Each container element descendant of that top-level element inherits the Body paragraph format, plus any antecedent format changes to it which are not overridden by the element's own format rules. Consequently, any template created from such an EDD will have a single paragraph tag named Body in its paragraph catalog. Obviously, the influence of such a template on paragraph formatting is nil.

3.2 The "Use Paragraph Format" Method

When this method is employed, all container element format changes are in rules that specify "Use paragraph format". This was the only method available in FM+SGML's predecessor, FrameBuilder. In this case, the template has virtually total control over paragraph formatting.

3.3 The "Middle-of-the-Road" Method

Both of the two previously described methods have serious disadvantages. The middle-of-the-road method specifies a base paragraph format in the format rules for most container elements having <TEXT> in their general rule, so as to stop all antecedent format inheritance. The format rules required to modify the base paragraph format for each element context do not, in most cases, specify "Use paragraph format". Instead, they specify "Use format change list" (more about the advantages of format change lists in the next section).

Since many different container elements can specify the same base paragraph format, the huge proliferation in paragraph tags produced by the "Use Paragraph Format" method is avoided. Instead, a relatively small set of base paragraph formats is defined to accommodate variations in such formatting parameters as font family, font size, line spacing, table cell parameters, hyphenation, and Frame Above/Below for different types of container elements. Although there may be some exceptions, the EDD's format change lists generally do not override these template-defined formatting parameters, thus the template designer has control over them, making it possible to develop multiple templates for the same EDD in which the same set of base paragraph formats is formatted differently for different document types and/or different information delivery modes.

6.2.3 Advantages of Using Format Change Lists

When format change lists are utilized, virtually all EDD format rules for modifying base paragraph formats specify:

Use Format Change List: Name

Where "Name" is the name of a particular format change list.

Format Change Lists are modular formatting building blocks

Format change lists should be viewed as modular formatting building blocks that can be combined in many different ways to produce different formatting outcomes (e.g., the formatting for a particular element context can be specified in the element's All Context and Context rules to be the composite of two or more format change lists).

The Impact on EDD Design

Format rules in an EDD refer to format change lists, where the formatting details are provided. Since many format rules in many different elements can use the same format change lists, the volume of formatting details in the EDD can be greatly reduced, providing better management of the formatting problem, as well as making it easier to modify the EDD. In a modular EDD design, format change lists can facilitate reuse of structural modules between different EDDs.

Format Change Lists can provide even greater format adaptability

All of the format change lists in an EDD can be grouped together into categories at the end of the EDD. This makes it possible, for example, to easily clone multiple versions of an EDD, all of which have identical element names, structure rules, and format rules, plus the same set of named format change lists. However, the formatting details in those format change lists can vary from version-to-version so as to accommodate wide variations in format for different document types and delivery modes.

Suppose, for example, two versions of the EDD, each having its own template, are created, where one version is intended for producing printed documents, and the other version is used for producing PDF documents for on-line viewing. Each EDD is identical, except for the formatting details in their format change lists. These variations define the formatting differences between the two delivery modes. Further variations (e.g., differences in font family and font size) may exist in the formatting of the base paragraph set in the two templates.

Suppose further that you create the printed version first. Now, you want to produce the on-line version. All you have to do to accomplish the conversion is import into the document the formats and element definitions from the "on-line" template. Consequently, the same document files can be used to produce both the printed and on-line versions, even though they may differ widely in their formatting details.

6.2.4 Using Attributes to Provide Authors with Formatting Options

Without the use of formatting attributes, a container element's paragraph format is determined solely by its context within the structural hierarchy. If formatting attributes are not provided and different formatting options are required for the same basic element, multiple versions of that element (each with a different name, the same structure rules, and different format rules) must be created. This unnecessarily complicates the EDD, as well as the authoring task.

Choice-type formatting attributes provide an excellent way to avoid element proliferation, and can provide authors with the options they need to

Formatting Attributes for a Paragraph

optimize the presentation of complex information.

