December, 2009

testing experience

The Magazine for Professional Testers

Standards – What about it?

printed in Germany
print version 8,00 €
free digital version
www.testingexperience.com
ISSN 1866-5705

Test Automation: Salutations to the World

by David Harrison

In a previous article [1], the role of test automation in a software development project context was set out. In particular the success of test automation on Java-Swing projects was described.

Whilst this article hopefully gave you, the reader, a basic grasp of this challenging topic, which is often so poorly described in the testing and quality assurance literature, it left open the low-level details of how the industrial-strength approach described was architected. It is the intention of this article to demonstrate, via a practical example, highly relevant to the test automation domain, together with it's companion Part 2 article, the key elements of the pattern of solution. It should be noted that, as set out in [1], the approach to test automation is essentially a programmatic one, and the reader needs to be open to this aspect. Once this fact is grasped, the means to obtain efficiency, scalability and effectiveness of a solution is basically the same as that for any other software development task. The unique example application, showcased in this article, Hello World!, was crafted by Peter Walser who specialises in the construction of advanced applications using Java-Swing and Eclipse RCP technology. Peter can be contacted on pwalser@frostnova.ch.

Say Hello to the World

Our example Java-Swing application is a test automation version of the ubiquitous Hello World! example – after startup it displays a textual greeting to the planet Earth, see Figure 1, below:

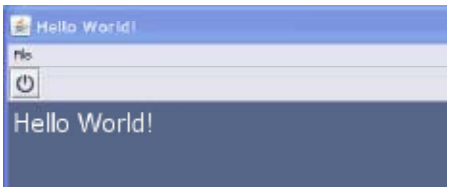

Figure 1 – Hello World!

However, to get this friendly, earthly greeting, we must first login immediately at application startup. This small piece of functionality is extremely common in industrial software and happens to be the, almost universal, example used in the automated testing literature when authors describe their approach to automation (e.g. [3]-[6]). So for these two reasons, it's a very useful feature to have in our example application. The login panel is as shown below:

Figure 2 – Login Panel

This application is designed to allow the user to obtain friendly earthly salutations. In particular we will login, initially, using the credentials "test/test123", in which case we see the eventual result as shown in Figure 1, the main application panel shows the greetings "Hello World!". However, the twist to this login story is that with alternative credentials "pluto/pluto123", we get a completely different message.

Getting different outcome on the UI of a software application depending upon the login credentials is a very common situation. For example, we might see the following:

- main panel title embellished with the logged-in user name
- a status message reflecting the logged-in user name
- other screen embellishments that reflect the access rights of the user

So this example embodies these issues.

Test Structure

In essence, then, our automated test must have the following structure, expressed below as pseudocode:

```
Start application
Wait for application to become
established
Wait for the Login panel to ap-
pear
Validate Login Panel
Enter user-name - "test"
Enter user-password - "test123"
Press Login Button
Wait for login panel to be dis-
missed
Wait for main application panel
to appear
Validate the UI elements of the
Main panel
Terminate the application
```

The structure shown here contains two Static Validation Test elements – validation of the Login panel and validation of the Main panel. This Static Validation Test concept is one explained in more detail in [1] [2]. Suffice it to say, that in this example the lack of real "business" Use Cases means that these forms of test represent the limit of what is possible.

The Automation Challenges

The above test case is deceptively simple. Why is that?

Well first of all our test, as well as starting the target application, the SUT, requires that we wait for the application to become established, in this case, within the context of the JVM, and then that we wait for the login panel to appear. These two steps require the test automation tool to have an intimate knowledge of when objects, the login and application panels, become fully instantiated as well having the capability of raising an event to signal this important outcome. In addition, we have the companion issues of intelligently tracking when objects are dismissed, essential in order to proceed to following test steps with the application in a known state. These issues, captured well in this seemingly simple example,

are extremely well handled by QF-Test[1], and this is the automation tool we use here. The singular attributes of this tool in the Java-Swing area have been noted elsewhere [7].

