



## Automation By Design: Applying the Reality Model to Automation

*by Jamie Mitchell*

*This column was originally published in the Journal of Software Testing Professionals, Volume 2, Issue3.*

In my last column (June 2001), I advanced the hypothesis that one of the main reasons the Record / Playback (R/P) methodology of test automation usually fails to achieve set testing goals is that the paradigm does not sufficiently model real world testing. I also laid out several behavioral models including the manual testing model (how do manual testers actually plan and run tests) and the event-driven architecture model (how do we actually interact with a GUI-based operating system.)

The manual testing model has two unique characteristics the R/P model is almost completely lacking; **context** and **reasonableness**, the manual tester's contributions to the test run. The manual test case<sup>1</sup> is often left somewhat vague as to the implementation details (more on this later.) The tester reads an instruction from the manual test case consisting of a task to be performed, data to use for the task, and the expected reaction of the system. The tester supplies domain knowledge suitable for the requested task. Based on that knowledge, the tester performs the task, interprets the actual result and decides whether it matches the expected result.

In this column, I would like to present an automation model which does match the manual testing model and supply a template which can be directly translated into successful automation code. In this model we will leverage the peculiarities of the event-driven architecture model which all GUI operating systems(OS) exhibit. That is, the application (through the OS) waits for the user to specify what is to be done, executes the specified task, and then waits again. Much like the robot, Number 5, in the movie Short Circuit, the operating system spends much of its time waiting for "more input."

A manual test case rarely specifies any details about the interface used in completing the task. For example, if we are testing a word processor package, the step is much more likely to read, "Open a File" than "Move the mouse to the file menu, click on the left mouse button, etc." Why is that?

---

<sup>1</sup> For definition purposes: A manual test case is a set of statements used to define actions a manual tester will take against the system under test. An automated test case is the [conceptually] same set of statements which are put into executable form. I will often use the term script interchangeably with automated test case. It needs to be noted that, in the case of table-driven testing, there may actually be no physical script for an automated test case.



On its own, the task “Open a File” is an abstract idea. No specifics are given. An assumption is made that the tester running the test will understand the task and know what discrete actions must be performed to complete the task. By abstracting the task, we ensure that when the interface changes, the test case will still be valid and executable *as long as the tester knows how to perform the task within the new context*. Think about that. In our theoretical word processor system under test (SUT), in every version since 1.0 we have had to open a file. The specific physical steps have changed from version to version. Certainly the actions performed by the OS in executing the task have changed. However, the logical, abstract idea of opening a file has not changed, nor have the test case instructions as long as the abstract ideas in the test case are still valid.

This is not by accident. A software package tends to allow the same abstract tasks to be performed from version to version. It is not as if the essential tasks that our word processor performs become database tasks when we upgrade to version 3. We buy software with the expectation that it will help us perform certain tasks. If the new version did not perform those tasks, we would probably not purchase the upgrade.

Our manual test cases are more valuable due to their abstract nature; they have very little direct correspondence with the physical GUI. Consider, however, an R/P automated test case. Here, there is a direct correspondence between the statements in the script and the physical actions that must occur in the interface. Any change in the interface or behavior of the system will break the R/P test script.<sup>2</sup> Ideally, we would like to be able to abstract our automation test cases so they will work from release to release with minimal changes.

Let us list this as a goal of our automation program:

**Each automated test case should be built such that changes in the physical interface of the system do not require changes in the test scripts to allow them to run.**

That brings us full circle to the main topic of this column: How do we model our automated tests to remain static even when the interface of the SUT changes? The answer, as stated before in this column, is in an effective **functional decomposition** of the testing domain into abstract tasks, each task captured into a programmed, callable function. The one to one correspondence of “line of code” to GUI element action does not then exist in the test; instead it exists in the functions called by the automated test case. A change in SUT interface will still break the test cases; we cannot prevent that as the test tool must work with each GUI element. However, broken test cases are fixed by

---

<sup>2</sup> For an excellent treatment of the problems of direct correspondence of script statements to physical GUI elements may be found in the June 2001 issue of The Journal of Software Testing Professionals, written by John Kent.



repairing the code in the functions and not the scripts. The maintenance on 1000 scripts should not be appreciably more than that on 50 scripts.

We want to model our automated test cases on manual test cases using the abstract task model. The better we do that, the closer we'll be to achieving successful and effective automation. A manual test case has very little overhead when compared to an automated test case; it pays back in positive return on investment (ROI) every time it's run. An automated test project must strive for that very outcome: reduced cost per test case through lower maintenance costs. Without a positive ROI over the course of the project, there is no valid reason to automate.

