

# Test Automation Frameworks

By Carl Nagle

*"When developing our test strategy, we must minimize the impact caused by changes in the applications we are testing, and changes in the tools we use to test them."*

--Carl J. Nagle

## 1.1 Thinking Past "The Project"

In today's environment of plummeting cycle times, test automation becomes an increasingly critical and strategic necessity. Assuming the level of testing in the past was sufficient (which is rarely the case), how do we possibly keep up with this new explosive pace of web-enabled deployment while retaining satisfactory test coverage and reducing risk? The answer is either more people for manual testing, or a greater level of test automation. After all, a reduction in project cycle times generally correlates to a reduction of time for test.

With the onset and demand for rapidly developed and deployed web clients test automation is even more crucial. Add to this the cold, hard reality that we are often facing more than one active project at a time. For example, perhaps the team is finishing up Version 1.0, adding the needed new features to Version 1.1, and prototyping some new technologies for Version 2.0!

Better still; maybe our test team is actually a pool of testers supporting many diverse applications completely unrelated to each other. If each project implements a unique test strategy, then testers moving among different projects can potentially be more a hindrance rather than a help. The time needed for the tester to become productive in the new environment just may not be there. And, it may surely detract from the productivity of those bringing the new tester up to speed.

To handle this chaos we have to think past the project. We cannot afford to engineer or reengineer automation frameworks for each and every new application that comes along. We must strive to develop a single framework that will grow and continuously improve with each application and every diverse project that challenges us. We will see the advantages and disadvantages of these different approaches in [Section 1.2](#)

### 1.1.1 Problems with Test Automation

Historically, test automation has not met with the level of success that it could. Time and again test automation efforts are born, stumble, and die. Most often this is the result of misconceived perceptions of the effort and resources necessary to implement a successful, long-lasting automation framework. Why is this, we might ask? Well, there are several reasons.

Foremost among this list is that automation tool vendors do not provide completely forthright demonstrations when showcasing the "simplicity" of their tools. We have seen the vendor's sample applications. We have seen the tools play nice with *those* applications. And we try to get the tools to play nice with *our* applications just as fluently. Inherently, project after project, we do not achieve the same level of success.

# Test Automation Frameworks

By Carl Nagle

This usually boils down to the fact that our applications most often contain elements that are not compatible with the tools we use to test them. Consequently, we must often mastermind technically creative solutions to make these automation tools work with our applications. Yet, this is rarely ever mentioned in the literature or the sales pitch.

The commercial automation tools have been chiefly marketed for use as solutions for testing an application. They should instead be sought as the cornerstone for an enterprise-wide test automation framework. And, while virtually all of the automation tools contain some scripting language allowing us to get past each tool's failings, testers have typically neither held the development experience nor received the training necessary to exploit these programming environments.

---

*"For the most part, testers have been testers, not programmers. Consequently, the 'simple' commercial solutions have been far too complex to implement and maintain; and they become shelfware."*

---

Most unfortunate of all, otherwise fully capable testers are seldom given the time required to gain the appropriate software development skills. For the most part, testers have been testers, not programmers. Consequently, the "simple" commercial solutions have been far too complex to implement and maintain; and they become shelfware.

Test automation must be approached as a full-blown software development effort in its own right. Without this, it is most likely destined to failure in the long term.

## Case Study: Costly Automation Failures

In 1996, one large corporation set out evaluating the various commercial automation tools that were available at that time. They brought in eager technical sales staff from the various vendors, watched demonstrations, and performed some fairly thorough internal evaluations of each tool.

By 1998, they had chosen one particular vendor and placed an initial order for over \$250,000 worth of product licensing, maintenance contracts, and onsite training. The tools and training were distributed throughout the company into various test departments--each working on their own projects.

None of these test projects had anything in common. The applications were vastly different. The projects each had individual schedules and deadlines to meet. Yet, every one of these departments began separately coding functionally identical common libraries. They made routines for setting up the Windows test environment. They each made routines for accessing the Windows programming interface. They made file-handling routines, string utilities, database access routines--the list of code duplication was disheartening!

# Test Automation Frameworks

By Carl Nagle

For their test designs, they each captured application specific interactive tests using the capture/replay tools. Some groups went the next step and modularized key reusable sections, creating reusable libraries of application-specific test functions or scenarios. This was to reduce the amount of code duplication and maintenance that so profusely occurs in pure captured test scripts. For some of the projects, this might have been appropriate if done with sufficient planning and an appropriate automation framework. But this was seldom the case.

With all these modularized libraries testers could create functional automated tests in the automation tool's proprietary scripting language via a combination of interactive test capture, manual editing, and manual scripting.

One problem was, as separate test teams they did not think past their own individual projects. And although they were each setting up something of a reusable framework, each was completely unique--even where the common library functions were the same! This meant duplicate development, duplicate debugging, and duplicate maintenance. Understandably, each separate project still had looming deadlines, and each was forced to limit their automation efforts in order to get *real* testing done.

As changes to the various applications began breaking automated tests, script maintenance and debugging became a significant challenge. Additionally, upgrades in the automation tools themselves caused significant and unexpected script failures. In some cases, the necessity to revert back (downgrade) to older versions of the automation tools was indicated. Resource allocation for continued test development *and* test code maintenance became a difficult issue.

Eventually, most of these automation projects were put on hold. By the end of 1999--less than two years from the inception of this large-scale automation effort--over 75% of the test automation tools were back on the shelves waiting for a new chance to try again at some later date.

## 1.1.2 Some Test Strategy Guidelines

Past failings like these have been lessons for the entire testing community. Realizing that we must develop reusable test strategies is no different than the reusability concerns of any good application development project. As we set out on our task of automating test, we must keep these past lessons forefront.

In order to make the most of our test strategy, we need to make it reusable and manageable. To that end, there are some essential guiding principles we should follow when developing our overall test strategy:

- Test automation is a fulltime effort, not a sideline.
- The test design and the test framework are totally separate entities.
- The test framework should be application-independent.
- The test framework must be easy to expand, maintain, and perpetuate.
- The test strategy/design vocabulary should be framework independent.
- The test strategy/design should remove most testers from the complexities of the test framework.

# Test Automation Frameworks

By Carl Nagle

These ideals are not earth shattering. They are not relatively new. Yet, it is seldom these principles are fully understood and instrumented.

So what do they mean?

## **1.1.3 Test automation is a fulltime effort, not a sideline.**

While not necessarily typical design criteria, it bears repeating. The test framework design and the coding of that design together require significant front-loaded time and effort. These are not things that someone can do when they have a little extra time here, or there, or between projects. The test framework must be well thought out. It must be documented. It should be reviewed. It should be tested. It is a full software development project like any other. This bears repeating--again.

Will our test framework development have all of these wonderful documentation, design, review, and test processes? Does our application development team?

We should continuously push for both endeavors to implement all these critical practices.

## **1.1.4 The test design and the test framework are totally separate entities.**

The test design details how the particular functions and features of our application will be tested. It will tell us what to do, how and when to do it, what data to use as input, and what results we expect to find. All of this is specific to the particular application or item being tested. Little of this requires any knowledge or care of whether the application will be tested automatically or manually. It is, essentially, the "how to" of what needs to be tested in the application.

On the other hand, the test framework, or specifically, the test automation framework is an execution environment for automated tests. It is the overall system in which our tests will be automated. The development of this framework requires completely different technical skills than those needed for test design.

## **1.1.5 The test framework should be application-independent.**

Although applications are relatively unique, the components that comprise them, in general, are not. Thus, we should focus our automation framework to deal with the common components that make up our unique applications. By doing this, we can remove all application-specific context from our framework and reuse virtually everything we develop for every application that comes through the automated test process.

# Test Automation Frameworks

By Carl Nagle

*"We should focus our automation framework to deal with the common components that make up our unique applications."*

---

Nearly all applications come with some form of menu system. They also have buttons to push, boxes to check, lists to view, and so on. In a typical automation tool script there is, generally, a very small number of component functions for each type of component. These functions work with the component objects independent of the applications that contain them.

Traditional, captured automation scripts are filled with thousands of calls to these component functions. So the tools already exist to achieve application independence. The problem is, most of these scripts construct the function calls using application-specific, hard coded values. This immediately reduces their effectiveness as application-independent constructs. Furthermore, the functions by themselves are prone to failure unless a very specific application state or synchronization exists at the time they are executed. There is little error correction or prevention built-in to these functions.

