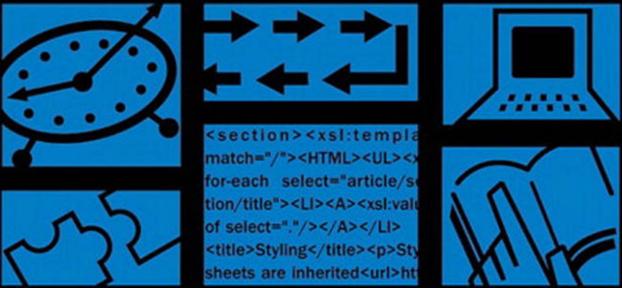
XML

BY EXAMPLE



```
<section><xsl:templa
match="/"><HTML><UL><x
for-each  select="article/se
tion/title"><LI><A><xsl:valu
of select="."/></A></LI>
<title>Styling</title><p>Sty
sheets are inherited<url>ht
```

QUE®

Benoit Marchal

XML BY EXAMPLE

que®
201 West 103rd Street
Indianapolis, Indiana 46290

Benoît Marchal

XML by Example

International Standard Book Number: 0-7897-2242-9

Library of Congress Catalog Card Number: 99-66449

Printed in the United States of America

First Printing: December 1999

01 00 4

Trademarks

Warning and Disclaimer

Publisher
John Pierce

Acquisitions Editor
Todd Green

Development Editor
Susan Hobbs

Technical Editor
Karl Fast

Managing Editor
Thomas F. Hayes

Project Editor
Karen S. Shields

Copy Editor
Sossity Smith

Indexer
Tina Trettin

Proofreader
Maribeth Echard

Team Coordinator
Julie Otto

Media Developer
Jay Payne

Interior Designer
Karen Ruggles

Cover Designer
Rader Design

Copy Writer
Eric Borgert

Production
Louis Porter Jr.

Contents at a Glance

Table of Contents

Dedication

To Pascale for her never-failing trust and patience.

Acknowledgments

This book is an important station on a long journey. I would like to thank all the people who have helped me and trusted me along the way. In chronological order, Ph. Capelle, who helped a confused student; Ph. van Bastelaer and J. Berge, who were curious about SGML; H. Karunaratne and K. Kaur and the folks at Sitpro, who showed me London; S. Vincent, who suggested I get serious about writing; V. D'Haeyere, who taught me everything about the Internet; Ph. Vanhoolandt, who published my first article; M. Gonzalez, N. Hada, T. Nakamura, and the folks at Digital Cats, who published my first U.S. papers; S. McLoughlin, who helps with the newsletter; and T. Green, who trusted me with this book.

Thanks the XML/EDI Group and, in particular, M. Bryan, A. Kotok, B. Peat, and D. Webber.

Special thanks to my mother for making me curious.

Writing a book is a demanding task, both for a business and for a family. Thanks to my customers for understanding and patience when I was late. Special thanks to Pascale for not only showing understanding, but also for encouraging me!

About the Author

Benoît Marchal runs the consulting company, Pineapplesoft, which specializes in Internet applications, particularly e-commerce, XML, and Java. He has worked with major players in Internet development such as Netscape and EarthWeb, and is a regular contributor to `developer.com` and other Internet publications.

In 1997, he cofounded the XML/EDI Group, a think tank that promotes the use of XML in e-commerce applications. Benoît frequently leads corporate training on XML and other Internet technologies. You can reach him at `bmarchal@pineapplesoft.com`.

Tell Us What You Think!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As a Publisher for Que, I welcome your comments. You can fax, email, or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or fax number. I will carefully review your comments and share them with the author and editors who worked on the book.

Fax:	317-581-4666
Email:	`que.programming@macmillanusa.com`
Mail:	John Pierce Publisher Que-Programming 201 West 103rd Street Indianapolis, IN 46290 USA

This page intentionally left blank

Introduction

The *by Example* Series

How does the *by Example* series make you a better programmer? The *by Example* series teaches programming using the best method possible. After a concept is introduced, you'll see one or more examples of that concept in use. The text acts as a mentor by figuratively looking over your shoulder and showing you new ways to use the concepts you just learned. The examples are numerous. While the material is still fresh, you see example after example demonstrating the way you use the material you just learned.

The philosophy of the *by Example* series is simple: The best way to teach computer programming is using multiple examples. Command descriptions, format syntax, and language references are not enough to teach a newcomer a programming language. Only by looking at many examples in which new commands are immediately used and by running sample programs can programming students get more than just a feel for the language.

Who Should Use This Book

XML by Example is intended for people with some basic HTML coding experience. If you can write a simple HTML page and if you know the main tags (such as `<P>`, `<TITLE>`, `<H1>`), you know enough HTML to understand this book. You don't need to be an expert, however.

Some advanced techniques introduced in the second half of the book (Chapter 7 and later) require experience with scripting and JavaScript. You need to understand loops, variables, functions, and objects for these chapters. Remember these are advanced techniques, so even if you are not yet a JavaScript wizard, you can pick up many valuable techniques in the book.

This book is for you if one of the following statements is true:

- You are an HTML whiz and want to move to the next level in Internet publishing.

- You publish a large or dynamic document base on the Web, on CD-ROM, in print, or by using a combination of these media, and you have heard XML can simplify your publishing efforts.

- You are a Web developer, so you know Java, JavaScript, or CGI inside out, and you have heard that XML is simple and enables you to do many cool things.

- You are active in electronic commerce or in EDI and you want to learn what XML has to offer to your specialty.
- You use software from Microsoft, IBM, Oracle, Corel, Sun, or any of the other hundreds of companies that have added XML to their products, and you need to understand how to make the best of it.

You don't need to know anything about SGML (a precursor to XML) to understand *XML by Example*. You don't need to limit yourself to publishing; *XML by Example* introduces you to all applications of XML, including publishing and nonpublishing applications.

This Book's Organization

This book teaches you about XML, the eXtensible Markup Language. XML is a new markup language developed to overcome limitations in HTML.

XML exists because HTML was successful. Therefore, XML incorporates many successful features of HTML. XML also exists because HTML could not live up to new demands. Therefore, XML breaks new ground when it is appropriate.

This book takes a hands-on approach to XML. Ideas and concepts are introduced through real-world examples so that you not only read about the concepts but also see them applied. With the examples, you immediately see the benefits and the costs associated with XML.

As you will see, there are two classes of applications for XML: publishing and data exchange. Data exchange applications include most electronic commerce applications. This book draws most of its examples from data exchange applications because they are currently the most popular. However, it also includes a very comprehensive example of Web site publishing.

I made some assumptions about you. I suppose you are familiar with the Web, insofar as you can read, understand, and write basic HMTL pages as well as read and understand a simple JavaScript application. You don't have to be a master at HTML to learn XML. Nor do you need to be a guru of JavaScript.

Most of the code in this book is based on XML and XML style sheets. When programming was required, I used JavaScript as often as possible. JavaScript, however, was not appropriate for the final example so I turned to Java.

You don't need to know Java to understand this book, however, because there is very little Java involved (again, most of the code in the final example is XML). Appendix A, "Crash Course on Java," will teach you just enough Java to understand the examples.

Conventions Used in This Book

EXAMPLE

Examples are identified by the icon shown at the left of this sentence:

Listing and code appears in `monospace` font, such as

```
<?xml version="1.0"?>
```

NOTE
Special notes augment the material you read in each chapter. These notes clarify concepts and procedures.

TIP
You'll find numerous tips offering shortcuts and solutions to common problems.

CAUTION
The cautions warn you about pitfalls that sometimes appear when programming in XML. Reading the caution sections will save you time and trouble.

What's Next

XML was introduced to overcome the limitations of HTML. Although the two will likely coexist in the foreseeable future, the importance of XML will only increase. It is important that you learn the benefits and limitations of XML so that you can prepare for the evolution.

Please visit the *by Example* Web site for code examples or additional material associated with this book:

```
<http://www.quecorp.com/series/by_example/>
```

Turn to the next page and begin learning XML by examples today!

3

XML Schemas

In Chapter 2, “The XML Syntax,” you learned how to write and read XML documents. More importantly, you learned that XML emphasizes the structure of documents.

This chapter further develops that theme by looking at the DTD, short for *Document Type Definition*, a mechanism to describe the structure of documents. Specifically, you will learn how to

- model XML documents
- express the model in a DTD
- validate a document against its model

The DTD is the original modeling language or schema for XML. However, for historical reasons, the DTD is somewhat limited and people are looking for solutions to overcome these limitations. The W3C is working on an alternative to DTD. We will review the current status of that effort.

