Chapter 7

XML and Data Access Integration

The `DataSet` class interoperates with XML schema and data. The `XmlDataDocument` combines `XmlDocument` and `DataSet` functionality.

7.1 XML and Traditional Data Access

The preceding chapters have talked about the data access stack mostly as it relates to traditional data access and relational data. Each time, I've mentioned nonrelational data (homogeneous and heterogeneous hierarchies and semistructured data) almost as an afterthought, or to illustrate that it would be a stretch to support it by using the classes in `System.Data`. But a wide variety of data can be represented in a nonrelational way. Here are a few examples.

- LDAP readable directories, such as Active Directory, contain multivalued attributes. This violates relational theory's first normal form.
- Each item in the NT file system is either a directory or a file. This is an example of a heterogeneous hierarchy. A related case is the reading of data from an Exchange store. Not only are contacts in the Contacts folder structured differently from mail messages in the Inbox, but also each mail message can contain 0–N attachments. The attachments can also vary in data format. Exchange—or other IMAP (Internet Mail Access Protocol) mail systems—can also expose hierarchical folders. All these data structures are analogous to the multiset in IDMS.
- Screen scraping from HTML or XML pages consists of reading through a combination of text and tags and extracting only the data you need—for example, the number in the third column of the fourth row of an HTML table and the contents of the third `<h3>` tag. This is an example of semistructured data.

There are ways to approximate each different data type—with the possible exception of semistructured data—by using a variation of a relational concept. Sometimes, however, you need to present the data in an alternative, nonrelational format. For example, suppose you're managing an electronic student registration form that contains data that affects the value of 15 different normalized relational tables. In addition, the form may contain information, such as a request for low-fat, vegetarian meals, that has no correlation in the relational schema. You may want to store the information in the request into multiple tables and reproduce the original request on demand. This might require that you retain additional information or even the entire request in its original form. It might also be nice if the information could be transmitted in a platform-independent, universally recognized format. Enter XML.

7.2 XML and ADO.NET

One of the most useful features of ADO.NET is its integration with portions of the managed XML data stack. Traditional data access and XML data access have the following integration points:

- The `DataSet` class integrates with the XML stack in schema, data, and serialization features.
- The XML schema compiler lets you generate typed `DataSet` subclasses.
- You can mix nonrelational XML data and the relational `DataSet` through the `XmlDataDocument` class and do XPath queries on this data using `DataDocumentXPathNavigator` class.
- ADO.NET supports SQL Server 2000 XML integration features, both in the `SqlClient` data provider and in an add-on product called SQLXML. The latter product features a series of `SqlXml` managed data classes and lets you update SQL Server via updategram or DiffGram format.

These features, although unrelated in some aspects, work to complete the picture of support of nonrelational as well as relational data in ADO.NET, and direct support of XML for marshaling and interoperability. Let's look first at integration in the `System.Data.DataSet` class and its support for XML.

7.2.1 Defining a `DataSet`'s Schema

In many ways, the ADO.NET `DataSet` class mimics a relational database. Each `DataSet` instance contains a schema—the set of tables, columns, and relationships—as does a relational database. You can define the schema of an ADO.NET `DataSet` instance in at least four ways:

- Use the `DataSet` APIs directly to create `DataTable`s, `DataColumn`s, and `DataRelation`s. This approach is similar in concept to using DDL in relational databases.
- Infer the schema using database metadata through a `DataAdapter` class. Using `DataAdapter.Fill` creates tables and columns matching the metadata from `DataAdapter`'s `SelectCommand`. For this to work, `DataAdapter`'s `MissingSchemaAction` property must be set to `Add` or `AddWithKey`.
- Define the desired `DataSet` schema using XSD (XML Schema Definition language), and use `DataSet.ReadXmlSchema` to load the schema definition into the `DataSet`. The schema may not use nonrelational data definition styles, or else `ReadXmlSchema` will throw an error.
- Use `DataSet.InferXmlSchema`. The `DataSet` class will use a set of schema inference rules to infer a `DataSet` schema from a single XML document.

You can also define `DataSet`'s schema incrementally by using a combination of these methods, as shown in Figure 7–1. Note that in each case the result is the same: `DataSet` contains a set of tables, columns, constraints, and relationships that comply with relational rules.

`DataSet` is not aware of the source of the schema, and therefore any method of defining the schema works as well as any other. For example, let's define a simple schema that includes a customers table, an orders table, and a one-to-many relationship between customers and orders. Listing 7–1 uses the four schema-definition methods to accomplish this. Note that, when using `DataSet` or `DataAdapter`, you need additional code to set up the relationship, whereas in the case of XML schema or document inference, this information may be available in the schema or exemplar document.

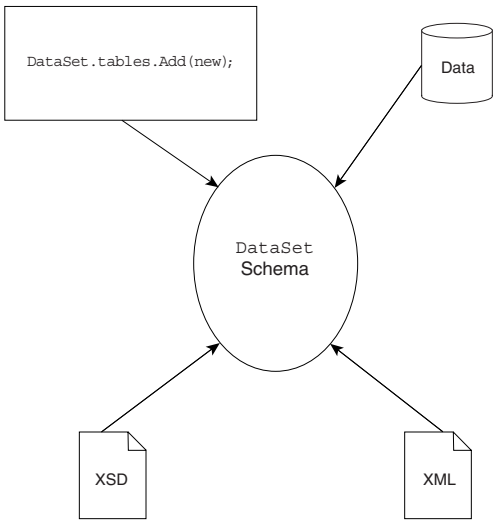

Figure 7–1 Ways to fill the `DataSet`'s schema

Listing 7–1 Ways to create a `DataSet` schema

```
DataSet ds1, ds2, ds3, ds4;

// Use DataSet APIs
ds1 = new DataSet();
ds1.Tables.Add("Customers");
ds1.Tables[0].Columns.Add("custid", typeof(int));
ds1.Tables[0].Columns.Add("custname", typeof(String));
ds1.Tables.Add("Orders");
ds1.Tables[1].Columns.Add("custid", typeof(int));
ds1.Tables[1].Columns.Add("orderid", typeof(int));
ds1.Relations.Add(
  ds1.Tables["Customers"].Columns["custid"],
  ds1.Tables["Orders"].Columns["custid"]);

// Create schema from SQL resultset metadata
ds2 = new DataSet();
SqlDataAdapter da = new SqlDataAdapter(
  "select * from customers;select * from orders",
  "server=localhost;uid=sa;database=northwind");
da.FillSchema(ds2, SchemaType.Source);
ds2.Tables[0].TableName = "Customers";
ds2.Tables[1].TableName = "Orders";
ds2.Relations.Add(
  ds2.Tables["Customers"].Columns["customerid"],
```

```
    ds2.Tables["Orders"].Columns["customerid"]);

// Read schema from a file
// Contains customers and orders
ds3 = new DataSet();
ds3.ReadXmlSchema(
  @"c:\xml_schemas\customers.xsd");

// Infer schema from exemplar document
// Contains customers and orders
ds4 = new DataSet();
ds4.InferXmlSchema(
  @"c:\xml_documents\customers.xml",
  null);
```

Past Microsoft (and other) APIs let you map XML to relational data but require that you specify the XML in a special format. ADO classic, for example, requires the XML to be specified in the format used by ADO's XML support. Some XML support systems require manual coding for each case, based on the programmer's knowledge of the underlying structure and the underlying types of all the elements or attributes involved. Some systems use a document type definition to specify type, but because the DTD is designed around XML's original use as a document markup language it is largely type-ignorant in the traditional programming sense, and the programmer still must know each type that is not `string`.

ADO classic's XML support uses a Microsoft-specific predecessor of XSD schemas; it's known as XDR (XML Data Reduced). XDR's type system is limited. Simple types are based loosely on OLE DB types, and there is no notion of type derivation, precluding the use of the object-oriented concepts of inheritance and polymorphism. .NET object persistence and `DataSet` take full advantage of XSD's improvements in these areas. This "object-relational" mapping layer includes two main classes: `System.Data.DataSet`'s XML support and `System.Xml.XmlDataDocument`, a class that represents a hybrid of the DOM/`DataSet` model. Unlike the ADO `Recordset`, which can consume and produce a single XML format based on a single schema style, `DataSet` can read and write XML corresponding to almost any schema.

Because XSD supports complex user-defined types as well as type derivation by extension and restriction, using XSD to define types supports a schema-based

representation of objects and facilitates a natural mapping to object-based data stores, such as object databases. The hierarchical nature of XML makes this a natural for mapping to homogeneous hierarchies and databases such as IMS. Adding support for navigation and multisets brings CODASYL and object-relational data into the picture. Of course, data can also be stored in files using XML's native serialization format.

Using the XML native serialization format, which is defined in the XML 1.0 and Namespaces recommendations, is arguably a specialization of the object database case, although it enables standards compliance and a measure of code similarity to other platforms using the same object models to access serialized documents. In addition, including `DataSet` in the mix enables codifying of a set of default rules for mapping XML to relational databases. The relational structure of any Infoset can be defined directly using an XML schema for mapping, and it can also be inferred by using a rule-based approach. You can use relations to map XML hierarchies to multiple relational tables. As you've seen, you can map columns to either attributes or subelements of a particular element.

XML schema definition language is more flexible than the rules of a relational database. This means that when you load a `DataSet` via an XSD schema, you use a simple set of rules to map the XSD schema to a relational schema:

- `ComplexType`s are mapped to `DataTable`s.
- Nested `ComplexType`s are mapped to related `DataTable`s.
- `Key` or `Unique` constraints are mapped to `UniqueConstraint`s.
- `KeyRef`s are mapped to `ForeignKeyConstraint`s.

Schema inference is the attempt to deduce a schema from a single exemplar document. Because a single document is used, this technique is more error-prone than simply supplying the schema, and using a predefined schema should always be preferred over schema inference. When you attempt to infer a nonrelational XSD schema into `DataSet` format, it is either coerced into a relational schema (as in Listing 7–2) or you receive an error (as in Listing 7–3). Just as the schema compiler (`XSD.exe`) can infer an XSD schema for an XML document, `DataSet` uses a set of schema inference rules to infer a relational schema for an XML document. The complete set of rules is described in Appendix C. The

`DataSet.InferSchema` method permits you to specify as an optional parameter, a set of namespaces that will be excluded from schema inference.

Listing 7–2 Schema coercion to relational

```
<!-- This results in a table with three columns -->
<root>
  <document>
     <name>Bob</name>
     <address>111 Any St</address>
  </document>
  <document>
     <name>Bird</name>
     <livesin>tree</livesin>
  </document>
</root>
```

Listing 7–3 Attempt to infer a nonrelational schema

```
// Attempt to infer a schema from this document
/*
<book>
<chapter>
<title>Testing your <noun>typewriter</noun></title>
<p>The quick brown <noun>fox</noun> jumps over
the lazy <noun>dog</noun></p>
</chapter>
</book>
*/
DataSet ds = new DataSet();
ds.InferXmlSchema(
  @"c:\xml_documents\semistruct.xml",
  null);

// produces error:
// System.ArgumentException,
// The same table (noun) cannot be the child
// table in two nested relations.
```

Methods that read XML data in .NET usually have the same four overloads. You can read or write XML schema using `Stream`, `String`, `TextReader`, or `XmlReader`. The XML methods in `DataSet` follow the same pattern, as shown in Listing 7–4. The `ReadXmlSchema` method can read the schema in either XDR format or XSD (XML Schema Definition language) format. `InferXmlSchema`

can exclude a set of namespaces from the schema inference process; there is no way to do this with `ReadXmlSchema`.

Listing 7–4 Overloads of `ReadXmlSchema`

```
public class DataSet
{
  // ... other methods omitted
  public void ReadXmlSchema(Stream stream);
  public void ReadXmlSchema(string);
  public void ReadXmlSchema(TextReader reader);
  public void ReadXmlSchema(xmlReader reader);
}
```

`DataSet` also exposes symmetric `WriteXmlSchema` methods. `WriteXmlSchema` uses a hard-coded algorithm to write schemas (see "Writing XML Schemas from `DataSet`" later in this chapter). You can also obtain schema information by using `GetXmlSchema`, which returns the XML schema as a string.

7.2.2 Refining `DataSet`'s XML Schema

You can further refine `DataSet`'s XML schema by using the properties of `DataColumn`, `DataTable`, and `DataSet` that relate to XML. Properties that are useful for setting the schema include `Namespace`, `Prefix`, `ColumnMapping`, and `DataSet`'s `DataRelations` collection. Listing 7–5 shows an example of changing the XML schema for `DataSet`. The default namespace for `DataSet` and its underlying `DataColumn`s and `DataTable`s is no namespace.

Listing 7–5 Changing a schema using column mappings

```
SqlDataAdapter da = new SqlDataAdapter(
     "select * from authors",
     "server=localhost;uid=sa;database=pubs");
DataSet ds = new DataSet();

da.TableMappings.Add("authors", "AllAuthors");
da.TableMappings[0].ColumnMappings.Add("au_id","AuthorID");

da.Fill(ds, "authors");
ds.Tables[0].Namespace = "http://www.develop.com/dmauth";
ds.Tables[0].Prefix = "dmauth";
ds.WriteXmlSchema("myauthors.xsd");
```

XML data can take the form of elements, attributes, or element text. The default in `DataSet` is elements, but you can override it by specifying an XML schema or by using the `InferXmlSchema` method. For those `DataSet`s that do not populate their schema from XML, it can be specified by using `DataColumn`'s `ColumnMapping` property. `ColumnMapping` choices are `MappingType.Element`, `Attribute`, `SimpleText`, or `Hidden`. The latter indicates that this column will not be serialized when `DataSet` is serialized as XML. Listing 7–6 shows the results of using the XML properties of `DataSet`.

Listing 7–6 Using namespace, prefix, and mappings

```
// This code:

DataSet ds = new DataSet();
SqlDataAdapter da = new SqlDataAdapter(
  "select top 2 * from jobs",
  "server=localhost;uid=sa;database=pubs");
da.Fill(ds);
DataTable t = ds.Tables[0];
t.Columns[1].Namespace = "http://www.develop.com";
t.Columns[1].Prefix = "DM";
t.Columns[2].ColumnMapping = MappingType.Attribute;
ds.WriteXml(@"c:\xml_documents\dmjobs.xml");

// Produces this document:

<?xml version="1.0" standalone="yes"?>
<NewDataSet>
  <Table min_lvl="10">
    <job_id>1</job_id>
    <DM:job_desc
     xmlns:DM="http://www.develop.com">
    Vice Chairman</DM:job_desc>
    <max_lvl>10</max_lvl>
  </Table>
  <Table min_lvl="200">
    <job_id>2</job_id>
    <DM:job_desc
     xmlns:DM="http://www.develop.com">
    Grand Poobah</DM:job_desc>
    <max_lvl>250</max_lvl>
  </Table>
</NewDataSet>
```

When you write a `DataSet` containing multiple `DataTable`s in XML format, each `DataTable` is an immediate child of the root element, as shown in Listings 7–7 and 7–8.

