Choosing a test automation framework

Michael Kelly

November 20, 2003

A test automation framework is a set of assumptions, concepts, and practices that provide support for automated software testing. This article describes and demonstrates five basic frameworks.

Basing an automated testing effort on using only a capture tool such as IBM Rational® Robot to record and play back test cases has its drawbacks. Running complex and powerful tests is time consuming and expensive when using only a capture tool. Because these tests are created ad hoc, their functionality can be difficult to track and reproduce, and they can be costly to maintain.

A better choice for an automated testing team that's just getting started might be to use a test automation framework, defined as a set of assumptions, concepts, and practices that constitute a work platform or support for automated testing. In this article I'll attempt to shed a little light on a handful of the test automation frameworks I'm familiar with -- specifically, test script modularity, test library architecture, keyword-driven/table-driven testing, data-driven testing, and hybrid test automation. I won't evaluate which framework is better or worse but will just offer a description and a demonstration of each and, where appropriate, some tips on how to implement it in the IBM Rational toolset.

The Test Script Modularity Framework

The test script modularity framework requires the creation of small, independent scripts that represent modules, sections, and functions of the application-under-test. These small scripts are then used in a hierarchical fashion to construct larger tests, realizing a particular test case.

Of all the frameworks I'll review, this one should be the simplest to grasp and master. It's a well-known programming strategy to build an abstraction layer in front of a component to hide the component from the rest of the application. This insulates the application from modifications in the component and provides modularity in the application design. The test script modularity framework applies this principle of abstraction or encapsulation in order to improve the maintainability and scalability of automated test suites.

To demonstrate the use of this framework, I'll automate a simple test case for the Windows Calculator program (see Figure 1) to test the basic functions (add, subtract, divide, and multiply).
Figure 1. The Windows Calculator

At the bottom level of the script hierarchy are the individual scripts testing addition, subtraction, multiplication, and division. As examples, the first script that follows tests addition and the second script tests subtraction.

```vba
Sub Main
    Window Set Context, "Caption=Calculator", ""
    '5
    PushButton Click, "ObjectIndex=10"
    '+'
    PushButton Click, "ObjectIndex=20"
    '6
    PushButton Click, "ObjectIndex=14"
    '='
    PushButton Click, "ObjectIndex=21"
    '11
    Result = LabelUP (CompareProperties, "Text=11.", "UP=Object Properties")
End Sub

Sub Main
    Window Set Context, "Caption=Calculator", ""
    '20
    PushButton Click, "ObjectIndex=11"
    PushButton Click, "ObjectIndex=8"
    '-'
    PushButton Click, "ObjectIndex=19"
    '10
    PushButton Click, "ObjectIndex=7"
    PushButton Click, "ObjectIndex=8"
    '='
    PushButton Click, "ObjectIndex=21"
    '10
    Result = LabelUP (CompareProperties, "Text=10.", "UP=Object Properties")
```

Choosing a test automation framework
The two scripts at the next level of the hierarchy would then be used to represent the Standard view and the Scientific view available from the View menu. As the following script for the Standard view illustrates, these scripts would contain calls to the scripts we built previously.

```vbnet
' Test Script Modularity Framework
' Script for Standard View
Sub Main
    ' Test Add Functionality
    CallScript "Test Script Mod Framework - Add"
    ' Test Subtract Functionality
    CallScript "Test Script Mod Framework - Subtract"
    ' Test Divide Functionality
    CallScript "Test Script Mod Framework - Divide"
    ' Test Multiply Functionality
    CallScript "Test Script Mod Framework - Multiply"
End Sub
```

And finally, the topmost script in the hierarchy would be the test case to test the different views of the application.

```vbnet
' Test Script Modularity Framework
' Top level script - represents test case
Sub Main
    ' Test the Standard View
    CallScript "Test Script Mod Framework - Standard"
    ' Test the Scientific View
    CallScript "Test Script Mod Framework - Scientific"
End Sub
```

From this very simple example you can see how this framework yields a high degree of modularization and adds to the overall maintainability of the test suite. If a control gets moved on the Calculator, all you need to change is the bottom-level script that calls that control, not all the test cases that test that control.

