Seven Steps to Test Automation Success

Bret Pettichord
bret@pettichord.com
http://www.pettichord.com

Revised version of a paper originally presented at STAR West, San Jose, November 1999.
Version of 26 June 2001.

Abstract

Test automation raises our hopes yet often frustrates and disappoints us. Although automation promises to deliver us from a tough situation, implementing automated tests can create as many problems as it solves. The key is to follow the rules of software development when automating testing. This paper presents seven key steps: improve the testing process, define requirements, prove the concept, champion product testability, design for sustainability, plan for deployment, and face the challenges of success. Follow these steps as you staff, tool, or schedule your test automation project, and you will be well on your way to success.

A Fable

I've seen lots of different problems beset test automation efforts. I've worked at many software companies, big and small. And I've talked to a people from many other companies. This paper will present ways to avoid these problems. But first we need to understand them. Let me illustrate with a fable.

Once upon a time, we have a software project that needs test automation. Everyone on the team agrees that this is the thing to do. The manager of this project is Anita Delegate. She reviews the different test tools available, selects one and purchases several copies. She assigns one of her staff, Jerry Overworked, the job of automating the tests. Jerry has many other responsibilities, but between these, he tries out the new tool. He has trouble getting it to work with their product. The tool is complicated and hard to configure. He has to make several calls to the customer support line. He eventually realizes that they need an expert to set it up right and figure out what the problem is. After more phone calls, they finally send an expert. He arrives, figures out the problem and gets things working. Excellent. But many months have passed, and they still have no automation. Jerry refuses to work on the project any further, fearing that it will never be anything but a time sink.

Anita reassigns the project to Kevin Shorttimer, who has recently been hired to test the software. Kevin has a recent degree in computer science and is hoping to use this job as a step up to something more challenging and rewarding. Anita sends him to tool training so that he won't give up in frustration the way Jerry did. Kevin is very excited. The testing is repetitive and boring so he is glad to be automating instead. After a major release ships, he is allowed to work full time on test automation. He is eager for a chance to prove that he can write sophisticated code. He builds a testing library and designs some clever techniques that will support lots of tests. It takes longer than planned, but he gets it working. He uses the test

suite on new builds and is actually able to find bugs with it. Then Kevin gets an opportunity for a development position and moves on, leaving his automation behind.

Ahmed Hardluck gets the job of running Kevin's test suite. The sparse documentation he finds doesn't help much. It takes a while for Ahmed to figure out how to run the tests. He gets a lot of failures and isn't sure if he ran it right or not. The error messages aren't very helpful. He digs deeper. Some of the tests look like they were never finished. Others have special setup requirements. He updates the setup documentation. He plugs away with it. He finds that a couple failures are actually due to regression bugs. Everyone is happy that the test suite caught these. He identifies things in the test suites that he'd like to change to make it more reliable, but there never seems to be the time. The next release of the product has some major changes planned. Ahmed soon realizes that the product changes break the automation. Most of the tests fail. Ahmed works on this for a while and then gets some help from others. They realize that it's going to take some major work to get the tests to run with the new product interface. But eventually they do it. The tests pass, and they ship the product. And the customers start calling right away. The software doesn't work. They come to realize that they reworked some tests so that error messages were being ignored. These tests actually failed, but a programming error had dismissed these errors. The product is a failure.

That's my fable. Perhaps parts of the story sound familiar to you. But I hope you haven't seen a similar ending. This paper will suggest some ways to avoid the same fate. (James Bach has recounted similar stories of test automation projects [Bach 1996].)

The Problems

This fable illustrates several problems that plague test automation projects:

Spare time test automation. People are allowed to work on test automation on their own time or as a back burner project when the test schedule allows. This keeps it from getting the time and focus it needs.

Lack of clear goals. There are many good reasons for doing test automation. It can save time, make testing easier and improve the testing coverage. It can also help keep testers motivated. But it's not likely to do all these things at the same time. Different parties typically have different hopes. These need to be stated, or else disappointment is likely.

Lack of experience. Junior programmers trying to test their limits often tackle test automation projects. The results are often difficult to maintain.

High turnover. Test automation can take a while to learn. But when the turnover is high, you lose this experience.

Reaction to desperation. Problems are usually lurking in the software long before testing begins. But testing brings them to light. Testing is difficult enough in itself. When testing is followed by testing and retesting of the repaired software, people can get worn down. Will the testing ever end? This desperation can become particularly acute when the schedule has dictated that the software should be ready now. If only it weren't for all the testing! In this environment, test automation may be a ready answer, but it may not be the best. It can be more of a wish than a realistic proposal.

Reluctance to think about testing. Many find automating a product more interesting than testing it. Some automation projects provide convenient cover stories for why their contributors aren't more involved in the testing. Rarely does the outcome contribute much to the test effort.

Technology focus. How the software can be automated is a technologically interesting problem. But this can lose sight of whether the result meets the testing need.

