Enterprise Test Engine Suite Technology (E-TEST)

Table Driven Automation

James Stephen Schaefer, CCP
Automation Test Manager

Capital One Services, Incorporated
Attn: 12019-0215
4820 Lakebrook Drive
Glen Allen, VA 23060

jim.schaefer@capitalone.com

Abstract

Many companies invest heavily in test automation in order to verify the functionality of their complex client/server and web applications, only to find that anticipated cost savings and higher reliability remain illusively out of reach.

This paper is a guide on how to create Table Driven Test automation with off-the-shelf utilities and commercially available GUI testing tools. It demonstrates the benefits of using a table driven approach and presents various engines, utilities and documents that enhance or support this third generation testing architecture, which I call Enterprise Test Engine Suite Technology (E-TEST).

1. Overview

The off the shelf software used to develop E-TEST consists of Mercury's WinRunner GUI testing tool, Microsoft's Access DB, and Sun's Java language.

WinRunner was chosen because it contains a robust 'C' like scripting language (TSL), which allows us to do more than record/playback automation. It also supports Structure Query Language (SQL) which is essential to implementing any robust table driven architecture.

After defining the tables E-TEST uses we will look at how these tables can be automatically generated from GUI maps and see how the GUI Engine can reformat tables when GUI maps change without destroying existing test data that has already been entered.

In order to isolate object names E-TEST produces three tables for every application page. This feature of the architecture can produce hundreds of tables associated with the Application Under Test (AUT) and makes it almost impossible to manually enter test data into the database without a Data Interface. To alleviate this problem you will be shown a Data Interface that simplifies data entry by presenting test case centric views of appropriate tables to testers using E-TEST.

Finally, we will discuss other engines and utilities that support this testing methodology and some of its advantages and limitations.

2. The Application Under Test (AUT)

In order to do any testing, albeit manual or automated, we need an application to serve as the Application Under Test (AUT). Luckily, WinRunner comes complete with just such an application, which is modeled after a flight reservation system.

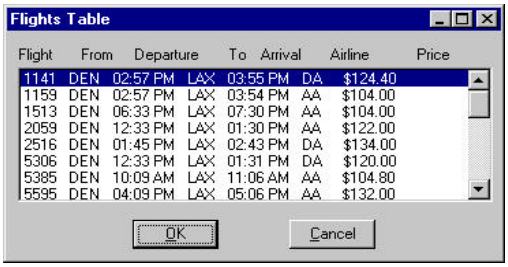
When the Flight Reservation application is launched the first page that appears is the Login window, shown in Figure 2.1. This window has two edit fields and three push buttons.

Figure 2.1 – Login Window

Figure 2.2 shows the main page of this application, which is sprinkled with many different types of window objects (e.g. edit, list boxes, push buttons, etc.). Looking at the figure you can see that "Date of Flight", "Fly From", and "Fly To" input objects have been populated with some initial data. When these three fields are populated and the "Flights" button is clicked a second page titled "Flights Table" (See Figure 2.3) is launched which allows the user to view and select an available flight.

Figure 2.2 – Flight Reservation

Figure 2.3 – Flights Table



This application is included with WinRunner and can be found in its default location: C:\Program Files\Mercury Interactive\WinRunner\samples\flight\app\flight1b.exe.

2.1 Launching Different Types of Applications

Because E-TEST was written to support many different types of automated testing it uses specific key words to launch types of applications. Since the Flight Reservation program is a Windows based Executable we need to store the "CLIENT" key word along with its Full Path Name in the proper table inside the MasterDB (Section 4.3).

In this case: CLIENT("C:\ProgramFiles\MercuryInteractive\WinRunner\samples\flight\app\flight1b.exe")

If we were testing a web based application we could use a browser specific key word with the Universal Resource Locator (URL).

Browser Based:
IE("http://www.capitalone.com")

Netscape("http://www.capitalone.com")

These formats are used to launch the Application Under Test each time a test case is run and are stored in the PAGE_FLOW table, which will be discussed later and are provided here to introduce some of the versatility incorporated in E-TEST.

3. WinRunner Fundamentals

Now that we have an application to test it's time to introduce the WinRunner automation tool. Figure 3.1 shows the tool's main window and Figure 3.2 shows the GUI Map Editor, which is used to map each Window and Window Object contained in the application.

