Generation of Automated Test Programs Using System Models

John Kent M.Sc.
Simply Testing Ltd
Email: johnkent@simplytesting.com
Web: http://www.simplytesting.com/

Abstract

Many organisations have encountered difficulties when attempting to create realistic automated tests using modern, script-based automated testing tools. As a consequence, the way these tools are used has begun to change and more effective automation *architectures* are beginning to be developed. Test automation is now often *data-driven* with actions and navigation in the test data. Test programs are *wrappers* for either business level or screen/window level objects. This paper reviews progress in this field and offers an architectural approach that may indicate future directions. The architecture is based upon *screen/window level wrappers* and *UI object wrappers* which allow for test program generation from *models* of the system under test.

The Promise of Test Automation

Automated testers are in a fortunate position in that they can have their tests running overnight while they are at home with their feet up, having a beer. Automated testers work when they are not at work. That is a productivity level which no one else can match and a great reason for being in this exciting area of software development. A large number of tests that used to have to be painstakingly done manually at a snail's pace and probably involving many testers, can now be done by an automated tester whilst he or she is at home sound asleep.

This is the promise of automation: greatly increased productivity during test execution. Software is becoming so complex that the cost of adequately testing every release of a product manually is becoming prohibitive for many development organisations. Good software is extremely expensive to create and testing it 'fully' could cost just as much again. Because testing does not have a visible deliverable in the way that development does, this is a cost that many organisations are not prepared to meet. What is needed to help test our software properly is a great increase in tester's productivity. The industry needs software to do the repetitive, boring test execution tasks and the vendors have not been shy in supplying the basic tools to do this.

Where Are We Now?

It is often a good idea to ask "where are we?" and "how did we get here?" Let's do that here for automated testing.

Has the promise of test automation been fulfilled? Unfortunately, no. The promised increased in testing productivity has been attained only by very few of those who have taken up the challenge of automated testing. This has been for a variety of reasons. Test automation is a very young branch of computing and it is easy to get it wrong. Automated test tools are sometimes believed to offer 'testing at the press of a button' and in extreme cases they are even thought of as a way to squeeze testing schedules even further. These are complete misconceptions which have not always been discouraged by the tool vendors. Many people feel that this lack of success has been due to use of the wrong tools and there has been a lot of noise about tool evaluation in the testing community. But surely we can't all be using the wrong tools, can we?

Automated testing can give real benefits but it involves an up-front and ongoing investment and the software development industry is slowly coming to realise this. Rather than simply a way of squeezing testing, automated testing should be seen as a way to do more thorough testing[1].

Many of the problems with automated testing are primarily due to the *approach* adopted rather than the fault of any particular tool[1].

The test automation community is coming to realise that the structure or *architecture* of the automated test programs and data has a major influence on the success or otherwise of an automation project[1, 2, 3]. Building automated tests with certain structures (or the lack of them) is doomed to failure within the resource constraints of most software projects. This paper explains why certain approaches don't work and describes the new architectures and the key factors in building an advanced architecture.

How Did We Get Here?

Automated Testing of Mainframe Systems

Tools for automated testing of user interfaces were first built for text-based, mainframe systems and were called 'Capture/Playback' (or 'Record/Playback') because recordings of each screen were captured when the user pressed the Enter key (or a function, 'F' key). These could then be used to play the data back into the system in order to retest the software. In a mainframe system, the user communicates with the processor (and thus the software to be tested) only when he or she presses Enter or a function key. In between each press of the Enter key, data can be input into the terminal but it is not sent to the processor until the function key is pressed. Thus the mainframe text-based automated tools did not actually record user actions, they recorded terminal/processor conversations. The tests were recordings of what was sent to the processor on pressing the Enter key and what was sent back by the software under test.

These tools were 'true' record and playback – we shall see that PC tools were not. The better tools would stop the test on a difference between the current run and the recording and allow you to accept the changes or record some more screens so that you could, for example, change incorrect reference data. Other tools would simply terminate the test run when a difference was found.

Although some of these tools were quite basic, they could be made to work, and work well. The author was involved in a test project on a mainframe system at an insurance company in the late 1980's where recordings of over 100,000 screens were successfully used to find bugs during regression testing.

Automated Testing On PCs

With the introduction of PCs and then GUIs, the user/software interaction became much more complex. With PCs, each user has his or her own processor rather than having to share one processor with many others, as with mainframes. This means that on PCs the user/software interaction is much closer. For example, in a GUI the application software is invoked not just when the user presses Enter—a GUI system can respond to any key press or mouse click or even just putting the mouse over a particular part of a window. GUI software is said to be *event-driven* because the application responds to a variety of user actions including mouse clicks, mouse movements and key presses—i.e. it is driven by user created events. Because of this closer user/software interaction, automated test tools could no-longer simply play-back the text on a screen; rather, they needed to be able to play-back the users' *actions*. The only way to do this is to have a list of actions which are performed on the SUT (System Under Test). These lists were called *Scripts* and thus the tools became *script-based*.

How Scripted Tools Work

Scripts contain actions such as *Type 'Smith' in the surname text box* and *Click the OK button.* In Visual Test, for example, a script might look like this:

```
Play "Smith"
WButtonClick("&OK")
```

You can record scripts with most of the test tools on the market. You set the test tool into *record* mode and then perform actions on the SUT. The recording will consist of a list of your actions.

These scripts are actually written in programming languages and you can write scripts manually as well —this is often called *Scripting* but perhaps ought to be called programming, because that is what it really is—more on this later.

