

Internets

- “Introduction” on page 5
- “Why XML” on page 6
- “Structure of the Book” on page 8
- “Let’s Talk: Internets Are for Communicating” on page 9
- “The Velocity of Information” on page 10
- “Into the Smart Network” on page 11
- “Current Approaches—Can the Web Help?” on page 12
- “Where the Web Needs Help” on page 17
- “Beyond the Traditional Document” on page 18
- “Toward the Active Document” on page 19
- “Down to the Nitty-Gritty” on page 25
- “What Do We Do with Documents?” on page 26
- “DTDs and Content Specifications—A Short Excursion” on page 31
- “Conclusion” on page 49

Chapter 1

XML is a pivotal technology in the development of the Internet. In this first chapter we cover the philosophy behind XML and how it will affect your organization. This includes a new approach to documents and what we do with them. We will also include a roadmap of the rest of the book.

1.1 | Introduction

Despite the ever-changing panoply of tools, the reason businesses use computer technology remains the same: coordinate mission-critical resources and information to further the corporation's goals. From the "big iron" of the 1960s to the Internet of today, and on into the buzzwords of the future, this remains constant. Each new technology is measured ultimately by how it supports the corporation and is

either kept or dropped (possibly after a fair amount of blood, hopefully not yours, is spilled) as this holds true.

The information systems world is awash in talk of corporate internets, intranets, and extranets, and companies across the globe are rushing to take advantage of this “new” technology. But in this rush, the technologies have been hyped to death with very little serious thought given to just what the Internet is used for. The user’s goal, as in any corporate network, is to improve the information and resource sharing to further corporate objectives.

1.2 | Why XML

In this book we will introduce you to XML—the tool of choice for structured information in the networked age. We will show you how to use this important strategic technology to exploit the Internet to serve you. XML can not only help you in dealing with the new flashy buzzword technologies of the moment—such as groupware and the Internet—but also help insulate you from the booms and busts of the hot new products ready to drop in your lap in the future. Because we consider improving your life the primary goal of this book, we will not just blithely describe how to run a few products and then drop you but show just how XML can help you organize your information and resources better. We will introduce you to a *philosophy* of XML to help you get the most out of your Internet investment.

The ideas behind XML have been developing since the 1960s, culminating in the approval of XML’s parent, SGML, as an international standard in 1986. SGML found its niches in publishing, defense contractors, and large multinationals (such as aircraft manufacturers). It was perceived (wrongly, its practitioners would claim) as a difficult technology for high-end systems by corporations with deep pockets. However, by the mid-1990s, large-scale transmission of SGML-based electronic documents around the World Wide Web had become commonplace through HTML (an SGML application built from the

grass roots up). While HTML has played a key role in enabling the rapid development of the Web, the full power of SGML for the Internet remained largely untapped. In 1996 a working group under the aegis of the World Wide Web Consortium started developing XML, a streamlined version of SGML designed to simplify transmission of structured documents over the Web. In many ways XML is a “purification” of SGML, removing portions of it with limited application that complicate its very powerful and basically simple central ideas. Because of this, almost everything we have to say will apply equally to both and examples, except where explicitly stated, will conform to both. Unless the content clearly indicates otherwise the term “XML” will mean “XML and SGML.”

But before getting into the nitty-gritty of XML and the Internet (in later chapters) we need to convince you to spend time and money implementing XML technology. We will approach this in two ways:

1. Describe the qualities of the ideal Internet—the Internet you hoped you’d get with just HTML, a Web server, and some browsers, or with your “Internet-in-a-box” solution, but didn’t. Then we’ll show you just how much closer you can get with XML, both the state of the art and future possibilities. There will be a small philosophical detour as we build the list of requirements. You can skip this, if you’d like, and get right to the list, but it will also explain how the pieces of this book fit together and show you how to lead your technology, rather than letting it lead you.
2. Tell you of all the whiz-bang possibilities XML has to offer. This gee-whiz approach shows you all the incremental capabilities you can have without having to “buy into” the whole concept (although you should buy the book).

In the end these are the same—it is the whiz-bang features which allow you to implement an Internet, and it is the Internet which makes those whiz-bang features useful.

1.3 | Structure of the Book

This book is divided into five parts.

In the first part we will introduce you to the XML universe. Here you will find a discussion of the role of XML in the internet and a quick-start on the XML recommendation and XML tools. We don't assume prior knowledge of either XML or SGML but our task here is not to provide an extended tutorial or reference on the language syntax. What we do do is develop the perspective of the XML internet application designer and provide any background that is needed to comprehend the subsequent chapters.

The next three parts consist of a series of projects using XML in actual internet applications. Working through the projects the reader will gain concrete experience in the design of XML applications, DTDs, and programming. We also delve into standards related to XML and the internet wherever relevant.

This first project spans five chapters as the construction of several types of components is involved including a bulletin board, forms processing tools, a search engine, and transformation filters.

Most of the work is done in Perl and the approach is less rigorous than that used in subsequent projects. Our intention here is to introduce XML programming in the most simple and “exposed” form possible.

We have chosen to use Perl in this first part for various reasons.

It is the closest thing we know of to a lingua franca for internet programmers, it is extremely compact allowing us to construct complete examples in relatively few lines of code, and, most significantly, Perl is the most versatile XML scripting language.

The second project implements SGML/XML email and digs into the topics of entity management, catalogs, MIME, and full-scale SGML/XML parsing. Code is presented in Perl and C++.

Lest the reader think we are Perl bigots the third project plunges us into Java and XML, building an application based on the Document Object Model and making use of an Java XML parser API. Java

is the language in which most of the new XML internet infrastructure is being built.

The fifth and final section of the book takes a rigorous, formal look at the role of XML in software architectures and agents based on the paradigm of negotiation.

Full source code for all the projects has been included on the CD-ROM as have all the public domain tools used in the book.

1.4 | Let's Talk: Internets Are for Communicating

Shorn of its aura, the Internet is just a lot of sand-running programs and sending electrons on wires. However people do not knowingly spend enormous sums of money on such devices without some purpose in mind.

The essential business purpose of the Internet, never to be lost sight of, is communication. Before the computer, even before the printing press, there were international corporations with impressive networks spanning continents. The velocity of information was much different in those days, but the goals of the network haven't changed—to get important information to the concerned parties as quickly as possible and enable people to collaborate at a distance. Many of the problems and concerns which arise in the modern Internet already existed in the twelfth century: Security and encryption, network failures, baud rate (or gallop rate), common carrier versus private network, data format (written or spoken, English or French), gateways to other networks, and viruses (Bubonic plague, etc.) were all of importance back then as now, and one could look back even further. Their solutions may be of little current interest, but the similarities show that our concerns go beyond merely implementing the technology du jour and can give us a benchmark for judging our solutions.

If the Internet exists to support communication in an organization, what are the current requirements of this communication, and how do current solutions stack up against these requirements? There are

two essential characteristics of the modern corporate network (as opposed to twelfth- or early twentieth-century corporate networks) really mandating a move to XML. The first is the incredible increase in the velocity of information, and the second is a shift from a dumb network linking information processing humans to a smart network where information is as likely (or more likely) to be processed by a computer as by a human. We will examine each of these in turn.

1.5 | The Velocity of Information

It was perfectly acceptable in the slow-paced days of the early nineteenth century for troops on the fringes of empires to continue fighting wars months after peace treaties had been signed, simply because it could take that long for word to spread; nowadays information travels at the speed of light and must be accessible any time, any place, anywhere. In particular, this means information must be able to find you and be presented to you in as meaningful a way as possible.

