

CHAPTER

# 11

## The Internet and XML

The Future  
of Metadata

It is critical that you understand the meaning of the data that goes into the warehouse. ... You also need to know if your data is a complete record of the organization's activity. Maybe the data you really need is not recorded in your database at all.

—IBM Software Solutions

Part 1 covered Enterprise Portal Design, while Part 2 addressed data warehouse development, for later evolution to Enterprise Portals. In Part 3 we focus on this evolution of data warehouses and their deployment as Enterprise Portals. We will examine technologies that offer new ways of delivering information from the data warehouse. This chapter addresses the Internet and intranets, and a vital technology for deploying data warehouses and Enterprise Portals: Extensible Markup Language (XML). Our brief introduction to XML in Chapter 1 will now be expanded to cover XML more completely in this chapter.

In earlier chapters we discussed that the only thing stable today ... is change itself. Organizations must structure themselves to respond rapidly to change. They must change to a market-driven and customer-driven focus—rather than be organization-driven or product-driven as in the past. New business process opportunities can emerge from this customer-oriented focus, with new processes crossing previous functional boundaries. These cross-functional processes can lead to dramatic breakthroughs with reengineered business processes. The Enterprise Engineering Portals (EEP) methodology uses XML for Business Reengineering in Chapter 12, and for integrated Business and Systems Reengineering in Chapter 13.

## Transformations of the 1990s

We begin this chapter by considering three major transformations, or shifts, that have occurred in the computer industry throughout the 1990s. Their impact extends far beyond that industry. They are also transforming business and society. They are moving us rapidly from the Industrial Age to the Information Age.

### The First Shift: The Internet

The First Shift has already occurred: the impact that the World Wide Web is having on business today. With the introduction of Web browsers in

the early 1990s, the Internet—already over 20 years old at that time—moved into the mainstream as organizations rushed to establish their own Web sites.

First-generation Web sites—using Hypertext Markup Language (HTML)—were used as billboards to the world. They provided static advertising and marketing information for the benefit of customers and suppliers. They implemented online information that was also available in print advertisements, or as documentation in book or manual formats. While effective with those static media, when transferred to a Web site they offered no benefit—only glitzy eye candy. These static Web sites also suffered from another disadvantage. While they were easy to visit, they were also easy to leave with the click of a mouse—when potential customers could not find what they needed.

Second-generation Web sites added interactivity and more content to provide further assistance. But alone, animated images or sounds and movie clips do not provide real benefit to visitors. They are still essentially “static” in their ability to bring real, bottom-line benefit to the business. They need to be integrated into the main purpose of the Web site—as demonstration aids, sales aids, or information aids, for example. When they provide this purpose-focused capability, they move their Web sites to the third generation.

Electronic Commerce sites that are extensively being established today are part of these third-generation Web sites. They have potential to generate major revenue and profit for the business. But many of these electronic storefronts are like the Lemonade stands of our childhood—the first tentative ventures into a New World of business. More is needed before the full potential of Electronic Commerce can be realized, as this and later chapters will explain.

## The Second Shift: Java

The mid-1990s saw the start of the Second Shift: the emergence of Java as a programming language able to be executed anywhere regardless of hardware platform or operating system. Java was first developed by a team led by James Gosling at Sun Microsystems in 1991. It was planned as a portable language that could be executed from embedded devices such as TV set-top boxes. But its potential to become a major programming language that could transcend the hardware platform and operating system dependencies of other languages was also recognized. This

saw the introduction by Sun in early 1995 of Java as a portable programming language. It was seen as the “Holy Grail of Computing”: a hardware- and operating-systems-independent language.

Java presented a potential threat to Microsoft, as it could offer an alternative operating environment to Windows and threaten its desktop monopoly. Microsoft therefore embraced Java, but it added extensions to use Windows-specific capabilities—so limiting the portability of the language. This was the subject of a suit brought by Sun against Microsoft in 1997, decided against Microsoft in late 1998. The legal judgment required that Microsoft remove its Windows-specific Java extensions within 90 days of the ruling.

Java today is being adopted widely as a major object-oriented language across the industry. Java virtual machines are now available for all major operating system and hardware environments. Java compilers are also available for most operating systems: desktop, server, and mainframe. The shift to Java is gathering steam, but it will be many years before its full promise of “write once, run anywhere” can be fully realized.

## **The Third Shift: Extensible Markup Language (XML)**

The Third Shift is the emergence of the Extensible Markup Language (XML) in the late 1990s. This shift is just starting. It promises to be as significant as the first two. It has the ability to bring real, bottom-line benefits to business—in cost reduction, in greater efficiency, in greater competition, and in greater revenue.

XML is one of the most significant developments of the computer industry since the World Wide Web and Java moved to their present positions of importance. For the next 2–5 years this will be one of the most important aspects of the Internet, and of systems development in general. It has the potential to move metadata and data administration also into the mainstream of systems development. XML will present major business opportunities, when used with the Internet, as a delivery channel for information from Data Warehouses and Enterprise Portals.

XML will be the successor to HTML for the Internet, intranets, and for secure extranets between customers, suppliers, and business partners. XML incorporates metadata in any document, to define the content and structure of that document and any associated (or linked) resources. It has the potential to transform the integration of structured data (such as

in legacy files or relational databases) with unstructured data (such as in text documents, reports and emails, graphics and images, audio and video resources, and Web pages). XML will be a significant technology for the deployment of Data Warehouses and Enterprise Portals.

XML uses the Extensible Style Language (XSL) and the Extensible Linking Language (XLL) to achieve this integration. We will see that XML, XSL, and XLL allow the easy integration of dissimilar systems for multiple worldwide customers and suppliers in any industry. It permits the ready integration of those systems, regardless of whether they are legacy systems and databases, Electronic Data Interchange (EDI) systems, or Electronic Commerce. It represents the future direction of metadata and the important role that data administration will take in systems development in the years ahead.

There are steps that you can take now, to prepare today for the coming shift to XML.

## **Preparing for an XML World**

XML assumes that your metadata has already been defined. This is necessary not only for the new systems that you want to develop, but also for the legacy systems and databases that you need to integrate with those new systems. XML will enable this integration to be carried out dynamically.

The knowledge of data modeling and strategic modeling that you gained in Part 1 from Chapters 3 and 4 will help you to define the metadata required by XML using these Forward Engineering methods of EEP. This will also enable you to eliminate redundant data versions and redundant processes, to develop integrated databases for the Internet and intranets. This is not just the responsibility of data administrators. It requires business knowledge also, gained by your knowledge of strategic business planning from Chapter 2.

The knowledge of the metadata types, metadata activities, and metadata capture techniques that you learned in Part 2 using the Reverse Engineering methods of EEP will also help you to extract the metadata from existing legacy systems and databases, or from relational or object databases. XML will enable you to combine reverse-engineered metadata from Part 2 with the forward-engineered metadata from Part 1, for the seamless structured and unstructured data integration that characterizes truly effective Enterprise Portals.

Interest in XML, metadata, and data administration will grow strongly. The XML specifications are now essentially complete [XML], while the XSL and XLL specifications were still evolving at the time of writing. These specifications are defined by the World Wide Web Consortium and are all available from their Web site [W3C].

Some browser support for XML was first included in Microsoft Internet Explorer 4.0. The Channel Definition Format (CDF) capability of Internet Explorer 4.0 was based on the use of XML. More complete support for XML is provided in Microsoft Internet Explorer 5.0 and Netscape Communicator 5.0. We will also see wide XML support added to DBMS products, to CASE tools, to Data Warehouse tools and also to Client/Server development tools. We will see a new generation of Knowledge Management tools evolve rapidly to take advantage of the structured/unstructured data integration opportunities offered by XML.

Several books provide good treatment of XML. An initial introduction to XML (and also Cascading Style Sheets) is provided by *XML: A Primer* [St Laurent 1998]. XML used for Web site development, with HTML, XSL, and XLL, is addressed in *XML: Extensible Markup Language* [Harold 1998]. *XML Complete* [Holzner 1998] covers the use of XML with Java. These can be used as detailed references in conjunction with the remainder of this chapter. *Web Farming for the Data Warehouse* [Hackathorn 1998] uses the Internet, intranets and XML for access to external data sources for warehouse deployment. This is addressed in more detail in Chapter 15, together with a discussion of XML-related products.

We will now examine XML concepts. In one chapter, of necessity this can only be an overview. More detail is available from the references above and at the end of the chapter. We will start with the initial purpose of XML, which was to provide a more effective capability for defining document content than that offered by HTML.

## Some Problems Using HTML

Tim Berners-Lee at CERN, the originator of the World Wide Web (WWW) in 1990, developed Hypertext Markup Language (HTML) as a subset of the Standard Generalized Markup Language (SGML). A standard for the semantic tagging of documents, SGML evolved out of work done by IBM in the 1970s. It is used in Defense and other industries that deal with large amounts of structured data. SGML is powerful, but it is also very complex and expensive.

HTML was defined as a subset of SGML—specifically intended as an open architecture language for the definition of WWW text files transmitted using Hypertext Transport Protocol (HTTP) across the Internet. HTML defines the layout of a Web page to a Web browser running as an open architecture client. Microsoft Internet Explorer and Netscape Communicator share over 90 percent of the Web browser market; both are now available free.

An HTML page contains text as the content of a Web page, as well as tags that define headings, images, links, lists, tables, and forms to display on that page. These HTML tags also contain attributes that define further details associated with a tag. An example of such attributes is the location of an image to be displayed on the page, its width, depth, and border characteristics, and alternate text to be displayed while the image is being transmitted to the Web browser.

Because of this focus on layout, HTML is recognized as having some significant problems:

1. *No effective way to identify content of page* HTML tags describe the layout of the page. Web browsers use the tags for presentation purposes, but the actual text content has no specific meaning associated with it. To a browser, text is only a series of words to be presented on a Web page for display purposes.
2. *Problems locating content with search engines* Because of a lack of meaning associated with the text in a Web page, there is no automatic way that search engines can determine meaning—except by indexing relevant words, or by relying on manual definition of keywords.
3. *Problems accessing databases* We discussed earlier that Web pages are static. But when a Web form provides access to online databases, that data needs to be displayed dynamically on the Web page. Called “Dynamic HTML” (DHTML), this capability enables dynamic content from a database to be incorporated “on the fly” into an appropriate area on the Web page.
4. *Complexity of dynamic programming* DHTML requires complex programming to incorporate dynamic content into a Web page. This may be written as CGI, Perl, ActiveX, JavaScript, or Java logic, executed in the client, the Web server, the database server, or all three.

5. *Problems interfacing with back-end systems* This is a common problem that has been with us since the beginning of the Information Age. Systems written in one programming language for a specific hardware platform, operating system, and DBMS may not be able to be migrated to a different environment without significant change or a complete rewrite. Even though it is an open architecture, HTML also is affected by our inability to move these legacy systems to new environments.

Recognizing these limitations of HTML, the W3C SGML working group (now called the XML working group) was established in mid-1996. The purpose of this group was to define a way to provide the power of SGML, while also retaining the simplicity of HTML. The XML specifications were born out of this activity [XML].

XML retains much of the power and extensibility of SGML, while also being simple to use and inexpensive to implement. It allows tags to be defined for special purposes, with metadata definitions embedded internally in a Web document—or stored separately as a Document Type Definition (DTD) script. A DTD is analogous to the Data Definition Language script (DDL) used to define a database, but it has a different syntax.

As we discussed earlier, data modeling and metadata are key enablers in the use and application of XML. The Internet and intranets allow us to communicate easily with other computers. Java allows us to write program logic once, to be executed in many different environments. But these technologies are useless if we cannot easily communicate with and use existing legacy systems and databases.

In Chapter 1 we used an analogy based on the telephone. We can now make a phone call, instantly, anywhere in the world. The telephone networks of every country are interconnected. When we dial a phone number, a telephone assigned to that number will ring in Russia, or China, or Outer Mongolia, or elsewhere. It will be answered, but we may not understand the language used by the person at the other end.

So it is also with legacy systems. We need more than the simple communication between computers afforded by the Internet. True, we could rewrite the computer programs at each end in Java, C, C++, or some other common language. But that alone would not enable effective and automatic communication between those programs. Each program must know the metadata used by the other program and its databases so that they can communicate with each other.

Considerable work has been carried out to address this problem. Much effort has gone into definition and implementation of Electronic Data Interchange (EDI) standards. EDI has now been widely used for business-to-business commerce for many years. It works well, but it is complex and expensive. As a result, it is cost-justifiable generally only for larger corporations.

XML now also provides this capability. It allows the metadata used by each program and database to be published as the language to be used for this intercommunication. But distinct from EDI, XML is simple to use and inexpensive to implement. Because of this simplicity, as discussed in Chapter 1 we like to think of XML as:

XML is EDI for the Rest of Us

XML will become a major part of the application development mainstream. It provides a bridge between structured databases and unstructured text, delivered via XML then converted to HTML during a transition period for display in Web browsers. Web sites will evolve over time to use XML, XSL, and XLL natively to provide the capability and functionality presently offered by HTML, but with far greater power and flexibility. XML components are listed in Table 11.1.