Object Identification

Early on it became apparent that the tools needed to 'understand' the objects in the user interfaces (called *controls* in Windows—e.g. text boxes and list boxes) and to be able to interact with them quite closely. Selecting an item in a drop-down combo could be recorded as a series of mouse clicks:

```
Play "{Click 211, 55, Left}"
Sleep(7.168)
Play "{Click 184, 134, Left}"
```

but this is not very helpful or maintainable. It is better to have something like:

```
WComboItemClk("@1", "Mr")
```

which means select the item "Mr" in the combo box identified by the symbolic name "@1". It is equally important that the automation tools can retrieve data from these controls as well as input it.

The tool vendors have spent a lot of their time improving what some of them call 'object recording' like this. The automated test tool you use should include a library of functions you can call from the test tool's programming language which will interact with the objects in the system you are testing; for example, PowerBuilder's Data Windows or non-standard Visual Basic controls. The tool vendors have to produce these libraries for any new, non-standard controls. Some tools have generic functions which interact with non-standard controls but this functionality is often limited.

Automated Test Methods

Because these tools come with their own programming languages, they can be used in a variety of ways and that is what this paper is really about—how do you use the tools for the best results?

Paradigm Lost: Record/Playback

Automated testing tools are often called 'Capture/Playback' or 'Record/Playback'. It sounds great doesn't it? I've heard it a thousand times: "…all you have to do is simply record your tests and replay them whenever you like." It is a sad fact that this is not the whole truth. Record and playback looks great as well, but it only really does what is it is supposed to do in one situation: the tool vendor's sales demo. It is the ideal sales feature—you will see a great demonstration of a robot doing the work a highly paid technician used to do. It's simple and you don't need expertise to do it.

There is a growing body of opinion in the test automation community that recognises that it is not practical to create automated tests by simply recording them with script-based tools[2, 3, 4, 5, 6]. Recording does not work for a number of reasons, the most important of which is that re-runnable tests must cope with what *may* happen, not simply what *has* happened. For example, 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 recognises errors and this must be *programmed* into them. A test cannot simply be a recording played back into the SUT; rather it is an *interaction* between the test tool and the SUT. You can use recording as a way of helping to create automated tests but you will have to take them and modify them by programming. Record/Playback tools are not really Record/Playback at all, they are Program/Playback tools.

Recording also produces unmaintainable scripts that consist of a long list of references to objects on your user interface. For example, I recorded typing the above sentence with Visual Test and this is what the recording looks like:

```
CurrentWindow = WFndWndC("Microsoft Word - QW99Pap.rtf", "OpusApp",  FIND_AND_MAX,
Timeout)
Play "{Click 538, 0, Left}"
Sleep(2.374)
Play "{ENTER}Recording also pr"
Play "oduces i{BS}unmaintaai{BS}{BS}inable scripts"
Play " that"
Play " "
Play "consist of a long list of references to objects on your ue{BS}ser interface"
Play "."
```

The '{BS}'s are where I have typed a backspace. I'll try to play this back (it should type the next line) and see what I get:

Recording also produces unmaintainable scripts that consist of a long list of references to objects on your user interface.

Brilliant, so it works. So now I'll underline the word ***unmaintainable*** here and italicise it by selecting it with the mouse and clicking the Underline and Italics buttons. This is what I get when recording with Visual Test:

```
Play "{BtnDown 409, 136, Left}"
Sleep(5.232)
Play "{BtnUp 513, 137, Left}"
Sleep(2.106)
Play "{Click 302, 53, Left}"
Play "{Click 320, 53, Left}"
```

I tried to replay this and eventually, after some editing of the script got it to work. I had to remove the first sleep statement in order ensure that the word was selected and of course I took the underline and italics off in order to return to the initial state. It worked after editing, but look at this script. Not only is it unpleasant code, it is also unmaintainable (well, at least very difficult to maintain) because it contains XY co-ordinates which makes it almost impossible to identify what it is doing and to which controls. More importantly for re-running this, the SUT—in this case Word 97—has to be exactly the same state as when I recorded. That means the document should be open with the word *unmaintainable* in exactly the same place. Whilst this is not impossible, it is rather difficult and would be even more so in the middle of a large automated test. What is needed is a different approach here. For example, it may be better to select the word using the 'Find' dialog box (Edit|Find menu item) in order to test the Underline and Italics buttons.

Another reason that Record/Playback does not work is that the script can easily get out of sequence with the SUT. The script may need to wait for a window to appear—if it doesn't appear in time the script may just carry on clicking and keying anywhere. 'Wait' conditions may need to be programmed into the script.

If an error is encountered during a test run it is likely that the whole test schedule will come to a stop. It is indeed ironic that, although the objective of automated testing is to find bugs, if the SUT behaves in an unexpected way—i.e. there are bugs—then these automated tests will not work.

Finally, with Record/Playback you end up with data hard-coded into your test program … which we know from software engineering is not a good approach to building software[6].

Record/Playback may be OK for knocking together short, simple demonstrations but for large, real-life automated regression tests it is a non-starter. On the old, text-based mainframe tools it was possible to create replayable tests by recording hours of user input from more than one terminal. Try recording just one hour of input with one of the modern GUI script-based tools and see how well it replays. Without knowing anything about your system I can guarantee that it will not 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.

Disadvantages of Record/Playback (script-based tools):

- Does not work for realistic tests
- Tests need to expect the unexpected
- Replays can get out of sequence with SUT
- Recordings cannot recover from an error
- Scripts are unmaintainable jumble of object references
- Test data is hard-coded

Test Translation

Once you accept that Record/Playback is primarily a sales gimmick for script-based automation tools, you have to accept that you will need to do some programming in order to create automated tests. You may use Recording as a starting point for these programs (to give you the basic object references), but no more than this.

