Test Automation: From Record/Playback to Frameworks

John Kent M.Sc.
Email: john.kent@simplytesting.com
Web: http://www.simplytesting.com

Abstract

Script-based automated software testing tools which act upon User Interfaces (UI) have been available since the early 1990's. Organisations have become increasingly aware of the short-comings of these tools as they are out-of-the-box. Maintenance is a key issue in automation. The format and behaviour of a system's user interface changes with each new release and so the automated tests must also change. The basic approaches to building test automation code inevitably lead to an excessive maintenance burden as the size of the test suite increases. Another problem is outlined by the test automation paradox: the automated tests must be tested. Some automation structures lend themselves to producing highly tested automation code and some do not. Software engineering principles have been applied to the building of automation code structures or *frameworks* in order to overcome these and other problems. Test automation *frameworks* are gaining acceptance and they can offer *keyword data-driven* automation with actions and navigation specified in the test data. They can minimise the maintenance overhead, provide highly tested code and even offer script-less automation. This paper discusses the different automation code structures and their relative merits. It describes test frameworks and reviews progress in the field.

Introduction

This paper concerns itself with script-based test automation tools which act upon User Interfaces (UI). This includes tools such as WinRunner, QuickTest Pro, QARun, TestPartner, Robot and Functional Tester, amongst others. The comments here do not necessarily apply to test frameworks such as NUnit, JUnit, etc which are for testing objects at the unit test level.

Software test automation seems to be on every Test Manager's list of things to have. Almost every test organisation wants it. The test tool companies have made themselves large by selling tools to do it. It is often, mistakenly, the first thing people start with when they embark upon improving their testing procedures. All of this illustrates how attractive test automation is—not how easy it is. This attraction to test automation is strange when you consider how little success it has had. How many automation projects return real benefits for their investments? In automation circles there has been a debate about why the success rate is so low. Is it because the wrong tools are being used? Is it because of a lack of management commitment? Is it a failure to manage expectations? All of these things are important, but the major factor in the success, or otherwise, of automation projects is the way in which the tools are used. The structure of the code developed using these tools can either overcome or exacerbate the underlying problems of test automation.

Scripted tools running under Windows have only been around for a short while. Microsoft Test (later Visual Test) version 1.0 came off beta test in early 1992[9], at about the same time as many of the other tools we now have. The technology is difficult to use, but new approaches to its use are evolving and it is an exciting time to be involved in this area. One element in these new ways to build test automation is a software engineering approach to building test automation which has led to test frameworks. This paper builds on one of the authors' previous papers[16, 18] in the light of growing acceptance of test automation frameworks.

How Scripted Automation Tools Work

These test tools work by acting on the System Under Test's (SUT) user interface. A user interface consists of many discreet objects.

User Interface Objects

Windows or screens which form a user interface (UI) for a computer software system are collections of user interface objects. In Microsoft Windows these objects are called *Windows Controls*. In Unix XWindows they are called *Widgets*. Green screens have UI Objects called *Fields*. Many different types or *classes* of window control are available. Standard classes of Windows controls include:

- Text-boxes
- Combo-boxes
- Buttons
- Option buttons
- Check boxes
- Menus
- List-boxes
- Grids

There are many more types of controls, which may, or may not be implemented in non-standard ways. The user interface also includes the keyboard and mouse which are used to perform actions on UI Objects.

Recording, Scripting and Playing-Back

The automated test tools we are interested in here, perform actions on UI objects. These actions are performed by running a *script*. Scripts may be recordings of user actions. They may have been programmed (or scripted) manually in the language of the test tool. The ability to record scripts is available with most of the test tools on the market. The test tool is set to *record* mode and the user performs actions on the SUT. The recording or script will consist of a list of actions. This can then be replayed back into the SUT's UI, thereby executing tests against the system.

User Interface Object Functions

The recorded or scripted actions are written using functions which form part of the test tools scripting language. These functions act on the UI objects. Below is a fragment of IBM Rational Robot recording where an 'OK' button is clicked and a File|Exit menu item is selected.

```
PushButton Click, "Text=OK"
MenuSelect "File->Exit"
```

In WinRunner TSL the same actions look like this:

```
button_press ("OK");
menu_select_item ("File;Exit");
```

In QARun the actions look like this:

```
Button "OK", 'Left SingleClick'
MenuSelect "File~Exit"
```

The above demonstrates how similar the test tools are in that they all provide functions which perform actions on UI objects. The test tool attempts to identify each UI object uniquely, using a variety of different object properties.

UI Object Maps

In order to improve script maintainability, some tool vendors have built UI Object Map functionality into their test tools. A UI Object Map lists the objects in each window/page. Each UI object is given a

logical name and also contains the information that the test tool needs to identify that object uniquely. The test tool script uses the logical name in scripts when performing actions on UI Objects. If the UI object properties change so that the identification changes, only the Object Map is changed; not references to it within the scripts. The UI object map is called the *GUI Map* in WinRunner and the *Object Repository* in QuickTest Pro. IBM Rational Robot does not have UI object map but Compuware's QARun does. The figure below shows a logon window and the WinRunner GUI Map which defines the UI Objects for that window.

