

Chapter 1

Introduction

You know what a “legacy application” is? It’s one that works.—Bill Cafiero

This chapter introduces the approach presented in the book and discusses the business and technical rationales for it. Based on these, the overall features of the approach and the major design decisions are reviewed. The chapter concludes with a description of the primary audiences and suggestions for how to use the book.

■ The Problem

So, what’s the problem? You probably picked up this book (or are browsing this sample chapter on the Web) because the title sounded like it might have something to do with your situation. The Preface probably fleshed out that impression a bit more. By now you probably think you have the book pretty well scoped out, and you may not be that far from the truth. But to save you a bit of time, let me be very specific about the problem (or, more accurately, the problems) that this book addresses.

The primary problem is this: You have one business application that imports and exports data in XML formats, and one that doesn’t. You need to make these two applications talk to each other. The XML-enabled application may be yours, or it may belong to someone with whom you (or your application’s users) need to exchange data. (Imagine an important customer or government agency sending

you a letter that says, “You will receive orders from us in XML format by January 15 or be assessed a \$50 penalty for each paper order.”) The legacy application is most probably yours. However, the shoe may be on the other foot: Your application speaks XML, but the other guy’s application doesn’t. He expects *you* to deal with it. (Imagine an important customer sending you a letter that says, “You will receive orders from us by EDI by January 15 or be assessed a \$50 penalty for each paper order.”)

There are really two perspectives to this primary problem, and this book addresses both. According to one perspective, the application developer should just fix the stupid application so it can speak XML. From an end user’s viewpoint, this is perfectly reasonable, and I’ll deal with the problem from this perspective.

From the other perspective, whoever developed the application can’t or won’t support XML by the time you need it (which might have been last week). There may be several reasons for this. The vendor of a commercial package may not see sufficient market demand. The vendor may have retired the product or, even worse, gone out of business. If the application was developed in-house, the original developers may be gone, retired, or dead. So, there are quite a few reasons why you might need to come up with a solution on your own. In this book I’ll also deal with the problem from this perspective.

That, in about 400 words, sums up the primary problem this book addresses. However, in solving this problem from the perspective of someone who needs to come up with a solution on their own (as opposed to that of the developer who just needs to fix the stupid application), we find that the solution might apply to similar problems. Why is that? Well, if we have a method to go from a legacy flat file to XML and a method to go from XML to a legacy flat file, we have a method to go from a legacy flat file to a different legacy flat file (with XML in between). The same thing holds for other common formats such as comma-separated values (*CSV*) or Electronic Data Interchange (*EDI*). By coming up with a solution to the primary problem, we find that we have a general solution for all sorts of file format conversion problems. That is also what this book is about.

Before we leave the problem definition and start addressing the solution, there are just a couple more points we need to clarify. First, what, exactly, is a legacy application? Aside from Mr. Cafiero’s pithy observation, for the purposes of this book we’ll refer to a legacy application as any working application that doesn’t currently provide native support for XML (that is, the application can’t produce XML documents as output and consume XML documents as input).

Second, why use XML with a legacy application? That is a reasonable question, but answering it is beyond the scope of this book. In this book I assume that you have already answered that question for yourself. You’re going to use XML and you have figured out why; all you want to know is *how*. There has been enough good general material written on the benefits of application integration and electronic commerce that I feel there’s very little I can add for justification.

In regard to XML in particular, I assume that you have one application that uses XML and a legacy application that doesn't, and that you need to integrate them. To do that you need either to convert between XML and the format required by the legacy application or to build native XML support into the legacy application. You know what you need to do and why you need to do it. This book is intended to help you with how to do it.

■ What Do We Need in a Solution?

When we ask, "What do we need?" we're talking about requirements. There are two types of requirements: functional and nonfunctional (the latter are also known as quality requirements or system constraints). The former have to do with what the system is supposed to do. The latter have to do with how it does what it does. Both are important, and both determine the overall approach of this book.

Beyond the overall dictate of solving the problem, two distinct sets of requirements are imposed on the solution by technical end users on the one hand and by application developers on the other. I'll talk a little later about why I'm dealing with both, but for now if you don't care about the other group you can just skip the relevant paragraphs.

Functional Requirements

The technical end user who has an application that doesn't speak XML more than likely needs the solution to do one or more of the following:

- Convert an XML-formatted file to a flat file
- Convert a flat file to an XML-formatted file
- Convert an XML-formatted file to a CSV file
- Convert a CSV file to an XML-formatted file
- Convert an EDI-formatted file to an XML-formatted file
- Convert an XML-formatted file to an EDI-formatted file

A user may want the solution to support other formats, but CSV, flat file, and EDI should handle most cases. For example, an end user may also need to get data out of a database (relational, hierarchical, or otherwise) and put it into an XML format, or go back the other way. Sorry, but these types of problems are a bit beyond our scope. I will, however, give in Chapter 12 an overview of some approaches for doing things like this. When I present the approaches, you'll understand why problems like this exceed our scope a bit.

The developer who has an application that doesn't speak XML has some fairly simple requirements:

- Enable the application to export data in an XML format
- Enable the application to import data from an XML format

The word “an” is very important here. I don't say “a specific XML format”; I say “an” XML format. Why? Because once we have data in an XML format, it is fairly easy to convert it to another XML format. You want to make your life simple? Don't try to anticipate all the XML formats your users will want. Give them one, and let them convert. I'll have more to say about this later.

Those are the primary functional requirements. Both groups may have a secondary functional requirement to be able to validate the format of an XML document (which conventionally is referred to as an **instance document**). When validating an instance document, the format of the document is usually defined in an XML 1.0 **Document Type Definition (DTD)** or a schema written in the World Wide Web Consortium (W3C) **XML Schema language**. Both of these define things such as the Element and Attribute names used in the document and the overall structure of the document. The W3C XML Schema language allows documents to be defined in much greater detail than is possible with a DTD. These will both be discussed in Chapter 4.

Validation may need to occur either before the document is read or after it is produced. For going to and from EDI, end users may want to check that the EDI-formatted file complies with the relevant standard either before they read it or after they write it. The approach presented in this book satisfies most such requirements.

Some other functional requirements have to do with enabling business applications to support the exchange of business data with other organizations electronically. Those requirements are within the scope of this book, and I'll discuss them in Chapters 12 and 13.

So, we should be pretty clear by now about what we want the solution to do. We now need to figure out how we want it to do it.

Nonfunctional Requirements: Good, Fast, and Cheap

Remember the old joke about good, fast, and cheap, that you can pick only two out of the three? Well, with any luck we might be able to get you all three. Good is pretty relative (as well as unspecific), so I won't go into a lot of detail here. The same applies for fast. So let's talk about cheap and a few other typical requirements.

If you're an end user and you're paying a relatively small price for this book instead of buying a full-featured Enterprise Application Integration (EAI) system

like Mercator, we understand each other. If you find the word *cheap* a bit harsh, think of it this way: You can't cost-justify purchasing a full-featured package to convert a few files on an infrequent basis. You can't justify spending more on utilities than you did on the business application. Take your pick, but cheap is fine with me. If you're a developer, you want to support XML without spending the next release's entire new features budget. Return on investment is what it's all about. The bottom line is the bottom line.

So, beyond cheap, what's important? Simple usually goes hand in hand with cheap, as does easy. Beyond these biggies, there are several more that developers and information technology staff usually care about. I'll assume that you do too.

- **Maintainability:** We need to be able to keep the solution running and add new functionality without too much trouble. Code that we add to the application should be easy to fix and enhance.
- **Reusability:** When we develop code to solve one problem, we would like to be able to easily reuse it to solve similar problems. We don't want to keep reinventing the wheel or repeating the same snippets of code in several routines.
- **Flexibility:** We shouldn't be straightjacketed into a single approach that handles just one type of situation. We should be able to modify it to handle changing circumstances.
- **Modularity:** The solution should be nicely broken up into manageable pieces. This helps with reusability and maintainability.
- **Portability (or platform independence):** The solution should work on a variety of platforms so we can move it to a different platform with few, if any, changes.