Let's see how we meet, and overcome, these challenges…

The Test Project

Within the QF-Test tool, the test structure, the test project, is organized as shown below:

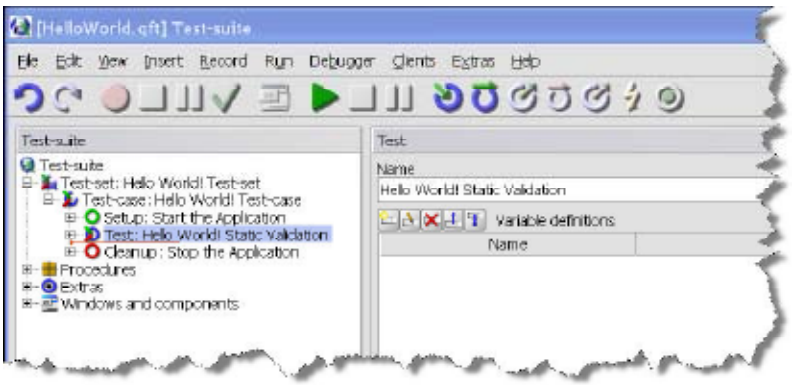

Figure 3 – Test Project

Here you can see that the test, the `Test-case`, is somewhat similar to that which would be recognizable to a developer involved in Unit testing. Within the `Test-case` we have a `Setup` element, then a `Test` element followed by a `Cleanup` element. As with Unit testing, the strategy here is that *before* each `Test`, the `Setup` action is executed and *after* each `Test` the `Cleanup` is executed. This is a *very* good structure for test automation, since prior to each test we start the application and following each test (successful or not) the application is terminated. Hence, each Test is executed with the application in a known and clean state. This is a fundamental strength of our pattern of solution.

The Detail

Let's take a look in more detail at the parts of our Test project (Figure 3).

The Setup Element

The `Setup` element starts our Hello World! application via a specified batch file. The mechanism here follows the documented approach from QFS. Expanding the `Setup` node in the project structure, the content is as shown in Figure 4 below:

Figure 4 – Setup Element

As can be seen, we "call" a Procedure to start our application, with the parameters specifying the name and location of the appropriate batch file (using Jython conventions for variables, in this case specified globally). Why do we use this approach? Well, proceeding like this means that we gain the benefit of reusability of the test "code" we write – the Procedure neatly encapsulates everything needed to securely start any application via a batch file, and thus can be re-used in any testing project.

Within the Start Procedure we use a built-in QF-Test Start SUT node. Once the application is started, a login panel appears and our test design must take account of this. Let's look at what this Procedure does in pseudocode form:

```
Try
        Use Start SUT client
node using supplied parameters
        Use Wait for client node
        Call Procedure to per-
form the login step
        Return true
Catch ClientNotConnected
        Return false
Catch DuplicateClientException
        Return true
```

As you can see the structure is fairly simple, the task of starting the target application is broken down into a number of steps; one that calls a QF-Test Start SUT function which literally starts the executable via its batch file, one that waits for the application to announce that it is fully started, and yet another which then deals with the details of the login task. Here, we may base the design on the strong exception handling capability of QF-Test. The pseudocode above is expressed in the project as shown in Fig 5-.

The procedure that logs into the application with specific (globally defined) credentials is as shown in Figure 6.

In this Procedure, we wait for the login dialog to become *fully established* and then enter the credentials. It should be noted that we take account of any (*unexpected*) blocking Modal Dialogs that arise e.g. user name not valid, password not valid etc.

It should also be noted that we take account of the special characteristics of the `JPasswordField`, when writing the password text by using generic library calls.

The Test Element

The intention of our test is to perform a Static Validation [1] of our main panel. This form of test is a *very important* one in the test automation domain. Why is this? Well, prior to performing any sort of business-related workflow test, we *must* ensure that the state of the application is as expected. In particular, we will need to verify elements such as:

- the main menu – both structure and enablement
- the toolbar – both button enablement as well as tooltip text
- main panel decorations – such things as dates, title text, memory monitoring elements and any user-specific elements

In this example, for simplicity we restrict ourselves to validation of the message displayed and the main panel title.

(See Fig 7)

Some important observations can be made about our Test element:

- we use Procedures to encapsulate the logic of main panel validation (`Test-case Support.ValidateMainPanelText`)
- we use a Procedure to validate the (`JLabel`) textual greeting (`Utility.Label.CheckText`), one parameter of which specifies the expected greetings
- as well as catching exceptions (`UserException`, `UnexpectedClientException`), we throw them too – specifically to signal that the test cannot continue (`UserException`) – a very clean and elegant approach which leaves our test uncluttered by tedious logic, thus improving readability and maintainability.
- we use QF-Test scripts to perform logging

The Cleanup Element

In this element of our test, we simply "kill" the application (it may be in a bad state, any menus may not be working). Figure 8 shows the details of the Stop Procedure, called from our Cleanup element.