The rest of this chapter provides an introduction to XML and DTDs, with only brief coverage of XSL, XLL, DOM, and RDF. Further information in each of these areas can be obtained from the book and Web site references provided at the end of the chapter.

**TABLE 11.1**

Components of XML

Acronym	Name	Description
XML	Extensible Markup Language	Defines document content using metadata tags and namespaces
DTD	Document Type Definition	Defines XML document structure (analogous to DDL schema)
XSL	Extensible Style Language	XSL or Cascading Style Sheets (CSS) separate layout from data
XLL	Extensible Linking Language	XLL implements multidirectional links (single or multiple)
DOM	Document Object Model	Implements a standard API for processing XML in any language
RDF	Resource Description Framework	W3 Interoperability Project for data content interchange

## A Simple XML Example

We will start our introduction to XML with a customer example in Figure 11-1. This illustrates some basic XML concepts. It shows customer data (in italics), such as entered from an online Web form or accessed from a customer database. It shows the inclusion of metadata “tags” (surrounded by < and >)—such as <customer\_name>.

The tag: <customer\_name> is a start tag; the text following it is the actual content of the customer name: *XYZ Corporation*. It is terminated by an end tag: the same tag-name, but now preceded by “/”—such as </customer\_name>. Other fields define <customer\_address>, <street>, <city>, <state>, and <postcode>. Each of these tags is also terminated by an end tag, such as </street>, </city>, </state>, and </postcode>. The example concludes with </customer\_address> and </CUSTOMER> end tags.

From this simple example of XML metadata, we can see how the meaning of the text between start and end tags is clearly defined. We can also see that search engines can use these definitions for more accuracy in identifying information to satisfy a specific query.

Even more effective applications become possible. For example, an organization can define the unique metadata used by its suppliers’ legacy inventory systems. This will enable that organization to place orders via the Internet directly with those suppliers’ systems, for automatic fulfillment of product orders. XML is enabling technology to integrate unstructured text and structured databases for next-generation E-Commerce and EDI applications. We will see examples of this and other XML applications in Chapters 12 and 15. The project example we discussed in Chapter 10 will be developed further with XML in Chapters 12 and 13.

The following pages now examine the XML syntax in more detail.

**Figure 11-1**

A simple XML example.

```
<CUSTOMER>
  <customer_name>XYZ Corporation</customer_name>
  <customer_address>
    <street>123 First Street</street>
    <city>Any Town</city>
    <state>WA</state>
    <postcode>12345</postcode>
  </customer_address>
</CUSTOMER>
```

## XML Naming Conventions

An XML document must be “well formed.” To be well formed, a document must obey the following rules:

- A tag name must start with a letter or underscore, with no spaces. Thus *person\_id* is correct, but not *person id* or *1st name*.
- XML names are case sensitive. For example, *PERSON*, *Person*, and *person* are all different names.
- Each tag must have surrounding < and > indicators, as in the start tag `<tag_name>`.
- Each start tag must also have an end tag, as in `</tag_name>`.
- If a tag is empty, it must still have an end tag or empty tag such as `<CUSTOMER></CUSTOMER>` or `<country/>` (i.e. Empty).
- Attribute values are preceded by an = sign and are surrounded by double or single quotes, such as `version="1.0"` `standalone="YES"`.
- The characters <, >, &, “, or ‘ cannot be used in XML except when replaced by their “escaped” versions. Thus the character string `&lt;` represents “<” at all times until it is to be displayed. Similarly `&gt;` is “>,” `&amp;` is “&,” `&quote;` is “,” and `&apos;` is ‘. These character sequences are called “predefined entity references.”

A well-formed document example follows in Figure 11-2.

Notice that double quote characters in Figure 11-2 surround the attribute values of *PERSON*, declared on the first line with the values: `person_id="p1100" sex=".M"`.

## The XML Document Prolog

Every XML document starts with an XML declaration as part of its prolog. This declaration must be the first statement on the first line of the document. It is defined as a processing instruction (surrounded by `<? ... ?>` tags) such as:

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

**Figure 11-2**

Example of a Well-Formed XML Document.

```
<PERSON person_id="p1100" sex="M"> (Attributes in Element)
  <person_name> (Children of
    <given_name>Clive</given_name> "person_name"
    <surname>Finkelstein</surname> Element)
  </person_name>
  <email>cfink@ies.aust.com</email>
  <company>
    Information Engineering Services Pty Ltd
  </company>
  <country>Australia</country>
  <phone>+61-8-9309-6163</phone>
  <fax>+61-8-9309-6165</fax>
</PERSON>
```

The `<?xml` specifies that the document uses XML syntax. An XML parser or application can analyze the content of the document prior to it being processed. The tag “XML,” “xml,” or any upper and lower-case combination of this sequence of letters is reserved and cannot be used in any tag name.

The version number is specified for compatibility with future XML versions. The standalone specification indicates whether a Document Type Definition is included in-line (“standalone=yes”) or out-of-line in an external file (“standalone=no”). We will discuss this shortly in relation to *DOCTYPE Declarations*.

The “encoding” statement specifies the language-encoding format used by the XML document. XML has been defined so it can be used with any language, such as English, European, and Middle Eastern languages, as well as double byte Asian languages—Japanese, Chinese, or Korean.

## DOCTYPE Declarations

A Document Type declaration (“DOCTYPE”) immediately follows the `<?XML . . . ?>` statement. Every XML document contains a root name, which includes all other XML tag names. The DOCTYPE statement identifies the specific root name used by the document. It also identifies the location of the Document Type Definition (DTD) file that is to be used with the document.

A DOCTYPE declaration has the following formats, with examples:

```
<!DOCTYPE root_element_name [ . . . ]>
OR
<!DOCTYPE root_element_name SYSTEM "DTD_URL">
```

1. `<!DOCTYPE CUSTOMER [ ... ]>`
2. `<!DOCTYPE CUSTOMER SYSTEM "customer.dtd">`
3. `<!DOCTYPE supplier PUBLIC  
"http://www.ind-xml.com/supplier.dtd">`

The first example specifies that the DOCTYPE is declared internally in the same document. We will see an example of this format shortly.

The second example declares that an external DTD is used as a private file ("SYSTEM"). It is the DTD file that is located at the relative Uniform Resource Locator (URL) "customer.dtd" within the same Web site directory.

The third example specifies that the DTD is PUBLIC. It is the DTD file at the absolute URL "http://www.ind-xml.com/supplier.dtd".

## URL and URI

These DOCTYPE examples use relative or absolute URLs to identify the location of an external DTD file. But files and other resources can be moved to different URL locations. With HTML Web pages, every link that refers to a moved resource must be updated to refer to its new URL. HTML links can be from Web sites anywhere in the world. These can all refer to the same URL. Relocating a resource to a different URL can therefore require considerable maintenance work.

To overcome this problem, in time XML and XLL will enable resources to be located instead by a Uniform Resource Identifier (URI). Distinct from a URL, a URI can never change. XLL, with XLinks and XPointers, defines a URI. The URI always points to that resource. We cover these in *Extensible Linking Language*, later in this chapter.

## XML Comments

Comments can be used in an XML document to describe the purpose, intent, and use of different statements. Comments can also document and separate logical sections of a document.

Comments in XML are defined similarly to HTML comments, surrounded by `<!-- . . . -->` tags. For example:

```
<!-- This is a comment and is not processed -->
```

Comments can contain any data except the literal string “-->” but may not be placed inside an XML tag. In the next two examples, the first comment is wrong; the second comment is correct:

```
<customer_name <!-- Defines customer name --> >
  (incorrect-comment is inside tag)
<!-- Defines customer name --> <customer_name>
  (correct-comment is outside tag)
```

However, comments can be used to surround and hide tags, such as:

```
<!-- The following tag is used only for retail customers
<retail-code>2</retail-code>
and is ignored for wholesale customers -->
```

An XML parser or XML application cannot process the `<retail-code>` tag until the surrounding comment is removed, or until the tag is moved outside the comment. While it remains within the comment, for XML processing purposes the tag does not exist.

## Processing Instructions

Processing instructions (PIs) declare applications that will be used to process part (or all) of an XML document. Like comments, they are not part of the XML document. An XML processor must pass a PI unchanged to the relevant XML application. A PI has the format:

```
<?PI_target_name PI_data?>
```

The *PI\_target\_name* identifies the application. The *PI\_data* following the *PI\_target\_name* is optional; it is specified by and used by the PI application. We saw a PI example in *XML Document Prolog*. A document to be processed by an XML parser or processor was declared by the PI statement:

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

XML applications should process only the targets they recognize. PI names that begin with “XML”, in any combination of upper or lower case, are reserved for use in XML standards. PIs are used for document-specific application processing.

PIs can also utilize NOTATION declarations and NOTATION attributes. We will discuss these briefly later.

## CDATA Sections

A document may contain markup characters such as <, >, &, “, or ‘ that should be ignored by an XML parser or processor. An example is a source code listing within an XML document. To prevent characters in the listing from being recognized incorrectly as XML markup characters, a CDATA section can be used. For example:

```
<![CDATA [  
*x = &a;  
c = (i <= 5);  
]]>
```

The CDATA section begins with the string: “<![CDATA [” and ends with the string “] ]>.” All other character data between these strings is passed directly to the relevant application. Clearly, the only character string that cannot occur in a CDATA section is “] ]>.”

We will see later that an XML Attribute containing character data is declared with “CDATA.” This is not to be confused with the use of a CDATA section, which is intended only to isolate markup characters so they can be passed unprocessed to an XML application.

## XML Elements, Attributes, and Entities

XML defines metadata tags using elements, attributes, and entities. In the following sections we will first learn how XML uses these to declare metadata tags. We will then discuss how this use differs from data modeling. The chapter concludes by describing how CASE tools can be used to define data modeling entities, attributes, and associations, for automatic generation of XML declarations.

### Declaring XML Elements

The tags that we have seen are all examples of XML “elements.” An element is a named metadata tag that is declared in a DOCTYPE statement. As we have seen, a DOCTYPE can be defined externally in a DTD file, located using a relative or absolute URL. Alternatively, a DOCTYPE can be defined internally. It is included in-line, immediately following the

**Figure 11-3**  
DOCTYPE declaration  
for the Customer  
example in Figure  
11-1.

```
<?XML version="1.0" standalone="YES"?>
<!DOCTYPE CUSTOMER
[
<!ELEMENT CUSTOMER ANY>
<!ELEMENT customer_name (#PCDATA)>
<!ELEMENT customer_address EMPTY>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT postcode (#PCDATA)>
]>
```

XML processing declaration. Figure 11-3 shows an internal declaration of the DOCTYPE statement that is used with the customer example in Figure 11-1.

The DOCTYPE statement in Figure 11-3 defines the XML root name as `<CUSTOMER>`. Square left and right brackets (`[ ... ]`) follow, surrounding all element declarations. The DOCTYPE root name `<CUSTOMER>` is declared as an ELEMENT, specified to be ANY (case-sensitive). This indicates that any element, as well as parsed character data (shown as italics in Figure 11-1) can appear in a `<CUSTOMER>` element.

Each element must be uniquely named within an XML document. As a DOCTYPE declaration can be defined using more than one DTD, the concept of namespaces has been included in XML. This allows an alias to be assigned to a DTD. The namespace alias can then be used to qualify named elements that would otherwise violate this rule, so ensuring uniqueness. This is discussed later in the chapter.

The declaration of an XML Element in Figure 11-3 has the format:

```
<!ELEMENT element-name content_type>
```

As with all XML tags, the *element-name* starts with a letter or underscore, can have no spaces and all names are case sensitive. Thus *Customer*, *customer*, and *CUSTOMER* are different XML element-names. Because of this and to avoid confusion, it is recommended that an element-name be declared using a case-sensitive name that always refers to that same element. Once declared, that *element-name* is used as the *tag-name*; the terms *element-name* and *tag-name* are therefore synonymous.

The *content\_type* of an Element can have values of ANY, EMPTY, (`#PCDATA`), or a (Child List) as discussed next.

We discussed an example of ANY in relation to the root name element <CUSTOMER> in Figure 11-3. This indicates that any element, as well as parsed character data, can appear within it.

By default, tags are *non-empty* and are followed by data (see italics in Figure 11-1). An element is declared EMPTY if it normally has no data. For example, the element <customer-address> in Figure 11-1 contains <street>, <city>, <state>, and <postcode> elements within it. These are called child elements. As the parent element, the data for <customer-address> is provided by its child elements. The <customer-address> element is therefore declared in Figure 11-3 to be EMPTY.

Figure 11-3 declares the <customer\_name> element is (#PCDATA). This specifies that the element contains “Parsed Character Data.” In Figure 11-1, we now know that *XYZ Corporation* is character data that is parsed by an XML parser, or processed by an XML application.

