Totally Data-Driven Automated Testing

A White Paper

By

Keith Zambelich

Sr. Software Quality Assurance Analyst

Automated Testing Evangelist

Professional History and Credentials:

I have been involved in Software Testing and Software Quality Assurance for the past 15 years, and have tested a number of software applications on a variety of platforms. I have also been involved in some form of automated testing or another during this period.

At First Interstate Services Corporation, I was involved in testing Electronic Funds Transfer (EFT) applications written in ACP/TPF. Here I developed the "Transaction Simulation Test Facility", which allowed testers to simulate transactions to and from all of First Interstate's connections (VISA, Cirrus, Great Western, etc.). This consisted of over 400 programs written in ACP/TPF, and enabled testers to verify all application modifications, including the front-end switching (Tandem) software.

At the Pacific Stock Exchange, I was in charge of testing all Exchange software, including their new PCOAST trading system. I developed and implemented a method of simulating Broker transactions, eliminating the need for live testing with Brokers. This resulted in a greatly improved implementation success rate.

During my employment at Omnikron Systems, Inc. (a software consulting company based in Calabasas, CA) I successfully implemented Automated Testing solutions using Mercury

Interactive's WinRunner® test tool for a variety of companies, including Transamerica Financial Services, J. D. Edwards Co., IBM Canada, PacifiCare Health Systems, and Automated Data Processing (ADP). While at Omnikron Systems, I developed a totally data-driven method of automated testing that can be applied to any automated testing tool that allows scripting.

I have been certified as a WinRunner® Product Specialist (CPS) by Mercury Interactive, Inc.

Introduction:

The case for automating the Software Testing Process has been made repeatedly and convincingly by numerous testing professionals. Most people involved in the testing of software will agree that the automation of the testing process is not only desirable, but in fact is a necessity given the demands of the current market.

A number of Automated Test Tools have been developed for GUI-based applications as well as Mainframe applications, and several of these are quite good inasmuch as they provide the user with the basic tools required to automate their testing process. Increasingly, however, we have seen companies purchase these tools, only to realize that implementing a cost-effective automated testing solution is far more difficult than it appears. We often hear something like "It looked so easy when the tool vendor (salesperson) did it, but my people couldn't get it to work.",or, "We spent 6 months trying to implement this tool effectively, but we still have to do most of our testing manually.", or, "It takes too long to get everything working properly. It takes less time just to manually test.". The end result, all too often, is that the tool ends up on the shelf as just another "purchasing mistake".

The purpose of this document is to provide the reader with a clear understanding of what is actually required to successfully implement cost-effective automated testing. Rather than engage in a theoretical dissertation on this subject, I have endeavored to be as straightforward and brutally honest as possible in discussing the issues, problems, necessities, and requirements involved in this enterprise.

What is "Automated Testing"?

Simply put, what is meant by "Automated Testing" is automating the *manual* testing process currently in use.This

requires that a formalized "manual testing process" currently exists in your company or organization. Minimally, such a process includes:

Detailed test cases, including predictable "expected results", which have been developed from Business Functional Specifications and Design documentation

A standalone Test Environment, including a Test Database that is restorable to a known constant, such that the test cases are able to be repeated each time there are modifications made to the application

If your current testing process does not include the above points, you are never going to be able to make any effective use of an automated test tool.

So if your "testing methodology" just involves turning the software release over to a "testing group" comprised of "users" or "subject matter experts" who bang on their keyboards in some ad hoc fashion or another, then you should not concern yourself with testing automation. There is no real point in trying to automate something that does not exist. You must *first* establish an effective testing process.

The real use and purpose of automated test tools is to automate *regression testing*. This means that you must have or must develop a database of *detailed* test cases that are *repeatable*, and this suite of tests is run every time there is a change to the application to ensure that the change does not produce unintended consequences.

An "automated test script" is a *program*. Automated script development, to be effective, must be subject to the same rules and standards that are applied to software development. Making effective use of any automated test tool requires at least one trained, *technical* person – in other words, a *programmer*.

