



## TAC Automation Architecture Theory

### Overview

High quality test automation does not happen by accident. It is the result of intense planning, concentrated design and solid software engineering. The automation architecture can be seen as the culmination of the planning and design phases.

There are several competing definitions of software architecture that have been advanced. Cem Kaner's definition is my favorite:

**The architecture of a software product specifies the overall technical strategy for the product's development. For example the architecture:**

- **Provides a strategy for subdividing the overall product into a set of components.**
- **Determines the means by which one component can pass data or a request for service to another.**
- **Specifies the relationship between code and data...**

**...decisions about the architecture of the software (in this case, the test automation application) are fundamental to the rest of your design and implementation of the program.<sup>1</sup>**

There are several concerns that we want to address with our architecture choice:

1. Effectiveness: ensuring that every individual test case actually performs the test it is designed to and verifies the results correctly.
2. Survivability: ensuring that a non-trivial number of test cases in a suite can run unattended end-to-end.
3. Scalability: ensuring that the maintenance of the test suite as it grows does not overwhelm the automation group or cost an inordinate amount.

### Competing Architecture Schemes

There are currently several commonly used automation architectures. The following list is certainly not exhaustive, but is representative of most projects using GUI automation tools:

---

<sup>1</sup> Drawn from the paper, "Avoiding Shelfware: A Manager's View of Automated GUI Testing". Cem Kaner. Keynote address, STAR '98 (SEVENTH ANNUAL SOFTWARE TESTING ANALYSIS & REVIEW), Orlando, May 6, 1998. The definitions of architecture types are drawn from this paper, and other discussions with Cem Kaner and other attendees of the Austin Workshop on Test Automation.



Innovative solutions to maximize your testing investment.

- Record/Playback (quick and dirty)
- Framework-based
- Data-driven
- Table-driven

There is, unfortunately, no definite nomenclature in automation. The names listed above are commonly used, but certainly not accepted by all automation professionals. For the purposes of this document, we will use them; each will be described below.

It must be understood that the architectures described below do not represent single, monolithic entities. Instead, they should be seen as broad ranges on a continuum. For example, there is no single defining separation line between a Framework-based and Data-driven architecture. Many features are common to several architectures. The following descriptions should be seen as generalities used for descriptive purposes.

### **Record/Playback (Quick and Dirty)**

The basic automation architecture which many organizations start with is called Record / Playback (R/P.) In reality, this architecture is actually a lack of architecture; each test case consists of an individual script which is recorded using the automation tool. There are no allowances for fault tolerance or error recovery beyond what the tool supplies.

The tool is placed in recording mode whereby each interaction of a tester with the system under test (SUT) is recorded. When the tester clicks a mouse button, a comparable software function is placed into a text file called a script. When keystrokes are entered in an edit box, a list item selected, a button clicked: each action is captured and a comparable function is placed into the script. When the test is completed, the tool's record mode is shut off. At this point, a text file exists which has coded into it a model of the exact actions performed by the tester.

By compiling and executing this file, the exact actions of the tester may be recreated. If the application is set to the correct starting point, and everything in the environment is the same, replaying of the script should result in the same test being repeated. At any rate, that is the theory.

The problem, of course, is that everything is *never* the same. The environment may have changed in thousands of subtle ways. The application has likely changed (why else would it be under test?) Any relevant change in the environment or the application may result in the replayed script doing the wrong thing (or possibly the right thing at the wrong time) and causing a test failure.

If the test organization were only responsible for a single test, this would not be a major problem. Re-record the test and you are back in business. Unfortunately, most organizations need more than one test. Due to the high initial costs of automation, the organization likely needs hundreds or thousands of test cases to justify the automation



Innovative solutions to maximize your testing investment.

startup expenses. In this kind of an environment, the time and effort to maintain all of the tests rapidly becomes an impossible investment. And the automation project either evolves to a more complex architecture or is halted.

Analysis of typical failures highlights the problems with R/P. Each script depends on statements that have a one-to-one correspondence with the user interface. Type into this edit box or click that button. Any change to a control anywhere in the system can cause the failure of every script using that control. Any change in system logic or application flow can break every recorded script which goes through that changed area. Of all the possible architectures, R/P is by far the most expensive over the lifetime of the software, mainly due to the maintenance costs.

The conceptual problem of one-to-one correspondence between scripts and GUI operations was solved many years ago in the field of software engineering; **functional decomposition** was developed to avoid the “do the same thing repeatedly” dilemma. Write a single block of code that performs the task and then invoke it whenever the task needs doing. When the task changes, fix the code only in that single block. That design model is the basis of the next automation architecture.