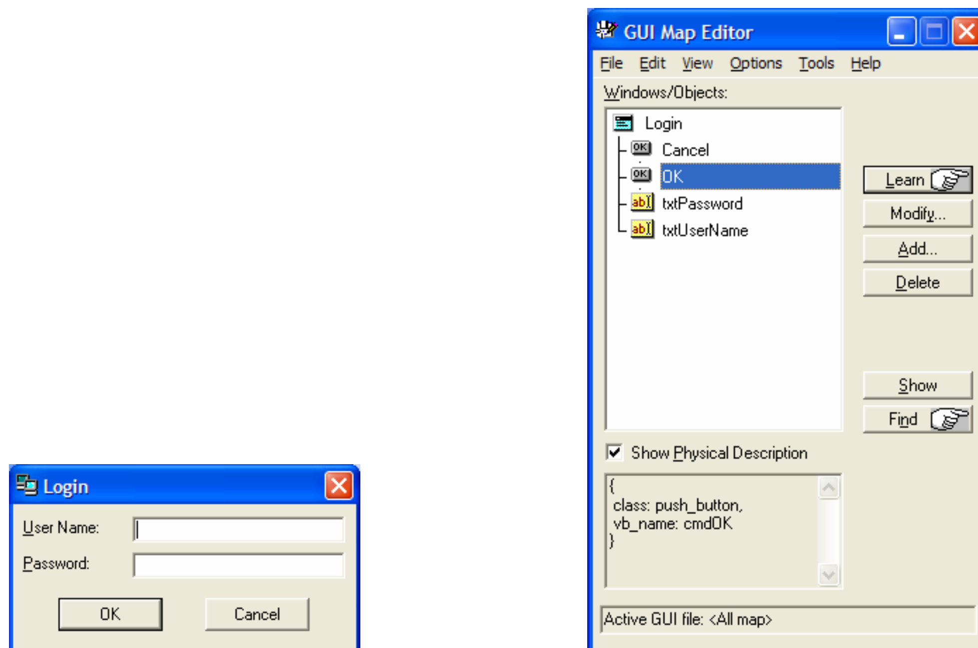


Figure 1: A window and its equivalent WinRunner GUI Map

Figure 1 (above) shows a Login window and its definition in the WinRunner GUI Map. You can see that the OK and Cancel buttons are in the map along with two textboxes called txtPassword and txtUserName, which correspond to the objects in the window. The OK button is selected and the 'Physical Description' is given. The OK button is identified by its *class* ('push_button') and its vbname ('cmdOK').

Record/Playback

As we have already seen the tools can create automated tests by recording the actions a user performs against a user interface and these actions can then be played-back into the UI, hence the term: *Record/Playback*. The recorded script will be a series of actions in the tool's language. As an example, the following is a recording of actions against the Login window shown in Figure 1.

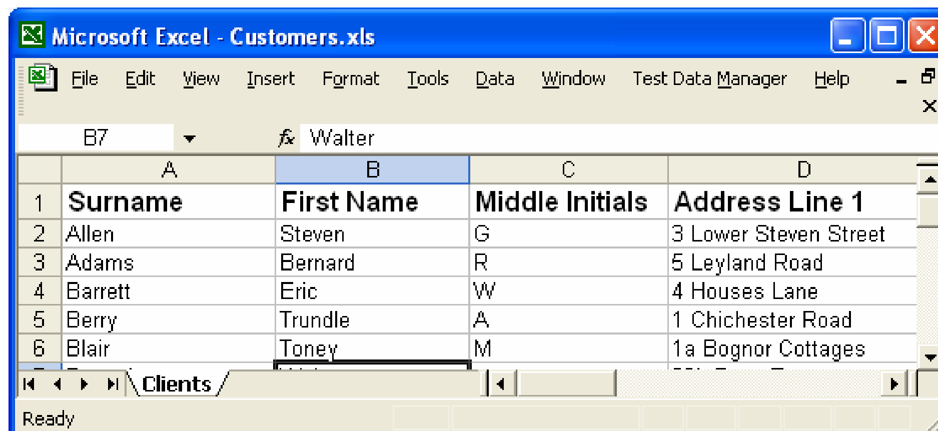
```
# Login
	set_window ("Login", 2);
	edit_set ("txtUserName", "JohnSmith");
	password_edit_set("txtPassword", "hygsfzgsbzotmyisivjs");
	button_press ("OK");
```

The code above again shows the functions used to provide actions against the UI Objects. The edit_set function is used to set the text in the txtUserName textbox to 'JohnSmith'. Once the script is recorded it can be modified manually by scripting (programming) and all the standard programming constructs such as conditionals, loops, variables, etc can be used.

Basic Data-Driven

A more mature architectural approach called 'data-driven' automation was developed very early on in the history of this type of test tool. In data-driven automation the test data is held in separate files

which are read in by the automated test code and then used as input into the SUT. See the example test data file below in Fig 2. It has a heading line and you can see that column A contains the surname, column B the first name, etc, etc.



	A	B	C	D
1	Surname	First Name	Middle Initials	Address Line 1
2	Allen	Steven	G	3 Lower Steven Street
3	Adams	Bernard	R	5 Leyland Road
4	Barrett	Eric	W	4 Houses Lane
5	Berry	Trundle	A	1 Chichester Road
6	Blair	Toney	M	1a Bognor Cottages

Figure 2: Simple 'data-driven' data

In the test tool there will be a user programmed function which repeatedly reads in this data and then inputs it into the 'add customer' windows or screens. The program (or a driver program) might then call another program, which would open another file and do the same thing for another business entity.

For example, automation of a system which manages insurance policies may look like this:

(i) The driver program calls the test function *add_customer*.

(ii) The *add_customer* program navigates to the customer screens (windows or web pages) and repeatedly reads the customer data and inputs it into the System Under Test (SUT).

(iii) The driver program then calls the *add_policy* function.

(iv) The *add_policy* function repeatedly reads the policy data file and inputs it to the policy screens of the SUT.
and so on for each business task…..

Each action like *add_customer* will be performed by a function which must be written (scripted or programmed) by an automation specialist. User written functions are a way of introducing code re-use into the automation code. In record/playback each automated test is a long sequence of recorded actions with the test data hard-coded. If you want another automated test, you record another script which gives two scripts to maintain as the SUT UI changes over time. In the data-driven situation, the test data can contain a vast variety of different data combinations and so you can increase your test coverage in a way that simply would not be possible with manual testing. You can add more tests without having to write anymore test script (albeit only for the actions you already have). In the above example you only have two test programs to maintain: add_customer and add_policy, but you can now exercise both of these business scenarios automatically, repeatedly and also increase the test coverage of the code in comparison to Record/playback.

The test programs can also be made to check values in the UI, automating the examination of expected results.

An improvement to this basic approach is to control the order test data is input into the SUT. A control file is introduced which specifies the data sequence. This instructs the automation for example, to add a customer then add a policy for that customer and then add another customer, and so on. In this case the sequence of actions looks more like that of a manual test.

