XSLT in Context

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, and how it came to be that way; 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 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.

I shall begin 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.

The chapter then moves on to 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.

Finally, I'll have a few things to say about 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.

What is XSLT?

XSLT, which stands for **eXtensible Stylesheet Language: Transformations**, is a language which, according to the very first sentence in the specification (found at http://www.w3.org/TR/xslt), is primarily designed for transforming one XML document into another. However, XSLT is more than 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 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.

Why Transform XML?

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. This need is becoming ever more urgent as the range of internet-capable devices grows. Organizations that have invested in creating valuable information sources need to be able to deliver them not only to the traditional PC-based web browser (which itself now comes in many flavors), but also to TV sets and WAP phones, not to mention the continuing need to produce print-on-paper.
- **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 probably 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.

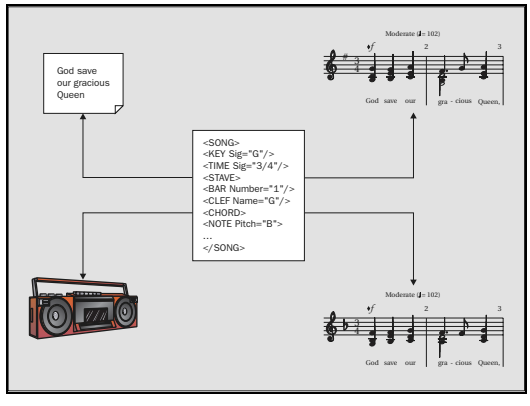
In order to transfer data between different applications we need to be able to transform data from the data model used by one application to the model used in another. To load the data into an application, the required format might be a comma-separated-values file, a SQL script, an HTTP message, or a 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, so 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 newspaper industry is likely to use different formats for exchanging news articles from the format used in the broadcasting industry. Equally, there will always be a need to do things like extracting an address from a purchase order and adding it to an invoice. 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. Before we move on to discuss XSLT in more detail and have a first look at how it works, let's take a look at an example that clearly demonstrates the variety of formats to which we can transform XML, using XSLT.

An Example: Transforming Music

There is an excellent registry of XML vocabularies and schemas at http://www.xml.org/xmlorg_registry/index.shtml.

If you look there, you will find half a dozen different XML schemas for describing music. These were all invented with different purposes in mind: a markup language used by a publisher for printing sheet music has different requirements from one designed to let you listen to the music from a browser. MusicML, for example, is oriented to displaying music notation graphically; ChordML is designed for encoding the harmonic accompaniment to vocal lyrics; MusicXML is designed to represent musical scores, specifically western musical notation from the 17th century onwards, while the rather more academic Music Markup Language (MML) from the University of Pretoria is designed for serious musicological analysis, embracing Eastern and African as well as Western musical idioms.



So you could use XSLT to process marked-up music in many different ways:

- You could use XSLT to convert music from one of these representations to another, for example from MusicXML to MML.
- You could use XSLT to convert music from any of these representations into visual music notation, by generating the XML-based vector graphics format SVG.
- You could use XSLT to play the music on a synthesizer, by generating a MIDI (Musical Instrument Digital Interface) file.
- You could use XSLT to perform a musical transformation, such as transposing the music into a different key.
- You could use XSLT to extract the lyrics, into HTML or into a text-only XML document.

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

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:

- 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.
- 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 VoxML output, for aural rendition of your data.

Links to information about Motorola's VoxML Voice Markup Language can be found at http://www.oasis-open.org/cover/voxML.html

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.

> *Both the DOM and the SAX APIs are fully described in the Wrox Press book Professional XML, ISBN 1-861003-11-0.*

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 – be it a DOM parser or a SAX-compliant one – 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 etc.) 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.*

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

I like to think of an analogy with relational databases. In a relational database, the data consists of a set of tables. By themselves, the tables are not 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 that language 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 26).

The XPath query 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 (rearranging selected nodes, constructing new nodes etc).

There are further similarities between XSLT and SQL:

- Both languages augment the basic query facilities with useful additions for performing basic arithmetic, string manipulation, and comparison operations.
- Both languages supplement the declarative query syntax with semi-procedural facilities for describing the sequence of 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 (with the new XSLT 1.1 specification, currently a working draft) by capturing the output of one transformation phase as a temporary tree, and using that temporary tree as the input of another transformation phase.

