1

XSLT in Context

COPYRIGHTED MATERIAL

This chapter is designed to put XSLT in context. It's about the purpose of XSLT and the task it was designed to perform. It's about what kind of language it is, how it came to be that way, and how it has changed in version 2.0; and it's about how XSLT fits in with all the other technologies that you are likely to use in a typical Web-based application. I won't be saying very much in this chapter about what an XSLT stylesheet actually looks like or how it works: that will come later, in Chapters 2 and 3.

The chapter starts by describing the task that XSLT is designed to perform—**transformation**—and why there is the need to transform XML documents. I'll then present a trivial example of a transformation in order to explain what this means in practice.

Next, I cover the different ways of using XSLT within the overall architecture of an application, in which there will inevitably be many other technologies and components, each playing their own part. We then discuss the relationship of XSLT to other standards in the growing XML family, to put its function into context and explain how it complements the other standards.

I'll describe what kind of language XSLT is, and delve a little into the history of how it came to be like that. If you're impatient you may want to skip the history and get on with using the language, but sooner or later you will ask "why on earth did they design it like that?" and at that stage I hope you will go back and read about the process by which XSLT came into being.

What is XSLT?

XSLT (which stands for eXtensible Stylesheet Language: Transformations) is a language that, according to the very first sentence in the specification (found at `http://www.w3.org/TR/xslt20/`), is primarily designed for transforming one XML document into another. However, XSLT is also capable of transforming XML to HTML and many other text-based formats, so a more general definition might be as follows:

XSLT is a language for transforming the structure and content of an XML document.

Why should you want to do that? In order to answer this question properly, we first need to remind ourselves why XML has proved such a success and generated so much excitement.

XML is a simple, standard way to interchange structured textual data between computer programs. Part of its success comes because it is also readable and writable by humans, using nothing more complicated than a text editor, but this doesn't alter the fact that it is primarily intended for communication between software systems. As such, XML satisfies two compelling requirements:

- *Separating data from presentation*: the need to separate information (such as a weather forecast) from details of the way it is to be presented on a particular device. The early motivation for this arose from the need to deliver information not only to the traditional PC-based Web browser (which itself comes in many flavors), but also to TV sets and WAP phones, not to mention the continuing need to produce print-on-paper. Today, for many information providers an even more important driver is the opportunity to syndicate content to other organizations that can republish it with their own look-and-feel.
- *Transmitting data between applications*: the need to transmit information (such as orders and invoices) from one organization to another without investing in bespoke software integration projects. As electronic commerce gathers pace, the amount of data exchanged between enterprises increases daily, and this need becomes ever more urgent.

Of course, these two ways of using XML are not mutually exclusive. An invoice can be presented on the screen as well as being input to a financial application package, and weather forecasts can be summarized, indexed, and aggregated by the recipient instead of being displayed directly. Another of the key benefits of XML is that it unifies the worlds of documents and data, providing a single way of representing structure regardless of whether the information is intended for human or machine consumption. The main point is that, whether the XML data is ultimately used by people or by a software application, it will very rarely be used directly in the form it arrives: it first has to be transformed into something else.

In order to communicate with a human reader, this something else might be a document that can be displayed or printed: for example, an HTML file, a PDF file, or even audible sound. Converting XML to HTML for display is the most common application of XSLT today, and it is the one I will use in most of the examples in this book. Once you have the data in HTML format, it can be displayed on any browser.

In order to transfer data between different applications we need to be able to transform information from the data model used by one application to the model used by another. To load the data into an application, the required format might be a comma-separated-values file, a SQL script, an HTTP message, or a sequence of calls on a particular programming interface. Alternatively, it might be another XML file using a different vocabulary from the original. As XML-based electronic commerce becomes widespread, the role of XSLT in data conversion between applications also becomes ever more important. Just because everyone is using XML does not mean the need for data conversion will disappear.

There will always be multiple standards in use. For example, the NewsML format for exchanging news stories (`http://www.newsml.org/pages/index.php`) has wide support among Western newspaper publishers and press agencies, but attracts little support from broadcasters. Meanwhile, broadcasters in Japan are concentrating their efforts on the Broadcast Markup Language (`http://xml.coverpages.org/bml.html`). This has a very different scope and purpose; but ultimately, it can handle the same content in a different form, and there is therefore a need for transformation when information is passed from one industry sector to the other.

Even within the domain of a single standard, there is a need to extract information from one kind of document and insert it into another. For example, a PC manufacturer who devises a solution to a customer problem will need to extract data from the problem reports and insert it into the documents issued to field engineers so they can recognize and fix the problem when other customers hit it. The field

P1: IML
WY020-01 WY020-Kay WY020-Kay-v2.cls July 7, 2004 17:53

engineers, of course, are probably working for a different company, not for the original manufacturer. So, linking up enterprises to do e-commerce will increasingly become a case of defining how to extract and combine data from one set of XML documents to generate another set of XML documents: and XSLT is the ideal tool for the job.

At the end of this chapter we will come back to specific examples of when XSLT should be used to transform XML. For now, I just wanted to establish a feel for the importance and usefulness of transforming XML. If you are already using XSLT, of course, this may be stale news. So let's take a look now at what XSLT version 2.0 brings to the party.

Why Version 2.0?

XSLT 1.0 came out in November 1999 and has been highly successful. It was therefore almost inevitable that work would start on a version 2.0. As we will see later, the process of creating version 2.0 has been far from smooth and has taken rather longer than some people hoped.

It's easy to look at version 2.0 and see it as a collection of features bolted on to the language, patches to make up for the weaknesses of version 1.0. As with a new release of any other language or software package, most users will find some features here that they have been crying out for, and other additions that appear surplus to requirements.

But I think there is more to version 2.0 than just a bag of goodies; there are some underlying themes that have guided the design and the selection of features. I can identify three main themes:

❑ *Integration across the XML standards family*: W3C working groups do not work in isolation from each other; they spend a lot of time trying to ensure that their efforts are coordinated. A great deal of what is in XSLT 2.0 is influenced by a wider agenda of doing what is right for the whole raft of XML standards, not just for XSLT considered in isolation.

❑ *Extending the scope of applicability*: XSLT 1.0 is pretty good at rendering XML documents for display as HTML on screen, and for converting them to XSL Formatting Objects for print publishing. But there are many other transformation tasks for which it has proved less suitable. Compared with report writers (even those from the 1980s, let alone modern data visualization tools) its data handling capabilities are very weak. The language is quite good at doing conversions of XML documents if the original markup is well designed, but it's much weaker at recognizing patterns in the text or markup that represent hidden structure. An important aim of XSLT 2.0 is to increase the range of applications that you can tackle using XSLT.

❑ *Tactical usability improvements*: Here we *are* into the realm of added goodies. The aim here is to achieve productivity benefits, making it easier to do things that are difficult or error-prone in version 1.0. These are probably the features that existing users will immediately recognize as the most beneficial, but in the long term the other two themes probably have more strategic significance for the future of the language.

Before we discuss XSLT in more detail and have a first look at how it works, let's study a scenario that clearly demonstrates the variety of formats to which we can transform XML, using XSLT.

A Scenario: Transforming Music

As an indication of how far XML has now penetrated, Robin Cover's index of XML-based application standards at `http://xml.coverpages.org/xmlApplications.html` today runs to over

580 entries. (The last one is entitled *Mind Reading Markup Language*, but as far as I can tell, all the other entries are serious.)

I'll follow just one of these 580 links, *XML and Music*, which takes us to `http://xml.coverpages.org/xmlMusic.html`. On this page we find a list of no less than 17 standards, proposals, or initiatives that use XML for marking up music.

Some of this diversity is unnecessary, and many of these initiatives will bear little fruit. Even the names of the standards are chaotic: there is a Music Markup Language, a MusicML, a MusicXML, and a MusiXML, all of which appear to be quite unrelated. There are at least two really serious contenders: the Music Encoding Initiative (MEI) and the Standard Music Description Language (SMDL). The MEI derives its inspiration from the Text Encoding Initiative, which is widely used by the library community for creating digital text archives, while SMDL is related to the HyTime hypermedia standards and takes into account requirements such as the need to synchronize music with video or with a lighting script.

The diversity of standards is inevitable before the industry can come up with a standard that works for everyone. Without variety, there can be no innovation or experimentation. In fact, the likely outcome is not a single standard, but a collection of three or four different standards that are optimized for different needs. The different notations were invented with different purposes in mind: a markup language used by a publisher for printing sheet music has different requirements from the one designed to let you listen to the music from a browser.

For most of us, music may be fun, a diversion from the world of work. But for others, it is a very serious billion-dollar business. Standards that make information interchange in this business easier have an enormous economic impact. Whether you're interested in the music or the money, we're not dealing here with something that's trivial. So it shouldn't be surprising that so much effort is going into the process of creating standards in this area.

In earlier editions of this book I introduced the idea of using XSLT to transform music as a theoretical possibility, something to make my readers think about the range of possibilities open for the language. Today, it is no longer a theoretical possibility—people are actually doing it.

With 17 different schemas for music in existence, all with different strengths and weaknesses (and fan clubs), there is a big need to convert information from one of these formats to any of the others. There is also a need to convert information from any of these formats to a printable score or an audible performance of the music, as well as a need to create XML representations of music from non-XML sources such as MIDI files (Figure 1-1). XSLT has a role to play in all of these conversions.

So you could use XSLT to:

- ❑ Convert music from one of these representations to another, for example from MEI to SMDL.
- ❑ Convert music from any of these representations into visual music notation, by generating the XML-based vector graphics format SVG.
- ❑ Play the music on a synthesizer, by generating a MIDI (Musical Instrument Digital Interface) file.
- ❑ Perform a musical transformation, such as transposing the music into a different key or extracting parts for different instruments or voices.

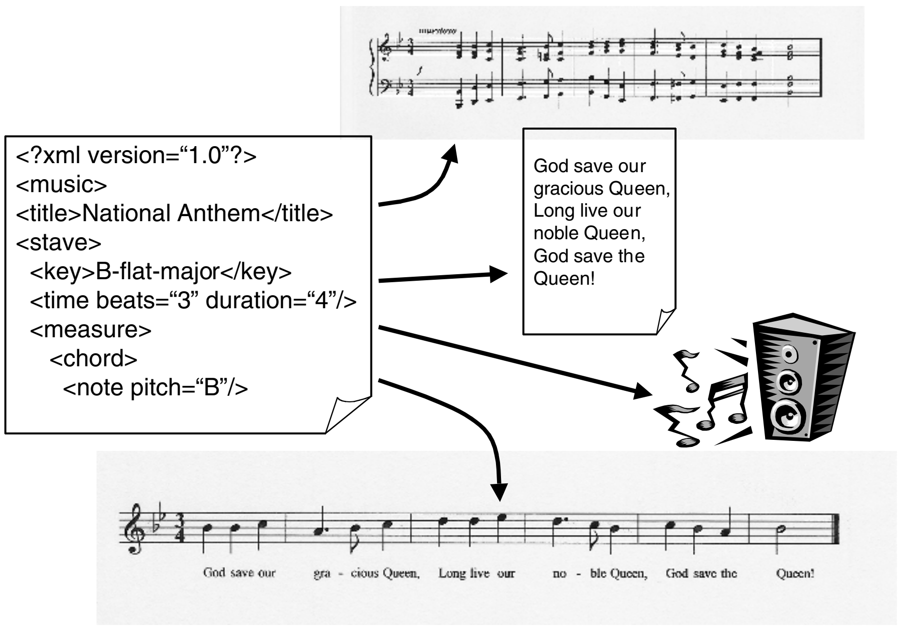

<?xml version="1.0"?>
<music>
<title>National Anthem</title>
<stave>
<key>B-flat-major</key>
<time beats="3" duration="4"/>
<measure>
<chord>
<note pitch="B"/>
God save our gracious Queen, Long live our noble Queen, God save the Queen!
God save our gra - cious Queen, Long live our no - ble Queen, God save the Queen!

Figure 1-1

- ❑ Extract the lyrics, into HTML or into a text-only XML document.
- ❑ Capture music from non-XML formats and translate it to XML (XSLT 2.0 is especially useful here).

As you can see, XSLT is not just for converting XML documents to HTML.

For some real examples of XSLT stylesheets used to transform music, take a look at a thesis written by Baron Schwartz at the University of Virginia (`http://www.cs.virginia.edu/~bps7j/thesis/`).

How Does XSLT Transform XML?

