



Automation By Design: Designing Infrastructure Functions

by Jamie Mitchell

This column was originally published in the Journal of Software Testing Professionals, Volume 1, Issue3.

In the last issue, I put together an organizational blooper film: exposing a hypothetical automation attempt at a fictional company in which every step taken destined the project to failure. In reality, the example was a composite of many failed automation projects that I have seen or been party to.

This issue, we will start discussing how to create a test automation *infrastructure* which will allow us to successfully automate some of our testing. An *infrastructure* (sometimes called a *Framework*) is a set of functions that can be called by your automated test scripts to do various tasks during a test run. These functions replace recorded code to make your scripts more robust and have more functionality. My previous column discussed some of the problems of maintaining and updating recorded test scripts because they are hard coded. See the June issue of the JSTP for more information.

The question to be asked, then, is how do we avoid hard-coding our test scripts? One answer comes from the engineering discipline of software development that had to solve the exact same problem several decades ago.

Functional decomposition is the technical name given to the process of determining which execution behaviors are likely to be used repeatedly and coding them as separate, callable functions. For test automation, a *behavior* is simply a series of interactions with the application under test (AUT) that is done repeatedly. *Aggregate bundling* is a fancy way of saying that we are putting several different but related steps together into a single function; this function can then be used abstractly as a single action. Our infrastructure will consist of many functions that are built using these two concepts. Following is an example of the design considerations that go into one of these infrastructure functions.

Consider the abstract idea of opening a file in an application, in this case, since many people have worked with it, MS Word™. In the manual test case which we are automating, this step would likely show up as:

Open file C:\Dir1\Dir2\XXX.Doc



The author of the manual testcase assumes that the tester would know how to open this file. An automated test script which was recorded would have a number of steps that execute this behavior.

Let us imagine that your task is to test Microsoft Word™. During the process of planning your testing¹, you realize that you will often be opening many document files into Word. My first principle of Functional Decomposition states:

If you perform an action more than a few times, create a function to do it and then call that function each time you perform the action.

Opening a Word file is a good candidate for such a function.

Each of the commercially available automation tools allows you to create such a function; giving it a descriptive name so that, when you see it called in the code, the function documents what it does. For this example, we will name it:

OpenWordFile(str FileName)

The basic syntax of the function may change with each tool, but it is safe to say that the function declaration will look something like the above. Note that a parameter (string) is passed into the function: this specifies the path and name of the file to be opened. When the function is called, it will appear as follows:

OpenWordFile("C:\Dir1\Dir2\XXX.Doc");

WinRunner will execute the function we write and the file will be opened.

To design this function, consider how a file is opened in Word.

- Move mouse to file menu item
- Press left button
- Drag mouse to "**Open ...**" submenu
- Release left button
- Set focus to the Open File dialog when it pops
- Click on "look in" combo box
- Click on list view to get to desired path
- Click on file view to select file
- Double click on file to select
- Wait to ensure file opens

¹ You do, of course, plan your testing before starting it?!



Not a bad start; consider, however, how the list view and file view controls work. When using these type controls, the user needs to make decisions based on visual feedback (i.e. double click on this directory, then that one, move here, etc.) Is there an easier alternative to the list and tree views of the open file dialog? How about if we just type the file name and path into the “File Name” edit box? That will give us exactly the same behavior, and is much easier to write. This illustrates my second rule:

When automating a behavior, use the simplest steps possible (unless the test case explicitly calls for a specific way.)

When you think about this particular example, there is another rule we can extrapolate. This is a bit more subtle and it may take a developer to see that it applies here. This rule states:

Test the functionality which you are being paid to test; unless the testcase specifically calls for it, do not test operating system code or other code not part of the AUT.

The file open dialog is supplied by the operating system as a common dialog. While it may appear to be good methodology to thoroughly test the Open File dialog, understand that the AUT calls a Windows Application Programming Interface (API) to put up the dialog. The implication here is that Microsoft has already tested the dialog; any extra time spent testing it by you is not a good use of resources. You should limit yourself to ensuring the file opens correctly.

So, let’s revise the steps we are going to put in our function:

- Move mouse to file menu item
- Press left button
- Drag mouse to “**Open ...**” submenu
- Release left button
- Set focus to the Open File dialog when it pops
- Type the “FileName” argument into the File Name edit box
- Click on OK button
- Wait to ensure file opens

These steps can be placed into the function body using the record method of the tool. At this point, you likely have something that looks like the following (syntax will be different in each tool.)²

OpenWordFile(str FileName) {

² This recording was made with Mercury-Interactive WinRunner™ 6.0.



Innovative solutions to maximize your testing investment.

```
set_window ("Functional Decomposition.doc – Microsoft Word");      (1)
obj_mouse_click ("Menu Bar", 23, 13, LEFT);                      (2)
win_mouse_drag ("File", 34, 37, 35, 37);                        (3)
set_window ("Open 1", 3);                                       (4)
obj_type ("RichEdit20W", "c:\\autoexec.bat");                  (5)
win_mouse_drag ("Open 1", 635, 394, 635, 393);                (6)
}
```

Not a pretty sight! So let's see if we can pretty it up some. Remember, we want all of our code to be self-documenting if possible. That means that if I look at the code a year from now, or if a junior automator looks at it, it should be completely understandable. So we are going to massage it one line at a time and add comments where needed.