This chapter is probably the most abstract chapter in this book. You might want to temporarily skip the second half (starting from the section “Entities and Notations”) and revisit it after you have read through the book.

The DTD Syntax

EXAMPLE

The syntax for DTDs is different from the syntax for XML documents. Listing 3.1 is the address book introduced in Chapter 2 but with one difference: It has a new `<!DOCTYPE>` statement. The new statement is introduced in the section “Document Type Declaration.” For now, it suffices to say that it links the document file to the DTD file. Listing 3.2 is its DTD.

Listing 3.1: An Address Book in XML

```
<?xml version="1.0"?>
<!DOCTYPE address-book SYSTEM "address-book.dtd">
<!-- loosely inspired by vCard 3.0 -->
<address-book>
   <entry>
      <name>John Doe</name>
      <address>
         <street>34 Fountain Square Plaza</street>
         <region>OH</region>
         <postal-code>45202</postal-code>
         <locality>Cincinnati</locality>
         <country>US</country>
      </address>
      <tel preferred="true">513-555-8889</tel>
      <tel>513-555-7098</tel>
      <email href="mailto:jdoe@emailaholic.com"/>
   </entry>
   <entry>
      <name><fname>Jack</fname><lname>Smith</lname></name>
      <tel>513-555-3465</tel>
      <email href="mailto:jsmith@emailaholic.com"/>
   </entry>
</address-book>
```

Listing 3.2: The DTD for the Address Book

```
<!-- top-level element, the address book
     is a list of entries                    -->
<!ELEMENT address-book  (entry+)>

<!-- an entry is a name followed by
     addresses, phone numbers, etc.        -->
```

```
<!ELEMENT entry  (name,address*,tel*,fax*,email*)>

<!-- name is made of string, first name
     and last name. This is a very flexible
         model to accommodate exotic name       -->
<!ELEMENT name   (#PCDATA ¦ fname ¦ lname)*>
<!ELEMENT fname  (#PCDATA)>
<!ELEMENT lname  (#PCDATA)>

<!-- definition of the address structure
     if several addresses, the preferred
         attribute signals the "default" one   -->
<!ELEMENT address      (street,region?,postal-code,locality,country)>
<!ATTLIST address      preferred (true ¦ false)  "false">
<!ELEMENT street       (#PCDATA)>
<!ELEMENT region       (#PCDATA)>
<!ELEMENT postal-code  (#PCDATA)>
<!ELEMENT locality     (#PCDATA)>
<!ELEMENT country      (#PCDATA)>

<!-- phone, fax and email, same preferred
     attribute as address                    -->
<!ELEMENT tel       (#PCDATA)>
<!ATTLIST tel       preferred (true ¦ false)  "false">
<!ELEMENT fax       (#PCDATA)>
<!ATTLIST fax       preferred (true ¦ false)  "false">
<!ELEMENT email     EMPTY>
<!ATTLIST email     href  CDATA                #REQUIRED
                     preferred (true ¦ false)  "false">
```

Element Declaration

EXAMPLE

1. DTD is a mechanism to describe every object (element, attribute, and so on) that can appear in the document, starting with elements. The following is an example of element declaration:

```
<!ELEMENT address-book  (entry+)>
```

After the `<!ELEMENT` markup comes the element name followed by its *content model*. The element declaration is terminated with a right angle bracket.

Element declarations are easy to read: The right side (the content model) defines the left side (the element name). In other words, the content model lists the children that are acceptable in the element.

The previous declaration means that an `address-book` element contains one or more `entry` elements. `address-book` is on the left side, `entry` on the right. The plus sign after `entry` means there can be more than one `entry` element.

2. Parentheses are used to group elements in the content model, as in the following example:

EXAMPLE

```
<!ELEMENT name (lname, (fname ¦ title))>
```

Element Name

As we saw in Chapter 2, XML names must follow certain rules. Specifically, names must start with either a letter or a limited set of punctuation characters (“`_`”,“`:`”). The rest of the name can consist of the same characters plus letters, digits and new punctuation characters (“`.`”, “`-`”). Spaces are not allowed in names.

Names cannot start with the string “xml,” and as we will see in Chapter 4, “Namespaces,” the colon plays a special role so it is advised you don’t use it.

Special Keywords

For most elements, the content model is a list of elements. It also can be one of the following keywords:

- `#PCDATA` stands for parsed character data and means the element can contain text. `#PCDATA` is often (but not always) used for leaf elements. Leaf elements are elements that have no child elements.
- `EMPTY` means the element is an empty element. `EMPTY` always indicates a leaf element.
- `ANY` means the element can contain any other element declared in the DTD. This is seldom used because it carries almost no structure information. ANY is sometimes used during the development of a DTD, before a precise rule has been written. Note that the elements must be declared in the DTD.

Element contents that have `#PCDATA` are said to be *mixed content*. Element contents that contain only elements are said to be *element content*. In Listing 3.2, `tel` is a leaf element that contains only text while `email` is an empty element:

```
<!ELEMENT tel    (#PCDATA)>

<!ELEMENT email  EMPTY>
```

EXAMPLE

Note that CDATA sections appear anywhere `#PCDATA` appears.

The Secret of Plus, Star, and Question Mark

The plus (“+”), star (“*”), and question mark (“?”) characters in the element content are *occurrence indicators*. They indicate whether and how elements in the child list can repeat.

- An element followed by no occurrence indicator must appear once and only once in the element being defined.
- An element followed by a “+” character must appear one or several times in the element being defined. The element can repeat.
- An element followed by a “*” character can appear zero or more times in the element being defined. The element is optional but, if it is included, it can repeat indefinitely.
- An element followed by a “?” character can appear once or not at all in the element being defined. It indicates the element is optional and, if included, cannot repeat.

EXAMPLE

The `entry` and `name` elements have content model that uses an occurrence indicator:

```
<!ELEMENT entry    (name,address*,tel*,fax*,email*)>
<!ELEMENT address  (street,region?,postal-code,locality,country)>
```

Acceptable children for the `entry` are `name`, `address`, `tel`, `fax`, and `email`. Except for `name`, these children are optional and can repeat.

Acceptable children for `address` are `street`, `region`, `postal-code`, `locality`, and `country`. None of the children can repeat but the `region` is optional.

The Secret of Comma and Vertical Bar

The comma (“,”) and vertical bar (“¦”) characters are *connectors*. Connectors separate the children in the content model, they indicate the order in which the children can appear. The connectors are

- the “,” character, which means both elements on the right and the left of the comma must appear in the same order in the document.
- the “¦” character, which means that only one of the elements on the left or the right of the vertical bar must appear in the document.

EXAMPLE

The `name` and `address` elements are good examples of connectors.

```
<!ELEMENT name     (#PCDATA ¦ fname ¦ lname)*>
<!ELEMENT address  (street,region?,postal-code,locality,country)>
```

Acceptable children for `name` are `#PCDATA` or a `fname` element or a `lname` element. Note that it is one or the other. However, the whole model can repeat thanks to the "`*`" occurrence indicator.

Acceptable children for `address` are `street`, `region`, `postal-code`, `locality`, and `country`, in exactly that order.

The various components of mixed content must always be separated by a "`¦`" and the model must repeat. The following definition is incorrect:

```
<!ELEMENT name  (#PCDATA, fname, lname)>
```

It must be

```
<!ELEMENT name (#PCDATA ¦ fname ¦ lname)*>
```

Element Content and Indenting

In the previous chapter, you learned that the XML application ignores indenting in most cases. Here again, a DTD can help.

If a DTD is associated with the document, then the XML processor knows that spaces in an element that has element content must indent (because the element has element content, it cannot contain any text). The XML processor can label the spaces as *ignorable whitespaces*. This is a very powerful hint to the application that the spaces are indenting.

Nonambiguous Model

The content model must be *deterministic* or unambiguous. In plain English, it means that it is possible to decide which part of the model applies to the current element by looking only at the current element.

EXAMPLE

For example, the following model is not acceptable:

```
<!ELEMENT cover  ((title, author) ¦ (title, subtitle))>
```

because when the XML processor is reading the element

```
<title>XML by Example</title>
```

in

```
<cover><title>XML by Example</title><author>Benoît Marchal</author></cover>
```

it cannot decide whether the `title` element is part of `(title, author)` or of `(title, subtitle)` by looking at `title` only. To decide that `title` is part of `(title, author)`, it needs to look past `title` to the `author` element.

In most cases, however, it is possible to reorganize the document so that the model becomes acceptable:

```
<!ELEMENT cover  (title, (author ¦ subtitle))>
```

Now when the processor sees `title`, it knows where it fits in the model.

