CHAPTER 4

Test Harness Design Patterns

4.0 Introduction

One of the advantages of writing lightweight test automation instead of using a third-party testing framework is that you have great flexibility in how you can structure your test harnesses. A practical way to classify test harness design patterns is to consider the type of test case data storage and the type of test-run processing. The three fundamental types of test case data storage are flat, hierarchical, and relational. For example, a plain-text file is usually flat storage; an XML file is typically hierarchical; and SQL data is often relational. The two fundamental types of test-run processing are streaming and buffered. Streaming processing involves processing one test case at a time; buffered processing processes a collection of test cases at a time. This categorization leads to six fundamental test harness design patterns:

- Flat test case data, streaming processing model
- Flat test case data, buffered processing model
- Hierarchical test case data, streaming processing model
- Hierarchical test case data, buffered processing model
- Relational test case data, streaming processing model
- Relational test case data, buffered processing model

Of course, there are many other ways to categorize, but thinking about test harness design in this way has proven to be effective in practice. Now, suppose you are developing a poker game application as shown in Figure 4-1.

Figure 4-1. *Poker Game AUT*

Let's assume that the poker application references a `PokerLib.dll` library that houses classes to create and manipulate various poker objects. In particular, a `Hand()` constructor accepts a string argument such as "`Ah Kh Qh Jh Th`" (ace of hearts through ten of hearts), and a `Hand.GetHandType()` method returns an enumerated type with a string representation such as "`RoyalFlush`". As described in Chapter 1, you need to thoroughly test the methods in the `PokerLib.dll` library. This chapter demonstrates how to test the poker library using each of the six fundamental test harness design patterns and explains the advantages and disadvantages of each pattern. For example, Section 4.3 uses this hierarchical test case data:

```
<?xml version="1.0" ?>
<testcases>
  <case id="0001">
    <input>Ac Ad Ah As Tc</input>
    <expected>FourOfAKindAces</expected>
  </case>
  <case id="0002">
    <input>4s 5s 6s 7s 3s</input>
    <expected>StraightSevenHigh</expected>
  </case>
  <case id="0003">
    <input>5d 5c Qh 5s Qd</input>
    <expected>FullHouseFivesOverQueens</expected>
  </case>
</testcases>
```

and uses a streaming processing model to produce this result:

```
<?xml version="1.0" encoding="utf-8"?>
<TestResults>
  <case id="0001">
    <result>Pass</result>
  </case>
  <case id="0002">
    <input>4s 5s 6s 7s 3s</input>
    <expected>StraightSevenHigh</expected>
    <actual>StraightFlushSevenHigh</actual>
    <result>*FAIL*</result>
  </case>
  <case id="0003">
    <result>Pass</result>
  </case>
</TestResults>
```

Although the techniques in this chapter demonstrate the six fundamental design patterns by testing a .NET class library, the patterns are general and apply to testing any type of software component.

The streaming processing model, expressed in pseudo-code, is

```
loop
  read a single test case from external store
  parse test case data into input(s) and expected(s)
  call component under test
  determine test case result
  save test case result to external store
end loop
```

The buffered processing model, expressed in pseudo-code, is

```
loop // 1. read all test cases
  read a single test case from external store into memory
end loop

loop // 2. run all test cases
  read a single test case from in-memory store
  parse test case data into input(s) and expected(s)
  call component under test
  determine test case result
  store test case result to in-memory store
end loop

loop // 3. save all results
  save test case result from in-memory store to external store
end loop
```

The streaming processing model is simpler than the buffered model, so it is often your best choice. However, in two common scenarios, you should consider using the buffered processing model. First, if the aspect in the system under test (SUT) involves file input/output, you often want to minimize test harness file operations. This is especially true if you are monitoring performance. Second, if you need to perform any preprocessing of your test case input (for example, pulling in and filtering test case data from more than one data store) or postprocessing of your test case results (for example, aggregating various test case category results), it's almost always more convenient to have data in memory where you can process it.

4.1 Creating a Text File Data, Streaming Model Test Harness

Problem

You want to create a test harness that uses text file test case data and a streaming processing model.

Design

In one continuous processing loop, use a `StreamReader` object to read a test case data into memory, then parse the test case data into input and expected values using the `String.Split()` method, and call the component under test (CUT). Next, check the actual result with the expected result to determine a test case pass or fail. Then, write the results to external storage with a `StreamWriter` object. Do this for each test case.

Solution

Begin by creating a tagged and end-of-file delimited test case file:

```
[id]=0001
[input]=Ac Ad Ah As Tc
[expected]=FourOfAKindAces

[id]=0002
[input]=4s 5s 6s 7s 3s
[expected]=StraightSevenHigh

[id]=0003
[input]=5d 5c Qh 5s Qd
[expected]=FullHouseFivesOverQueens

*
```

Then process using `StreamReader` and `StreamWriter` objects:

```
Console.WriteLine("\nBegin Text File Streaming model test run\n");

FileStream ifs = new FileStream("..\\..\\..\\TestCases.txt",
                                FileMode.Open);
StreamReader sr = new StreamReader(ifs);
FileStream ofs = new FileStream("TextFileStreamingResults.txt",
                                FileMode.Create);
StreamWriter sw = new StreamWriter(ofs);

string id, input, expected, blank, actual;

while (sr.Peek() != '*')
{
  id = sr.ReadLine().Split('=')[1];
  input = sr.ReadLine().Split('=')[1];
  expected = sr.ReadLine().Split('=')[1];
  blank = sr.ReadLine();

  string[] cards = input.Split(' ');
  Hand h = new Hand(cards[0], cards[1], cards[2], cards[3], cards[4]);
  actual = h.GetHandType().ToString();

  sw.WriteLine("====================");
  sw.WriteLine("ID       = " + id);
  sw.WriteLine("Input    = " + input);
  sw.WriteLine("Expected = " + expected);
  sw.WriteLine("Actual   = " + actual);
```

```
  if (actual == expected)
    sw.WriteLine("Pass");
  else
    sw.WriteLine("*FAIL*");
}
sw.WriteLine("====================");

sr.Close(); ifs.Close();
sw.Close(); ofs.Close();

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

Comments

You begin by creating a test case data file. As shown in the techniques in Chapter 1, you could structure the file with each test case on one line:

```
0001:Ac Ad Ah As Tc:FourOfAKindAces
0002:4s 5s 6s 7s 3s:StraightSevenHigh:deliberate error
0003:5d 5c Qh 5s Qd:FullHouseFivesOverQueens
```

When using this approach, notice that the meaning of each part of the test case data is implied (the first item is the case ID, the second is the input, and the third is the expected result). A more flexible solution is to provide some structure to your test case data by adding tags such as `"[id]"` and `"[input]"`. This allows you to easily perform rudimentary validity checks. For example:

```
string temp = sr.ReadLine(); // should be the ID
if (temp.StartsWith("[id]"))
  id = temp.Split('=')[1];
else
  throw new Exception("Invalid test case line");
```

You can perform validity checks on your test case data via a separate program that you run before you run the test harness, or you can perform validity checks inside the test harness itself. In addition to validity checks, structure tags also allow you to deal with test case data that has a variable number of inputs.

This technique assumes that you have added a project reference to the `PokerLib.dll` library under test and that you have supplied appropriate `using` statements so you don't have to fully qualify classes and objects:

```
using System;
using PokerLib;
using System.IO;
```

You should also always wrap your test harness code in `try-catch-finally` blocks:

```
static void Main(string[] args)
{
  // Open any files here
  try
  {
    // main harness code here
  }
  catch(Exception ex)
  {
    Console.WriteLine("Fatal error: " + ex.Message);
  }
  finally
  {
    // Close any open streams here
  }

} // Main()
```

When the code in this section is run with the preceding test case input data, the output is

```
====================
ID       = 0001
Input    = Ac Ad Ah As Tc
Expected = FourOfAKindAces
Actual   = FourOfAKindAces
Pass
====================
ID       = 0002
Input    = 4s 5s 6s 7s 3s
Expected = StraightSevenHigh
Actual   = StraightFlushSevenHigh
*FAIL*
====================
ID       = 0003
Input    = 5d 5c Qh 5s Qd
Expected = FullHouseFivesOverQueens
Actual   = FullHouseFivesOverQueens
Pass
====================
```

Test case #0002 is an intentional failure. Using a special character token in the test case data file to signal end-of-file is an old but effective technique. With such a token in place, you can use the `StreamReader.Peek()` method to check the next input character without actually consuming it from the associated stream.

To create meaningful test cases, you must understand how the SUT works. This can be difficult. Techniques to discover information about the SUT are discussed in Section 4.8. This solution represents a minimal test harness. You can extend the harness, for example, by adding

summary counters of the number of test cases that pass and the number that fail by using the techniques in Chapter 1.

4.2 Creating a Text File Data, Buffered Model Test Harness

Problem

You want to create a test harness that uses text file test case data and a buffered processing model.

Design

Read all test case data into an `ArrayList` collection that holds lightweight `TestCase` objects. Then iterate through the test cases `ArrayList` object, executing each test case and storing the results into a second `ArrayList` object that holds lightweight `TestCaseResult` objects. Finally, iterate through the results `ArrayList` object, saving the results to an external text file.

Solution

Begin by creating lightweight `TestCase` and `TestCaseResult` classes:

```
class TestCase
{
  public string id;
  public string input;
  public string expected;

  public TestCase(string id, string input, string expected)
  {
    this.id = id;
    this.input = input;
    this.expected = expected;
  }
} // class TestCase

class TestCaseResult
{
  public string id;
  public string input;
  public string expected;
  public string actual;
  public string result;
```

```
  public TestCaseResult(string id, string input, string expected,
                        string actual, string result)
  {
    this.id = id;
    this.input = input;
    this.expected = expected;
    this.actual = actual;
    this.result = result;
  }
} // class TestCaseResult
```

Notice these class definitions use `public` data fields for simplicity. A reasonable alternative is to use a C# `struct` type instead of a `class` type. The data fields for the `TestCase` class should match the test case input data. The data fields for the `TestCaseResult` class should generally contain most of the fields in the `TestCase` class, the fields for the actual result of calling the CUT, and the test case pass or fail result. Because of this, a design option for you to consider is placing a reference to a `TestCase` object in the definition of the `TestCaseResult` class. For example:

```
class TestCaseResult
{
  public TestCase tc;
  public string actual;
  public string result;

  public TestCaseResult(TestCase tc, string actual, string result)
  {
    this.tc = tc;
    this.actual = actual;
    this.result = result;
  }
} // class TestCaseResult
```

You may also want to include fields for the date and time when the test case was run. You process the test case data using three loop control structures and two `ArrayList` objects like this:

```
Console.WriteLine("\nBegin Text File Buffered model test run\n");

FileStream ifs = new FileStream("..\\..\\..\\TestCases.txt",
                                FileMode.Open);
StreamReader sr = new StreamReader(ifs);
FileStream ofs = new FileStream("TextFileBufferedResults.txt",
                                FileMode.Create);
StreamWriter sw = new StreamWriter(ofs);

string id, input, expected = "", blank, actual;
TestCase tc = null;
TestCaseResult r = null;
```

```
// 1. read all test case data into memory
ArrayList tcd = new ArrayList(); // test case data
while (sr.Peek() != '*')
{
  id = sr.ReadLine().Split('=')[1];
  input = sr.ReadLine().Split('=')[1];
  expected = sr.ReadLine().Split('=')[1];
  blank = sr.ReadLine();
  tc = new TestCase(id, input, expected);
  tcd.Add(tc);
}
sr.Close(); ifs.Close();

// 2. run all tests, store results to memory
ArrayList tcr = new ArrayList(); // test case result
for (int i = 0; i < tcd.Count; ++i)
{
  tc = (TestCase)tcd[i];
  string[] cards = tc.input.Split(' ');
  Hand h = new Hand(cards[0], cards[1], cards[2], cards[3], cards[4]);
  actual = h.GetHandType().ToString();

  if (actual == tc.expected)
    r = new TestCaseResult(tc.id, tc.input, tc.expected,
                           actual, "Pass");
  else
    r = new TestCaseResult(tc.id, tc.input, tc.expected,
                           actual, "*FAIL*");

  tcr.Add(r);
} // main processing loop

// 3. emit all results to external storage
for (int i = 0; i < tcr.Count; ++i)
{
  r = (TestCaseResult)tcr[i];
  sw.WriteLine("====================");
  sw.WriteLine("ID       = " + r.id);
  sw.WriteLine("Input    = " + r.input);
  sw.WriteLine("Expected = " + r.expected);
  sw.WriteLine("Actual   = " + r.actual);
  sw.WriteLine(r.result);
}
sw.WriteLine("====================");

sw.Close(); ofs.Close();

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

Comments

The buffered processing model has three distinct phases. First, you read all test case data into memory. Although you can do this in many ways, experience has shown that your harness will be much easier to maintain if you create a very lightweight class for the test case data. Don't get carried away and try to make a universal test case class that can accommodate any kind of test case input, however, because you'll end up with a class that is so general it's too awkward to use effectively.

You have many choices of the kind of data structure to store your `TestCase` objects into. A `System.Collections.ArrayList` object is simple and effective. Because test case data is processed strictly sequentially in some situations, you may want to consider using a `Stack` or a `Queue` collection.

In the second phase of the buffered processing model, you iterate through each test case in the `ArrayList` object that holds `TestCase` objects. After retrieving the current `TestCase` object, you execute the test and determine a result. Then you instantiate a new `TestCaseResult` object and add it to the `ArrayList` that holds `TestCaseResult` objects. Although it's not a major issue, you do need to take some care to avoid confusing your objects. Notice that you'll have two `ArrayList` objects, a `TestCase` object and a `TestCaseResult` object, both of which contain a test case ID, test case input, and expected result.