To deal with this in traditional scripts we must place additional code before and/or after the command, or a set of commands, to insure the proper application state and synchronization is maintained. We need to make sure our window has the current focus. We need to make sure the component we want to select, or press, or edit exists and is in the proper state. Only then can we perform the desired operation and separately verify the result of our actions.

For maximum robustness, we would have to code these state and synchronization tests for every component function call in our scripts. Realistically, we could never afford to do this. It would make the scripts huge, nearly unreadable, and difficult to maintain. Yet, where we forego this extra effort, we increase the possibility of script failure.

What we must do is develop a truly application-independent framework for these component functions. This will allow us to implement that extra effort just once, and execute it for every call to any component function. This framework should handle all the details of insuring we have the correct window, verifying the element of interest is in the proper state, doing something with that element, and logging the success or failure of the entire activity.

We do this by using variables, and providing application-specific data to our application-independent framework. In essence, we will provide our completed test designs as executable input into our automation framework.

Does this mean that we will *never* have to develop application-specific test scripts? Of course not. However, if we can limit our application-specific test scripts to some small percentage, while reusing the best features of our automation framework, we will reap the rewards project after project.

## **1.1.6 The test framework must be easy to expand, maintain, and perpetuate.**

# Test Automation Frameworks

By Carl Nagle

One of our goals should be a highly modular and maintainable framework. Generally, each module should be independent and separate from all the other modules. What happens inside one is of no concern to the others.

With this modular black-box approach, the functionality available within each module can be readily expanded without affecting any other part of the system. This makes code maintenance much simpler. Additionally, the complexity of any one module will likely be quite minimal.

However, modularity alone will not be enough to ensure a highly maintainable framework. Like any good software project, our design must be fully documented and published. Without adequate, published documentation it will be very difficult for anyone to decipher what it is the framework is designed to do. Any hope of maintenance will not last far beyond the departure of the original framework designers. Our test automation efforts will eventually become another negative statistic.

To prevent this, we should define documentation standards and templates. Wherever possible, module documentation should be developed "in-context". That is, directly in the source code itself. Tools should be retained, or designed and developed, so that we can automatically extract and publish the documentation. This will eliminate the task of maintaining two separate sets of files: the source code, and its documentation. It will also provide those doing the code maintenance quite a ready reference. Nearly everything they need to know should exist right there in the code.

---

*"We must always remember: our ultimate goal is to simplify and perpetuate a successful automation framework."*

---

We must always remember: our ultimate goal is to simplify and perpetuate a successful test automation framework. To put something in place that people will use and reuse for as long as it is technically viable and productive.

## **1.1.7 The test strategy/design vocabulary should be framework independent.**

As noted before, the framework refers to the overall environment we construct to execute our tests. The centerpiece is usually one of many commercially available automation tools. In good time, it may be more than one. In some rare circumstances, it might even be a proprietary tool developed or contracted specifically for our test automation needs.

The point is, different tools exist and some will work better for us than others in certain situations. While one tool might have worked best with our Visual Basic or C/C++ applications, we may need to use a different tool for our web clients. By keeping a specific tool consideration out of our test designs, we avoid limiting our tests to that tool alone.

The overall test strategy will define the format and *low-level* vocabulary we use to test *all* applications much like an automation tool defines the format and syntax of the scripting language it

# Test Automation Frameworks

By Carl Nagle

provides. Our vocabulary, however, will be independent of any particular test framework employed. The same vocabulary will migrate with us from framework to framework, and application to application. This means, for example, the syntax used to click a button will be the same regardless of the tool we use to execute the instruction or the application that contains the button.

The test design for a particular application, however, will define a *high-level* vocabulary that is specific to that application. While this high-level vocabulary will be application specific, it is still independent of the test framework used to execute it. This means that the high-level instruction to login to our website with a particular user ID and password will be the same regardless of the tool we use to execute it.

When we provide *all* the instructions necessary to test a particular application, we should be able to use the exact same instructions on any number of different framework implementations capable of testing that application. We must also consider the very likely scenario that some or all of this testing may, at one time or another, be manual testing. This means that our overall test strategy should not only facilitate test automation, it should also support manual testing.

Consequently, the format and vocabulary we use to test our applications should be intuitive enough for mere mortals to comprehend and execute. We should be able to hand our test over to a person, point to an area that failed, and that person should be able to manually reproduce the steps necessary to duplicate the failure.

---

*"A good test strategy can remove the necessity for both manual and automated test scripts. The same 'script' should suffice for both."*

---

A good test strategy, comprised of our test designs and our test framework, can remove the necessity for both manual and automated test scripts for the same test. The same "script" should suffice for both. The important thing is that the vocabulary is independent of the framework used to execute it. And the test strategy must also accommodate manual testing.

## **1.1.8 The test strategy/design should remove most testers from the complexities of the test framework.**

In practice, we cannot expect all our test personnel to become proficient in the use of the automation tools we use in our test framework. In some cases, this is not even an option worth considering. Remember, generally, testers are testers--they are not programmers. Sometimes our testers are not even professional testers. Sometimes they are application domain experts with little or no use for the technical skills needed for software development.

Sometimes testers are application developers splitting time between development and test. And when application developers step in to perform testing roles, they do not want or need a complex test scripting language to learn. That is what you get with commercial automation tools. And that

# Test Automation Frameworks

By Carl Nagle

may even be counter-productive and promote confusion since some of these scripting languages are modified subsets of standard programming languages. Others are completely unique and proprietary.

Yet, with the appropriate test strategy and vocabulary as discussed in the previous section, there is no reason we should not be able to use all our test resources to design tests suitable for automation without knowing anything about the automation tools we plan to deploy.

The bulk of our testers can concentrate on test design, and test design only. It is the automation framework folks who will focus on the tools and utilities to automate those tests.

## 1.2 Data Driven Automation Frameworks

Over the past several years there have been numerous articles done on various approaches to test automation. Anyone who has read a fair, unbiased sampling of these knows that we cannot and *must not* expect pure capture and replay of test scripts to be successful for the life of a product. We will find nothing but frustration there.

Sometimes this manifesto is hard to explain to people who have not yet performed significant test automation with these capture\replay tools. But it usually takes less than a week, often less than a day, to hear the most repeated phrase: "It worked when I recorded it, but now it fails when I play it back!"

Obviously, we are not going to get there from here.

### 1.2.1 Data Driven Scripts

Data driven scripts are those application-specific scripts captured or manually coded in the automation tool's proprietary language and then modified to accommodate variable data. Variables will be used for key application input fields and program selections allowing the script to drive the application with external data supplied by the calling routine or the shell that invoked the test script.

#### **Variable Data, Hard Coded Component Identification:**

These data driven scripts often still contain the hard coded and sometimes very fragile recognition strings for the window components they navigate. When this is the case, the scripts are easily broken when an application change or revision occurs. And when these scripts start breaking, we are not necessarily talking about just a few. We are sometimes talking about a great many, if not all the scripts, for the entire application.

Figure 1 is an example of activating a server-side image map link in a web application with an automation tool scripting language:

# Test Automation Frameworks

By Carl Nagle

```
Image Click "DocumentTitle=Welcome;\;ImageIndex=1" "Coords=25,20"
```

Figure 1

This particular scenario of clicking on the image map might exist thousands of times throughout all the scripts that test this application. The preceding example identifies the image by the title given to the document and the index of the image on the page. The hard coded image identification *might* work successfully all the way through the production release of *that* version of the application. Consequently, testers responsible for the automated test scripts may gain a false sense of security and satisfaction with these results.

However, the next release cycle may find some or all of these scripts broken because either the title of the document or the index of the image has changed. Sometimes, with the right tools, this might not be too hard to fix. Sometimes, no matter what tools, it will be frustratingly difficult.

Remember, we are potentially talking about thousands of broken lines of test script code. And this is just one particular change. Where there is one, there will likely be others.

## **Highly Technical or Duplicate Test Designs:**

Another common feature of data driven scripts is that virtually all of the test design effort for the application is developed in the scripting language of the automation tool. Either that, or it is duplicated in *both* manual and automated script versions. This means that everyone involved with automated test development or automated test execution for the application must likely become proficient in the environment and programming language of the automation tool.

## **Findings:**

A test automation framework relying on data driven scripts is definitely the easiest and quickest to implement *if* you have *and* keep the technical staff to maintain it. But it is the hardest of the data driven approaches to maintain and perpetuate and very often leads to long-term failure.