By now you are probably wondering exactly how XSLT goes about processing an XML document in order to convert it into the required output. There are usually two aspects to this process:

1. The first stage is a structural transformation, in which the data is converted from the structure of the incoming XML document to a structure that reflects the desired output.
2. The second stage is formatting, in which the new structure is output in the required format such as HTML or PDF.

The second stage covers the ground we discussed in the previous section; the data structure that results from the first stage can be output as HTML, a text file, or as XML. HTML output allows the information to be viewed directly in a browser by a human user or be input into any modern word processor. Plain text output allows data to be formatted in the way an existing application can accept, for example comma-separated values or one of the many text-based data interchange formats that were developed before XML arrived on the scene. Finally, XML output allows the data to be supplied to one of the new breed of applications that accepts XML directly. Typically, this will use a different vocabulary of XML tags from the original document: for example, an XSLT transformation might take the monthly sales figures as its XML input and produce a histogram as its XML output, using the XML-based SVG standard for vector graphics. Or, you could use an XSLT transformation to generate Voice XML output, for aural rendition of your data.

Information about VoiceXML can be found at `http://www.voicexml.org/`.

Let's now delve into the first stage, transformation—the stage with which XSLT is primarily concerned and which makes it possible to provide output in all of these formats. This stage might involve selecting data, aggregating and grouping it, sorting it, or performing arithmetic conversions such as changing centimeters to inches.

So how does this come about? Before the advent of XSLT, you could only process incoming XML documents by writing a custom application. The application wouldn't actually need to parse the raw XML, but it would need to invoke an XML parser, via a defined Application Programming Interface (API), to get information from the document and do something with it. There are two principal APIs for achieving this: the Simple API for XML (SAX) and the Document Object Model (DOM).

The SAX API is an event-based interface in which the parser notifies the application of each piece of information in the document as it is read. If you use the DOM API, then the parser interrogates the document and builds a tree-like object structure in memory. You would then write a custom application (in a procedural language such as C#, Visual Basic, or Java, for example), which could interrogate this tree structure. It would do so by defining a specific *sequence of steps* to be followed in order to produce the required output. Thus, whatever parser you use, this process has the same principal drawback: every time you want to handle a new kind of XML document, you have to write a new custom program, describing a different sequence of steps, to process the XML.

So, how is using XSLT to perform transformations on XML better than writing custom applications? Well, the design of XSLT is based on a recognition that these programs are all very similar, and it should therefore be possible to describe what they do using a high-level *declarative* language rather than writing each program from scratch in C#, Visual Basic, or Java. The required transformation can be expressed as a set of rules. These rules are based on defining what output should be generated when particular patterns occur in the input. The language is declarative in the sense that you describe the transformation you require, rather than providing a sequence of procedural instructions to achieve it. XSLT describes the required transformation and then relies on the XSLT processor to decide the most efficient way to go about it.

XSLT still relies on an XML parser—it might be a DOM parser or a SAX-compliant one, or one of the new breed of "pull parsers"—to convert the XML document into a tree structure. It is the structure of this tree representation of the document that XSLT manipulates, not the document itself. If you are familiar with the DOM, then you will be happy with the idea of treating every item in an XML document (elements, attributes, processing instructions, and so on) as a node in a tree. With XSLT we have a high-level language that can navigate around a node tree, select specific nodes, and perform complex manipulations on these nodes.

The XSLT tree model is similar in concept to the DOM but it is not the same. The full XSLT processing model is discussed in Chapter 2.

P1: IML
WY020-01 WY020-Kay WY020-Kay-v2.cls July 7, 2004 17:53

The description of XSLT given thus far (a declarative language that can navigate to and select specific data and then manipulate that data) may strike you as being similar to that of the standard database query language, SQL. Let's take a closer look at this comparison.

XSLT and SQL: An Analogy

In a relational database, the data consists of a set of tables. By themselves, the tables are not of much use, the data might as well be stored in flat files in comma-separated values format. The power of a relational database doesn't come from its data structure: it comes from the language that processes the data, SQL. In the same way, XML on its own just defines a data structure. It's a bit richer than the tables of the relational model, but by itself it doesn't actually do anything very useful. It's when we get a high-level language expressly designed to manipulate the data structure that we start to find we've got something interesting on our hands, and for XML data the main language that does that is XSLT.

Superficially, SQL and XSLT are very different languages. But if you look below the surface, they actually have a lot in common. For starters, in order to process specific data, be it in a relational database or an XML document, the processing language must incorporate a declarative query syntax for selecting the data that needs to be processed. In SQL, that's the `SELECT` statement. In XSLT, the equivalent is the *XPath expression*.

The XPath expression language forms an essential part of XSLT, though it is actually defined in a separate W3C Recommendation (`http://www.w3.org/TR/xpath`) because it can also be used independently of XSLT (the relationship between XPath and XSLT is discussed further on page 21). For the same reason, I cover the details of XPath in a companion book, *XPath 2.0 Programmer's Reference*.

The XPath syntax is designed to retrieve nodes from an XML document, based on a path through the XML document or the context in which the node appears. It allows access to specific nodes, while preserving the hierarchy and structure of the document. XSLT is then used to manipulate the results of these queries, for example by rearranging selected nodes and constructing new nodes.

There are further similarities between XSLT and SQL:

- ❑ Both languages augment the basic query facilities with useful additions for performing arithmetic, string manipulation, and comparison operations.
- ❑ Both languages supplement the declarative query syntax with semiprocedural facilities for describing the processing to be carried out, and they also provide hooks to escape into conventional programming languages where the algorithms start to get too complex.
- ❑ Both languages have an important property called *closure*, which means that the output has the same data structure as the input. For SQL, this structure is tables, for XSLT it is trees—the tree representation of XML documents. The closure property is extremely valuable because it means operations performed using the language can be combined end-to-end to define bigger, more complex operations: you just take the output of one operation and make it the input of the next operation. In SQL you can do this by defining views or subqueries; in XSLT you can do it by passing your data through a series of stylesheets, or by capturing the output of one transformation phase as a temporary tree, and using that temporary tree as the input of another transformation phase. This last feature is new in XSLT 2.0, though most XSLT 1.0 processors offered a similar capability as a language extension.

In the real world, of course, XSLT and SQL have to coexist. There are many possible relationships but typically, data is stored in relational databases and transmitted between systems in XML. The two languages don't fit together as comfortably as one would like, because the data models are so different. But XSLT transformations can play an important role in bridging the divide. A number of database vendors have delivered (or at least promised) products that integrate XML and SQL, and some standards are starting to emerge in this area. Check the vendor's Web sites for the latest releases of Microsoft SQL Server, IBM DB2, and Oracle 10*g*.

Before we look at a simple working example of an XSLT transformation, we should briefly discuss a few of the XSLT processors that are available to effect these transformations.

XSLT Processors

The job of an XSLT processor is to apply an XSLT stylesheet to an XML source document and produce a result document.

With XSLT 1.0, there are quite a few good XSLT processors to choose from, and many of them can be downloaded free of charge (but do read the licensing conditions).

If you're working in the Microsoft environment, there is a choice of two products. The most widely used option is MSXML3/4 (Google for "Download MSXML4" to find it). Usually, I'm not Microsoft's greatest fan, but with this processor it's generally agreed that they have done an excellent job. This product comes as standard with Internet Explorer and is therefore the preferred choice for running transformations in the browser. The XSLT processor differs little between MSXML3 and MSXML4: the differences are in what else comes in the package. MSXML3 also includes support for an obsolete but still-encountered dialect of XSLT called WD-xsl (which isn't covered in this book), while MSXML4 includes more extensive support for XML Schema processing. For the .NET environment, however, Microsoft has developed a new processor. This doesn't have a product name of its own, other than its package name within the .NET framework, which is `System.Xml.Xsl`. This processor is often said to be slower than the MSXML3 product, but if you're writing your application using .NET technologies such as C# and ASP.NET, it's probably the one that you'll find more convenient.

In the Java world, there's a choice of open-source products. There's my own Saxon product (version 6.5.3 is the version that supports XSLT 1.0) available from `http://saxon.sf.net/`, there's the Xalan-J product available from Apache at `http://xml.apache.org/`, there is a processor from Oracle, and there is the less well known but highly regarded `jd.xslt` product from Johannes Döbler at `http://www.aztecrider.com/`, which is sadly no longer available for free download. The standard for all these products was set by James Clark's original xt processor, which has been updated by Bill Lindsay and is now available at `http://www.blnz.com/xt/index.html`. A version of Xalan-J is bundled with Sun's Java JDK software from JDK 1.4 onwards.

Other popular XSLT processors include the `libxslt` engine (`http://xmlsoft.org/XSLT/`), Sablotron (`http://www.gingerall.com/charlie/ga/xml/p_sab.xml`), and 4XSLT (part of 4suite, see `http://4suite.org/index.xhtml`).

Most of these products are XSLT interpreters, but there are two well-known XSLT compilers: XSLTC, which is distributed as part of the Xalan-J package mentioned earlier, and Gregor, from Jacek Ambroziak (`http://www.ambrosoft.com/gregor.html`).

The XSLT processor bundled with the Netscape/Mozilla browser is Transformiix (`http://www.mozilla.org/projects/xslt/`).

P1: IML
WY020-01 WY020-Kay WY020-Kay-v2.cls July 7, 2004 17:53

There are a number of development environments that include XSLT processors with custom editing and debugging capabilities. Notable examples are XML Spy (`http://www.altova.com`) and Stylus Studio (`http://www.stylusstudio.com/`).

However, these are all XSLT 1.0 processors, and this book is about XSLT 2.0. With XSLT 2.0, at the time of writing, you really have one choice only, and that is my own Saxon product. By the time you read this, you will be able to get the Saxon XSLT 2.0 processor in two variants, corresponding to the two conformance levels defined in the W3C specification: Standard Saxon 8.x (`http://saxon.sf.net/`) is an open-source implementation of a basic XSLT 2.0 processor, and Saxon-SA 8.x (`http://www.saxonica.com/`) is a commercial implementation of a schema-aware XSLT processor. You can run most of the examples in this book, using a basic XSLT processor, but Chapter 4 and Chapter 11 focus on the additional capability of XSLT when used with XML Schemas, with examples that will only run with a schema-aware product.

There are several other XSLT 2.0 processors under development, though it's not my job to pass on unofficial rumors. Oracle has released a beta implementation (`http://otn.oracle.com/tech/xml/xdk/xdkbeta.html`). Apache has said that they are developing an XSLT 2.0 version of Xalan, and although things have been very quiet in public since they announced this, if you poke around the archives of their internal developers' mailing list (which are all openly available) you can see that work is proceeding, slowly but steadily.

The biggest uncertainty at the moment is what Microsoft will do. The MSXML3/4 products have been highly successful technically, though whether Microsoft considers them a commercial success is anyone's guess. The .NET processor also appears to be a competent piece of engineering. However, the unofficial rumor from Reston is that there is no XSLT 2.0 version of this technology under development. Instead, most of Microsoft's efforts seem to be going into the development of their XQuery engine, designed as part of the SQL Server release code-named Yukon. I have heard speculation that if and when Microsoft implements XSLT 2.0, it is likely to be done using this engine. XSLT 2.0 and XQuery 1.0 are so close in their semantics that it's entirely possible to implement both languages, using the same processing engine, as Saxon has already demonstrated.

It shouldn't really be a surprise that commercial vendors are keeping fairly quiet about their plans while the specifications are still working drafts. When a specification first comes out at version 1.0, there is a great deal of commercial advantage to be gained by releasing beta implementations as early as possible. With a 2.0 draft, it's more prudent to wait until you know exactly what's in the final specification. W3C specifications these days spend a long time in their "Candidate Recommendation" phase, and this is the time to look for evidence of implementation activity. Although XSLT 2.0 is close to reaching that phase, it isn't there yet at the time of writing.

I think it's a little unlikely that there will be quite as many XSLT 2.0 processors as there are for XSLT 1.0 (there is bound to be some shakeout in a maturing market), but I'm confident there will be four or five, which should be enough.

Meanwhile, there is Saxon, and that's what I will be using for all the examples in this book.

An XSLT 1.0 Stylesheet

We're now ready to take a look at an example of using XSLT to transform a very simple XML document.

Example: A "Hello, world!" XSLT Stylesheet

Kernighan and Ritchie in their classic *The C Programming Language* (Prentice-Hall, 1988) originated the idea of presenting a trivial but complete program right at the beginning of the book, and ever since then the `Hello world` program has been an honored tradition. Of course, a complete description of how this example works is not possible until all the concepts have been defined, and so if you feel I'm not explaining it fully, don't worry—the explanations will come later.

