BIO

PAPER

T7

November 7, 2002
11:30 AM

KEYWORD TESTING AT THE OBJECT LEVEL

Brian Qualters
TurboTesting Concepts

International Conference On
Software Testing Analysis & Review
November 4-8, 2002
Anaheim, CA USA

Brian Qualters

Brian Qualters has been certified as a Mercury Interactive Product Instructor and Specialist since 1999 (versions 5 thru 7). After starting my own consulting company, TurboTesting Concepts in 2000, I have travelled extensively for MI performing training classes and consulting at various companies including Boeing and AT&T. TurboTesting Concepts developed it's own application called TurboTester ™. Used in combination with WinRunner, TurboTester allows for simultaneous development of manual and automated scripts using the data driven engine approach.

Name: Brian Qualters
Company: TurboTesting Concepts

Title: Keyword Testing at the Object Level
Time: Thursday, Nov 7, 11:30 a.m.

Track: Test Automation

<u>Presentation Description:</u>

It's time to put a new spin on the technique of keyword testing using a data-driven engine. Brian Qualters shows you how to effectively place your focus not on the action or process to be completed, but rather on the object type that's to be manipulated. This redirected focus lets you avoid the pitfalls and resource requirements encountered when you move to test another application. He gives a demonstration of how this modified approach can be integrated into the manual test case creation as a way to tremendously improve efficiency.

- Learn to avoid some of the maintenance issues of automation
- Make scripts easily maintainable
- Keep the process unobtrusive to the manual test effort

This paper looks to build on what is already out there in terms of information regarding the keyword driven method and attempts to provide more of the 'how-to' information than the 'why' information found in many papers. I would highly recommend reading the white paper by Keith Zambelich of Automated Testing Specialists, Inc. entitled 'Totally Data-Driven Automated Testing - A White Paper' to get an understanding of the various approaches to automated testing. (http://www.sqa-test.com/w_paper1.html) I had not run across this article until after I created my tool, but found it very informative.
For further assistance in creating your own or employing the TurboTester engine, contact Brian Qualters at bqualter@turbotester.com or call 336-292-9838.

Taking Advantage of Object Oriented Programming in Test Automation Efforts

When I started automating five years ago, test automation was simply seen as an extension of the manual testing process. A written document was created defining what and how to test for a particular objective, and the automation team would take the document and create an automated script to accomplish the actions defined in the document. Usually the process involved recording the actions as defined by the document, then adding synchronization and verification code to ensure proper playback and a valid evaluation of the test execution. This one-to-one relationship between the manual 'business process' test and automated test was the direct result of a failure to recognize that automation is not just an extension of the manual test process and a failure to recognize the power of the automation languages.

After several regression cycles, we realized that we often spent as much if not more time maintaining the "suite" of automated scripts as we did researching and creating new tests to evaluate the new code for the latest release. Automation itself became a development process. And unfortunately we were not following a planned out process. Recognizing this, we moved to what we hoped would be a more efficient method to manage changes in the application. Scrapping the old method where individual business processes had individual scripts, we embarked on creating a more modular approach where a script was created for every screen within the application. Built into each script was a checkpoint and synchronization statements. Also, all data values were parameterized and a reference was made to an external data file. Logic was built in that checked for a data value for each object and if a data value did not exist then the object would be skipped.

Individual test case scenarios were then created by calling the required screen scripts and sending data files to those scripts. This approach took away some of the maintenance of individual test cases by placing the emphasis on the screen scripts. If something about a screen changed from build to build we would update the script thereby automatically fixing all test case scenarios touching that screen. The maintenance process was streamlined, as instead of having to identify every business process that was affected by AUT changes, it was now a matter of relating screen change information to screen scripts. Data files also were changed when needed to reflect screen changes.

Modularization had come into our process and it was then a quick leap from screen based scripts to object based scripts afterward. This modularization can be looked at as the second phase of automation. Along with it came the recognition that automation itself is a development process that needs to be managed and planned to fully take advantage of the power of the tools.

