



Automation By Design : Refining the Infrastructure

by Jamie Mitchell

This column was originally published in the Journal of Software Testing Professionals, Volume 2, Issue 1.

In our last column, we talked about the need for designing our automation scripts in such a way that they would be effective, maintainable and affordable. There are several different automating methodologies available to the automator; the one we have chosen to illustrate this time is called functional decomposition.

In this column, we will discuss how to verify that the functions we write have completed their task and handle both expected and unexpected error conditions. So much of the success of automation depends on responding correctly to error conditions. We will also talk about how to design our function to be expandable, allowing us to continuously update it without worrying about breaking existing functionality. Change happens! We must be ready when it does.

Previously, we had designed a function to illustrate some of the techniques and thought processes that go into the design of a function. The example used was that of opening up a Word™ file. The code (as we left it) is below, written in Mercury™ WinRunner; the concepts are applicable to all major automation tools.

```
OpenWordFile(str FileName) {  
  
    # Direct the input to the MS Word application  
    set_window ("MSWord Main Window");  
  
    # Type in hot keys to open the Open File dialog  
    obj_type ("MSWord Main Edit", "<kAlt L-f>o");  
  
    # Connect up to Open File Dialog  
    set_window ("MSWord Open Dlg", 15);  
  
    # Type in the file name and press Open button  
    obj_type ("MSWord Open Dlg FileName Edit", FileName & "<Return>");  
}
```

In addition, we started to talk about the verification step for this kind of function. When a human tester performs a task, the verification step is reasonably trivial, “did we get the response we expected to get? Yes or No?” The tester looks at the screen and makes a determination. If any problem occurs, the tester can see that *something else* happened. What happened? The manual tester can describe it easily based on the state of the system. Thousands of different outcomes may be possible, and the tester’s brain usually can deal with all of them. Unfortunately, automation is not that straight forward.



Remember that we stated as a basic rule of infrastructure programming that a function should be seen as an implicit contract between the user (scripter) and the author (automator.) This contract states that the function will perform the action it is written to do successfully, or it will notify the caller (and the log) of the exact failure. *The function, as written above, does not fulfill the contract, as it returns no indication to the caller as to success or failure.* We then introduced some pseudo code to be added to ensure the contract is fulfilled. Let's walk through this code to see how it works.

```
1.   While (TRUE)
2.       If (successfully completed) Then
3.           Return success
4.           Exit loop

5.       Else If (x) Then
6.           Do something

7.       Else if (Error Dialog pops) Then
8.           Log the error
9.           Exit Loop

10.      Else if (Warning dialog pops) Then
11.          Accept the warning
12.          Continue

13.      Else if (have we waited long enough) Then
14.          Document machine state
15.          Return Time out failure
16.          Exit

17.      Else
18.          Wait a bit
19.          Increment the counter
20.      EndIf
21.  Wend
```

An important point to notice about this code is that this is a loop that executes only a finite number of times. On line 13, we will test to see if we have looped through the maximum number of times (as defined externally to the function) without getting to the expected state. If we have, then we will determine that the function has failed and return that failure. This is important enough to codify a rule:

No infinite loops must be allowed, as we must be able to run unattended!

Since the unexpected may happen after triggering an event in the application under test (AUT), we must always ensure that we do not wait forever for a known outcome. We



will test for all known outcomes, but will always allow for an outcome that we did not expect. The absence of a known outcome after waiting the maximum allowed time will be treated as a failure for this test, but also as a learning experience.

As stated above, in manual testing, this step is not explicitly codified. The manual tester performing this step will notice that an unexpected outcome has occurred, and handle it without prior planning. If the outcome was unexpected, the tester will stop the test, write an error report, and go on to the next testcase. But, in automation, we do not have that luxury. Assume that we may expect one of thirteen outcomes when opening a file, but the machine performs a fourteenth. We must capture the event – somehow – and log it so a person can later investigate what happened.

This style of coding, which loops looking for expected outcomes, but still handles the unexpected, we call a **Learning Architecture**. The loop itself we call a **Single-Action While Loop (SAWL)**.

At its most simple, the single action portion of the SAWL is a simple syntactical construct which repeats itself as follows:

```
If (Condition) Then
    Statement
Else If (Condition) Then
    Statement
Else If (Condition) Then
    Statement
...
Else
    Statement
End If
```

Take a look at the pseudo-code above. The key to understanding this is that each condition is evaluated, from the top to the bottom, sequentially until one is found to be TRUE. The statement that is associated with the TRUE condition is then run, and the execution thread moves beyond the construct. If none of the conditions evaluate to TRUE, then the final statement after the “else” is executed. Compare this with the verification code above and see that they are essentially alike.

So what is a condition to be evaluated? What are the possibilities that can happen when we try to open a file? The file could open. For a positive test, where we are not expecting an error, we would consider this the desirable end result and would therefore end the function by returning a SUCCESS code. How do we tell that the file opened? We might notice that the main window has become enabled again coupled with a change of title bar to include the name of the opened file. The question we ask is, “how does the manual tester know that the file opened successfully?” There are almost always visual



cues shown by the application to signify the completion of an event: sometimes just a line in the status bar (OK, or Done.)

