CHAPTER 12

XML Testing

12.0 Introduction

This chapter presents a variety of test automation techniques that involve XML data. The most common XML-related tasks in test automation situations are reading/parsing test case data that has been stored as XML, writing test results to external storage as XML, programmatically modifying XML files to match a new test harness or output format, validating XML files, and comparing two XML files for equality to determine a test case pass/fail result. The screenshot in Figure 12-1 demonstrates XML validation and parsing. The program that generated the output shown in Figure 12-1 is presented in Section 12.12.

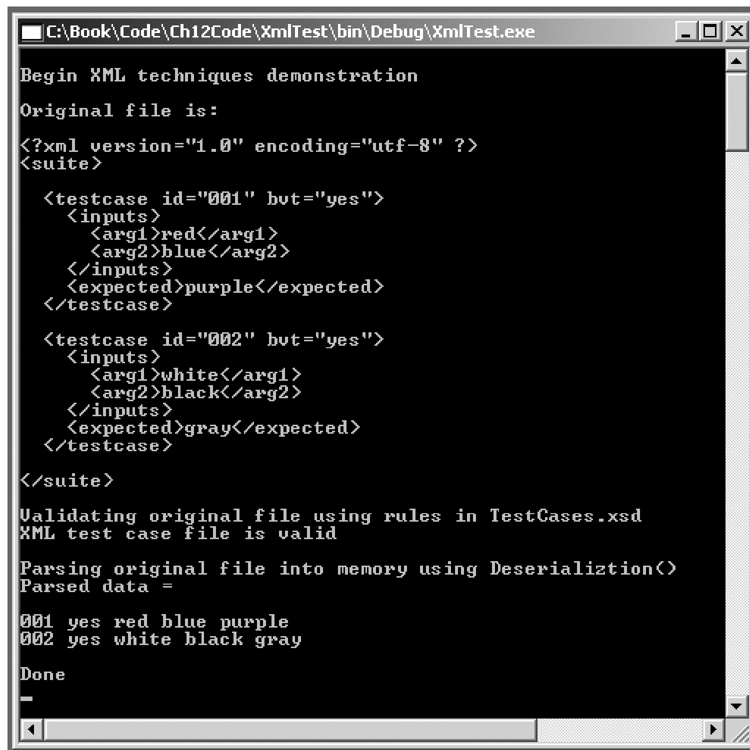

Figure 12-1. *Validating and parsing an XML file*

Most of the example code in this chapter will use a slightly expanded version of the file shown in Figure 12-1:

```
<?xml version="1.0" encoding="utf-8" ?>
<suite>

  <testcase id="001" bvt="yes">
    <inputs>
      <arg1>red</arg1>
      <arg2>blue</arg2>
    </inputs>
    <expected>purple</expected>
  </testcase>

  <testcase id="002" bvt="no">
    <inputs>
      <arg1>blue</arg1>
      <arg2>yellow</arg2>
    </inputs>
    <expected>green</expected>
  </testcase>

  <testcase id="003" bvt="yes">
    <inputs>
      <arg1>white</arg1>
      <arg2>black</arg2>
    </inputs>
    <expected>gray</expected>
  </testcase>

</suite>
```

The preceding example is a dummy XML file of hypothetical test case data. Notice that XML data is stored as either an element (such as `<arg1>red</arg1>`) or as an attribute in an element (such as `<testcase id="001" bvt="yes">`). Dealing with elements and attributes, and with a nested/hierarchical structure, are key tasks when working with XML. The five parsing techniques in this chapter (Sections 12.1 through 12.5) all parse file testCases.xml into a test case `Suite` object defined as:

```
namespace Utility
{
  public class TestCase
  {
    public string id;
    public string bvt;
    public string arg1;
    public string arg2;
    public string expected;
  }
```

```
  public class Suite
  {
    public ArrayList cases = new ArrayList();
    public void Display()
    {
      foreach (TestCase tc in cases)
      {
        Console.Write(tc.id + " " + tc.bvt + " " + tc.arg1 + " ");
        Console.WriteLine(tc.arg2 + " " + tc.expected);
      }
    }
  } // class Suite
} // ns Utility
```

The `TestCase` class represents a single test case and the `Suite` class represents a collection of `TestCase` objects. Encapsulating test case data in this way instead of using individual variables usually makes your test harnesses easier to maintain.

12.1 Parsing XML Using XmlTextReader

Problem

You want to parse an XML file using the `XmlTextReader` class.

Design

Iterate through each node of the XML file using the `Read()` and `ReadElementString()` methods of the `XmlTextReader` class. Use the `GetAttribute()` method to fetch attribute data, and use the return value from `ReadElementString()` to fetch element data.

Solution

This code parses file testCases.xml (shown in the introduction to this chapter) into a `Suite` collection of `TestCase` objects (also shown in the introduction):

```
Console.WriteLine("Start\n");

Utility.Suite suite = new Utility.Suite();

XmlTextReader xtr = new XmlTextReader("..\\..\\testCases.xml");
xtr.WhitespaceHandling = WhitespaceHandling.None;
xtr.Read(); // read the XML declaration node, advance to <suite> tag

while (!xtr.EOF) //load loop
{
  if (xtr.Name == "suite" && !xtr.IsStartElement()) break;
```

```
  while (xtr.Name != "testcase" || !xtr.IsStartElement() )
  xtr.Read(); // advance to <testcase> tag

  Utility.TestCase tc = new Utility.TestCase();
  tc.id = xtr.GetAttribute("id");
  tc.bvt = xtr.GetAttribute("bvt");
  xtr.Read(); // advance to <inputs> tag
  xtr.Read(); // advance to <arg1> tag
  tc.arg1 = xtr.ReadElementString("arg1"); // consumes the </arg1> tag
  tc.arg2 = xtr.ReadElementString("arg2"); // consumes the </arg2> tag
  xtr.Read(); // advance to <expected> tag
  tc.expected = xtr.ReadElementString("expected"); // consumes </expected> tag
  // we are now at an </testcase> tag
  suite.cases.Add(tc);
  xtr.Read(); // and now either at <testcase> tag or </suite> tag
} // load loop

xtr.Close();
suite.Display(); // show the suite of TestCases

Console.WriteLine("\nDone");
```

When run, this solution will produce this output:

```
Start

001 yes red   blue   purple
002 no  blue  yellow green
003 yes white black  gray

Done
```

The XML file has been parsed into its individual data pieces which can then be used as needed, typically as input to a method under test.