To do this, the network not only needs to know how to find you, but it must also have some idea of the importance of the information it is passing around and of the equipment you have at hand. In the old days, importance was determined by a secretary and presentation meant a sheet of paper with ink on it. Now few secretaries remain, so the network, or some components, must be able to examine and partly understand messages. The equipment at hand might still be paper, but it is more likely a workstation, a laptop, a PDA, a beeper, or even an audio response unit over the telephone. The variety of possible equipment means it must be possible to present the same message in a variety of formats.

1.6 | Into the Smart Network

While humans communicate with words, companies communicate with documents. Contracts, SEC filings, advertisements, memos, purchase orders, all of these are documents an organization uses to communicate both with other organizations and internally with itself.

Premodern networks communicated only among humans, and in the early computer age it was pretty obvious which nodes in the communication system were humans and which were not. No accounting program was likely to pass itself off as human. In corporations, computer usage was confined mostly to database and transactional applications.

However, as the power of computers increases, computers become capable of more flexible tasks. We will soon see computerized agents afloat in the corporate network becoming more and more central to a corporation's ability to respond quickly and accurately to changing circumstances. These agents will receive and process orders and invoices, schedule appointments, and perform other complex tasks integrated with humans. Their sophistication is limited only by their ability to understand and manipulate the information presented to them, in turn limited by our ability to program them.

The existence of these agents is inevitably driven by the increasing velocity, and amount, of information. Smaller decisions may no longer be delayable until a human can examine information; people may need to search through huge amounts of information and require agents to help sift it down to a manageable amount. In many cases there is a tremendous time savings if humans can pass information directly to their computers. For example, if an e-mail contains a meeting proposal, the proposed time should be checked automatically against the recipient's current schedule. Other kinds of interesting

information, such as the biographies of the participants, should also be readily at hand. A decision to have the meeting should automatically start travel arrangements. Alternatively, incoming documents, such as SEC document filings, will be handled first by computers, then later by humans, who may then request further handling from computers, as well as sending the information to other humans.

Documents have traditionally been only human-readable; they have been used exclusively to pass information among the human participants of a corporation's networks. With the increase in computer power and performance, this view is too limited. For computers to do their tasks, they need to understand much the same information as the humans. Most document technologies, such as word processors, developed by and for humans, result in documents that are "illegible" to computers. Computers require large-scale artificial intelligence front ends to understand these, making agents difficult to write. Any document technology that bridges the gap between the human structures and the kinds of structure computerized agents need to act on will become strategically important. XML bridges the gap.

1.7 | Current Approaches— Can the Web Help?

Recently two approaches to distributed technologies, hypertext (represented by HTML) and object-oriented programming (represented by Java), have collided to provide the technological substrate for the current World Wide Web architecture. Unfortunately they have collided just where they are most incompatible. HTML is reasonably useful for displaying hyperlinked information to humans, while Java, an object-oriented language in the style of C++, functions well for controlling computers. But they don't talk to each other. Let's examine each.

1.7.1 *HTML*

Hypertext Markup Language (HTML), first developed in 1989 by Tim Berners-Lee at CERN and since extended in both a standard and ad hoc fashion by numerous programmers and researchers around the world, is the undisputed lingua franca of the Web and has introduced the concept of distributed hypertext to people around the globe. Despite this success, the language started out with simple goals; in many ways it is a paradigmatic example of a prototype too quickly made into a product.

We stated at the beginning of this book that HTML is an SGML application. SGML applications can and do have a wide variety of intended purposes; the purpose of HTML is to enable the transmission and display of hypertext documents across a network. (A hypertext document is a document which contains explicit links to other documents. On a command from the user, such as a mouse click, a computer can retrieve and display these other documents. Although many documents contain links to other documents, such as footnotes, hypertext is different because the links can be automatically, and immediately, traversed by a computer.) Unlike the majority of SGML applications, it is not suitable for representing the structure of the information contained in the document for purposes other than display. A typical HTML document (we will deal with exceptions later) is a string of text separated by *tags* (Figure 1-1). For the most part, the text specifies *what* to display, while the tags specify *how* to display it. There are additional tags to specify placement of pictures and tags to specify hyperlinks. An HTML document is like a standard Word Processing (WP) document with links added. WP documents are also strings separated by tags, except that the tags are usually unreadable control characters. In both cases the tags specify the layout of the document—where pieces of text and graphics are to be placed, which font to use, what size, color, and so on. In essence, HTML and WP documents are specifications for how to draw a picture on a screen or piece of paper for viewing by a human. HTML provides the

additional functionality of hyperlinking to automatically retrieve other documents when requested by the user.

```
<!DOCTYPE HTML PUBLIC "-//ISBN 82-7640-037::WWW//DTD
HTML//EN//2.0" "html.dtd">
<HTML>
<HEAD>
<TITLE>Sane Corporation: How to Contact us</TITLE>
```

Figure 1-1 HTML Fragment

However this picture-oriented view of documents is very limited. Any number of perfectly reasonable tasks are difficult to impossible:

- Automatically find all the section titles in a document. This appears like an easy task, but HTML's heading tags are identified in authors' minds with a particular appearance, used wherever the desired text appearance is desired. Collecting all the H1, H2, and so on, tags in a document may retrieve all the section headings but might gather a lot more lines as well. In addition, font tags can also be used for sections.
- Automatically number document sections. Doing this requires solving the previous problem.
- Break a document into its constituent parts for storage in a database. As with finding section headings, HTML gives little idea of the document structure; there is no obvious way to break it up.
- Radically reformat the document for other purposes besides Web browsing. For example, a document might be part of a book or magazine as well as be displayed on-line.
- Hand a document, or part of a document, such as a schedule or purchase order to a specialized agent for further processing. This kind of capability is particularly useful for business forms.

- Specify in a systematic way what kind of link a URL points to, that is, how does it fit into the document at hand.
- Instantly change the way a particular item of information is displayed every place it occurs among your documents—without changing a single document.

The basic point of these examples is to show that HTML is good for little beyond its original purpose of showing a particular picture to humans on a particular type of screen. This makes HTML a very brittle format, particularly when one considers the large scaffolding necessary to support this particular task in an industrial setting.

HTML is particularly bad at communicating information to computational agents. While humans can communicate by sending HTML, humans and computers can barely communicate through the use of HTML forms; it is ludicrous to consider computational agents sending each other HTML documents as a means of communicating—they wouldn't know what to do besides draw a picture.

1.7.2 *Java*

The Java programming language, developed by James Gosling and his team at Sun Microsystems, is a fairly standard object-oriented programming language with a couple of very interesting features, making it ideal for developing computer programs to be delivered over the Web.

In object-oriented programming, the application is composed of many objects sending each other messages. Similar objects are organized in groups called *classes*. Each class accepts a specific group of messages. To create objects in a class that accept other messages, a *subclass* can be created with some additional messages. In most object-oriented languages, including Java, each message is handled by a chunk of computer code called a *method* (essentially a func-

tion), and sending a message is called *method invocation*. The contents of a message is a combination of program values, strings, references to other objects, and so on, corresponding to the values the method is expecting. For example, an e-mail object would have methods to retrieve senders, receivers, and contents. A multimedia e-mail would have additional methods to handle the different kinds of media it could contain.