If the file has not opened yet on our trip through the SAWL, we may have had an undesirable event occur. We could have gotten any one of several error messages in the standard error dialog, or any one of several warning messages in a standard warning dialog box. A condition to check for, then, is a dialog box popping up.

If an error message were to pop (detected by the code in line 7), we are done trying to open the file. After all, if the scripter wants to open a file, the only condition that could be seen as a success is the file opening. Therefore, the error signifies that the open is not going to happen; we need to identify the error and return the failure so we can move on.

The message in the error box will consist of a string (or possibly several) explaining what the application found and why the file is not going to open. This is of obvious interest to the tester who expected the file to open. We want to extract the string(s) from the dialog and place it directly into the log for later review. Each one of the major tools has a way to get the string; in WinRunner, we use the following lines to pick it up into a variable:

```
set_window(Win, 10);      # Win is the error dialog logical name  
static_get_text(Ctrl, Text); # Ctrl is the logical name of the static control
```

The variable **Text** now contains the string explaining the error. This can be directly written to the log, and a FAILED return code may be returned to the caller of the function.

Suppose, for a moment, that we wanted to run a negative test. Imagine that we wanted to check the failure handling of our AUT by having it open a known bad file and we want to check to make sure we get the correct error message. We could write another function; however, that is rather inefficient.

Let's step back and think about this for a moment. There are four separate cases we must handle.

Case 1: If we are running a positive test case, then we expect no error messages of any sort. The existence of an error message anywhere in the script would signify a failure. No error messages allowed, and the first one to pop kills the script and we mark it failed.

Case 2: If we are running a negative test case, then we expect an error message somewhere through the execution path. We may be testing a business rule or some kind of error handling feature in our application; when we trigger the event that we know will fail, we want to ensure we get back the correct error message. We know what we expect the error message to say; we just want to verify it. When the error condition occurs, we pull out the message and compare it with a global variable string containing the expected



message. If they are the same, we return a SUCCESS to the caller and end the function. It is, of course, up to the scripter to write the script such that the script only executes up to the function where the failure is expected.

Case 3: We are running a negative test and we get an error message that does not match the one we are expecting. In other words, we are running a negative test and we have a [different] failure. We compare the actual failure message with the expected string in our global variable – and they do not match. At this point we must log that we were expecting the error in the global variable, but actually got a different error and return a failure to the caller.

Case 4: We are running a negative test but the failure does not occur. We get to the end of the test and no error message has popped. At this point, we must log the absence of the error and fail the test.

In each of the four cases above, we can add negative testing to the function we have already wrote by adding a single global variable containing a string; if the test case is positive, then the variable is NULL. If it is a negative test case, then the variable contains the expected string. Each function that we write will have a condition to check in case an error dialog pops and a couple lines of code to log the erroneous conditions.

Back to the SAWL. If we get a warning dialog, we will pull out the message in the same way as we recovered the error message. However, in the case of a warning, we may need to make a decision. What is the nature of the warning? Is it a nag message, put in the application to test the nerves of the user: *how many worthless warnings can we make you handle before you go and buy a competitor's product?* Or, is it an important warning indicates not all is right with the state of the application. It may say that the file version is from a previous release or the file is not a Word document. These we may want to log as a warning and then go on to see if the rest of the test case works. Or we may want to end right here and return an error. Each case may be handled differently and coded to do the right thing. Notice that, if we decide to go on after clearing a warning, all we have to do is get rid of the message box (i.e. click the OK button.)

After the dialog goes away, we loop back up to the top for the next iteration of the SAWL. If the file now opens, the SAWL will act as before, determining that the file did open, and the function can return the SUCCESS.

I called this a learning architecture. Here's why. What happens if an unknown condition occurs? We loop through the SAWL the maximum number of times, but we never kick out of the loop because we do not get a known dialog nor does the file open. At this point, we are going to return a failure for the function. Notice in the pseudo code, at line 14, that we are going to document the state of the machine before returning the failure. I tend to do this by taking a snapshot of the entire screen and placing it in the log. This



Innovative solutions to maximize your testing investment.

will allow the scripter to see exactly what the screen looked like at the point of failure. If an unknown window had popped, it will be in the snapshot.

The automator can now add another condition to the SAWL (shown in Lines 5,6) that can handle the previously unknown condition. By looking at the code, it should be clear that the addition of the new lines in no way changes the execution of the other lines. This is essential, as we do not want to introduce any new bugs to our code by changing that code which works.

We have walked through the verification steps which will allow our function to be much more robust than anything we can script ad hoc.

Certainly, the first time you try to script a function like this, it takes a while. After writing a few, however, you will find it gets easier to the point where it will become second nature. Then you will be on your way to becoming an automator.

Next issue we will discuss how we decide which functionality we are going to decompose into functions, and lay out some templates to simplify our coding. We will also discuss what to do in our script when we have a failure in a function.