Similarly, the <street>, <city>, <state>, and <postcode> elements in Figure 11-3 are also declared to be (#PCDATA). Note that <postcode> in Figure 11-1 contains the numeric characters: “12345,” not the numeric value. For example, consider the following element declaration and corresponding tag:

```
<!ELEMENT customer_balance (#PCDATA)>
.....
<customer_balance>$15,500.00</customer_balance>
```

Before the <customer\_balance> data of \$15,500.00 can be processed by an XML application, it must first be converted from the numeric characters “\$15,500.00” to the numeric currency value of \$15,500.00.

When we consider address data in an application, there are many variations. For example, in addition to the street number and name, some customers may have a floor or level number, and/or an apartment, suite, or flat number. We could define each of these as separate elements within <CUSTOMER>. But this data could be considered as part of the normal content for the <street> element. Some customers may need two or more <street> elements. For others, <postcode> may not be available and so could be omitted.

To this point there is nothing in the declaration of CUSTOMER to control whether any or all of the declared elements must exist. We can provide extra control by specifying a *content model* (also called a *child list*).

**Figure 11-4**  
Constraints added to  
the CUSTOMER  
DOCTYPE  
Declaration.

```
<?XML version="1.0" standalone="YES"?>
<!DOCTYPE CUSTOMER
[
<!ELEMENT CUSTOMER ANY>
<!ELEMENT customer_name (#PCDATA)>
<!ELEMENT customer_address (street, city, state, postcode)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT postcode (#PCDATA)>
]>
```

We indicate in Figure 11-4 that `<customer_address>` has a (*child list*). This child list is a content model, which specifies that `<customer_address>` has child elements of (`street`, `city`, `state`, `postcode`). This comma-delimited format indicates that each customer address has only one `<street>`, `<city>`, `<state>`, and `<postcode>` element. Each element must appear in the specified sequence.

When a child list is defined using commas, each element is mandatory and must exist in that sequence. Alternatively, if any elements validly may not exist, they can be separated by “|” to indicate optionality, such as:

```
<!ELEMENT customer_address (street | city | state |
postcode)>
```

If we also add a `<contacts>` element, with child elements of `<phone>`, `<fax>`, `<mobile>` (cell phone), and `<email>` elements, we find even more variations. We therefore need to be able to specify the number of occurrences of child elements that are valid.

Figure 11-5 adds validity constraints to child elements by including a suffix character attached to the child name. A suffix of “?” specifies that zero or one occurrence of the child element may exist within the parent element. A suffix of “\*” specifies that zero or more occurrences may exist, while a suffix of “+” specifies that at least one or more occurrences of the relevant child element must exist within the parent element. No suffix indicates that the element must exist only once.

Examining Figure 11-5 further, we see that `<customer_address>` must have at least one `<street>` element, but it can have more `street` occurrences (`street+`). There must be only one

**Figure 11-5**

Further constraints added to the CUSTOMER DOCTYPE declaration for *customer\_address* and *contacts*.

```
<?XML version="1.0" standalone="YES" ?>
<!DOCTYPE CUSTOMER
[
<!ELEMENT CUSTOMER ANY>
<!ELEMENT customer_name (#PCDATA)>
<!ELEMENT customer_address (street+, city, state, postcode?)>
<!-- ? = zero or one; * = zero or more; + = one or more -->
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT postcode (#PCDATA)>
<!ELEMENT contacts (phone+, fax*, mobile?, email?)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT fax (#PCDATA)>
<!ELEMENT mobile (#PCDATA)>
<!ELEMENT email (#PCDATA)>
]>
```

`<city>` and one `<state>` element (no suffix). But the `<postcode>` element is optional; there may be none, or one occurrence (`postcode?`). The comma delimiters specify that the elements must appear in the declared sequence.

We may want to show that a valid *customer\_address* can have several addresses within it. We can place these child elements all within brackets, with the relevant group suffix character following the right bracket. We also use surrounding brackets to group other elements.

```
<!ELEMENT customer_address
(street+ | (city, state) | postcode)+>
```

The above fragment indicates by outer brackets with a suffix “+” that there must be at least one or more groups of addresses. Within an address group, there must be one or more *street* elements (`street+`) OR a *city* element followed by a *state* element OR a *postcode* element). Of course, all elements can also exist in the above example.

In Figure 11-5 we also saw new element declarations of `<contacts>`: `<phone>`; `<fax>`; `<mobile>`; and `<email>`. We see that `<contacts>` has a content model with child elements of (`phone+, fax*, mobile?, email?`).

Based on the suffix attached to each of the `<contacts>` child names, Figure 11-5 specifies that there must be at least one or more `<phone>` occurrences (`phone+`) and zero or more `<fax>` occurrences

(*fax*\*). There can be zero or one *<mobile>* occurrence (*mobile?*), and also zero or one *<email>* occurrence (*email?*).

We can use a content model that includes PCDATA. For example, we can alternatively specify *<phone>* and *<fax>* by the fragment:

```
<!ELEMENT phone (#PCDATA | (country-code, area-code,
phone-number))*>
<!ELEMENT fax (#PCDATA| (country-code, area-code, phone-
number))*>
  <!ELEMENT country-code (#PCDATA)>
  <!ELEMENT area-code (#PCDATA)>
  <!ELEMENT phone-number (#PCDATA)>
```

There can be zero or more *<phone>* and *<fax>*—by the suffix “\*” after the outer brackets. These can contain parsed character data, or they may optionally have a child element group in the sequence of (*<country-code>*, *<area-code>*, and *<phone-number>*). All content models that include PCDATA must have this format: PCDATA must come first, vertical bars must separate all elements or element groups, and the entire outer group must be optional.

These constraints enable an XML parser or XML application to confirm the validity of the document, by checking the number of child element occurrences within each parent element. They validate these occurrences against those specified by the child list constraints associated with the parent element in an internal DOCTYPE declaration, or a DOCTYPE in an external DTD.

## Declaring XML Attributes

An element may contain one or more attributes to provide additional details about that element. Figure 11-2 earlier included an example of attributes for the PERSON element. This specified a PERSON occurrence, with a unique identification attribute called *person\_id* and another attribute called *sex*, repeated now in Figure 11-6.

**Figure 11-6**  
The PERSON  
Element, with  
Attributes.

```
<PERSON person_id='p1100' sex='M'>
```

This example shows that attributes and their values are enclosed within the < and > characters of the start tag for an element, immediately following the element name.

Each attribute of an element is declared in DOCTYPE ATTLIST, using the format in Figure 11-7. The ATTLIST format specifies the *element\_name* and then defines an *attribute\_name* as a unique XML name within all of the element's attributes. It observes all of the rules detailed earlier in *XML Naming Conventions*.

The *type* specification in Figure 11-7 is defined from an enumerated list of valid values: (CDATA | ID | IDREF | IDREFS | ENTITY | ENTITIES | NMTOKEN | NMTOKENS | NOTATION).

CDATA represents "Character Data," as a character data type that is non-markup text. This is somewhat analogous to a Data Definition Language (DDL) SQL data type of VARCHAR, as used by DBMS products.



**NOTE** that XML does not support the other DDL data types such as numeric or decimal (with a defined length and precision), or money, currency, or CHAR (with a defined length), or float, bit, Boolean, or other data types. XML is used and read as text. An XML application must convert and validate these other data types. In the topics following When to Use Elements, Attributes, or Entities we will later discuss how these other data types can be supported.

We will continue with the other *type* declarations for Figure 11-7. ID represents an identifying attribute such as a primary key, with a unique name within the element. There can only be one attribute in an element that is specified with a *type* of ID.

Where an element must have a compound primary key for uniqueness (see Chapter 3), a single unique primary key is defined. In data modeling this is called a "surrogate key." The compound primary keys are instead defined as foreign keys with a *type* of IDREF or IDREFS. These are discussed shortly.

Furthermore, the value of each ID attribute must be unique for all occurrences of the relevant element. This follows the uniqueness rule of primary keys that we discussed in Chapter 3: a primary key cannot have

**Figure 11-7**  
The ATTLIST Format.

```
<!ATTLIST element_name attribute_name type "default value">
Where type = (CDATA | ID | IDREF | IDREFS | ENTITY | ENTITIES
| NMTOKEN | NMTOKENS | NOTATION)
```

duplicates. The earlier PERSON example has a unique value of “p1100” for the attribute named `person_id`, repeated as Figure 11-6.

An attribute can be defined as a foreign key, with a type of IDREF. Or several attributes can all be specified as foreign keys, each with a type of IDREFS. This offers more flexibility. It is used to specify many foreign keys. The referenced IDREF attribute name must also exist elsewhere, in an element where it is also declared as an ID or IDREF attribute. As we discussed earlier, IDREF or IDREFS can be used to specify compound primary keys, where a single primary (surrogate) key is specified with a type of ID.

In Figure 11-7 the type declarations ENTITY and ENTITIES define an attribute name, or attribute names, with associated substitution text. These declare entity references. The defined entity name can be used as a shorthand notation, analogous to a macro; it is replaced by the substitution text wherever it is used as an attribute value for the declared element. Entities can be used within the main body of the XML document, or in a DTD. We will cover *Entity Declarations* shortly.

Note that the use of ENTITY and ENTITIES by XML is different to the use of these terms in data modeling and normalization. We will consider these differences further in the topic *When to Use Elements, Attributes, or Entities*.

NMTOKEN and NMTOKENS types specify that the value of an attribute must be a valid XML name (NMTOKEN) or valid multiple XML names (NMTOKENS). A program can use an attribute of this type to manipulate XML data. For example, it can be used to associate a Java class with an element. A Java API can then be used to pass the data to a method for that class.

A NOTATION type typically is used to specify an application to process an unparsed value of an attribute. A NOTATION attribute is associated with a NOTATION declaration in a DTD. This declares the specific application program name to be invoked. We saw earlier that applications can be declared in a Processing Instruction (PI). This declares the `PI_target_name` as the application, with associated `PI_data`.

We will now discuss the “default value” specification in Figure 11-7. This is used to define a list of valid values for an attribute, or it can declare an attribute as being `#REQUIRED`, `#IMPLIED`, or `#FIXED`.

For example, attributes of PERSON in Figure 11-6 are specified by an ATTLIST declaration in Figure 11-8. We can see that `person_id` is an ID attribute. Every PERSON occurrence must have a unique

**Figure 11-8**

A list of valid attribute values.

```
<!ELEMENT PERSON EMPTY>
<!ATTLIST PERSON person_id ID #REQUIRED>
<!ATTLIST PERSON sex (M | F) #IMPLIED>
<!ATTLIST PERSON status (employee | trainee) "employee">
<!ATTLIST PERSON company CDATA #FIXED "XYZ">
```

*person\_id* value. Further, this ID attribute is mandatory (*#REQUIRED*).

The attribute *sex* in Figure 11-8 has valid values of “M” (Male) or “F” (Female). Any other values are invalid. This attribute example is *#IMPLIED*. It is not mandatory for a value to be supplied. The *sex* attribute can be omitted if it is not known.

A default value can be provided if an attribute is able to be omitted. In the example, *status* can only have valid values of “employee” or “trainee.” If not specified, *status* defaults to “employee”.

Finally, an attribute can be declared as *#FIXED*. This allows a default value to be supplied for an attribute, which cannot be changed. Figure 11-8 shows that *company* is character data (CDATA). It has a default value (*#FIXED*). This attribute is not provided in a document. It is automatically supplied as the value “XYZ”.

Another example of element and attribute declarations is provided in Figure 11-9. This defines a PHOTO element in XML, so it can be used by HTML to display an image on a Web page. The *src* attribute specifies the location of the photo image source file. It contains character data (CDATA) and is mandatory (*#REQUIRED*). The *width*, *depth*, *border*, and *alt* specify the image dimensions and border thickness, as well as alternate text that is displayed while the image file is being transmitted. These are all character data (CDATA) and are optional (*#IMPLIED*).

**Figure 11-9**

Attribute declarations for the PHOTO element.

```
<!ELEMENT PHOTO EMPTY>
<!ATTLIST PHOTO src CDATA #REQUIRED>
<!ATTLIST PHOTO width CDATA #IMPLIED>
<!ATTLIST PHOTO depth CDATA #IMPLIED>
<!ATTLIST PHOTO border CDATA #IMPLIED>
<!ATTLIST PHOTO alt CDATA #IMPLIED>
```

## Valid XML Documents

An XML document must not only be well formed as discussed earlier, it must also be valid. An XML document is valid if the document tags and their data content agree with the ELEMENT and ATTLIST declarations in the Document Type Definition (DTD). We discussed that a DTD is analogous to a DDL schema for a DBMS, but with different syntax. A DOCTYPE declaration for the earlier PERSON examples, together with the defined document tags and data content, is shown in Figure 11-10.

