Reviews | Essay

Structure rules!

Why DTDs matter after all

Chet Ensign
Manager, Data Architecture
Matthew Bender & Company, Inc.
Two Park Avenue
New York, NY 10016

EMAIL censign@bender.com
FAX 212-448-2469
TEL 212-448-2466
WEB www.bender.com

EXTENSIBLE MARKUP LANGUAGE or XML, a simpler form of SGML, has introduced the concept of a *well-formed document*, one that doesn't need a DTD. This sounds wonderful — all the benefits of SGML without the expense or restrictions of that darned DTD. But to dismiss DTDs as arbitrary, restrictive constraints on creative freedom is to miss their role as documentation of the truly valuable parts/aspects of your content and as descriptions that enable users to build lights-out processing systems for leveraging that content. Assuming your goal is to make your content more generally useful and to make it valuable across your entire enterprise, you will quickly find that DTDs are not enemies but allies. And once you make that discovery, you'll want these two books in your library.

Eve Maler & Jeanne El Andaloussi
Developing SGML DTDs From Text To Model To Markup
1996 Prentice-Hall

David Megginson
Structuring XML Documents
1998 Prentice-Hall

You looked into SGML years ago. The payoffs sounded promising, but the expense and effort to get started looked daunting. In particular, SGML demanded that you create this thing called a DOCUMENT TYPE DEFINITION or DTD. The word on the street was that creating one of these things was a big undertaking. You would have to analyze virtually every document you produce in detail and look closely at all the different ways you wanted to use them. Worse, you heard that DTDs would restrict your flexibility — they would force you to create all your documents the same way! You decided to wait for something better.

Now there is EXTENSIBLE MARKUP LANGUAGE or XML, a simpler form of SGML. XML has introduced the concept of a *well-formed document*, one that doesn't

Markup Languages: Theory & Practice 1.1 (1999): 101–112

need a DTD. This sounds like what you were looking for — all the benefits of SGML without the expense or restrictions of that darned DTD.

XML has indeed advanced the state of the art of structured information. In particular, it has recognized that DTDs are not required at *every* point in the information's processing cycle. But to dismiss DTDs as arbitrary, restrictive constraints on your creative freedom is to miss their role as documentation of the truly valuable parts/aspects of your content and as descriptions that enable users to build lights-out processing systems for leveraging that content. Assuming your goal is to make your content more generally useful and to make it valuable across your entire enterprise, you will quickly find that DTDs are not enemies but allies. The act of creating them will teach you things about your editorial and production systems that you never knew, and the act of using them will make your data predictable so that you and others can build systems that use it. And once you make that discovery, you'll want these two books in your library.

Imagining life without a DTD

Finally; the magic bullet we've been looking for — XML. State-of-the-art document processing without the DTD. Cutting edge technology with none of that restrictive, expensive, picayune type-A obsessive detailing of every nuance of every document you would ever want to create. SGML for the rest of us. It *does* sound appealing.

For those coming to structured markup languages for the first time, a DTD can look like an elaborate tool used to transfer as much of their project budget as possible to vendors and consultants. It can also look arbitrarily restrictive. Especially to those whose main experience has been HTML, where a creative ability to sequence tags in new and inventive ways is a hard-won design skill, the insistence on something that looks like a prescriptive, pre-defined set of rules for document markup probably seems unreasonably constricting. For someone who has that impression, it must be very tempting to interpret XML as "SGML without the DTD."

The XML standard itself, of course, makes no such claim. The authors of the standard simply recognized one of the premier lessons of the World Wide Web — that you don't need the DTD at every stage of the information's life. Especially where final delivery is over the Web, the less exacting concept of the *well-formed document* can be applied without compromising system effectiveness.

