

The Art of **KEYWORD DRIVEN Testing**



TABLE OF CONTENT

| | |
|--|-----------|
| Why Keywords? | 02 |
| Economic Advantage | 03 |
| Criteria for a Good KDT Solution | 07 |
| A Real-world Problem <ul style="list-style-type: none"> • Example scenario • Old-school KDT frameworks • Advantages of old-school KDT approach | 08 |
| Disadvantages of old-school KDT approach <ul style="list-style-type: none"> • Recursive Keyword Combination • Action-Based Testing™ (ABT) • Example test case in Action-Based Testing • Advantages of Action-Based Testing • Methodology should be embedded in a product • The combo of Action-Based Testing & TestArchitect™ | 11 |
| Conclusion | 18 |
| Author Section | 18 |

Keyword-Driven testing has served us well for decades but, like everything else, it evolves. This paper inspects the common mistakes in implementing Keyword-Driven Testing solutions. Most significantly, we'll analyze the setbacks of the noun-based keyword approach along with the direct mapping between a keyword and its code. Later on we'll discuss a modern approach to Keyword-Driven Testing namely Action-Based Testing.

Why Keywords?

Before the invention of *Keyword-Driven Testing (KDT)*, Test Automation was a little messier. For each test scenario, you had to code all of the interactive steps and check points in a specific programming language such as VBScript, Java or C#. That meant everyone on our team must be a professional coder. Or worse yet, (as we often joke) a full stack super human *a.k.a. professional tester slash coder*. We invented that nickname because one had the glorious task of handling both *testing* and *coding* simultaneously back then.

Furthermore, you couldn't reuse previously coded steps and checkpoints in similar test scenarios. Every test scenario authored by different *tester-slash-coder super human* had a different implementation. *Poor Reusability* not only reduced test production throughput but also devastated *Maintainability* of test scripts. Test Automation artifacts became a liability more than an asset.

Keyword-Driven Testing was born out of those needs to achieve Decoupling, Reusability and Maintainability.

- KDT enables clear division of labor. Keywords are awesome because they allow you to separate test automation into two distinct activities, Test Design and Test Automation, which can be worked on in parallel. Programming staff who chooses to specialize in coding will implement the keywords of which the rest of the team can reuse. On the other hand, non-programming staff whose expertise is on the domain knowledge and business logic side are not required to engage with coding. They just need to focus on designing the right test cases for applicable business processes.
- KDT makes our code base "leaner". Once a piece of code doing a specific function is abstracted and encapsulated, you can call it over and over again with different parameters, free from duplicating your code.
- KDT allows us to embrace inevitable changes in Agile development. Since you're recalling the piece of code, not duplicating it, you only need to modify one isolated piece of code when the app under test (AUT) changes. It doesn't matter if you've called it thousands of times from thousands of places. This is similar to the superiority of *Procedural Programming over Unstructured Programming ("spaghetti code")*.

All of these benefits yielded improved Productivity and Speed for test teams who were among the early adopters of KDT. Since your automation framework needs very little overhead to keep running, your hands are now free to focus on refactoring old tests and writing new tests to increase *Test Quality* and *Test Coverage* (especially when combined with *Data-Driven Testing*). Over time, your test project will scale with flying colors.

Economic Advantage of Keyword-Driven Testing

Let's dive into how *Keyword-Driven Testing* saves you cost. In the early-days stage of test automation, automating a test case means translating all test steps written in English to a code file that will be executed against the AUT over and over again. Thus, you will notice the direct correlation between the number of test cases designed, and the number of test cases coded.

| Test Cases | Test Cases Designed | Test Cases Coded |
|------------|---------------------|------------------|
| 10 | 10 | 10 |
| 50 | 50 | 50 |
| 100 | 100 | 100 |
| 500 | 500 | 500 |
| 1000 | 1000 | 1000 |
| 2000 | 2000 | 2000 |
| 5000 | 50000 | 5000 |

With KDT, you still have to design & code the test cases as well as the keywords but the amount of code has been reduced dramatically. You can expect the number of keywords coded in KDT to increase at a slower pace.

