# Getting More Mileage Out Of Your Automation

**By**
**Kelly Whitmill**

**IBM Printing Systems Division**
**6300 Diagonal Highway – 003G**
**Boulder, Colorado 80301**

**Email: whitmill@us.ibm.com**

# Getting More Mileage Out of Your Automation

## Abstract

Normally, by the time you have your test automated, it has already found most of the bugs that it will ever find. How do you get automated testing to be more effective, productive and actually improve with age?

In a very practical way, this paper will introduce:

- How to add real-time variety to your test cases.
- How to add enough randomness to your test cases to keep them from being stale, worn out and ineffective, yet still be targeted and meaningful.
- What you can do with those data-driven test cases to make them more dynamic and productive.
- Test case generation for the common practitioner. You don't have to have formal models, fancy tools, and be a guru from research to employ test case generation.
- A testing mindset. How you think about testing may either open new doors or set up road blocks. Some of those old test paradigms may be stumbling blocks that are getting in the way of your best testing. Food for thought will be provided that may open up new avenues for you.
- Hints, guidelines, and tradeoffs for automated verification. Ultimately, you have to know whether the test case passed or failed. How do you do automated verification when you can't know the expected results until runtime?

Don't settle for rerunning the same test cases over and over again. Get more mileage out of your automation!

**Introduction**

Do you really believe that running test cases for problems that have already been found and fixed is your most productive means of testing? What other business would consider it good practice to spend enormous amounts of resources to merely try to rediscover problems that have already been found and fixed? Do you think that re-running test cases that didn't find anything previously is a productive means of testing? If you step back and consider most automated test efforts, you will realize that they are architected to not teach us anything new about the software. We spend vast resources to relearn what we already knew.

Many articles on automation suggest that if the test is not repetitive and mundane then it is not a good candidate for automation. Though automated regression tests have their place in certifications and in fallible processes that cause breakage, it is evident that automated testing must become more effective. Too many automated tests drop by the wayside because they are too expensive to keep up to date. Too many automated tests judge their worth by how many times they run rather than what they teach us.

Each of the following items will be presented to show how they can enhance your automation.

- Probabilities and Weights
- Intelligent Randomness
- Data-driven Testing
- Generate Test Cases
- Varying the Environment
- Automated Verification
- Assessment Philosophy

**Probabilities and Weights**

Automated tests can be made much more dynamic and flexible by adding probabilities and/or weights. Simply put, instead of forcing test cases down the same tired path every time, throw in some variety.

For example, take something as simple as bringing up a document in a word processor. Based on probabilities/weights you can bring up a) an empty document, b) a small document, c) a medium sized document or d) a large document. You could also invoke the word processor via either GUI commands or the command line or by double clicking on the file. Consider what would happen if, based on probabilities, you chose which functions to performs on the document and the order to perform the functions. If the selected function is to insert text, you may, based on probabilities/weights, determine where in the document to insert the text, the type and amount of text to insert, and the manner of saving the text. It may be one test case, but you could run it numerous times, and each time have the possibility of learning something new about the word processor.

Your test cases start looking something like the following: (Note: This is not representing any particular programming language)

```
While (not done testing)
Do
        Switch(SelectAction())
                Case NoAction:        DoNoAction()
                Case InsertText:      Call InsertText()
                Case DeleteText:      Call DeleteText()
                Case ChangeFormat:    Call ChangeFormat()
                        .
                        .
                        .
                Case DoXxx():         Call DoXxx()
End
```

To accomplish this we need a routine to select an action based on a probability or weight.

A simplified SelectAction() could look something like this:

```
SelectAction()
Num = GenerateRandomNumber(1,100)
Case Num:
        1-2:     return NoAction
        3-20:   return InsertText
        21-25: return DeleteText
        26-30: return ChangeFormat
                  .
                  .
                  .
        92-100: return DoXxx
```

Another approach to randomly select actions could be something like the following:

```
Switch(SelectAction(10,10,20,50,10))
        Case 1:  DoNoAction()
        Case 2: Call InsertText()
        Case 3: Call DeleteText()
        Case 4: Call ChangeFormat()
        Case 5: Call DoXxx()
```

This SelectAction() sums all the probabilities/weights specified as input. The SelectAction() function then generates a number between 0 and the total inclusive. The first case whose weight/probability summed with all the previous weights/probabilities is greater than or equal to the randomly selected number is the case chosen.
This second form of SelectAction(arg1,arg2,…,argn) is a more generalized implementation and is easier to use when modifying probabilities/weights on the fly.