However, let us assume that you are interested in XML because you want faster, more efficient or more *effective* systems for turning your company's information into end products. (If your interest is in unique, hand-crafted products, neither XML nor SGML is likely to be of much use or interest to you.) You want to get as close as possible to a write-once/use-often system that turns

authorized content into a variety of information products at the push of a button. You want to move content from your experts to your editors to your clients or customers without all the file converting and reformatting that you currently do by hand. You want to use metadata to filter or classify your content in powerful new ways or to build interactive expert systems. If any of this is true, then you'll need DTDs. They are keys to building systems that fit that description. Here is why, and here is how they will benefit you.

Tag wars, format crashes and other symptoms of the unexamined (document processing) life

The idea that a DTD is an arbitrary attempt to limit the creativity of authors and designers is a product of our continuing concentration on the look of publications instead of on the information they contain. Of course, the people responsible for producing those products *should* be concerned with their product's visual quality. However, that concern should not constrain the effectiveness of the organization as a whole. To make content a true enterprise-wide resource, *you* must place *your* value on the quality of the information resource itself.

Let's face it; one of the key goals of the exercise (adopting XML or SGML) is *rules-based processing*. Indeed, this is about as basic as you can get. If I can articulate a precise and unambiguous description of my data, you can build software that turns that data into end products without requiring lots of manual intervention. This is a fundamental characteristic of the word "data." To qualify as data, information must come in some predictable form.

Why focus on rules-based systems? Because they are more efficient than we are—orders of magnitude more efficient. By letting the computer do the grunt work, and putting our people resources where they do the most good, where creative talent is truly required, we can create end products faster. We can cut the distance between the experts and the audience. But rules demand reliability, and building rules-based systems is impossible if we can't rely on information to be structured the same way twice in a row. To achieve that kind of reliability in an organization of any size, you have to start with a close look at the information you are currently producing and at how you are (or want to be) using it.

Unfortunately, most organizations have foregone that kind of self-examination, opting instead to pick a "corporate standard" word processor in the mistaken expectation that this will result in "data." However, the proprietary formats of word processors and desktop publishing programs can be considered "data" only within the confines of their own world. Go outside those confines—convert to a different word processor, turn the files over to your electronic publishing team, use that "Save as HTML" option, or even upgrade your product

(a learning experience some of us may have been through recently)—and the results are usually only an approximation of what you expected.

Sometimes not even that. I vividly recall spending hours on the phone with a word processor's technical support, trying to discover why merging word processing files from a contract author with mine was crashing my computer. We were both using the same version of the same word processor and sticking to basic system fonts, yet trying to merge portions of his files with mine consistently hung my PC. It turned out to be a subtle style sheet conflict (which, by the way, could not be fixed except by saving his work as ASCII and reformatting), but that was cold comfort to someone working in a supposedly "standard" environment.

By now, this is part of the motherhood and apple pie anthem for those of us who believe that open standards are the way to solve these problems. But that doesn't explain the need for DTDs. Why pre-define tags at all, or restrict how they can relate to one another? There are at least three good reasons:

1. Preventing "tag wars";
2. Creating a formal specification for users anywhere in the organization;
3. Defining metadata that enables the next generation of information products.

Tag wars

Tag wars are what you get if you define a list of "official" tags and hope that your end users will stick to them. Tag wars are what you get when your UNIX division calls it <man-page> and your PC division calls it <reference>. Tag wars are what Netscape and Microsoft gave us when they decided to differentiate their browsers in the market place, for an especially acute example.

Tag wars aren't all that much different from creating a standard style sheet and giving it to all your authors and editors. It is better than nothing, but it does nothing to ensure reliability or predictability in the content. In fact, approved tag lists, standard style sheets, and house style guides are often most useful as gauges of how far off the mark any single writer's work turns out to be.

In an environment that allows total freedom, people inevitably deviate from the rules, either deliberately, in pursuit of some special effect, or inadvertently, because there are just too many rules to keep in mind. A DTD tackles the tag wars problem by providing a mechanism for checking the markup against the standard. If you do your homework, involve your end users in the design and apply some creativity of your own, you can even develop a DTD that your end users will accept, respect and use.

Describing your data

Formal specifications become important when we start looking at document-type data as a global resource of the organization. Looking at corporate databases gives a useful parallel.