Follow the Rules of Software Development

You may be familiar with the five step maturity models that can be used to classify software development organizations. The Capabilities Maturity Model from the Software Engineering Institute is a well known example. Jerry Weinberg has his own organizational model, in which he adds an additional level which he calls Pattern Zero. A Pattern Zero organization is *oblivious* to the fact that it is actually developing software; there is no distinction between the users and the developers [Weinberg 1992]. This is the place that test automation often finds itself. Thus, dedicating resources to test automation and treating it like a development activity elevates it to the first level. This is the core of the solution to the problems of test automation. We need to run test automation projects just as we do our other software development projects. Like other software development projects, we will need to have developers dedicated to developing our test automation. Like other software development projects, test automation automates a task in which the programmer is probably not an expert. Therefore, expert testers should be consulted and should provide the requirements. Like other software development projects, test automation benefits if we design our approach before we start coding. Like other software development projects, test automation code needs to be tracked and safeguarded. Therefore, we need to use source code management. Like other software development projects, test automation will have bugs. Therefore, we need to plan to track them and test for them. Like other software development projects, users will need to know how to use it. Therefore, we need user documentation.

It's not my place to tell you how to develop software. I assume that you are part of a software organization that already has some idea as to what reasonable and effective methods should be used for developing software. I am simply urging you to abide by whatever rules are established for software development in your own test automation. This paper will be organized by the normal steps that we all use for our software development projects, making special notes of the considerations and challenges that are particular to test automation.

1. Improve the Testing Process
2. Define Requirements
3. Prove the Concept
4. Champion Product Testability
5. Design for Sustainability
6. Plan for Deployment
7. Face the Challenges of Success

Step 1. Improve the Testing Process

If you are responsible for improving the efficiency of a business task, you first want to make sure that the process is well defined. You also want to see if there are simple and cheap ways to make things go easier before you invest the time and money in automating the system using computers. The same, of course, holds for test automation. Indeed, I like to think that the term "test automation" refers to anything that streamlines the testing process, allowing things to move along more quickly and with less delay. Automated test scripts running on a machine are just one alternative.

For example, many teams start by automating their regression tests. These are the tests that are frequently run and rerun, checking to make sure that things that used to work aren't broken by new changes. They are run often and are tedious. How well are your regression tests documented? It is common to use lists of features that are to be checked. This is a great start. A reminder of what you need to test suits someone who knows the product and understands the test approaches that need to be used.

But, before you start to automate, you'll need to improve this documentation. Make your test approach explicit. Specify what names and data should be used for the tests or provide guidelines for making them up. It is probably safe to assume that the tester has basic product knowledge. This is surely documented elsewhere. But you need to be specific about the details of the test design. You also need to state the expected results. This is often unstated, suggesting that the tester should know. Too many testers don't realize what they are missing or are too embarrassed to ask. This kind of detailed documentation is going to be an immediate benefit to your team, because now anyone who has a basic understanding of the product can execute the tests. It is also going to need to be done before you do a more thorough automation of the tests. Your test design is going to be the primary requirements statement for your automation, so it's important that it be explicit. It's possible to go overboard here and spell out every step that needs to be taken to execute the test. It is safe to presume that someone who understands how to operate the software will execute the tests. But don't assume that they understand your ideas on how it should be tested. Spell these out.

I once had the job of automating tests for a software module. This module had some features that made it hard to automate. When I realized that I wasn't going to be finished in a short amount of time, I decided I needed a detailed regression test design. I went through the closed defects for the module and for each one I wrote a description of a test that would have been able to find the defect. My plan was that this would provide me with a detailed list of automation requirements that would help me decide what parts of the module most needed automation support. Well, I never got a chance to write the automation. But when we needed to run a full regression on the module, we were able to give the test specifications to a couple people who knew the product but had no experience testing it. Using the detailed test descriptions, they were able to go off and test independently. They found bugs. This required almost no supervision. In a way, it was great automation. Indeed, on this project we had better luck handing off these documented test cases, than we did the automated test scripts we had for other product modules. We learned that the automated scripts required too much training for others to just pick them up and run them. If the automated tests were better designed, this wouldn't have been a problem, but we found that it was much easier to create well designed test *documentation* than it was to create well designed test *automation.*

Another easy way to improve the efficiency of the testing is to get more computers. Many testers can easily keep a couple of machines busy. This is an obvious point, but I make it because I've seen some misguided automation attempts that resulted from trying too hard to

maximize the testing done on a single machine. Test automation can be an expensive and risky way of dealing with an equipment shortage. Better, would be to focus on making a case for the equipment you need.

My final suggestion for improving the testing process is to improve the product to make it easier to test. There are many improvements that will help both the users and the testers. Later I'll discuss testability needs for automation. Here I want to suggest identifying product improvements that will help manual testing.

Some products are hard to install, and testers find themselves spending lots of time installing and reinstalling. Rather than automating the install process, maybe it would be better to improve the install program. That way the customer gets the benefit too. Another way of putting this is to consider developing your automation in a form that can be delivered with the product. Indeed there are many commercial tools available that are specifically designed to create install programs.