Input

What kind of transformation would we like to do? Let's try transforming the following XML document.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<?xml-stylesheet type="text/xsl" href="hello.xsl"?>
<greeting>Hello, world!</greeting>
```

This document is available as file `hello.xml` in the download directory for this chapter.

A simple node-tree representation of this document is shown in Figure 1-2.

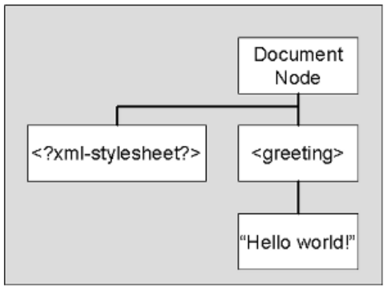


Figure 1-2

There are four nodes in this tree: a document node that represents the document as a whole; an `<?xml-stylesheet?>` processing instruction that identifies the stylesheet to be used; the `<greeting>` element; and the text within the `<greeting>` element.

The document node in the XSLT model performs the same function as the document node in the DOM model (it was called the root node in XSLT 1.0, but the nomenclature has been brought into line with the DOM). The XML declaration is not visible to the XSLT processor and, therefore, is not included in the tree.

I've deliberately made it easy by including an `<?xml-stylesheet?>` processing instruction in the source XML file. Many XSLT processors will use this to identify the stylesheet if you don't specify a different stylesheet to use. The `href` attribute gives the relative URI of the default stylesheet for this document.

Output

Our required output is the following HTML, which will simply change the browser title to `"Today's Greeting"` and display whatever greeting is in the source XML file:

```
<html>
<head>
   <title>Today's greeting</title>
</head>
<body>
   <p>Hello, world!</p>
</body>
</html>
```

XSLT Stylesheet

Without any more ado, here's the XSLT stylesheet `hello.xsl` to effect the transformation.

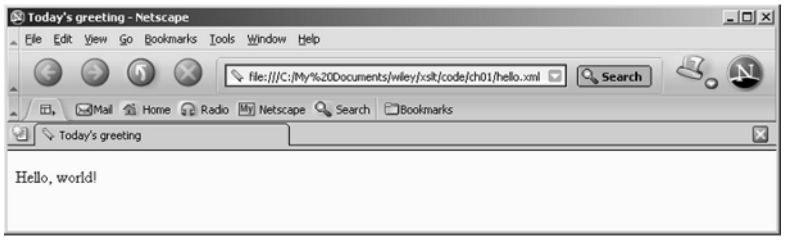
```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet
   version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <html>
   <head>
      <title>Today's greeting</title>
   </head>
   <body>
      <p><xsl:value-of select="greeting"/></p>
   </body>
   </html>
</xsl:template>

</xsl:stylesheet>
```

Running the Stylesheet

You can run this stylesheet in a number of different ways. The easiest is simply to load the XML file `hello.xml` into any recent version of Internet Explorer or Netscape. The browser will recognize the `<?xml-stylesheet?>` processing instruction and will use this to fetch the stylesheet and execute it. The result is a display like the one in Figure 1-3.


Today's greeting - Netscape
File Edit View Go Bookmarks Tools Window Help
file:///C:/My%20Documents/wiley/xslt/code/ch01/hello.xml
Search
Mail Home Radio My Netscape Search Bookmarks
Today's greeting

Hello, world!

Figure 1-3

Note that this example is an XSLT 1.0 stylesheet. Current versions of Internet Explorer and Netscape don't yet support XSLT 2.0.

Saxon

Running the stylesheet with Saxon is a bit more complicated, because you first need to install the software. It's worth going through these steps, because you will need Saxon to run later examples in this book. The steps are as follows:

1. Ensure you have the Java SDK version 1.4 or later installed on your machine. You can get this from `http://java.sun.com/`. Saxon is pure Java code, and so it will run on any platform that supports Java, but I will usually assume that you are using a Windows machine.
2. Download the Saxon processor from `http://saxon.sf.net/`. Any version of Saxon will run this example, but we'll soon be introducing XSLT 2.0 examples, so install a recent version, say Saxon 7.9.
3. Unzip the download file into a suitable directory, for example `c:\saxon`..
4. Within this directory, create a subdirectory called `data`.
5. Using Notepad, type the two files mentioned earlier into `hello.xml` and `hello.xsl` respectively, within this directory (or get them from the Wrox Web site at `http://www.wrox.com`).
6. Bring up an MSDOS-style console window (using Start | Programs | MSDOS Prompt on Windows 98 or NT, or look in the Accessories menu under Windows 2000 or Windows XT).
7. Type the following at the command prompt.

   ```
   java -jar c:\saxon\saxon8.jar -a hello.xml
   ```

8. Admire the HTML displayed on the standard output.

If you want to view the output using your browser, simply save the command line output as an HTML file, in the following manner.

```
java -jar c:\saxon\saxon8.jar -a hello.xml >hello.html
```

(Using the command prompt in Windows isn't much fun. I would recommend acquiring a good text editor: most editors have the ability invoke a command line processor that is usually much more usable than the one provided by the operating system. I have recently started using `jEdit`, which is free and can be downloaded from `http://www.jedit.org/`. Be sure to install the optional Console plug-in.)

How It Works

If you've succeeded in running this example, or even if you just want to get on with reading the book, you'll want to know how it works. Let's dissect it.

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

This is just the standard XML heading. The interesting point is that an XSLT stylesheet is itself an XML document. I'll have more to say about this, later in the chapter. I've used `iso-8859-1` character encoding (which is the official name for the character set that Microsoft sometimes calls "ANSI") because in Western Europe and North America it's the character set that most text editors support. If you've got a text editor that supports `UTF-8` or some other character encoding, feel free to use that instead.

```
<xsl:stylesheet
   version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

This is the standard XSLT heading. In XML terms it's an element start tag, and it identifies the document as a stylesheet. The `xmlns:xsl` attribute is an XML Namespace declaration, which indicates that the prefix `xsl` is going to be used for elements defined in the W3C XSLT specification. XSLT makes extensive use of XML namespaces, and all the element names defined in the standard are prefixed with this namespace to avoid any clash with names used in your source document. The `version` attribute indicates that the stylesheet is only using features from version 1.0 of the XSLT standard.

Let's move on.

```
<xsl:template match="/">
```

An `<xsl:template>` element defines a template rule to be triggered when a particular part of the source document is being processed. The attribute «`match="/"`» indicates that this particular rule is triggered right at the start of processing the source document. Here «`/`» is an XPath expression that identifies the *document node* of the document: an XML document has a hierarchic structure, and in the same way as UNIX uses the special filename «`/`» to indicate the root of a hierarchic file store, XPath uses «`/`» to represent the root of the XML content hierarchy.

```
<html>
<head>
   <title>Today's greeting</title>
</head>
<body>
   <p><xsl:value-of select="greeting"/></p>
</body>
</html>
```

Once this rule is triggered, the body of the template says what output to generate. Most of the template body here is a sequence of HTML elements and text to be copied into the output file. There's one exception: an `<xsl:value-of>` element, which we recognize as an XSLT instruction, because it uses the namespace prefix `xsl`. This particular instruction copies the textual content of a node in the source document to the output document. The `select` attribute of the element specifies the node for which the value should be evaluated. The XPath expression «`greeting`» means "find the set of all `<greeting>` elements that are children of the node that this template rule is currently processing." In this case, this means the `<greeting>` element that's the outermost element of the source document. The `<xsl:value-of>` instruction then extracts the text of this element and copies it to the output at the relevant place—in other words, within the generated `<p>` element.

All that remains is to finish what we started.

```
</xsl:template>

</xsl:stylesheet>
```

In fact, for a simple stylesheet like the one shown earlier, you can cut out some of the red tape. Since there is only one template rule, the `<xsl:template>` element can actually be omitted. The following is a complete, valid stylesheet equivalent to the preceding one.

```
<html xsl:version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<head>
   <title>Today's greeting</title>
</head>
<body>
   <p><xsl:value-of select="greeting"/></p>
</body>
</html>
```

This simplified syntax is designed to make XSLT look familiar to people who have learned to use proprietary template languages that allow you to write a skeleton HTML page with special tags (analogous to `<xsl:value-of>`) to insert variable data at the appropriate place. But as we'll see, XSLT is much more powerful than that.

Why would you want to place today's greeting in a separate XML file and display it using a stylesheet? One reason is that you might want to show the greeting in different ways, depending on the context; for example, it might be shown differently on a different device, or the greeting might depend on the time of day. In this case, you could write a different stylesheet to transform the same source document in a different way. This raises the question of how a stylesheet gets selected at run-time. There is no single answer to this question: it depends on the product you are using.

With Saxon, we used the `-a` option to process the XML document using the stylesheet specified in its `<?xml-stylesheet?>` processing instruction. Instead, we could simply have specified the stylesheet on the command line:

```
java -jar c:\saxon\saxon8.jar hello.xml hello.xsl >hello.html
```

The same thing can also be achieved with the Microsoft XSLT product. Like Saxon, this has a command line interface; though, if you want to control it programmatically in the browser you will need to write an HTML page containing some script code to control the transformation. The `<?xml-stylesheet?>` processing instruction which I used in the example described earlier works only if you want to use the same stylesheet every time.

Having looked at a very simple XSLT 1.0 stylesheet, let's now look at a stylesheet that uses features that are new in XSLT 2.0.

An XSLT 2.0 Stylesheet

This stylesheet is very short, but it manages to use four or five new XSLT 2.0 and XPath 2.0 features within the space of a few lines. I wrote it in response to a user enquiry raised on the xsl-list at `http://www.mulberrytech.com/` (an excellent place for meeting other XSLT developers with widely varying levels of experience); so it's a real problem, not an invention. The XSLT 1.0 solution to this problem is about 60 lines of code.

Example: Tabulating Word Frequencies

The problem is simply stated: given any XML document, produce a list of the words that appear in its text, giving the number of times each word appears, together with its frequency.

Input

The input can be any XML document. I will use the text of Shakespeare's *Othello* as an example; this is provided as `othello.xml` in the download files for this book.

Output

The required output is an XML file that lists words in decreasing order of frequency. For *Othello*, the output file starts like this.

```
<?xml version="1.0" encoding="UTF-8"?>
<wordcount>
   <word word="i" frequency="899"/>
   <word word="and" frequency="796"/>
   <word word="the" frequency="764"/>
   <word word="to" frequency="632"/>
   <word word="you" frequency="494"/>
   <word word="of" frequency="476"/>
   <word word="a" frequency="453"/>
   <word word="my" frequency="427"/>
   <word word="that" frequency="396"/>
   <word word="iago" frequency="361"/>
   <word word="in" frequency="343"/>
   <word word="othello" frequency="336"/>
   <word word="it" frequency="319"/>
   <word word="not" frequency="319"/>
   <word word="is" frequency="309"/>
   <word word="me" frequency="281"/>
   <word word="cassio" frequency="254"/>
```

Stylesheet

Here is the stylesheet that produces this output. You can find it in `wordcount.xsl`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet
    version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
<xsl:output method="xml" indent="yes"/>

<xsl:template match="/">
  <wordcount>
    <xsl:for-each-group group-by="." select="
          for $w in tokenize(string(.), '\W+') return lower-case($w)">
      <xsl:sort select="count(current-group())" order="descending"/>
      <word word="{current-grouping-key()}"
            frequency="{count(current-group())}"/>
    </xsl:for-each-group>
  </wordcount>
</xsl:template>

</xsl:stylesheet>
```

Let's see how this works.

The `<xsl:stylesheet>` element introduces the XSLT namespace, as before, and tells us that this stylesheet is designed to be used with an XSLT 2.0 processor.

The `<xsl:output>` element asks for the XML output of the stylesheet to be indented, which makes it much easier for humans to read.

There is one `<xsl:template>` element, as before, which defines the code to be executed when the document node of the source document is encountered. This generates a `<wordcount>` element in the result, and within this it puts the word frequencies.

To understand the `<xsl:for-each-group>` instruction, which is new in XSLT 2.0, we first need to look at its `select` attribute. This contains the XPath 2.0 expression