The key to understanding this solution is to understand the `Read()` and `ReadElementString()` methods of `XmlTextReader`. To an `XmlTextReader` object, an XML file is a sequence of nodes. For example:

```
<?xml version="1.0" ?>
<alpha id="001">
  <beta>123</beta>
</alpha>
```

There are six nodes, without counting whitespace: the XML declaration, `<alpha id="001">`, `<beta>`, `123`, `</beta>`, and `</alpha>`. Notice that attributes (`id="001"`) are not considered XML nodes by an `XmlTextReader` object.

The Read() method advances one node at a time. Unlike many Read() methods in other classes, the System.XmlTextReader.Read() method does not return significant data. The ReadElementString() method on the other hand returns the data between begin and end tags of its argument and advances to the next node after the end tag. Because XML attributes are not nodes, we have to extract attribute data using the GetAttribute() method. The statement:

```
xtr.WhitespaceHandling = WhitespaceHandling.None;
```

is important. It instructs the XmlTextReader object to ignore whitespace characters such as blanks, tabs, and newlines. Without this statement you would have to Read() over all whitespace, which is very troublesome and error-prone.

Comments

The main loop in this solution:

```
while (!xtr.EOF) //load loop
{
  if (xtr.Name == "suite" && !xtr.IsStartElement()) break;
  // etc.
}
```

is not particularly elegant but is more readable than alternatives. The loop exits when at EOF or a </suite> tag.

When marching through an XML file, you can either Read() your way one node at a time, or get a bit more sophisticated with code such as this:

```
while (xtr.Name != "testcase" || !xtr.IsStartElement() )
  xtr.Read(); // advance to <testcase> tag
```

The choice of technique you use is mostly a matter of style. Parsing an XML file with XmlTextReader has a traditional, pre-.NET feel to it. You walk sequentially through the file using Read(), and extract data with ReadElementString() and GetAttribute(). Using XmlTextReader is straightforward and effective and is appropriate when the structure of the XML file being parsed is relatively simple and consistent, and when you only need to process the XML in a forward-only manner. In general, using XmlTextReader is the fastest technique when compared with the other parsing techniques in this chapter. Notice that because the logic in this solution depends quite a bit on the XML file having a consistent structure, using XmlTextReader is usually not a good idea if your XML file has an inconsistent structure. Compared to other parsing techniques in this chapter, XmlTextReader operates at a lower level of abstraction, meaning it is up to you as a programmer to keep track of where you are in the XML file and Read() correctly.

12.2 Parsing XML Using XmlDocument

Problem

You want to parse an XML file using the XmlDocument class.

Design

Read the entire XML file into memory using the `XmlDocument.Load()` method. Fetch node collections using the `SelectNodes()` method then use the `Attributes.GetNamedItem()` and `SelectSingleNode()` methods combined with the `InnerText` property to get the values of attributes and elements.

Solution

This code parses file testCases.xml (shown in the introduction to this chapter) into a `Suite` collection of `TestCase` objects (also shown in the introduction):

```
Utility.Suite suite = new Utility.Suite();

XmlDocument xd = new XmlDocument();
xd.Load("..\\..\\testCases.xml");

// get all <testcase> nodes
XmlNodeList nodelist = xd.SelectNodes("/suite/testcase");
foreach (XmlNode node in nodelist) // for each <testcase> node
{
  Utility.TestCase tc = new Utility.TestCase();

  tc.id = node.Attributes.GetNamedItem("id").Value;
  tc.bvt = node.Attributes.GetNamedItem("bvt").Value;

  XmlNode n = node.SelectSingleNode("inputs"); // get <inputs> node
  tc.arg1 = n.ChildNodes.Item(0).InnerText;
  tc.arg2 = n.ChildNodes.Item(1).InnerText;

  tc.expected = node.ChildNodes.Item(1).InnerText;

  suite.cases.Add(tc);
} // foreach <testcase> node

suite.Display();
```

When run, this solution will produce the exact same output as Section 12.1 (parsing with `XmlTextReader`):

```
Start

001 yes red   blue   purple
002 no  blue  yellow green
003 yes white black  gray

Done
```

XmlDocument objects are based on the notion of XML nodes and child nodes. Instead of sequentially navigating through a file, we select sets of nodes with the SelectNodes() method, or individual nodes with the SelectSingleNode() method. Notice that because XML distinguishes between attributes and elements, we must get the id and bvt attribute values with an Attributes.GetNamedItem() method applied to an element node.

Comments

After loading the XmlDocument, we fetch all the testcase nodes at once with:

```
XmlNodeList nodelist = xd.SelectNodes("/suite/testcase");
```

Then we iterate through this list of nodes and fetch each <input> node with:

```
XmlNode n = node.SelectSingleNode("inputs");
```

and then extract the arg1 (and similarly arg2) value using:

```
tc.arg1 = n.ChildNodes.Item(0).InnerText;
```

In this statement, n is the <inputs> node, ChildNodes.Item(0) is the first element of <inputs>, i.e., <arg1>, and the InnerText property gets the value between <arg1> and </arg1>.