## **1.2.2 Keyword or Table Driven Test Automation**

Nearly everything discussed so far defining our ideal automation framework has been describing the best features of "keyword driven" test automation. Sometimes this is also called "table driven" test automation. It is typically an application-independent automation framework designed to process our tests. These tests are developed as data tables using a keyword vocabulary that is independent of the test automation tool used to execute them. This keyword vocabulary should also be suitable for manual testing, as you will soon see.

## **Action, Input Data, and Expected Result ALL in One Record:**

The data table records contain the keywords that describe the actions we want to perform. They also provide any additional data needed as input to the application, and where appropriate, the benchmark information we use to verify the state of our components and the application in general.

# Test Automation Frameworks

By Carl Nagle

For example, to verify the value of a user ID textbox on a login page, we might have a data table record as seen in Table 1:

WINDOW	COMPONENT	ACTION	EXPECTED VALUE
LoginPage	UserIDTextbox	VerifyValue	"MyUserID"

Table 1

## **Reusable Code, Error Correction and Synchronization:**

Application-independent component functions are developed that accept application-specific variable data. Once these component functions exist, they can be used on each and every application we choose to test with the framework.

Figure 2 presents pseudo-code that would interpret the data table record from Table 1 and Table 2. In our design, the primary loop reads a record from the data table, performs some high-level validation on it, sets focus on the proper object for the instruction, and then routes the complete record to the appropriate component function for full processing. The component function is responsible for determining what action is being requested, and to further route the record based on the action.

Framework Pseudo-Code
<p><b>Primary Record Processor Module:</b></p> <pre>Verify "LoginPage" Exists. (Attempt recovery if not) Set focus to "LoginPage". Verify "UserIDTextbox" Exists. (Attempt recovery if not) Find "Type" of component "UserIDTextbox". (It is a Textbox) Call the module that processes ALL Textbox components.</pre>
<p><b>Textbox Component Module:</b></p> <pre>Validate the action keyword "VerifyValue". Call the Textbox.VerifyValue function.</pre>
<p><b>Textbox.VerifyValue Function:</b></p> <pre>Get the text stored in the "UserIDTextbox" Textbox. Compare the retrieved text to "MyUserID". Record our success or failure.</pre>

Figure 2

# Test Automation Frameworks

By Carl Nagle

## Test Design for Man and Machine, With or Without the Application:

Table 2 reiterates the actual data table record run by the automation framework above:

WINDOW	COMPONENT	ACTION	EXPECTED VALUE
LoginPage	UserIDTextbox	VerifyValue	"MyUserID"

Table 2

Note how the record uses a vocabulary that can be processed by both man and machine. With *minimal* training, a human tester can be made to understand the record instruction as deciphered in Figure 3:

**On the LoginPage, in the UserIDTextbox,  
Verify the Value is "MyUserID".**

Figure 3

Once they learn or can reference this simple vocabulary, testers can start designing tests without knowing anything about the automation tool used to execute them.

Another advantage of the keyword driven approach is that testers can develop tests without a functioning application as long as preliminary requirements or designs can be determined. All the tester needs is a fairly reliable definition of what the interface and functional flow is expected to be like. From this they can write most, if not all, of the data table test records.

Sometimes it is hard to convince people that this advantage is realizable. Yet, take our login example from Table 2 and Figure 3. We do not need the application to construct any login tests. All we have to know is that we will have a login form of some kind that will accept a user ID, a password, and contain a button or two to submit or cancel the request. A quick discussion with development can confirm or modify our determinations. We can then complete the test table and move on to another.

We can develop other tests similarly for any part of the product we can receive or deduce reliable information. In fact, if in such a position, testers can actually help guide the development of the UI and flow, providing developers with upfront input on how users might expect the product to function. And since the test vocabulary we use is suitable for both manual and automated execution, designed testing can commence immediately once the application becomes available.

It is, perhaps, important to note that this does not suggest that these tests can be executed *automatically* as soon as the application becomes available. The test record in Table 2 may be perfectly understood and executable by a person, but the automation framework knows nothing

# Test Automation Frameworks

By Carl Nagle

about the objects in this record until we can provide that additional information. That is a separate piece of the framework we will learn about when we discuss application mapping.

## **Findings:**

The keyword driven automation framework is initially the hardest and most time-consuming data driven approach to implement. After all, we are trying to fully insulate our tests from both the many failings of the automation tools, as well as changes to the application itself.

To accomplish this, we are essentially writing enhancements to many of the component functions already provided by the automation tool: such as error correction, prevention, and enhanced synchronization.

Fortunately, this heavy, initial investment is mostly a one-shot deal. Once in place, keyword driven automation is arguably the easiest of the data driven frameworks to maintain and perpetuate providing the greatest potential for long-term success.

Additionally, there may now be commercial products suitable for your needs to decrease, but not eliminate, much of the up-front technical burden of implementing such a framework. This was not the case just a few years ago. We will briefly discuss a couple of these in [Section 1.2.4](#)

## **1.2.3 Hybrid Test Automation (or, "All of the Above")**

The most successful automation frameworks generally accommodate both keyword driven testing as well as data driven scripts. This allows data driven scripts to take advantage of the powerful libraries and utilities that usually accompany a keyword driven architecture.

The framework utilities can make the data driven scripts more compact and less prone to failure than they otherwise would have been. The utilities can also facilitate the gradual and manageable conversion of existing scripts to keyword driven equivalents when and where that appears desirable.

On the other hand, the framework can use scripts to perform some tasks that might be too difficult to re-implement in a pure keyword driven approach, or where the keyword driven capabilities are not yet in place.

## **1.2.4 Commercial Keyword Driven Frameworks**

Some commercially available keyword driven frameworks are making inroads in the test automation markets. These generally come from 3<sup>rd</sup> party companies as a bridge between your application and the automation tools you intend to deploy. They are not out-of-the-box, turnkey automation solutions just as the capture\replay tools are not turnkey solutions.

They still require some up-front investment of time and personnel to complete the bridge between the application and the automation tools, but they can give some automation departments and professionals a huge jumpstart in the right direction for successful long-term test automation.

# Test Automation Frameworks

By Carl Nagle

Two particular products to note are the TestFrame™ product led by Hans Buwalda of CMG Corp, and the Certify™ product developed with Linda Hayes of WorkSoft Inc. These products each implement their own version of a keyword driven framework and have served as models for the subject at international software testing conferences, training courses, and user-group discussions worldwide. I'm sure there are others.

It really is up to the individual enterprise to evaluate if any of the commercial solutions are suitable for their needs. This will be based not only on the capabilities of the tools evaluated, but also on how readily they can be modified and expanded to accommodate your current and projected capability requirements.

## 1.3 Keyword Driven Automation Framework Model

The following automation framework model is the result of over 18 months of planning, design, coding, and sometimes trial and error. That is not to say that it took 18 months to get it working--it was actually a working prototype at around 3 person-months. Specifically, one person working on it for 3 months!

The model focuses on implementing a keyword driven automation framework. It does not include any additional features like tracking requirements or providing traceability between automated test results and any other function of the test process. It merely provides a model for a keyword driven execution engine for automated tests.

The commercially available frameworks generally have many more features and much broader scope. Of course, they also have the price tag to reflect this.

### 1.3.1 Project Guidelines

The project was informally tasked to follow the guidelines or practices below:

- Implement a test strategy that will allow reasonably intuitive tests to be developed and executed both manually and via the automation framework.
- The test strategy will allow each test to include the step to perform, the input data to use, *and* the expected result all together in one line or record of the input source.
- Implement a framework that will integrate keyword driven testing and traditional scripts, allowing both to benefit from the implementation.
- Implement the framework to be completely application-independent since it will need to test at least 4 or 5 different applications once deployed.
- The framework will be fully documented and published.

# Test Automation Frameworks

By Carl Nagle

- The framework will be publicly shared on the intranet for others to use and eventually (hopefully) co-develop.

## 1.3.2 Code and Documentation Standards

The first thing we did was to define standards for source code files and headers that would provide for in-context documentation intended for publication. This included standards for how we would use headers and what type of information would go into them.

Each source file would start with a structured block of documentation describing the purpose of the module. Each function or subroutine would likewise have a leading documentation block describing the routine, its arguments, possible return codes, and any errors it might generate. Similar standards were developed for documenting the constants, variables, dependencies, and other features of the modules.