What makes Java so interesting for Internet applications are

- Its use of a byte-code interpreter, allowing the same program code to be run on any workstation in the network, and
- Its run-time linker, allowing a client to dynamically load classes from anywhere on the network to create new objects.

Extensions to the basic language allow objects to invoke methods on objects residing on remote servers.

What interests us here is the manner in which Java objects communicate—by method invocation. As with other OO languages, the Java world is divided up into objects, and each object belongs to a class. The class defines a set of messages that member objects are able to receive and process. Method invocations in Java, as in C++, are similar to procedure calls in more traditional languages. Figure 1–2 gives a tiny example of a Java class.

Just as HTML is an acceptable means of communicating among humans using browsers, but lousy for communicating with computer programs, method invocation is a good means for software objects to communicate among themselves but lousy for communicating with or among humans, even if they are professional programmers. Just as it would be ludicrous for objects to communicate by passing HTML, it is equally ludicrous for humans to communicate by method invocation with binary data. Humans don't reach into each other's heads to manipulate objects when they communicate.

```
class HelloWorldClass {
    String message;
    public static void main(String [] args){
        HelloWorldClass instance = new HelloWorldClass("hello
world");
        instance.method();
    }
    public HelloWorldClass(String msg){
        message = msg;
    }
    public void method(){
        System.out.println(message);
    }
}
```

Figure 1-2 Java

1.8 | Where the Web Needs Help

The core problem of the current set of Web technologies is a lack of closure. HTML is acceptable for displaying information to humans, but not to computational agents. Method invocation [either Java, or the industry-standard Common Object Request Broker Architecture (CORBA), which is extending into the Internet with the IIOP protocol] provides a mechanism for fairly fine-grained communication among computational objects, but not among humans.

Each of these technologies is effective for communicating among a subset of the participants in the Internet, but none is capable of handling all of them. Handling the mismatches among them is a major programming effort, as each, to some degree, requires reinventing the wheel. These incompatibilities remain a major impediment. In a sense, we are in the position of application development before the invention of the database, when each application kept its own files and passing information among them was very difficult.

There is, however, one technology out there which has the possibility of bridging the gap between humans and computers. This is XML (and its parent SGML), designed to encode information in ways compatible to both man and machine. To really see how this works, though, we need to reconsider the meaning of a document.

1.9 | Beyond the Traditional Document

Documents are traditionally viewed in information systems as pictures shown to the user at the leaves of the network. Communication with users is through these pictures, which are not directly treatable by machine, and communication among programs is through means incomprehensible to humans (i.e., HTML versus Java). This relegates humans to receptive leaves and reduces their ability to impact the total system. And yet, the same information is being passed around in two incompatible forms (again, HTML and Java). By unifying how we pass information to humans with how we pass information to computational agents, we can remove humans from the periphery of the network and place them in the center.

We can unify these diverse technologies by viewing documents as *intentional* messages sent by agents (human or computational) out into the world to have some effect. In some cases, the desired effect is merely to convey information. In others it is to purchase goods and services, gather people together to meet, schedule production, and the like. From this perspective, sending the memo, document, or method invocation, *is* the action of offering, requesting, or informing, and not just a reflection of it, with the actual event done entirely behind the scenes in software. These messages can be passed among any parties in the network, as well as stored and later referenced.

This view of communication has its origin in *speech act theory*, first suggested by the English philosopher John Austin in his book *How to Do Things with Words* and later extended by the American philoso-

pher, John Searle. More recently speech act theory has been influential in a number of areas of computer science, particularly artificial intelligence and groupware. The basic idea is a break with traditional logic, which views speech as only making statements about what is true or false in the world, and adopt a more active view where statements, themselves, can actually do things. The classic example is wedding vows. Once the couple has said “I do,” they are married; two small statements said in the right context have changed the couple’s status in the world and created a whole new set of rights and obligations. There are a variety of speech acts, such as promising, offering, requesting, and informing. We are less interested here in the exact list than in the viewpoint.

As we can see, our documents already behave as speech acts, but our computer systems only support this in the most tortured fashion. The goal of the next section is to begin to show how to implement this richer concept of document.

1.10 | Toward the Active Document

We can view documents as coming in three levels of structure:

1. Unformatted, or very loosely structured.

Examples of such a document include e-mail and pure images. The structure is understood only by a human reader through his or her knowledge of grammar or the picture presented. There is little information for a computer to manipulate. They are extensions of older kinds of communications such as talking and drawing; they are perfect when humans communicate informally.

E-mail can convey a certain amount of structure through the use of indentation; however it is limited through reliance on a single font. E-mail will normally

have the underlying structure of the human language it is written in, easily understood by human recipients willing to read it.

2. Formatted documents.

These are typically word processed documents or forms. The presentation of the document is well specified, so the appropriate computer program can correctly display this document, but no more than that. Such documents are extensions of traditional print media, which themselves extended unformatted text with visual cues to aid readers. Unlike e-mail, formatted text can convey many visual cues using color, point size, pictures, and so on. Word processed documents, however, are very much like pictures, so the recipient can understand some of the structure without reading. The processing instructions tell the computer how to paint the picture of the document (formatting some text as centered, bold, 14 point, helvetica) but don't say what the picture means (a chapter title). This makes formatted documents extremely difficult for computers to understand beyond display, especially when the same instruction string (i.e., identical formatting) can mean different things in different places. HTML essentially belongs in this category.

3. Structured documents.

These are documents where the document itself contains enough added information for either a computer or a human to easily understand the underlying structure. For this to be possible,

- The markup should use names that are *semantically significant*, meaning they designate what something *is* (such as title, section, part, etc.), rather than how to

display it as a picture (bold, helvetica, centered, etc.). Although a computer doesn't care what labels are given to markup, the humans programming the computer do. We can distinguish the semantics into two types, the *abstract* semantics, given by the element type name, and the *operational* semantics, which is what the program does with the information. Formatting information for printing is one kind of operational semantics, often derived from the abstract semantics (for example, we may associate the formatting semantics "24 pt. helvetica, centered" from the fact that the abstract semantics specifies a "Chapter Title").

- The markup used must delineate the logical structure of the document, so a *book* element contains a *title* followed by one or more *chapters*, each containing some number of *paragraphs* and *figures*, and so on.
- The markup must be *grammatical*. This means there is a set of rules available specifying how elements relate to each other, which elements can contain other ones, and what order they come in. Otherwise there is no way for the computer (and its users) to be sure that the title comes first in a book, and that chapters cannot, themselves, contain books.

Using this kind of hierarchical structure makes it possible for a computer to parse the pieces of a document and process them. The document becomes one large data structure, with these tags identifying each piece and what composes it. These structured documents subsume the two other types, as minimal markup could just indicate a string of ASCII text, or the markup terms could expressly indicate presentation format.

