



Automation By Design: Simplicity: What a Concept

by Jamie Mitchell

This column was originally published in the Journal of Software Testing Professionals, Volume 3, Issue 3.

OK, I'll admit it – I think everyone should be able to cook a gourmet dinner, kill a bear, fly an airplane, and program test automation¹. Unfortunately, this is yet another case of where our school system has failed us!² Specialization appears to be the bane of civilization.

In the last issue, we discussed how to utilize the architectural concepts (covered previously) to mass produce automated test cases in those non-trivial numbers that we need. In this issue, we will discuss a concept that is absolutely essential if we want our automation architecture to be successfully utilized by the other members of our group.

Many people in software development are specialists at what they do. They are not only programmers; they are C programmers for Unix systems. They are not just network specialists; they are experts at 8255x Ethernet connected networks. One of the few exceptions to specialization in the IT game is testers. In many cases, testers are expected to be generalists. Technical enough to perform white box testing, business savvy enough to understand the needs of the end users, mathematically inclined to handle the metrics and chart the trends. My experience is that testers usually have a very good set of skills – often, they just do not happen to include programming skills.

Sure it would be nice if every person could perform every task. Simply plug the jack into the socket in the back of your head, run the Java programming training program, and voila: instant expert. There is no spoon ...³

In today's world, it is an economic reality that test organizations usually have to make do with the talent that they have on staff. When there is a critical need for a specific skill, they may sometimes hire a specialist, tempered, of course, by the fact that they have to pay relatively more for that person and will likely have to make a cut elsewhere.

Test automation is software development; to program test automation requires very specific skills. Let's face it, not many testers have that broad range of development skills needed to succeed using today's tools. So how do we automate⁴? Few organizations face

¹ With apologies to Robert Heinlein

² Mostly just kidding: please don't send any nasty letters.

³ If you don't get this quote, please run, do not walk, to your local video store and purchase a copy of "The Matrix." Your mind will thank you.

⁴ In many cases, badly!



Innovative solutions to maximize your testing investment.

up to the fact that skills are needed beyond that three day class supplied by the tool vendor, and it costs them in failed automation projects and wasted effort.

Is there an answer? Of course! Make test automation something other than a software development task. Oops – now we are back to record / playback. Or are we?

Linda Hayes uses an outstanding metaphor to describe the skills dilemma. She points out that most of us drive cars, but very few of us are mechanics. The driving interface is separate from the maintenance interface. When the car starts making a funny noise, we drive it into a repair shop; a few arm waves by a specialist, some magic incantations, a hefty fee and we are back on the road.

When I was in the Navy, some thirty years ago, I bought a 1966 Ford Galaxy 500 with the police interceptor engine: 428 cubic inches of Detroit heavy metal with a four barrel carburetor. That car would run rings around most of today's cars; it drank premium gasoline and got reasonable mileage for the time. One of my most vivid memories of that car was spending almost every weekend under the hood, tinkering with the engine. Tune it, clean it, pull something off and then put it back on, doing this and that under the hood. Fast forward to now. I own a 1996 Buick Park Avenue with a 3800 cc engine. I wouldn't even start to think about trying to work on that engine myself. Opening up the hood reveals a dizzying amount of stuff, little of which I can identify. Computers contained in beige boxes, high tech sensors for this and that – at least that is what I am told. As technology advanced, more and more stuff was added under my hood.

I never really thought of this as progress until I took the car in for a 75,000 mile checkup. I asked the mechanic to tune up the engine; I was told that there was nothing to tune until 100,000 miles. It suddenly occurred to me that I had never done anything to the engine other than getting the regular 3,000 mile oil changes and lubes. No greasy Saturdays, no oily Sundays.

The point? By adding high-tech stuff to the engine during the creation of the car, GM removed the onerous responsibilities of constant maintenance to my ownership. The owner / driver interface was simplified at the expense of the mechanic / maintenance interface. I got the benefit of going from here to there without the pain of fixing it all myself. I am responsible for starting the car and navigating it to where I want to go without having to care what the squirrels under the hood are doing. Another way to put it was that I was able to purchase personal simplicity from GM and they delivered it by adding complexity to the design.

Getting simplicity by adding complexity.

Back about 12-13 years ago, when I tried to first connect my computer to the internet, I was stymied at the complexity. Get in and change this flag at this offset, reboot and set this value to ... well, you get my drift. I never was able to do it myself. Today, when



you buy a computer, the little MS wizard dude does all of the connection stuff for you. Automatically! It obviously took some programming skills to handle all of the connection possibilities automatically; it makes it possible, however, for my sister to connect up to send me an email when she has trouble even spelling computer.