### The Test Library Architecture Framework

The test library architecture framework is very similar to the test script modularity framework and offers the same advantages, but it divides the application-under-test into procedures and functions instead of scripts. This framework requires the creation of library files (SQABasic libraries, APIs, DLLs, and such) that represent modules, sections, and functions of the application-under-test. These library files are then called directly from the test case script.
To demonstrate the use of this framework, I'll automate the same test case as above but use an SQABasic library. The library contains a function to perform the operations. Following are the header file (.sbh) and the library source file (.sbl).

'Header File
'Test Library Architecture Framework
"Functions Library
Declare Sub StandardViewFunction BasicLib "Functions Library" (OperandOne As Integer, _
    OperandTwo As Integer, _
    Operation As String)

'Library Source File
'Test Library Architecture Framework
'Functions Library
Sub StandardViewFunction (OperandOne As Integer, _
    OperandTwo As Integer, _
    Operation As String)

    'Click on first operand
    Select Case OperandOne
        Case 0
            PushButton Click, "ObjectIndex=8"
        Case 1
            PushButton Click, "ObjectIndex=7"
        Case 2
            PushButton Click, "ObjectIndex=11"
        Case 3
            PushButton Click, "ObjectIndex=15"
        Case 4
            PushButton Click, "ObjectIndex=6"
        Case 5
            PushButton Click, "ObjectIndex=10"
        Case 6
            PushButton Click, "ObjectIndex=14"
        Case 7
            PushButton Click, "ObjectIndex=5"
        Case 8
            PushButton Click, "ObjectIndex=9"
        Case 9
            PushButton Click, "ObjectIndex=13"
    End Select

    'Click on first operand
    Select Case OperandOne
        Case "+"
            PushButton Click, "ObjectIndex=8"
        Case "-"
            PushButton Click, "ObjectIndex=7"
        Case "*"
            PushButton Click, "ObjectIndex=11"
        Case "/"
            PushButton Click, "ObjectIndex=15"
    End Select

    'Click on first operand
    Select Case OperandOne
        Case 0
            PushButton Click, "ObjectIndex=8"
        Case 0
            PushButton Click, "ObjectIndex=7"
        Case 0
            PushButton Click, "ObjectIndex=11"
        Case 0
Using this library, the following test case script can be made.

```
' Test Library Architecture Framework
' Test Case script
' $Include "Functions Library.sbh"

Sub Main

' Test the Standard View
Window Set Context, "Caption=Calculator", ""

' Test Add Functionalty
StandardViewFunction 3,4,"+"
Result = LabelVP (CompareProperties, "Text=7.", "VP=Add")

' Test Subtract Functionalty
StandardViewFunction 3,2,"-"
Result = LabelVP (CompareProperties, "Text=1.", "VP=Sub")

' Test Multiply Functionalty
StandardViewFunction 4,2,"*"
Result = LabelVP (CompareProperties, "Text=8.", "VP=Mult")

' Test Divide Functionalty
StandardViewFunction 10,5,"/"
Result = LabelVP (CompareProperties, "Text=2.", "VP=Div")

End Sub
```

From this example, you can see that this framework also yields a high degree of modularization and adds to the overall maintainability of the test suite. Just as in test script modularity, if a control gets moved on the Calculator, all you need to change is the library file, and all test cases that call that control are updated.

**The Keyword-Driven or Table-Driven Testing Framework**

*Keyword-driven testing* and *table-driven testing* are interchangeable terms that refer to an application-independent automation framework. This framework requires the development of data tables and keywords, independent of the test automation tool used to execute them and the test script code that "drives" the application-under-test and the data. Keyword-driven tests look very
similar to manual test cases. In a keyword-driven test, the functionality of the application-under-test is documented in a table as well as in step-by-step instructions for each test.