Another product improvement can be to utilize tools for scanning install or execution logs for errors. Visually scanning through pages and pages of logs looking for error messages gets tedious very quickly. So let's automate it, right? Writing a scanning tool is easy if you know exactly what form the error messages will take. But if you aren't sure, you are setting yourself up for disaster. Remember the fable about the test suite that missed the failures? Customers don't want to scan through logs looking for errors, either. Adding an error scanner to the product will likely result in a more reliable scanner, possibly requiring modifications to the error logging system itself to ensure all errors are caught. This is a tool your testing can depend on.

Performance is another area where product improvement can help the testing. Surely this is obvious. If product sluggishness is delaying your testing, identify the slow functionality, measure it, and report it as a defect that's blocking testing.

These are some of the things you can do to improve test efficiency without having to build a test automation system. Improving the test process may buy you some time for test automation and will certainly make your automation project go more smoothly.

Step 2. Define Requirements

In our fable, we saw that automators can have different goals than the sponsors. To avoid this situation, we'll need to make sure we have agreement on the requirements for test automation. We'll have test requirements, which will describe what needs to be tested. These will be detailed in our test designs. And we will have automation requirements, which will describe the goals for automation. Too many people think test automation is obviously the right thing and don't bother to state what they hope to get. There are several reasons why people choose to automate tests:

- Speed up testing to accelerate releases
- Allow testing to happen more frequently
- Reduce costs of testing by reducing manual labor
- Improve test coverage
- Ensure consistency

- Improve the reliability of testing
- Allow testing to be done by staff with less skill
- Define the testing process and reduce dependence on the few who know it
- Make testing more interesting
- Develop programming skills

Goals will often differ between development management, test management, the testers themselves and whoever is automating the tests. Clearly success will be elusive unless they all come to some agreement.

Of course, some of these automation goals will be easier to meet than others. Test automation often actually increases the required skill level for testers, as they must be able to understand the automated tests sufficiently that they can reproduce the defects found. And automation can be a frustrating means of extracting test knowledge from your staff. Regardless, be clear in advance on what people will count as success.

Manual testers do a lot of things that can go unnoticed when they run tests. They plan and obtain required resources. They setup and execute tests. They notice if anything unusual happens. They compare test results. They log results and reset the system to get ready for the next test. They analyze failures and investigate curious behavior. They look for patterns of failure and devise and execute additional tests to help locate defects. And then they log defect reports in order to get fixed and summary reports so that others can know what's been covered.

Don't feel compelled to automate every part of the tests. Look for where you are going to get the biggest payback. Partial automation is OK. You may find it's best to automate the execution and leave the verification to be done manually. Or you may choose to automate the verification and leave the execution to be done manually. I've heard some people say that it's not real automation unless it does everything. That's hogwash. If you are simply looking for challenge, then you can try to do it all. But if you are looking for success, focus on where you can quickly get automation that you can use again and again.

Defining requirements for your test automation project will force these various tradeoffs to be made explicit. It will also help set different party's expectations reasonably. By defining your goals, you have taken another step towards test automation success.

Step 3. Prove the Concept

In our fable, we saw that the automators dived into the automation project without knowing for sure where they were headed. But they also got mixed support for their project.

You may not realize it, but you have to prove the feasibility of your test automation project. It always takes longer than people would like. To get the commitment it needs, you'll need the support of various people in your organization.

Many years ago, I worked on a test automation project where we had all kinds of great ideas. We designed a complex testing system and worked hard on many of its components. We periodically gave presentations describing our ideas and the progress we were making. We

even demonstrated the pieces we had working. But what we didn't do was demonstrate actual tests. Eventually the project was cancelled. It's a mistake I haven't repeated since.

You'll need to validate your tools and approach as soon as possible. Is it even possible to automate tests for your product? It's often difficult. You need to find the answer to this question as soon as possible. You need to find out whether your tools and approach will work for your product and staff. What you need is a proof of concept – a quick, meaningful test suite that demonstrates that you are on the right track. Your proof-of-concept test suite will also be an excellent way to evaluate a test tool.

For many people, test automation means GUI test automation. That's not my meaning. I've done both GUI and non-GUI automation and I've been surprised to learn that most of the planning concerns are shared by both. But GUI test tools are much more expensive and finicky. It is hard to predict what difficulties they will encounter. Consequently, choosing the right GUI test tool is an important decision. Elisabeth Hendrickson has provided excellent guidelines for selecting one [Hendrickson 1999]. I can suggest that your proof-of-concept will be an important part of your evaluation. This will require at least a one-month trial license for the tool. You may even want to purchase one copy now and wait to buy additional copies until after the evaluation. You want to find the tool problems before you've shelled out the big bucks. You'll get better help from the vendor, and you won't feel trapped if you find you have to switch to a different tool.

Here are some candidates for your proof of concept:

Regression testing. Do you have tests you run on every build? These kinds of tests are excellent candidates for automation.

Configuration tests. How many different platforms does your software support? And are you expected to test them all? Some automation may help.

Test bed setup. The same setup procedures may be used for lots of different tests. Before automating the tests, automate the setup.

Non-GUI testing. It's almost always easier to automate command line and API tests than it is to automate GUIs.