In the real world, of course, XSLT and SQL have to coexist. There are many possible relationships, but typically data will be 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 products that integrate XML and SQL, though there are no standards in this area as yet. Check the vendor's web sites for the latest releases of Microsoft SQL Server 2000, and Oracle 9i.

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

XSLT Processors

The principal role of an XSLT processor is to apply an XSLT stylesheet to an XML source document and produce a result document. It is important to note that each of these is an application of XML and so the underlying structure of each is a tree. So, in fact, the XSLT processor handles three trees.

There are several XSLT processors to choose from. Here I'll mention three: Saxon, Xalan, and Microsoft MSXML3. All of these can be downloaded free of charge (but do read the licensing conditions).

> *These three processors and several others are described in detail in the Appendices to this book.*

Saxon is an open source XSLT processor developed by the author of this book. It is a Java application, and can be run directly from the command prompt; no web server or browser is required. The Saxon program will transform the XML document to, say, an HTML document, which can then be placed on a web server. In this example, both the browser and web server only deal with the transformed document.

If you are running Windows (95/98/NT/2000) the simplest way to use it is to download Instant Saxon, which is packaged as a Windows executable. You will need to have Java installed, but that will be there already if you have any recent version of Internet Explorer. On non-Windows platforms you will need to install the full Saxon product and follow the instructions that come with it. You can download Instant Saxon for free from http://users.iclway.co.uk/mhkay/saxon/index.html. Saxon will run with any XML parser that implements the SAX2 interface (in its Java form), but it comes with a copy of the Ælfred parser, so you don't need to install one separately.

Xalan is another open source XSLT processor, available from the Apache organization at http://xml.apache.org/. Xalan was originally derived from an IBM product called LotusXSL , but it has since developed an open-source life of its own. Xalan is available in both Java and C++ versions. Like Saxon, Xalan-Java is a Java application that can be run from the command prompt. Xalan too can operate with any SAX2-compliant parser; it comes with a copy of its stable-mate parser, Xerces.

Saxon and Xalan-Java both implement the same Java interface, known as TrAX, which I will describe in Appendix F: this means you can write applications that work with either processor. Both are highly conformant to the XSLT 1.0 specification, so your stylesheets will be fully portable; but they have different capabilities in terms of what they can do beyond the requirements of the standard. Both products are more fully described in Appendices to this book.

Alternatively, you can run XSLT stylesheets actually within Internet Explorer. The production versions of IE5 and IE5.5 are shipped with a processor that handles a Microsoft dialect of XSLT loosely based on an early 1998 working draft of the W3C specification: this language (which I'll call *working-draft XSL* or *WD-xsl*) is quite different from the final version, and I strongly recommend you to avoid it. Microsoft now provide a fully-conformant implementation of the final XSLT 1.0 specification, which is known as MSXML3. In due course this will probably be shipped as standard with Internet Explorer 6, but in the meantime you can download it separately from http://msdn.microsoft.com and install it for use with IE5 or IE5.5.

Download and install both the SDK and the run-time package. You should also download and install a program called `xmlinst.exe`. Run this program to establish MSXML3 as the default XML processor to be used by Internet Explorer (if you don't do this, IE5 will try to use the old WD-xsl processor). The big advantage of Microsoft's technology is that the XSLT processing can take place on the browser.

I've avoided talking about specific products in most of the book, because the information is likely to change quite rapidly. The products that I've described in the Appendices are reasonably stable by now, but new products are constantly appearing. It's best to get the latest status from the web. Some good places to start are:

- http://www.w3.org/Style/XSL
- http://www.xslinfo.com/
- http://www.xml.com/
- http://www.oasis-open.org/cover

An Example Stylesheet

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

Example: A "Hello, world!" XSLT Stylesheet

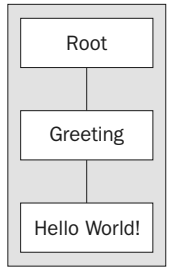
Kernighan and Ritchie in their classic *The C Programming Language* originated the idea of presenting a trivial but complete program right at the beginning of the book, and ever since then the "Hello world" program has been an honored tradition. Of course, a complete description of how this example works is not possible until all the concepts have been defined, 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>
```

A simple node-tree-representation of this document would look as follows:



There is one root node per document. The root node in the XSLT model performs the same function as the document node in the DOM model. The XML declaration is not visible to the parser 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 using any of the three processors described in the previous section.

Saxon

With Saxon, the steps (for a Windows platform) are:

- Download the Instant Saxon processor
- Install the executable `saxon.exe` in a suitable directory, and make this the current directory
- Using Notepad, type the two files above into `hello.xml` and `hello.xsl` respectively, within this directory (or get them from the Wrox web site at http://www.wrox.com)
- 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)

- Type the following at the command prompt:
  `saxon hello.xml hello.xsl`
- 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:

```
saxon hello.xml hello.xsl >hello.html
```

Because you're using the default stylesheet for this source XML document, you could also write:

```
saxon -a hello.xml >hello.html
```

Xalan-Java

The procedure is very similar if you use Xalan-Java. However, Xalan doesn't have a special version packaged for the Windows platform in the way Saxon does, so you'll have to run the application explicitly under Java (you can also do this with Saxon if you use the full version).

You will need to install a Java Virtual Machine. Xalan doesn't support the Microsoft Virtual Machine that comes with Internet Explorer (because it requires features from later versions of Java), so you'll need to do this separately. I usually use Sun's JDK 1.3, which you can get from http://java.sun.com/j2se/1.3/. Installing this is very straightforward, though it's a large download.

Having downloaded Xalan-Java 2 as a `.zip` file, you need to unpack it into a suitable directory. There are two important files that must be accessible to the Java Virtual Machine. These are `xalan.jar` (containing the code of the XSLT processor) and `xerces.jar` (holding the XML parser), both found in the `bin` directory. To tell the Java VM where to find this file, they must be present in the `CLASSPATH` environment variable. You can set this variable in your `autoexec.bat` file, or whatever other script your operating system executes when you log in; on Windows NT and Windows 2000 you set environment variables using the `System` icon on the control panel. An alternative is to enter the classpath as part of the command that invokes the application.

I usually move all the `.jar` files I need to a directory such as `c:\jars` and then set the classpath as follows:

```
SET CLASSPATH=.;c:\jars\xalan.jar;c:\jars\xerces.jar
```

Note that it is the `.jar` files themselves that must be on the classpath, not the directory containing them.

The command to run the "hello world" transformation under Xalan is then:

```
java org.apache.xalan.xslt.Process -in hello.xml -xsl hello.xsl
```

It should give the same result as Saxon. Because you're using the default stylesheet, you could simplify this to:

```
java org.apache.xalan.xslt.Process -in hello.xml
```

If you want to direct the output to a specific file, use the `-out` option.

MSXML3

Finally, you can run the stylesheet actually within Internet Explorer.

Once you have installed MSXML3 as your default XSLT processor, you should simply be able to double-click on the `hello.xml` file, which will bring up IE5 and load `hello.xml` into the browser. IE5 reads the XML file, discovers what stylesheet is needed, loads the stylesheet, executes it to perform the transformation, and displays the resulting HTML. If you don't see the text "Hello, world!" on the screen, but just the XML file, this is because you're using the original WD-xsl interpreter that Microsoft issued with IE5, not the MSXML3 version. If you see the stylesheet displayed, this also indicates that you haven't completed the installation process correctly. Remember to run the `xmlinst.exe` program.

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 which identifies the **root 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 filestore, XPath uses «/» to represent the root of the XML content hierarchy. The DOM model calls this the Document object, but in XPath it is called the root.

```
<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 value 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 node 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 above, 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 which 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. 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. As we saw above, Saxon and Xalan have interfaces that allow you to nominate both the stylesheet and the source document to use. The same thing can also be achieved with the Microsoft XSLT product, though it requires you to write an HTML page containing some script code to control the transformation: the **<?xml-stylesheet?>** processing instruction which I used in the example above only works if you want to use the same stylesheet every time.

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 some people prefer to call it **XSL-FO** (**XSL Formatting Objects**) – for the formatting stage.

XSL Formatting 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 Formatting is outside the scope of this book. It's a big subject (the draft specification currently available is far longer than XSLT), the standard is not completely finalized, and the products that implement it are still incomplete. What's more, you're far less likely to need it than to need XSLT. XSL Formatting provides wonderful facilities to achieve high-quality typographical output of your documents. However, for most 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 Style Sheets (CSS or CSS2), which I shall return to shortly.

