CHAPTER 8

XML Schemas

MOST OF US WHO ARE INVOLVED in XML development are all too familiar with using Document Type Definition (DTD) to enforce the structure of XML documents. Basically, a DTD is a schema that defines the content model of an XML document. However, DTD has some shortcomings.

- First, a DTD is not an XML document. It does not allow DTD to be easily processed by XML parsers.

- Second, DTD does not support data typing. All content is treated as strings. This lack of support for data types means that additional workload must be placed on applications handling the XML document to verify the content of the XML document.

Consider the following code in Listing 8-1:

Listing 8-1. XML document containing an internal DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Product [
   <!ELEMENT Name (#PCDATA)>
   <!ELEMENT Price (#PCDATA)>
   <!ELEMENT Product (Name, Price)>
]>
<Product>
   <Name>LCD Monitor</Name>
   <Price>500</Price>
</Product>
```

There is no way a DTD can validate that the price is a numeric value; hence <Price> will be validated as correct. The following document in Listing 8-2 will still be validated correctly:

Listing 8-2. DTD does not support data types

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Product [
```

```
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Price (#PCDATA)>
<!ELEMENT Product (Name, Price)>
]>
<Product>
        <Name>LCD Monitor</Name>
        <Price>xyz</Price>
</Product>
```

Due to all of these reasons, DTD is slowly being phased out. In the meantime, several alternate solutions have been proposed:

- Microsoft Extensible Data Reduced (XDR) Schema Language (XDR is discussed in Appendix C)
- Document Content Description (DCD)
- XML Schema Definition Language (XSD)

An ideal XML schema should have the following features:

- Supports data types
- Uses XML syntax
- Able to be processed by XML processors
- Supports elements of global and local scope (for example, the use of identical element names)

The W3C's XML Schema Definition Language (XSD) was formally approved as a Recommendation on May 2, 2001. In this chapter, we discuss the W3C's XSD.

CAUTION *Do not confuse the term XML schema with W3C's XML Schema. An XML schema is basically a set of rules that defines the structure of an XML document. The W3C's XML Schema is also one that has been drafted and recommended for adoption by vendors to enhance the functionality and interoperability of the Web.*

XML Schema and XDR Schema

Before the W3C's XSD was approved as a Recommendation, Microsoft implemented an interim solution to XSD—the Extensible Data Reduced (XDR) Schema Language. XDR was supported in many Microsoft products like BizTalk server, Office 2000, Internet Explorer 5.0 and SQL Server. However, Microsoft is expected to support the W3C's XSD in its future release of its products.

CAUTION *The MSXML release 4 supports the XSD and XDR languages. MSXML3, however, supports only XDR.*

The W3C's XML Schema specification can be found on W3C's Web site. There are three parts to the specification (please refer to the end of this chapter for links to more information on the W3C's XML Schema):

- XML Schema Part 0: Primer
- XML Schema Part 1: Structures
- XML Schema Part 2: Data types

The *XML Schema Part 0: Primer* is an easy to read description of the XML Schema**.** *XML Schema Part 1: Structures* and *XML Schema Part 2: Data types* provide the complete description of the XML Schema language.

First Look at XML Schema

Before we dive deep into the details of XML Schema, let us illustrate XSD using a simple example. Consider the XML document that is shown in Listing 8-3.

Listing 8-3. Magazine.xml

```
<?xml version="1.0"?>
<Magazines>
    <Magazine Price="5.95">
       <Title>SQL Server magazine</Title>
       <Publisher>Penton</Publisher>
    </Magazine>
```

```
    <Magazine Price="4.95">
       <Title>Web Techniques</Title>
       <Publisher>CMP</Publisher>
    </Magazine>
    <Magazine Price="5.00">
       <Title>Wireless Business and Technology</Title>
       <Publisher>Sys-con media</Publisher>
    </Magazine>
    <Magazine Price="5.95">
       <Title>MSDN</Title>
       <Publisher>CMP</Publisher>
    </Magazine>
</Magazines>
```

The corresponding XML Schema for the preceding XML document is shown in Listing 8-4.

Listing 8-4. Magazine.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="Magazines" type="MagazineType"/>
   <xsd:complexType name="MagazineType">
      <xsd:sequence>
         <xsd:element name="Magazine" type="MagazineDetails"
                      minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="MagazineDetails">
      <xsd:sequence>
         <xsd:element name="Title" type="xsd:string"/>
         <xsd:element name="Publisher" type="xsd:string"/>
      </xsd:sequence>
      <xsd:attribute name="Price" type="xsd:float"/>
   </xsd:complexType>
</xsd:schema>
```

Note that the Schema file is also an XML document, which has the advantage of allowing XML parsers to process it. You might also have noticed the use of the "xsd" prefix. The use of the "xsd" prefix is similar to using the "xsl" prefix for XSLT stylesheets. Although using "xsd" as a prefix is the norm, you can use any prefix you want by modifying the namespace declaration.

Validating Against a Schema Using MSXML4

MSXML3 does not support the W3C Schema. At the time of writing, the MSXML4 has just been released. The MSXML4 supports the W3C Schema. In this section, we discuss how you can use the MSXML4 to validate an XML document against a Schema. We show our example in Visual Basic 6. For examples using other languages, please refer to the documentation provided with MSXML4.

We will use the examples that are shown in Listings 8-3 and 8-4. To demonstrate validating documents with namespaces, we have added a namespace in the magazine.xml document.