Whatever your approach, define a demonstrable goal and then focus on it. Proving your test automation concept will move you one step further on the road to success.

Step 4. Champion Product Testability

Three different interfaces a product might have are command line interfaces (CLIs), application programming interfaces (APIs), and graphical user interfaces (GUIs). Some may have all three, but many will have only one or two. These are the interfaces that are available to you for your testing. By their nature, APIs and command line interfaces are easier to automate than GUIs. Find out if your product has either one; sometimes these are hidden or meant for internal use only. If not, championing product testability may require you to encourage your developers to include a CLI or API in your product.

But first, let me talk a little more about GUI test automation. There are several reasons why GUI test automation is more difficult than people often realize. The first reason is that GUI test automation requires some manual script writing. Most GUI automation tools have a feature called ‘record and playback’ or, ‘capture replay’. The idea is great. You execute the test manually while the test tool sits in the background and remembers what you do. It then generates a script that you can run to re-execute the test. It's a great idea that rarely works. Many authors have concluded that although usable for learning and generating small bits of code, various problems prevent recorders from being effectively used for complete test generation [Bach 1996, Pettichord 1996, Kaner 1997, Linz 1998, Hendrickson 1999, Kit 1999, Thomson 1999, Groder 1999]. As a result, you will need to create your GUI tests primarily by hand.

A second reason for the difficulty of GUI test automation regards the technical challenge of getting the tool to work with your product. It often takes considerable expertise to get GUI test tools to work with the latest user interface technologies. This difficulty is also one of the main reasons why GUI test tools are so expensive. They have a hard job. Non-standard or custom controls can present added difficulties. Solutions can usually be found, but often require modifications to the product source code or updates from the tool vendor. Bugs in the test tool may require analysis and patches or workarounds. The test tool may also require considerable customization to make it work effectively with customized elements of your product interface. The difficulty of this work often comes as a surprise. You also may find yourself redesigning your tests to avoid difficult controls.

A third complication for GUI test automation involves keeping up with design changes made to a GUI. GUIs are notorious for being modified and redesigned throughout the development process. It is often a very good idea, as the first version of the GUI can be awful. But keeping the automation running while the GUI keeps changing can feel like running in place. You can spend a lot of time revising your tests to match the changing interface. Yet, you don't want to be in the position of arguing against helpful improvements. I've been in this situation and it is mighty uncomfortable to be suggesting that improvements be withheld from the product just so that the tests can keep running. Programmable interfaces tend to exhibit less volatility after the original design has been worked through.

These are reasons not to depend on GUI test automation as the basis for testing your product functionality. The GUI still needs to be tested, of course, and you may choose to automate these tests. But you should have additional tests you can depend on to test core product functionality that will not break when the GUI is redesigned. These tests will need to work through a different interface: a command line or API. I’ve seen people choose GUI test automation because they didn't want to have to modify the product. But they eventually learned that product modification was necessary to get the GUI test automation to work. Automation is likely to require product modification whichever way you go. So demand a programmable interface that will be dependable.

To make it easier to test an API, you may want to bind it to an interpreter, such as TCL or Perl or even Python. This enables interactive testing and should also speed up the development cycle for your automated tests. Working with API’s may also allow you to automate unit tests for individual product components.

An example of a possibly hidden programmable interface regards InstallShield, a popular tool for developing install programs. InstallShield has command line options that enable what's

called a silent install. This allows install options to be read from a response file you've created in advance. Using this is likely to be easier and more dependable than automating the InstallShield GUI itself.

Another example of how you could avoid GUI automation relates to Web-based software. GUI tools are available to manipulate Web interfaces through a browser. But it can be easier to directly test the HTTP protocol that Web browsers use to communicate to Web servers. Perl is but one language tool that can directly connect to a TCP/IP port, enabling this kind of automation. Applications using advanced interface technology, such as client-side Java or ActiveX won't be able to take advantage of this kind of approach. But when this approach is suitable, you may find that your automation is cheaper and easier than working through a GUI.

I was once hired to write automated tests for a product GUI. The product also had a command line interface, for which they already had automated tests. After some investigation I learned that it wasn't hard to find GUI bugs, but that the customers didn't care much as they were happy using the CLI. I also learned that we had no automation for the latest features (which could be accessed from either the GUI or the CLI). I decided to put off the GUI test automation and extended the CLI test suite to cover the latest features. Looking back, I sometimes think of this GUI test automation project I chose not to do was one of my bigger successes, because of all the time and effort that could have been wasted on it. They were all ready for the GUI automation; they had bought a tool and everything. But I know it would have faced various difficult obstacles, while providing extremely limited value.

Whether you need support for GUI, CLI, or API automation, you are going to be much more successful in getting your testability features designed right into the product if you ask early, while the product is still being designed. Enlightened developers will realize that testability is a product requirement. Getting an early start on your test automation project puts you on the road to success.