From Figure 11-10, we see that a PERSON document has two attributes: a *person\_id* which must be unique (ID #REQUIRED) and *sex*. This is an optional attribute (#IMPLIED), but if provided it can only have the values (M | F).

A PERSON must have at least one or more *names* (name+). A *name* has zero or more *given\_name* (given\_name\*) and at

**Figure 11-10**

A valid internal DTD, with defined XML tags and data content.

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE PERSON
[
<!ELEMENT PERSON
  (name+, email*, company?, country?, phone*, fax?, mobile*)>
  <!ATTLIST PERSON person_id ID #REQUIRED>
  <!ATTLIST PERSON sex (M | F) #IMPLIED>
<!ELEMENT name (given_name*, surname+)>
<!ELEMENT given_name (#PCDATA)>
<!ELEMENT surname (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT company (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT fax (#PCDATA)>
]>
<PERSON person_id="p1100" sex="M">
  <person_name>
    <given_name>Clive</given_name>
    <surname>Finkelstein</surname>
  </person_name>
  <email>cfink@ies.aust.com</email>
  <company>Information Engineering Services Pty Ltd </company>
  <country>Australia</country>
  <phone>+61-8-9309-6163</phone>
  <phone>(08) 9309-6163</phone>
  <fax>+61-8-9309-6165</fax>
  <mobile>+61-411-472-375</mobile>
  <mobile>0411-472-375</mobile>
</PERSON>
```

least one or more `surnames` (`surname+`). The document shows examples of these tags with the relevant data content.

A `PERSON` can have zero or more `email` addresses (`email*`), zero or one `company`, `country`, or fax number (`company?`, `country?`, `fax?`), and zero or more `phone` or `mobile` numbers (`phone*`, `mobile*`). We can see in Figure 11-10 that two phone numbers and two mobile numbers are provided as part of the `PERSON` document content. The data tags and content agree with the `DOCTYPE` declaration. The `PERSON` root name and its contents therefore comprise a valid XML document.

## Entity Declarations

XML uses the term `ENTITY` to declare a substitution name for insertion of predefined values. This is quite different from the use of the term “entity” in data modeling. We discuss this later in the chapter, in the topic *When to Use Elements, Attributes, or Entities*.

There are two types of entity declarations. The first is a *General Entity* declaration. This can be used inside the main body of an XML document or in a DTD section, where it is called an *internal entity reference*. It can be used externally to a document, when it is called an *external entity reference*. A general entity reference is distinguished by a prefix “&.”

We saw internal general entity references earlier, in *XML Naming Conventions*. XML supplies five predefined entities—`&lt;` (which is replaced by `<`), `&gt;` (by `>`), `&amp;` (by `&`), `&quot;` (by `“`) and `&apos;` (by `’`). The replacement text replaces an internal entity reference only when it is displayed, or is about to be processed by an application. For example, the internal entity reference “`&IES;`” can be declared with text “Information Engineering Services Pty Ltd.” This text automatically replaces “`&IES;`” wherever it occurs, but only when that entity is displayed or passed to an application. Internal entities can contain references to other internal entities, but they cannot be recursive.

Distinct from an internal entity reference, the replacement text for an external entity reference immediately replaces that entity wherever it occurs. An XML parser or processor processes the replaced text as if it was an original part of the document.

An entity name must be a unique XML name. It is declared together with the replacement text. This text is substituted for the entity

**Figure 11-11**

General entity declaration and usage examples.

```

Format:
  <!ENTITY entity_name "replacement text">           (Internal)
  <!ENTITY entity_name SYSTEM URL">                 (External)

Declaration Examples:
  <!ENTITY IES "Information Engineering Services Pty Ltd">
                                                    (Internal)
  <!ENTITY copy99 "© Copyright 1999">               (Internal)
  <!ENTITY refl SYSTEM http://www.ref.com/ref1.xml"> (External)

Usage Examples:
  "&IES;" is replaced later by Information Engineering
    Services Pty Ltd
  "&copy99;" is replaced later by * Copyright 1999
  "&ref1;" is replaced immediately by the content of document
    ref1.xml

```

wherever it occurs. Figure 11-11 shows the internal and external format and examples for a general entity.

The format and two examples of an internal entity are illustrated in Figure 11-11. The first declares “&IES;” as an internal entity reference for *Information Engineering Services Pty Ltd*. The second declares “&copy99;” as a shorthand for the text © *Copyright 1999*. Whenever “&IES;” or “&copy99;” are found internally within a document they are replaced by that text, but only when the document is about to be processed or displayed.

The third example in Figure 11-11 declares an external entity “&ref1;” as a shorthand reference for the document “*ref1.xml*.” This is located externally at “*http://www.ref-xml.com/ref1.xml*.” (This is a fictitious URL.) Because it is an external entity reference, it is immediately replaced by the content of the document “*ref1.xml*.”

An entity can be a convenient shorthand way of including a much larger amount of text, as shown in Figure 11-11. It also provides an XML document with a single point for declaration of text that can change. If volatile text appears in many places of an XML document, a general entity can be used in each place. The replacement text is defined once only when the entity is declared. Whenever that text is later changed, the updated text automatically replaces every occurrence of that entity.

We discussed that there are two types of entities. The second type is a *Parameter Entity* declaration, which can only be declared inside a DTD. A parameter entity reference is distinguished by a prefix “%”.

Figure 11-12 now shows the declaration format and examples of a parameter entity, which uses a prefix “%”—distinguished from general

**Figure 11-12**

Parameter entity declaration and usage examples.

```

Format:
  <!ENTITY % entity_name "replacement text">           (Internal)
  <!ENTITY % entity_name SYSTEM URL">                 (External)
Declaration Format:
  <!ENTITY % person SYSTEM person.dtd">             (External)
  <!ENTITY % idr 'ID #REQUIRED' >                   (Internal)
Usage Examples:
  <!DOCTYPE PERSON SYSTEM %person;>                 (External)
  .....
  <!ATTLIST PERSON person_id %idr;>                 (Internal)

```

entities that use a prefix of “&.” A parameter entity is declared in a DTD, which can be internal or external. If declared in an internal DTD, it is used within that same document similar to a general entity. If declared in an external DTD, it references a URL where the DTD exists.

In the first example of Figure 11-12, the % character—followed by a space—declares that *person* is an external Parameter Entity. It specifies that content for “%*person*;” (no spaces) is located in the DTD file “*person.dtd*.” The content of this DTD file immediately replaces “%*person*;” as if it was an original part of the document.

The second example declares “%*idr*;” as an internal Parameter Entity that is to be immediately replaced by the text ‘ID #REQUIRED.’ The example shows an ATTLIST declaration for *person\_id* (as an attribute of PERSON) with “%*idr*;” as an internal Parameter Entity. This is replaced immediately by “ID #REQUIRED,” as if it had been written:

```
<!ATTLIST PERSON person_id ID #REQUIRED>
```

Any amount of replacement text can be declared for general entities and for parameter entities. This text is surrounded by quotes. As we have seen, entities can be declared to insert fragments or complete paragraphs of standard “boiler-plate” text in a document. That insertion is immediate for parameter entities or external general entities. Insertion is deferred for internal general entities; the entity is replaced by the text only when it is about to be displayed, printed or passed to an application for processing.

## Namespaces

XML requires that each element name is unique within a document, and also that each attribute name is unique within an element. These names

can be declared in an internal DOCTYPE for the document, or can be specified in a DOCTYPE declared externally in a DTD whose location is referenced by the document DOCTYPE. Once defined, the element and attribute names constitute a “markup vocabulary”.

A problem exists if different XML processors or applications all use the same names for different purposes. A single document may also include tags for more than one application. Because of these different markup vocabularies, there are potential problems of recognition and name collision. Software modules must be able to recognize the tags and attributes that they are designed to process, even in the face of naming collisions.

The role of XML namespaces is to provide a mechanism that overcomes these problems. Names can be qualified by a namespace prefix so that they are universally unique. XML Namespaces are now documented as a W3C “recommendation,” the first step for acceptance as a standard [Namespaces]. This recommendation specifies that:

An XML namespace is a collection of names, identified by a URI reference, which are used in XML documents as element types and attribute names.

XML namespaces differ from the “namespaces” conventionally used in computing disciplines in that the XML version has internal structure and is not, mathematically speaking, a set.

Names from XML namespaces may appear as qualified names, which contain a single colon separating the name into a namespace prefix and a local part. The prefix, which is mapped to a URI reference, selects a namespace. The combination of the universally managed URI namespaces and the document’s own namespace produces identifiers that are universally unique. [Namespaces]

An example of a qualified name as an element type of *price* for use in EDI applications is illustrated in Figure 11-13. (This is not a formal EDI specification, and the referenced URI is fictitious.)

Figure 11-13 is an example of a local namespace with a limited scope. It declares an XML namespace (`xmlns:`) prefix of “*edi*,” located at the URI ‘`http://ecommerce.org/schema`’. As it is declared within the element *retail\_price*, its scope applies until the end tag `</retail_price>` is reached. The example prefixes “*edi:*” to the *price* element and to the *status* attribute of the *tax* element. These are both children within the *retail\_price* parent element.

A namespace declaration applies to the element where it is specified and to all elements within the content of that element, unless overridden by another namespace declaration. With the *edi:* prefix in Figure 11-13, the *price* element and *status* attribute each now

**Figure 11-13**  
Namespace definition  
and use for elements.

```
<retail_price xmlns:edi="http://ecommerce.org/schema">
  <!-- The namespace for the 'price' element, and the
       'status' attribute of the 'tax' element, is located at
       'http://ecommerce.org/schema' -->
  <edi:price unit="Euro">2.50</edi:price>
  <tax edi:status="Exempt">Meat</tax >
</retail_price>
```

has a unique name within the scope of the *retail\_price* parent element.

More detail about namespaces can be obtained from the W3C recommendation and associated references, available from the [Namespaces] Web site.

## Extensible Style Language (XSL)

The XSL specification is based on the Document Style Semantics and Specification Language (DSSSL) and Cascading Style Sheets (CSS). It is simpler than DSSSL and more powerful than CSS. While not yet a recommendation at the time of writing, it has progressed through a number of drafts. It will soon become an integral part of XML. We will briefly introduce XSL in this section. Further detail can be found in [Harold 1998]. The latest XSL specifications are available from the [XSL] Web site.

An XSL style sheet is itself an XML document. It has elements that are rules defining how the data contents of specific tags are to be displayed on screen or printed. An XSL processor uses a specified XSL style sheet to process an XML document. The output that is generated is an HTML file that can then be stored in a Web server, ready to be transmitted to a Web client and viewed with a Web browser.

A single XSL style sheet can be used by many XML documents. An organization may choose to define a corporate XSL style sheet that is to be used by all departments. If required, a department can override certain corporate styles by defining its own styles first. A parameter entity is then used to include the corporate style sheet automatically. XML always processes the first occurrence of a declaration that it encounters, ignoring later occurrences of that same declaration. Because they are processed first, only the departmental styles that have been specified therefore override those same relevant corporate styles.

**Figure 11-14**  
XSL Style Sheet  
format.

```
<xsl>
  <rule>
    <target-element type="tagname"/>
    action
  </rule>
  <rule>
    <target-element type="tagname"/>
    action
  </rule>
</xsl>
```

The root element of an XSL style sheet is called `<xsl>`. Each `<xsl>` element contains one or more `<rule>` elements. Each rule specifies a *target* and an *action*. The *target* is an expression that defines which XML elements the *rule* applies to. The *action* specifies the list of *flow objects* that are to be generated when the rule is applied. A *flow object* is typically HTML markup and/or text that will become part of the output. Figure 11-14 shows the format of an XSL style sheet.

An XSL processor uses the style sheet to process tags in an XML document. Each time a `<tagname>` is found in the document that matches a `<target-element>` for a *rule* in the style sheet, the processor carries out the specified action for that rule. The action usually specifies HTML tags that will surround the data content of the located XML `<tagname>`.

HTML compresses white space (such as carriage returns, new lines or space characters) all as a single space character. This can be overridden by surrounding preformatted text with `<PRE> ... </PRE>` tags. A new line can be forced in an HTML file with a `<BR>` tag. A new paragraph can be forced with a `<P>` tag.

The HTML tags `<DIV> ... </DIV>` and `<SPAN> ... </SPAN>` can be used to apply formatting to regions of text. The `<SPAN>` tag is used to apply specified formatting in-line, as part of a sentence for example. The `<DIV>` tag is used to separate a block of text from other text, as if it was surrounded by `<P> ... </P>` tags for a new paragraph.

In contrast, XML preserves white space. To continue to maintain this white space after an XML document is converted to HTML, the tags can be included as part of the action for an XSL rule. Figure 11-15 illustrates the use of a `<DIV>` tag for all tags in an XML document, setting the style for the entire document to a font size of 12 point and using the font family of Arial.

**Figure 11-15**

Converting from XML to HTML, with a global font size and font family.