However, we do need to draw the line somewhere and note some things that are of lesser importance. In this book I assume that you don't care as much about the following requirements.

- **Performance:** In performance we're concerned with resource usage such as CPU time required to perform conversions, memory usage, and disk space usage. This is most often a concern when a system is used for many purposes simultaneously or is otherwise somewhat constrained in a key resource such as disk space or memory. Due to the relatively low price of hardware these days, and due to the fact that the typical machine is a stand-alone PC, we're not going to be overly concerned about performance.
- **Real-time processing:** If we think of performance as applying to resource usage, then real-time processing (as opposed to old-fashioned batch

processing) is more relevant to the elapsed time the system takes to complete an action. We aren't concerned with a user clicking a mouse or hitting a key and waiting for an immediate response. We are concerned with batch, file-oriented interfaces.

- Support for Web services, frameworks, and other bleeding-edge technology: We're talking legacy applications here. If you want to make your applications do WSDL, UDDI, .NET (to be discussed in Chapters 12 and 13), or whatever, that's not cheap, simple, and easy. If you're concerned about this type of stuff, you need something beyond this book. Many of the techniques discussed in this book are quite relevant to getting data into and out of the XML formats typically used by Web services, but what happens after the handoff to the Web service is beyond our scope.

Summing it all up into just a few words, in this book I present a fairly simple, pragmatic approach that can be implemented relatively quickly and at fairly low cost.

■ The Overview of a Solution

At the heart of this approach are a number of independent programs that can be used alone or together to build a solution appropriate for the specific problem you need to solve. The programs we will develop in this book are:

- XML to CSV Converter: Reads and validates an XML document, and writes it out in CSV format
- CSV to XML Converter: Reads a file in CSV format, produces an equivalent document in XML format, and validates it
- XML to Flat File Converter: Reads and validates an XML document, and writes it out as a flat file in a user-specified structure (read from another XML file)
- Flat File to XML Converter: Reads a flat file in a user-specified structure (read from another XML file), writes an equivalent document in XML format, and validates it
- XML to EDI Converter: Reads and validates an XML document, and writes it out in EDI format according to a user-specified structure (read from another XML file)
- EDI to XML Converter: Reads an EDI interchange in a user-specified structure (read from another XML file), writes an equivalent document in XML format, and validates it

In addition, we'll rely on an XSLT transformation program to transform between different XML formats. *XSLT* is the W3C recommendation on **Extensible Stylesheet Language Transformations**. The good news is that such XSLT utilities are easily (and freely) available, and we'll use an existing one rather than build one. I will present the essential techniques you need to know to code XSLT stylesheets that drive the transformations between different XML formats.

How does the solution measure up so far against our requirements? As you can easily see, it meets all the functional requirements for technical end users. Because I'll walk you through the coding techniques used in each of the programs, it should help developers meet their requirements too.

Cheap, simple, and easy? You bet. As I said before, you can download some pretty decent XSLT transformation programs for free from the Web. You can download executable versions of the programs developed in this book from the book's Web site. If you want to tweak them a bit or even use them as a basis for your own slightly different programs, you can also download the source code. The solution is very modular, since we use a number of independent programs. It is also maintainable because most of what you have to change to deal with different file organizations is done in XML files rather than in code. The XML to XML transformations are driven entirely by XSLT stylesheets, which are themselves nothing more than XML documents in the XSLT language. XSLT also offers a great deal of portability (as long as you don't use nonstandard extensions!). Other aspects of portability and platform independence are determined more by the specific implementation, so I'll discuss those later.

■ Architecture

According to the seminal book on software architecture, written by Shaw and Garlan [1996], software architecture “involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns. In general, a particular system is defined in terms of a collection of components and interactions among those components.” What we're most concerned with here is what Shaw and Garlan refer to as the architectural pattern or style.

Style is the general model of how the system is composed and how the different parts interact with each other. You've probably run across object-oriented, layered, and client-server styles of software architecture. We'll use a style that should be familiar to you (especially if you do a lot of work from a UNIX shell prompt) but that you may not know by the name used here: **pipe and filter**.

The basic idea is that you have a number of different programs, or filters, that read input in one format, transform it, and produce output in a different format.

The output of one filter is connected to the input of another via a pipe. The filters are independent of each other, with only the constraint that the output of one filter be useable as the input of the next. In our approach, the pipes are disk files or sets of files in directories, but they could just as easily be in-memory data structures, database tables, or something else. Figure 1.1 shows the basic pipe and filter style.

A modified pipe and filter style (Figure 1.2) allows the filters to accept input or parameters other than just the input files. This allows the filter programs to produce different types of output. In reality, most pipe and filter systems in use are of this modified style; ours is too.

Want to build a system that converts CSV to XML? Easy—just use the CSV to XML Converter program as the filter (Figure 1.3).

You don't like the XML that this produces? You want to produce an XML file that conforms to your customer's invoice specification? No problem. Add the XSLT transformer to build a more general purpose application (Figure 1.4).

Now let's say you have a customer (or maybe you're a consultant and your client has a customer) who wants to send you orders using the X12 EDI standard. The version of Peachtree Accounting you're using processes only CSV files, and you don't want to buy an EDI system for just this one application. We can easily build a system to solve this problem too (Figure 1.5).