Address as unformatted text

Matthew Fuchs
 110 Some St.
 Anytown, CA 91100

```
{\rtf1\ansi\ansicpg1252\uc1
\deff0\deflang1033\deflangfe1033{\fonttbl{\f0\froman\charset0\prq2{\*\panose 02020603050405020304}Times
New Roman;}{\f16\froman\charset238\prq2 Times New Roman
CE;}{\f17\froman\charset204\prq2 Times New Roman Cyr;}{
\f19\froman\charset161\prq2 Times New Roman
Greek;}{\f20\froman\charset162\prq2 Times New Roman
Tur;}{\f21\froman\charset186\prq2 Times New Roman Bal-
tic;}}{\col-
ortbl;\red0\green0\blue0;\red0\green0\blue255;\red0\green
255\blue255;
\red0\green255\blue0;\red255\green0\blue255;\red255\green
0\blue0;\red255\green255\blue0;\red255\green255\blue255;\
red0\green0\blue128;\red0\green128\blue128;\red0\green128
\blue0;\red128\green0\blue128;\red128\green0\blue0;\red12
8\green128\blue0;
\red128\green128\blue128;\red192\green192\blue192;}}{\styl
esheet{\nowidctlpar\widctlpar\adjustright \fs20\cgrid
\next0 Normal;}{\*\cs10 \additive Default Paragraph
Font;}}{\info{\title Matthew Fuchs}{\author matthew
fuchs}{\operator matthew fuchs}
{\crea-
tim\yr1998\mo1\dy22\hr2\min23}{\revtim\yr1998\mo1\dy22\hr
2\min24}{\version1}{\edmins1}{\nofpages1}{\nofwords0}{\no
fchars0}{\*\company }{\nofcharsws0}{\vern71}}\widowc-
trl\ftnbj\aeenddoc\form-
shade\viewkind4\viewscale100\pgbrdrhead\pgbrdrfoot \fet0
\sectd \linex0\endnhere\sectdefaultcl {\*\pnseclvl1\pnu-
crm\pnstart1\pnindent720\pnhang{\pntxta
.}}{\*\pnseclvl2\pnu-
cltr\pnstart1\pnindent720\pnhang{\pntxta
.}}{\*\pnseclvl3\pndec\pnstart1\pnindent720\pnhang{\pntxt
a .}}{\*\pnseclvl4
```

Figure 1-3 Document Taxonomy —The Three Levels of Structure

Address as formatted text in RTF

```

\pnlcltr\pnstart1\pnindent720\pnhang{\pntxta
)}}{\*\pnseclvl5\pndec\pnstart1\pnindent720\pnhang{\pntxt
b ({\pntxta
)}}{\*\pnseclvl6\pnlcltr\pnstart1\pnindent720\pnhang{\pnt
xtb ({\pntxta
)}}{\*\pnseclvl7\pnlcrm\pnstart1\pnindent720\pnhang{\pntx
tb ({
{\pntxta
)}}{\*\pnseclvl8\pnlcltr\pnstart1\pnindent720\pnhang{\pnt
xtb ({\pntxta
)}}{\*\pnseclvl9\pnlcrm\pnstart1\pnindent720\pnhang{\pntx
tb ({\pntxta )}}\pard\plain \nowidctlpar\widctlpar\
adjustright \fs20\cgrid {Matthew Fuchs
\par 110 Some St.
\par Anytown CA, 91100
\par }}

```

Address as structured text - XML

```

<address>
<name>Matthew Fuchs</name>
<street>110 Some St.</street>
<state>CA</state><zip>91100</zip>
</address>

```

Figure 1-3 Document Taxonomy (*continued*)—The Three Levels of

It is only the last type, structured documents, which can really function as general intentional messages among humans and computers. There are some significant differences between these messages and, say, e-mail messages:

- They contain a significant amount of structure. A fair amount of explicit structure is necessary for messages to be manipulated in any meaningful way by computer systems. Although hope for AI springs eternal, the

systems available in the near term can only take advantage of the most obvious implicit structure.

- The structure conforms to a publicly available grammar. This way, any party can retrieve a description of the message syntax and parse it. Without such a grammar, it is impossible to agree on what is a correct document.
- The receiver is free to apply his own set of operational semantics to the message.
- The message may contain or reference more than one type of abstract semantics. Each abstract semantic type corresponds to a subdocument whose structure is interpreted separately by the receiving application. For example, an announcement of a meeting may contain an agenda for that meeting. The announcement would be the containing structure but the agenda would have its own structure which could be interpreted independently by the receiver's software.

A good example of these, in fact, are computer programs, which are highly structured documents humans use to communicate with computers and with each other (if subsequent programmers are included). The structure is partly for the benefit of humans, since the final program is always the same object code. The public availability of the grammar permits the development of compilers, pretty printers, revision control systems, and other development tools. Each one of these tools has a different semantics for the information—compilers see it as instructions for generating object code while pretty printers or class browsers see it as formatting instructions.

Programming languages, however, are large and complex beasts. They are also too highly structured to be really useful for communications among humans. Our goal is to

develop far more manageable languages corresponding to the kinds of things your organization does.

1.11 | Down to the Nitty-Gritty

In the course of its business, a corporation may deal with a variety of different document types—purchase orders, memos, legal documents, and the like. Each of these represents a separate minilanguage the corporation must be able to understand, each with a different grammar and semantics.

It is not possible to escape this multilingual environment; your corporation speaks several dialects already. However, to keep order among these dialects, they should be defined so that the same software tools can be applied to all of them regardless of domain. This is accomplished through the use of a single metagrammar system for defining all the languages. This means that “under the hood” they all look the same; a metagrammar is essentially a grammar for defining grammars.

We now have the outline of a multilayer applications architecture. The components are

- The metagrammar itself. This is the system used to define the grammars for the various languages. In this book we will use one called XML because it has shown itself to be ideal for communication among humans and computers.
- The particular language definition for documents in the problem domain. In XML jargon, this is called a Document Type Definition (DTD). For our approach to work, this grammar must be *public*, so that all the applications which need access to it can find it.

- A particular document conforming to the language.
- A parser which, when given a DTD, can parse conforming documents. The parser recognizes the structural elements of the document and makes them available to the application which processes them.
- One of several potential applications which can work on the output of the parser. Each application applies a separate semantics to the parsed document, as discussed above.
- All the resources of the Internet.

These elements together form the kernel of the XML Internet architecture. We will show how all these pieces fit together through a small example. To do this we will discuss the core elements of XML and information markup; a more thorough tutorial on all of XML is coming up soon.

1.12 | What Do We Do with Documents?

Once we accept the fact that documents are the communication medium of the corporation, and the Internet is the channel by which they are passed around, our next step is to show how we do things with documents.

In the old world we wrote documents, we read them, we filed (or threw) them away, we sent them around, and occasionally we looked through our files and retrieved them. Sometimes we even took different actions based on the contents of the documents. Now we need to look at a wider repertoire of operations and create document-centric business applications by combining them. Some of these have been performed previously by computers; others were performed by the

humans using the documents. Now we need to make them all explicit in order to let our computers do them as well.

There are 11 basic operations we will discuss here. We will list them briefly and then describe them at greater length.

- Search through a document or a set of documents.
- Retrieve either from a database or through the Internet.
- Store a document in a database or in files.
- Transform a document from one document type to another.
- Link information in one document to information in another document, or even in the same one.
- Send/receive a document to/from another “agent” on the network.
- Compare two or more documents, or pieces from within a single document.
- Import/export documents between XML and some other format, such as Microsoft’s Rich Text Format (RTF).
- Interpret a document as if it were a program.
- Define a DTD or document language.
- Create a document either individually (using an editor) or automatically through an application.

While discussing these operations, we will refer to a small, but universally known, example: the organization chart (org-chart). The org-chart, which lists the employees of the corporation and chain of command, is fairly ubiquitous. In this scenario, which is intentionally kept simple and general, Widget Co. already has an internet with employee home pages on-line. They now wish to place the corporate org-chart on-line, with links to all the home pages. Employees, of course, will also want to be able to include the org-chart information in their own pages or to link to appropriate locations in it.

