



## **Automation By Design:**

### **Why Use Script-Based Testing When We Can Use Table-Driven (or Scriptless Automation?)**

**by Jamie Mitchell**

*This column was originally published in the Journal of Software Testing Professionals, Volume 3, Issue 2.*

Henry Ford invented the automobile. Just ask any elementary school student who has suffered through Miss Waxman's twisted idea of history class. Benjamin Franklin invented electricity while flying a kite, Marconi invented the wireless radio, and Don Ameche invented the telephone.

History is rife with examples of people getting credit for the invention of an object, technique or process; sometimes even justifiably. As I grew older and was able to sort out the subtle distinctions between historical facts, I learned that there were many horseless carriages before Ford's, electricity was not invented nor discovered by Franklin and that Marconi likely took credit for others' work. Of course, Don Ameche did invent the telephone: I saw the movie.

Henry Ford did not invent software test automation; but many of his ideas are still applicable to efficient, cost effective automation. He is credited with understanding that if we were to build each individual car by hand, using skilled craftsmen, we would build very few cars. So he engineered the assembly line. Each worker performs a limited set of tasks; when many workers each do their specialized jobs together once; a usable automobile is the result. When they each perform the same tasks many times, an empire is built.

A single test case may be considered valuable, and if we handcraft an automated test case lovingly over that test, we may have the best automated test case ever written. Most organizations do not have, nor can they afford individual craftsman to build individual automated tests, however. We need to come up with a way to automate automation; that is, we need to generate a great many test cases efficiently and cost effectively. When was the last time you heard someone say, "Sorry, you've done too much testing on this system?" There is only one answer when someone asks a tester how much testing needs to be done, "MORE!"

In the last column, we discussed different types of test automation architecture. In some ways, automation architecture addresses the question, "How are we going to run a non-trivial number of test cases unattended?" In this column we will start discussing how to



utilize the architectural concepts to mass produce automated test cases in those non-trivial numbers that we need.

The Henry Ford metaphor came to mind when thinking about the origins of advanced automation techniques. There are those who attribute certain automation methods to specific individuals; he invented this or she invented that. Concerning who came up with the original ideas for automation architectures, there is a [sort of] joke among experienced automators; each one of us individually invented the framework architecture, the data-driven (DD) architecture, and the table-driven architecture. In reality, I think that is exactly what happened.

The nature of test automation evolution is that many of us tended to start with record / playback. When that failed, we analyzed the failure and started applying software engineering concepts and came up with the framework architecture. The framework architecture replaces areas of recorded code in scripts with pre-designed, pre-built functions that perform tasks against the system being tested. A few examples:

- `OpenFile(FileName)`
- `CreateRecord(FirstName, LastName, Address, etc.)`
- `VerifyReturnCode(Expected, Location)`

By replacing recorded code with tested, reusable code, we reduce maintenance, and increase scalability at the expense of doing more programming.

When we needed more testing quicker, we came up with data-driven techniques. These allowed us to reuse code more often; we could essentially run the same scripts using different data sets to change the actual testing being done. This gave us more testing with less redundancy at the cost of more complexity in our data-access routines.

When we needed non-technical people to be able to create automated test cases, we came up with table-driven concepts. By describing the individual steps in a test in a stylized, disciplined way, we were able to create automated test cases without using actual scripts. Testers and business analysts could now define the tests without knowing programming; all it cost us was a lot of very technical programming to translate and execute their test descriptions.

The point is that automation has taken a logical, evolutionary path, and many who have stuck with it have likely gone through these stages<sup>1</sup>. Looking at the big picture, each step can be said to allow more testing using less technical people to create the tests at the expense of higher and higher technical work by the automators.

---

<sup>1</sup> One thing that is common to almost all successful, long-time automators is that most of us have had the luxury of being allowed to fail. It is learning from our failures that has allowed us to make any progress.



I offer that common evolutionary path as a reason for discussing many different architectures in this column. Several people have said to me that table-driven (or scriptless automation) is the only worthwhile architecture to discuss; talking about lesser techniques is doing a disservice to the reader. After all, why use a propeller plane when a jet is available?

Well, anyone who wants to get from Columbus, Ohio to Lancaster, Pa. can tell you that prop planes still work. My belief is that there is still a place for script-based testing. If a survey were to be taken, I am sure that there are many more script-based automation projects going on than there are table-based. I believe that a solid understanding of the basic fundamentals of automation is a good idea; a good automator should be able to work with any architecture for any project (hopefully, she will have the freedom to choose the best architecture to do the best testing.).

