Realistic Expectations of Test Automation

Mark Broihier
October 18, 2016
Revision 2

Table of Contents

1 Introduction

I've been doing successful test automation for over thirty years and the most common problem I've run into is the difficulty of aligning people's expectation of test automation with the reality of test automation. Often the expectation is that test automation will be easy to implement, will perform better testing than is already being done, that it will be implemented and maintained by a separate team independent of the developers, and that the testing can be performed by unskilled technicians.

All of these expectations, to some degree, are wrong and historically I've had to implement the automation between product releases and often without the full understanding or awareness of the managers that I've worked under.

So if test automation isn't easy, won't perform better testing than is already being done, requires time and resources from the development team, and requires people with superior skills, why would a software project ever want to implement test automation? The answer is that so a lean team of people can evolve the software safely and efficiently. Once established, the team can be reduced in size and still be productive and cost effective.

1.1 Why Isn't it Easy?

Test automation is not easy because any nontrivial application is complex and is often built or designed without considering how it is to be tested.

By definition, a system can't be tested if no one knows what it is supposed to do. All tests require well known initial conditions, some sort of input sequence, and an expected behavior/output of the system under test. If the test is to be automated, the initial conditions have to be established by the test automation system, the inputs have to be supplied by the test automation system, and the output of the system under test has to be “measured” for correctness. This is very tedious work. Work that computers do well once they are programmed to do it.

The defense systems I worked on at Texas Instruments were quite different from the telecommunication systems I worked on at DSC, Alcatel, Alcatel-Lucent, and Infinite. The tools used to test the systems and to document requirements, design, and test procedures were also very different. The frameworks for building the automated test environments, however, were quite similar from system to system and was incrementally refined from project to project.

There are two tasks that take time in test automation:

- Understanding the system to be tested
- The development of the specific tools necessary to test the specific application

1.2 Why Isn't it Better?

Automated testing isn't better than manual testing because typically it is not possible to automate the test case design. The test cases themselves can often be generated by

automation tools, but the design of what is going to be tested and how to do the testing largely requires human analysis.

The comparator “Better” requires some constraints. It is certainly the case that it is possible to write testing tools that will generate a nearly endless stream of test cases. That can certainly be done very well automatically. So if a test suite's quality is measured by bulk, automating test can certainly always be better than manual testing. I have worked in many groups where quantity of testing was considered the measure of quality. But if the tests don't happen to cover a failure scenario that exists, that nearly endless stream of tests is not particularly useful. Also, the mere fact that automated testing often produces more consistent results is a problem in itself. Often the actions of a test engineer doing something a little different uncovers something unexpected.

What automated testing is particularly good at is regression testing. It is very good at detecting differences in product releases.

1.3 Why should the best people be doing this?

Simply put, the best people should be doing this because they should understand the system. If they don't understand the system, they are probably not the best people on the project. These people should be the system engineers, lead software developers, human factors engineers, mechanical engineers – all those that took part in visualizing and designing the system. There is no special unique test skill that is required. One's knowledge of the system and what it is supposed to do is skill enough. Having a set of people who can realize this group's knowledge into an automated testing system is fine to start with, but those people soon become unnecessary. They become unnecessary as the group begins understanding how the test framework is implemented and how to alter the tools to fit new needs that arise. Often these people become eager to do this work because it gives them insight into how system requirement changes affect the entire system and insight as to how to express those requirements in a more universally meaningful way.

1.4 If it's automated, why can't unskilled people test the system?

The answer to this is question is that they can, but they will be doing the things that only require seconds or minutes a day to do, so why have the additional overhead.

2 Framework

The following paragraphs summarize the framework I've used and refined over the last 30 plus years. The framework has been very much a constant over the years, but the “tooling” of the framework has changed significantly. Initially the target systems that were being tested were written in TI 990 assembly language and installed on custom TI 9900 microprocessor boards in boxes connected to our development system over serial interfaces to a “front panel” of the microprocessor board. There was 6K of RAM and 12 of ROM on our boards and the clock speed was 3 MHz. Our documentation tool was “roff”. One of the last systems I worked on was a cluster of 4 Oracle T4's each with 64 G of RAM, one 8 core 8 thread CPU running at 1.65 GHz, nearly a terabyte of disk space, and connected to our development system over a 1 GBit LAN interface. Our documentation tools were Word and HTML.

On that first system, there were 8 test cases that took about 8 hours each to run. It took us about two weeks to run the tests and confirm the results.

On the cluster of T4s, there were 45,000 test cases. Most of them were automated and the automated portion could finish in about a week. Manual tests tended to extend the test period to about 6 weeks.

2.1 What do managers want to know?

Managers want to know “where are you” and “when you will be done”. Automated systems are very good about providing this information. They are typically too good. That is, my experience has been that objective measures given by an automated test system are not the measures managers want because they tend to be too truthful. Programmed metrics that are produced with measurable information can often tell managers that a schedule problem exists. Historically, I've passed lead test jobs onto others as a system matures and I move onto something new. Automated status reports are often dropped by the new lead person because they often can not take the suffering that objective status reports often produce.

