TEST AUTOMATION: AN ARCHITECTED APPROACH

Dan Young
Schwab Performance Technologies, Raleigh, NC. USA

Abstract

In the world of automated testing, everyone throws around buzzwords like "data-driven," "data flows" and "error handling," but what does it take to produce automation that is efficient, maintainable and usable?

Successful automated testing requires a considerable financial investment. Simply installing an automated testing tool and recording scripts may jumpstart the initial automated testing effort, but this approach will become difficult to maintain and therefore more expensive.

A more cost-effective solution is an architected solution. Providing the right architectural framework for automation development means that the automation code can be used for longer periods of time with less maintenance than a simple record/playback solution. This translates to a significant savings over the course of longer projects, and the ability to more thoroughly test an application with less employee overhead.

The particular architecture championed here is based on the idea of automation code as an application in its own right. Code reuse, encapsulation (on many levels), recursion, object-oriented concepts, testing maturity and usability (of automation by non-technical business analysts) are covered. The result of this architecture is reliable automation code with scripts that can last the entire life of the product (not just the project) and that can be used and enhanced by business analysts who have little to no knowledge of automated testing.

Contents

Introduction 3
Assumptions: 3
Overview: 3
When Good Automation Projects Go Bad 3
Planning an Automation Project 4
Requirements: 4
Balancing Personnel: 5
The Automation Plan: 5
ROI: A Simple Rule of Thumb 6
The cost of manual testing: 6
The cost of automated testing: 6
The Architected Solution 6
Encapsulation, Encapsulation and More Encapsulation 7
Using functions 7
Encapsulated test cases 9
Managing Results 10
Pre-Scripting 11
Selling the Architected Solution at Your Workplace 11
Conclusion 12

Introduction

Assumptions:

This document begins with the assumption that the reader will have some knowledge of automated testing and/or programming.

Overview:

Automated testing can be a very valuable tool to gauge and improve the quality of any software product. If it were as simple as recording and playing back some test scripts, every company would have a vast array of in-depth test suites that covered the testing of their products from front to back, but they don't. This document will attempt to explore some of the pitfalls of automated testing as well as present an architectural framework that has produced proven results.

When Good Automation Projects Go Bad

There are many things that can derail an automation project. To start with, I blame the companies who make commercial automation software. When an automated test tool vendor comes to your company, they want, first and foremost, to sell you on their software. They seem to believe that the best way to do this is to make you believe that by purchasing and installing their software, and then pressing a few simple buttons, you'll have a robust test suite that will meet your needs.

The second group of people that must bear a large part of the burden for this type of derailment is the management that buys in to the sales pitch. Certain managers hear how simple something can be, and they want it to be true so badly that they start believing it. If you're one of those managers, be very clear, if anyone tells you that you can have your automated test suite up and running in a couple of days, don't believe them.

The third group that has to take responsibility for the derailment of potentially good automation projects is people like me, the ones who use the tools directly. Often we are so caught up with deadlines and management expectations that we fail to research and implement the tools to their maximum benefit.

Perhaps the most difficult obstacle to good automation is the mindset that automation is merely a part of the overall development cycle. On some level this is true, but if you

want automation that will be stable and maintainable through the life of the product and not just something that you throw out when the current project is over, automation must be approached with a broader view. Automation that will last through the life of the product needs to be approached as a software development project in its own right.

This paper could take the whole Rational Unified Process and correlate it piece by piece, but that would be tedious and out of scope. Instead I'm going to focus on a few key ideas that will go the farthest toward insuring a successful automation project.

Planning an Automation Project

The best automation is accomplished when approached and justified as a software project in its own right. I often describe my job to friends and family as "I write a program within a program that tests another program." This is usually the point where people roll their eyes and decide to give up understanding what I do. Anyone who has read this far probably understands the above statement and realizes that automation *is* a form of development. Below are a few key elements of a software project that are essential to a good automation project.

Requirements:

No successful software company would think of asking a developer to just sit down and create a software tool that does what the customer needs; and yet this approach is often taken with automated testing. A lot of times the business analysts who plan and conduct the manual tests also double as automation engineers. As a combined analyst/engineer, that person is expected to simply understand what his or her customers (e.g. product managers, project managers and development staff) need, and provide automated testing that meets those needs.