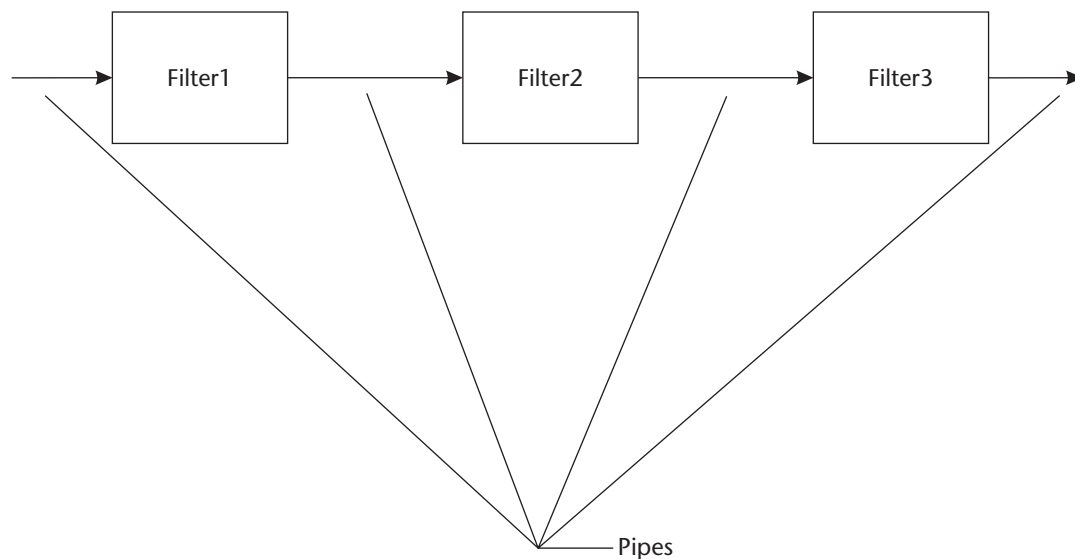


Figure 1.1 Basic Pipe and Filter Style

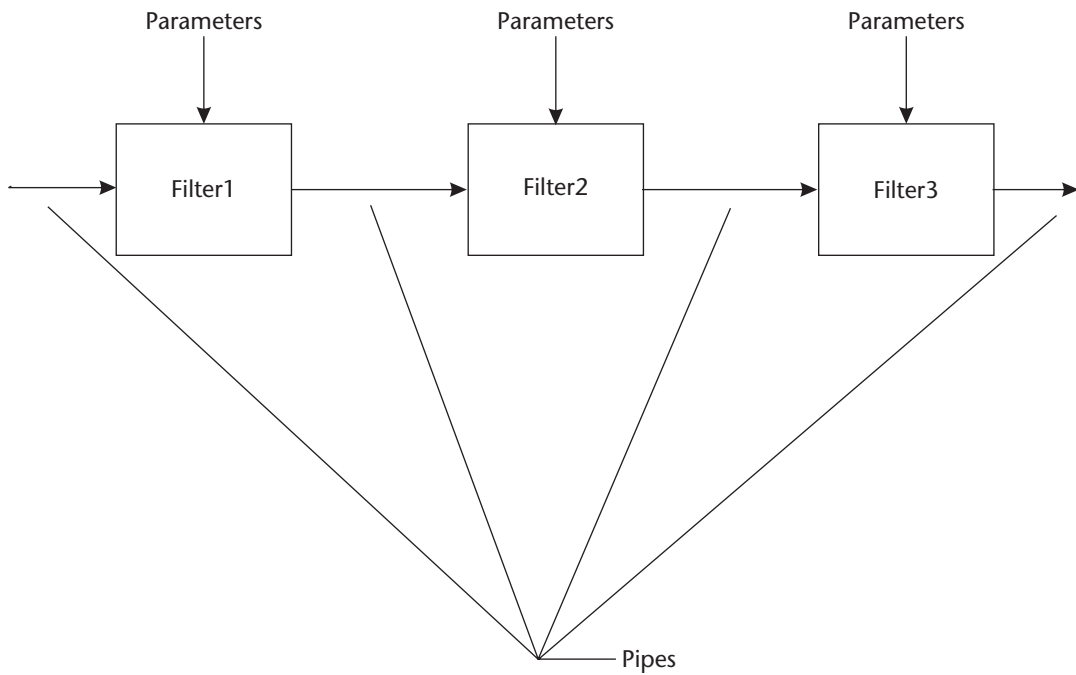


Figure 1.2 Modified Pipe and Filter Style

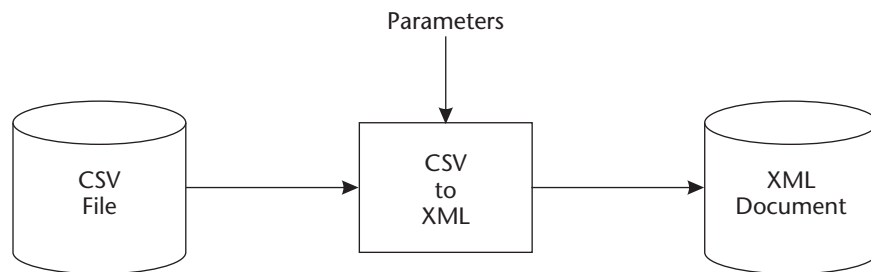


Figure 1.3 CSV to XML Application

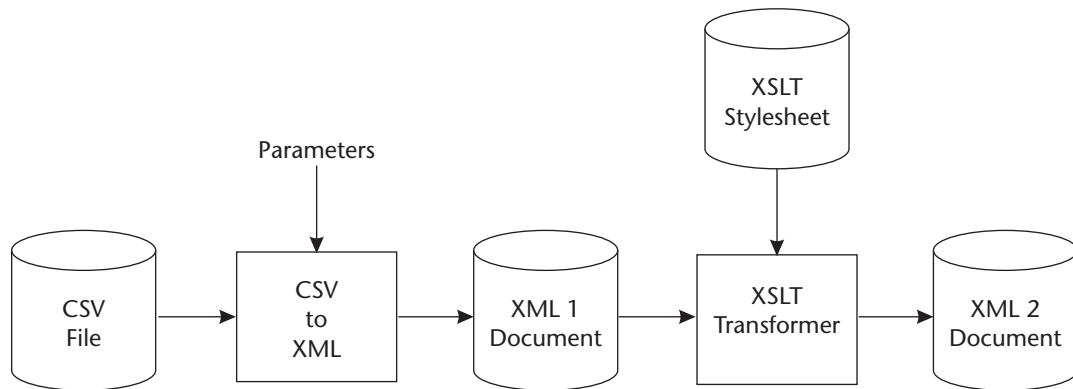


Figure 1.4 CSV to XML to XML Application

By now it should be pretty obvious that you can take the building blocks here and make pretty much whatever pipe and filter system you might need. Once you think about it, the pipe and filter style is a very logical choice for the type of format conversion problems we're dealing with in this book. Even if you're a developer who wants to enable your product to work with XML, knowing that the XML your application reads or writes is probably going to live in this type of environment can help you make some better-informed design decisions.

Beyond our basic pipe and filter approach, it would be nice to have something that helps us build these specialized little pipe and filter applications. In Chapter 11 we'll put a lot of the pieces together and go over just how such a system might look.

Why Not Use XSLT for Everything?

There are two different opinions about using XSLT for non-XML transformations where the source or target is a format other than XML. One camp says that XSLT can and should be used for everything. The other camp says that XSLT is wonderful for transforming an XML document into a different flavor of XML document but that there are better ways to deal with non-XML formats.

So, why am I in the latter camp? It's an issue of using the right tool for the job. If you're camping and you forget an essential tool, you can open a can of beans with a hatchet or maybe even cut down a tree with your Swiss Army knife. But those aren't exactly the easiest ways to do things. It's the same way with XSLT. XSLT has a fairly concise language for identifying items in source XML documents and creating corresponding items in target XML documents. However, when dealing with other formats the language is not so elegant. For example, to

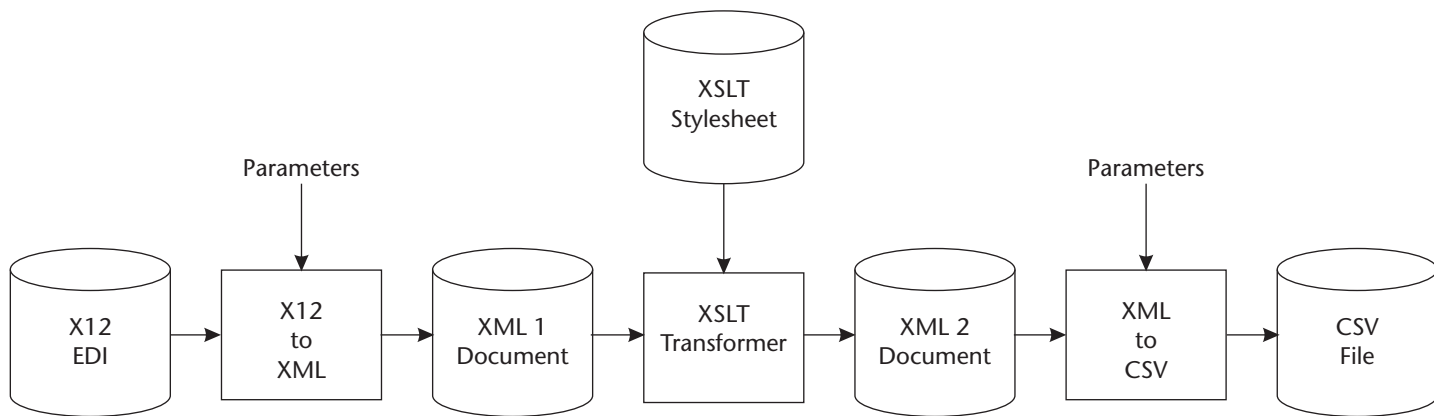


Figure 1.5 EDI to XML to CSV Application

parse an input fixed record length file it is necessary to use things like substring functions. Parsing an input CSV file requires repeated use of a search function to find the delimiter and then substring functions to isolate individual fields. Creating target documents in non-XML formats requires similar operations.