If we can, we would like to construct our automated test case by filling in a "document" using a similar model used for a manual test case:

**task | data to use for the task | expected action**

Note the similarity between the manual test case step and the function call:

**Function\_Name(Data1, Data2, ...)**

Where the expected action is captured either in the data passed in or the function code logic.

Note that it does not matter whether or not a script is used. Many automators believe that scripts should never be used, and that a table-driven architecture should replace individual scripts when creating test cases.<sup>3</sup> However, the structure of the test case does not really depend on the run-time architecture used; if we abstract the task correctly, we should be able to use any execution structure efficiently.

Having decided to try to emulate a manual test case, it is imperative that we understand how the manual tester adds context and reasonableness to the test case in order to create an effective test. Since we will be using functional decomposition to create executable tasks for our test cases, we must figure out how to programmatically institute the needed context and reasonableness inside each function.

In the last column, we discussed the way a human tester interacts with the event-driven GUI. Let's review it here quickly as the base for the functions we will build. A good tester does not just bang on the keys to run a test. They actually cycle through a fixed set of physical steps, causing the SUT to run through the test scenario. At each step, they receive feedback from the system that may cause them to alter the next step subtly to

---

<sup>3</sup> I will discuss the issues of table-driven architecture vs. scripts in an upcoming column.



ensure they can continue. Through a synthesis of context, reasonableness, a step-by-step test case and the SUT, the test is run to completion by the tester.

At the very lowest level, the cycle performed by the tester consists of the following four steps:

1. **Ensure that the SUT is in correct state to perform the task.**
2. **Fill in the interface elements necessary to instruct the system how to perform the next task.**
3. **Trigger the task.**
4. **Wait for completion or some state that indicates the task is not going to complete.**

Step 1 essentially deals with context: can we physically perform the action in the SUT's current state? Step 4 deals mainly with reasonableness: Did we get what we expected? Did we get a steady state other than what we expected? Did we even finish?

In the following sections, we will deal with how we can make our automated test functions follow these same four steps so that we might add both context and reasonableness to them. If we can develop a template for dealing with these four steps, we should be able to automate just about any task. If we can automate tasks, then we can stack them (much like a manual script) and create full test cases.

### **Step 1: Where the Heck Are We?**

The first step simply ensures that we can actually start to perform the task. For example, if we are suppose to try to open a file, but we are not sitting in the main window with access to the menu item, "Open File...", we have to change the state of the system to get to the main window.

This action is simple for manual testers. They determine the state of the system as reflected by the visual cues on the screen. If the main window is not active, they apply keystrokes and/or mouse actions to move the state of the SUT where it needs to be.

Unfortunately, for our automation needs, we do not have this luxury. Conceivably, we could interpret the state of the SUT and apply an algorithm to it forcing a state change by moving to the place where we need to be. Consider, however, the complexity this would add to each function! Since there could be in innumerable states other than the required one, we must be able to test for where we actually are, and then calculate how we can move to the state where we need to go.

Complexity is the enemy of all programming tasks; it increases maintenance and reduces ROI.



Fortunately, when taken in the context of our task-based automation, this is a solvable problem. If the test case is correctly designed, the “step 4” of each task will always leave the state of the SUT in exactly the correct position for the next task’s “step 1.”

Therefore, if the SUT is not in the correct state, it shows either a logic error in the script design, or an execution error. Either way, we want to let the user know that there was an error. If we are not in the correct state: we will log the problem, throw an error, and immediately fail the test case. Period. Do not try to figure out what went wrong, nor try to move to the correct state programmatically. Keep each function as simple as possible!

So how do we determine that we are in the correct state to continue? If the SUT is well designed, there should always be a visual cue to the system’s current state. Usually we are looking for the correct window to exist and be enabled for input.

Each tool has a way of determining the existence and state of a window. I don’t particularly like any of the built in functions, so I have written my own “where am I” for each tool I use.<sup>4</sup> In pseudo code, the function is shown below:

```
WaitForWindow(Window, TimeToWait, IsEnabled)  
  If (IsEnabled = TRUE) Then State = “Enabled”  
  Else State = “Exists”  
  Do  
    If (WindowName is in State) then Return 0  
    Else Wait 1 second  
  While (--TimeToWait > 0)  
  Return 1  
End Function
```

Note that the function will return a 0 if the window is found in the expected state, else it will continue to cycle around waiting for the window to move into that state. Each time the window is not found in the correct state, we wait 1 second and then try again. The scripter calls this function with the name of the window, the time they are willing to wait, and the state required: either the window exists, or it both exists and is enabled. If WaitForWin times out, it returns a 1. The scripter forces an error and returns failure from the function which then ripples up and forces a failure of the script. An invocation of the function follows:

```
If (WaitForWin(MainWin, 10, TRUE) = 1) Then  
  Error Step1Synch, “Expected the “ + MainWin + “but did not get it.  
EndIf
```

---

<sup>4</sup> I will discuss why I do not use the tool built in functions for synchronization when we come to step 4.