Where Are We…

In this relatively simple example of test automation , a number of important themes have been touched upon. These can be summarized as follows:

- using QF-Test we have a genuine capa-

1 www.qfs.de

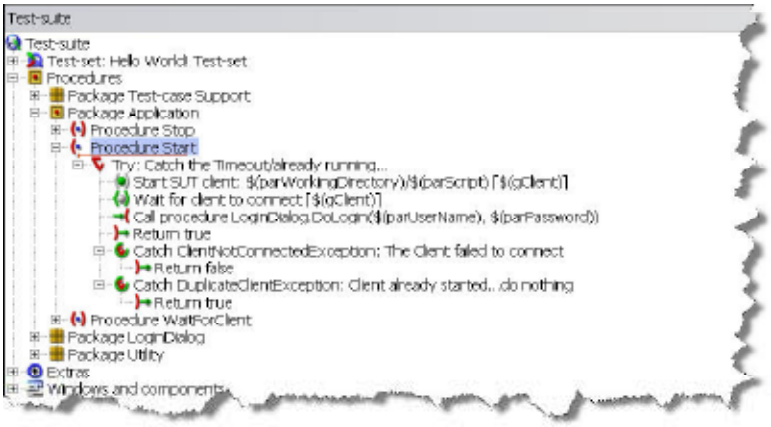

Figure 5 – Start Procedure

Figure 6 – LoginDialog.DoLogin Procedure

Figure 7 – Test Element

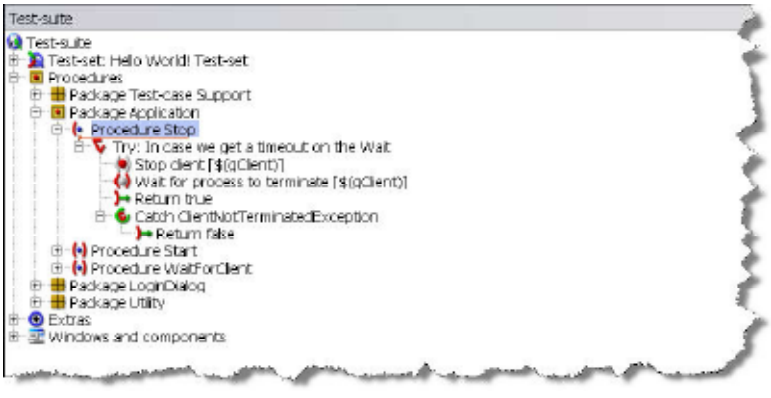

Figure 8 – Stop Element

bility to *design* our tests

- the use of Procedures delivers to us as test automators, the same benefits that they do to the general software development community
- the availability of a first-class exception architecture provides crucial benefits in responding to the SUT state
- although only hinted at in this example, we have the capability of developing a generic automation library of Procedures which allow us to interact with Java-Swing objects such as `JLabel`, `JComboBox`, `JPasswordField`, `JTable` etc. This is a crucial observation for real-world, project-based automation efforts.

Say Hello to Pluto

In real-world projects the automator is very often faced with software that, to one extent or other, contains UI elements that change in relation to such things as:

- The currently "logged in" user
- The current date/time
- The currently open file/document within the application
- The current "pending edit" state
- The machine on which the application is running
- The current number of elements in a collection, e.g. number of open documents, number of results found by searching

A good example of this variation is the way in which most applications which deal with documents, embellish the main application frame with a title that carries the current document title as well as a mark, often '*', that indicates that there are pending edits. When QF-Test responds to object creation, the *default component naming mechanism* takes these specializations into account. So, for example, if we attempt to automate a Search functionality for which the Search panel carries a result `JLabel` like:

```
Hits = 120
```

to inform the user that the Search resulted in 120 result elements, then this represents a case for which the *default* Name Resolving process will produce different values of component ID as differing result counts are found through searching. In such a case, if we recorded the `JLabel` component, we might see a default component ID like:

```
Label_Hits_=_120
```

But in the case where we see an outcome involving 1200 result elements, we would see:

```
Label_Hits_=_1200
```

Given this situation, if we develop our tests using the former component ID, then when we want to address the count `JLabel` again for the situation with a different result count, *the component will not be found*. On first reflection, this situation may seem somewhat catastrophic for successful test automation. Well, QF-Test provides us with a very *elegant*, *practical* and *powerful* solution to this problem – the Name Resolver API.

