CHAPTER 6

Script-Based Web UI Testing

6.0 Introduction

The simplest form of Web application testing is manual testing through the UI; however, because manual testing is often slow, inefficient, and tedious, a good strategy is to supplement manual testing with basic Web application UI test automation. You can do this in several ways. The oldest technique uses JavaScript to manipulate a Web application's controls through the Internet Explorer Document Object Model (IE DOM). The best way to demonstrate this type of testing is visually, so Figure 6-1 shows a sample run of a script-based Web UI test harness.

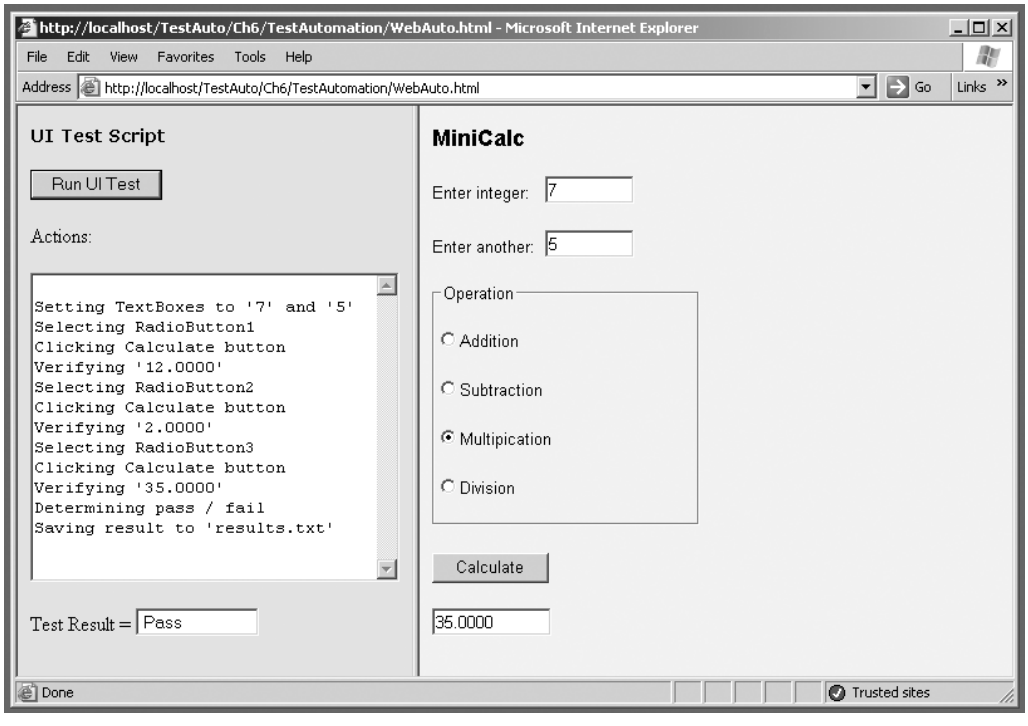

Figure 6-1. *Script-based Web application UI testing*

If you examine Figure 6-1, you'll see that the test harness is a Web page with two frames. The right frame hosts the Web AUT; its display title is MiniCalc. In this example, the application is a simple calculator program. The left frame hosts JavaScript functions that manipulate the Web AUT, examine the resulting state of the application, and log test results to an external file. This chapter presents the various techniques you need to perform script-based Web UI test automation.