Cost-Effective Automated Testing

Automated testing is *expensive* (contrary to what test tool vendors would have you believe). It does *not* replace the need for manual testing or enable you to "down-size" your testing department. Automated testing is an *addition* to your testing process. According to Cem Kaner, in his paper entitled "Improving the Maintainability of Automated Test Suites" (www.kaner.com), it can take between 3 to 10 times as long (or longer) to develop, verify, and document an automated test case than to create and execute a manual test case. This is especially true if you elect to use the "record/playback" feature

(contained in most test tools) as your primary automated testing methodology. Record/Playback is the *least* cost-effective method of automating test cases.

Automated testing can be made to be cost-effective, however, if some common sense is applied to the process:

Choose a test tool that best fits the testing requirements of your organization or company. An "Automated Testing Handbook" is available from the Software Testing Institute (www.ondaweb.com/sti) which covers all of the major considerations involved in choosing the right test tool for your purposes.

Realize that it doesn't make sense to automate some tests. Overly complex tests are often more trouble than they are worth to automate. Concentrate on automating the majority of your tests, which are probably fairly straightforward. Leave the overly complex tests for manual testing.

Only automate tests that are going to be repeated. One-time tests are not worth automating.

Avoid using "Record/Playback" as a method of automating testing. This method is fraught with problems, and is the most costly (time consuming) of all methods over the long term. The record/playback feature of the test tool is useful for determining how the tool is trying to process or deal with the application under test, and can give you some ideas about how to develop your test scripts, but beyond that, its usefulness ends quickly.

Adopt a *data-driven* automated testing methodology. This allows you to develop automated test scripts that are more "generic", requiring only that the input and expected results be updated. There are 2 data-driven methodologies that are useful. I will discuss both of these in detail in this paper.

The Record/Playback Myth

Every automated test tool vendor will tell you that their tool is "easy to use" and that your non-technical user-type testers can easily automate all of their tests by simply recording their actions, and then playing back the recorded scripts. This one statement alone is probably the most responsible for the majority of automated test tool software that is gathering dust on shelves in companies around the world. I would just love to see one of these salespeople try it themselves in a real-world scenario. Here's why it doesn't work:

The scripts resulting from this method contain *hard-coded values* which must change if anything at all changes in the application.

The costs associated with maintaining such scripts are astronomical, and unacceptable.

These scripts are not reliable, even if the application has not changed, and often fail on replay (pop-up windows, messages, and other things can happen that did not happen when the test was recorded).

If the tester makes an error entering data, etc., the test must be re-recorded.

If the application changes, the test must be re-recorded.

All that is being tested are things that *already work*. Areas that have errors are encountered in the recording process (which is manual testing, after all). These bugs are reported, but a script cannot be recorded until the software is corrected. So what are you testing?

After about 2 to 3 months of this nonsense, the tool gets put on the shelf or buried in a desk drawer, and the testers get back to manual testing. The tool vendor couldn't care less – *they are in the business of selling test tools, not testing software*.

Viable Automated Testing Methodologies

Now that we've eliminated Record/Playback as a reasonable long-term automated testing strategy, let's discuss some methodologies that I (as well as others) have found to be effective for automating functional or system testing for most business applications

The "Functional Decomposition" Method

The main concept behind the "Functional Decomposition" script development methodology is to reduce all test cases to their most fundamental tasks, and write *User-Defined Functions, Business Function Scripts,* and *"Sub-routine"* or *"Utility" Scripts* which perform these tasks *independently* of one another. In general, these fundamental areas include:

1. **Navigation (e.g. "Access Payment Screen from Main Menu")**
2. **Specific (Business) Function (e.g. "Post a Payment")**
3. **Data Verification (e.g. "Verify Payment Updates Current Balance")**

4. **Return Navigation (e.g. "Return to Main Menu")**

In order to accomplish this, it is necessary to separate *Data* from *Function*. This allows an automated test script to be written for a *Business Function*, using data-files to provide the both the input and the expected-results verification. A hierarchical architecture is employed, using a structured or modular design.

The highest level is the Driver script, which is the engine of the test. The Driver Script contains a series of calls to one or more "Test Case" scripts. The "Test Case" scripts contain the test case logic, calling the Business Function scripts necessary to perform the application testing. Utility scripts and functions are called as needed by Drivers, Main, and Business Function scripts.

.

- **Driver Scripts:**

  Perform initialization (if required), then call the *Test Case* Scripts in the desired order.