Take, for example, the ubiquitous Para element that serves as the general-purpose text container in many DTDs/EDDs. Here are some of the formatting options an author might like to have for the Para element:

1. Select the horizontal alignment of the paragraph as Left, Right, Center, Justified, or aligned on a decimal point.
2. Select the table cell vertical alignment of the paragraph as Top, Middle, or Bottom
3. Specify the amount of indenture of the paragraph from the left margin (e.g., Level 1, Level 2, etc.)
4. Change the font size of the entire paragraph from Regular (the default font size) to 2 points larger (Large) or 2 points smaller (Small) than Regular, with a corresponding change in the line spacing
5. Change the font of the entire paragraph to Courier or some other special font to represent, for instance, a computer message or a typed command
6. Make the style of the entire paragraph bold, italics, or underlined
7. Make the paragraph span all columns, both the sidehead and normal column, or only a single column
8. Force the paragraph to appear at the top of a column (Column Break) or at the top of a page (Page Break).

All of these options can be readily provided by choice-type attributes, in which the default value for each attribute produces the format specified by the applicable base paragraph format in the template.

Formatting Attributes at the Highest Level

Formatting attributes can also be used at the highest level to provide formatting options for an entire document. For example:

1. Autonumbering options for chapter and appendix titles (e.g., None, Alpha, or Number), where, if the title is numbered or lettered, the number or letter is prefixed to the section head numbers, if any.
2. Autonumbering options for section heads (e.g., None, Number Major Sections Only, or Number All Sections).
3. Section head styling options needed for different document types or delivery modes.
4. Autonumbering options for chunk structures below the section level, allowing titled chunks to be used, for example, as additional numbered section levels.
5. Figure, Table, and Equation autonumbering options (e.g., Restart at Chapter, Restart at Each Major Section, or Number From Start of Book), where:
 - a. If the first option is chosen, numbering is restarted at 1 in each chapter or appendix, and, if the chapter or appendix title is auto-numbered, that autonumber is prefixed to the figure, table, and equation autonumbers.
 - b. If the second option is chosen, numbering is restarted at 1 in each major numbered section, and the section number is prefixed to the figure, table, and equation autonumbers.

- c. If the third option is chosen, there is no prefix to the figure, table, and equation numbers, and they are numbered consecutively from the start of the book.

Implementation in the EDD

Formatting attributes for elements provide an effective way to give authors a wide range of formatting options without a concomitant proliferation in the number of elements in the EDD. Moreover, by using formatting attributes, EDD modifications to accommodate additional formatting requirements are easily implemented, either by adding new attributes, or by adding more choices to existing attributes, thus the DTD/EDD structure rules are unaffected.

Here is a typical format rule for a Style attribute in a Para element:

```
If context is: [Style = "Bold"]
  Use format change list: Bold
Else, if context is: [Style = "Italics"]
  Use format change list: Italics
Else, if context is: [Style = "Underline"]
  Use format change list: Underline
Else if context is: [Style = "Normal"]
  Use format change list: Normal
```

Note that many different container elements could have a Style attribute, and all would reference the same format change lists in their identical format rules for the Style attribute.

6.3 Style Guide Enforcement

An EDD's format rules constitute an auto-enforced style guide, freeing authors from concerns about style guide compliance. If authors attempt to override the format rules by applying unstructured character formatting within paragraphs, or by making *ad hoc* changes to paragraph formats, those overrides can be removed. This is accomplished by re-importing the document's Element Definitions (with Remove Format Overrides turned on). This action restores the affected document to full compliance with the EDD's format rules, including removal of all *ad hoc* character and paragraph formatting.

6.4 FM+SGML's Utilities and Developer's Tools

The following utilities and tools are built into FM+SGML:

- **Batch Conversion Utilities** to convert FM+SGML documents to SGML, or to convert SGML documents to FM+SGML
- **Generate Structure Rules Tables** for an unstructured document, and then convert it to a structured document using those structure rules
- **Generate and Apply Paragraph and Character Format Tags** to a structured document in preparation for mapping those tags for an HTML conversion
- **Create a new EDD**
- **Create an EDD** from an existing DTD
- **Open a DTD** for editing

- **Create a DTD** from an existing EDD
- **Parse an Existing SGML Document Instance**, and log all detected anomalies
- **Edit the SGML import/Export application file**
- **Create a new Read/Write Rules file** for SGML import/export
- **Check an Existing Read/Write Rules file**, and log all syntax errors.