> *The XSL Formatting specifications, which at the time of writing are at Candidate Recommendation status, can be found at* http://www.w3.org/TR/xsl*. A Candidate Recommendation is a specification that W3C has published for final comments, which will normally become a formal Recommendation when those comments have been dealt with. The conditions for XSL Formatting Objects to become a full Recommendation also require evidence that all its constructs have been successfully implemented, and that different implementations are interoperable.*

XSLT and XPath

Halfway through the development of XSLT, 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 version 1.0 was published on the same day as XSLT, 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="sum(//book/@price) div count(//book)"/>
```

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 total of the **`price`** attributes on all the **`<book>`** elements, divided by the number of **`<book>`** elements.

The XPath specification is increasingly taking on a life of its own, separate from XSLT. For example, several DOM implementations (including Microsoft's) allow you to select nodes within a DOM tree structure using a method such as `selectNodes(XPath)`, and such a feature is being considered for the next version of the standard, DOM3. XPath is also used within other W3C specifications including XPointer and XQuery.

The separation of XPath from XSLT works reasonably well, but there are places where the split seems awkward, and there are many cases where it's difficult to know which document to read to find the answer to a particular question. For example, an XPath expression can contain a reference to a variable, but creating the variable and giving it an initial value is the job of XSLT. Another example: XPath expressions can call functions, and there is a range of standard functions defined. Those whose effect is completely freestanding, such as **`string-length()`**, are defined in the XPath specification, whereas additional functions whose behavior relies on XSLT definitions, such as **`key()`**, are defined in the XSLT specification.

Because the split is awkward, I've written this book as if XSLT+XPath were a single language. For example, all the standard functions are described together in Chapter 7. In the reference sections, I've tried to indicate where each function or other construct is defined in the original standards, but the working assumption is that you are using both languages together and you don't need to know where one stops and the other one takes over. The only downside of this approach is that if you want to use XPath on its own, for example when using the DOM `selectNodes()` method, then you need to check the description of the function in Chapter 7 to see whether it is part of the XPath core or whether it is an XSLT addition.

XSLT and Internet Explorer 5

Very soon after the first draft proposals for XSL were published, back in 1998, Microsoft shipped a partial implementation as a technology preview for use with IE4. This was subsequently replaced with a rather different implementation when IE5 came out. This second implementation, known as MSXSL, remained in the field essentially unchanged until very recently, and is still being shipped with every copy of IE5 and IE5.5, and also with Windows 2000. Unfortunately, though, Microsoft jumped the gun, and the XSLT standard changed and grew, so that when the XSLT Recommendation version 1.0 was finally published on 16 November 1999, it bore very little resemblance to the initial Microsoft product.

> **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.**

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.

So the Microsoft IE5 dialect of XSL, which I refer to as WD-xsl, is also outside the scope of this book. Please don't assume that anything in this book is relevant to the original Microsoft XSL as even where the syntax appears similar to XSLT, the meaning of the construct may be completely different.

> *You can find information about WD-xsl in the Wrox book XML IE5 Programmer's Reference, ISBN 1-861001-57-6.*

As we've already seen, Microsoft's MSXML3 is a full implementation of the XSLT 1.0 specification, but at the time of writing you have to download and install this separately. MSXML3 actually supports both the old WD-xsl dialect and the newer XSLT 1.0 language; it decides which is in use based on the namespace URI of the `<xsl:stylesheet>` element.

Microsoft has also released a converter to upgrade stylesheets from WD-xsl to XSLT 1.0. However, this isn't the end of the story, because, of course, there are millions of copies of IE5 installed that only support the old version. If you want to develop a web site that delivers XML to the browser and relies on the browser interpreting its XSLT stylesheet, you've currently got your work cut out to make sure all your users can handle it. Microsoft do provide a CAB file installer to help with the roll-out of MSXML3 — check the MSDN web site for details.

If you are using Microsoft technology on the server, there is an ISAPI extension called XSLISAPI that allows you to do the transformation in the browser where it's supported, and on the server otherwise. Unless you're in control of the browser configuration, however, **server-side** transformation of XML to HTML, driven from ASP pages or from Java servlets, is really the only practical option today for a serious project.

There's more information about Microsoft's XML products in Appendix A – but do be aware that it will become out of date very rapidly.

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 7 of the Wrox Press book* Professional XML, ISBN 1-861003-11-0.

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 (see, for example, http://www.biztalk.org) is a URN, for example `urn:java:com.icl.saxon`. 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.

- 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 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 virtually incompatible with serious use of Document Type Definitions, 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, choosing instead to wait for the replacement facility, XML Schemas.

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 in which some aspects are visible and others are not. 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 recent, and 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 this 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. I shall describe in Chapter 2 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 piece of jargon you will start hearing more often 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. XSLT isn't yet able to use this information, but it features strongly in the published requirements list for the next version, XSLT 2.0.

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 has been 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, it cannot 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, so it doesn't need to build a tree representation of the document in memory, and it can 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 (today) 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 as well as postponing the next expensive bandwidth increase.

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/.