Most of the sections in this chapter reference the Web application shown in the right frame in Figure 6-1. The application is named `WebApp.aspx`. The entire code for the application is

```
<%@ Page Language="C#" Debug="true" %>

<script language="c#" runat="server">
  private void Button1_Click(object sender, System.EventArgs e)
  {
    int alpha = int.Parse(TextBox1.Text.Trim());
    int  beta = int.Parse(TextBox2.Text.Trim());

    if (RadioButton1.Checked)
    {
      TextBox3.Text = Sum(alpha, beta).ToString("F4");
    }
    else if (RadioButton2.Checked)
    {
      TextBox3.Text = Diff(alpha, beta).ToString("F4");
    }
    else if (RadioButton3.Checked)
    {
      TextBox3.Text = Product(alpha, beta).ToString("F4");
    }
    else
     TextBox3.Text = "Select method";
    }

    private static double Sum(int a, int b)
    {
      double ans = 0.0;
      ans = a + b;
      return ans;
    }

    private static double Diff(int a, int b)
    {
      double ans = 0.0;
      ans = a - b;
      return ans;
    }
```

```
    private static double Product(int a, int b)
    {
      double ans = 0.0;
      ans = a * b;
      return ans;
    }
</script>

<html>
  <head><title>WebApp.aspx</title></head>
  <style type="text/css">
   fieldset { width: 16em }
   body { font-size: 10pt; font-family: Arial }
  </style>
  <body bgColor="#ccffff">
    <h3>MiniCalc</h3>
    <form method="post" name="theForm" id="theForm" runat="server">
      <p><asp:Label id="Label1" runat="server">Enter integer:  
        </asp:Label>
      <asp:TextBox id="TextBox1" size="6" runat="server" /></p>
      <p><asp:Label id="Label2" runat="server">Enter another: 
        </asp:Label>
      <asp:TextBox id="TextBox2" size="6" runat="server" /></p>
      <p></p>

      <fieldset>
       <legend>Operation</legend>
       <p><asp:RadioButton id="RadioButton1" GroupName="ops"
         runat="server"/>Addition</p>
       <p><asp:RadioButton id="RadioButton2" GroupName="ops"
         runat="server"/>Subtraction</p>
       <p><asp:RadioButton id="RadioButton3" GroupName="ops"
         runat="server"/>Multipication</p>
       <p><asp:RadioButton id="RadioButton4" GroupName="ops"
         runat="server"/>Division</p>
       <p></p>
      </fieldset>

      <p><asp:Button id="Button1" runat="server" text=" Calculate "
        onclick="Button1_Click" /> </p>

      <p><asp:TextBox id="TextBox3" size="10" runat="server" />
    </form>
  </body>
</html>
```

For simplicity, all the Web application code is contained in a single source file rather than the more usual approach of separating HTML display code and C# (or other .NET-compliant language) code into two separate files. If you examine this code, you'll see that the UI contains two text fields where the user enters two integers, four radio button controls that the user selects to indicate which of four arithmetic operations to perform (addition, subtraction, multiplication, division), a button control to submit the calculation request, and a third text field that displays a result with four decimals.

■ **Note** Notice that the label next to the multiplication radio button control is misspelled as "Multipication". Typographical errors in AUTs are common during the testing phase, so be prepared to deal with them when writing automation.

This Web application is simplistic, and your Web AUTs are likely to be much more complex. However, this application has all the key components necessary to demonstrate script-based UI testing. Even if your Web AUT does sophisticated numerical processing or fetches complex data from a SQL data store, each HTTP request-response will result in a new page state that is reflected in the UI.

The code in this chapter assumes that the automation is organized with a root folder containing two subfolders named `TheWebApp` and `TestAutomation`. The `TheWebApp` folder holds the Web AUT (`WebApp.aspx`). The `TestAutomation` folder contains the main test harness structure as a single Web page (`WebAuto.html`) and the page that houses the JavaScript code which runs the test scenario (`TestCode.html`).

Related but lower-level techniques to test a Web application through its UI are presented in Chapter 7. The techniques in this chapter can handle most basic UI testing situations but cannot deal with configurations that have JavaScript disabled. These techniques also cannot manipulate objects that are outside the browser client area (such as alert dialog boxes). The test harness that produced the test run shown in Figure 6-1 is presented in Section 6.8.

6.1 Creating a Script-Based UI Test Harness Structure

Problem

You want to create a structure for a script-based Web application UI test harness that allows you to programmatically manipulate, synchronize, and examine the Web AUT.

Design

Create an HTML page with two frames. One frame hosts the AUT. The second frame hosts the test harness script. The containing HTML page holds synchronization variables and test scenario meta-information.