Key Features of Software Test Automation

Automation Is Software Engineering

The tool vendors promote these tools as record/playback and they use the tester-friendly term *scripting* to describe the programming process. So what is a script? 'Script' is another word for Program—they are the same thing. These test tools have *scripting* languages that are usually 'VB like' or 'C like' and

contain all the standard programming structures such as statements, variables, conditionals and operators. Script-based test automation tools are, in fact, program development environments with compilers or interpreters, code editing and debugging facilities like any other IDE. What makes them test automation tools is that they have special functions in the test language which can 'talk' to your system's UI Objects. QuickTest Pro for example, uses VBScript as its scripting language which is hidden in the standard view but revealed if the 'Expert View' tab is selected. What we build when we create test automation is software. Test automation is, therefore, Software Engineering. This fact is fundamental to the activity of building and using software test automation and should shape our thoughts about the activities we carry out. Software products in software engineering have to be developed, maintained and as testers we should be well aware that they must be *tested* as well.

To emphasise and summarise: the test automation we build, no-matter what structure it is built in, must be:

- Developed
- Maintained
- Tested

Because software test automation using script-based tools is software engineering, the tools are really program/playback.

The Software Test Automation Paradox

The last item in the list of software engineering tasks above, leads us to something of a paradox. This is that we must *'test the automated tests'*. When we run an automated test against a new release of the software under test, we are testing two software systems. Firstly we are testing the SUT. Secondly and there is no way around this, we are testing the test automation. The test automation is a software system that has a requirement to be able to test the latest release of the SUT but this automation itself may be faulty and failures may occur. This paradox implies that we must build automation code structures in such a way that they can be tested to some degree, so that we can have a high degree of confidence that the automation software is reliable.

Comparison of the record/playback and data-driven approaches nicely illustrate how some automation structures become more tested with use. In Record/Playback, each test is run once per release and that means each script is run once and so it is not exercised very often compared to the data-driven situation. Data driven functions are run iteratively as each line of test data is processed. This increases our confidence in the reliability of the test code. If there are possible synchronisation problems, for example, they are more likely to become apparent in the data driven case. Other code structures in more advanced frameworks give more highly tested code and therefore, even more confidence in the test automation's reliability.

It is ironic that these tools are used by test teams who often do not recognise that the software produced using these tools needs to be tested. As testers we know that we must strive for the highest test coverage we can manage for any software product we have—including test automation software.

The paradox that we must 'test the test automation' should not be underestimated. It is one of the two fundamental problems that must be resolved for successful, large-scale test automation.

Maintenance of Automation Code Is Critical

The other fundamental issue in automation is that of test code maintenance. With test automation we are interested in testing the system as it is changed. Generally, the user interface objects and behaviour change with each release of the system and so you must change your automation to work with these changes. In test Record/playback you are interested in subsequent releases of the SUT because you have already proved that the system does what you've recorded. The payback with automation starts when you test later releases of the system. In short: payback only comes with playback. If we cannot maintain our automated tests and bring them up to date quickly enough, we cannot run the automation. Maintenance is crucial to test automation and this fact is one of the main motivating factors in building test automation frameworks for user interfaces.

So how do we measure maintainability of automated tests? One measure we can use is the *UIObject-to-Reference Ratio*. This is a ratio of the number of UI Objects automated versus the number of times they are referenced in the automation code. Imagine a number of tests created using Record/Playback which all include automation of the logon window in Fig 1 (above). Each and every script would have a reference to the username textbox. If the username textbox changed in some way, changes would need to be made to every script (GUI maps may go some way to improving this situation but not for every change). The ideal for the UIObject-to-Reference Ratio is 1:1 or one reference in the test code for each UI Object in the SUT.

Limitations of Basic Approaches

Limitations of Record/Playback

It is now generally accepted in the automated testing community that it is not practical to create automated tests by simply recording them with script-based tools[2, 3, 4, 5, and 6]. Defining what exactly is wrong with Record/Playback has not been straight-forward. The problems are seen as:

(i) Recording produces scripts that are difficult to maintain because they consist of long lists of actions on objects in your user interface which are then difficult to decipher.

(ii) The recording may not re-run due to synchronisation problems. Synchronisation is an important issue in automated UI testing. The script may need to wait for a window to appear. If it doesn't appear in time the script may just carry on attempting to perform actions on the wrong windows. Synchronisation points need to be programmed into the script. Try recording just one hour of input with one of the modern GUI script-based tools and then see how well it replays. It is unlikely to run all the way through. Another enlightening exercise is to try to fix the recorded script in order to see what you would have to do to get it to replay correctly and how long it would take you.

(iii) With Record/Playback you end up with data hard-coded into your test program. This is not a good thing according to software engineering principles[6].

(iv) Automated tests must cope with what *may* happen, not simply with what happened when the script was recorded. One of the objectives of running tests against a system is to find errors that were not there before. The automated tests must have code that can recognise unexpected errors and this must be *programmed* into them. With basic Record/playback, if an error is encountered during a test run it is likely that the whole test sequence will come to a stop. It is indeed ironic that, although the objective of automated testing is to find bugs, if the System Under Test (SUT) behaves in an unexpected way—i.e. there are bugs—then these automated tests will not work. A test cannot therefore, simply be a recording played back into the SUT; rather it should be an intelligent *interaction* between the test tool and the SUT.

(v) Record/playback scripts are not as highly tested as with other code structures that take advantage of code re-use.

The reason it is difficult to pin down the fundamental problem with Record/playback is because it can actually work on a small scale. The problem of record playback is really a problem of scale. As stated above, when a test is automated using recording, script lines are generated. Thus the number of lines of script you have is proportional to the number of tests you automate and we get the relationship:

$$\boxed{LinesOfAutomationCode \propto NumberOfTests}$$

This says that the number of lines of automation code is *proportional to* the number of tests.

The more tests you record, the more automation code you have to maintain until you reach a point where the maintenance burden will inevitably become too great for your budget. See Fig 3 which shows this relationship graphically. For low numbers of tests, maintenance of the scripts, whilst difficult, is possible. However, if an organisation is to invest large sums of money in test automation, it is highly likely to want to automate a large number of tests in order to produce a return on its investment. An automated regression test pack is likely to contain enough tests in it to take us well up the graph in Figure 3.

