

## Tools & Automation

### INFO TO GO

- The layers of code that make up a test framework can be reduced if developers build in test support. For example, window maps give window controls meaningful names, something developers could do themselves.
- When using task libraries, don't sacrifice ease of understanding for expediency.
- You may want to use a tried-and-true framework to avoid costly test suite errors.



# Deconstructing GUI Test Automation

Four framework-building abstraction layers, their advantages and pitfalls, and ways developers may be able to help you avoid them altogether.

*by Bret Pettichord*

User:  User:

TESTERS OFTEN LOOK TO AUTOMATION AS A WAY to avoid repetitive manual testing of the Graphical User Interface (GUI). And although commercial GUI test tools include built-in support for capture/replay, many times a more sophisticated approach is needed. That's right. Sometimes you have to write code. Code to encapsulate common testing tasks. Code to execute easy-to-read test case descriptions. And code to supply layers to ease the task of updating the test suite when the inevitable user-interface changes occur.

Why layers? The sophisticated automator separates code that addresses different concerns into layers, each loosely coupled to the others, in order to make code easier to update and test. Yes. Just like any other kind of code, automated test suites need to be tested.

These layers of code, or abstractions, are often combined to create frameworks: external scaffolding surrounding the software application under test. Frameworks make tests easier to create and maintain, but they also take time and effort to create. How much effort to put into developing frameworks and how to justify that effort are key design issues for automated test suites. To help you decide those issues, this article presents the pros and cons of four abstraction layers often used in automation frameworks.

### Window Mapping

A window map is an abstraction layer that associates names with *window elements*: the controls or widgets that make up the user interface. These include text labels, push buttons, menus, list boxes, and all the other elements that make up modern graphical interfaces.

## Four Abstraction Techniques to Try

**Window mapping** gives elements specific names so tests are easier to update and understand. However, if developers give window controls meaningful internal names that are accessible to your tool from the start, there is little need for window maps.

**Task libraries** group sequences of steps that make up part of user tasks when those sequences show up in multiple tests. On the other hand, if developers create or expose analogous service APIs that the tests can call directly, task libraries are unnecessary.

**Data-driven test automation** separates the parameters of a test case from the test script so that the test script can be reused for many related tests. However, data-driven test automation can get bogged down in routing and make test drivers overly complicated.

**Keyword-driven test automation** formats tests as tables or spreadsheets and creates parsers to read and execute the test descriptions. One drawback: keyword-driven test automation requires a parser, or test driver, which can be difficult to create.

Sometimes automators find they need to use a variety of such techniques to reliably reference window elements. The choice of the best technique often depends on the nature of the interface technology (Web-based, Visual Basic, Java, etc.), the way that the developers have created the window elements, and the capabilities of the test tool being used. Window maps make tests independent of the chosen techniques by providing meaningful element names for use in test scripts. Tests use the element names and let someone else worry about translating the name into a specific window element. Window maps make test scripts easier to understand. And they make test suites easier to update when window controls are rearranged or renamed: the window map can be updated without having to make further changes to the tests.

Window elements can be accessed by many techniques:

- *Adjacent Labels*—the edit field following the text “state”
- *Ordinal Position*—the second button on the dialog, read from top to bottom, left to right
- *Absolute Position*—the checkbox located 120 pixels across and 45 pixels down from the upper-left corner of the dialog
- *Internal Names*—assigned by the programmers
- *Other Properties of the Controls*—such as color, font, justification, or even non-visible properties

Now suppose you are automating an application with a login window. It has two fields: Login and Password. Suppose that the best available way to describe those fields is by ordinal position. So, in our code, we tell our test to enter a specific string of text in textbox #1, and to enter another string of text in textbox #2. Will it be obvious to others just by looking at the test that “textbox #1” represents a login name? No. What happens if later the developers add a new field, Server, to the login screen, in between the first and second textboxes? All of a sudden there are three textboxes instead of two. Textbox #2, which used to describe the password field, now refers to the server field. We now have to go back and change every instance of textbox #2 to textbox #3—but only those instances that refer to this dialog. What a nightmare!

