



TAC Architecture and High-Level Design

Overview

Test & Automation Consulting LLC (TAC) custom crafts each automation project to ensure the highest return on investment (ROI) for each client. However, it is not cost effective for our clients to “reinvent the wheel” each time around. Therefore, TAC brings certain automation testing artifacts to each non-trivial engagement.

TAC has a core architecture and high-level design methodology to solve most common automation problems. This architecture is not tied into a specific tool or SDLC. It can be used with any automation tool¹ that is supported by the TAC **Test-Automation Programming Interface (T-API)**. It is independent of the test methodology used. T-API is simply a pragmatic solution for automation problems that have been seen in the past.

To the standard TAC architecture, we add custom solutions for the unique problems that occur in each project.

This document will discuss standard TAC architecture and high-level design topics. It will be helpful to have read the architectural theory document, **TAC Automation Architecture Theory.doc** before attempting to read this document. Individual techniques to implement the TAC architecture and processes are documented in the **TAC Processes and Techniques.Doc**. A separate document discusses the reasons for the design choices that have been made: **TAC Managing Automation Complexity.doc**.

TAC Architecture

At present, the main TAC architecture is essentially a Framework-based architecture that uses data-driven techniques where applicable. In the following sections, we will discuss various architectural and design topics.

Infrastructure vs. Framework

The TAC framework on any given project is a combination of the standard TAC Infrastructure and a custom built framework for the application’s specific needs. We need to be very specific about the nomenclature we use.

The TAC **Infrastructure** is a combination of code, tools, templates, and processes that is application neutral. That is, a total of all the “automation stuff” which Test & Automation Consulting LLC brings to an engagement. It includes what we commonly refer to as the Test – Automation Programming Interface (T-API):

¹ Currently Mercury WinRunner™, Compuware QARun™, and Rational Robot™ are supported. The TAC Infrastructure will be ported to other automation tool sets when warranted.



Innovative solutions to maximize your testing investment.

- Native Tool code libraries (see **TAC X Native Code T-API Libraries.doc** where X is the tool name)
- TAC custom DLL code libraries (see **TAC Custom DLL Libraries.doc**)
- TAC external Log (see **TAC External Log Design.Doc**)
- TAC Script Templates (see **TAC X Script Templates.doc** where X is a specific tool name.)
- TAC test header data structures (see **TAC Testcase Header Data Structure.Doc**) that is used to control the test suite execution.

On the other hand, when we talk about the **Framework**, we will be discussing the application-specific code and artifacts that are built to support a particular project for a client. This includes such things as **abstract functions** (see discussion below), **test header implementations**, and **data access** solutions.

Often, code that is created explicitly for a specific project will be added to the TAC Infrastructure rather than the application framework. This will occur when the code is deemed to have wider relevance than to just a single application.

Separation between the Infrastructure and the framework is usually done by placing the code in separate scripts (or include files, depending on the automation tool being used.) Typically, the Infrastructure code is placed in scripts or files with TAC in the name. Framework specific code is typically placed in scripts or include files with the organization or company name embedded in the file name.

At present, most TAC automation implementations are script based. That is, individual test scripts contain executable code that is run through an automation tool. In the future, we expect to move closer towards a **Table-driven** architecture where the executable code is stored in database tables or Excel spreadsheets for certain projects when warranted.

Functional Decomposition

The heart of the TAC automation methodology is the logical decomposition of an application into its individual tasks. This should not be conceptually difficult; we usually think of an application as an interrelated set of tasks anyway.

When an application is designed, it normally performs a variety of jobs. It might create and destroy records, sort and collate items, and transform data in many different ways. A person who is going to use the application is given training in all of the different tasks that they are expected to perform on the system. Often, many different people perform a variety of tasks using the same system. The training manuals are often a good source for identifying tasks for decomposition.

Another way to start the decomposition process is to use the menu structure of the system. Typically, the menu items are verbs or action words: do this, do that. These tend to be tasks. Looking in the help files also can help identify discrete tasks.



Innovative solutions to maximize your testing investment.

For automation purposes, we want to decompose an application into its separate tasks. The two main reasons for this decomposition are **Complexity Management** and **Suite Scalability**.

Complexity Management

When you look at a large application in its totality, no human mind can grasp all of the details or all of the ramifications of all the state changes that are possible. That is essentially why we want to break a larger system into individual tasks in the first place: so that each individual who has responsibilities in using the system can do their job successfully.