Again, this assumes an error handler (such as QARun™ and Mercury WinRunner™ have.) If the tool does not contain an error handler, then other code must be written to simulate the behavior.

As stated before, this code may not strictly be necessary for step 1, as the previous task's "step 4" should have left the SUT in the correct state, or else thrown its own failure. I tend to be redundant when it comes to sanity checking, however, so I normally do use it.

Assuming we are in the correct place, the function execution now moves to step 2.

### **Step 2: Fill in Stuff**

This step is very similar whether it is taken by a manual tester or an automated test. The interface interactions necessary to set up the state change are performed sequentially. Edit boxes get typed in, buttons get clicked, list items get selected, etc.

For automation, the scripting method may be simple brute force where each control is listed and data is passed in according to the tool's standard methods. In fact, the individual steps may simply be recorded. Or, there are a number of clever programming tricks which can be taken to minimize the amount of code in the function, such as using data-driven techniques where the data being read in from the data base / spreadsheet are fed to the correct controls automatically.

The only critical element to this step is ensuring that the automation mechanism used to deal with interface elements monitor for failure to feed each control correctly. For example, if an attempt is made to feed an edit box that is disabled, the automator must make sure that a run-time error is generated, the function failed, and consequently the test case failed.

Each automation tool allows different control feeding possibilities; I will discuss some of the different techniques in an upcoming column.

Once all of the interface changes have been completed, we ready for step 3.

### **Step 3: Go**

Again, this step is identical whether performed by an automated script or by a manual tester. Whether the trigger action is pushing a button, clicking on a menu item, or some other kind of action, it represents the last point where the task may be aborted. After the trigger action, the operating system working with the SUT will take over and complete the action. Step 4 starts immediately upon triggering the action.



### **Step 4: Are We There Yet (Repeat as Necessary)**

This step is by far the most problematic in all of automation. If every test case worked exactly the way we expected, this step would not even be necessary. Unfortunately, far too often we do not get what we expect, either through intentional change in interface or actual defects.

For manual testers, the waiting time after triggering the action is often not even noticeable. They await one of four possible outcomes:

1. If the action completes correctly, it is almost a seamless slide into the next action.
2. If the action does not complete, and goes into never-never land, the tester will tend to wait an appropriate time, reset the application and/or the workstation, and move onto the next test case after recording the error.
3. A third possibility is an unexpected reaction, such as an error dialog or message dialog signifying a fatal condition for the test.
4. The fourth possibility is a message box signifying a condition that may be rectified by a positive action by the tester / automated test case. If we are opening a file, an example might be a message that the file cannot be opened because the drive is not mapped. If the user maps the drive to the correct server, the file could yet open. Of course, having mapped the drive, the action may yet cause an error or go to never-never land.

It is obvious that the current generation of automation tools are not designed to compete with the manual tester in handling all of the possible outcomes to an action. The unenlightened automator and the recorded test case will assume that actions will complete correctly. Fortunately for them, the task often will result in the expected reaction in the short term. Unfortunately, an optimistic automator and a recording script will usually create automated tests which will eventually languish in the bit bin because they fail to handle the unexpected in the long term and require too much maintenance over the lifetime of the project.

The successful automator must be a pessimist – or perhaps a realist. They understand that if there are 20 lines of code in the function that there are 21 different places it could fail. When running on a half dozen different platforms with varying levels of operating system patches, with SUT changes from build to build, with unproven test environments and uncertain networks, the possibilities of the function always running successfully are extremely slim. The difference between successful automation and shelf-ware is often the error handling in the step 4 phase of each function.

Automation tools do not have a good way of dealing with step 4 problems built-in; they are built using the assumption that we are always waiting for a single thing to happen. Luckily, the tools do give automators the facilities needed to create very robust solutions. The heart of the solution is in the WaitForWin() function defined earlier.



Consider the problem. We have triggered an action and the operating system has taken over performing the task. The test case must wait for the action to complete before continuing on. If the expected reaction to the task occurs, we can go on to the next task. If, however, something else happens, or no known reaction occurs within a reasonable time, the function must be able to decipher the event, log the discrepancy, and inform the script that the test case has failed.