```
<?xml version="1.0"?>
<m:Magazines xmlns:m="urn:Mags">
    <Magazine Price="5.95">
       <Title>SQL Server magazine</Title>
       . . .
```

Listing 8-5 shows the code for XML Schema validation.

Listing 8-5. Validating an XML document against an XML Schema using VB6

```
Private Sub Form_Load()
  '--Create a schema cache--
  Dim xmlschema As New MSXML2.XMLSchemaCache40
  '--and add magazines.xsd to it--
  xmlschema.Add "urn:Mags", App.Path & "\magazine.xsd"
  '--Create an XML DOMDocument object--
  Dim xmldom As New MSXML2.DOMDocument40

  '--Assign the schema cache to the DOM document schemas collection.--
  Set xmldom.schemas = xmlschema
  '--Load magazines.xml document--
  xmldom.async = False
  xmldom.Load App.Path & "\magazine.xml"

  '--check for error--
  If xmldom.parseError.errorCode <> 0 Then
     MsgBox xmldom.parseError.errorCode & " " & xmldom.parseError.reason
  Else
     MsgBox "No Error"
  End If
End Sub
```

If the XML document does not conform to the rules that are specified in the Schema, an error message would be displayed. For example, if the price contains a nonnumeric, such as

```
<Magazine Price="A.95">
```

the XML document would display an error message:

"-1072897661 The attribute: 'Price' has an invalid value according to its data type."

Dissecting the Schema

Let's now dissect the Schema and see how it defines the model of our XML document.

We start the XML Schema by first declaring the <xsd:schema> root element. Also note the XSD namespace that we use. This namespace conforms to the latest W3C's recommendation of XML Schema on May 2, 2001.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

Next, we define the <Magazines> root element with the <xsd:element> element. The `type` attribute indicates that the <Magazines> element belongs to a complex type (more on this shortly) named `MagazineType`. This is very similar to declaring a variable to be of a certain data type in a programming language.

```
<xsd:element name="Magazines" type="MagazineType"/>
```

We then go on to declare the complex type MagazineType. All elements that contain attributes and child elements are known as `complexType`. Within the <xsd:complexType> element, we use the <xsd:sequence> element to list the sequence in which child elements must appear.

```
<xsd:complexType name="MagazineType">
   <xsd:sequence>
      <xsd:element name="Magazine" type="MagazineDetails" minOccurs="0"
                   maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
```

So the preceding declaration means that <Magazines> contains child element(s) called <Magazine>. The `minOccurs` attribute indicates the minimum

number of time (0 in this case; i.e., this element is optional) this element must appear while the `maxOccurs` attribute indicates the maximum number of time (any number in this case) this element can appear.

Table 8-1 lists some of the values possible for the `minOccurs` and `maxOccurs` attributes.

Table 8-1. Possible Values for the `minOccurs` *and* `maxOccurs` *Attributes*

MINOCCURS	MAXOCCURS	DESCRIPTION
0	1	Element is optional, but only one at most can be present
1	1	One and only one occurrence of the element
0	Unbounded	Can have any number of occurrences of the element
1	Unbounded	Element must appear at least once
4	9	Element must appear at least four times, subject to a maximum of nine

The <Magazine> element belongs to another complex type called `MagazineDetails`.

```
   <xsd:complexType name="MagazineDetails">
      <xsd:sequence>
         <xsd:element name="Title" type="xsd:string"/>
         <xsd:element name="Publisher" type="xsd:string"/>
      </xsd:sequence>
      <xsd:attribute name="Price" type="xsd:float"/>
   </xsd:complexType>
</xsd:schema>
```

The declaration for the `MagazineDetails` complex type is very similar to the previous one, except for the declaration of the `Price` attribute. In addition, the <Title> and <Publisher> elements are known as *simple type*, as they contain neither child elements nor attributes. The `type` attribute in <xsd:element> and <xsd:attribute> allows data types to be specified. This is one huge advantage XML Schema has over DTD.

Based on the first look at XML Schema, we can see that an XML Schema is itself an XML document, unlike a DTD (which has its own syntax for describing content model). An interesting thing (or irony!) to note is that an XML Schema has its own DTD! The official location for the DTD can be found at `http://www.w3.org/2001/XMLSchema.dtd`.

Hey! Who says DTD is dead?

Rearranging the Schema

The previous example takes the top-down approach of creating a schema. That is, examine the XML document from top to bottom and as elements and attributes are encountered, declare them in the schema. This approach, though simple to use, is not very suitable for large documents. Schemas created using this approach are often complicated and difficult to understand.

To solve this problem, let's rewrite our schema (Listing 8-6).

Listing 8-6. Magazine1.xsd

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:element name="Title" type="xsd:string"/>
<xsd:element name="Publisher" type="xsd:string"/>
<xsd:attribute name="Price" type="xsd:float"/>

<xsd:element name="Magazine">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref="Title"/>
         <xsd:element ref="Publisher"/>
      </xsd:sequence>
      <xsd:attribute ref="Price"/>
   </xsd:complexType>
</xsd:element>

<xsd:element name="Magazines">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref="Magazine" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>

</xsd:schema>
```

Let's now dissect the schema.

