Data Driven Test Automation

Craig McKenzie

June 13, 2008

Abstract

Data driven test automation is considered to be one of the best automated test implementation or framework types due to its extensibility and relatively low maintenance requirements. But what exactly does data driven mean? Among the many different implementation strategies that result in a data driven script, can we identify one that is truly data driven?

1 Introduction

There are a variety of implementation strategies for creating a data driven automated functional test suite. It is not possible to list all potential implementation options so in this reference, we'll create a bridge from the simplest form of script through some of the more complex implementation options.

To make sense of the options, a simple sample application will be used as the base against which to establish the various scripting options.

2 Sample Application

The token application used in this exercise is as illustrated in Figure 1. It represents a general web-based account transfer application.

Figure 1: Transfer Application

The basic steps in this application are:

1. Select account from which the money will be transferred
2. Select account to which the money will be transferred
3. Enter the amount
4. Click on the button

3 Data Driven Implementation Techniques

3.1 Non-Data Driven

To establish a basis for comparison, it is worthwhile spending a little time on looking at the most basic of all automated scripting techniques. While the technique need not necessarily be a *recorded* script it, ultimately shares a number of its characteristics.

Regardless of how the script is generated, it is likely to have the form as illustrated in Figure 2.

```
list("FromAccount").Select "12345678"
list("ToAccount").Select "87654321"
edit("Amount").Set "120.00"
button("Transfer").Click
```

Figure 2: Generic Script Form

The things to note about this code are:

- The data is within the code
- In order to run multiple data cases, multiple elements of code will have to be created
- The object identifiers are internal to the code
- The map between object and data is the code

3.2 List Based Data

It is fairly simple to enable a script to execute through multiple code instances. All one needs is a construct that will generate a list or array of data values and then a loop to iterate through them.

Given the above basic description, the above code (see Figure 2) can be changed to the form as illustrated in Figure 3.

```
fromList = ("123456789", "987654321")
toList = ("987654321", "123456789")
amountList = ("120.00", "240.00")
for i = 1 to length(fromList)
    list("From Account").Select fromList(i)
    list("To Account").Select toList(i)
    edit("Amount").Set amountList(i)
    button("Transfer").Click
```

Figure 3: List Data Script Form

While we have eliminated the problem of having to rescript for individual data instances, there are still things to note about this code:

- The data is within the code
- The object identifiers are internal to the code
- The mapping between object and data is the code

3.3 File Based Data

Removing the data from the script has the advantage of making the script maintenance easier and safer to perform. Moving data outside of the script is fairly simple depending on the file capabilities of

From_Account	To_Account	Amount
123456789	987654321	120.00
987654321	123456789	240.00

Table 1: File-Based Data

```
objectList = datTable.GetParameters(inputData)
for i = 1 to length(objectList)
    list("From Account").Select datTable(objectList(i), inputData)
    list("To Account").Select datTable(objectList(i), inputData)
    edit("Amount").Set datTable(objectList(i), inputData)
    button("Transfer").Click
```

Figure 4: File Data Script Form

the test tool. In general some form of spreadsheet or delimited file is supported. Using a spreadsheet file format as an example, the data table as illustrated in Table 1 can be derived.

Using a file changes the script code to something along the lines of the listing in Figure 4.

While we have further eliminated the problem of data being within the code, there are still things to note about this code:

- The object identifiers are internal to the code
- The mapping between the object and data is the code

3.4 Scripted Data Driven

In order to remove the dependency between the code and the object a few things need to be considered about the test automation tool.

The most important consideration is the mechanism used by the tool to identify the objects within the application. What is being sought is a single reference key that identifies the object. If the tool does not inherently support a single reference, then there may well be some means of manually creating this form of reference through, for example, a data map.

A solution is also needed for identifying the type of object. This is necessary as tools generally use separate functions for setting values in the different types of fields (list fields, edit fields, check boxes, etc.) If there is no function for identifying the object type from its reference within the tool, then it should be possible to use an object naming scheme on the reference key to simulate this. In this last instance, for example, it would entail using something like “ed” and “ls” prepended to the reference in order to identify an edit and list respectively.

The final step is to create a mapping between the data and the object. The simplest, and a highly efficient means of doing this, is to ensure that the data file parameters are the object reference keys.

Armed with solutions to the aspects listed above, a function needs to be developed which will be able to capture the data values irrespective of the field type. The function will have to handle the following steps and could be something like the code sample in Figure 5:

1. Identify the object’s type
2. Perform the tool specific action based upon the object’s type and use the data value

If we assume that the above function is called *processDataValue*, then the script is reduced to the form as in the listing in Figure 6.

There are some things to note about this code:

- The object identifiers are outside of the code
- The objects and data are mapped by the data parameters
- The object types are considered to be part of the data and not a fundamental property of the code

```
function processDataValue( objID, dataValue )
    retVal = "ok"
    select case objID.type()
        case "edit"
            objID.set dataValue
        case "list"
            objID.select dataValue
        case "check"
            objID.set dataValue
        default
            retVal = "error"
    return retVal
```

Figure 5: Data Processing Function

```
objectList = datTable.GetParameters(inputData)
for i = 1 to length(objectList)
    processDataValue(objectList(i), datTable(objectList(i), inputData)
button("Transfer").Click
```

Figure 6: Script-Based Data Driven Form

4 Summary

The complexity of the illustrated options increases as the layers of abstraction are added. The benefits of the added complexity are:

1. Removing the data from the script means that it can be maintained independently of the script code
2. Mapping the data and the objects together removes a step in maintenance. It is only necessary to maintain the data and the object reference as the code is not included in the map
3. The physical code from the script is reusable for nearly all applications on the same platform
4. The code base can be easily extended to cover all application platforms

Increased complexity has a habit of introducing additional problems. The more abstraction added to further the ability of the data to drive the script, includes the following challenges:

1. The automation scripts don't immediately follow the manual process which means that its more difficult to follow the scripts
2. Debugging may be more difficult but this can be handled through thorough error management