A single task, however, is usually small enough to completely understand. What are the inputs, outputs, side effects, expectations, possible error conditions, etc? It turns out that most of this information is required to correctly automate the task anyway. What controls need to be manipulated? What kind of data is needed? How does performing the task change the state of the application? What are the error conditions that could occur?

If we were performing record / playback (R/P) automation, our only real concern would be the data used to perform the task. The script would input the data into the correct controls, trigger the task, and assume that everything would work correctly.

In real testing, however, everything does not always work perfectly. Our concern needs to be for those times when things do not go well, as those are the times that a test script can fail leading to full test suite stoppage. When there is a failure to perform a task successfully, we need the ability to recognize the failure, log the problem completely, and continue the suite by going on to the next test case. That can only be done by first gathering the knowledge to fully understand the task, and then encapsulating the task into a callable function using that knowledge.

Scalability

Each individual task performed in an application is liable to be executed many times in many different contexts. As we build up our test suite by adding more test cases, it is inevitable that we will be performing the same tasks multiple times (probably using different data each time.)

If the method of doing the task changes from release to release, we must repair the code performing the task in each place it is used. By decomposing the application into discrete tasks, and then scripting each task as a function, repairing the suite when functionality changes is much easier. No matter how many times a task is performed, there is still only one point of maintenance. That adds the property of scalability to the test suite.



Abstract Functions

In the above section, we discussed the functional decomposition of an application into its component tasks. In TAC T-API, the functions written to support individual tasks are called **Abstract Functions**. In other automation methodologies, these may be called **Action Words** or **Keywords**.

We tend to call these abstract functions because the task that is to be performed is in many ways based on an abstract idea. Consider the example of opening a file in a word processor.

There clearly is not a physical file (i.e. a manila folder physically existing in a sliding drawer in a filing cabinet.) There is a stream of bits, existing somewhere on a hard drive that will be interpreted by the application as a single entity. When the correct command is given, the operating system (OS) will retrieve the stream of bits and manipulate it in such a way that a view of it can be placed in the main window (itself only an abstraction.)

So the idea of opening a file is really a metaphor for some canned routines that the application will perform in conjunction with the operating system. Beyond that however, what really makes this an abstract idea is that we have given no firm details on how to perform the task. Open a file! Do we use the toolbar? Menu structure? Once in the open dialog, do we use the file list? The directory tree?

We keep the idea abstract because we do not want to deal with the details at the script level. After all, we opened a file in Word for DOS, in Word for Windows 3.1, in Word for Windows 97, 2000, XP, etc. Ideally, we would like to use the same scripts to test the application through several versions. By separating the implementation details from the interface, we can avoid having to change the scripts performing the tests.

Yes, we certainly must change the function implementation when the application interface changes. This is unavoidable since an automation tool actually works by creating a one-to-one correspondence between the interface controls of an application and the commands to manipulate those controls in a script. Type a string into this edit, choose an item from a combo box, click on this radio button. At its most basic, an automation test script is a series of commands telling the controls of an application what to do. We cannot break the one-to-one correspondence between automation code and GUI objects. What we can do, however, is move that relationship out of the test script and into an include file (containing the decomposed functions.) Since the included function by its very nature is a one to many relationship (one function used in multiple places), we have essentially made the line of code / control correspondence one to many.

Without this ability to isolate and reuse code, test automation is just not practical. An organization very quickly reaches the point of diminishing returns where it spends more time maintaining existing code (tests) and is no longer creating more. The failure to make automation scalable has killed many projects. At Test & Automation Consulting,



Innovative solutions to maximize your testing investment.

scalability is one of the most important considerations when designing an automation suite, and the heart of that scalability is our abstract functions.

The structure of the abstract function will usually follow the “4-step dance” model as explained in “**TAC Automation Models.Doc**”.

Script Templates

Test automation is a programming task. Unfortunately, most test teams do not have programmers available to write the automation, nor would that likely be an efficient use of resources.

Instead, we want to create an framework that allows lesser-trained, non-developer class scripters to create the test cases. See the document, **TAC Managing Automation Complexity.doc**, and the section on architecture above.

Each test case will be captured in a script. Each script must execute a number of architectural elements as well as the test case. To simplify the task for the scripter, the architectural elements and the error recovery tasks will be built into every script before any test case elements are added.

