

IMPLEMENTING COST EFFECTIVE AUTOMATED TESTING

Joseph Anjo Porathur
IBS Software Services Pvt Ltd,
Trivandrum, Kerala, India
josepha@ibsplc.com

Abstract:

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 to just test manually". The end result, all too often, is that the tool ends up on the shelf as just another "purchasing mistake".

According to popular mythology, people with little or no programming experience can use GUI-level regression test tools to quickly and competently create extensive black box test suites that are easy to maintain. Though some efforts to use these tools have been successful, several have failed miserably.

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. It also covers the issues, problems, necessities and requirements involved in this regard.

1 Introduction

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*.

2 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*.

3 Problems with the basic paradigm:

Here is the basic paradigm for GUI-based automated regression testing:

- Design a test case, and then run it.
- If the program fails the test, write a bug report. Start over after the bug is fixed.
- If the program passes the test, automate it. Run the test again (either from a script or with the aid of a capture utility). Capture the screen output at the end of the test. Save the test case and the output.
- Next time, run the test case and compare its output to the saved output. If the outputs match, the program passes the test.

First problem: this is not cheap. It usually takes between 3 and 10 times as long (and can take much longer) to create, verify, and minimally document the automated test as it takes to create and run the test once by hand. Many tests will be worth automating, but for all the tests that you run only once or twice, this approach is inefficient.

Some people recommend that testers automate 100% of their test cases. I strongly disagree with this. I create and run many black box tests only once. To automate these one-shot tests, I would have to spend substantially more time and money per test. In the same period of time, I wouldn't be able to run as many tests. Why should I seek lower coverage at a higher cost per test?

Second problem: this approach creates risks of additional costs. We all know that the cost of finding and fixing bugs increases over time. As a product gets closer to its (scheduled) ship date more people work with it, as in-house beta users or to create manuals and marketing materials. The later you find and fix significant bugs, the more of these people's time will be wasted. If you spend most of your early testing time writing test scripts, you will delay finding bugs until later, when they are more expensive.

Third problem: these tests are not powerful. The only tests you automate are tests that the program has already passed. How many new bugs will you find this way? The estimates that I've heard range from 6% to 30%. The numbers go up if you count the bugs that you find while creating the test cases, but this is usually manual testing, not related to the ultimate automated tests.

Fourth problem: in practice, many test groups automate only the easy-to-run tests. Early in testing, these are easy to design and the program might not be capable of running more complex test cases. Later, though, these tests are weak, especially in comparison to the increasingly harsh testing done by a skilled manual tester.

Now consider maintainability:

Maintenance requirements don't go away just because your friendly automated tool vendor forgot to mention them. Two routinely recurring issues are

- When the program's user interface changes, how much work do you have to do to update the test scripts so that they accurately reflect and test the program?
- When the user interface language changes (such as English to French), how hard is it to revise the scripts so that they accurately reflect and test the program?

We need strategies that we can count on to deal with these issues.

Here are two strategies that don't work:

Creating test cases using a capture tool: The most common way to create test cases is to use the capture feature of your automated test tool. This is absurd.

In your first course on programming, you probably learned not to write programs like this:

```
SET A = 2
```

```
SET B = 3
```

```
PRINT A+B
```

Embedding constants in code is obviously foolish. But that's what we do with capture utilities. We create a test script by capturing an exact sequence of exact keystrokes, mouse movements, or commands. These are constants, just like 2 and 3. The slightest change to the program's user interface and the script is invalid. The maintenance costs associated with captured test cases are unacceptable.

Capture utilities can help you script tests by showing you how the test tool interprets a manual test case. They are not useless. But they are dangerous if you try to do too much with them.

Programming test cases on an ad hoc basis: Test groups often try to create automated test cases in their spare time. The overall plan seems to be, "Create as many tests as possible." There is no unifying plan or theme. Each test case is designed and coded independently, and the scripts often repeat exact sequences of commands. This approach is just as fragile as the capture/replay.