The same processes that are used to generate requirements for the software project that your company is engaged in should be used to generate automation requirements. The automation requirements can be as formal and specific as the software requirements, or they can be as informal as agreeing what modules or portions of the application should and should not be automated and which paths particular test cases will take through the application.

Balancing Personnel:

With the exception of very small companies, or very small projects, development tasks are generally divided and assigned to the most qualified persons. A company may have a system architect who directs coding standards and designs the application. There may be specialists who bring certain knowledge that serve mainly as resources, and then there may be developers who use the artifacts created by the specialists and implement the designs. In some cases portions of the coding are outsourced to other companies.

Generally, it seems that the Quality Assurance staff is just expected to wear all available hats. Even in companies with larger QA departments, it's not uncommon for a single person to be expected to design the tests, write the test plan, manually test the application and write the automation.

A more logical and successful approach, even in QA departments as small as three people, is to assign some staff to focus on manual testing (or the business-analyst-oriented tasks) and assign others to focus on automated testing (or the more technically oriented tasks) .

This approach offers several advantages. First, it gives management better tools to manage resources in a multi-project environment. Since the automation staff is not bound to a particular project, a manager can balance needs between projects. By leveraging this balance, a manager can insure that the growth of automation is consistent with the priority of each project, and not just left to a given staff member's propensity to automate his or her work.

Second, this approach increases employee satisfaction and productivity by allowing people to focus in areas of interest. I have rarely met anyone who truly enjoys both manual testing and writing automated testing. Just about everyone has a clear preference; and by allowing staff to pursue areas of interest, employee retention and satisfaction is increased thereby decreasing the overall project risk.

The Automation Plan:

Since functional automation is predominantly dependant on the GUI, the automation plan will always have dependencies on the development plan. It's difficult, but not impossible to begin the process of automated testing before the GUI is in place. That said, a clear and separate automation plan will help track the success of the automation project. The automation plan should include assumptions, considerations, time estimates and estimated ROI.

ROI: A Simple Rule of Thumb

Return on investment is a very hot topic in the world of automated testing. There is far more material than can be covered in this paper, but this paper would be incomplete without some mention of it. An important part of the automation plan is to provide business justification. A very simple method of calculating ROI is illustrated in the following section.

The cost of manual testing:

The cost of manual testing can be calculated by estimating how person/days are required for a single testing cycle or component of a cycle. Multiply this by the number of test cycles anticipated during the project and you will know how many person/days are required for manual testing during the life of the project itself. If a single test cycle takes 7 person/days and there will be 10 test cycles during the project. Then there will be 70 person/days of effort to support the project.

7 person/days per cycle x 10 cycles = 70 person/days of manual testing over the life of the product

The cost of automated testing:

If the same project can be automated in 30 person/days (which would not be at all unreasonable if the manual test cycle takes 7 days), then ½ a day of regression review is required for each testing cycle, the total cost is reduced to only 35 person/days.

30 person/days to automate + (1/2 day regression review x 10 cycles) = 35 days over the life of the project

In this simple example, the cost of automated testing is 50% of the cost of manual testing provided that all other things are equal. This return grows exponentially as the product matures and other projects are able to build on the tests that are already automated.

The Architected Solution

The foremost factor in the long-term success of automated testing is very simple. If the code is maintainable, then it can be used easily throughout the project and reused in future projects. The question then becomes how to create automation that will be accessible to non-technical business analysts, easy to use and easy to maintain.

Encapsulation, Encapsulation and More Encapsulation

Simply put, encapsulation is using a part or a piece without knowing, or needing to know, exactly how that piece works. If you were going to build a computer, you probably wouldn't build each component. For instance, you would probably buy a motherboard. You don't need to know the specifics of how each circuit works. All you need to know is how to install it, and how to set a few jumpers and how to hook other components to it. For you, the motherboard is an encapsulated piece of your computer.

Using functions

Different automation tools use different terminology to describe their features. For this paper, I'll use the term “Function” to mean a block of code, which may accept parameters, performs a specific task and returns a value (in some cases “void”). Since functions can call other functions, encapsulation on this level is very powerful in reducing maintenance.

