

Test Automation Snake Oil

V2.1 6/13/99

Copyright © 1999, James Bach

This article is revised from earlier versions published in Windows Tech Journal (10/96) and the proceedings of the 14th International Conference and Exposition on Testing Computer Software, respectively. The author thanks ST Labs for supporting this work.

Case #1: A product is passed from one maintenance developer to the next. Each new developer discovers that the products design documentation is out of date and that the build process is broken. After a month of analysis, each pronounces it to be poorly engineered and insists on rewriting large portions of the code. After several more months, each quits or is reassigned and the cycle repeats.

Case #2: A product is rushed through development without sufficient understanding of the problems that it's supposed to solve. Many months after it is delivered, a review discovers that it costs more to operate and maintain the system than it would have cost to perform the process it automates by hand.

Case #3: \$100,000 is spent on a set of modern integrated development tools. It is soon determined that the tools are not powerful, portable, or reliable enough to serve a large scale development effort. After nearly two years of effort to make them work, they are abandoned.

Case #4: Software is written to automate a set of business tasks. But the tasks change so much that the project gets far behind schedule and the output of the system is unreliable. Periodically, the development staff is pulled off the project in order to help perform the tasks by hand, which makes them fall even further behind on the software.

Case #5: A program consisting of many hundreds of nearly independent functions is put into service with only rudimentary testing. Just prior to delivery, a large proportion of the functions are deactivated as part of debugging. Almost a year passes before anyone notices that those functions are missing.

These are vignettes from my own experience, but I bet they sound familiar. It's a common complaint that most software projects fail, and that should not surprise us— from the outside, software seems so simple, but the devil is in the details, isn't it? Seasoned software engineers know that, and approach each new project with a wary eye and skeptical mind.

Test automation is hard, too. Look again at the five examples, above. They aren't from product development projects. Rather, each of them was an effort to automate testing. In the nine years I spent managing test teams and working with test automation (at some of the hippest and richest companies in the software business, mind you), the most important insight I gained was that test software projects are as susceptible to failure as any other software project. In fact, in my experience, they fail more often, mainly because most organizations don't apply the same care and professionalism to their testware as they do to their shipping products.

Strange, then, that almost all testing pundits, practicing testers, test managers, and of course, companies that sell test tools recommend test automation with such overwhelming enthusiasm. Well, perhaps "strange" is not the right word. After all, CASE tools were a big fad for a while, and test tools are just another species of CASE. From object-orientation to "programmerless" programming, starry-eyed advocacy is nothing new to our industry. So maybe the poor quality of public information and analysis about test automation is not so much strange as it is simply a sign of the immaturity of the field. As a community, perhaps we're still in the phase of admiring the cool idea of test automation, and not yet to the point of recognizing its pitfalls and gotchas.

Let me hasten to agree that test automation is a very cool idea. I enjoy doing automation more than any other testing task. Most full-time testers and probably all developers dream of pressing a big green button and letting a lab full of loyal robots do the hard work of testing, freeing themselves for more enlightened pursuits, such as playing games over the network. However, if we are to achieve this Shangri-La, we must proceed with caution.

This article is a critical analysis of the "script and playback" style of automation for regression testing of GUI applications.

Debunking the Classic Argument for Automation

"Automated tests execute a sequence of actions without human intervention. This approach helps eliminate human error, and provides faster results. Since most products require tests to be run many times, automated testing generally leads to significant labor cost savings over time. Typically a company will pass the break-even point for labor costs after just two or three runs of an automated test."

This quote is from a white paper on test automation published by a leading vendor of test tools. Similar statements can be found in advertisements and documentation for most commercial regression test tools. Sometimes they are accompanied by impressive graphs, too. The idea boils down to just this: computers are faster, cheaper, and more reliable than humans; therefore, automate.

This line of reasoning rests on many reckless assumptions. Let's examine eight of them:

Reckless Assumption #1

Testing is a "sequence of actions."