In the third phase of the buffered processing model, you iterate through each test case result in the result `ArrayList` object and write information to an external text file. Of course, you can also easily emit results to an XML file, SQL database, or other external storage. If you run this code with the test case data file from Section 4.1

```
[id]=0001
[input]=Ac Ad Ah As Tc
[expected]=FourOfAKindAces
etc.
```

you will get the identical output as in Section 4.1:

```
====================
ID       = 0001
Input    = Ac Ad Ah As Tc
Expected = FourOfAKindAces
Actual   = FourOfAKindAces
Pass
====================
etc.
```

You can modularize this technique by writing three helper methods that wrap the code in the section. With these helper methods, your harness might look like:

```
class Class1
{
  static void Main(string[] args)
  {
    ArrayList tcd = null; // test case data
    ArrayList tcr = null; // test case results
    tcd = ReadData("..\\TestCases.txt");
    tcr = RunTests(tcd);
    SaveResults(tcr, "..\\TestResults.txt");
  }
  static ArrayList ReadData(string file)
  {
    // code here
  }
  static ArrayList RunTests(ArrayList testdata)
  {
    // code here
  }
  static void SaveResults(ArayList results, string file)
  {
    // code here
  }
}
class TestCase
{
  // code here
}
class TestCaseResult
{
  // code here
}
```

4.3 Creating an XML File Data, Streaming Model Test Harness

Problem

You want to create a test harness that uses XML file test case data and a streaming processing model.

Design

In one continuous processing loop, use an `XmlTextReader` object to read a test case into memory, then parse the test case data into input and expected values using the `GetAttribute()` and `ReadElementString()` methods, and call the CUT. Next, check the actual result with the

expected result to determine a test case pass or fail. Then, write the results to external storage using an `XmlTextWriter` object. Do this for each test case.

Solution

Begin by creating an XML test case file:

```
<?xml version="1.0" ?>
<testcases>
  <case id="0001">
    <input>Ac Ad Ah As Tc</input>
    <expected>FourOfAKindAces</expected>
  </case>
  <case id="0002">
    <input>4s 5s 6s 7s 3s</input>
    <expected>StraightSevenHigh</expected>
  </case>
  <case id="0003">
    <input>5d 5c Qh 5s Qd</input>
    <expected>FullHouseFivesOverQueens</expected>
  </case>
</testcases>
```

Then process the test case data using `XmlTextReader` and `XmlTextWriter` objects:

```
Console.WriteLine("\nBegin XML File Streaming model test run\n");

XmlTextReader xtr = new XmlTextReader("..\\..\\..\\TestCases.xml");
xtr.WhitespaceHandling = WhitespaceHandling.None;
XmlTextWriter xtw = new XmlTextWriter("XMLFileStreamingResults.xml",
                                      System.Text.Encoding.UTF8);
xtw.Formatting = Formatting.Indented;
string id, input, expected, actual;

xtw.WriteStartDocument();
xtw.WriteStartElement("TestResults"); // root node

while (!xtr.EOF) // main loop
{
  if (xtr.Name == "testcases" && !xtr.IsStartElement())
    break;

  while (xtr.Name != "case" || !xtr.IsStartElement())
    xtr.Read(); // go to a <case> element if not there yet
```

```
    id = xtr.GetAttribute("id");
    xtr.Read(); // advance to <input>
    input = xtr.ReadElementString("input"); // go to <expected>
    expected = xtr.ReadElementString("expected"); // go to </case>
    xtr.Read(); // go to next <case> or </testcases>

    string[] cards = input.Split(' ');
    Hand h = new Hand(cards[0], cards[1], cards[2], cards[3], cards[4]);
    actual = h.GetHandType().ToString();

    xtw.WriteStartElement("case");

    xtw.WriteStartAttribute("id", null);
    xtw.WriteString(id); xtw.WriteEndAttribute();

    xtw.WriteStartElement("input");
    xtw.WriteString(input); xtw.WriteEndElement();

    xtw.WriteStartElement("expected");
    xtw.WriteString(expected); xtw.WriteEndElement();

    xtw.WriteStartElement("actual");
    xtw.WriteString(actual); xtw.WriteEndElement();

    xtw.WriteStartElement("result");
    if (actual == expected)
      xtw.WriteString("Pass");
    else
      xtw.WriteString("*FAIL*");
    xtw.WriteEndElement(); // </result>

    xtw.WriteEndElement(); // </case>
  } // main loop

  xtw.WriteEndElement(); // </TestResults>
  xtr.Close();
  xtw.Close();

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

The `XmlTextReader.Read()` method advances one XML node at a time through the XML file. Because XML is hierarchical, keeping track of exactly where you are within the file is a bit tricky. To write results, you use an `XmlTextWriter` object with the `WriteStartElement()`, the `WriteString()`, and the `WriteEndElement()` methods, along with the `WriteStartAttribute()` and `WriteEndAttribute()` methods.

Comments

The use of XML for test case storage has become very common. The key to understanding this technique is to understand the `Read()` and `ReadElementString()` methods of the `System.Xml.XmlTextReader` class. To an `XmlTextReader` object, an XML file is a sequence of nodes. For example, if you do not count whitespace, the XML file

```
<?xml version="1.0" ?>
<foo att="x">
  <bar>99</bar>
</foo>
```

has six nodes: the XML declaration, `<foo>`, `<bar>`, `99`, `</bar>`, and `</foo>`. This means that the statement

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

in your harness is critical because without it you would have to keep track of blank lines, tab characters, end-of-line sequences, and so on. The `Read()` method advances one node at a time. Unlike many `Read()` methods in other classes, the `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, you have to extract attribute data using the `GetAttribute()` method.

When run with the preceding test case data, this code produces the following as output:

```
<?xml version="1.0" encoding="utf-8"?>
<TestResults>
  <case id="0001">
    <input>Ac Ad Ah As Tc</input>
    <expected>FourOfAKindAces</expected>
    <actual>FourOfAKindAces</actual>
    <result>Pass</result>
  </case>
  <case id="0002">
    <input>4s 5s 6s 7s 3s</input>
    <expected>StraightSevenHigh</expected>
    <actual>StraightFlushSevenHigh</actual>
    <result>*FAIL*</result>
  </case>
  <case id="0003">
    <input>5d 5c Qh 5s Qd</input>
    <expected>FullHouseFivesOverQueens</expected>
    <actual>FullHouseFivesOverQueens</actual>
    <result>Pass</result>
  </case>
</TestResults>
```

Because XML is so flexible, you can use many alternative structures. For example, you can store all data as attributes:

```
<?xml version="1.0" ?>
<testcases>
 <case id="0001" input="Ac Ad Ah As Tc" expected="FourOfAKindAces"/>
 <case id="0002" input="4s 5s 6s 7s 3s" expected="StraightSevenHigh"/>
 etc.
</testcases>
```

This flexibility characteristic of XML is both a strength and a weakness. From a lightweight test automation point of view, the main disadvantage of XML is that you have to slightly modify your test harness code for every XML test case data structure.

Processing an XML test case file with this loop structure:

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

  // process file here
}
```

may look a bit odd at first glance. The loop exits on end-of-file or when at the `</testcases>` tag. But this structure is more readable than alternatives. When marching through the XML file, you can either `Read()` your way one node at a time or get a bit more sophisticated with code such as:

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

The choice of technique you use is purely a matter of style. Writing an XML element with `XmlTextWriter` tends to be a bit wordy but is straightforward. For example:

```
xtw.WriteStartElement("alpha");
xtw.WriteStartElement("beta");

xtw.WriteString("b");

xtw.WriteEndElement(); // writes </beta>
xtw.WriteEndElement(); // writes </alpha>
```

would create

```
<alpha>
  <beta>b</beta>
</alpha>
```

Notice that the `WriteEndElement()` method does not accept an argument; the end element written is kept on an internal stack structure and popped off the stack.

Writing an XML attribute follows a pattern similar to writing an element. For example:

```
xtw.WriteStartElement("alpha");
xtw.WriteStartAttribute("beta", null);
xtw.WriteString("b");
xtw.WriteEndAttribute();
xtw.WriteEndElement();
```

produces as output:

```
<alpha beta="b" />
```

4.4 Creating an XML File Data, Buffered Model Test Harness

Problem

You want to create a test harness that uses XML file test case data and a buffered processing model.

Design

To create a harness structure that uses a buffered processing model with XML test case data, you follow the same pattern as in Section 4.2 combined with the XML reading and writing techniques demonstrated in Section 4.3. You read all test case data into an `ArrayList` collection that holds lightweight `TestCase` objects, iterate through that `ArrayList` object, execute each test case, store the results into a second `ArrayList` object that holds lightweight `TestCaseResult` objects, and finally save the results to an external XML file.

Solution

With lightweight `TestCase` and `TestCaseResult` classes in place (see Section 4.2), you can write:

```
Console.WriteLine("\nBegin XML File Buffered model test run\n");

XmlTextReader xtr = new XmlTextReader("..\\..\\..\\TestCases.xml");
xtr.WhitespaceHandling = WhitespaceHandling.None;
XmlTextWriter xtw = new XmlTextWriter("XMLFileStreamingResults.xml",
                                      System.Text.Encoding.UTF8);
xtw.Formatting = Formatting.Indented;

string id, input, expected, actual;
TestCase tc = null;
TestCaseResult r = null;
```

```
// 1. read all test case data into memory
ArrayList tcd = new ArrayList();
while (!xtr.EOF) // main loop
{
  if (xtr.Name == "testcases" && !xtr.IsStartElement()) break;

   while (xtr.Name != "case" || !xtr.IsStartElement())
     xtr.Read(); // advance to a <case> element if not there yet

   id = xtr.GetAttribute("id");
   xtr.Read(); // advance to <input>
   input = xtr.ReadElementString("input"); // advance to <expected>
   expected = xtr.ReadElementString("expected"); // advance to </case>
   tc = new TestCase(id, input, expected);
   tcd.Add(tc);

   xtr.Read(); // advance to next <case> or </TestResults>
}
xtr.Close();

// 2. run all tests, store results to memory
ArrayList tcr = new ArrayList();
for (int i = 0; i < tcd.Count; ++i)
{
  tc = (TestCase)tcd[i];
  string[] cards = tc.input.Split(' ');
  Hand h = new Hand(cards[0], cards[1], cards[2], cards[3], cards[4]);
  actual = h.GetHandType().ToString();

  if (actual == tc.expected)
    r = new TestCaseResult(tc.id, tc.input, tc.expected, actual, "Pass");
  else
    r = new TestCaseResult(tc.id, tc.input, tc.expected, actual, "*FAIL*");

  tcr.Add(r);
} // main processing loop

// 3. emit all results to external storage
xtw.WriteStartDocument();
xtw.WriteStartElement("TestResults"); // root node

for (int i = 0; i < tcr.Count; ++i)
{
  r = (TestCaseResult)tcr[i];
  xtw.WriteStartElement("case");

  xtw.WriteStartAttribute("id", null);
  xtw.WriteString(r.id); xtw.WriteEndAttribute();
```

```
  xtw.WriteStartElement("input");
  xtw.WriteString(r.input); xtw.WriteEndElement();

  xtw.WriteStartElement("expected");
  xtw.WriteString(r.expected); xtw.WriteEndElement();

  xtw.WriteStartElement("actual"); xtw.WriteString(r.actual);
  xtw.WriteEndElement();

  xtw.WriteStartElement("result");
  xtw.WriteString(r.result); xtw.WriteEndElement();

  xtw.WriteEndElement(); // </case>
}
xtw.WriteEndElement(); // </TestResults>

xtw.Close();

Console.WriteLine("\nEnd test run\n");
```

Comments

All the pertinent details to this technique are discussed in Sections 4.2 (buffered processing models) and 4.3 (reading and writing XML). If this code is run using the XML test case data file from Section 4.3:

```
<?xml version="1.0" ?>
<testcases>
  <case id="0001">
    <input>Ac Ad Ah As Tc</input>
    <expected>FourOfAKindAces</expected>
  </case>
  etc.
</testcases>
```

the output will be identical to that produced by the technique code in Section 4.3:

```
<?xml version="1.0" encoding="utf-8"?>
<TestResults>
  <case id="0001">
    <input>Ac Ad Ah As Tc</input>
    <expected>FourOfAKindAces</expected>
    <actual>FourOfAKindAces</actual>
    <result>Pass</result>
  </case>
  etc.
</TestResults>
```

Notice that this technique is starting to get a bit messy, mostly due to the large number of statements required to read and write XML. This makes it an excellent candidate for modularization by wrapping the code to read data, run tests, and save data into three helper methods. Furthermore, because the technique uses helper classes `TestCase` and `TestCaseResult`, recasting this solution to an OOP design is an attractive option. Such a design could take many forms, but here is one possibility:

```
class XMLBufferedHarness
{
    private ArrayList tcd = null; // test case data
    private ArrayList tcr = null; // test case results
    private XmlTextReader xtr = null;
    private XmlTextWriter xtw = null;

    public XMLBufferedHarness(string datafile, string resultfile)
    {
      // initialize tcd, tcr, xtr, xtw here
    }

    public void ReadData()
    {
      // use xtr to read datafile into tcd here
    }

    public void RunTests()
    {
      // run tests, store results to tcr here
    }

    public void SaveResults()
    {
      // save results to resultfile here
    }

    class TestCase
    {
      // see Section 4.2
    }

    class TestCaseResult
    {
      // see Section 4.2
    }

}
```

With this class in place, you can write very clean harness code like this:

```
static void Main(string[] args)
{
  string data = "TestCases.xml";
  string result = "TestResults.xml";
  XMLBufferedHarness h = new XMLBufferedHarness(data, result);
  h.ReadData();
  h.RunTests();
  h.SaveResults();
} // Main()
```

This approach has the advantage of being more modular than a non-OOP approach. However, the methods are very specific to a particular test scenario, meaning you'd have to significantly rewrite the methods for each CUT and associated XML test case file.