### **Framework Based Architecture**

Most business software consists of tasks that are repeated using different data (or in different context.) If we can isolate a particular task in our system and record or program that task into a callable sub-routine, then we can utilize it in our test scripts every time we need to perform that task during a test.

Of course, by simply putting the code into a sub-routine, we are now in the business of programming. An evaluation of R/P class tools reveals, at their heart, they are basically programming languages. This is a fundamental tenet of successful test automation: **automation is software development and must be treated as such!**

Now that we have a sub-routine that performs a particular task in our system, we can record our scripts as before. By replacing any recorded code which performs the task with our sub-routine, we have taken a giant step. If something changes in the way this task is performed, or a control it uses changes, we have a single point of repair – the code inside the sub-routine. The more change we are likely to see, the more functional decomposition will help us control change.

When we run functional decomposition based scripts after system changes, we will still have the same number of failures we would have had with straight recording. Every script executing the logic containing the changed task will fail. The big difference is that we can repair a single section of code (our sub-routine) to fix all the broken scripts at once. We now have scalability. All other things being equal, we have the same amount of maintenance with 1000 scripts as we do 2000 or 5000.



Innovative solutions to maximize your testing investment.

Obviously, one sub-routine is not sufficient for our needs. We perform many different tasks in the standard business software package; perhaps thousands. The automation project can quickly become a full-scale software development project. At this point, the organization must determine the worth of automation; like any other testing effort, risk must be balanced against cost.

Some business software does not really break down easily into individual tasks. Instead, it is much easier to organize the automation project using a screen-based approach. Then the logical unit of an automation sub-routine becomes a single screen rather than an individual task. The routine *knows* how to deal with a particular screen full of controls. It can populate the screen by using any of the controls based on data input. This approach is especially useful in web type automation.

The framework-based architecture is fundamentally different from the R/P architecture in that it no longer simply depends on a business analyst creating a script by performing actions; it requires both the test logic of the analyst synthesized with the programming acumen of a developer.

The framework developed by the automator(s) will usually contain decomposed system tasks and any number of other routines performing administrative, logistical, and other support tasks.

In this architecture, creating test scripts changes from a recording task to a scripting process. A script may still contain some recording for those tasks only done occasionally. It may also contain recording whereby the testcase logic is captured, and the individual recorded tasks are replaced with the programmed code.

However the test case is created, the automator has far more execution control through the code; synchronization and fault tolerance can be added to prevent some execution errors, and control those errors which do occur. A framework-based test suite will have a much better chance of meeting the objectives we want from our architecture: effectiveness, survivability, and scalability.

### **Data-Driven Architecture**

The business tasks that are scripted in the framework-based architecture often consist of doing the same thing over and over again, using only different data. By moving away from recording, we suddenly have a large programming effort to support the automation. It makes good sense to try to reduce the amount of code we need to generate. One way of doing this is to analyze and filter out commonalities in tasks, then write code to perform the generalized set of tasks. Then repeat that code using different data each time through.

At a lower level, that is exactly what we have done by writing our functional decomposition sub-routines. We generalized each task and then made it specific by passing in different data for each function invocation.



Data-driven architecture is a logical extension to framework-based architecture. Conceptually, we are building larger sub-routines that aggregate multiple tasks into a full test. In the framework-based architecture, sub-routines are usually built as individual functions; their behavior being modified by passing in arguments when they are invoked. In the data-driven architecture, entire scripts essentially become sub-routines that have data passed in through an external mechanism; often via spreadsheets, flat files, or database tables.

One possibility of a test set benefiting from data-driven techniques is the testing of a data input screen. The same controls will be traversed by each test case testing the screen. In a script based on this architecture, each test case is actually defined by the different data used. For example, one test may ensure that the error handler works correctly when a social security number contains letters rather than numbers. Another test may ensure the business rule requiring a first and last name works correctly. System error handling testing (negative testing) using data-driven techniques is particularly effective.

Each time through the screen, different data tests different business rules and system behaviors. The script itself, however, is executing the same code each time through. By using a common script, with different data for each test, we can reduce the number of scripts to be maintained.

Once again, we have increased the scope of the programming part of the automation task. We now need data access routines, driver scripts, and other programmatic support artifacts. Luckily, most of this support scaffolding is written once and then reused throughout the test suite. The overall result is less code written, netting down the overall cost per test case.

### **Table-Driven Architecture**

For each of the three architectures discussed so far, we have a commonality: test cases are encapsulated by scripts: a text file containing both discrete programmatic steps and calls to sub-routines. The script is self-limiting in some ways: the more scripts you have the more maintenance (and hence cost) you incur. The sheer magnitude of maintaining hundreds or thousands of test scripts becomes limiting to the project.