Although managers don't often want this raw honesty and young team leaders are happy to make up their more favorable reports, this framework inherently provides continuous status reporting by having the ability to scan the test results periodically and send/post results that can be inspected.

2.2 What do testers want to know?

Testers want to know “how do I do it” and “how do I know the results are good”. Good test procedures are hard to come by and are absolutely necessary for automated testing. If the test procedures aren't precisely defined, it's nearly impossible to get consistent results especially in an automated environment. Another thing an automated test environment requires is a very precise way to know that a result is good.

This framework, by its nature, requires a tester to think end to end about what the test is and what should be expected. This attribute of the testing task is what tends to weed out those who are purely “testers” and again illustrates the reason why the people who should be doing this task should be the most talented and knowledgable people working on the application.

2.3 The tools

The tools developed to support the framework have been simplified over time as attempts to keep “current” were abandoned to instead support what was needed. For example, document processors such as roff gave way to products such as PDWS, WordStar, Word Perfect, FrameMaker, Interleaf, and Word. It became clear that company “strategic strategies” were not stable and to keep from developing document generators instead of defense or telecommunication products, my strategy was to develop a framework around a simple database and simple tools to alter and use it.

2.3.1 The database

The primary component of the framework is the test database. It is very simple and requires no commercial tool. The test database is simply sets of flat files. A test database is defined for each test category for a product and contains the test identification, the requirements, title, objective, procedures, setup, expected results, and actual results.

2.3.2 Planning

No matter what software development model a project uses, planning your testing should be seriously performed. Ad hoc testing results in needless test growth, delays in schedules, and gaps.

2.3.2.1 Requirements

All testing should be requirements based as the first step. That is, given no other information as to design and implementation, the only thing one can rely on is the requirements. As the product matures, that will change, but to start with, the product requirements are to be used to define what to test and thereby bound the problem.

So the first step of testing is to have requirements. All requirements should have a unique ID. All requirements should be testable which means that they have clear statements of conditions, processing, and results.

Since all kinds of products are used to write requirements, it is important to write requirements such that some sort of scanner can extract them from the documents and place them into the test database. What that means is that it is to everyone's benefit to write the requirements such that the unique identifier is parsable within the document file and that relevant text can be extracted. CASE tools can do this directly, but my experience has been that companies change CASE tools more often than word processing tools and in most cases have abandoned them. What I've been left with in recent years are documents that have largely been developed on an individual engineer's laptop often in Word. Once in a while, the engineer has some decade old macro that is used to synthesize unique requirement IDs which tend to inherently be parsable. More often, the engineer writing the requirements is simply doing what he or she is told and makes up an ID (if told to) that may be difficult to consistently find with a parser. With a little bit if training, it's not hard to write requirements with parsable IDs that make design and test mapping relatively easy.

One of the most important tools in this framework is a tool to extract requirement IDs and description text and put that into the test database.

2.3.2.2 Mapping Requirements to Test Cases

As test categories and cases are developed (not after), a mapping is made to

the requirements that is placed in the database. A test case always has an objective and that objective should be to test a requirement or potentially a set of requirements. In simple terms, most test cases should say something like, “the objective of this test case is to test this requirement: ….....” and should then say if there are any special conditions related to the test that may affect the expected behavior of the system.

The database holds requirement IDs for each test case. With that mapping in place, a tool can be written to map test cases to requirements and requirements to test cases. These “matrices” can then be checked for completeness – no traceability matrix should ever be done by hand. There is no need and it will often be wrong if it is done by hand.

2.3.2.3 Writing Test Case Procedures

Given rudimentary test cases with objectives formulated by requirements mapping, the tester is now in a position to be able to write procedures to test the requirements under the conditions specified in the objectives.

2.3.2.3.1 Manual

The simplest way to write test case procedures is to do it manually. That is, use a text editor to document the procedure step by step and to produce the test input and expected output. The test database format is such that it can easily be edited. This is a very tedious and error prone process, but there are times when this is the simplest most cost effective way to complete a test design.

2.3.2.3.2 Simulation/Emulation

Another way to write test case procedures is to have a simulation or emulator perform the function. In this discussion I'm using simulation and emulation interchangeably, but the two are different. Most automated test systems I've done for real-time related projects have used simulations to produce the test cases because there has been a significant effort put into a simulation of the system for the purpose of estimating performance and proving concepts. In other situations which I consider non real-time/semi-real-time like telecom switching systems, I've used emulations which only run the intended algorithms but are not meant to be simulations of the network.

In scenarios where the project is going to be long term and there are many paths to follow/test, having emulations write the test cases is the approach I take. The process I've found to work the best is to have a member of the test team write the emulation slightly ahead of the person writing the code that will be tested often while requirements are in flux/not approved. The emulator's job will be to:

- produce test input
- produce expected output
- document the scenario entered into the database objective field of

the test case

The most valuable outcome of this process is the built in process of validating the requirements – are the requirements complete and do they do what what was intended. Without putting in the full effort necessary to implement the new feature into the application, the feature can be sanity checked by building an emulator that can produce input and expected output for your test cases.