This technique uses an `XmlTextReader` object to iterate through the XML test case data file and store test case data into memory. You have two significant alternatives: the `XmlSerializer` class and the `XmlDocument` class. The techniques to use these classes to read and parse test case data into memory are explained in Chapter 12.

4.5 Creating a SQL Database for Lightweight Test Automation Storage

Problem

You want to create a SQL database for a lightweight test automation harness.

Design

Write a lightweight T-SQL script and run it using the Query Analyzer or the `osql.exe` programs.

Solution

```
-- makeDbTestPoker.sql
use master
go

if exists (select * from sysdatabases where name='dbTestPoker')
 drop database dbTestPoker
go

create database dbTestPoker
go

use dbTestPoker
go
```

```
create table tblTestCases
(
caseid char(4) primary key,
input char(14) not null,
expected varchar(35) not null,
)
go

insert into tblTestCases
 values('0001','Ac Ad Ah As Tc','FourOfAKindAces')
insert into tblTestCases
 values('0002','4s 5s 6s 7s 3s','StraightSevenHigh')
insert into tblTestCases
 values('0003','5d 5c Qh 5s Qd','FullHouseFivesOverQueens')
go

create table tblTestResults
(
resultid int identity(1,1) primary key,
caseid char(4) not null,
input char(14) not null,
expected varchar(35) not null,
actual varchar(35) not null,
result char(4) not null,
runat datetime not null
)
go
```

Comments

An alternative to using text files or XML files for your test case storage is to use a SQL database. SQL is particularly appropriate when you have many test cases (making the use of huge text files awkward) or when your SUT has a long development cycle (making management of many test case result files awkward).

To run a SQL script, you can paste the code into the Query Analyzer program that ships with Microsoft SQL Server, and execute it directly. An alternative is to run the script using the `osql.exe` command-line program, which also ships with SQL Server. If the preceding script is saved as `makeDbTestPoker.sql`, you can run it like this:

```
>osql -S(local) -E -i makeDbTestPoker.sql
```

The `-S` switch specifies the name of the SQL Server machine. The `-E` switch means to use a *trusted connection* (explained later in this section). The `-i` switch specifies the name of the SQL script to run.

The preceding script starts by setting the current database context to the "master" database, which is necessary to create or drop a database. Next, you check to see if the database `dbTestPoker` already exists by querying the `sysdatabases` system database. If `dbTestPoker` exists, then you drop it. Dropping a SQL database is surprisingly easy, so when using SQL for

test automation, be sure to back up your databases often. After creating database `dbTestPoker`, you switch context to that database. A common mistake is to forget to switch context, when all subsequent SQL commands will be directed at the master database. Next, you create a SQL table to hold test case data. The `primary key` argument to the `caseid` column means that each `caseid` value must be unique. The `not null` arguments mean that each test case must have an input and expected value. After creating the test case data table, you use the T-SQL `insert` command to populate the table. The last step is to create a table to hold test case results. Because each test run adds additional test results to the table, you usually want to include a column that holds the date and time when the test case result was added to the SQL database:

```
runat datetime not null
```

This technique creates a single database with a single test results table. An alternative approach is to create a new table for each test harness run. As a general rule, however, placing all harness run results into a single table is better than creating multiple tables—one table with thousands of rows of data is easier to manage than thousands of tables with any number of rows of data. If you do plan to put all test results into a single table, then you should create a column that uniquely identifies the test case result. The simplest way to do this is by adding an identity column to your test case data table definition:

```
resultid int identity(1,1) primary key
```

The `identity(1,1)` modifier instructs SQL Server to automatically generate an integer value for the `resultid` column, starting with value `1`, and increasing by 1 on each insert operation.

The technique in this section assumes that your test harness will be using a trusted connection. SQL Server requires a default Windows Authentication mode, which means in essence to integrate Windows security with SQL. This mode is the one used by a trusted connection. But SQL Server also supports an optional, additional SQL Authentication mode that can be used to gain access to SQL databases. The interaction between Windows Authentication and SQL Authentication modes can be tricky and is outside the scope of this book.

4.6 Creating a SQL Data, Streaming Model Test Harness

Problem

You want to create a test harness that uses SQL test case data and a streaming processing model.

Design

In one continuous processing loop, use a `SqlDataReader` object from the `System.Data.SqlClient` namespace to read a test case into memory from SQL, then parse the test case data into input and expected values using the `GetString()` method, and call the CUT. Next, check the actual result with the expected result to determine a test case pass or fail. Write the results to external storage using a `SqlCommand` object with a SQL `insert` statement as an argument. Do this for each test case. This technique assumes you have previously prepared a SQL database with test case data and a table to hold test results. See Section 4.5 for details.

Solution

```
using System.Data.SqlClient;
Console.WriteLine("\nBegin SQL Streaming model test run\n");

SqlConnection isc = new SqlConnection("Server=(local);
  Database=dbTestPoker;Trusted_Connection=yes");
SqlConnection osc = new SqlConnection("Server=(local);
  Database=dbTestPoker;Trusted_Connection=yes");

SqlCommand scSelect = new SqlCommand("SELECT * FROM tblTestCases", isc);
isc.Open();
osc.Open();
SqlDataReader sdr;
sdr = scSelect.ExecuteReader();
string caseid, input, expected, actual, result;

while (sdr.Read()) // main loop
{
  caseid = sdr.GetString(0); // parse input
  input = sdr.GetString(1);
  expected = sdr.GetString(2);
  string[] cards = input.Split(' ');

  Hand h = new Hand(cards[0], cards[1], cards[2], cards[3], cards[4]);
  actual = h.GetHandType().ToString();

  if (actual == expected) // emit results
    result = "Pass";
  else
    result = "FAIL";

  string runat = DateTime.Now.ToString("s");
  string insert = "INSERT INTO tblTestResults
    VALUES('" + caseid + "','" + input + "','" + expected +
            "','" + actual + "','" + result + "','" + runat + "')";
  SqlCommand scInsert = new SqlCommand(insert, osc);
  scInsert.ExecuteNonQuery();
} // while

sdr.Close();
isc.Close();
osc.Close();

Console.WriteLine("\nEnd test run\n");
```

Comments

Although there are several ways to iterate through a SQL table, the simplest is to use the `SqlDataReader` class. A `SqlDataReader` object gives you a way of reading a forward-only stream of rows from a SQL Server database. Notice that to create a `SqlDataReader` object, you use a factory mechanism by calling the `ExecuteReader()` method of the `SqlCommand` object, rather than directly by using a constructor and the `new` keyword. You also must prepare the `SqlCommand` object by passing in a T-SQL `select` statement to the `SqlCommand` constructor, so that the resulting `SqlDataReader` object knows how to traverse through the rows of its associated table.