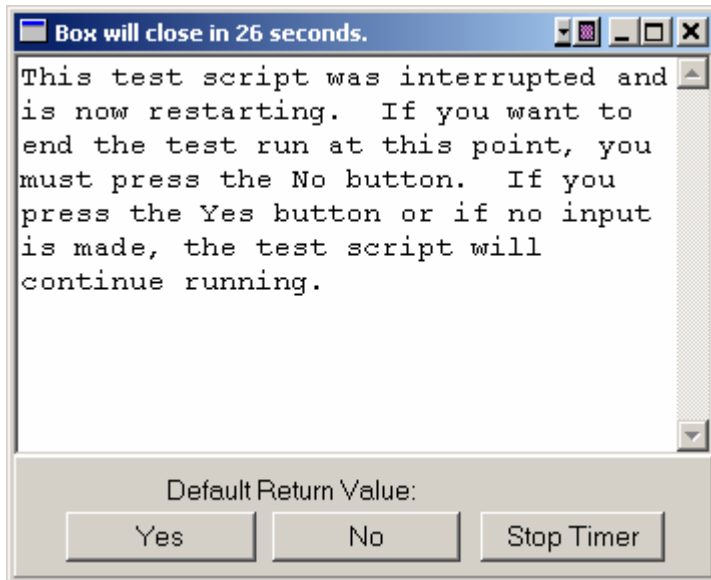
To build a complete test suite, several different kinds of templates exist. These are discussed in the documents, “**TAC XXX Script Templates.doc**”, where **XXX** is the name of the automation tool being used.

Suite Restartability

The TAC architecture has extensive error handling and fault tolerance built in. However, there is always the possibility of a catastrophic failure that cannot be handled by the architecture. Such cases include general protection faults (GPF) or the dreaded “blue screen of death (BSOD.) In these cases, the operating system itself is stopped. Obviously, a task running on the OS cannot handle this level of failure.

A TAC T-API exclusive feature is the ability to restart the test suite after such a catastrophic failure. This feature is controlled by a variable [**TAC_UseRestartFeature**], usually initialized in the startup script. When this variable is set to TRUE, the restartable feature is enabled.

This feature works by storing suite run-time information in the system Registry. This information is a running account of the current state of the suite. Should a catastrophic failure occur, human intervention must be taken (usually by rebooting the workstation.) To restart the suite, simply start the same script that was run originally. When the suite starts, it will detect that an incomplete run had occurred. If the restart feature is enabled, a prompt box will pop up notifying the test runner that there had been a failure, and that the suite execution could be continued, or the suite run could be concluded immediately. The following is the prompt:



Default action is to simply continue the suite from where the failure occurred. If no input is given or the Yes button is pressed, the suite will fast forward through all scripts which have already been executed until it gets to the exact test case that was running. At that point, another prompt will occur, asking the user if they want to run the exact test case that was running when the failure occurred, or to skip it and go to the next test case.

Obviously, if the test case itself caused the problem, it is best to skip the test case. If there is no proof that the test case caused the problem directly, then choosing to run it again is the best decision.

The suite may be restarted any number of times; at the end of the suite, there will still be only one log containing all suite-run information. A restart notification line is added to the log each time a restart occurs; otherwise, there is no difference between a log with restarts and a standard log.

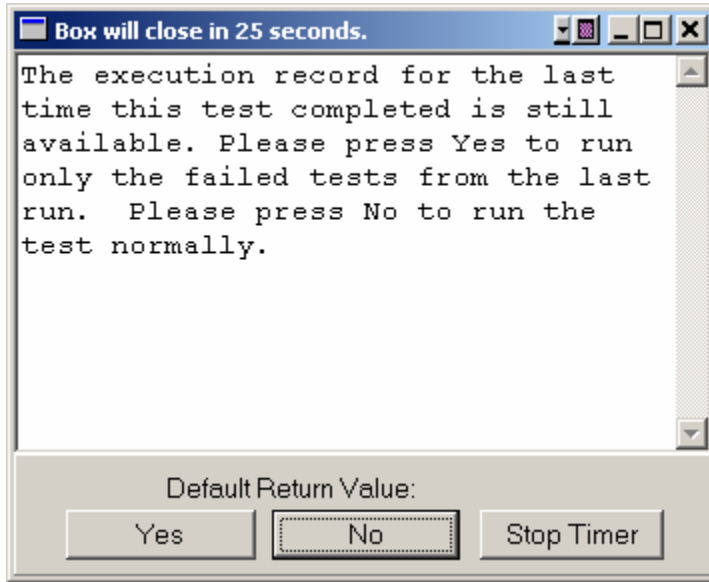
The restart feature allows the test runner to stop the suite intentionally if desired, and restart it later at any time. This feature adds extra flexibility that is very useful when sharing workstations with manual testers.