Attributes

EXAMPLE

Attributes also must be declared in the DTD. Element attributes are declared with the ATTLIST declaration, for example:

```
<!ATTLIST tel  preferred (true ¦ false)  "false">
```

The various components in this declaration are the markup (`<!ATTLIST`), the element name (`tel`), the attribute name (`preferred`), the attribute type (`(true ¦ false)`), a default value (“`false`”), and the right angle bracket.

EXAMPLE

For elements that have more than one attribute, you can group the declarations. For example, email has two attributes:

```
<!ATTLIST email  href      CDATA           #REQUIRED
                  preferred (true ¦ false)  "false">
```

Attribute declaration can appear anywhere in the DTD. For readability, it is best to list attributes immediately after the element declaration.

CAUTION

If used in a valid document, the special attributes `xml:space` and `xml:lang` must be declared as

```
xml:space (default¦preserve) "preserve"
xml:lang  NMTOKEN  #IMPLIED
```

EXAMPLE

The DTD provides more control over the content of attributes than over the content of elements. Attributes are broadly divided into three categories:

- string attributes contain text, for example:
  ```
  <!ATTLIST email  href CDATA #REQUIRED>
  ```
- tokenized attributes have constraints on the content of the attribute, for example:
  ```
  <!ATTLIST entry id ID #IMPLIED>
  ```
- enumerated-type attributes accept one value in a list, for example:
  ```
  <!ATTLIST entry preferred (true ¦ false)  "false">
  ```

Attribute types can take any of the following values:

- `CDATA` for string attributes.
- `ID` for identifier. An identifier is a name that is unique in the document.
- `IDREF` must be the value of an `ID` used elsewhere in the same document. IDREF is used to create links within a document.
- `IDREFS` is a list of `IDREF` separated by spaces.

- `ENTITY` must be the name of an external entity; this is how you assign an external entity to an attribute.
- `ENTITIES` is a list of `ENTITY` separated by spaces.
- `NMTOKEN` is essentially a word without spaces.
- `NMTOKENS` is a list of `NMTOKEN` separated by spaces.
- Enumerated-type list is a closed list of `nmtokens` separated by ¦, the value has to be one of the `nmtokens`. The list of tokens can further be limited to `NOTATION`s (introduced in the section "Notation," later in this chapter).

Optionally, the DTD can specify a default value for the attribute. If the document does not include the attribute, it is assumed to have the default value. The default value can take one of the four following values:

- `#REQUIRED` means that a value must be provided in the document
- `#IMPLIED` means that if no value is provided, the application must use its own default
- `#FIXED` followed by a value means that attribute value must be the value declared in the DTD
- A literal value means that the attribute will take this value if no value is given in the document.

EXAMPLE

NOTE

Information that remains constant between documents is an ideal candidate for `#FIXED` attributes. For example, if prices are always given in dollars, you could declare a price element as

```
<!ELEMENT price (#PCDATA)>
<!ATTLIST price currency NMTOKEN #FIXED "usd">
```

When the application reads

```
<price>19.99</price>
```

in a document, it appears as though it reads

```
<price currency="usd">19.99</price>
```

The application has received additional information but it didn't require additional markup in the document!

Document Type Declaration

EXAMPLE

The *document type declaration* attaches a DTD to a document. Don't confuse the document type declaration with the document type definition (DTD). The document type declaration has the form:

```
<!DOCTYPE address-book SYSTEM "address-book.dtd">
```

It consists of markup (`<!DOCTYPE`), the name of the top-level element (`address-book`), the DTD (`SYSTEM` “`address-book.dtd`”) and a right angle bracket. As Listing 3.1 illustrates, the document type declaration appears at the beginning of the XML document, after the XML declaration.

EXAMPLE

The top-level element of the document is selected in the declaration. Therefore, it is possible to create a document starting with any element in the DTD. Listing 3.3 has the same DTD as Listing 3.1, but its top-level element is an `entry`.

Listing 3.3: An Entry

```
<?xml version="1.0"?>
<!DOCTYPE entry SYSTEM "address-book.dtd">
<entry>
   <name>John Doe</name>
   <address>
      <street>34 Fountain Square Plaza</street>
      <region>OH</region>
      <postal-code>45202</postal-code>
      <locality>Cincinnati</locality>
      <country>US</country>
   </address>
   <tel preferred="true">513-555-8889</tel>
   <tel>513-555-7098</tel>
   <email href="mailto:jdoe@emailaholic.com"/>
</entry>
```

Internal and External Subsets

The DTD is divided into internal and external subsets. As the name implies, the internal subset is inserted in the document itself, whereas the external subset points to an external entity.

The internal and the external subsets have different rules for parameter entities. The differences are explained in the section “General and Parameter Entities,” later in this chapter.

The internal subset of the DTD is included between brackets in the document type declaration. The external subset is stored in a separate entity and referenced from the document type declaration.

The internal subset of a DTD is stored in the document, specifically in the document type declaration, as in

```
<!DOCTYPE address [
<!ELEMENT address  (street,region?,postal-code,locality,country)>
<!ATTLIST address  preferred (true ¦ false)  "false">

<!ELEMENT street       (#PCDATA)>
<!ELEMENT region       (#PCDATA)>
<!ELEMENT postal-code  (#PCDATA)>
<!ELEMENT locality     (#PCDATA)>
<!ELEMENT country      (#PCDATA)>
]>
```

The external subset is not stored in the document. It is referenced from the document type declaration through an identifier as in the following examples:

```
<!DOCTYPE address-book SYSTEM "http://www.xmli.com/dtd/address-book.dtd">


<!DOCTYPE address-book PUBLIC "-//Pineapplesoft//Address Book//EN"
➥"http://catwoman.pineapplesoft.com/dtd/address-book.dtd">


<!DOCTYPE address-book SYSTEM "../dtds/address-book.dtd">
```

There are two types of identifiers: *system identifiers* and *public identifiers*. A keyword, respectively SYSTEM and PUBLIC, indicates the type of identifier.

- A system identifier is a *Universal Resource Identifier* (URI) pointing to the DTD. URI is a superset of URLs. For all practical purposes, a URI is a URL.
- In addition to the system identifier, the DTD identifier might include a public identifier. A public identifier points to a DTD recorded with the ISO according to the rules of ISO 9070. Note that a system identifier must follow the public identifier.

The system identifier is easy to understand. The XML processor must download the document from the URI.

Public identifiers are used to manage local copies of DTDs. The XML processor maintains a catalog file that lists public identifiers and their associated URIs. The processor will use these URIs instead of the system identifier.

Obviously, if the URIs in the catalog point to local copies of the DTD, the XML processor saves some downloads.

Listing 3.4 is an example of a catalog file.

Listing 3.4: A Catalog File

```
<XMLCatalog>
   <Base HRef="http://catwoman.pineapplesoft.com/dtd/"/>
   <Map PublicId="-//Pineapplesoft//Address Book//EN"
        HRef="address-book.dtd"/>
   <Map PublicId="-//Pineapplesoft//Article//EN"
        HRef="article.dtd"/>
   <Map PublicId="-//Pineapplesoft//Simple Order//EN"
        HRef="order.dtd"/>
   <Extend Href="http://www.w3.org/xcatalog/mastercat.xml"/>
</XMLCatalog>
```

EXAMPLE

Finally, note that a document can have both an internal and an external subset as in

```
<!DOCTYPE address SYSTEM "address-content.dtd" [
<!ELEMENT address  (street,region?,postal-code,locality,country)>
<!ATTLIST address  preferred (true ¦ false)  "false">
]>
```

Public Identifiers Format

EXAMPLE

The following public identifiers point to the address book:

```
"-//Pineapplesoft//Address Book//EN"
```

There are four parts, separated by "`//`":

- The first character is `+` if the organization is registered with ISO, `-` otherwise (most frequent).
- The second part is the owner of the DTD.
- The third part is the description of the DTD; spaces are allowed.
- The final part is the language (`EN` for English).

Standalone Documents

As you have seen, the DTD not only describes the document, but it can affect how the application reads the document. Specifically, default and fixed attribute values will add information to the document. Entities, which are also declared in the DTD, modify the document.

If all the entries that can influence the document are in the internal subset of the DTD, the document is said to be standalone. In other words, an XML processor does not need to download external entities to access all the information (it might have to download external entities to validate the document but that does not impact the content).

Conversely, if default attribute values or entities are declared in the external subset of the document, then the XML processor has to read the external subset, which might involve downloading more files.