For a concrete example, we will look at the file open we have been discussing through out this column. We have walked through the interface in the guise of the Open File Dialog box and triggered the open action. Now we are waiting for the file to open. But it might not open. We may get one of a dozen different error messages (e.g. File is corrupted.) We may get a warning message with a range of choices (e.g. file is from a previous release. Do you want to open it, convert it, or cancel?) We may get nothing right away; the file may be quite large and taking a long time to process. We may get nothing ever; perhaps the word processor has had a catastrophic failure and frozen the process. How will we handle all of the eventualities?

For previous readers of this column, the answer was covered in the March 2001 issue: the **Single-Action While Loop (SAWL.)** This programming construct allows us to wait for any number of different actions and deal with any eventuality. The SAWL will wait for a given number of seconds (determined by the automator) for a resolution to the task. As it cycles through the known possibilities, it acts much like the manual tester who can deal with any known eventuality as well as the unknown. To get a complete understanding of this construct, please refer to the March 2001 column. For our purposes here, it can be seen as a number of WaitForWin() functions in a while loop that executes once each second until either a pre-determined reaction occurs or the loop times out.

The [pseudo] code follows:

```
1  While (--Tim > 0)
2      If (WaitForWin(MainWin, 0, TRUE) = 0) Then
3          Return File opened correctly
4      Else If (WaitForWin(ErrorWin, 0, FALSE) = 0) Then
5          Handle Error and Exit
6      Else If (WaitForWin(MessageWin, 0, FALSE) = 0) Then
7          Handle Message and Maybe Exit
8      Else
9          Wait 1 second
10     EndIf
11 End While
12 If Timed out, log error, take picture
```



A couple of notes. Each of the WaitForWin() functions has a time argument of 0 seconds. This permits us to check for several specific conditions very quickly with a minimum of resources being expended. Only if the condition exists does the clause handling the condition fire. This is why I prefer using my own construct, the WaitForWin(), rather than the built-in synchronization functions. I can explicitly control the timing and return value of the function and then stack them in any order I want.

In line 2, we check for the main window to be enabled (the File Open dialog has closed and there was no other dialog opened.) This would signify the file opening successfully.

In line 4, we check for the existence of an error box; if an error box has popped, we pick up the error message (and possibly compare it to the expected error message) [in line 5] and exit the function.

In line 6 the existence of a message box is checked for. If there is a message box active, we may handle it in line 7 and continue on in the loop or we may exit the function if we were not prepared to handle the message.

By adding more “else if” statements, we can handle any eventuality we know about. The time out handler is in case the action never finishes (i.e. freezes) or we get an unknown steady state reaction (i.e. a new kind of dialog box we never saw before.)

As stated before, this is not meant to be a complete explanation of the loop; instead it is meant to point out that we must have a way of traversing step 4 satisfactorily. We must add the reasonableness that the manual tester adds through their experience and senses.

The entire function structure will look like the following:

**Function DoSomething(Arg1, Arg2, ...)**

```
// Step 1  
If (WaitForWin(MainWin, 10, TRUE) = 1) Then  
  Log the error and exit  
EndIf  
  
// Step 2  
Handle the interface elements  
  
// Step 3  
Press Go button  
  
// Step 4  
While (Not timed out)
```



*Innovative solutions to maximize your testing investment.*

```
If (WaitForWin(Expected_reaction_Win, 0, TRUE) = 0) Then  
  Exit Function with success code  
Else If (WaitForWin(ErrorBox, 0, FALSE) = 0) Then  
  Handle and log error, exit with error code  
Else If (WaitForWin(MessageBox, 0, FALSE) = 0) Then  
  Handle message- if recoverable, continue, else exit with error  
Else  
  Wait for 1 second  
  Decrement counter  
End If  
While End  
  
// If we get here, we timed out  
Log error, take picture of screen, exit with error code  
End Function
```

This template may be used for almost any functional task that we might have to create for our automation task. Note that it follows the logic of the manual tester very closely, by adding context checks and reasonableness to an automated test case through every step that is taken.

In the next issue we will start to deal with different automation architectures. After all, it matters not whether this one test can execute intelligently. What will really determine the success of our automation program is whether we will be able to run 100 or 1000 test cases end to end without human intervention. Do not be frightened off by the term architecture; consider it as simply a term which answers the question, “can we run 1000 of these scripts seamlessly and unattended?” We’ll talk about it next time.