The XmlDocument class is modeled on the W3C XML Document Object Model and may have a somewhat different feel to it than many .NET Framework classes that you are familiar with. Using the XmlDocument class is appropriate if you need to extract data in a nonsequential manner, or if you are already using XmlDocument objects and want to maintain a consistent approach to your test harness code. Because using XmlDocument reads an entire XML document into memory at the same time, using it may not be suitable in situations where the XML file being parsed is very, very large.

In addition to the XmlDocument class, the System.Xml namespace contains a closely related XmlDataDocument class. It is derived from the XmlDocument class and is primarily intended for use in conjunction with DataSet objects. So, in this solution, we could have used the XmlDataDocument class but we would not have gained any advantage by doing so.

12.3 Parsing XML with XPathDocument

Problem

You want to parse an XML file using the XPathDocument class.

Design

Read the entire XML file into memory using the XPathDocument() constructor. Create an XPathNodeIterator object and use it to move through the XPathDocument object with the MoveNext() method. Fetch attribute values using the GetAttibute() method. Fetch element values using the SelectChildren() method and the Current.Value property.

Solution

This code parses file testCases.xml (shown in the introduction to this chapter) into a `Suite` collection of `TestCase` objects (also shown in the introduction):

```
Utility.Suite suite = new Utility.Suite();

XPathDocument xpd = new XPathDocument("..\\..\\testCases.xml");
XPathNavigator xpn = xpd.CreateNavigator();
XPathNodeIterator xpi = xpn.Select("/suite/testcase");

while (xpi.MoveNext()) // each testcase node
{
  Utility.TestCase tc = new Utility.TestCase();
  tc.id = xpi.Current.GetAttribute("id", xpn.NamespaceURI);
  tc.bvt = xpi.Current.GetAttribute("bvt", xpn.NamespaceURI);

  XPathNodeIterator tcChild =
    xpi.Current.SelectChildren(XPathNodeType.Element);
  while (tcChild.MoveNext()) // each part of <testcase>
  {
    if (tcChild.Current.Name == "inputs")
    {
      XPathNodeIterator tcSubChild =
 tcChild.Current.SelectChildren(XPathNodeType.Element);
      while (tcSubChild.MoveNext()) // each part of <inputs>
      {
        if (tcSubChild.Current.Name == "arg1")
          tc.arg1 = tcSubChild.Current.Value;
        else if (tcSubChild.Current.Name  == "arg2")
          tc.arg2 = tcSubChild.Current.Value;
      }
    }
    else if (tcChild.Current.Name == "expected")
      tc.expected = tcChild.Current.Value;
  }
  suite.cases.Add(tc);

} // each testcase node

suite.Display();
```

When run, this solution will produce the same output as Section 12.1 (parsing with `XmlTextReader`) and Section 12.2 (parsing with `XmlDocument`):

```
Start

001 yes red   blue   purple
002 no  blue  yellow green
003 yes white black  gray

Done
```

After loading the `XPathDocument` object, we get what is, in essence, a reference to the first `<testcase>` node into an `XPathNodeIterator` object with:

```
XPathNavigator xpn = xpd.CreateNavigator();
XPathNodeIterator xpi = xpn.Select("/suite/testcase");
```

Because `XPathDocument` does not maintain "node identity," we must iterate through each `<testcase>` node with this loop:

```
while (xpi.MoveNext())
```

Similarly, we have to iterate through the children nodes with:

```
while (tcChild.MoveNext())
```

Comments

Using an `XPathDocument` object to parse XML has a hybrid feel that is part procedural and lower-level (as in `XmlTextReader`), and part object oriented and higher-level (as in `XmlDocument`). You can select parts of the document using the `Select()` method of an `XPathNavigator` object and also move through the document using the `MoveNext()` method of an `XPathNodeIterator` object.

The `XPathDocument` class is optimized for `XPath` data model queries. So using it is particularly appropriate when the XML file to parse is deeply nested, has a complex structure, or requires extensive searching. You might also consider using `XPathDocument` if other parts of your test harness code use that class, so that you maintain a consistent coding look and feel. An `XPathDocument` object is read-only, so using `XPathDocument` is not appropriate if you want to do any direct, in-memory processing of the XML file you are parsing.

12.4 Parsing XML with XmlSerializer

Problem

You want to parse an XML file using the `XmlSerializer` class.

Design

Prepare a class that is defined so it will accept the result of calling the `Deserialize()` method of the `XmlSerializer` class. Then create an instance of the receptacle class and use `Deserialize()` with a `StreamReader` object.

Solution

This code parses file testCases.xml (shown in the introduction to this chapter) into a `Suite` collection of `TestCase` objects (also shown in the introduction):

```
XmlSerializer xs = new XmlSerializer(typeof(SerializerLib.Suite));
StreamReader sr = new StreamReader("..\\..\\testCases.xml");
SerializerLib.Suite suite = (SerializerLib.Suite)xs.Deserialize(sr);
sr.Close();
suite.Display();
```

where:

```
namespace SerializerLib
{
  [XmlRootAttribute("suite")]
  public class Suite
  {
    [XmlElementAttribute("testcase")]
    public TestCase[] items; // changed name from xsd-generated code
    public void Display() // added to xsd-generated code
    {
      foreach (TestCase tc in items)
      {
        Console.Write(tc.id + " " + tc.bvt + " "  + tc.inputs.arg1 + " ");
        Console.WriteLine(tc.inputs.arg2 + " " + tc.expected);
      }
    }
  }

  public class TestCase  // changed name from xsd-generated code
  {
    [XmlAttributeAttribute()]
    public string id;
    [XmlAttributeAttribute()]
    public string bvt;
    [XmlElementAttribute("inputs")]
    public Inputs inputs; // change from xsd-generated code: no array
    public string expected;
  }

  public class Inputs // changed name from xsd-generated code
  {
    public string arg1;
    public string arg2;
  }
} // ns SerializerLib
```

When run, this solution will produce the same output as in Section 12.1 (parsing with `XmlTextReader`), Section 12.2 (parsing with `XmlDocument`), and Section 12.3 (parsing with `XPathDocument`):

```
Start

001 yes red   blue   purple
002 no  blue  yellow green
003 yes white black  gray

Done
```

Using the `XmlSerializer` class is significantly different from using any of the other five fundamental classes that parse XML, because the in-memory data store must be carefully prepared beforehand. Observe that pulling the XML data into memory is accomplished in a single statement:

```
SerializerLib.Suite suite = (SerializerLib.Suite)xs.Deserialize(sr);
```

This example uses a `SerializerLib` namespace to hold the definition for a `Suite` class that corresponds to the testCases.xml file so that the `XmlSerializer` object can store the XML data into it. The trick of course is to set up this `Suite` class.