Simplicity from complexity.

We need to learn the “simplicity from complexity” lesson when dealing with automation. We certainly need to have the automation specialists who know how to twiddle the bits and bytes; the wizard who solves the technical problems so that the rest of the staff does not have to worry about them. Then we need the ‘drivers’, the essentially non-technical personnel who know how to drive the automation infrastructure to actually do the testing. There is absolutely no reason for the average automated test creator to know all of the technical magic going on behind the scenes; no more than there is a good reason for the average driver to be an expert mechanic.

We, as the automation experts, need to do more than come up with a way to automate test cases; we need to always be coming up with easier and better ways to create the test cases, kick off the automation, to evaluate it while it’s running, and determine what the end results are.

Most of the columns I have written for this journal have dealt with making the creation of automated tests easier for non-technical testers. In this issue, I will discuss ways of making the automated test cases easier to run and evaluate. I believe that both issues are actually one and the same. Reduce complexity for users – at every level. This reduction will typically be done by adding complexity [through programming] to the automation suite.

Simplicity from complexity.

I once consulted at a company that had the requirement that non-technical people needed to be able to add automated test cases to the suite – and to run them. These non-technical people included testers, analysts, and ... lawyers. Lawyers! *Insert your own lawyer joke here.* It started me thinking, however. How often had I seen the case where it was desirable for the automation to be run but the owner / automator was not available to kick it off? So how do we make it so simple to run that a lawyer can do it?

The tool vendors have tried to simplify the actions needed to run the automation suite, mainly by creating a tool to manage the testing process. Mercury Test Director™, Compuware QA Director™ and Rational TestManager™ are examples of this class of tool. When used intelligently, these tools can help a test organization plan their testing more efficiently. However, as a solution for the problem of starting up a test suite, they fall short (in my humble opinion.) Why? They substitute learning how to start one tool



with learning how to start another tool. Another complex tool with complicated screens and shortcuts and icons and toolbars and menus and tab sets and ... lots of stuff.

Maybe the easiest way to start a complex test suite is to consider it from the viewpoint of simplicity. Assume that the person who wants to run some tests is intelligent but is afraid of all technology. Assume that they can read and follow simple instructions. Assume that they know the test cases that they want to run.

Our test suite interface, using these assumptions, looks radically different than it might if we were to simply use a purchased R/P tool.

Starting the Suite

Double click on an icon. I think I am on safe ground by assuming that anyone who uses a GUI driven operating system can perform this task. Therefore, I try to always give the test runner a simplified interface to start the suite: that of an icon on the desktop to double-click.

It turns out that this is a fairly simple thing to set up in most of today's tools. The tool can be started by specifying a command line that specifies the tool and startup script, and then executing it. The icon is created; the startup directory and command line are specified. One problem with this simplified scenario, however, is that we may want to run different flavors of test suites. Each prospective test runner certainly has their own agenda, a set of tests that they want to run. This agenda is likely to change depending on project cycle or a host of other reasons.

We could create an icon for each different permutation of test cases that we might want to support. Of course, this rapidly degenerates into specialized training on selecting the correct icon from the hundreds that now populate the screen. Not very simple.

Instead, let's add some complexity to simplify the interface. We need a way to determine from the user just what tests they want to run. We need a way to [conceptually] ask the user, "Where do you want to go today?" We need a way to intelligently interact with the test runner. We need ...

Prompts

Interactive prompts, allowing the test runner to execute exactly what automated tests they care to, are an integral part of my automation strategy. This means that I want to build a framework of interactive prompts that any person capable of reading can understand and respond to. By interacting with the prompts, a person can choose to run as many or as few test cases as they need to. All of the suite options are available for choice during the prompting session; and the instructions are given in plain English (or any other language) for easy selection. This allows any person in the organization to kick off a set of tests



when needed; not coincidentally, it makes the automation suite more likely to be run in different circumstances, thus improving the possibilities of positive ROI from the suite.