First, we declare all the simple type elements first:

```
<xsd:element name="Title" type="xsd:string"/>
<xsd:element name="Publisher" type="xsd:string"/>
<xsd:attribute name="Price" type="xsd:float"/>
```

We then declare the <Magazine> element as a complex type. Note that we use the `ref` attribute to reference the simple type elements declared earlier.

```
<xsd:element name="Magazine">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref="Title"/>
         <xsd:element ref="Publisher"/>
      </xsd:sequence>
      <xsd:attribute ref="Price"/>
   </xsd:complexType>
</xsd:element>
```

And finally, we declare the <Magazines> root element:

```
<xsd:element name="Magazines">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref="Magazine" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

Compared to the last example, this schema is much more readable and organized. We first start with the most basic elements and progressively declare more complex elements.

Sequencing

Previous examples have illustrated ordering the elements using the <xsd:sequence> element. The <xsd:sequence> element enforces the order in which all the elements appear. Sometimes, you may not want to enforce the order. To do so, use the <xsd:all> element.

```
      <xsd:all>
         <xsd:element ref="Title"/>
         <xsd:element ref="Publisher"/>
      </xsd:all>
```

For example, the preceding <xsd:all> element allows either the <Title> or <Publisher> element to appear first.

XML Schema Data Types

One of the strengths of the XSD is its support of data types. Data types in XSD can be classified as:

- Primitive data types
- Derived data types

Let's examine them in more detail.

Primitive Data Types

Primitive data types are the basic types in which other data types derive on. Some of the primitive data types are:

- String: Characters strings in XML
- Boolean: True or false, 1 or 0
- Float: Corresponds to the IEEE single-precision 32-bit floating point type
- Double: Corresponds to IEEE double-precision 64-bit floating point type
- Decimal: Represents arbitrary precision decimal numbers

Derived Data Types

As the name implies, derived data types are types that derive from the primitive data types. Examples of derived data types are:

- integer: Represents a sequence of digits with optional + or – sign. It is derived from the `decimal` type.
- long: Represents a range of numbers (-9223372036854775808 to 9223372036854775807). It is derived from `integer`.
- int: Represents a range of numbers (from -2147483648 to 2147483647). It is derived from `long`.

As you can see, a derived data type can both derive from a primitive data type or it can derive from another derived data type.

XML Schema Simple Types

In the earlier example (Listing 8-6), we saw the declaration of simple types using something like this:

```
<xsd:element name="Title" type="xsd:string"/>
```

And to make use of that simple type, we simply reference the simple type using the `ref` attribute:

```
<xsd:element ref="Title"/>
```

However, there are times when you want to use the same simple type but with a different tag name, for example "BookTitle." In this case, it is useful to define the "Title" element as a simple type using the <simpleType> element:

```
<xsd:simpleType name="TitleType">
   <xsd:restriction base="xsd:string"/>
</xsd:simpleType>
```

To make use of the simple type but with a different tag name, you can now reference it like this:

```
<xsd:element name="BookTitle" type="TitleType"/>
```

Using the <xsd:simpleType> element allows simple types to be reused.

Restricting Range

In the first example (shown previously in Listing 8-6), we declare the price of a magazine as a floating-point number:

```
<xsd:attribute name="Price" type="xsd:float"/>
```

However, we want to further impose some restrictions on the range of values that can be permitted. For example, the price must be greater than zero and not exceed $40 (it is not very likely to have such an expensive magazine). In this case, we need to further restrict the "xsd:float" data type. Consider the example:

```
<xsd:simpleType name="priceType">
   <xsd:restriction base="xsd:float">
      <xsd:minExclusive value="0" />
      <xsd:maxInclusive value="40"/>
   </xsd:restriction>
</xsd:simpleType>
```

Here we use the <xsd:simpleType> element to define a simple type called `priceType`. Within this simple type, we impose a restriction on the range of values permissible using the <xsd:minExclusive> and <xsd:maxInclusive> `facet` elements. A facet element defines constraints for a data type.

We also need to change the declaration for the <Price> element to use the new type:

```
<xsd:attribute name="Price" type="priceType"/>
```

With this restriction, our magazine price must be more than 0 (minExclusive) and less than or equal to 40 (minInclusive).

Enumeration

We might also want to restrict the list of publishers to three. In this case, we define another simple type element, like this:

```
<xsd:simpleType name="publisherType">
   <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Penton"/>
      <xsd:enumeration value="CMP"/>
      <xsd:enumeration value="Sys-con media"/>
   </xsd:restriction>
</xsd:simpleType>
```

We used the <xsd:enumeration> element to set the allowable values for the "publisherType" element. Just like before, we need to modify the declaration for the <Publisher> element to use the new publisherType:

```
<xsd:element name="Publisher" type="publisherType"/>
```

So in this case, Publishers are limited to three—Penton, CMP, and Sys-con media.

Deriving New Simple Types

Simple types can be extended from another simple type. The following example illustrates this.

Suppose we have the AgeType derived from the short data type, with a minimum value of 0 and maximum value of 150

```
<xsd:simpleType name="AgeType">
   <xsd:restriction base="xsd:short">
      <xsd:minInclusive value="0"/>
      <xsd:maxInclusive value="150"/>
   </xsd:restriction>
</xsd:simpleType>
```

We want to have another type called AgeGroup1, which extends on the AgeType simple type. We shall impose another restriction, which states that the maximum value for this type cannot exceed 12. Here is the definition:

```
<xsd:simpleType name="AgeGroup1">
   <xsd:restriction base="AgeType">
   <xsd:maxInclusive value="12"/>
   </xsd:restriction>
</xsd:simpleType>
```

As usual, change the `type` attribute to AgeGroup1:

```
<xsd:element name="Age" type="AgeGroup1"/>
```

So the following text content for <Age> is invalid:

```
<Age>13</Age>
```

since it is more than 12.

XML Schema Complex Types

We have seen the use of complexType in the earlier sections. We shall now take a closer look.

Consider the following XML document for storing shipping and ordering information (Listing 8-7):

Listing 8-7. XML document for storing shipping and ordering information

```
<?xml version="1.0" encoding="UTF-8"?>
<OrderInfo>
   <ShipTo>
      <Name>Wei Meng Lee</Name>
      <Phone>065-4606872</Phone>
      <Address>
         <Street1>Ngee Ann Polytechnic</Street1>
         <Street2>535 Clementi Road</Street2>
         <Street3>Singapore</Street3>
         <Postal>599489</Postal>
      </Address>
   </ShipTo>
   <BillTo>
      <Name>Sales @ Apress</Name>
      <Phone>510-5495930</Phone>
      <Address>
         <Street1>Apress L.P.</Street1>
         <Street2>901 Grayson Street Suite 204</Street2>
         <Street3>Berkeley, CA</Street3>
         <Postal>94710</Postal>
      </Address>
   </BillTo>
</OrderInfo>
```

Obviously, there are a couple of repeating elements. Both the <ShipTo> and <BillTo> elements contain the <Name>, <Phone>, and <Address> elements. If you are familiar with object-oriented programming, then the notion of creating classes and deriving objects from them will come to mind. The W3C XML Schema allows us to define data types by giving the simple type or complex type a `name` attribute. The XML Schema for the preceding XML document appears in Listing 8-8 as follows:

Listing 8-8. Schema for the XML document in Listing 8-7

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema
    elementFormDefault="qualified">

<xsd:simpleType name="StreetType">
   <xsd:restriction base="xsd:string">
      <xsd:maxLength value="30"/>
```

```
   </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="PostalType">
   <xsd:restriction base="xsd:string">
      <xsd:maxLength value="6"/>
   </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="AddressType">
   <xsd:sequence>
      <xsd:element name="Street1" type="StreetType"/>
      <xsd:element name="Street2" type="StreetType"/>
      <xsd:element name="Street3" type="StreetType"/>
      <xsd:element name="Postal" type="PostalType"/>
   </xsd:sequence>
</xsd:complexType>

<xsd:simpleType name="PhoneType">
   <xsd:restriction base="xsd:string">
      <xsd:pattern value="[0-9]{3}-[0-9]{7}"/>
   </xsd:restriction>
</xsd:simpleType>

<xsd:complexType  name="Contact">
   <xsd:sequence>
      <xsd:element name="Name"/>
      <xsd:element name="Phone" type="PhoneType"/>
      <xsd:element name="Address" type="AddressType"/>
   </xsd:sequence>
</xsd:complexType>

<xsd:element name="OrderInfo">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element name="ShipTo" type="Contact"/>
         <xsd:element name="BillTo" type="Contact"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
</xsd:schema>
```

The first thing to note is that we have an additional attribute, `name`, in the <xsd:simpleType> element:

```
<xsd:simpleType name="StreetType">
   <xsd:restriction base="xsd:string">
      <xsd:maxLength value="30"/>
   </xsd:restriction>
</xsd:simpleType>
```

The preceding simple type element defines a data type called "StreetType." This is analogous to defining a class in an object-oriented programming language. Next comes the restriction element that expresses that this data type is derived from "xsd:string." Within the restriction element is the facet. The facet

```
<xsd:maxLength value="30"/>
```

states that the maximum length of this element is 30 characters.

Besides the simple type definition, we also have complex type definitions:

```
<xsd:complexType name="AddressType">
   <xsd:sequence>
      <xsd:element name="Street1" type="StreetType"/>
      <xsd:element name="Street2" type="StreetType"/>
      <xsd:element name="Street3" type="StreetType"/>
      <xsd:element name="Postal" type="PostalType"/>
   </xsd:sequence>
</xsd:complexType>
```

Here, the "AddressType" element is defined to contain four elements, three of which reference the "StreetType" type and one of which references the "PostalType" type. Also note that the <xsd:complexType> element has the `name` attribute. The inclusion of this `name` attribute allows this type to be reused and is known as a *named data type*. Complex type elements that do not contain the `name` attribute are known as *anonymous data types*.