Listing 7–7 Data relations and XML

```
SqlDataAdapter da = new SqlDataAdapter(
  "select au_id, au_fname, au_lname from authors;" +
  "select au_id, title_id from titleauthor",
  "server=localhost;uid=sa;database=pubs");
DataSet ds = new DataSet();
da.Fill(ds, "foo");
ds.Tables[0].TableName = "authors";
ds.Tables[1].TableName = "titleauthor";

// non-nested relation
ds.Relations.Add(ds.Tables[0].Columns["au_id"],
  ds.Tables[1].Columns["au_id"]);
ds.Relations[0].Nested = false;
ds.WriteXml("myfile.xml");
```

Listing 7–8 With `Nested=false` (default)

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
  <authors>
    <au_id>213-46-8915</au_id>
    <au_fname>Bob</au_fname>
    <au_lname>Green</au_lname>
  </authors>
  <authors>
    <au_id>472-27-2349</au_id>
    <au_fname>Burt</au_fname>
    <au_lname>Gringlesby</au_lname>
  </authors>
  <titleauthor>
    <au_id>213-46-8915</au_id>
    <title_id>BU1032</title_id>
  </titleauthor>
  <titleauthor>
    <au_id>213-46-8915</au_id>
    <title_id>BU2075</title_id>
  </titleauthor>
  <titleauthor>
    <au_id>486-29-1786</au_id>
```

```
      <title_id>PS7777</title_id>
    </titleauthor>
  </NewDataSet>
```

To produce hierarchical XML from multiple `DataTable`s, you must define a `DataRelation` between them, and the `DataRelation`'s `Nested` property must be set to `true`. This is illustrated in Listing 7–9.

Listing 7–9 With `Nested=true`

```
<NewDataSet>
  <authors>
    <au_id>213-46-8915</au_id>
    <titleauthor>
      <au_id>213-46-8915</au_id>
      <title_id>BU1032</title_id>
    </titleauthor>
    <titleauthor>
      <au_id>213-46-8915</au_id>
      <title_id>BU2075</title_id>
    </titleauthor>
    <au_fname>Bob</au_fname>
    <au_lname>Green</au_lname>
  </authors>
  <authors>
    <au_id>472-27-2349</au_id>
    <titleauthor>
      <au_id>472-27-2349</au_id>
      <title_id>TC7777</title_id>
    </titleauthor>
    <au_fname>Burt</au_fname>
    <au_lname>Gringlesby</au_lname>
  </authors>
</NewDataSet>
```

7.2.3 Reading XML into `DataSet`

XML document data can be read into `DataSet` just as data from `DataAdapter` can, using one of the overloads of `DataSet.ReadXml`. There are the usual four overloads: `String`, `Stream`, `TextReader`, and `XmlReader`. To refine the process, you can use another set of overloads via a second `XmlReadMode` parameter. There is also a set of mostly symmetric `WriteXml` methods.

There are two ways to read and write the contents of `DataSet`. The default is to read or write all the current values in `DataSet` that are not specified as `DataColumn.ColumnMapping = MappingType.Hidden`. An example is shown earlier in Listing 7–6. A second format, known as DiffGram, includes all the current rows, before images for those rows that you have changed since the last time you called `AcceptChanges`, and a set of error elements that contains errors for specific rows. Listing 7–10 shows the general layout of a DiffGram.

Listing 7–10 General format of a DiffGram

```
<?xml version="1.0"?>
<diffgr:diffgram
 xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
 xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <DataInstance>
   </DataInstance>

  <diffgr:before>
  </diffgr:before>

  <diffgr:errors>
  </diffgr:errors>
</diffgr:diffgram>
```

The DiffGram format adds annotations from the namespace `urn:schemas-microsoft-com:xml-diffgram-v1` to the XML output. The most important annotation is the `id` attribute, which is used to tie before images and errors to specific rows. Listing 7–11 shows a typical DiffGram. DiffGrams are discussed further in "Writing XML Data from `DataSet`" later in this chapter.

Listing 7–11 A DiffGram in which both rows have been changed

```
<?xml version="1.0" standalone="yes"?>
<diffgr:diffgram
 xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
 xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1">
  <NewDataSet>
    <Table diffgr:id="Table1" msdata:rowOrder="0"
           diffgr:hasChanges="modified">
      <job_id>1</job_id>
      <job_desc>Different value</job_desc>
```

```
      <min_lvl>10</min_lvl>
      <max_lvl>10</max_lvl>
    </Table>
    <Table diffgr:id="Table2" msdata:rowOrder="1"
           diffgr:hasChanges="modified">
      <job_id>2</job_id>
      <job_desc>Different value</job_desc>
      <min_lvl>200</min_lvl>
      <max_lvl>250</max_lvl>
    </Table>
  </NewDataSet>
  <diffgr:before>
    <Table diffgr:id="Table1" msdata:rowOrder="0">
      <job_id>1</job_id>
      <job_desc>Vice Chairman</job_desc>
      <min_lvl>10</min_lvl>
      <max_lvl>10</max_lvl>
    </Table>
    <Table diffgr:id="Table2" msdata:rowOrder="1">
      <job_id>2</job_id>
      <job_desc>Grand Poobah</job_desc>
      <min_lvl>200</min_lvl>
      <max_lvl>250</max_lvl>
    </Table>
  </diffgr:before>
</diffgr:diffgram>
```

When you read an XML document into a `DataSet` using `ReadXml`, you might have a document with an *embedded* schema (one that is contained in the same file or stream as the document) or a document that contains only the data. In addition, you can use `ReadXml` to read from a document that contains a DiffGram or a "normal" XML document. `XmlReadMode` specifies how the `ReadXml` method works. The following `XmlReadMode`s are affected by the existence of a schema:

- `InferSchema`: Ignores an inline schema and infers the schema from the data. If tables already exist in the `DataSet` but have incompatible properties, an error is thrown. If the `DataSet`'s existing schema overlaps, tables or columns will be added.
- `IgnoreSchema`: Uses only the `DataSet`'s existing schema. Data in the XML input that does not match the existing schema is thrown away.

- `ReadSchema`: Uses the inline schema if one exists. If tables with the same name already exist in the `DataSet`'s schema, an exception is thrown.
- `DiffGram`: Applies the changes assuming DiffGram format as input. If the input is in DiffGram format, this works in the same way as the `DataSet.Merge` method. If the XML input is not in DiffGram format, this works the same as `IgnoreSchema`.

If the `XmlReadMode` is not specified, the default is `XmlReadMode.Auto`, which works as follows:

- If the XML input is a DiffGram, it is read as a DiffGram, possibly populating multiple `RowVersion`s in the `DataSet`.
- If the input is not in DiffGram format and if the `DataSet` already has a schema or there is an inline schema, `ReadSchema` is used.
- If no schema exists in the `DataSet` or inline, `InferSchema` is used.

Schemas used by `ReadXml` can be in either XDR or XSD format. Either XDR or XSD format can use inline schemas. XSD can also use the `xsi:SchemaLocation` attribute to specify the schema location.

Filling the `DataSet` by using `ReadXml` works differently than filling the `DataSet` using `DataAdapter.Fill`. When you use `DataAdapter`, by default, changes are implicitly accepted when `Fill` is finished; you can override this by setting `DataAdapter`'s `AcceptChangesDuringFill` property. Calling `RejectChanges` immediately after calling `Fill` has no effect. When `ReadXml` is used, the changes are not accepted until `AcceptChanges` is explicitly called. The reasoning behind this behavior is that `ReadXml` can be used to read rows into the `DataSet`, and these rows can be used to update a database through a `DataAdapter`, as shown in Listing 7–12. If `AcceptChanges` were implicit, attempting to update a database in this way would fail because the rows in the `DataSet` would be marked as original rows only. Calling `RejectChanges` after using `ReadXml` rolls back any rows added but leaves the schema intact, as shown in Listing 7–13.

Listing 7–12 Changes are not accepted during `ReadXml`

```
DataSet ds = new  DataSet();
SqlDataAdapter da = new SqlDataAdapter(
```

```
    "select * from jobs",
    "server=localhost;uid=sa;database=pubs");
SqlCommandBuilder bld = new SqlCommandBuilder(da);

da.MissingSchemaAction = MissingSchemaAction.AddWithKey;
da.Fill(ds, "jobs");

ds.ReadXml(@"c:\xml_documents\more_jobs.xml");

// adds jobs in more_jobs.xml to database
da.Update(ds, "jobs");
```

Listing 7–13 Calling `RejectChanges` deletes rows added by `ReadXml`

```
SqlDataAdapter da = new SqlDataAdapter(
  "select * from jobs",
  "server=localhost;uid=sa;database=pubs");
DataSet ds = new DataSet();
da.Fill(ds);
// 15 rows here
Console.WriteLine("{0} rows in table 0",
  ds.Tables[0].Rows.Count);
ds.RejectChanges();
// still 15 rows here
Console.WriteLine("{0} rows in table 0",
  ds.Tables[0].Rows.Count);

DataSet ds2 = new DataSet();
ds2.ReadXml(@"c:\xml_documents\jobs.xml");
// 15 rows here
Console.WriteLine("{0} rows in table 0",
  ds2.Tables[0].Rows.Count);
ds2.RejectChanges();
// table still exists but 0 rows
Console.WriteLine("{0} rows in table 0",
  ds2.Tables[0].Rows.Count);
```

The DiffGram format works differently from "ordinary" XML, no matter which `XmlReadMode` is specified. With any of the `XmlReadMode`s, if the table or tables that the DiffGram refers to do not already exist in the `DataSet`'s schema, `ReadXml` results in a `System.Data.DataException` being thrown. In addition, you must be very careful when mixing DiffGrams with `DataSet`s that are filled using `DataSet.Fill`. If all the metadata (such as `PrimaryKey` or `Unique` constraint) is not present when the DiffGram is read, the `DataSet` may

contain duplicate rows, and the resulting call to `DataAdapter.Update` may fail or produce the wrong results. For best results when using `DataAdapter` to update a database, always retrieve the metadata by specifying `MissingSchemaAction.AddWithKey` or explicitly calling `DataAdapter.FillSchema`. The rule is to remember that using `XmlReadMode.DiffGram` acts like `DataSet.Merge`.

7.2.4 Writing XML Schemas from `DataSet`

When you write XML from `DataSet`, you have similar options as when reading XML. The instructions for writing the XML data and schema are specified as properties of `DataSet` and related classes, so the process is straightforward. `WriteXmlSchema` has the same four overloads as `ReadXmlSchema`; it can write to `Stream`, `String`, `TextWriter`, or `XmlWriter`. In contrast to reading, `WriteXmlSchema` does not allow a choice of schema format; only XSD schemas are supported. The schemas written reflect the `Namespace` and `Prefix` properties of the `DataSet`, `DataTable`s, and `DataColumn`s as well as the `DataRelation`'s `Nested` property and the `DataColumn`'s `ColumnMapping`. Although the schema is a full-fidelity XML schema, it does reflect the relational nature of the `DataTable`s in the `DataSet`. In addition, the `DataSet`'s schema contains annotations from a Microsoft-specific namespace, making XML documents that use the schema load faster into the `DataSet` than those that don't use the schema.

Let's look at a sample `DataSet` containing two `DataTable`s and a primary-foreign key relationship between them. The code that produced this `DataSet` is shown in Listing 7–14.

Listing 7–14 Code to produce an XML schema

```
SqlConnection conn = new SqlConnection(
  "server=(local);uid=sa;pwd=;database=pubs");
SqlDataAdapter da = new SqlDataAdapter(
  "select * from authors;select * from titleauthor",
  conn);

da.TableMappings.Add("Table", "authors");
da.TableMappings.Add("Table1", "titleauthors");
```

```
DataSet ds = new DataSet("OneMany");
da.MissingSchemaAction =
  MissingSchemaAction.AddWithKey;
da.Fill(ds);
ds.Relations.Add(
  "au_ta",
  ds.Tables[0].Columns["au_id"],
  ds.Tables[1].Columns["au_id"],
  true);
ds.Relations[0].Nested = true;
ds.WriteXmlSchema("one_to_many.xsd");
```

Listing 7–15 shows the schema produced by calling `DataSet.WriteXmlSchema`. Examining the schema in detail elucidates the relationship between relational data and XML. The most interesting point is that the schema is nonrelational! Although both the `authors` and the `titleauthors` `DataTable`s are represented as `xsd:ComplexType`, the `titleauthors` table is represented as a nested `xsd:ComplexType`, an embedded member of `authors`. This representation is more similar to an object-oriented, embedded table view than to a relational view, where the tables are not hierarchical. This schema does not allow a "child" type without an existing parent, something that is consistent with the primary-foreign key constraint in a relational database.

Listing 7–15 Schema produced by the code in Listing 7–14

```
<?xml version="1.0" standalone="yes"?>
<xsd:schema id="OneMany" targetNamespace=""
   xmlns="" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xsd:element name="OneMany" msdata:IsDataSet="true">
    <xsd:complexType>
      <xsd:choice maxOccurs="unbounded">
        <xsd:element name="authors">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="au_id"
                           type="xsd:string" />
              <xsd:element name="au_lname"
                           type="xsd:string" />
              <xsd:element name="au_fname"
                           type="xsd:string" />
              <xsd:element name="phone"
```

```
                    type="xsd:string" />
          <xsd:element name="address"
                    type="xsd:string"
                    minOccurs="0" />
          <xsd:element name="city"
                    type="xsd:string"
                    minOccurs="0" />
          <xsd:element name="state"
                    type="xsd:string"
                    minOccurs="0" />
          <xsd:element name="zip"
                    type="xsd:string"
                    minOccurs="0" />
          <xsd:element name="contract"
                    type="xsd:boolean" />
          <xsd:element name="titleauthors"
                    minOccurs="0"
                    maxOccurs="unbounded">
           <xsd:complexType>
             <xsd:sequence>
               <xsd:element name="au_id"
                            type="xsd:string" />
               <xsd:element name="title_id"
                            type="xsd:string" />
               <xsd:element name="au_ord"
                            type="xsd:unsignedByte"
                            minOccurs="0" />
               <xsd:element name="royaltyper"
                            type="xsd:int"
                            minOccurs="0" />
             </xsd:sequence>
            </xsd:complexType>
           </xsd:element>
         </xsd:sequence>
       </xsd:complexType>
     </xsd:element>
   </xsd:choice>
</xsd:complexType>
<xsd:unique name="titleauthors_Constraint1"
 msdata:ConstraintName="Constraint1"
 msdata:PrimaryKey="true">
  <xsd:selector xpath=".//titleauthors" />
  <xsd:field xpath="au_id" />
  <xsd:field xpath="title_id" />
```