Solution

```
<html>
  <head>
    <script language="JavaScript">
      var description = "Description of test scenario";
      var loadCount = 0;
      var pass = true;
    </script>
  </head>

  <frameset cols="40%,*">
    <frame src="TestCode.html" name="leftFrame">
    <frame src="../TheWebApp/WebApp.aspx" name="rightFrame"
      onload="leftFrame.updateState();">
  </frameset>
</html>
```

Comments

Although you can structure a script-based Web application UI test harness in several ways, the organization presented here has proven simple and effective in practice. The `<script>` portion of the HTML harness holds three key variables. Notice we use the `language="JavaScript"` attribute. In a pure Microsoft technology environment, you might want to use `"JScript"` to emphasize the fact that you are using the IE DOM to access Web page controls. The first variable, `description`, is test scenario meta-information. You may want to include other meta-information here such as a test scenario ID or the date and time when the scenario was run. The second variable, `loadCount`, is the key to test harness synchronization. Because HTTP is a stateless protocol, each request-response pair is independent. You need some way to know which state the Web application is in. The easiest way to do this is to use a global variable where a value of `0` indicates an initial state, a value of `1` indicates the next state (after a user clicks a submit button for example), and so on. Observe that when the Web page/document under test finishes loading into the right frame of the test harness, control is transferred to a function `updateState()`located in the page in the left (test code) frame:

```
onload="leftFrame.updateState();"
```

Section 6.2 describes the `updateState()` function. The third variable in the HTML harness page, `pass`, represents the test scenario pass or fail result.

The body of the HTML test harness page just contains two frames, `leftFrame` and `rightFrame`, in this solution. The frames are organized into two columns with the first (left) column receiving 40% of the display area. There's nothing special about the column organization or frame names used here. Using the names `leftFrame` and `rightFrame` implies you have the frames organized in a particular way, but experience has shown that using positionally oriented frame names tends to be easier to read than functionally oriented names such as `frameWithWebApp` and `frameWithHarnessCode`, although this is a matter of personal preference. Frame `rightFrame` holds the AUT. The application does not need to be instrumented in any way, and the techniques in this chapter apply to both classic ASP Web applications and ASP.NET Web applications. Frame `leftFrame` holds the test scenario JavaScript code that manipulates the AUT.

A common mistake when performing script-based UI testing is to attempt to synchronize events by using the `setTimeout()` method to pause the test automation. Calling `setTimeout()` stops the thread of execution. Unfortunately, because IE runs under a single thread of execution (with a few rare exceptions), you end up pausing both your test automation and the AUT.

6.2 Determining Web Application State

Problem

You want to determine the state of the Web AUT.

Design

In the `TestCode.html` page that holds the JavaScript test harness code in the preceding solution, write a function `updateState()` that increments the global state counter variable and then calls the main test logic.

Solution

```
<html>
  <head>
    <script language="JavaScript">
      function updateState()
      {
        parent.loadCount++;
          if (parent.loadCount > 1) // > 0 for full-automation
            runTest();
      } // updateState()

      function runTest()
      {
        // runTest() code here
      }

      // other test functions here

    </script>
  </head>
  <body bgColor="#aaff99">
    <h3 style="font-size: 14; font-family: Verdana">UI Test Script
    </h3>
    <p><input type="button" value="Run UI Test" onclick="runTest();">
    </p>
    <p>Actions:</p><p><textarea id="comments" rows="15" cols="34">
    </textarea></p>
    <p>Test Result = <input type="text" name="result" size="12"></p>
  </body>
</html>
```

Comments

If you create a test harness structure as described in Section 6.1, when the Web AUT finishes loading into the test harness right frame, control is transferred to function `updateState()` located in the script part of the page located in the left frame. This state-updating function first increments the global application state counter:

```
parent.loadCount++;
```

Because the state counter is located in the main test harness structure page, you must access it using the `parent` keyword. Next, the `updateState()` function checks if the value of the global state counter is greater than 1. Because the counter is initialized to `0`, the counter has a value of `1` when the test harness first launches, which, in turn, loads the Web AUT. If you check for a value greater than 1, the condition is false on the initial page load so the thread of execution stops. This allows you to manually view the test harness and Web AUT, and then launch the test automation manually. If you want full automation, you can edit the condition to

```
if (parent.loadCount > 0)
  runTest();
```

This condition is true on the initial application page load into the test harness structure, and control is immediately transferred to function `runTest()`.