Figure 1–4 gives a short example DTD for an org-chart, Figure 1–5 shows the same DTD in a graphic version generated with Near & Far

Designer, and Figure 1–6 describes a very small organization using our org-chart DTD. For the sake of clarity (and brevity) we have exploited an SGML feature and defined element types with common context models together. In XML, each element type is defined separately, leading, in this case, to a much larger DTD. It is not necessary for the reader to understand the DTD in detail at this point. We provide a microtutorial in the following section which will enable you to identify the portions of it we reference in subsequent sections and all discussion in this chapter is at a functional rather than syntactic level. This DTD is simple but still adequate for expressing basic organizational information. We will see that this baseline structure can be extended in any number of ways depending on the particularities of the implementation for a specific company. Elaborations and augmentation of this DTD can be constructed so that those additions and elaborations will be systematically derived from the base DTD. A base DTD often serves as the vehicle for interchange information between organizations, each of which has developed its own more specialized applications for internal use.

```
<!ELEMENT business-unit (name,address,officer,
description,business-unit*)>
<!ELEMENT (name|address|position|phone|email)
(#PCDATA)>
<!ELEMENT officer (employee,(officer|employee)*>
<!ELEMENT employee (name,address,position,phone,
email,home-page?,responsibilities)>
<!ELEMENT (responsibilities|description)
(htext|essay)>
<!ELEMENT home-page EMPTY>
<!ELEMENT essay (title,section+)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT section (title?, p+)>
<!ELEMENT htext (#PCDATA|a)*>
<!ELEMENT p (htext|ul|ol)*>
<!ELEMENT a (#PCDATA)>
<!ELEMENT (ul|ol) (li+)>
<!ELEMENT li (htext)>
```

Figure 1–4 Org-Chart DTD

```
<!ATTLIST (a|home-page) href CDATA #REQUIRED>
<!ATTLIST (phone|email)
  visibility (internal|external) "internal">
<!ATTLIST address
  visibility (internal|external) "external"
  location (hq|other) "other">
```

Figure 1-4 Org-Chart DTD

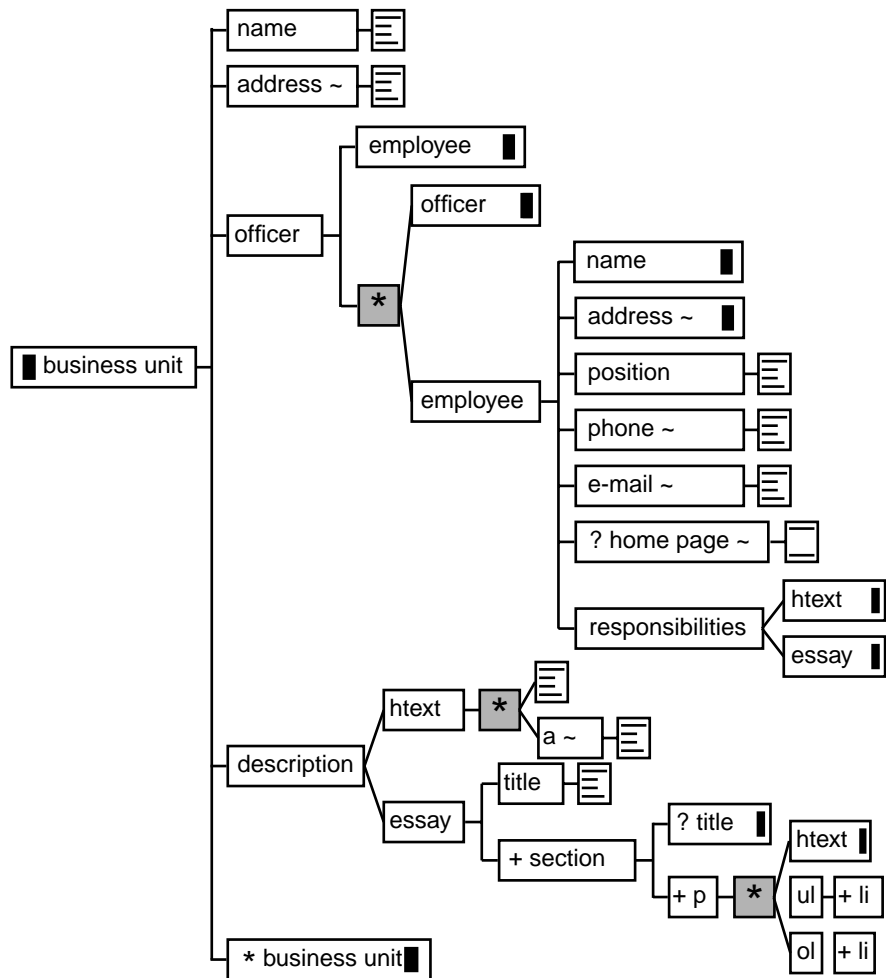


Figure 1-5 Org-Chart DTD—Graphic View

```
<business-unit>
<name>Sales</name><address>Main HQ</address>
<officer><name>John Bialystock</name>
<address>Corner Office</address><position>EVP</position>
<phone>951-555-1212</phone>....
</officer>
<description>They sell stuff</description>
  <business-unit> <!-- children -->
    <name>Consumer Sales</name>
    <officer><name>Alan Chuzzlewitz</name></officer>
    <address>In the Booneys</address>
    ...
  </business-unit>
</business-unit>
```

Figure 1-6 An Organization and Its Org-Chart

1.13 | DTDs and Content Specifications—A Short Excursion

A structured document is composed of a tree of elements. Each element can contain subelements or data. The entire document, such as a book, is a single element containing smaller elements, such as chapters, and they contain even smaller ones, down to the individual characters of the book's text.

The placement of the elements in this large, even enormous, tree is scarcely random—chapters cannot contain books, nor can footnotes contain chapters. Determining whether or not a document is a valid book requires rules stating which elements (if any) can go in other elements.

These rules create Book, a language for books, with its own grammar for generating or parsing books. While an English sentence has a noun, a verb, and an optional direct object, a sentence of Book may have a title, a list of chapters, and optional appendices.

As XML is traditionally used for documents, an XML grammar for a language such as Book is called a Document Type Definition, or DTD. The rule in a DTD which describes each structural component is called an element type declaration; the rule has a left-hand side, which names the element type being defined, and a right-hand side, called a content specification, which specifies the valid subelements. A DTD corresponding to the small book tree is in Figure 1–7.

```
<!ELEMENT book (front-matter, chapter+, index?)>
<!ELEMENT front-matter (title, author, dedication)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT dedication (#PCDATA)>
<!ELEMENT chapter (title, paragraph+)>
<!ATTLIST chapter revision CDATA #REQUIRED>
<!ELEMENT paragraph (#PCDATA)>
<!ELEMENT index (entry+)>
<!ELEMENT entry (phrase, location+)>
<!ELEMENT phrase EMPTY>
<!ATTLIST phrase text CDATA #REQUIRED>
<!ELEMENT location EMPTY>
<!ATTLIST location pageno CDATA #REQUIRED>
```

Figure 1–7 Book DTD

The exact anatomy of an actual element type declaration in a DTD is as follows:

- The opening string “<!ELEMENT” specifying this will be an element type declaration.
- The name of the element type being defined.
- The content specification specifying the subelement. An element can either
 - Be empty. This requires a content specification of EMPTY.
 - Contain only character data. This requires a content specification of #PCDATA (Parsed Character Data).
 - Contain only other elements.