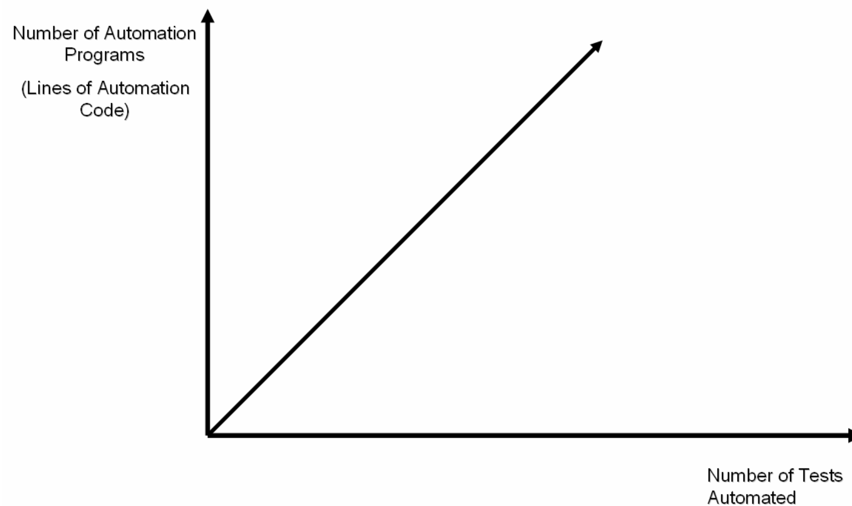


Figure 3: A graph of LOC against the number of tests automated

The graph also shows that the code has zero re-use, because each test has its own dedicated script and it is difficult to get highly tested code with such a structure. It is very difficult to see how we could test recorded scripts enough to give high reliability.

So it is really the cost of maintenance that condemns Record/Playback to little more than a sales gimmick. Yes, it is possible to maintain recorded automation code but it is the cost of doing this for large recorded test packs that makes it impractical. Record/Playback may work for very small scale automation but for large, real-life automated regression testing it is a non-starter. This also applies to manually programmed scripts where the automation script to test ratio is 1:1. The graph above shows that Record/playback offers the worst possible maintenance situation of one script to one test.

Limitations of Data-Driven

Look again at the data in the table in Fig 2. It looks nothing like a series of tests and—it isn't. It is simply test data. Most regression or system tests are not about repetitively inputting data into the user interface, rather they seek to fully exercise all of the business functionality of the system under test in a realistic way. Data-driven automation does not provide a good platform for this.

Automated testers have recognised this for a while and some provide a solution for it by adding another layer to the automation. This uses a control file to specify the order in which data is used from the various data files. An example of this is given in the data-driven section above where an insurance system has a control file to select the customer data to add, then to select the insurance policy data to add for that customer and so on. This is a step in the right direction as it gives Test Analysts the facility to specify actual tests rather than just a repetitive data load. This approach however, becomes a mess architecturally.

Advanced Frameworks

Since the introduction of script-based UI automation tools, a number of test automation frameworks[3, 4, 5, 6, 8, 10, 11, 12, 13, 16, 17, 18] have been proposed. An important aim of advanced automation frameworks has been to move the creation of automated tests away from the test tool language to higher levels of abstraction. These frameworks allow Test Analysts to specify tests without having to know anything about the automated test tool and how to write scripts using it. Advanced frameworks also reduce the *UI Object-to-Reference Ratio* towards 1:1 by reducing the referencing of UI Objects.

What is a Framework?

A test framework can be considered to be *software architecture* for test automation. Linda Hayes in the *Automated Testing Handbook*[19] states that:

> *"The test framework is like an application architecture"*

Mark Fewster and Dorothy Graham[15] talk in terms of a Testware Architecture. This paper's author has been using the term *Automation Architecture* for some time[5, 16, 18], however the term *test automation framework* appears to have gained general acceptance.

If a framework is a software architecture, what is a software architecture? Bass, Clements, and Kazman of Carnegie Mellon's Software Engineering Institute (SEI) state in their book *Software Architecture in Practice*[21] that:

> *"The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them."*

So a software architecture is the structure of the source code and the relationship between the components. In civil engineering, the architecture is the overall structure of the building. The architecture provides the space for the functional use of the building and ultimately, is the major factor in how successful it is at fulfilling its purpose.

Taking the definition literally, Record/Playback and data driven automation could be said to have software architectures and are therefore, test automation frameworks—just very poorly designed ones. The term *Framework* however, is usually applied to one of the more sophisticated automation code structures.

Poor software architecture is one of the biggest factors in test automation project failures. Design of software architecture is specific skill based on programming knowledge and experience. Automation projects can become problematic if the automation builders do not work towards a sound architecture. They can become too focused on the low level detail to be able to see the overall structure and may not have the software engineering experience to design architectures. They may be builders not architects.

A good test automation framework provides structures for logging, error reporting and allows recovery to be incorporated (dropping failed tests, navigating the system to a base point and continuing with the next test). It will have libraries of reusable functions much in the same way as any other software system. Most importantly of all, its structure will minimise the code maintenance overhead and provide a platform to create testable structures. It is the starting point for success with test automation.

Examples of test automation frameworks/architectures:

- ATAA from Simply Testing
- TestFrame from LogicaCMG
- EMOS_GPL open source
- SAFS open source

What is Required?

To summaries the above, the test automation frameworks we build must include:

(i) A sound software engineering approach
(ii) Highly maintainable automation code structures – a *UIObject-to-Reference Ratio* as close to 1:1 as possible.
(iii) Automation code structures which allow the code to be highly tested
(iv) An 'elegant' way to create tests - a way to automate tests easily

An automated test framework should also provide:

(v) A script structure that is not inherently error prone.
(vi) Ability to recover from unexpected conditions.
(vii) Basic infrastructure components such as logging
(viii) Ability to run tests from multiple PCs at once

The test frameworks detailed below, implement all of the above requirements to some degree. They provide a stable basis for easily maintainable, testable and error free script code.

Keyword Driven Automation

Advanced test automation frameworks are really the logical place to go to when you take a software engineering approach to building test automation. So how do they differ from the basic data driven approach?