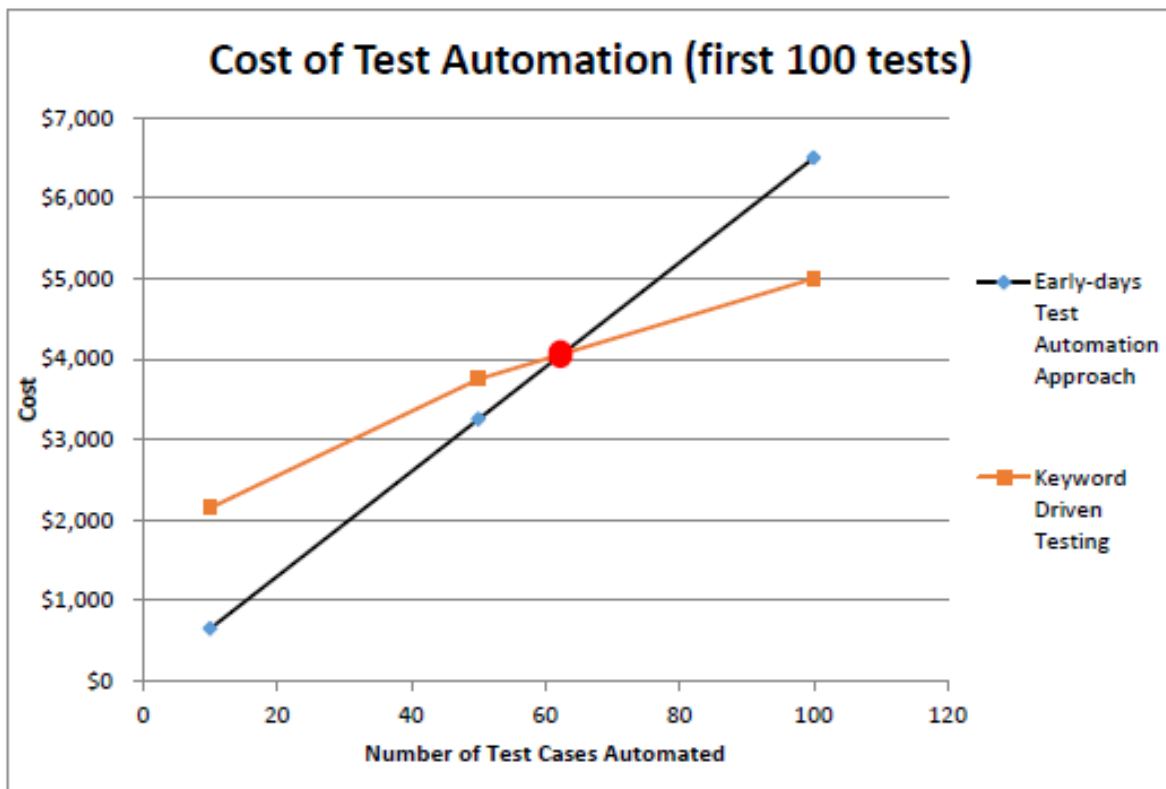
| Test Cases | Test Cases Designed | Reusable Keyword Coded |
|------------|---------------------|------------------------|
| 10 | 10 | 40 |
| 50 | 50 | 60 |
| 100 | 100 | 70 |
| 500 | 5000 | 150 |
| 1000 | 1000 | 200 |
| 2000 | 2000 | 200 |
| 5000 | 5000 | 200 |

Our first 10 test cases might require 40 keywords. But automating 1,000 test cases will only need 200 keywords. And tests beyond that will require no additional keywords, reducing work for the automation engineers.

To visualize the cost savings of *Keyword-Driven Testing* in this particular scenario, let's assume the following:

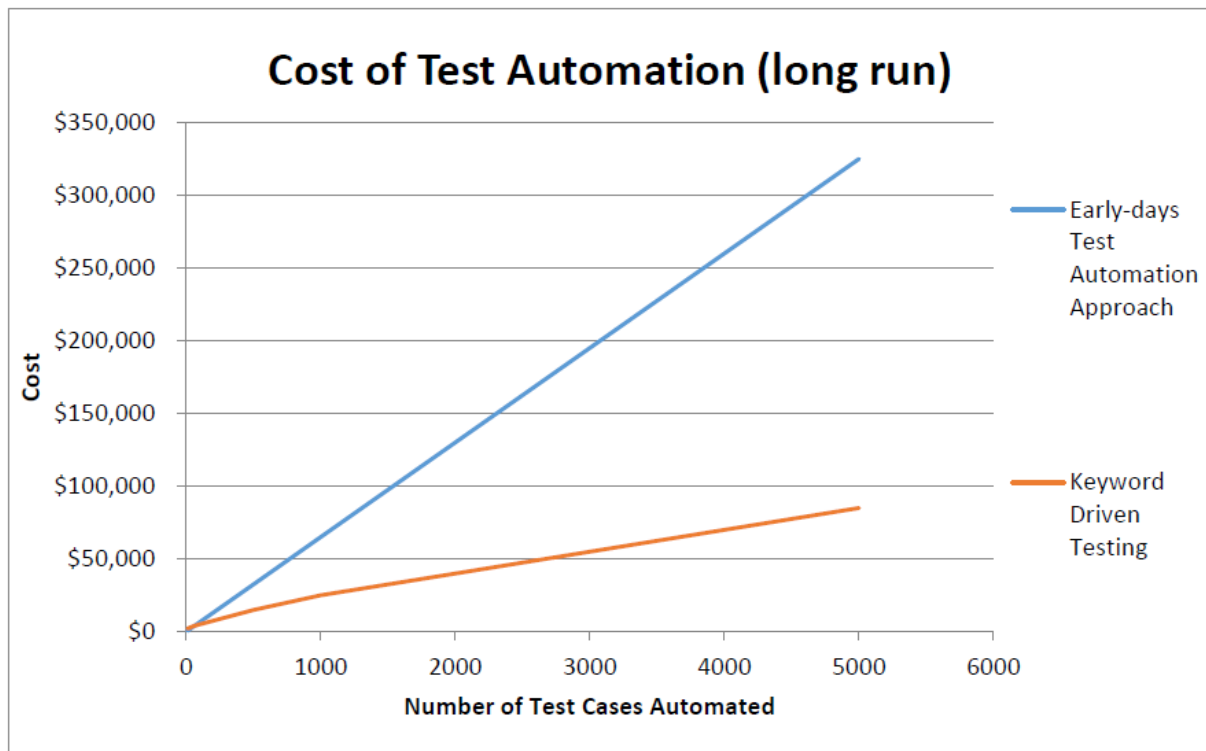
- A business tester can design 4 test cases per hour (with the salary of \$60/hour)
- An experienced automation engineer can code 2 keywords per hour (with the salary of \$100/hour)

Using these figures, we can make the following comparison between the costs of test development using the *early-days test automation approach* versus *Keyword-Driven Testing*.



As the chart illustrates, the early-days test automation approach is cheaper if we automate a very small number of test cases. But as the number of test cases grows, KDT becomes far cheaper. In this example, KDT is more expensive until we've created roughly 63 tests (at the cost of \$4,095).

In the long run, KDT becomes significantly cheaper than the early-days test automation approach as illustrated in the graph below.



When Keyword-Driven Fails

Keyword-Driven Testing is sweet but... sometimes it comes with a pinch of salt. The methodology is not inherently faulty in itself. Actually, KDT principles are very simple and effective as we've analyzed earlier. However, successfully building and running a KDT framework in reality is not a trivial task. Just like great ideas such as *Agile* and *DevOps*, there's usually a huge fracture between theory and practice.

In-house KDT framework

Plenty of open-source KDT solutions are out there and they are quite easy to adopt. If we choose to build our own KDT framework from scratch, we should keep an eye on these challenges:

Labor division separates *testing* from *coding* but we should remember that it doesn't completely eliminate *coding*. Someone on the team has to "eat the frog" and learn to code from Day-1.

For maximum *reusability*, it's worthwhile to make an upfront investment (usually a substantial one) on the architecture of the KDT framework. As you might already know, system architects experienced in test framework designing are certainly not easy to find or cheap, but this is for a good reason. Only after we realize that we are stuck in endless rabbit holes, we regret not hiring a more experienced architect from the beginning.

Since objects in a well-architected framework are usually more abstract, new team members encounter a steeper learning curve. If we neglect sufficient knowledge through

transferring and hands-on practices, productivity simply diminishes. You can also expect more frustration, conflicts, and demoralization within the team.

Because implementing keywords falls in the hands of automation engineers and automation engineers don't always understand the business processes in sufficient detail, they might create redundant keywords. For example, check account and verify profile keywords probably refer to the same thing. *Redundancy* means poor *maintainability*. The solution ought to be frequent feedback-loops between the two sub-teams (business testers and automation engineers).