If we were to map out the actions we perform with the mouse when we test our Windows Calculator functions by hand, we could create the following table. The "Window" column contains the name of the application window where we're performing the mouse action (in this case, they all happen to be in the Calculator window). The "Control" column names the type of control the mouse is clicking. The "Action" column lists the action taken with the mouse (or by the tester). And the "Arguments" column names a specific control (1, 2, 3, 5, +, -, and so on).

<table>
<thead>
<tr>
<th>Window</th>
<th>Control</th>
<th>Action</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculator</td>
<td>Menu</td>
<td></td>
<td>View, Standard</td>
</tr>
<tr>
<td>Calculator</td>
<td>Pushbutton</td>
<td>Click</td>
<td>1</td>
</tr>
<tr>
<td>Calculator</td>
<td>Pushbutton</td>
<td>Click</td>
<td>+</td>
</tr>
<tr>
<td>Calculator</td>
<td>Pushbutton</td>
<td>Click</td>
<td>3</td>
</tr>
<tr>
<td>Calculator</td>
<td>Pushbutton</td>
<td>Click</td>
<td>=</td>
</tr>
<tr>
<td>Calculator</td>
<td>Verify Result</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Calculator</td>
<td>Clear</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Calculator</td>
<td>Pushbutton</td>
<td>Click</td>
<td>6</td>
</tr>
<tr>
<td>Calculator</td>
<td>Pushbutton</td>
<td>Click</td>
<td>-</td>
</tr>
<tr>
<td>Calculator</td>
<td>Pushbutton</td>
<td>Click</td>
<td>3</td>
</tr>
<tr>
<td>Calculator</td>
<td>Pushbutton</td>
<td>Click</td>
<td>=</td>
</tr>
<tr>
<td>Calculator</td>
<td>Verify Result</td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

This table represents one complete test; more can be made as needed in order to represent a series of tests. Once you've created your data table(s), you simply write a program or a set of scripts that reads in each step, executes the step based on the keyword contained the Action field, performs error checking, and logs any relevant information. This program or set of scripts would look similar to the pseudocode below:

**Main Script / Program**
- Connect to data tables.
- Read in row and parse out values.
- Pass values to appropriate functions.
- Close connection to data tables.

**Menu Module**
- Set focus to window.
- Select the menu pad option.
- Return.

**Pushbutton Module**
- Set focus to window.
- Push the button based on argument.
- Return.

**Verify Result Module**
- Set focus to window.
- Get contents from label.
- Compare contents with argument value.
- Log results.
- Return.
From this example you can see that this framework requires very little code to generate many test cases. The data tables are used to generate the individual test cases while the same code is reused. The IBM Rational toolset can be extended using interactive file reads, queries, or datapools, or you can use other tools (freeware, other development tools, and such) along with IBM Rational tools in order to build this type of framework.

The Data-Driven Testing Framework

Data-driven testing is a framework where test input and output values are read from data files (datapools, ODBC sources, cvs files, Excel files, DAO objects, ADO objects, and such) and are loaded into variables in captured or manually coded scripts. In this framework, variables are used for both input values and output verification values. Navigation through the program, reading of the data files, and logging of test status and information are all coded in the test script.

This is similar to table-driven testing in that the test case is contained in the data file and not in the script; the script is just a "driver," or delivery mechanism, for the data. Unlike in table-driven testing, though, the navigation data isn't contained in the table structure. In data-driven testing, only test data is contained in the data files.

The IBM Rational toolset has native data-driven functionality when using the SQABasic language and the IBM Rational datapool structures. To demonstrate the use of this framework, we'll test the order form from the test sample application Classics A (see Figure 2).

Figure 2. Order form from the sample application Classics A

If we record data entry into this window, we get the following:

'Data Driven Framework
'Test Case Script
We can use datapools to set up test cases that test valid and invalid credit card numbers and expiration dates. The datapool shown in Figure 3, for example, would be for a test case that would test the date field.