There are some drawbacks to using XSLT for everything. The most efficient design approach using just XSLT is to try to do everything with one XSLT stylesheet. In this approach the code that parses the input or formats the output is mixed up with the code that does the *logical* transformation between documents. By “logical” I mean correlating input fields to output fields, along with any manipulation or value conversions required. One example of a logical transformation is concatenating a person’s first and last names into a single customer name. This approach of trying to do everything with a single stylesheet doesn’t support modularity, reusability, understandability, or maintainability very well.

A less efficient approach that is somewhat more modular and reusable is to use two XSLT transformations. One transforms the non-XML format to or from XML, and the other performs the logical transformation of XML to XML. This approach is somewhat better than a single XSLT stylesheet that does everything. However, it is still hampered by a limitation in the XSLT 1.0 language that restricts it to generating just one output file per transformation. As we’ll see in later chapters, several types of transformations require us to generate multiple XML documents from one non-XML input file. Some XSLT 1.0 processors include nonstandard functions that provide this functionality. There is discussion of supporting it in XSLT 1.1, but it isn’t there yet as a standard. In addition, for more complex operations like dealing with EDI acknowledgments and keeping track of the documents exchanged with different trading partners, it is much more appropriate to use a programming language like Java, C, or C++ than XSLT.

An advantage of using a more conventional programming language for handling non-XML files is that we can implement a general purpose solution rather than a series of XSLT stylesheets that are each useful for only one particular file organization. In our general purpose solution fairly simple parameter files handle the differences in file organization. These files are themselves coded as simple XML documents.

In addition to these considerations, presenting the techniques for handling non-XML files with XML files is a good introduction to the type of XML programming that will be required to XML-enable legacy business applications. In other words, it’s a good teaching tool.

I should mention one final caveat regarding XSLT. There are, of course, wide variations among different XSLT processors, but as a group they tend to have poorer performance than commercial EAI or EDI conversion utilities. XSLT, and therefore the architecture, is probably not well suited to production situations that need to handle large amounts of data in a short amount of time. However,

earlier in the chapter I noted that in this book I assume that cost and portability are more important considerations than performance. XSLT fits the bill.

Two Implementations of the Architecture: Java and C++

In this book I present two different implementations of our architecture. Why? Basically, for reasons of platform independence and portability. However, I'm also doing it out of a pragmatic recognition that there isn't just one dominant programming language, and I want to be able to speak to both camps.

We can use an analogy to home building in order to understand the two implementations of a single software architecture. The analogy isn't exact, but there are enough high-level similarities between building houses and building software to serve our purpose.

When an architect designs a house, he or she first establishes the number and types of rooms that the client wants as well as the general characteristics. Then the architect develops a detailed floor plan with approximate room dimensions, locations of windows and fixtures, and so on. At this point the architect may not have decided yet on specific materials. The actual house could be constructed out of wood, brick, or even recycled tires; the materials will be selected during a second phase of home design. The first phase of home design corresponds to software architecture. Just as we could build two houses with the same floor plans but with different materials, we can implement architecture in two different languages.

So, this book presents implementations in Java and C++. The Java implementation uses the Sun Microsystems and Apache Foundation XML libraries, and the C++ implementation uses Microsoft's MSXML library.

Why Java and C++? Well, most applications these days are written in either Java or C++. Most environments in which those applications are run use standard Java or C++ libraries. "But," you say, "my legacy application is written in COBOL!" This gets to the bottom line of cheap, simple and easy. Reading (parsing), writing (serializing), and validating XML documents are not easy tasks. This is especially true in the case of validating against schemas written in the W3C XML Schema language. It takes a lot of complicated, hard-to-write, and hard-to-debug code to do such things. We don't want to write that code. We want to use libraries that someone else has written. And because we're concerned about portability and platform independence we would like to use standard libraries rather than proprietary approaches. These standard libraries are primarily object-oriented, which again makes them a good fit for Java and C++. In Chapter 12 I'll talk a bit more about some other alternatives, including some for procedural languages such as COBOL. However, our most important nonfunctional requirements point us to standard, object-oriented application programming interface (API) libraries.

Even with the standard API libraries there are implementation choices other than Java and C++. For example, Perl, Python, and Visual Basic all have fairly sophisticated XML support. Visual Basic even uses the same API library, MSXML, that we'll use for C++. The techniques discussed in the next two chapters on basic conversions involving CSV files and XML could be implemented in these languages. However, the more sophisticated conversions and utilities built later in the book are complex enough that these languages would not be appropriate. In addition, very few *real* applications are written in these languages. So, we'll keep our focus fairly limited and be concerned primarily with the Java and C++ implementations. However, the basic techniques will still be relevant if you want to cobble together some simple utilities using Perl or whatever. As I'll discuss shortly, the basic operations using the standard APIs are presented in pseudocode that is language independent. Even though I won't offer the specific Perl or Visual Basic syntax, you could still port the techniques fairly directly to these languages from the pseudocode design.

If you are an end user, you really don't care much whether the program is written in Java, C++, Perl, COBOL, ALGOL, or fuggly old IBM 360 Assembler. In this case, the types of libraries and utilities you can install and run in your environment dictate your choice of implementation. Java can run pretty much anywhere, so you may want to install the relevant Java libraries and filter programs. If you can't install things directly where your application is running, maybe you can hijack a PC networked to your application's box and install the conversion libraries and utilities there. In that case, the WIN32 C++ implementation might be a bit easier since there are fewer pieces to install.

Before wrapping up this section there are a few other developer-oriented issues we need to discuss. We've already decided that we're going to use a standard API library. Even with this restriction we still have a few choices. I have chosen to use the W3C's Document Object Model (the *DOM*). But before discussing why I'm using the DOM, let's look at what it is.

The Document Object Model

Like nearly all other basic XML technologies, the DOM is a creation of the W3C, which defines the DOM as:

a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page. [W3C 2002]

What does this mean? Basically, the DOM specifies a set of program routines, or an API. It is not itself an object; but similar to how Java does things, it

specifies an interface to the object model, with that interface composed of a number of different methods. Since it is just a specification, there isn't any actual W3C-produced code. So, in general terms, why use the DOM? Again, the W3C says:

“Dynamic HTML” is a term used by some vendors to describe the combination of HTML, style sheets and scripts that allows documents to be animated. The W3C has received several submissions from members companies on the way in which the object model of HTML documents should be exposed to scripts. These submissions do not propose any new HTML tags or style sheet technology. The W3C DOM WG is working hard to make sure interoperable and scripting-language neutral solutions are agreed upon. [W3C 2002]

What does this really mean? It means they developed the DOM so that application programmers can have a standardized way to deal with documents. It is also notable that the DOM was developed first to deal with *HTML* documents and not *XML* documents. However, the XML world has embraced it wholeheartedly.

The hope was that if enough tool vendors (like Microsoft, Sun, Oracle, and so on) developed DOM implementations, application developers would have a fairly easy, standardized way to deal with documents programmatically. This has come to pass, and there are enough different implementations that porting HTML or XML code between implementations is a lot easier than it would have been if there were no DOM. The DOM also makes it fairly easy to talk about coding in a generic way; most of the XML processing logic discussed in the book's utilities is applicable to both C++ and Java environments.

The W3C has defined several different “levels” of the DOM. Level 1 was released in October 1998, Level 2 was released in November 2000, and Level 3 was still a working draft at the time of this book's preparation. Basically, each level has a few more features and a bit more functionality than the previous version. For the most part, we don't need to be too worried about the DOM level we're using. Both the C++ and Java libraries used in this book support Level 2, with a few additional features from Level 3.