Keyword implementations are written in a programming language thus they are not very *business-readable*. As a result, although business testers and domain experts want to help automation engineers to eliminate redundant keywords, their hands are mostly tied. Again, we should rely on better communication between the two sub-teams.

When your project starts to grow, new *large-scale challenges* will come in:

The test team will have to invest more time and money on each additional utility such as *Parallel Execution*, *Test Dispatching*, *DevOps Integration*, *Error Logging*, *Test Environment Management*, and *Statistical Reporting*. It is important to prepare for these utilities starting from the architectural design stage.

There will be time when the test team needs to scale to another app platform, e.g. from web to mobile. Again we must invest more time and money on implementing platform-specific keywords. Beware of this hidden cost. *Free-of-charge open-source solutions are not completely free after all*.

Test production must achieve a *critical mass* before it can pay off the initial cost of developing the KDT framework. The acceleration process usually takes months, if not years, to achieve the *break-even point*. Clear end goals should be defined from the beginning of the project to objectively measure the project's success. Maximizing *reusability* can also boost test development speed so that we can achieve the *break-even point* faster.

Off-the-shelf KDT solutions

On the other hand, if we choose to buy a commercial KDT product, we should try to avoid the old-school style of implementing KDT because it poses certain limitations:

- In some tools, business testers define keywords in Keyword View but those keywords have to be implemented in Code View using a programming language such as VBScript. Therefore, coding is still necessary. It's easy at first but it tends to be less productive over time.
- You might receive the promise of low marginal cost in increasing test volume, but if

the KDT framework in question stops at mapping one keyword to one function, the cost is still high because you cannot recursively combine existing keywords to create new keywords. This setback slows down test development speed and limits the scope of the tests. We call this approach **Direct Keyword-Code Mapping**.

- Some products map each **TestStep** in a **TestCase** to one **Test-Object** representing the GUI of the app under test (e.g. a web page). We call it the **Noun-based Keyword** approach. Not only these noun-based keywords are quite hard to understand, but they also create unnecessary maintainability problems.

Note that these problems are not limited to off-the-shelf products. If we choose an in-house KDT framework ourselves, we still need to keep an eye on these overlooked mistakes.

No matter whether we stick with an in-house KDT framework or purchase a commercial KDT product, if we don't pay enough attention to these lessons learned, the next thing we know our test project has become a train wreck. By then, most of our time will be spent on firefighting instead of productive work.

Criteria for a Good KDT Solution

From the holistic viewpoint, regardless of your tool choice, a comprehensive KDT solution should possess these characteristics:

- **Platform-agnostic keyword implementations.**

To limit coding as much as possible and scale to other app platforms, test teams need to invest in low-level automation libraries. The key success factor is *Abstraction*. *Keyword-producers* should hide the technology layer from *keyword-users*. From the *keyword-users's* point of view, keywords should be self-explanatory and platform-agnostic.

- **Deliberate architectural design.**

The test framework itself is an actual software product. Similar to your app under test, it must be carefully designed to boost *reusability* and *maintainability*. Also, we should keep in mind that we might need to add new utilities into the framework later on. The principle here is solving today's problems while laying the bricks for future growth.

- **Enable collaborative test design.**

To avoid creating redundant keywords, automation engineers and business testers should build a common understanding of the *business objects* and *business processes*. They also need to work hand-in-hand in defining common test assets (reusable keywords that both parties comprehend). This is more like a *process solution* than a *product solution* but if your KDT framework enables creating *business-readable* keywords, communication will be much easier.

- **Neither Noun-based Keywords nor Direct Keyword-Code Mapping.**

These approaches might sound very tempting at first but in the long run, they tend to do more harm than good. **Noun-based Keyword** approach promises that after automatically capturing the AUT's GUI, you can drag-n-drop keywords to create new