Figure 3.1 – WinRunner Tool

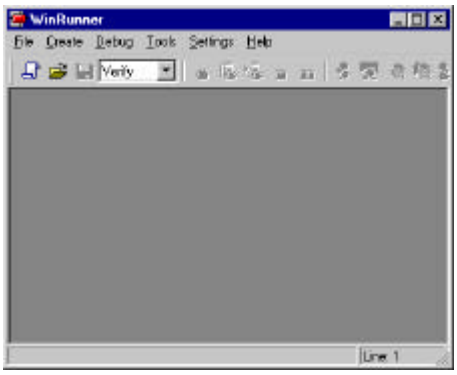


Figure 3.2 – Mapping the AUT

3.1 GUI Map Editor

The GUI Map Editor is selectable from the main menu under Tools and is shown with the objects that were learned from the main page of the Flight Reservation application (See Figure 2.1). WinRunner uses the Physical Description to identify each object, while allowing users to create their own Logical Name for that object. These Logical Names are used as variable names in the TSL scripts created with WinRunner.

Figure 3.2 shows the Physical Description of the Flight Reservation Window, which is displayed when its Logical Name ("Flight Reservation") is highlighted. In this case it shows that the "Flight Reservations" class is a Window and its label is "Flight Reservation". The "Date of Flight" is an edit object, "FLIGHT" is a button object, and "Fly From" and "Fly To" are both in a list objects. This should give you an idea of how WinRunner keeps track of the windows and objects that make up the AUT.

3.2 GUI File Format

Figure 3.2.1 below, shows the simple file structure that WinRunner uses to store each GUI map. This structure consists of the Logical Name followed by the Physical Description, which is enclosed in braces. Each class of objects has a default set of physical attributes that the GUI Map Editor learns.

The first line of each GUI map contains the window name, in this case, "Flight Reservation". Window names and objects will have quotation marks around them if they contain more than one word. A colon denotes the end of the Logical Window Name.

The Physical description follows the Logical Name Line and contains Class Type and other class specific information. Class specific information can vary from one class to another, but remains consistent within a class.

Section 6.1 describes how the GUI Engine uses GUI map structure to automatically create the Page Abbreviation, Page Flow, Test Data, Seized Data, and “Action” Objects Data tables in the MasterDB.

Figure 3.2.1 – GUI Map File Format

4. Tables

The tables used by the E-TEST architecture consists of six basic table types: Functions, Page Abbreviation, Page Flow, Test Data, Seized Data, and “Action” Object Data tables. With the exception of the Functions table, which is delivered with the MasterDB, all application tables are generated automatically by the GUI Engine and are listed in increasing order of complexity.

4.1 Functions Table

The Functions Table (Figure 4.1.1) is created by the Automation Team, which consists of a small team of highly trained technicians that are responsible for creating and maintaining E-TEST. It consists of a single column of Function Prototypes and is the only table delivered with the MasterDB.

This table is used to populate the PRE_ACTION and POST_ACTION sections of the Data Interface (Figures 5.2.1 and 5.2.3) which allows E-TEST users to optionally include predefined functions in each step of a test case.

Figure 4.1.1 – Functions Table

FUNCTION_PROTOTYPE
Print_Screen()
Seize_Data()
Verify_Text(Page,Row,Col,Text)

4.2 Page Abbreviation Table (One per AUT)

The Page Abbreviation table (Figure 4.2.1) contains two columns FULL_NAME and ABBR_NAME. The abbreviated name is used to populate the Available Pages section of the Data Interface (See Section 5.1). This reduces the space needed to display the page names and makes setting up the Page Flow sequences easier.

The GUI Engine generates each page abbreviation from the capital letters and numbers contained in its Logical Window Name as defined in the WinRunner GUI Map Editor. Before entering the generated full

and abbreviated information of a page into this table the GUI Engine checks for duplicates. If a duplicate abbreviation is found, a number is added to the Logical Window Name of that GUI map and its abbreviation prior to its insertion into the table. This insures uniqueness of abbreviated names.

Figure 4.2 – PAGE_ABBR Table

ABBR_NAME	FULL_NAME
LW	Login Window
FR	Flight Reservation
FT	Flights Table

4.3 Page Flow Table (One per AUT)

Next, the GUI Engine constructs the PAGE_FLOW Table (Figures 4.3.1, 4.3.2, and 4.3.3) from variables stored in a Constants script. The Alpha Section of this table (Figure 4.3.1) provides a means for Mercury's TestDirector (TD_ID) to control test case execution via the Test Engine. FLOW_ID is equivalent to the test case identifier. RUN_COUNT controls the number of times the test case will be run and is useful when a test case must be run multiple times to setup a test condition (e.g. Login Lockout) or continually run a single test case (e.g. Heartbeat, Timing Log). The FLOW_DESC column holds a brief freeform text description of the test case. The LAUNCH_APPL column holds the function prototype that launches the AUT that was mentioned in Section 2.1.