Corporate databases storing things like financial, sales, or operating data are a shared resource of the organization. No one would seriously suggest that individual employees should be able to change those databases at will, or decide that they don't like the way data input is structured and make up their own input fields instead. Too many other critical systems depend on the databases to allow unmanaged changes. Everyone accepts the fact that databases must be managed through policy-driven processes and procedures. Those processes, procedures, forms, and bureaucracies may seem to slow things down, but ultimately they make the organization more effective, in part because they ensure the integrity and reliability of the data.

When we were the sole users of our content, decisions about its structure, format, etc. could reside solely in our hands. But the world we are moving into is changing its attitude towards our content. Information we author looks increasingly like a corporate resource, fed to a growing number of users and distribution channels. Through document management systems and Intranets, it is becoming more universally accessible. It is looking a lot more like data that we have to get serious about managing.

In most organizations, the rate-limiting factor interfering with broader use of content is its inconsistency. There is no specification for content that tells (or assures) downstream users what the content will look like from one department to the next, one writer to the next, even one version to the next. If potential publishers of the material (print production, electronic publishing, the Web site team, the Intranet team) know that content will never follow any rules, then they have no incentive to build automated systems to publish it. (And, in fact, they can't.) The best they can do is use conversion utilities to get them close. After that, they have to clean up by hand. The ad-hoc nature of the word processing or desktop publishing files that the authors and editors produce directly results in the delays in getting it to the consumer.

This problem even extends down to the intra-sentence level. My company's business is legal publishing. In our field, citations to court opinions and legal statutes are critical pieces of information, the keys to making linkages between different publications that add tremendous value for our customers. Yet our industry is still struggling to grab that key and use it simply because the way citations are written is so unreliable. And this despite the fact that there are

standard style guides that address those issues, and despite significant investments in trying to write AI software that recognizes the components of citations with 100% reliability. Believe me; somewhere, somehow the same is true in your industry.

From this perspective, DTDs are the formal specifications of our data. They are the "rules of the game", the documents' counterpart to the schemas that describe corporate databases. DTDs are the roadmaps that tell anyone else building a system what our data will look like, and how it will be tagged and structured. In an environment where our data must be broadly available, where its integrity and reliability determines whether other groups can successfully use it or not, developing and using DTDs are crucial to the overall effectiveness of our organization.

Metadata — The z-axis of information

Consider this: any particular unit of content that you select can have one of a number of characteristics that we can group into several categories: format, organization, navigation, narrative, referencing, and metadata. Format obviously is the way the text looks. Navigation is text that acts as finding aids — tables of contents, indices, and the like. Organization is the way content is grouped into coherent units. Narrative is content-bearing structures; paragraphs, lists, tables, etc. And referencing is text that acts as a gateway to other, related content.

So far, so good. Most systems and formats can handle these to greater or lesser degrees. Metadata is another matter entirely.

Metadata is like the Z-axis of your documents. It is the information in front of or behind the words on display that makes it possible for your to exploit the content in novel and sophisticated ways. For example, metadata about a publication can be used to automate assembly and publishing. Metadata about graphics can allow the print product to use full-page TIFF images while the Web site uses the thumbnail JPEGs linked to a full-sized images.

More significantly, metadata about intellectual expertise can be used to classify content, filter it in more effective ways, or to drive sophisticated interactive systems. For example, metadata about a diagnostic test (test equipment and technician level required, dependencies or prerequisites, average time needed to perform the test, etc.) can drive expert diagnostic and repair systems. Metadata about a mutual fund (risk category, type of fund, typical investor type) can be used to automatically present lists of funds to potential investors or drive pointcast-style information services.

The metadata that is important is going to be different for every organization. The metadata important in my field (legal publishing) is going to be

completely different from the metadata needed by the pharmaceutical companies, semiconductor manufacturers, or the financial services industry. And the only people who are going to really know what metadata is critical are your experts.