A more useful way to think about testing is as a sequence of interactions interspersed with evaluations. Some of those interactions are predictable, and some of them can be specified in purely objective terms. However, many others are complex, ambiguous, and volatile. Although it is often useful to conceptualize a general sequence of actions that comprise a given test, if we try to reduce testing to a rote series of actions

the result will be a narrow and shallow set of tests.

Manual testing, on the other hand, is a process that adapts easily to change and can cope with complexity. Humans are able to detect hundreds of problem patterns, in a glance, an instantly distinguish them from harmless anomalies. Humans may not even be aware of all the evaluation that they are doing, but in a mere "sequence of actions" every evaluation must be explicitly planned. Testing may *seem* like just a set of actions, but good testing is an interactive cognitive process. That's why automation is best applied only to a narrow spectrum of testing, not to the majority of the test process.

If you set out to automate all the necessary test execution, you'll probably spend a lot of money and time creating relatively weak tests that ignore many interesting bugs, and find many "problems" that turn out to be merely unanticipated correct behavior.

Reckless Assumption #2

Testing means repeating the same actions over and over.

Once a specific test case is executed a single time, and no bug is found, there is little chance that the test case will ever find a bug, unless a new bug is introduced into the system. If there is variation in the test cases, though, as there usually is when tests are executed by hand, there is a greater likelihood of revealing problems both new and old. Variability is one of the great advantages of hand testing over script and playback testing. When I was at Borland, the spreadsheet group used to track whether bugs were found through automation or manual testing-consistently, over 80% of bugs were found manually, despite several years of investment in automation. Their theory was that hand tests were more variable and more directed at new features and specific areas of change where bugs were more likely to be found.

Highly repeatable testing can actually minimize the chance of discovering all the important problems, for the same reason stepping in someone else's footprints minimizes the chance of being blown up by land mine.

Reckless Assumption #3

We can automate testing actions.

Some tasks that are easy for people are hard for computers. Probably the hardest part of automation is interpreting test results. For GUI software, it is very hard to *automatically* notice all categories of significant problems while ignoring the insignificant problems.

The problem of automatability is compounded by the high degree of uncertainty and change in a typical innovative software project. In market-driven software projects it's common to use an incremental development approach, which pretty much guarantees that the product will change, in fundamental ways, until quite late in the project. This fact, coupled with the typical absence of complete and accurate product specifications, make automation development something like driving through a trackless forest in the family sedan: you can do it, but you'll have to go slow, you'll do a lot of backtracking, and you might get stuck.

Even if we have a particular sequence of operations that can in principle be automated, we can only do so if we have an appropriate tool for the job. Information about tools is hard to come by, though, and the most critical aspects of a regression test tool are impossible to evaluate unless we create or review an industrial size test suite using the tool. Here are some of the factors to consider when selecting a test tool. Notice how many of them could never be evaluated just by perusing the users manual or watching a trade show demo:

- ◆ **Capability:** Does the tool have all the critical features we need, especially in the area of test result validation and test suite management?
- ◆ **Reliability:** Does the tool work for long periods without failure, or is it full of bugs? Many test tools are developed by small companies that do a poor job of testing them.
- ◆ **Capacity:** Beyond the toy examples and demos, does the tool work without failure in an industrial environment? Can it handle large scale test suites that run for hours or days and involve thousands of scripts?

- ◆ **Learnability:** Can the tool be mastered in a short time? Are there training classes or books available to aid that process?
- ◆ **Operability:** Are the features of the tool cumbersome to use, or prone to user error?
- ◆ **Performance:** Is the tool quick enough to allow a substantial savings in test development and execution time versus hand testing.
- ◆ **Compatibility:** Does the tool work with the particular technology that we need to test?
- ◆ **Non-Intrusiveness:** How well does the tool simulate an actual user? Is the behavior of the software under test the same with automation as without?

Reckless Assumption #4:

An automated test is faster, because it needs no human intervention.

All automated test suites require human intervention, if only to diagnose the results and fix broken tests. It can also be surprisingly hard to make a complex test suite run without a hitch. Common culprits are changes to the software being tested, memory problems, file system problems, network glitches, and bugs in the test tool itself.