In this solution, the page located in the left frame of the containing test harness page is named `TestCode.html`. In a fully automated situation, such as just described, you do not need any UI for page `WebAuto.html`. However, some minimal UI is required if you want to manually launch the test automation:

```
<body bgColor="#aaff99">
  <h3 style="font-size: 14; font-family: Verdana">UI Test Script</h3>
  <p><input type="button" value="Run UI Test" onclick="runTest();"></p>
  <p>Actions:</p>
  <p><textarea id="comments" rows="15" cols="34"></textarea></p>
  <p>Test Result = <input type="text" name="result" size="12"></p>
</body>
```

You give a title to the page containing the JavaScript automation code so that other testers and developers can clearly distinguish which frame holds the AUT and which frame holds the test automation. You supply a button control so that testers can manually launch the test automation as described previously. An HTML `<textarea>` element is handy to display messages containing information about the progress of the test automation as shown in Figure 6-1. Finally, you add a text field so that the overall test scenario pass/fail result can be displayed in a way that stands out from other messages.

6.3 Logging Comments to the Test Harness UI

Problem

You want to display messages that detail the progress of the test automation.

Design

Write a JavaScript helper function `logRemark()` that uses the IE DOM `value` property to set a value into an HTML `<textarea>` comments field.

Solution

```
function logRemark(comment)
{
  var currComment = document.all["comments"].value;
  var newComment = currComment + "\n" + comment;
  document.all["comments"].value = newComment;
} // logRemark()
```

Comments

Although the goal of any test scenario is to produce a pass or fail result, it's useful to have a way to display the progress of the automation. This helps you diagnose the inevitable problems you'll run into and sometimes reveals bugs in the Web AUT as well. The simple `logRemark()` function accepts a comment to log as the single input argument. Notice that JavaScript is a nontyped language, so you don't specify exactly what data type the comment parameter is. The function first grabs any existing content in the `textarea` named `"comments"` using the `value` property and the `document.all` collection. See Section 6.2 for the definition of the comments HTML `<textarea>` element. The function then appends a newline character to the existing comments contents and then appends the text of the input argument comment using the JavaScript + string concatenation operator. The `logRemark()` function finishes by replacing the value of the old comments contents with the newly updated value.

With this helper function in hand, you can enhance the readability and clarity of your test harness output by displaying various messages as the test scenario runs. For example:

```
logRemark("Starting test automation");
logRemark("About to set TextBoxes to '7' and '5'");
```

6.4 Verifying the Value of an HTML Element on the Web AUT

Problem

You want to verify that an HTML element on the Web AUT has a certain value and set a test scenario pass/fail result to the appropriate value.

Design

Write a function `verify()` that accepts a reference to a control element and an expected value for the element and sets a global pass/fail result variable that has been initialized to true (to false if the actual value of the control does not equal the expected value).

Solution

```
function verify(ctrl, val)
{
  if (parent.rightFrame.document.all[ctrl].value != val)
    parent.pass = false;
}
```

Comments

The `verify()` function accepts a reference to a control and an expected value for the control. The function assumes the existence of a global variable `pass` located in the containing harness structure Web page as described in Section 6.1. Notice that to access a control in the Web AUT from the left frame, you must "go up" one page using the `parent` keyword and then "down" one page into the application using the frame name. If the actual value in the specified control is not equal to the expected value argument, the global pass variable is set to false. This scheme assumes that variable `pass` has been initialized to true. In other words, the logic used here is that you assume the test scenario will pass. After each state change, you check one or more controls looking for an inconsistent value; if you find such a problem, you set `pass` to false. An alternative approach is to assume the test scenario will fail. Then after all the state changes, you check for a series of consistency values and set `pass` to true only if all expected conditions/values are met.