To avoid this, we create a window map. Using this map, we can name textbox #1 “Login\_Name” and textbox #2 “Password.”

Name	Access Method
Login_Name	textbox#1
Password	textbox#2

## Use task libraries when you have the time and resources to design them well and sufficient expected usage to justify them.

Our test scripts will be easier to understand since we can refer to the Login field as “Login\_Name” rather than “textbox#1.” Also, when those developers go changing things and add a server field to the dialog, we only have to update the window map, not all the references in the tests. The new map would look like this:

Name	Access Method
Login_Name	textbox#1
Server	textbox#2
Password	textbox#3

Window maps also help when you are testing translated interfaces, particularly when the window elements are being identified by their human-language labels. A separate set of window maps can be created for each language—or localization—allowing the same test suite to run against different versions of the software.

### Window-Mapping Tools

Window maps go by a variety of names, and many tools support them:

**GUI maps** (WinRunner)

**Window declarations** (SilkTest)

**Application maps** (Certify)

Other tools can be extended to employ window maps using the tools’ scripting languages.

In our example, not having access to meaningful internal window-element names necessitates creating an external mechanism to provide it. If the programmers had given meaningful internal names, it would have, in effect, allowed for the elimination of duplicate code. The tests and the product code could have used the same names to refer to the same controls. When you find yourself having to create external scaffolding like window maps to make your tests work, see if there are ways the product design team could refine their code to eliminate the duplicate effort.

### Task Libraries

Window maps allow you to write tests that look something like this: select menu item x, enter “foo” in text field y, press the z pushbutton. The syntax may be a bit more technical, but tests are written at this level of detail. Understanding tests like this can be difficult unless you know the software well or else can follow along and operate it manually as you read the test.

Task libraries help you write tests as a sequence of mid-level tasks rather than low-level steps. A well-designed task library will group tasks according to users’ conceptions of how the software usage is organized, making tests shorter and easier to

write and comprehend. And the functions will be named in terms that users understand. When library interfaces are designed up front, tests can be written before the software is ready for testing and even before the final user interface designs are complete. This brings large advantages.

Consider the following example of low-level pseudo code for logging in to the system under test.

```
menu_select ("System Login")
enter_text ("Login_Name", "mrbunny")
enter_text ("Password", "jake")
push_button ("OK")
```

Instead of repeating these lines at the beginning of each test, you could create a function that would abstract these steps into a single step:

```
system_login ("mrbunny", "jake")
```

Or you might create a function that defaults to an available test account on the system under test:

```
system_login ()
```

The most difficult aspect of creating task libraries is design. It’s easy to create an ad hoc library as you go along. It’s harder to create a library that provides a clear set of abstractions. A well-designed library can make it much easier for the tester to think about testing concerns when writing tests, rather than interface details or automation quirks.

Use task libraries when you have the time and resources to design them well and sufficient expected usage to justify them. Some believe that to avoid duplicate code, a new function should be added to a task library whenever two or more tests share the same steps. But it may be riskier to have a hastily written test library than to have duplicate test code. Moreover, the general rule against duplicating code applies more to

### Need for Speed?

Like all abstraction layers, task libraries add complexity. When done well, they make each layer easier to understand. When done poorly, tests may be harder to understand than if they weren’t used at all.

Suppose a test calls the `system_login()` function with our normal test account: “mrbunny.” But on one particular test system, we have to use the “frodo” account instead.

The quick fix is to add some code to the `system_login()` function to substitute “frodo” for “mrbunny” when we’re running on this particular system.

Seems reasonable. But, imagine trying to debug why a test is failing if the test accounts are later reconfigured.

Don’t sacrifice ease of understanding for the sake of expediency.