Automated testing has quickly entered the world of “third-generation” scripting, where data driven engines power the test building process. Regardless of the vendor or tool utilized, utilizing a data driven engine, where specially formatted external data files, passed to a single set of automation code for processing, determine the individual business process actions to take place within a test script, has become the preferred methodology to deploy by the automation test team. The leading automation tool vendors currently do not provide the engines themselves, preferring instead to provide tools that allow users to take their testing in any direction they wish. There are however, a few companies and individuals that have begun to offer their own versions of “third-generation” automated scripting tools, with TurboTesting Concepts TurboTester™ among them.

The approaches taken in providing this “third-generation” tool vary with each offering, but many of the current offerings differ in their approach in terms of focus level. The majority of offerings I’ve seen follow a methodology where the business processes of the individual application(s) under test determine the construct of the engine. This paper will attempt to bring out the need to shift this focus from the individual application under test (AUT) to the very basic building blocks of the application, the objects, when creating your own data driven engine.

There are many reasons to place the focus at the object level instead of the process level, the primary reasons however will always remain portability and ease of maintenance. The incorporation of an automated testing tool into the testing process requires the recognition that automation in and of itself is a development process. Much as application developers need well defined requirements to establish the framework and logic of the application under development, so does the automation process require defined requirements to build a viable data driven engine.

However, the main information required is that which relates to the framework of the application. For example, how will users navigate from window to window? Once in a window how will the user interact to extract and provide information? What technique will be used to update, save and delete information? Application logic, in terms of what will trigger events and what will those events do, is not needed for the data driven engine as that information is wholly testable by the individual test cases created to exercise the business logic. It is this separation of logic from objects, whether at the window or standard/custom object level that allows the automation effort to create an engine that can be portable and expandable across applications.

Today’s development environment, regardless of whether in a web or gui based client/server scenario, is still mainly a reflection of Object Oriented Programming techniques. At a basic level, regardless of whether your tool is built using Visual Basic, Visual C++, JavaScript, HTML or what have you, the development process usually revolves around three stages. First there is the stage of requirements gathering where information relating to purpose, look and feel and process/logic are defined. This is the launch pad for the building phase of the application under development. Screens are created using development tool specific objects and custom objects where necessary to allow for interactivity with the user. Finally the logic is employed, relating object interaction with the user to event triggers that will ideally cause requested business processes to occur.

It is the second phase, the building phase of object oriented programming that should be concentrated on by the automation test team to allow for a viable data driven engine to be created. Once an understanding of the objects and navigation items available to the user is attained, it is then a matter of mapping the functionalities or properties of these objects to the engine. For example, data entry operations vary based on the class of object being interacted with. Combo boxes, regardless of the development tool that created it, allow the user to select a value from a self contained list or directly edit into the field a unique value if desired. Edit fields require that users enter values directly into the field. Navigation is often made through the selection of menu items. The point of all of this is that specific object classes, combo boxes, edit fields, lists, menu items etc, exercise the same functionality whenever called. The uniqueness of the individual action is found in the data being used.

Let's look at Mercury Interactive TSL language used within WinRunner to bring this point out further. As with other automation tools, the functions used within WinRunner are reliant on the relationship between the GUI map, the code and the application under test (AUT) internals. Ideally, the user will 'learn', or record information regarding objects on each window in the AUT prior to creating a script that executes any actions against these objects/windows. Once learned and saved into a GUI map file, the information regarding the objects manipulated in the script is referenced during playback to allow WinRunner to locate and operate on the application at run time.

The process of 'learning' objects informs WinRunner to store physical attribute information along with a logical name for reference in the script coding. WinRunner only stores as much information as is needed to uniquely identify each object encountered, with a hierarchy of attributes to record based on a configuration file. The one common attribute that is always recorded, or learned, regardless of object type, is the object class. This object class is WinRunner's tip off as to which functions to use against the object when manipulating or checking it. For example, list items use the **list_select_item** function to pick a value from a list; edit fields utilize the **edit_set** function to activate and enter data in an edit field.