The heart of the techniques in this chapter is the capability to access the HTML elements on a Web page using the IE DOM. This is a large topic because the IE DOM has more than 500 properties and nearly as many methods. From a testing point of view, you'll use the `value` property most often to verify the state of an HTML element, but you'll find other properties useful too. For example, suppose you need to check whether the background color of the Web AUT is pure red. You can write code like this:

```
if (parent.rightFrame.document.bgColor == "#FF0000")
  backgroundIsRed = true;
```

As another example, suppose you want to check whether some `Label` element is visible to the user. You can write code like this:

```
if (parent.rightFrame.document.all["Label1"].visibility == "visible")
  logRemark("The Label control is visible");
```

After you understand the test structure presented in the techniques in this chapter, your next step is to get a firm grasp of the IE DOM. The better you understand the DOM, the more powerful automation you'll be able to write.

One common situation that can cause trouble is when you need to access text on a Web page/application that is not part of any HTML element other than the body. One way to do this is to use the `document.body.innerText` property. Another way is to use the `createTextRange()` and `findText()` methods:

```
var trange = parent.rightFrame.document.body.createTextRange();

if (trange.findText("foo") == true)
  logRemark("Found 'foo' on the Web page");
else
  logRemark("No 'foo' found");
```

6.5 Manipulating the Value of an HTML Element on the Web AUT

Problem

You want to manipulate an HTML element on the Web AUT to simulate user actions such as typing data into a text field and clicking on buttons.

Design

Use methods and properties of the IE DOM, such as the `checked` property and the `click()` method. You need to take into account the state of the Web AUT.

Solution

For example:

```
function runTest()
{
  try {
    if (parent.loadCount == 1)
    {
      parent.rightFrame.document.theForm.TextBox1.value = "7";
      parent.rightFrame.document.theForm.TextBox2.value = "5";
      parent.rightFrame.document.all["RadioButton1"].checked = true;
      parent.rightFrame.document.theForm.Button1.click();
    }
    else if (parent.loadCount == 2)
    {
      parent.rightFrame.document.all["RadioButton3"].checked = true;
      parent.rightFrame.document.theForm.Button1.click();
    }
    else if (parent.loadCount == 3)
    {
       // determine pass or fail result here
       // save test scenario results here
    }
  }
  catch(e) {
    logRemark("Unexpected fatal error: " + e);
  }
} // runTest ()
```

This code simulates a user typing 7 and 5 into the input fields, checking the `RadioButton1` control (for addition), clicking the `Button1` control (to calculate), and then after the Web application reloads, checking `RadioButton3` for multiplication and clicking the `Button1` control again. On the third page load, you verify the state of the application (see Section 6.4) and save the scenario result (see Sections 6.6 and 6.7).

Comments

To simulate user interaction with a Web AUT, you first need to determine what user action you want to simulate. In the case of test automation, this is usually placing text into an HTML element to simulate typing, selecting an option from a drop-down control, clicking on a button control, or checking a radio button control. Each of these actions has an intuitively named method or property such as `value`, `click()`, and `checked`. There are hundreds of other properties and methods too. For example, you can simulate a user scrolling a scrollbar component using the `scrollTo()` method, you can set the focus of an element using the `focus()` method, and you can highlight text using the `select()` method. The IE DOM gives you virtually full control over the client area of an HTML Web page. The real trick is knowing which method or property to use. This is a combination of art and science because it's not practical to learn the details of all the IE DOM methods and properties. Fortunately, the methods and properties have meaningful names and are well documented.

Notice that you are constructing a test scenario here rather than a test case. The terms are often used interchangeably, but in general the term *test scenario* refers to test automation that changes the SUT through several states. For instance, in the techniques in this chapter, each part of the test code triggers a new HTTP request-response, which creates a new state of the application that is reflected in the UI. *Test case* normally refers to a testing situation/item in which the test automation manipulates the SUT through one (or possibly two) state changes. For example, in API test cases, inputs are sent to the method under test, and a return value is produced. Web application UI testing is usually performed as a test scenario because most bugs are found when transitioning through multiple states of the Web application; single state bugs are usually detected during the development process.