We then developed a tool that would extract and publish the documentation in HTML format directly from the source and header files. We did this to minimize synchronization problems between the source code and the documentation, and it has worked very well.

It is beyond the scope of this work to illustrate how this is done. In order to produce a single HTML document we parse the source file and that source file's primary headers. We format and link public declarations from the headers to the detailed documentation in the source as well as link to any external references for other documentation. We also format and group public constants, properties or variables, and user-defined types into the appropriate sections of the HTML publication.

One nice feature about this is that the HTML publishing tool is made to identify the appropriate documentation blocks and include them pretty much "as is". This enables the inclusion of HTML tags within the source documentation blocks that will be properly interpreted by a browser. Thus, for publication purposes, we can include images or other HTML elements by embedding the proper tags.

## 1.3.3 Our Automation Framework

Figure 4 is a diagram representing the design of our automation framework. It is followed by a description of each of the elements within the framework and how they interact. Some readers may recognize portions of this design. It is a compilation of keyword driven automation concepts from several sources. These include Linda Hayes with WorkSoft, Ed Kit from Software Development Technologies, Hans Buwalda from CMG Corp, myself, and a few others.

# Test Automation Frameworks

By Carl Nagle

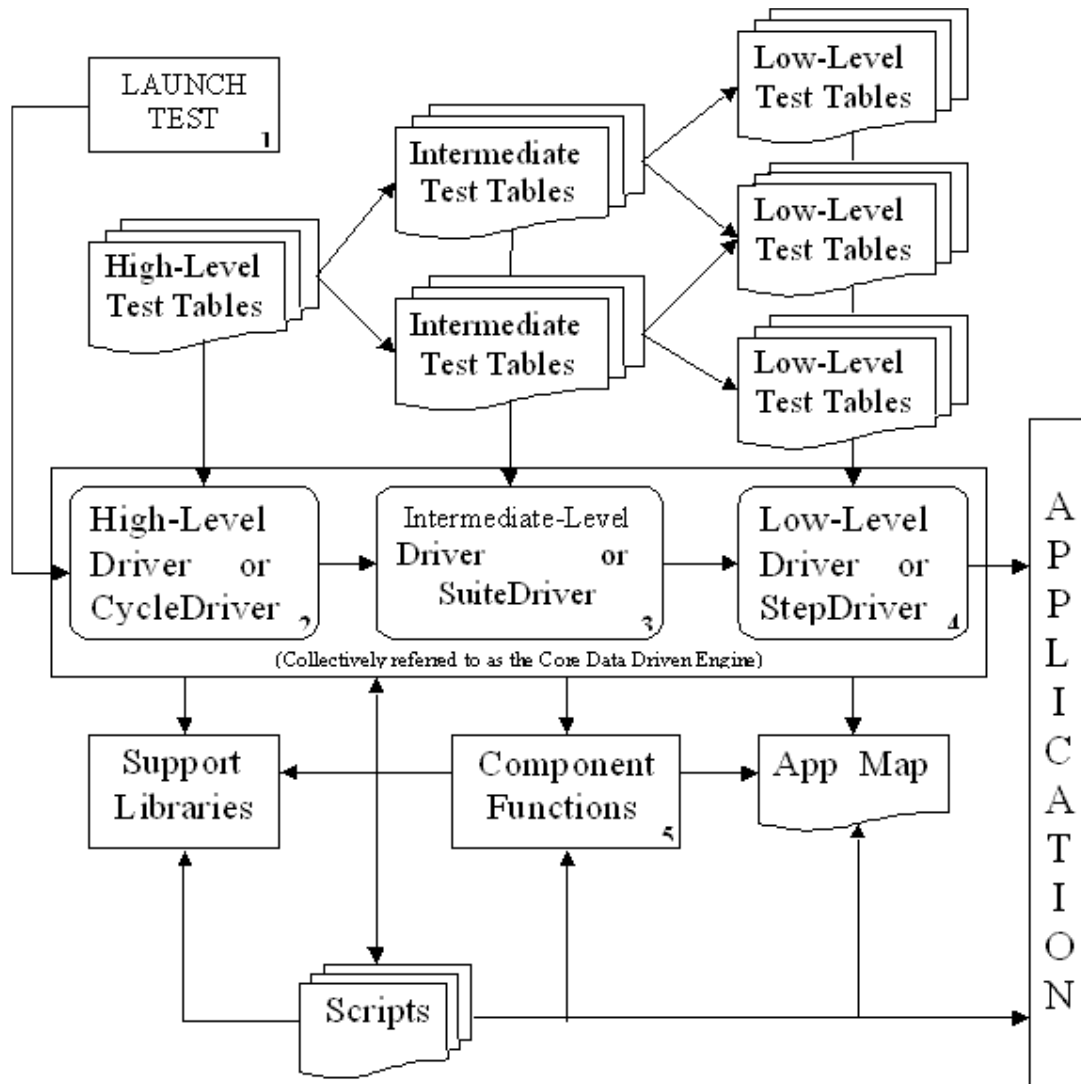


Figure 4

In brief, the framework itself is really defined by the *Core Data Driven Engine*, the *Component Functions*, and the *Support Libraries*. While the *Support Libraries* provide generic routines useful even outside the context of a keyword driven framework, the core engine and *Component Functions* are highly dependent on the existence of all three elements.

The test execution starts with the LAUNCH TEST(1) script. This script invokes the *Core Data Driven Engine* by providing one or more *High-Level Test Tables* to *CycleDriver*(2). *CycleDriver* processes these test tables invoking the *SuiteDriver*(3) for each *Intermediate-Level Test Table* it encounters. *SuiteDriver* processes these intermediate-level tables invoking *StepDriver*(4) for each *Low-Level Test Table* it encounters. As *StepDriver* processes these low-level tables it attempts to keep the application in synch with the test. When *StepDriver* encounters a low-level command for a

# Test Automation Frameworks

By Carl Nagle

App Map

specific component, it determines what Type of component is involved and invokes the corresponding *Component Function(5)* module to handle the task.

All of these elements rely on the information provided in the *App Map* to interface or bridge the automation framework with the application being tested. Each of these elements will be described in more detail in the following sections.

### **1.3.4 The Application Map:**

The *Application Map* is one of the most critical items in our framework. It is how we map our objects from names we humans can recognize to a data format useful for the automation tool. The testers for a given project will define a naming convention or specific names for each component in each window as well as a name for the window itself. We then use the *Application Map* to associate that name to the identification method needed by the automation tool to locate and properly manipulate the correct object in the window.

Not only does it give us the ability to provide useful names for our objects, it also enables our scripts and keyword driven tests to have a single point of maintenance on our object identification strings. Thus, if a new version of an application changes the title of our web page or the index of an image element within it, they should not affect our test tables. The changes will require only a quick modification in one place--inside the *Application Map*.

Figure 5 shows a simple HTML page used in one frame of a HTML frameset. Table 3 shows the object identification methods for this page for an automation tool. This illustrates how the tool's recorded scripts might identify multiple images in the header frame or top frame of a multi-frame web page. This top frame contains the HTML document with four images used to navigate the site. Notice that these identification methods are literal strings and potentially appear many times in traditional scripts (a maintenance nightmare!):



Figure 5

### **Script referencing HTML Document Components with Literal Strings**

OBJECT	IDENTIFICATION METHOD
--------	-----------------------

# Test Automation Frameworks

By Carl Nagle

Window	"WindowTag=WebBrowser"
Frame	"FrameID=top"
Image	"FrameID=top;\;DocumentTitle=topFrame"
Image	"FrameID=top;\;DocumentTitle=topFrame;\;ImageIndex=1"
Image	"FrameID=top;\;DocumentTitle=topFrame;\;ImageIndex=2"
Image	"FrameID=top;\;DocumentTitle=topFrame;\;ImageIndex=3"
Image	"FrameID=top;\;DocumentTitle=topFrame;\;ImageIndex=4"

**Table 3**

This particular web page is simple enough. It contains only four images. However, when we look at Table 3, how do we determine which image is for Product information, and which is for Services? We should not assume they are in any particular order based upon how they are presented visually. Consequently, someone trying to decipher or maintain scripts containing these identification strings can easily get confused.

An *Application Map* will give these elements useful names, and provide our single point of maintenance for the identification strings as shown in Table 4. The *Application Map* can be implemented in text files, spreadsheet tables, or your favorite database table format. The *Support Libraries* just have to be able to extract and cache the information for when it is needed.