An alternative approach to Record/Playback is to 'translate' manual test procedures into automated test scripts written in the test tool's scripting language. Each step in a *manual* test procedure is the basis for one or more automated test scripts. This is the approach that many automated testers originally took, is still widely used, and is about as advanced as most test tool vendors get in their understanding of automated testing. The 'Test Translation' method can work but it has some fundamental limitations. It can ultimately become a victim of its own success in that the more tests that are automated, the more programming and maintenance that has to be done. If you have 50 windows in the GUI system you are testing, you may wish to create 500 tests for a system test. So now the development group have 50 programs to maintain and you have 500. Say that you get to be very good at automated testing; you may create 1000 automated tests. When you wish to test the next release of the system, the development group still has about 50 programs to maintain but now you have 1000.

The 'Test Translation' method leads to large numbers of automated test scripts which have test data hard-coded into them. You wouldn't hard-code data into programs so why hard-code test data into automated tests? Automated test scripts need to be developed, maintained and *tested* in exactly the same way as any other software. Thus the Test Translation method becomes a victim of its own success. In the example above you would be testing a system with 50 programs using 1000 programs which would require maintaining and testing (more on this later).

Disadvantages of the Test Translation method:

- Test data is hard-coded
- Script to test procedure ratio is 1:1 or worse—the more tests, the more scripts to maintain
- Many test programs (scripts) to maintain
- Many test programs (scripts) to test

A Note on Horses

Well, not really horses but horses for courses. Microsoft have used the test translation method to good effect for some time. This is due to many factors but there are perhaps three major ones. First they have a great deal of resources to throw at testing and have developer-to-tester ratios[7] of about 1:1. Second, they have an iterative development life cycle[7], repeating tests many times, and so it is more cost effective to use Test Translation than to do manual testing. Third, and perhaps more importantly, they are testing different types of applications from most of the rest of us. Most testers in corporations are employed to test database systems. Testing an insurance database for example, requires a very different approach from that used for testing a Word Processor. The main factor governing insurance database system testing is that lots of test data is required which is used in Time Travel tests because these systems manipulate data over time. This Time Travel involves—resetting the business date for different parts of the system test. It is very different from testing the functionality of a software product such as a word processor.

Generally the testing world needs a better way to automate tests than this.

Reasons For New Approaches:

I have listed some of the reasons for taking a new approach above; but there are more, so I will list them all here:

Automation Not Fulfilling Its Promise

As we have seen, lots of organisations buy automated testing tools, but the number of those who have benefited from them is low and the number who have used them to fulfil the promise of automation—

large automated tests—is even lower. Any new approach to automated testing must provide the facilities to be able to:

- Run large numbers of repeatable tests
- Run those tests unattended

As I mentioned above, when re-running a test you must cope with what *may* happen, not simply what happened when you ran the test the last time. Many testers build automated scripts in a way that is in conflict with what they are trying to achieve. They are looking for unexpected behaviour in a system, i.e. bugs, and yet they build automated tests that will stop (or worse) when something unexpected actually happens. This means that an automated test script could stop at any point and the tester cannot rely on it completing. One of the greatest benefits of automated testing will be lost—that is, having your system tested while you are at home tucked up in bed. One of the primary objectives of automated testing should be to create automated tests that can be run unattended. When your automated test finds that the SUT has behaved in an unexpected way, it should be able to *recover*. Recovery means getting the system back to the starting state (closing all the children of an MDI window for example) and then continuing with the next test.

Testing The Tests

When a new release of a system is handed to the test team and the automated tests are run against it, two things are being tested. Of course the SUT is under test as you would expect but, importantly for us here, the automated tests themselves are also being tested. In the real world most failures in automated tests are due to bugs in the test code or test data—not in the system being tested. There is no way around this; it is a fundamental fact about automated testing. We as automated testers need to embrace this fact rather than ignore it, hoping it will go away. Automated tests need to be built in a way that takes this fundamental truth into account. A major problem with the Test Translation approach is that the more tests you have, the more tests you have to test.

This is so important that we'll summarise it here:

> When you run automated tests against a new release of a system you are testing:
>
> - The system under test
> - The automated tests themselves

The Importance of Maintenance

Automated testing is a bit like trying to hit a moving target. The target, the SUT, keeps changing. There are changes between subsequent releases as well as changes during the development of a release. This is another fundamental fact about automated testing that we need to embrace[6]. Automated tests need to be light on their feet so that they can be changed quickly in order to cope with their constantly changing target.

Object-to-Object Reference Ratios

The key factor in the maintenance of automated tests is the number of times the they refer to an individual object on the user interface. The more references you have to User Interface (UI) objects, such as test boxes and combo boxes in Windows, the higher your maintenance burden (and incidentally your 'testing the tests' burden). This is the *Object-to-Object Reference Ratio*—the object to the number of times it is referenced in the test automation.

Record/Playback may have references to an individual object many times as the user interacts with that object, thus its Object-to-Object Reference Ratio is one to many. Test Translation also scores low here because the test scripts are based upon the tests themselves and an individual user interface object may be referred to by many tests. With either of these methods, increasing the number of tests your

automation performs is likely to make the Object-to-Object Reference Ratio more unfavourable. In the example given above, where the number of tests is increased from 500 to 1000 the Object-to-Object Reference Ratio could, if all of the tests use that UI object, go from 1:500 to 1:1000.

An advanced approach tries to minimise the number of times an object or control on the user interface is referenced in the automated tests—bringing the Object-to-Object Reference Ratio as close to 1:1 as possible, thus minimising maintenance and testing of the tests.