Each function for data manipulation allows the user to define the logical object name and the value to enter in the field as parameters. It is here that we can take advantage of object oriented programming, as regardless of which test scenario we are trying to execute the operation is taking place at the object level. Sequences change, windows change and objects change from test to test. There are even situations where only the data used changes, creating a whole new test scenario. The constant throughout all of this though is that the object classes within the application do not change and therefore the functions used to manipulate them do not change.

With this knowledge in hand, it is now possible to create a data driven engine using what I call Object Oriented Scripting. Remember, developers employ Object Oriented Programming techniques that also relate to the object level and power applications through the event based logic. It is much more difficult to create the application because of this requirement to understand what the logic is behind each event requirement. We do not want to recreate the application using test scripting languages, we merely want to create the ability to exercise the application internals and check expected results based on test case definitions. This is where most people who start out to create an engine get tripped up as they can not let go of what the application is designed to do. Separating application functionality from application design is difficult but necessary and is step one in creating a reusable engine.

To illuminate this further, imagine two separate applications; one a Visual Basic application designed to provide financial information and a Visual C++ application designed for human resource management purposes. Each application has a different purpose and was designed with a different development tool. The object oriented programming nature of the development tools gives them similar design capabilities. In the VB application for example, a textbox object may be used to gather information from the user, whereas in the Visual C++ application an edit object is used to do the same job. Once configured in the automation tool (I will always refer to WinRunner methodology in this paper, but the concept is universal) as a standard edit class object, these varied objects will have access to the same set of functions. Which specific functions you use within that set will be determined by the action to be accomplished.

By variablizing the parameters, any edit object can reference the individual code for each action, thereby taking advantage of the ‘script-once’ object oriented scripting concept to optimize consistency and ease of maintenance. This concept basically dictates that by scripting code at the object class level, independence of the application processes is achieved. Creating code that performs specific actions like data entry, data verification, and menu navigation with the flexibility to utilize the same code against any specific object that belongs to the object class negates the need to change code once the engine has been stabilized. Stabilization will occur after all objects used in the AUT have been integrated into all action cases.

To properly utilize the code you must ensure that synchronization and verification statements are built in to actions. Also, it is suggested that you build in code that dynamically recognizes the object class type being referenced. Once this is known you can use the class type as the switch value that determines which case to use within the action case construct. This allows your action cases to logically be broken up based on class names. The action will serve as the keyword in your data driven engine, but the objects will drive the action capabilities.