Function Example:

At the lowest level, a function may simply write a line that is passed in as a parameter to a table in a database. Any function that needs to write to this table in this database can call this low level function. If, at some point, you need to change which table gets written to, you can simply make this change in one function, and no other code needs to be touched. If you've recorded 200 test cases, and added all the code to write this data in each test case, you'll need to edit 200 test cases. The five minute fix in the encapsulated version of this example could easily become a two or three hour fix in a record/playback environment. The more times this table changes, the more time encapsulation saves. Functions at this level are only called by other functions. I refer to these as non-scripted functions.

A more involved function may be recursive. A recursive function is a function that calls itself. The simplest example that I've seen of this is a function I wrote to delete a directory. The automation tool that I work with gives the user the functionality to delete a directory, but only on a single level, and only if that directory is empty. If I wanted to delete the directory “C:\Automation\MyDirectory”, but the directory had other files and directories within it, the test case would stop with an error.

The automation developer may not know how many directory layers there will be in the “C:\Automation\MyDirectory” tree at runtime. In a recursive “DeleteDirectory” function, the path “C:\Automation\MyDirectory” would be passed as a parameter. The function would make a system call to get the contents of that directory. If the particular content file is not a directory, a system call will be made to delete the file. When all contents are deleted, a system call will be made to remove the directory itself. If the

content file is a directory, the “DeleteDirectory” function will be called passing the original path, plus the content file as the new path. See figure 3.1 for pseudo-code.

```
void DeleteDirectory(string sDirectory)

 if (SYSTEM_DirectoryExists(sDirectory))

         // Variables
         list of FILES Directory = SYSTEM_GetDirContents(sDirectory)
         FILE FileName

         // Cycle through each item in sDirectory.
         for each FileName in Directory

              // Item is a subdirectory. Worm into it.
              if (FileName is a directory)
                 DeleteDirectory("{sDirectory}\{FileName}")

               // Item is a file. Remove it.
              else
                  SYSTEM_RemoveFile("{sDirectory}\{FileName}")

// Delete the starting directory
     SYSTEM_RemoveDir(sDirectory)
```

Figure 3.1

These low-level or non-scripted functions make great building blocks that can vastly streamline the creation of higher level functions. Be aware, however, of the tendency that some automation developers have to encapsulate too much. Encapsulation overkill can be expensive. If an automation developer writes a function that sets a variable, calls a second function, retrieves and returns the second function’s value, nothing was gained. In fact, now this function will be more difficult to troubleshoot because an unnecessary layer of complexity has been added. See Figure 3.2 for an example of this.

```
boolean FunctionA()
  // set varable
   string sString = "Hello"
   boolean bReturn

   bReturn FunctionB(sString)

   return bReturn

boolean FunctionB(string sString)
  if( sString = "Hello")
    return true
  else
    return false
```

Figure 3.2

The highest level function is what I call a scripted function. These are the functions that someone actually entering a test case would use. The scripted function navigates a full path through the application, performs the entire test, and returns the application to a known base state. After this, the function performs any clean up necessary to prepare for another test.

If you are automating, for example, a contact management application (Figure 3.3), one test case might be to enter and verify a new contact. The scripted function would perform all the tasks associated with this test case. The function would set up the proper database and launch the application if necessary, navigate through the data entry, verify that the data entered was saved properly, then return the application to the base state and perform any clean-up as dictated by the automation plan.

Figure 3.3

Encapsulated test cases

With the entirety of the test case functionality encapsulated in the function, a test case in a script becomes simply a test case declaration, passing in required variables, and making a single function call. From the example above, a business-analyst could create hundreds or thousands of test cases by simply varying which database information gets written to, or varying which fields of information are passed (such as first name, last name, address, etc.) or what exactly is passed to each field(e.g. pass alphabet characters in the phone number field).

Figure 3.4 shows pseudo-code for a simple test case that picks a database and adds only a first name and last name parameter to the contact management software. The "AddNewContact" function would select the "TestData" database, navigate to the new contact screen, populate the first and last name fields, save the contact, verify that the expected behavior occurred, then clean the data.

```
testcase AddNewContact_001()

list of string lsInformation = {...}
"Database = TestData"
"FirstName =  Bill"
"LastName =  Smith"

AddNewContact(lsInformation)
```

Figure3.4

This means that test case entry becomes accessible to everyone. Even people with no experience in or desire to deal with automated testing can enter test cases. In our organization, during crunch times, we've recruited help from our technical support department and our administrative department to enter test cases for us. With about 10 minutes of preparation and training, anyone can enter automated test cases.