Figure 4.3.1 – Alpha Section of the PAGE_FLOW Table

TD_ID	FLOW_ID	RUN_COUNT	FLOW_DESC	LAUNCH_APPL

The Page Sequence Section of this table (Figure 4.3.2) currently has the capacity to store up to twelve Page Abbreviations that makeup the sequence of pages associated with a test case.

Figure 4.3.2 – Page Sequence Section of the PAGE_FLOW Table

PAGE_1	PAGE_2	PAGE_3	...	PAGE_10	PAGE_11	PAGE_12
			...			

The final section of the PAGE_FLOW table is known as the Omega Section (Figure 4.3.3). Although this section is included in the PAGE_FLOW, Test Data, Seized Data, and "Action" Object Data tables – it is only used in the Test Data tables. The "Action" objects are selected classes (e.g. button, html_text_link, etc.) that initiate a jump to the next page in a test case. Jumps to same page are allowed, as when incorrect data results in the same page being redisplayed with highlighted error message(s). The Action Section of the PAGE_FLOW table is not used at this time.

Figure 4.3.3 – Action Section of the PAGE_FLOW Table

PRE_ACTION	ACTION	POST_ACTION
N/A	N/A	N/A

Once the PAGE_ABBR and PAGE_FLOW tables are created the GUI Engine turns to its final task, that of creating a Test Data, Seized Data, and Object Data table for each GUI map found in the GUI directory.

Unique table names are created by prefixing the page ABBR_NAME with a "type tag" (TD_, SD_, and OD_). Each of these tables has identical Alpha and Omega sections, but differ in the data that is contained between these sections in the center.

4.4 Test Data (TD_*ABBR*) Tables (One for each Page in AUT)

Looking at Figure 4.4.1, we see an Alpha section of what could be a Test Data, Seized Data, or Object Data table, since all are identical. Each of these tables also begins with a TD_ID column, which is not used at this time.

Figure 4.4.1 – Alpha Section of all TD_ Tables

TD_ID	FLOW_ID	PAGE_ID	PAGE_ABBR	TimeStart	TimeStop	VerifyObject

Unique combinations of **FLOW_ID** and **PAGE_ID** in these tables match FLOW_ID and Page Sequence (PAGE_1-PAGE_12) columns in the PAGE_FLOW table and identify which row of data is to be used when populating data or initiating an action. Since we are discussing the Test Data tables here, we are interested in finding the correct row of data that will be used to populate a specific page in a test case.

Once the proper row is identified the **PAGE_ABBR**, **TimeStart**, and **VerifyObject** columns are updated with Page Abbreviation, Current Time, and Full_Page Name data.

TimeStop is populated immediately after the page "Action" is initiated. Recording these times allows us to calculate the amount of time it takes to populate a page (input time) and how long it takes to get from one page to the next (transition time). Of the two, transition time is the most important and primarily to report web based applications response speeds. It can also provide feedback when tweaking backend systems or monitoring a heartbeat transaction.

The Data Input Section of the Flight Reservation Test Data (TD_FR) table (Figure 4.4.2) contains objects used for data input. The specific column names contained in this section of the Test Data Tables match the logical object names defined in each GUI map. Thus the number of columns in this section varies. In application page tables without any user-input object this section will not exist.

Figure 4.4.2 – Data Input Section of the TD_FR Test Data Table

Date of Flight	Fly From	Fly To
10/31/01	Denver	Los Angeles

The Omega section of the TD_ Tables (Figure 4.4.3) stores optional Pre-action and Post-action Function prototypes that can be used to perform verification (e.g. string comparison) and/or initiate tasks (e.g. printing). These function prototypes are available in the Data Interface from pull down lists, which are populated from the Functions table.

The Action column stores "Action" Data Objects that take us to the next test case page. These Data Objects are stored in the Object Data tables, in this case the OD_FR table, and are selectable in the Data Interface via a pull down list (Figure 5.2.2).