```
    </xsd:unique>
    <xsd:unique name="Constraint1"
                msdata:PrimaryKey="true">
      <xsd:selector xpath=".//authors" />
      <xsd:field xpath="au_id" />
    </xsd:unique>
    <xsd:keyref name="au_ta"
                refer="Constraint1"
                msdata:IsNested="true">
      <xsd:selector xpath=".//titleauthors" />
      <xsd:field xpath="au_id" />
    </xsd:keyref>
  </xsd:element>
</xsd:schema>
```

Because the code does not specify otherwise, all `DataColumn`s are represented as elements. This results in `DataColumn`s that can be `NULL`, using the `minOccurs="0"` facet in the schema. `NULL` values will be represented, not by empty elements but instead by the element's absence in the data. This distinguishes a `NULL` value from an empty string, as in relational databases. Both the `authors` and the `titleauthors` primary keys are represented by the `xsd:unique` production. The relationship between the two tables is represented by an `xsd:keyref` production. Note that the `xsd:key` production is not used to indicate that the unique fields are primary keys. Instead, this is indicated by a Microsoft-specific annotation.

7.2.5 Microsoft-Specific Annotations

The `urn:schemas-microsoft-com:xml-msdata` namespace annotations used in the `DataSet`'s schema are the most controversial part of the schema, although they are in complete compliance with the schema recommendation document. XML purists argue that they are a "hack" and should not be included in an XML schema at all. Another opinion is that they are convenience features that can be safely ignored by systems that do not support .NET `DataSet`s. The truth falls somewhere in between.

Let's look at some of these annotations. The first annotation, `msdata:IsDataSet="true"`, appears on the element that will become the `DataSet`. Although it may appear at first that there should be a `DataSet xsd:ComplexType`, it is

impossible because `DataSet` is a generic container. The number of `DataTable`, `DataColumn`, and `DataRelation` items can vary with each instance, although each `DataSet` instance can be precisely described. The `msdata:IsDataSet` annotation assists in deserialization for systems that support such a container; other systems can safely ignore it, although the resulting structure graph will not have `DataSet` semantics, such as the ability to dynamically add a `DataTable`.

The `msdata:ConstraintName="Constraint1"` annotation is an interesting way to specify a relational constraint. It combines with the `"name"` attribute of the `authors` table `PrimaryKey` and the `"refer"` attribute of the `xsd:keyref` production to define a relationship. This definition is nonstandard and outside the realm of XSD, but it is a compliant schema element. Using the annotation that results in using `msdata:PrimaryKey="true"` on an `xsd:unique` production instead of the `xsd:key` production is not technically the best way to describe the key in XML schemas, but it does permit the `DataSet` to have primary-foreign key relationships using unique fields that are not primary keys. Because `DataSet` allows this, using the `PrimaryKey` production simplifies deserialization.

The last annotation, `msdata:IsNested="true"`, allows `DataSet` to serialize hierarchical XML when the relationship is not actually a nested table hierarchy. This is a convenience feature for XML processors that "prefer" a hierarchical representation.

Using Microsoft-specific annotations to coerce relational data into different shapes is a step you can take toward bridging the gap between relational and other data representations. This step is necessary because strict relational is a simplification of the rich set of data representations possible in XML. I discuss this issue further in Chapter 10.

7.2.6 Writing XML Data from `DataSet`

XML data can be written from `DataSet` in a variety of formats. Each format reflects the `Prefix`es, `Namespace`s, `Relation`s, and `ColumnMapping`s and can be validated using the schema that would be written using `WriteXmlSchema` against the same `DataSet`. The `DataSet.WriteXml` method has two series of the same four overloads as `DataSet.ReadXml`; the second parameter is the

`XmlWriteMode` enumeration. The `XmlWriteMode` is a simple enumeration; a `DataSet` can be written with or without the corresponding XSD schema. In addition, it can be written as DiffGram format.

DiffGram format is an XML representation of an entire `DataSet` with extra rows that correspond to pending changes. Listing 7–16 shows an example that adds, deletes, and changes rows in an existing `DataSet`, along with the DiffGram produced. Note that this code uses annotations from the `msdata` prefixed namespace and that the DiffGram itself comes from a new namespace (`urn:schemas-microsoft-com:xml-diffgram-v1`). The `msdata` annotation is used to define the order in which the rows occur in the `DataSet`. Everything else is defined in the DiffGram's namespace. The DiffGram assigns each row a unique ID composed of the `TableName` and an ordinal, in this case `Table1X`, where `Table1` is the `TableName` and `X` is the unique-ifier. The DiffGram consists of a set of current rows in the `DataSet` followed by a set of rows contained within a `diffgram:before` element. Changes are represented by `before` images and the `diffgram:hasChanges` attribute. Changed rows have a `before` and `current` row, and each has a `hasChanges="modified"` attribute. Inserted rows appear in the `current` set of rows only and have a `hasChanges="inserted"` attribute. Deleted rows appear in the `before` section only and are marked with `hasChanges="deleted"`. From this format, an entire `DataSet` of rows and changes is represented. A DiffGram can be persisted with or without schema.

Listing 7–16 DiffGram produced after making changes

```
//
// This code:

DataSet ds = new DataSet();
SqlDataAdapter da = new SqlDataAdapter(
  "select * from jobs",
  "server=localhost;uid=sa;database=pubs");

da.MissingSchemaAction = MissingSchemaAction.AddWithKey;
da.Fill(ds);

ds.Tables[0].Rows[1][1] = "newer description";
ds.Tables[0].Rows[2].Delete();
```

```
ds.Tables[0].Rows.Add(
  new object[4] { null, "new row", 40, 40 });

ds.WriteXml(@"c:\xml_documents\writeDiffGram.xml",
  XmlWriteMode.DiffGram);

<!-- produces this DiffGram -->
<?xml version="1.0" standalone="yes"?>
<diffgr:diffgram
 xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
 xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1"
>
  <NewDataSet>
    <Table diffgr:id="Table1" msdata:rowOrder="0">
      <job_id>1</job_id>
      <job_desc>Vice Chairman</job_desc>
      <min_lvl>10</min_lvl>
      <max_lvl>10</max_lvl>
    </Table>
    <Table diffgr:id="Table2" msdata:rowOrder="1"
           diffgr:hasChanges="modified">
      <job_id>2</job_id>
      <job_desc>newer description</job_desc>
      <min_lvl>200</min_lvl>
      <max_lvl>250</max_lvl>
    </Table>
    <Table diffgr:id="Table4" msdata:rowOrder="3">
      <job_id>4</job_id>
      <job_desc>Chief Financial Officier</job_desc>
      <min_lvl>175</min_lvl>
      <max_lvl>250</max_lvl>
    </Table>
    <Table diffgr:id="Table5" msdata:rowOrder="4">
      <job_id>5</job_id>
      <job_desc>Publisher</job_desc>
      <min_lvl>150</min_lvl>
      <max_lvl>250</max_lvl>
    </Table>
    <Table diffgr:id="Table6" msdata:rowOrder="5">
      <job_id>6</job_id>
      <job_desc>Managing Editor</job_desc>
      <min_lvl>140</min_lvl>
      <max_lvl>225</max_lvl>
    </Table>
```

```
    <!-- some rows (6-14) deleted from diagram here... -->
    <Table diffgr:id="Table16" msdata:rowOrder="15"
           diffgr:hasChanges="inserted">
      <job_id>0</job_id>
      <job_desc>new row</job_desc>
      <min_lvl>40</min_lvl>
      <max_lvl>40</max_lvl>
    </Table>
  </NewDataSet>
  <diffgr:before>
    <Table diffgr:id="Table2" msdata:rowOrder="1">
      <job_id>2</job_id>
      <job_desc>zzz</job_desc>
      <min_lvl>200</min_lvl>
      <max_lvl>250</max_lvl>
    </Table>
    <Table diffgr:id="Table3" msdata:rowOrder="2">
      <job_id>3</job_id>
      <job_desc>Business Operations Manager</job_desc>
      <min_lvl>175</min_lvl>
      <max_lvl>225</max_lvl>
    </Table>
  </diffgr:before>
</diffgr:diffgram>
```

Finally, you can produce a second `DataSet` consisting only of changes and then write the changes as a DiffGram. This technique is useful when you're marshaling changes because the entire `DataSet` need not be passed across the wire. To implement this, you use `DataSet`'s `GetChanges` method. Listing 7–17 shows an example of the code and resulting DiffGram.

Listing 7–17 Using `GetChanges` and DiffGrams

```
//
// This code

DataSet ds = new DataSet();
SqlDataAdapter da = new SqlDataAdapter(
  "select * from jobs",
  "server=localhost;uid=sa;database=pubs");
da.MissingSchemaAction = MissingSchemaAction.AddWithKey;
da.Fill(ds);

ds.Tables[0].Rows[1][1] = "newer description";
```

```
ds.Tables[0].Rows[2].Delete();
ds.Tables[0].Rows.Add(
  new object[4] { null, "new row", 40, 40 });

DataSet ds2 = new DataSet();
ds2 = ds.GetChanges();
ds2.WriteXml(
 "c:\\xml_documents\\writeDiffGramChanges.xml",
 XmlWriteMode.DiffGram);

<!-- produces this DiffGram -->
<?xml version="1.0" standalone="yes"?>
<diffgr:diffgram
 xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
 xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1"
>
  <NewDataSet>
    <Table diffgr:id="Table1" msdata:rowOrder="0"
           diffgr:hasChanges="modified">
      <job_id>2</job_id>
      <job_desc>newer description</job_desc>
      <min_lvl>200</min_lvl>
      <max_lvl>250</max_lvl>
    </Table>
    <Table diffgr:id="Table3" msdata:rowOrder="2"
           diffgr:hasChanges="inserted">
      <job_id>0</job_id>
      <job_desc>new row</job_desc>
     <min_lvl>40</min_lvl>
      <max_lvl>40</max_lvl>
    </Table>
  </NewDataSet>
  <diffgr:before>
    <Table diffgr:id="Table1" msdata:rowOrder="0">
      <job_id>2</job_id>
      <job_desc>zzz</job_desc>
      <min_lvl>200</min_lvl>
      <max_lvl>250</max_lvl>
    </Table>
    <Table diffgr:id="Table2" msdata:rowOrder="1">
      <job_id>3</job_id>
      <job_desc>Business Operations Manager</job_desc>
      <min_lvl>175</min_lvl>
      <max_lvl>225</max_lvl>
```

```
    </Table>
  </diffgr:before>
</diffgr:diffgram>
```

After you have produced a `DataSet` as a result of using `GetChanges`, you can persist it as a DiffGram (current and changed rows) or as a normal XML document (current rows only). You can also use `GetChanges` to get the rows that have a certain `RowState`, such as only `Added` rows. Listing 7–18 shows the use of `GetChanges` to marshal changes made on the client tier back to a middle tier, where they are used in an update. Note that, although all data is always marshaled in XML format, only changed data is sent across the wire, thereby cutting down on network traffic. Finally, the changed rows are returned to the client to refresh the client's copy of the `DataSet`.

Listing 7–18 Round-trip update; only changes are marshaled in XML format

```
// 1. middle tier
DataSet ds = new DataSet();
SqlDataAdapter da = new SqlDataAdapter(
  "select * from jobs",
  "server=localhost;uid=sa;database=pubs");

da.MissingSchemaAction = MissingSchemaAction.AddWithKey;
da.Fill(ds);
// 2. pass to client here

ds.Tables[0].Rows[1][1] = "newer description";
ds.Tables[0].Rows[14].Delete();
ds.Tables[0].Rows.Add(
  new object[4] { null, "new row", 40, 40 });

DataSet ds2 = new DataSet();
ds2 = ds.GetChanges();
// 3. pass ds2 back to middle tier here
// just pass back the changes

SqlCommandBuilder bld = new SqlCommandBuilder(da);
da.InsertCommand = bld.GetInsertCommand();
da.UpdateCommand = bld.GetUpdateCommand();
da.DeleteCommand = bld.GetDeleteCommand();

// make sure we get the identity column on insert
```

```
da.InsertCommand.CommandText +=
  ";select * from jobs where job_id = @@identity";
da.InsertCommand.UpdatedRowSource =
  UpdateRowSource.FirstReturnedRecord;

// update on middle tier using changes only.
// refresh with most current rows
// this is only needed for insert,
// update has most current rows
da.Update(ds2);
// 4. pass changes only back to client

// client now contains latest changes
ds.Merge(ds2);
```

7.3 Serialization, Marshaling, and `DataSet`

Rather than use a binary format by default, as in ADO classic, the `DataSet` default is XML serialization and marshaling. This means that you can populate `DataSet` from non-Microsoft data sources, and the data is consumable from non-Microsoft platforms. Because `DataSet` marshals as an XML document, it is natively supported without transformation by Web Services.

The .NET platform includes two libraries that serialize and deserialize classes in XML format:

- `System.Runtime.Serialization` is used for marshaling in .NET implementations.
- `System.Xml.Serialization` is used in Web Services to support unlike implementations.

`DataSet` is compatible with both of them.

`System.Runtime.Serialization` serializes .NET classes using two formatters included in the .NET framework: the Binary formatter and the SOAP formatter. The Binary formatter uses a .NET-specific format and protocol to optimize size by reducing the number of bytes transmitted. The SOAP formatter uses an XML-based format and the Simple Object Access Protocol. The standardization of SOAP details is in progress under the auspices of the W3C XML-SP committee. `System.Runtime.Serialization.SoapFormatter` uses SOAP 1.1 as its format. SOAP 1.1 is currently a W3C note; an updated version

(SOAP 1.2) has been released. The SOAP formatter is CLR type-centric. It can serialize any CLR type to SOAP format but cannot serialize any arbitrary XML; some XML types cannot be processed, and others are serialized differently from the expected XSD-defined format. For example, arrays are serialized according to SOAP section 5, which is not consistent with XSD schemas.

`System.Xml.Serialization` is XML-centric in its approach. It can serialize any XML simple or complex type that can be represented in an XML schema, but it may not be able to serialize all CLR types with 100 percent fidelity. It is used in the `System.Web.Services` library for greatest compatibility with unlike platforms. The inability of CLR serialization to serialize all schema types, and the inability of `XML.Serialization` to handle all CLR types, is not a deficiency of the implementation; rather, it's a result of the inherent difference between the schema type system and the CLR type system.

To indicate support for serialization using `System.Runtime.Serialization`, the class must mark itself with the `[Serializable]` attribute. Classes that use the `[Serializable]` attribute can either accept the system's default serialization mechanism or implement `ISerializable` in a class-specific manner. Listing 7–19 shows how to use the `[Serializable]` attribute and implement a custom version of `ISerializable`.

Listing 7–19 A class that implements `ISerializable`

```
[Serializable]
public class Foo : ISerializable
{
  public int x, y;
  public Foo() {}

  internal Foo(SerializationInfo si,
               StreamingContext context)
  {
        //Restore our values.
        x = si.GetInt32("i");
        y = si.GetInt32("j");
  }

  public void GetObjectData(SerializationInfo si,
                            StreamingContext context)
  {
```

```
        //Add our three scalar values;
        si.AddValue("x", x);
        si.AddValue("y", y);