Failure Only Mode

An unfortunate problem with test automation is that often test cases fail during a suite execution for reasons that have nothing to do with SW defects in the system being tested. Timing glitches, spurious windows, temporary discontinuities, etc., all can cause test scripts to fail. After the suite has been run, the first step in accepting the test results is often re-running all failed test cases to see if they actually failed due to defects, or due to the afore mentioned glitches.



The TAC T-API has built into it a special mode that allows the test suite to be restarted and run only those test cases that were not run or failed in the first run. This mode is controlled by a global variable, **TAC_AllowFailedTestRun**, usually set in the startup script. When set to TRUE, the persistent information saved off to the Registry is not destroyed at the end of the test run; instead, it is left intact. When the suite is again started, its existence triggers a prompt as shown below.



If the test runner presses the Yes button, the automation suite will start running using the exact same parameters as the previous run. As each test is started, its previous results are checked in the Registry. If the test case had passed, a message is placed in the log to the effect that it will not be run here. If, however, the test had not been run (for whatever reason) or if it had failed, it will be run now. The only exception is if the prerequisites are still not met, or the test case is explicitly set to not run.

At the end of the Failed only run, all run records are removed from the Registry.

Test Level

Typically, during the software development life cycle (SDLC), there are many different times when test cases may need to be run. It may not be efficient to run all test cases each cycle. In the real world, we want to be able to selectively run certain test cases without having to run the entire suite every time.

The TAC T-API handles this by classifying each test case by level. When starting the suite, the test runner is prompted to enter which level of testing is to be executed. Once a level is chosen, the test cases are filtered and only those at the desired level or lower are actually called.



In the test header table, each test case is assigned a level, from one to five. A one corresponds to the lowest test level, usually designated as a **smoke** or **build verification** test. This is usually a very small subset of all the available tests; this level is selected when we want to run a few test cases to test the viability of a new build. It typically is very wide ranging; a few tests from each functional area, simply performing a sanity test to decide whether the build has enough quality to put it into the formal test process.

At the other extreme, a five means a full regression set. Every test case that is defined will be run when this level is selected.

The intermediate levels can be defined in any way the organization wants. One standard level may be defined as the critical set of tests; those that must pass before the build is certified for release.

It must be understood when defining levels that, when a level is chosen, it implicitly includes all lower levels. For example, choosing a level 3 will mean that all test cases with defined levels of 3, 2 and 1 will be executed.

Header pickup standard

As noted elsewhere, each test case must have a header defined that contains test meta data (including test case name, description, prerequisites, level, etc.) This header is automatically picked up at run time via a function call in the script. The function, **PickupTestHdr_TAC(HdrArr, GlobValArr)**, is called from the template part of the script. There is no reason for a scripter to ever touch this area of the script.

Two arrays (**HdrArr** and **GlobValArr**) are filled during the execution of this function. The indexes into these arrays are the column headers from the spreadsheets / database tables. The GlobValArr will contain any global values defined in the optional header data store.

Header pickup DD

Each data-driven (DD) testcase also has a header defined and an optional set of global values. These are picked up at the beginning of a DD test case by the function, **PickupDDTestHdr_TAC(Iteration, DDTCHdr, DDGlobVars)**. As in the case of the standard header, this function is executed in the template portion of the script. Therefore, the scripter should never touch this function.