- **Test Case Scripts:**

  Perform the application test case logic using Business Function Scripts

- **Business Function Scripts:**

  Perform specific Business Functions within the application;

- **Subroutine Scripts:**

  Perform application specific tasks required by two or more Business scripts;

- **User-Defined Functions:**

  General, Application-Specific, and Screen-Access Functions;

  Note that Functions can be called from any of the above script types.

Example:

The following steps could constitute a "Post a Payment" Test Case:

1. **Access Payment Screen from Main Menu**
2. **Post a Payment**
3. **Verify Payment Updates Current Balance**
4. **Return to Main Menu**
5. **Access Account Summary Screen from Main Menu**
6. **Verify Account Summary Updates**
7. **Access Transaction History Screen from Account Summary**

8. **Verify Transaction History Updates**

9. **Return to Main Menu**

A "Business Function" script and a "Subroutine" script could be written as follows:

Payment:

Start at Main Menu

- **Invoke a "Screen Navigation Function" to access the Payment Screen**
- **Read a data file containing specific data to enter for this test, and input the data**
- **Press the button or function-key required to Post the payment**
- **Read a data file containing specific expected results data**
- **Compare this data to the data which is currently displayed (actual results)**
- **Write any discrepancies to an Error Report**
- **Press button or key required to return to Main Menu or, if required, invoke a "Screen Navigation Function" to do this.**

Ver-Acct (Verify Account Summary & Transaction History):

- **Start at Main Menu**
- **Invoke a "Screen Navigation Function" to access the Account Summary**
- **Read a data file containing specific expected results data**
- **Compare this data to the data which is currently displayed (actual results)**
- **Write any discrepancies to an Error Report**
- **Press button or key required to access Transaction History**
- **Read a data file containing specific expected results data**
- **Compare this data to the data which is currently displayed (actual results)**
- **Write any discrepancies to an Error Report**
- **Press button or key to return to Main Menu or, invoke a "Screen Navigation Function"**

The "Business Function" and "Subroutine" scripts invoke "User Defined Functions" to perform navigation. The "Test Case" script would call these two scripts, and the Driver Script would call this "Test Case" script some number of times required to perform all the

required Test Cases of this kind. In each case, the only thing that changes are the *data* contained in the files that are read and processed by the "Business Function" and "Subroutine" scripts.

Using this method, if we needed to process 50 different kinds of payments in order to verify all of the possible conditions, then we would need only 4 scripts which are *re-usable for all 50 cases*:

.

1. **The "Driver" script**
2. **The "Test Case" (Post a Payment & Verify Results) script**
3. **The "Payment" Business Function script**
4. **The "Verify Account Summary & Transaction History" Subroutine script**

If we were using Record/Playback, we would now have 50 scripts, each containing hard-coded data, that would have to be maintained.

This method, however, requires only that we add the data-files required for each test, and these can easily be updated/maintained using Notepad or some such text-editor. Note that updating these files does not require any knowledge of the automated tool, scripting, programming, etc. meaning that the non-technical testers can perform this function, while one "technical" tester can create and maintain the automated scripts.

It should be noticed that the "Subroutine" script, which verifies the Account Summary and Transaction History, can also be used by other test cases and business functions (which is why it is classified as a "Subroutine" script rather than a "Business Function" script) – Payment reversals, for example. This means that if we also need to perform 50 "payment reversals", we only need to develop *three* additional scripts.

.

5. **The "Driver" script**
6. **The "Test Case" (Reverse a Payment & Verify Results) script**
7. **The "Payment Reversal" Business Function script**

Since we already had the original 4 scripts, we can quickly clone these three new scripts from the originals (which takes hardly any time at all). We can use the "Subroutine" script as-is without any modifications at all.

It ought to be obvious that this is a much more cost-effective method than the Record/Playback method.