<b>An Application Map Provides Named References for Components</b>	
<b>REFERENCE</b>	<b>IDENTIFICATION METHOD</b>
Browser	"WindowTag=WebBrowser"
TopFrame	"FrameID=top"
TopPage	"FrameID=top;\;DocumentTitle=topFrame"
CompInfoImage	"FrameID=top;\;DocumentTitle=topFrame;\;ImageIndex=1"
ProductsImage	"FrameID=top;\;DocumentTitle=topFrame;\;ImageIndex=2"
ServicesImage	"FrameID=top;\;DocumentTitle=topFrame;\;ImageIndex=3"

# Test Automation Frameworks

By Carl Nagle

## Component Functions

SiteMapImage	"FrameID=top;\;DocumentTitle=topFrame;\;ImageIndex=4"
--------------	---

Table 4

With the preceding definitions in place, the same scripts can use variables with values from the *Application Map* instead of those string literals. Our scripts can now reference these image elements as shown in Table 5. This reduces the chance of failure caused by changes in the application and provides a single point of maintenance in the *Application Map* for the identification strings used throughout our tests. It can also make our scripts easier to read and understand.

Script Using Variable References Instead of Literal Strings	
OBJECT	IDENTIFICATION METHOD
Window	Browser
Frame	TopFrame
Document	TopPage
Image	CompInfoImage
Image	ProductsImage
Image	ServicesImage
Image	SiteMapImage

Table 5

### **1.3.5 Component Functions:**

*Component Functions* are those functions that actively manipulate or interrogate component objects. In our automation framework we will have a different *Component Function* module for each Type of component we encounter (Window, CheckBox, TextBox, Image, Link, etc..).

Our *Component Function* modules are the application-independent extensions we apply to the functions already provided by the automation tool. However, unlike those provided by the tool, we

# Test Automation Frameworks

By Carl Nagle

add the extra code to help with error detection, error correction, and synchronization. We also write these modules to readily use our application-specific data stored in the *Application Map* and test tables as necessary. In this way, we only have to develop these *Component Functions* once, and they will be used again and again by every application we test.



Another benefit from *Component Functions* is that they provide a layer of insulation between our application and the automation tool. Without this extra layer, changes or "enhancements" in the automation tool itself can break existing scripts and our table driven tests. With these *Component Functions*, however, we can insert a "fix"--the code necessary to accommodate these changes that will avert breaking our tests.

## **Component Function Keywords Define Our Low-Level Vocabulary:**

Each of these *Component Function* modules will define the keywords or "action words" that are valid for the particular component type it handles. For example, the Textbox *Component Function* module would define and implement the actions or keywords that are valid on a Textbox. These keywords would describe actions like those in Table 6:

<b>Some Component Function Keywords for a Textbox</b>	
<b>KEYWORD</b>	<b>ACTION PERFORMED</b>
InputText	Enter new value into the Textbox
VerifyValue	Verify the current value of the Textbox
VerifyProperty	Verify some other attribute of the Textbox

**Table 6**

Each action embodied by a keyword may require more information in order to complete its function. The `InputText` action needs an additional argument that tells the function what text it is suppose to input. The `VerifyProperty` action needs two additional arguments, (1) the name of the property we want to verify, and (2) the value we expect to find. And, while we need no additional information to `Click` a `Pushbutton`, a `Click` action for an `Image` map needs to know where we want the click on the image to occur.

These *Component Function* keywords and their arguments define the low-level vocabulary and individual record formats we will use to develop our test tables. With this vocabulary and the *Application Map* object references, we can begin to build test tables that our automation framework and our human testers can understand and properly execute.

### **1.3.6 Test Tables:**

*Low-level Test Tables* or *Step Tables* contain the detailed step-by-step instructions of our tests.

# Test Automation Frameworks

By Carl Nagle

Using the object names found in the *Application Map*, and the vocabulary defined by the *Component Functions*; these tables specify what document, what component, and what action to take on the component. The following three tables are examples of *Step Tables* comprised of instructions to be processed by the *StepDriver* module. The *StepDriver* module is the one that initially parses and routes all low-level instructions that ultimately drive our application.

<b>Step Table: LaunchSite</b>		
<b>COMMAND/DOCUMENT</b>	<b>PARAMETER/ACTION</b>	<b>PARAMETER</b>
LaunchBrowser	Default.htm	
Browser	VerifyCaption	"Login"

Table 7

<b>Step Table: Login</b>			
<b>DOCUMENT</b>	<b>COMPONENT</b>	<b>ACTION</b>	<b>PARAMETER</b>
LoginPage	UserIDField	InputText	"MyUserID"
LoginPage	PasswordField	InputText	"MyPassword"
LoginPage	SubmitButton	Click	

Table 8

<b>Step Table: LogOffSite</b>		
<b>DOCUMENT</b>	<b>COMPONENT</b>	<b>ACTION</b>
TOCPage	LogOffButton	Click

Table 9

Note:

In Table 7 we used a System-Level keyword, `LaunchBrowser`. This is also called a *Driver Command*. A *Driver Command* is a command for the framework itself and not tied to any particular

# Test Automation Frameworks

By Carl Nagle

document or component. Also notice that test tables often consist of records with varying numbers of fields.

*Intermediate-Level Test Tables* or *Suite Tables* do not normally contain such low-level instructions. Instead, these tables typically combine *Step Tables* into *Suites* in order to perform more useful tasks. The same *Step Tables* may be used in many *Suites*. In this way we only develop the minimum number of *Step Tables* necessary. We then mix-and-match them in *Suites* according to the purpose and design of our tests, for maximum reusability.

The *Suite Tables* are handled by the *SuiteDriver* module which passes each *Step Table* to the *StepDriver* module for processing.

For example, a *Suite* using two of the preceding *Step Tables* might look like Table 10:

<b>Suite Table: StartSite</b>	
<b>STEP TABLE REFERENCE</b>	<b>TABLE PURPOSE</b>
LaunchSite	Launch web app for test
Login	Login "MyUserID" to app

Table 10

Other *Suites* might combine other *Step Tables* like these:

<b>Suite Table: VerifyTOCPage</b>	
<b>STEP TABLE REFERENCE</b>	<b>TABLE PURPOSE</b>
VerifyTOCContent	Verify text in Table of Contents
VerifyTOCLinks	Verify links in Table of Contents

Table 11

<b>Suite Table: ShutdownSite</b>	
<b>STEP TABLE REFERENCE</b>	<b>TABLE PURPOSE</b>

# Test Automation Frameworks

By Carl Nagle

LogOffSite	Logoff the application
ExitBrowser	Close the web browser

Table 12

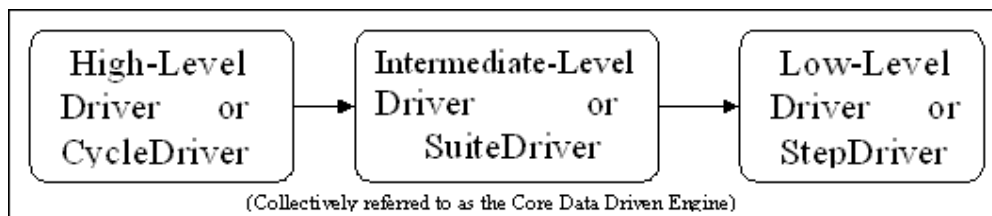
*High-Level Test Tables* or *Cycle Tables* combine intermediate-level *Suites* into *Cycles*. The *Suites* can be combined in different ways depending upon the testing *Cycle* we wish to execute (Regression, Acceptance, Performance...). Each *Cycle* will likely specify a different type or number of tests. These *Cycles* are handled by the *CycleDriver* module which passes each *Suite* to *SuiteDriver* for processing.

A simple example of a *Cycle Table* using the full set of tables seen thus far is shown in Table 13.

Cycle Table: SiteRegression	
SUITE TABLE REFERENCE	TABLE PURPOSE
StartSite	Launch browser and Login for test
VerifyTOCPage	Verify Table of Contents page
ShutdownSite	Logoff and Shutdown browser

Table 13

## 1.3.7 The Core Data Driven Engine:



We have already talked about the primary purpose of each of the three modules that make up the *Core Data Driven Engine* part of the automation framework. But let us reiterate some of that here.