When constructing test scenarios like the one in this section, you can organize your test effort in one of two ways. You can hard-code the scenario input values into the test script and maintain a lot of separate scenario scripts. A second approach is to write just a few scripts, which are then parameterized to read input files, and maintain a lot of scenario input files. In practice, most test efforts primarily use the first approach, even though it has the disadvantage of requiring you to manage a large number of test scripts.

6.6 Saving Test Scenario Results to a Text File on the Client

Problem

You want to save your test scenario pass/fail result to an external text file on the test client machine.

Design

Instantiate a `Scripting.FileSystemObject` ActiveX object and then use the object's `CreateTextFile()` and `WriteLine()` methods.

Solution

```
function saveResults()
{
  var fso = new ActiveXObject("Scripting.FileSystemObject");
  var f = fso.CreateTextFile("C:\\results.txt", true, false);
  f.WriteLine("Description = " + parent.description);
  if (parent.pass == true)
    f.WriteLine("Result = Pass");
  else
    f.WriteLine("Result = FAIL");
  f.Close();
}
```

The JavaScript language by itself does not contain any native file IO routines, so if you want to save results to file, you must use a JavaScript add-on. Microsoft created script-friendly libraries called ActiveX technologies that essentially extend JavaScript functionality. To write to a text file, the first step is to instantiate a `Scripting.FileSystemObject` object. Next, you use that object to create a handle to a file object. The `CreateTextFile()` method accepts one required argument and two optional arguments. The required argument is the name of the text file to create. The first of the two optional arguments is a Boolean flag to indicate whether any existing file with the same name should be overwritten. In this example, you specify true. The second optional argument is a Boolean flag indicating whether to use Unicode encoding. In this example, you set that argument to false, which causes the text file to use the default ASCII encoding.

Comments

When running script-based Web application UI test automation, you'll often want to write test scenario results to external storage. The simplest way to save results is to write the results to a text file on the client machine.

The technique given here assumes the existence of a global variable `pass`. As a general rule, the use of global variables is not recommended because it makes your code harder to read and maintain. In this case, the simplicity gained by using a global variable seems to outweigh the readability and maintainability penalty.

To write to a file from a JavaScript function, you may have to modify IE's security settings. By default, these settings typically do not allow JavaScript to write to the client machine's hard drive. Go to IE's Security settings and modify the Trusted Sites and ActiveX object scripting execution properties. (The exact process to do this varies depending upon your client configuration.)

One alternative to saving your test results to a text file on the test client machine using ActiveX technology is to save results as a `Cookie` object on the client machine. This approach is more troublesome in general than saving results as a text file because cookies are stored in a binary format, so you have to write an auxiliary JavaScript helper program to read the cookie from disk and then parse the results. In general, you should use this approach only when other approaches are not feasible. A second alternative to saving scenario results as a text file on the client is to save the results into a lightweight database. This technique is described in Section 6.7.

Several of the techniques in Chapter 1 show how to time-stamp the file name of a results file and how to create a time-stamped folder to hold results files. You can adapt the techniques

in Chapter 1 to a JavaScript coding environment by using the `Date` object and the `Date.toDateString()` and `Date.toTimeString()` methods.

6.7 Saving Test Scenario Results to a Database Table on the Server

Problem

You want to save your script-based UI test scenario results into a lightweight database on the Web server.

Design

Create an Access database on the Web server. Then post the test results from the client machine via an HTML `Form` element back to the server and execute an ASP/VBScript program on the server to save the results into the Access database.

Solution

First you create an Access database on the Web server. For example, you could create an Access database named `dbResults.mdb` with two columns. The first column is named `scenarioID`, has type `AutoNumber`, and is a primary key. The second column is named `scenarioResult` and is type `Text`. Of course, you may want to add other columns to hold information, such as the date and time of the test run, and so on.