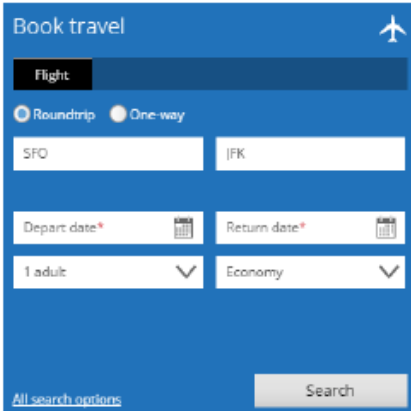
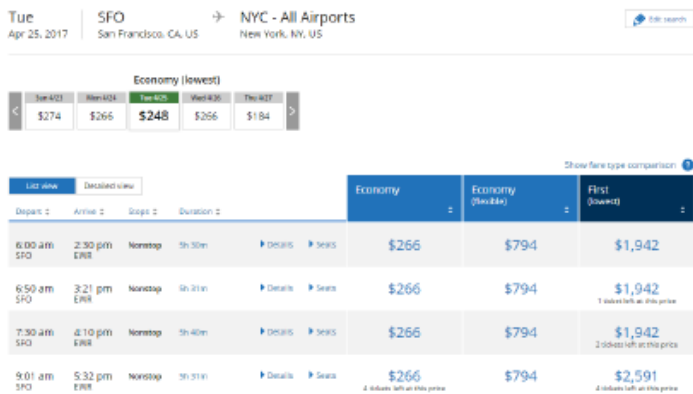
tests right away. The catch is your tests become very *hard to refactor*. On the other hand, Direct Keyword-Code Mapping *hinders reusability* by tight-coupling a keyword with a piece of code (usually a function in a programming language). So it's better off for us to stay away from these rabbit holes at all cost.

A Real-world Problem

In the following sections, we'll analyze the Noun-based keyword and Direct Keyword-Code Mapping approaches in more detail. Then we'll discuss their cures: Verb-based Keywords, and Recursive Keyword Combination.

Example scenario

Suppose we are testing the flight cost calculation function of an airline website. After you choose the departure and return flights, you will be given the total cost. That number of should be the sum of the departure flight's cost, return flight's cost as well as taxes and fees. We'll verify whether the total cost is calculated correctly. The test flow will look like this.

| Step | Screenshot |
|--|--|
| <p>On the Home page:</p> <ul style="list-style-type: none"> Fill in the information to search: <ul style="list-style-type: none"> A round-trip flight from SFO to JFK Departs on 25-Apr and returns on 30-Apr Click Search button |  |
| <p>On the Departure page:</p> <ul style="list-style-type: none"> Record the departure cost of the first applicable flight (\$266) Choose the first applicable flight in the displayed result |  |

On the Return page:

- Record the return cost of the first applicable flight (\$187)
- Choose the first applicable flight in the displayed result

Sun
Apr 30, 2017

NYC - All Airports
New York, NY, US

→ SFO
San Francisco, CA, US

Economy (lowest)

| Flight | Price | Class | Price | Class | Price |
|----------|-------|---------|----------|-------|---------|
| Mon 4/24 | \$187 | Economy | Tue 4/25 | \$187 | Economy |
| Wed 4/26 | \$187 | Economy | Thu 4/27 | \$187 | Economy |
| Fri 4/28 | \$187 | Economy | Sat 4/29 | \$187 | Economy |

Show fare type comparison

| Depart | Arrive | Stops | Duration | Economy | Economy (flexible) | First (2-cabin, lowest) |
|-------------|--------------|---------|----------|---------|--------------------|-------------------------|
| 6:00 am EWR | 9:15 am SFO | Nonstop | 6h 15m | \$187 | \$797 | \$1,202 |
| 6:45 am EWR | 10:00 am SFO | Nonstop | 6h 15m | \$187 | \$797 | \$1,202 |
| 7:47 am EWR | 10:45 am SFO | Nonstop | 5h 58m | \$187 | \$797 | \$1,742 |
| 8:30 am EWR | 11:45 am SFO | Nonstop | 6h 15m | \$242 | \$797 | \$1,742 |

2 tickets left at this price

On the Review Flight page:

- Get the taxes & fees (\$57.91)
- Verify that the total cost is equal to the sum of departure cost, return cost and taxes & fees (\$510.91)

Total \$510.91

| | |
|-----------------|----------|
| 1 adult (18-64) | \$453 |
| Taxes and fees | \$57.91 |
| Total | \$510.91 |

Total \$510.91

Continue

Your account may give you access to travel benefits. Sign in | Compare benefits

Trip summary

Tue, Apr 25, 2017

| Economy | Economy + | Economy ++ |
|----------------------|-----------|------------|
| No additional charge | + \$144 | + \$240 |

