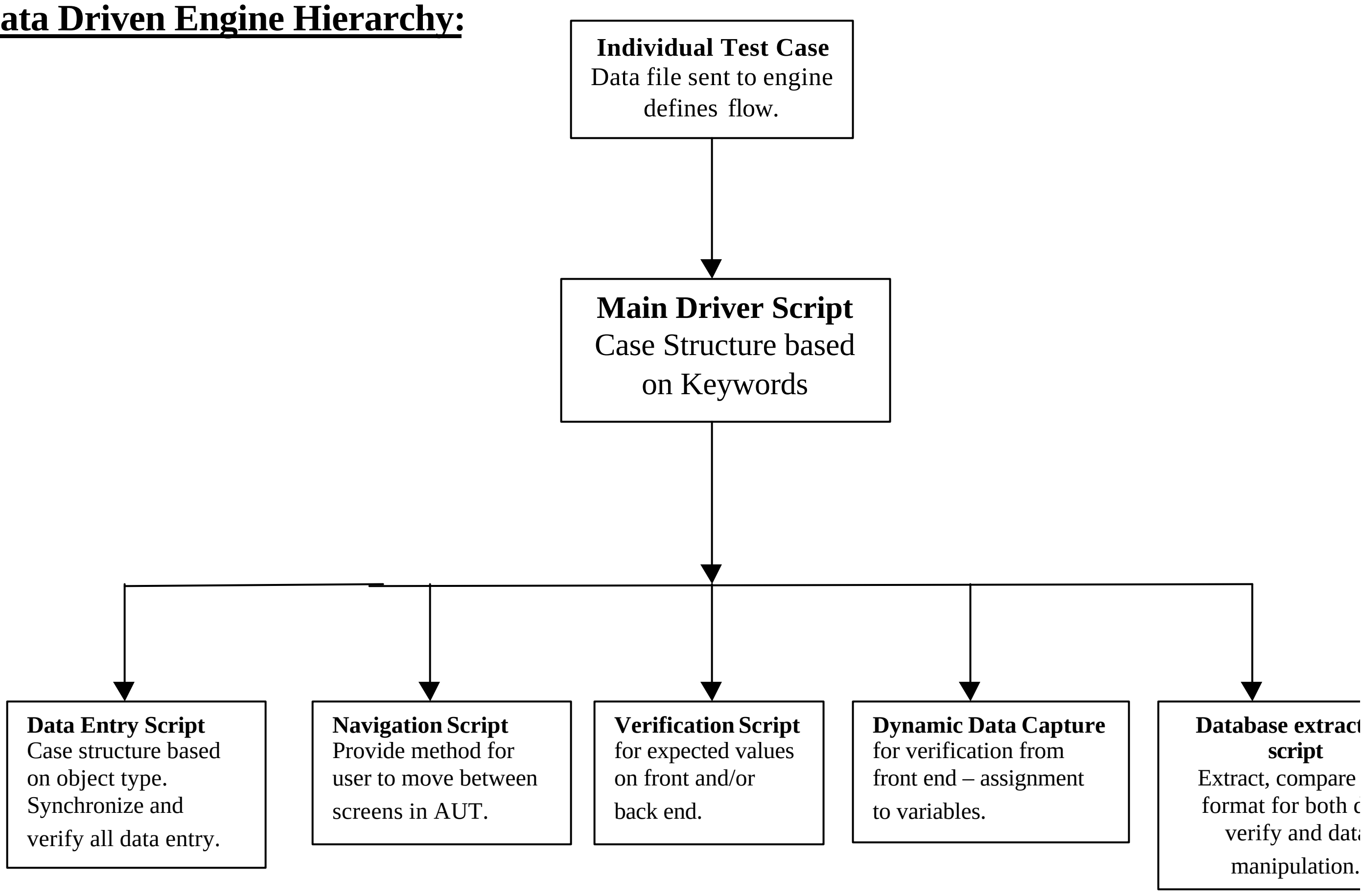
Let's take a look at the three tiered keyword driven engine automation approach. Keep in mind that every automation tool offers its own unique capabilities and therefore may guide you to massage the hierarchy to be explained, but also remember that because every tool must make a connection or recognition of application objects the blueprint is pretty static.

The main driver script is the organization or the "flow management" script. This script should be created after first reviewing what your application does and what your test plan dictates that you test. This is often referred to as the "Test Plan Method" where the test plan dictates the requirements for actions needed during testing. For example, UNIX applications are command prompt based, where actions taken are defined by function statements inserted at the command prompt. The commands themselves are testable by checking the result of each command. A basic example would be a copy command entered at the prompt to create a copy of a directory in a new location in the file structure. Data entry is not accomplished through object manipulation but through command line manipulation. In this situation you may create a case structure with keywords such as 'Enter Command', 'Check Outfile', 'Check Screen Output' etc.

Applications where the user interface is made up of context sensitive objects that require user interactivity, regardless of the end goal of the interaction, will require a different keyword structure based on the interaction goals. Some examples of keywords to use in the case structure include 'Data Entry', 'Capture', 'Check Calculation', 'Navigate', 'Check Object Attributes', 'Verify Data' etc. Other keyword cases may need to be created based on the automation tool in place, for example 'Load GUI' and 'Unload GUI', 'GUI Check', 'Bitmap Check', etc. It is the recognition of what your application does and what your automation tool is capable of that will lead to the overall case structure. The one thing that you will notice for object based applications is that by building the action cases based on the various objects employed in the application, you allow for easy expansion when new object types are employed in other applications to be tested that activate the action capabilities.

The individual test case script in the diagram merely serves as a holding place to define the data file to be used to exercise a specific functionality. The data file will refer to the keyword action to process for each line of data. Where needed, the main driver script will direct control to the appropriate subscript to execute the action. Once completed, control will return to the main driver script which will iterate through to the next line of data for processing until there is no more data in the data file.

