Test Run
Five Ways to Emit Test Results as XML
James McCaffrey

Code download available at: TestRun2006_06.exe (164KB)

Contents

The Problem
Technique 1: Using the XmlTextWriter Class
Technique 2: Using the XmlDocument Class
Technique 3: Using the XPathDocument Class
Technique 4: Using the DataSet Class
Technique 5: Using the XmlSerializer Class
Summing It Up

The use of XML files in software testing has steadily increased over the past few years. Test case data, test harness configuration information, and test result data are now stored as XML. Recently I was writing some .NET test automation that saved test results to XML storage, and in conversations with colleagues I discovered that there are a variety of opinions on the best way to emit XML files. I set out to enumerate the various ways that exist to emit XML in a .NET testing environment and to understand the pros and cons of each technique. The "best" technique depends primarily on how your test result data is stored in memory, the type of processing you are performing on the data, and to some extent the style of programming you prefer.

In the sections that follow, I will demonstrate how to emit test result data to an XML file using five different techniques: XmlTextWriter, XmlDocument, XPathDocument, DataSet, and XmlSerializer. Each is based on a different Microsoft® .NET Framework class and its associated methods. These five techniques are by no means the only ways to emit test results to XML, but they are fundamental and will be a valuable addition to your .NET skill set.

The Problem

Let's begin by examining the basis for each of the five. Suppose you are developing a dice game application named Boatzee, as shown in Figure 1. Behind the scenes of the application there is a library named BoatzeeLib that contains classes such as a Die class to represent a single die, and a Hand class to represent a set of five dice.

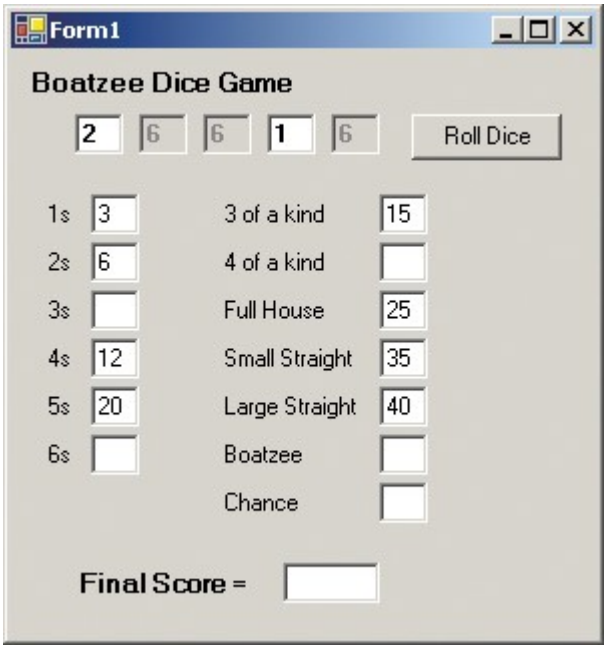

Figure 1 Application Under Test

One of the methods in the BoatzeeLib library you need to test is Hand.GetHandType, which returns a member of an enumeration. For example, if a Hand object, h, has the five dice shown in Figure 1 (2, 6, 6, 1, 6), then h.GetHandType returns an enumeration with string representation "ThreeOfAKind".

Now suppose you have test case data stored in a simple text file named TestCases.txt:

```
0001:3:3:3:3:3:Boatzee
0002:5:5:2:2:2:FullHouse
0003:6:5:4:3:1:SmallStraight
```

Each line of the test case data represents a single test case. The first field is a test case ID, the second through sixth fields are inputs to the Hand constructor, and the last field is an expected result when Hand.GetHandType is called. Of course, in a real-life situation, your test case data would probably be stored in an XML file, and you would have many hundreds of test cases. I'm using a text file for test case storage just to avoid confusion with the XML test result files. I will demonstrate several ways to emit test results to an XML file using this test case data. Each of the techniques will produce the XML test results file shown in Figure 2.