        Type t = this.GetType();
        si.AddValue("TypeObj", t);
      }
    }
```

Note that implementing `ISerializable` requires two things: implementing the `GetObjectData` method to fill in the `SerializationInfo` property bag, and implementing a constructor that takes the `SerializationInfo` and `StreamingContext` parameters. Custom serialization methods can be implemented to optimize serialization based on the `StreamingContext`. The `DataSet` class implements a custom version of `ISerializable`.

XML schema-centric serialization is controlled by the `XmlSerializer` class in the `System.Xml.Serialization` namespace. This class can generate custom `XmlSerializationReader`/`XmlSerializationWriter` pairs on a per-type basis. By default, `XmlSerializer` uses a one-to-one CLR-class-to-XML-complex-type mapping. Classes can customize the exact serialization by decorating their class declarations with a series of CLR attributes from the `System.Xml.Serialization` namespace. `DataSet` uses a custom mechanism to interact with `XmlSerializer`.

`DataSet` supports both `System.Runtime.Serialization` and `System.Xml.Serialization`. It supports each one through its implementations of `ReadXmlSchema`/`ReadXml` and `WriteXmlSchema`/`WriteXml`. When `System.Runtime.Serialization` is used, `GetObjectData` uses the `WriteXmlSchema` and `WriteXml` methods directly. In addition, `DataSet` has the appropriate constructor for custom serialization and invokes `ReadXmlSchema` and `ReadXml` to populate itself from `SerializationInfo`. There are no optimizations for different streaming contexts; `DataSet` is marshaled by value even across `appdomain` boundaries.

`DataSet` supports custom XML-centric serialization by implementing a special interface, `IXmlSerializable`. Currently it is the only class in the base class libraries to implement this interface. `IXmlSerializable` has three methods—`ReadXml`, `WriteXml`, and `GetSchema`—which are implemented in

`DataSet` by calling the appropriate `Read` or `WriteXml` and `Read` or `WriteXmlSchema`, just as in `System.Runtime.Serialization`.

If you want to use complex types as `DataColumn`s, it is useful to know exactly how `DataSet` is serialized. When `DataSet` is serialized, `WriteXml` calls `XmlDataTreeWriter`, which eventually writes each row with an `XmlDataRowWriter`. Then `XmlDataRowWriter` calls `DataColumn.ObjectToXml` on every column. `DataColumn.ObjectToXml` calls only `System.Data.Common.DataStorage.ObjectToXml`. The `System.Data.Common.DataStorage` class has a static method called `CreateStorage`. It creates `Storage` classes for any of the concrete types it supports—that is, it calls the constructor on the concrete classes: `System.Data.Common.XXXStorage`.

A final storage class is called `ObjectStorage`. Any class that is not directly supported by `DataSet` will use the `ObjectStorage` class. This is important when you think back to the example in Chapter 4 that stores `Object` types in `DataSet`.

Every `DataColumn` value in a `DataTable` is represented as XML by calling its `ToString` method. It is rehydrated from XML by using a constructor that takes a single string as input. Therefore, to use arbitrary objects as `DataColumn` types, they must have a `ToString` method that renders their value as XML and a single string constructor. This is a difficult design decision because a method (`ToString`) that may produce string output for reports must be reserved for XML, but the decision must be tempered by the fact that a complex type usually cannot be represented as a single string. Listing 7–20 illustrates this type of object using the `Person` class from Chapter 4.

Listing 7–20 Producing correct XML with the `Person` class

```
public class Person
{
  public String name;
  public int age;

  public Person(String serstr)
    {
      Person p;
      XmlSerializer ser = new XmlSerializer(typeof(Person));
      p = (Person)ser.Deserialize(new StringReader(serstr));
```

```
        this.age = p.age;
        this.name = p.name;
      }

    public override string ToString()
      {
        String s;
        StringBuilder mysb =
          new StringBuilder();
        StringWriter myStringWriter =
          new StringWriter(mysb);
        XmlSerializer ser = new XmlSerializer(this.GetType());
          ser.Serialize(myStringWriter, this);

         s = myStringWriter.ToString();
         return s;
      }
}
```

To use an embedded `DataTable` in a `DataColumn`, as you did in Chapter 4, you must override the `DataTable`'s implementation of these two methods. Unfortunately, the `DataTable` has a single string constructor, and to implement this constructor in such a way changes the semantics of the base class and is suboptimal. SQL Server's `UNIQUEIDENTIFIER` class is an example of using this pair of methods to map to `System.Guid`, which has the appropriate constructor and `ToString` method to be correctly marshaled as a column inside `DataSet`. The `DataSet` class implements two additional public methods—`ShouldSerializeTables` and `ShouldSerializeRelations`—to allow `Serialization` to work with subclasses, such as strongly typed `DataSet`s.

7.4 Typed `DataSet`s

One of the functions performed by `XSD.exe`, the XML schema generation tool, is to generate a *typed* `DataSet` from an XSD schema. This functionality is also available in Visual Studio as a menu item and context menu entry on an existing "`DataAdapter` object." What exactly is a typed `DataSet`?

A typed `DataSet` is a subclass of `System.Data.DataSet` in which the tables that exist in the `DataSet` are derived by reading the XSD schema information. The difference between a typed `DataSet` and an "ordinary" `DataSet` is

that the DataRows, DataTables, and other items are available as strong types; that is, rather than refer to DataSet.Tables[0] or DataSet.Tables["customers"], you code against a strongly typed DataTable named, for example, MyDataSet.Customers. Typed DataSets have the advantage that the strongly typed variable names can be checked at compile time rather than causing errors at runtime. A short example will illustrate this concept.

Suppose you have a DataSet that should contain a table named customers. It should have columns named custid and custname. You can refer to the table and the columns by ordinal or by name. As shown in Listing 7–21, the data is loosely typed when referred to by ordinal or name, meaning that the compiler cannot guarantee that you've spelled the column name correctly or used the correct ordinal. The problem is that the error informing you of this occurs at runtime rather than at compile time. If the DataSet items were strongly typed, misspelling the column name or using the wrong ordinal would be prevented because the code simply would not compile.

Listing 7–21 Referring to DataTables and Columns

```
DataSet ds = new DataSet();
// some action to load the DataSet...

// this will fail if second table does not exist
String name = ds.Tables[1].TableName;
// this will fail if the table is named customers
DataTable t = ds.Tables["customesr"];

// This will fail if the DataRow r has fewer than 5 columns
// or if column 5 is a String data type
DataRow r;
int value = (int)r[4];
// This will fail if there is a column named "custname"
String value = r["custnam"].ToString();
```

This does not solve every problem; mismatches can still occur if the database schema changes between the time the typed DataSet was generated and runtime. But because the structure of the DataSet is built into the names, the compiler can catch the misspellings. The examples in Listing 7–22 illustrate this.

Listing 7–22 Strong typing in `DataSet`

```
// this fails at compile time if the name of
// the table should be "customers"
DataTable t = MyDataSet.Customesr;

// so does this, should be custname
String value = MyDataSet.Customers[0].custnam;
```

The easiest way to generate a typed `DataSet` corresponding to an existing database resultset is through Visual Studio or `XSD.exe`, using an existing table, stored procedure, SQL statement, or in the case of `XSD.exe`, an XML schema. In Visual Studio, you can create a typed `DataSet` from any `DataAdapter` object that has been dropped on a form, as shown in Figure 7–2. The Visual Studio designer instantiates the `DataAdapter`, calls `FillSchema` internally, and feeds the results into a code generator (you can produce typed `DataSet`s in C# or VB.NET).

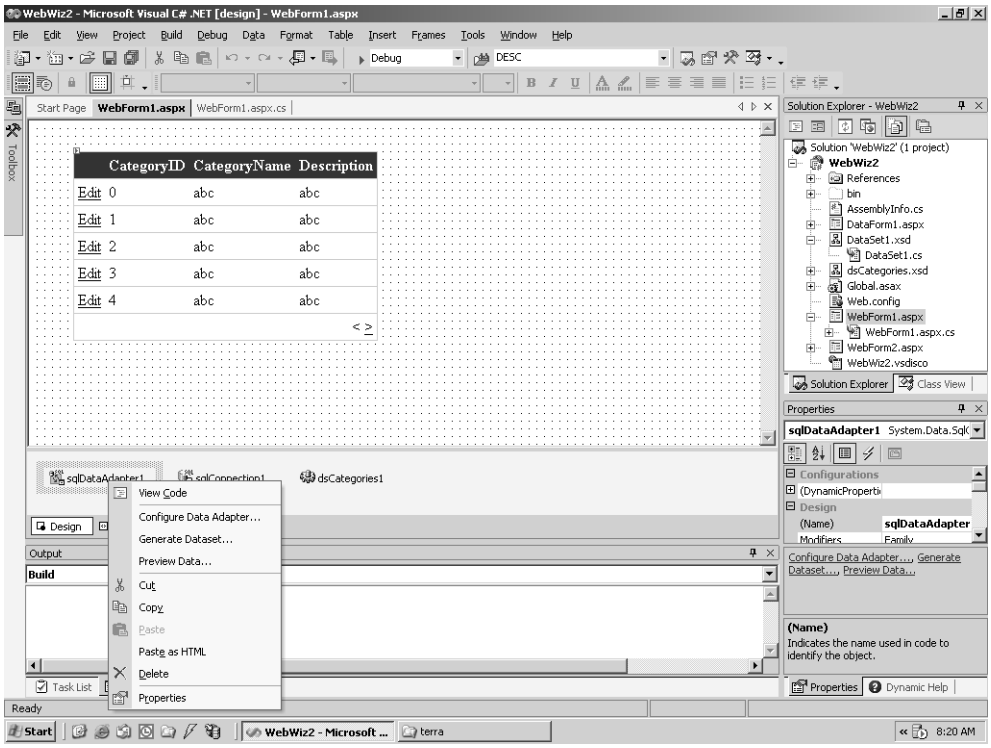

Figure 7–2 Generating a typed `DataSet` in Visual Studio

The manual equivalent of this is to `Fill` a `DataSet`, save the schema with `DataSet.WriteXmlSchema`, and then use the schema as input into `XSD.exe`. For example, let's generate a typed `DataSet` for the simple one-table case shown in Listing 7–23 and see what we get.

Listing 7–23 Producing input for a typed `DataSet`

```
SqlDataAdapter da = new SqlDataAdapter(
  "select * from jobs",
  "server=localhost;uid=sa;database=pubs");

// name the DataSet MyDS
DataSet ds = new DataSet("MyDS");

// name the table MyTable
da.Fill(ds, "MyTable");

ds.WriteXmlSchema("myschema.xsd");
```

Listing 7–24 shows the complete source code for the sample typed `DataSet`.

Listing 7–24 A typed `DataSet` subclass generated by `XSD.exe`

```
//
// This source code was auto-generated by xsd
//
using System;
using System.Data;
using System.Xml;
using System.Runtime.Serialization;


[Serializable()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
[System.Diagnostics.DebuggerStepThrough()]
[System.ComponentModel.ToolboxItem(true)]
public class JobsDS : DataSet {

  private jobsDataTable tablejobs;

    public JobsDS() {
        this.InitClass();
        System.ComponentModel.CollectionChangeEventHandler
          schemaChangedHandler = new
          System.ComponentModel.CollectionChangeEventHandler(
```

```
            this.SchemaChanged);
        this.Tables.CollectionChanged += schemaChangedHandler;
        this.Relations.CollectionChanged += schemaChangedHandler;
    }

    protected JobsDS(SerializationInfo info, StreamingContext context) {
        string strSchema = ((string)(info.GetValue("XmlSchema",
          typeof(string))));
        if ((strSchema != null)) {
            DataSet ds = new DataSet();
            ds.ReadXmlSchema(new XmlTextReader(new
               System.IO.StringReader(strSchema)));
            if ((ds.Tables["jobs"] != null)) {
            this.Tables.Add(new jobsDataTable(ds.Tables["jobs"]));
            }
            this.DataSetName = ds.DataSetName;
            this.Prefix = ds.Prefix;
            this.Namespace = ds.Namespace;
            this.Locale = ds.Locale;
            this.CaseSensitive = ds.CaseSensitive;
            this.EnforceConstraints = ds.EnforceConstraints;
           this.Merge(ds, false, System.Data.MissingSchemaAction.Add);
            this.InitVars();
        }
        else {
            this.InitClass();
        }
        this.GetSerializationData(info, context);
        System.ComponentModel.CollectionChangeEventHandler
          schemaChangedHandler = new
            System.ComponentModel.CollectionChangeEventHandler(
              this.SchemaChanged);
        this.Tables.CollectionChanged += schemaChangedHandler;
        this.Relations.CollectionChanged += schemaChangedHandler;
    }

    [System.ComponentModel.Browsable(false)]
    [System.ComponentModel.DesignerSerializationVisibilityAttribute(
     System.ComponentModel.DesignerSerializationVisibility.Content)]
    public jobsDataTable jobs {
        get {
            return this.tablejobs;
        }
    }
```

```
    public override DataSet Clone() {
        JobsDS cln = ((JobsDS)(base.Clone()));
        cln.InitVars();
        return cln;
    }

    protected override bool ShouldSerializeTables() {
        return false;
    }

    protected override bool ShouldSerializeRelations() {
        return false;
    }

   protected override void ReadXmlSerializable(XmlReader reader)
{
        this.Reset();
        DataSet ds = new DataSet();
        ds.ReadXml(reader);
        if ((ds.Tables["jobs"] != null)) {
           this.Tables.Add(new jobsDataTable(ds.Tables["jobs"]));
        }
        this.DataSetName = ds.DataSetName;
        this.Prefix = ds.Prefix;
        this.Namespace = ds.Namespace;
        this.Locale = ds.Locale;
        this.CaseSensitive = ds.CaseSensitive;
        this.EnforceConstraints = ds.EnforceConstraints;
        this.Merge(ds, false, System.Data.MissingSchemaAction.Add);
        this.InitVars();
    }

    protected override System.Xml.Schema.XmlSchema
      GetSchemaSerializable() {
       System.IO.MemoryStream stream = new System.IO.MemoryStream();
        this.WriteXmlSchema(new XmlTextWriter(stream, null));
        stream.Position = 0;
        return System.Xml.Schema.XmlSchema.Read(new
          XmlTextReader(stream), null);
    }

    internal void InitVars() {
        this.tablejobs = ((jobsDataTable)(this.Tables["jobs"]));
```

```
    if ((this.tablejobs != null)) {
        this.tablejobs.InitVars();
    }
}

private void InitClass() {
    this.DataSetName = "JobsDS";
    this.Prefix = "";
    this.Namespace = "";
    this.Locale = new System.Globalization.CultureInfo
         ("en-US");
    this.CaseSensitive = false;
    this.EnforceConstraints = true;
    this.tablejobs = new jobsDataTable();
    this.Tables.Add(this.tablejobs);
}

private bool ShouldSerializejobs() {
    return false;
}

private void SchemaChanged(object sender,
 System.ComponentModel.CollectionChangeEventArgs e) {
    if ((e.Action ==
 System.ComponentModel.CollectionChangeAction.Remove)) {
        this.InitVars();
    }
}

public delegate void jobsRowChangeEventHandler(object sender,
  jobsRowChangeEvent e);

[System.Diagnostics.DebuggerStepThrough()]
public class jobsDataTable : DataTable,
             System.Collections.IEnumerable {

    private DataColumn columnjob_id;

    private DataColumn columnjob_desc;

    private DataColumn columnmin_lvl;

    private DataColumn columnmax_lvl;
```

```
internal jobsDataTable() :
        base("jobs") {
    this.InitClass();
}

internal jobsDataTable(DataTable table) :
        base(table.TableName) {
  if ((table.CaseSensitive != table.DataSet.CaseSensitive)) {
        this.CaseSensitive = table.CaseSensitive;
    }
    if ((table.Locale.ToString() !=
      table.DataSet.Locale.ToString())) {
        this.Locale = table.Locale;
    }
    if ((table.Namespace != table.DataSet.Namespace)) {
        this.Namespace = table.Namespace;
    }
    this.Prefix = table.Prefix;
    this.MinimumCapacity = table.MinimumCapacity;
    this.DisplayExpression = table.DisplayExpression;
}