Line 1: `set_window ("Functional Decomposition.doc – Microsoft Word");`

This is a WinRunner function telling Windows which window is going to be getting input. Each automation tool has an equivalent function call. The main problem with this line as it stands is that it is specific to one particular document. We want to ensure that the function will work with any file we might want to open. We need to define the window with a regular expression³ for the caption since that will always be different. In some tools, the string in this function is a logical name which defines a specific window in the object repository or GUI map. In this case, we want the logical name to be self-explanatory. Therefore, we will change the logical name to one better understood.

Line 1[r]: `set_window ("MSWord Main Window");`

Moving on ...

Line 2,3: `obj_mouse_click ("Menu Bar", 23, 13, LEFT);`
 `win_mouse_drag ("File", 34, 37, 35, 37);`

These lines are what was recorded when I clicked on the "File | Open ..." menu item. For many applications we could be testing, WinRunner would have recorded a simple, one line function call such as:

`menu_select_item("File;Open...");`

Line 2 indicates that the menu bar in Word is not an actual, standard menu object as many applications use; instead it is surfaced by a COM object and, as such, is not seen by the tool as a real menu. It is only seen as a generic object. This can be problematic, as the numbers following the object name, "Menu Bar", are indicating a low-level mouse

³ A Regular Expression is a string which uses specific symbols to allow a search on strings like it (i.e. like using the "*" in a DOS file search.) Your tool will have a fuller explanation and the necessary syntax.



click was all that the tool understood. If the tool bar is moved or resized, the menu that is clicked will likely be wrong.

This is not a major problem; we can use the alternative of typing in the short-cut keys to do exactly the same thing as clicking on the menu. That allows a single line to do the work of both:

Line 2[r]: `obj_type ("MSWord Main Edit","<kAlt L-f>o");`

Note that the keystrokes are sent to the main edit window of Word. They consist of an (Alt+F)⁴ followed by a small (o). This follows the second rule of Functional Decomposition listed above.

Line 4: `set_window ("Open 1", 3);`

Once again we are directing the tool to send the next key and mouse operations to a specific window (in this case, the Open File dialog box); we want to change the logical name of that window to something meaningful. In addition, we want to extend the time allowable for the window to pop up (the second argument in the function call) to a larger number. If we are running from the network, or have a lot of stuff in memory, the box may take a while to pop. The new line will allow 15 seconds for the action to complete, and will look like this:

Line 4[r]: `set_window ("MSWord Open Dlg", 15);`

The next operation is to type in the actual path and file name (FileName) to the edit box in the dialog. Once again we want to use a meaningful logical object name to facilitate understanding.

Line 5: `obj_type ("RichEdit20W","c:\\autoexec.bat");`

Line 5[r]: `obj_type ("MSWord Open Dlg FileName Edit", FileName);`

Finally, the last line is the mouse click on the Open button.

`win_mouse_drag ("Open 1", 635, 394, 635, 393);` (6)

In this case, the tool cannot explicitly see the Open button, (it is surfaced by the COM object(s) painting the rest of the dialog.) When actually using the Open File dialog manually, there is another way to accept the input and open the file. Pressing the Enter key after entering the file name does this. This will be our simplifying decision – once again choosing an easy alternative to make the automation more robust. This eliminates line 6 altogether.

⁴ Pressing the “Alt” key at the same time as pressing the “F” key.



Line 5[r]: `obj_type("MSWord Open Dlg FileName Edit",FileName & "<Return>");`

At this point, since the function may still appear a bit obtuse to anyone who is not a programmer, it is a good idea to add some comments to the function, including a header to explain what the function does.

```
#-----  
# Function OpenWordFile  Open a file in MS Word 2000.  Uses hot  
# keys to access the Open dialog.  Types the name into the edit  
# box and types and Enter key to implicitly press the Open button.  
# At this point, there is no exception or error handling.  
#  
# Input:      [str] FileName: Path and name of the file to be opened  
# Output:     None at this time  
#  
# Requires:   None at this time  
# Side Effects:  None at this time  
#  
# Created:    07/03/2000  Jamie Mitchell  
# Changed:    MM/DD/YYYY  
#-----  
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>");  
}
```

To use this function in a script, record the script as before, delete all of the lines having to do with the file open and add the function call with the actual file as the argument passed in.

```
OpenWordFile("C:\\Autoexec.bat");
```

Well there you have it. The real value of this function will be evident when the next version of MS Word is released and the steps required to open a file radically change. Then, all of the scripts that open a file will be broken; however, fixing the code in just one spot (i.e. this function) will fix all of the broken scripts.

Next issue, we will add some error handling and add some negative testing capability to this function.