Technique 1: Using the XmlTextWriter Class

The most traditional technique is to use the XmlTextWriter class. The basic plan is to write XML elements (and attributes) to the output stream as the data for the XML elements becomes available to the test harness. The example code is shown in Figure 3. When run, the output will be as shown in Figure 2.

I start coding by creating an XmlTextWriter object, then use the WriteStartDocument method to write an XML declaration, and the WriteStartElement method to write the <TestResults> root element to the output stream. Next I iterate through each line in the test case data file and parse out the test case ID, the five inputs to the Die constructor, and the expected result.

Inside the main test loop, I instantiate a Hand object using the test case inputs. Then I use WriteStartElement to create a <result> tag, and the WriteAttributeString method to insert the test case ID into the <result> tag as an attribute. I use the WriteElementString method to add the entire <input> and <expected> elements. Then I call the Hand.GetHandType method under test, fetch the actual return value, and compare it to the expected value to determine a pass/fail test result.

Once I know the pass/fail result, I can create the entire <passfail> element and add the closing </result> tag to the output stream. After all the test cases have been executed, I add the final </TestResults> tag using the WriteEndElement method. Finally, I close the XmlTextWriter object, which causes any internally buffered output to be written to the file.

Emitting test results to XML with XmlTextWriter has a traditional, pre-.NET feel. As each piece of test result data becomes available, you add it to the output stream. Using XmlTextWriter is simple and straightforward, and is appropriate when test result data is streamed to your test harness. Because the various Write methods of the XmlTextWriter class are forward-only mechanisms, using XmlTextWriter to emit test results is not usually appropriate in situations where the data to write is not streamed. For example, in a testing situation where a particular test result cannot be fully determined until after some subsequent test case executes, you are probably better off using one of the other four techniques presented in this column. Compared to the other techniques covered here, XmlTextWriter 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 output XML file, and write elements and attributes correctly. Instead of using the methods in the XmlTextWriter class, you could simply write all output using a WriteLine method, along the lines of the following, but you would lose a lot of error-checking, power, flexibility, and clarity:

```
Console.WriteLine("<?xml version ='1.0' ?>");
Console.WriteLine("<TestResults>");
...
```

Technique 2: Using the XmlDocument Class

The second way to emit test result data to an XML file is to use the XmlDocument class. The plan is to create a complete in-memory representation of the entire XML results file, and then write it to a file (see Figure 4). When run, the output will be exactly the same as the output from the previous XmlTextWriter technique, as shown in Figure 2.

I start by instantiating an XmlDocument object. Next I obtain a reference to the virtual root element which is returned from the XmlDocument.DocumentElement property. The real purpose of this root element is to provide an anchor with which to insert an XmlDeclaration object with the InsertBefore method (because an XML declaration must come before the root element). Next I create the actual <TestResults> root element using the CreateElement method. After these preliminaries, I process the test cases exactly as I do in the previous section: read a test case, parse the fields, call the method under test, and determine a pass/fail result.

My next step is to create the <input>, <expected>, and <passfail> elements using the CreateElement method, and supply values using the XmlElement.InnerText property. I use the SetAttribute method to insert the test case ID attribute into each <result> element. Each element is added to the in-memory XmlDocument object by using the AppendChild method or the InsertAfter method, as appropriate.

The XmlDocument class is modeled on the W3C XML Document Object Model (XML DOM) and may have a different feel to it than non-XML .NET Framework classes that you are familiar with. Using the

XmlDocument class to emit test results to XML is appropriate if you need to construct result data in a nonsequential manner, as you can use InsertBefore, InsertAfter, AppendChild and other methods to place data anywhere in the in-memory XML document representation.

Using XmlDocument to emit test results is also appropriate if you are already using XmlDocument objects (for test case data, for example) and want to maintain a consistent feel in your test code. Because using XmlDocument builds the entire XML file in memory, using it here may not be a good choice if your XML result set is very large.

Let me note that in discussions with my colleagues, there was some minor confusion about the role of the XmlDataDocument class. It is derived from the XmlDocument class and is intended for use in conjunction with DataSet objects. So, in this example, I could have used the XmlDataDocument class but would not have gained any significant advantage by doing so.