Figure 4.4.3 – Omega Section of the TD_FR Test Data Table

PRE ACTION	ACTION	POST ACTION
Print_Page()	Flights	

4.5 Seized Data (SD_*ABBR*) Tables (One for each Page in AUT)

The Seized Data table layouts are identical to their corresponding Test Data tables. The difference is in how they are used. Storing an "X" in a column of the Seized Data table tells the Test Engine you want to know what data was in that object. This allows us to verify that data entered in one page matches that data when it is displayed on another page. Figure 4.5.1 show the Flight Reservation's Seized Data table setup to record the Fly From city. When the Test Engine completes this step of the test case the Fly From column will contain the data entered from the Test Data table. In this case the "X" would be replaced with "Denver".

Figure 4.5.1 – Seized Data Section of the SD_FR

Date of Flight	Fly From	Fly To
	X	

4.6 Action Object Data (OD_*ABBR*) Tables (One for each Page in AUT)

Finally, the GUI Engine constructs the action Object Data table from a specific GUI map. The Alpha and Omega sections of this table are identical to those of the TD_ and SD_ tables, but the center section stores only "Action" words (See Figure 4.6.1)

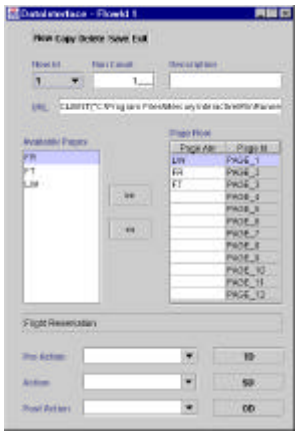
Figure 4.6.1 – OD_FR Table

Flights

5 The Data Interface

Figure 5.1 show the Data Interface, which was built to help alleviate the problems associated with keeping track of all these tables. A test case has been setup that involves three Flight Reservation application pages (Login Window, Flight Reservation, and Flights).

Figure 5.1 – Data Interface



Since this is the first time the Data Interface was run against the new MasterDB no data is stored in the PAGE_FLOW, Tests Data (TD_FR, TD_FT, TD_LW) and Seized Data (SD_FR, SD_FT, SD_LW) tables.

Clicking the "New" button creates a new test case by adding a row to the PAGE_FLOW table and setting the Flow ID to the next available index (in this case 1). Next "Run Count" is set. Usually this is set to one, but it can be set to higher number, which causes the test case to be repeated that many times. This can be handy when locking accounts (e.g. five unsuccessful login attempts, etc.) and reduces setup time.

Next the launch the AUT information is entered (Section 2.1).

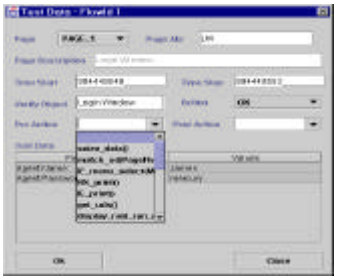
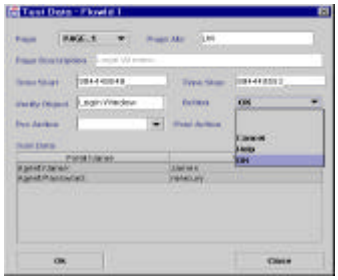
Then the page sequence is setup by copying available page abbreviations over to the Page Flow list. This is done by highlighting the proper rows in the **Available Pages** lists and then clicking the >> button to move then into the **Page Flow** in the desired sequence. The << button is used to remove a page from the sequence.

The **Description** is optional, but is usually filled in with a short description of the test.

Once the **URL** "Application Type("Full Path Executable Name") data is entered we can move on to the Test and Seized Data tables associated with each application page.

Figures 5.2A and 5.2C show a partial list of the functions that are included with the tool. You may remember that the prototypes of these functions are stored in the Functions table in the database. Figure 5.3B lists the Actions available on this page of the application, which are stored in the OD_LW table.

Figure 5.2.1 -TD w/ Pre Action **Figure 5.2.2 – TD w/ Action** **Figure 5.2.3 – TD w/ Post Action**


Remember we are looking at the Test Data table here. Hence the actual Agent Name and Password is displayed under the Values column in the test data area in the lower third of the Data Interface page. Had we been viewing the Seized Data tables the Values would initially contain an "X" where we wanted to retrieve output data. These "X" values change to the actual displayed data values once the test case is run.

Hopefully, this short description of the Data Interface conveys the ease with which test cases can be setup and shows how the number of tables generated in the database is hidden from the user.