[System.ComponentModel.Browsable(false)]
public int Count {
    get {
        return this.Rows.Count;
    }
}

internal DataColumn job_idColumn _
    get {
        return this.columnjob_id;
    }
}

internal DataColumn job_descColumn {
        return this.columnjob_desc;
    }
}

internal DataColumn min_lvlColumn {
    get {
        return this.columnmin_lvl;
    }
```

```
}

internal DataColumn max_lvlColumn {
    get {
        return this.columnmax_lvl;
    }
}

public jobsRow this[int index] {
    get {
        return ((jobsRow)(this.Rows[index]));
    }
}

public event jobsRowChangeEventHandler jobsRowChanged;

public event jobsRowChangeEventHandler jobsRowChanging;

public event jobsRowChangeEventHandler jobsRowDeleted;

public event jobsRowChangeEventHandler jobsRowDeleting;

public void AddjobsRow(jobsRow row)
    this.Rows.Add(row);
}

public jobsRow AddjobsRow(string job_desc, System.Byte min_lvl,
   System.Byte max_lvl) {
    jobsRow rowjobsRow = ((jobsRow)(this.NewRow()));
    rowjobsRow.ItemArray = new object[] {
            null,
            job_desc,
            min_lvl,
            max_lvl};
    this.Rows.Add(rowjobsRow);
    return rowjobsRow;
}

public jobsRow FindByjob_id(short job_id) {
    return ((jobsRow)(this.Rows.Find(new object[] {
                job_id})));
}

public System.Collections.IEnumerator GetEnumerator() {
```

```
    return this.Rows.GetEnumerator();
}

public override DataTable Clone()
  jobsDataTable cln = ((jobsDataTable)(base.Clone()));
    cln.InitVars();
    return cln;
}

protected override DataTable CreateInstance() {
    return new jobsDataTable();
}

internal void InitVars() {
    this.columnjob_id = this.Columns["job_id"];
    this.columnjob_desc = this.Columns["job_desc"];
    this.columnmin_lvl = this.Columns["min_lvl"];
    this.columnmax_lvl = this.Columns["max_lvl"];
}

private void InitClass() {
   this.columnjob_id = new DataColumn("job_id", typeof(short),
       null, System.Data.MappingType.Element);
    this.Columns.Add(this.columnjob_id);
    this.columnjob_desc = new DataColumn("job_desc",
       typeof(string), null, System.Data.MappingType.Element);
    this.Columns.Add(this.columnjob_desc);
    this.columnmin_lvl = new DataColumn("min_lvl",
       typeof(System.Byte), null,
       System.Data.MappingType.Element);
    this.Columns.Add(this.columnmin_lvl);
    this.columnmax_lvl = new DataColumn("max_lvl",
      typeof(System.Byte), null,
      System.Data.MappingType.Element);
    this.Columns.Add(this.columnmax_lvl);
    this.Constraints.Add(new UniqueConstraint
     ("Constraint1", new DataColumn[] {this.columnjob_id}, true));
    this.columnjob_id.AutoIncrement = true;
    this.columnjob_id.AllowDBNull = false;
    this.columnjob_id.ReadOnly = true;
    this.columnjob_id.Unique = true;
    this.columnjob_desc.AllowDBNull = false;
    this.columnjob_desc.MaxLength = 50;
    this.columnmin_lvl.AllowDBNull = false;
```

```
        this.columnmax_lvl.AllowDBNull = false;
    }

    public jobsRow NewjobsRow() {
        return ((jobsRow)(this.NewRow()));
    }

    protected override DataRow NewRowFromBuilder(
      DataRowBuilder builder) {
        return new jobsRow(builder);
    }

    protected override System.Type GetRowType() {
        return typeof(jobsRow);
    }

    protected override void OnRowChanged(
      DataRowChangeEventArgs e)
    {
        base.OnRowChanged(e);
        if ((this.jobsRowChanged != null)) {
            this.jobsRowChanged(this,
              new jobsRowChangeEvent(
                ((jobsRow)(e.Row)), e.Action));
        }
    }

protected override void OnRowChanging(DataRowChangeEventArgs e)
    {
        base.OnRowChanging(e);
        if ((this.jobsRowChanging != null)) {
            this.jobsRowChanging(this,
              new jobsRowChangeEvent(
                ((jobsRow)(e.Row)), e.Action));
        }
    }

    protected override void OnRowDeleted(DataRowChangeEventArgs e) {
        base.OnRowDeleted(e);
        if ((this.jobsRowDeleted != null)) {
            this.jobsRowDeleted(this,
              new jobsRowChangeEvent(
                ((jobsRow)(e.Row)), e.Action));
        }
```

```
    }

    protected override void OnRowDeleting(DataRowChangeEventArgs e)
        base.OnRowDeleting(e);
        if ((this.jobsRowDeleting != null)) {
            this.jobsRowDeleting(this,
              new jobsRowChangeEvent(
                ((jobsRow)(e.Row)), e.Action));
        }
    }

    public void RemovejobsRow(jobsRow row) {
        this.Rows.Remove(row);
    }
}

[System.Diagnostics.DebuggerStepThrough()]
public class jobsRow : DataRow _par
    private jobsDataTable tablejobs;

    internal jobsRow(DataRowBuilder rb) : base(rb) {
        this.tablejobs = ((jobsDataTable)(this.Table));
    }

    public short job_id {
        get {
           return ((short)(this[this.tablejobs.job_idColumn]));
        }
        set {
            this[this.tablejobs.job_idColumn] = value;
        }
    }

    public string job_desc {
        get {
          return ((string)(this[this.tablejobs.job_descColumn]));
        }
        set {
            this[this.tablejobs.job_descColumn] = value;
        }
    }

    public System.Byte min_lvl
```

```
            get {
              return ((System.Byte)
                 (this[this.tablejobs.min_lvlColumn]));
            }
            set {
                this[this.tablejobs.min_lvlColumn] = value;
            }
        }

        public System.Byte max_lvl
            get {
               return ((System.Byte)
                 (this[this.tablejobs.max_lvlColumn]));
            }
            set {
                this[this.tablejobs.max_lvlColumn] = value;
            }
        }
    }

    [System.Diagnostics.DebuggerStepThrough()]
    public class jobsRowChangeEvent : EventArgs {

        private jobsRow eventRow;

        private DataRowAction eventAction;

        public jobsRowChangeEvent(jobsRow row, DataRowAction action) {
            this.eventRow = row;
            this.eventAction = action;
        }

        public jobsRow Row {
            get {
                return this.eventRow;
            }
        }

        public DataRowAction Action {
            get {
                return this.eventAction;
            }
        }
    }
}
```

The typed `DataSet` accomplishes strong typing by generating a class `MyDS`, which derives from `DataSet` (1). The name of the subclass of the `DataSet` class is equal to `DataSet.DataSetName` in the original `DataSet` that produced the XML schema. Four public nested classes are exposed:

- `MyDS.MyTabDataTable:DataTable, IEnumerable`
- `MyDS.MyTabRow:DataRow`
- `MyDS.MyTabRowChangeEvent:EventArgs`
- `MyDS.MyTabRowChangeEventHandler`

where

- `MyDS` is the `DataSet.DataSetName`
- `MyTab` is the `DataTable.TableName`
- `MyTabRow` is `DataTable.TableName + Row`

`MyDS.MyTabDataTable` has a series of private `DataColumn` members; one data member per column is the table or resultset. There are getters for these, but they are marked `internal` because you are not allowed to add or delete `DataColumn`s at runtime. There are also four typed delegates for `Changing`, `Changed`, `Deleting`, and `Deleted` rows.

The typed `DataTable` has the following methods:

- An `Indexer` for `Rows` and a `GetEnumerator` method.
- Two add methods, both called `AddMyTabRow` but each used a little differently. `AddMyTabRow(row)`, which takes a `Row`, is used with `NewMyTabRow`, an empty typed row. `AddMyTabRow(n1,n2,n3)` takes `N parms`, where `N` is the number of columns in the table and `MyTab` is a placeholder. For example, if the table name were equal to `Jobs`, the method name would be `AddJobsRow`.
- `RemoveMyTabRow`, a delete method.

The `DataColumn`s are created and added to the `DataTable` in the `DataTable`'s `InitClass` method. If metadata is available, it is also filled in at that time. If there is a primary key or unique column, there is a method called `FindBykeycolname` that uses the primary key as input.

The `MyDSRow` class exposes columns as public properties. If the column is nullable, there are two predefined helper functions—`IsColumnnameNull` and `SetColumnnameNull`—where `Columnname` is a placeholder for the name of the column.

To delete a `DataRow` provided by strongly typed `DataSet`s, you would use the convenience method `DataRowCollection.Remove` rather than `DataRow.Delete`. But the two methods have different semantics. The difference between the two is that calling `DataRowCollection.Remove` is the same as calling `DataRow.Delete` followed by `AcceptChanges`. If you use `Remove` and then use the `DataSet` to update a database through a `DataAdapter`, the rows that you deleted in the `DataSet` using `Remove` will not be deleted in the database. If this is the desired behavior, you should use `DataRow.Delete` instead of the convenience `RemoveMyTabRow` method.

A strongly typed `DataSet` can also contain more than one table. If you have tables with parent-child relationships—specified by the existence of a `DataRelation` in the `DataSet`'s `Relations` collection—some additional information and methods are generated. When the `DataSet` contains a `Relation`, the following things happen:

- The `PrimaryKey` property is added to `DataColumn` properties for the parent table, a `ForeignKeyConstraint` is added for the child table, and `DataRelation` is added. If the `DataSet`'s `Nested` property was set in the original schema, it is preserved in the typed `DataSet`.
- `ChildTabRow` has a property of type `ParentTabRow`. A property's `get` method calls `GetParentRow`, and the setter calls `SetParent`.
- `ParentRow` has a method, `GetChildTabRow`, that returns an array of typed child rows by calling `GetChildRows`.

where

- `ParentTab` is the `DataTable.TableName` of the parent table.
- `ChildTab` is the `DataTable.TableName` of the child table.

Finally, the strongly typed `DataSet` has certain methods and a property that are related to XML persistence. These override the `DataSet`'s methods.

- A protected constructor takes a `SerializationInfo` info and a `StreamingContext` context. This constructor calls `InitClass` before calling `GetSerializationData`. This is a requirement when you implement `ISerializable`.
- `ReadXmlSerializable` simply calls the base class's `ReadXml` method.
- `GetSchemaSerializable` calls `WriteXmlSchema` to write the schema to an `XmlTextWriter`. Then it reads it back into a `System.Xml.Schema.XmlSchema`. This is similar to the code in the base class (`DataSet`).
- The properties `ShouldSerializeTables` and `ShouldSerializeRelations`, and an additional property called `ShouldSerialize[MyTable]`, return `false`.

The example in Listing 7–25 uses every method of a one-`DataTable` typed `DataSet` and a hierarchical typed `DataSet` with a parent-child relationship. Typed `DataSet`s can also be used as ordinary `DataSet`s, with a corresponding loss of compile-type syntax checking.

Listing 7–25 Using a Typed `DataSet`

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace UseDataSet
{
  class Class1
  {
    static void Main(string[] args)
      {
        Class1 c = new Class1();
        c.instanceMain();
      }

      void instanceMain()
      {
        UseJobsWithDBMS();
        UseJobsDS();
        UseAuTitleDS();
      }
```

```
void UseJobsWithDBMS()
{
  try
  {
    JobsDS j = new JobsDS();
    SqlDataAdapter da = new SqlDataAdapter(
     "select * from jobs",
     "server=localhost;uid=sa;database=pubs");

    SqlCommandBuilder bld = new SqlCommandBuilder(da);
    da.Fill(j.jobs);
    Console.WriteLine(j.jobs.Rows.Count);

    JobsDS.jobsRow found_row = j.jobs.FindByjob_id(156);
    Console.WriteLine(j.jobs.Rows.Count);

    //j.jobs.RemovejobsRow(found_row);
    found_row.Delete();
    Console.WriteLine(j.jobs.Rows.Count);
    da.Update(j.jobs);
  }
  catch (Exception e)
  {
    Console.WriteLine(e.Message);
  }
}

protected void T_Changing(object sender,
  JobsDS.jobsRowChangeEvent e)
{
  if (e.Row.RowState == DataRowState.Deleted)
   Console.WriteLine("Row Changing: Action {0}, State {1}",
      e.Action,  e.Row.RowState);
  else
   Console.WriteLine("Row Changing: {0} id = {1}, State {2}",
      e.Action, e.Row[0], e.Row.RowState);
}

protected void T_Changed(object sender,
  JobsDS.jobsRowChangeEvent e)
{
  if (e.Row.RowState == DataRowState.Detached)
    Console.WriteLine("Row Changed: Action {0}, State {1}",
```

```
      e.Action,  e.Row.RowState);
  else
   Console.WriteLine("Row Changed: {0} id = {1}, State {2}",
      e.Action, e.Row[0], e.Row.RowState);
}

protected void T_Deleting(object sender,
  JobsDS.jobsRowChangeEvent e)
{
 Console.WriteLine("Row Deleting: {0} id = {1}, State {2}",
    e.Action, e.Row[0], e.Row.RowState);
}

protected void T_Deleted(object sender,
  JobsDS.jobsRowChangeEvent e)
{
  Console.WriteLine("Row Deleted: Action {0}, State {1}",
    e.Action, e.Row.RowState);
}

void UseJobsDS()
{
  // 1. One public class JobsDS
  // 2. Four public nested classes:
  //    JobsDS.jobsDataTable;
  //    JobsDS.jobsRow;
  //    JobsDS.jobsRowChangeEvent;
  //    JobsDS.jobsRowChangeEventHandler;

  // Generates one named high-level type
  // the DataSet
  JobsDS j = new JobsDS();
 Console.WriteLine(j.DataSetName);

  // event handlers
  j.jobs.jobsRowChanging +=
    new JobsDS.jobsRowChangeEventHandler(T_Changing);
  j.jobs.jobsRowChanged +=
    new JobsDS.jobsRowChangeEventHandler(T_Changed);
  j.jobs.jobsRowDeleting +=
    new JobsDS.jobsRowChangeEventHandler(T_Deleting);
  j.jobs.jobsRowDeleted +=
    new JobsDS.jobsRowChangeEventHandler(T_Deleted);
```

```
// The DataSet has a single new property named "jobs"
// It's also a public nested class
JobsDS.jobsDataTable t = j.jobs;
Console.WriteLine(j.jobs.TableName);

// add a row
//j.jobs.AddjobsRow(99, "new job", 20, 20);

// when you have metadata, it's smarter about this
// you can't add the identity column
j.jobs.AddjobsRow("new job", 20, 20);

// or add a row
// through the jobsRow public nested class
JobsDS.jobsRow r = j.jobs.NewjobsRow();

// convenience columns
//r.job_id = 100;
r.job_desc = "job 100";
r.max_lvl = 90;
r.min_lvl = 89;

j.jobs.AddjobsRow(r);

// convenience IsNull functions
// only if it can be null
//if (r.Isjob_descNull() == true)
//  Console.WriteLine("desc is null");

// and SetNull functions
//r.Setjob_idNull();

// jobs exposes a public property Count == table.Rows.Count
Console.WriteLine("row count is " + j.jobs.Count);

// convenience find function
JobsDS.jobsRow found_row = j.jobs.FindByjob_id(1)
Console.WriteLine(j.jobs.Rows.Count);

// strongly typed
//j.jobs.RemovejobsRow(r);
j.jobs.RemovejobsRow(found_row);
Console.WriteLine(j.jobs.Rows.Count);
```

```
        // 4 DataColumns as members.
        //j.jobs.job_idColumn;
        //j.jobs.job_descColumn;
        //j.jobs.max_lvlColumn;
        //j.jobs.min_lvlColumn;
        if (j.jobs.job_idColumn.ReadOnly == true)
          Console.WriteLine("its read only");
        Console.WriteLine(j.jobs[0].job_desc);

        // change the first column
        // this would fail, column is readonly
        //j.jobs[0].job_id = 98;
        j.jobs[0].job_desc = "new description";
        j.AcceptChanges();

        j.WriteXmlSchema("theschema.xsd");
        j.WriteXml("thedocument.xml");

        JobsDS j2 = new JobsDS();

        // fails, the typed DataSet already contains the typed
table.
        //j2.ReadXmlSchema("jobsds.xsd");
        j2.ReadXml("jobsds.xml");
      }

      void UseAuTitleDS()
      {
        try
        {
          SqlDataAdapter da = new SqlDataAdapter(
            "select * from authors;select * from titleauthor",
            "server=localhost;uid=sa;database=pubs");

          AuTitleDS om = new AuTitleDS();

          // we still must map these because the
          // mapping is on the DataAdapter
          da.TableMappings.Add("Table", "authors");
          da.TableMappings.Add("Table1", "titleauthors");
          da.Fill(om);
          Console.WriteLine("{0} tables", om.Tables.Count);

          AuTitleDS.authorsRow r = om.authors[0];
```

```
            // get array of children
            AuTitleDS.titleauthorsRow[] cr = r.GettitleauthorsRows();
            foreach (AuTitleDS.titleauthorsRow tr in cr)
            {
              Console.WriteLine("author {0}, title {1}",
                                tr.au_id, tr.title_id);
              AuTitleDS.authorsRow ar = tr.authorsRow;
              Console.WriteLine("author {0} is the parent", ar.au_id);
            }
          }
          catch (Exception e)
          {
            Console.WriteLine(e.Message);
          }
        }
      }
}
```