An analysis of the structure of a script causes us to realize that the physical script itself may be expendable. The script captures the test case logic in a linear way; it might be replaced (conceptually) by an ordered group of steps coming from a database or spreadsheet. After all, the actual components of a script (before compiling) are nothing more than textual data!

Table-driven architecture is still very much a new technology. There are several different approaches currently being used. All of them entail a tester or business analyst using a front-end tool to develop the test case logic. This logic is encapsulated into tabular form



Innovative solutions to maximize your testing investment.

(database table or spreadsheet.) At execution time, the table is read and executed sequentially, likely using data stored elsewhere, by a back-end tool (QARun™, WinRunner™, Robot™, or similar tool.)

Test case execution requires a single, very intricate script, running on the back-end tool which acts (for all intents and purposes) as a virtual machine. Conceptually, this approach is similar to Java execution; a stream of statements created on one platform is executed on a different platform. The statement stream is taken from the data store, processed to determine the executable meaning, and then actions are taken based on that meaning. This all requires some highly complex programming.

Once again, if we choose to use this architecture, we will have added a great deal of low-level complexity to our automation project. The value we derive from this added complexity is leverage. With a few very technical people, we can build an automation suite that allows a great many non-technical people (testers, business analysts, domain experts, etc.) to create robust and maintainable automated tests. With a table-based architecture, users (other than the development team) only need to know the system under test and how to use the front-end tool to be able to successfully create test cases.

One huge benefit to this architecture is the possibility of moving automated testing up in the testing schedule. The other architectures generally require that the application be available before test case creation can begin. By the time the test scripts are designed and built, it is usually very late in the testing cycle. Indeed, most automation suffers from the inability to do significant testing in the first test cycle; usually the testing of features lags their introduction by a full cycle.

However, because the test case description is decoupled from the executable code in the table-driven method, there is nothing to prevent table-driven test cases from being designed and written before the application is ready for test. The only requirement is the design be fixed such that the interface the test case is going to execute against is stable. The test cases obviously cannot be unit tested until the system is available, and any design changes in the system will certainly cause rework; however, in an environment where there are stable processes, the risk of change may be well worth the ability to get benefits from the earlier automation usage.

At its most simple, table-driven test cases consist of a series of test step descriptions. Each test step is a 5-tuple, consisting of:

1. The name of the window containing a control to be used
2. The name of the control to be used
3. The description of the action to be taken on the control
4. The data (or data location) to be used for the step
5. The expected reaction of the control to the step



Innovative solutions to maximize your testing investment.

A spreadsheet or database with five columns allows the serialization of these 5-tuples into a linear track through the test case.

Front-end support is an absolute necessity for table-driven testing. It is not reasonable for the test case creator to have detailed low-level knowledge of all of the windows and objects of the system. One common approach is to create a front-end tool using Excel whereby prompts are used to guide the test creator step by step. To select a window in the first column, a drop down box is actuated, giving the user access to the names of all the windows in the system. The cell is populated by choosing one item in the list. The drop down in the control cell is now populated with only those controls contained in the selected window. Each ensuing cell contains a drop down box listing only those items which logically follow from the previous choices in the 5-tuple.

A common enhancement to the simple table-driven scheme is to supply a set of framework functions which can be used as well as the Window/Object 5-tuple. This gives the test case creator the best of both worlds: the ability to script by control as well as by task or screen.

### ***Simplicity Through Complexity***

When we take a look at the architecture continuum, we can see two common threads running through all of the architectures. These are to handle the complexity of trying to test a computer program with another program, and to simplify the human effort needed to create new test cases.

Both of these imperatives are handled through programming. It is essential to understand that we do not add software development techniques to automation projects just to solve individual problems. That would be a reactive strategy. Instead, we should choose a proactive strategy where we try to make the automation work correctly at the same time we try to reduce the technical skills necessary for a person to create, run, and maintain each testcase. It is only by reducing the technical skills needed to create more test cases that we can reduce the cost per test case, and thus increase our overall return on investment in the entire automation program.

### ***TAC Architecture***

At the present time, The Test & Automation Consulting LLC Test – Application Programming Interface (T-API) supports a combination of the Framework / Data-Driven architectures. In the future, we expect to build a front-end tool that will allow us to implement a table-driven architecture as well. We are always trying to increase the complexity of our automation offerings to ensure reduced complexity for our clients.

For a detailed view of the design of the TAC architecture, please see the document, **TAC Architecture and High Level Design.doc**.