



Automation By Design: Engineering the Infrastructure

by Jamie Mitchell

This column was originally published in the Journal of Software Testing Professionals, Volume 1, Issue 4.

In the last issue, we discussed designing test automation that can be maintained more cost effectively than recording-only scripts. The software engineering methodology we are going to adopt is called *functional decomposition*. We will use the test tool language directly to build an infrastructure containing sub-routines that directly execute tasks in the application(s) we are testing. The functions we build will be analogous to building blocks or Legos™; the scripts we execute will be largely made up of sets of Legos™ combined in different ways. Although the examples are written in Mercury Interactive WinRunner™, this approach works with any of the major test tools.

By creating and programming with this infrastructure, we can quickly react to the inevitable changes in our application, environment or platform. When changes that prevent the successful execution of test cases do occur, we can directly revise the infrastructure functions rather than the test scripts. This maintenance indirection will increase the scalability of our effort; whether we have twenty scripts or five hundred scripts, the maintenance is the same (well, not completely, but good enough for now.)

Before getting into the technical side, let's spend a moment on the political side. There is only one reason to automate our testing: we expect a return on the investment (ROI.) Management will fund resources (tools and person-hours) for the automation effort only if they can earn an ROI. Organizations never do something for nothing; and although the proximate reason may not appear to be about money, it is. Need higher quality, less time to market? Money. Need to automate to prove due diligence for the lawyers? Money. Life or property endangering software where the quality is essential? Money (lawsuits are expensive!)

So how does this affect you, the Automator?

Considering the investment made in tools and time, we have to succeed at test automation; failure is not a viable option! We have established that record / playback is not going to work for us over the long term. We need to invest some time and effort up front to build our infrastructure and plan our test suites; this means that we must delay the return we hope to eventually realize on our investment. But, management has been promised immediate, short term results – if not by us, then by the tool vendor. ***“Buy the tool; get immediate return on the investment! Automatically get higher quality and quicker time to market? What the heck! Buy two.”***



Cem Kaner addresses this very eloquently:

... Don't forget to clear up the fantasies that have been spoonfed to your management.

... some vendor's salespeople will make fantastic claims that only a senior executive could believe. You have to *reset expectations* [emphasis added by me] or you will fail even if you do superb technical work.¹

Setting and resetting expectations is a political task we ignore at our own peril.

If we were building a car, and we built the best family sedan on the market, it would do us no good if the management team is expecting a sporty two-seater with 400 horses under the hood. It is my experience that, too often, we over-promise the capability of automation in an attempt to get buy-in from those who set the budgets and pay the bills. Of course, the vendors are more than happy to embellish our embellishments to the point that management is ready to enter our Chevy in the Tallahassee 400.

We must convince management that automation is **not** a short-term investment, **not** the silver bullet to fix all development shortcomings. If we allow automation to be targeted as a short term fix to our testing problems, we will fail. We must treat automation as a long-term project, where the return is dependent on good engineering and planning, on good processes and good people. ***You have been warned!***

Now to the technical side.

In the last issue, we designed our first infrastructure function to open a file in Word. The code we wrote is repeated below.

```
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>");  
}
```

¹ Cem Kaner, [Avoiding Shelfware: A Managers' View of Automated GUI Testing](#). 1998. Available at kaner@kaner.com. If you haven't read this yet, download it now. It is perhaps the best treatment of automation I have ever seen.



This is not bad for a first effort. However, this function has a serious problem. After pressing the OK button (implicitly, done by pressing the Return key after typing in the name of the file), the function ends and the script moves on to execute the next line of code. What happened with the file open? We don't know. If we are lucky, the file opened correctly. However, what happens if the file does not exist? It might not be a Word file. It might be on a drive not currently mapped. In fact, I have identified thirteen different outcomes of the OK button press. The scripter who uses this function is in serious trouble – they must write more lines of code to handle possible outcomes than the function contains; and they have to do it in every script.

Consider the name of the function, **OpenWordFile**. The person who uses this function assumes that after the function has executed, the desired file will be open. The belief that a function performs a task correctly is essential to successful automation. We must put in place a rule such that this belief is enforced. That is:

A function call must be seen as a contract between the Automator who writes it and the scripter who uses it. When the function is invoked, the Automator must guarantee that it will only return a success code if the task has been completed successfully. If the task cannot be completed, the function must return an explicit error code, or not return at all.²

It turns out that almost every function we write will follow the same pattern; we can generalize the function call into a template as shown below.

- **Ensure we are in the correct starting position in the application (in correct state.)**
- **Manipulate the controls required to perform the task.**
- **Trigger the task (press the OK button, click the Next, etc.)**
- **Cycle until the task is complete: successfully or not.**

Note that almost every tester performing a manual test case follows this algorithm. They may not consciously think about it, but if you look closely enough, their actions almost always boil down to these 4 steps. Their eyes, looking at the screen, evaluate the current state of the application to make sure they are in the correct place to perform a task. If they are not, they manipulate the application until it enters the correct state. Then they mouse click and type to various controls to enter the correct instructions and data necessary to perform the task. Then, using a key or mouse click, they trigger the action. The application begins processing, using existing data, entered data, programmed logic and entered instructions. Eventually, the application finishes computing and reaches a

² Some tools allow an Error to be thrown inside of a function such that the cleanup code is executed directly without ever going back to the script



steady state. At this point, the tester evaluates the resultant state and decides if it is correct or incorrect.

Obviously, with an experienced tester, this set of steps occurs faster than I can describe it. Every task in the test case repeats the same steps. Our job as automators is to simulate these actions in software (the test script).

So how does our function fulfill the steps in this template?

The first bullet is completed by the “set_window()” function call. The function will wait until such time as the main Word window both exists and is enabled; this ensures we are in the right starting position. Now we will use the controls required, starting with a menu click (in this case by using the accelerator keys) and filling out an edit control in the file open dialog. Note that we have another “set_window()” to verify the file open dialog has appeared correctly; this is perfectly valid. Next we trigger the action (by pressing the Return key.)

What we are missing is the final step. We need to write some code to verify that the file has opened; alternatively, if the file does not open correctly, we need to evaluate what did happen. If an error occurred, we want to notify the user (via the log) that an expected action (opening the file) failed. If an informational message pops which does not prevent the file from opening - perhaps to inform us that the file is a back level file and will be updated to the new version - then perhaps we want to clear the message and wait again for the file to open.

The essential point to understand is that we have an implied contract with the scripter who uses this function: They will not get control again until such time as the file is open and we return a success code, or until an error is logged and an error code is returned. Therefore, the final bullet in our algorithm actually contains a loop which waits for completion. The following pseudo code presents the algorithm to evaluate the resultant state of any action which we have triggered.

While (NOT DONE)

If (the task has completed successfully) Then

Return_code = SUCCESS

Exit

Else If (Known error A occurs) Then

Write Message to log about this error

Return_code = FAILED

Exit

Else If (Known error B occurs) Then

Write Message to log about this error

Return_code = FAILED



Innovative solutions to maximize your testing investment.

```
Exit
Else If (Known error C occurs) Then
  Write Message to log about this error
  Rreturn_code = FAILED
  Exit
Else If (Known info message Q occurs) Then
  Clear info message Q dialog
Else
  If (we haven't waited too long) Then
    Wait a little bit more
  Else
    Write Message that we timed out waiting
    Return_code = FAILED
    Exit
  EndIf
EndIf
Wend
```

To many of the readers of this column, this code may appear daunting or even scary. **Do not fear!** In the next issue, we will parse out this code and show that, not only is it **not** difficult to understand, but it is the key to successful infrastructure building.