Step 5. Design for Sustainability

We saw in our fable that test automation efforts are prone to being dropped. This happens when automators focus on just getting the automation to work. Success requires a more long-term focus. It needs to be maintained and expanded so that it remains functional and relevant as new releases of your product are developed. Concern for the future is an important part of design. The integrity of the tests is also paramount. People must trust that when the automation reports a test as passed, it actually did. I have seen far too many cases where parts of tests were silently skipped over or where errors failed to be logged. This is the worst kind of automation failure. It's the kind of failure that can lead to disaster for the whole project. Yet, it can happen when people build test automation that is poorly designed or carelessly modified. This can often happen as a result of a misguided focus on peformance or ease of analysis.

Performance. Improving code performance often increases its complexity, thus threatening its reliability. This is a particular concern with test automation because rarely is much attention placed on testing the automation itself. My analysis of test suite performance has also shown that many test suites spend most of their time waiting for the product. This places a limit on how much the test execution can be sped up without improving the performance of

the product. I suspect that the concern I've seen amongst test automators with performance stems from overemphasis of this characteristic in computer science curriculums. If test suite performance is a genuine concern, get more hardware or reduce the number of tests in your test suite. They often contain a fair amount of redundancy.

Ease of Analysis. A common bugbear is what to do when automated tests fail. Failure analysis is often difficult. Was this a false alarm or not? Did the test fail because of a flaw in the test suite, a mistake in setting up for the tests, or an actual defect in the product? I see several ways to aid failure analysis, one of which can lead to problems. You could improve the reliability of the test suite by having it explicitly check for common setup mistakes before running tests. You could improve the serviceability of the test suite by improving its error reporting. You could repair known problems in your test harness. You could train people on how to analyze failures. You might even be able to find unreliable tests that can be safely removed because they are redundant or test obsolete functionality. These are all positive ways of reducing false alarms or improving test analysis. A mistaken approach would be to build a results post-processor that conducted its own analysis and filtered the information. Although this approach can be made to work, it complicates the test automation system. Moreover, bugs in the post-processing could seriously impair the integrity of the tests. What happens if it dismisses or mischaracterizes bona fide failures? I've seen this approach taken a couple times by groups wary of modifying the test suites and reluctant to conduct training. This misguided tactic can be very appealing to managers looking for testing that occurs at the push of a button. Resist suggestions to hide the complexity of your tests.

That said, let's focus on what it takes to make a sustainable test suite. It takes reviewability, maintainability, integrity, independence and repeatability.

Reviewability. A common situation is to have an old test suite that's been around for years. It was built before the current staff were on the project. We could call this a wise oak tree [Bach 1996]. People depend on it, but don't quite know what it does. Indeed, it invariably turns out that the test suite is rather modest in its accomplishments, but has attained oracular status with time. These wise oak test suites suffer from poor reviewability. It is hard to determine what they actually test, and people tend to overestimate their abilities. It's critical that people be able to review a test suite and understand what is being tested. Good documentation is one way to achieve this. Code coverage analysis is another. I used a variation of this on one project. I instrumented the test suite to log all product commands. We then analyzed the logs to determine which commands were being exercised and with what options. It provided a nice summary of what was and wasn't being covered by the tests. Without reviewability, it's easy to become overly dependent on a test suite you don't really understand. You can easily start thinking that it is doing more than it really is. Being able to review your test suite also facilitates peer review.

Maintainability. I once worked with a test suite that stored all the program output to files. These output files would then be compared to previously generated output files, termed "gold files." The idea was that this would be a good way to detect any regression bugs. But this approach was so sensitive, it generated many false alarms. The problem was that with time, the developers intentionally made small changes to many of the output messages. A test failure would result whenever one of these changed messages appeared in the test output. Clearly the gold files needed to be updated, but this required lots of analysis. A more maintainable approach would only select specific product outputs to be checked. Rather than comparing all the output, the tests could just compare the output relating to the specific

features being tested. Product interfaces can also change and prevent old tests from running. I mentioned that this is a particular challenge for GUI automation. Building an abstraction barrier to minimize the changes to your tests due to product interface changes is a common approach for addressing this problem. This can take the form of a library used by all the tests. Product changes will then only require that this library be updated to make the tests current.

Integrity. When your automation reports that a test passed, did it really? This is what I call test suite integrity. In our fable, we saw a dramatic example of what could happen when due attention isn't given to the integrity of the tests. How well can you depend on its results? False alarms can be a big part of this problem. People hate it when test suites fail and it just turns out to be just a problem with the tests or the setup. It's hard to do much about false alarms. You want your tests to be sensitive and report a failure if things don't look right. Some test harnesses help by supporting a special test result for when the test isn't setup right to run. They have PASS, FAIL and a third result called something like NOTRUN or UNRESOLVED. Whatever you call it, it's handy to be able to easily sort the tests that were blocked from the tests than ran but failed. Getting the correct result is part of integrity. The other part is making sure that the right test was run. I once found a bug in a test dispatcher that caused it to skip parts of some tests. No errors were generated. I stumbled across this bug while reviewing the code. Had I not noticed this, I imagine that we could have been running partial tests for a long time before we realized something was wrong with the automation.

