



Automation By Design

by Jamie Mitchell

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

Welcome to **Automation By Design**, a new column about software test automation for testers and other software development professionals. The automation of testing is an emerging field that can be very rewarding to both companies and individuals alike. Like many other new technologies, however, there is more than meets the eye for the uninitiated; success is not automatically guaranteed simply because your organization has bought a tool.

If you are reading this column, you probably already believe that the addition of test automation could help you improve your software development process. As a long time automator, I have seen cases where test automation has returned many times its investment in benefits. Unfortunately, I have also seen and participated in many failures.

Since failure is just an opportunity in disguise (I have repeatedly been told), we need to learn from these failures to avoid them in the future. In this column, we will analyze why some automation projects fail and try to set up some automation development rules to help you succeed.

So, what separates the winners from the losers? In upcoming columns, we will look at specific techniques and processes that separate the good from the bad, what works from what fails. I hope to share my experiences with test automation and help you learn from my mistakes.

The term automation tool can be applied to many different classes of software packages. For the purposes of this column, I will use the term to mean the record / playback class of tools, such as WinRunner™ by Mercury-Interactive, QARun™ by Compuware, SQA Robot™ by Rational, QAPartner™ by Segue, etc. We will talk about the most effective ways to write, run, and maintain automation scripts created by these tools.

Often, the first attempt at automation in an organization is to buy one of these tool sets and start batting out scripts simply by recording the manual execution of existing test cases. Now, recording is a very valuable method in an automator's toolbox; however, there is more to a robust script than just recording manual actions. Let's see why.

Creating a test automation script via recording is a fairly simple task. The end result of the recording process is a script (essentially a software program) consisting of statements in a proprietary programming language that represent the manual actions of the tester and



the reactions of the application under test (AUT) to those actions. Depending on the tool, some extra “stuff” may be inserted to handle synchronization, modest application changes, and the like.

Consider what happens when this script is played back.

The automation tool executes each statement in turn, recreating the actions that the tester originally made. If everything is exactly the same as before, the script faithfully recreates the entire run. Small changes are usually handled successfully; for example if interface objects are moved, resized, or even drawn slightly differently, the script will likely still run. Consider the possibilities, however, in a real-life scenario. The network may run slower, the playback machine may be slower, the application might run differently on other versions of Windows, the starting environment may be different. In fact, the chances of non-trivial differences occurring between script recording and running are almost a certainty. Even without a change in the application, we could have problems in the playback.

And remember why we are creating an automation script in the first place! We expect that the application is going to change; we know that the environment will. We expect that there will be bugs (or else why test?)

A manual tester executing the same scenario understands these small differences and allows for them almost unconsciously. Step by step, synchronization is handled by a mental process that views with the eyes, processes with the brain, and only starts the next step of the test case when the machine is ready for input. If a label on a control is different or an interface object changes behavior, the human brain can automatically make assumptions and allow for the difference.

But our script has no brain, no way of handling trivial changes. It is a literal minded automaton, [almost] blindly executing its way from one statement to the next. If it gets lost during the execution, it will walk right off the path to Never-Never land. We, as automators, need to take that script and add artificial intelligence to it to heighten its chances of surviving to the last statement.

We need to figure out a way to create scripts that will work when the unexpected happens, to enhance the originally recorded scripts to make them more robust and able to handle any eventuality. And, we need to make sure that scripts running unattended can survive anything that can happen in the environment.

So why wouldn't we automatically assume that these type of script enhancements are called for? Why do we so often see organizations who do not succeed with pure recording techniques give up on automation all together? My belief is that we have made a fundamental error in the way we think about automation tools.



Suppose your organization needs a mission critical, multiple-tier, client-server data warehouse system to solve the problems it is experiencing in marketing its products. How would that need be fulfilled? One possibility is for the organization to buy the latest version of MS Visual Basic™, hire a sharp new college graduate with the latest skills and put the two together. *And, incidentally, we would like the system finished next month!* So what are the odds that the solution written would solve the problem?

This argument is specious, since no organization expecting to be in business next quarter would even entertain the thought. Yet, consider how we handle the automation of testing! We buy a very expensive set of test tools, take our best tester, give her 3-5 days of hands on training, and ask her to automate all of our testing. *And, incidentally, we would like the test suite finished for our next release!* Is it a wonder that we tend to see failures in automation?

The failure is not in the tools. The tools today are extremely good at helping the user do the right things. The problem is more in the organization that expects relatively non-technical people, no matter how good their test skills are, to complete a very technical task successfully. My experience is that there are two types of people in software development: those that like to create and those that like to destroy (then there are managers, but that is a different column.☺) The ones who like to create become developers, the ones who enjoy to destroy become testers. The skill sets between these two are disjoint; what makes a good tester is different than what makes a good developer.

No value judgment here, simply an observation.

Software automation is inherently complex. It is technically complex. To be successful, the automator must develop software (a set of scripts) that executes flawlessly on top of a complex operating system, running software that is known to be buggy.

Sounds as if we might as well give up, right?

Well, the good news is that, over the last 50 years, we have created a set of processes which have solved all these problems (and others we will get to later.)

These, in their entirety, are called Software Engineering techniques; I am willing to wager that your organization already has a mature set of them in use every day. All we need to do is take the best techniques your organization already uses and apply them to the domain of software testing through automation.

For now, lets simply list just some of the reasons we need to handle the engineering side of automation development to be successful:

1. **Maintenance**: if you have a lot of test scripts, you will have to maintain them. Like any software, better design will facilitate maintenance.



Innovative solutions to maximize your testing investment.

2. **Logistics**: The more test scripts you have, the more difficult it becomes to successfully run them.
3. **Fault Tolerance**: The more scripts you have, the more certain it is that you will have failures. You must not let a single script failure stop all the other scripts from running.
4. **Multiple environments**: you must be able to leverage your testing from platform to platform.
5. **Multiple languages**: when you localize your application, it is essential that you can leverage the investment you have already made in automation to test it.

The good news is that software test automation is not brain surgery or even rocket science. We can apply well-understood techniques to make automation work. We can help software-testing keep up with internet time, RAD development (and any other buzzwords currently circulating.) Check in next issue and we will get started on the road to success.