This is where XML and SGML separate themselves from all other data formats. There is simply no other reliable and flexible mechanism that enables you to define, capture and store the metadata that is meaningful to you at the precise place in the content where you need it. Some metadata may appear as part of the actual content of the publication (such as the grade of technician required for a test, or the risk category of a mutual fund), but much will not, or will appear in ways that can not be reliably identified. Some metadata will apply to the entire document or file (such as its chapter or appendix number or the name of its author), but much will apply to small, specific components within the document (such as legal citations in a court's opinion, or special effects in a screen play).

The DTD is your vehicle for defining the metadata that is important to *your* organization in *your* language. Some of it will be in the form of elements, much in the form of attributes about those elements. The DTD is where those sorts of choices will be worked out and expressed, in ways that can be used by all the systems in your enterprise.

What is really involved in *document analysis*?

So far, so good. We're looking at DTDs in a new light: as tools for documenting our data in precise and unambiguous ways, for preventing tag wars and other types of disjoins, and for defining the metadata that is mission-critical to our current—and next—generation products. But this view suggests that we will want more than just canned DTDs. We'll want to create our own or modify industry standard DTDs to suit our objectives. So the obvious next question is: "How do you come up with the right DTD?" The answer is: "Through document analysis."

Many organizations have been doing document analysis for years. The difference between these efforts and designing a DTD is the end goal of the exercise. Historically, analysis has served the efforts to create an editorial style guide, to establish a uniform "look & feel" for an organization's publications. It has dealt with questions about page size, type face, font families, etc. There are still many companies where significant effort goes into this exercise.

With DTDs, the focus of the effort expands to incorporate the identity, role, organization and structure of the content. The effort involved in document analysis and DTD development, while it can be challenging, is worthwhile because of what you learn about your content and your current systems for

producing it. It is also worthwhile because it is where you pull your experts together and identify the important metadata that only they can add to the documents.

How you go about doing this is not necessarily obvious. You can't simply have your data architect sit down and write what he or she thinks the DTD ought to be. You need to involve the participants in the information system in its design and execution because only they can state the requirements of their systems, the terminology of their expertise, and the identity of the critical metadata. The role of your document architect is to be a facilitator, a catalyst, and then ultimately the documenter of the results.

Document architects are not in great supply; it is a relatively new business role. To develop the architectural expertise of internal staff (assuming you want to develop and keep that expertise in-house — a wise idea), they need to learn more than just the technical specifics of SGML. They also need to learn fundamental principles and guidelines and benefit from years of existing practice, and they need a methodology for running design workshops with end users and documenting the results. Fortunately, these two books provide the expertise: David Megginson's *Structuring XML Documents* gives you the one; Eve Maler and Jeanne Andaloussi's *Developing SGML DTDs* gives you the other.

Structuring XML Documents by David Megginson

At first glance, the title of David Megginson's book, *Structuring XML Documents*, sounds like it is about coding, like the almost unlimited supply of books on how to write HTML documents. In fact, other reviews have made precisely that complaint – that the book is all about this DTD thing with nothing about tagging. But Megginson is not out to tell you how to code XML; his purpose is to provide you with the background to guide the DTD design effort.

Megginson's focus is on setting down fundamental principles. He gives you a framework for articulating the basic objectives for your system, and then shows you how to realize them by looking at the DTD design from three perspectives:

1. How easy is it for authors to learn the DTD?
2. How effective is the DTD for producing the content?
3. How well can other information-processing specialists use the resulting data?

Principles may seem like frills or luxuries when you are planning a DTD development project, but their intrinsic importance will become obvious the first time you are in front of a room full of experts, each with different agendas, trying to reach consensus on some basic design question. As he says on page 99, "DTD writing is not for the faint of heart." Once the people who create or use

information start to understand the fundamental power of SGML and XML, they start to care deeply about how the DTD is designed. As Megginson makes clear, DTD design inevitably requires tradeoffs and choices. Principles that all accept help decision-making enormously.