- Contain mixed content, meaning both character data and other elements.
- A closing “>”.

In the actual XML document, the document tree can only be implicit; the document is just a string of characters. The hierarchical structure is created through the use of markup specifying the beginning and ending of each element. So the book element, for example, starts with the *start-tag* <book> and ends with the *end-tag* </book>; the chapter element with <chapter> and </chapter>.

```
<book>
<front-matter>
<title>Something about the Internet</title>
<author>John Doe</author>
<dedication>To my loving wife and adoring children
</dedication>
<chapter revision = "10"><title>Isn't the Internet Wonder-
ful</title>
<paragraph>What could be as wonderful as the Internet?
Look how the
whole world is abuzz! And what's at the middle of it all?
XML!</paragraph>
<paragraph>... more good stuff on XML ...</paragraph>
... more paragraphs ...
</chapter>
<chapter revision = "5"><title>HTML, or the Ways of our
Ancestors</title>
<paragraph>HTML is the primitive hypertext format which
helped spark
the Web</paragraph>...
</chapter>
<index>
<entry><phrase text="xml"><location pageno = "1"><location
pageno =
"2"></entry>...
</index>
</book>
```

Figure 1-8 XML Markup

The final XML concept we will need are attributes. Attributes allow one to annotate elements with additional information. Attributes are placed inside the element's start tag, as in Figure 1–9, where the chapter tag has a “revision” attribute. Attributes are defined by another type of DTD declaration, the ATTLIST. For the sake of brevity the syntax of an ATTLIST declaration is not described here but an example of an ATTLIST is shown in Figure 1–10.

```
<chapter revision="10">
```

Figure 1–9 Attributes Modify Elements

```
<!ATTLIST chapter revision NUMBER #REQUIRED>
```

Figure 1–10 ATTLIST

1.13.1 Search

Whether represented in XML or not, your documents are part of the greater corporate resources. And wherever there are data there is a need to search them. Queries on unstructured documents are limited in complexity to keyword search or string matching. It is possible to determine proximity of different phrases, but nothing as specific as:

- The first sentence in the third paragraph of the last chapter. This type of query is essential to any kind of group editing.
- All the meetings mentioned in memos to the president of Wild Widgets, Inc.
- All the lines Mercutio says to Romeo in *Romeo and Juliet*.

In the org-chart example, we might want to find out:

- For whom does employee X work? (Alternatively, does employee X work under employee Y?)
- How many employees work under X? What are their names?
- What department does X work in? (Alternatively, who is in charge of X's department?)

```
<business-unit>
<name>Sales</name><address>Main HQ</address>
<officer><name>John Bialystock</name>
<address>Corner Office</address><position>EVP</position>
<phone>951-555-1212</phone>....
</officer>
<description>They sell stuff</description>
  <business-unit> <!-- children -->
    <name>Consumer Sales</name>
    <officer><name>Alan Chuzzlewitz</name></officer>
    <address>In the Booneys</address>
    ...
  </business-unit>
</business-unit>
```

Figure 1-11 Answering the Query “Who does Alan Chuzzlewitz work for?”

These kinds of queries are traditionally performed against databases, not documents, and the current Web effectively separates the information to be displayed from the information to be queried, thereby requiring two or more technologies: HTML for display, forms for canned queries provided by the server, and possibly a Java interface. Suppose a client receives the org-chart in HTML. Some questions, such as the name of an employee's supervisors might be simple to answer from looking at the screen, but others, such as how

many people are in a particular department, are time-consuming to calculate solely from searching by hand. If the answers to these questions are not explicitly placed in the document, then either an HTML form or Java applet must perform the query. However, if the query is not one of a standard few, the user will need to know the database schema to compose the query and have general access to the database to perform it. As it is hard to anticipate all possible queries, it is quite likely that some desirable functionality is simply unimplemented.

If the org-chart is structured, however, many queries can be processed directly against the document. The information to display conforms to the same “schema” as the information to be queried. The same query can be processed either by the client or the server; the server need not implement all possible queries or make available all resources. A client can make queries against the org-chart by traversing the document, either up or down, without any server processing, except to return additional documents. Of course, the org-chart for a large multinational corporation would be several megabytes, but there are ways to divide up the document to require only portions of it.

Some of the queries possible with structured documents seem to require advanced AI technology not easily available to many companies, but that’s not the case. AI is necessary when the document’s structure provides no assistance to the query engine. This is particularly true when the document is just natural language. In a structured document, the markup is designed to highlight the structure by putting the semantics directly into the syntax, so the complexity of analysis is reduced to manageable levels. There is an upper limit to the amount of structure that can easily be imposed on a document intended for human consumption, but one of the nice aspects of XML is the ability to mix both structured and unstructured information, so a string of text with no explicit markup could also contain a chunk of highly structured information, such as a mention of a meeting in a paragraph of text.

1.13.2 *Retrieve*

Retrieval is closely linked to search but remains a separate, occasionally complex, activity. Both database and XML worlds share the concept (sometimes ignored for efficiency) that any particular piece of information should be stored only once, no matter how many other pieces it is linked to. Any particular document may be composed of a number of pieces to be (re)assembled for the finished product.

As an example, consider retrieving a form letter sent to a client. No copy of the letter itself may exist on-line, it was present in your system only for the few moments it took for it to be automatically composed and sent to the printer. Nevertheless, your system maintains a program or other skeleton which indicates how the various pieces can be reconstructed. The query “find me the letters sent to the Widget Corp. on June 23” may result in a number of records which are used, in turn, to retrieve the various pieces composing the actual letters.

Even where there is also a scanned version of the document stored locally, having access to the marked-up version remains important. It is more difficult to manipulate either the scanned version or a text version generated through Optical Character Recognition (OCR) than to manipulate the marked-up original. The marked-up version provides all the advantages of the structural search mentioned above, while the OCR version is no better than e-mail.

In the case of the org-chart, we must consider the potential size of the document. The complete document could comprise several megabytes of storage. Although the storage cannot be avoided, the time to load, format, and display the document for each retrieval can be controlled; only as much of the document should be loaded as is necessary.

With the org-chart we can deal with size issues by dividing the total document hierarchically. The topmost “hub” document contains only the upper echelons of the corporation. Departments and other subordinate entities are stored separately and only retrieved as required. Because the DTD defines both the entire organization and each department as a *business entity*, each department can exist as either an

independent document or as part of the larger document. A particular retrieval might access only a particular office of a department, rather than the entire document. This can be implemented either directly by the document management system (in which case it should be transparent to the user), or by extending the DTD to explicitly describe how the pieces fit together, a subject which will be explored further in the section on hyperlinking.

1.13.3 *Store*

Storing a document is the obvious flipside of retrieval, but these are not just mirror operations. When retrieving a document, various pieces are assembled. When storing a document, it is broken up again. How to do this is not obvious: the seams among the original pieces may no longer be evident or may be inappropriate (if it is a document received from elsewhere, the original pieces may not exist locally). The simplest solution is to store the whole document in one chunk, but this can have drawbacks.

If you consider the document part of your corporate information resources on a par with more traditional forms of data, then the document must be stored in a way which makes it easy to search and otherwise manipulate. Our ability to do this is vastly increased by using a database. However, it must always be possible to resurrect the original document from its dissected form. So we need to retain a skeleton of the original document indicating where the various pieces have been placed. This could be automatic in some databases, but if not, must be done explicitly.