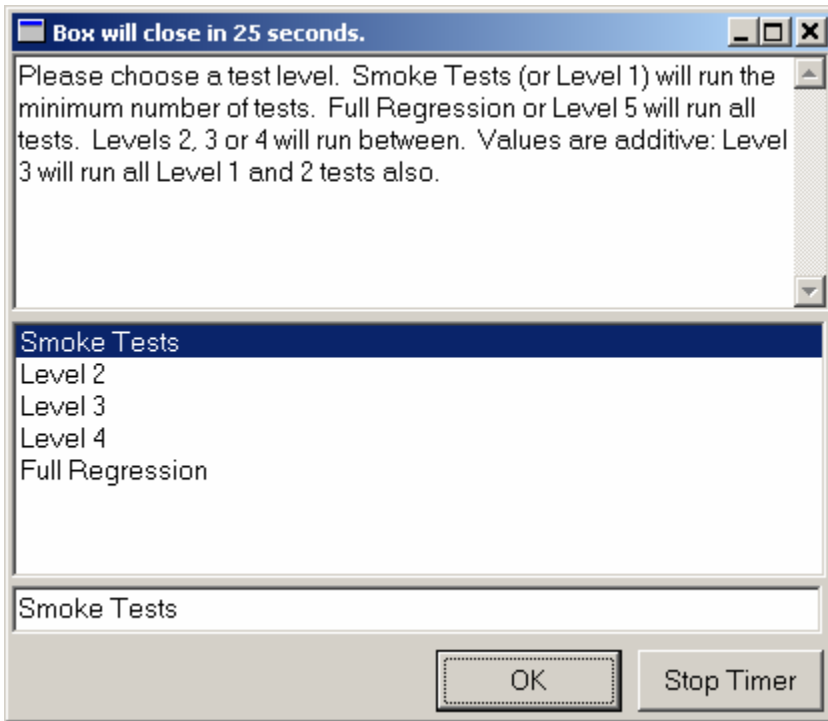
The entire suite starts by double-clicking on an icon on the desktop of a workstation. This triggers the prompt sequence. As the automator, it is incumbent on me to set up the prompts in such a way that a prospective test runner can perform any testing they need. That is why I have the fields in my test headers for test level and functional tasks (as explained in the last issue.) By categorizing each test in different, general ways, I can ask the test runner to make rational decisions on the specific sets of test cases they want to run. Or, they can simply kick off the entire suite all at once.

All of the major automation tools do supply a way of prompting the user for information. I have found it more valuable to create my own prompts and place them into DLLs to make them callable from the automation tools. Why?

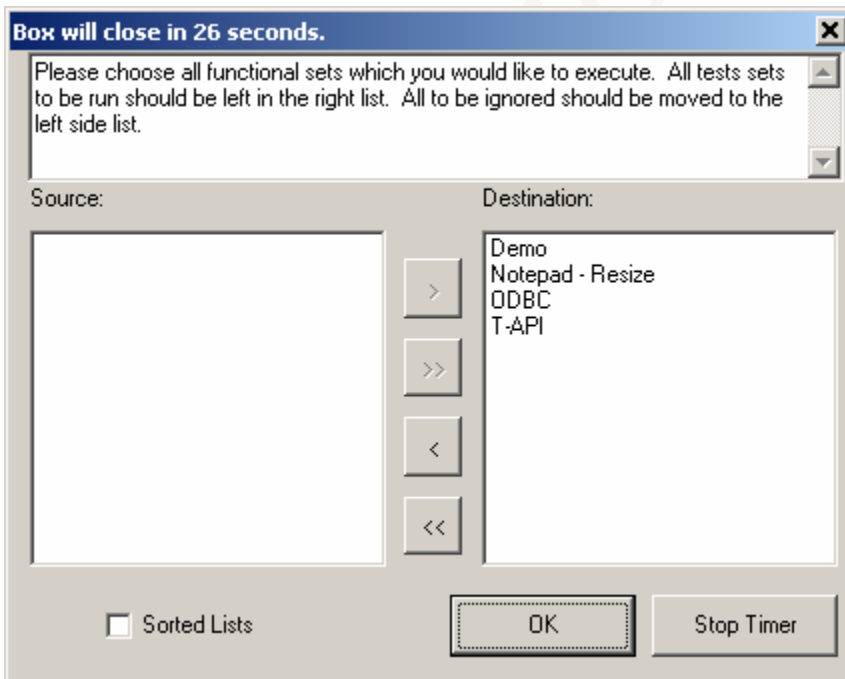
Disclaimer!!

Prompts are potentially dangerous in automation. Since automation is designed to be run unattended, putting a prompt up is an act of faith: faith that there will be a person available to respond to the prompt. As a person of little faith, I decided that I needed a better solution. The prompts that I wrote have a timer built into them. When an automation developer wants to use one of my prompts, they supply the prompt information plus a default value and the amount of time in seconds that they are willing to wait for a response. If no person answers the prompt within the specified time, it chooses the default value, closes, and the test case continues. If you have Mercury WinRunner, you have seen this kind of a timed prompt already; they use it in their start up routine when they prompt for addin types to select.