Although strongly typed `DataSet`s are produced using the names in the schema, you can refine the naming process by using certain schema annotations. These attributes are specified on the element declaration that equates to the table. The annotations are as follows:

- `typedName`: Name of an object referring to a row
- `typedPlural`: Name of an object referring to a table
- `typedParent`: Name of a parent object in a parent-child relationship
- `typedChild`: Name of a child object in a parent-child relationship

There is also an annotation, `nullValue`, that refers to special handling in a strongly typed `DataSet` when the value in the underlying table is `DBNull`.

7.5 The `XmlDataDocument` Class

Having come at the problem of data representation from the point of view of the data storage mechanism—that is, the relational database—we've thus far represented the in-memory object model as though it, too, were a relational database. Chapter 2 touches on the XML Infoset as a different abstraction for data, and you looked at the XML DOM, one of its in-memory data representations. The classes that are used in the relational model parallel those in a relational database; you have a `DataSet` consisting of `DataTable`s, `DataColumn`s, `DataRow`s, and `DataRe-`

lations. You even have a mechanism—DataView—to filter and sort DataTables in memory. The filtering mechanism uses a language similar to SQL.

In the DOM model, XmlDocuments consist of XmlNodes. The XmlDocument can use the XML query language, XPath, to produce either single nodes or nodesets. You can also transform an entire XmlDocument using the XSLT transformation language, producing XML, HTML, or any other format of text output. The .NET class that encapsulates this function is XslTransform. You can traverse the XmlDocument structure either sequentially or using a navigation paradigm. Navigation is represented by a series of classes that implement the XPathNavigator interface. XPathNavigator is optimized for XPath queries; its queries can return XPathNodeIterator or scalar values.

Sometimes it would be useful to integrate these two models—for example, to update a portion of a DOM document based on data in a relational database, or to query a DataSet using XPath as though it were a DOM. The class that lets you treat data as though it were both a DOM and a DataSet, exposing updatability but maintaining consistency in each model, is XmlDataDocument.

The XmlDataDocument class works around the limitation of the strict relational model by enabling partial mapping on DataSet. The DataSet class (and its underlying XML format) works only with homogeneous rowsets or hierarchies, in which all rows contain the same number of columns in the same order. When you attempt to map a document in which columns are missing in rows of the same type, as in Listing 7–26, the XmlRead function compensates by mapping every combination of columns, and setting the ones that do not exist in any level of hierarchy to DBNull.Value.

Listing 7–26 Missing columns in rows

```
<root>
  <document>
     <name>Bob</name>
     <address>111 Any St</address>
  </document>
  <document>
     <name>Bird</name>
     <livesin>tree</livesin>
  </document>
</root>
```

The XML Infoset has no limitation to homogeneous data. When data is semi-structured or contains mixed content (elements and text nodes mixed), as in Listing 7–27, coercing the data into a relational model will not work. An error, "The same table (noun) cannot be the child table in two nested relations.," is produced in this case. You can still integrate, at least partially, XML data that is shaped differently; you use the `DataSet` through the `XmlDataDocument` class. In addition, you can preserve white space and maintain element order in the `XmlDocument`, but when such a document is mapped to a `DataSet`, these extra representation semantics may be lost. XML comments and processing instructions will also be lost in the `DataSet` representation.

Listing 7–27 A document containing mixed content

```
<book>
<chapter>
<title>Testing your <noun>typewriter</noun></title>
<p>The quick brown <noun>fox</noun> jumps over
the lazy <noun>dog</noun></p>
</chapter>
</book>
```

7.5.1 `XmlDataDocument`s and `DataSet`s

As shown in Figure 7–3, an `XmlDataDocument` is an `XmlDocument`. That's because `XmlDataDocument` extends `XmlDocument` and contains a `DataSet` as a member.

Data can be loaded into an `XmlDataDocument` through either the `DataSet` interfaces or the `XmlDocument` interfaces. You can import the relational part of the XML document into `DataSet` by using an explicit or implied mapping schema, as shown in Listing 7–28. Whether changes are made through `DataSet` or through `XmlDataDocument`, the changed values are reflected in both objects. The full-fidelity XML is always available through the `XmlDataDocument`.

Listing 7–28 Loading a `DataSet` through the `XmlDataDocument`

```
XmlDataDocument datadoc = new XmlDataDocument();
datadoc.DataSet.ReadXmlSchema("c:\\authors.xsd");
datadoc.Load ("c:\\authors.xml");

DataSet ds = datadoc.DataSet;
```

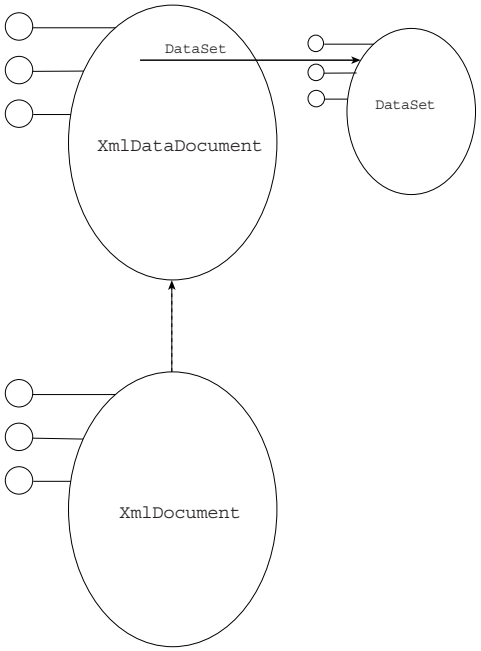

Figure 7–3 `XmlDataDocument` and related classes

```
// use DataSet as usual
foreach (DataTable t in ds.Tables)
  Console.WriteLine(
    "Table " + t.TableName + " is in dataset");
```

In addition to the DOM-style navigation supported by `XmlDocument`, the `XmlDataDocument` adds methods to let you get an `Element` from a `DataRow` or a `DataRow` from an `Element`. Listing 7–29 shows an example.

Listing 7–29 Using `GetElementFromRow`

```
XmlDataDocument datadoc = new XmlDataDocument();

datadoc.DataSet.ReadXmlSchema(
  "c:\\xml_schemas\\cust_orders.xsd");

datadoc.Load(new XmlTextReader(
  "http://localhost/northwind/template/modeauto1.xml"));

XmlElement e = datadoc.GetElementFromRow(
    datadoc.DataSet.Tables[0].Rows[2]);
```

```
Console.WriteLine(e.InnerXml);
```

You can create an `XmlDataDocument` using a prepopulated `DataSet`, as shown in Listing 7–30. Any data in the `DataSet` is used to construct a DOM representation. This DOM is exactly the same as the XML document that would be serialized using `DataSet.WriteXml`. The only difference, a trivial one, is that `DataSet.WriteXml` writes an XML directive at the beginning and `XmlDocument.Save` does not.

Listing 7–30 Creating an `XmlDataDocument` from a `DataSet`

```
DataSet ds = new DataSet();

// load the DataSet
SqlDataAdapter da = new SqlDataAdapter(
          "select * from authors;select * from titleauthor",
          "server=localhost;database=pubs;uid=sa");
da.MissingSchemaAction = MissingSchemaAction.AddWithKey;
da.Fill(ds);

// tweak the DataSet schema
ds.Tables[0].TableName = "authors";
ds.Tables[1].TableName = "titleauthor";
ds.Relations.Add(
   ds.Tables[0].Columns["au_id"],
   ds.Tables[1].Columns["au_id"]);
ds.Relations[0].Nested = true;

XmlDataDocument dd = new XmlDataDocument(ds);

// write the document
dd.Save("c:\\temp\\xmldoc.xml");
// write the dataset
dd.DataSet.WriteXml("c:\\temp\\dataset.xml");
```

An `XmlDataDocument` can also be populated by an XML document through its `Load` method, as shown in Listing 7–31. What makes `XmlDataDocument` unique is that you can load an entire XML document by using `XmlDataDocument.Load`, but the `DataSet` member contains only the tables that existed in the `DataSet`'s schema at the time that you called `Load`. Removing the `ReadXmlSchema` line in Listing 7–31 results in a complete document in the

DOM, but a `DataSet` that, when serialized, contains only an empty root element. Reading a schema that contains only authors will result in a complete DOM and a `DataSet` containing only authors. Attempting to use an "embedded schema plus document"–style XML document produced by using `DataSet.WriteXml` with `XmlWriteMode.WriteSchema`, or using an XSD inline schema recognized with the `XmlValidatingReader`, works to load the document, but this schema is not used to populate the `DataSet`; the `DataSet` contains no data.

Listing 7–31 Loading an `XmlDataDocument` from a document

```
XmlDataDocument dd = new XmlDataDocument();

dd.DataSet.ReadXmlSchema(
   "c:\\xml_schemas\\au_title.xsd");
dd.Load("c:\\xml_documents\\au_title.xml");

// write the document
dd.Save("c:\\temp\\xmldoc.xml");
// write the dataset
dd.DataSet.WriteXml("c:\\temp\\dataset.xml");
```

Here are a few rules to keep in mind when you're using `XmlDataDocument`:

- The mapping of `Document` to `DataSet` (using `XmlDataDocument.DataSet.ReadXmlSchema` or other means) must already be in place when you load the XML document using `XmlDataDocument.Load`.
- Each named piece of data represented in the XML schema can be a child of only one element. In general, this means that an XML schema cannot use global `xsd:element` elements.
- Tables cannot be added to the schema mapping after a document is loaded.
- Documents cannot be loaded after data has been loaded, either through `XmlDataDocument.Load` or by a `DataAdapter`.

You can use `XmlDataDocument` to coerce mixed content and other semistructured data into a somewhat relational form. This technique could help you use the nonrelational document that you looked at in the beginning of this section. The relational schema must be defined so that each simple element type appears unambiguously in any table. Using the "Testing Your Typewriter"

document in Listing 7–27 as an example, you cannot map the document so that `noun` elements appear under both `title` elements and `p` elements, as you saw earlier. Neither can you use a schema that maps `noun` under only `title` or only `p` elements. When you try to use a `DataSet` schema with `noun` only as a child of `p` (hoping to map only nouns that appear in paragraphs), you receive the error "`SetParentRow` requires a child row whose table is `"p"`, but the specified row's `Table` is `title`." The way to map all `noun` elements is to use a schema in which `noun` is not a child of any other element. This produces a single `noun` table containing `noun` elements from (`title`)s and (`p`)aragraphs.

Another possible use for `XmlDataDocument` is to merge new data from a relational database into an existing XML document. This works, but with the limitations noted earlier. Consider a `DataSet` schema containing authors and rows from a nonrectangular document. You would first try to load both documents into the `XmlDataDocument`, but a second document cannot be loaded when the `XmlDataDocument` already contains data. The code in Listing 7–32 fails when trying to load the second document.

Listing 7–32 Merging data with `XmlDataDocument`; this fails

```
try
{
  XmlDataDocument dd = new XmlDataDocument();

  // schema contains authors and document
  dd.DataSet.ReadXmlSchema(@"c:\xml_schemas\au_nonr.xsd");

  // this document contains document
  dd.Load(@"c:\xml_documents\nonrect.xml");

  // this document contains authors
  // this fails
  dd.Load(@"c:\xml_documents\authors_doc.xml");

  foreach (DataTable t in dd.DataSet.Tables)
    Console.WriteLine("table {0} contains {1} rows",
      t.TableName, t.Rows.Count);
}
catch (Exception e)
{
```

```
    Console.WriteLine(e.Message);
  }
```

Attempting to populate the `DataSet` with a `DataAdapter` after a document has been loaded produces partial success. The code in Listing 7–33 produces a `DataSet` containing two tables, but a document containing only the data originally loaded by calling `XmlDataDocument.Load`.

Listing 7–33 Merging documents with `XmlDataDocument`; this doesn't synchronize

```
try
{
  XmlDataDocument dd = new XmlDataDocument();

  // schema contains authors and document
  dd.DataSet.ReadXmlSchema(@"c:\xml_schemas\au_nonr.xsd");

  // this document contains document
  dd.Load(@"c:\xml_documents\nonrect.xml");

  // add authors
  SqlDataAdapter da = new SqlDataAdapter(
    "select * from authors",
    "server=localhost;uid=sa;database=pubs");

  da.Fill(dd.DataSet, "authors");

  // both appear in the DataSet
  foreach (DataTable t in dd.DataSet.Tables)
    Console.WriteLine("Table {0} contains {1} rows",
      t.TableName, t.Rows.Count);

  // no authors in the document
  dd.Save(@"c:\temp\au_nonr.xml");
}
catch (Exception e)
{
  Console.WriteLine(e.Message);
}
```

The correct way to accomplish a merge of two documents is to use two `DataSet`s and the `DataSet.Merge` method. The second `DataSet` can be

either standalone or part of an `XmlDataDocument`, as shown in Listing 7–34. If data is merged into a `DataDocument`'s `DataSet`, however, the schema for both tables must be loaded when the original document is loaded or else an error message will result during `DataSet.Merge`.