One example of the flexibility of information markup is the possibility to create a DTD for documents which describe how to break up other documents, which can be transmitted with the original document. This could be particularly useful if certain pieces are just “boilerplate” or are time-sensitive and should always be regenerated on the fly.

1.13.4 *Send/Receive*

Sending and receiving documents are analogous to storing and retrieving, as the storage system can be seen as just another agent. In practical terms, however, they are quite different. When sending or receiving, we are quite possibly sending documents over the network, so it is important to determine just what to send and how to package it so the document can be reconstructed on the other side.

In our scenario, changes in the org-chart can trigger memos to be sent to partners and employees, probably via e-mail.

Of course, sending the document as raw XML may not serve the needs of the recipient. If the org-chart uses a standard DTD, known to the recipient, then they can use their existing software, but if not, the sender must either send it with the document, or inform the receiver where to look for it.

In an internet, some of this problem can be mitigated by ensuring that standard organizational entities are present at all locations. Another approach is through standardizing on an XML MIME type. MIME allows the creation of multipart e-mail messages. XML MIME messages can contain all the pieces, such as entities, necessary for reassembling a document. Defining all possible pieces as URLs ensures that the recipient can eventually find everything necessary.

Beyond reconstructing a document, the other interesting question for the recipient is deciding what to do with an incoming document. It can simply be routed for display to a human, or it can be manipulated by an automated agent and stored in a database, or otherwise acted upon. For example, an incoming purchase order can be printed in a less automated organization or automatically processed to produce the corresponding work orders in a more sophisticated one.

1.13.5 *Import/Export*

It is unfortunate that not all information is encoded in some form of XML. Therefore it is occasionally, sometimes even frequently, neces-

sary to convert between XML and some other format. Very common is conversion between XML and some word processing format, such as RTF or Postscript. Another issue is the exact mechanics of storage and retrieval—exporting from XML to a database format and importing back in again.

Importing and exporting may not be necessary for our org-chart unless we need to import it initially if it is in a database or project management software before we convert it to XML.

There are other uses for import and export. An important one is electronic commerce—converting between a rich internal XML format and a narrower EDI format, such as EDIFACT or ASC X12, both international standards, but very limited to fixed format fields—sufficient, perhaps for encoding a transmission, but not for internal use.

1.13.6 *Type Transformation*

Type transformation means converting a document of one type (i.e., conforming to a particular document type definition) to one or more new documents conforming to one or more DTDs. Transformation usually takes place when you put your information to some kind of use.

The most common form of transformation is converting a document encoded with high-level information markup to one with low-level presentation markup as preparation for final delivery to a bit-mapped display or printer. In the current Web world, the low-level presentation markup of choice appears to be HTML, and there are both public domain and for-profit tools for converting from a private DTD to HTML. If your company needs to deliver information in more than one format, such as for internal publication, manuals, instructional guides, and on-line documentation, as well as on the Web, you may need to convert from your private DTD to several other, more presentation-oriented DTDs.

Sometimes this can have more than one layer. In the case of the org-chart, conversion to an HTML page is an obvious start, but what

should that HTML page look like? HTML lends itself well to a text-based version, perhaps with embedded lists but cannot easily handle a more graphical representation, such as a real chart. A graphical chart, such as corporations love to distribute after reorganization, would be very useful. The direct conversion from an org-chart document to a graphic, though, is not so obvious. However the org-chart is really a Directed Acyclic Graph, or DAG (like a tree, but a node can have more than one parent), and it is straightforward to convert from an org-chart to a different DTD for describing DAGs, such as in Figure 1–12 to 1–14. Since many kinds of structures (such as book contents) can be represented as DAGs, software to convert a DAG DTD to HTML or a graphic image would be very useful. Many documents can be transduced to a DAG, so the hard work of converting a DAG to a display format is spread over each of these, and transforming from any initial format to a DAG is much easier than going straight to display in one step.

Instance of Org Chart

```
<business-unit><name>Operations</name>
...
<business-unit><name>North America</name>
...
  <business-unit><name>Northeast</name>
  </business-unit>...
</business-unit>
<business-unit><name>Asia</name>
...
  <business-unit><name>Japan</name>....
  </business-unit>
</business-unit>
</business-unit>
```

Figure 1-12 Transformation (continued)

Org Chart as tree

```

<tree>
<node name = "Operations">
<children>
<node name = "North America">
<children>
<node name = "Northeast">...
</children>
</node>
<node name = "Asia">
<children><node name = "Japan">
...</children>
</node>
</node>
</tree>

```

Figure 1-13

Org Chart as Directed Graph

```

<directed-graph>
<nodelist>
<node name = "Operations">
<node name = "North America">
<node name = "Northeast">...
<node name = "Asia">
<node name = "Japan">
</nodelist>
<edges>
<edge tail = "Operations" head = "North America">
<edge tail = "Operations" head = "Asia">
<edge tail = "North America" head = "Northeast">
<edge tail = "Asia" head = "Japan">
...
</edges>

```

Figure 1-14

Type transformation also becomes important when you start to truly integrate documents with your corporation's workflow processes. An incoming purchase order may be converted internally into several work orders sent to various departments. Other documents, such as for scheduling and reporting, may be combined into a single document, with only abstracts remaining from each. In our org-chart example, we might want to generate memos for each person describing their position and responsibilities. Later in the book we'll explore some of the issues around the relationship between XML and the commercial workflow systems which are beginning to generate interest. We confine ourselves here to making the observation that the actual work which gets accomplished by a workflow is the transduction of information objects, that is, the transformation of knowledge into products and services. XML offers the most precise way to describe and create information objects and to make those objects available to process-oriented workflow software. It is therefore critical that your workflow software does not force you to use a proprietary and limited document data representation or, worse still, to lock your workflow data into program code.

1.13.7 *Hyperlinking*

One of the important capabilities provided by hypermedia is the ability to hyperlink arbitrary pieces of information. In hypermedia terminology, these pieces of information are called the *anchors* of the hyperlink. As hyperlinks are an essential way to express relationships among objects in your domain, you want a technology that expresses all the ones important to you.

Different implementations of hyperlinking lead to different capabilities, however. The WWW, for example, has a very limited model. It requires all hyperlinks to be placed explicitly in the source docu-

ment, and all hyperlinks to a particular location in the destination document require an explicitly named anchor in that document. This means all hyperlinks are unidirectional pointers from an HTML document to either a complete object (such as another HTML page, a Java applet, or graphic) or the interior of another HTML document. This limitation does not exist for XML.

XML can support far more general models of linking. One of the most complete implementations of linking is found in HyTime, a companion standard to SGML (much of which will probably become part of XML over time). In HyTime, a link may have any number of anchors, each filling a particular role. As these roles are not limited to *source* and *destination*, they can describe far more complex relationships than HTML links. XML has a baseline of hyperlinking functionality that is less sophisticated than HyTime but is still far richer than HTML.

The link element itself can be stored either in one of the anchors or in a separate document. Once the link element is physically separated from its anchors, the anchors no longer need to be XML documents. In XML we can use links in one document to talk about relationships among other documents. For example, a user's manual could link a program, sample input, and sample output together, each residing in a separate file. By giving this particular kind of relationship a name, such as `test run`, we can treat them as a class, so we can, for example, find all the test runs, or substitute C++ test runs for Cobol test runs.