The "PhoneType" type contains an interesting facet:

```
<xsd:simpleType name="PhoneType">
   <xsd:restriction base="xsd:string">
      <xsd:pattern value="[0-9]{3}-[0-9]{7}"/>
   </xsd:restriction>
</xsd:simpleType>
```

The <xsd:pattern> element specifies the specific pattern that the element must contain. The pattern is specified in the `value` attribute. The `value` attribute contains a *regular expression*. In our example, we specify that the phone number must start with three digits followed by a - and then followed by another seven digits.

To make use of the new data types, simply reference them using the `type` attribute as illustrated in the example:

```
<xsd:complexType  name="Contact">
   <xsd:sequence>
      <xsd:element name="Name"/>
      <xsd:element name="Phone" type="PhoneType"/>
      <xsd:element name="Address" type="AddressType"/>
   </xsd:sequence>
</xsd:complexType>
```

Mixed Content

How do you define mixed content in an XML document? Consider the following code that is shown in Listing 8-9:

Listing 8-9. Mixed content in an XML document

```
<?xml version="1.0"?>
<Synopsis>
   <Author>Eric Gunnerson</Author>, the test lead for and member of Microsoft's
 C# design team, has written a comprehensive <Topic>C#</Topic> tutorial for
 programmers to help them get up to speed.
</Synopsis>
```

Here the <Synopsis> element contains both child elements as well as text. Within the <Synopsis> element are <Author> and <Topic>. The Schema for the preceding XML document is shown in Listing 8-10.

Listing 8-10. Schema for the XML document in Listing 8-9

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Author" type="xsd:string"/>
   <xsd:element name="Topic" type="xsd:string"/>
   <xsd:element name="Synopsis">
      <xsd:complexType mixed="true">
```

```
            <xsd:choice minOccurs="0" maxOccurs="unbounded">
               <xsd:element ref="Author"/>
               <xsd:element ref="Topic"/>
            </xsd:choice>
         </xsd:complexType>
      </xsd:element>
</xsd:schema>
```

To indicate that the <Synopsis> element is of mixed content, we add the `mixed` attribute to the <xsd:complexType> element.

```
      <xsd:complexType mixed="true">
```

We then declare the elements that can appear within the <Synopsis> element.

```
<xsd:choice minOccurs="0" maxOccurs="unbounded">
   <xsd:element ref="Author"/>
   <xsd:element ref="Topic"/>
</xsd:choice>
```

Note that we use the <xsd:choice> element to encapsulate the two elements. The <xsd:choice> element allows one and only one of the elements contained within it to be present in the XML instance document. In our example, it essentially means that either <Author> or <Topic> can appear, in any sequence, and with unlimited occurrences. The following is valid:

```
<Author>Eric Gunnerson</Author>, the test lead for and member of
Microsoft's C# design team, has written a comprehensive <Topic>C#</Topic>
tutorial for programmers to help them get up to speed. <Author>Andrew
Troelsen</Author> has also written a book on C#.
```

You might be tempted to use the <xsd:sequence> element, for example:

```
<xsd:sequence minOccurs="0" maxOccurs="unbounded">
   <xsd:element ref="Author"/>
   <xsd:element ref="Topic"/>
</xsd:sequence>
```

However, if you use the <xsd:sequence> element, then both the <Author> and <Topic> elements must appear in pairs, like this:

```
<Author>Eric Gunnerson</Author>, the test lead for and member of
Microsoft's C# design team, has written a comprehensive <Topic>C#</Topic>
```

```
tutorial for programmers to help them get up to speed. <Author>Daniel
Appleman</Author> is the president of Desaware, Inc., a developer of add-on
products and components for Microsoft Visual Development Tools including
<Topic>Visual Basic</Topic>.
```

Empty Elements

For elements that do not have any content, that is, empty elements, you simply define a <xsd:complexType> within the element definition like this:

```
        <xsd:element name="Empty">
      <xsd:complexType/>
   </xsd:element>
```

The schema defines an empty element called <Empty/>.

Defining and Declaring Elements and Attributes

Till this point, we have frequently used the terms *declare* and *define*. But what is the distinction between them? A *declaration* describes the structure of an XML document. A *definition* creates new types to be used by other elements.

The following example illustrates a definition:

```
<xsd:simpleType name="PhoneType">
   <xsd:restriction base="xsd:string">
      <xsd:pattern value="[0-9]{3}-[0-9]{7}"/>
   </xsd:restriction>
</xsd:simpleType>
```

Here, we are defining a new type, which is based on the `string` primitive data type.

The following is a declaration:

```
   <xsd:element name="Phone" type="PhoneType"/>
```

We are declaring an element called "Phone," which is of the "PhoneType" type.

Groupings

W3C Schema allows elements and attributes to be grouped so that they can be used as a "container." Consider the following example of an attributes group:

```
<?xml version="1.0" encoding="UTF-8"?>
<Library>
   <Magazine Publisher="Penton" Price="5.95" PubDate="2000-12-01">
      <Name>SQL Magazine</Name>
   </Magazine>
   <Book Publisher="Apress" Price="32.95" PubDate="2001-09-25">
      <Name>Microsoft XML Programming</Name>
   </Book>
</Library>
```

In the preceding example, both the <Magazine> and <Book> elements contain the same set of attributes. Rather than defining the attributes twice, it is neater to group them, as shown in the following Schema (Listing 8-11):

Listing 8-11. Grouping attributes in a Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:attributeGroup name="AttributeGroup">
   <xsd:attribute name="Publisher" type="xsd:string"/>
   <xsd:attribute name="Price" type="xsd:number"/>
   <xsd:attribute name="PubDate" type="xsd:date"/>
</xsd:attributeGroup>

<xsd:element name="Magazine">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element name="Name"/>
      </xsd:sequence>
      <xsd:attributeGroup ref="AttributeGroup"/>
   </xsd:complexType>
</xsd:element>

<xsd:element name="Book">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element name="Name"/>
```

```
      </xsd:sequence>
      <xsd:attributeGroup ref="AttributeGroup"/>
   </xsd:complexType>
</xsd:element>

<xsd:element name="Library">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref="Magazine"/>
         <xsd:element ref="Book"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>

</xsd:schema>
```

We first define an attribute group using the <xsd:attributeGroup> element. Within it, we declare all the attributes belonging to the group.

```
<xsd:attributeGroup name="AttributeGroup">
   <xsd:attribute name="Publisher" type="xsd:string"/>
   <xsd:attribute name="Price" type="xsd:number"/>
   <xsd:attribute name="PubDate" type="xsd:date"/>
</xsd:attributeGroup>
```

To make use of the attribute group, we use the same <xsd:attributeGroup> element with the `ref` attribute.

```
<xsd:element name="Magazine">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element name="Name"/>
      </xsd:sequence>
      <xsd:attributeGroup ref="AttributeGroup"/>
   </xsd:complexType>
</xsd:element>
```

Besides using the <xsd:attributeGroup> element to group attributes, you can also group elements using the <xsd:group> element. Its usage is similar to that of using the <xsd:attributeGroup> element, except that for element group definition you need to use the <xsd:sequence> element to enforce the order of the elements.

Linking Schemas and Redefining Definitions

Schemas might be shared among various people. To facilitate sharing, you can include an external Schema by using the <xsd:include> element. For example, we can include another Schema by using:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:include schemaLocation="Address.xsd"/>
   <xsd:complexType name="AddressType">
      <xsd:sequence>
         <xsd:element name="Street1" type="StreetType"/>
         <xsd:element name="Street2" type="StreetType"/>
      <xsd:element name="Street3" type="StreetType"/>
         <xsd:element name="Postal" type="PostalType"/>
      </xsd:sequence>
   </xsd:complexType>
   . . .
```

In this example, the definition for `StreetType` and `PostalType` are defined in the XSD file Address.xsd.

You can also redefine a definition from the loaded Schema. For example, you might want to redefine the PostalType to contain at most four characters:

```
<?xml version="1.0" encoding="UTF-8"?>
   <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <xsd:redefine schemaLocation="Address.xsd">
         <xsd:simpleType name="PostalType">
            <xsd:restriction base="xsd:string">
               <xsd:maxLength value="4"/>
            </xsd:restriction>
         </xsd:simpleType>
      </xsd:redefine>
   <xsd:complexType name="AddressType">
      <xsd:sequence>
         <xsd:element name="Street1" type="StreetType"/>
         <xsd:element name="Street2" type="StreetType"/>
      <xsd:element name="Street3" type="StreetType"/>
         <xsd:element name="Postal" type="PostalType"/>
      </xsd:sequence>
   </xsd:complexType>
   . . .
```

Here, we use the <xsd:redefine> element to change the definition of the "PostalType" type.

Documenting the Schema Using <annotation>

Documentation is an often-overlooked area in software development. Until this point, we have not really talked about documenting your Schema. To document your Schema, you may use the <!-- and --> pair, since an XSD document is actually an XML document. For example:

```
<!-- Declaring a complex type -->
<xsd:complexType name="MagazineDetails">
   <xsd:sequence>
      <xsd:element name="Title" type="xsd:string"/>
      <xsd:element name="Publisher" type="xsd:string"/>
   </xsd:sequence>
   <xsd:attribute name="Price" type="xsd:float"/>
</xsd:complexType>
```

However, applications processing your XML Schema might not be able to access the comment, since an XML parser can choose to ignore the comments in an XML document. In that case, the comment is not accessible for further processing (say, for documentation purposes). To document your Schema, consider the following example:

```
...
<xsd:annotation>
   <xsd:documentation xml:lang="en">
       Declaring a complex type
   </xsd:documentation>
   <xsd:appinfo>
       Declaration for the Synopsis element
   </xsd:appinfo>
</xsd:annotation>
<xsd:complexType name="MagazineDetails">
  <xsd:sequence>
     <xsd:element name="Title" type="xsd:string"/>
     <xsd:element name="Publisher" type="xsd:string"/>
  </xsd:sequence>
  <xsd:attribute name="Price" type="xsd:float"/>
</xsd:complexType>
...
```

The <xsd:annotation> element contains two child elements:

- <xsd:appinfo>
- <xsd:documentation>

The <xsd:appinfo> element is for the XML parsers to process the comments while the <xsd:documentation> element allows you to add in comments for human consumption.

Tools for Validating XML Schemas

At the time of writing, there aren't many tools out in the market that are able to validate an XML document against an XML Schema. Most tools, if available, do not support the latest XSD recommendation from W3C. Apart from using the MSXML Preview Release 4, you can use XMLSpy version 4.0 from Altova (`www.xmlspy.com`). XMLSpy is an Integrated Development Environment for XML that includes all major aspects of XML in one powerful and easy-to-use product. XMLSpy features support for:

- XML document editing and validation
- W3C's XSD Recommendation
- XSLT

In this section, we briefly explain how you can validate an XML document against an XML Schema using XMLSpy 4.0.

Creating the XML Schema

A common question that is often asked is which comes first—XML document or XML Schema? Well, creating XML documents and Schema can be likened to creating databases. In creating database applications, the first thing to do is to design the database before populating the database. Similarly, it is natural that you first design your XML Schema before you create your XML document.

XMLSpy provides an IDE to create XML Schemas easily. Figure 8-1 shows an XML Schema that is created in XMLSpy.

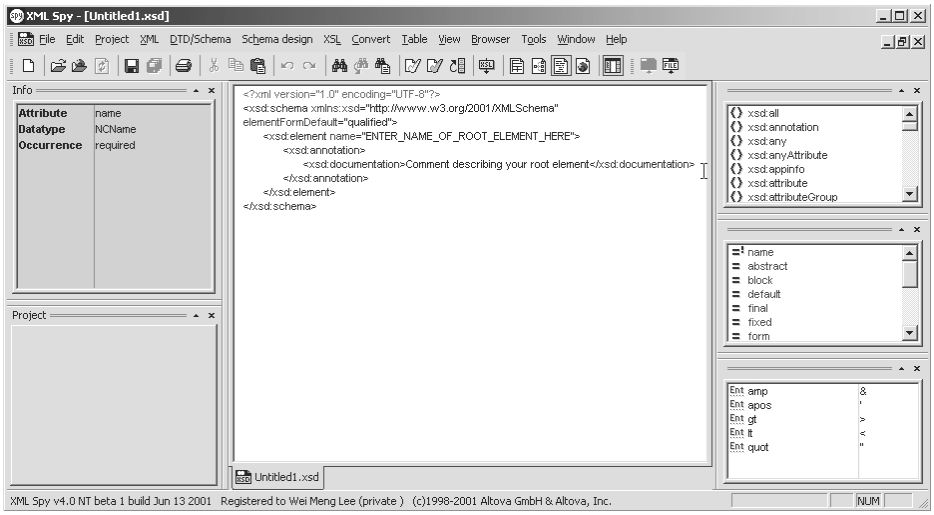

Figure 8-1. Creating an XML Schema in XMLSpy

We won't be going into details on how to create an XML Schema using XMLSpy. For more information, you can refer to the documentation that comes with XMLSpy.

Creating an XML Document Based on an XML Schema