Pre-history

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**), whose architecture in turn was influenced by the elaborate (and never implemented) work done in the ISO Open Document Architecture (ODA) standards.

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 (I pronounce it *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 towards driving equally expensive phototypesetters, so that the technology was only really taken up by the big publishing houses.

C. M. 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 style sheet language should be declarative, not procedural, and should allow style sheets 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". They may even need to vary with the attribute values of other elements: items in numbered lists will look different from items in bulleted lists.*
>
> *At the same time, the language has to be reasonably easy to interpret in a procedural way: implementing the style sheet 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 style sheets by adding new specifications to some existing (possibly standard) style sheet. This should not require copying the entire base style sheet; instead, the user should be able to store locally just the user's own changes to the standard style sheet, and they should be added in at browse time. This is particularly important to support local modifications of standard DTDs.*
>
> *Syntactically, the style sheet language must be very simple, preferably trivial to parse. One obvious possibility: formulate the style sheet language as an SGML DTD, so that each style sheet 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 style sheets 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 style sheets, 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.

What follows is 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.*

Both simple and complex designs should be possible, and the styles should be suitable for batch formatting as well as interactive applications. Existing systems should be able to support DSSSL with only minimal changes (a DSSSL parser is obviously needed).

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

A DSSSL style sheet very precisely describes a function from SGML to a flow object tree. It allows partial style sheets 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 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 21 August 1997. It can be found at http://www.w3.org/TR/NOTE-XSL.html.

There are eleven authors listed. They include five from Microsoft, three from Inso Corporation, plus Paul Grosso of ArborText, James Clark (who works for himself), and Henry Thompson of the University of Edinburgh.

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:

- *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 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 37, 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 their first technology preview five 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.

The bulk of his paper is given over to a long list of the typographical features that the language should support, following the tradition both before and since that the formatting side of the language gets a lot more column inches than the transformation side. But as XSLT fans that need not worry us because the success of standards has always been inversely proportional to their length.

What Walsh has to say on the transformation aspects of the language is particularly terse, and although he clearly had reasons for thinking these features were necessary, it's a shame that he doesn't tell us why he put these in and left others, such as sorting, grouping, and totaling, out:

- Ancestors, children, siblings, attributes, content, disjunctions, negation, enumerations, computed select based upon arbitrary query expressions.
- Arithmetic Expressions; arithmetic, simple boolean comparisons, boolean logic, substrings, string concatenation.
- Data Types: Scalar types, units of measure, Flow Objects, XML Objects.
- Side effects: No global side effects.
- Standard Procedures: The expression language should have a set of procedures that are built in to the XSL language. These are still to be identified.
- User Defined Functions: For reuse. Parameterized, but not recursive.

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.

The XSLT 1.0 Recommendation is still the current version at the time of writing, and the one to which most current products conform, although a working draft of XSLT 1.1 was published on December 12th, 2000. I've included the new features of XSLT 1.1 in this book, because they are already starting to appear in some products; but I've flagged them as XSLT 1.1 features to warn you that you won't find them in every product. In most cases, however, these features are based on extensions that many vendors provided in their XSLT 1.0 products, so you are likely to find something similar.

So let's look now at the essential characteristics of XSLT as a language.

XSLT 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 three 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, and the fact that processing is described as a set of independent pattern-matching rules.

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:

- There is already an XML parser in the browser, so it keeps the footprint small if this can be re-used.
- 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; 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.
- All 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. (Such tools are starting to become available, and I've described some of them in Appendix E.)

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

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 608.

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, given the flexibility of the expressions and patterns that are now permitted in the language. Equally, all existing XSLT processors require the whole document to be loaded into memory. However, it would be a mistake to expect too much too soon. 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 fifteen 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. At the time of writing, we are starting to see the first signs that implementors are exploiting this capability: Xalan-Java 2, for example, runs 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, which is described in Appendix E, tracks the dependencies between parts of the output document and the template rules that were used to generate them, so one can start to see the potential to regenerate the output selectively when small changes are made.

What being side-effect free means in practice is that you cannot update the value of a variable. This restriction is something you may 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, 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.

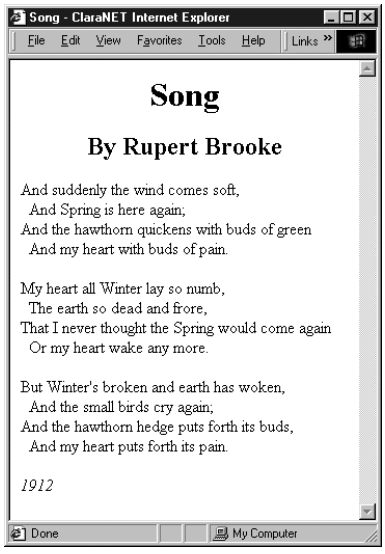
Input

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

We'll write a stylesheet such that this document appears in the browser as shown below:



Stylesheet

It starts with the standard header:

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

Now we'll 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>
      <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>
```