Comments

There are two ways to create a class that is defined so it will accept the result of calling the `Deserialize()` method of the `XmlSerializer` class. The first way is to carefully examine the structure of the source XML file and then code the destination/receptacle class by hand. A much easier approach is to use the xsd.exe command line tool that ships with Visual Studio .NET. First, (assuming file testCases.xml is in the C: folder) issue the command:

```
C:\>xsd.exe testCases.xml /o:.
```

This means create an XSD schema definition of file testCases.xml and save the result with default name testCases.xsd in the current directory. The intermediate .xsd file will contain a complete structure definition of the XML file. Next, issue the command:

```
C:\>xsd.exe testCases.xsd /c /o:.
```

This means use the testCases.xsd definition file to generate a set of class definitions that are compatible with the `Deserialize()` method, using the default C# language, and save with default name testCases.cs in the current directory. Here is the original testCases.cs before some editing:

```
[System.Xml.Serialization.XmlRootAttribute(Namespace="",
  IsNullable=false)]
public class suite
```

```
{
  [System.Xml.Serialization.XmlElementAttribute("testcase",
    Form=System.Xml.Schema.XmlSchemaForm.Unqualified)]
  public suiteTestcase[] Items;
}

public class suiteTestcase
{
  [System.Xml.Serialization.XmlElementAttribute(Form=
    System.Xml.Schema.XmlSchemaForm.Unqualified)]
  public string expected;

  [System.Xml.Serialization.XmlElementAttribute("inputs",
    Form=System.Xml.Schema.XmlSchemaForm.Unqualified)]
  public suiteTestcaseInputs[] inputs;

  [System.Xml.Serialization.XmlAttributeAttribute()]
  public string id;

  [System.Xml.Serialization.XmlAttributeAttribute()]
  public string bvt;
}
public class suiteTestcaseInputs
{
  [System.Xml.Serialization.XmlElementAttribute(Form=
    System.Xml.Schema.XmlSchemaForm.Unqualified)]
   public string arg1;

  [System.Xml.Serialization.XmlElementAttribute(Form=
    System.Xml.Schema.XmlSchemaForm.Unqualified)]
  public string arg2;
}
```

If you examine the resulting class definition code carefully, you will eventually be able to see the relationship between the code and the original XML file:

```
<suite>

  <testcase id="001" bvt="yes">
    <inputs>
      <arg1>red</arg1>
      <arg2>blue</arg2>
    </inputs>
    <expected>purple</expected>
  </testcase>

  (other <testcase> data here)

</suite>
```

At this point you can copy and paste the newly created class definitions directly into your test harness and use them as is to instantiate an object to receive the result of the Deserialize() method. Alternatively, you can edit the auto-generated file by removing unneeded code, changing names to those that better match your original XML file, and adding additional methods (such as a display method or get and set properties). The class definition in the solution to this section was created using this approach.

Using the XmlSerializer class provides a very elegant solution to the problem of parsing an XML file. Compared with the other four techniques in this chapter, XmlSerializer operates at the highest level of abstraction, meaning that the algorithmic details are largely hidden from you. But this gives you somewhat less control over the XML parsing process.

Using XmlSerializer for parsing is most appropriate for situations when fine-grained control is not required, the test harness program does not make extensive use of XmlDocument objects, the XML file is relatively shallow rather than deeply nested, and the application is not primarily an ADO.NET application.

12.5 Parsing XML with a DataSet Object

Problem

You want to parse an XML file using a DataSet object.

Design

Read the entire XML file into a DataSet object using the ReadXml() method. Then iterate through each DataTable in the DataSet, and extract related data by using the GetChildRows() method in conjunction with table relation names.

Solution

This code parses file testCases.xml (shown in the introduction to this chapter) into a Suite collection of TestCase objects (also shown in the introduction):

```
DataSet ds = new DataSet();
ds.ReadXml("..\\..\\testCases.xml");

Utility.Suite suite = new Utility.Suite();
foreach (DataRow row in ds.Tables["testcase"].Rows)
{
  Utility.TestCase tc = new Utility.TestCase();
  tc.id = row["id"].ToString();
  tc.bvt = row["bvt"].ToString();
  tc.expected = row["expected"].ToString();

  DataRow[] children = row.GetChildRows("testcase_inputs"); // relation name
```

```
  tc.arg1 = (children[0]["arg1"]).ToString(); // there is only 1 row in children
  tc.arg2 = (children[0]["arg2"]).ToString();

  suite.cases.Add(tc);
}

suite.Display();
```

When run, this solution will produce the same output as in Section 12.1 (parsing with `XmlTextReader`), Section 12.2 (parsing with `XmlDocument`), Section 12.3 (parsing with `XPathDocument`), and Section 12.4 (parsing using `XmlSerializer`):

```
Start

001 yes red   blue   purple
002 no  blue  yellow green
003 yes white black  gray

Done
```

We start by reading the XML file directly into a `System.Data.DataSet` object using the `ReadXml()` method. A `DataSet` object can be thought of as an in-memory relational database. The key to parsing XML using a `DataSet` object is to understand how XML, which is inherently hierarchical, is mapped to a set of `DataTable` objects, which are inherently flat. Each level of the source XML file will generate a table in the `DataSet`. Recall the structure of the source XML file:

```
<suite>

  <testcase id="001" bvt="yes">
    <inputs>
      <arg1>red</arg1>
      <arg2>blue</arg2>
    </inputs>
    <expected>purple</expected>
  </testcase>

  (other <testcase> nodes

</suite>
```

The top-level, `<testcase>`, produces a `DataTable` named `testcase`. The next level, `<inputs>`, produces a `DataTable` named `inputs`. A relation named `testcase_inputs` is created which links the `DataTable` objects. Notice that the XML root level does not generate a table and that the lowest level (in his case the `<arg>` data) does not generate a table either.