```

<xsl>
  <rule>
    <root/>
    <HTML/>
    <BODY>
      <children/>
    </BODY>
  </HTML>
</rule>
<rule>
  <target-element/>
  <DIV style="font-size: 12pt; font-family: Arial;">
    <children/>
  </DIV>
</rule>
<rule>
  <target-element type="person_name"/>
  <H2>
    <children/>
  </H2>
</rule>
</xsl>

```

The first `<rule>` in Figure 11-15 specifies an empty `<root/>` tag. The action therefore applies to all tags and their content in the document. It declares `<HTML> ... </HTML>` and `<BODY> ... </BODY>` tags to surround `<children/>`, so applying these tags around the whole document. This converts the XML document to an HTML file.

The second `<rule>` provides a global style for every `<target-element>` in the document. This uses a font-size of 12 pt and a font-family of Arial. `<DIV>` also places the `<children/>` data content each on a new line, for each XML tag in the document.

The third `<rule>` specifies the `<target-element type="person_name,"` such as in Figure 11-10. It specifies that all `<person_name>` elements will be surrounded by `<H2> ... </H2>` tags. The output HTML file will display each person's name (all data content between `<person_name> ... </person_name>` tags) in the HTML Heading 2 format for the relevant browser.

XSL also allows elements to be targeted by attribute. This is very flexible as it can target all elements that have a certain attribute, or that have a certain attribute with a specific value, or an element whose ancestors have an attribute with a specific value, and more. Figure 11-16 documents the format, with a typical example.

Figure 11-16 locates all PERSON elements. Of these elements, it selects the `person_id` attribute with the value of "p1100" and

**Figure 11-16**

An XSL rule based on an Attribute Value.

```

Format:
<attribute name="attribute-name" value="attribute-value"/>
Example:
<rule>
  <element name="PERSON"
    <attribute name="person_id" value="p1100"/>
    <target-element type="person_name"/>
    <SPAN style="font-weight: bold">
      <children/>
    </SPAN>
  </element>
</rule>

```

locates the target element of `<person_name>`. `<SPAN>` includes all of the target-element and its children, which are made **bold** by the style specification `font-weight: bold`.

This example located a specific occurrence of an element based on the value of the attribute, `<person_id>`. If instead `<person_id>` was declared as an element of type ID, it could be explicitly referenced by an XSL ID attribute, such as:

```
<id attribute="attribute-name"/>
```

Formatting rules can also be specified to apply to elements based on their position in the parent element. This is specified by adding `only` and `position` attributes to the element or target-element tags.

The value `of-type` used with `only` specifies that the target be the only element of its type that is a child of its parent. The value `of-any` specifies that the target be the only element of any type of its parent.

The `position` attribute has four valid values: `first-of-type`, `last-of-type`, `first-of-any`, and `last-of-any`. The first rule locates the target that is the first child of a specified type of its parent. The second rule locates the last child of the specified type of its parent. The third and fourth rules locate the first child (or the last child) of any type of the parent.

In addition, content can be added as part of an XSL action following a rule. Additional text can be included that is not present in the XML document. For example, the internal general entity `&M;` (defined as `AUD`) adds this prefix to prices in Figure 11-17, denominating prices in Australian dollars as AUD\$15,500.00 (say).

If required, prices can instead be denominating in US dollars just by changing the declaration of the entity `&M;` instead to `USD`.

**Figure 11-17**

Adding content to  
<price> in an XSL  
action.

```
<rule>
  <target-element type="price"/>
    &M; <children/>
</rule>
```

To this point, we have used <children/> to include children of the specified element in the output. We can be more specific with *Select*.

## Selected Elements

When a target is matched, an action that includes a <children/> tag outputs all children of the element that matched the target, perhaps also applying other rules recursively to the children. These children are output in the order in which they appear in the XML document. The <select-elements> tag allows some children to be included in the output, while others are excluded. The order that included children are output can also override their order in the XML document.

The <target-element> tag applies patterns to the input; the <select-elements> tag applies patterns to the output to organize, reorder, and select the exact elements to appear in the rendered output. Figure 11-18 shows how we use <select-elements> to present the PERSON details of Figure 11-10 in a specific order.

Figure 11-18 has now changed the order of the tags in Figure 11-10 to display a person's *name* first at the start of a new paragraph <P>, followed by *company*, then *phone*, *fax*, and *email*. <BR> places

**Figure 11-18**

Presenting XSL output  
in a specific order.

```
<rule>
  <target-element type="PERSON"/>
  <select-elements>
    <P>
      <target-element type="name"/> <BR/>
      <target-element type="company"/> <BR/>
      <target-element type="phone"/> <BR/>
      <target-element type="fax"/> <BR/>
      <target-element type="email"/>
    </P>
  </select-elements>
</rule>
```

each of these on a new line. Note that neither `country` nor `mobile` is a specified `<target-element>` in the list of `<select-elements>`. They are therefore omitted from the output.

## XSL Macros

In some cases, you may want to apply the same action on multiple elements. For example, you may have specific elements for managers, other elements for salespersons and still different elements for clerks. But you may want each of these separate job titles all to be formatted in the same way. XSL *macros* can be defined to apply the same group of complex rules to different objects.

The XSL `<define-macro>` tag uses an attribute name that becomes the name of the macro. The macro specifies relevant replacement text together with one or more `<contents/>` tags that specify where the contents of the relevant element are to be placed. Macros are declared in an XSL style sheet outside of a rule.

## XSL Named Styles and Embedded Scripts

*Named styles* can be used similar to macros, to apply style properties. For example, the `<define-style>` tag allows a series of Cascading Style Sheet (CSS) style values to be used with a defined style name. Styles can also be included in XML tags; but this tends to defeat the intent of XML and XSL, which is to separate content from layout.

XSL style sheets can also contain *embedded scripts* that are written in JavaScript. These scripts can be executed by XSL as it processes an XML document, or they may be passed to the output HTML file so they can be executed by a Web browser.

This overview of XSL has provided a small insight into its power. As we have seen, it can be used to provide great flexibility for processing and formatting an XML document as an HTML file. And it can be expanded further with JavaScript. Using the power of XML with the syntax of XSL, very powerful rules can be defined to format the data content of an XML document. The XSL concepts are covered well in [Harold 1998]. The complete XSL specifications are available from the [XSL] Web site.

We have examined XSL syntax and examples, but in practice we expect that XSL will be generated using other tools. For example, we

anticipate that developers of word processing software packages will offer an XSL generation option. This could convert existing or new style templates automatically into XSL style sheets. HTML editing packages should also allow style templates to be defined and generated as XSL style sheets using a similar approach.

Once generated, the look and feel of an XML document—converted to an HTML Web file under control of an XSL style sheet—can be changed merely by making appropriate style changes using the word processor or HTML editor. The changed XSL code will convert the generated HTML file then to the new style.

## Document Object Model (DOM)

We have seen that an XML parser or processor is used to analyze an XML document. If it is well formed, and agrees with its DTD (and so is valid), the document is then passed to another application for processing. XSL is an application that is used to process and convert an XML document, formatting its output as an HTML file. Enterprise and other applications may need to carry out other processing against an XML document or an HTML file. A way is needed to access the content of XML or HTML files using a programming language.

Recognizing this need, the W3C has produced specifications for the Document Object Model (DOM). This is now a recommendation of the W3C; the DOM specifications are now available from the [DOM] URL.

### DOM Specifies XML Program Interface

XML can be used to define the metadata vocabulary used by many disciplines. The structure of an XML document is clearly defined by its metadata, as well as the meaning of its data content. DOM defines a standard programming interface that can be used by a wide variety of environments and applications. DOM is a language-independent specification using the Object Management Group (OMG) Interface Definition Language (IDL), defined in the CORBA 2.2 specification. Any language can therefore use the DOM. In addition to the OMG IDL, language bindings are provided for Java and for ECMAScript—an industry-standard scripting language based on JavaScript and Jscript.

The DOM is an object model based on object-oriented design. The DOM recommendation specifies that *documents are modeled using objects, and the model encompasses not only the structure of the document, but also the behavior of a document and the objects of which it is composed. . . . As an object model, the DOM identifies:*

- *the interfaces and objects used to represent and manipulate a document*
- *the semantics of these interfaces and objects—including both behavior and attributes*
- *the relationships and collaborations among these interfaces and objects.*

## DOM Core and DOM HTML

The DOM has two parts: *DOM Core* and *DOM HTML*. The DOM Core represents the functionality used by XML documents. It also serves as the basis for DOM HTML. It documents functions to *create, set, get, replace, and remove* any objects of an XML document that can be written using XML, XSL, or XLL syntax. Using the DOM, a complete XML document—or any object within it—can be created, read, updated, or deleted as if it was part of a file or database.

It is important to note that DOM is not a binary specification. DOM programs written in the same language will be source code compatible across platforms, but DOM does not define any form of binary interoperability. It is not a set of data structures; it is an object model that specifies interfaces for managing XML and HTML documents.

The DOM is not a competitor to the Component Object Model (COM). It can be implemented using language-independent systems like COM or CORBA, or it can be implemented with language-specific bindings like Java or ECMAScript bindings as discussed earlier.

## Extensible Linking Language (XLL)

The main reason for the rapid growth of the WWW and browsers in the early 1990s was largely due to HTML links. The ability to point and click

on a link in a document, and be taken immediately to another document anywhere in the world, is now a familiar concept.

We discussed earlier that links based on a Uniform Resource Locator (URL) are vulnerable to change. If the targeted resource is moved, all URL links pointing to the target must be explicitly updated to point each URL to the target in its new location. In contrast, a Uniform Resource Identifier (URI) need never be changed.

If a link must point not to the start of a target HTML document, but instead to a point within that document, an anchor point must be used. The HTML anchor point must be inserted within the target document, at the relevant point. This target anchor point is then included as a suffix to the link, separated from the URL by a “#.”

In HTML a link is specified with the <A> tag. In XML, almost any tag can be a link; elements that include links are called *linking elements*. The Extensible Linking Language (XLL) specifies how linking elements and URIs are defined and used. XLL is implemented using *XLinks* and *XPointers*. A linking element can specify a *Simple Link* or *Extended Links*.

## Simple XLinks

A simple linking element is specified by `xlink:form="simple"`. Each linking element contains an `href` attribute; its value is the URI of the target (linked) resource. Three linking examples follow in Figure 11-19. These are examples of *simple* XLinks, similar to standard HTML links.

The first example in Figure 11-19 uses a `<store_link>` element and relative URL to link to the HTML page “store.html” when the active underlined text: Enter our Online Store is clicked. The start tag is terminated by the end tag `</store_link>`.

**Figure 11-19**  
Example of “simple”  
Xlinks defined by XLL.

```
<store_link xlink:form="simple" href="store.html">
  Enter our Online Store</store_link>
<home_page xlink:form="simple" href="http://
www.visible.com.au/">
  Visible Systems Australia Pty Ltd Home Page</
home_page>
<image xlink:form="simple" src="visible-logo.gif"
href="storeohtml"=/>
```

The next example uses a `<home_page>` element to specify an absolute URL for the active underlined Visible Systems Australia Pty Ltd Home Page text. This link is terminated by `</home_page>`.

The final example is an image link using the relative URL for the GIF file `visible-logo.gif`. This last example is simplified by declaring the `xlink:form` attribute `FIXED` in a DTD, as shown in Figure 11-20.

Link elements may contain three optional attributes that specify how the link is to be used and how the target resource will be displayed in relation to the current page. These are `show`, `actuate`, and `behavior`.

The `show` attribute has three values: `replace`, `new`, and `embed`. With `show="replace"`, the target document is displayed in the same window as presently used by the current page. This is analogous to the default behavior of standard HTML links. In contrast, `show="new"` displays the target document in a new window—similar to the behavior of an HTML link set to `_blank`. Or with `show="embed"`, the target document is inserted into the current document—similar to an HTML server `include` statement.

The `actuate` attribute has two values: `user` and `auto`. A specification of `actuate="user"` indicates that the link is to be traversed only when the user clicks on it. This is the default action. The alternative specification of `actuate="auto"` indicates that the link is to be followed automatically whenever another targeted resource of that same link element is traversed.

The `behavior` attribute passes included data to an application specified to read that data. The specific data and how it is used is determined by the application. For example, the sound file `welcome.au` could play when the relevant link is traversed.

**Figure 11-20**

Example of a `FIXED` `xlink:form` attribute.

```
<!ELEMENT image EMPTY>
<!ATTLIST image
  xlink:form CDATA #FIXED "simple"
  src CDATA #REQUIRED
  href CDATA #REQUIRED
  alt CDATA #IMPLIED
  height CDATA #IMPLIED
  width CDATA #IMPLIED>

<image src "visible-logo.gif" href="store.html"/>
```

**Figure 11-21**

Using the *show*, *actuate*, and *behavior* XLink attributes.