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 (` `). 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 **
** element to end the line.

```
   <xsl:template match="line">
      <xsl:if test="position() mod 2 = 0">&amp;#160;&amp;#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.

Beyond XSLT 1.0

I've talked about where XSLT 1.0 came from, but where is it going next?

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 it necessary to add to their products as extensions, while XSLT 2.0 would handle the more strategic requirements that needed further research.

XSLT 1.1

A working draft of XSLT 1.1 was published on December 12th 2000. It describes four 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. (All these products are described in the appendices to this book.)
- 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.
- 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).
- Support for the **`xml:base`** construct, a late addition to the XML core standards, which allows an XML document to specify the Base URI that should be used to resolve any relative URIs contained within the document.

At the time of writing, XSLT 1.1 is still a working draft, and there is no official statement about when it is likely to be finalized. There have been suggestions that it would be a good idea to explore in more depth the implications of the XSLT 2.0 requirements before freezing the XSLT 1.1 spec, to avoid constraining the way the XSLT 2.0 requirements are met. One can also speculate that some vendors feel the market is still only really beginning to take XSLT 1.0 on board, and that in reality users want stability more than they want enhancements.

Despite this, we've decided to go ahead and include definitions of XSLT 1.1 facilities in this book. In doing this, we're taking a calculated gamble that the spec will be ratified in very similar form to its published draft, and that it will be implemented in products during the course of the year (2001) – there are signs that is already starting to happen. Meanwhile, we've taken care to flag XSLT 1.1 enhancements for what they are, so that you can check the specifications in this book against the latest W3C recommendations and the facilities offered by your chosen product before using them.

As we went to press, the W3C had decided that XSLT 1.1 would not be progressed further than working draft stage, but would instead be used as a baseline for developing XSLT 2 (see overleaf)

Throughout this book, references to "the XSLT 1.1 working draft" mean the draft dated December 12th 2000.

XSLT 2.0 and XPath 2.0

The W3C XSL Working Group, working closely with other working groups, published the requirements for XSLT 2.0 and XPath 2.0 on 14 February 2001; you can find the documents at the following URLs:

http://www.w3.org/TR/xslt20req
http://www.w3.org/TR/xpath20req

The reason W3C working groups publish requirements documents is to try and get user input into the standardization process, so it would be a mistake to regard these lists of proposed enhancements as definitive. Nevertheless, they give a good indication of the thinking. 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.
- Addition of features developed by the XML Query working group. XQuery is a specification for an SQL-like language to query a collection of XML documents and return the results, also as an XML document. The first public working draft was released on 15 February 2001 (see http://www.w3.org/TR/2001/WD-xquery-20010215/), but it is based on ideas from a number of previous proposals, notably a language called Quilt, the work of Don Chamberlin, Jonathan Robie, and Daniela Florescu. Although XQuery is designed for use in a different context, there are many overlaps in functionality with XSLT and XPath. W3C likes to keep its different standards as coherent as possible, and there are many features in the proposed XQuery specification which could usefully be added to the XPath specification to keep them in step with each other. There's a heavy dose of compromise involved here, on both sides, because some of the improvements the XML Query group has identified look great in theory, but are difficult to retrofit into a language that is already in widespread use.