In this column we will start to discuss some aspects of the test automation suite which are applicable to any of the different architectures. These techniques work equally well with DD or table-based suites.

Assume for the sake of argument that we have come up with a way to create stand-alone test cases. The tests can be run unattended, without human intervention. We likely are using some form of framework architecture, meaning that there are likely to be locally written functions that are performing tasks against the system under test. These test cases may be reusable, pulling data out of the data store for each iteration executed. They may even exist as a sequential set of rows taken from a database or spreadsheet. Now let's figure out how we will run a bunch of these suckers.

### **Suite Administration**

Let's look at the big picture. We are planning on having hundreds [thousands?] of automated test cases. We need to consider the details of running and administering this many test cases. We have three choices: we can ignore the administration details until we have a huge mess (this method is unfortunately chosen by default by many automation teams.) We can buy an expensive test planning tool which will sort of allow us to run suites and sub-suites (at the cost of doing things "their" way.) Or we can come up with a logical, easy to implement scheme of our own<sup>2</sup>. Whatever we decide on, it must be robust enough to last a long time, through changes in methodology, process and even architecture.

For administrative purposes, we want to be able to:

- Build a hierarchical execution graph of tests to run. Our scheme should be flexible enough to allow different stakeholders to run different sets of tests based

---

<sup>2</sup> Is it tough to detect which of these schemes I favor?



*Innovative solutions to maximize your testing investment.*

on their individual needs. Ideally, I want a turnkey solution for any person who wants to run a set of tests.

- Allow a non-technical person to run the test cases desired. No training or skills implied.
- Allow a non-automator (non-technical tester) to handle administration details of the suite. Administration should never require going in and changing code.
- Ensure that we can specify order relationships between test cases (test A executes, producing something, and then test B runs and uses what was produced.)
- Ensure that we can easily mark tests to not run (when they are broken or cause known errors in the system under test.)
- Maintain the suite easily over the long term, especially considering personnel turnover and tool changes.

In addition, we would ideally like to come up with a scheme that will survive architectural changes as we evolve our automation (eventually) to a higher order architecture (DD or table-driven.)

My design choice to fulfill these requirements is a record-based, test header system which I have used for several years now; it has evolved into a very stable scheme that I have implemented in many locations, using each of the different automation architectures and all of the most popular automation tools. The test headers can be created in spreadsheets or database tables. I have found that smaller implementations are easier using Excel™. When the number of test cases reaches 900-1000, I find that database implementation becomes more efficient. Implemented correctly, it is easy to switch from Excel™ to Access™ when it becomes necessary.

The header record is designed to allow testers to create new headers without the technical expertise of an automator. It is independent of the test case itself. You might think of it as an indexing service; it describes the test case, it contains relationships among other test cases, and may even supply some high-level data to the tests<sup>3</sup>.

During execution of every test, the information from the header is loaded into the script and governs its execution. This requires that the automator create some custom data access routines; it is well worth the time spent as we can use the same code to load in data for DD and table-driven tests. Because the header data is actually used in the execution of the test cases, the data in the header record usually consists of strings, numeric, or Boolean<sup>4</sup> values. Each header (row) in the sheet or table represents a single test. The data access function has the following signature:

---

<sup>3</sup> I tend to put data that may change periodically in the header. For example, if the security passwords change often, I put them in the header to make them easier to administrate. More on that below.

<sup>4</sup> A Boolean value is programming talk for a two member set of conditions: {TRUE | FALSE}, {YES | NO}, {0 | Non-zero}. These are used for decision making.



LoadTestHeader(LineNumber)

It picks up the row and places it in a global array available to the test script. Indexes into the array are the column names for those languages that support associative arrays:

HdrArr["Description"]  
HdrArr["Unique Number"] etc.

For Robot, which does not allow this type of array, we use a 2D array of strings:

HdrArr[0,1] = "Description"      HdrArr[1,1] = "xxxxxxxxxxx"  
HdrArr[0,2] = "Unique Number"    HdrArr[1,2] = "234"    etc.

In the following sections, we will discuss the header fields and how they fit into the automation architecture. There are two flavors of headers: the header record for individual test cases (I will call them "main test cases") and a modified header record for data-driven (DD) test cases. I believe it is essential that each DD test case have its own header record. Too often, data-driven test cases are treated as sub-tests and simply represented by iterations in a spreadsheet. To my mind, a test case is a test case, however it is automated. If the test is important enough to automate, it is important enough to have its own header and documentation.