*CycleDriver* processes *Cycles*, which are high-level tables listing *Suites* of tests to execute. *CycleDriver* reads each record from the *Cycle Table*, passing *SuiteDriver* each *Suite Table* it finds during this process.

# Test Automation Frameworks

By Carl Nagle

*SuiteDriver* processes these *Suites*, which are intermediate-level tables listing *Step Tables* to execute. *SuiteDriver* reads each record from the *Suite Table*, passing *StepDriver* each *Step Table* it finds during this process.

*StepDriver* processes these *Step Tables*, which are records of low-level instructions developed in the keyword vocabulary of our *Component Functions*. *StepDriver* parses these records and performs some initial error detection, correction, and synchronization making certain that the document and/or the component we plan to manipulate are available and active. *StepDriver* then routes the complete instruction record to the appropriate *Component Function* for final execution.

Let us again show a sample *Step Table* record in Table 14 and the overall framework pseudo-code that will process it in Figure 6.

<b>Single Step Table Record</b>			
<b>DOCUMENT</b>	<b>COMPONENT</b>	<b>ACTION</b>	<b>EXPECTED VALUE</b>
LoginPage	UserIDTextbox	VerifyValue	"MyUserID"

Table 14

<b>Framework Pseudo-Code</b>
<p><b>Primary Record Processor Module:</b></p> <pre>Verify "LoginPage" Exists. (Attempt recovery if not) Set focus to "LoginPage". Verify "UserIDTextbox" Exists. (Attempt recovery if not) Find "Type" of component "UserIDTextbox". (It is a Textbox) Call the module that processes ALL Textbox components.</pre>
<p><b>Textbox Component Module:</b></p> <pre>Validate the action keyword "VerifyValue". Call the Textbox.VerifyValue function.</pre>
<p><b>Textbox.VerifyValue Function:</b></p> <pre>Get the text stored in the "UserIDTextbox" Textbox. Compare the retrieved text to "MyUserID". Record our success or failure.</pre>

Figure 6

# Test Automation Frameworks

By Carl Nagle

In Figure 6 you may notice that it is not *StepDriver* that validates the action command or keyword *VerifyValue* that will be processed by the Textbox *Component Function* Module. This allows the Textbox module to be further developed and expanded without affecting *StepDriver* or any other part of the framework. While there are other schemes that might allow *StepDriver* to effectively make this validation dynamically at runtime, we still chose to do this in the *Component Functions* themselves.

## **System-Level Commands or Driver Commands:**

In addition to this table-processing role, each of the Driver modules has its own set of System-Level keywords or commands also called *Driver Commands*. These commands instruct the module to do something other than normal table processing.

For example, the previously shown *LaunchSite Step Table* issued the *LaunchBrowser StepDriver* command. This command instructs *StepDriver* to start a new Browser window with the given URL. Some common *Driver Commands* to consider:

<b>Common Driver Commands</b>	
<b>COMMAND</b>	<b>PURPOSE</b>
UseApplicationMap	Set which Application Map(s) to use
LaunchApplication	Launch a standard application
LaunchBrowser	Launch a web-based app via URL
CallScript	Run an automated tool script
WaitForWindow	Wait for Window or Browser to appear
WaitForWindowGone	Wait for Window or Browser to disappear
Pause or Sleep	Pause for a specified amount of time
SetRecoveryProcess	Set an AUT recovery process
SetShutdownProcess	Set an AUT shutdown process
SetRestartProcess	Set an AUT restart process
SetRebootProcess	Set a System Reboot process
LogMessage	Enter a message in the log

**Table 15**

# Test Automation Frameworks

By Carl Nagle

These are just some examples of possible *Driver Commands*. There are surely more that have not been listed and some here which you may not need implemented.

## **All for One, and One for All:**

This discussion on the *Core Data Driven Engine* has identified three separate modules (*StepDriver*, *SuiteDriver*, and *CycleDriver*) that comprise the Core. This does not, however, make this three-module design an automation framework requirement. It may be just as valid to sum up all the functionality of those three modules into one module, or any number of discreet modules more suitable to the framework design developed.

## **1.3.8 The Support Libraries:**

The *Support Libraries* are the general-purpose routines and utilities that let the overall automation framework do what it needs to do. They are the modules that provide things like:

- File Handling
- String Handling
- Buffer Handling
- Variable Handling
- Database Access
- Logging Utilities
- System\Environment Handling
- Application Mapping Functions
- System Messaging or System API Enhancements and Wrappers

They also provide traditional automation tool scripts access to the features of our automation framework including the *Application Map* functions and the keyword driven engine itself. Both of these items can vastly improve the reliability and robustness of these scripts until such time that they can be converted over to keyword driven test tables (if and when that is desirable).

## **1.4 Automation Framework Workflow**

We have seen the primary features of our automation framework and now we want to put it to the test. This section provides a test workflow model that works very well with this framework. Essentially, we start by defining our high level *Cycle Tables* and provide more and more detail down to our *Application Map* and low-level *Step Tables*.

# Test Automation Frameworks

By Carl Nagle

For this workflow example, we are going to show the hypothetical test design that might make up security authentication tests for a web site. The order in which we present the information and construct the tables is an ideal workflow for our automation framework. It will also illustrate how we do not even need a functioning application in order to start producing these tests.

## 1.4.1 High-Level Tests -- *Cycle Table*

We will start out by defining our high-level *Cycle Table* in Table 16. This will list tests that verify user authentication functionality. There is no application yet, but we do know that we will authenticate users with a user ID and password via some type of Login page. With this much information we can start designing some tests.

At this level, our tables are merely keywords or actions words of what we propose to do. The keywords represent the names of *Suites* we expect to implement when we are ready or when more is known about the application.

We will call this particular high-level test, `VerifyAuthenticationFunction` (Table 16). It will not show all the tests that should be performed to verify the user authentication features of a web site. Just enough to illustrate our test design process.

<b>Cycle Table: VerifyAuthenticationFunction</b>	
<b>KEYWORDS (Suite Tables)</b>	<b>TABLE PURPOSE</b>
<code>VerifyInvalidLogin</code>	Tests with Invalid UserID and/or Password
<code>VerifyBlankLogin</code>	Tests with Missing UserID and/or Password
<code>VerifyValidLogin</code>	Tests with Valid UserID and Password

**Table 16**

The preceding table illustrates that we plan to run three tests to verify the authentication functionality of our application. We may add or delete tests from this list in the future, but this is how we currently believe this functionality can be adequately verified.

## 1.4.2 Intermediate-Level Tests -- *Suite Tables*

# Test Automation Frameworks

By Carl Nagle

When we are ready to provide more detail for our authentication test effort we can begin to flesh-out the intermediate-level *Suite Tables*. Our high-level `VerifyAuthenticationFunction` *Cycle Table* defined three *Suites* that we must now expand upon.

First, we plan to verify invalid login attempts--those with an invalid user ID, invalid password, or both. This first *Suite* was aptly dubbed, `VerifyInvalidLogin` (Table 17).

Second, we want to run a simple test without any user ID or password whatsoever. This second *Suite* was called, `VerifyBlankLogin` (Table 18).

The last *Suite* we must implement is the one that insures we can successfully login when we provide a valid user ID and password. This one was named, `VerifyValidLogin` (Table 19).

<b>Suite Table: VerifyInvalidLogin</b>		
<b>KEYWORDS (Step Tables)</b>	<b>USERID</b>	<b>PASSWORD</b>
LaunchSite		
Login	BadUser	GoodPassword
VerifyLoginError		
Login	GoodUser	BadPassword
VerifyLoginError		
Login	BadUser	BadPassword
VerifyLoginError		
ExitLogin		

Table 17

<b>Suite Table: VerifyBlankLogin</b>		
<b>KEYWORDS (Step Tables)</b>	<b>USERID</b>	<b>PASSWORD</b>
LaunchSite		
Login	""	""
VerifyLoginError		

# Test Automation Frameworks

By Carl Nagle

ExitLogin		
-----------	--	--

Table 18

Suite Table: VerifyValidLogin		
KEYWORDS (Step Tables)	USERID	PASSWORD
LaunchSite		
Login	GoodUser	GoodPassword
VerifyLoginSuccess		
ShutdownSite		

Table 19

Notice that we start to see a large amount of test table reuse here. While not yet defined, we can see that we will be using the `LaunchSite`, `Login`, `VerifyLoginError`, and `ExitLogin` tables quite frequently. As you can probably imagine, `VerifyLoginSuccess` and `ShutdownSite` would also be used extensively throughout all the tests for this web site, but we are focusing just on user authentication here.