EXAMPLE

Obviously, a standalone document is more efficient for communication over a network because only one file needs to be downloaded. The XML declaration has an attribute, `standalone`, that declares whether the document is a standalone document or not. It accepts only two values: `yes` and `no`. The default is `no`.

```
<?xml version="1.0" standalone="yes"?>
```

Note that a standalone document might have an external DTD subset but the external subset cannot modify how the application reads the document. Specifically, the external subset cannot

- declare entities
- declare default attribute values
- declare element content if the elements include spaces, such as for indenting. The last rule is the easiest to break but it is logical: If the DTD declares element content, then the processor reports indenting as ignorable whitespaces; otherwise, it reports as normal whitespaces.

Why Schemas?

Why do we need DTDs or schemas in XML? There is a potential conflict between flexibility and ease of use. As a rule, more flexible solutions are more difficult, if only because you have to work your way through the options. Specific solutions might also be optimized for certain tasks.

Let's compare a closed solution, HTML, with an open one such as XML. Both can be used to publish documents on the Web (XML serves many other purposes as well). HTML has a fixed set of elements and software can be highly optimized for it. For example, HTML editors offer templates, powerful visual editing, image editing, document preview, and more.

XML, on the other hand, is a flexible solution. It does not define elements but lets you, the developer, define the structure you need. Therefore, XML editors must accept any document structure. There are very little opportunities to optimize the XML editors because, by definition, they must be as generic as XML is. HTML, the close solution, has an edge here.

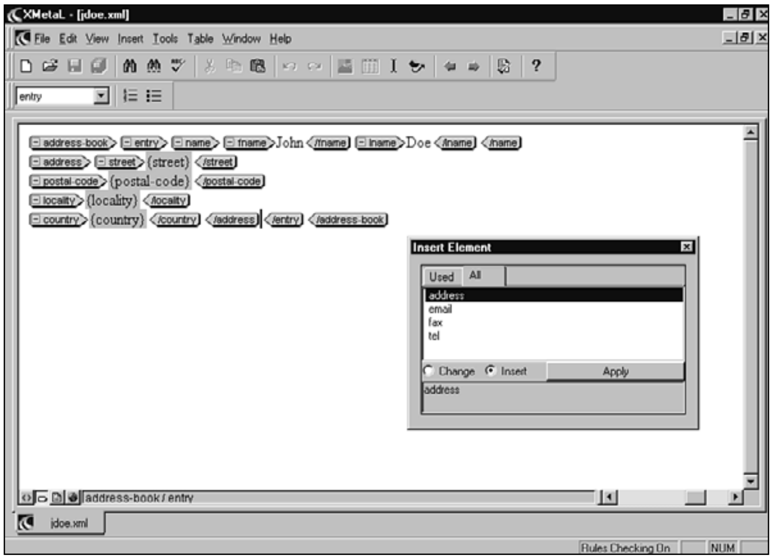
The DTD is an attempt to bridge that gap. DTD is a formal description of the document. Software tools can read it and learn about the document structure. Consequently, the tools can adapt themselves to better support the document structure.

For example, some XML editors use DTDs to populate their element lists as well as adopt default styling, based on the DTD. Finally, these XML editors will guide the author by making certain the structure is followed.

In other words, the editor is a generic tool that accepts any XML document, but it is configured for a specific application (read specific structure) through the DTD.

EXAMPLE

Figure 3.1 is a screenshot from a DTD-aware editor. Notice that the editor prompts for elements based on the structure.



```
<address-book> <entry> <name> <fname>John </fname> <lname>Doe </lname> </name>
<address> <street> (street) </street>
<postal-code> (postal-code) </postal-code>
<locality> (locality) </locality>
<country> (country) </country> </address> </entry> </address-book>
```

```
address
email
fax
tel
```

address

address-book / entry

Figure 3.1: XML editor uses the DTD to guide the user.

Well-Formed and Valid Documents

XML recognizes two classes of documents: *well-formed* and *valid*. The documents in Chapter 2 were well-formed, which in XML jargon means they follow the XML syntax. Well-formed documents have the right mix of start and end tags, attributes are properly quoted, entities are acceptable, character sets are properly used, and so on.

Well-formed documents have no DTD, so the XML processor cannot check their structure. It only checks that they follow the syntax rules.

Valid documents are stricter. They not only follow the syntax rules, they also comply with a specific structure, as described in a DTD.

Valid documents have a DTD. The XML processor will check that the documents are syntactically correct but it also ensures they follow the structure described in the DTD.

Why two classes of documents? Why not have only valid documents? In practice, some applications don't need a DTD. Also, among those applications that do, they need the DTD only at specific steps in the process.

The DTD is useful during document creation, when it makes sense to enforce the document structure. However, it is less useful after the creation. For example, in most cases, it is useless to distribute the DTD with the document. Indeed, a reader cannot fix errors in the structure of a document (that's the role of the author and editor), so what is a reader to do with the DTD?

Relationship Between the DTD and the Document

Unless it's overlooked, let me stress the relationship between the DTD and the XML document. The role of the DTD is to specify which elements are allowed where in the document.

EXAMPLE

The documents in Listings 3.6 and 3.7 are valid and respect the DTD in Listing 3.5. The document in Listing 3.6 has a `region` element, whereas the one in Listing 3.7 has none. It works because `region` is a conditional element in the DTD.

Listing 3.5: The DTD

```
<!ELEMENT address  (street,region?,postal-code,locality,country)>
<!ATTLIST address  preferred (true ¦ false)  "false">

<!ELEMENT street       (#PCDATA)>
<!ELEMENT region       (#PCDATA)>
<!ELEMENT postal-code  (#PCDATA)>
<!ELEMENT locality     (#PCDATA)>
<!ELEMENT country      (#PCDATA)>
```

Listing 3.6: A Valid Document

```
<?xml version="1.0"?>
<!DOCTYPE address SYSTEM "address.dtd">
<address>
   <street>34 Fountain Square Plaza</street>
   <region>OH</region>
   <postal-code>45202</postal-code>
   <locality>Cincinnati</locality>
   <country>US</country>
</address>
```

Listing 3.7: Another Valid Document

```
<?xml version="1.0"?>
<!DOCTYPE address SYSTEM "address.dtd">
<address>
   <street>Rue du Lombard 345</street>
   <postal-code>5000</postal-code>
   <locality>Namur</locality>
   <country>Belgium</country>
</address>
```

However, Listings 3.8 and 3.9 are not valid documents. Listing 3.8 is missing a `country` element and `country` is not optional. In Listing 3.9, the `region` element has a `code` attribute that is not declared in the DTD.

Listing 3.8: An Invalid Document

```
<?xml version="1.0"?>
<!DOCTYPE address SYSTEM "address.dtd">
<address>
   <street>34 Fountain Square Plaza</street>
   <region>OH</region>
   <postal-code>45202</postal-code>
   <locality>Cincinnati</locality>
</address>
```

Listing 3.9: An Invalid Document

```
<?xml version="1.0"?>
<!DOCTYPE address SYSTEM "address.dtd">
<address>
   <street>34 Fountain Square Plaza</street>
   <region code="OH">Ohio</region>
   <postal-code>45202</postal-code>
   <locality>Cincinnati</locality>
   <country>US</country>
</address>
```

Another way to look at the relationship between DTD and document is to say that the DTD describes the tree that is acceptable for the document. Figure 3.2 shows the tree described by the DTD in Listing 3.5.

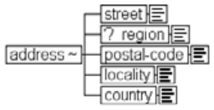


Figure 3.2: *The tree for the address*

Benefits of the DTD

The main benefits of using a DTD are

- The XML processor enforces the structure, as defined in the DTD.
- The application accesses the document structure, such as to populate an element list.
- The DTD gives hints to the XML processor—that is, it helps separate indenting from content.
- The DTD can declare default or fixed values for attributes. This might result in a smaller document.

Validating the Document

You can validate documents with an XML processor. I invite you to download XML for Java from the IBM Web site at `www.alphaworks.ibm.com`. There are other XML processors, but I will use the IBM one in Chapter 5, “XSL Transformation,” and Chapter 8, “Alternative API: SAX.”

XML for Java is a Java application. You don’t need to be a Java programmer to use it, but you must have installed a Java runtime on your system. You can download a Java runtime from `java.sun.com`.

Tools are sometimes updated. If the status of XML for Java changes, we will post an update on the Macmillan Web site at `www.quecorp.com/series/by_example`. If you experience a problem finding the tool, visit `www.quecorp.com/series/by_example`.

