



## **Automation By Design: A Model Of Reality**

**by Jamie Mitchell**

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

So I'm sitting here in the Minneapolis airport, waiting for a jet plane to take me to yet another place to talk about how to's: how to solve this test automation problem using that technique on this platform with this operating system, yadda yadda yadda. It struck me that I am always talking about how and rarely about why. Why doesn't this automation technique work as advertised? Why doesn't record / playback ever work (well, so very rarely that it is hardly worth talking about.) This column is about test automation, and mostly we talk about how to make it work, how to get the greatest return on the investment we make. Today we are going to talk about why we need to do what we do. Rather than looking at the trees as we have in the last several columns, let's look at the forest.

A basic tenet of good software engineering is that we must model reality to build good software. Test automation is software development; however, I rarely hear anyone speak about modeling when speaking about automation. Today we will.

Ask just about any automator why the basic record/playback (R/P) methodology does not work well, and you are likely to hear a number of well-rehearsed mutterings: hard coded values, static synchronization values, can't handle changes, etc. While all of these are true to some extent, perhaps we can get to the heart of the matter by basing the discussion on the model that R/P represents.

A recorded script is based on a model that the tool vendor envisioned when creating the tool. That model appears to include a world where:

- Each test case is completely repeatable, with only very minor variations.
- The script recorder does not really need any special skills – often the vendor recommends that a business analyst is sufficient to record the script.
- The environment is always stable enough that a recorded script is repeatable.

To be fair, I do not believe that tool vendors (or at least their technical people) believe these lines. A few weeks ago, I started an engagement with a client who had purchased QA Center from Compuware. It has been two years since I had last used QARun, so I sat down to reread the manual (my standard first step before starting to use a tool; one I highly recommend.) There was a section in the manual that clearly listed the



shortcomings of record/playback and recommended using a functional decomposition model to build a framework architecture. In talks with high-level people at Mercury, Segue, and Compuware, I have found that they consistently acknowledge that automation is more than recording. No, I don't believe that the vendors actually believe the R/P model; but they sure do sell a lot of tools by claiming the methodology works. Marketing! *[Said with a rising inflection with a finger in the air, a la Jon Lovitz speaking about acting.]*

At any rate, what does this model say about manual testers? Do manual testers really do exactly the same thing each time through a test case? Do they add no extra value such that a business analyst can do the same testing by operating the system as if using it to execute business logic? Frankly, I believe that this particular model of testing implies that a tester is not much more than a monkey, a key-banger. If we had enough monkeys (oh, say an infinite number) then they could simply bang on enough key boards and all of the software errors would be found.

The time-tested fact is that manual testing works and that professional testers do add great value to software testing. This testing is not random and seldom accidental; there are some very good reasons that manual testing methodology and manual test cases have worked so well for at least half a century. So what part of the manual testing model is lost when ported to the R/P automation tool? More importantly, what must we add to the basic playback script to make it match the manual test case model? To understand those questions, we must look at a couple of inter-related models.

### **Manual Testing Model**

Why does the manual test which is being recorded by a R/P tool work the first time and then so often fail when played back exactly the same way by the tool? Believe it or not, there is actually a model that can be derived from observing how professional testers do their testing.

Look at a typical manual test case. In almost every case, the test consists of a couple columns of text captured in a Word or Excel file, as shown below.

1	Add a record to the database	First Name: Gerry Last Name: Franklin SSN: 234-34-5678	Record created: record # returned
2	Search for the name	Franklin, Gerry	Expect to find it
3	Edit the record	Occupation: Lawyer Income: \$125,000	Expect dialog verifying change
4	Check ordering	Record # from step 1	Expect valid ordering
5	etc		



The first column names a task to be performed; a business task that the test creator assumes is clearly understood by the tester, business analyst, or user of the system. This task is usually an action that is a single step in using the system, changing the application state from one to another. Note that there is usually no direct, one to one correspondence with the graphical user interface; the text does not say anything about moving the mouse, clicking on a key or mouse button. The assumption is made that performing the task will consist of the user performing those kind of physical actions based on their knowledge of the system and of Windows. Rarely does the test step explicitly explain out the action so fully that someone who does not understand the application can run the step. Why? Well, for one thing, the test case would become so long that it would take almost forever to write it. This would reduce the number of test cases which a tester could write. It would also make the test case brittle; any change would invalidate the test case to the point it would no longer be correct. Instead, the usual test case assumes that the tester will add some system knowledge to make up for the lack of specificity.

Contrast this with an automated test case which includes every single action that the user performs, tied intimately to the interface. Clearly, the two models diverge on the amount of detail included. The question is whether we can do anything about the need of the computer for all of the detail. Read on.

The second column of the test case consists of the data to use in performing the action; again, no direct mapping is made to the GUI. The final column, in general terms describes what reaction is expected from the system under test.

Note the beauty of this model. When there are minor changes to the system or the environment, the test case rarely needs to be modified. This is because it is just an outline of a test, an incomplete recipe or algorithm that is combined with something else in order to create an executable test case.

Now, obviously, this test case is not automatically executable. It must be taken and executed by a manual tester to have value. The tester adds *something* to the manual test case to execute the test. I believe that the intangibles that are added by the tester can be narrowed down to two things: context and reasonableness. The manual tester reads a step from the manual test case: perform this task. The relative order of several tasks is given sequentially. The tester understands the context of the action within the confines of the system and the environment and performs the physical actions on the workstation, through the GUI, to achieve the task using the data in the following column of the test case.