Main test case headers can represent stand-alone test cases, shell test scripts<sup>5</sup> (a script whose main job is to call other test cases), and data-driven test driver scripts (a script that actually traverses through the system under test, but depends on externally described data for its execution.<sup>6</sup>)

Here then is a description of my header record fields:

## Test Identification

### Unique Number (both)

A unique index number for the test: This number allows a 'join' to be created, associating the test with data in another sheet or table. This becomes essential when the suite is moved to data-driven technology, as a single test script can represent dozens or even hundreds of individual test cases. The easiest way to create a unique number is to simply give the header record the row number<sup>7</sup> of the Excel table, or create an 'automatic number' in the database table. The data-driven test headers also require unique values.

<sup>5</sup> Sometimes called Master or Parent scripts.

<sup>6</sup> There is nothing particularly scientific about these names; they are simply what I have gotten used to calling them.

<sup>7</sup> If the row number is used for unique numbering, rows obviously can not be inserted into the sheet; they must be added at the end. This is an example of a process rule that must be followed by all on the automation team.



### **Join (DD only)**

This column is a reference to the main test which “owns” the DD test case. One way to think of this is that the main test case represents a physical script and a data-driven test case is a single execution of that script with a specific data set. The Join column in the DD header simply consists of the unique number of the owning main test case. There are usually many DD test cases that reference the same main test script (a many-to-one relationship.) In a future column, I will discuss how to use the DD headers to facilitate a tester creating new test cases without the need for a technical automator. Since the unique value is available to the main test case at run time, it can go out and pick up all of the DD test cases it “owns” dynamically (as long as it knows that they exist – see the Data Driven field below.)

### **Test Name (both)**

This name may be the physical script which contains the test or it might be a logical name representing a set of instructions in a table or spreadsheet. This can be used by the top level script which coordinates the execution of the suite, via the “call” functionality of the automation tool, or by the virtual machine script running the table-driven tests. The test name may be encoded to specify some special meaning to the test group using a described standard, or it might simply be a name chosen by the tester because it is meaningful to her. Naming conventions have become a religion to some; my belief is that the standards of the organization should determine how test cases are named; therefore, I have no specific naming convention defined.

### **Test Reference (both) [Optional]**

If the test group uses a formal test planning tool, this field allows the tracking identifier be associated with the test. This can be used to report results back to the test planning tool or for formal reports. If no such tool is used, I omit the field from the header.

### **Description (both)**

Description: a short description of the test case itself. This is placed into the log to help the test runner understand the results of a test, especially when the test fails. I have found that result logs are much easier to read when they contain a short description of each test case run rather than forcing the log viewer to refer back to other documentation or translate cryptic test names. Again, if the test case is worth automating, there should be no difficulty describing it.

### **Hierarchy Field (Main test only)**

#### **Parent Name**

When every test knows who its parent is, an execution graph of any complexity may be created. I have gotten as deep as 12 levels of scripts already (test 1 calls test 2 who calls test 3 who calls test 4 etc.) This allows subsets of tests to be created dynamically for



testing different branches of system functionality, different error conditions, different data configurations, etc.

Generally, I have one script that is considered the parent of the entire suite. It knows how to call any number of children scripts through the test header records. Any child script may also be a parent of other scripts, which may be parents of other scripts, etc. By starting execution of the top level script (the one without a parent), any or all test cases in the entire suite may be run.

Using hierarchical grouping allows testers to easily run just the testing desired. As the test suite grows larger, it becomes more important to classify test cases by function, sub-function, sub-sub-function, etc.

I try very hard not to 'hard-code' shell scripts. I use the header records exclusively to determine which parent scripts know how to run which child scripts. Why? A tester should be able to easily plan the test suite and create the hierarchy without going into the scripts at all. This is a feature of most of the automation I write: give the power of planning, running, changing, and reviewing automated testing to the testers who own it wherever possible. The testers should not have to be technically capable of understanding all of the internals of the automation; the test header system makes them the master of their domain.

### **Selection Fields (Main test only)**

Fields that allow test runners to select the testing they want to do.

### **Test Level**

The test level is a number (1..5) that is assigned to each main test script. When a test runner wants to run the suite, they can choose the amount of testing that they want to perform through interactive prompting. Setting the value to a '1' designates the test suite to be run for a smoke (or build acceptance) test. A '5' means that the test case is only to be run when doing a full regression set. A number in between is a level of testing between minimum and maximum. Each number includes all those tests represented by numbers below them (a 3 will also execute 2s and 1s.) This is designed to allow the test runner to easily choose the desired level of testing or perhaps, the amount of time that the tests will run.