Each randomly selected action may use weights and probabilities to decide their specific actions, attributes, parameters and so forth. For example, InsertText() may choose whether to insert at the beginning, end, or somewhere in the middle of the document. It may choose the type and amount of text to insert and so forth.

You can see that even if you ran this multiple times, each time it may look and act like a new test case. Each time you run, your test case may consist of a different set of actions, actions run in a different order, and actions implemented with differing details.

However, this is only the beginning of the power and flexibility that probabilities and weights can add to your test cases.

- You can adjust the probabilities and weights to match the usage profiles of your customers. You can adjust and readjust multiple times to represent the usage profiles of different types of customers. If a set of customers tend to do a lot more inserting and deleting you could increase the probabilities of those actions. If another set of customers emphasizes another set of actions then you can adjust the probabilities/weights to reflect their usage profile.
- You can set the probability to zero for an item that is not yet ready or is broken and shouldn't be tested until it is fixed.
- If you want to focus in on certain areas of testing you can increase their probabilities/weights.
- This enables automation in an incremental approach. For example, add automation for a base set of functions. As time permits, you add automation for more functions. With this approach, if 20% is automated, 20% is runnable. You don't have to wait for everything to be automated.
- This enables updates to update and improve all tests. If you add another function to your word processor you don't have to go back and update numerous test cases. You can update your loop for selecting actions and all test cases are automatically updated to incorporate that function. Thus your test cases improve with age rather than diminish in usefulness over time.

If needed, you can add scripted test cases (hand written or generated) as a selectable action.

I know what you are thinking, but don't worry. Yes, you can still verify the results, and it is not an undue burden, but that will be covered later.

**Intelligent Randomness**
Intelligent randomness is interspersing random commands and actions that make sense in context throughout the tasks.
This is probably the most important key. This is what gives life and mileage to test cases. Test cases wear out because they get stale. They do things the exact same way, with the exact same sequence, in the exact same state, over and over. We intentionally make them do this so that they are repeatable and predictable. Unfortunately, they are too often repeatably and predictably ineffective. Fifty percent of the time throw in random actions, but actions that make sense. What if immediately before or after inserting text we did some other action or actions in our word processor. It could be anything that makes sense, deleting text, highlighting text, importing, exporting, changing the font, creating a new page, starting another application, clicking on certain areas of the document and so forth. Could your insert text test take on new meaning if it is on a new page? Could you have written separate test cases for all of these variations anyway? Only if you have infinite time and resources.

If test cases are set up like the example in probabilities/weights you get a lot of this randomness naturally. However, you still need to consider adding in some additional actions that may or may not be directly related to your test case. For example, you may need to add in a ThinkTime() action. Users don't execute everything at computer speed. It is sometimes useful to model the customer behavior and add some idle time to your test cases. This is especially important in web based applications. Consider all the things that could happen before, during, and after the action/task./function you are testing. Throw in as many of those things as are reasonable.

More typically, automated test cases are written for specific tasks or functions. To rerun the same task over and over again is not likely to teach you something new each time. Even when you are randomly picking which task to run and the order to run them in, test cases tend toward ineffectiveness because they are just reruns of the same thing. We really shouldn't expect to learn anything new, unless we change something.