6.5 Customization

FM+SGML's Customization capabilities include:

- Customizing menus
- Customizing graphic filters
- Developing API clients with the Frame Developer's Kit (FDK)
- Developing SGML Import/Export Applications for accomplishing conversions between SGML/XML and FM+SGML
- Using FM+SGML 5.5.6's ODMA interface to create bridges to ODMA-compatible database repositories. Such bridges could allow FM+SGML to:
 - Check SGML/XML documents (or portions thereof) out of the database, and import them into FM+SGML for editing
 - Check FM+SGML documents (or portions thereof) back into the database as SGML/XML.
 - Query the database
 - Navigate around in the database.

7 XML for the New Paradigm

Extensible Markup Language (XML) will revolutionize the way in which information is handled and processed. It promises to make information "smarter" by including machine-readable data about the structure and content of information objects (hereafter called "resources"). Resources can include documents, document fragments, and external entities such as graphics and individual database records. Resources are always identified by Universal Resource Identifiers (URIs), plus optional anchor IDs. The extensibility of URIs allows the introduction of identifiers for almost any resource imaginable.



Note: Some of the text in this section paraphrases information that was originally contained in "A New Dawn", an article about XML by Glyn Moody.

XSL Stylesheets

It is a basic rule of XML that content and presentation are separate, thus XML tags contain no hint about how the information contained therein should be formatted/displayed. One candidate for creating stylesheets uses an XML language called eXtensible Style Language (XSL). XSL stylesheets format the data for display or printing, but also promise much more. For example, different stylesheets could be applied to the same data; each stylesheet could hide some information chunks, and display others.

Hypertext Links

Another XML application, called XLink, makes hypertext links much more robust than they are in HTML or PDF. XLink offers a number of new features such as links that indicate (before you click the mouse) what kind of link they are, and links that provide a pull-down menu of options. Each link specifies the unique URI of its destination node, thus any node anywhere on the web or within a site is reachable.

Unicode

Unicode eliminates the need for using entity references to represent *glyphs* (i.e., characters) whose code points are outside the range of printable ASCII characters. Instead, Unicode provides a unique codespace for each of the world's languages, plus many archaic languages. Each glyph in each language has a unique code point. Glyphs that are common to more than one language (e.g., punctuation) have a single code point that is used by all languages.



Note: The Unicode standard defines the code point for each glyph, not the glyph itself. Separate code points are provided for each diacritical mark, thus characters having diacritical marks can be produced by specifying the code point for the character glyph, followed by the the code point for the diacritical mark glyph.

Different languages can be freely intermixed within the same document. Unicode-compliant fonts are already available which permit the intermixing of as many as 40 different languages with a single font.

For all of the reasons cited above, Unicode will provide a superior solution to the translation of information into different languages.

New Languages

Unicode also provides reserved blocks of codespace for different disciplines to create new language options. Musical, mathematical, and chemical notation languages have already been developed. More special languages for other disciplines will undoubtedly follow. Reserved blocks of codespace are also available for enterprises to create special typograph-

	<p>ical symbols (e.g., logos, and other enterprise-unique icons and characters). Users (human and non-human) will be able to treat information in these new languages just like ordinary text, analyzing and manipulating it in any way they see fit.</p>
<p>Resource Description Framework (RDF)</p>	<p>The most potent new feature of XML involves the handling of metadata. RDF is a flexible model for representing named properties and their property values. It allows information about resources to be stored as if it were in a structured database so that it is machine readable. An RDF instance is defined by a named <i>model</i> that specifies the syntax and property set for a given type of resource. Complex relationships can exist between resources and properties within an RDF.</p>
<p>Benefits</p>	<p>The benefits of RDF include:</p> <ul style="list-style-type: none"> • Much smarter searching • Greatly improved transfer and pooling of data, such as amalgamation of bibliographies from different enterprises to create global library catalogs • Greatly improved information management • Many other novel automation functions are made possible by the machine readability of RDF.
<p>Syntax</p>	<p>RDF uses standard XML encoding as its interchange syntax. The RDF wrapper element marks the boundaries in an XML document between which the content is explicitly intended to be mappable into an RDF data model instance that defines a property set. A <code>Description</code> container element child of the RDF wrapper contains the property set.</p> <p>Suppose, for instance, that an RDF model named <code>rdf</code> were defined for this document, and that this model specifies two properties: <code>Title</code> and <code>Author</code>, both of which are defined in a schema named "s". The complete XML document containing the RDF element would be as follows:</p> <pre><?xml version="1.0"?> <rdf:RDF> xmlns:rdf="URI₁" xmlns:s="URI₂" <rdf:Description about="URI₃"> s>Title="FM+SGML Information Design" s:Author="Dan Emory" </Description> </RDF></pre> <p>Where:</p> <ul style="list-style-type: none"> <code>xmlns:</code> is followed by the name of an RDF model syntax or a property schema <code>about</code> is an attribute of the <code>Description</code> element that specifies the URI (<code>URI₃</code>) of the document being described <code>Title</code> and <code>Author</code> are properties specified (in this example) as attributes of the <code>Description</code> container element <code>URI₁</code> is the URI containing the RDF model named <code>rdf</code>