Scripts Are Really Programs

The term *script* is rather unfortunate. Automated test *scripts* are really *programs*[3, 5, 6]. A script is a list of step-by-step instructions for something, but automated test scripts need to contain conditional processing (if statements and loops—do whiles etc.). An actor in a play has a script which is a list of sequential instructions. There aren't any plays where the script says "if the audience look, bored skip to Act 3." In a play, one scene follows another, whereas an automated test should make *decisions* which are dependent upon the behaviour of the system being tested. The test cannot simply be a recording played back into the system, rather it is an *interaction* between the test tool and the SUT. The SUT may behave differently on the next test, perhaps due to an error or a change and the script should generally be able to handle new responses from the system. Thus automated test scripts are really *programs* and have to be developed and maintained in the same way as any other software.

Once you accept that automated test are really programs, you, as testers, should not have difficulty with the idea that they need to be tested. You will know that the more you test your automated tests, the more confident you can be that they will work under all possible conditions.

Redefining terms:

- Test Program—the software part of an automated test, usually written in the test tool language (e.g. Test BASIC, TSL, SQA BASIC).
- Script—a list of sequential actions to be performed in order to exercise a test. Often written using a spreadsheet, word processor or test management tool.

A Software Engineering Approach To Test Automation

Test automation has been sent down some blind alleys by the Record/Playback paradigm, encouraged along by the idea that automated tests can be 'scripts'. Test automation is a software development process and should conform to the disciplines of software engineering. The part of the automated tests written in the so-called scripting languages provided by these tools are really *programs* and must be *developed, maintained and tested* in the same way as any other software.

We need to take a software engineering approach to test automation. We certainly will not improve things if we start writing test programs from scratch each time we want to automate the testing of a new system. No, we should be taking what we already have and improving on it. We need a structured approach that aims to get the *Object-to-Object Reference Ratio* as close to 1:1 as possible. We should also have a library of functions that plug into the test automation in the same way that C++ developers get Microsoft Foundation Classes.

Advanced Architectures

In recent years a number of test automation architectures[3, 4, 5, 6, 8, 10, 11, 12, 13] have been proposed. The whole thrust of advanced automation techniques has been to move the creation of automated tests to higher levels of abstraction. These architectures allow test analysts to specify tests without having to know anything about the automated test tool and how it interacts with the SUT UI objects. The test programs deal with the low-level interaction—the test analyst uses symbolic names for objects used in the tests. Advanced architectures also move the *Object-to-Object Reference Ratio* closer to 1:1 than

with standard architectures by referencing objects in test programs rather than in the tests themselves.

An advanced automated test script architecture should provide:

- An Object-to-Object Reference Ratio as close to 1:1 as possible
- A test program structure that promotes easy maintenance
- A test program structure that is easy to test (testing the tests)
- Ability to recover from unexpected conditions
- Basic infrastructure components
- Tests specified at higher level of abstraction—using symbolic UI object names

Framework Based Architectures

One way to improve upon the Test Translation Architecture is to use a Framework Architecture. In a Framework[6, 8, 9]—called *The Functional Decomposition Method* by Zambelich[3]—system functionality is automated by functions held in the test library. Tests can then be created by calling these functions from a test program which passes parameters to them. The functions can automate low- or high-level tasks. They can be business task based—for example, Add_Cust("Smith", "John")—or system screen/window based—for example, OpenFile("xxx.doc"). These functions act as *wrappers* around what they are automating. 'Wrapper' is an Object Oriented term which is used loosely here to mean a module that handles all the interaction with the thing it 'wraps' around. Thus the Add_Cust function is a wrapper for the process of adding a customer to your SUT. Similarly the Open_File function is a wrapper for the Open File dialog box. Individual objects on the Open File dialog box are only accessed by the Open_File function. The tests themselves do not actually interact directly with the object only with the function.

Data-Driven Architectures

An architectural feature that has been around for a while is that of *data-driven* tests where the test data is held in a separate file which is read and interpreted by the automated test code. A data-driven architecture reduces the burden of maintenance and 'testing the tests' because functions are repeatedly used to interface with the SUT; thus the Object-to-Object Reference Ratio is closer to 1:1 than with Test Translation methods.

In early data-driven architectures all of the navigation was handled in test programs. The test programs would navigate to a screen, read the first line of data, input it, read the second line of data and input it and so on until all of the data had been input. The program might then go to another screen, open up another file and do the same thing. Thus the test data was just that—data. The sequence of actions was governed by the test program.

Data Driven Architectures With Navigation Extracted From The Test Programs

The next step in the development of an automated testing architecture was to remove the navigation from the test code and move it, or at least some of it, to the test data[3, 4, 5, 10, 11, 12, 13]. The test programs are instructed what to do by the test data in the same way a manual tester is by a test procedure document. This means that it is possible to put behaviour into a test data file. Thus test data files can control the behaviour of the SUT and navigate around it as well as simply providing data to be input. Zambelich calls this *Totally Data-Driven Automated Testing*[3].

With this approach the test programs are no longer tests at all; rather they are wrappers that interface the tests to the system under test. The test data becomes the test script in that it is a list of instructions that drive the software.

This is really a combination of basic data-driven and framework architectures. The tasks are automated by wrappers that have parameters—test data—passed to them, only this time the test data is held in files rather than in a test program. As with framework based architectures this can be done at the *business object level*—for example, Buwalda's ActionWords[10] and Ottensooser's Data Independent Test Scripts[13]—or the *screen/window* level, as in the author's approach[4, 5] which will be discussed in depth below.