An example case study is an automation effort to test a print server system. We automated multiple tasks such as create a printer definition, delete printers, submit print jobs, move print jobs, etc.. Based on usage profiles, we randomly selected the order of running these automated tasks. After only a short while, the test cases stopped finding any new bugs. After considerable experimentation, we discovered that if we randomly executed print server commands in between tasks the test cases started finding errors again. The automation would, based on the current state of the system, do random actions that made sense (thus the term intelligent randomness). For example, it would submit jobs to existing printers, delete printers, change printers, delete jobs and so forth. Now, the task test cases started teaching us new things. The task to delete a printer would behave differently if jobs were on the printer than if there were no jobs. It may behave differently based on the number of jobs and the type of jobs queued to the printer. The "move job' task took on different meaning if the destination printer was just deleted, or if it was in a held state or active state or processing state etc.. The task verification had to modified, but not significantly. The lesson learned is that existing automated test cases can be much more effective if you surround them with intelligent randomness, (i.e. actions/commands that make sense in context). The more likely the random commands are to affect the state or operation of the tasks the more likely they are to reveal new information when the task test case is run. We found that we needed as many random actions as we did automated tasks. This trend has been consistent with follow-on projects. Your situation may vary, but the concept seems to apply to many if not most test situations.

When using intelligent randomness, there can be a tendency for the system under test to drift to a state that is not conducive to testing. For our word processor test, the document may tend to become too large or to have too little data to be meaningful. In the print server example, there may be too many printers in a non-printing state. In these cases, you need to periodically reset the environment to a more conducive state for testing. The print server automation may periodically need to make sure at least 70% of the printers are in a state to be able to print. In the word processor example, periodically you may need to load or open up a new document with many of the attributes that are important to your test

**Data Driven Testing**
You know the routine here. Don't hard code the data into the test case. However, here is the twist. Don't use the same old data file over and over again either. Generate new data.

Put new files (perhaps updated files that better reflect the customer) in the directory where you randomly select files to open. Randomly generate the text to insert. Have a data file of example text segments to insert, but keep updating it with new, and hopefully better, but at least different inputs for the test case to use.

A table of static data used as data-driven input for a test case is not much different than hard-coding the data into the test case. If our data doesn't change and improve as time goes on we shouldn't expect our testing to improve as time goes on either.

At the very least, we should change or add to the data that is driving the test cases. Better yet, select the data to be used from a pool of data that can be constantly updated. For example, don't put the name of the document to open in table to be read into the test case. Rather, select a document from a directory of documents. As you acquire good customer data add it to the document directory and it is automatically integrated into your tests.

**Generate Test Cases**

If you write your test cases using a series of probabilities and weights, you are generating a new set of test cases each time you run. If you must rerun old test cases, save the seed. Using the seed, you can regenerate and rerun the same tests. No formal models are needed, just some common programming skills that you have anyway if you are automating.

If you generate test cases at run time then you eliminate much of the complexity needed for test cases generated ahead of time. If you generate test cases prior to run time you have to keep track of all the expected states of the elements you are working with so that you can generate actions that make sense for the current context. If you generate at run time then you can usually bypass tracking all that state information. Instead, you can typically make queries to determine appropriate actions and expected results.

For example, it is easier to just query a printer for its state and generate an appropriate action than it is to track every state change for every printer. It is easier to check the read/write status of a file when you need it than to record and track the state of file throughout all of your testing.

**Vary the Environment/Configuration/Inputs**

Rerunning the same test cases in different environments can reveal a lot of new information and extend the usefulness of your test cases. When designing your test cases consider what inputs, configuration elements, and environmental considerations will have a tendency to vary in the customer's environments.

Running a "create printer" test case may behave differently in an environment with one printer than in an environment with 10,000 printers. Inserting text into a text only page may behave differently than inserting text into a mixed text/image page.

Even if the test case stays the same, when you open a different file for editing, you may learn something new. In testing a Print Server system, you could run the same test cases with different types, numbers, and states of printers. By running the same test cases with a variety of printers intended to resemble different customers, you would learn new information about how your code would behave for different customers.

**Automated Verification**

Automated verification is hard, but perhaps not as hard -- or as impossible -- as you might think. Even if you randomly opened a file, randomly selected text to insert, randomly selected a point to insert it, it is still not that hard to look at the saved file and see if the text you selected is saved there. You don't have to over verify. Many times you can write simple verification routines, as in the inserted text example. Sometimes you will need to keep track of object names and states, but even this is still a practical exercise.

There is no cookbook answer as to how to do automated verification. However, there are a few principles to apply.