```

<!ENTITY % link
    "xlink:form CDATA #FIXED 'simple'
    href CDATA #REQUIRED
    show CDATA (new | replace | embed) 'replace'
    actuate CDATA (user | auto) 'user'
    behavior CDATA #IMPLIED"

<!ELEMENT home_page (#PCDATA)>
<!ATTLIST home_page %link;>

<home_page href="http://www.ies.aust.com/~ieinfo/"
    show="new" behavior="sound:welcome.au">

```

Like all attributes, *show*, *actuate*, and *behavior* must be declared in an ATTLIST for the link element. Figure 11-21 illustrates this, using the parameter entity “%link;” for convenience.

The parameter entity “%link;” is a single declaration that specifies all of the XLink attributes. The <home\_page> ELEMENT then uses “%link;” to declare its ATTLIST. This inserts the ATTLIST XLink declarations into the <home\_page> ELEMENT.

The <home\_page> tag can now be used in an XML document to link to the IES Web site as shown in Figure 11-21. The IES home page is displayed in a *new* window and the sound “welcome.au” is played. The default *show* = “replace” has been overridden, while the second default attribute *actuate* = “user” has been used as declared.

So far we have discussed *simple* links. These are very similar to the standard links used by HTML. They are in-line links that are unidirectional; they are traversed from one link to the target. XLL is more powerful than HTML. It also supports *extended* links. These can define multidirectional links—with many links to many targets, as well as out-of-line links that reference a link file.

## Extended XLinks

Distinct from HTML, an *extended* link can point to more than one target. It is declared by the attribute *xlink:form* = “extended”. The targets are defined in locator elements as children of the linking element, rather than use an *href* attribute in the linking element. The *show*,

**Figure 11-22**

Using child elements to implement XLL extended links.

```

<!ELEMENT home_page (website)*>
  <!ATTLIST home_page
    xlink:form      CDATA    #FIXED "extended"
  >
  <!ELEMENT website EMPTY>
    <!ATTLIST website
      xlink:form    CDATA    #FIXED "simple"
      show          CDATA    (new | replace | embed) "replace"
      actuate       CDATA    (user | auto) "user"
      behavior      CDATA    #IMPLIED
    >

<home_page>IES and Visible Home Pages
  <website href="http://www.ies.aust.com/~ieinfo/">IES
    Australia
  </website>
  <website href="http://www.visible.com.au/">Visible
    Australia
  </website>
  <website href="http://www. www.visible.com/">Visible USA
  </website>
</home_page>

```

*actuate*, and *behavior* attributes can be specified with each locator element.

Figure 11-22 specifies an extended `<home_page>` parent element with multiple `<website>` child links within the parent. Each child link now specifies the *href* attribute for its relevant text link. Defaults are used for *show* and *actuate*, but optional *behavior* attributes have not been included.

## Implementing Uniform Resource Identifiers (URIs)

All of the links defined in this section, both simple and extended, are *in-line* links based on URLs. XLL also supports *out-of-line* links. The links between documents are not present in the documents themselves, but are stored in separate XLL documents or files. Thus they become URIs. They never change in the linking documents, which always specify the same XLL document or file. This is analogous to indirect addressing. Any

required changes are made only in the XLL document; the linking documents are unaffected.

This capability will lead to innovative new applications, as links can be maintained quite separate from the content of the document. For example, an XML document can have its links defined out-of-line in an XLL document or file. An HTML page can then be generated, using XML to specify the structure and content of data in a database. As the data changes dynamically in the generated HTML, the links embedded in that data can also change dynamically using XLL. Additional logic can also be provided using XSL `<rules>` or with embedded JavaScript. The application potentials are endless.

## XPointers

To this point we have used *XLinks* to point to a specific resource, which may be an HTML page. XPointers are used to point to precise locations within a resource: a section, part, chapter, clause, or specific text in an unstructured textual document. Or it may point to a specific record in a legacy file, or a specific row and column of a certain table in a database. They are more powerful than HTML anchors.

HTML requires that anchors be physically inserted at the relevant position in the target document. In contrast, XPointers are specified without requiring any changes to the target document. Unlike HTML anchors, XPointers can point to ranges or spans rather than a specific point. For example, an XPointer can select a particular part of a document so that it can be copied or loaded into a program.

*Absolute XPointers:* An XPointer location can be specified as *id*, *root*, *html*, or *origin*. An *id* location selects the document element that has an ID-type attribute with the specified value. A *root* location selects the root element of a document and has no arguments; it selects the entire document. An *html* location selects a named anchor in an HTML document, for compatibility with HTML. The *origin* pointer is used in conjunction with one or more relative location terms—discussed next.

*Relative XPointers:* The *id*, *root*, *html*, and *origin* specify absolute locations: they can find an element in a document regardless of other contents of the document. But sometimes you may need to locate the first or the last element of a given type. Or instead, the next or preceding element, or the first (or last, next, or previous) child (or parent)

of a specified type. These are all relative locations. They utilize the following location terms:

- *child*: searches the immediate children of the specified source element.
- *descendent*: searches all descendents of the source, not just the immediate children.
- *ancestor*: searches all ancestors of the source, starting from the nearest.
- *preceding*: searches all elements that occur before the source element in the document.
- *following*: searches all elements that occur after the source element in the document.
- *psibling*: selects the element that precedes the source element in the same parent element.
- *fsibling*: selects the element that follows the source element in the same parent element.

These relative XPointers provide a powerful location capability within unstructured text documents, as well as within structured legacy files or databases. They can specify ID values or particular values of elements and/or attributes.

Finally a selection can be made by *string*. This can be used to point into non-XML data, or into XML data that contains large amounts of text. A specific character, or range of characters, can be selected.

Figure 11-23 provides several examples that illustrate the use of XPointers. Each URL can be used as the *href* attribute value in a simple or extended XLink.

The first URL example in Figure 11-23 specifies the HTML document *articles.htm* in the IES Web site, at the anchor point “#TEN.” When activated, this link positions the browser to display a list of

**Figure 11-23**

Examples of XPointers specified in Xlink URLs.

```
http://www.ies.aust.com/~ieinfo/articles.htm#html(TEN)
http://www.ies.aust.com/~ieinfo/ten.xml|id(ten03)
http://www.ies.aust.com/~ieinfo/ten.xml|root(),following(2)
http://www.ies.aust.com/~ieinfo/ten.xml|root(),string(2,
“IES,” 1, 3)
```

published free quarterly newsletters, called: *The Enterprise Newsletter (TEN)*.

The second URL example locates an element in the document *ten.xml*, which has an *id* value of *ten03*. The “|” character (instead of the anchor point character “#”) selects only the specified element.

The third URL example selects the second element (2) following the root element of *ten.xml*.

Finally, the fourth example uses a *string* to select the second occurrence of the string “IES.” The two digits after the string specify the first character, and the number of characters, that are to be selected. This selects the entire string “IES”.

## Resource Description Framework (RDF)

Metadata that is used by various industries, communities, or bodies can be used with XML, XSL, and XLL to define markup vocabularies. The W3C is developing a standard framework that can be used to define these vocabularies. This is called the *Resource Description Framework (RDF)*. It is a model for metadata applications that can support XML applications. RDF is an attempt by the W3C to build standards for XML applications so that they can interoperate more easily. The current status of RDF is available from the [RDF] Web site.

### General Markup Vocabularies

We earlier referred to *Channel Definition Format (CDF)*, supported by Microsoft in Internet Explorer 4.0. This is based on XML. Other examples are Netscape’s *Meta Content Framework (MCF)*, the *Open Software Description (OSD)* defined by Marimba and Microsoft, and the *Web Interface Definition Language (WIDL)* by webMethods. These vocabularies use unique language for communication.

*Channel Definition Format (CDF)*: Microsoft has submitted CDF to the W3C as a standard, but it may remain a Microsoft-only standard. CDF provides a standard set of tags for defining push channels. These

automate the flow of data from a Web server, pushed to a browser. CDF comprises a DTD that points the browser to relevant content, with descriptive content information and a schedule for downloads.

*Meta Content Framework (MCF)*: Netscape extended original work done by Apple to navigate in 3D through a Web site. They converted it to XML as the Meta Content Framework. The current MCF model uses XML to create information nodes describing Web sites and pages. These nodes can include other nodes and can link across multiple files, creating a web of metadata that reflects the web of HTML underneath.

*Open Software Description (OSD)*: The OSD was developed to deliver software and software updates over the Internet, not just Web pages. OSD can automatically download and install programs and packages in Java and in platform-specific code. Marimba is interested in using OSD for Java; Microsoft will use it to deliver Windows software. The OSD specifies information that is used to install the same software on multiple platforms, even if the code to be downloaded will change depending on the platform. OSD files specify program dependencies, allowing Java programs to download packages and Windows programs to download required DLLs. It expands the current OBJECT and APPLET HTML tags. OSD is designed to work with CDF as well; channels can be used with OSD files to update software automatically and regularly.

*Web Interface Definition Language (WIDL)*: WIDL provides information about available Web services to client machines, allowing them to automate web-based processes. Programs use WIDL without needing a browser. For example, a shipping clerk who needs to track thousands of packages can use WIDL to connect lists of tracking numbers to the relevant FedEx, DHL, or UPS Web sites. We will see other applications that use WIDL in Chapter 12.

WIDL makes it easy to connect clients to back-end systems through Web interfaces. It is a tool to connect and automate systems, converting applications using sophisticated interfaces to access data and databases. Parts of WIDL may become obsolete as XML is more widely used, when programs are able to parse XML data easily without the need for a separate interface. WebMethods has promised WIDL interface capabilities for C, C++, Java and Visual Basic. We could expect them also to update WIDL to reflect advances in XML.

## Special Purpose Markup Vocabularies

There is considerable effort in some industries to define a standard vocabulary, using XML for their metadata. Markup languages have been defined for: *Mathematics Markup Language* (MathML); *Chemical Markup Language* (CML); *Open Financial Exchange* (OFX); *Internet Content Exchange* (ICE); *Speech Markup Language* (SpeechML); *JavaBean Markup Language* (JBL); *Synchronized Multimedia Integration Language* (SMIL); and *Wireless Markup Language* (WML).

The [W3C] and [RDF] Web sites are two good starting points for more information. They will point you to specific Web sites that provide additional details about the above markup languages.

## XML Resolves Many HTML Problems

Early in this chapter we discussed a number of problems associated with the use of HTML. XML resolves many of these problems, as discussed next.

1. *XML defines content of page* We now know that XML offers a powerful way to define tags describing the content of a document. This document can be unstructured text, or it can be graphics, images, audio or video files, or it can be structured data in legacy files, relational or object databases.
2. *Search engines can locate XML content* Search engines can precisely locate required content based on defined XML tags. This content has more meaning than earlier search methods that rely only on word indexes or manually defined keywords.
3. *XML can integrate dissimilar data sources* XML can be used to define the structure of legacy files, relational and object databases, as well as unstructured text, graphics, images, audio, or video. This makes it easier to integrate data content sourced from dissimilar systems and databases.
4. *Easier dynamic programming* XML and DOM simplify programming to incorporate dynamic content from different data

sources. DOM offers a language-independent interface for processing XML documents.

5. *Easier interfacing with back-end systems:* XML and DOM have been designed to interface with back-end systems. Markup languages such as WIDL also provide this capability.

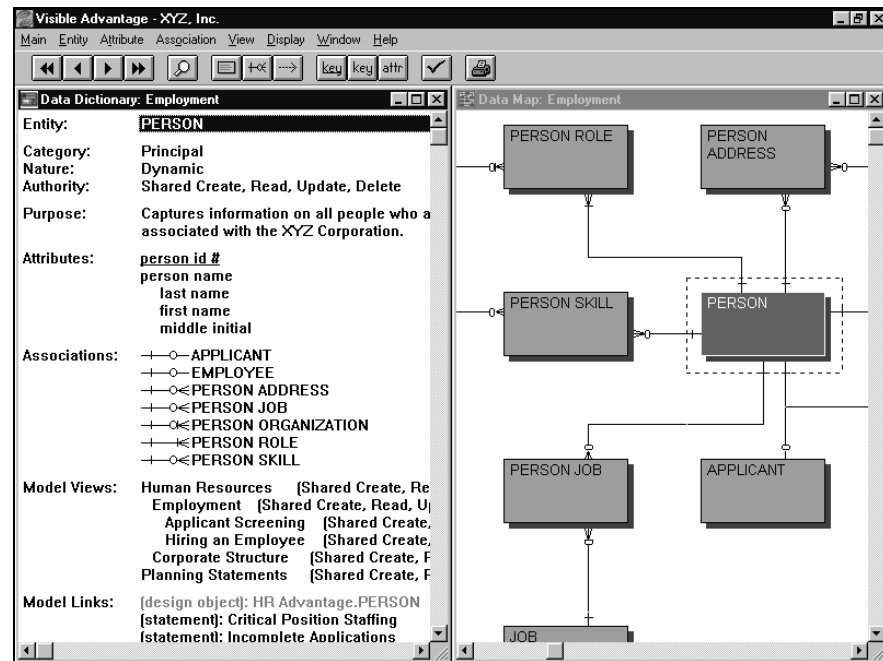
## When to Use Elements, Attributes, or Entities

The terminology used by XML differs in context from the terminology we used for data modeling in Parts 1 and 2. In data modeling, an entity contains attributes. With XML, an element can represent both data modeling entities and attributes. For example, XML can declare data modeling attributes as elements. These are then declared hierarchically as the children of a parent element that represents the data modeling entity.

To illustrate further, Figure 11-24 shows a typical data map and data dictionary as used by Visible Advantage, the CASE tool we used in Parts

**Figure 11-24**

CASE Tool details for the Element PERSON.



1 and 2 for Data Warehouse and Enterprise Portal design and development. In Figure 11-24 a Data Map window is displayed for the *Employment* area within XYZ. A Data Dictionary window is synchronized with it and tiled to its left.

When an entity is selected in the Data Map window of Figure 11-24, the metadata details of that entity are automatically displayed in the Data Dictionary window. We can see that the selected PERSON entity contains a number of attributes. In data modeling entity list notation (see Chapter 3), this is documented as:

```
PERSON(person id#, person name (last name, first name,
middle initial))
```

Note that *person name* is a group attribute, containing each attribute that follows in brackets. Expressing this entity in XML, PERSON and its attributes can be declared as elements, hierarchically nested as shown in Figure 11-25. The *Employment* model view in the figure is declared as the root name of the document, and is also called *Employment*.



**NOTE.** *PERSON* is an XML element, but it is called an “entity” in data modeling terminology (see Chapter 3). The metadata details describing the element PERSON in Figure 11-24 can be declared as XML attributes.

**Figure 11-25**

An Internal DTD for Figure 11-24, with XML tags and data content.

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE Employment
[
<!ELEMENT Employment ANY>
  <!ELEMENT PERSON (person_id, person_name)>
    <!ELEMENT person_id (#PCDATA)>
    <!ELEMENT person_name
      (last_name, first_name, middle_initial)>
      <!ELEMENT last_name (#PCDATA)>
      <!ELEMENT first_name (#PCDATA)>
      <!ELEMENT middle_initial (#PCDATA)>
    ]>
<Employment>
  <PERSON>
    <person_id>771144</person_id>
    <person_name>
      <last_name>Finkelstein</last_name>
      <first_name>Clive</first_name>
      <middle_initial>B</middle_initial>
    </person_name>
  </PERSON>
</Employment>
```

Figure 11-25 contains the data modeling entity PERSON, declared as XML element PERSON. As XML names are case-sensitive, the element name is in upper case—following the data modeling notation convention for entities in capitals as discussed in Chapter 3.

The *Employment* document in Figure 11-25 can also declare the other data modeling entities displayed in Figure 11-24. For example, SKILL and PERSON SKILL entities could be declared as the XML elements SKILL and PERSON\_SKILL.

The data modeling attributes *person id#* and *person name* in the entity list are declared as XML elements *person\_id* and *person\_name*. These element names have been declared in lower case, following the convention for data modeling attributes (see Chapter 3). The child list (*person\_id*, *person\_name*) within the PERSON element declares—with a comma-separator—that these are mandatory.

The group attribute (*person name*) in the entity list is declared in Figure 11-25 as the XML element *person\_name*, with child elements of (*last\_name*, *first\_name*, *middle\_initial*). These child elements are all mandatory. XML tags and data content at the end of Figure 11-25 illustrate the hierarchical nature of these element declarations through nesting. The *person\_id* element has a value of “771144” as an actual value for this primary key element.

## Creating XML DTD Files

We can now see why knowledge of the metadata used by an organization is essential if XML is to be used effectively. The examples in this chapter focused on metadata for one data modeling entity: PERSON, and declared some of its data modeling attributes. In real life, we saw in Parts 1 and 2 that an enterprise has many hundreds (or thousands) of entities, and thousands of attributes. To manually declare each of these to XML is impractical; an automated approach is needed.

The earlier chapters of the book used various methods to identify this metadata. In Part 1 we used Forward Engineering methods: strategic business planning (Chapter 2); data modeling (Chapter 3); strategic modeling (Chapter 4); and Decision Early Warning (Chapter 5). In Part 2 we used Reverse Engineering methods to extract and capture metadata that exists in legacy databases and systems. We used a CASE tool, Visible Advantage, to capture this metadata in a repository. With this CASE tool, we can use the defined metadata to generate automatically the

database tables, columns, and indexes needed to implement a Data Warehouse or Enterprise Portal.

Through our Forward Engineering and Reverse Engineering focus in Parts 1 and 2, the hard work has already been done. The metadata has been identified and captured. It can be used to generate the databases needed by the data warehouse or enterprise portal.

So why can't the metadata in the CASE tool repository also be used to generate the DTD files needed for XML?

We believe that an automatic XML DTD generation capability will be supported soon by some CASE tool vendors. It is not a difficult task as we will soon see. To misquote Neil Armstrong, *One small step for a CASE tool; one giant leap for business!* CASE vendors that provide this capability will be the winners in the new business and computing environment that is emerging with Internet, intranet and extranet technologies, with Java, and with XML.

Let us examine how this could be achieved. We will continue to use Visible Advantage as our CASE tool example; the concepts apply also to other CASE tools. The approach that we will use next is only one of several possible methods.

## Using CASE Tools to Generate XML DTD Files

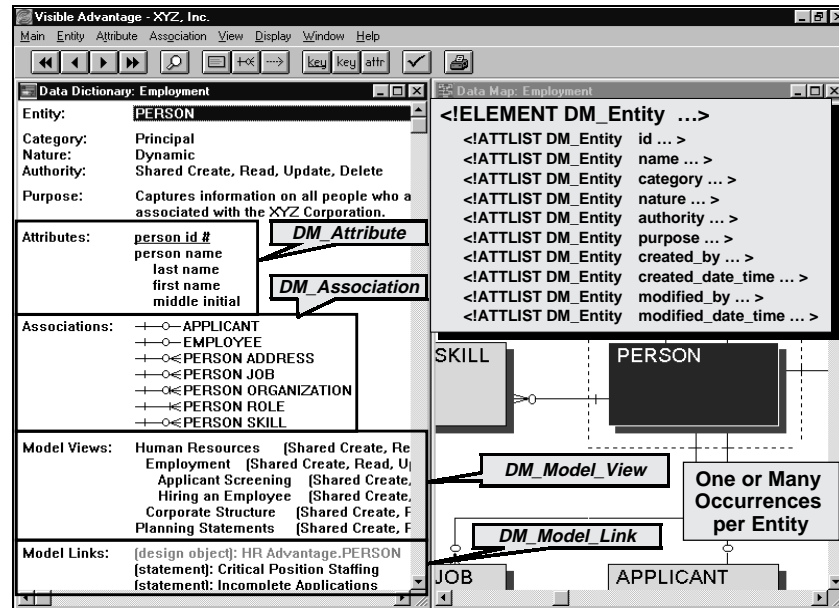
In Figure 11-24 the Data Dictionary window at left contains data modeling details, and also: all attributes of PERSON; associations to other entities related to PERSON; model views interested in PERSON; and design object and planning statement model links from PERSON. One or many occurrences of these exist for each entity as we discussed in Chapter 4. This is illustrated in Figure 11-26 by a box surrounding each group of multiple occurrences. We will discuss the labels attached to each box shortly.

The CASE tool uses these details as its metadata (*CASE metadata*), to manage details about enterprise entities, attributes, and associations (*enterprise metadata*) defined in its repository. *CASE metadata* can be used by XML applications when they process the enterprise data content identified by XML data tags declared from the *enterprise metadata*. This is the "giant leap" above.

Visible Advantage uses enterprise metadata definitions of entities and attributes in its repository to generate Data Definition Language (DDL)

**Figure 11-26**

CASE Tool details for the Element PERSON. Boxes surround multiple occurrences of CASE metadata details.



schema scripts automatically for DBMS products. As the DDL syntax can differ between DBMS products, Visible Advantage uses a scripting language to generate the DDL CREATE TABLE, CREATE INDEX and other DDL statements for the target DBMS. These specify table and column definitions generated from data modeling entity and attribute metadata, to install these in a physical database.

The logical source of metadata definitions for XML is the enterprise metadata captured in CASE tools as discussed in Parts 1 and 2. We believe that XML element and attribute declarations in the future will be automatically generated by CASE tools, using CASE metadata and similar scripting capabilities that these tools already use to generate DDL schemata automatically for different DBMS products. This will be a key enabler of XML, from the wider use of enterprise metadata defined by data administrators.

## Specifying Data Modeling Entities

The CASE metadata details in Figure 11-26 can be declared as attributes in an XML ATTLIST for the XML element: *DM\_Entity* (say). We have used this element name with the prefix *DM\_* (for data modeling) to

distinguish it from other CASE metadata elements. CASE tools typically support many methodologies. They may need to use *OO\_Entity* (or *OO\_Class*) for elements generated from object-oriented CASE metadata, for example. The prefix identifies the methodology.

Do not confuse this use of methodology prefixes with the prefixes that are used for namespaces, as discussed earlier. Each CASE tool vendor will define its CASE metadata. Namespaces can then be used (as defined for XML) to prevent name collisions with the XML names used by different vendors for their specific CASE metadata.

A fragment of the *DM\_Entity* XML ELEMENT and ATTLIST declarations defined from CASE metadata is displayed in Figure 11-26 and is defined in more detail in Figure 11-27.

Several incomplete XML attribute declarations in Figure 11-26 follow the *DM\_Entity* element declaration. Figure 11-27 shows that the XML attributes *id*, *name*, *category*, *nature*, *authority*, *created\_by*, and *created\_date\_time* are specified as “REQUIRED;” there must be one, and only one, occurrence of each of these XML attributes in a declared *DM\_Entity*. The attributes *purpose*, *modified\_by*, and *modified\_date\_time* are optional and so are specified as “IMPLIED.”

```
<?XML version="1.0" standalone="YES"?>
  <!DOCTYPE Employment
  [
    <!ELEMENT Employment ANY>
    <!ELEMENT DM_Entity
      (DM_Attribute+, DM_Association+, DM_Model_View+, DM_Model_Link*)>
      <!ATTLIST DM_Entity id ID #REQUIRED>
      <!ATTLIST DM_Entity name CDATA #REQUIRED>
      <!ATTLIST DM_Entity category (Principal | Type | Intersecting |
        Secondary | Structure | Role) #REQUIRED>
      <!ATTLIST DM_Entity nature (Dynamic | Static) #REQUIRED>
      <!ATTLIST DM_Entity authority (Create | Read | Update |
        Delete) #REQUIRED>
      <!ATTLIST DM_Entity purpose CDATA #IMPLIED>
      <!ATTLIST DM_Entity created_by CDATA #REQUIRED>
      <!ATTLIST DM_Entity created_date_time CDATA #REQUIRED>
      <!ATTLIST DM_Entity modified_by CDATA #IMPLIED>
      <!ATTLIST DM_Entity modified_date_time CDATA #IMPLIED>
    ]
```

**Figure 11-27**

XML ELEMENT and ATTLIST CASE Metadata Declarations for *DM\_Entity* in Model View Employment.

In Figure 11-26, the box surrounding the attributes for `PERSON` is labelled to indicate that each *Attribute* is defined by an XML element: `DM_Attribute` (say). Because many attributes exist for each entity, each is defined as a child element of `DM_Entity`. Similarly each *Association* is defined as a child element: `DM_Association` (say). Additionally, *Model View* and *Model Link* entries in Figure 11-26 are defined as child elements: `DM_Model_View` and `DM_Model_Link` (say).

Figure 11-27 shows that there must be at least one `DM_Attribute` child element (`DM_Attribute+`) for each `DM_Entity`. Each data modeling entity must at least have a primary key. The `+` suffix indicates that there can be more occurrences of `DM_Attribute`. These occurrences define other primary keys, foreign keys, or non-key attributes (see Chapter 3).

The `+` suffix for `DM_Association` (`DM_Association+`) specifies there must be one or more association occurrences in a `DM_Entity`. Good data modeling practice ensures that a data modeling entity does not exist as an isolated island of data; it is associated with at least one or more other entities. There must also be one or more `DM_Model_View` (`DM_Model_View+`) child elements, while there may be zero or more `DM_Model_Link` child elements (`DM_Model_Link*`) for each `DM_Entity`.

Figure 11-27 lists these as valid child element declarations of `DM_Entity` for each of these child elements. We will not refer to these further, except for `DM_Attribute` which will be used with `DM_Entity` to generate an XML DTD as shown in Figure 11-25.