You should also notice that our `Login` keyword is describing a *Step Table* that will accept up to two arguments--the user ID and password we want to supply. We have not gone into the specific design details of how the automation framework should be implemented, but tests accepting variable data like this provide significant reuse advantages. This same table is used to login with different user ID and password combinations.

## 1.4.3 Application Maps

Up to this point there was little need for an *Application Map*. Nearly all the high-level and intermediate-level test designs are abstract enough to forego references to specific window components. But as we move towards our final level of test design we indeed must reference specific elements of the application. Thus, for test automation, we need to make our *Application Maps*.

### Naming the Components:

Unless the application has formal design documentation that explicitly provides meaningful names for the windows and components, the test team is going to have to develop this themselves.

# Test Automation Frameworks

By Carl Nagle

The team will discuss and agree upon the names for each window and component. Sometimes we can use the names developers have already given the components in the code. But the team can often come up with more meaningful and identifiable names. Regardless, once defined, the names chosen should not change. However, if the need arises, an item can always have more than one name.

## **Test Design Application Map:**

For low-level test design and manual testing, an *Application Map* suitable for human consumption should be created. This is often done with screen captures of the various application windows. These screen captures are then clearly annotated with the names given to each component. This may also indicate the Type for each component.

While many components may be exactly what they look like, this is not always the case. Many controls may simply be windowless, painted elements that the underlying environment knows nothing about. They can also be custom components or COM objects that are functionally different from the controls they appear to emulate. So care must be taken when identifying the Type of a component.

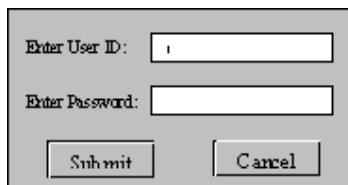
## **Automation Framework Application Map:**

For automated testing, an *Application Map* suitable for the automation framework will be created. We will use the *Test Design Application Map* as our reference for the names used to identify the components. The identification syntax suitable for the automation tool will then be mapped to each name as discussed in [Section 1.3.4](#).

Our test design so far has indirectly identified three documents or windows that we will interact with: (1) the Login page, (2) an Error dialog or message displayed during our failed Login attempts, and (3) the Home page of our web site displayed following a successful login attempt. We will also need to reference the browser window itself throughout these tests.

We must now create an *Application Map* that will properly reference these items and their components for the automation framework. For the sake of simplicity, our Login page in Figure 7 will contain only the two textbox components for user ID and password, and two buttons; Submit and Cancel.

## **Login Page**



The image shows a screenshot of a login form. It has a light gray background. At the top left, there is a label "Enter User ID:" followed by a white rectangular text box. Below that, there is a label "Enter Password:" followed by another white rectangular text box. At the bottom left, there is a button labeled "Submit". At the bottom right, there is a button labeled "Cancel".

Figure 7

# Test Automation Frameworks

By Carl Nagle

The Error dialog in Figure 8 is actually a system alert dialog. It has a Label containing the error message and an OK button to dismiss the dialog.

## Error Dialog

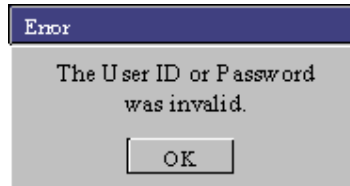


Figure 8

The Home Page of the application may contain many different elements. However, for the purpose of these tests and this example, we only need to identify the document itself. We will not interact with any of the components in the Home Page, so we will not define them in this sample *Application Map*.

Table 20 shows the *Application Map* we will use to identify the browser window and the three documents in a manner suitable for the automation framework.

Site Application Map	
OBJECTNAME	IDENTIFICATION METHOD
Browser:	
Browser	WindowID=WebBrowser
LoginPage:	
LoginPage	FrameID=content;\;DocumentTitle=Login
UserID	FrameID=content;\;DocumentTitle=Login;\;TextBoxIndex=1
Password	FrameID=content;\;DocumentTitle=Login;\;TextBoxIndex=2
SubmitButton	FrameID=content;\;DocumentTitle=Login;\;ButtonText=Submit
CancelButton	FrameID=content;\;DocumentTitle=Login;\;ButtonText=Cancel

# Test Automation Frameworks

By Carl Nagle

ErrorWin:	
ErrorWin	WindowCaption=Error
ErrorMessage	LabelIndex=1
OKButton	ButtonText=OK
HomePage:	
HomePage	FrameID=content;\;DocumentTitle=Home

**Table 20**

The physical format of the *Application Map* will be determined by how the automation framework will access it. It may be delimited files, database tables, spreadsheets, or any number of other possibilities.

Notice that we have identified each document with its own section in the *Application Map* of Table 20. We do this to avoid naming conflicts among the many components of our application. For instance, there may be many forms that contain a `Submit` button. Each form can name this button the `SubmitButton` because the names only have to be unique within each individual section of the *Application Map*.

With this we have now uniquely identified our components for the framework. The `OBJECTNAME` is how we reference each component in our low-level *Step Tables*. The `IDENTIFICATION METHOD` is the information the automation tool needs to identify the component within the computer operating system environment.

The syntax for these component identifications is different depending upon the automation tool deployed. The example syntax we used is not for any particular automation tool, but is representative of what is needed for automation tools in general. The *Application Map* designer will need a thorough understanding of the automation tool's component identification syntax in order to create effective *Application Maps*.

## **New Component Types:**

As we examine the application and develop an *Automation Framework Application Map* we may find new component types not handled by existing *Component Functions*. To deal with this, we need to either implement a new *Component Function* module for each new component type, or

# Test Automation Frameworks

By Carl Nagle

determine how best they can be handled by existing *Component Functions*. This, of course, is a job for the automation technical staff.

## 1.4.4 Low-Level Tests -- Steps Tables

Now that the design of our application is known and our *Application Map* has been developed, we can get down to the step-by-step detail of our tests. We can still develop *Step Tables* without an *Application Map*. But without the map, we cannot automatically execute the tests.

Looking back at all the *Suites* that detailed our high-level `VerifyAuthenticationFunction` *Cycle Table*, we see they invoke six reusable, low-level *Step Tables* that we are now ready to implement. These are listed in Table 21.

Step Tables Invoked by Suites	
STEP TABLE	TABLE PURPOSE
LaunchSite	Launch the web app for testing
Login	Login with a User ID and Password
VerifyLoginError	Verify an Error Dialog on Invalid Login
ExitLogin	Exit the Login Page via Cancel
VerifyLoginSuccess	Verify Home Page up after Successful Login
ShutdownSite	Logoff and close the browser

Table 21

We develop these *Step Tables* using the component names defined in our *Application Map* and the keyword vocabulary defined by our *Component Functions*. Where test designers find missing yet needed keywords, actions, or commands in our *Component Functions*; the technical staff will need to expand the appropriate *Component Function* module, *Driver Commands*, *Support Libraries*, or find some other solution to provide the needed functionality if possible.

Now we can provide the detailed test steps that make-up our six *Step Tables*.

### Step Table: LaunchSite

# Test Automation Frameworks

By Carl Nagle

COMMAND/DOCUMENT	ARG/COMPONENT	ACTION	EXPECTED RESULTS
LaunchBrowser	Default.htm		
Browser		VerifyCaption	"Login"

Table 22

Step Table: Login &user &password			
DOCUMENT	COMPONENT	ACTION	INPUT DATA
LoginPage	UserID	InputText	&user
LoginPage	Password	InputText	&password
LoginPage	SubmitButton	Click	

Table 23

Step Table: VerifyLoginError			
DOCUMENT	COMPONENT	ACTION	EXPECTED RESULT
ErrorWin		VerifyCaption	"Error"
ErrorWin	ErrorMessage	VerifyText	"The User ID or Password was invalid."
ErrorWin	OKButton	Click	

Table 24

Step Table: ExitLogin		
DOCUMENT	COMPONENT	ACTION

# Test Automation Frameworks

By Carl Nagle

LoginPage	CancelButton	Click
-----------	--------------	-------

Table 25

Step Table: VerifyLoginSuccess			
DOCUMENT	COMPONENT	ACTION	EXPECTED RESULT
HomePage		VerifyTitle	"Home "
Browser		VerifyCaption	"Welcome "

Table 26

Step Table: ShutdownSite			
DOCUMENT	COMPONENT	ACTION	INPUT DATA
Browser		SelectMenuItem	File>Exit

Table 27

As you can see, our high-level test `VerifyAuthenticationFunction` has been detailed down to a very sparse set of highly reusable test steps. Our test designs have taken a high-level test specification, given that some abstract scenarios to perform, and finally provided the step-by-step instructions to execute those scenarios. We did all of this without resorting to difficult scripting languages, or complex software development environments.