Business Object Level Architectures

In Business Object Level Architectures, business *tasks* are automated. Test data for an automated test of this type might look like:

    Create_Cust   "Smith", "John", …
    Create_Account "Smith", "John", "Cheque",….
    Debit_Account "Smith", "John", "$1000",….

Thus much of the navigation is in the test data. As you can see from the above example, test data in Business Object Level Architectures is at the business language level and therefore understandable by end users, which is a great advantage[10]. This approach also gives a lower Object-to-Object Reference Ratio than for Test Translation because the UI objects are referenced only on a per business task basis.

Screen/Window Level Architectures

Figure 1 shows the Screen/window Level Architecture approach. In this architecture there is one test program that deals with each screen/window in the system—it acts as a *wrapper* for that screen/window. It handles all of the input and output (getting values) from it. The test data is held in a separate text file and is passed to each test program as it is needed. Thus the navigation is in the test data. This structure gives a ratio of Window-to-test program of 1:1, and thus also an Object-to-Object Reference Ratio of 1:1, so if a change is made to a window in the SUT, only one program needs to be changed and tested.

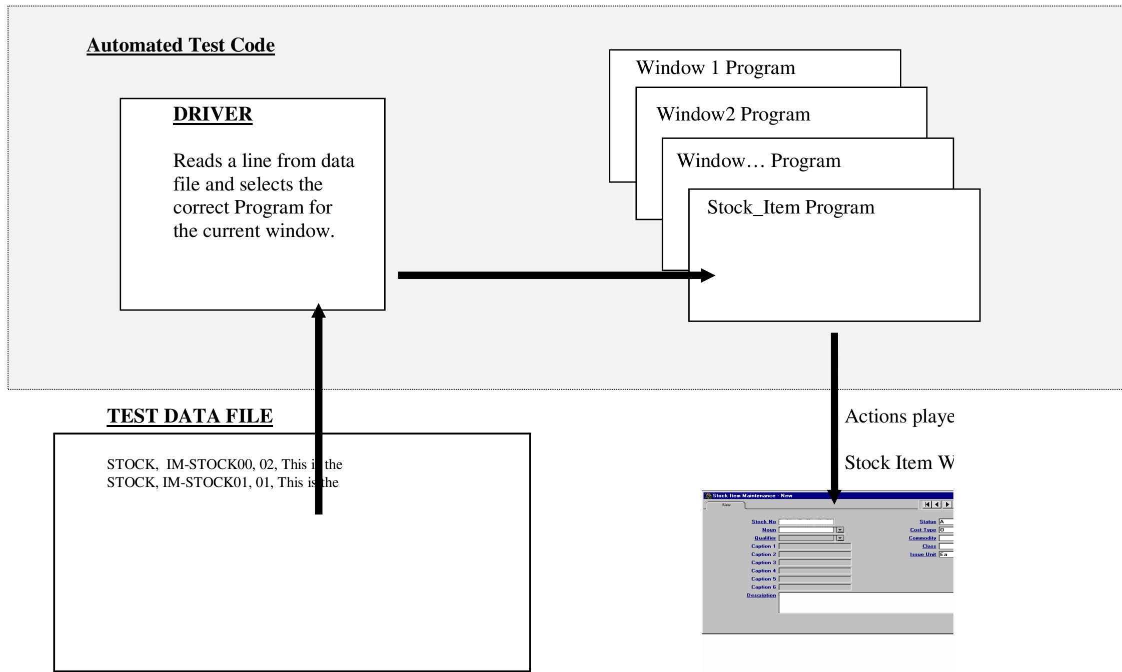

Fig. 1 Data-driven tests: **One test program per window in the SUT in this case.**

An Advanced Automated Test Architecture

The author has been developing automated test architectures since 1992 for both GUI and text-based SUTs using script-based automation tools. Simply Testing Ltd has developed a tool called *TestCoda* which is used to implement this architecture and incorporates test management, system modelling and test code generation facilities (more on this later). The architecture incorporates all the advanced architectural features we have discussed so far.

- Object-to-Object Reference Ratio is 1:1 or 1:2—minimising maintenance and testing
- Completely data-driven, including all navigation
- All test and user actions are in the test data
- It takes the Window/screen-level approach
- Each *class* of UI object is 'wrapped' in just one (or sometimes two) functions
- It has infrastructure components; e.g. recovery, etc.

This architecture is implementation-independent—i.e. it can be implemented with any script-based test tool.

Architectural Overview

The Data

The data needed for each test is held in a comma-separated file created using Excel spreadsheets. Figure 2 shows an example of part of an Excel spreadsheet with test data for an actual test carried out recently on a textbased AS/400 system. Although this example is for a text-based system, automated tests of GUI systems have been done using the same principles of this architecture but implemented in slightly different way.

#Screen	IMOM00 =>IMACS/400 Main Menu							
#	TEST	TESTSTEP	AUTOSTEP	RUNDATE	ACTIONKEY	TESTACTION	DFMOPT	
#							B L=2	
IMOM00	POLGIT01	1	1		ENTER		21	
#Screen	IMOM21 =>Underwriting Maintenance						===>	
#	TEST	TESTSTEP	AUTOSTEP	RUNDATE	ACTIONKEY	TESTACTION	DFMOPT	
#							B L=2	
IMOM21	POLGIT01	1	2		ENTER		2	
#Screen	IMPL02 =>Register New Policy						Renewal/ New Policy	Policy No.
#	TEST	TESTSTEP	AUTOSTEP	RUNDATE	ACTIONKEY	TESTACTION	DFOPTN	PLPLNO
#							B L=1	B L=7
IMPL02	POLGIT01	1	3		Enter		N	
IMPL02	POLGIT01	1	4		Enter			
IMPL02	POLGIT01	1	5		SHIFT+F2			

Fig. 2 Data-driven tests: EXCEL spreadsheet test data for a text based system.