Next, because you need to post the scenario results back to the Web server, you need to place the results text field in the test harness UI into an HTML `Form` element:

```
<form name="theForm" method="Post" action="..\\SaveResults.asp">
  <p>Test Result = <input type="text" name="result" size="12"></p>
  <p><input type="submit" name="sender" value="Save Results"></p>
</form>
```

You give the `Form` element a name and specify which script to run (`SaveResults.asp`) when the form data is posted to the Web server. You also need to edit any lines of code in the test harness that reference the result field to its new name `theForm.result`. For example:

```
theForm.result.value = " Pass ";
```

You then write a script called `SaveResults.asp` and save it on the Web server:

```
<html>
<body>
  <%
    strResult = Request.Form("result")
  %>
  <h4>Save Test Results Page</h4>
  <%
    Response.Write("<p>Scenario result = " & strResult & "<BR>")
    Response.Write("<p>Saving result to Access database dbResults.mdb</p>")
```

```
    Const adOpenStatic = 3
    Const adLockOptimistic = 3
    Set objConnection = CreateObject("ADODB.Connection")
    Set objRecordSet = CreateObject("ADODB.Recordset")
    objConnection.Open _
    "Provider = Microsoft.Jet.OLEDB.4.0; " & _
      "Data Source = C:\Inetpub\wwwroot\TestAuto\Ch6\dbResults.mdb"
    objRecordSet.Open "SELECT * FROM tblResults" , _
    objConnection, adOpenStatic, adLockOptimistic
    objRecordSet.AddNew
    objRecordSet("scenarioResult") = strResult
    objRecordSet.Update

    objRecordSet.Close
    objConnection.Close
    Response.Write("Done")
  %>
  </body>
<html>
```

With this code in place, to manually save results after the test scenario runs, you can click on the Save Results button. This action posts the `Form` element to the Web server and invokes the `SaveResults.asp` script that will retrieve the test scenario result from the posted `Form` object and save it into the `dbResults.mdb` Access database.

Comments

You may want to save your test scenario results on the Web server machine instead of the client machine as described in Section 6.6. One way to do this is to create a lightweight Access database on the Web server and write a script that saves scenario results into the database.

VBScript is used traditionally for server-side scripts, but you can use JavaScript if you prefer. The `SaveResults.asp` script grabs the test scenario result from the `Form` using the `Request.Form()` method. Next, you open a connection to the `dbResults.mdb` database using the `CreateObject()` method, which is part of ADO technology. You can think of ADO (ActiveX Data Objects) as a code library that adds database functionality to JavaScript and VBScript. You can add data to a database using ADO in several ways. The simplest technique, as used in this section, is to create a `RecordSet` object, fill it with the existing data in the database, add a new row to the `RecordSet`, add the test scenario result to the new row, and then insert using the `RecordSet.Update()` method.

If you want to save test results programmatically, you can write a `saveResults()` function that directly submits the `Form` element containing the scenario result:

```
function saveResults()
{
  document.all["theForm"].submit();
}
```

You can also indirectly submit the `Form` element by simulating a user click on the save-results button:

```
function saveResults()
{
  document.all["sender"].click();
}
```

When saving results to the SUT Web server, instead of saving to a database, you can save as a text file. You would use the techniques presented in Section 6.6 by creating a `Scripting.FileSystemObject`. If you use this approach, you must modify the Web server security permissions to allow the virtual user/context under which the saving script executes to have permission to write to the hard drive. Exactly how to do this varies from system to system and can be tricky. Additionally, when creating the results file with the `CreateTextFile()` method, you'll either have to specify the full path to the file or use the `MapPath()` method because the IIS Web Server interprets relative paths incorrectly for use with file operations.

6.8 Example Program: ScriptBasedUITest

This program combines several of the techniques in this chapter to create a lightweight test automation harness to test the ASP.NET Web application shown in the right frame in Figure 6-1. The test automation consists of three files. The first file, `WebForm.aspx`, is presented in the introduction section of this chapter. It is a simple client-server calculator demonstration program. The second file is named `WebAuto.html`. This HTML container houses two frames, one for the Web AUT and one for the test harness code. Following is the code for `WebAuto.html`:

```
<html>
  <head>
    <script language="JavaScript">
      var description = "Demo Test Scenario";
      var whenRun = new Date();
      var loadCount = 0;
      var pass = true;
    </script>
  </head>

  <frameset cols="40%,*">
    <frame src="TestCode.html" name="leftFrame">
    <frame src="../TheWebApp/WebApp.aspx" name="rightFrame"
      onload="leftFrame.updateState();">
  </frameset>
</html>
```