EXAMPLE

The XML for Java comes with a command-line version that you can use to validate documents against their DTD. To validate the document in Listing 3.1, save it in a file called “abook.xml,” save its DTD in the file called “address-book.dtd,” and issue the command:

```
java -classpath c:\xml4j\xml4j.jar;c:\xml4j\xml4jsamples.jar
➥XJParse -p com.ibm.xml.parsers.ValidatingSAXParser abook.xml
```

This looks like a long and complex command line. If you are curious, Appendix A breaks it into smaller pieces.

OUTPUT

This command assumes XML for Java is installed in the `c:\xml4j` directory. You might have to update the classpath for your system. If everything goes well, the result is a message similar to

```
abook.xml: 1420 ms (24 elems, 9 attrs, 105 spaces, 97 chars)
```

If the document contains errors (either syntax errors or it does not respect the structure outlined in the DTD), you will have an error message.

CAUTION

The IBM for Java processor won't work unless you have installed a Java runtime.

If there is an error message similar to "Exception in thread "main" java.lang.NoClassDefFoundError," it means that either the `classpath` is incorrect (make sure it points to the right directory) or that you typed an incorrect class name for XML for Java (`XJParser` and `com.ibm.xml.parsers.ValidatingSAXParser`).

If there is an error message similar to "Exception in thread "main" java.io.FileNotFoundException: d:\xml\abook.xm", it means that the filename is incorrect (in this case, it points to "abook.xm" instead of "abook.xml").

TIP

You can save some typing with batch files (under Windows) or shell scripts (under UNIX). Adapt the path to your system, replace the filename (abook.xml) with "%1" and save in a file called "validate.bat". The file should contain the following command:

```
java -classpath c:\xml4j\xml4j.jar;c:\xml4j\xml4jsamples.jar
➥XJParse -p com.ibm.xml.parsers.ValidatingSAXParser %1
```

Now you can validate any XML file with the following (shorter) command:

```
validate abook.xml
```

Entities and Notations

As already mentioned in the previous chapter, XML doesn't work with files but with entities. Entities are the physical representation of XML documents. Although entities usually are stored as files, they need not be.

In XML the document, its DTD, and the various files it references (images, stock-phrases, and so on) are entities. The document itself is a special entity because it is the starting point for the XML processor. The entity of the document is known as the *document entity*.

XML does not dictate how to store and access entities. This is the task of the XML processor and it is system specific. The XML processor might have to download entities or it might use a local catalog file to retrieve the entities.

In Chapter 7, "The Parser and DOM," you'll see how SAX parsers (a SAX parser is one example of an XML processor) enable the application to retrieve entities from databases or other sources.

XML has many types of entities, classified according to three criteria: *general* or *parameter entities, internal* or *external entities*, and *parsed* or *unparsed entities*.

General and Parameter Entities

EXAMPLE

General entity references can appear anywhere in text or markup. In practice, general entities are often used as macros, or shorthand for a piece of text. External general entities can reference images, sound, and other documents in non-XML format. Listing 3.10 shows how to use a general entity to replace some text.

Listing 3.10: General Entity

```
<?xml version="1.0"?>
<!DOCTYPE address-book [
<!ENTITY jacksmith
'<entry>
    <name><fname>Jack</fname><lname>Smith</lname></name>
    <tel>513-555-3465</tel>
    <email href="mailto:jsmith@emailaholic.com"/>
 </entry>'>
]>
<address-book>
   &jacksmith;
</address-book>
```

General entities are declared with the markup `<!ENTITY` followed by the entity name, the entity definition, and the customary right angle bracket.

TIP

General entities also are often used to associate a mnemonic with character references as in

```
<!ENTITY icirc "&#238;">
```

As we saw in Chapter 2, "The XML Syntax," the following entities are predefined in XML: "`&lt;`", "`&amp;`", "`&gt;`", "`&apos;`", and "`&quot;`".

Parameter entity references can only appear in the DTD. There is an extra `%` character in the declaration before the entity name. Parameter entity references also replace the ampersand with a percent sign as in

```
<!ENTITY % boolean  "(true ¦ false) 'false'">
<!ELEMENT tel       (#PCDATA)>
<!ATTLIST tel       preferred %boolean;>
```

Parameter entities have many applications. You will learn how to use parameter entities in the following sections: “Internal and External Entities,” “Conditional Sections,” “Designing DTDs from an Object Model.”

CAUTION

The previous example is valid only in the external subset of a DTD. In the internal subset, parameter entities can appear only where markup declaration can appear.

Internal and External Entities

XML also distinguishes between internal and external entities. Internal entities are stored in the document, whereas external entities point to a system or public identifier. Entity identifiers are identical to DTD identifiers (in fact, the DTD is a special entity).

The entities in the previous sections were internal entities because their value was declared in the entity definition. External entities, on the other hand, reference content that is not part of the current document.

TIP

External entities might start with an XML declaration—for example, to declare a special encoding.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

External general entities can be parsed or unparsed. If parsed, the entity must contain valid XML text and markup. External parsed entities are used to share text across several documents, as illustrated by Listing 3.11.

EXAMPLE

In Listing 3.11, the various entries are stored in separate entities (separate files). The address book combines them in a document.

Listing 3.11: Using External Entities

```
<?xml version="1.0"?>
<!DOCTYPE address-book [
<!ENTITY johndoe   SYSTEM "johndoe.ent">
<!ENTITY jacksmith SYSTEM "jacksmith.ent">
]>
<address-book>
   &johndoe;
   &jacksmith;
</address-book>
```

Where the file “johndoe.ent” contains:

```
<entry>
   <name>John Doe</name>
   <address>
      <street>34 Fountain Square Plaza</street>
      <region>OH</region>
      <postal-code>45202</postal-code>
      <locality>Cincinnati</locality>
      <country>US</country>
   </address>
</entry>
```

And “jacksmith.ent” contains

```
<entry>
   <name><fname>Jack</fname><lname>Smith</lname></name>
   <tel>513-555-3465</tel>
   <email href="mailto:jsmith@emailaholic.com"/>
</entry>
```

EXAMPLE

However, unparsed entities are probably the most helpful external general entities. Unparsed entities are used for non-XML content, such as images, sound, movies, and so on. Unparsed entities provide a mechanism to load non-XML data into a document.

The XML processor treats the unparsed entity as an opaque block, of course. By definition, it does not attempt to recognize markup in unparsed entities.

A notation must be associated with unparsed entities. Notations are explained in more detail in the next section but, in a nutshell, they identify the type of a document, such as GIF, JPEG, or Windows bitmap for images. The notation is introduced by the NDATA keyword:

```
<!ENTITY logo SYSTEM "http://catwoman.pineapplesoft.com/logo.gif"
               NDATA GIF>
```

EXAMPLE

External parameter entities are similar to external general entities. However, because parameter entities appear in the DTD, they must contain valid XML markup.

External parameter entities are often used to insert the content of a file in the markup. Let’s suppose we have created a list of general entities for every country, as in Listing 3.12 (saved in the file `countries.ent`).

Listing 3.12: A List of Entities for the Countries

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ENTITY be "Belgium">
```

```
<!ENTITY ch "Switzerland">
<!ENTITY de "Germany">
<!ENTITY it "Italy">
<!ENTITY jp "Japan">
<!ENTITY uk "United Kingdom">
<!ENTITY us "United States">
<!-- and more -->
```

EXAMPLE

Creating such a list is a large effort. We would like to reuse it in all our documents. The construct illustrated in Listing 3.13 pulls the list of countries from `countries.ent` in the current document. It declares a parameter entity as an external entity and it immediately references the parameter entity. This effectively includes the external list of entities in the DTD of the current document.

Listing 3.13: Using External Parameter Entities

```
<?xml version="1.0"?>
<!DOCTYPE address SYSTEM "address.dtd" [
<!ENTITY % countries SYSTEM "countries.ent">
%countries;
]>
<address>
   <street>34 Fountain Square Plaza</street>
   <region>Ohio</region>
   <postal-code>45202</postal-code>
   <locality>Cincinnati</locality>
   <country>&us;</country>
</address>
```

CAUTION

Given the limitation on parameter entities in the internal subset of the DTD, this is the only sensible application of parameter entities in the internal subset.

Notation

Because the XML processor cannot process unparsed entities, it needs a mechanism to associate them with the proper tool. In the case of an image, it could be an image viewer.

EXAMPLE

Notation is simply a mechanism to declare the type of unparsed entities and associate them, through an identifier, with an application.