4 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. 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.softwaretestinginstitute.com) 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 interact 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.

4.1 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 have found to be effective for automating functional or system testing for most business applications

4.1.1 Framework – Based Architecture

The framework provides an entirely different approach, although it is often used in conjunction with one or more data-driven testing strategies.

The framework isolates the application under test from the test scripts by providing a set of functions in a shared function library. The test script writers treat these functions as if they were basic commands of the test tool's programming language. They can thus program the scripts independently of the user interface of the software.

For example, a framework writer might create the function, `openfile(p)`. This function opens file `p`. It might operate by pulling down the file menu, selecting the Open command, copying the file name to the file name field, and selecting the OK button to close the dialog and do the operation. Or the function might be richer than this, adding extensive error handling. The function might check whether file `p` was actually opened or it might log the attempt to open the file, and log the result. The function might pull up the File Open dialog by using a command shortcut instead of navigating through the menu. If the program that you're testing comes with an application programmer interface (API) or a macro language, perhaps the function can call a single command and send it the file name and path as parameters. The function's definition might change from week to week. The scriptwriter doesn't care, as long as `openfile(x)` opens file `x`.

Many functions in your library will be useful in several applications (or they will be if you design them to be portable). Don't expect 100% portability. For example, one version of `openfile()` might work for every application that uses the standard File Open dialog but you may need additional versions for programs that customize the dialog.

Frameworks include several types of functions, from very simple wrappers around simple application or tool functions to very complex scripts that handle an integrated task. Here are some of the basic types:

4.1.1.1 Define every feature of the Application

You can write functions to select a menu choice, pull up a dialog, set a value for a variable, or issue a command. If the UI changes how one of these works, you change how the function works. Any script that was written using this function changes automatically when you recompile or relink.

Frameworks are essential when dealing with custom controls, such as owner-draw controls. An owner-draw control uses programmer-supplied graphics commands to draw a dialog. The test-automation tool will know that there is a window here, but it won't know what's inside. How do you use the tool to press a button in a dialog when it doesn't know that the button is there? How do you use the tool to select an item from a listbox, when it doesn't know the listbox is there? Maybe you can use some trick to select the third item in a list, but how do you select an item that might appear in any position in a variable-length list? Next problem: how do you deal consistently with these invisible buttons and listboxes and other UI elements when you change video resolution?

These kludges are a complex, high-maintenance, aggravating set of distractions for the script writer. I call them distractions because they are problems with the tool, not with the underlying program that you are testing. They focus the tester on the weaknesses of the tool, rather than on finding and reporting the weaknesses of the underlying program.

If you must contend with owner-draw controls, encapsulating every feature of the application is probably your most urgent large task in building a framework. This hides each kludge inside a function. To use a feature, the programmer calls the feature, without thinking about the kludge. If the UI changes, the kludge can be redone without affecting a single script.

4.1.1.2 Define commands or features of the tool's programming language.

The automation tool comes with a scripting language. You might find it surprisingly handy to add a layer of indirection by putting a wrapper around each command. A wrapper is a routine that is created around another function. It is very simple, probably doing nothing more than calling the wrapped function. You can modify a wrapper to add or replace functionality, to avoid a bug in the test tool, or to take advantage of an update to the scripting language.

Consider the example of `wMenuSelect`, a Visual Test function that selects a menu. You can write a wrapper function, `SelMenu()` that simply calls `wMenuSelect`. This provides flexibility. For example, you can modify `SelMenu()` by adding a logging function or an error handler or a call to a memory monitor or whatever you want. When you do this, every script gains this new capability without the need for additional coding. This can be very useful for stress testing, test execution analysis, bug analysis and reporting and debugging purposes.

4.1.1.3 Define small, conceptually unified tasks that are done frequently.

The `openfile()` function is an example of this type of function. The scriptwriter will write hundreds of scripts that require the opening of a file, but will only consciously care about how the file is being opened in a few of those scripts. For the rest, he just wants the file opened in a fast, reliable way so that he can get on with the real goal of his test. Adding a library function to do this will save the scriptwriter time, and improve the maintainability of the scripts.