Reckless Assumption #5

Automation reduces human error.

Yes, some errors are reduced. Namely, the ones that humans make when they are asked carry out a long list of mundane mental and tactile activities. But other errors are amplified. Any bug that goes unnoticed when the master compare files are generated will go systematically unnoticed every time the suite is executed. Or an oversight during debugging could accidentally deactivate hundreds of tests. The dBase team at Borland once discovered that about 3,000 tests in their suite were hard-coded to report success, no matter what problems were actually in the product. To mitigate these problems, the automation should be tested or reviewed on a regular basis. Corresponding lapses in a hand testing strategy, on the other hand, are much easier to spot using basic test management documents, reports, and practices.

Reckless Assumption #6

We can quantify the costs and benefits of manual vs. automated testing.

The truth is, hand testing and automated testing are really two different processes, rather than two different ways to execute the same process. Their dynamics are different, and the bugs they tend to reveal are different. Therefore, direct comparison of them in terms of dollar cost or number of bugs found is meaningless. Besides, there are so many particulars and hidden factors involved in a genuine comparison that the best way to evaluate the issue is in the context of a series of real software projects. That's why I recommend treating test automation as one part of a multifaceted pursuit of an excellent test strategy, rather than an activity that dominates the process, or stands on its own.

Reckless Assumption #7

Automation will lead to "significant labor cost savings."

"Typically a company will pass the break-even point for labor costs after just two or three runs of an automated test." This loosey goosey estimate may have come from field data or from the fertile mind of a marketing wonk. In any case, it's a crock.

The cost of automated testing is comprised of several parts:

- ◆ The cost of developing the automation.
- ◆ The cost of operating the automated tests.
- ◆ The cost of maintaining the automation as the product changes.
- ◆ The cost of any other new tasks necessitated by the automation.

This must be weighed against the cost of any remaining manual testing, which will probably be quite a lot. In fact, I've never experienced automation that reduced the need for manual testing to such an extent that the manual testers ended up with less work to do.

How these costs work out depend on a lot of factors, including the technology being tested, the test tools used, the skill of the test developers, and the quality of the test suite.

Writing a single test script is not necessarily a lot of effort, but constructing a suitable test harness can take weeks or months. As can the process of deciding which tool to buy, which tests to automate, how to trace the automation to the rest of the test process, and of course, learning how to use the tool and then actually writing the test programs. A careful approach to this process (i.e. one that results in a useful product, rather than gobbledygook) often takes months of full-time effort, and longer if the automation developer is inexperienced with either the problem of test automation or the particulars of the tools and technology.

How about the ongoing maintenance cost? Most analyses of the cost of test automation completely ignore the special new tasks that must be done just because of the automation:

- ◆ Test cases must be documented carefully.
- ◆ The automation itself must be tested and documented.
- ◆ Each time the suite is executed someone must carefully pore over the results to tell the false negatives from real bugs.
- ◆ Radical changes in the product to be tested must be reviewed to evaluate their impact on the test suite, and new test code may have to be written to cope with them.
- ◆ If the test suite is shared, meetings must be held to coordinate the development, maintenance, and operation of the suite.
- ◆ The headache of porting the tests must be endured, if the product being tested is subsequently ported to a new platform, or even to a new version of the same platform. I know of many test suites that were blown away by hurricane Win95, and I'm sure many will also be wiped out by its sister storm, Windows 2000.

These new tasks make a significant dent in a tester's day. Most groups I've worked in that tested GUI software tried at one point or another to make all testers do part-time automation, and every group eventually abandoned that idea in favor of a dedicated automation engineer or team. Writing test code and performing interactive hand testing are such different activities that a person assigned to both duties will tend to focus on one to the exclusion of the other. Also, since automation development is software development, it requires a certain

amount of development talent. Some testers aren't up to it. One way or another, companies with a serious attitude about automation usually end up with full time staff to do it, and that must be figured in to the cost of the overall strategy.

Reckless Assumption #8

Automation will not harm the test project.