The lines in the spreadsheet that start with '#' are comments and ignored by the automation program's

outlines. All other lines are actual test data and instructions. The comments in this case are *screen descriptions* which have three lines. The first line in the screen description has the screen name and the field labels on the AS/400 screens to be tested. The second screen comment line contains the internal AS/400 program names of the fields (e.g. “DFMOPT”, “PLPLNO” and “TESTSTEP”). The third line contains B or O for each field meaning “**B**oth input and output” or “**O**utput only” for each field. This line also has the maximum length of the field (e.g. “L=7” means length is 7 characters maximum). The comment line’s only function is to give information on each screen to the person creating the test data.

The first seven columns of the actual test data are standard. Column 1 contains the name of the AS/400 screen to which that particular line applies. Columns 2, 3 and 4 contain the test name, test step number and automation step number. Column 6 contains the ActionKey—this is the function key that is to be pressed after all of the data for that line has been keyed in or checked. For example “ENTER” or “F3”. Column 7 contains the TestAction which can be set to perform actions like ‘CHECK’, which checks the values in the spreadsheet against values on the screen.

How The Test Automation Works

This data is read by a driver program which calls the test program for the screen named in column 1 of that line of the data. It passes all of the data to this program, which completes its tasks and returns processing to the driver. The driver then repeats the process for each line of test data (see Figure 1 for an overview of this process). So, for the data in Figure 2, the driver will read the first line and call the test program IMOM00 (in this case the test program has the same name as the screen it tests) which deals with the IMOM00 screen—the main menu of the SUT. The IMOM00 test program keys ‘21’ into the DFMOPT fields, presses Enter (the ActionKey) and returns control to the driver. The driver reads the next line of data and calls the test program specified in that line—IMOM21 and so on. Thus, it can be seen that there will be one test program for each screen in the system (this is a slight simplification for the sake of clarity—it’s one program per logical screen).

Architectural Structures

Each test program is a wrapper for the screen it tests. It contains all the information it needs on each UI object in the screen/window so that it can do the tasks required of it, such as input test data and check values. Thus the Object-to-Object Reference Ratio is the ideal 1:1.

When a test program runs it does not actually input test data or check data values itself; rather, it calls functions that deal with each type of UI (see Figure 3), These functions are object wrappers for each *class* of UI object. There will, for example, be one function for text boxes that will handle all of the test data input into all text boxes in the SUT.

Thus there are wrappers within wrappers. There are wrappers at the screen/window level—the test programs which themselves call wrappers for each type of user interface object—the functions for that object class.

In the large AS/400 test that the test data example is taken from there were only four functions that input or checked test data values—two for each of the user interface object types:

- 2 functions for text fields
- 2 functions for Subfile fields (AS/400s version of scrollable list boxes)

For the whole of a large regression test lasting over 60 hours of automated testing, there were only four functions actually inputting the test data and getting screen values.