Comments

In practice, when parsing XML using a `DataSet` object, a good approach is to do some preliminary investigation. Although you could create a custom `DataSet` object with completely known characteristics, it is much quicker to let the `ReadXml()` method do the work and then examine the result. Read the source XML file into a `DataSet` and then programmatically examine it to determine the number and names of the `DataTable` objects that are created. This utility method will usually reveal all the information you need:

```
// names of tables, columns, relations in ds
public static void DisplayInfo(DataSet ds)
{
  foreach (DataTable dt in ds.Tables)
  {
    Console.WriteLine("\n==========================================");
    Console.WriteLine("Table = " + dt.TableName + "\n");
    foreach (DataColumn dc in dt.Columns)
    {
      Console.Write("{0,-14}", dc.ColumnName);
    }
    Console.WriteLine("\n------------------------------------------");

    foreach (DataRow dr in dt.Rows)
    {
      foreach (object data in dr.ItemArray)
      {
        Console.Write("{0,-14}", data.ToString());
      }
      Console.WriteLine();
    }
    Console.WriteLine("==========================================");
  } // foreach DataTable

  foreach (DataRelation dr in ds.Relations)
  {
    Console.WriteLine("\n\nRelations:");
    Console.WriteLine(dr.RelationName + "\n\n");
  }

} // DisplayInfo()
```

The first table, `testcase`, holds the data that is one level deep from the XML root: `id`, `bvt`, and `expected`. The second table, `inputs`, holds data that is two levels deep: `arg1` and `arg2`. In general if your XML file is n levels deep, `ReadXml()` will generate n-1 tables (or n-2 tables, depending on your exact definition of levels).

Extracting the data from the parent `testcase` table is easy. Just iterate through each row of the table and access by column name. To get the data from the child table `inputs`, get an array of rows using the `GetChildRows()` method:

```
DataRow[] children = row.GetChildRows("testcase_inputs"); // relation name
```

Because each <testcase> node has only one <inputs> child node, the children array will only have one row. The trickiest aspect of this technique is to extract the child data:

```
tc.arg1 = (children[0]["arg1"]).ToString(); // there is only 1 row in children
```

Using the DataSet class to parse an XML file has a very relational database feel. Compared with the other parsing techniques in this chapter, it operates at a middle level of abstraction. The ReadXml() method hides a lot of details, but you must traverse through relational tables.

Using a DataSet object to parse XML files is particularly appropriate when your test harness program is using ADO.NET classes so that you maintain a consistent look and feel. Using a DataSet object has relatively high overhead and would not be a good choice if performance is an issue. Because each level of an XML file generates a table, if your XML file is deeply nested, then using DataSet would not be a good choice. If you need to perform extensive in-memory processing of the XML file being parsed and the XML is not deeply nested, using a DataSet approach is generally a good choice because you can easily manipulate the data stored in DataTable objects.

12.6 Validating XML with XSD Schema

Problem

You want to validate an XML file using an XSD schema definition.

Design

Read through the XML file you wish to validate using an XmlValidatingReader object. If the XML file is invalid, control is transferred to a delegate method where you can print an error message. If the XML file is valid, control does not transfer to the delegate.

Solution

This code will validate file testCases.xml (shown in the introduction to this chapter) using a schema file named testCases.xsd:

```
try
{
  Console.WriteLine("\nStarting XML validation");
  XmlSchemaCollection xsc = new XmlSchemaCollection();
  xsc.ValidationEventHandler += new ValidationEventHandler(ValidationCallBack);
  xsc.Add(null, "..\\..\\testCases.xsd");
  XmlTextReader xtr = new XmlTextReader("..\\..\\testCases.xml");
  XmlValidatingReader xvr = new XmlValidatingReader(xtr);
  xvr.ValidationType = ValidationType.Schema;
  xvr.Schemas.Add(xsc);
  xvr.ValidationEventHandler += new ValidationEventHandler(ValidationCallBack);
  while (xvr.Read()); // note empty loop
```

```
    Console.WriteLine("If no error message then XML is valid");
    Console.WriteLine("Done");
    Console.ReadLine();
  }
  catch(Exception ex)
  {
    Console.WriteLine("Generic exception: " + ex.Message);
    Console.ReadLine();
  }

  Console.WriteLine("\nDone");
  Console.ReadLine();
```

where:

```
private static void ValidationCallBack(object sender, ValidationEventArgs ea)
{
  Console.WriteLine("Validation error: " + ea.Message);
  Console.ReadLine();
}
```

and file testCases.xsd is:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="suite" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="suite" msdata:IsDataSet="true">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="testcase">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="inputs" minOccurs="1"
              maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="arg1" type="xs:string"
                    minOccurs="1" />
                    <xs:element name="arg2" type="xs:string"
                    minOccurs="1" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element name="expected"
                type="xs:string" minOccurs="0"
                msdata:Ordinal="1" />
            </xs:sequence>
            <xs:attribute name="id" type="xs:string" />
            <xs:attribute name="bvt" type="xs:string" />
```

```
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

You can generate an XSD schema definition by hand, or you can use the xsd.exe tool as described in Section 12.4 to generate one for you to use as a starting point.