Note how important the context of the action is. If the tester is told to open a file, but the system is not in a state such that a file can be opened directly, the tester changes the state of the system to move to a state where the file can be opened. The test case did not direct her to do it, but the tester understands the context of the step text and acts to make it happen. Let's take it a step further. As the tester waits for the file to open, there certainly



are several possible outcomes which can occur whereby the file does not open successfully. The manual tester does not get lost or confused when the file does not open exactly as expected. If it takes longer to open one time than another time, the tester does not panic and fail the test case; she applies reasoning to the problem. Perhaps the network is slow, perhaps the file is huge. If the file does not open immediately, but pops up an intermediate error or message, the tester applies rational thought to the window's contents and determines, based on experience and rationality, whether the reaction of the system constitutes a problem or an annoyance.

The point I am trying to make is that the tester applies a standard of reasonableness to the reactions he sees, and a sense of context to the tasks which she is to perform. The very two features not built into the test tool! Our task will be to find an automation model where we can simulate these two properties of the manual tester in code.

### ***Event-driven Architecture model***

Let's look at another model which has a direct bearing on automated testing; that of the event driven or state model that all GUI architectures exhibit. In the above explanation of the manual testing model, I used the term state several times. Just what does state mean when using it in this context?

Consider how Windows works. It is usually in a steady state, just sitting, waiting for the user to do something. Certainly, in the background, Windows is busy, executing clean up tasks, updating the screen, etc. Certainly necessary stuff, but nothing much of direct interest to the user or the application at hand. Let's assume that the user wants to write a letter to someone. First step is to start the word processor. It does not start automatically - we cannot think it on. The user must instruct Windows to start up the app.

We perform a positive action, perhaps finding the icon in the start menu and double clicking on it. Windows does the rest; finding the application image on disk, loading it, initializing data structures, drawing the screen, etc. More on this in a moment. The point right now is that, after the word processor has been started, it does nothing. It does not start writing the letter; it is in a steady state; it waits for the user to initiate a new action. Every step of a test case, or a user working on a business task follows this same model: direct the application to do something, and then wait for it to finish. Each time the computer finishes a step, Windows is very patient; it waits for the user to order it to do something else. Now, when a programmer speaks about an event-driven architecture, this is not strictly what she is speaking about. It is an apt description, however, of how Windows actually works; set up an action, tell it to go. Set up an action, tell it to go. Repeat as necessary. There is no actual smooth flow through a GUI architecture; instead there is this jerky, stop and go action which consists of an order from a user and a follow through by the system.

99.9 percent of applications work this way in Windows. Now, let's get a bit more granular. What does it mean to set up an action and then tell it to go?



With Windows, we have a set of metaphors that we use to interact with an application; we call the “physical manifestation” of these metaphors controls. Each control can either help set up an action or give us an indication of the reaction to an action; some do both. Edit or list box, radio or check button, push button or label, menu item, etc. Each of these metaphors is well understood by an experienced user so that little training need be given for most applications. This was the advantage of Windows which killed DOS; every application had a common (!?) interface so that if one knew how to execute one Windows application, they likely could figure out any app they had to use.

These controls are usually how we instruct Window to move from one state to another. We interact with the controls to input data, we view data in the controls to understand the current state we are in. Think about it; want to know what state your application is in right now? What is the top window? What are the available menus? What are in the edit boxes, radio and check boxes?

Once we understand what the current state is, we are ready to direct the system to move to another state. This directly corresponds to the first model illustrated here; the tester interacts with the application being tested by seeing what state we are in currently, evaluating what the next task is to be done, and interacting with the system (via keyboard and mouse) to change the current state to the next state in the test.

The change of state for each step of the execution can be split up into four distinct parts. First the user must make the determination that the application is sitting in the correct state to initiate the change to the next state. If we are to open a file, we had better be in the main window with no other dialogs open. If we are not sitting in the main window, we must either get there (by driving the application using the same four steps) or we must fail the test case. We cannot advance to opening the file from any other state.

Secondly, we must indicate to the system what it is, exactly, that we want it to do. We do this by interfacing with controls of the application. Continuing with the file open example, we will use the menu structure to bring up an open file dialog box, then use the controls of the dialog to specify a single file to open; either by manipulating the tree view and tree list controls, or by typing in the file name and path to the edit box.

We eventually get to the point where we have supplied Windows with sufficient data to clearly explain what we want. At this point we move to the third part of the action, that of triggering it. There is always some point at which we declare we have finished with the set up and now the system should begin its work. Often the trigger consists of pushing a button, clicking a menu, or pressing the return key.

At this point, Windows takes over. No longer is the user in control; the application in conjunction with Windows is going to start cranking in an attempt to fulfill the wishes of the user. Whatever the task, whatever the context, at this point Windows is in full control



and cannot be stopped. There are three possibilities at this point: the action completes as expected (i.e. the file opens), we get an intermediate action other than the file opening (error or message box which will convey information to the user as to why the file is not yet or may never be able to open, or the action never does finish (in technical terms, the system goes to never never land.)

What does the automation tool do in a R/P script? It generally feeds in commands blindly (or almost blindly within the limits of the tools basic capability.) There is no feedback loop as in the manual model where anomalies can be handled. And there are almost always anomalies. Applications change from build to build, release to release. Environments change (Win 95 to Win 98 to Win NT to Win 2K to Win ME.) Is it any wonder that R/P fails?

If we can completely understand exactly how the manual tester deals with the event driven architecture model, we can use that knowledge to program a solution to the R/P dilemma. Next month we will continue this discussion of models and lay out a model that we can use for automation. Understanding the four step dance and the manual test case model will allow us to build a fairly robust automated test model which might just provide us with some return on our automation investment.