Why Use the DOM?

We already agreed that we don't want to reinvent the wheel. So, from a programming perspective, the DOM fits the bill. However, it's not the only language-independent API we can use. There is one other predominate API, called *SAX* (Simple API for XML, not the musical instrument that John Coltrane and a certain past U.S. president played). *SAX*, like the DOM, specifies an interface that can be used to process XML documents in a somewhat platform-independent fashion. It doesn't matter a lot in practical terms since it's becoming customary

for most XML libraries to support both, but the DOM is an “official” W3C Recommendation and SAX isn’t.

How are the DOM and SAX different, and why have I chosen the DOM? The DOM handles a complete document as an object in memory, specifically a tree. SAX, on the other hand, is an event-driven API. It calls a specific method (via callbacks) for each type of XML construct it encounters as it reads in an XML instance document. The most important difference between the two APIs is that the DOM makes it fairly easy to build or manipulate a document in memory. Some SAX implementations offer ways to build documents, but there isn’t a standard SAX approach. So, SAX is not as well suited in this respect. On the other hand, SAX is well suited to parsing very large XML documents that might cause performance problems (or even crashes) if we tried to read them into memory using the DOM. Since it’s much simpler to use just one API if possible, I’m using the DOM in this book. I’ll discuss this topic again in Chapter 12, but the choice of the DOM for this book’s utilities will become more understandable as the overall design approach progresses.

I should note, though, that while the current versions of the DOM make it fairly easy to build and manipulate XML documents in memory, through Level 2 the DOM doesn’t specify how XML (or HTML) documents are actually read or written. Curious but true. I would have thought those things to be fairly fundamental, but I guess the HTML heritage as well as different priorities in the W3C left those details to the implementers. The draft Level 3 requirements do finally deal with such things. The fact that actually reading or writing XML documents isn’t specified in Level 2 really isn’t too much of a problem since most XML libraries provide the functionality. However, because it isn’t specified there are often differences in the particular methods used. We’ll see this as we develop the C++ and Java code later in the book.

One final note on the DOM: While many implementations use the exact W3C method and object names in their libraries, Microsoft’s MSXML library sometimes slightly modifies the names specified by the DOM. It also allows many of the DOM object properties (or attributes) to be accessed directly rather than just through get methods. This means that the C++ code in this book may have to be modified if you want to use it with a different C++ DOM library. (At least a global search and replace will be required. Other changes will probably be required as well, but those are beyond the scope of this book.)

■ How to Use This Book

How you use this book depends very much on what you want out of it. As I said earlier, the book is primarily focused on the requirements of two distinct sets of

readers: technical end users and developers. I've attempted to structure the book so that both groups can easily use it. I'll first cover how the chapters are organized, then I'll suggest how someone from either group might want to approach a chapter.

Chapter Organization

Some chapters (like this one) are mostly general text that might be relevant to the needs of both groups. These are organized in a fashion appropriate to the topic.

All chapters contain a section of references or of resources (sometimes both). These provide information on references cited in text and suggestions for further study. I generally try to be as specific with resource URLs as I can, but in many cases I can give only general pointers. Some Web sites are reorganized so frequently that links to specific internal pages quickly become invalid. In such cases the referenced URL should have either a link to the desired resource or a search facility to help you find it.

The chapters that present utilities are organized slightly differently. Here's the general organization by sections. (*Note:* This is the general organization; some chapters may not have each of these sections.)

- **Requirement:** A description of what the utility is supposed to do, including general specifications of the input, processing, output, parameters, and restrictions.
- **Running the Utility:** A guide to running the utility from the command line, including description of the arguments and options.
- **Sample Input and Output:** A review of sample input and output to help further illustrate the utility's functionality.
- **Design:** The design of the utility. A high- to mid-level design overview is presented in text, with a more detailed pseudocode description that highlights DOM usage. Chapters 6 through 9, in which more complex utilities are developed, break the design section into two aspects. The first deals with high-level design or overall design considerations, while the second deals with detail design. The detail sections are not intended to be read front to back (unless you happen to need help falling asleep); they are included for reference.
- **Java Implementation:** Highlights of the Java implementation of the code, with listings of important code fragments and explanation.
- **C++ Implementation:** Highlights of the C++ implementation of the code, with listings of important code fragments and explanation.
- **Enhancements and Alternatives:** Suggestions for how the utility might be enhanced plus discussion of different design and coding approaches.

End users will probably use the utilities chapters differently than developers will, as described in the next section.

Notes for Primary Audiences

Technical End Users

If you're a technical end user, you probably mostly care about what a utility does and how to run it. Therefore, scan the beginning of the chapter, and review the requirements. If the utility provides what you need, look at how to run it. You can skip the rest of the chapter.

Developers

If you're a developer, you probably care less about what a utility does than about how it does it. Scan the beginning of the chapter, review the requirements, and study the design. Then look at the Java or C++ implementation, depending on your language choice. You'll probably also find the enhancement suggestions interesting. You can find out how to download the source code by referring to the book's Web site, and you can get more information on related topics from the Resources section.

Other than those specific suggestions, I'll offer only this: This book is meant to be a tool kit. I don't expect anyone to read it cover to cover. Use what you need, scan what looks useful for the future, and ignore the rest.

Chapter Summaries

To aid you in exploring this tool kit, here is a chapter-by-chapter guide to what you'll find in it.

- Chapter 1, Introduction: You're reading it.
- Chapter 2, Converting XML to CSV: This chapter presents a very basic approach for converting an XML document to a CSV format file. I start with this topic for two reasons: (1) CSV is still the most common universal format used by desktop applications, and (2) reading an XML document is a bit easier than writing one, so we'll start with the easy stuff first. This chapter covers the basic techniques for reading and parsing XML files using the DOM, so it is a foundation chapter for developers. The example file used is a generic address book listing, since it is most amenable to the type of processing performed by this utility.
- Chapter 3, Converting CSV to XML: This is the converse of the preceding chapter. Here we take a CSV file as input and convert it to an XML document. It is also a foundation chapter for developers since it deals with the basic

techniques for creating an XML document in memory using the DOM and writing it out to disk. I again use the generic address book listing as an example.

- Chapter 4, Learning to Read XML Schemas: This book is intended primarily for people who are going to be reading schemas, not writing them (or, at most, writing fairly simple schemas). The W3C XML Schema Recommendation is pretty obtuse, and there are some big, fat (and good) books out now that cover it in gory detail. (My Web site, <http://www.rawlinsecconsulting.com>, has recommendations for a few.) I cover here the basics of what you need to know, going over the essential features and a few different approaches to schema design that you're likely to encounter. This chapter is important because many of the remaining chapters use schemas.
- Chapter 5, Validating against Schemas: In this chapter the utilities developed in Chapters 2 and 3 are modified to support validating an XML instance document against a schema on input and validating it against the schema before writing it out. Various examples of validation failures are presented with the types of error messages that the different APIs offer.
- Chapter 6, Refining the Design: The utilities presented in Chapters 2 and 3 are fairly basic in functionality and simple in design and coding. The other utilities developed in the book have more sophisticated functionality and therefore require more complex designs and coding. This chapter lays the foundation for these utilities by discussing several issues related to processing XML documents and non-XML grammars and presenting the base classes used by the utilities developed in Chapters 7, 8, and 9.
- Chapter 7, Converting CSV Files to and from XML, Revisited: This chapter builds on the foundation of Chapter 6 and develops CSV conversion utilities that are more capable than those developed in Chapters 2 and 3. For example, XML works best when a disk file contains a single XML business document. However, most batch imports and extracts in CSV format bundle many separate business documents into one disk file. These utilities combine many XML documents into one CSV file and split one CSV file into many XML documents. The concept of driving the conversion with a parameter file, coded as an XML document, is presented in this chapter. For the XML to CSV conversion, a purchase order is used as the sample document since small and medium-sized businesses commonly receive such documents from larger customers. The CSV to XML conversion uses an invoice as the sample document. In this fashion we deal with the two most basic documents used in the procurement cycle.
- Chapter 8, Converting Flat Files to and from XML: Most common CSV files have a uniform format. Every row is in the same format. Flat files are not so simple. Each document usually has at least a header record, at least

one type of detail record, and often a trailer or summary record. They may have many, many more types of records. They therefore require more complicated processing than CSV files. As with our CSV utilities, an XML file describes the structure of the flat file. We'll again use a purchase order as the sample document for converting from XML and an invoice as the sample document for converting from a flat file.

