Chapter 1

Why XML?

In which it is revealed where my personal experience of markup languages began.

In this chapter, I take you through some of my initial experiences with markup languages, experiences that led me to be such an advocate of information standards in general and markup languages in particular. We discuss a simple example of the power of markup, and throughout the chapter, I cover some basic definitions and concepts.

The Lesson of SGML

In early 1995, I helped start a company, E-Doc, with a subversive business plan based on the premise that big publishing companies (in this case, in the scientific-technical-medical arena) might want to publish on the World Wide Web. I say "subversive" because at the time it was just that—the very companies we were targeting with our services were the old guard of the publishing world, and they had every reason in the world to suppress and reject these new technologies. A revolution was already occurring, especially in the world

of scientific publishing. Through the Internet, scientists were beginning to share papers with other scientists. While the publishing companies weren't embracing this new medium, the scientists themselves were, and in the process they were bypassing traditional journal publication entirely and threatening decades of entrenched academic practice. Remember, the Internet wasn't seen as a viable commercial medium back then; it was largely used by academics, although we were starting to hear about the so-called "information superhighway." Despite the assurance of all my friends that I was off my rocker, I left my secure career in the client/server software industry to follow my nose into the unknown. In my two years at E-Doc, I learned a great deal about technology, media, business, and the publishing industry, but one lesson that stands out is the power of SGML.

An international standard since 1986, SGML (Standard Generalized Markup Language) is the foundation on which modern markup languages (such as HTML or Hypertext Markup Language, the language of the Web) are based. SGML defines a structure through which markup languages can be built. HTML is a flavor of SGML, but it is only one markup language (and not even a particularly complex one) that derives from SGML. Since its inception, SGML has been in use in publishing, as well as in industry and governments throughout the world.

Because many of the companies we were dealing with at E-Doc had been using flavors of SGML to encode material such as books and journal articles since the late 1980s, they had developed vast storehouses of SGML data that was just waiting for the Internet revolution. Setting up full-text Web publishing systems became a matter of simply translating these already existing SGML files. It's not that the decision makers at these companies were so forward-thinking that they knew a global network that would redefine the way we think about information would soon develop. The lesson of SGML was precisely that these decision makers did not know what the future would hold. Using SGML "future-proofed" their data so that when the Web came around, they could easily repurpose it for their changing needs.

It's been a wild ride over the past six years, but as we begin a new century and a new millennium, that idea of future-proofing data seems more potent and relevant than ever. The publishing industry will continue to transform and accelerate into new areas, new platforms, and new paradigms. As technology

professionals, we have to start thinking about future-proofing now, while we're still at the beginning of this revolution.

What About XML?

So what do SGML and the Internet revolution have to do with XML? Let me tell you a secret: XML is just SGML wearing a funny hat; XML is SGML with a sexy name. In other words, XML is an evolution of SGML. The problem with SGML is that it takes an information management professional to understand it. XML represents an attempt to simplify SGML to a level where it can be used widely. The result is a simplified version of SGML that contains all the pieces of SGML that people were using anyway. Therefore, XML can help anyone future-proof content against future uses, whatever those might be.

That's power, baby!

Why HTML Is Not the Answer

I hear you saying to yourself, "Ah, Dan, but what about HTML? I can use HTML for managing information, and I get Web publishing for free (because HTML is the language of the Web). Isn't HTML also derived from SGML, and isn't it also a great, standardized way of storing documents?" Well, yes on one, no on two. HTML is wonderful, but for all its beauty, HTML is really good only at describing layout—it's a display-oriented markup. Using HTML, you can make a word **bold** or *italic*, but as to the reason that word might be bold or italic, HTML remains mute. With XML, because you define the markup you want to use in your documents, you can mark a certain word as a person's name or the title of a book. When the document is *represented*, the word will appear bold or italic; but with XML, because your documents know all the locations of people's names or book titles, you can capriciously decide that you want to underline book titles across the board. You have to make this change only once, wherever your XML documents are being represented. And that's just the beginning. Your documents are magically transformed from a bunch of relatively dumb HTML files to documents with intelligence, documents with muscle.

If I hadn't already learned this lesson, I learned it again when migrating TheStreet.com (the online financial news service that I referred to in the Introduction) from a relatively dumb HTML-based publishing system to a relatively smart XML-based content management system. When I joined TheStreet.com, it had been running for over two years with archived content (articles) that needed to be migrated to the new system. This mass of content was stored only as HTML files on disk. A certain company (which shall remain nameless) had built the old system, apparently assuming that no one would ever have to do anything with this data in the future besides spit it out in exactly the same format. With a lot of Perl (then the lingua franca of programming languages for the Web and an excellent tool for writing data translation scripts) and one developer's hard-working and largely unrecognized efforts over the course of six months, we managed to get most of it converted to XML. Would it have been easier to start with a content management system built from the ground up for repurposing content? Undoubtedly!