Listing 7–34 Merging documents with `XmlDataDocument`; this works

```
XmlDataDocument dd = new XmlDataDocument();

// schema contains authors and document
dd.DataSet.ReadXmlSchema(@"c:\xml_schemas\au_nonr.xsd");

// this document contains document
dd.Load(@"c:\xml_documents\nonrect.xml");

// 1. either of these will work

// add authors
SqlDataAdapter da = new SqlDataAdapter(
  "select * from authors",
  "server=localhost;uid=sa;database=pubs");

DataSet ds = new DataSet();
da.Fill(ds, "authors");
dd.DataSet.Merge(ds);

// 2. either of these will work
XmlDataDocument dd2 = new XmlDataDocument();
dd2.DataSet.ReadXmlSchema(@"c:\xml_schemas\au_nonr.xsd");

// add authors
dd2.Load(@"c:\xml_documents\authors_doc.xml");
dd.DataSet.Merge(dd2.DataSet);

// both appear in the DataSet
foreach (DataTable t in dd.DataSet.Tables)
  Console.WriteLine("Table {0} contains {1} rows",
    t.TableName, t.Rows.Count);

// both in the document
dd.Save(@"c:\temp\au_nonr.xml");
}
```

```
catch (Exception e)
{
  Console.WriteLine(e.Message);
}
```

7.5.2 `XmlDataDocument` and `DataDocumentXPathNavigator`

An additional advantage of using `XmlDataDocument` is that you can query the resulting object model using either the SQL-like syntax of `DataView` filters or the XPath query language. `DataView` filters produce sets of rows and have simple support for parent-child relationships via the `CHILD` keyword. XPath is a more full-featured query language that can produce sets of nodes or scalar values. You can use XPath directly via the `SelectSingleNode` and `SelectNode` methods that `XmlDataDocument` inherits from `XmlDocument`. The `XPathNavigator` class also lets you use precompiled XPath queries. Resultsets from XPath queries are exposed as `XPathNodeIterator`s. You can use `XPathNavigator`s in input to the XSLT transformation process exposed through the `XslTransform` class. You can also use `XPathNavigator`s to update nodes in their source document, using the presence of an `IHasXMLNode` interface on a result node and using `IHasXMLNode.GetNode` to get the underlying (updatable) `XmlNode`.

`DataDocumentXPathNavigator` is a private subclass of `XPathNavigator` that provides cursor-based navigation of the "XML view" of the data in an `XmlDataDocument`. As with the `XPathNavigator` returned by `XmlDocument.CreateNavigator`, multiple navigators can maintain multiple currency positions; in addition, a `DataDocumentXPathNavigator`'s position in the `XmlDocument` is synchronized with its position in the `DataSet`. Programs that depend on positional navigation under classic ADO's client cursor engine or `DataShape` provider can be migrated to this model. Listing 7–35 shows an example of using `XPathNavigator` with `XmlDataDocument`.

Listing 7–35 Using `XPathNavigator`

```
XmlDataDocument datadoc = new XmlDataDocument();

datadoc.Load(new XmlTextReader(
  "http://localhost/nwind/template/modeauto1.xml"));
```

```
XPathNavigator nav = datadoc.CreateNavigator();
XPathNodeIterator i = nav.Select("//customer");
Console.WriteLine(
  "there are {0} customers", i.Count);
```

A final example, Listing 7–36, combines all the `XmlDataDocument` features shown so far. A `DataSet` is created from multiple results obtained from a SQL Server database. An `XmlDataDocument` and an `XPathNavigator` are created over the `DataSet`. Using the `XPathNavigator`, an XPath query returns a set of nodes in the parent (`contract` columns in the `authors` table) based on criteria in the children. The resulting `XPathNodeIterator` is used to update the `XmlDocument` nodes. Because the `XmlDocument` stays synchronized with the `DataSet`, the rows are then updated using a `DataAdapter`.

Listing 7–36 Updating through an `XPathNavigator`

```
DataSet ds = new DataSet("au_info");

// load the DataSet
SqlDataAdapter da = new SqlDataAdapter(
  "select * from authors;select * from titleauthor",
  "server=localhost;database=pubs;uid=sa");
da.MissingSchemaAction = MissingSchemaAction.AddWithKey;
da.Fill(ds);

// tweak the DataSet schema
ds.Tables[0].TableName = "authors";
ds.Tables[1].TableName = "titleauthor";
ds.Relations.Add(
  ds.Tables[0].Columns["au_id"],
  ds.Tables[1].Columns["au_id"]);
ds.Relations[0].Nested = true;

XmlDataDocument dd = new XmlDataDocument(ds);
// This must be set to false
// to edit through the XmlDocument nodes
dd.DataSet.EnforceConstraints = false;
XPathNavigator nav = dd.CreateNavigator();

// get the "contract" column (node)
// for all authors with a royalty percentage < 30%
XPathNodeIterator i = nav.Select(
```

```
 "/au_info/authors/contract[../titleauthor/royaltyper<30]");
while (i.MoveNext() == true)
{
  XmlNode node = ((IHasXmlNode)i.Current).GetNode();
  node.InnerText = "false";
}

SqlCommandBuilder bld = new SqlCommandBuilder(da);
da.Update(dd.DataSet, "authors");
```

7.6 Why Databases and XML?

Relational databases are good for storing data in a controlled, administered manner. They have built-in support for fast concurrent access and optimized set-based query capabilities. However, the protocol and packet formats are database-specific. XML is an almost universally supported method for passing data around. It is supported by heterogeneous architectures; for example, a big-endian Sun workstation can easily parse an XML document created on a little-endian Intel architecture machine. Given that each XML document must have a single root element, it must be somewhat hierarchical by default.

Some database servers have built-in XML features, and the underlying APIs also have built-in integration features. An ASP application can facilitate the sending and receiving of SQL results over HTTP and the formatting of output as XML, optionally adding stylesheets.

A rectangular resultset can easily be stored in XML format for transmission to any platform. Because an XML document can be a hierarchical representation of a complete graph of data, there must be a method to decompose this data when it is stored into the database. Conversely, to serve an XML document as output, it is often useful to compose information from two or more tables into a hierarchy.

7.6.1 XML as a Distinct Type

Object-relational databases and extensions to relational databases let you use an XML document as a distinct type. To do this, you use an XML DataBlade in Informix 9, or an XML Extender in DB2 6.0 and later.

When you use XML as a distinct type, you store the entire XML document as a CLOB column. Special user-defined functions, schemas, tables, and API

extensions allow optimization of the XML type. For example, when a DB2 database is defined as XML `Extender`-aware, an XML `distinct` type is added at the database level. This equates to a CLOB. In addition, a series of user-defined functions (UDFs) and stored procedures is added to the database. These database objects take care of the addition and maintenance of the XML user-defined type (UDT) and keep optional tables of information (called *sidetables*) up-to-date when new XML column instances are added.

7.6.2 Document Composition and Decomposition

XML documents are *organic* types, meaning that they most closely represent a graph of objects in an ODBMS or a network DBMS. You can decompose the information contained in a document into multiple relational tables. Document decomposition can serve to reduce database round-trips because you can pass in the entire document at once and parse it into multiple relational tables.

Going the other way, when you need to present data as an XML document, composition of multiple tables is required. The easiest approach is to use extensions to SQL that know how to produce an XML hierarchy based on the individual tables in a join. Some databases provide extension functions that enable document decomposition. Special logic in stored procedures can be used to store extra data that is provided in the document but does not correspond to a specific relational table.

Document composition is often combined with services provided by APIs and XSLT stylesheets to enable direct output of XML-based HTTP pages and XML-based input formatting through Web browsers. This strategy is used by XML for SQL Server's Internet Services API (ISAPI) application and Oracle's XTK (XML toolkit).

7.7 SQL Server, XML, and Managed Data Access

SQL Server 2000 and the ensuing Web-released extensions, called SQLXML, have many kinds of support for XML. This topic could take an entire book by itself. Almost all the support is available through ADO.NET. First, let's enumerate them and then go over how each one is supported in ADO.NET.

7.7.1 The FOR XML Keyword

SQL Server added a FOR XML keyword to the SQL SELECT statement. This keyword can produce XML in four formats: RAW, AUTO, NESTED, and EXPLICIT. The AUTO, NESTED, and EXPLICIT formats can produce hierarchical nested XML output and attribute-normal or element-normal form. Listing 7–37 shows examples of using SELECT ... FOR XML and the results obtained.

Listing 7–37 Using SQL Server's FOR XML syntax

```
-- 1. raw mode:
-- this query:
SELECT Customers.CustomerID, Orders.OrderID
FROM Customers, Orders
WHERE Customers.CustomerID = Orders.CustomerID
ORDER BY Customers.CustomerID
FOR XML RAW

-- produces this XML output document fragment
  <row CustomerID="ALFKI" OrderID="10643" />
  <row CustomerID="ALFKI" OrderID="10692" />
  <row CustomerID="ALFKI" OrderID="10703" />
  <row CustomerID="ALFKI" OrderID="10835" />
  <row CustomerID="ANATR" OrderID="10308" />

-- 2. auto mode
-- this query:
SELECT Customers.CustomerID, Orders.OrderID
FROM Customers, Orders
WHERE Customers.CustomerID = Orders.CustomerID
ORDER BY Customers.CustomerID
FOR XML AUTO

-- produces the following XML document fragment
  <Customers CustomerID="ALFKI">
    <Orders OrderID="10643" />
    <Orders OrderID="10692" />
    <Orders OrderID="10702" />
    <Orders OrderID="10835" />
  </Customers>
  <Customers CustomerID="ANATR">
    <Orders OrderID="10308" />
  </Customers>
```

```
-- 3. explicit mode
-- this query:
SELECT          1 as Tag, NULL as Parent,
  Customers.CustomerID as [Customer!1!CustomerID],
  NULL as [Order!2!OrderID]
FROM            Customers
UNION ALL
SELECT          2, 1,
                Customers.CustomerID,
                Orders.OrderID
FROM            Customers, Orders
WHERE Customers.CustomerID = Orders.CustomerID
ORDER BY [Customer!1!CustomerID]
FOR XML EXPLICIT

-- produces this output document fragment
<Customer CustomerID="ALFKI">
   <Order OrderID="10643"/>
   <Order OrderID="10692"/>
   <Order OrderID="10702"/>
</Customer>
```

7.7.2 OpenXML

SQL Server 2000 can decompose XML passed in to a stored procedure using a user-defined function, `OpenXML`. This technique uses the normal stored procedure mechanism, so I don't discuss it further.

7.7.3 The `SQLOLEDB` Provider

The `SQLOLEDB` provider (that is, the native OLE DB provider for SQL Server) accepts two new query dialects: XPath and MSSQLXML. MSSQLXML consists of XPath or SQL queries surrounded by XML wrapper elements. Because SQL Server does not support XPath directly, XPath support requires an XML mapping schema that maps an XML view of a single SQL Server database. Multiple tables and relationships are supported in mapping schemas. The `SQLOLEDB` provider also supports streamed input and output. An XSLT transform can automatically be run on the output stream by means of a property on the XML query.

7.7.4 The `SqlXml` Managed Classes

The `SqlXml` set of managed classes, which provide some functionality similar to an ADO.NET data provider, encapsulate all the XML support in the OLE DB provider, listed earlier. We'll talk a lot more about this one.

7.7.5 The SQLXML Web Application

An ISAPI application exposes the ability to obtain an XML result through the HTTP protocol. The URL endpoint exposed can accommodate MSSQLXML templates in files, direct queries, and HTTP `POST` requests. This functionality works by calling the OLE DB provider from within the ISAPI application.

7.7.6 Updategrams

An update to the OLE DB provider accepts an XML dialect called *updategrams*. This functionality works either directly through the provider or through the ISAPI application. Several dialects of updategram are supported. Listing 7–38 shows a sample updategram document. Updategrams are similar in concept to ADO.NET DiffGrams.

Listing 7–38 Updategram formats

```
<DocumentElement
  xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
  xmlns:sql="urn:schemas-microsoft-com:xml-sql">
   <sql:ssync>
   <!-- Deleted -->

  <sql:before>
      <Teachers sql:id="1">
         <ID>0</ID>
         <Name>Mr Apple</Name>
      </Teachers>
   </sql:before>
   <sql:after></sql:after>

   <!-- Unchanged -->

   <sql:before>
      <Teachers sql:id="2">
         <ID>1</ID>
```

```
            <Name>Mrs Blue</Name>
         </Teachers>
      </sql:before>
      <sql:after>
      <Teachers sql:id="2"></Teachers>
      </sql:after>

      <!-- New -->

      <sql:before></sql:before>
      <sql:after>
         <Courses sql:id="7">
            <ID>6</ID>
            <Name>Home Ec 200</Name>
            </Courses>
            </sql:after>

      <!-- Modified -->

         <sql:before>
         <Students sql:id="1">
            <ID>0</ID>
            <Name>Abe</Name>
         </Students>
         </sql:before>
         <sql:after>
         <Students sql:id="1">
            <ID>0</ID>
            <Name>Abby</Name>
         </Students>
         </sql:after>

         <!-- Removed -->

         <sql:before>
            <Students sql:id="2"></Students>
            </sql:before>
            <sql:after>
            </sql:after>
       </sql:ssync>
  </DocumentElement>
```

7.7.7 FOR XML in the SQLXMLOLEDB Provider

A new OLE DB provider, SQLXMLOLEDB, allows the same processing of FOR XML output as the SQLOLEDB provider. The difference is that the FOR XML processing and conversion to XML occur on the client rather than inside SQL Server. This arrangement lets you optimize data transmission because data is transmitted using SQL Server's TDS protocol rather than XML. Because this client-side processing is exposed as an OLE DB service provider, it is possible that it may support providers other than SQLOLEDB in the future.

7.7.8 Bulk Loading

Bulk loading of XML to SQL Server is provided in SQLXML Web release 1. Because this is a COM interface available in .NET only through interoperability, I don't discuss this one further.

7.7.9 Future Support

Future plans for integration of SQL Server and XML include using SOAP as an output protocol (SQLXML3.0) and support of the relatively new XQuery language in addition to SQL and XPath.

7.8 Using SQLXML and .NET

Now let's look at some of these techniques in detail. The SqlClient data provider supports ExecuteXmlReader, a provider-specific method on the SqlCommand class. Rather than provide a SqlDataReader to process the result of a SQL query, ExecuteXmlReader produces an XmlReader, which can be used to directly consume the results of a SELECT ... FOR XML query. The XmlReader might be used directly—for example, to serialize the resulting document to a Stream for transmission to a BizTalk server. The document could be serialized to disk by using an XmlTextReader. It could be read directly into the DataSet by using the DataSet's ReadXml method. Listing 7–39 shows an example. The interesting point of ExecuteXmlReader is that, if you use a FOR XML query that produces nested hierarchies of XML output (AUTO or EXPLICIT mode), it takes only a single SELECT statement to produce multiple DataTables with the appropriate DataRelations in the DataSet.