URI₂ is the URI containing the property schema named *s*

Observe that the above RDF example is machine-translatable into any of the following English sentences:

Dan Emory is the author of the resource URI₃, whose title is "FM+SGML Information Design".

OR

Resource URI₃, which is a document entitled "FM+SGML Information Design", was created by Dan Emory.

OR

The document entitled "FM+SGML Information Design", which is identified as resource URI₃, was created by Dan Emory.

If the top-level element for this document has (as it does) attributes named *Title* and *Author*, those attribute values could be machine-extracted, and inserted as the values of the corresponding properties in the RDF. Alternatively, values of the *Title* and *Author* properties of the RDF could be machine-extracted, and inserted into the corresponding attributes of the top-level element for this document. Whichever way it's done, this would assure that the values of properties in the RDF always agree with the corresponding attribute values included in the top-level element of the resource being described.

From the foregoing, it's also evident that RDFs could be produced for *MDU_Wrapper* and *MRU_Wrapper* elements within a document. Each such RDF would have an unique URI value in its *about* attribute.

The RDF syntax also accommodates more complex structures than that in the example above to handle, for instance:

- Cases where a single property has multiple values (e.g., two or more authors)
- Cases where there are nested description elements (e.g., the *Author* property could itself be defined as a resource having a nested *Description* element whose *about* attribute identifies the URI for Dan Emory. This *Description* element for the *Author* resource defines two properties: *Name* and *Email*).

Automated Creation and Delivery of Custom Documents from a Database Repository

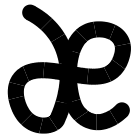
If all XML-based resources and their RDFs are stored in a searchable database repository, well-formed XML documents, consisting of miscellaneous document fragments (e.g., encapsulated MDUs and/or MRUs, each having its own RDF) could be assembled on-the-fly and delivered to the requesting user. The user originates a database query specifying value(s) in one or more RDF properties. Each time a database hit occurs, the URI in the *about* attribute of that RDF is used to fetch the fragment. The fetched fragments would then be assembled into a document, with the sequence in which the fragments appear being determined by some criterion (e.g., hit rating or parent document).

Automatic Access

Automatic access allows application programs to extract data held between pairs of tags and then manipulate that data automatically for any purpose imaginable. For example, equations or chemical information could be extracted from a document, modified, and then sent to a computer-controlled process of some sort.

Using these automatic access capabilities, many believe that XML could provide superior solutions for applications such as:

- Electronic Data Interchange (EDI), which attempts to define standard ways for companies to exchange orders electronically.
- Electronic Record Keeping in Healthcare, where the ability to pool medical information from many different sources to search for patterns of disease or successful treatments could transform epidemiology.

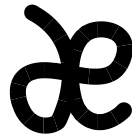


Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com



Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com

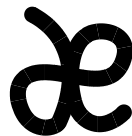


Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com

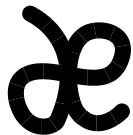


Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com

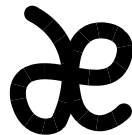


Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com



Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com

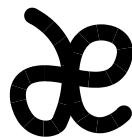


Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com



Dan Emory & Associates

Information Design Specialists

Dan Emory

10044 Adams Ave. #208
Huntington Beach, CA 92646
Voice/Fax: 949-722-8971
Email: danemory@primenet.com