If this tale doesn't motivate you sufficiently, consider the problem of the wireless applications market. Currently, wireless devices (such as mobile phones, Research In Motion's Blackberry pager, and the Palm VII wireless personal digital assistant) are springing up all over, and content providers are hot to trot out their content onto these devices. Each of these devices implements different markup languages. Many wireless devices use WML (Wireless Markup Language, the markup language component of WAP, Wireless Application Protocol), which is built on top of XML. Any content providers who are already working with XML are uniquely positioned to get their content onto these devices. Anyone who isn't is going to be left holding the bag.

So HTML or WML or whatever you like becomes an output format (the display-oriented markup) for our XML documents. In building a Web publishing system, display-oriented markup happens at the representation stage, the very last stage. When our XML document is represented, it is represented in HTML (either on the fly or in a batch mode). Thus HTML is a "representation" of the root XML document. Just as a music CD or tape is a representation of a master recording made with much more high-fidelity equipment, the display-oriented markup (HTML, WML, or whatever) is a representation for use by a consumer. As a consumer, you probably don't have an 18-track digital recording deck in your living room (or pocket). The CD or tape (or MP3 audio file, for that matter) is a representation of the original recording for you

to take with you. But the music publisher retains the original master recording so that when a new medium comes out (like Super Audio CD, for instance), the publisher can convert the high-quality master to this new format. In the case of XML, you retain your XML data forever in your database, but what you send to consumers is markup specific to their current needs.

The Basics of XML

If you know HTML already, then you're familiar with the idea of tagging content. Tags are interspersed with data to represent "metadata" or data about the data. Let's start with the following sentence:

> Homer's Odyssey is a revered relic of the ancient world.

Imagine you never heard of the *Odyssey* or Homer. I'll reprint the sentence like this:

> Homer's Odyssey is a revered relic of the ancient world.

I've added metadata that adds meaning to the sentence. Just by adding one underline, I've loaded the sentence with extra meaning. In HTML, this sentence would be marked up like this:

```
Homer's <u>Odyssey</u> is a revered relic of the ancient world.
```

This markup indicates that the word "Odyssey" is to appear underlined. As described in the last section, HTML is really good only at describing layout—a display-oriented markup. If you're interested only in how users are *viewing* your sentences, that's great. However, if you want to give your documents part of a system, so that they can be managed intelligently and the content within them can be searched, sorted, filed, and repurposed to meet your business needs, you need to know more about them. A human can read the sentence and logically infer that the word "Odyssey" is a book title because of the underline. The sentence contains metadata (that is, the underline), but it's ambiguous to a computer and decodable only by the human reader. Why? Because computers are *stupid!* If you want a computer to know that "Odyssey"

is a book title, you have to be much more explicit; this is where XML comes in. XML markup for the preceding sentence might be the following:

```
Homer's <book>Odyssey</book> is a revered relic of the ancient world.
```

Aha! Now we're getting somewhere. The document is marked up using a new tag, `<book>`, which I've made up just for this application, to indicate where book titles are referenced. This provides two important and powerful tools: You can centrally control the style of your documents, and you have machine-readable metadata—that is, a computer can easily examine your document and tell you where the references to book titles are. You can then choose to style the occurrences of book titles however you want—with underlines, in italics, in bold, with quotes around them, in a different color, whatever.

Let's say you want every book title you mention to be a hyperlink to a page that enables you to buy the book. The HTML markup would look something like this:

```
Homer's <u><a href="http://some.store.com/buybook.cgi?ISBN=0987-
2343">Odyssey</a></u> is a revered relic of the ancient world.
```

In this example, you've *hard-coded* the document with a specific Uniform Resource Locator (URL) to a script on some online bookstore somewhere. What if that bookstore goes out of business? What if you make a strategic partnership with some other online bookstore and you want to change all the book titles to point to that store's pages? Then you've got to go through all of your documents with some kind of half-baked Perl script. What if your documents aren't all coded consistently? There are about a hundred things that can and will go wrong in this scenario. Believe me—I've been there.

Let's look at XML markup of the same sentence:

```
Homer's <book isbn="0987-2343">Odyssey</book> is a revered relic of the
ancient world.
```

Now isn't that a breath of fresh air? By replacing the hard-coded script reference with a simple indication of ISBN (International Standard Book Number,

a guaranteed unique number for every book printed[1]), you've cut the complexity of markup in half. In addition, you have enabled centralized control over whether book titles should be links and, if so, where they link. Assuming central control of how XML documents are turned into display-oriented markup, you can make a change in this one place to effect the display of many documents. As a special bonus, if you store all your XML documents in a database and properly *decompose*, or extract, the information within them (as we'll discuss next), you can also find out which book titles are referred to from which documents.