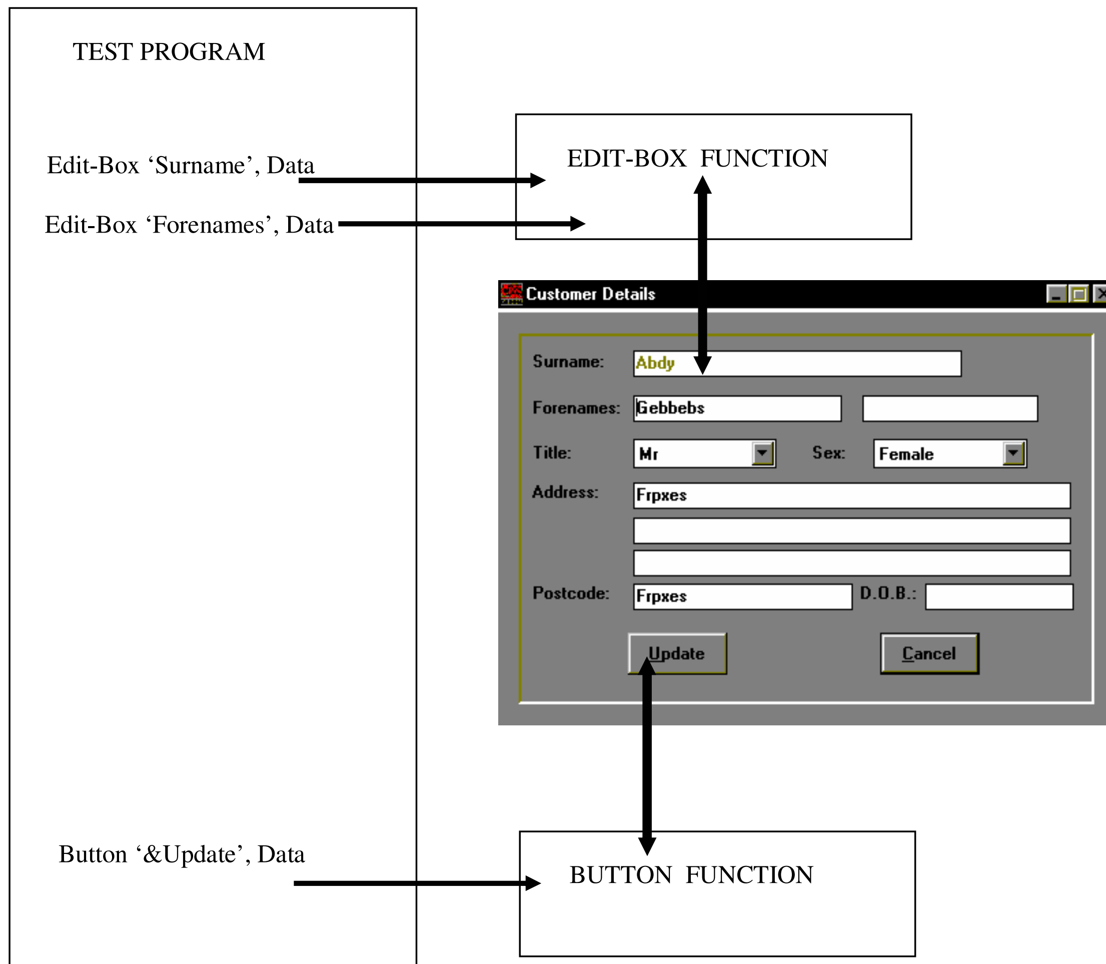

Fig. 3 Test programs call functions for each UI object class.

UI Object Wrappers

The behaviour of UI objects—text boxes, combo boxes, etc.—is well understood because development build the system using these components. Thus generic wrappers for these objects can be built based upon their behaviour. Functionality can be built to get and set their properties and initiate their methods—i.e. do things to an object that a user might do.

The Object-to-Object Reference is still determined by the test programs because they contain the references to each individual object (each instance), but the program code that actually interacts with those objects is held in the function that deals with that object class. Because the input of test data and the checking of results are handled by one (or maybe two) functions per UI class—e.g. one function inputs data into all of the Windows text boxes in the system—a high level of confidence can be attained in these functions. They are repeatedly used and thus tested.

Test Program Generation

A screen/window level approach has been chosen for this architecture because it offers one major

advantage over the business object level architecture. Because the behaviour of the UI objects within a system are defined and known and the behaviour of the screens or windows can be defined, it is possible to generate the test programs for each screen/window from descriptions of those screens. The *CISSTest* tool is used to generate the test programs. It does not actually run the tests itself; it generates test code for the chosen automation tool. Currently it has been used to generate Test BASIC(MS-Test and Visual Test), TSL (WinRunner) SQA BASIC (SQA Robot) and Autotester Outlines. The generators are constantly being improved and *TestCoda* is now close to generating 100% of the test code and in some cases actually 100%. Some manual programming effort has been necessary in order to cope with unpredictable system behaviour. This unpredictable behaviour is usually in the area of system generated keys (see the section on Key Translation below).

The basic architectural components can be used to test many different types of system; only the actual screen/window programs must be generated for each SUT, but this generation can be based upon descriptions of the SUT that can come from the system's source code, or from automatic inspection of the actual running executable. Alternatively, these descriptions can be derived from system design documents in order to start building test automation for a system that has not yet been delivered. In the AS/400 example, the test programs were generated from the AS/400 screen definition files (DDSs).

Actions At The User Interface Level

Another reason for choosing the screen/window level for this architecture is that all of the UI objects (and therefore their methods) that a user can access on a screen or window can be made available to a tester when creating the automated test data and actions. This includes selecting any button or menu item available. Business Object Level automated testers believe that this level of detail at the test creation stage is too much of an overhead and impossible to automate, but if your test programs are *generated* from the SUT it need not cost you anything to have access to any object in the SUT. The benefit of having all of the possible UI objects available to the test analyst at the test data creation stage is that it gives them complete freedom to perform any action on the system. With the Business Object Level, only the business tasks that have been automated can be performed.

Automation Is Based Upon A Model of The System

The screen/window descriptions provide a model of the system which is used not only to generate the test programs, but also to create the spreadsheet format for test data creation. The screen descriptions in the test data in Figure 2 were generated from AS/400 source files (DDSs) downloaded onto a PC. The TestCoda tool provides system modelling and test program generation facilities as well as test management and test *data* management. It has functionality which loads the screen/window descriptions, generates the test programs for each screen/window and generates the test data spreadsheet format for each screen/window.

The automated tests (the test data) are based upon *test procedures* which are documented within the TestCoda database in the same way that manual tests might be. The test procedures consist of *test steps* and the automated test data for each test step is contain in a series of 'AutoSteps' (see col. 4, Figure 2).

Test Object Re-Use.

One of the great advantages of the Business Object Level Architectures is that the test data is created at the business language level and therefore understandable by end users. The architecture we are describing here is built at a lower level—the screen/window level—and so the actions in the test data are at this low level. You can see from Figure 2 that the actions in this real test data are at the screen level and quite detailed. This provides the advantage that *all* of the UI objects—and therefore the possible user actions—are available for the tester to specify in the test data but it also means that the test data cannot be specified at the business language level.

Some facilities have been provided in the TestCoda tool so that tests can be re-used. Basic business

tasks are automated and the test data is saved off as test objects or *Templates* which can then be used to create tests. This is not re-use in the Object Oriented sense at all—there is no inheritance here; it is really just copying but it currently serves its purpose of allowing a test analyst to create tests by stringing together business level test objects. One of the more important tasks for the future is to build true inheritance into test objects, thereby allowing us to build tests at higher and higher levels.

