Common Mistakes in Test Automation

Mark Fewster
Grove Consultants
Llwyncynhwyra
Cwmdu
Llandeilo
SA19 7EW
UK

Tel: +44 1558 685180
Fax: +44 1558 685181
Email: mark@grove.co.uk
www.grove.co.uk

Abstract

Automating the execution of tests is becoming more and more popular as the need to improve software quality amidst increasing system complexity becomes ever stronger. The appeal of having the computer run the tests in a fraction of the time it takes to perform them manually has led many organisations to attempt test automation without a clear understanding of all that is involved.

Consequently, many attempts have failed to achieve real or lasting benefits. This paper highlights a few of the more common mistakes that have contributed to these failures and offers some thoughts on how they may be avoided.

1. Confusing automation and testing

Testing is a skill. While this may come as a surprise to some people it is a simple fact. For any system there are an astronomical number of possible test cases and yet practically we have time to run only a very small number of them. Yet this small number of test cases is expected to find most of the bugs in the software, so the job of selecting which test cases to build and run is an important one. Both experiment and experience has told us that selecting test cases at random is not an effective approach to testing. A more thoughtful approach is required if good test cases are to be developed.

What exactly is a good test case? Well, there are four attributes that describe the quality of a test case, that is, how good it is. Perhaps the most important of these is its effectiveness, whether or not it finds bugs, or at least, whether or not it is likely to find bugs. Another attribute reflects how much the test case does. A good test case should be exemplary, that is, it should test more than one thing thereby reducing the total number test cases required. The other two attributes are both cost considerations: how economical a test case is to perform, analyse and debug; and how evolvable it is, that is, how much maintenance effort is required on the test case each time the software changes.

These four attributes must often be balanced one against another. For example, a single test case that tests a lot of things is likely to cost a lot to perform, analyse and debug. It may also require a lot of maintenance each time the software changes. Thus a high measure on the exemplary scale is likely to result in low measures on the economic and evolvable scales.

Thus testing is indeed a skill, not only must testers ensure that the test cases they use are going to find a high proportion of the bugs but they must also ensure that the test cases are well designed to avoid excessive costs.

Automating tests is also a skill but a very different skill which often requires a lot of effort. For most organisations it is expensive to automate a test compared with the cost of performing it once manually, so they have to ensure that each test automated will need to be performed many times throughout its useful life.

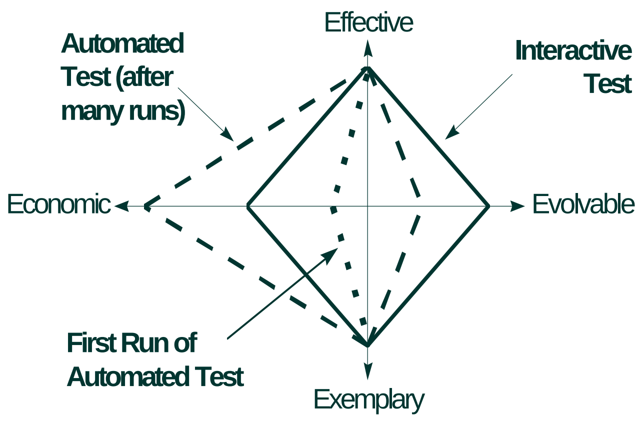

Figure 1 The 'goodness' of a test case can be illustrated by considering the four attributes in this Keviat diagram. The greater the measure of each attribute the greater the area enclosed by the joining lines and the better the test case.

Whether a test is automated or performed manually affects neither its effectiveness nor how exemplary it is. It doesn't matter how clever you are at automating a test or how well you do it, if the test itself achieves nothing then all you end up with is a test that achieves nothing faster.

Automating a test affects only how economic and evolvable it is. Once implemented, an automated test is generally much more economic, the cost of running it being a mere fraction of the effort to perform it manually. However, automated tests generally cost more to create and maintain. The better the approach to automating tests the cheaper it will be to implement new automated test in the long term. Similarly, if no thought is given to maintenance when tests are automated, updating an entire automated test suite can cost as much, if not more, than the cost of performing all the tests manually.

For an effective and efficient automated suite of tests you have to start with the raw ingredient of a good test suite, a set of tests skilfully designed by a tester to exercise the most important things. You then have to apply automation skills to automate the tests in such a way that they can be created and maintained at a reasonable cost.

Figure 1 depicts the four quality attributes of a test case in a Keviat diagram and compares the likely measures of each on the same test case when it is performed manually (shown as an interactive test in the figure) and after it has been automated.