The third file that makes up the test scenario is named `TestCode.html`. This HTML page houses the JavaScript test harness code. The entire page is provided in Listing 6-1. When run, the output will be as shown in Figure 6-1 in the introduction section of this chapter. The code in this section assumes that the automation is organized with a root folder containing two subfolders named `TheWebApp` and `TestAutomation`. The `TheWebApp` folder holds the Web AUT (`WebApp.aspx`). The `TestAutomation` folder contains the main test harness structure as a single Web page (`WebAuto.html`) and the page that houses the JavaScript code which runs the test scenario (`TestCode.html`).

Listing 6-1. *Test Harness File* TestCode.html

```
<html>
  <head>
    <script language="JavaScript">
      function updateState()
      {
        parent.loadCount++;
          if (parent.loadCount > 1)
            runTest();
      } // updateState()

      function runTest()
      {
      try {
        if (parent.loadCount == 1)
        {
          logRemark("Setting TextBoxes to '7' and '5'");
          logRemark("Selecting RadioButton1");
          logRemark("Clicking Calculate button");
          parent.rightFrame.document.theForm.TextBox1.value = "7";
          parent.rightFrame.document.theForm.TextBox2.value = "5";
          parent.rightFrame.document.all["RadioButton1"].checked = true;
          parent.rightFrame.document.theForm.Button1.click();
        }
        else if (parent.loadCount == 2)
        {
          logRemark("Verifying '12.0000'");
          verify("TextBox3", "12.0000");
          logRemark("Selecting RadioButton2");
          logRemark("Clicking Calculate button");
          parent.rightFrame.document.all["RadioButton2"].checked = true;
          parent.rightFrame.document.theForm.Button1.click();
        }
        else if (parent.loadCount == 3)
        {
          logRemark("Verifying '2.0000'");
          verify("TextBox3", "2.0000");
          logRemark("Selecting RadioButton3");
          logRemark("Clicking Calculate button");
          parent.rightFrame.document.all["RadioButton3"].checked = true;
          parent.rightFrame.document.theForm.Button1.click();
        }
```

```
    else if (parent.loadCount == 4)
    {
      logRemark("Verifying '35.0000'");
      verify("TextBox3", "35.0000");
      logRemark("Determining pass / fail");
      if (parent.pass == true)
        theForm.result.value = " Pass ";
      else
        theForm.result.value = " *FAIL* ";
      logRemark("Saving result to 'results.txt'");
      saveResults();
      logRemark("Run at " + parent.whenRun);
    }
  }
  catch(e) {
    logRemark("Unexpected fatal error: " + e);
  }
} // runTest()

function logRemark(comment)
{
  var currComment = document.all["comments"].value;
  var newComment = currComment + "\n" + comment;
  document.all["comments"].value = newComment;
} // logRemark()

function verify(ctrl, val)
{
  if (parent.rightFrame.document.all[ctrl].value != val)
    parent.pass = false;
}

function saveResults()
{
  var fso = new ActiveXObject("Scripting.FileSystemObject");
  var f = fso.CreateTextFile("C:\\results.txt", true, false);
  f.WriteLine("Description = " + parent.description);
  if (parent.pass == true)
    f.WriteLine("Result = Pass");
  else
    f.WriteLine("Result = FAIL");
  f.Close();
  // document.all["sender"].click();
  //document.all["theForm"].submit();
} // saveResults()
```

```
    </script>
  </head>
  <body bgColor="#aaff99">
    <h3 style="font-size: 14; font-family: Verdana">UI Test Script
    </h3>
    <p><input type="button" value="Run UI Test" onclick="runTest();">
    </p>
    <p>Actions:</p><p><textarea id="comments" rows="15" cols="34">
    </textarea></p>

    <form name="theForm" method="Post" action="..\\SaveResults.asp">
      <p>Test Result = <input type="text" name="result" size="12"></p>
      <p><input type="submit" name="sender" value="Save Results"></p>
    </form>
    <p>
  </body>
</html>
```