Once the XML Schema is created, you can create XML documents based on that Schema. When you create a new XML document, XMLSpy will ask if you have an existing DTD or Schema that you may want to use as shown in Figure 8-2.

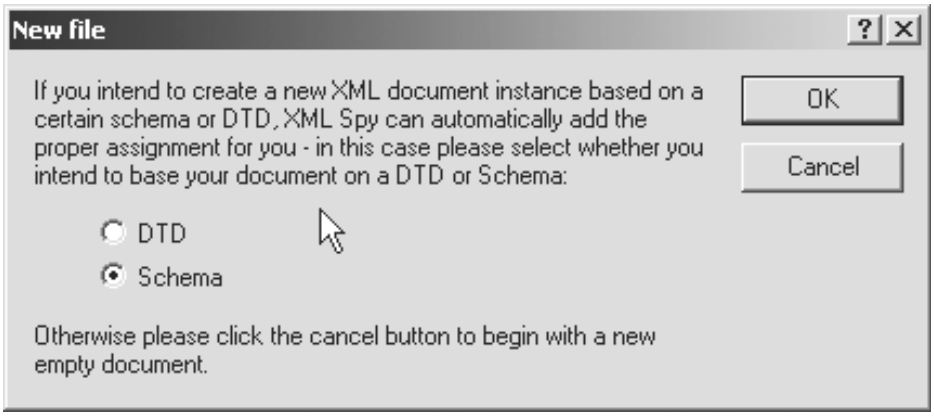

Figure 8-2. Creating an XML document

When a Schema is selected, XMLSpy will automatically generate an XML template that is based on the Schema. Figure 8-3 shows the template.

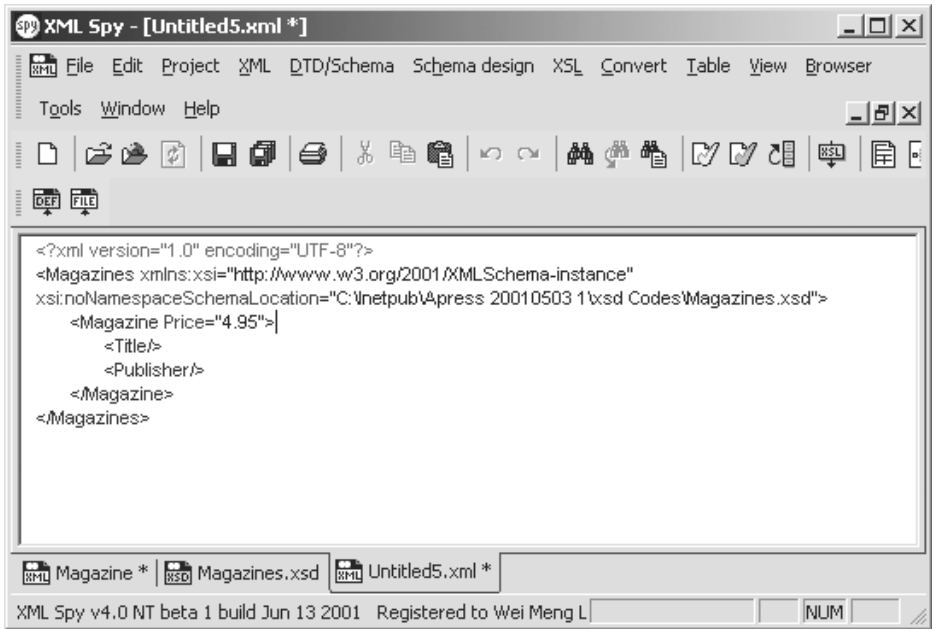

Figure 8-3. Template for XML document

To validate the XML document against the XML Schema, simply click on the "Validate File button," as shown in Figure 8-4.

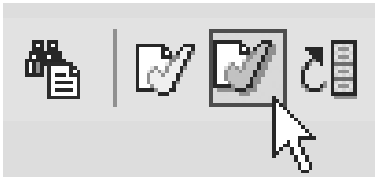

Figure 8-4. Validating an XML document

Using an XML Document to Generate an XML Schema

Although we said that it is more logical to create an XML Schema before you create the XML document, XMLSpy enables you to generate an XML Schema that is based on an XML document, as shown in Figure 8-5.

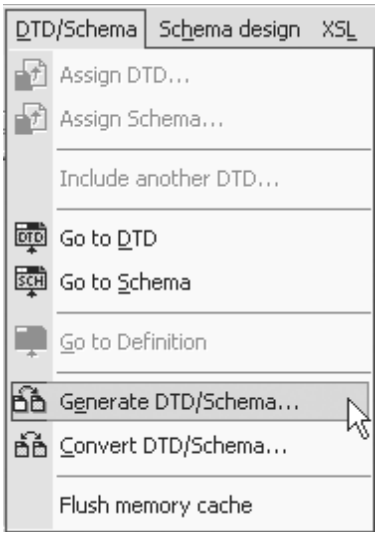

Figure 8-5. Generating an XML Schema

Once an XML Schema is generated, you can simply assign it to the XML document, as shown in Figure 8-6.

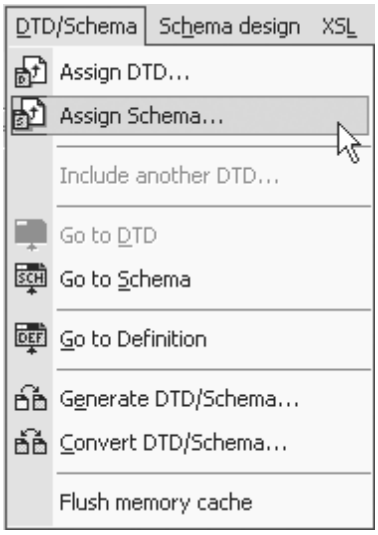

Figure 8-6. Assigning an XML Schema to an XML document

Generating an XML Schema from an XML document is helpful to beginners who are getting started with XML Schema.

Useful Web Links

- XML Schema Part 0: Primer at `http://www.w3.org/TR/xmlschema-0/`
- XML Schema Part 1: Structures at `http://www.w3.org/TR/xmlschema-1/`
- XML Schema Part 2: Data types at `http://www.w3.org/TR/xmlschema-2/`
- XMLSpy Web site at `http://www.xmlspy.com`
- Download MSXML4 at `http://msdn.microsoft.com/xml`

Summary

That's it! In this chapter, we provided an overview of how XML Schema is being used to replaceDTD for validating XML documents. We showed you how to define the structure of an XML document using simpleType and complexType elements. In addition, we showed you how to use XMLSpy from Altova for validating purposes.

Through the many examples that we presented in this chapter, we hope we have provided you with a jump start to getting XML Schema to work for you. Have fun!

http://www.springer.com/978-1-893115-42-2

XML Programming Using the Microsoft XML Parser

Lee, W.-M.; Foo, S.M.

2002, XIV, 451 p. 196 illus., Softcover
ISBN: 978-1-893115-42-2
A product of Apress