## Detecting Misuses of Task Libraries

Focus on start and end states when designing task library functions. Try to include code to verify these states.

- In what state must the product be when the function is called?
- What screens or dialogs must be active?
- What objects or records must have already been created?
- In what state does the function leave the product?

production code than test code. *Production* code can have hidden bugs when a change that should be made to all instances of the duplicated code is only made to one. What happens if duplicate *test* code is later updated in only one place? The un-updated tests should fail. The problem isn't hidden. Duplicated test code does not have the same risk for hidden bugs as duplicated production code. Indeed, hasty test libraries are more likely to lead to hidden bugs than hasty test code duplication.

Like window maps, the external scaffolding of task libraries is unnecessary if code is designed with testing in mind. Indeed, development teams charged with creating the fixtures to support automated testing by customers or analysts often choose to expose or create Application Programming Interfaces (APIs) for testing, rather than construct a scaffolding framework. Assuming that the appropriate APIs exist, this is often more direct and easier to implement, especially for scenario and usage testing. Of course, these tests don't test the GUI directly. But they provide an efficient way of accomplishing a lot of automated testing. Creating and using APIs for testing requires changes in the traditional relationships between programmers and testers:

working more closely together and with greater mutual trust and cooperation.

### Data-Driven Test Automation

Data-driven testing begins with the observation that much of our testing is data-intensive. Often collecting the right test data defines the tests. For example, consider the following tests:

Login Name	Password	Result
mfayad	xyz	login
dschmidt	123	expired
rjohnson	abc	reject

You could automate these tests by creating a test script for each test. Or you could build a data-driven test driver that would load the data in from an external file and then execute each test, entering the name and password in the login dialog and then checking whether the expected result occurred.

Data-driven test drivers are particularly useful when you have a lot of tests that vary predictably. The data may be created by hand, computed in a spreadsheet, exported from another application, or generated from usage logs. The approach's strength is its ability to make use of data obtained from various sources. This approach is popular enough that most commercial test tools include wizards for creating these test drivers.

This approach becomes strained, however, when different tests take separate routes through the application. I've seen some test formats that include columns for flags that indicate the path the test needs to take, but this approach really breaks down as the paths multiply. The test data becomes harder to read and understand in isolation, and the test drivers get overly complicated.

### Keyword-Driven Test Automation

A better technique for these situations is to allow for procedural commands to be included with the test data. This allows tests to still be written in spreadsheets (or other tabular formats).



## Parlez-vous Test Description?

All four abstraction techniques support the creation of *test description languages*. Determining the kind of test description language to use is a key question for test automation architecture. There is great value in being able to express tests so they are understandable and reviewable by lots of people on the team. The actual language and format that works best will often depend on the background and preferences of the testers, analysts, and developers working on a project.

Defining test description languages, of one sort or another, can be a major benefit. They allow more people to be directly involved in the creation and review of automated tests. And if the format is defined early, test creation doesn't have to wait until after the software is working.

Test formats may be spreadsheet-based or may use a more conventional language syntax. The best format is often a design tradeoff between ease of use and difficulty of implementation.

In many cases, the least accessible choice is to encode tests in a proprietary scripting language that only a few test automation specialists understand. For example, the keyword-based approach requires a sophisticated test driver, or parser. These are difficult to create because many of the test tools use proprietary scripting languages, or *vendorscripts*, which have been stripped down to make them easier or safer to use (or so they say). In the process, the language features that most naturally support parser development have also been removed.

At the same time, supporting a homebrew test language may be more trouble than it is worth. A sign that a more conventional programming language would be more appropriate is when keywords and syntax to support variables and control logic appear in the language design. The value of the spreadsheet-based languages lies in their simplicity. If you need more power, consider using a standard scripting language.

Our industry has a history of creating languages designed for testing. Some commercial test tools are now using standardized scripting languages. This is a step in the right direction. We don't need proprietary *vendorscripts* in our test tools, and we need to avoid creating elaborate third-generation test languages.