Advantages:

8. **Utilizing a modular design, and using files or records to both input and verify data, reduces redundancy and duplication of effort in creating automated test scripts.**

9. **Scripts may be developed while application development is still in progress. If functionality changes, only the specific "Business Function" script needs to be updated.**

10. **Since scripts are written to perform and test individual Business Functions, they can easily be combined in a "higher level" test script in order to accommodate complex test scenarios.**

11. **Data input/output and expected results is stored as easily maintainable text records. The *user's* expected results are used for verification, which is a requirement for System Testing.**

12. **Functions return "TRUE" or "FALSE" values to the calling script, rather than aborting, allowing for more effective error handling, and increasing the robustness of the test scripts. This, along with a well-designed "recovery" routine, enables "unattended" execution of test scripts.**

Disadvantages:

13. **Requires proficiency in the Scripting language used by the tool (technical personnel);**

14. **Multiple data-files are required for each Test Case. There may be any number of data-inputs and verifications required, depending on how many different screens are accessed. This usually requires data-files to be kept in *separate directories* by Test Case.**

15. **Tester must not only maintain the Detail Test Plan with specific data, but must also re-enter this data in the various required data-files.**

    1. **If a simple "text editor" such as *Notepad* is used to create and maintain the data-files, careful attention must be paid to the format required by the scripts/functions that process the files, or script-processing errors will occur due to data-file format and/or content being incorrect.**

---

1. The "Key-Word Driven" or "Test Plan Driven" Method

This method uses the actual Test Case document developed by the tester using a spreadsheet containing special "Key-Words". This method preserves most of the advantages of the "Functional Decomposition" method, while eliminating most of the disadvantages. In this method, the entire process is *data-driven*,

including functionality. *The Key Words control the processing*.

Consider the following example of our previous "Post a Payment" Test Case:

.

.

COLUMN 1 Key_Word	**COLUMN 2** Field/Screen Name	**COLUMN 3** Input/Verification Data	**COLUMN 4** C
Start_Test:	Screen	Main Menu	Verify Starting P
Enter:	Selection	3	Select Payment
Action:	Press_Key	F4	Access Payment
Verify:	Screen	Payment Posting	Verify Screen ac
Enter:	Payment Amount	125.87	Enter Payment d
	Payment Method	Check	
Action:	Press_Key	F9	Process Paymen
Verify:	Screen	Payment Screen	Verify screen ren

Verify_Data:	**Payment Amount**	**$ 125.87**	**Verify updated d**
	Current Balance	**$1,309.77**	
	Status Message	**Payment Posted**	
Action:	**Press_Key**	**F12**	**Return to Main M**
Verify:	**Screen**	**Main Menu**	**Verify return to M**

.

Each of the "Key Words" in Column 1 causes a "*Utility Script*" to be called which processes the remaining columns as input parameters in order to perform specific functions. Note that this could also be run as a manual test. The test engineer must develop and document the test case anyway – *why not create the automated test case at the same time*?

The data in red indicates what would need to be changed if one were to copy this test case to create additional tests.

.

How in the World Does This Work?

- **"Templates" like the example above are created using a spreadsheet program (e.g. MS Excel®), and then copied to create additional test cases.**
- **The Spreadsheet is saved as a "tab-delimited" file (input/verification data often contains "commas", so "tab-delimited" is preferable).**
- **This file is read by a "Controller" script for the application, and processed. When a *Key Word* is encountered, a list is created using data from the remaining columns. This continues until a "null" (blank) in column-2 is encountered.**
- **The "Controller" script then calls a Utility Script *associated with the Key Word*, and passes the "list" as an *input parameter*.**
- **The Utility Script continues processing until "end-of-list", then returns to the "Controller" script, which continues processing the file until "end-of-file" is reached.**

Architecture

The architecture of the "Test Plan Driven" method appears similar to that of the "Functional Decomposition" method, but in fact, they are substantially different:

- **Driver Script**
  - **Performs initialization, if required;**
  - **Calls the Application-Specific "Controller" Script, passing to it the file-names of the Test Cases (which have been saved from the spreadsheets as a "tab-delimited" files);**
- **The "Controller" Script**
  - **Reads and processes the file-name received from Driver;**
  - **Matches on "Key Words" contained in the input-file**
  - **Builds a *parameter-list* from the records that follow;**
  - **Calls "Utility" scripts associated with the "Key Words", passing the created parameter-list;**
- **Utility Scripts**
  - **Process input parameter-list received from the "Controller" script;**
  - **Perform specific tasks (e.g. press a key or button, enter data, verify data, etc.), calling "User Defined Functions" if required;**
  - **Report any errors to a Test Report for the test case;**
  - **Return to "Controller" script;**
- **User Defined Functions**
  - **General and Application-Specific functions may be called by any of the above script-types in order to perform specific tasks;**

Advantages:

This method has all of the advantages of the "Functional Decomposition" method, as well as the following:

1. **The Detail Test Plan can be written in *Spreadsheet* format containing all input and verification data. Therefore the tester only needs to write this *once*, rather than, for example, writing it in *Word*, and then creating input and verification files as is required by the "Functional Decomposition" method.**
2. **Test Plan does not necessarily have to be written using *Excel*. Any format can be used from which either "tab-delimited" or "comma-delimited" files can be saved (e.g. Access Database, etc.).**

3. **If "utility" scripts can be created by someone proficient in the Automated tool's Scripting language *prior* to the Detail Test Plan being written, then the tester can use the Automated Test Tool *immediately* via the "spreadsheet-input" method, without needing to learn the Scripting language. The tester need only learn the "Key Words" required, and the specific format to use within the Test Plan. This allows the tester to be productive with the test tool very quickly, and allows more extensive training in the test tool to be scheduled at a more convenient time.**

4. **If the Detail Test Plan *already exists* in some other format, it is not difficult to translate this into the "spreadsheet" format.**

5. **After a number of "generic" Utility scripts have already been created for testing an application, we can usually re-use most of these if we need to test another application. This would allow the organization to get their automated testing "up and running" (for most applications) within a few days, rather than weeks.**

Disadvantages:

1. **Development of "customized" (Application-Specific) Functions and Utilities requires proficiency in the tool's Scripting language. Note that this is also true of the "Functional Decomposition" method, and, frankly of any method used *including* "Record/Playback".**

2. **If application requires more than a few "customized" Utilities, this will require the tester to learn a number of "Key Words" and special formats. This can be time-consuming, and may have an *initial* impact on Test Plan Development. Once the testers get used to this, however, the time required to produce a test case is greatly improved.**

Cost-Effectiveness:

In the example we gave using the "Functional Decomposition" method, it was shown that we could use previously created "Test Case" and "Business Function" scripts to create scripts for additional Test Cases and Business Functions. If we have 100 Business Functions to test, this means that we must create a minimum of 200 scripts (100 Test Case scripts, and 100 Business Function scripts).

Using the "Test Plan Driven" method, I currently am using 20 Utility scripts, that I have been able to use in *every single automated testing engagement* that I have been sent on. Let us examine what it takes on an average for me to walk into a company, and implement Automated Testing:

- **Normally, I have to create a minimum of 3 application-specific utility scripts (a "Controller" script, a "Start_Up" script, and an "End_Test" script).**

- **I may also have to create application-specific versions of several of the 20 "general" Utility scripts.**

- **It is also usually necessary to develop between 10 and 20 application-specific "functions" (depending on how complex or strange the application under test is). These functions include such things as activating and shutting down the application, logging in, logging out, recovering from**

unexpected errors ("return to main window"), handling objects that the tool doesn't recognize, etc.

- **A number of "prototype" test cases must be created as a "proof of concept". This includes developing the spreadsheet data. Sometimes test cases that have already been developed can be used, other times I have to create the test cases myself.**

  Depending on the complexity of the application, and how well the test tool works with the application, this process normally takes me no more than 3 days – 5 days is worst-case. At this point, testers can be trained to create the spreadsheet data (usually takes about a week) and then they are in business. It also takes about a week to train the "test tool technician" to use this methodology, provided that this person is a relatively competent programmer and has already been sufficiently trained by the tool vendor in the use of the tool.

  > **What this demonstrates is that an organization can implement cost-effective automated testing if they go about it the right way.**

Managing Resistance to Change

One of the main reasons organizations fail at implementing automated testing (apart from getting mired down in the "record/playback" quagmire) is that most testers do not welcome what they *perceive* as a fundamental change to the way they are going to have to approach their jobs. Typically, decisions as to what tool to use and how to implement it are made by management, often without consulting the people who are actually doing the testing, and who are now going to have to cope with all of this. This usually meets with a great deal of resistance from the testers, especially when management does not have a clearly defined idea of how to implement these changes effectively. Let us examine some concerns that might be expressed by testers, and some answers to these:

- **The tool is going to replace the testers**

  This is not even remotely true. The automated testing tool is just another tool that will allow testers to do their jobs better by:

  - **Performing the boring-type test cases that they now have to do over and over again**
  - **Freeing up some of their time so that they can create better, more effective test cases**

  The testers are still going to have to perform tests manually for specific application changes. Some of these tests may be automated afterward for regression testing.