- Features designed to exploit and integrate with XML Schema. The W3C XML Schema specification is now at an advanced stage (it became a Candidate Recommendation on 20 October 2000), and implementations are starting to appear in products. The schema acts as a replacement for the DTD, giving a much richer way of specifying the data types of the elements and attributes that may appear in a document. The thinking is that if the schema specifies 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. Achieving this without turning the current XPath language upside down (at present it is a very loosely typed language) is going to be a major challenge, and for this reason alone I would expect that development of XPath 2.0 may well take a significant length of time.

Where to use XSLT

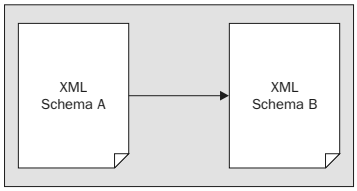
In the final section of this chapter I shall try and identify what tasks XSLT is good at, and by implication, tasks for which a different tool would be more suitable. I shall also look at alternative ways of using XSLT within the overall architecture of your application.

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

Data Conversion Applications

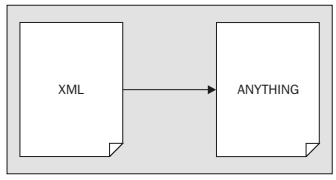
Data conversion is not something that will go away just because XML has been invented. Even though an increasing number of data transfers between organizations or between applications within an organization are likely to be encoded in XML, there will still be different data models, different ways of 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 XML data set, then XSLT is an obvious choice.

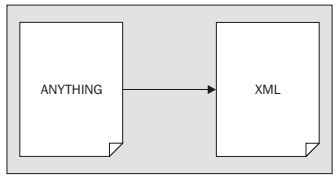


It can be used for extracting the data selectively, reordering it, turning attributes into elements or vice versa, or any number of similar tasks. It can also be used simply for validating the data. As a language, XSLT is best at manipulating the structure of the information as distinct from its content: it's a good language for turning rows into columns, but for string handling (for example removing any text that appears between square brackets) it's rather laborious compared with a language like JavaScript or Perl that supports regular expressions. However, you can always tackle these problems by invoking procedures written in other languages, such as Java or Javascript, from within the stylesheet.

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



Perhaps more surprising is that XSLT can often be useful to convert from non-XML formats into XML or something else:



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 10, page 656, 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.

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 as much as ten times the original data size, so in practice, the limit on data size for an XSLT conversion is a few megabytes. 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 might be better off, for example, loading the data into a relational or object database, 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.

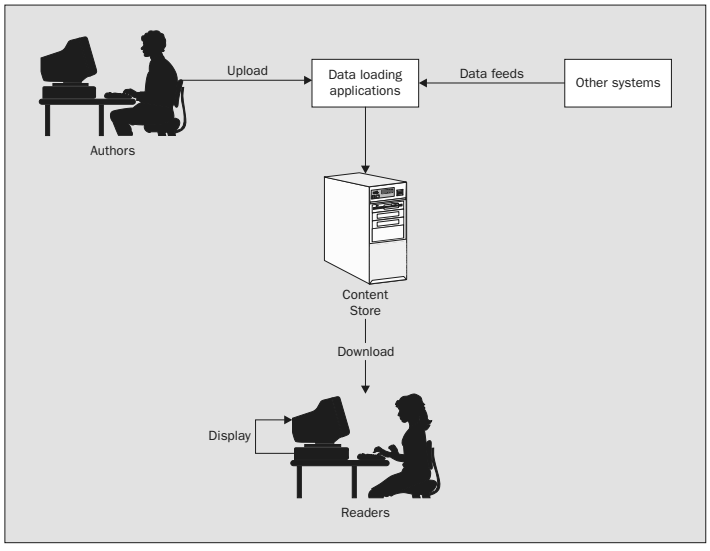
XML was designed to enable information to be held independently of the way it is presented, which sometimes leads people into the fallacy of thinking that using XML for presentation details is somehow bad. Far from it: if you were designing a new format for downloading fonts to a printer today, you would 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 complete, and will probably take even longer to implement in its entirety. 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 how quick the take-up of XHTML will be remains to be seen.

When to do the Conversion?

The process of publishing information to a user is illustrated in the diagram below:



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 use the XSL ISAPI extension available from http://msdn.microsoft.com/xml, or if you want more application control, you can invoke the transformation from script on ASP 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 once XSLT becomes widely available within browsers, but that is still some way off. Even when this is done, there may still be a need for some server-side processing to deliver the XML in manageable chunks, and to 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 pre-constructed HTML page.

Summary

This introductory chapter described the whys and wherefores of XSLT. It tried to answer questions such as:

- 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'll 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.