This is straightforward code re-use, which is just as desirable in test automation as in any other software development.

4.1.1.4 Define larger, complex chunks of test cases that are used in several test cases.

It may be desirable to encapsulate larger sequences of commands. However, there are risks in this, especially if you overdo it. A very complex sequence probably won't be re-used in many test scripts, so it might not be worth the labor required to generalize it, document it, and insert the error-checking code into it that you would expect of a competently written library function. Also, the more complex the sequence, the more likely it is to need maintenance when the UI changes. A group of rarely-used complex commands might dominate your library's maintenance costs.

4.1.1.5 Define utility functions.

For example, you might create a function that logs test results to disk in a standardized way. You might create a coding standard that says that every test case ends with a call to this function.

Each of the tools provides its own set of pre-built utility functions. You might or might not need many additional functions.

Some framework risks

You can't build all of these commands into your library at the same time. You don't have a big enough staff. Several automation projects have failed miserably because the testing staff tried to create the ultimate, gotta-have-everything programming library. Management support (and some people's jobs) ran out before the framework was completed and useful. You have to prioritize. You have to build your library over time.

Don't assume that everyone will use the function library just because it's there. Some people code in different styles from each other. If you don't have programming standards that cover variable naming, order of parameters in function interfaces, use of global variables, etc., then what seems reasonable to one person will seem unacceptable to another. Also, some people hate to use code they didn't write. Others come onto a project late and don't know what's in the library. Working in a big rush, they start programming without spending any time with the library. You have to manage the use of the library.

Finally, be careful about setting expectations, especially if your programmers write their own custom controls. In Release 1.0 (or in the first release that you start automating tests), you will probably spend most of your available time creating a framework that encapsulates all the crazy workarounds that you have to write just to press buttons, select list items, select tabs, and so on. The payoff from this work will show up in scripts that you finally have time to write in Release 2.0. Framework creation is expensive. Set realistic expectations or update your resume.

5 Strategies for Success

Here are some suggestions for developing an automated regression test strategy that works:

1. Reset management expectations about the timing of benefits from automation
2. Recognize that test automation development is software development.
3. Maintain a Strong Test Environment
4. Manage Resistance to Change
5. Recognize Staffing Realities

5.1 Reset management expectations about the timing of benefits from automation.

We all agreed that when GUI-level regression automation is developed in Release N of the software, most of the benefits are realized during the testing and development of Release N+1. Some benefits are realized in release N. For example:

- There's a big payoff in automating a suite of acceptance-into-testing (also called "smoke") tests. You might run these 50 or 100 times during development of Release N. Even if it takes 10x as long to develop each test as to execute each test by hand, and another 10x cost for maintenance, this still creates a time saving equivalent to 30-80 manual executions of each test case.
- You can save time, reduce human error, and obtain good tracking of what was done by automating configuration and compatibility testing. In these cases, you are running the same tests against many devices or under many environments. If you test the program's compatibility with 30 printers, you might recover the cost of automating this test in less than a week.
- Regression automation facilitates performance benchmarking across operating systems and across different development versions of the same program.

Take advantage of opportunities for near-term payback from automation, but be cautious when automating with the goal of short-term gains. Cost-justify each additional test case, or group of test cases.

If you are looking for longer term gains, across releases of the software, then you should seriously thinking about setting your goals for Version N as:

- providing efficient regression testing for Version N in a few specific areas (such as smoke tests and compatibility tests);
- developing scaffolding that will make for broader and more efficient automated testing in Version N+1.

5.2 Recognize that test automation development is software development.

You can't develop test suites that will survive and be useful in the next release without clear and realistic planning.

You can't develop extensive test suites (which might have more lines of code than the application being tested) without clear and realistic planning.

You can't develop many test suites that will have a low enough maintenance cost to justify their existence over the life of the project without clear and realistic planning.

Automation of software testing is just like all of the other automation efforts that software developers engage in—except that this time, the testers are writing the automation code.