```
<!NOTATION GIF89a PUBLIC "-//CompuServe//NOTATION Graphics
➥ Interchange Format 89a//EN" "c:\windows\kodakprv.exe">
```

EXAMPLE

This declaration is unsafe because it points to a specific application. The application might not be available on another computer or it might be available but from another path. If your system has defined the appropriate file associations, you can get away with a declaration such as

```
<!NOTATION GIF89a SYSTEM "GIF">
<!NOTATION GIF89a SYSTEM "image/gif">
```

The first notation uses the filename, while the second uses the MIME type.

Managing Documents with Entities

External entities are helpful to modularize and help manage large DTDs and large document sets.

The idea is very simple: Try to divide your work into smaller pieces that are more manageable. Save each piece in a separate file and include them in your document with external entities.

Also try to identify pieces that you can reuse across several applications. It might be a list of entities (such as the list of countries) or a list of notations, or some text (such as a copyright notice that must appear on every document). Place them in separate files and include them in your documents through external entities.

Figure 3.3 shows how it works. Notice that some files are shared across several documents.

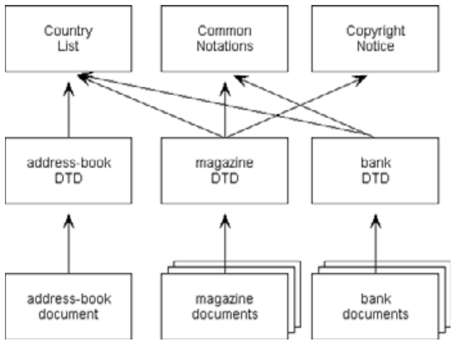


Figure 3.3: *Using external entities to manage large projects*

This is like eating a tough steak: You have to cut the meat into smaller pieces until you can chew it.

Conditional Sections

As your DTDs mature, you might have to change them in ways that are partly incompatible with previous usage. During the migration period, when you have new and old documents, it is difficult to maintain the DTD.

To help you manage migrations and other special cases, XML provides *conditional sections*. Conditional sections are included or excluded from the DTD depending on the value of a keyword. Therefore, you can include or exclude a large part of a DTD by simply changing one keyword.

EXAMPLE

Listing 3.13 shows how to use conditional sections. The `strict` parameter entity resolves to `INCLUDE`. The `lenient` parameter entity resolves to `IGNORE`. The application will use the definition of name in the `%strict;` section (`(fname, lname)`) and ignores the definition in the `%lenient;` section (`(#PCDATA ¦ fname ¦ lname)*`).

Listing 3.13: Using Conditional Sections

```
<!ENTITY % strict 'INCLUDE'>
<!ENTITY % lenient 'IGNORE'>

<![%strict;[
<!-- a name is a first name and a last name -->
<!ELEMENT name  (fname, lname)>
]]>
<![%lenient;[
<!-- name is made of string, first name
     and last name. This is a very flexible
     model to accommodate exotic name        -->
<!ELEMENT name  (#PCDATA ¦ fname ¦ lname)*>
]]>
```

However, to revert to the lenient definition of name, it suffices to invert the parameter entity declaration:

```
<!ENTITY % strict 'IGNORE'>
<!ENTITY % lenient 'INCLUDE'>
```

Designing DTDs

Now that you understand what DTDs are for and that you understand how to use them, it is time to look at how to create DTDs. DTD design is a creative and rewarding activity.

It is not possible, in this section, to cover every aspect of DTD design. Books have been devoted to that topic. Use this section as guidance and remember that practice makes proficient.

Yet, I would like to open this section with a plea to use existing DTDs when possible. Next, I will move into two examples of the practical design of practically designing DTDs.

Main Advantages of Using Existing DTDs

There are many XML DTDs available already and it seems more are being made available every day. With so many DTDs, you might wonder whether it's worth designing your own.

I would argue that, as much as possible, you should try to reuse existing DTDs. Reusing DTDs results in multiple savings. Not only do you not have to spend time designing the DTD, but also you don't have to maintain and update it.

However, designing an XML application is not limited to designing a DTD. As you will learn in Chapter 5, "XSL Transformation," and subsequent chapters, you might also have to design style sheets, customize tools such as editors, and/or write special code using a parser.

This adds up to a lot of work. And it follows the "uh, oh" rule of project planning: Uh, oh, it takes more work than I thought." If at all possible, it pays to reuse somebody else's DTD.

The first step in a new XML project should be to search the Internet for similar applications. I suggest you start at `www.oasis-open.org/sgml/xml.html`. The site, maintained by Robin Cover, is the most comprehensive list of XML links.

In practice, you are likely to find DTDs that almost fit your needs but aren't exactly what you are looking for. It's not a problem because XML is extensible so it is easy to take the DTD developed by somebody else and adapt it to your needs.

Designing DTDs from an Object Model

I will take two examples of DTD design. In the first example, I will start from an object model. This is the easiest solution because you can reuse the objects defined in the model. In the second example, I will create a DTD from scratch.

EXAMPLE

Increasingly, object models are made available in UML. UML is the Unified Modeling Language (yes, there is an ML something that does not stand for markup language). UML is typically used for object-oriented applications such as Java or C++ but the same models can be used with XML.

An object model is often available when XML-enabling an existing Java or C++ application. Figure 3.4 is a (simplified) object model for bank accounts. It identifies the following objects:

- "Account" is an abstract class. It defines two properties: the balance and a list of transactions.
- "Savings" is a specialized "Account" that represents a savings account; interest is an additional property.
- "Checking" is a specialized "Account" that represents a checking account; rate is an additional property.
- "Owner" is the account owner. An "Account" can have more than one "Owner" and an "Owner" can own more than one "Account."

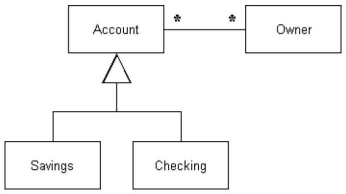


Figure 3.4: *The object model*

The application we are interested in is Web banking. A visitor would like to retrieve information about his or her various bank accounts (mainly his or her balance).

The first step to design the DTD is to decide on the root-element. The top-level element determines how easily we can navigate the document and access the information we are interested in. In the model, there are two potential top-level elements: Owner or Account.

Given we are doing a Web banking application, Owner is the logical choice as a top element. The customer wants his list of accounts.

Note that the choice of a top-level element depends heavily on the application. If the application were a financial application, examining accounts, it would have been more sensible to use account as the top-level element.

At this stage, it is time to draw a tree of the DTD under development. You can use a paper, a flipchart, a whiteboard, or whatever works for you (I prefer flipcharts).

In drawing the tree, I simply create an element for every object in the model. Element nesting is used to model object relationship.

Figure 3.5 is a first shot at converting the model into a tree. Every object in the original model is now an element. However, as it turns out, this tree is both incorrect and suboptimal.

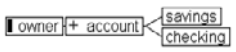


Figure 3.5: *A first tree for the object model*

Upon closer examination, the tree in Figure 3.5 is incorrect because, in the object model, an account can have more than one owner. I simply cannot add the owner element into the account because this would lead to infinite recursion where an account includes its owner, which itself includes the account, which includes the owner, which… You get the picture.

The solution is to create a new element co-owner. To avoid confusion, I decided to rename the top-level element from owner to accounts. The new tree is in Figure 3.6.

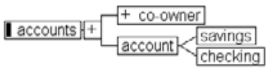


Figure 3.6: *The corrected tree*

The solution in Figure 3.6 is a correct implementation of the object model. To evaluate how good it is, I like to create a few sample documents that follow the same structure. Listing 3.14 is a sample document I created.

Listing 3.14: Sample Document

```
<?xml version="1.0"?>
<accounts>
  <co-owner>John Doe</co-owner>
  <co-owner>Jack Smith</co-owner>
  <account>
    <checking>170.00</checking>
  </account>
  <co-owner>John Doe</co-owner>
  <account>
    <savings>5000.00</savings>
  </account>
</accounts>
```

This works but it is inefficient. The checking and savings elements are completely redundant with the account element. It is more efficient to treat account as a parameter entity that groups the commonality between the various accounts. Figure 3.7 shows the result. In this case, the parameter entity is used to represent a type.

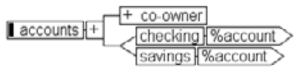


Figure 3.7: *The tree, almost final*

We're almost there. Now we need to flesh out the tree by adding the object properties. I chose to create new elements for every property (see the following section "On Elements Versus Attributes").

Figure 3.8 is the final result. Listing 3.15 is a document that follows the structure. Again, it's useful to write a few sample documents to check whether the DTD makes sense. I can find no problems with this structure in Listing 3.15.

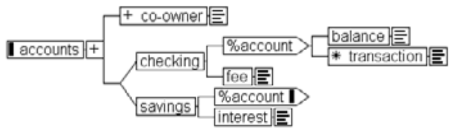


Figure 3.8: *The final tree*

Listing 3.15: A Sample Document