Independence. Automation cannot truly replicate manual tests. A written manual test procedure assumes you have an intelligent, thinking, observant human being running the tests. With automation, a dumb computer will be running the tests instead. You have to tell it what a failure looks like. And you have to tell it how to recover so that it can keep running the test suite. Automated tests need to be able to be run as part of a suite or individually. The only way to reliably implement this is to make tests independent. Each test needs to setup its test environment. Manual regression tests are usually documented so that each test picks up after the preceding test, taking advantage of any objects or records that may already have been created. Manual testers can usually figure out what is going on. A common mistake is to use the same approach with automated tests. This results in a "domino" test suite. A failure in one test, will topple successive tests. Moreover, these tests also cannot be run individually. This makes it difficult to use the automated test to help analyze legitimate failures. When this happens, people will start questioning the value of automated tests in the first place. Independence requires adding repetition and redundancy to the tests. Independent tests will be convenient for developers to use when diagnosing reported defects. It may seem inefficient to structure tests this way, but the important thing is to maintain independence without sacrificing reliability. If the tests can be run unattended, efficiency becomes less of a concern.

Repeatability. There's not much that can be done with a failure report that only hits an error intermittently. So, you need to make sure that your tests work the same way every time they are run. This principle indicts careless use of random data. Random numbers built into common language libraries often hide the initialization process. Using this can make your tests run differently each time. This can frustrate failure analysis. There are two ways of using random number generators to avoid this. One would be to use a constant value to initialize the random number generator. If you wanted to generate a variety of tests, you could set this up to vary in a predictable and controlled way. The other technique would be to generate your random test data ahead of time in a file or database. This is then fed to your test procedure. This may seem obvious enough, but I've seen too many violations of this

principle. Let me explain what I've seen. When you execute tests manually, you often make up names for files and records on the fly. What do you do when you automate this test? One approach would be to define a fixed name for the records in the test. If they are going to persist after the test completes, you'll need to use a naming convention to ensure that different tests don't collide. This is usually the wise thing to do. However, I've seen several cases where the tests randomly generated the names for the records. Unfortunately, this turned out to be an unreliable way of avoiding name collisions that also impaired the repeatability of the tests. The automators had apparently underestimated the likelihood of a name collision. In two cases, four digit numbers were used as random elements of record names. Some basic probability calculations show that it only takes 46 of such records to generate a 10% chance of a name collision. With 118 records, the odds go up to 50%. I suspect that these tests used random names in a lazy attempt to avoid having to write code to clean out old test records before rerunning the tests. But this only introduced problems that damaged the reliability and integrity of the tests.

Placing a priority on these design concerns will help ensure that your automated test suite will continue to be usable for the life of the product it tests.

Let me now turn to discussing a few test automation architectures that have been used to support these design goals:

Libraries. A common strategy is to develop libraries of testing functions that can be used in lots of different tests. I've reviewed a lot of these libraries and have written my own. These can be particularly helpful when they allow tests to be insulated from product interface design changes. But my general observation is that these tend to be overdeveloped. The libraries are overly ambitious in what they cover and are under-designed. They are often poorly documented and tests that use them can be hard to follow. When problems are later found, it is hard to tell whether the error lies in the function or its usage. Because of their complexity, maintainers can be reluctant to modify them even when they look broken. The obvious conclusion is to make sure your libraries are not poorly designed. But the practical conclusion is to realize that test automation often doesn't get the luxury of having well-designed libraries. I often find that open-coding is a better option than using under-designed libraries. I've also seen too many libraries that included functions that were unused or only used once. This squares with the Extreme Programming principle, "You're not going to need it." [Jeffries 1997] This may result in some duplication of code between test cases, but I've found that small variations may still exist that are difficult to handle elegantly with library functions. You want to have some variety amongst your test cases and open coding makes this easier to do. If I have several tests that do some of the same things, I use cut and paste to duplicate my code. Some people think that this practice is heresy. Oh well. It allows me to modify the common code as needed, and I don't have to try and guess how I'm likely to reuse code ahead of time. I think my tests are easier to read, because the reader doesn't have to know the semantics of some library. Another advantage of this approach is that it is easier for others to understand and extend the test suite. Rather than writing from scratch, most programmers find code that does something similar to what they want to do and then modify it. This is an excellent approach for writing test suites that open coding actually encourages. I do like to write small libraries of functions I'll be using again and again. These need to be conceptually well defined and well documented, especially with regard to start and end states. And I test them thoroughly before I use them in my tests. This matter is, of course, all a matter of balance. But don't plan a large testing library with the hope that hordes of test automators will come someday to write lots of tests. They aren't coming.

Data-Driven Tests. A technique that is becoming widely discussed allows tests to be written in a simplified table format. This is know variously as table-driven, data-driven or even "third generation" automation. It requires that a parser be written to interpret and execute test statements. One of the primary benefits of this architecture is that it allows tests to be specified in a format that is easy to write and review. It's well suited for test teams with domain experts who may not be strong programmers. However, a data-driven test parser is basically a test library with a small language on top. Thus, the comments I made about test libraries apply here as well. There is also the difficulty of designing, developing and testing a small language for the tests. Invariably, these languages grow. A tester decides that they want to use the output from the first part of a test as input to a second part. Thus variables are added. Then someone decides that they want to repeat something a hundred times. So loops are added to the language. You can end up with yet another language. If it looks like you are headed down this route, it's probably better to hook in a publicly available language like Perl, Python or TCL than to try and design your own.