Principle 1: Just Enough Verification

You don't have to verify everything and you don't need to verify everything. Consider what is essential to verify and just verify that. For example, when inserting text, your typical approach might be to insert text and then compare the entire document to a master with inserted text. You won't be able to do that if the content is dynamic. You don't need to do that. It may be enough to verify that the document contains the inserted text. If you are concerned about side effects in other parts of the document, you may check the overall byte count of the document to see that it incremented the correct number or some similar verification activity. Automation is expensive. Verification is one of the most difficult and expensive parts of automation. You need to provide just enough verification, but no more.

Principle 2: Build on Less Dynamic Testing

You may use hard-coded, less dynamic testing or even manual testing to show correctness of an element then base further automated testing on the assumption that the element works correctly. For example, you may hard code a single test case to test inserting text. This test case may verify the contents of the entire document to ensure there are no unintended side effects in the rest of the document. Then the remainder of your automated test cases just test that the intended data is inserted, but not try to detect if there were any side effects throughout the rest of the document.

Principle 3: Sometimes Less is More

Even though the verification for a more dynamic test may be less precise than a hard coded traditional test, it may still tell you more in the long run. (Note: It is not a given that more dynamic testing will be less precise, but it is a condition that will not be uncommon.) In the inserted text example, you may give up detecting a side effect that pops up in the document, but you will learn how inserting text works in so many different contexts that you wouldn't have known before. You won't have the over burdening maintenance of master files or equivalent costs associated with 100% verification. This follows the Pareto model. You can get 80% of the benefit for 20% of the cost. It will probably cost you the extra 80% to go after the 20% more precision in verification.

When considering tradeoffs in automated verification, don't just look at what can't be verified with a more dynamic approach. Look at all the new things that will be exposed and verified using a more dynamic approach.

Principle 4: State Matters

It is often the case that you will need some state information to help determine the expected results. Much of the state information can be determined on the fly. Some of it you will have to keep track of. To determine if a print submission should work or not, it would be necessary to know if the printer exists and is in a condition to successfully print. This can be checked on the fly, the print submission exercised and then verified for correctness.

When saving a document in a word processor, it may be necessary to know if the file is read-only. This can be determined on the fly.

If the state or condition cannot be determined at run-time, it may be necessary for the automation to track that information.

More dynamic verification may require a change in mindset. It may require a different set of tradeoffs than we are accustomed to. It will also lead to more effective automated testing than we are accustomed to.

**Assessment Philosophy**

Are you testing to "test in quality" or to "assess quality"? The answer is probably both. But, which takes priority? How often have you heard that every test must start and end in a known state? How many customers do that? The "known-state" approach is good for re-creates and bad for assessing how the code works in a customer environment. Perhaps in our system tests we should lean more towards running tests that build upon each other and not always operate in predictable and known states, and thus get a better assessment of how the code will work for the customer. This change in philosophy may in itself change the structure and effectiveness of tests to continue finding problems. Though it may be difficult to fix a problem that is hard to re-create, you have no chance of fixing a problem that you don't even know exists.

My experience has been that a testing philosophy that puts first priority on maximizing the effectiveness of finding defects then doing everything you can to accommodate fixing defects within that context is more productive than prioritizing fixing defects. A test environment that is more realistic and doesn't depend on tests always starting and stopping in known states will do a better job of  exposing risks and provide earlier opportunities to be aware of and deal with the hard problems.

## Filling In the Gaps

There will be some additional items to consider in your automation. How do you know what has been tested? In practice, we have found it useful to have counters that track each task, options, etc. Each time an item is exercised it is recorded along with a message as to whether it was used successfully or not. You can then generate reports to show what has and has not been tested, as well as how much an item has been tested.

You will need to add very good logging so that you can have a good trail of what happened when an error occurs and needs debugging.

## Summary

Each key by itself seems rather unremarkable. Together they are powerful and effective in getting more mileage out of your automation. Instead of wearing out over time, the automation improves with age. Considering the significant investment required for automation, it makes sense to start making those tests more productive. The known-state, repeatable regression tests have their place, but there are many opportunities for more productive testing that are being ignored. Now you have some keys to start getting better mileage out of your automation.