- Chapter 9, *Converting EDI to and from XML*: This chapter presents the most complex type of conversion we'll cover, between XML and EDI formats. The grammar of EDI syntaxes is analyzed, and algorithms appropriate for processing EDI are presented. This chapter also covers other issues around EDI, such as processing Functional Acknowledgments and handling control numbers, and the preliminary functionality needed to support these requirements.
- Chapter 10, *Converting from One XML Format to Another with XSLT*: This chapter presents the basics you need to know about using XSLT to transform a document from one XML format to another XML format. It covers the most commonly used features as well as a few things to watch out for. This chapter is primarily targeted for technical end users, though developers need to be aware of what XSLT can do so they can design around it.
- Chapter 11, *Using the Conversion Techniques Together*: This chapter presents a few use cases and simple script examples for using the utilities together to solve some common conversion problems. It also presents the requirements and high-level design for the initial version of the Babel Blaster open source EDI/EC/EAI file conversion program. It builds on the utilities presented in the previous chapters to develop a comprehensive system for solving many types of file conversion problems.
- Chapter 12, *Building XML Support into a Business Application*: The preceding chapters presented various techniques for reading and writing XML documents. However, other issues need to be addressed before deciding on an overall approach to XML-enabling an application. This chapter discusses issues such as integrating with existing code in the least disruptive manner, selecting data for import or export, deciding whether and what to validate, choosing design options other than the DOM, and processing data in other formats (such as relational databases).
- Chapter 13, *Security, Transport, Packaging, and Other Issues*: Having the data in XML isn't enough. You also have to be able to get it to and from your trading partners. They may want to receive and send the data over a private Value Added Network (VAN) if they are using EDI, but more than likely they will want to use the public Internet. To do that you'll need

ways to package several XML documents into one bundle (or unpack them) and handle security. This chapter addresses these other B2B considerations. I'll present a basic approach for assessing your real needs in these areas and suggest some practical strategies for supporting those needs without getting too much over your head into bleeding-edge technology.

Conventions

To help make it clear what you're looking at, I adopted some formatting conventions in this book as shown below.

Fragments of source code in Java, C++, pseudocode, and various types of XML syntax:

Source Code Program

```
main()
{
    // This is a C++ program
    ...
}
```

Single lines of code:

```
myDoc = new DocBuilder;
```

Fragments of non-XML files:

```
Doe,John,12 Lee Street,Boston,MA,01303
```

Command line program execution:

```
java MyProgram input output -a argument
```

The first time an important term is used it appears in **bold** font, for example, **pipe and filter**. The first time an acronym is used it appears in *italics*, for example, *W3C*, accompanied by the full name set in regular font. In some cases, important acronyms, for example, *XSLT*, first appear in both bold and italic.

Several of the terms used in this book have different meanings, depending on the context. For example, we can talk about an element's attributes in an XML instance document, or when discussing the DOM we can talk about the attributes (or properties) defined for any of its interfaces, including a DOM element. To help

keep things straight, whenever I refer to a named DOM entity I will capitalize the term. In addition, Elements and Attributes discussed in the context of instance documents will always be capitalized.

■ What You Need to Use This Book

In this book I assume that you have certain prerequisite knowledge. You also need to have several software tools, source code, and some XML files in order to make full use of this book. This section tells you what you need to already know as well as what those additional resources are, where they are, and how to get them.

What You Should Already Know

All readers should already have a basic understanding of XML. Familiarity with XML 1.0 DTDs may be helpful but is not absolutely necessary. If you aren't already familiar with XML, there are several good introductions to the subject. On my Web site I give a few recommendations. If you don't mind doing a bit of somewhat technical reading, you can also review the XML Recommendation at the W3C Web site, <http://www.w3.org>.

End users should be comfortable with the general concept of file format conversion, that is, taking a file that exists in one format and converting it to another file in a different format. Some rudimentary programming experience will be helpful if you plan to develop XSLT stylesheets or write scripts that use the utilities developed in the book.

Application developers should be familiar with either Java or C++ and their respective basic library functions. For those interested in the C++ implementation, prior experience with Microsoft's Component Object Model (COM) is helpful but not required.

Web Site and Contact Information

The Web site for this book has nearly everything you need (or pointers to the information you want). There you will find:

- Executable versions of the utilities in both Java jar file format and Win32 binary format
- Full source code in C++ and Java

- Example schemas and XSLT stylesheets
- Sample input and output files
- Other sample and support files
- Updated links (current at the time of final editing)

The URL for the book's Web site is: <http://awprofessional.com/titles/032115940>.

You can find additional examples and frequently updated references to other good books at <http://www.rawlinseconsulting.com/booksupplement>.

Please feel free to contact me with any questions, comments, errata, bug reports, and so on via e-mail at mike@rawlinseconsulting.com.

General Software

If the only XML syntax work you expect to do is to code parameter files for running the utilities in this book, probably all you need is a good text or programming editor such as emacs. For Windows users, Microsoft's XML Notepad (Beta version 1.5, May 3, 1999) is available as a free download. I would give you a URL for it, but Microsoft has an annoying tendency to move things around on its site. So, I won't go any farther than pointing you to <http://www.microsoft.com> and suggesting that you search for "XML Notepad". XML Notepad should ensure that you create a **well-formed XML document**, that is, one that complies with W3C's XML Recommendation in certain key aspects. (*Note:* When used with Internet Explorer version 4.01 and earlier, XML Notepad converted all characters to uppercase. Very few people are likely to have this problem now since most people run later versions of Internet Explorer. If you run into this problem, you'll need to upgrade Internet Explorer to version 5.0 or later.) The Windows Notepad, emacs, or vi will probably do just fine for very light use.

However, if you're going to do much with XSLT or schema design, spending a couple hundred dollars for a good XML tool will probably be a good investment. I list my favorites below.

- XMLSPY by Altova GmbH: All you need is the Integrated Development Environment (*IDE*). It helps you design schemas in various schema syntaxes, create instance documents, validate both, and develop and test XSLT stylesheets. XMLSPY runs only on Win32 platforms (Windows NT, 2000, XP, 98, ME). A free evaluation download is available from <http://www.xmlspy.com>. I used version 4.3 for this book.
- TurboXML (formerly XML Authority) by TIBCO: Similar in functionality to XMLSPY, as a Java-based application it runs on many platforms including

Windows, Linux, Solaris, HP-UX, and other UNIX platforms. A free evaluation download is available from <http://www.tibco.com>.

Other tools with similar functionality also exist (for example, eXcelon's Stylus Studio). However, I've not tested or extensively reviewed any of them, so I don't have much to say about them. I have heard of some free, open source tools, but they were still in development when I wrote this book. If I find any good ones I'll post links on my Web site.

If you would like a different way to view an XML document (but not edit it), Internet Explorer version 5.5 is handy. It allows you to expand or contract elements with children so that you can view the complete document tree or just focus on certain branches.