I've left for last the most thorny of all the problems that we face in pursuing an automation strategy: it's dangerous to automate something that we don't understand. If we don't get the test strategy clear before introducing automation, the result of test automation will be a large mass of test code that no one fully understands. As the original developers of the suite drift away to other assignments, and others take over maintenance, the suite gains a kind of citizenship in the test team. The maintainers are afraid to throw any old tests out, even if they look meaningless, because they might later turn out to be important. So, the suite continues to accrete new tests, becoming an increasingly mysterious oracle, like some old Himalayan guru or talking oak tree from a Disney movie. No one knows what the suite actually tests, or what it means for the product to "pass the test suite" and the bigger it gets, the less likely anyone will go to the trouble to find out.

This situation has happened to me personally (more than once, before I learned my lesson), and I have seen and heard of it happening to many other test managers. Most don't even realize that it's a problem, until one day a development manager asks what the test suite covers and what it doesn't, and no one is able to give an answer. Or one day, when it's needed most, the whole test system breaks down and there's no manual process to back it up. The irony of the situation is that an honest attempt to do testing more professionally can end up assuring that it's done blindly and ignorantly.

A manual testing strategy can suffer from confusion too, but when tests are created dynamically from a relatively small set of principles or documents, it's much easier to review and adjust the strategy. Manual testing is slower, yes, but much more flexible, and it can cope with the chaos of incomplete and changing products and specs.

A Sensible Approach to Automation

Despite the concerns raised in this article, I do believe in test automation. I am a test automation consultant, after all. Just as there can be quality software, there can be quality test automation. To create good test automation, though, we have to be careful. The path is strewn with pitfalls. Here are some key principles to keep in mind:

- ◆ Maintain a careful distinction between the automation and the process that it automates. The test process should be in a form that is convenient to review and that maps to the automation.
- ◆ Think of your automation as a baseline test suite to be used in conjunction with manual testing, rather than as a replacement for it.
- ◆ Carefully select your test tools. Gather experiences from other testers and organizations. Try evaluation versions of candidate tools before you buy.
- ◆ Put careful thought into buying or building a test management harness. A good test management system can really help make the suite more reviewable and maintainable.
- ◆ Assure that each execution of the test suite results in a status report that includes what tests passed and failed versus the actual bugs found. The report should also detail any work done to maintain or enhance the suite. I've found these reports to be indispensable source material for analyzing just how cost effective the automation is.
- ◆ Assure that the product is mature enough so that maintenance costs from constantly changing tests don't overwhelm any benefits provided.

One day, a few years ago, there was a blackout during a fierce evening storm, right in the middle of the unattended execution of the wonderful test suite that my team had created. When we arrived at work the next morning, we found that our suite had automatically rebooted itself, reset the network, picked up where it left off, and finished the testing. It took a lot of work to make our suite that bulletproof, and we were delighted. The thing is, we later found, during a review of test scripts in the suite, that out of about 450 tests, only about 18 of them were truly useful.

It's a long story how that came to pass (basically the wise oak tree scenario) but the upshot of it was that we had a test suite that could, with high reliability, discover nothing important about the software we were testing. I've told this story to other test managers who shrug it off. They don't think this could happen to them. Well, it *will* happen if the machinery of testing distracts you from the craft of testing.

Make no mistake. Automation is a great idea. To make it a good investment, as well, the secret is to think about testing first and automation second. If testing is a means to the end of understanding the quality of the software, automation is just a means to a means. You wouldn't know it from the advertisements, but it's only one of many strategies that support effective software testing.



James Bach (j.bach@computer.org, <http://www.jamesbach.com>) is an independent testing and software quality assurance consultant who cut his teeth as a programmer, tester, and SQA manager in Silicon Valley and the world of market-driven software development. He has worked at Apple, Borland, a couple of startups, and a couple of consulting companies. He currently edits and writes the Software Realities column in Computer magazine. Through his models of Good Enough quality, testcraft, exploratory testing, and heuristic test design, he focuses on demystifying software projects, and helping individual software testers answer the questions "What am I doing here? What should I do now?"