In order to illustrate this, let's first look again at the data used in data driven automation (see Figure 2). In this example of Excel test data, every line (except the heading line) has customer details in it. The table in Fig 2 is really just data. All the decisions on what to do and when, are taken by the automation code; the data does not give instructions to the code on what to do.

What advanced automation frameworks do, is take (at least some) of the navigation and actions out of the test programs and put them into the test data. The test data becomes the script—the sequence of actions to be followed. It tells the automation code what to do and in what order. When the Test Analysts create the test data they can, to some degree, choose what to do and in what order, just as they would when creating a manual test. The level of choice is dependent upon how sophisticated the framework is.

Figure 4 illustrates data from a keyword-driven framework. Note that if there is a hash at the beginning of the left hand column which means that the line is a comment and not test data. Lines beginning Supplier_Add or Stock_Item_Add are the actual test data which will be read by the automation code. The format of the file is very different to that in the data-driven case in Figure 2 because the meaning of the data in each column is dependent upon what type of line it is and this is defined in the first column which contains the *keyword*. The commented lines give the format so that the Test Analyst knows what each column represents. For example in the seventh line which has a Stock_Item_Add keyword, column four is the 'Name' field because this is a Stock_Item_Add format. Thus the Test Analyst can choose the order of the actions when creating the Excel data.

#Supplier_Add	Sup Name	Description	Type	External Flag
Supplier_Add	SC-SUP00	SC-SUP00 Supplier Company		OFF
Supplier_Add	SC-SUP01	SC-SUP01 Supplier Company		ON
# Loc_Add	Loc Name	Description	Corp	Corporate Y/N
Loc_Add	SC-SXX-1	Test LO Stock Room only	ON	OFF
#	SI Name	Item Description	Name	External Flag
Stock_Item_Add	SC-STCK00	CABLE	COAX	Test SI.
Stock_Item_Add	99	PUBLICATION	BOOK	Wind in the ...
#	SI Name	Description	Type	Quantity
Stock_Loc_Add	SC-STCK00	SC-SXX-1	1-01	10
Stock_Loc_Add	99	SC-SXX-1	1-01	10

Figure 4: Advanced architecture data

When the automation test runs, a driver program reads through the data and calls the function specified in column 1, passing it the data for that particular line. For example, there will be a function Stock_Item_Add, which will have been written by the test programmer (scripter) in order to perform all of the actions required to add a stock item. These functions are known as *Wrappers*.

Wrappers

Wrapper is an OO term that means some software that is used to interface with an object. In programming you call the wrapper if you wish to use the object but don't want to, or are unable to call the object directly. The object is wrapped by the wrapper.

In test automation, wrappers are programs or functions written in the language of the test tool which perform discreet automation tasks as instructed by the test data and actions. They implement the keywords and provide the interface between the test data and the user interface. In our previous data example, four wrappers will have been written – Stock_Item_Add, Loc_Add, Supplier_Add and Stock_Loc_Add. The Stock_Item_Add wrapper for example, is a function which performs all of the

actions necessary to perform the business task of adding a stock item. These are business level wrappers – they 'wrap' business scenarios and are written by the test programmer (scripter). The *keyword* tells the automation which *wrapper* to call.

There are two distinct types of keyword/wrappers: *Business Scenario Level Wrappers* and *Window Level Wrappers*. Dwyer and Freeburn[12] talk of '*Business Object Scenarios'* and *'Window-centric Scenario Libraries'*. This is dependant upon how the SUT is carved up in an automation sense–at the window level or the business scenario level.

Business Scenario Level Keywords/Wrappers

The data in Figure 4 is at Business Scenario Level. With this type of wrapper, one automation function is written in the test tool for each business scenario. These functions '*wrap'* the business scenarios.

Usually a functional decomposition of the system is the first step in building this approach (see Zambelich[3]). In a functional decomposition, the basic business scenarios of the system are defined. Then the wrappers for each business task are manually programmed (scripted) and the data format for each task is created.

As you can see from the above example, Business Object Scenario Level test data is at the business language level and therefore understandable by end users, which is a great advantage.

Window Level Keywords/Wrappers

With this type of keyword/wrapper there is one automation function that deals with each window, screen or web page in the system—it acts as a wrapper for that screen/window. It handles all of the input and output (getting expected results) for its window or screen. See Figure 5 for an example of window level data. Again, lines with a hash in the left hand side are comments, used to show the format of the data. The other lines are the test data which are passed to the automation wrappers for that particular screen. Unlike the business level data, window level data conforms to a set format:

(i) The column headed 'Wrapper' specifies the screen or window to be acted upon.
(ii) The 'Object' column contains the name of the User Interface Object.
(iii) The Action column gives the action to be performed on that UI Object.
(iv) The 'Data' Column contains any data that is to be used.

In a sense the Keyword is made up of the first three of the above. Each class of UI Object has certain allowable actions against it.

	E	F	G	H
1	Wrapper	Object	Action	Data
2	Home | The National Lottery	MyAccount	CLICK	
3	Sign In | The National Lottery	Open Account	CLICK	
4	About You | Open Account | The National Lottery	title	SELECTROW	Mr.
5	About You | Open Account | The National Lottery	firstName	SET	John
6	About You | Open Account | The National Lottery	lastName	SET	Kent
7	About You | Open Account | The National Lottery	DOBDay	SELECTROW	05
8	About You | Open Account | The National Lottery	DOBMonth	SELECTROW	Jun
9	About You | Open Account | The National Lottery	DOBYear	SELECTROW	1958
10	About You | Open Account | The National Lottery	gender	SELECTROW	Male
11	About You | Open Account | The National Lottery	email	SET	john.kent@simplytesting.com
12	About You | Open Account | The National Lottery	emailRetype	SET	john.kent@simplytesting.com
13	About You | Open Account | The National Lottery	telephone	SET	0797 4865 48648
14	About You | Open Account | The National Lottery	quit	CLICK	
15	Quit? | Open Account | The National Lottery	quit	CLICK	
16	Security Alert	Yes	CLICK	
17				
18				

Figure 5: Screen\Window Level Advanced Architecture data

In line 3 of the example above the 'Open Account' object is clicked, then in line 4 the title combobox is set to the row which equals the text 'Mr', and so on….