Other Features of The Architecture

Because each component of the architecture has a standard structure, it is relatively easy to include lots of additional test facilities in it. As structured architectures such as this one mature, they start to become very feature rich in terms of test automation features. A good source of information on such features is provided by Bach's *Useful Features of a Test Automation System*[14]. Here's a list of some of the standard features currently included in this test automation architecture:

Checks

It is not enough to test a system by simply inputting data. A test consists of input and output—checks against expected values. This is especially true in test automation where inputting data to a system is often much easier than getting output from it. This can be done using the 'CHECK' TestAction where a screen value is checked against a value in the test data. In Figure 4, below, the test automation will check that the risk class field contains the value 'APL' and the Underwriting Year field contains '98'.

#Screen	IMPL02 =>Register New Policy						Risk Class	Underwriting Year
#	TEST	TESTSTEP	AUTOSTEP	RUNDATE	ACTIONKEY	TESTACTION	RISKCL	UWYEAR
#							B L=3	B L=2
IMPL02	POLGIT01	1	23		F3	CHECK	APL	98

Fig. 4 A Check in test data.

Having the expected values for a test in the test data is a much better way to do things than having them hidden in some 'check file', as is suggested by many tool vendors' user manuals. The standard automated check functionality provided by many tools does not lend itself to easy maintenance and is similar to hard-coding the test data.

Key Translation

The architecture can cope with system-generated fields. This usually applies to key fields—for example Policy Number or Customer Number—where the SUT generates a value for them. While writing tests, a test analyst may wish to create a business object (e.g. a Policy or Customer entry) and then, later in the test, refer to it for some reason. If a business object uses a generated key, the test analyst has no way of knowing in advance what the value will be. These values can, however, be represented using symbolic names. The names are specified in the test data and, once set, can be used later in the tests. The test programs save the symbolic name and its system generated value, looking it up when-ever it is used in the test data. This is known as Key Translation and is extremely useful for some SUTs.

Recovery

Automated tests should, as much as possible, be built in a fault-tolerant manner so that they can run unattended. If the system behaves in an unexpected way, the test programs should be able to recover and continue. *Recovery* functionality provides a means of dropping a test and getting the SUT back to a known base state so that the next test can be run—e.g. closing all the children of an MDI window. It is used whenever the automated test programs discover that they are in the wrong place—i.e. that the wrong screen/window is displayed and the SUT is in an unexpected state.

In this architecture the SUT is checked at the beginning of every test program in order to ensure that the

screen/window displayed is the appropriate one for the test program. If it is not, recovery is invoked.

Reporting

Large automated tests can generate massive report logs, especially if you use the vendor supplied logs which detail events at a very low level. The generated test programs within the architecture described above report events in a way that allows the tester to view results at different levels of detail. The resultant reports can be tied back to the test procedures on the TestCoda database. If any AutoStep within a test step fails then the test step itself fails.

Performance Measures

Because the architecture is screen/window-based it is relatively easy to incorporate performance measuring functionality in the generator so that a standard set of measures can be applied to each SUT action.

Test Data Management Support

TestCoda provides support for the test data spreadsheet creation process in the form of EXCEL macros. These provide facilities to:

- Copy screen layouts into test data spreadsheets
- Copy Templates for test object 're-use'
- Roll dates forward (much used recently in Y2K projects)
- Scan data for specified values
- Scan data for occurrences of specified fields

Once you have overcome the major problems of automated testing and actually build and run large automated tests you find different problems. Test data management becomes more important as the failures in the automated tests tend to be related to incorrect test data.

Implementation of The Architecture For GUI Systems

The test data examples given (see Figures 2 and 4) are from a text-based system. Most of the actions in this case are specified in the ActionKey field. There is only one ActionKey column per screen in the spreadsheets because the possible user actions for a textbased system of this sort are essentially at the screen/window level—i.e. pressing Enter or function keys. For GUIs this will not do, as we need to set actions for each object on a window, and therefore the implementation of the architecture is different in terms of test data creation though the fundamental principles remain the same.

Conclusion

Test automation is in its infancy and there is a long way to go before we can say that most organisations benefit from it. Although the tool vendors have delivered some helpful new features over the past few years the main problem has been due to the wrong architectural approach when implementing a test automation tool. Test automation needs to be recognised for what it is—a software engineering activity—and well-understood software engineering principles apply just as much to the creation of automated tests as to the system under test. Once advanced architectures are fully understood, realistic, large-scale automation will gain wider acceptance.

Test automation is already moving away from simple 'scripting' towards using the test tools to create wrappers for the objects on the user interfaces under test. Test actions have been moved out of the test program code and into the test data. Along with this, the actual tests themselves are being specified at a higher level.

This main thesis of this paper has been that architectures that create wrappers at the screen/window level can allow for test program generation to be used to create most of the actual test program code. This can be done because the behaviour of UI objects is well defined and the automation functionality required

can be provided in a wrapper for each class of UI object. The generation is based upon a model of the system which can be derived from source files such as screen or window definitions.

Another advantage of building automation architectures at the screen/window level is that all of the UI objects and possible user actions can be made available to the test analyst at test data creation time. Although this also means that the actions are not at the business object level, test creation can be moved up to this level using test object re-use facilities. This is not yet fully provided for. Once this level is reached, automated test creation will be very similar to the creation of *manual* test procedures except that the format and names of the user interface objects will be derived from the system model.

The key to the effective implementation of new architectures is to base them on a model of the system to be tested. The model can then be used both to generate the object wrappers (the test programs) and as a basis for specifying the test procedures in a format that the automation programs understand. These models can be created from the definitions of the system's user interfaces or can based upon function specifications if the system is still under development..

References

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

Field Code Changed

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/

Field Code Changed

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: *Software Testing with Visual Test 4.0*—IDG Books 1997

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

Field Code Changed