As a simple example of what all this means, let's revisit Hello World!, our good friend from earlier discussions. By design, with this application we can login as a different user, other than "test/test123", the credentials we used earlier, and see a different set of important visual outcomes. In particular, we now say hello to a different planet! Let's see how this works…

Saying Hello Again

The alternative set of credentials we can use for Hello World! is "pluto/pluto123". In real situations, the application being automated may have a large set of credentials that are permissible when it comes to logging in. This "unboundedness" (for all practical considerations), if it leads to variations in the visual elements of the application, should be the alarm bell that we must enhance the name resolving process.

If we take a look in the "Windows and components" section of the Hello World! project, and expand the tree view, then the `JLabel` component (when we logged in with "test/test123") can be located, as shown below:

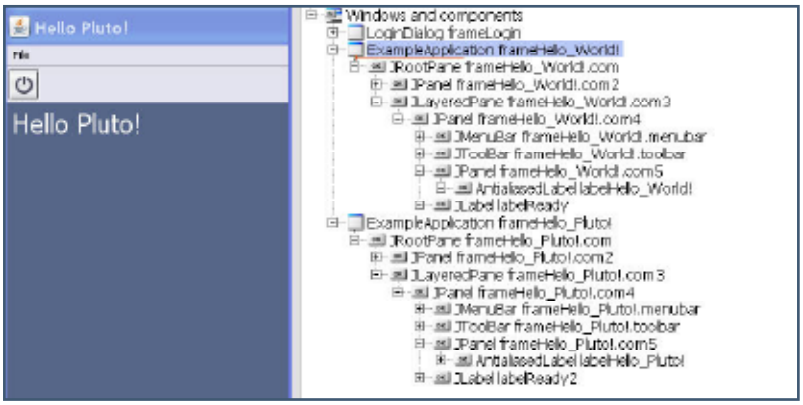

Figure 9 – Hello World Label

In this figure we can see that the main panel of Hello World!, as well as the relevant Main Window JLabel component are highlighted. The component ID of the JLabel is

```
labelHello_World!
```

just the form we described earlier. In addition, we can see that the main application frame is also decorated with the specialized text, e.g. frameHello_World!. In addition, all the components below the main application frame are given names that involve the specialized text. If we now look at the situation when we use the "pluto/pluto123" credentials, and re-record this entire main application frame [8], then the situation as shown below emerges:

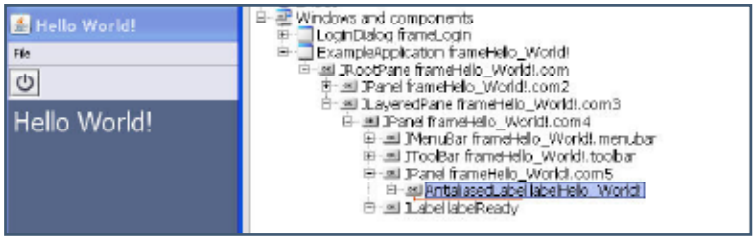

Figure 10 – Hello Pluto! Components

So now we have our "Windows and components" section containing two top-level elements, essentially the same components, but having different names/IDs, only brought about because we have logged in with different credentials. As an alternative strategy to extending the basic QF-Test Name Resolver, we could get the developers to do setName() on the UI controls. When this is done, such controls do not have their name overridden by QF-Test.

This is a great strategy to solve this component naming problem. However, typically on projects, the test automation workstream gets active only later in the overall project life cycle, and this makes sense as the software, particularly in relation to its UI, is becoming stable. At this point, developers are often heavily involved in getting the business functionality integrated, for example, and asking them to go through the code and do a setName() on controls is probably not a practical proposition. Having a strategy for resolving this naming problem that is in the hands of the QA workstream, then, has definite appeal.