Managing Results

Any automated test tool will give some type of result when a test script is run. There are two main problems I've found with these results. The first problem is that the result files generally only give a pass/fail status on the test. I would maintain that there are actually three possible results to a test -- pass, fail and crash. If a test case passes, the automation and the AUT (Application Under Test) have worked flawlessly. If a test case fails, it is most likely that the automation worked flawlessly and the AUT has regressed in some way. If the test case crashes, or fails to complete, the automation code becomes the place to begin the troubleshooting process. This sounds simple, but I've literally spent days in five or ten minute increments troubleshooting failed test cases where the automation ran without a hitch and the AUT was the culprit.

The other problem with using the test tool result file is that anyone analyzing the results is tied to using and understanding the test tool. The tool vendors encourage this of course, as increased tool dependence means increased need for licenses of their product. Our company has made what is probably a controversial departure from this. Every test case produces a separate result in the form of a text or csv file. This means that after a test case is run the first time, all data in the resulting file can be verified manually, and then the file can be used as a baseline. With each subsequent test run, the files produced can be compared using one of any number of third party text comparison tools. This means that any business analyst can quickly see the test results and begin assessing how the code regressed. The result is a decrease in the time between the discovery of an issue and that issue being logged for development. Additionally, any developer can

easily look at the results thereby reducing the chance for miscommunication between departments and eliminating the time a developer may have to spend waiting for a QA person to answer questions.

When taking results into your own hands, there are some considerations that must be taken into account. Within this architecture, all result files are written into a particular directory structure. The directory structure is divided by areas of functionality in the AUT. Reports will get written to a "reports" directory. License tests will be written to a "license" directory. This makes it simple to compare entire directories of results at once and the responsibility for reviewing particular areas of functionality can easily be divided among the QA staff.

Another thing to consider is how you will handle successive results. Keeping track of successive results gives the test team the ability to analyze trends in issue discovery as well as to isolate when a bug was introduced into the product. Isolating when an issue is introduced helps development more easily narrow down which code changes created or revealed the problem. We handle this in a rather simple fashion. We use the product version number as the root directory for result files. The entire directory structure described above is written under a directory with the product version.

Pre-Scripting

In a preceding section I mentioned that it is difficult but not impossible to begin automation prior to the production and stabilization of the GUI. Using the architecture described above, it is possible to script test cases prior to any code development on the application. Once the project plan is in place, and the manual test plan is written, the automation plan will specify which scripted functions are needed, and what parameters get passed to that function. Using the simple Contact Management example from above, it would be simple to take all of the test cases that add a client, define them in a script, add the proper variables and set up a call to the function which doesn't exist yet. After the GUI has stabilized, and the function code has been written, the test can be run, verified and a baseline can be put in place. This process makes great use of the time at the beginning of the project when there isn't much to test and nothing that can be automated in a traditional record/playback model.

Selling the Architected Solution at Your Workplace

As good as this solution may sound; it is not always easy to sell. It was not always supported by management at my company. A good bit of this architecture was put into

place surreptitiously as portions of the automation needed rewritten. After it was in place, it was agreed that the approach was good, but it is a lot harder to sell up front.

Most managers prefer quick results. At the same time, good automation takes considerable planning and investment before results can be achieved. Sometimes compromise is the only way to achieve the maximum benefit. When automating a new project, it may be easier to record a few things for quick results, and then build in the architecture after some results have been generated. For products that have existing automation, the best approach is to build in the architecture as areas of the regressions need revisited.

The first step of migrating to the Architected Solution is the most difficult. Planning and building the infrastructure takes time, and the results are not visible. But after this hurdle has been cleared, and the architecture gets applied to more and more of the test suite, the time for maintaining the suite decreases, and there is more time available for improving and expanding the architecture. Since maintenance is low, the automation can be maintained through the life of the entire product, not just a single project. There are few things that project managers like to hear more than that part of the testing for their newly-assigned project is already done.

Conclusion

Properly planned and implemented automated testing can significantly lower project risk and cost. In the course of completing a project, there are many temptations to take shortcuts. While appearing to offer short-term savings, these shortcuts actually turn out to be quite expensive in the long-term.

Anyone in the software industry understands that solid, efficient automated testing can greatly reduce the risk of a project and a product. If that automation is well designed and implemented, it can not only reduce the risk of the current project, but can actually reduce the risk and cost of future projects.