The actions can also check values in the UI Objects so the Test Analyst could specify actions like 'Check' that the object contains the data equal to 'Smith'.

One of the biggest advantages of Window Level Keyword/Wrappers is that all of the user interface objects can be made available to the test analyst in Excel and thus the analyst has complete control over what the navigation and actions should be, rather than being dependent upon what the test automation programmer (scripter) has put into the wrapper, as with Business Object Level Wrappers.

Maintenance with the Keyword/Wrappers Approach

In keyword-based frameworks the wrappers are written for each business scenario or window in the system under test. Thus we get the relationships:

$$LOC \propto NumberOfBizScenarios$$

$$OR$$

$$LOC \propto NumberOfWindows$$

For business scenario level wrappers the number of lines of automation is *proportional to* the number of business scenarios. With window level wrappers the number of lines of automation is *proportional to* the number of windows in the system. In Figure 6 we can see that the number of LOC goes up proportionally with the number of tests, until all the windows or business scenarios have been automated after which the LOC then remains constant as more tests are automated. Figure 6 assumes that window or business scenario wrappers are built progressively as more tests are added (which might not quite be the case), however, the graph shows that a point is reached when effectively all of the automation code has been written.

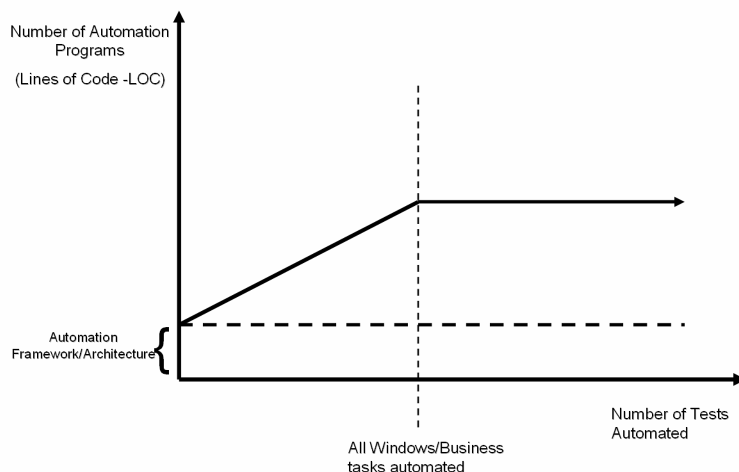


Figure 6: Automation LOC against the number of tests

UI Object Map Based Frameworks

The Limits of Program/Playback

If we implement a wrappers based framework by programming Business Scenario Wrappers or the Window Wrappers, we will still have to maintain the automation code because of changes to the UI of the SUT. Just as we hit limits with record/playback where it becomes impossible to maintain the automation code for a large system, we can hit limits with program/playback in wrapper based

frameworks. These limits are limits in terms of being able to build the automation pack, and limits in terms of being able to maintain it.

For small systems we don't hit these limits. Imagine, for example, a trading system which has only fifteen windows. Assume 20 UI objects per window and you get 20x15= 300 UI Objects. If you are using window wrappers that takes 8 lines of code for each UI Object, then you get 300 x 8 = 2400LOC. This is manageable and maintainable within many test team budgets (the full picture of maintainability of automation code needs to take into account the rate of change of the user interface but we don't need to go into that in detail here). Most of the systems we are testing, however, are large and for these systems we need to find easier ways to build automation.

Map Based Automation

Once we arrive at windows level wrappers, we are one small step away from removing SUT specific scripting of automation code from the process altogether. This may sound strange at first but it is possible. The way to achieve this is by having a driver program which interprets the '*UI Object Map'* of the user interface. We discussed UI Object Maps above. Many of the tools separate out the user interface object definitions from the actual scripts or code and put them in a 'Map'. The object identifiers (the way the test tool recognises which UI object is which) are defined in the map and the scripts or code use a logical name for each object. WinRunner for example has the GUI Map which contains UI object definitions. WinRunner TSL code uses the names of these mapped objects, rather than the actual object identifiers themselves.

UI Object Map based frameworks have an automation *engine* which, when fed with instructions from test data, recognises which object in the map is to be acted upon and then performs the specified action upon the object in the user interface.

Look at the test data shown below. The automation engine reads in each line of this data. It knows which window the instruction is for and it has the logical name of the object within that window. The actual way the object is identified is defined in the UI object map but the automation engine does not need that information, only the name of the object. The automation engine reads the UI object Map to find out what class the object is and then calls a Control Wrapper which deals with objects of that class and performs the required action upon the object, using the data if required. No automation code for a specific *instance* of an object is needed, only for *classes* of object.

Window	Object	Action	Data
Login	UserID	SET	user1
Login	Password	SET	password
Login	OK	CLICK	
MDIMain	Open	CLICK	

Figure 7: Test data for map based automation is at the UI Object level

Control Wrappers

The actions are performed on the UI Objects by *Control Wrappers* – functions which wrap UI object classes. A control wrapper has code which implements all the actions for a particular class of UI object. There will be a control wrapper for each class of UI object and all of the actions performed on the user interface are performed by the control wrappers. Nagle[17] calls control wrappers *'Component Functions'*. When the automation is running, the engine gets a line of data, gets the UI object's class from the map and calls the control wrapper for that class of UI object, passing the actions and data to it.

The importance of control wrappers should not be underestimated. Control wrappers are the key to solving the two fundamental test automation problems of maintainability and tested-ness (i.e. the automation paradox). Once a control wrapper has been written for a class of UI object, it can then be used in any automation project where that class of wrapper is used. This is automation code re-use taken to its ultimate extreme.

The Maintenance Problem Solved By Control Wrappers

The maintenance problem is addressed by control wrappers because, when the system under test's UI changes, only the UI object map and the data have to be changed, *not* the control wrappers. Because a control wrapper automates a *class* of UI object, it only needs to be changed when the behaviour of that *class* of UI object changes, which is very rare. A .Net application may have changes to its user interface in a new release, but changes to the behaviour of the classes of windows controls are very unlikely to happen. For example, Button objects in Windows can be clicked. Another possible automation action is to check that the text of a button is equal to something. A third action could be to check that the button is enabled. All of these actions will be provided by the Button control wrapper. The Button UI object class behaviour is not, and has not been for quite some time, subject to change. The behaviour of textboxes, combo-boxes, list-boxes, option buttons, radio buttons and other standard windows controls are similarly constant. For a particular development platform such as .Net or Delphi or HTML in Internet Explorer, the behaviour of the UI objects is virtually unchanging. A Delphi combo-box may behave differently to a .Net combo-box and so will require different control wrappers but once these control wrappers are built and tested, they are unlikely to need to be changed.