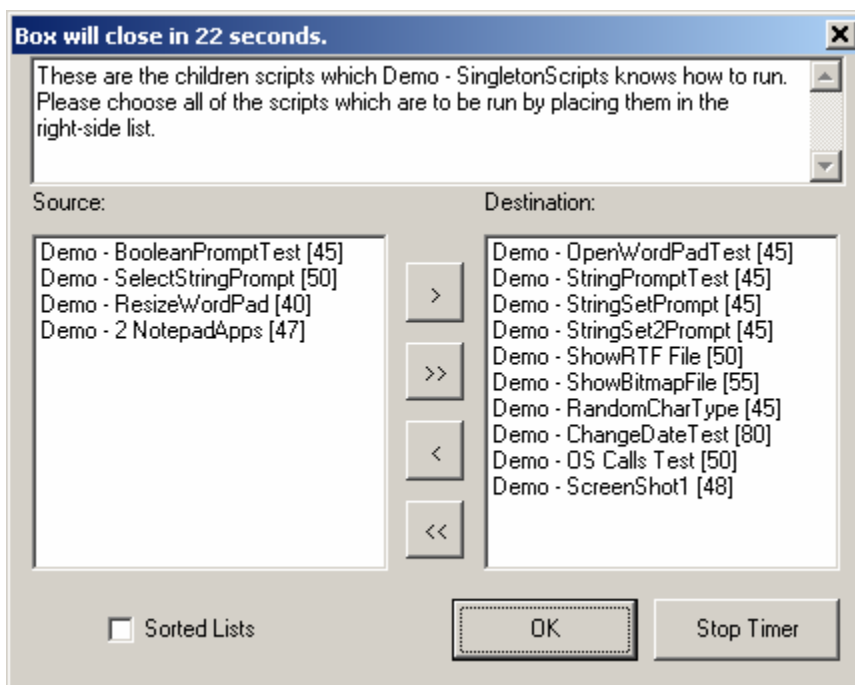
Script Selection Prompting

One of the most convenient features of the TAC T-API is the ability of the test runner to easily choose the test cases to execute at run time. In addition to the standard filters of test level and functional areas, there may be other prompts popped up for the user. To make the suite as interactive as possible, the automator may allow the test runner to choose individual test cases or test sets directly.



Innovative solutions to maximize your testing investment.

This is done by using a **Promptable Parent** script template. During the execution of the template, a function (**PromptForScripts_TAC()**) is executed. This iterates through the header table and finds all scripts that are defined for this parent. Each child script of a promptable parent with a value of 1 in the “Prompt” column will be represented in the prompt dialog (a value of 0 in the prompt column will cause the script to be run automatically, without giving the tester a choice.) Another column in the header is “Selected”. The value of this column determines where in the prompt box the test case will initially be placed. If the value is 0, the test case will be placed in the left list; all test cases in this list box are available but not currently selected.



All test cases with a 1 in the “Selected” column will be placed in the right list; these test cases are considered selected, and will be executed unless the test runner moves the test into the left list. The test runner may freely move any or all test cases from one list to the other, using the buttons supplied, drag-and-drop, or simply by double clicking on any item. When the OK button is pressed, or when the timer expires, all test cases in the right list will be executed.

Each script name in the list has appended to it the relative ordering value in brackets. These values are picked up from the “RelOrder” column and used during the actual run to determine the order of execution; the current order in the list has no effect..

GetNextCurlIndex

When executing a set of data-driven (DD) test cases, T-API needs to have a way of iterating through the list of tests. Normally, iteration though any list is done using a



Innovative solutions to maximize your testing investment.

“FOR” loop. Unfortunately, a FOR loop is insufficient for T-API. The ability to restart a suite mandates that the current iteration number be persistent. If there is a catastrophic failure while looping through the test cases, we must be able to determine which iteration failed for restart purposes.

The function, `GetNextCurIndex_TAC()`, is used to determine the iteration to run next. The only argument passed in is the maximum number of iterations defined. On first call, the function returns 1 (always start at the first iteration.) Every call thereafter returns the next possible numerical value until the maximum value is reached. The iteration after the maximum has been reached results in a return value of 0.

The iteration value is stored in the registry; therefore, a restart is always possible.

Running a child script

Every automation tool has the capacity of allowing a script (called the parent) to invoke another script (called the child.) The function, `RunChildScript_TAC()`, is defined in native code for each different tool.

In T-API, we want to make sure that the execution invocation is always done through this special function. T-API has several reasons for using this method of indirection:

- In case of suite restart, T-API allows us to iterate very quickly through those test cases that had already been executed.
- Certain critical global variables can be saved off so that any changes to the variables in the child scripts can be rolled back when the called scripts return.
- Updates to the TAC HeadsUp Display are done in this function

Finalize script

At the end of every test case, we need to be able to clean up the environment to prepare for the next test case. We also need to log the results of the test case, as well as save off the results to the Registry for prerequisite, restart, and failed run usage. This functionality is supplied through the `Finalize()` function call. This function may be customized for each application that is tested.