- It is code, even if the programming language is funky.
- Within an application dedicated to testing a program, every test case is a feature.
- From the viewpoint of the automated test application, every aspect of the underlying application (the one you're testing) is data.

As we've learned on so many other software development projects, software developers (in this case, the testers) must:

- understand the requirements;
- adopt an architecture that allows us to efficiently develop, integrate, and maintain our features and data;
- adopt and live with standards. (I don't mean grand schemes like ISO 9000 or CMM. I mean that it makes sense for two programmers working on the same project to use the same naming conventions, the same structure for documenting their modules, the same approach to error handling, etc.. Within any group of programmers, agreements to follow the same rules are agreements on standards);
- be disciplined.

Of all people, testers must realize just how important it is to follow a disciplined approach to software development instead of using quick-and-dirty design and implementation. Without it, we should be prepared to fail as miserably as so many of the applications we have tested.

5.3 Maintain a strong Test Environment

It assumes that adequate preparations have been made by the organization before beginning the testing automation process. This is rarely the case, however. If adequate preparations have not been made, then the "ramp-up" time required is increased dramatically. What then, does an organization do to prepare themselves for this effort?

- An adequate Test Environment must exist that accurately replicates the Production Environment. This can be a small-scale replica, but it must consist of the same *types* of hardware, programs and data.
- The Test Environment's database must be able to be *restored* to a known baseline, otherwise tests performed against this database will not be able to be repeated, as the data will have been altered.
- Part of the Test Environment includes *hardware*. The automated scripts must have dedicated PC's on which to run. If one is developing scripts, then these scripts themselves *must be tested* to ensure that they work properly. It takes time to run these scripts, especially after a number of them have been developed. If the script developers must run these scripts on their development machines, then what will they do while the scripts are running? Watch them? Now that's an effective use of one's time! If the company is purchasing say, 5 licenses for the test tool, then it would be prudent to assign at least 3 PCs to run the automated scripts on. The other 2 can be used for script development (assuming 2 developers). A company can easily spend up to \$150K on test tools (including load testing). It doesn't make sense to then start getting "frugal" with the hardware. Since test tool software requires a lot of processing speed and memory to run correctly, you're going to want to get the biggest, fastest machines you can afford to run this software.
- Detailed Test Cases that are able to be converted to an automated format must exist. If they do not, then they will need to be developed, adding to the time required. The test-tool is not a thinking entity. You must tell it exactly what to do, how to do it, and when to do it. Data to be entered and verified must be *specific* data, not "post a payment to an account, and verify the results".
- The person or persons who are going to be developing and maintaining the automated scripts must be hired and trained. Normally, test tool vendors provide training courses. While it makes sense to hire a consulting firm or a contractor to help you get started, who are they going to turn things over to? If no one is there to take over, then what is going to happen to the automated testing effort? If the consulting firm or contractor is supposed to train someone, then that someone better be there *before* the consultants start their work. If you need to, have the consultants help you hire a person that can do the job - and have them do that *first*.

5.4 Manage 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 "Framework based Architecture" method (described above) is used, the testers will not have to learn how to use the tool in detail. All that they have to learn is a different method of executing the test scripts, using the data driven approach. It is not that difficult when considering the fact that the complex part of the framework will have already been developed by experts in the field. It is just a matter of learning how to create scripts using this framework and execute them.

- 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.

5.5 Recognize Staffing Requirements

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 similar position must be created and staffed with at least one senior-level programmer. It does not really matter what language 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 want 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 written by the testers
- Running the automated tests, and providing the testers with the results

It is often useful to hire a contractor 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 be 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.

6 Summary

- 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
 - Framework Based Architecture may be used provided there is an Automated Testing Engineer competent enough to develop the framework
 - The scripts that are created should be data driven. Hard coded values should be fully eliminated
- 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.
- Realize the fact that script development is equivalent to software development. All the processes used in software development should also be incorporated in script development.
- A version control tool may also be used to track changes made during the course of time.