At its most fundamental level, an org-chart specifies the manager-employee links among people. Our example DTD specifies the organizational hierarchy with individual home pages as anchors. An employee's home page (were that to be in a DTD with the appropriate markup) could, in turn link back to the employee's entry in the org-chart document. A complete corporate reorganization can be reflected by changing a single document, the org-chart. (This assumes links from home pages are followed at query time. If pages are cached, then a program would need to perform a pass through the home pages to bring everything up to date.) No one would need to go

through the laborious process of editing everyone's home page, possibly introducing errors on each one.

1.13.8 *Compare*

Comparing two documents can be a very difficult, but sometimes necessary, task. At the limit, it is just as difficult as comparing two programs to determine if they will have the same output, but it is often much simpler than that. Also, comparisons may need to be made only between sections of documents, as opposed to whole documents.

In many cases comparisons will be among different versions of the same document. The most common example of this is in revision control systems, such as are commonly used in software development. In these systems, members of a group working on a common set of documents can “check out” and “check in” documents. Whenever a revised version of a document is checked in, all the changes from the old to the new versions are recorded. In this way it is always possible to revert to a previous version of any document. As another example, by comparing before and after versions of the org-chart, we can determine who was hired, who was fired, who was promoted, and who changed areas. Similar comparisons can be used to determine editing changes. In the new field of electronic commerce, rival vendors will make bids and proposals. If these are documents conforming to the same DTD (or architecture, as will be explained later), the two bids can be compared automatically.

Comparisons which do not exhaustively compare the entire documents exist in combination with searching, as the areas to compare first need to be found.

1.13.9 *Interpret*

So far we have discussed many operations which can be performed on documents but have not yet discussed the operations that documents can perform on our information systems, or rather, the actions performed once the documents are understood, or interpreted. From this perspective, we can see a document as instructions for another program which will interpret it.

In the area of publishing, a document is normally interpreted as instructions to some kind of formatting engine, and the result of interpreting the document is a book. In a way, this subsumes our other operations, since they have all required having a document present to drive some process, so we have discussed many different ways of interpreting an org-chart. This is one of the great strengths of XML—the ability to apply a variety of different semantics to the same information according to the different ways it is needed.

However, there are applications where the document is treated more closely to a program. Forms are a very good example of this. In HTML, forms are just a set of data entry fields with optional labels. Each field corresponds to some piece of data the server requires of the client, but (as usual) it is not obvious to the client's program what these data are. Therefore it is always necessary for a human to interpret the displayed fields and enter the information.

If, however, there is a DTD describing the kind of information in the domain, a form can be a specification for exactly which data items it requires. A style sheet could display this to a human with a series of blank fields, but a local computational agent could also read the form, understand what information is requested, immediately retrieve it from a database without requiring any human intervention, and return the filled-in form to the server. Medical insurance forms are an important example of this. Once the kinds of information requested in insurance forms is standardized in a DTD, applications at the service provider's facilities can automatically fill in forms sent by insur-

ance companies from the database. Where these applications do not exist, the insurance company can provide an applet to help a human perform the task manually. The two cases can be mixed, if there are questions of access rights to certain values, with an application filling in some and leaving the rest to a human.

1.13.10 *Define*

Data are not useful until they are structured and structure cannot exist until they are defined. The task of structuring corporate data has always been challenging, requiring specialized skills and knowledge. It has also always been acknowledged that a poor execution of that task may introduce expensive inefficiencies into the processing of data and can have an adverse impact on many aspects of business operations. For this reason a great deal of attention has been devoted to the science of structuring data: Countless Ph.D.s have been hired for work in this area, great competing schools of philosophy have sprung into existence, and an ocean of ink has been spilled in books on this subject.

The SGML community has devoted a great deal of energy to developing sound techniques for structuring your documents. This has been inherited by XML. The basic unit of document structure is, of course, the DTD. As in database design, creating a DTD is often an extended exercise in domain analysis, that is, the analysis of your business operations as it relates to your data needs. We'll discuss this later in the book and we'll point to some of the major reference works.

In addition to the techniques that have been developed, often called "document analysis," SGML is unique in having developed a large number of DTDs specific to various types of information (for example, technical publishing, software documentation, semiconductor data, aircraft maintenance, chemistry). It is often the case that you will be able

to start from one of these public DTDs and, not infrequently, you may find that the existing work is sufficient for your needs.

Also parallel to the database world, sophisticated tools exist to help you design or modify DTDs.

The org-chart DTD presented here is deliberately small, for pedagogical reasons. Actually, a great deal of work is done in XML with such small, simple DTDs—often a little structure may go a long way.

One of the most interesting developments in this area is the growing unification of database and XML design concepts. Object-oriented databases seem to be particularly suited for representing document types and the incentive for doing so has become compelling as SGML becomes more and more essential for defining and circulating information in the electronic age.

1.13.11 *Create*

Once database schemas have been developed it is, of course, necessary to capture the data which will be stored in the database. Database Management Systems (DBMSs) typically provide a variety of tools, such as form builders, for this purpose. Form interfaces on the Web, that is, HTML forms, have begun to replace an older generation of client/server tools. The underlying objective of such tools is to present the data schema to users of the database in a way which shields the user from the complexity of the actual representation of the data in the database. The analogue to this in the XML world is the XML editor (currently adapted from existing SGML editors), a kind of form interface which adapts to the particular requirements of any DTD applied to it. The XML world is actually better off than the traditional database world in that there is only one schema language which is used to define document structures while database vendors each have their own schema language. Any XML editor can be used with any DTD.

There are other ways to create XML documents. Another advantage XML enjoys over databases is that XML documents can be created in any text editor. Many people, for example, use emacs to create XML. While the sophisticated SGML editors that are available are the easiest way to create XML documents, the fact that any user on any computing platform has the ability to create XML ensures that the technology is accessible to everyone.

Many documents, and probably most documents in the near future, are automatically generated by software from databases or in reaction to different events. (Many of these documents will be ephemeral responses to structured database queries and can't compete with humans on artistic merit. They are documents nonetheless.) Now that there are new Internet protocols based on XML, such as the Channel Definition Format proposed by Microsoft, the day is fast approaching when documents will routinely be exchanged unseen by human eyes.

With the advent of automatic document generation, on the one hand, and the development of smarter software on the other, we reach a point where computers and humans can communicate among themselves using a common language.

1.14 | Conclusion

This chapter has had two goals. The first has been to “take the high ground” in explaining XML almost from first principles by exposing its foundations and showing how it addresses the inescapable problems built into the current architecture of the Web. We hope from this you will better understand where generic markup, as exemplified by SGML and XML, differ from specific markup languages, such as HTML, and why the latter is inherently limited in its expressibility. We also hope you will appreciate how SGML can enable far more

sophisticated distributed information systems, working in conjunction with applications, whether Java or Cobol.

Assuming we had accomplished the first goal, our second was to give a perspective on how to use documents based on a number of fundamental behaviors that present a kind of taxonomy of document handling. In the age of the electronic document, document-centric applications will be complex, multistage affairs. Categorizing these stages into a finite set of behaviors gives you, the Internet document engineer, a set of tools you can use in building these applications.

The chapters that follow will be much less concerned with such lofty concerns and much more interested in getting down and dirty with real software and problems to show how these different behaviors work in practice. At the end you will be able to join them together to create real Internet applications.