This completes my recommendations for general purpose software. The developer recommendations for the Java and C++ environments are discussed next, but I must raise an important cautionary flag first.

CAUTION Don't Change Horses in the Middle of a Stream!

I feel a bit of a nag when reminding people to pay attention to the basics. But it seems that we often forget them or make exceptions when dealing with new technology. After you have downloaded the latest tools and APIs, don't update them until you've finished your project! With updates seeming to come every month or two in the XML world, there is a great temptation to apply them just as soon as they come out. However, I've had too many experiences on other projects over the years of starting to work in the morning and discovering that code that worked last night doesn't work today, even though I didn't change anything. After hours of debugging I found that new libraries had been installed and the problem wasn't my code at all. Save the updates of the APIs until your code is stable and thoroughly tested. Then download the latest, rebuild, and retest.

Java Software

The Java code in this book was developed using the libraries listed below. It may compile and run under later versions, but it hasn't been tested that way. Due to changes in support for the W3C XML Schema language, the versions listed are the minimum that I recommend.

- Java 2 Standard Edition Version 1.3.1_03 (<http://java.sun.com/j2se/1.3>): To compile the code you'll need the full software development kit (SDK). If you're only going to download and run the book utilities, you'll just need the Java Runtime Environment (JRE).

- Java XML Pack, Spring 02 (<http://java.sun.com/xml/downloads>): If you're only going to run the Java utilities, you don't need the full kit. However, you will need the Java jar archive files from the kit.
- Xerces2 Java Parser 2.0.1 (<http://xml.apache.org>): This software is required for all Java XML development and runtime presented in this book. For the most part I use standard Java API for XML Processing (JAXP) features, but the routines to serialize a DOM document use specific Xerces2 routines. (Although the precise name of the parser is "Xerces2," I'll observe the common custom of referring to it as just "Xerces" unless I mean to refer specifically to this version of Xerces.)
- Xalan-Java 2.3.1 (<http://xml.apache.org>): This software is required for Java-based XSLT transformations. Other Java-based XSLT implementations may work but have not been tested.

Refer to my Web site for specific instructions about how to install the jar files for your particular version of Java.

Note: The book's Java code was developed using version 1.2 of Sun's JAXP. The Java XML Pack has an implementation of the JAXP 1.2 specification that uses Xerces2 as the default parser. (Xerces2 conforms to the JAXP 1.1 specification.) To be sure that you get the latest bug fixes I recommend downloading the most recent version of Xerces2 from the Apache site.

The Java code was developed and tested with Borland's JBuilder Version 6 on Windows NT Workstation version 4. It was also tested on Windows 98.

C++ Software

As previously mentioned, the C++ implementation presented here is based on Microsoft's MSXML library, listed below. The code was developed and tested with MSXML version 4.0. This is the minimum version required. The code may work with later versions but has not been tested. Changes will be required to make it run with different DOM implementations.

- MSXML 4.0 Microsoft XML Core Services (<http://www.microsoft.com/downloads>): This is required for both development and runtime. MSXML runs as a COM service.
- MSXSL.EXE Command Line Transformation Utility (no version number, last updated 9/10/2001; <http://msdn.microsoft.com/downloads>). This is required for runtime XSLT transformations from command line.

The C++ code in this book was developed under Visual C++ 6.0 and tested on Windows NT Workstation 4.0 and Windows 98. (No, you don't need .NET!)

MSXML, Win32, and COM as Legacy Technologies?

Incredible but true! When I conceived this project, MSXML 4.0 (with full support for the final W3C XML Schema Recommendation) was still pretty new. However, before I really got going, Microsoft came out with .NET. Not only does .NET offer a different DOM API than MSXML (though reportedly it still uses MSXML under the hood), it is a completely new and different framework. XML is only a small piece.

When this happened I became a bit concerned that working with the DOM via MSXML's COM interfaces would be dealing with obsolete technology. However, on reflection I decided that it fit very well with the thrust of the book. I just didn't expect to run into legacy applications and technology in this particular area! Considering the paradigm shift that .NET imposes and the impact of migrating from Visual Studio 6.0 to Visual Studio .NET, I think there may be an audience for this book for some time to come.

Bottom line: This book gives you a way to do XML from Win32 without having to migrate to .NET.

Alternatives to MSXML?

While struggling to get the COM-related stuff working and fully debugged, I began to question my decision to use MSXML instead of some other DOM implementation that worked with C++. But, given the realities of C++ applications on Win32, I still think that MSXML is the best choice for this book. Many development shops are all Microsoft and wouldn't look at APIs from any other source. MSXML is fairly mature, installs easily, and is widely considered to offer some of the best support for W3C XML Schema. However, if you are comfortable with the licensing terms of the Apache C++ Xerces implementation, you can save yourself a lot of the COM headaches by using it instead of MSXML.

■ For Developers

This section describes the coding approach, conventions, and style I chose to use in this book. If you're reading this book as a technical end user, you may want to skip ahead to the next section.

General Coding Approach and Conventions

The Java and C++ code presented in this book uses object-oriented techniques. The DOM and all its various parts are object-oriented, as are the Java file operations and

the C++ file operations. (I have chosen to use the C++ classes instead of the old-style C libraries.) That said, most of what we programmers really care about is procedural in nature, that is, the code that lives and works inside methods. We care most about how to call methods that manipulate DOM objects. We don't care as much about everything else.

I have not gone out of my way to make this code object-oriented. In the beginning it is fairly simple and not necessarily very heavily object-oriented. However, as the design progresses and matures through the book, I do use more object-oriented techniques when they promote reuse and extension. I'm enough of an old-school programmer to be a bit concerned about the performance implications of declaring and freeing a lot of objects dynamically. I generally tend to avoid doing so when I can. However, we're valuing reusability and extensibility over performance in this design. So, there are cases when we do create a lot of objects dynamically at runtime rather than declaring them statically at compile time. If the code lives on and it turns out to be a dog at runtime, we can investigate more efficient designs. A modular, object-oriented approach also helps us in that regard.

I have not chosen to construct elaborate object models for the non-XML entities manipulated in these programs. In addition there aren't very many class diagrams or other things as found in the Unified Modeling Language (*UML*). We're going to keep it simple and focus on the essentials.

That said, here are a few other notes and general rules on coding approach and style.

- **Clarity over cleverness:** As any seasoned programmer knows, there are ways to write programs that are exceedingly clever. Such cleverness often yields short, efficient programs. However, in many cases these come at the expense of clarity, since it sometimes makes it harder to understand what the program is really doing. In this book where being clear means using a bit more code at the expense of being as clever as possible, I do the former rather than the latter. You can be clever in your own programs if you want, but in trying to get across some basic concepts I would rather be clear. We will follow the KISS principle: Keep it simple, stupid!
- **Error handling:** Since these are basic utilities intended for light use and teaching concepts, I did not implement elaborate error handling. However, all parsing and DOM exceptions are caught and enough information is reported to enable someone reasonably experienced in XML to fix a problem.
- **DOM extensions:** Some DOM implementations, notably Microsoft's MSXML in our case, offer extensions to the methods, functions, and interfaces defined in the DOM. To try to make the code platform independent and

to keep the Java and C++ implementations as alike as possible, I avoid using these extensions in the code. However, where the extensions offer alternatives to the approaches I take in the code, I often comment on the alternative in the text.