The Data Driven Engine Hierarchy:



Individual Test Case
Data file sent to engine defines flow.
Main Driver Script
Case Structure based on Keywords
Data Entry Script
Case structure based on object type. Synchronize and verify all data entry.
Navigation Script
Provide method for user to move between screens in AUT.
Verification Script
for expected values on front and/or back end.
Dynamic Data Capture
for verification from front end – assignment to variables.
Database extract script
Extract, compare format for both c verify and data manipulation.

Below is a snippet from a data file that can be processed through the TurboTester™ data driven engine from TurboTesting Concepts (www.turbotester.com). Notice that the first column contains the keyword that defines the action or case to execute within the test case. By defining the order of actions to take you define the test case scenario itself. Notice also the flexibility of the actions that can be taken, for example, running an SQL statement, scrolling the Vertical scrollbar on the Leave of Absence Z screen down three positions, turning an exception handler off, etc. These action cases were developed based on the information provided by looking through applications to see what actions are performed.

ntry	Leave of Absence Z	From Date	<sysdate>	Num of Days	5	
et New Date	<sysdate>	5				
et Info	Leave of Absence Z	Consecutive Days	enabled	ON	Eff Date	displayed
un SQL	GrievanceCheck	shared QRY folder				
croll	Leave of Absence Z	Inside ScrollBar	Vertical	-3		
xceptions	MenuDefinitionManagerGet	OFF				
apture	Compensation	Change Amount	100	Change Perce	15	
erify	Close a Grievance	A	Same	Priority Item?	Date Close	Descriptior
se Function	button_get_value(Effective Date(B)-enabled)					
eyboard	Compensation	Backspace	Delete	Insert		

More importantly though, notice the Entry and Capture lines. These action cases allow the user to perform the integral actions to execute the test case. The Entry case calls the data entry subscript, sets the focus on the Leave of Absence Z screen and proceeds to enter data into the 'From Date' and 'Num of Days' fields. Likewise, the Capture case sets the focus on the Compensation screen and captures the data in the 'Change Amount' and 'Change Percent' fields. To ensure that the correct values exist in those fields, expected values are defined after each field to be captured and a comparison is processed.

Building the subscripts to perform these actions is simply a matter of relating the objects to the appropriate tool functions. Your verification script for example could easily be made of statements that grab or retrieve information from the application at run time in regards to the attributes of the objects in question. In most cases, this involves performing something similar to WinRunner's '_get_info' statements with the attribute retrieved usually being the 'value'. By setting up the script to recognize the object type in question the appropriate '_get_info' statement can be employed to gather the information. For example, if the object being operated on is of class list, then the 'list_get_info' statement would be used to provide greatest reliability. Finally a comparison or conditional statement could be built in that checks to ensure that the retrieved value equals the expected value sent in the data file. If a negative case is being executed the logic could easily be reversed. Again, the point of all of this is that the code for each object information retrieval could be separated by the object type in a case structure within the Verification action script. This is the same formula to follow for all of the subscripts you will need to create as noted above in the hierarchy diagram.

<u>Guidelines to follow when creating your own data driven engine:</u>

1. Explore the application under test to uncover the object types employed. Reference any available design documentation that explains user interface standards and integration with the database structure.
2. Explore custom objects and Active X controls to allow for creation of custom functions and where possible reconfiguration to standard object types to gain access to existing functionality supplied by the automation tool.
3. Explore methods of database interaction. How do you extract, update, delete and/or insert information?
4. What languages can you use for dynamic data extraction and creation of external files?
5. Determine the most reliable way to navigate around through applications. Is there menu based navigation, command prompt navigation, link navigation?
6. Allow for negative and positive test conditions by creating variable that is defined at beginning of test.
7. Use case structure throughout for all actions and then subsequent subscripts that perform those actions. Concentrate on the object as the case switch value whenever possible.
8. Develop method to allow for iteration of individual test cases and suites of cases.