The only time control wrapper automation needs to be changed is when new UI objects classes are introduced or when new actions for UI objects are required. When new UI objects are introduced, then new control wrappers for those classes have to be written. When new actions for UI objects are required, the code to perform the actions has to be added to the control wrapper for that class.

The Testedness Problem Solved By Control Wrappers

As we have seen, the actions for each UI object type are always performed by the control wrapper for that class of UI object. For a Windows application with 1000 textboxes in its user interface, all the actions on all of the textboxes in all of the windows will be performed by just one small piece of automation code—the textbox control wrapper. So these control wrappers are tested by testing each action on each UI Object type. The more the control wrappers are used, the more highly tested they become.

Maintenance with the UI Object Map Approach

Map based automation frameworks contain no code specific to the SUT. The UI of the SUT is defined in the UI Object Map and the control wrappers perform all of the actions. This means that the number of lines of code in the framework is fixed. The number of LOC is not proportional to the number of UI objects in a system or even the number of windows. Actually the number of LOC is proportional to the number of different UI Object types or classes:

$$\boxed{LOC \propto NumberOfUIObjectClasses}$$

If a new UI Object type is added to the system under test, then a new control wrapper would have to be built. Also, if a new action on a UI Object is required, that would also have to be coded. However, over time, it is likely that for a development environment (HTML objects in a web page, for example) the introduction of a new class of UI Object will be fairly rare and a mature framework is likely to contain control wrappers for all the classes of UI Object. Thus, in practical terms, we can consider the number of LOC to be constant (see graph below).

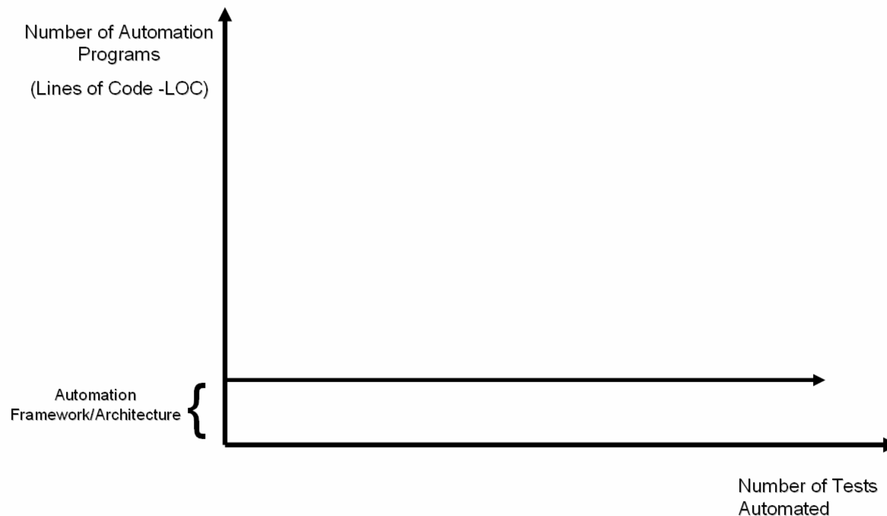


Figure 8: LOC versus number of tests for map based automation frameworks

Thus maintenance of this type of automation framework only involves keeping the map and the test data up-to-date with the changes to the system under test. Not only can you create more tests without having to build automation code, you can define more windows/screens/web pages in the GUI map and not have to increase the number of code lines at all (unless a new *class* of UI Object is introduced). All that is necessary is to define the new windows in the UI Object Map. In fact you could use the same automation engine on many different systems with the same UI Object Classes, without having to write any more automation code.

Test Re-Use

The test data for UI Object Map based test automation in Figure 7 can be considered to be at quite a low level. UI object map based frameworks can offer the ability to reuse test automation data at a higher level of abstraction: i.e. at the business language level. This also means that test data can be treated as an object itself and re-used.

Automated Map Generation

A further saving can be made in the creation and maintenance of test automation. UI object maps can be generated automatically by examination of the SUT's user interface. This can be done by building software to extract the UI definitions from a variety of sources, including from source code or from the .exe. Microsoft's .Net provides technology called Reflection which can access an application's user interface from its .exe file or a running application[23]. Once the UI objects have been extracted, UI object maps can be generated. This technique is only successful for pre-defined, non-dynamic user interfaces. Systems where the UI objects are generated as the system runs (as in many web systems) do not lend themselves to automated map generation.

Automation Maturity Model

This paper provides a route map in the search for better test automation. It can be summarised in Figure 9, below, which provides a maturity model for test automation.