Flight 1 Boeing 757-200

Revise flight | Details | Seats

Old-school KDT frameworks

In some of the famous old-school KDT frameworks, here's how to automate this test scenario:

- Noun-based Keyword Approach. You'd typically construct your test by dragging-n-dropping noun-based keywords from a pre-defined library to a TestCase. Each TestStep in a TestCase is a keyword call. And each keyword represents a window or a web page.
- Direct Keyword-Code Mapping. Inside one keyword, you cannot call other keywords. Only TestCase can contain keyword calls. This is because each keyword is tied to a piece of code and that piece of code is not exposed to you for modification.

| Test case #1 | Value | Action | Note |
|---|-------------------------------|-----------|-------------------------------------|
|  <i>home page</i> | | | |
|  round trip radio btn | {click} | input | |
|  one-way radio btn | - | - | |
|  departure airport txt | SFO | input | |
|  arrival airport txt | JFK | input | |
|  departure datepicker | 25-Apr-17 | input | |
|  return datepicker | 30-Apr-17 | input | |
|  passenger cbx | - | - | |
|  class cbx | Economy | input | |
|  search button | {click} | input | |
|  <i>departure page</i> | | | |
|  price table | | | |
| \$firstRow | | | |
| \$cell <8> | >> dep cost | get value | Example depart cost to be \$ 266 |
| \$cell <8> | {click} | input | |
|  <i>return page</i> | | | |
|  price table | | | |
| \$firstRow | | | |
| \$cell <8> | >> ret cost | get value | Example return cost to be \$ 187 |
| \$cell <8> | {click} | input | |
|  <i>review flight page</i> | | | |
|  summary table | | | |
| \$row <3> | | | |
| \$cell <2> | >> taxes | get value | E.g. \$ 57.91 |
| \$firstRow | | | |
| \$cell <2> | # dep cost + ret cost + taxes | verify | Expected total cost to be \$ 510.91 |

Advantages of old-school KDT approach

Admittedly, the old-school KDT approach is still more convenient than the early-days automation technique, mostly because:

- Methods like input, get value, and verify have been implemented so you don't have to code them yourself. The underlying technology specifics of these methods are hidden from view so that you can focus on writing the tests. This is the platform-agnostic criterion we discussed earlier.
- The framework "scans" the app's GUI to automatically generate the keywords for you, e.g. home page, departure page, return page, etc. This scanning feature is useful to speed up test development. If your app under test is not complicated and your business logics are straightforward, you can finish test automation in the matter of days.
- You can reuse keywords like home page in another test scenario without duplicating any code. Just drag-n-drop them into the next test scenario. *Reusability* boosts test development speed.

Disadvantages of old-school KDT approach

Albeit, the old-school KDT approach poses several setbacks:

Problems resulting from Noun-based Keyword Approach

| | |
|------------------------|---|
| Readability | <p>As you might notice, each step of the test case is a noun – name of the web page. If you have to execute the test manually, you don't actually know what to "do". Noun-based keyword approach hinders the test's readability. We have to eye-witness the app's GUI in order to understand the test. What if during the early phase of development, the developers haven't finished the app GUI? In such case, testers usually have to wait for the GUI to "stabilize" before they can start automating. Testing the business logics alone from Day-1 is impossible.</p> |
| Maintainability | <p>Let's say all of the sudden, business requires that the departure page and return page must be combined into one page. How would we handle that? Normally we have to:</p> <ul style="list-style-type: none"> • Capture a new keyword for the new combined page, e.g. travel dates page • For every test scenario that previously went through the Departure and Return pages, we have to replace departure page and return page keywords by one travel dates page keyword. <p>Needless to say, this is a mundane and time consuming task.</p> |

Redundancy

Although you don't need to interact with the whole page, you still have to drag-n-drop the entire keyword into the test case. For instance, in the above example, we didn't need to tick the **one-way** radio button or change the default value ("Economy") of the **class** combobox. But since they exist in the **home page** keyword (in case another **TestCase** needs them), they still appear in this particular **Test-Case**.

*Problems resulting from Direct Keyword-Code Mapping***Abstraction Level Constraint**

Since keywords cannot recursively call other keywords due to **Direct Keyword-Code Mapping**, our tests stay at the low-level. We cannot create a business-process keyword which interacts with different GUIs across the app under test. For instance, you might want to create a keyword namely **rent car** spanning from "Welcome" window to "Select Date" window to "Select Car" window and so on as below.