It may seem simple enough, but that is because we have the underlying automation framework doing so much of the work for us.

Now that we have all of this defined, what does it actually do?

Remember, the automation framework is invoked with one or more high-level *Cycle Tables*. These tables call the test *Suites* that, in turn, call the *Step Tables* to perform the low-level test steps that drive the application.

We developed our own *Cycle Table* called `VerifyAuthenticationFunction`. This calls our own collection of *Suites*, and ultimately the *Step Tables* we just finished developing. If you look

# Test Automation Frameworks

By Carl Nagle

over all of these tables, you can see they form a relatively small set of reusable instructions that make up this test. But they generate a large instruction set at time of execution.

The following section will actually illustrate the automated execution flow of the test we have just implemented.

Table 28

<b>Execution Flow of VerifyAuthenticationFunction Test</b>				
<b>Cycle Table</b>	<b>DOCUMENT</b>	<b>COMPONENT</b>		<b>INPUT DATA</b>
<b>Suite Table</b>	<b>Or</b>	<b>Or</b>	<b>ACTION</b>	<b>Or</b>
<b>Step Table</b>	<b>COMMAND</b>	<b>COMMAND ARG</b>		<b>EXPECTED RESULTS</b>
VerifyAuthenticationFunction				
VerifyInvalidLogin				
LaunchSite				
	LaunchBrowser	Default.htm		
	Browser		VerifyCaption	"Login"
Login	&user=BadUser	&password=GoodPassword		
	LoginPage	UserID	InputText	&user
	LoginPage	Password	InputText	&password
	LoginPage	SubmitButton	Click	
VerifyLoginError				
	ErrorWin		VerifyCaption	"Error"

# Test Automation Frameworks

By Carl Nagle

	ErrorWin	ErrorMessage	VerifyText	"The User ID or Password was Invalid."
	ErrorWin	OKButton	Click	
Login	&user=GoodUser	&password=BadPassword		
	LoginPage	UserID	InputText	&user
	LoginPage	Password	InputText	&password
	LoginPage	SubmitButton	Click	
VerifyLoginError				
	ErrorWin		VerifyCaption	"Error"
	ErrorWin	ErrorMessage	VerifyText	"The User ID or Password was Invalid."
	ErrorWin	OKButton	Click	
Login	&user=BadUser	&password=BadPassword		
	LoginPage	UserID	InputText	&user
	LoginPage	Password	InputText	&password
	LoginPage	SubmitButton	Click	
VerifyLoginError				
	ErrorWin		VerifyCaption	"Error"
	ErrorWin	ErrorMessage	VerifyText	"The User ID or Password was Invalid."

# Test Automation Frameworks

By Carl Nagle

	ErrorWin	OKButton	Click	
ExitLogin				
	LoginPage	CancelButton	Click	
VerifyBlankLogin				
LaunchSite				
	LaunchBrowser	Default.htm		
	Browser		VerifyCaption	"Login"
Login	&user=""	&password=""		
	LoginPage	UserID	InputText	&user
	LoginPage	Password	InputText	&password
	LoginPage	SubmitButton	Click	
VerifyLoginError				
	ErrorWin		VerifyCaption	"Error"
	ErrorWin	ErrorMessage	VerifyText	"The User ID or Password was Invalid."
	ErrorWin	OKButton	Click	
ExitLogin				
	LoginPage	CancelButton	Click	
VerifyValidLogin				

# Test Automation Frameworks

By Carl Nagle

LaunchSite				
	LaunchBrowser	Default.htm		
	Browser		VerifyCaption	"Login"
Login	&user=GoodUser	&password=GoodPassword		
	LoginPage	UserID	InputText	&user
	LoginPage	Password	InputText	&password
	LoginPage	SubmitButton	Click	
VerifyLoginSuccess				
	HomePage		VerifyTitle	"Home"
	Browser		VerifyCaption	"Welcome"
ShutdownSite				
	Browser		SelectMenuItem	File>Exit

Table 28

## 1.5 Summary

In order to keep up with the pace of product development and delivery it is essential to implement an effective, reusable test automation framework.

We cannot expect the traditional capture/replay framework to fill this role for us. Past experience has shown that capture\replay tools alone will never provide the long-term automation successes that other more robust test automation strategies can.

A test strategy relying on data driven automation tool scripts is definitely the easiest and quickest to implement if you have and keep the technical staff to handle it. But it is the hardest of the data driven approaches to maintain and perpetuate and often leads to long-term failure.

The most effective test strategies allow us to develop our structured test designs in a format and vocabulary suitable for both manual *and* automated testing. This will enable testers to focus on effective test designs unencumbered by the technical details of the framework used to execute them,

# Test Automation Frameworks

By Carl Nagle

while technical automation experts implement and maintain a reusable automation framework independent of any application that will be tested by it.

A keyword driven automation framework is probably the hardest and potentially most time-consuming data driven approach to implement *initially*. However, this investment is mostly a one-shot deal. Once in place, keyword driven automation is arguably the easiest of the data driven frameworks to maintain and perpetuate providing the greatest potential for long-term success. There are also a few commercially available products maturing that may be suitable for your needs.

What we really want is a framework that can be both keyword driven while also providing enhanced functionality for data driven scripts. When we integrate these two approaches the results can be *very* impressive!

The essential guiding principles we should follow when developing our overall test strategy (or evaluating the test strategy of a tool we wish to consider):

- The test design and the test framework are totally separate entities.
- The test framework should be application-independent.
- The test framework must be easy to expand, maintain, and perpetuate.
- The test strategy/design vocabulary should be framework independent.
- The test strategy/design should isolate testers from the complexities of the test framework.

## **Bibliography:**

1. Kit, E. & Prince, S. "A Roadmap for Automating Software Testing" Tutorial presented at STAR'99East Conference, Orlando, Florida, May 10, 1999.
2. Hayes, L. "Establishing an Automated Testing Framework" Tutorial presented at STAR'99East Conference, Orlando, Florida, May 11, 1999.
3. Kit, E. "The Third Generation--Integrated Test Design and Automation" Guest presentation at STAR'99East Conference, Orlando, Florida, May 12, 1999.
4. Mosley, D. & Posey, B. *Just Enough Software Test Automation* New Jersey: Prentice Hall PTR, 2002.
5. Wust, G. "A Model for Successful Software Testing Automation" Paper presented at STAR'99East Conference, Orlando, Florida, May 12, 1999.
6. Dustin, E. *Automated Software Testing: Introduction, Management, and Performance*. New York: Addison Wesley, 1999.
7. Fewster & Graham *Software Test Automation: Effective use of test execution tools* New York: Addison Wesley, 1999.
8. Dustin, E. "Automated Testing Lifecycle Methodology (ATLM)" Paper presented at STAR EAST 2000 Conference, Orlando, Florida, May 3, 2000.
9. Kit, E. & Buwalda, H. "Testing and Test Automation: Establishing Effective Architectures" Presentation at STAR EAST 2000 Conference, Orlando, Florida, May 4, 2000.
10. Sweeney, M. "Automation Testing Using Visual Basic" Paper presented at STAR EAST 2000 Conference, Orlando, Florida, May 4, 2000.
11. Buwalda, H. "Soap Opera Testing" Guest presentation at STAR EAST 2000 Conference, Orlando, Florida, May 5, 2000.

# Test Automation Frameworks

By Carl Nagle

12. Pollner, A. "Advanced Techniques in Test Automation" Paper presented at STAR EAST 2000 Conference, Orlando, Florida, May 5, 2000.
13. Cem Kaner, <http://www.kaner.com>
14. Zambelich, K. *Totally Data-Driven Automated Testing* 1998  
[http://www.sqa-test.com/w\\_paper1.html](http://www.sqa-test.com/w_paper1.html)
15. SQA Suite Users, Discussions and Archives, 1999-2000, <http://www.dundee.net/sqa/>
16. Nagle, C. *Data Driven Test Automation: For Rational Robot V2000* 1999-2000  
[DDE Doc Index](#)