Comments

When using XML in lightweight software test automation situations, you will often need or want to check that various XML files are valid. For example, if your test case data is stored as XML you will likely want to validate it before launching a test run. Or if you store test results as XML, you may want to validate the results file before distributing the results. Validating XML with XSD schema is relatively easy. You create an `XmlValidatingReader` object and set the `ValidationType` property to `ValidationType.Schema`. You add the validating schema definition file through an `XmlSchemaCollection`; this approach allows multiple schema definitions to be used against a single XML file. The only unusual aspect of the validation process is that when you read through the XML file being validated, instead of getting a return result indicating success or failure, a delegate method will be called if the XML is invalid, and nothing will happen if the XML is valid. So you have to create a callback method to handle the validation error. In this example, we would simply print the validation message.

Generating XSD schema definition files from scratch is not so much fun. A better approach is to use the xsd.exe tool to generate an initial XSD file to be used as a starting point, and then manually edit the generated file as needed. For example, when xsd.exe was applied to the testCases.xml file (presented in the introduction to this chapter), the resulting XSD file contained this:

```
<xs:element name="testcase">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="expected" type="xs:string" minOccurs="0"
    msdata:Ordinal="1" />
    <xs:element name="inputs" minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="arg1" type="xs:string" minOccurs="0" />
          <xs:element name="arg2" type="xs:string" minOccurs="0" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
   </xs:sequence>
    <xs:attribute name="id" type="xs:string" />
    <xs:attribute name="bvt" type="xs:string" />
 </xs:complexType>
</xs:element>
```

This is close to, but not exactly, what is needed for this solution. Notice that the expected result is mistakenly defined to come before the inputs, and that `arg1` and `arg2` were defined to allow 0 occurrences. You can easily make changes because the XML format of XSD files is very readable (for example, `minOccurs=0`) and is self-explanatory.

An alternative approach to validating XML files using XSD schema is to validate using DTD (Document Type Definition) files. DTD is an older technology that is somewhat easier to use than XSD, but not as powerful as XSD schema validation.

12.7 Modifying XML with XSLT

Problem

You want to generate a modified version of an XML file using XSLT (Extensible Stylesheet Language Transformations).

Design

Create an XSLT template file, then create an `XslTranform` object. Use the `Load()` and `Transform()` methods to generate the modified version of the original XML file.

Solution

Suppose you wish to modify the testCases.xml file from its original form:

```
<?xml version="1.0" encoding="utf-8" ?>
<suite>

  <testcase id="001" bvt="yes">
    <inputs>
      <arg1>red</arg1>
      <arg2>blue</arg2>
    </inputs>
    <expected>purple</expected>
  </testcase>

  (other <testcase> nodes here)

</suite>
```

to a modified version that looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<allOfTheCases>
  <aCase caseID="001">
    <bvt>yes</bvt>
    <expRes>purple</expRes>
```

```
    <inputs>
      <input1>red</input1>
      <input2>blue</input2>
    </inputs>
  </aCase>

  (other <aCase> nodes here)

</allOfTheCases>
```

The names of all nodes are different in the modified XML file; the `bvt` attribute in the original file is replaced by an element in the modified file; and the expected result comes before the inputs in the modified file. First, create an XSLT file like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" indent="yes"/>

<xsl:template match="/">
<allOfTheCases>
  <xsl:for-each select="//testcase">
    <aCase>
      <xsl:attribute name="caseID"><xsl:value-of select="@id"/></xsl:attribute>
      <bvt><xsl:value-of select="@bvt"/></bvt>
        <expRes><xsl:value-of select="expected"/></expRes>
        <inputs>
          <xsl:for-each select="inputs">
            <input1><xsl:value-of select="arg1"/></input1>
            <input2><xsl:value-of select="arg2"/></input2>
          </xsl:for-each>
        </inputs>
    </aCase>
  </xsl:for-each>
</allOfTheCases>

</xsl:template>
</xsl:transform>
```

and then programmatically apply the transform using C# code like this:

```
Console.WriteLine("\nStarting XSLT Transformation");
XslTransform xst = new XslTransform();
xst.Load("..\\..\\testCasesModifier.xslt");
xst.Transform("..\\..\\testCases.xml", "..\\..\\testCasesModified.xml");
Console.WriteLine("Done. New XML file is testCasesModified.xml");
```

Comments

You may want to generate an XML file that is a modified version of some other XML file. For example, in a testing situation you may want to use an existing test case data file created by some other group as input to one of your test harnesses, but you need to modify the XML to conform to the structure expected by your harness. One way to do this is to use XSLT technology. The problem boils down to creating the appropriate .xslt transform template. If you examine the example in this section, you'll see that XSLT is fairly intuitive. The `xsl:for-each` tag is used for iteration; the `xsl:value-of` tag is used for assignment; and XPath syntax is used for specifying particular attributes and elements. Once you have created the XSLT modification file, applying it with the `XslTranform` class is also very obvious.

The potential problem with XSLT is not so much technical as it is psychological; using XSLT has a very different feel than normal procedural-style programming. Because of this, many testers prefer a completely different approach to generating a modified version of an XML file, which does not use XSLT—they parse the original XML file into memory using one of the techniques presented in this chapter, modify the in-memory image of the original file to match the target structure, then write the modified image to file. This alternate technique is common. However, there may be situations in which you inherit a system that makes heavy use of XSLT.

12.8 Writing XML Using XmlTextWriter

Problem

You want to write to an XML file using the `XmlTextWriter` class.

Design

Use the `WriteStartElement()` method to write XML element tags. Use the `WriteAttributeString()` method to write attribute values. Use the `WriteString()` method to write element values.