Doing this is impossible in old-school KDT frameworks. Typically, we have to create 5 keywords for this case.

Technology Dependency

At the higher level, you also duplicate keywords due to different app platforms. For instance, you have to maintain all of these keywords: **HTML5 home page**, **WPF home page**, and **Java home page**, if your app happens to exist on those three platforms.

A Modern KDT Solution**Verb-based Keywords**

To avoid the aforementioned undesirable consequences of the Noun-based Keyword approach, we propose an alternative namely Verb-based Keyword approach. Particularly in this approach, keywords begin with a verb (specifying intent) instead of a noun. Each test step is a call to a verb-based keyword, which follows by several arguments so the test tool knows what to do with the target object in a specific circumstance.

The result is better *readability*. Your tests become friendlier to business testers and domain experts. They can guess what the actions are supposed to perform by merely looking at their names and arguments such as **search flight or select flight**.

Recursive Keyword Combination

To fix **Direct Keyword-Code Mapping's** setbacks, we'd propose a different approach called **Recursive Keyword Combination**. Essentially, it means the inside content of a keyword should be exposed to business testers so that they can "program" it themselves by simply calling other keywords. This frees business testers from depending on automation engineers. Given this power, business testers can take matters into their own hands to create more business-level keywords from Day 1 without waiting for automation engineers or having to eye-witness the app's GUI. All activities, including test designing, keyword implementing, and app developing can be done in parallel.

Additionally, when new test scenarios pop up, business testers are the ones who know best about which keywords should be reused or newly created. Automation engineers only need to focus on the technology specifics of implementing the keywords which the business testers ask for. They don't need to wonder whether creating a certain keyword is necessary or not. This eliminates redundancy, at least at the business-level.

Action-Based Testing™ (ABT)

With the mindset of overcoming old-school KDT's shortcomings while staying loyal to KDT core principles, we put together a methodology namely **Action-Based Testing™ (ABT)**. ABT is the better version of old-school *Keyword-Driven Testing* because it incorporates the aforementioned **Verb-based Keyword and Recursive Keyword Combination** approaches.

The core concept in ABT is Actions – verb-based keywords. Test cases consist of actions. Each action accepts a set of parameters and executes one atomic function, such as clicking on a control in a window. An action is similar to a procedure in *Procedural Programming*. Below is an example of an ABT action:

| | window | control | row | Cell |
|----------------------|---------------|---------------|-----|------|
| Get table cell value | review flight | summary table | 3 | 2 |

As you might notice, window and control are the most important arguments of an action call. In ABT, one window is mapped to one Interface Entity in which each Interface Element is mapped to a control.

Your test scenarios, which share the same test objectives or requirements, are bundled in a container called Test Module. This allows you to design and organize your tests in a modularized fashion. The end goal is an efficient and maintainable KDT framework.

The illustration below describes the relations among Test Module, Action and Interface Entity concepts.

| TEST MODULE | | | ACTION DEFINITION | | | INTERFACE ENTITY | | |
|-------------|--------------------------------|----------------------|-------------------|----------|----|--------------------------|------------------|-----------|
| 1 | Test Module | Action Based Testing | 1 | login | 2 | login | Car Rental-Login | |
| 2 | OBJECTIVES | | 2 | | 3 | interface entity setting | | |
| 3 | test objective | TO 01 | 3 | argument | 4 | user name | textbox | User name |
| 4 | test objective | TO 02 | 4 | argument | 5 | password | password text | Password |
| 5 | INITIAL | Setting up | 5 | enter | 6 | login | button | Login |
| 6 | start application | | 6 | enter | 7 | | | |
| 7 | login | | 7 | click | 8 | | | |
| 8 | TEST CASE | TC 1 | 8 | | 9 | | | |
| 9 | test objective | TO 01 | 9 | | 10 | | | |
| 10 | rent car | first name | 10 | | 11 | | | |
| 11 | check total payment | last name | 11 | | | | | |
| 12 | TEST CASE | TC 2 | 12 | | | | | |
| 13 | test objective | TO 02 | 13 | | | | | |
| 14 | get number of available cars | car | 14 | | | | | |
| 15 | rent car | first name | 15 | | | | | |
| 16 | check number of available cars | car | 16 | | | | | |
| 17 | FINAL | Cleaning up | 17 | | | | | |
| 18 | close application | window | 18 | | | | | |

- 1 'Action-Based Testing' is a Test Module
- 2 'login' is a user-defined action
- 3 'enter' is a pre-coded action
- 4 'password' is the logical name of the password control in the Login form

As you might notice, the inside content of an action is exposed in the spreadsheet format (e.g. login action). This helps business testers instruct the test tool on how to interact with the app-under-test themselves by calling other user-defined or pre-coded keywords (Recursive Keyword Combination).