Writing DTDs is not a science, but an art. Megginson points out that often, the principles that make a DTD easier to learn—relatively few logical units, internal consistency, and names that are intuitive to the authors—conflict with the principles that make it most useful—minimal mental effort needed to choose among logical units, and flexibility and completeness to deal withall information requirements. This is especially true when the DTD will be used across a number of authoring groups. To some extent, this can be ameliorated by the SGML editing tool used or by developing DTDs for authoring. Nevertheless, there are choices to be made and the guidelines in Megginson's book will help you make the best ones available.

The book assumes that you have a basic knowledge of SGML and know how to collect business information from clients. (There is a basic introduction to DTD syntax in the first chapters of the book, but you can safely skip over those if you are already familiar with it.) You will want to find that from other sources, if either is an area that needs development. However, there are now more than enough books available on the market to fill in those gaps (and "Developing SGML DTDs" in particular addresses the second issue) and his book is stronger for staying focused on his key objectives.

While XML may seem like a new technology to many of this book's readers, the SGML community has been developing DTDs for over a decade. There is a lot of practical experience in place in a number of well-exercised industry-standard DTDs. Ideally, you would want your document architects to profit from those years of experience. Megginson provides a big dose of that experience by including five major DTDs in his book: *ISO 12083* (four general-purpose DTDs for electronic publishing), *DocBook* (a book-oriented DTD for technical manuals), *TEI-Lite* (a subset of the full Text Encoding Initiative DTD, an extremely rich structure designed to support research into scholarly texts like plays, poems, and lexicographical texts), *MIL-STD-38784* (the CALS DTD for technical documentation accompanying military weapons systems), and *HTML 4.0*. He starts off with a general overview of each of these DTDs, then uses them throughout to illustrate the points made in the book. While you are unlikely to use any of these DTDs as is, this makes each of his key points more concrete, and gives you a basis for deciding how to solve particular problems.

The third and forth parts of the book deal with advanced topics like dealing with document fragments and subdocuments, developing architectural forms, etc. These present important material, but they are topics that you can pick and choose from as you find the material important to what you are doing. The one

topic I would have liked to see more fully addressed is DTD maintenance. From experience, I can tell you that DTDs are always, to some degree or another, in flux, not only during their development, but throughout their subsequent life. Just as database schemas will change to meet changing requirements and new product objectives, you will need to change your DTDs over time. In the early days of your system, change is relatively manageable because you probably know everyone who is interested in your data personally. Once content starts to be used across your organization, you'll need to move to a formal process for collecting, testing, and broadcasting changes. People will need to be notified of pending changes in advance, so that they can test their systems and alert you to any potentially negative consequences of the new design. (Our email list for testing and announcing DTD changes has over 100 names and is still growing.) This is a whole topic by itself, and Megginson's book could have presented the basic issues to consider.

But that is a minor concern. David Megginson's book teaches you how to think about designing DTDs—and by extension, how to think about the related issues of creating documents using those DTDs. It is a resource you will want to read and keep at hand throughout the process of developing DTDs that your organization can, and will, use.

Developing SGML DTDs by Eve Maler and Jeanne El Andaloussi

Once you have the guidelines and policies for developing your DTDs, you are going to need procedures for the development project itself. You are about to invest time and energy in documenting how your authors and other users think about the DTDs you produce. You are going to look at how your documents are supposed to be constructed and how they really *are* constructed (almost guaranteed to be two different things), and you will ask people to decide how they *ought* to be constructed (yet a third). Discussions, both cool and passionate, are going to happen and decisions are going to be made every day. This is not the sort of stuff you want to jot down on the backs of envelopes. You want a history and a paper trail to refer to throughout the project and after its completion.

Structuring XML Documents gave you a roadmap, but little guidance on how to actually cross the terrain. In *Developing SGML DTDs From Text to Model to Markup*, Eve Maler and Jeanne El Andaloussi give you more than enough process and procedure to organize and carry out the development project and live to tell the tale.