2. Believe capture/replay = automation

Capture / replay technology is indeed a useful part of test automation but it is only a very small part of it. The ability to capture all the keystrokes and mouse movements a tester makes is an enticing proposition, particularly when these exact keystrokes and mouse movements can be replayed by the tool time and time again. The test tool records the information in a file called a script. When it is replayed, the tool reads the script and passes the same inputs and mouse movements on to the software under test which usually has no idea that it is tool controlling it rather than a real person sat at the keyboard. In addition, the test tool generates a log file, recording precise information on when the replay was performed and perhaps some details of the machine. Figure 2 depicts the replay of a single test case.

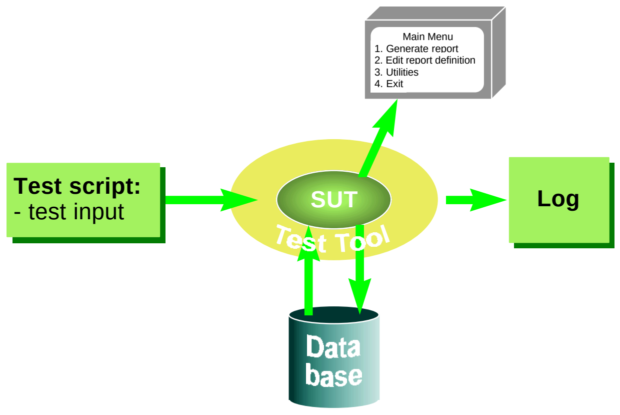

Figure 2 Capture/replay tools offer an inviting way to automate tests but it is checking the results that may be overlooked.

For many people this seems to be all that is required to automate tests. After all, what else is there to testing but entering a whole series of inputs? However, merely replaying the captured input to the software under test does not amount to performing a whole test.

For a start there is no verification of the results. How will we know if the software generated the same outputs? If the tester is required to sit and watch each test be replayed he or she may as well have been typing them in as they are unlikely to be able to keep up with the progress of the tool, particularly if it is a long test. It is necessary for the tool to perform some checking of the output from the application to determine that its behaviour is the same as when the inputs were first recorded. This implies that as well as recording the inputs the tool must record at least some of the output from the software under test. But which particular outputs? How often and is an exact match required every time? These are questions that have to be answered by the tester as the inputs are captured, or possibly (depending on the particular test tool in use) during a replay.

Alternatively, the testers may prefer to edit the script, inserting the required instructions to the tool to perform comparison between the actual output from the software under test and the expected output now determined by the tester. This pre-supposes that the tester will be able to understand the script sufficiently well to make the right changes in the right places. It also assumes that the tester will know exactly what instructions to edit into the script, their precise syntax and how to specify the expected output.

In either approach, the tests themselves may not end up as particularly good tests. Even if it was thought out carefully at the start, the omission of just one important comparison or the inclusion of one unnecessary or erroneous comparison can destroy a good test. Such tests may then never spot that important bug or may repeatedly fail good software.

Scripts generated by testing tools are usually not very readable. Well, OK, they may be readable ("click left mouse button", "enter 17645", and "click OK") but will the whole serious of possibly hundreds of individual actions really convey what has been going on and where comparison instructions are to be inserted? Scripts are programming languages so anyone editing them has to have some understanding of programming. Also, it may be possible for the person who has just recorded the script to understand it immediately after they have recorded it, but after some time has elapsed or for anyone else this will be rather more difficult.

Even if the comparison instructions are inserted by the tool under the testers control, the script is likely to need editing at some stage in its life. This is most likely when the software under test changes. A new field here, a new window there, will soon cause untold misery for testers who then have to trawl through each of their recorded scripts looking for the places that need updating. Of course, the scripts could be re-recorded but then this rather defeats the object of recording them in the first place.

Recording test cases that are performed once manually so they can be replayed is a cheap way of starting test automation which is probably why it is so appealing to those who opt for this approach. However, as they soon discover, even if they do overcome the test quality problems the cost of maintaining the automated tests becomes prohibitive as soon as the software changes. If we are to minimise the growing test maintenance costs, it is necessary to invest more effort up front implementing automated tests in a way that is designed to avoid maintenance costs rather than avoid implementation costs. Figure 3 depicts this in the form of a graph.

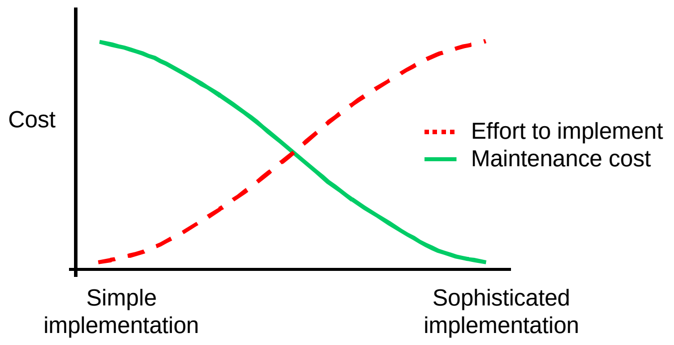

Figure 3 The cost of test maintenance is related to the cost of test implementation. It is necessary to spend time building the test in order to avoid high maintenance costs later on.