The ability to scale the amount of testing easily often makes the automation more likely to be run. As a hypothetical, consider the case of a developer who would like to do some testing against a private build that she has incorporated changes in. While she is not likely to kick off the entire suite, imagine what would happen if she could double click on an icon on her desktop to start the department automation suite. A set of prompts allow her to choose exactly the level of testing she wants to run over her lunch hour. She chooses level 2 and selects the functional areas that she is interested in (see the next



section), and leaves for lunch knowing that the testing will be selected intelligently based on the guidelines she chose. If we make it easy enough for people to run automation, this does not have to be hypothetical.

How does a person know what a level means? The tester, who plans the automation, should gather input from different users and determine what level of testing they want. Not everyone wants to run all of the tests all of the time. Developers should be able to run automation against private builds. Testers should be able to target specific testing for specific reasons. Business analysts may want to run some tests to create data, build environments, or such. The test suite should then be engineered so that it can be broken into levels of testing and the meaning of the levels should be clearly documented. I see this as part of the test planning process.

### **Functional Class (Main test only)**

Each test case is designed to test inside a specific functional area. This field is a text field holding a string describing the functional area the test covers. When the top level parent runs, it picks up all of the strings from this column and puts them into a prompt. The user can then select those functional areas to be tested. The test cases that correspond to the functional areas chosen are executed. How to do this will be covered in a future column.

A tester may want to test one or more different functional areas rather than the whole system. By allowing the test runner to specify the functional area(s) as well as the test level, we have a simple, easy to maintain way of slicing and dicing the testing to be run. A tester may say that they want to run a smoke test only on the record sorting area, or they want to all of the available test cases on the interest calculations, or they might even select a single test case to run; any scenario they can come up with should be supported by the suite.

To me, automation should be considered a service to the testing area. We should try to come up with innovative ways to give better service to those who want or need to test. That means coming up with innovations to make them want to run the automation more often.

### **Other Selection Criteria Fields [optional]**

In addition, I allow for extra columns to be added to the header record for extra run selection criteria. For example, I once added a column to a real estate web site test suite that had multi-list values in it; sometimes, the test runner needed to be able to select sets of tests which narrowed down the available listings to be tested.



## Execution Control

### Relative Order

This column contains a numerical value from 1..100. When test cases are selected to run through the prompting scheme, their order of execution is determined by a number of factors. This column allows the tester a certain control over the run order. A test case with a lower value in this column will be executed before all test cases with a higher number. By using a relative system, the automator does not have to micromanage the suite. This comes in handy when adding test cases to an established suite. Often, a new test case may be designed to use data that earlier test cases create or the environment that they set. We could micromanage the suite such that we know exactly what order each test case runs in. That strikes me as overkill, however. We really don't care what order most test cases run in unless there are specific dependencies involved.

Every test case is given a relative number. Most of the assigned numbers are likely to be between 40 and 50. When a new test case is added that will use existing test cases, it is a simple matter to look at the existing test cases, and assign the new test a relative number that is greater than those. The run-time system will order the tests that have been chosen to run by these relative numbers, lower to higher<sup>8</sup>.

### Prerequisites (both)

This field is used in conjunction with the relative order field; while that field is used to cause the tests to run in a certain order, this field is used to ensure that this test case does not run if the prerequisite tests did not complete correctly. Suppose that test X is dependent on tests Q and R running and creating data that will be used in X. It is not enough just to run them in order; it is essential to X running correctly that both Q and R ran correctly. If one or both of them failed, there is a really good chance that X will also fail. Rather than allowing the test to fail, we want a mechanism to prevent it from running at all.

This prereq field allows the test suite administrator to explicitly define prerequisite test cases (any number of them) that must run and pass before this test is allowed to run. This is done by placing the names of all such prereq tests in this field separated by a delimiter (I tend to use a pipe character.) The run-time system, when it comes upon a test case with prereqs, checks to ensure that each test listed did pass correctly during this run. If one or more did not, then the test case is not run. Instead, a message is placed in the log stating that the test was chosen to run, but did not because of a prereq failure. Once again, we will discuss the mechanism in a future column.

---

<sup>8</sup> In a future column, we will discuss how to code this type of run-time behavior. It is easier than you might think.