6. The Engines

There are three engines in this suite of tools and each was built to handle a specific table related task. They are: the GUI Engine which generates tables from the GUI maps, the Test Engine which executes the tests, and the Data Engine which populates a single table with test data from external sources.

6.1 The GUI Engine

The GUI Engine translates the GUI maps created with the GUI Map Editor into the tables as described in Section 4 above. The first thing it does is build a list of all the window GUI names it finds in the GUI directory.

6.1.1 Creating Empty Tables from A New MasterDB

Assuming this is the first time any tables will be generated, the GUI Engine creates the PAGE_ABBR table. This table initially contains only column headings (e.g. ABBR_NAME and FULL_NAME), but is updated with abbreviation and full name information each time a group of TD, SD, and OD tables are generated from a GUI map.

Next the PAGE_FLOW table is generated. This table is the primary table used to setup and drive test cases through the desired page sequences.

Once PAGE_ABBR and PAGE_FLOW tables have been created, the GUI Engine's final step should begin to generate groups of TD, SD, and OD tables for each GUI map found in the GUI directory. However, before that process can begin the GUI Engine scans each GUI map object to check for special characters that can't be used as column headings in the database. This is not as simple as it may seem, since each logical name in every GUI map must be checked and the map edited to remove any offending characters.

When all the logical names have been checked and edited so they can be used as column headings, the GUI Engine generates the page abbreviation, appending sequential numbers to duplicate names to insure uniqueness, where appropriate.

Finally, TD, SD, and OD group table generation begins with user input objects (e.g. edits, radio buttons, lists, etc.) inserted into the TD and SD tables and 'Action' words (e.g. push buttons, html rectangle, etc.) placed in the OD table.

Now you know how the GUI Engine script creates empty tables from a new MasterDB. But what happens when the tables are full of test data?

6.1.2 Redefining Tables When GUI Maps Change

The method of generating empty tables introduced above works fine, but applications are born to change and changing application pages means that associated GUI maps will also need to change. Since E-TEST wasn't built to map the application pages in the first place, I leave the re-mapping to you, but will offer some mapping tips at the end of this paper.

Just as in the first case, where we were generating empty tables, the GUI Maps are still the key when the PAGE_FLOW, PAGE_ABBR, TD, SD, and OD tables are chuck full of data. The difference now is that the GUI Engine may need to insert or delete tables, with associated changes to the PAGE_ABBR entries, and add and delete column names according to the new GUI map definitions.

The logic that powers this database reconfiguration is shown below in Table 6.1.2.1.

Table 6.1.2.1 – MasterDB Table Reconfiguration Rules

GUI Map	MasterDB Table	Action
Exists	Does Not Exist	Create Table & Add to PAGE_ABBR
Does Not Exist	Exists	Delete Table & Remove From PAGE_ABBR
Exists	Exists	Reformat Tables & Copy Matching Data

While this method of updating database table configurations is not foolproof, it does offer an alternative to creating new empty tables and then moving the data back into the tables manual or via SQL script. The only thing it does guarantee is that test cases composed of unchanged tables will continue to run (if they ran before the database was changed).

6.2 The Test Engine

When the Test Engine runs, it reads down the rows of test cases setup in the PAGE_FLOW table looking for RUN_COUNTS greater than zero. Once the iteration loop is set, the Application is launched. When it becomes stable the Test Engine sets the TimeStart in the TD Table of the active page. Window based programs, like "Flight Reservation", seem to be stable the instant they are launched, but web-based applications can take several seconds to become stable.

Next the Test Engine loops through each page found in the Page Sequence Section of the PAGE_FLOW table - entering data, calling PRE_ACTION functions, initiation ACTIONS, setting TimeStop, and initiating POST_ACTIONS, as defined in each Test Data Table visited. Object classes are derived from the GUI maps. Thus, when the input data is associated with a list the Test Engine calls the appropriate TSL

list_select_item function and when we click on Flights push button it calls button_press function, housed in TSL function scripts in the Function Directory.

6.3 The Data Engine

Up to this point we have been looking at how E-TEST saves time when setting up test cases individually then running them automatically. However, you may want to run one test case repeatedly while varying the test data.

In many cases the input data for this type of Testing comes from production systems. This engine translates single production data fields (e.g. Housing = O, R, or X) to multiple radio buttons (Own, Rent, and Other) within the targeted application page. Another interesting feature of the Data Engine is its ability to replicate a test case while importing data. This frees the user from manually setting up multiple tables before testing can begin.