Solution

For example, this code:

```
string caseID = "0001";
string result = "Pass";
string whenRun = "01/23/2006";

XmlTextWriter xtw = new XmlTextWriter("..\\..\\Results1.xml",
  System.Text.Encoding.UTF8);
xtw.Formatting = Formatting.Indented;
xtw.WriteStartDocument();
xtw.WriteStartElement("Results");
xtw.WriteStartElement("result");
xtw.WriteAttributeString("id", caseID);
xtw.WriteStartElement("passfail");
xtw.WriteString(result);
```

```
xtw.WriteEndElement();
xtw.WriteStartElement("whenRun");
xtw.WriteString(whenRun);
xtw.WriteEndElement();
xtw.WriteEndElement();
xtw.WriteEndElement();
xtw.Close();
```

will produce as output:

```
<?xml version="1.0" encoding="utf-8"?>
<Results>
  <result id="0001">
    <passfail>Pass</passfail>
    <whenRun>01/23/2006</whenRun>
  </result>
</Results>
```

Comments

Writing XML results using an `XmlTextWriter` object is simple and straightforward. In theory, all you need is the `XmlTextWriter.WriteString()` method, which simply writes its argument to output. But if you only use `WriteString()` you will not get the benefit of the `XmlTextWriter` class and you could just as well have used a series of `StreamWriter.WriteLine()` statements to write the XML file. The preceding solution uses explicit `WriteStartElement()` and `WriteEndElement()` calls like this:

```
xtw.WriteStartElement("whenRun");
xtw.WriteString(whenRun);
xtw.WriteEndElement();
```

Alternatively, you can use `WriteElementString()` like this:

```
xtw.WriteElementString("whenRun" , whenRun);
```

The `XmlTextWriter` class has many useful methods such as `WriteComment()` and `WriteCData()`.

12.9 Comparing Two XML Files for Exact Equality

Problem

You want to compare two XML files for exact equality.

Design

Write a helper method that iterates through each file using two `FileStream` objects. Read each file byte-by-byte, and return false if you hit a byte mismatch.

Solution

This method will compare two XML files for exact equality:

```
private static bool XMLExactlySame(string file1, string file2)
{
  FileStream fs1 = new FileStream(file1, FileMode.Open);
  FileStream fs2 = new FileStream(file2, FileMode.Open);

  if (fs1.Length != fs2.Length) // number bytes
    return false;
  else
  {
    int b1 = 0;
    int b2 = 0;

    while ((b1 = fs1.ReadByte()) != -1)
    {
      b2 = fs2.ReadByte();
      //Console.WriteLine("b1 = " + b1 + " b2 = " + b2);
      if (b1 != b2)
      {
        fs1.Close();
        fs2.Close();
        return false;
      }
    }
    fs1.Close();
    fs2.Close();
    return true;
  }
} // XMLExactlySame()
```

This code assumes the two files passed in as input arguments exist. First we check the size of the two files; if the sizes are different, the two files cannot possibly be identical. Next, we iterate through both files, and read one byte from each, and compare the two byte values. If the byte values differ, we know the files are different, so we can close the `FileStream` objects and return false. If we make it all the way through both files, they must be identical.

Comments

In software test automation, if the system under test produces an XML file as output, you will have to compare an actual XML file with an expected XML file. One of several ways to do this is to store an expected XML file and then compare byte-by-byte. Because XML files are just a particular type of text file, the technique in this section will work for any text file.

12.10 Comparing Two XML Files for Exact Equality, Except for Encoding

Problem

You want to compare two XML files for exact equality except for their encoding.

Design

Read each of the two files being compared into a string variable. Then compare the two strings using the ordinary == Boolean comparison operator.

Solution

```
private static bool XMLExactlySameExceptEncoding(string file1, string file2)
{
  FileStream fs1 = new FileStream(file1, FileMode.Open);
  FileStream fs2 = new FileStream(file2, FileMode.Open);
  StreamReader sr1 = new StreamReader(fs1);
  StreamReader sr2 = new StreamReader(fs2);

  string s1 = sr1.ReadToEnd();
  string s2 = sr2.ReadToEnd();
  //Console.WriteLine(s1);
  //Console.WriteLine(s2);
  sr1.Close();
  sr2.Close();
  fs1.Close();
  fs2.Close();

  return (s1 == s2);
}
```

Comments

In testing situations, you may want to compare an actual XML file with an expected XML file but you do not care if the encoding schemes are different. In other words, if the actual and expected XML files both have the same character data but one file is encoded using UTF-8 and the other is encoded using ANSI, the files are equivalent from your perspective. One way to perform such a comparison is to simply read both files into string variables and compare using the overloaded == operator. The Boolean == operator is overloaded to take into account character encoding. This approach may not be feasible if the two XML files being compared are very, very large. In this situation, you can adapt Section 12.9 by reading through each file a character at a time and doing a character-by-character comparison.

12.11 Comparing Two XML Files for Canonical Equivalence

Problem

You want to compare two XML files for canonical equivalence. You can think of canonical equivalence as meaning "the same for most practical purposes."

Design

Perform a C14N canonicalization on the two XML files being compared using the `XmlDsigC14NTransform` class and then compare the two files in memory using two `MemoryStream` objects.