```
for $w in tokenize(string(.), '\W+') return lower-case($w)
```

This first calculates `string(.)`, the string-value of the node currently being processed, which in this case contains the whole text of the input document, ignoring all markup. It then tokenizes this big string: that is, it splits it into a sequence of substrings. The tokenizing is done by applying the regular expression «`\W+`». Regular expressions are new in XPath 2.0 and XSLT 2.0, though they will be very familiar to users of other languages such as Perl. They provide the language with greatly enhanced text handling capability. This particular expression, «`\W+`», matches any sequence of one-or-more "non-word" characters, a convenient category that includes spaces, punctuation marks, and other separators. So the result of calling the `tokenize()` function is a sequence of strings containing the words that appear in the text.

The XPath «`for`» expression now applies the function `lower-case()` to each of the strings in this sequence, producing the lower-case equivalent of the word. (Almost everything in this XPath expression is new in XPath 2.0: the `lower-case()` function, the `tokenize()` function, the «`for`» expression, and indeed the ability to manipulate a sequence of strings.)

The XSLT stylesheet now takes this sequence of strings and applies the `<xsl:for-each-group>` instruction to it. This processes the body of the `<xsl:for-each-group>`

instruction once for each group of selected items, where a group is identified as those items that have a common value for a grouping key. In this case the grouping key is written as «`group-by="."`», which means that the values (the words) are grouped on their own value. (In another application, we might have chosen to group them by their length, or by their initial letter.) So, the body of the instruction is executed once for each distinct word, and the `<xsl:sort>` instruction tells us to sort the groups in descending order of the size of the groups (that is, the number of times each word appears). For each of the groups, we output a `<word>` element with two attributes: one attribute is the value we used as the grouping key, the other is the number of items in the group.

Don't worry if this example seemed a bit bewildering: it uses many concepts that haven't been explained yet. The purpose was to give you a feeling for some of the new features in XSLT 2.0 and XPath 2.0, which will all be explained in much greater detail elsewhere in this book (in the case of XSLT features) or in the companion book *XPath 2.0 Programmer's Reference*.

Having dipped our toes briefly into some XSLT code, I'd now like to step back to take a higher-level view, to discuss where XSLT as a technology fits into the big picture of designing applications for the Web.

Where to Use XSLT

This section identifies what tasks XSLT is good at, and by implication, tasks for which a different tool would be more suitable. I also look at alternative ways of using XSLT within the overall architecture of your application.

As I discussed at the beginning of the chapter, there are two main scenarios for using XSLT transformations: data conversion and publishing. We'll consider each of them separately.

Data Conversion Applications

Data conversion is not something that will go away just because XML has been invented. Even though an increasing number of data transfers between organizations or between applications within an organization are likely to be encoded in XML, there will still be different data models, different ways of representing the same thing, and different subsets of information that are of interest to different people (recall the example at the beginning of the chapter, where we were converting music between different XML representations and different presentation formats). So, however enthusiastic we are about XML, the reality is that there are going to be a lot of comma-separated-values files, EDI messages, and any number of other formats in use for a long time to come.

When you have the task of converting one XML data set into another, then XSLT is an obvious choice (Figure 1-4).

It can be used for extracting the data selectively, reordering it, turning attributes into elements or vice versa, or any number of similar tasks. It can also be used simply for validating the data. As a language, XSLT 1.0 was best at manipulating the structure of the information as distinct from its content: it was a good language for turning rows into columns, but for string handling (for example, removing any text that appears between square brackets) it was rather laborious compared with a language like JavaScript

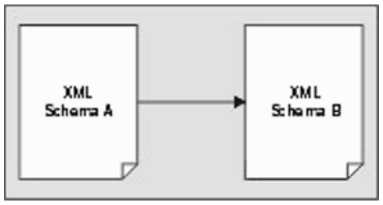


Figure 1-4

or Perl that offered support for regular expressions. This has changed considerably in version 2.0, and now there are few XML transformation tasks that I wouldn't tackle using XSLT.

XSLT is also useful for converting XML data into any text-based format, such as comma-separated values, or various EDI message formats (Figure 1-5). Text output is really just like XML output without the tags, so this creates no particular problems for the language.

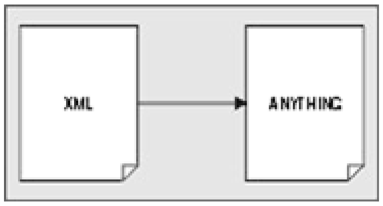


Figure 1-5

Perhaps more surprising is that XSLT can often be useful to convert from non-XML formats into XML or something else (Figure 1-6). In this case you'll need to write some kind of parser that understands the input format; but you would have had to do that anyway. The benefit is that once you've written the parser, the rest of the data conversion can be expressed in a high-level language. This separation also increases the chances that you'll be able to reuse your parser next time you need to handle that particular input format. I'll show you an example in Chapter 11, page 703, where the input is a rather old-fashioned and distinctly non-XML format widely used for exchanging data between genealogy software packages. It turns out that it isn't even necessary to write the data out as XML before using the XSLT stylesheet to process it: all you need to do is to make your parser look like an XML parser, by making it implement one of the standard parser interfaces: SAX or DOM. Most XSLT processors will accept input from a program that implements the SAX or DOM interfaces, even if the data never saw the light of day as XML.

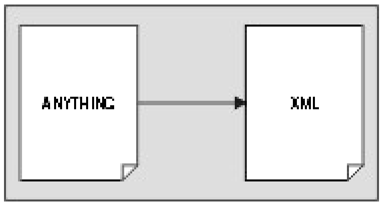


Figure 1-6

One caveat about data conversion applications: today's XSLT processors all rely on holding all the data in memory while the transformation is taking place. The tree structure in memory can be anything up to ten times the original data size, and so if you have 512MB of memory, I wouldn't advise tackling a

P1: IML
WY020-01 WY020-Kay WY020-Kay-v2.cls July 7, 2004 17:53

transformation larger than 50MB, unless you do some performance tests first. Even at this size, a complex conversion can be quite time-consuming; it depends very much on the processing that you actually want to do.

One way around this is to split the data into chunks and convert each chunk separately—assuming, of course, that there is some kind of correspondence between chunks of input and chunks of output. But when this starts to get complicated, there comes a point where XSLT is no longer the best tool for the job. You would probably be better off-loading the data into an XML database such as Tamino or Xindice, and using the database query language to extract it again in a different sequence.

If you need to process large amounts of data serially, for example extracting selected records from a log of retail transactions, then an application written using the SAX interface might take a little longer to write than the equivalent XSLT stylesheet, but it is likely to run many times faster. Very often the combination of a SAX filter application to do simple data extraction, followed by an XSLT stylesheet to do more complex manipulation, can be the best solution in such cases.

Publishing

The difference between data conversion and publishing is that in the former case, the data is destined for input to another piece of software, while in the latter case it is destined to be read (you hope) by human beings. Publishing in this context doesn't just mean lavish text and multimedia, it also means data: everything from the traditional activity of producing and distributing reports so that managers know what's going on in the business, to producing online phone bills and bank statements for customers, and rail timetables for the general public. XML is ideal for such data publishing applications, as well as the more traditional text publishing, which was the original home territory of SGML.

XML was designed to enable information to be held independently of the way it is presented, which sometimes leads people into the fallacy of thinking that using XML for presentation details is somehow bad. Far from it, if you were designing a new format for downloading fonts to a printer today, you would probably make it XML-based. Presentation details have just as much right to be encoded in XML as any other kind of information. So, we can see the role of XSLT in the publishing process as being converting data-without-presentation to data-with-presentation, where both are, at least in principle, XML formats.

The two important vehicles for publishing information today are print-on-paper and the Web. The print-on-paper scene is the more difficult one, because of the high expectations of users for visual quality. XSL Formatting Objects attempts to define an XML-based model of a print file for high-quality display on paper or on screen. Because of the sheer number of parameters needed to achieve this, the standard has taken a while to come to maturity. But the Web is a less demanding environment, where all we need to do is convert the data to HTML and leave the browser to do the best it can on the display available. HTML, of course, is not XML, but it is close enough so that a simple mapping is possible. Converting XML to HTML is the most common application for XSLT today. It's actually a two-stage process: first convert to an XML-based model that is structurally equivalent to the target HTML, and then serialize this in HTML notation rather than strict XML.

The emergence of XHTML 1.0, of course, tidies up this process even further, because it is a pure XML format. But the emergence of XSLT has arguably reduced the need for XHTML, because once HTML becomes merely a transient protocol used to get information from the XSLT engine to the Web browser, its idiosyncrasies cease to matter so much.

When to Do the Conversion?

The process of publishing information to a user is illustrated in Figure 1-7.

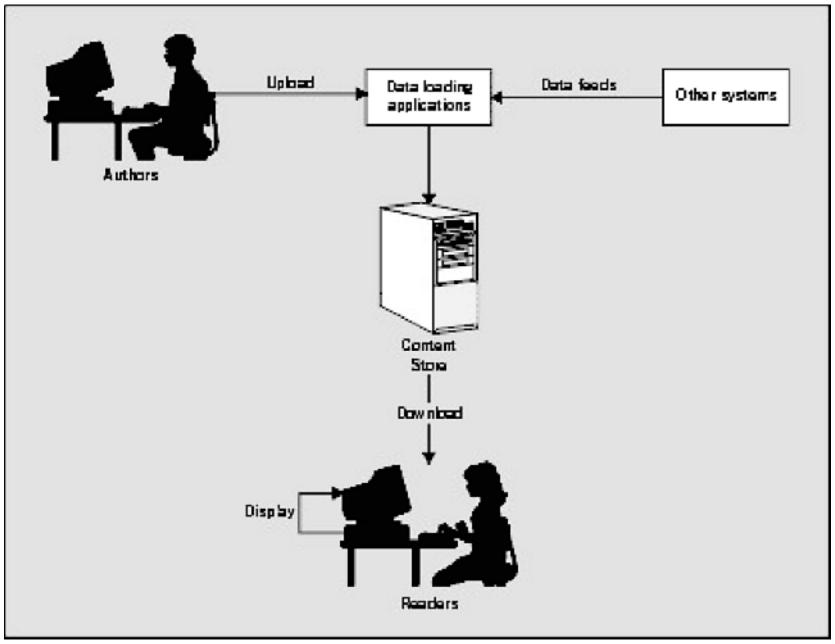


Figure 1-7

There are several points in such a system where XSLT transformations might be appropriate:

- Information entered by authors using their preferred tools, or customized form-filling interfaces, can be converted to XML and stored in that form in the content store.
- XML information arriving from other systems might be transformed into a different flavor of XML for storage in the content store. For example, it might be broken up into page-size chunks.
- XML can be translated into HTML on the server, when the users request a page. This can be controlled using technology such as Java servlets or Java Server Pages. On a Microsoft server you can invoke the transformation from script on ASP.NET pages.
- XML can be sent down to the client system and translated into HTML within the browser. This can give a highly interactive presentation of the information and remove a lot of the processing load from the server, but it relies on all the users having a browser that can do the job.
- XML data can also be converted into its final display form at publishing time and stored as HTML within the content store. This minimizes the work that needs to be done at display time and is ideal when the same displayed page is presented to very many users.

There isn't one right answer, and often a combination of techniques may be appropriate. Conversion in the browser is an attractive option when XSLT is widely available within browsers, but we still don't have universal availability of XSLT 1.0 in all browsers, let alone 2.0. Even when client-side conversion is done, there may still be a need for some server-side processing to deliver the XML in manageable chunks and to

P1: IML
WY020-01 WY020-Kay WY020-Kay-v2.cls July 7, 2004 17:53

protect secure information. Conversion at delivery time on the server is a popular choice, because it allows personalization, but it can be a heavy overhead for sites with high traffic. Some busy sites have found that it is more effective to generate a different set of HTML pages for each section of the target audience in advance, and at page request time to do nothing more than select the right preconstructed HTML page.

It's time now to take a closer look at the relationship between XSLT and XPath and other XML-related technologies.

The Place of XSLT in the XML Family

XSLT is published by the World Wide Web Consortium (W3C) and fits into the XML family of standards, most of which are also developed by W3C. In this section I will try to explain the sometimes-confusing relationship of XSLT to other related standards and specifications.

XSLT and XSL

XSLT started life as part of a bigger language called *XSL* (*Extensible Stylesheet Language*). As the name implies, XSL was (and is) intended to define the formatting and presentation of XML documents for display on screen, on paper, or in the spoken word. As the development of XSL proceeded, it became clear that this was usually a two-stage process: first a structural transformation, in which elements are selected, grouped and reordered; and then a formatting process in which the resulting elements are rendered as ink on paper, or pixels on the screen. It was recognized that these two stages were quite independent, so XSL was split into two parts: XSLT for defining transformations; and "the rest"—which is still officially called XSL, though most people prefer to call it *XSL-FO* (*XSL Formatting Objects*)—for the formatting stage.

XSL-FO is nothing more than another XML vocabulary, in which the objects described are areas of the printed page and their properties. Since this is just another XML vocabulary, XSLT needs no special capabilities to generate this as its output. XSL-FO is outside the scope of this book. It's a big subject (the specification is longer than XSLT). What's more, you're probably less likely to need it than to need XSLT. XSL-FO provides wonderful facilities to achieve high-quality typographical output of your documents. However, for many people translating documents into HTML for presentation by a standard browser is quite good enough, and that can be achieved using XSLT alone, or if necessary, by using XSLT in conjunction with Cascading Stylesheets (CSS or CSS2), which I shall return to shortly.

The XSL-FO specification became a Recommendation on 15 October 2001. It can be found at `http://www.w3.org/TR/xsl`.

XSLT and XPath

Halfway through the development of XSLT 1.0, it was recognized that there was a significant overlap between the expression syntax in XSLT for selecting parts of a document and the XPointer language being developed for linking from one document to another. To avoid having two separate but overlapping expression languages, the two committees decided to join forces and define a single language, *XPath*, which would serve both purposes. XPath 1.0 was published on the same day as XSLT 1.0, 16 November 1999.

XPath acts as a sublanguage within an XSLT stylesheet. An XPath expression may be used for numerical calculations or string manipulations, or for testing Boolean conditions, but its most characteristic use (and the one that gives it its name) is to identify parts of the input document to be processed. For example, the following instruction outputs the average price of all the books in the input document:

```
    <xsl:value-of select="avg(//book/@price)"/>
```

Here, the `<xsl:value-of>` element is an instruction defined in the XSLT standard, which causes a value to be written to the output document. The `select` attribute contains an XPath expression, which calculates the value to be written: specifically, the average value of the `price` attributes on all the `<book>` elements. (The `avg()` function too is new in XPath 2.0.)

Following its publication, the XPath specification increasingly took on a life of its own, separate from XSLT. Several DOM implementations (including Microsoft's) allowed you to select nodes within a DOM tree structure, using a method such as `selectNodes(XPath)`, and this feature is now included in the current version of the standard, DOM3. A subset of XPath is used within the XML Schema language, and bindings of XPath to other languages such as Perl are multiplying. The language has also proved interesting to academics, and a number of papers have been published analyzing its semantics, which provides the basis for optimized implementations.

The separation of XPath from XSLT works reasonably well, but as the earlier example shows, you need to understand the interaction between the two languages to see how a stylesheet works. In previous editions of this book I covered both languages together, but this time I have given each language its own volume, mainly because the amount of material had become too large for one book, and also because there are an increasing number of people who use XPath without also using XSLT. For the XSLT user, though, I'm afraid that at times you may have to keep both books open on your desk at once.

XSLT and XML

XSLT is essentially a tool for transforming XML documents. At the start of this chapter we discussed the reasons why this is important, but now we need to look a little more precisely at the relationship between the two. There are two particular aspects of XML that XSLT interacts with very closely: one is XML Namespaces; the other is the XML Information Set. These are discussed in the following sections.

XML Namespaces

XSLT is designed on the basis that *XML namespaces* are an essential part of the XML standard. So when the XSLT standard refers to an XML document, it really means an XML document that also conforms to the XML Namespaces specification, which can be found at `http://www.w3.org/TR/REC-xml-names`.

For a full explanation of XML Namespaces, see Chapter 11 of XML 1.1 Bible, Third Edition (Wiley, 2004).

Namespaces play an important role in XSLT. Their purpose is to allow you to mix tags from two different vocabularies in the same XML document. For example, in one vocabulary `<table>` might mean a two-dimensional array of data values, while in another vocabulary `<table>` refers to a piece of furniture. Here's a quick reminder of how they work:

❑ Namespaces are identified by a Uniform Resource Identifier (URI). This can take a number of forms. One form is the familiar URL, for example `http://www.wrox.com/namespace`. Another form, not fully standardized but being used in some XML vocabularies, is a URN, for example `urn:biztalk-org:biztalk:biztalk_1"`. The detailed form of the URI doesn't matter, but it is a good idea to choose one that will be unique. One good way of achieving this is to use the URL of your own Web site. But don't let this confuse you into thinking that there must be something on the Web site for the URL to point to. The namespace URI is simply a string that you have chosen to be different from other people's namespace URIs; it doesn't need to point to anything.

- ❑ The latest version, XML Namespaces 1.1, allows you to use an International Resource Identifier (IRI) rather than a URI. The main difference is that this permits characters from any alphabet; it is no longer confined to ASCII. In practice, most XML parsers have always allowed you to use any characters you like in a namespace URI.
- ❑ Since namespace URIs are often rather long and use special characters such as «`/`», they are not used in full as part of the element and attribute names. Instead, each namespace used in a document can be given a short nickname, and this nickname is used as a prefix of the element and attribute names. It doesn't matter what prefix you choose, because the real name of the element or attribute is determined only by its namespace URI and its local name (the part of the name after the prefix). For example, all my examples use the prefix `xsl` to refer to the namespace URI `http://www.w3.org/1999/XSL/Transform`, but you could equally well use the prefix `xslt`, so long as you use it consistently.
- ❑ For element names, you can also declare a default namespace URI, which is to be associated with unprefixed element names. The default namespace URI, however, does not apply to unprefixed attribute names.

A namespace prefix is declared using a special pseudo-attribute within any element start tag, with the form:

```
xmlns:prefix = "namespace-URI"
```

This declares a namespace prefix, which can be used for the name of that element, for its attributes, and for any element or attribute name contained in that element. The default namespace, which is used for elements having no prefix (but not for attributes), is similarly declared using a pseudo-attribute:

```
xmlns = "namespace-URI"
```

XSLT can't be used to process an XML document unless it conforms to the XML Namespaces Recommendation. In practice this isn't a problem, because most people are treating XML Namespaces as an intrinsic part of the XML standard, rather than a bolt-on optional extra. It does have certain implications, though. In particular, serious use of Namespaces is difficult to combine with serious use of Document Type Definitions (DTDs), because DTDs don't recognize the special significance of prefixes in element names; so, a consequence of backing Namespaces is that XSLT provides very little support for DTDs, having chosen instead to wait for the replacement facility, XML Schemas.

XML Namespaces 1.1 became a Recommendation on 4 February 2004, and the XSLT 2.0 specification makes provision for XSLT processors to work with this version, though it isn't required. Apart from the largely cosmetic change from URIs to IRIs mentioned earlier, the main innovation is the ability to undeclare a namespace, using a namespace undeclaration of the form «`xmlns:prefix=""`». This is particularly intended for applications like SOAP messaging, where an XML payload document is wrapped in an XML envelope for transmission. Without namespace undeclarations, there is a tendency for namespaces used in the SOAP envelope to stick to the payload XML when this is removed from the envelope, which can cause validation failures and other problems. For example, it can invalidate a digital signature attached to the document.

The XML Information Set

XSLT is designed to work on the information carried by an XML document, not on the raw document itself. This means that, as an XSLT programmer, you are given a tree view of the source document. This tree view is an abstraction of the original lexical XML, in which information that's deemed significant is retained, and other information is discarded. For example, you can see the attribute names and values, but you can't see whether the attribute was written in single or double quotes, you can't see what order the attributes were in, and you can't tell whether or not they were written on the same line.

One messy detail is that there have been many attempts to define exactly what constitutes the *essential* information content of a well-formed XML document, as distinct from its accidental punctuation. All attempts so far have come up with slightly different answers. The most definitive attempt to provide a common vocabulary for the content of XML documents is the *XML Information Set* definition (usually called the InfoSet), which may be found at `http://www.w3.org/TR/xml-infoset`.

Unfortunately, the InfoSet came too late to make all the standards consistent. For example, some treat comments as significant, others not; some treat the choice of namespace prefixes as significant, others take them as irrelevant. In Chapter 2, I shall describe exactly how XSLT (or more accurately, XPath) defines the tree model of XML, and how it differs in finer points of detail from some of the other definitions such as the Document Object Model or DOM.

One new piece of jargon is the concept of the *post schema validation infoset* or PSVI. This contains the significant information from the source document, augmented with information taken from its XML schema. It therefore allows you to find out not only that the value of an attribute was «`17.3`», but also that the attribute was described in the schema as a non-negative decimal number. A major change in XSLT 2.0 is that if you choose to use a schema processor to validate your documents, your XSLT stylesheets now make use of this additional information in the PSVI.

XSL and CSS

Why are there two stylesheet languages, XSL (that is, XSLT plus XSL Formatting Objects) as well as Cascading Style Sheets (CSS and CSS2)?

It's only fair to say that in an ideal world there would be a single language in this role, and that the reason there are two is that no one was able to invent something that achieved the simplicity and economy of CSS for doing simple things, combined with the power of XSL for doing more complex things.

CSS (by which I include CSS2, which greatly extends the degree to which you can control the final appearance of the page) is mainly used for rendering HTML, but it can also be used for rendering XML directly, by defining the display characteristics of each XML element. However, it has serious limitations. It cannot reorder the elements in the source document, it cannot add text or images, it cannot decide which elements should be displayed and which omitted, neither can it calculate totals or averages or sequence numbers. In other words, it can only be used when the structure of the source document is already very close to the final display form.

Having said this, CSS is simple to write, and it is very economical in machine resources. It doesn't reorder the document, and so it doesn't need to build a tree representation of the document in memory, and it can

P1: IML
WY020-01 WY020-Kay WY020-Kay-v2.cls July 7, 2004 17:53

start displaying the document as soon as the first text is received over the network. Perhaps, most important of all, CSS is very simple for HTML authors to write, without any programming skills. In comparison, XSLT is far more powerful, but it also consumes a lot more memory and processor power, as well as training budget.

It's often appropriate to use both tools together. Use XSLT to create a representation of the document that is close to its final form, in that it contains the right text in the right order, and then use CSS to add the finishing touches, by selecting font sizes, colors, and so on. Typically, you would do the XSLT processing on the server and the CSS processing on the client (in the browser); so, another advantage of this approach is that you reduce the amount of data sent down the line, which should improve response time for your users and postpone the next expensive bandwidth increase.

XSLT and XML Schemas

One of the biggest changes in XSLT 2.0, and one of the most controversial, is the integration of XSLT with the XML Schema language. XML Schema provides a replacement for DTDs as a way of specifying the structural constraints that apply to a class of documents; unlike DTDs, an XML Schema can regulate the content of the text as well as the nesting of the elements and attributes. Many of the industry vocabularies being used to define XML interchange standards are specified using XML Schema definitions. For example, several of the XML vocabularies for describing music, which I alluded to earlier in the chapter, have an XML Schema to define their rules, and this schema can be used to check the conformance of individual documents to the standard in question.

When you write a stylesheet, you need to make assumptions about the structure of the input documents it is designed to process and the structure of the result documents it is designed to produce. With XSLT 1.0, these assumptions were implicit; there was no formal way of stating the assumptions in the stylesheet itself. As a result, if you try applying a stylesheet to the wrong kind of input document, the result will generally be garbage.

The idea of linking XSLT and XML Schema was driven by two main considerations:

- ❑ There should, in principle, be software engineering benefits if a program (and a stylesheet is indeed a program) makes explicit assertions about its expected inputs and outputs. These assertions can lead to better and faster detection of errors, often enabling errors to be reported at compile time that otherwise would only be reported the first time the stylesheet was applied to some test data that happened to exercise a particular part of the code.
- ❑ The more information that's available to an XSLT processor at compile time, the more potential it has to generate optimal code, giving faster execution and better use of memory.

So why the controversy? It's mainly because XML Schema itself is less than universally popular. It's an extremely complex specification that's very hard to read, and when you discover what it says, it appears to be full of rules that seem artificial and inconsistent. It manages at the same time to be specified in very formal language, and yet to have a worryingly high number of bugs that have been fixed through published errata. Although there are good books that present XML Schema in a more readable way, they achieve this by glossing over the complications, which means that the error messages you get when you do something wrong can be extremely obscure. As a result, there has been a significant amount of support for an alternative schema language, Relax NG, which as it happens was co-developed by the designer of XSLT and XPath, James Clark, and is widely regarded as a much more elegant approach.

The XSL and XQuery working groups responded to these concerns by ensuring that support for XML Schema was optional, both for implementors and for users. However, this has not entirely quelled the voices of dissent. Some have asked for XSLT to offer a choice of schema languages. However, this is technically very difficult to achieve, since the structure of data and the semantics of the operations that can be performed on the data are so closely coupled with each other.

The signs are that XML Schema is here to stay, whether people like it or not. It has the backing of all the major software vendors such as IBM, Oracle, and Microsoft, and it is being adopted by many of the larger user organizations and industries. And like so many things that the IT world has adopted as standards, it may be imperfect but it does actually work. Meanwhile, to simplify the situation rather cruelly, Relax NG is taking the role of the Apple Macintosh: the choice of the cognoscenti who judge a design by its intrinsic quality rather than by its list of heavyweight backers.

As I've already mentioned, W3C is not an organization that likes to let a thousand flowers bloom. It is not a loose umbrella organization in which each working group is free to do its own thing. There are strong processes that ensure the working groups cooperate and strive to reconcile their differences. There is therefore a determination to make all the specifications work properly together, and the message is that if XML Schema has its problems, then you work together to get them fixed. XSLT and XML Schema come from the same stable, so they are expected to work together. And I hope to show in this book that they can work together beneficially.

Chapter 4 provides an overview of how stylesheets and schemas are integrated in XSLT 2.0, and Chapter 11 provides a worked example of an application that uses this capability. In developing this application for the book (which I did at the same time as I developed the underlying support in Saxon) I was pleasantly surprised to see that I really was getting benefits from the integration. At the simplest level, I really liked the immediate feedback you get when a stylesheet generates output that does not conform to the schema for the result document, with error messages that point straight to the offending line in the stylesheet. This makes for a much faster debugging cycle than does the old approach of putting the finished output file through a schema validator as a completely separate operation.

The History of XSL

Like most of the XML family of standards, XSLT was developed by the World Wide Web Consortium (W3C), a coalition of companies orchestrated by Tim Berners-Lee, the inventor of the Web. There is an interesting page on the history of XSL, and styling proposals generally, at `http://www.w3.org/Style/History/`.

Writing history is a tricky business. Sharon Adler, the chair of the XSL Working Group, tells me that her recollections of what happened are very different from the way I describe them. This just goes to show that the documentary record is a very crude snapshot of what people were actually thinking and talking about. Unfortunately, though, it's all that we've got.

Prehistory

HTML was originally conceived by Berners-Lee as a set of tags to mark the logical structure of a document; headings, paragraphs, links, quotes, code sections, and the like. Soon, people wanted more control over how the document looked; they wanted to achieve the same control over the appearance of the delivered publication as they had with printing and paper. So, HTML acquired more and more tags and attributes to control presentation; fonts, margins, tables, colors, and all the rest that followed. As it evolved, the documents being published became more and more browser-dependent, and it was seen that the original goals of simplicity and universality were starting to slip away.

The remedy was widely seen as separation of content from presentation. This was not a new concept; it had been well developed through the 1980s in the development of *Standard Generalized Markup Language* (*SGML*).

Just as XML was derived as a greatly simplified subset of SGML, so XSLT has its origins in an SGML-based standard called *DSSSL (Document Style Semantics and Specification Language)*. DSSSL (pronounced *Dissel*) was developed primarily to fill the need for a standard device-independent language to define the output rendition of SGML documents, particularly for high-quality typographical presentation. SGML was around for a long time before DSSSL appeared in the early 1990s, but until then the output side had been handled using proprietary and often extremely expensive tools, geared toward driving equally expensive phototypesetters, so that the technology was really taken up only by the big publishing houses.

Michael Sperberg-McQueen and Robert F. Goldstein presented an influential paper at the WWW '94 conference in Chicago under the title *A Manifesto for Adding SGML Intelligence to the World-Wide Web*. You can find it at `http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/Autools/sperberg-mcqueen/sperberg.html`.

The authors presented a set of requirements for a stylesheet language, which is as good a statement as any of the aims that the XSL designers were trying to meet. As with other proposals from around that time, the concept of a separate transformation language had not yet appeared, and a great deal of the paper is devoted to the rendition capabilities of the language. There are many formative ideas, however, including the concept of fallback processing to cope with situations where particular features are not available in the current environment.

It is worth quoting some extracts from the paper here:

> *Ideally, the stylesheet language should be declarative, not procedural, and should allow stylesheets to exploit the structure of SGML documents to the fullest. Styles must be able to vary with the structural location of the element: paragraphs within notes may be formatted differently from paragraphs in the main text. Styles must be able to vary with the attribute values of the element in question: a quotation of type "display" may need to be formatted differently from a quotation of type "inline"*
>
> *At the same time, the language has to be reasonably easy to interpret in a procedural way: implementing the stylesheet language should not become the major challenge in implementing a Web client.*
>
> *The semantics should be additive: It should be possible for users to create new stylesheets by adding new specifications to some existing (possibly standard) stylesheet. This should not require copying the entire base stylesheet; instead, the user should be able to store locally just the user's own changes to the standard stylesheet, and they should be added in at browse time. This is particularly important to support local modifications of standard DTDs.*
>
> *Syntactically, the stylesheet language must be very simple, preferably trivial to parse. One obvious possibility: formulate the stylesheet language as an SGML DTD, so that each stylesheet will be an SGML document. Since the browser already knows how to parse SGML, no extra effort will be needed.*
>
> *We recommend strongly that a subset of DSSSL be used to formulate stylesheets for use on the World Wide Web; with the completion of the standards work on DSSSL, there is no reason for any community to invent*

their own style-sheet language from scratch. The full DSSSL standard may well be too demanding to implement in its entirety, but even if that proves true, it provides only an argument for defining a subset of DSSSL that must be supported, not an argument for rolling our own. Unlike home-brew specifications, a subset of a standard comes with an automatically predefined growth path. We expect to work on the formulation of a usable, implementable subset of DSSSL for use in WWW stylesheets, and invite all interested parties to join in the effort

In late 1995, a W3C-sponsored workshop on stylesheet languages was held in Paris. In view of the subsequent role of James Clark as editor of the XSLT Recommendation, it is interesting to read the notes of his contribution on the goals of DSSSL, which can be found at `http://www.w3.org/Style/951106_Workshop/report1.html#clark`.

Here are a few selected paragraphs from these notes.

DSSSL contains both a transformation language and a formatting language. Originally the transformation was needed to make certain kinds of styles possible (such as tables of contents). The query language now takes care of that, but the transformation language survives because it is useful in its own right.

The language is strictly declarative, which is achieved by adopting a functional subset of Scheme. Interactive stylesheet editors must be possible.

A DSSSL stylesheet very precisely describes a function from SGML to a flow object tree. It allows partial stylesheets to be combined ("cascaded" as in CSS): some rule may override some other rule, based on implicit and explicit priorities, but there is no blending between conflicting styles.

James Clark closed his talk with the remark:

Creating a good, extensible style language is hard!

One suspects that the effort of editing the XSLT 1.0 Recommendation didn't cause him to change his mind.

The First XSL Proposal

Following these early discussions, the W3C set up a formal activity to create a stylesheet language proposal. The remit for this group specified that it should be based on DSSSL.

As an output of this activity came the first formal proposal for XSL, dated 27 August 1997. Entitled *A Proposal for XSL*, it lists 11 authors: James Clark (who works for himself), five from Microsoft, three from Imso Corporation, one from ArborText, and one (Henry Thompson) from the University of Edinburgh. The document can be found at `http://www.w3.org/TR/NOTE-XSL.html`.

The section describing the purpose of the language is worth reading.

XSL is a stylesheet language designed for the Web community. It provides functionality beyond CSS (e.g. element reordering). We expect that CSS will be used to display simply structured XML documents and XSL will be used where more powerful formatting capabilities are required or for formatting highly structured information such as XML structured data or XML documents that contain structured data.

Web authors create content at three different levels of sophistication given as follows:

P1: IML
WY020-01 WY020-Kay WY020-Kay-v2.cls July 7, 2004 17:53

- *markup: relies solely on a declarative syntax*
- *script: additionally uses code "snippets" for more complex behaviors*
- *program: uses a full programming language*

XSL is intended to be accessible to the "markup" level user by providing a declarative solution to most data description and rendering requirements. Less common tasks are accommodated through a graceful escape to a familiar scripting environment. This approach is familiar to the Web publishing community as it is modeled after the HTML/JavaScript environment.

The powerful capabilities provided by XSL allow:

- *formatting of source elements based on ancestry/descendency, position, and uniqueness*
- *the creation of formatting constructs including generated text and graphics*
- *the definition of reusable formatting macros*
- *writing-direction independent stylesheets*
- *extensible set of formatting objects*

The authors then explained carefully why they had felt it necessary to diverge from DSSSL and described why a separate language from CSS (Cascading Style Sheets) was thought necessary.

They then stated some design principles:

- XSL should be straightforwardly usable over the Internet.
- XSL should be expressed in XML syntax.
- XSL should provide a declarative language to do all common formatting tasks.
- XSL should provide an "escape" into a scripting language to accommodate more sophisticated formatting tasks and to allow for extensibility and completeness.
- XSL will be a subset of DSSSL with the proposed amendment. *(As XSL was no longer a subset of DSSSL, they cannily proposed amending DSSSL so it would become a superset of XSL.)*
- A mechanical mapping of a CSS stylesheet into an XSL stylesheet should be possible.
- XSL should be informed by user experience with the FOSI stylesheet language.
- The number of optional features in XSL should be kept to a minimum.
- XSL stylesheets should be human-legible and reasonably clear.
- The XSL design should be prepared quickly.
- XSL stylesheets shall be easy to create.
- Terseness in XSL markup is of minimal importance.

As a requirements statement, this doesn't rank among the best. It doesn't read like the kind of list you get when you talk to users and find out what they need. It's much more the kind of list that designers write when they know what they want to produce, including a few political concessions to the people who might raise objections. But if you want to understand why XSLT became the language it did, this list is certainly evidence of the thinking.

The language described in this first proposal contains many of the key concepts of XSLT as it finally emerged, but the syntax is virtually unrecognizable. It was already clear that the language should be based on templates that handled nodes in the source document matching a defined pattern, and that the language should be free of side effects, to allow "progressive rendering and handling of large documents." I'll explore the significance of this requirement in more detail on page 36, and discuss its implications on the way stylesheets are designed in Chapter 9. The basic idea is that if a stylesheet is expressed as a collection of completely independent operations, each of which has no external effect other than generating part of the output from its input (for example, it cannot update global variables), then it becomes possible to generate any part of the output independently if that particular part of the input changes. Whether the XSLT language actually achieves this objective is still an open question.

Microsoft shipped its first technology preview 5 months after this proposal appeared, in January 1998.

To enable W3C to make an assessment of the proposal, Norman Walsh produced a requirements summary, which was published in May 1998. It is available at `http://www.w3.org/TR/WD-XSLReq`. It largely confirms the thinking already outlined.

The bulk of his paper is given over to a long list of the typographical features that the language should support, following the tradition that the formatting side of the language originally got a lot more column inches than did the transformation side.

Following this activity, the first Working Draft of XSL (not to be confused with the Proposal) was published on 18 August 1998, and the language started to take shape, gradually converging on the final form it took in the 16 November 1999 Recommendation through a series of Working Drafts, each of which made radical changes, but kept the original design principles intact.

A Recommendation is the most definitive of documents produced by the W3C. It's not technically a standard, because standards can only be published by government-approved standards organizations. But I will often refer to it loosely as "the standard" in this book.

The Microsoft WD-xsl Dialect

Before the Recommendation came out, however, Microsoft took a fateful decision to ship an early implementation of their XSLT processor as an add-on to Internet Explorer 4, and later as a built-in feature of IE5. Unfortunately, Microsoft was too early, and the XSLT standard changed and grew. When the XSLT Recommendation version 1.0 was finally published on 16 November 1999, it had diverged significantly from the initial Microsoft product.

Many of the differences, such as changes of keywords, are very superficial but some run much deeper; for example, changes in the way the equals operator is defined.

Fortunately, the Microsoft IE5 dialect of XSL (which I refer to as WD-xsl) is now almost completely obsolete. Microsoft no longer actively promotes it, and their more recent products are very closely aligned with the W3C specifications. It's still possible, however, that you will come across stylesheets written in this language, or developers who aren't aware of the differences. You can recognize stylesheets written in this dialect by the namespace URI on the `<xsl:stylesheet>` element, which is «`http://www.w3.org/TR/WD-xsl`».

Saxon

At this point it might be a good idea to clarify how I got involved in the story. In 1998 I was working for the British computer manufacturer ICL, a part of Fujitsu. Fujitsu, in Japan, had developed an object database system, later marketed by Computer Associates as Jasmine, and I was trying to find applications for this technology in content management applications for large publishers. We developed a few successful large systems with this technology, but found that it didn't scale downwards to the kind of project that wanted something working in 6 weeks rather than 6 months. So I was asked to look at what we could do with XML, which was just appearing on the horizon.

I came to the conclusion that XML looked like a good thing, but that there wasn't any software. So I developed the very first early versions of Saxon to provide a proof-of-concept demonstration. At that stage Saxon was just a Java library, not an XSLT processor, but as the XSL standards developed I found that my own ideas were converging more and more with what the W3C working group was doing, and I started implementing the language as it was being specified. ICL had decided that its marketing resources were spread thinly over too many products, and so the management took the imaginative decision to make the technology available as open source. Seventeen days after the XSLT 1.0 specification was published in November 1999, I announced the first conformant implementation. And on the day it was published, I started work on the first edition of this book.

When the book was published, the XSL Working Group invited me to join and participate in the development of XSLT 1.1. Initially, being based in the United Kingdom and with limited time available for the work, my involvement was fairly sporadic. But early in 2001 I changed jobs and joined Software AG, which wanted me to take a full role in the W3C work. The following year James Clark pulled out of the Working Group, and I stepped into his shoes as editor.

The reason I'm explaining this sequence of events is that I hope it will help you to understand the viewpoint from which this book is written. When I wrote the first edition I was an outsider, and I felt completely free to criticize the specification when I felt it necessary. I have tried to retain an objective approach in the present edition, but as editor of the language spec it is much more difficult to be impartial. I've tried to keep a balance: it wouldn't be fair to use the book as a platform to push my views over those of my colleagues of the working group, but at the same time, I've made no effort to be defensive about decisions that I would have made differently if they had been left to me.

Software AG continued to support my involvement in the W3C work (on the XQuery group as well as the XSL group), as well as the development of Saxon and the writing of this book, through till February 2004, at which point I left to set up my own company, Saxonica.

Beyond XSLT 1.0

After XSLT 1.0 was published, the XSL Working Group responsible for the language decided to split the requirements for enhancements into two categories: XSLT 1.1 would standardize a small number of urgent features that vendors had already found necessary to add to their products as extensions, while XSLT 2.0 would handle the more strategic requirements that needed further research.

A working draft of XSLT 1.1 was published on December 12, 2000. It described three main enhancements to the XSLT 1.0 specification:

- *Multiple output documents*: an `<xsl:document>` instruction, modeled on extensions provided initially in Saxon and subsequently in other products including xt, Xalan and Oracle, allowing a source document to be split into multiple output documents. This instruction has become `<xsl:result-document>` in XSLT 2.0.
- *Temporary trees*: the ability to treat a tree created by one phase of processing as input to a subsequent phase of processing. This enhancement was modeled on the `node-set()` extension function introduced first in xt and subsequently copied in other products. It is retained largely unchanged in XSLT 2.0.
- *Standard bindings to extension functions*: written in Java and ECMAScript. XSLT 1.0 allowed a stylesheet to call external functions, but did not say how such functions should be written, with the result that extension functions written for Xalan would not work with xt or Saxon, or vice versa. XSLT 1.1 defined a general framework for binding extension functions written in any language, with specific mappings for Java and ECMAScript (the official name for JavaScript). This feature of the XSLT 1.1 draft has been dropped completely from XSLT 2.0. It proved highly controversial, particularly as it coincided with Microsoft's U-turn in its Java strategy.

For a number of reasons XSLT 1.1 never got past the working draft stage. This was partly because of the controversy surrounding the Java language bindings, but more particularly because it was becoming clearer that XSLT 2.0 would be a fairly radical revision of the language, and the working group didn't want to do anything in 1.1 that would get in the way of achieving the 2.0 goals. There were feelings, for example, that the facility for temporary trees might prejudice the ability to support sequences in 2.0, a fear which as it happens proved largely unfounded.

XQuery

By the time work on XSLT 2.0 was starting, the separate XQuery working group in W3C had created a draft of its own language.

While the XSL working group had identified the need for a transformation language to support a self-contained part of the formatting process, XQuery originated from the need to search large quantities of XML documents stored in a database.

Different people had different motivations for wanting an XML Query Language, and many of these motivations were aired at a workshop held in December 1998. You can find all 66 position papers presented at this workshop at `http://www.w3.org/TandS/QL/QL98/pp.html`. Quite how a consensus emerged from this enormous variety of views is difficult to determine in retrospect. But it's interesting to see how the participants saw the relationship with XSL, as it was then known. The Microsoft position paper states the belief that a query language could be developed as an extension of XSLT, but in this it is almost alone. Many of the participants came from a database background, with ideas firmly rooted in the tradition of SQL and object database languages such as OQL, and to these people, XSL didn't look remotely like a query language. But in the light of subsequent events, it's interesting to read the position paper from the XSL Working Group, which states in its summary:

1. The query language should use XSL patterns as the basis for information retrieval.
2. The query language should use XSL templates as the basis for materializing query results.

3. The query language should be at least as expressive as XSL is, currently.
4. Development of the pattern and transformation languages should remain in the XSL working group.
5. A coordination group should ensure either that a single query language satisfies all working group requirements or that all W3C query languages share an underlying query model.

(Remember that XPath had not yet been identified as a separate language, and that the expressions that later became XPath were then known as patterns.)

This offer to coordinate, and the strong desire to ensure consistency among the different W3C specifications, can be seen as directly leading to the subsequent collaboration between the two working groups to define XPath 2.0.

The XQuery group started meeting in September 1999. The first published requirements document was published the following January (`http://www.w3.org/TR/2000/WD-xmlquery-req-20000131`). It included a commitment to compatibility with XML Schema, and a rather cautiously worded promise to "take into consideration the expressibility and search facilities of XPath when formulating its algebra and query syntax." July 2000 saw a revised requirements document that included a selection of queries that the language must be able to express. The first externally visible draft of the XQuery language was published in February 2001 (see `http://www.w3.org/TR/2001/WD-xquery-20010215/`) and it was at this stage that the collaboration between the two working groups began in earnest.

The close cooperation between the teams developing the two languages contrasts strangely with the somewhat adversarial position adopted by parts of the user community. XSLT users were quick to point out that XSLT 1.0 satisfied every single requirement in the first XQuery requirements document, and could solve all the use cases published in the second version in August 2000. At the same time, users on the XQuery side of the fence have often been dismissive about XSLT, complaining about its verbose syntax and sometimes arcane semantics. Even today, when the similarities of the two languages at a deep level are clearly apparent, there is very little overlap between their user communities: I find that most users of the XQuery engine in Saxon have no XSLT experience. The difference between XSLT and XQuery is in many ways a difference of style rather than substance, but users often feel strongly about style.

XSLT 2.0 and XPath 2.0

The requirements for XSLT 2.0 and XPath 2.0 were published on 14 February 2001. In the case of the XPath 2.0 requirements, the document was written jointly by the two working groups. You can find the documents at the following URLs:

```
http://www.w3.org/TR/2001/WD-xslt20req-20010214

http://www.w3.org/TR/2001/WD-xpath20req-20010214
```

Broadly, the requirements fall into three categories:

❑ Features that are obviously missing from the current standards and that would make users' lives much easier, for example, facilities for grouping related nodes, extra string-handling and numeric functions, and the ability to read text files as well as XML documents.

P1: IML
WY020-01 WY020-Kay WY020-Kay-v2.cls July 7, 2004 17:53

- ❑ Changes desired by the XML Query working group. The difficulty at this stage was that the Query group did not just want additions to the XPath language; they wanted fundamental changes to its semantics. Many members of the XQuery group felt they could not live with some of the arbitrariness of the way XPath handled data types generally, and node-sets in particular, for example the fact that «`a = 1`» tests whether there is some «`a`» that equals one, whereas «`a - 1 = 0`» tests whether the first «`a`» equals one.
- ❑ Features designed to exploit and integrate with XML Schema. The W3C XML Schema specification had reached an advanced stage (it became a Candidate Recommendation on 20 October 2000), and implementations were starting to appear in products. The thinking was that if the schema specified that a particular element contains a number or a date (for example), then it ought to be possible to use this knowledge when comparing or sorting dates within a stylesheet.

It has sometimes been suggested that the adoption of XML Schema was forced on XSLT by the XQuery group. I don't think there is any truth in this. The big three database companies (IBM, Oracle, and Microsoft) were very active in both working groups, along with middle-tier players such as BEA and Software AG. Most of these companies saw XML Schema as a strategic way forward, and they carried both groups along with them. What is almost certainly true, however, is that when the XSL Working Group made the decision to work with XML Schema, this was based on a rather vague idea of the potential benefits; none of the members at that stage had a clear idea as to the detailed implications of this decision on the design of the language.

The development of XSLT 2.0 has been a long drawn out process. The timescale was dictated largely by the pace at which agreement could be reached with the XQuery group on the details of XPath 2.0. This took a long time firstly, because of the number of people involved; secondly, because of the very different places where people were coming from (the database community and the document community have historically been completely isolated from each other, and it took a lot of talking before people started to understand each others' positions); and finally, because of the sheer technical difficulty of finding a workable design that offered the right balance between backwards compatibility and rigorous, consistent semantics. A great deal of the credit for finding a way through these obstacles goes to Mary Fernandez, who chaired the joint XPath task force with remarkable patience and persistence.

So much for the history. Let's look now at the essential characteristics of XSLT 2.0 as a language.

XSLT 2.0 as a Language

What are the most significant characteristics of XSLT as a language, which distinguish it from other languages? In this section I shall pick four of the most striking features: the fact that it is written in XML syntax, the fact that it is a language free of side effects, the fact that processing is described as a set of independent pattern-matching rules, and the fact that it has a type system based on XML Schema.

Use of XML Syntax

As we've seen, the use of SGML syntax for stylesheets was proposed as long ago as 1994, and it seems that this idea gradually became the accepted wisdom. It's difficult to trace exactly what the overriding arguments were, and when you find yourself writing something like:

```
<xsl:variable name="y">
   <xsl:call-template name="f">
      <xsl:with-param name="x"/>
   </xsl:call-template>
</xsl:variable>
```

to express what in other languages would be written as «`y=f(x);`», then you may find yourself wondering how such a decision came to be made.

In fact, it could have been worse; in the very early drafts, the syntax for writing what are now XPath expressions was also expressed in XML, so instead of writing «`select="book/author/first-name"`» you had to write something along the lines of:

```
<select>
   <path>
      <element type="book">
      <element type="author">
      <element type="first-name">
   </path>
</select>
```

The most obvious arguments for expressing XSLT stylesheets in XML are perhaps as follows:

- ❑ There is already an XML parser in the browser; so it keeps the footprint small if this can be reused.
- ❑ Everyone had got fed up with the syntactic inconsistencies between HTML/XML and CSS and didn't want the same thing to happen again.
- ❑ The Lisp-like syntax of DSSSL was widely seen as a barrier to its adoption; so it would be better to have a syntax that was already familiar in the target community.
- ❑ Many existing popular template languages (including simple ASP and JSP pages) are expressed as an outline of the output document with embedded instructions; so this is a familiar concept.
- ❑ The lexical apparatus is reusable, for example Unicode support, character and entity references, whitespace handling, namespaces.
- ❑ It's occasionally useful to have a stylesheet as the input or output of a transformation (witness the Microsoft XSL converter as an example); so it's a benefit if a stylesheet can read and write other stylesheets.
- ❑ Providing visual development tools easily solves the inconvenience of having to type lots of angle brackets.

Like it or not, the XML-based syntax is now an intrinsic feature of the language that has both benefits and drawbacks. It does make the language verbose, but in the end, the number of keystrokes has very little bearing on the ease or difficulty of solving particular transformation problems.

In XSLT 2.0, the long-windedness of the language has been reduced considerably by increasing the expressiveness of the non-XML part of the syntax, namely XPath expressions. Many computations that required five lines of XSLT code in 1.0 can be expressed in a single XPath expression in 2.0. Two constructs in particular led to this simplification: the conditional expression (`if..then..else`) in XPath 2.0; and the ability to define a function in XSLT (using `<xsl:function>`) that can be called directly from an

XPath expression. To take the example discussed earlier, if you replace the template «f» by a user-written function «f», you can replace the five lines in the example with:

```
<xsl:variable name="y" select="f($x)"/>
```

The decision to base the XSLT syntax on XML has proved its worth in several ways that I would not have predicted in advance:

- It has proved very easy to extend the syntax. Adding new elements and attributes is trivial; there is no risk of introducing parsing difficulties when doing so, and it is easy to manage backwards compatibility. (In contrast, extending XQuery's non-XML syntax without introducing parsing ambiguities is a highly delicate operation.)
- The separation of XML parsing from XSLT processing leads to good error reporting and recovery in the compiler. It makes it much easier to report the location of an error with precision and to report many errors in one run of the compiler. This leads to a faster development cycle.
- It makes it easier to maintain stylistic consistency between different constructs in the language. The discipline of defining the language through elements and attributes creates a constrained vocabulary with which the language designers must work, and these constraints impose a certain consistency of design.

No Side Effects

The idea that XSL should be a declarative language free of side effects appears repeatedly in the early statements about the goals and design principles of the language, but no one ever seems to explain *why*: what would be the user benefit?

A function or procedure in a programming language is said to have side effects if it makes changes to its environment; for example, if it can update a global variable that another function or procedure can read, or if it can write messages to a log file, or prompt the user. If functions have side effects, it becomes important to call them the right number of times and in the correct order. Functions that have no side effects (sometimes called pure functions) can be called any number of times and in any order. It doesn't matter how many times you evaluate the area of a triangle, you will always get the same answer; but if the function to calculate the area has a side effect such as changing the size of the triangle, or if you don't know whether it has side effects or not, then it becomes important to call it once only.

I expand further on this concept in the section on Computational Stylesheets in Chapter 9, page 625.

It is possible to find hints at the reason why this was considered desirable in the statements that the language should be equally suitable for batch or interactive use, and that it should be capable of *progressive rendering*. There is a concern that when you download a large XML document, you won't be able to see anything on your screen until the last byte has been received from the server. Equally, if a small change were made to the XML document, it would be nice to be able to determine the change needed to the screen display, without recalculating the whole thing from scratch. If a language has side effects then the order of execution of the statements in the language has to be defined, or the final result becomes unpredictable. Without side effects, the statements can be executed in any order, which means it is possible, in principle, to process the parts of a stylesheet selectively and independently.

Whether XSLT has actually achieved these goals is somewhat debatable. Certainly, determining which parts of the output document are affected by a small change to one part of the input document is not easy,

P1: IML
WY020-01 WY020-Kay WY020-Kay-v2.cls July 7, 2004 17:53

given the flexibility of the expressions and patterns that are now permitted in the language. Equally, most existing XSLT processors require the whole document to be loaded into memory (there is a version of `jd.xslt` that is disk-based, but this is the exception to the rule). However, there has been research work that suggests the goals are achievable (search for papers on "Incremental XSLT Transformation" or "Lazy XSLT Transformation"). When E. F. Codd published the relational calculus in 1970, he made the claim that a declarative language was desirable because it was possible to optimize it, which was not possible with the navigational data access languages in use at the time. In fact, it took another 15 years before relational optimization techniques (and, to be fair, the price of hardware) reached the point where large relational databases were commercially viable. But in the end he was proved right, and the hope is that the same principle will also eventually deliver similar benefits in the area of transformation and styling languages.

Of course, there will always be some transformations where the whole document needs to be available before you can produce any output; examples are where the stylesheet sorts the data, or where it starts with a table of contents. But there are many other transformations where the order of the output directly reflects the order of the input, and progressive rendering should be possible in such cases. Xalan-J made a start on this by running a transformation thread in parallel with the parsing thread, so the transformer can produce output before the parser has finished, and MSXML3 (from the evidence of its API) seems to be designed on a similar principle. The Stylus Studio debugging tool tracks the dependencies between parts of the output document and the template rules that were used to generate them, and so one can start to see the potential to regenerate the output selectively when small changes are made.

What it means in practice to be free of side effects is that you cannot update the value of a variable. This restriction is something many users find very frustrating at first, and a big price to pay for these rather remote benefits. But as you get the feel of the language and learn to think about using it the way it was designed to be used, rather than the way you are familiar with from other languages, you will find you stop thinking about this as a restriction. In fact, one of the benefits is that it eliminates a whole class of bugs from your code. I shall come back to this subject in Chapter 9, where I outline some of the common design patterns for XSLT stylesheets and, in particular, describe how to use recursive code to handle situations where in the past you would probably have used updateable variables to keep track of the current state.

Rule-Based

The dominant feature of a typical XSLT stylesheet is that it consists of a sequence of template rules, each of which describes how a particular element type or other construct should be processed. The rules are not arranged in any particular order; they don't have to match the order of the input or the order of the output, and in fact there are very few clues as to what ordering or nesting of elements the stylesheet author expects to encounter in the source document. It is this that makes XSLT a declarative language, because you specify what output should be produced when particular patterns occur in the input, as distinct from a procedural program where you have to say what tasks to perform in what order.

This rule-based structure is very like CSS, but with the major difference that both the patterns (the description of which nodes a rule applies to) and the actions (the description of what happens when the rule is matched) are much richer in functionality.

Example: Displaying a Poem

Let's see how we can use the rule-based approach to format a poem. Again, we haven't introduced all the concepts yet, and so I won't try to explain every detail of how this works, but it's useful to see what the template rules actually look like in practice.

Input

Let's take this poem as our XML source. The source file is called `poem.xml`, and the stylesheet is `poem.xsl`.

```
<poem>
   <author>Rupert Brooke</author>
   <date>1912</date>
   <title>Song</title>
   <stanza>
      <line>And suddenly the wind comes soft,</line>
      <line>And Spring is here again;</line>
      <line>And the hawthorn quickens with buds of green</line>
      <line>And my heart with buds of pain.</line>
   </stanza>
   <stanza>
      <line>My heart all Winter lay so numb,</line>
      <line>The earth so dead and frore,</line>
      <line>That I never thought the Spring would come again</line>
      <line>Or my heart wake any more.</line>
   </stanza>
   <stanza>
      <line>But Winter's broken and earth has woken,</line>
      <line>And the small birds cry again;</line>
      <line>And the hawthorn hedge puts forth its buds,</line>
      <line>And my heart puts forth its pain.</line>
   </stanza>
</poem>
```

Output

Let's write a stylesheet such that this document appears in the browser, as shown in Figure 1-8.

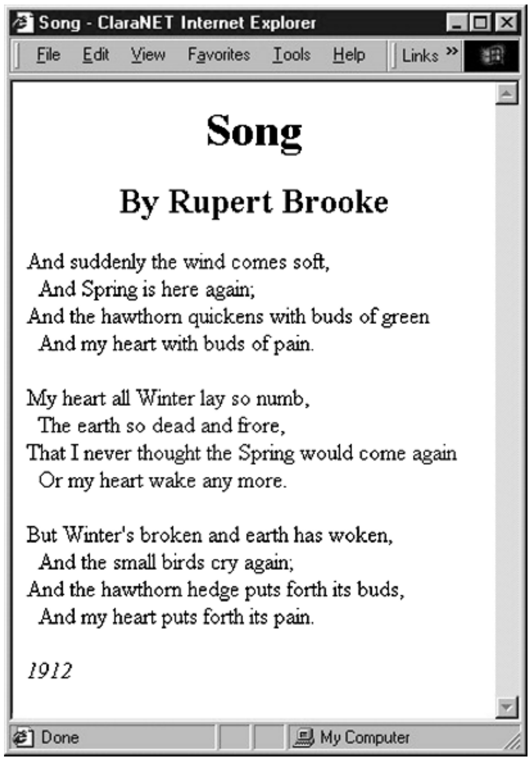
Stylesheet

It starts with the standard header.

```
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">
```

Now we write one template rule for each element type in the source document. The rule for the `<poem>` element creates the skeleton of the HTML output, defining the ordering of the elements in the output (which doesn't have to be the same as the input order). The `<xsl:value-of>` instruction inserts the value of the selected element at this point in the output. The `<xsl:apply-templates>` instructions cause the selected child elements to be processed, each using its own template rule.

```
<xsl:template match="poem">
   <html>
   <head>
```


Song - ClaraNET Internet Explorer
File Edit View Favorites Tools Help Links
Done
My Computer

Song

By Rupert Brooke

And suddenly the wind comes soft,
  And Spring is here again;
And the hawthorn quickens with buds of green
  And my heart with buds of pain.

My heart all Winter lay so numb,
  The earth so dead and frore,
That I never thought the Spring would come again
  Or my heart wake any more.

But Winter's broken and earth has woken,
  And the small birds cry again;
And the hawthorn hedge puts forth its buds,
  And my heart puts forth its pain.

1912

Figure 1-8

```
        <title><xsl:value-of select="title"/></title>
     </head>
     <body>
        <xsl:apply-templates select="title"/>
        <xsl:apply-templates select="author"/>
        <xsl:apply-templates select="stanza"/>
        <xsl:apply-templates select="date"/>
     </body>
     </html>
  </xsl:template>
```

In XSLT 2.0 we could replace the four `<xsl:apply-templates>` instructions with one, written as follows:

```
  <xsl:apply-templates select="title, author, stanza, date"/>
```

This takes advantage of the fact that the type system for the language now supports ordered sequences. The «`,`» operator performs list concatenation and is used here to form a list containing the `<title>`, `<author>`, `<stanza>`, and `<date>` elements in that order. Note that this includes all the `<stanza>` elements, so in general this will be a sequence containing more than four items.

The template rules for the `<title>`, `<author>`, and `<date>` elements are very simple; they take the content of the element (denoted by «`select="."`»), and surround it within appropriate HTML tags to define its display style.

```
<xsl:template match="title">
   <div align="center">
      <h1><xsl:value-of select="."/></h1>
   </div>
</xsl:template>

<xsl:template match="author">
   <div align="center">
      <h2>By <xsl:value-of select="."/></h2>
   </div>
</xsl:template>

<xsl:template match="date">
   <p><i><xsl:value-of select="."/></i></p>
</xsl:template>
```

The template rule for the `<stanza>` element puts each stanza into an HTML paragraph, and then invokes processing of the lines within the stanza, as defined by the template rule for lines:

```
<xsl:template match="stanza">
<p><xsl:apply-templates select="line"/></p>
</xsl:template>
```

The rule for `<line>` elements is a little more complex: if the position of the line within the stanza is an even number, it precedes the line with two non-breaking-space characters (`&#160;`). The `<xsl:if>` instruction tests a boolean condition, which in this case calls the `position()` function to determine the relative position of the current line. It then outputs the contents of the line, followed by an empty HTML `<br>` element to end the line.

```
<xsl:template match="line">
   <xsl:if test="position() mod 2 = 0">&#160;&#160;</xsl:if>
   <xsl:value-of select="."/><br/>
</xsl:template>
```

And to finish off, we close the `<xsl:stylesheet>` element.

```
</xsl:stylesheet>
```

Although template rules are a characteristic feature of the XSLT language, we'll see that this is not the only way of writing a stylesheet. In Chapter 9, I will describe four different design patterns for XSLT stylesheets, only one of which makes extensive use of template rules. In fact, the `Hello World` stylesheet I presented earlier in this chapter doesn't make any real use of template rules; it fits into the design pattern I call *fill-in-the-blanks*, because the stylesheet essentially contains the fixed part of the output with embedded instructions saying where to get the data to put in the variable parts.

Types Based on XML Schema

I have described three characteristics of the XSLT language (the use of XML syntax, the principle of no side-effects, and the rule-based processing model) that were essential features of XSLT 1.0 and that have been retained essentially unchanged in XSLT 2.0. The fourth characteristic is new in XSLT 2.0, and creates a fundamental change in the nature of XSLT as a language. This is the adoption of a type system based on XML Schema.

There are two aspects to the type system of any programming language. The first is the set of types that are supported (for example, integers, strings, lists, tuples), together with the mechanisms for creating user-defined types. The second aspect is the rules that the language enforces to ensure type-correctness.

XSLT 1.0 had a very small set of types (integers, booleans, strings, node-sets, and result tree fragments), and the rules it applied were what is often called "weak typing": this means that the processor would always attempt to convert the value supplied in an expression or function call to the type that was required in that context. This makes for a very happy-go-lucky environment: if you supply an integer where a string is expected, or vice versa, nothing will break.

XSLT 2.0 has changed both aspects of the type system. There is now a much richer set of types available (and this set is user-extensible), and the rules for type-checking are stricter.

We will look at the implications of this in greater detail in Chapter 4.

Summary

This introductory chapter answered the following questions about XSLT:

- ❑ What kind of language is it?
- ❑ Where does it fit into the XML family?
- ❑ Where does it come from and why was it designed the way it is?
- ❑ Where should it be used?

You now know that XSLT is a declarative high-level language designed for transforming the structure of XML documents; that it has two major applications: data conversion and presentation; and that it can be used at a number of different points in the overall application architecture, including at data capture time, at delivery time on the server, and at display time on the browser. You also have some idea why XSLT has developed in the way it has.

Now it's time to start taking an in-depth look inside the language to see how it does this job. In the next chapter, we look at the way transformation is carried out by treating the input and output as tree structures, and using patterns to match particular nodes in the input tree and define what nodes should be added to the result tree when the pattern is matched.