## The next big step may be the development of test APIs for system testing.

Thus our tests might look like this:

Test	1	
Login	mfayad	xyz
VerifyScreen	MainAccount	
Logoff		
Test	2	
Login	dschmidt	123
VerifyError	"Your user account is expired."	
Test	3	
Login	rjohnson	abc
VerifyError	"Unknown user"	

These tests are somewhat more verbose than their data-driven counterparts, but they offer the possibility of being expanded in different ways. The commands that appear in the first column of this expanded style of data-driven tests are often called *keywords* or *action words*.

These tests require a more generic test driver that reads the test file, looks up the library function (system\_login) associated with the keyword (Login), and then executes it using the rest of the data on the line ("mfayad," "xyz") as arguments to the function. This approach to test automation is sometimes called "third-generation" test automation. Some see it as the next big step.

Keyword-driven tests work by providing yet another layer of abstraction. Many testers and analysts find this kind of spreadsheet-based test format easier to create and understand. It's valuable if it can improve the productivity of the testers and help them focus on testing concerns, rather than the mechanics of the automation.



## Object-Aware GUI Testing Tools

Commercial GUI test tools include Mercury's WinRunner, Rational's Robot, Compuware's QARun, Segue's SilkTest, Empirix's e-Tester, and several others. All of these tools are *object-aware*. This means that they are able to detect and identify window elements, or controls, as they execute. They can detect that an OK button is present before pushing it. The first generation of GUI test tools, appearing in the early '90s, did not include this technology and ran into synchronization problems. The tools would push buttons before the buttons actually appeared, leading to spurious failures.

In a way, this object-aware technology represents an abstraction layer itself. It translates control-based commands into a stream of keyboard inputs, mouse clicks, and mouse movements. Test automators, who create support for non-standard controls (e.g., controls programmers create specially for a particular product), often find that they have to create similar translations.

## False Alarms and Silent Horrors

Test suite errors fall into two categories: false alarms and silent horrors.

**False alarms** occur when a test reports that it failed, but the failure can be tracked down to an error in the test, the tool, the environment, or something else other than the product under test. False alarms are a nuisance, but not nearly as worrisome as silent horrors.

**Silent horrors** are the errors that cause tests not to be run, to be run differently than specified (and so not encountering the bugs they were designed to find), or to fail to report bugs detected by the tests. The more complex a test automation system, the more likely that it will contain both kinds of bugs.

Because of these concerns, many automators reuse frameworks that have been used and tested. Frameworks supporting data-driven and keyword-driven testing are available from several sources. These frameworks support layers of abstraction, as well as other useful features including error recovery systems that enforce test case independence and prevent a single test failure from interfering with other tests.

## Conclusion

There are many good reasons for using abstraction layers to support GUI test automation. They can make test suites easier to update when the product user interface changes. And they help testers to focus on testing rather than automation.

However, these layers of code are external scaffolding built solely to support testing. While window maps provide a way for test scripts to use simple names and ignore the variety of techniques needed to translate those names into actual controls, they are unnecessary when test scripts can reference the internal names of the controls directly. Similarly, task libraries could be eliminated if APIs were made available for system testing. Data-driven testing is valuable, but can become cumbersome. Keyword testing solves the problems of data-driven testing, but does require the development of a test driver, or parser, which is not an easy task, especially with stripped-down vendorscripts.

Although the present state of the art for GUI test automation is to use testing frameworks, the next big step may be the development of test APIs for system testing. Only time will tell. **STQE**

*Bret Pettichord is co-author of Lessons Learned in Software Testing and a columnist for Stickyminds.com. He has worked as a test automation lead for several companies and now works as an independent consultant based in Austin, Texas. Contact him at [bret@pettichord.com](mailto:bret@pettichord.com) or <http://www.pettichord.com>.*

**STQE magazine is produced by  
Software Quality Engineering.**