### **Data-Driven (Main test only)**

This field contains a Boolean value indicating that this particular test case knows how to run data-driven test cases (if set to TRUE), or it is a stand-alone test case or shell script itself. Note that the automation suite can have any combination of DD and stand-alone tests during its evolution.

If this field is marked as TRUE, the unique value is used as a 'join' value to specify all of the tests owned by the script in the data-driven sheet / table. The run-time system is designed to detect this value and automatically cycle through the available DD test cases, running each one in turn. Note that this is another case where the header record is actually determining the way the suite executes. The header records are not just documentation, although they work very well at that task. They also actually determine the way that the suite executes.

### **DoNotRun (both)**

This field allows each test case to be 'commented out'. If the test would otherwise be scheduled to run as part of the suite, a TRUE value in this field will cause the run-time system to simply log that it cannot run (and log the reason, see below), and then skip the test. This might be needed if the test causes a known error, the functionality that the test covers is not working, the test is broken or even not ready yet.

It may seem to be simpler to some to go into the script (or the shell script) and simply comment out the call to this script. Why do we want to be able to specify skipping a test in the header? Once again, the basic philosophy that this header scheme fulfills is that we do not want the administration of the test suite to require an automator. A tester who is non-technical should be able to administrate the suite without having to modify scripts. It is much easier to start Excel and maintain a spreadsheet than it is to find the correct script, make a programmatic change, and save it off.

### **Reason (both)**

This field is a text field that should document the reason a test case is selected not to run using the DoNotRun field. This information is put into the log to ensure that commented out test cases are simply not forgotten. It is my experience that commented-out test cases often do not make it back into the suite; they get forgotten. By documenting the absence of the test each time the suite is run, we keep it in mind for fixing. Of course, if the test is not chosen for execution during the selection process, no log message is written.

### **Data**

#### **Global Values (both)**

This field is a Boolean that, when marked TRUE, indicates that there are variable values stored in the header mechanism. These may be URLs, passwords, high level data, or any other value that can be imported into the test case when the header is read in. Since



accessing the header sheet or data table is much easier than accessing the script itself, I tend to place data here that may change during the testing cycle. Once again, it also allows a non-technical person to set dynamic or changing values without having the programming skills needed to change a script or the experience to go through all of the data tables chasing down the values to change.

If this field is set to TRUE, the unique number of the test indexes into another sheet or table containing fields to be input. The sheet (or table) contains columns that match the field name of the variable. When the run-time system reads a TRUE value in this field, it references the sheet or table and picks up all of the columns in the row that matches the join value. If there is a security concern over passwords in the header, a scheme for encoding the password may be used, or the global field may simply point to another, more secure location where the password can be accessed.

### **Conclusion**

Using headers like this makes administration of the test set manageable. When a new test case is to be added, the test planner simply makes available to the suite administrator some of the information that was used in the planning process. The header for the new test is created using that material.

For this header scheme to be useful, the automator must have some functionality available for dealing with the data store, whether spreadsheets or data table. Each of the major automation tools does supply enough functionality to access spreadsheets or databases if you only know how to use it. Not all of these supplied methods are very elegant, however. My recommendation, and my favorite way of dealing with data store access issues, is to write custom code in a general programming tool and encapsulate it in DLLs. Then, you can call the routines from each automation tool as needed. Some of the necessary routines include:

- Picking up a row of data (from spreadsheet or database) and loading it into an array where the indexes into the array are the column names. The row to be chosen can be specified by number, or by the value in any given column. This is actually how the header record is loaded into the script so that it can control the execution of the script and [optionally] use the global values defined.
- Reading from and writing to Registry, INI files, or flat files.
- Picking up all of the data in a column and returning the distinct values.
- Selecting all of the rows that meet specific criteria.
- Interactive prompts

It should be clearly understood: writing headers and access code, prompts and other advanced techniques that we will discuss in the future has nothing and yet everything to do with automation. Directly, these are not automation techniques. You can easily write



*Innovative solutions to maximize your testing investment.*

an automated test case the same way that our mythical horseless carriage craftsman can build an excellent roundabout.

But single test cases do not solve our testing problems. We need to mass produce test cases and run them efficiently without human intervention. They need to be kicked off by non-experts, evaluated by those who understand testing, not automation. They need to be written by those with few development skills and likely maintained by those with fewer skills. The more clever the development that we use to support our automation, the more likely we are to get long-term, positive ROI from it.