The following are a couple of the prompts that are triggered when I start a little demo test suite. Note that these are called from my custom DLLs and look exactly the same no matter the automation tool or Windows platform that I am using for the test. The first prompt allows the user to choose the test level. The user does not have to understand any of the technical details of the automation as long as they know how the tests themselves were categorized. Since the categorization has nothing to do with the automation, it is specified by the test personnel when planning the testing, the test runner needs to know only the test domain.

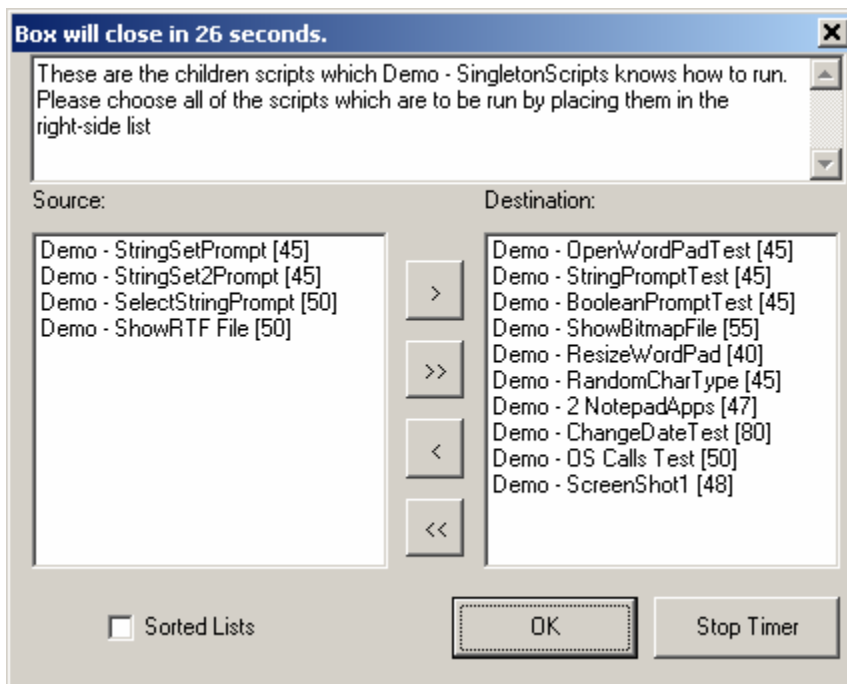


Likewise, the next prompt has to do with functionality to be tested; again, the test runner does not need to know anything about automation for this step.





Once the gross characteristics of the suite execution have been selected, then we can put up a series of prompts that allow the test runner to drill into those areas that they really want to test. All of the items in all of these prompts are picked up directly from the test headers that we spoke of in the last issue. The number appended to the end of each selection is the relative order number.



The relative order value is simply a numeric value (0..100), given at test creation time, that allows the tester to specify a general order. If test X absolutely must be run after test Y, for whatever reason, the relative order value for X is set to a higher value than Y. At run time, the tests are then sorted according to this value.

Developing the prompts themselves was fairly simple once the idea occurred to me. Using Borland Delphi™, I created a prompt dialog, and attached a timer thread to it. Putting it in the DLL was a little more complex, but easily doable; the code came directly from a Delphi book. Total time to create the first working prompt: about 20 hours. Each successive, different prompt: about 2 hours. Because the prompts are contained in a DLL, they can be used from any automation tool.

Can any automator create such prompts? On a scale of 1 to 10, with 1 being a simple user and 10 being a programming guru, I would estimate that the skill needed to create these prompts is about a 5.5.

By creating a hierarchy of test cases, represented in the test header store, we can allow the test runner to slice and dice the suite to run only the test cases that they select via



Innovative solutions to maximize your testing investment.

point and click. They read the prompt, move test case names from list to list and press go.

Simplicity from complexity. This is a theme that I will continue to visit in future columns; I believe that the key to successful automation hinges on this concept. As automators, we create simplicity for our clients (test creators and test runners) through programming. We have to start thinking bigger. Too often I hear automators talking about how little programming they have to do. Get it done quickly and move on!

We ignore simplicity at our own peril. A positive ROI depends on robust test cases – lots of them. Tests that can be created and run and evaluated and maintained easily. Without obtaining such a positive ROI flow from our automated testing, we might want to start planning our next career.