Why You Don't Need to Throw Away Your RDBMS

People often come up to me on the street and say, "Tell me, Dan, if I decide to build XML-based systems, what happens to my relational database?" A common misconception is that XML, as a new way of thinking about and representing data, means an end to the relational database management system (RDBMS) as we know it. Well, don't throw away your relational database just yet. XML is a way to format and bring order to data. By mating the power of XML with the immense and already well-understood power of SQL-based relational database systems, you get the best of both worlds. In the following chapters, I'll discuss some approaches to building this bridge between XML and your good old relational database.

Relational databases are great at some things (such as maintaining data integrity and storing highly structured data), while XML is great at other things (for example, formatting data for transmission, representing unstructured data, and ordering data). Using both XML and SQL (Structured Query Language) together enables you to use the best parts of both systems to create robust, data-centric systems. Together, XML and relational databases help you answer the fundamental question of content management and of data-oriented systems in general. That question is "*What do I have?*" Once you know what you have, you can do anything. If you don't know what you have, you

[1] I realize that Homer's *Odyssey* has been reprinted thousands of times in many languages by different publishers and that all of the modern reprintings have their own ISBNs. This is simply an example.

essentially don't have anything. You'll see this question restated throughout this book in different ways.

A Brief Example

For convenience, let's say that I want to keep track of books by ISBN. ISBNs are convenient because they provide a unique numbering scheme for books. Let's take the previous example of the book references marked up by ISBN:

```
<document id="1">Homer's <book isbn="0987-2343">Odyssey</book> is a revered
relic of the ancient world.</document>
```

I've added `<document id="1">` and `</document>` tags around the body of my document so each document can uniquely identify itself. Each XML document I write has an ID number, which I've designated should be in a tag named "document" that wraps around the entire document. Again, remember that I'm just making these tags up. They're not a documented standard; they're just being used for the purpose of these examples.

For easy reference, I want to keep track of which ISBN numbers, are referred to from which documents; thus I design an SQL table to look something like this:

doc_id	ISBN
1	0987-2343
2	0872-8237

`doc_id` has referential integrity to a list of valid document ID numbers, and the `isbn` field has referential integrity to a list of valid ISBN numbers. "Great," I hear you saying, "this is a lot of complexity for a bunch of stupid book names. Explain to me *why* this is better than using HTML again."

Suppose I have a thousand documents (book reviews, articles, bulletin board messages, and so on), and I want to determine which of them refer to a spe-

cific book. In the HTML universe, I can perform a textual search for occurrences of the book name. But what if I have documents that refer to Homer's *Odyssey* and Arthur C. Clark's *2001: A Space Odyssey*? If I search for the word "odyssey," my search results list both books. However, if I've marked up all my references to books by ISBN and I've decomposed or extracted this information into a table in a database, I can use a *simple SQL query* to get the information I need quickly and reliably:

```
select doc_id from doc_isbn where isbn = '0987-2343'
```

The search results are a set of document ID numbers. I can choose to display the title of each document as a hyperlink, clickable to the actual document, or I can concatenate the documents and display them to the user one after another on a page—whatever the requirements of my application. By combining the power of XML machine-readable metadata with the simplicity and power of my relational database, I've created a powerful document retrieval tool that can answer the question, "What do I have?" Creating such a tool simply required a little forethought and designing skill.

If I'm going too fast for you, don't worry. I discuss these topics in detail in the following chapters.

Great! How Do I Get Started?

The four essential steps to building an XML-based system or application are the following:

1. Requirements gathering (described in Chapter 3)
2. Abstract data modeling (Chapter 4)
3. Application design, including DTD (document type definition) and schema design (Chapters 5 and 6)
4. Implementation (Chapters 8 and 9)

If you follow this plan, you won't write one line of application code until step 4. Building an XML-based application is writing software and requires the same rigorous approach.

The four steps don't mention platform at all. Are we implementing on UNIX or Windows NT? Oracle or MySql? Java or Perl? XML and database design free you from platform-dependent approaches to data storage and manipulation, so take advantage of that freedom, and don't even choose a platform until at least midway through step 2. Base that platform decision on what features it includes to get you closer to your goal—built-in features that fit into your business requirements—how easy the platform is to support ongoing operations (operational considerations).

You can incorporate the same methodology when integrating XML into an existing RDBMS-based application. Throughout the following chapters, we'll examine how to build an XML-based application. You'll learn how to collect your requirements, build an abstract data model around these requirements, and then build an XML DTD and a relational schema around this data model. We'll get into implementation only in the abstract, describing how your system must interact with the DTD and schema.

Summary

If I've done my job, you're excited about the raw potential of XML now. You've seen how it can work to turn dumb documents into smart documents—documents with oomph. You should understand where some of my passion for these systems comes from. I've seen them work and work well. In the next chapter, we'll step back into the history of both XML and the relational database to provide a bit more context before moving forward with application design and development.