```
<?xml version="1.0"?>
<accounts>
  <co-owner>John Doe</co-owner>
  <co-owner>Jack Smith</co-owner>
  <checking>
         <balance>170.00</balance>
         <transaction>-100.00</transaction>
         <transaction>-500.00</transaction>
         <fee>4.00</fee>
  </checking>
  <co-owner>John Doe</co-owner>
  <savings>
         <balance>5000.00</balance>
         <interest>212.50</interest>
  </savings>
</accounts>
```

Having drawn the tree, it is trivial to turn it into a DTD. It suffices to list every element in the tree and declare their content model based on their children. The final DTD is in Listing 3.16.

Listing 3.16: The DTD for Banking

```
<!ENTITY % account     "(balance,transaction*)">
<!ELEMENT accounts     (co-owner+,(checking ¦ savings))+>
<!ELEMENT co-owner     (#PCDATA)>
<!ELEMENT checking     (%account;,fee)>
<!ELEMENT savings      (%account;,interest)>
<!ELEMENT fee           (#PCDATA)>
<!ELEMENT interest     (#PCDATA)>
<!ELEMENT balance      (#PCDATA)>
<!ELEMENT transaction  (#PCDATA)>
```

Now I have to publish this DTD under a URI. I like to place versioning information in the URI (version 1.0, and so on) because if there is a new version of the DTD, it gets a different URI with the new version. It means the two DTDs can coexist without problem.

It also means that the application can retrieve the URI to know which version is in use.

```
http://catwoman.pineapplesoft.com/dtd/accounts/1.0/accounts.dtd
```

If I ever update the DTD (it's a very simplified model so I can think of many missing elements), I'll create a different URI with a different version number:

```
http://catwoman.pineapplesoft.com/dtd/accounts/2.0/accounts.dtd
```

You can see how easy it is to create an XML DTD from an object model. This is because XML tree-based structure is a natural mapping for objects.

As more XML applications will be based on object-oriented technologies and will have to integrate with object-oriented systems written in Java, CORBA, or C++, I expect that modeling tools will eventually create DTDs automatically.

Already modeling tools such as Rational Rose or Together/J can create Java classes automatically. Creating DTDs seems like a logical next step.

On Elements Versus Attributes

As you have seen, there are many choices to make when designing a DTD. Choices include deciding what will become of an element, a parameter entity, an attribute, and so on.

Deciding what should be an element and what should be an attribute is a hot debate in the XML community. We will revisit this topic in Chapter 10, "Modeling for Flexibility," but here are some guidelines:

- The main argument in favor of using attributes is that the DTD offers more controls over the type of attributes; consequently, some people argue that object properties should be mapped to attributes.
- The main argument for elements is that it is easier to edit and view them in a document. XML editors and browsers in general have more intuitive handling of elements than of attributes.

I try to be pragmatic. In most cases, I use element for "major" properties of an object. What I define as major is all the properties that you manipulate regularly.

I reserve attributes for ancillary properties or properties that are related to a major property. For example, I might include a currency indicator as an attribute to the balance.

Creating the DTD from Scratch

Creating a DTD without having the benefit of an object model results in more work. The object model provides you with ready-made objects that you just have to convert in XML. It also has identified the properties of the objects and the relationships between objects.

However, if you create a DTD from scratch, you have to do that analysis as well.

A variant is to modify an existing DTD. Typically, the underlying DTD does not support all your content (you need to add new elements/attributes) or is too complex for your application (you need to remove elements/attributes).

This is somewhat similar to designing a DTD from scratch in the sense that you will have to create sample documents and analyze them to understand how to adapt the proposed DTD.

On Flexibility

When designing your own DTD, you want to prepare for evolution. We'll revisit this topic in Chapter 10 but it is important that you build a model that is flexible enough to accommodate extensions as new content becomes available.

The worst case is to develop a DTD, create a few hundred or a few thousand documents, and suddenly realize that you are missing a key piece of information but that you can't change your DTD to accommodate it. It's bad because it means you have to convert your existing documents.

To avoid that trap you want to provide as much structural information as possible but not too much. The difficulty, of course, is in striking the right balance between enough structural information and too much structural information.

You want to provide enough structural information because it is very easy to degrade information but difficult to clean degraded information.

Compare it with a clean, neatly sorted stack of cards on your desk. It takes half a minute to knock it down and shuffle it. Yet it will take the best part of one day to sort the cards again.

The same is true with electronic documents. It is easy to lose structural information when you create the document. And if you lose structural information, it will be very difficult to retrieve it later on.

EXAMPLE

Consider Listing 3.17, which is the address book in XML. The information is highly structured—the address is broken down into smaller components: street, region, and so on.

Listing 3.17: An Address Book in XML

```
<?xml version="1.0"?>
<!DOCTYPE address-book SYSTEM "address-book.dtd">
<!-- loosely inspired by vCard 3.0 -->
<address-book>
   <entry>
      <name>John Doe</name>
      <address>
         <street>34 Fountain Square Plaza</street>
         <region>OH</region>
         <postal-code>45202</postal-code>
         <locality>Cincinnati</locality>
         <country>US</country>
      </address>
      <tel preferred="true">513-555-8889</tel>
      <tel>513-555-7098</tel>
      <email href="mailto:jdoe@emailaholic.com"/>
   </entry>
   <entry>
      <name><fname>Jack</fname><lname>Smith</lname></name>
      <tel>513-555-3465</tel>
      <email href="mailto:jsmith@emailaholic.com"/>
   </entry>
</address-book>
```

Listing 3.18 is the same information as text. The structure is lost and, unfortunately, it will be difficult to restore the structure automatically. The software would have to be quite intelligent to go through Listing 3.18 and retrieve the entry boundaries as well as break the address in its components.

Listing 3.18: The Address Book in Plain Text

```
John Doe

34 Fountain Square Plaza

Cincinnati, OH 45202

US

513-555-8889 (preferred)

513-555-7098

jdoe@emailaholic.com

Jack Smith

513-555-3465

jsmith@emailaholic.com
```

However, as you design your structure, be careful that it remains usable. Structures that are too complex or too strict will actually lower the quality of your document because it encourages users to cheat.

Consider how many electronic commerce Web sites want a region, province, county, or state in the buyer address. Yet many countries don't have the notion of region, province, county, or state or, at least, don't use it for their addresses.

Forcing people to enter information they don't have is asking them to cheat. Keep in mind the number one rule of modeling: Changes will come from the unexpected. Chances are that, if your application is successful, people will want to include data you had never even considered. How often did I include for "future extensions" that were never used? Yet users came and asked for totally unexpected extensions.

There is no silver bullet in modeling. There is no foolproof solution to strike the right balance between extensibility, flexibility, and usability. As you grow more experienced with XML and DTDs, you also will improve your modeling skills.

My solution is to define a DTD that is large enough for all the content required by my application but not larger. Still, I leave hooks in the DTD—places where it would be easy to add a new element, if required.

Modeling an XML Document

EXAMPLE

The first step in modeling XML documents is to create documents. Because we are modeling an address book, I took a number of business cards and created documents with them. You can see some of the documents I created in Listing 3.20.

Listing 3.20: Examples of XML Documents