**Figure 3. Sample datapool for a test case that would test the date field**

If we modify the script to accept this data, we get the following:

```vbscript
Sub Main
    'Make An Order
    Window Set Context, "Name=frmOrder", ""
    'Card Number
    EditBox Click, "Name=txtCreditCard", "Coords=16,9"
    InputKeys "3334445556666"
    'Expiration Date
    EditBox Click, "Name=txtExpirationDate", "Coords=6,7"
    InputKeys "3334445556666"
    'Place Order
    PushButton Click, "Name=cmdOrder"
    'Confirmation Screen
    Window SetContext, "Name=frmConfirm", ""
    PushButton Click, "Name=cmdOK"
End Sub
```

We can use datapools to set up test cases that test valid and invalid credit card numbers and expiration dates. The datapool shown in Figure 3, for example, would be for a test case that would test the date field.

**Figure 3. Sample datapool for a test case that would test the date field**

If we modify the script to accept this data, we get the following:

```vbscript
'Sub Main
    'Data Driven Framework
    'Test Case Script
    '$Include "SQAUTIL.SBH"

    Dim Result As Integer
    Dim DatapoolHandle As Long
    Dim DatapoolReturnValue As Variant

    'Open the datapool
    DatapoolHandle = SQADatapoolOpen("OrderFormDP")
    '...Add error checking....

    'Loop through the datapool
    While SQADatapoolFetch(DatapoolHandle) = dqaDpSuccess
        'Open Order Form
        Window SetContext, "Name=frmMain", ""
        PushButton Click, "Name=cmdOrder"
        Window SetContext, "Name=frmOrder", ""

        'Card Number
        Dim CardNumber As String
        CardNumber = SQADatapoolValue(DatapoolHanle, "Credit Card Number", DatapoolReturnValue)
        '...Add error checking....
        EditBox Click, "Name=txtCreditCard", "Coords=16,9"

        'Expiration Date
        Dim ExpirationDate As String
        ExpirationDate = SQADatapoolValue(DatapoolHanle, "Expiration Date", DatapoolReturnValue)
        '...Add error checking....
        EditBox Click, "Name=txtExpirationDate", "Coords=6,7"

        'Place Order
        PushButton Click, "Name=cmdOrder"

        'Confirmation Screen
        Window SetContext, "Name=frmConfirm", ""
        PushButton Click, "Name=cmdOK"
```

Choosing a test automation framework
I had to add SQABasic commands to manipulate the datapools. I also added a **while** loop to allow for the processing of each row in the datapool. I should also mention the use of the SQABasic command **UCase** within the **If Then** statement. **UCase** is used to make the argument (in this case, the datapool return value) all uppercase. This way the comparisons aren't case sensitive, making the code more robust.

This framework tends to reduce the overall number of scripts you need in order to implement all of your test cases, and it offers the greatest flexibility when it comes to developing workarounds for bugs and performing maintenance. Much like table-driven testing, data-driven testing requires very little code to generate many test cases. This framework is very easy to implement using the IBM Rational toolset, and there’s a lot of detailed documentation available with how-tos and examples.

**The Hybrid Test Automation Framework**

The most commonly implemented framework is a combination of all of the above techniques, pulling from their strengths and trying to mitigate their weaknesses. This hybrid test automation framework is what most frameworks evolve into over time and multiple projects. Figure 4 gives you an idea of how you could combine the approaches of the different frameworks within the IBM Rational toolset.
Roundup

I've described five test automation frameworks that an automated testing team might consider using instead of relying only on a capture tool. You can use just one or a combination of the frameworks. You can implement modularity by nesting test scripts and using the SQABasic library files to implement functions and procedures. You can use datapools to implement whichever data-driven technique you choose, or you can extend Robot to work with other types of data stores. The trick is to use the best framework(s) for the job, and the only way to figure that out is to jump in and start using them.
Related topics

- "Improving the Maintainability of Automated Test Suites" by Cem Kaner (paper presented at Quality Week 97)
- "Test Automation Frameworks" by Carl Nagle
- "Using Cost-Benefit Analysis to Compare Different Test Structures for Rational Robot" by Mike Kelly
- "Totally Data-Driven Automated Testing" by Keith Zambelich (Automated Testing Specialists Web site)

© Copyright IBM Corporation 2003
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)