7. Utilities

Several utilities further enhance E-TEST functionality. Two of these are the Page Flow Logic, Object Coverage, and Data Dictionary Verification scripts. These scripts are used after the tables have been setup to test an application, but before the Test Engine script is run.

7.1 Page Flow Logic Utility

The Page Flow Logic utility is run to verify that uniqueness and correctness of the FLOW_ID and PAGE_ID pointers prior to testing. Running this utility is especially important if anyone has gone into the MasterDB and made changes to the tables without using the Data Interface tool.

7.2 Object Coverage Utility

The Object Coverage utility is used to verify that all the windows, data inputs and action objects defined in the MasterDB tables have been used in at least one test case. This utility generates an Object Coverage Report. While it doesn't provide path coverage, but does report on any defined objects that have not been used.

7.3 Date Dictionary Verification Utility

One problem with any kind of test automation that is used by more than one person is keeping object names standard. An old solution for this problem is to establish a Data Dictionary. The dictionary is a repository for Logical Object Names and their definitions. These could be built into a new table in the MasterDB or in a word processor like MS Word.

If you create a Data Dictionary that has an appendix listing all the objects it contains; it is easy to generate an **Object Discrepancy Report** using the Data Dictionary Verification script. This script matches objects in the MasterDB tables with objects contained in the appendix. Unmatched objects are printed out in a report that lists objects not in the tables and not in the appendix.

8. Advantages of Table Driven Automation

The chief advantage of table driven architecture is that it is built and maintained by a small group of automation experts. In the case of E-TEST these technicians would be TestDirector, WinRunner, SQL, and Java experts. As automation experts they are more focused on building the automation engines and have less involvement with actual testing.

Much of the saving associated with table driven automation results from the fact that one script (Test Engine) is used to conduct many tests on many different applications. The automation technicians create and maintain this script and others that are easily controlled. Thus the functional test team is insulated from

the need to become scripting experts and can focus on translating the requirements and business use cases into test cases setup in E-TEST.

Since the automation team creates and maintains E-TEST they are responsible for creating and maintaining the documentation for the tool and training materials used to educate testers on its use. This shortens class time and reduces training costs. It also insulates the company from maintaining huge numbers of scripts that develop over time and whose initial usefulness begins to fade as staff turnover and parallel development eat into savings.

9. Disadvantages of Table Driven Automation

Since I spent all this time promoting table drive automation, it seems only fair to point out a few of its weaknesses. One of the chief weaknesses of this methodology is the small group of automation experts that created all this wonderful stuff in the first place.

Like most experts they tend to be a little high strung and head strong, feeling they are a step above average and entitled to more freedoms than those who use their creations. Finding these individuals is hard enough, but trying to manage them can be a real wakeup call for micro-managers who are use to telling someone what to do and expecting them to smile and say OK.

Testers who see the automation in action soon get interested and want to start developing some automation of their own. After all how hard could it be? The answer unfortunately is that it is very easy to create 1st generation automation. That's why companies selling the stuff continually trot it out in demos to perspective customers. It's only slightly harder to jump into 2nd generation automation, so the scripts begin to pile up and you start sliding down the slippery slope into reduced cost savings.

The best solution to keeping the automation team small while still giving the functional tests a career path toward automation is to rotate people in and out of both positions. This mixing of roles can actually be beneficial to both teams, as new perspectives generate new ideas that can be incorporated into the 3rd generation Table Driven Tool.

10. Summary

Table Driven Test Automation can be used to improve testing in organizations by reducing the costs associated with staff training and implementation. Once established, it is easier to maintain than capture/playback or data driven automation and represents the next step in the continuing evolution of testing.

Jim Schaefer

Jim Schaefer is currently the manager of the Automation Test Team at Capital One and is the inventor of Capital One's table driven automation test tool - Enterprise Test Engine Suite Technology (E-TEST). He has been working in the computer industry for the past twenty years and has specialized in software testing for the last eight. He has taught, lectured, and appeared in various technical and trade publications while working in the areas of banking and telephony.

Jim is currently the President of the Richmond Area Mercury Users Group and co-recipient of Telephony magazine's First Fiber in the Loop Award. He currently enjoys working locally as a volunteer with FIRST (For Inspiration and Recognition of Science and Technology) whose mission is to excite young people about the fun, accessibility, and importance of science and engineering through robotics competitions at the pre-college level.

James S. Schaefer, CCP
Capital One E-Commerce
jim.schaefer@capitalone.com