Solution

```
// using System.Security.Cryptography.Xml;

string f1 = "..\\..\\Books1.xml";
XmlDocument xd1 = new XmlDocument();
xd1.Load(f1);

XmlDsigC14NTransform t1 = new XmlDsigC14NTransform(true);
// true = include comments

t1.LoadInput(xd1);
Stream s1 = t1.GetOutput() as Stream;
XmlTextReader xtr1 = new XmlTextReader(s1);
MemoryStream ms1 = new MemoryStream();
XmlTextWriter xtw1 = new XmlTextWriter(ms1, System.Text.Encoding.UTF8);
xtw1.WriteNode(xtr1, false);
// false = do not copy default attributes

xtw1.Flush();
ms1.Position = 0;
StreamReader sr1 = new StreamReader(ms1);
string str1 = sr1.ReadToEnd();
//Console.WriteLine(str1);

//Console.WriteLine("\n======\n");

string f2 = "..\\..\\Books2.xml";
XmlDocument xd2 = new XmlDocument();
xd2.Load(f2);
XmlDsigC14NTransform t2 = new XmlDsigC14NTransform(true);
t2.LoadInput(xd2);
```

```
Stream s2 = t2.GetOutput() as Stream;
XmlTextReader xtr2 = new XmlTextReader(s2);
MemoryStream ms2 = new MemoryStream();
XmlTextWriter xtw2 = new XmlTextWriter(ms2, System.Text.Encoding.UTF8);
xtw2.WriteNode(xtr2, false);
xtw2.Flush();
ms2.Position = 0;
StreamReader sr2 = new StreamReader(ms2);
string str2 = sr2.ReadToEnd();
Console.WriteLine(str2);

if (str1 == str2)
  Console.WriteLine("Files canonically equivalent");
else
  Console.WriteLine("Files NOT canonically equivalent ");
```

Comments

Suppose an XML file Books1.xml looks like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<books>

  <book>
    <title isbn='1111'   storeid="A1A1">
All About Apples</title>
    <author>
      <last>Anderson</last>
      <first>Adam</first>
    </author>
  </book>
</books>
```

and suppose that a second XML file, Books2.xml, looks like this:

```
<books>
  <book>
    <title storeid="A1A1" isbn="1111">
      All About Apples
    </title>
    <author>
      <last>Anderson</last>
      <first>Adam</first>
    </author>
  </book>

</books>
```

If the code in this solution is run against these two files, the message "Files canonically equivalent" would be displayed—these two files are canonically equivalent. The whitespace differences do not matter; the use of single-quote and double-quote characters does not matter; XML declarations do not matter; and the order of attributes does not matter. C14N canonical equivalence is fairly complex. It is defined by the W3C and is primarily used in security contexts. In order to determine if an XML file has been accidentally or maliciously changed during transmission over a network, you can compare crypto-hashes of the transmitted file and the received file. However, because networks may modify the files, we need a way to determine canonical equivalence. This explains why the `XmlDsigC14NTransform` class is in the System.Security.dll assembly.

12.12 Example Program: XmlTest

The program in Listing 12-1 demonstrates XML validation using an XSD schema definition, and XML parsing using the `XmlSerializer` class. When run, the output will be that shown in Figure 12-1 in the introduction to this chapter.

Listing 12-1. *Program XmlTest*

```
using System;
using System.IO;
using System.Xml;
using System.Xml.Schema; // validation
using System.Xml.Serialization; // deserialization

namespace XmlTest
{
  class Class1
  {
    [STAThread]
    static void Main(string[] args)
    {
      try
      {
        Console.WriteLine("\nBegin XML techniques demonstration\n");

        Console.WriteLine("Original file is: \n");
        FileStream fs = new FileStream("..\\..\\TestCases.xml",
                                       FileMode.Open);
        StreamReader sr = new StreamReader(fs);
        string line;
        while((line = sr.ReadLine()) != null)
        {
          Console.WriteLine(line);
        }
        sr.Close(); fs.Close();
```

```
      Console.WriteLine("\nValidating original file using
                        rules in TestCases.xsd");
      XmlSchemaCollection xsc = new XmlSchemaCollection();
      xsc.ValidationEventHandler +=
       new ValidationEventHandler(ValidationCallBack);
      xsc.Add(null, "..\\..\\testCases.xsd");
      XmlTextReader xtr = new XmlTextReader("..\\..\\testCases.xml");
      XmlValidatingReader xvr = new XmlValidatingReader(xtr);
      xvr.ValidationType = ValidationType.Schema;
      xvr.Schemas.Add(xsc);
      xvr.ValidationEventHandler +=
       new ValidationEventHandler(ValidationCallBack);
      while (xvr.Read()); // note empty loop

      Console.WriteLine("XML test case file is valid");

      Console.WriteLine("\nParsing original file into memory
                        using Deserialization()");
      XmlSerializer xs =
        new XmlSerializer(typeof(SerializerLib.Suite));
      sr = new StreamReader("..\\..\\TestCases.xml");
      SerializerLib.Suite suite =
       (SerializerLib.Suite)xs.Deserialize(sr);
      sr.Close();
      Console.WriteLine("Parsed data = \n");
      suite.Display();

      Console.WriteLine("\nDone");
      Console.ReadLine();
    }
    catch(Exception ex)
    {
      Console.WriteLine("Fatal error: " + ex.Message);
      Console.ReadLine();
    }

      } // Main()

  private static void ValidationCallBack(object sender, ValidationEventArgs ea)
  {
    Console.WriteLine("Validation error: " + ea.Message);
    Console.ReadLine();
  }
  } // class

namespace SerializerLib
{
```

```
    [XmlRootAttribute("suite")]
    public class Suite
    {
      [XmlElementAttribute("testcase")]
      public TestCase[] items; // changed name from xsd-generated code
      public void Display() // added to xsd-generated code
      {
        foreach (TestCase tc in items)
        {
          Console.Write(tc.id + " " + tc.bvt + " "  + tc.inputs.arg1 + " ");
          Console.WriteLine(tc.inputs.arg2 + " " + tc.expected);
        }
      }
    }

    public class TestCase  // changed name from xsd-generated code
    {
      [XmlAttributeAttribute()]
      public string id;
      [XmlAttributeAttribute()]
      public string bvt;
      [XmlElementAttribute("inputs")]
      public Inputs inputs; // change from xsd-generated code: no array
      public string expected;
    }

    public class Inputs // changed name from xsd-generated code
    {
      public string arg1;
      public string arg2;
    }
  } // ns SerializerLib

} // ns
```