Maler and Andaloussi have earned their stripes on the ground and their book is built on practical experience. Lots of it. Both authors have been heavily involved in numerous DTD development initiatives, both inside their respective

companies, and in various industry organizations. They were both heavily involved in precursors to the DocBook DTD, and Maler is one of the prime architects of the XML standard itself. The methods they describe are the result of plenty of trial and error and that experience shows in the tone and the construction of the book.

Like Megginson's book, *Structuring DTDs* begins with an introduction to DTD syntax. Again, if this is material you are familiar with, you can skip over it. (On the other hand, the book also includes a "Quick Reference" in the back that is concise and extremely useful.)

However, their overview of the DTD development process, project management and, in particular, considerations in selecting the teams and handling the politics are critical reading. Don't be tempted to skip it in favor of the "good stuff" – the sample forms, diagraming methods and the like. The steps they recommend you follow to organize the project will help you avoid pitfalls and traps that can cripple or kill the project. The process they recommend will help ensure that the project succeeds. Maler and Andaloussi don't hesitate to suggest that you protect your flank and to avoid the project entirely if it is not adequately supported. (And the questions they give you will let you know for sure whether the project is truly supported or not.) It may not be politically correct to state

> … do not invest in a DTD development project unless you have proof that it … has been officially acknowledged as necessary … and that the necessary means (attention, human resources, budget, and time) have been allocated through the end of the project.

but it is advice that could save you a great deal of angst, and in fact can help you make the case for adequate resources to your management.

Most of the book concentrates on the actual mechanics of developing a DTD. The second part of the book, *Document Type Design,* provides procedures for preparing for and running design workshops, identifying and modeling semantic components, selecting which of those components become actual elements or attributes, and for documenting the process. They provide sample forms and questions that you can use or adapt to your purposes. Don't expect to put everything they say into practice; to follow the entire process in all its detail is probably overkill for most projects. Instead, focus on finding those pieces that best fit the scope and requirements of your project. Your project will run the smoother for it.

The authors use a recipe book as their sample application throughout. While the problem of modeling recipes may seem simplistic at first, it suits their purposes well. It is a document type that almost everyone can relate to and it quickly proves to be far more complex than it seems. If you work in an organization where people's first reaction to the document analysis effort is "hey,

we know this stuff already; this is going to be a waste of time," you will appreciate the lessons of the recipe model. Your users will also quickly begin to realize that their information is not nearly as clear cut as they had thought it was, and you will find that Maler and Andaloussi have anticipated in their book the types of "aha" experiences through which you will guide your teams.

Like Megginson, Maler and Andaloussi do not spend as much time on maintenance as I would like. However, they do provide more specific technical advice, particularly on the topics of making your DTDs readable and setting up bug tracking and reporting systems. They also tackle the subjects of DTD documentation and training. These are two areas that often fall through the cracks during system development (and not in SGML or XML systems alone – it's a chronic oversight in application development in general). They give suggestions for the types of documentation and training that will be most useful, and clearly explain its importance to the success of your system over the long run.

Structuring DTDs was written before the XML standard, so it doesn't distinguish between the structures that are valid in both SGML and XML and those that are prohibited in XML. Keep that in mind as you read the book. Certain types of constructs that they discuss—minimizing end tags or content model exceptions—are not valid in XML DTDs. However, this should not pose any problems. The differences between SGML and XML are well documented elsewhere and the book does not espouse any particular way of writing the DTD itself.

Conclusion

SGML and XML can take your document development and delivery systems to an entirely new level. You can cost-effectively develop products for your customers that you could only dream about before. Your ability to define and then use your own markup languages, designed to meet your own exacting requirements and standards, puts new power into your content. You no longer have to wait on vendors to invent the future for you. You can invent it for yourself.

Historically, what we as content owners could do with our documents has been determined by what products we bought, and by the cost of crafting our content into the form each product required. Markup languages offer us the opportunity to reverse that relationship. SGML and XML give us the power to develop rich data, richer, in some ways, that any single product is capable of using. Far from avoiding DTDs, we should be glad that such a powerful and general purpose mechanism for creating reliable narrative data has been created and we should jump on the opportunity it gives us to make the most of the information we invest in every day.