- File organization: Each class is coded in its own source file. The name of the source file generally matches the name of the class. This is, of course, standard in Java but not in C++.
- Header files: For C++, each class has a separate header file.
- Naming conventions: Upper camel case is used in the XML examples and for class names in the programs. Lower camel case is used for method names and variables. These choices seem to follow the prevailing conventions. With few exceptions abbreviations are avoided. Variable names in programs are prefixed with an abbreviation indicating the data type. Although in Java and C++ the line between variables and classes is sometimes hard to determine, I won't use this prefixing convention for classes.
- Comments: The code is liberally commented. I have tried to err on the side of having too many comments rather than not enough. I hope you find them sufficient without being too tedious and obvious.
- Formatting: Opening and closing brackets for code blocks appear on their own lines. Lines generally break around column 65. I have tried to make indentation and continuations consistent in the source files, using spaces and no tabs.
- Pseudocode: The pseudocode presented here is based on Programming Design Language (PDL) [Caine and Gordon 1975]. This commonly used pseudocode detail design language is basically structured English with a few reserved words. I see no particular merits in it over other pseudocode languages, but I picked it to lend some consistency to the pseudocode. My usage extends PDL by including calls to specific DOM methods. Appendix B presents a summary of pseudocode conventions used in this book.

Additional C++ Considerations

A few words are in order regarding my approach to using C++. In particular, I want to point out handling of strings, exceptions, and constants.

The final ISO and ANSI C++ standard has extremely useful string classes (at last!). These are much easier to use and less prone to runtime exceptions than old C-style char arrays, char pointers, and the C string library. However, these string classes are still not universally supported “out of the box” by many major C++ compilers. They weren't supported natively by Visual C++ 6.0, which I'm

using for this book. Even though some add-on open source and proprietary class libraries do support these classes, many developers use only what is standard from the compiler vendor. In addition, a lot of the legacy C++ code currently in production doesn't use these string classes. Finally, neither MSXML nor the Apache Xerces C++ API that is its best alternative use the ISO/ANSI string classes. MSXML uses COM strings (as we'll discuss in Chapter 2 and Appendix C), and the Xerces C++ API uses its own XML char class. They both support conversions to and from char arrays, not ISO/ANSI strings. So, for all these reasons I'm sticking with char arrays, char pointers, and the old C string library.

In the early days of C++ compilers, throwing exceptions for common runtime errors was discouraged because exception handling wasn't very efficient. The general view is that compilers have gotten a lot more efficient in this regard. However, a lot of the standard C++ library functions still use status codes or return values rather than throw exceptions for every little thing the way that Java does. For this reason I generally use the approach of returning status values from functions rather than throwing exceptions.

Regarding constants, some programmers believe that using `#define` rather than `const` to define constants is "evil" [Cline 2002]. I certainly prefer the Java style of defining constant class members. However, I found it extremely awkward to define constants that could be shared across classes using this approach with Visual C++ 6.0. There may be better ways to do it with compilers that provide better support for the final ISO/ANSI C++ standard. I'm sure there are people who are more clever about C++ programming than I am. At any rate, I've taken the easy way out in this book and used old C style `#defines`.

The general picture you should draw from this discussion is that the C++ code presented here does not necessarily represent current best practices in C++ coding. It instead represents "good" practices (I hope) as of the mid-1990s. However, I'm not really too concerned about that. If your legacy C++ applications are very old, they probably don't reflect current best practices either. Your code may look very similar to mine. On the other hand, if you *are* using more up-to-date compilers and following current best practices (for example, using the string class instead of char arrays), by all means keep doing what you're doing and don't regress. You should, however, still be able to follow my code and update the techniques to your current best practices as appropriate.

■ How You Can Use the Utilities and Code

Welcome to the wonderful world of copyrights and licenses. All the utilities and code developed in this book are open source. Their copyrights are owned by the

good people at Addison-Wesley, but you can use the utilities and code under the terms of the GNU General Public License (GPL), Version 2, included as Appendix A. This is the same license under which Linus Torvalds makes the Linux operating system available. I hope you and your organization will feel comfortable using this stuff without having to get the lawyers involved.

In the final analysis the terms of the GPL take precedence, but here is what I intend.

- You may use the executable, binary forms of the utilities in any way you wish for any purpose, whether it is public, private, commercial, not-for-profit, and so on.
- If you use the binaries, you do so with the understanding that no warranty of any kind is offered by anyone. The only exception is the case where you have explicitly executed a contract for warranty support with someone. (Just so the lawyers and my editor at Addison-Wesley don't have anything to worry about, let me make it very clear that your purchase of this book doesn't imply such a warranty contract. If it did, the book would be a lot more expensive. However, we thank you for buying it anyway!)
- You may freely distribute the source code and binaries in their original form, provided that they are accompanied by the GPL and that you give appropriate notice of the Addison-Wesley copyright as well as a disclaimer of warranty. As is the case with Linux, you may charge a reasonable fee for your copy or distribution costs. You may offer warranty support for a fee.
- You may modify or use parts of the source code in your programs so long as you retain the existing copyright notices and make your modifications available under the GPL.

The synopsis above applies to everyone. Here are my intentions for more specific groups of readers and users.

- If you are a consultant, you may make the binaries available to your clients for a nominal fee to cover actual copy costs. You may modify the source code and offer the resulting binaries to your customers, again for only a nominal copy fee, so long as you observe the GPL and make those modifications freely available to others (like me!) under the terms of the GPL. You may accept payment from your clients for performing the modifications, but you need to make sure they understand that they don't own the modified code. You may contract with your clients to offer warranty support.

- If you are a vendor of proprietary business applications software that does not natively support XML, you may make the binaries available to your customers for a nominal fee to cover copy or distribution costs. If you want to build native XML import/export support into your business application, you may use the algorithms and DOM processing techniques in your proprietary product, but you may not use the source code in your product.
- If you are a vendor of proprietary EAI, EDI translation and management, or other middleware software, you may use the contents of this book as an aid to your development efforts. You may use the more common and generic techniques and algorithms. You may not use either the unique design approaches or the source code.

In short, I hope that you benefit indirectly from the programs and the source code and that you feel free to share them with others. However, don't look to me for free warranty support, and don't try to make money directly from this work.

(Oh, and just to make sure I haven't forgotten anything: Everything I just said applies to the binaries and the source code but not to the book itself. It, too, is copyrighted by Addison-Wesley, but we want everyone to pay for copies. We have to get *something* out of this!)

■ References

- Cafiero, Bill. 2002. Remarks while leading a panel discussion at the semi-monthly meeting of the Dallas/Fort Worth Electronic Commerce Forum, July 18, in Irving, Texas.
- Caine, S., and E. K. Gordon. 1975. "PDL—a Tool for Software Design." In the American Federation of Information Processing Societies' *Proceedings of the 1975 National Computer Conference, Anaheim, CA, 19–22 May 1975*. Montvale, NJ: AFIPS Press, pp. 271–276. Available online at <http://www.cfg.com/pdl81/pdlpaper.html>, accessed July 11, 2002.
- Cline, Marshall. 2002. "C++ FAQ Lite." Available online at <http://www.atd.ucar.edu/software/c++-faq/newbie.html#faq-29.7>, accessed March 27, 2003.
- Shaw, Mary, and David Garlan. 1996. *Software Architecture—Perspectives on an Emerging Discipline*. Upper Saddle River, NJ: Prentice Hall.
- W3C (World Wide Web Consortium). 2002. Document Object Model main page. Available online at <http://www.w3.org/DOM/>, accessed July 10, 2002.

■ Resources

Apache Software Foundation: <http://www.apache.org>
Essential XML Quick Reference, by Aaron Skonnard and Martin Gudgin (Boston, MA: Addison-Wesley, 2002). I don't directly recommend very many other books in this book, but this is one I suggest without reservation. If you deal with XML on any technical basis at all, this book needs to be in your collection.

Microsoft's XML Resources: <http://msdn.microsoft.com/xml>

Sun Microsystem's Java Technology and XML: <http://java.sun.com/xml>

World Wide Web Consortium: <http://www.w3.org>

The XML Cover Pages: <http://xml.coverpages.org>