- **It will take too long to train all of the testers to use the tool**

If the "test-plan-driven" method (described above) is used, the testers will not have to learn how to use the tool at all if they don't want to. All that they have to learn is a different method of documenting the detailed test cases, using the key-word/spreadsheet format. It is not that different from what they are doing currently, and takes only a few hours to learn.

- **The tool will be too difficult for testers to use**

  Perhaps, but as we have already discussed, they will not have to use it. What will be required is that a "Test Tool Specialist" will need to be hired and trained to use the tool. This can either be a person who is already an expert with the particular tool, or can be a senior-level programmer who can easily be trained to use it. Most test-tool vendors offer training courses.

  The "test-plan-driven" testing method will eliminate most of the testers' concerns regarding automated testing. They will perform their jobs exactly as they do now. They will only need to learn a different method of documenting their test cases.

.

.

<u>Staffing Requirements</u>

One area that organizations desiring to automate testing seem to consistently miss is the staffing issue. Automated test tools use "scripts" which automatically execute test cases. As I mentioned earlier in this paper, these "test scripts" are *programs*. They are written in whatever scripting language the tool uses. This might be C++ or Visual Basic, or some language unique to the test tool. Since these are programs, they must be managed in the same way that application code is managed.

To accomplish this, a "Test Tool Specialist" or "Automated Testing Engineer" or some such position must be created and staffed with at least one senior-level programmer. It does not really matter what languages the programmer is proficient in. What does matter, is that this person must be capable of designing , developing, testing, debugging, and documenting code. More importantly, this person must <u>want</u> to do this job – most programmers want nothing to do with the Testing Department. This is not going to be easy, but it is nonetheless absolutely *critical*. In addition to developing automated scripts and functions, this person must be responsible for:

- **Developing standards and procedures for automated script/function development**
- **Developing change-management procedures for automated script/function implementation**
- **Developing the details and infrastructure for a data-driven testing method**
- **Testing, implementing, and Managing the test scripts (spreadsheets) written by the testers**

- **Running the automated tests, and providing the testers with the results**

It is often useful to hire a contractor (like me) who knows how to set this all up, help train the "Automation Engineer", and the testing staff, and basically get things rolling. In my experience, this can take from two to three weeks, or as long as two to three months, depending on the situation. In any case, it should be a short-term assignment, and if you find someone who really knows what they're doing, it will be well worth it.

It is worth noting that no special "status" should granted to the automation tester(s). The non-technical testers are just as important to the process, and favoritism toward one or the other is counter-productive and should be avoided. Software Testing is a profession, and as such, test engineers should be treated as professionals. It takes just as much creativity, brain power, and expertise to develop effective, detailed test cases from business and design specifications as it does to write code. I have done both, and can speak from experience.

.

.

<u>Summary:</u>

- **Establish clear and reasonable expectations as to what can and what cannot be accomplished with automated testing in your organization.**
    - **Educate yourself on the subject of automated testing. Many independent articles have been written on the subject. Get a clear idea of what you are really getting into.**
    - **Establish what percentage of your tests are good candidates for automation**
        - **Eliminate overly complex or one-of-a kind tests as candidates**
- **Get a clear understanding of the requirements which must be met in order to be successful with automated testing**
    - **Technical personnel are required to use the tool effectively**
    - **An effective manual testing process must exist before automation is possible. "Ad hoc" testing cannot be automated. You should have:**
        - **Detailed, repeatable test cases, which contain exact expected results**
        - **A standalone test environment with a restorable database**
    - **You are probably are going to require short-term assistance from an outside company which specializes in setting up automated testing or a contractor experienced in the test tool being used.**
- **Adopt a viable, cost-effective methodology.**

- **Record/Playback is too costly to maintain and is ineffective in the long term**
- **Functional Decomposition method is workable, but is not as cost-effective as a totally data-driven method**
- **Test Plan driven method is the most cost-effective:**
  - **Requires a minimum of technical personnel**
  - **Requires the fewest automated scripts**
  - **Introduces the least amount of change for the manual testers**
  - **Test cases developed can be used for manual testing, and are automatically usable for automated testing**

- **Select a tool that will allow you to implement automated testing in a way that conforms to your long-term testing strategy. Make sure the vendor can provide training and support.**