3. Verify only screen based information

Testers are often only seen sat in front of a computer screen so it is perhaps natural to assume that it only the information that is output to the screen by the software under test that it checked. This view is further strengthened by many of the testing tools that make it particularly easy to check information that appears on the screen both during a test and after it has been executed.

However, this assumes that a correct screen display means that all is OK, but it is often the output that ends up elsewhere that is far more important. Just because information appears on the screen correctly does not always guarantee that it will be recorded elsewhere correctly.

For good testing it is often necessary to check these other outputs from the software under test. Perhaps not only the files and database records that have been created and changed, but also those that have not been changed and those that have (or at least should have) been deleted or removed. Checking some of these other aspects of the outcome of a test (rather than merely the output) will make tests more sensitive to unexpected changes and help ensure that more bugs are found.

Without a good mechanism to enable comparison of results other than those that appear on the screen, tests that undertake these comparisons can become very complex and unwieldy. A common solution is to have the information presented on the screen after the test has completed. This is the subject of the next common mistake.

4. Use only screen based comparison

Many testing tools make screen based comparisons very easy indeed. It is a simple matter of capturing the display on a screen or a portion of it and instructing the tool to make the same capture at the same point in the test and compare the result with the original version. As described at the end of the previous common mistake, this can easily be used to compare information that did not originally appear on the screen but was a part of the overall outcome of the test.

However, the amount of information in files and databases is often huge and to display it all on the screen one page at a time is usually impractical if not impossible. Thus, compromise sets in. Because it becomes so difficult to do, little comparison of the tests true outcome is performed. Where a tester does labour long and hard to ensure that the important information is checked, the test becomes complex and unwieldy once again, and worse still, very sensitive to a wide range of changes that frequently occur with each new release of the software under test. Of course, this in turn adversely impacts the maintenance costs for the test.

In one case, I came across a situation where a PC based tool vendor had struggled long and hard to perform a comparison of a large file generated on a mainframe computer. The file was brought down to the PC one page at a time where the tool then performed a comparison with the original version. It turned out that the file comprised records that exceeded the maximum record length that the tool could handle. This, together with the length of time the whole process took caused the whole idea of automated comparison of this file to be abandoned.

In this case, and many others like it, it would have been relatively simple to invoke a comparison process on the mainframe computer to compare the whole file (or just a part of it) in one go. This would have been completed in a matter of seconds (compared with something exceeding an hour when downloaded to the PC).

5. Let testware organisation evolve naturally

Like a number of other common mistakes, this one isn’t made through a deliberate decision (by choice) rather it is made through not realising the need to plan and manage where all the data files, databases, scripts, expected results, etc., etc., everything that makes up the tests, is required to run them and results from their execution, in short: the testware.

There are three key issues to address: scale, re-use; and multiple versions. Scale is simply the number of things that comprise the testware. For any one test there can be several (10, 15 or even 20) things (files) that are unique (files and records containing test input, test data, scripts, expected results, actual results and differences, log files, audit trails and reports). Figure 4 depicts one such test case.

Re-use is an important consideration for efficient automation. The ability to share scripts and test data not only reduces the effort required to build new tests but also reduces the effort required for maintenance. But, re-use will only be possible if testers can easily (and quickly) find out what there is to re-use, quickly locate it and understand how to use it. I'm told a programmer will spend up to 2 minutes looking for a re-useable function before he or she will give up and write their own. I'm sure this applies to testers and that it can be a lot less than 2 minutes. Of course, while test automation is implemented by only one or two people this will not be much of a problem if a problem at all, at least while those people remain on the automation team. But once more people become involved, either on the same project or on other projects, the need for more formal organisation (indeed a standard / common organisation) becomes much greater.

