

Test Automation Architectures

As with application development, test automation benefits from a structured approach. The benefits are enhanced code reuse, progressive reduction in test development time and lowered script maintenance activities.

A test script is a section of code that, when executed, has a direct impact on the application. The test script can be executed on its own or called by another script and should be able to complete a single task. The process involved in developing automated test scripts is not unlike that used in any application development.

An implementation strategy is the means through which the automated test script is created. The physical implementation is dependent upon the test tool's capabilities, but generally, you should at least be able to create test scripts and functions/procedures.

A function or procedure is a section of code that completes a specific task, but is flexible so that it can be used in many situations. An example of a function would be "open_file", in which the existence of the file is first established and then based on a parameter is opened in either read-only or read-write mode. Functions are beneficial because they can utilize parameters to reduce code and increase flexibility and they can return the status of an action performed so validation steps can be automated as well. However, function cannot be executed on its own. It has to be called from a test script.

There are two basic implementation strategies: Capture/Replay and Keyword-driven. These strategies can be used exclusively or in combination with each other within various test architectures. Most architectures are comprised of a framework where tests are driven by an event, a dialog or a model. Data-driven framework, though not an architecture or an implementation strategy by itself, is a fundamental part of any test automation approach.

Capture/Replay

Capture/Replay involves recording the steps from a test script as they are performed within the application. This recorded script is then replayed to reproduce the test. Most (if not all) test automation tools have capture/replay capability.

Use of the Capture/Replay strategy by itself frequently leads to the abandonment of test automation because of the high maintenance costs. When a change is made to an application, it is often easier to rerecord the entire script than to modify the script to match the application change. If this needs to be done for a number of scripts, the effort soon outweighs the benefit of test automation.

However, used within the defining process of a test architecture, Capture/Replay can provide value without the excessive maintenance requirements.

Keyword-driven

Keyword-driven involves abstracting the executable code from its intended purpose, similar to calling a function with a set of parameters. This strategy is characterized by a set of function libraries, commonly called an engine. The engine's basic purpose is to translate the keywords into actions. The actions themselves are either functions within the libraries or separate scripts.

Keyword coarseness describes the scope or functionality covered by each keyword. This can be large like "enter client details", or small in which case the keyword may simply be "enter". If a large coarseness is used, the functionality of the keyword may be physically implemented as a callable function or as a script, depending on the capabilities of the test tool. Additionally, the called function or script may be coded using either Capture/Replay or as Keyword-driven but with a smaller coarseness.

Input Data

Incorporating input data into the script itself is never a good idea. A test script that contains its input data can only test that input data. An alternative is known as data-driven testing.

A data-driven script is any script that separates its executable code from its input data. Data-driven techniques can be used with both Capture/ Replay and Keyword-driven implementation strategies.

Test Architectures

Test automation architectures consist of dividing the test requirements of an application into autonomous pieces. These pieces are then implemented individually, (in some manner). The pieces are then combined to form the test scripts that meet the test requirements.

The pieces of the framework can be discussed in terms of granularity. Each architecture runs into this question. Should the pieces be large and encompass large sections of the application or should they be small and have to be pieced together to form a whole “piece” as defined by the architecture?

This is typically answered by:

- Time allocated for implementation– Smaller pieces take longer to implement as there are more of them and there is overhead in their creation
- Expected amount of maintenance– Smaller pieces actually have less maintenance time as the likelihood of having to make the same change more than once is reduced
- Training needs – The more technical or abstract the implementation strategy, the more skillful the implementers need to be.
- Application complexity – A complex application is easier to handle in small chunks.
- Execution time – Larger pieces execute faster as the shut-down and reload overhead is reduced.
- Frequency of execution – Pieces that are frequently executed in the same order could possibly be incorporated into a larger piece.
- Re-usability – Small pieces can fit into more places than larger ones.
- Machine capability – Computers with limited resources require that the scripts be very efficient which is easier to accomplish with small pieces.
- Function versus script – Functions execute faster than scripts but they can take longer to implement and they have a tendency to add abstraction to the test environment.

It is the decision on how to define the pieces that identifies the architecture. Three often-used architectures are event-driven, dialog-driven and model-driven. Each of these uses its own means of identifying the pieces that form the framework.

Event-driven Architecture

In an event-driven architecture, the application is divided by its triggers into autonomous actions.

A trigger is any activity within the application that causes the basic processing to branch in some direction. They normally occur at a decision point within the application. At this point the processing or flow can be changed based upon some activity made either by the user or by the appearance of specific data values.

An event is the combined set of trigger and action. Some examples of events are “buyer selects to purchase” or “buyer selects to add more items to the purchase” which appear on a shopping cart webpage. Each of these is an event and triggers a specific path and set of validations. They are also static events and can be implemented into the test script and/or input data.