The *Category*, *Nature*, and *Authority* details in Figure 11-26 can only have certain values. These values are specified when an entity is first entered or later changed in Visible Advantage. The valid values are shown in Figure 11-28 as a dialog box. Figure 11-27 therefore declares the *Category*, *Nature*, and *Authority* values in brackets. Valid *category* values are (*Principal | Type | Intersecting | Secondary | Structure | Role*). Valid *nature* values are (*Dynamic | Static*). Valid *authority* values are (*Create | Read | Update | Delete*). Only these values can be used; each of these XML attributes is `#REQUIRED`. Other XML attributes are declared as `#REQUIRED` or `#IMPLIED` as discussed earlier.

**Figure 11-28**

CASE metadata valid values for definition of a data modeling entity.

## Specifying Data Modeling Attributes

CASE metadata is also captured for data modeling attributes and associations, as well as for planning statements, activities, processes, objects, methods, physical database designs, and more. Each of these is also declared as an XML element as shown in Figure 11-26. For our purposes, however, we only need to consider the *DM\_Attribute* child element. For example, metadata from Figure 11-26 for the primary key attribute *person id#* is shown in Figure 11-29 as a dialog box.

The CASE metadata in Figure 11-29 is declared in Figure 11-30 as an XML ATTLIST for the XML element *DM\_Attribute*. This shows that each dialog box detail of Figure 11-29 is declared as an XML attribute of the XML element *DM\_Attribute*. Attributes in the declaration that are mandatory are shown as #REQUIRED. Those details that are optional (such as length and precision) are shown as #IMPLIED as they depend on the specified domain value. For example, domain values of *character*, *numeric*, and *decimal* require a *length* attribute. Domains of *numeric* and *decimal* may also optionally specify the *precision* as a number of decimal places.

*Alias* and *purpose* are optional attributes and are shown as #IMPLIED. But as for data modeling entities, good practice recommends that all data modeling attributes should have a defined purpose description.

**Figure 11-29**

CASE metadata for the *person id#* primary key attribute. These valid values are declared in Figure 11-30.

The screenshot shows a dialog box titled "Edit Primary Key Attribute". The fields are as follows:

- Entity:** PERSON
- Attribute:** person id
- Alias:** (empty)
- Purpose:** Uniquely identifies an occurrence of person.
- Domain:** System Generated Id
- Length:** (empty)
- Precision:** (empty)
- System Controlled:**
- Model View Authority:**
  - Read
  - Update
- Buttons:** Edit Domain, Spell, More..., OK, Cancel

The XML *type* attribute in Figure 11-30 specifies valid values of *Primary-Key*, *Foreign-Key*, *Non-Key*, and *Group* for data modeling attributes. For attributes that are children of another *DM\_Attribute* of *type*="Group" (such as *person\_name*), each child *DM\_Attribute* specifies its parent in the XML *group* attribute (such as *group*="person\_name"). An example of this is shown later in Figure 11-31.

Notice in particular the list of valid values for the *domain* attribute in Figure 11-30. This is #REQUIRED: it specifies the data type of the attribute defined in the enterprise metadata. Typical valid values are listed here. Visible Advantage can easily be extended to add other enterprise-defined data types to the list of valid domains. Examples are *State*, *SSN*, and *Skill Level*. Once added, any domain value can be easily selected as the data type for an attribute used in the enterprise.

We discussed earlier that data content in an XML document is expressed as character data (CDATA). We considered an XML data example showing a *customer\_balance* of \$15,500.00. The numeric characters used for this value must be converted by an XML application to the numeric value of \$15,500.00 before it can be used for calculation purposes. The data type of each enterprise metadata attribute can be determined by the XML application from the XML *domain* attribute in the XML *DM\_Attribute* element, as declared in Figure 11-30.

Figure 11-31 shows the *CASE Metadata* XML element tags and attribute data content that has been automatically generated from the

**Figure 11-30**  
XML ELEMENT and  
ATTLIST  
Declarations for  
*DM\_Attribute*  
based on CASE  
metadata valid values  
in Figure 11-29.

```

<!ELEMENT DM_Attribute
  <!ATTLIST DM_Attribute id          ID          #REQUIRED>
  <!ATTLIST DM_Attribute entity      CDATA      #REQUIRED>
  <!ATTLIST DM_Attribute attribute   CDATA      #REQUIRED>
  <!ATTLIST DM_Attribute type        (Primary-Key|Foreign-Key|
  Non-Key|Group) #REQUIRED>
  <!ATTLIST DM_Attribute group       CDATA      #IMPLIED >
  <!ATTLIST DM_Attribute alias       CDATA      #IMPLIED >
  <!ATTLIST DM_Attribute purpose     CDATA      #IMPLIED >
  <!ATTLIST DM_Attribute domain      (Character | Text | Flag |
  Money | Numeric | Decimal |
  Float | Integer |
  System_Generated_Id|Date|
  Date_Time|Time) #REQUIRED>
  <!ATTLIST DM_Attribute length      CDATA      #IMPLIED>
  <!ATTLIST DM_Attribute precision   CDATA      #IMPLIED>
  <!ATTLIST DM_Attribute authority   (Read | Update) #REQUIRED>

```

declarations of *DM\_Entity* in Figure 11-27 and of *DM\_Attribute* in Figure 11-30. The *CASE Metadata* content for PERSON from Figure 11-26 is now included in Figure 11-31 as the data content for the *DM\_Entity* occurrence of PERSON. *CASE Metadata* content for the *person id#* primary key displayed in Figure 11-29—and other attributes earlier displayed in Figure 11-26—are shown in Figure 11-31 as XML attribute content for *DM\_Attribute* elements of *person id*, *person name*, *last name*, *first name*, and *middle initial*.

The resulting XML *CASE Metadata* in Figure 11-31 can be automatically generated by CASE tools from *Enterprise Metadata*. This is analogous to generated CREATE TABLE declarations as the DDL for a target DBMS. When processed by an XML processor or XML database (see Chapter 15), the final *Enterprise Metadata* XML tags and content are as shown in Figure 11-32.

Similarly, other CASE metadata can be declared as XML elements for planning statements, activities, processes, objects, methods, database designs, and more. The generated enterprise metadata in Figure 11-32 can be used for import and export data interchange between different enterprise data sources. This enables XML to integrate dissimilar legacy files or relational databases. Examples of XML applications that integrate different data sources are discussed in Chapters 12 and 15.

The XML declarations in Figure 11-27 and Figure 11-30 document the CASE metadata used to manage data modeling entities and attributes in

```

<Employment>
  <DM_Entity id="E1245"
    name="PERSON"
    category="Principal"
    nature="Dynamic"
    authority="Shared Create, Read, Update, Delete"
    purpose="Captures information on all people who are, have been or may be
    associated with the XYZ Corporation"
    created_by="CBF"
    created_date_time="Feb 12, 1999 10:15:00" >
  <DM_Attribute id="A3152" entity="PERSON" attribute="person id" type="Primary-Key"
    purpose="Uniquely identifies an occurrence of person."
    domain="System Generated Id" authority="Read, Update" >
  <DM_Attribute id="A3153" entity="PERSON" attribute="person name" type="Group"
    purpose="To identify the person." >
  <DM_Attribute id="A3154" entity="PERSON" attribute="last name" type="Non-Key"
    group="person_name" purpose="A person's last name used to identify the person."
    edit rule="add now & modify later" domain="Character" length="30"
    authority="Read, Update" >
  <DM_Attribute id="A3155" entity="PERSON" attribute="first name" type="Non-Key"
    group="person_name" purpose="The first name of a person."
    edit rule="add now & modify later" domain="Character" length="30"
    authority="Read, Update" >
  <DM_Attribute id="A3156" entity="PERSON" attribute="middle initial"
    type="Non-Key" group="person_name" purpose="Middle initial of a person's
    name, used for identification of the person."
    edit rule="add now & modify later" domain="Character" length="30"
    authority="Read, Update" >
</Employment>

```

---

**Figure 11-31**

Generated CASE Metadata XML tags and data content for *DM\_Entity* and *DM\_Attribute*.

---

the Visible Advantage CASE tool. The generated CASE metadata in Figure 11-31 is used to import and export enterprise metadata between this and other CASE tools that declare CASE metadata differently. Each vendor can publish its CASE metadata in a unique namespace. Once published, these CASE metadata declarations can be used to define a common CASE metadata vocabulary. An approach can be used to integrate CASE metadata in Figure 11-31 with other CASE metadata that is similar to the multiple supplier application described in Chapter 12. This will enable enterprise metadata to be easily imported and exported between different CASE tools in the future, using a common repository.

We will now use the XML concepts covered in this chapter for other purposes: using XML as a Business Reengineering technology in Chapter

**Figure 11-32**

Output from XML Processor or XML database of generated *Enterprise Metadata* tags from Figure 11-31, with relevant data content.

```
<?xml version="1.0" standalone="yes"?>
  <!DOCTYPE Employment
  [
    <!ELEMENT Employment ANY>
    <!ELEMENT PERSON (person_id, person_name)>
    <!ELEMENT person_id (#PCDATA)>
    <!ELEMENT person_name
      (last_name, first_name, middle_initial)>
    <!ELEMENT last_name (#PCDATA)>
    <!ELEMENT first_name (#PCDATA)>
    <!ELEMENT middle_initial (#PCDATA)>
  ]>
  <Employment>
    <PERSON>
      <person_id>771144</person_id>
      <person_name>
        <last_name>Finkelstein</last_name>
        <first_name>Clive</first_name>
        <middle_initial>B</middle_initial>
      </person_name>
    </PERSON>
  </Employment>
```

12, and for integrated Business and Systems Reengineering in Chapter 13. We consider enterprise quality initiatives in Chapter 14. We conclude the book by discussing other business and systems reengineering opportunities in Chapter 15 that are based on the use of XML, and the central role taken by Enterprise Portals in deploying these as XML applications.

## REFERENCES

- Document Object Model (DOM) Specifications—<http://www.w3.org/TR/REC-DOM-Level-1/>
- Hackathorn, R. (1998) *Web Farming for the Data Warehouse*, Morgan Kaufman, ISBN: 1-55860-503-7. Includes use of XML for data sources from Internet (368 pages).
- Harold, E. R. (1998) *XML: Extensible Markup Language*, IDG Books, ISBN: 0-7645-3199-9. Covers XML, XSL, and XLL with HTML (426 pages + CD-ROM).
- Holzner, S. (1998) *XML Complete*, McGraw-Hill, ISBN: 0-07-913702-4. Covers XML with focus on Java (516 pages + CD-ROM).

- XML Namespaces Specifications—<http://www.w3.org/TR/REC-xml-names/>
- Resource Description Framework (RDF) Specifications—<http://www.w3.org/Metadata/RDF/>
- St Laurent, S. (1998) *XML: A Primer*, MIS Press [IDG Books], ISBN: 1-55828-592-X. A good basic introduction to XML (348 pages).
- Extensible Linking Language (XLL) Specifications—<http://www.w3.org/XLL/>
- Extensible Markup Language (XML) Specifications—<http://www.w3.org/XML/>
- Extensible Style Language (XSL) Specifications—<http://www.w3.org/XSL/>
- W3C, WWW Consortium and associated specifications—<http://www.w3.org/>
- XML Books. The IES web site at <http://www.ies.aust.com/~ieinto/> and the Visible Australia web site at <http://www.visible.com.au/> both list many HTML, XML, Data Warehousing, and Corporate Portal books. Each listed book has a direct link to Amazon.com so that you can review these books then purchase those that interest you.

## XML Information Web Sites

- WWW Consortium—Specifications and Standards—<http://www.w3.org/>
- Microsoft XML Scenarios Web Site—<http://microsoft.com/xml/scenario/intro.asp>
- XML.com Web Site—<http://www.xml.com/>
- James Tauber's XMLINFO Web Site—<http://www.xmlinfo.com/>
- James Clark's XML Web Page—<http://www.jclark.com/xml/>
- Robin Cover's XML Resources—<http://www.sil.org/sgml/xml.html>
- Microsoft XML Site—<http://www.microsoft.com/xml/>
- Microsoft XML Workshop Web Site—<http://www.microsoft.com/workshop/xml/toc.htm>
- Web Farming Web Site—<http://www.webfarming.com/>

## XML Development Tools: Validating Parsers

- Microsoft's MSXML Parser—<http://www.microsoft.com/standards/xml/xmlparse.htm>
- IBM's Alphaworks XML for Java Parser—<http://www.alphaworks.ibm.com/formula/xml>
- Data Channel's DXP Parser—<http://www.datachannel.com/products/xdk/DXP/index.html>
- Object Design's eXcelon XML database—<http://www.objectdesign.com/>
- TclXML Parser (based on TCL—instead of Java or C)—<http://tcltk.anu.edu.au/XML/>

## XML Development Tools: XML Browsers

- Peter Murray Rust's Jumbo Browser—<http://vsms.nottingham.ac.uk/vsms/jumbo>
- Netscape's Mozilla Browser—<http://www.mozilla.org/>
- Microsoft Internet Explorer 5.0—<http://www.microsoft.com/>
- Netscape Communicator 5.0—<http://www.netscape.com/>