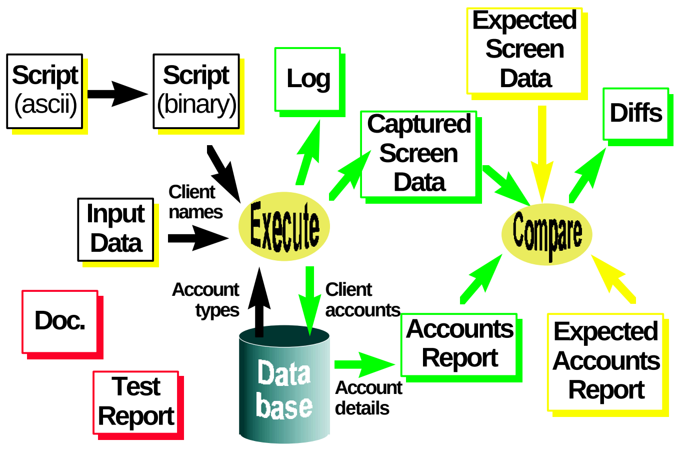

Figure 4 Executing a single test inevitably results in a large number of different files and types of information, all of which have to be stored somewhere. Configuration management is essential for efficient test automation.

Multiple versions can be a real problem in environments where previous versions of software have to be supported while a new version is being prepared. When an emergency bug fix is undertaken, we would like to run as many of our automated tests as seems appropriate to ensure that the bug fix has not had any adverse affects on the rest of the software. But if we have had to change our tests to make them compatible with the new version of the software this will not be possible unless we have saved the old versions of the tests. Of course the problem becomes even worse if we have to manage more than one old version or more than one software system.

If we have only a relatively few automated tests it will be practical to simply copy the whole set of automated tests for each new version of the software. Of course bug fixes to the tests themselves may then have to be repeated across two or more sets but this should be a relatively rare occurrence. However, if we have a large number of tests this approach soon becomes impractical. In this case, we have to look to configuration management for an effective answer.

6. Trying to automate too much

There are two aspects to this: automating too much too soon; and automating too much, full stop. Automating too much early on leaves you with a lot of poorly automated tests which are difficult (and therefore, costly) to maintain and susceptible to software changes.

It is much better to start small. Identify a few good, but diverse, tests (say 10 or 20 tests, or 2 to 3 hours worth of interactive testing) and automate them on an old (stable) version of software, perhaps a number of times, exploring different techniques and approaches. The aim here should be to find out just what the tool can do and how different tests can best be automated taking into account the end quality of the automation (that is, how easy it is to implement, analyse, and maintain). Next, run the tests on a later version (but still stable) of the software to explore the test maintenance issues. This may cause you to look for different ways of implementing automated tests that avoid or at least reduce some of the maintenance costs. Then run the tests on an unstable version of the software so you can learn what is involved in analysing failures and explore further implementation enhancements to make this task easier and therefore, reduce the analyse effort.

The other aspect, that of automating too much long term may at first seem unlikely. Intuitively, the more tests that are automated the better. But this may not be the case. Continually adding more and more automated tests can result in unnecessary duplication and redundancy and a cumulative maintenance cost. James Bach has an excellent way of describing this [BACH97]. James points out that eventually the test suite will take on a life of its own, testers will depart, new testers will arrive and the test suite grows ever larger. Nobody will know exactly what all the tests do and nobody will be willing to remove any of them, just in case they are important.

In this situation many inappropriate tests will be automated as automation becomes an end it itself. People will automate tests because “that’s what we do here - automate tests” regardless of the relative benefits of doing so.

James Bach [BACH97] reports a case history in which it was discovered that 80% of the bugs found by testing were found by manual tests and not the automated tests despite the fact that the automated tests had been developed of a number of years and formed a large part of the testing that took place. A sobering thought indeed.

Acknowledgements

My thanks to Brian Marick for unwittingly giving me the idea for this paper following his presentation of his own “Classic Testing Mistakes” paper at the Star’97 Conference [MARI97].

References

BACH97	James Bach, “Test Automation Snake Oil” presented at the 14th International Conference on Testing Computer Software, Washington, USA.
BEIZ90	Boris Beizer, “Software Testing Techniques”, 2nd Edition published by Van Nostrand Reinhold.
MARI97	Brian Marick, “Classic Testing Mistakes” presented at the 6th International Conference on Software Testing Analysis and Review, May 1997 San Jose, California, USA.

Mark Fewster

Mark has nearly 20 years of industrial experience in software testing specialising in the area of Software Testing Tools and Test Automation. This includes the development of a test execution tool and its successful introduction into general use within the organisation.

Since joining Grove Consultants in 1993, Mark has provided consultancy and training in software testing, particularly in the application of testing techniques and test automation. He has published papers in respected journals and is a popular speaker at national and international conferences and seminars.

Mark serves on the committee of British Computer Society's Specialist Interest Group in Software Testing (BCS SIGIST) and is also a member of the Information Systems Examination Board (ISEB) working on a qualification scheme for testing professionals.

Mark has co-authored a book with Dorothy Graham, "Software Test Automation" published by Addison-Wesley.