If the code in this section is run with the input data from Section 4.5:

```
insert into tblTestCases
 values('0001','Ac Ad Ah As Tc','FourOfAKindAces')
insert into tblTestCases
 values('0002','4s 5s 6s 7s 3s','StraightSevenHigh')
insert into tblTestCases
 values('0003','5d 5c Qh 5s Qd','FullHouseFivesOverQueens')
```

then table `tblTestResults` in database `dbTestPoker` will hold this result data:

```
resultid  caseid  input             expected
=================================================================
1         0001    Ac Ad Ah As Tc    FourOfAKindAces
2         0002    4s 5s 6s 7s 3s    StraightSevenHigh
3         0003    5d 5c Qh 5s Qd    FullHouseFivesOverQueens

          actual                    result  runat
=================================================================
          FourOfAKindAces           Pass    2006-06-15 07:50:20.000
          StraightFlushSevenHigh    FAIL    2006-06-15 07:50:20.000
          FullHouseFivesOverQueens  Pass    2006-06-15 07:50:20.000
```

The values in the `runat` column will be the date and time when the results were inserted into the SQL table.

You use the `SqlDataReader.GetString()` method to extract each column value as a string. The `GetString()` method accepts a zero-based column index rather than a column name as a string as you might expect. So you must write `caseid = sdr.GetString(0);` rather than `caseid = sdr.GetString("caseid");` which would be more readable.

If you insert all test case results into one SQL table rather than creating a new table to hold the results of each test run, you usually should time-stamp the result. A simple way to do this is to fetch the current system date and time:

```
string runat = DateTime.Now.ToString("s");
```

The `"s"` argument will format the `DateTime` object into a sortable pattern such as:

```
'2006-09-20T11:46:41.000'
```

SQL Server understands this format, and a C# string variable in this format is converted into a SQL `datetime` data type automatically when you insert it into a `datetime` column.

The technique used in this solution to insert a row of data into the SQL test results table is rather ugly. If you were inserting literals into the results table, code might look like this:

```
string insert = "INSERT INTO tblTestResults
 VALUES('0001', 'Ac Ad Ah As Kc', 'FourOfAKindAces',
        'FourOfAKindAces', 'Pass', '2006-09-20 11:46:41.000')";
```

But because you are inserting values stored in variables, you have to build up a fairly complex insert string like this:

```
string insert = "INSERT INTO tblTestResults VALUES('" + caseid +
 "','" + input + "','" + expected + "','" + actual + "','" + result +
 "','" + runat + "')";
```

Creating such SQL strings can be an error-prone process, so you must be careful when coding them. An alternative to writing such long strings is to create a SQL stored procedure and then call it from your harness. For example, if your SQL database creation script contains this user stored procedure T-SQL code:

```
create procedure usp_insert
@caseid char(4),
@input char(14),
@expected varchar(35),
@actual varchar(35),
@result char(4),
@runat datetime

as

insert into tblTestResults
values(@caseid, @input, @expected, @actual, @result, @runat)

go
```

then you can prepare a `SqlCommand` object like this:

```
SqlConnection osc = new SqlConnection("Server=(local);
  Database=dbTestPoker; Trusted_Connection=yes");

SqlCommand sp = new SqlCommand("usp_insert", osc);
sp.CommandType = CommandType.StoredProcedure;

SqlParameter paramCaseID = sp.Parameters.Add("@caseid",
                                             SqlDbType.Char, 4);
SqlParameter paramInput = sp.Parameters.Add("@input",
                                            SqlDbType.Char, 14);
SqlParameter paramExpected = sp.Parameters.Add("@expected",
                                               SqlDbType.VarChar, 35);
```

```
SqlParameter paramActual = sp.Parameters.Add("@actual",
                                             SqlDbType.VarChar, 35);
SqlParameter paramResult = sp.Parameters.Add("@result",
                                             SqlDbType.Char, 4);
SqlParameter paramRunAt = sp.Parameters.Add("@runat",
                                            SqlDbType.DateTime);

osc.Open();
```

And then in the main processing loop, you can insert test case results in SQL like this:

```
// read caseid, input, expected from test case data here
// run test and get actual, result here
string runat = DateTime.Now.ToString("s");

paramCaseID.Value = caseid;
paramInput.Value = input;
paramExpected.Value = expected;
paramActual.Value = actual;
paramResult.Value = result;
paramRunAt.Value = runat;

sp.ExecuteNonQuery(); // insert using usp_insert
```

This technique has the advantage of reducing complexity by eliminating an ugly SQL `insert` command string, but has the disadvantage of increasing complexity by adding many more lines of code to your test harness.

4.7 Creating a SQL Data, Buffered Model Test Harness

Problem

You want to create a test harness that uses SQL test case data and a buffered processing model.

Design

To create a harness structure that uses a buffered processing model with SQL test case data, you follow the same pattern as in Section 4.2 combined with the SQL reading and writing techniques demonstrated in Section 4.6. You use a `SqlDataReader` object to read all test case data into an `ArrayList` collection that holds lightweight `TestCase` objects. Next, you iterate through that `ArrayList` object, execute each test case, and store the results into a second `ArrayList` object that holds lightweight `TestCaseResult` objects. Then you save the results to an external SQL database.

Solution

With lightweight `TestCase` and `TestCaseResult` classes in place (see Section 4.2), you can write:

```
Console.WriteLine("\nBegin SQL Buffered model test run\n");

SqlConnection isc = new SqlConnection("Server=(local);
  Database=dbTestPoker; Trusted_Connection=yes");
SqlConnection osc = new SqlConnection("Server=(local);
  Database=dbTestPoker;Trusted_Connection=yes");
isc.Open();
osc.Open();

SqlCommand scSelect = new SqlCommand("SELECT * FROM tblTestCases", isc);
SqlDataReader sdr = scSelect.ExecuteReader();

string caseid, input, expected = "", actual;
TestCase tc = null; // see Section 4.2
TestCaseResult r = null;

// 1. read all test case data into memory
ArrayList tcd = new ArrayList();
while (sdr.Read()) // main loop
{
  caseid = sdr.GetString(0);
  input = sdr.GetString(1);
  expected = sdr.GetString(2);
  tc = new TestCase(caseid, input, expected);
  tcd.Add(tc);
}
isc.Close();

// 2. run all tests, store results to memory
ArrayList tcr = new ArrayList();
for (int i = 0; i < tcd.Count; ++i)
{
  tc = (TestCase)tcd[i];
  string[] cards = tc.input.Split(' ');
  Hand h = new Hand(cards[0], cards[1], cards[2], cards[3], cards[4]);
  actual = h.GetHandType().ToString();

if (actual == tc.expected)
  r = new TestCaseResult(tc.id, tc.input, tc.expected, actual, "Pass");
else
  r = new TestCaseResult(tc.id, tc.input, tc.expected, actual, "FAIL");

tcr.Add(r);
} // main processing loop
```

```
// 3. emit all results to external SQL storage
for (int i = 0; i < tcr.Count; ++i)
{
  r = (TestCaseResult)tcr[i];
  string runat = DateTime.Now.ToString("s");
  string insert = "INSERT INTO tblTestResults
    VALUES('" + r.id + "','" + r.input + "','" + r.expected +
           "','" + r.actual + "','" + r.result + "','" + runat + "')";
SqlCommand scInsert = new SqlCommand(insert, osc);
scInsert.ExecuteNonQuery();
}
osc.Close();

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

Comments

All the pertinent details to this technique are discussed in Sections 4.2 and 4.4 (buffered processing models), and Section 4.6 (reading and writing SQL). If the following code is run using the SQL test case data file from Section 4.5:

```
insert into tblTestCases
 values('0001','Ac Ad Ah As Tc','FourOfAKindAces')
insert into tblTestCases
 values('0002','4s 5s 6s 7s 3s','StraightSevenHigh')
insert into tblTestCases
 values('0003','5d 5c Qh 5s Qd','FullHouseFivesOverQueens')
```

then the output will be identical to that produced by the technique in Section 4.6:

```
resultid  caseid  input             expected
=================================================================
1         0001    Ac Ad Ah As Tc    FourOfAKindAces
2         0002    4s 5s 6s 7s 3s    StraightSevenHigh
3         0003    5d 5c Qh 5s Qd    FullHouseFivesOverQueens

          actual                    result  runat
=================================================================
          FourOfAKindAces           Pass    2006-06-15 07:50:20.000
          StraightFlushSevenHigh    FAIL    2006-06-15 07:50:20.000
          FullHouseFivesOverQueens  Pass    2006-06-15 07:50:20.000
```

Using a buffered test automation-processing model makes it easy for you to perform test case data filtering or test case results filtering. For example, suppose you want to filter your test cases so that only certain suites of tests are run rather than all your tests. *Test suite* means a collection of test cases, usually a subset of a larger set of tests. Following are examples of common test suite categorizations:

- Developer Regression Tests (DRTs): A set of tests run on some new code (typically a set of classes or methods) before a developer checks in the code to the main build system. Designed to verify that the new code has not broken existing functionality.
- Build Verification Tests (BVTs): A set of tests run on a new build of the SUT immediately after the build process. Designed to verify that the new build has minimal functionality and can be released to the test team for further testing.
- Daily Test Runs (DTRs): A set of tests run by the test team every day. Designed to verify that previous functionality is still correct, uncover new functionality and performance bugs, and so on.
- Weekly Test Runs (WTRs): A set of tests that is more extensive than Daily Test Run test cases but only run once a week due primarily to time constraints.
- Milestone Test Runs (MTRs): A comprehensive set of tests run before the release of a major or minor milestone. May require several days to run.
- Full Test Pass (FTP): Running every test case available. Typically requires several days to run.

Of course, there are many variations on these categories of test suites, but the general principle is that you'll have many test cases and you'll run various subsets of test cases at different times. This holds true whether you are working in a traditional spiral software development methodology environment or in any of a number of currently fashionable methodologies, such as test-driven development, extreme programming, agile development, and so on.

4.8 Discovering Information About the SUT

Problem

You want to discover information about the SUT so that you can create meaningful test cases.

Solution

One of the greatest challenges of software testing in almost any environment is discovering the essential information about the SUT (SUT) so that you can test it meaningfully. There are six primary ways to perform system discovery in a .NET environment:

- Read traditional specification documents.
- Examine SUT source code.
- Write experimental stub programs.
- Use XML auto-documentation.
- Examine .NET intermediate language code.
- Use reflection techniques.

Comments

In a very small production environment where developers test their own code, system discovery may not be an issue. As the size of a development effort increases, however, the discovery process becomes more difficult. The most common approach is for you to read traditional written specification documents that describe the SUT. In theory at least, every system has a set of documents, usually written by senior developers, managers, or architects, that completely and precisely describes the SUT. In reality, of course, such specification documents are often out-of-date, incomplete, or even nonexistent. Regardless, examining traditional specification documents is an important way to determine how to create meaningful test cases.

You can examine the source code of the SUT to gain insights on how to test your system, although in some cases, this may not be possible for security or legal reasons. Even when source code examination is possible, reviewing the source code for a complex SUT can be enormously time consuming. When you have access to system source code while developing test cases, the situation is sometimes called *white box* or *clear box* testing. When you do not have access to source code, the situation is sometimes called *black box* testing. When you have partial access to system source code, for example, the signatures of methods but not the body of the method, the situation is sometimes called *gray box* testing. These labels are some of the most overused but least-useful terms in software testing. However, the principles behind these labels are important. You cannot test every possible input to a system (see Chapter 10 for discussions of this idea), so the more you know about your SUT, the better your test cases will be. Although there has been much research in the area of automatic test case generation, currently test case development is still for the most part a human activity where experience and intuition play a big role.

A third discovery mechanism available to you is to experiment with the SUT by creating small stub programs. Again, this is not always possible for legal and security reasons and even when possible, it may not be a realistic technique: large software systems can be so complex that trying to understand them through experimentation just requires too much time. The development environment is often so dynamic that by the time you've figured a part of the system out, it has changed. This is not to say that experimentation is not important. On the contrary, initial experimentation with stub programs is usually the key first step when developing lightweight test automation.

The Visual Studio .NET IDE allows developers to add XML-based comments into their source code and have an XML-based document created automatically at project build time. In source code files, lines that begin with "`///`" and that precede user-defined items such as classes, delegates, interfaces, fields, events, properties, methods, or namespace declarations, can be processed as comments and placed in a file. There is a recommended set of tags. For example, the `<param>` tag is used to describe parameters. When used, the compiler verifies that the parameter exists and that all parameters are described in the documentation. This mechanism requires developers to expend extra effort, but the payoff is that system specs are always up to date.

Because .NET-compliant languages compile to an intermediate language, a terrific way to expose information about a SUT is to examine the SUT's intermediate language. The .NET environment provides developers and testers with a tool named ILDASM. The ILDASM tool parses .NET Framework `.exe` or `.dll` assemblies and shows the information in human-readable format. ILDASM also displays namespaces and types, including their interfaces. The use of ILDASM for system discovery is essential for any lightweight test automation situation.

The sixth primary way for you to discover information about the SUT is through the .NET reflection mechanism. Reflection means the process of programmatically obtaining information about .NET assemblies and the types defined within them. Using classes in the `System.Reflection` namespace, you can easily write short utility scripts that expose a wide range of data about the SUT. For example:

```
Console.WriteLine("\nBegin Reflection Discovery");
string assembly = "..\\..\\..\\LibUnderTest\\PokerLib.dll";
Assembly a = Assembly.LoadFrom(assembly);
Console.WriteLine("Assembly name = " + a.GetName());

Type[] tarr = a.GetTypes();
BindingFlags flags = BindingFlags.NonPublic | BindingFlags.Public |
                     BindingFlags.Static | BindingFlags.Instance;

foreach(Type t in tarr)
{
  Console.WriteLine(" Type name = " + t.Name);

  MemberInfo[] members = t.GetMembers(flags);
  foreach (MemberInfo mi in members) // fields, methods, ctors, etc.
  {
    if (mi.MemberType == MemberTypes.Field)
      Console.WriteLine("  (Field) member name = " + mi.Name);
  } // each member

  MethodInfo[] miarr = t.GetMethods(); // public only
  foreach (MethodInfo mi in miarr)
  {
    Console.WriteLine("   Method name = " + mi.Name);
    Console.WriteLine("    Return type = " + mi.ReturnType);
    ParameterInfo[] piarr = mi.GetParameters();
    foreach (ParameterInfo pi in piarr)
    {
      Console.WriteLine("    Parameter name = " + pi.Name);
      Console.WriteLine("    Parameter type = " + pi.ParameterType);
    }
  } // each method
} // each Type

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

This example loads the `PokerLib.dll` assembly and then iterates through each type (classes, enumerations, interfaces, and so on) in the assembly. Then for each type, you iterate through each member (fields, methods, properties, constructors, and so on), printing some information if you hit a field. After iterating through the members, you iterate through each method, printing the method's name, return type, parameter names, and parameter types.

4.9 Example Program: PokerLibTest

This demonstration program combines several of the techniques in this chapter to create a lightweight test automation harness to test the `PokerLib.dll` library described in Section 4.1. The harness reads test case data from a SQL database, processes test cases using a buffered model, and emits test results to an XML file. If the test case input is

```
caseid  input           expected
================================================
0001    Ac Ad Ah As Tc  FourOfAKindAces
0002    4s 5s 6s 7s 3s  StraightSevenHigh
0003    5d 5c Qh 5s Qd  FullHouseFivesOverQueens
```

then the resulting XML output (where the `runat` attribute will be the value of the date and time the harness executed) is

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

  <case id="0001" runat="2006-10-28T12:41:36">
    <input>Ac Ad Ah As Tc</input>
    <expected>FourOfAKindAces</expected>
    <actual>FourOfAKindAces</actual>
    <result>Pass</result>
  </case>

  <case id="0002" runat="2006-10-28T12:41:36">
    <input>4s 5s 6s 7s 3s</input>
    <expected>StraightSevenHigh</expected>
    <actual>StraightFlushSevenHigh</actual>
    <result>FAIL</result>
  </case>

  <case id="0003" runat="2006-10-28T12:41:36">
    <input>5d 5c Qh 5s Qd</input>
    <expected>FullHouseFivesOverQueens</expected>
    <actual>FullHouseFivesOverQueens</actual>
    <result>Pass</result>
  </case>

</TestResults>
```

The complete lightweight test harness is presented in Listing 4-1.

Listing 4-1. *Program PokerLibTest*

```
using System;
using System.Collections;
using System.Data.SqlClient;
using System.Xml;
using PokerLib;

namespace PokerLibTest
{
  class Class1
  {
    [STAThread]
    static void Main(string[] args)
    {
      try
      {
        Console.WriteLine("\nBegin PokerLibTest run\n");

        SqlConnection isc = new SqlConnection("Server=(local);
          Database=dbTestPoker;Trusted_Connection=yes");
        isc.Open();
        SqlCommand scSelect = new SqlCommand("SELECT * FROM tblTestCases",
                                             isc);
        SqlDataReader sdr = scSelect.ExecuteReader();

        string caseid, input, expected = "", actual;
        TestCase tc = null;
        TestCaseResult r = null;

        // 1. read all test case data from SQL into memory
        ArrayList tcd = new ArrayList();
        while (sdr.Read())
        {
          caseid = sdr.GetString(0);
          input = sdr.GetString(1);
          expected = sdr.GetString(2);
          tc = new TestCase(caseid, input, expected);
          tcd.Add(tc);
        }
        isc.Close();

        // 2. run all tests, store results to memory
        ArrayList tcr = new ArrayList();
        for (int i = 0; i < tcd.Count; ++i)
        {
          tc = (TestCase)tcd[i];
          string[] cards = tc.input.Split(' ');
```

```
  Hand h = new Hand(cards[0], cards[1], cards[2],
                    cards[3], cards[4]);
  actual = h.GetHandType().ToString();
  string runat = DateTime.Now.ToString("s");

  if (actual == tc.expected)
    r = new TestCaseResult(tc.id, tc.input, tc.expected,
                           actual, "Pass", runat);
  else
    r = new TestCaseResult(tc.id, tc.input, tc.expected,
                           actual, "FAIL", runat);
  tcr.Add(r);
}

// 3. emit all results to external XML storage
XmlTextWriter xtw = new XmlTextWriter("PokerLibResults.xml",
  System.Text.Encoding.UTF8);
xtw.Formatting = Formatting.Indented;
xtw.WriteStartDocument();
xtw.WriteStartElement("TestResults"); // root node

for (int i = 0; i < tcr.Count; ++i)
{
  r = (TestCaseResult)tcr[i];
  xtw.WriteStartElement("case");

  xtw.WriteStartAttribute("id", null);
  xtw.WriteString(r.id); xtw.WriteEndAttribute();

  xtw.WriteStartAttribute("runat", null);
  xtw.WriteString(r.runat); xtw.WriteEndAttribute();

  xtw.WriteStartElement("input");
  xtw.WriteString(r.input); xtw.WriteEndElement();

  xtw.WriteStartElement("expected");
  xtw.WriteString(r.expected); xtw.WriteEndElement();

  xtw.WriteStartElement("actual");
  xtw.WriteString(r.actual); xtw.WriteEndElement();

  xtw.WriteStartElement("result");
  xtw.WriteString(r.result); xtw.WriteEndElement();

  xtw.WriteEndElement(); // </case>
}
xtw.WriteEndElement(); // </TestResults>
xtw.Close();
```

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

    class TestCase
    {
      public string id;
      public string input;
      public string expected;

      public TestCase(string id, string input, string expected)
      {
        this.id = id;
        this.input = input;
        this.expected = expected;
      }
    } // class TestCase

    class TestCaseResult
    {
      public string id;
      public string input;
      public string expected;
      public string actual;
      public string result;
      public string runat;

      public TestCaseResult(string id, string input, string expected,
        string actual, string result, string runat)
      {
        this.id = id;
        this.input = input;
        this.expected = expected;
        this.actual = actual;
        this.result = result;
        this.runat = runat;
      }
    } // class TestCaseResult

  } // Class1
} // ns
```