Technique 3: Using the XPathDocument Class

The third technique uses the XPathDocument class. The basic plan is to start with an XPathDocument, which is an in-memory representation of an XML document, and use an XslTransform object in conjunction with an XSL stylesheet template to emit results. The following example emits XML using XPathDocument:

```
XPathDocument xpd = new XPathDocument("..\\..\\TestResultsOrig.xml");
XslTransform xslt = new XslTransform();
xslt.Load("..\\..\\TransformSheet.xsl");
XmlTextWriter xtw = new XmlTextWriter("..\\..\\TestResults3.xml",
  Encoding.UTF8);
xtw.Formatting = Formatting.Indented;
xtw.WriteStartDocument();
xslt.Transform(xpd, null, xtw, null);
  xtw.Close();
```

The associated XSL stylesheet is shown in Figure 5. When both of these are run, the resulting XML file will be the same as the one you saw in Figure 2. Before I explain the mechanics of the XPathDocument technique, let me first describe a few scenarios you may encounter.

An XPathDocument object is read-only; this means that you will never directly create an XPathDocument of test results from a test harness. The only practical way to get an XPathDocument into memory is to load it from an existing XML file or stream when you call the XPathDocument constructor. The purpose of an XPathDocument is to provide a highly efficient means of searching an XML file using XPath syntax. So the situations in which you emit test result data to XML using XPathDocument are very limited. The most common testing scenario I have encountered goes like this. You create an XML test results file using one of the other four techniques presented in this column (XmlTextWriter, XmlDocument, DataSet, or XmlSerializer). Later, you find that you must perform some analysis of the original XML test results file where search-efficiency is important, and you also find that the format of the original XML file is not precisely what you need. So, you read the original XML file into memory as an XPathDocument object, perform your read-only analysis, and finally emit the XPathDocument to a new XML file with the desired format.

Now let me explain the mechanics of using an XPathDocument object. First I create an XPathDocument object by passing an existing XML file to the XPathDocument constructor. Next I create an XslTransform object and pass an existing XSL stylesheet to the constructor. Creating the XSL stylesheet

is the key to the whole technique. Once again, I want the new XML file to look exactly like the one shown in Figure 2. As I just described, you only need to use this technique if your original XML results file is not in the correct format. Here I assume that file TestResultsOrig.xml looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<TheTestResults>
  <aresult id="0001">
    <theinput>[3] [3] [3] [3] [3]</theinput>
    <expected>Boatzee</expected>
    <whenrun>05/12/2006</whenrun>
    <passfail>Pass</passfail>
  </aresult>
   ...
</TheTestResults>
```

In other words, the root node of the original XML file is <TheTestResults> instead of the desired <TestResults>, the original file has <aresult> instead of <result>, <theinput> instead of <input>, and a <whenrun> element, which is not wanted. You must construct an XSL stylesheet to act as a mapping template from the original XML file to the desired XML file. I start building up the stylesheet by looking at the innermost elements in the original XML file that actually hold data I want to keep. In this example, those are the <theinput>, <expected>, and <passfail> elements. For each of these elements I create a matching template which identifies the original element, specifies the new element name, and assigns a value to the new element name like this:

```
<xsl:template match="theinput">
  <input><xsl:value-of select="." /></input>
</xsl:template>
```

Notice that I've placed these <xsl:template> tags at the bottom of my XSL stylesheet, so I am building my stylesheet from the bottom up. After doing this for each innermost data element, I move up one level of nesting. Here, that is the <aresult> element:

```
<xsl:template match="aresult">
  <result>
    <xsl:attribute name="id"><xsl:value-of
        select="@id"/></xsl:attribute>
      <xsl:apply-templates select="theinput" />
      <xsl:apply-templates select="expected" />
      <xsl:apply-templates select="passfail" />
  </result>
</xsl:template>
```

I specify the match criteria in the original XML file, and specify what to place inside the target XML file—in this case, the results of the innermost templates. The @ character in @id instructs the parser to retrieve the value of an attribute named id. I finish by moving up to the highest level of nesting, which is the root node in this example:

```
<xsl:template match="TheTestResults">
  <TestResults>
    <xsl:apply-templates select="aresult" />
  </TestResults>
</xsl:template>
```

If you haven't created XSL stylesheets before, you may find the process a bit bizarre at first. However, after you make one or two stylesheets, you'll get used to it. After constructing the XSL stylesheet,

emitting XML results from an XPathDocument is easy. I create an XmlTextWriter object, and call the Transform method of the XslTransform class.

There is a completely different approach to emitting test results from an XPathDocument to an XML file. As I mentioned, XPathDocument objects exist to promote efficient searching. Searching is most often performed using an XPathNodeIterator object. You can traverse through each node of an XPathDocument object using an XPathNodeIterator object and the MoveNext method, get a reference to each node as you traverse, and then write each node, modifying if necessary, using an XmlTextWriter object. This technique requires much more coding than using an XslTransform object, but if you want to avoid XSL stylesheets, this offers a good alternative approach.

Technique 4: Using the DataSet Class

The basic plan this time is to create an in-memory DataSet object, store all test result data into the DataSet, then emit the entire DataSet to an XML file. The example code, shown in Figure 6, will yield the same XML file as that produced by all the other techniques presented here.

The key to this technique is to set up a DataSet object with a format that maps to your desired XML file format. First I create a DataSet object called TestResults, which will become the XML root element. Next I create a DataTable object with the name result. Each table name will become a parent element name when the DataSet is emitted to XML. Next I add columns named id, input, expected, and passfail. Each column name will become a data-element name or data-attribute name. By default, column names become element names. In order to force a column name into becoming an attribute, as I want to do with the test case ID value, I use the DataColumn.ColumnMapping method and set its MappingType property to MappingType.Attribute. After setting up the DataTable object, I add it into my DataSet object with the DataTables.Add method.

Once the DataSet is ready, I process my test cases just as with the XmlTextWriter and XmlDocument techniques: read test case data, parse, call method under test, and determine pass/fail result. I store each test case id, input, expected, and passfail value into an object array, and then pass that array to the DataRow.Add method to insert the result data into the DataSet object. After all test cases have been processed, emitting the test result data to an XML file is simply a matter of calling the DataSet.WriteXml method.

Using this DataSet technique is most appropriate when your test harness naturally uses a DataSet to store test results. For example, storing test results into a DataSet is particularly useful if you need to perform processing of you results—if your results are in a DataSet, you can easily filter and process aggregate results such as the number of test cases of a particular type that failed.

Using the DataSet technique to emit test results to XML is often not a good choice if you have a predetermined XML format you must conform to, and that format is deeply nested. Each nested level of an XML file maps to a DataTable object. If, for example, your target XML format looked something like this:

```
<TestResults>
  <result>
    <alpha>
      <beta>
        <data>5</data>
      </beta>
    </alpha>
```

```
  </result>
  etc.
</TestResults>
```

then you would need a column named data in a DataTable named beta, and you'd need a second DataTable named alpha, and also a third DataTable named result. These three DataTable objects would have to be connected through DataRelation objects.

Technique 5: Using the XmlSerializer Class

The last fundamental way to emit test result data as an XML file I'll discuss is to use the XmlSerializer class. The basic plan is to store test results in memory in an array or ArrayList object, and then serialize the entire object to an XML file. The example code is shown in Figure 7. When this is run, the XML file will be the same as that produced by all the other techniques in this column.

Before I write the serialization code, I have to set up two utility classes to hold my test result data. The first class, named result, holds an individual test result:

```
public class result
{
    [XmlAttribute]
    public string id;

    [XmlElement("input")]
    public string input;

    public string expected;
    public string passfail;
}
```

Notice that I decorate the id field with an [XmlAttribute] attribute. This instructs the common language runtime (CLR) to interpret the id field as an attribute when an XmlSerializer object serializes a result object. I don't pass in a name argument to XmlAttribute so the id field will be serialized with name id. I decorate the input field with an [XmlElement] attribute. This is the default interpretation so I could have left [XmlElement] out altogether; I put it in so you can see how to rename element fields if necessary.

My second utility class, which is called TestResults, is a collection of result objects:

```
public class TestResults // becomes root node
{
    [XmlElement(ElementName = "result", Type = typeof(result))]
    public ArrayList Results = new ArrayList();
}
```

The class is really just a wrapper around an ArrayList object. The class name, TestResults, becomes the root element in the resulting XML file after serialization. I decorate the ArrayList field with an [XmlElement] attribute to instruct the CLR that each item in the ArrayList contains a result type, and that each item should produce an XML element named result.

When you are using the XmlSerializer technique, any field you want to decorate must be public. In a testing scenario this is usually not too much of a problem, but if you need to serialize private fields, you can create public get and set properties, and then decorate those properties.

Using XmlSerializer to emit test result data to an XML file is most appropriate when your test harness naturally requires a class that encapsulates test results. XML serialization is an elegant mechanism but it is sometimes tricky to set up the storage classes so that you get a particular XML format after serialization. As a general rule, you are better off letting the requirements of your particular testing situation determine the design of your storage classes. You can use serialization on those classes and then, if your resulting XML file format is not quite what you want, use an XSL transform to get to the desired format. For example, the previous TestResults class definition is somewhat artificial. A more natural design is:

```
public class TestResults // becomes root node
{
  [XmlArray("TestResults")]
  [XmlArrayItem(Type=typeof(result))]
  public ArrayList Results = new ArrayList();
}
```

The attribute decorations to the ArrayList object tell the CLR to interpret the ArrayList as an array-derived type where each item is a result object. With this class, the previous code will produce an XML file like this:

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

<TestResults xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <TestResults>
        <result id="0001">
            <input>[3] [3] [3] [3] [3]</input>
            <expected>Boatzee</expected>
            <passfail>Pass</passfail>
        </result>
        ...
    </TestResults>
</TestResults>
```

There is nothing at all wrong with this format; it just doesn't exactly match the format of the other four techniques presented in this column.

XML serialization tends to have slower performance than the other four techniques presented here, so serialization may not be appropriate if performance is a major concern.

Summing It Up

XML data files are a key component of the Microsoft .NET development environment. When testing in a .NET environment, you will often want to save test result data to an XML file. I have described five different ways to do this. Using XmlTextWriter is most appropriate when your test result data becomes available as a stream. Using XmlTextWriter is least appropriate when you need to perform some aggregate analysis of your test results. Using XmlDocument is useful when your test result data must be constructed nonsequentially. The XmlDocument technique, along with all techniques other than XmlTextWriter, is not a good choice if your result data is very large. The XPathDocument technique will be your least commonly used approach because it is useful only when you start with an in-memory XPathDocument object and you need to emit XML results in a different format. Using a DataSet object to emit XML is most appropriate when your test harness naturally stores result data into a DataSet, such

as when you need to aggregate analysis of your results. Using a DataSet object is not appropriate when you have a required XML format that is deeply nested. The XmlSerializer technique is useful when your test harness has one or more classes that store test result objects, but is less useful when performance is a primary concern.

You can also combine these techniques in many ways. For example, if your test result data is stored in a DataSet object, you can iterate through the object's DataTable and write to a file using an XmlTextWriter object. Or if your test results are stored in an ArrayList object, you can iterate through the list and build up an XmlDocument object in memory, and then emit the document to file.

The code in this column was designed to work with the .NET Framework versions 1.1 and 2.0. If you are working in a strictly .NET Framework 2.0 environment, you can take advantage of some neat new features. For example, the DataTable class now has many methods that previously were available only to the DataSet class, including a WriteXml method. Additionally, a DataTable object can now be serialized by itself, so you aren't forced to create a DataSet with a single table to serialize test result data. There is also support for reading and writing of XML where a single table or XML element is nested inside two or more parent tables.

.

James McCaffrey works for Volt Information Sciences Inc., where he manages technical training for software engineers working at Microsoft. He has worked on several Microsoft products including Internet Explorer and MSN Search. James can be reached at jmccaffrey@volt.com or v-jammc@microsoft.com.