Heuristic Verification. I've seen some test automation with no real results verification. This resulted from the difficulty of doing complete verification and the fact that the test design specifications failed to indicate expected results. Clearly, this is unfortunate. It's OK to depend on manual log verification, but this needs to be advertised. When I write tests that depend on external verification, I place a note to this effect in the execution logs. Gold files are another approach for results verification. The program output is captured, reviewed manually, and then archived as "gold". Later, results are then compared against it. The problem with this is that many of the differences will be due to changes in time, configuration, or product messages that are not indicative of problems. It leads to many false alarms. The best approach for verification is to look at specified portions of the output or results and then to make reasonable comparisons. Sometimes it is hard to know in advance what a correct result looks like, but you know what a wrong one looks like. Developing useful heuristics can be very helpful. I suspect that some people are reluctant to develop anything short of comprehensive results verification because of a fear that tests will be faulted if they let anything slip by. But of course we always make tradeoffs when we test, and we always need to face the risk we may have missed something. Automation doesn't change this. Automators who aren't used to making these kinds of tradeoffs need to have someone available to consult with on verification strategies. Creativity is often required as well. Many techniques are available that can find defects without raising false alarms.

By focussing on design goals of long-term sustainability and choosing an appropriate architecture, you will be moving along on the road to success.

Step 6. Plan for Deployment

In our fable, we saw some of the problems that can occur when automators defer packaging test suites for others to use. It doesn't happen, and then the next person needing to run the tests has to reverse engineer them to figure out how they should work.

As the automator, you know how to run the tests and analyze the failures. But to really get the payoff from test automation, the tests need to be packaged so that other people can use them. This will mean documenting the setup and maybe even making the test suite easier to install and run. Make sure you give helpful error messages in situations where the resources necessary for testing are unavailable.

Think of your test suite as a product. You'll have to test it, and make sure it doesn't depend on any special libraries or services you have installed on your machine.

Get someone else to use your test suite as soon as it's ready. Make sure that it does the testing in a way they think is appropriate. And make sure they understand the test results and can analyze failures. Some training and documentation may be in order.

As a manager, you want a chance to identify and remedy any major issues with the test suite before the automator moves on. Sooner or later they will, and then you won't have time to address this issue. If you don't address it, you risk owning another abandoned test suite.

A good test suite has many uses. Clearly it can be used to test new versions of the product. It can also be handy to assist with certifying your product on new platforms. Having a test suite that is easy to run can support a nightly build process or even one whereby developers are expected to run standard tests on their code before they check it in.

It can be hard to foresee which people might want to use your test suite. So make it widely available to the entire product team. Making it downloadable from an internal Web site is often ideal. People shouldn't have to talk to several people just to find out how to obtain a copy of the test suite. And too many test suites are kept secret because their owner doesn't think they are "ready." Finish the test suite and move on. It doesn't have to be perfect.

Planning for deployment and making your tests widely available sets you on the path to successful test automation that will be used again and again.

Step 7. Face the Challenges of Success

You’re done. Your test suite is documented and delivered. People understand the tests and how to run them. The tests are used regularly as your product is developed and maintained. Your success now brings additional challenges. Although you have certainly made some things easier, automation invariably complicates the testing process. Staff will need to learn how to diagnose failures found by the automated tests. If this doesn’t happen, the people running the tests are apt to presume that failures are automation problems and the automators will be called in to help diagnose every run. Developers are also prone to suspecting automation code they are unfamiliar with. So testers will need to learn how to reproduce failures either manually or with minimal test scripts.

The work with the test suites is not over. Tests will need to be added to improve the coverage or to test new features. Old tests will need to be repaired if broken or removed if redundant. Tests themselves may need to be ported to newly supported platforms. Ensuring that the test suites improve over time can be difficult. One idea is to plan a formal review of the test suites after each major release. If you already conduct a post-mortem as part of your process, make sure you include time to identify weaknesses of the test suite and then carry out the required improvements. Don't let the “old oak syndrome” set in. Just because a test suite has been around for a while doesn't mean it doesn't have blind spots.

With time, your tests are likely to stop finding problems. Developers will learn the tests and discover how to pass the tests the first time through. This phenomenon is known as the pesticide paradox. Your developers have improved their design and practices to the point

where they are no longer making the kinds of errors that your tests are designed to detect. Some people may doubt whether your tests are still working correctly. You may need to assess whether it is time to raise the bar.

Previously I mentioned the fantasy in which all the testing is done at the push of a button. I don't think that can ever really happen. There will always be a role for manual testing. For one, it is the only real way to sanity-test your automation itself.