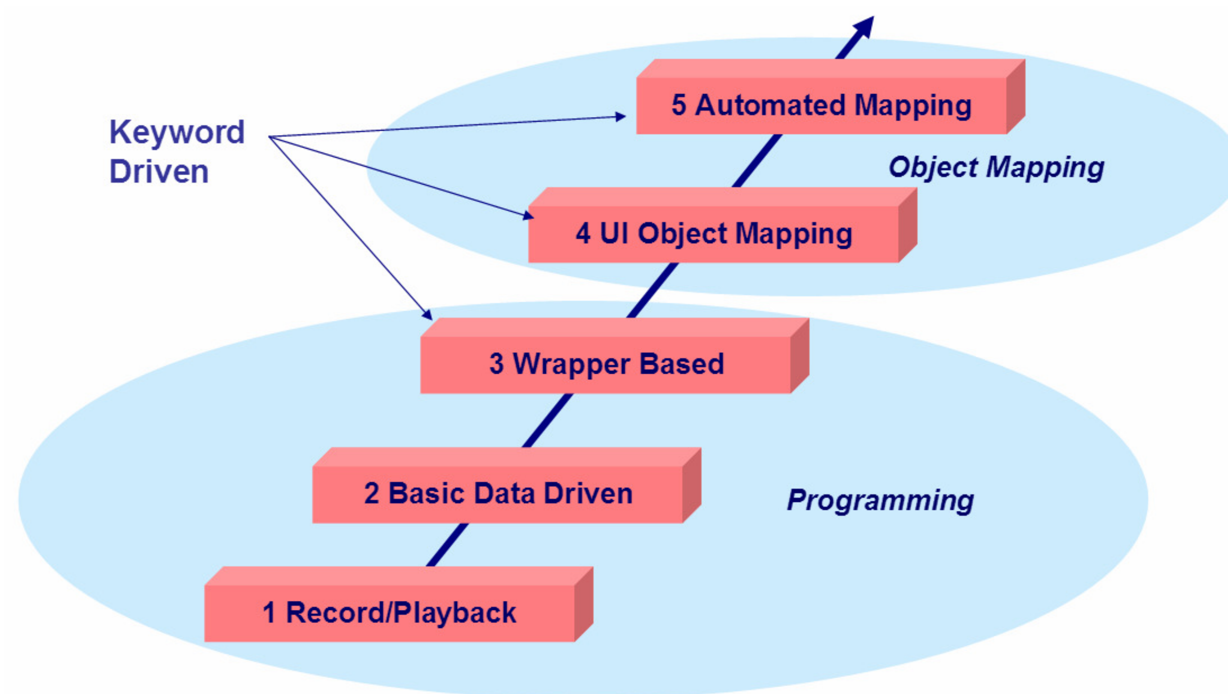
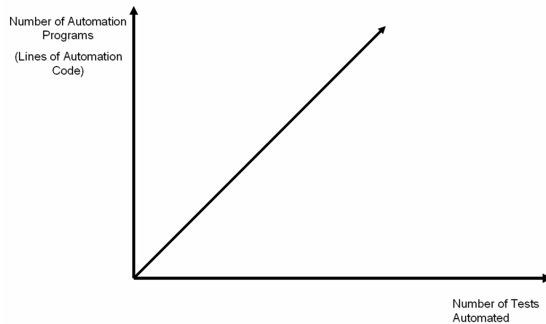


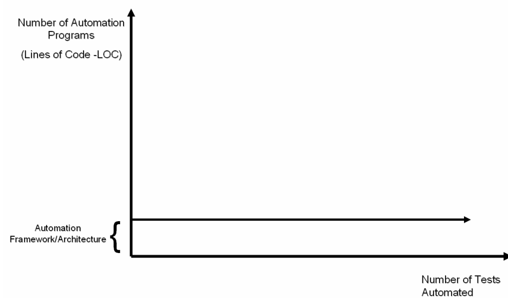
Figure 9: The Automation Maturity Model

Conclusions

- Test Automation is software engineering.
- Record/Playback automation inevitably hits limits of maintainability. It also does not provide a code structure which leads to well tested code. This is because the automation LOC is proportional to the number of tests.



- UI Object Map Approach allows scripting to be dropped from the picture altogether.
- Control Wrappers are the key to solving the maintenance and tested-ness problems.
- LOC not proportional to number of tests.



UI Object Map Automation Approach:

- Removes need for scripting.
- Reduces automation development time.
- Reduces automation maintenance.
- Provides highly tested automation code.
- Reduces risk: tried and tested automation frameworks.

References

1. Bach, James: *Test Automation Snake Oil*—http://www.stlabs.com/

2. Paul Gerrard: *Automated Testing: Past, Present and Future*—Presentation at EuroStar'98.

3. Zambelich, Keith: *Totally Data-Driven Automated Testing* http://www.sqa-test.com/

4. Kent, John: *Overcoming the Problems of Automated GUI Testing.* Presentations to STAR 1994 and British Computer Society SIGST Dec 1993.

5. Kent, John: *An Automated Testing Architecture.* Presentation to the British Computer Society SIGST July 1997.

6. Kaner, Cem: *Improving the Maintainability of Automated Test Suites*—Quality Week 1997

7. Cusmano, Michael A. and Selby, Richard W. *Microsoft Secrets: how the World's Most Powerful Software Company Creates Technology, Shapes Markets and Manages People*. Harper Collins 1997

8. Arnold Thomas, R.II: *Building an Automation Framework with Rational Visual Test*—ST Labs Report 1997

9. Arnold Thomas, R.II: *Visual Test 6 Bible*—IDG Books 1999

10. Buwalda, Hans *Testing with Action Words*—From *Automating Software Testing* (Chapter 22)—Addison Wesley Longman, 1999

11. Pettichord, Brett: *Success with Test Automation,* Proceedings of the Ninth International Quality Week (Software Research) 1996.
http://www.io.com/~wazmo/succpap.htm

12. Dwyer, Graham and Graham Freeburn: *Business Object Scenarios: a fifth-generation approach to automated testing*—From *Automating Software Testing* (Chapter 22)—Addison Wesley Longman, 1999

13. Ottensooser, Avner: *Data Independent Test Scripts*—paper presented at STAR 1997.

14. Bach, James: *Useful Features of a Test Automation System*—printed in Thomas Arnold's *Software Testing with Visual Test 4.0*—IDG Books. Also http://www.stlabs.com/

15. Fewster, Mark and Graham, Dorothy : *Software Test Automation*, 1999 Addison-Wesley

16. Kent, John: *Ghost in the Machine Parts 1-6*, Professional Tester, 2002 to 2004

17. Nagle, Carl J: *Test Automation Frameworks* (Web).

18. Kent, John: *Generation of Automated Test Programs Using System Models,* Quality Week 1999, and EuroSTAR 1999.

19. Hayes G. Linda: The Automated Testing Handbook

20. Michael Kelly: Choosing a Test Automation Framework - (Web)

21. Bass, Clements, and Kazman: *Software Architecture in Practice*, Addison-Wesley, 1997

22. Mosley, Daniel J and Posey, Bruce A: *Just Enough Software Test Automation*, Prentice Hall, 2002

23. Li, Kanglin and Wu, Mengqi: *Effective GUI Test Automation*, Sybex 2005