Let's set about resolving this conundrum using the Name Resolver extension API within QF-Test.

The two components for which we want to provide a new, neutral name are the main application window (Class: idx.loginapp.ExampleApplication) and the JLabel that ends up being displayed on the main application window area (Class: idx.gui.text.AntialiasedLabel), and you see below how a test is made in our Name Resolver Jython script, using methods of the Resolver Registry object, to determine if the class type of the current object is one of those in which we are interested.

(See Fig 11)

This Name Resolver script is "installed" as part of our Setup phase of each test [2].

In the case where we detect that we have an object of the main application class type, then we return the neutral name "APPLICATION_FRAME", and if the current component is of the label class and if the text property has the prefix "Hello " and the postfix of "!", then we return the neutral name "Hello_PLANET".

Once this Name Reseolver script is installed [8], then as these components are brought into existence within the application, the component Name/ID will be assigned these neutral names. When we record these specific UI objects [8], we will see these names/IDs instead of the default ones assigned by QF-Test.

Now we can, with confidence, develop our automated tests using these neutral names.

The importance of this Name Resolving approach cannot be overstated when considering its effectiveness at dealing with this "component naming" issue – one that bedevils test automation generally.

Now when we login with any alternative credentials, all is well - the elements we need to find can be found and their properties asserted.

Now that's a pretty good outcome for test automation…

Summary

Hopefully, in this article, a number of the crucial steps along the road to successful Java-Swing test automation have been illuminated. The issues highlighted, as well as the tool attributes which help us meet the challenges, cannot be overstated in their importance to a successful project-based approach. Good luck…

References

[1] Project-based Test Automation, Testing Experience, June 2009, www.testingexperience.com

[2] "Automated Functional Testing for Java-Swing; A Pattern of Solution", David Harrison, 2009, ISBN (978-1-4092-9068-1) www.LuLu.com

[3] A Simplified Automation Solution, Using WATIJ, Steven Troy et al, Testing Experience, December 2008, www.testingexperience.com

[4] The Record & Playback Fairytale, Koen Wellens, Testing Experience, December 2008, www.testingexperience.com

[5] Model-based Test Development and Automation, Claus Gittinger, Testing Experience, December 2008, www.testingexperience.com

```
#############################################################################
# Class: NRHelloWorld
# Author : D.Harrison
# Purpose: This class is the name resolver for the HelloWorld application
#
#############################################################################
class HelloWorldResolver(NameResolver):
    #constructor
    def __init__(self):
        self.data = []

#############################################################################
# Method: getName
# Author: David Harrison
# Purpose: overwrites the built-in Name Resolver.
#          If we return a non-None name from here, then the built-in Name Resolver will preserve the assigned name (but replacing WS).
# Parameter:
#       com ... the component to be examined and possibly named
# Returns
#    a replacement name
#############################################################################

    def getName(self,com):
        try:

            if not ResolverRegistry.getElementName( com ):
                #  ---------------------------------------------------------- Application Specic classes
                if ResolverRegistry.isInstance(com, "idx.loginapp.ExampleApplication" ):
                    return "APPLICATION_FRAME"

                elif ResolverRegistry.isInstance(com, "idx.gui.text.AntialiasedLabel"):
                    s = com.getText()
                    if  s.strip().startswith( "Hello " ) == 1:
                        if  s.strip().endswith( "!" ) == 1:
                            return "Hello_PLANET"
                #  ---------------------------------------------------------- Base Java-Swing classes
```

Figure 11 – Hello World! Name Resolver

[6] Robot Framework for Test Automation, Marcin Michalak et al, Testing Experience, December 2008, www.testingexperience.com

[7] Boon and Bane of GUI Test Automation, Mark Michaelis, Testing Experience, December 2008, www.testingexperience.com

[8] QFTest Documentation

Biography

David Harrison works as an independent software QA/Test Manager – currently at SwissRe, Zurich, Switzerland within the tools development group. This group has the mandate to develop and deploy globally reinsurance costing tools to the actuary and underwriter desktop. David can be contacted via email at dharrison_ch@yahoo.co.uk, or via his web site www.dexters-defect-dungeon.com

This article is based on material from his book: “Automated Software Testing for Java-Swing; A Pattern of Solution”, 2009, ISBN (978-1-4092-9068-1) www.LuLu.com