2.3.2.4 Status

Status reporting is built into this framework by automatically producing daily reports of the test state.

In the beginning, managers are able to see the test categories and test cases within each category. They can continuously see the test coverage and know when the test cases are nearing completion by watching the coverage converge on 100%.

During execution, they can see the progress by receiving reports of tests executed and their success or failure. Failed test cases can be inspected.

Status is objective. No person makes a decision of what to report. This sometimes is deceptive in that, for instance, early test results may project that testing will not complete on time or that it may complete much earlier than is being projected. That means that an experienced person does need to help interpret the raw data at times, but the data itself is perfectly objective and is not altered.

2.3.3 Execution

For this framework to be successful, it is necessary to assure that test hooks exist so that the driver can setup the conditions, initiate the test, and compare the results. This has been one of the big problems when purchasing commercial testing tools. They have often been GUI based, too general purpose to easily test a specific target system, or too specific and not flexible enough to handle variations in protocols or simply violated the protocols. The introduction of IP into these tools has gone a long way towards interfacing tools together and the introduction of command languages so that developers can tailor how test equipment works has helped eliminate most of these problems.

This task, the development of the driver and its interface, takes some upfront time at the beginning of a project, but once completed, it is rare that the driver need any adjustments.

2.3.4 Documentation

The contents of the database can easily be formatted into variety of document formats and in the framework, this is done daily in parallel with status generation. The purpose of doing it daily is so that test results of failed tests can be examined with the current set of procedures and test conditions. This helps

developers see if there are trends in what is failing.

Over the years I've used many different document languages (roff, PDWS, FrameMaker) but over the last decade or so I've settled on HTML and mainly because I've been able to “tar up” books and send them to the customer for review and because test case development can then be done on a wide number of platforms that simply have a text editor, perl interpreter, and browser.

3 Special Topics

3.1 Building in test hooks

One fundamental rule of software design is to isolate your interfaces. If this is done well, it simplifies where test hooks will appear and often dictates how they will work. As an example, one of the first systems I worked on retrieved all of its inputs from A/D converters and sent all of its outputs to D/A converters. When realized, this became two assembly language routines that accepted a channel (an integer that mapped to a device) and a source/destination address. These two routines became the interface to the simulation. When testing, the embedded software read from static memory areas populated by the simulation or wrote to static memory areas that the runtime would collect and record the data for analysis. Eventually this evolved such that the simulation directly accessed the data produced by the embedded system. The embedded software replaced the functionality of the simulated component. That is, it was then part of the simulation.

3.2 Real-time

My experience has been that most automated testing can be done in non real-time. There are certainly disadvantages to that. One of the first systems I tested processed imagery that, in real-time, was entering the system at 60 Hz. We were storing these test images on disk and loading them one image at a time every 60^{th} of a second was simply not possible with the technology 30 years ago. To perform a test that in real-time would have been 20 seconds or less took about 8 hours. At first glance this sounds unacceptable, but proving that the embedded software got identical results to the simulation was very useful and important information. It didn't verify that the embedded software actually ran in real-time, but it did go a long way to demonstrating that the embedded software was implemented correctly and the system would likely have the performance level predicted my simulation results. Real-time behavior was something that could be tested by other means in a real environment.

As technology improved, the 8 hours was reduced to minutes and eventually I started working on systems running faster than real-time, but my point is that often it is not necessary to run in real time and nearly all of the automated testing performed in the telecommunications world is semi-real-time.

3.3 Closed Loop

When we first started executing tests that compared simulation results with embedded results, we found that our biggest problem was divergence of internal closed loops. To this day, similar problems have been detected in vastly different systems.

In 1979, the processors being used in the embedded systems were integer processors and we emulated real number processing using techniques such a “Q arithmetic”. The simulations on the other hand were written taking advantage of the processors available on the simulation machines that were capable of doing floating point arithmetic. Further, the simulations took advantage of more memory, the fact that they didn't need to run in real time, and high order languages (FORTRAN) to simplify the implementation of the control algorithms that were being embedded.

As mentioned earlier, our inputs to the embedded system were digitized voltages that represented angles. Our outputs were digital values that needed to be converted to “command” voltages. It was simple to do the scaling of voltage values within the simulation to produce the digital inputs or expected outputs, but what we found that wasn't easy was the fact that the signal processing being done fed back into the simulation and embedded software equations independently. Our internal closed loop states changed even though our inputs were identical. Although this could be mitigated to a small degree by expanding tolerances in the expected results, this didn't offset the problems incurred by the fact that the dynamic range of the two floating point representations were not the same. If an internal state of a control equation went into saturation (this usually happened in the embedded system), the two systems failed to match from that point on.

We had to develop techniques to prevent this from happening that were mostly a process of making the simulation more like the embedded code. This detracted a bit from the usefulness of the simulation as an investigation tool but improved the ability of the simulation to predict more closely the results that would be experienced in the real world.

4 Summary

In this short article, I've attempted to sketch out what the test framework I've developed is like and some of the difficulties involved with developing an automated test system. In subsequent articles I'll give some examples of applying this framework to a project of my own making.