Listing 7–39 Using SQLXML through ExecuteXmlReader

```
SqlConnection conn = new SqlConnection(
  "server=.;uid=sa;database=pubs");
SqlCommand cmd = new SqlCommand(
  "select * from authors for xml auto, xmldata",
  conn);
conn.Open();

XmlTextReader rdr;
rdr = (XmlTextReader)cmd.ExecuteXmlReader();

DataSet ds = new DataSet();
ds.ReadXml(rdr,
   XmlReadMode.Fragment);
```

When using `ExecuteXmlReader` to obtain an `XmlReader` followed by `DataSet.ReadXml` to populate a `DataSet`, you must take certain precautions because the XML produced by SQL Server does not contain a root element. To obtain all the XML nodes, you must use `XmlReadMode.XmlFragment`, a special `XmlReadMode`. In addition, you must either prepopulate the `DataSet`'s schema with information that matches the incoming fragment or use the `XMLDATA` keyword in your SQL statement to prepend an XDR schema to your fragment. This XDR format schema will prepopulate the `DataSet` schema, as illustrated in Listing 7–40.

Listing 7–40 Using SQLXML through ExecuteXmlReader

```
// 1. This produces no rows
SqlConnection conn = new SqlConnection(
  "server=.;uid=sa;database=pubs");
SqlCommand cmd = new SqlCommand(
  "select * from authors for xml auto",
  conn);
conn.Open();

DataSet ds = new DataSet();
ds.ReadXml(
   (XmlTextReader)cmd.ExecuteXmlReader(),
    XmlReadMode.Fragment);

// 2. This produces 23 rows
SqlConnection conn = new SqlConnection(
```

```
  "server=.;uid=sa;database=pubs");
SqlCommand cmd = new SqlCommand(
  "select * from authors for xml auto, xmldata",
  conn);
conn.Open();

DataSet ds = new DataSet();
ds.ReadXml(
   (XmlTextReader)cmd.ExecuteXmlReader(),
    XmlReadMode.Fragment);

// 3. This produces 23 rows, 2 columns
//    because two columns are mapped
//

SqlConnection conn = new SqlConnection(
  "server=.;uid=sa;database=pubs");
SqlCommand cmd = new SqlCommand(
  "select * from authors for xml auto",
  conn);
conn.Open();

DataSet ds = new DataSet();

DataTable t = new DataTable("authors");
ds.Tables.Add(t);
t.Columns.Add("au_id", typeof(String));
t.Columns.Add("au_fname", typeof(String));

// "for xml" columns are attributes by default
for (int i=0; i<t.Columns.Count; i++)
  t.Columns[i].ColumnMapping =
    MappingType.Attribute;

ds.ReadXml(
   (XmlTextReader)cmd.ExecuteXmlReader(),
    XmlReadMode.Fragment);
```

SQL Server's XML ISAPI application can also be used as an endpoint to produce an `XmlTextReader`. You can then use this `XmlTextReader` to populate the `DataSet`, as shown in Listing 7–41. This method can be executed from any machine that supports .NET. No SQL Server client software need be installed because only ordinary XML is being produced.

Listing 7–41 Using SQL Server 2000's ISAPI application

```
DataSet ds = new DataSet();
XmlTextReader rdr = new XmlTextReader(
   "http://localhost/northwind/template/modeauto1.xml");

ds.ReadXml(rdr);
```

Updategrams are supported by the OLE DB provider or ISAPI application, and although they are similar to DiffGrams, DiffGrams could be used with SQL Server 2000's ISAPI application. (SQLXML Web release 2 adds support for DiffGrams in the ISAPI application.) The updategram format is fairly straightforward and can be created most easily from the information in an updated `DataSet`. This book's Web site contains an example of creating updategrams from a `DataSet` programmatically. Updategrams and DiffGram are especially useful for composing inserts, updates, and deletes to multiple SQL Server tables in a single round-trip to SQL Server.

Although SQL Server's ability to understand MSSQLXML and XPath queries and to use streaming input and output is part of the OLE DB provider, this functionality uses recent extensions to the OLE DB specification introduced in OLE DB version 2.6. The `OleDb` data provider supports most of the "base" OLE DB specification, but it does not support these extensions at all. Instead of adding these extensions to the `OleDb` data provider (they were used only by `SQLOLEDB`), Microsoft released a new set of `SqlXml` managed data classes as part of the SQLXML Release 2 Web release. These classes not only add support for the `SQLOLEDB` 2.6 extensions (by wrapping the original OLE DB code) but also support client-side transformation.

The `SqlXml` data provider does not implement a `Connection` class, implementing only `Command`, `Parameters/Parameter`, and `Adapter`. The special `Adapter` class, `SqlXmlAdapter`, does not derive from `System.Data.Common.DbDataAdapter`. You use the provider to execute a `FOR XML` query by creating a `SqlXmlCommand` and using one of its methods. Three methods of `SqlXmlCommand` produce XML output. `ExecuteStream` produces a new `System.IO.Stream` instance containing the results, as demonstrated in Listing 7–42.

Listing 7–42 Using `SqlXml`'s `ExecuteStream`

```
Stream s;
SqlXmlParameter p;
// note that provider keyword is required
SqlXmlCommand cmd = new SqlXmlCommand(
  "provider=sqloledb;server=localhost;" +
  "uid=sa;database=pubs");
cmd.CommandText =
 "select * from authors where au_lname = ?" +
 " For XML Auto";

p = cmd.CreateParameter();
p.Value = "Ringer";
s = cmd.ExecuteStream();
StreamReader sw = new StreamReader(s);
Console.WriteLine(sw.ReadToEnd());
```

`ExecuteToStream` populates an existing instance of `System.IO.Stream` rather than produce a new one, as shown in Listing 7–43. `SqlXmlCommand` also implements the `ExecuteNonQuery` and `ExecuteXmlReader` methods, which work the same as the corresponding methods in `SqlClient`, adding support for the MSSQLXML and XPath dialects.

Listing 7–43 Using `ExecuteToStream`

```
SqlXmlParameter p;
SqlXmlCommand cmd = new SqlXmlCommand(
  "provider=sqloledb;server=localhost;" +
  "uid=sa;database=pubs");
cmd.CommandText =
 "select * from authors where au_lname = ?" +
 " For XML Auto";

MemoryStream ms = new MemoryStream();
StreamReader sr = new StreamReader(ms);
p = cmd.CreateParameter();
p.Value = "Ringer";
cmd.ExecuteToStream(ms);
ms.Position = 0;
Console.WriteLine(sr.ReadToEnd());
```

`SqlXml` exposes all the extra functionality on the `Command` object that permits using streamed input, using MSSQLXML and XPath queries, specifying XML

mapping schemas for XPath queries, adding XML root elements, and post-processing through an XSL stylesheet. All these are exposed as properties of `SqlXmlCommand`. For example, Listing 7–44 shows how to use an XPath query and XML mapping schema to execute a command on SQL Server and fetch the results.

Listing 7–44 Using an XPath query with `SqlXml`

```
Stream strm;
SqlXmlCommand cmd = new SqlXmlCommand(
  "provider=sqloledb;uid=sa;server=localhost;" +
  "database=northwind");
cmd.CommandText = "Emp";
cmd.CommandType = SqlXmlCommandType.XPath;
cmd.RootTag = "ROOT";
cmd.SchemaPath = "c:\\xml_mappings\\MySchema.xml";
strm = cmd.ExecuteStream();
StreamReader sr = new StreamReader(strm);
Console.WriteLine(sr.ReadToEnd());

<!-- this is MySchema.xml -->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:sql="urn:schemas-microsoft-com:mapping-schema">
  <xsd:element name="Emp" sql:relation="Employees" >
   <xsd:complexType>
     <xsd:sequence>
        <xsd:element name="FName"
                     sql:field="FirstName"
                     type="xsd:string" />
        <xsd:element name="LName"
                     sql:field="LastName"
                     type="xsd:string" />
     </xsd:sequence>
     <xsd:attribute name="EmployeeID" type="xsd:integer" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
<!-- end of MySchema.xml -->
```

Although this is interesting from a "use XML everywhere" point of view, what actually happens when this command is executed is that the `SQLOLEDB` provider processes the XPath statement and mapping schema to produce a SQL `FOR`

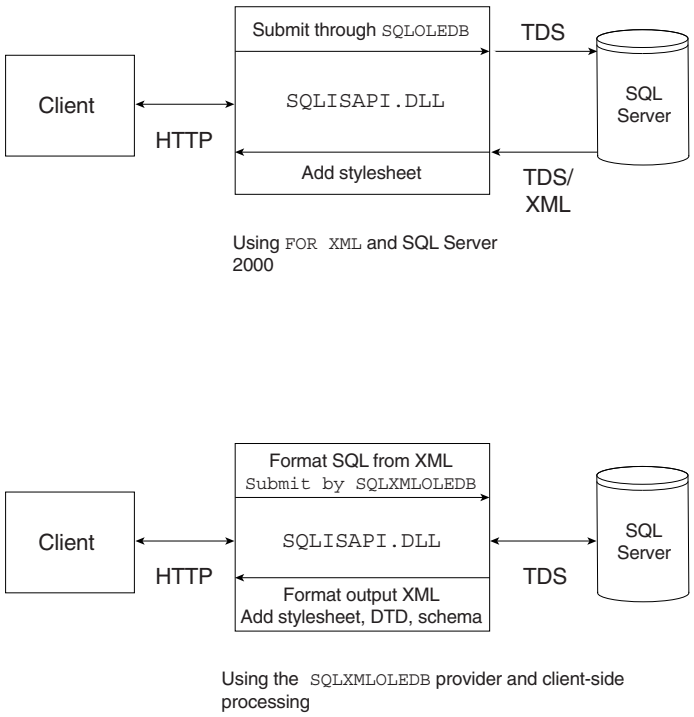

Figure 7–4 Database versus client transformations

`XML EXPLICIT` query, which is sent to SQL Server. In addition, an XML result (wrapped in a TDS packet) is returned from SQL Server to the client. Both of these behaviors might combine to make the execution quite a bit slower than using a SQL query and processing the result into XML (or HTML) on the client. When client processing is preferable, you can specify the `Command.ClientSideXml` property. When you use `Command.ClientSideXml`, the client (usually a Web server) must have SQL Server client libraries installed. The difference in processing is shown in Figure 7–4.

The `SqlXmlAdapter` has three constructors. One takes a single parameter, a `SqlXmlCommand`. The other two take three parameters. The first parameter is either a textual command or a `CommandStream`. The other two parameters are the same in both constructors: a `CommandType` (`SqlXmlCommandType.Sql`, `XPath`,

Template, or TemplateFile), and a ConnectionString. The SqlXmlAdapter implements single Fill and Update methods, each using XML to read or update based on all the DataSet's tables. Listing 7–45 shows an example of using SqlXmlAdapter.

Listing 7–45 Using SqlXmlAdapter

```
SqlXmlAdapter da;
SqlXmlCommand cmd = new SqlXmlCommand(
  "provider=sqloledb;uid=sa;server=localhost;" +
  "database=northwind");
cmd.RootTag = "ROOT";
cmd.CommandText = "Emp";
cmd.CommandType = SqlXmlCommandType.XPath;
cmd.SchemaPath = "MySchema.xml";
//load data set
DataSet ds = new DataSet();
da = new SqlXmlAdapter(cmd);
da.Fill(ds);
DataRow row = ds.Tables["Emp"].Rows[0];
row["FName"] = "Bob";
da.Update(ds);
```

Finally, SQLXML Web Release 2 and the SqlXml data provider support using DiffGrams, in addition to updategrams, to update SQL Server. This is supported both through SqlXmlCommand and through the ISAPI application. When you use SqlXmlCommand, DiffGram is supported as SqlXmlCommandType.Template or TemplateFile. To use a DiffGram to perform updates, you must specify an XSD format mapping schema to map the DiffGram to database tables. Unlike an updategram, the DiffGram format does not include sync elements, so you are more constrained in using transactions than you are with the updategram. Also, when exceptional conditions occur when you update SQL Server through updategrams or DiffGrams, an exception is not thrown in the provider code. Instead, the resulting XML output contains the XML nodes not used, such as the nodes that were used to attempt to add a row to the database where the add failed. Listing 7–46 shows an example of updating using a DiffGram, mapping schema, and HTTP endpoint.

Listing 7–46 Using a DiffGram to update SQL Server

```
try
{
SqlDataAdapter da = new SqlDataAdapter(
  "select CustomerID, CompanyName, " +
  "ContactName from customers",
  "server=localhost;uid=sa;database=northwind");

DataSet ds = new DataSet();
da.Fill(ds, "Customers");

// map this to an XML Attribute
// to match the mapping schema
ds.Tables[0].Columns[0].ColumnMapping =
   MappingType.Attribute;

// update the ninth row
ds.Tables[0].Rows[9][1] = "new customer name";
DataSet ds2 = ds.GetChanges();

HttpWebRequest r = (HttpWebRequest)WebRequest.Create(
  "http://zmv43/northwind/");
r.ContentType = "text/xml";
r.Method = "POST";

// MUST add mapping schema reference
String rootelem = "<ROOT " +
 "xmlns:sql='urn:schemas-microsoft-com:xml-sql'" +
 " sql:mapping-schema='diffgram1.xml'>";

String rootend = "</ROOT>";

StreamWriter s = new StreamWriter(
  r.GetRequestStream());
s.Write(rootelem, 0, rootelem.Length);
ds2.WriteXml(s, XmlWriteMode.DiffGram);
s.Write(rootend, 0, rootend.Length);
s.Close();

HttpWebResponse resp =
    (HttpWebResponse)r.GetResponse();
StreamReader rdr = new StreamReader(
     resp.GetResponseStream());
```

```
Console.WriteLine(rdr.ReadToEnd());
}
catch (Exception e)
{
  Console.WriteLine(e.Message);
}

<!-- here's the mapping-schema -->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:sql="urn:schemas-microsoft-com:mapping-schema">

<xsd:annotation>
  <xsd:documentation>
    Diffgram Customers/Orders Schema.
  </xsd:documentation>
</xsd:annotation>

<xsd:element name="Customers" sql:relation="Customers">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="CompanyName" type="xsd:string"/>
      <xsd:element name="ContactName" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="CustomerID"
     type="xsd:string" sql:field="CustomerID"/>
  </xsd:complexType>
</xsd:element>

</xsd:schema>
<!-- end of mapping-schema -->
```

7.9 Where Are We?

You've completed your in-depth exploration of ADO.NET by looking at the XML capabilities built into the data access stack at all levels. You've looked at integration of ADO.NET and XML in `DataSet` (including using XSD schemas to generate typed `DataSet`s) and the `XmlDocument`/`DataSet` hybrid called `XmlDataDocument`, including its implementation of an `XPathNavigator` class. Finally, you've seen that SQL Server directly supports `SELECT`, `INSERT`, `UPDATE`, and `DELETE` operations using XML. This support is built into two data providers: the `SqlClient` data provider and the new `SqlXml` managed classes.

What I hope you take away from this exposition is that ADO.NET not only supports relational data through the data provider, `DataSet`, and `Adapter` architecture but also adds support for all types of nonrelational data through its integration with XML. ADO.NET provides a wide integration layer between relational and nonrelational data via XML and also provides direct XML support of relational data.

You've learned the basic concepts of ADO.NET, but unless you started programming data access yesterday, you already have some data access code written using some other API. You may even have code that exposes your data through OLE DB providers. Chapter 8 discusses strategies for provider writers in the new .NET world and explores the ways existing OLE DB providers work with the `OleDb` managed provider. Chapter 9 explains migration strategies for consumer writers who use existing APIs.