```
<address-book>
   <entry>
      <name><fname>John</fname><lname>Doe</lname></name>
      <address>
         <street>34 Fountain Square Plaza</street>
         <state>OH</state>
         <zip>45202</zip>
         <locality>Cincinnati</locality>
         <country>US</country>
      </address>
      <tel>513-555-8889</tel>
      <email href="mailto:jdoe@emailaholic.com"/>
   </entry>
   <entry>
      <name><fname>Jean</fname><lname>Dupont</lname></name>
      <address>
         <street>Rue du Lombard 345</street>
         <postal-code>5000</postal-code>
         <locality>Namur</locality>
         <country>Belgium</country>
      </address>
      <email href="mailto:jdupont@emailaholic.com"/>
   </entry>
   <entry>
      <name><fname>Olivier</fname><lname>Rame</lname></name>
      <email href="mailto:orame@emailaholic.com"/>
   </entry>
</address-book>
```

As you can see, I decided early on to break the address into smaller components. In making these documents, I tried to reuse elements over and over again. Very early in the project, it was clear there would be a name element, an address element, and more.

Also, I decided that addresses, phone numbers, and so on would be conditional. I have incomplete entries in my address book and the XML version must be able to handle it as well.

I looked at commonalties and I found I could group postal code and zip code under one element. Although they have different names, they are the same concepts.

This is the creative part of modeling when you list all possible elements, group them, and reorganize them until you achieve something that makes sense. Gradually, a structure appears.

Building the DTD from this example is easy. I first draw a tree with all the elements introduced in the document so far, as well as their relationship. It is clear that some elements such as state are optional. Figure 3.9 shows the tree.

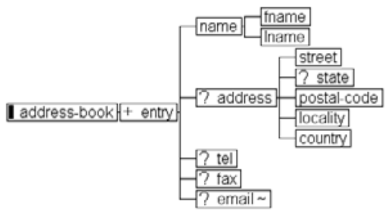


Figure 3.9: *The updated tree*

This was fast to develop because the underlying model is simple and well known. For a more complex application, you would want to spend more time drafting documents and trees.

At this stage, it is a good idea to compare my work with other similar works. In this case, I choose to compare with the vCard standard (RFC 2426). vCard (now in its third version) is a standard for electronic business cards.

vCard is a very extensive standard that lists all the fields required in an electronic business card. vCard, however, is too complicated for my needs so I don't want to simply duplicate that work.

By comparing the vCard structure with my structure, I realized that names are not always easily broken into first and last names, particularly foreign names. I therefore provided a more flexible content model for names.

I also realized that address, phone, fax number, and email address might repeat. Indeed, it didn't show up in my sample of business cards but there are people with several phone numbers or email addresses. I introduced a repetition for these as well as an attribute to mark the preferred address. The attribute has a default value of false.

In the process, I picked the name "region" for the state element. For some reason, I find region more appealing.

Comparing my model with vCard gave me the confidence that the simple address book can cope with most addresses used. Figure 3.10 is the result.

TIP

There is a group working on the XML-ization of the vCard standard. Its approach is different: It starts with vCard as its model, whereas this example starts from an existing document and uses vCard as a check.

Yet, it is interesting to compare the XML version of vCard (available from `www.imc.org/ietf-vcard-xml`) with the DTD in this chapter. It proves that there is more than one way to skin a cat.

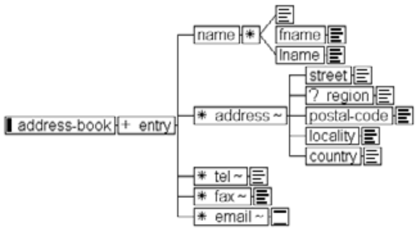


Figure 3.10: *The final tree*

Again converting the tree in a DTD is trivial. Listing 3.21 shows the result.

Listing 3.21: A DTD for the Address Book

```
<!ENTITY % boolean "(true ¦ false) 'false'">

<!-- top-level element, the address book
     is a list of entries                    -->
<!ELEMENT address-book  (entry+)>

<!-- an entry is a name followed by
     addresses, phone numbers, etc.         -->
<!ELEMENT entry  (name,address*,tel*,fax*,email*)>

<!-- name is made of string, first name
     and last name. This is a very flexible
         model to accommodate exotic name       -->
<!ELEMENT name   (#PCDATA ¦ fname ¦ lname)*>
<!ELEMENT fname  (#PCDATA)>
```

```
<!ELEMENT lname  (#PCDATA)>

<!-- definition of the address structure
     if several addresses, the preferred
         attribute signals the "default" one   -->
<!ELEMENT address      (street,region?,postal-code,locality,country)>
<!ATTLIST address      preferred (true ¦ false)  "false">
<!ELEMENT street       (#PCDATA)>
<!ELEMENT region       (#PCDATA)>
<!ELEMENT postal-code  (#PCDATA)>
<!ELEMENT locality     (#PCDATA)>
<!ELEMENT country      (#PCDATA)>

<!-- phone, fax and email, same preferred
     attribute as address                    -->
<!ELEMENT tel       (#PCDATA)>
<!ATTLIST tel       preferred (true ¦ false)  "false">
<!ELEMENT fax       (#PCDATA)>
<!ATTLIST fax       preferred (true ¦ false)  "false">
<!ELEMENT email     EMPTY>
<!ATTLIST email     href  CDATA                #REQUIRED
                     preferred (true ¦ false)  "false">
```

Naming of Elements

Again, modeling requires imagination. One needs to be imaginative and keep an open mind during the process. Modeling also implies making decisions on the name of elements and attributes.

As you can see, I like to use meaningful names. Others prefer to use meaningless names or acronyms. Again, as is so frequent in modeling, there are two schools of thought and both have very convincing arguments. Use what works better for you but try to be consistent.

In general, meaningful names

- are easier to debug
- provide some level of document for the DTD.

However, a case can be made for acronyms:

- Acronyms are shorter, and therefore more efficient.
- Acronyms are less language-dependent.

- Name choice should not be a substitute for proper documentation; meaningless tags and acronyms might encourage you to properly document the application.

A Tool to Help

I find drawing trees on a piece of paper an exercise in frustration. No matter how careful you are, after a few rounds of editing, the paper is unreadable and modeling often requires several rounds of editing!

Fortunately, there are very good tools on the market to assist you while you write DTDs. The trees in this book were produced by Near & Far from Microstar (`www.microstar.com`).

Near & Far is as intuitive as a piece of paper but, even after 1,000 changes, the tree still looks good. Furthermore, to convert the tree in a DTD, it suffices to save it. No need to remember the syntax, which is another big plus.

EXAMPLE

Figure 3.11 is a screenshot of Near & Far.

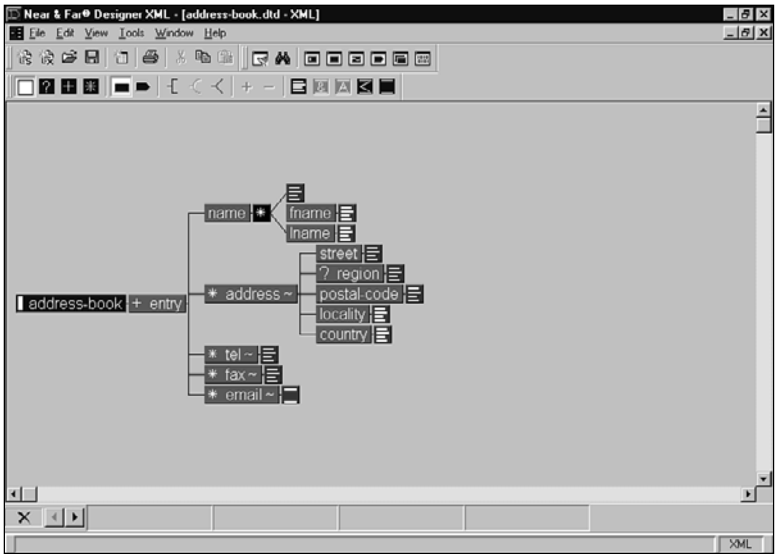


Figure 3.11: *Using a modeling tool*

New XML Schemas

The venerable DTD is very helpful. It provides valuable services to the application developer and the XML author. However, DTD originated in publishing and it shows.

For one thing, content is limited to textual content. Also, it is difficult to put in repetition constraints: You cannot say that an element can appear only four times. It's 0, 1, or infinite.

Furthermore, the DTD is based on a special syntax that is different from the syntax for XML documents. It means that it is not possible to use XML tools, such as editors or browsers, to process DTD!

These so-called limitations of the DTD are inherited directly from SGML. XML was originally designed as a subset of SGML and therefore it could not differ too much from the SGML DTD.

However, as XML takes a life of its own, people would like a new, more modern, replacement for the DTD. Various groups have made several proposals. Collectively, these proposals are known as schemas. The details of the proposals vary greatly but all:

- propose to use the same syntax as XML documents
- improve XML data typing to support not only strings but also numbers, dates, and so on
- introduce object-oriented concepts such as inheritance (an element could inherit from another)

The W3C has formed a working group to develop a new standard based on the existing proposals. At the time of this writing, the effort has just started and little is known about the final result.

You can find up-to-date information on new XML schemas on the W3C Web site at `www.w3.org/XML`.

The main proposals being considered are

- XML-Data, which offers types inspired from SQL types.
- DCD (Document Content Description), positioned as a simplified version of XML-Data.
- SOX (Schema for Object-oriented XML), as the name implies, is heavy on object-orientation aspects.
- DDML (Document Definition Markup Language), developed by the XML-Dev mailing list. It is intended as a simple solution to form a basis for future work.

What's Next

The next chapter is dedicated to the namespace proposal. Namespace is an often-overlooked but very useful standard that greatly enhances XML extensibility.

Index