The other reason for manual tests is that there will always be tests that are justified by the specific circumstances motivating the tests. This testing is often best done in an exploratory manner. And it's hard to say in advance that tests are worth repeating. There is always a cost involved. Don't fall into the classic error of trying to automate everything. [Marick 1997]

I've been urging you to maintain your investment in test automation. But a particular challenge lies in the timing for when test automation should be done. Test automation must be released to the testing staff in time for it to be useful. It's nice to release some automation early, but there comes a point where no new automation can be used, except for requested fixes. When the test effort is in full swing, testers can't afford to spend any time learning new tools or diagnosing tool errors. Identify this date in the project plan and let everyone know that you plan to meet this milestone. But after this date has been met, what should automators do? The focus on delivering the current release of the product may pull the automators into helping the test effort and executing tests. But once the testers know how to use the automation, it's a good time for automators to get a jump on the next release and improve the test tools and libraries. This is often the time when some developers start designing features for the next product release. The test automation is well served if the design work for new automation features also begins now. The idea is to keep the automators synchronized with the development cycle, rather than the testing cycle. Not doing this will result in fewer opportunities for improvement to the test automation. Explain the benefits of this schedule to the testers so they don't resent the fact that the automators aren't focussing on the release that is approaching its ship date.

Continuing to invest in automation will help you face the challenges of success and ensure that the road remains clear for continuing success as test automation becomes a dependable basis for your testing process.

Acknowledgements

Earlier versions of this paper were presented at the STAR West conference in San Jose, California, November 1999; Lucent Technologie's Automated Software Testing conference in Naperville, Illinois, November 1999; the Practical Software Quality Techniques conference in Austin, Texas, March 2000; and the Rational User's Conference in Philadelphia, Pennsylvania, August 2000.

Carol Schiraldi, Noel Nyman, Brian Marick and James Bach provided helpful comments on early drafts.

References

Bach, James. 1996. “Test Automation Snake Oil.” *Windows Technical Journal* , (October): 40-44. http://www.satisfice.com/articles/test_automation_snake_oil.pdf

Dustin, Elfriede. 1999. “Lessons in Test Automation.” *Software Testing and Quality Engineering* (September): 16-22. http://www.stickyminds.com/sitewide.asp?ObjectId=1802&ObjectType=ART&Function=edetail

Fewster, Mark and Dorothy Graham. 1999. *Software Test Automation*, Addison-Wesley.

Groder, Chip. “Building Maintainable GUI Tests” in [Fewster 1999].

Kit, Edward. 1999. “Integrated, Effective Test Design and Automation.” *Software Development* (February). http://www.sdmagazine.com/articles/1999/9902/9902b/9902b.htm

Hancock, Jim. 1997 “When to Automate Testing.” *Testers Network* (June). http://www.data-dimensions.com/Testers'Network/jimauto1.htm

Hendrickson, Elisabeth. 1999. “Making the Right Choice: The Features you Need in a GUI Test Automation Tool.” *Software Testing and Quality Engineering Magazine* (May): 21-25. http://www.qualitytree.com/feature/mtrc.pdf

Hoffman, Douglas. 1999. “Heuristic Test Oracles: The Balance Between Exhaustive Comparison and No Comparison at All.” *Software Testing and Quality Engineering Magazine* (March): 29-32.

Kaner, Cem. 1997. “Improving the Maintainability of Automated Test Suites.” Presented at Quality Week. http://www.kaner.com/lawst1.htm

Linz, Tilo and Matthias Daigl. 1998. “How to Automate Testing of Graphical User Interfaces.” European Systems and Software Initiative Project No. 24306 (June).http://www.imbus.de/html/GUI/AQUIS-full_paper-1.3.html

Jeffries, Ronald E., 1997, “XPractices,” http://www.XProgramming.com/Practices/xpractices.htm

Marick, Brian. 1998. “When Should a Test Be Automated?” Presented at Quality Week. http://www.testing.com/writings/automate.pdf

Marick, Brian. 1997. “Classic Testing Mistakes.” Presented at STAR. http://www.testing.com/writings/classic/mistakes.html

Pettichord, Bret. 1996. “Success with Test Automation.” Presented at Quality Week (May). http://www.io.com/~wazmo/succpap.htm

Thomson, Jim. “A Test Automation Journey” in [Fewster 1999]

Weinberg, Gerald M. 1992. *Quality Software Management: Systems Thinking*. Vol 1. Dorset House.

Comment [KB1]:

About the Author

Bret Pettichord is a consultant specializing in software testing, test automation and testability. Familiar with many test tools, he helps teams develop test automation strategies and architectures. He has helped develop automated tests for such companies as Texas Instruments, Rational, Netpliance, Whisperwire, Managemark, Tivoli, Unison, BMC, Segue and Interleaf. He also provides training in automated testing architectures and design.

Bret is the editor of the Software Testing Hotlist (http://www.io.com/~wazmo/qa/) and is a frequent speaker. He is the founder of the Austin Workshop on Test Automation and a founding participant in the Los Altos Workshop for Software Testing. He sits on the advisory board of Stickyminds.com and is certified in software quality engineering by the American Society of Quality*. He is also a member of the IEEE Computer Society. He has a bachelor's degree in philosophy and mathematics from New College, Sarasota, Florida.

*Not licensed to practice engineering by the state of Texas.