Events can also be dynamic, for example “purchase stock X when it has value Y” or “sell stock X when it has value Z.” These events rely on dynamic evaluation of the state of the application. Dynamic events are common in instances where the initial conditions of the application are not known or cannot be predefined for testing purposes. While, for the given examples, the subsequent actions are known, the time at which they will occur or activate is not. This means that they can only be implemented within the test-script code.

When using the event-driven architecture, the definition of an event is critical. It is simple to get caught up in thinking of everything as being an event. For example:

- Login successfully
- Login with invalid user name
- Login with invalid password
- Login with invalid user name & password

In the above listing, each of these could be considered as being an event but in actuality there is only one event—login. The rest are simply variations within the input data that need to be handled with appropriate validation checking.

Dialog-driven Architecture

In a dialog-driven architecture, the application is divided by its visible or user interactive components.

A dialog, in this context, is anything that is displayed by the application or allows interaction between the application and the user. This means that whole windows/pages, individual tab pages, logically separable screen areas, and pop-up messages/dialogs are looked at as being independent of each other. The definition of a dialog within the dialog-driven architecture is flexible so that the maximum amount of reuse can be obtained for the effort.

The division of pages should be logical within the application. Using a flexible page division dialog definition we see that we can reuse a test script for entering an individual's personal details for entering the spouse's details. One must keep in mind that while the personal details example would work where the two sets of information are nearly identical, it would not, however, achieve the same goal for instances where different information is gathered.

Pop-up dialogs do not always need to be defined as independent dialogs. This is especially true of message dialogs which form a generic category that can be dealt with in a standard manner. It is also true that if a dialog does not quite fit into the general handler, then it probably should not be forced into it. Forcing a dialog into a general category inevitably results in the general handler containing specialized code to handle one/two uncommon cases. This simply increases the effort required to maintain the code.

A dialog-driven architecture is typically static with all portions being definable at script creation time. Dynamic areas within an application can be considered as separate dialogs. This means that when the area changes, like say an additional row is added to a table, it is only a "dialog" that has to be changed and not an entire screen.

When using the dialog-driven architecture, the definition of a dialog is critical. The rules of thumb to keep in mind are:

- Avoid trivial dialog definitions that create an unnecessarily large number of test scripts
- Avoid overly large dialogs that contain a large number of fields
- Define dynamic areas as being a dialog separate from the rest of the dialog
- Attempt to use as many generic dialog definitions as possible

Model-driven Architecture

In a model-driven architecture the application is separated into logical or business areas.

The model is a stylized representation of the identified logical or business area. The internal workings within the model are not dealt with on an architectural level but are rather considered to be implementation details. This is done to separate the input and output data from the physical or working details.

Use of test cases form the basis of the model architecture. For example, in an e-commerce application you could have test cases for the following:

- Ordering items & then abandoning the cart
- Ordering items & then ordering additional items before proceeding to the checkout
- Ordering items & proceeding to checkout

Dividing an application into a set of models is a two-step process. In the first step you look at the overall business areas. This results in a high-level description of what a model represents. The entries in the list above are an adequate example of first level models. In the second step the models are examined to see if there are sub-models that can be defined. It is best to consider the sub-models for the possibility of reusability. So as a second phase description, we end with the following sub-models:

- Order items
- Abandon cart
- Checkout

Each of these sub-models has its own input and output. Where it makes business sense, the output of one easily becomes the input of another. This makes it easy to group the sub-models together to form the first-level models and thereby completing the required verification steps.

In using a model-driven architecture, the following should be considered:

- The model's processing activities are not dealt with at the definition level
- Each model can take any input
- Each model produces a well-defined output
- Sub-models are defined by considering the various points of input & output
- The output of one sub-model can be the input of the next
- Sub-models can be pieced together to form the required test cases

Each application has its own characteristics. Not every characteristic is going to map well to a chosen implementation style. If a specific portion of a test makes sense to be implemented by a Capture/Replay script then that is the best implementation and it should be used even if most of the implementation is done via Keyword-driven scripting.

Just as the implementation style is influenced by the application's characteristics, so is the architectural structuring. Architectures are a means through which the application can be broken into simplified, manageable, and reusable pieces. Since the goal is to create reusable scripts and thereby reduce maintenance, the most logical architecture for that portion of the application should be used when considering how to define and simplify its context.

For a given test requirement in any given application, there is no right/wrong way to implement the solution. The solution should be tailored to the application and the test requirements at that point.

Visit us at www.spherion.com to learn how your entire organization can benefit from applications that are delivered on schedule and within budget.