Example test case in Action-Based Testing

Back to our flight booking example, you can expect the test rewritten in *Action-Based Testing Language* as below.

| TEST MODULE | Flight Booking | | | | |
|---------------------------|-------------------|--|-------------|-----------------|-------|
| OBJECTIVES | | | | | |
| test objective | TO 01 | Verify that the total cost is calculated correctly | | | |
| INITIAL | Setting up | | | | |
| | location | | | | |
| navigate | www.myairline.com | | | | |
| TEST CASE | TC 01 | My first test case | | | |
| test objective | TO 01 | Verify that the total cost is calculated correctly | | | |
| | dep airport | arr airport | dep date | ret date | round |
| search flight | SFO | JFK | 25-Apr-17 | 30-Apr-17 | Yes |
| | window | row | variable | | |
| select flight | departure page | 1 | >> dep cost | | |
| | window | row | variable | | |
| select flight | return page | 1 | >> ret cost | | |
| | window | row | column | variable | |
| get value from summary | review page | β | 2 | >> taxes | |
| get value from summary | review page | 1 | 2 | >> actual total | |
| //// Check the total cost | | | | | |
| | value | expected | | | |
| check value | # actual total | # dep cost + ret cost + taxes | | | |

Advantages of Action-Based Testing

Besides the benefits of Verb-based Keyword and Recursive Keyword Combination we previously discussed, rewriting the test in ABT brings several more advantages:

| | |
|------------------------------------|---|
| Clearer Test Objectives | Looking at the test, we know the exact objective of our test case. All test objectives are defined in OBJECTIVES section at the beginning of each TEST MODULE . We then assign a pre-defined objective to each TEST CASE . In this example, the test objective is: TO 01 - Verify that the total cost is calculated correctly |
| Standardized Test Structure | The INITIAL section contains all pre-condition steps needed to run the ensuing test cases. In this example, we navigate to our web page at www.myairline.com . It's a common practice to wrap pre-recondition steps into a user-defined config action for better reusability across test modules. |
| Technology-Independent | We can execute the same set of tests across multiple platforms. Platform-specific parts (technology layer) shouldn't affect our business-level test flow. It'd be a nightmare if we have to duplicate one test flow for each app platform and maintain them all over the years. In ABT, if the app technology happens to change (e.g. from web to mobile), we don't need to update the test flow. We just go ahead and update the interface entities containing the definitions of the app's GUI. |

Better Maintainability

In the event of the aforementioned GUI change (**departure page** and **return page** are merged into **travel dates page**), we just need to replace the window arguments of the two **select flight** actions, like this:

| | window | row | variable |
|---------------|-------------------|-----|-------------|
| select flight | travel dates page | 1 | >> dep cost |
| | window | row | variable |
| select flight | travel dates page | 1 | >> ret cost |

Of course, the biggest difference is that our test flow remains intact (still two **select flight** test steps). In the **Noun-based Keyword** approach, two test steps are replaced by a new one in each and every test case concerning the two web pages.

Methodology should be embedded in a product

Up to now, we've seen the solutions for Direct Keyword-Code Mapping and Noun-based Keyword problems. But, we haven't discussed the solution for the coding requisite, which exists in both in-house KDT frameworks and many off-the-shelf KDT solutions on the market. We believe solving this problem requires more of a product solution than a methodology solution. To eliminate unnecessary coding and accelerate test development, we need a pre-coded keyword library.

[TestArchitect™](#) is the test automation framework embracing Action-Based Testing. It's shipped with features specifically designed to embody and empower ABT's ideology. One ought to agree that great ideas should be accompanied by a great product to actually close the gap between theory and practice.

To better understand the values which *TestArchitect™* can bring to the table, let's explore several kinds of Actions in *TestArchitect™*.

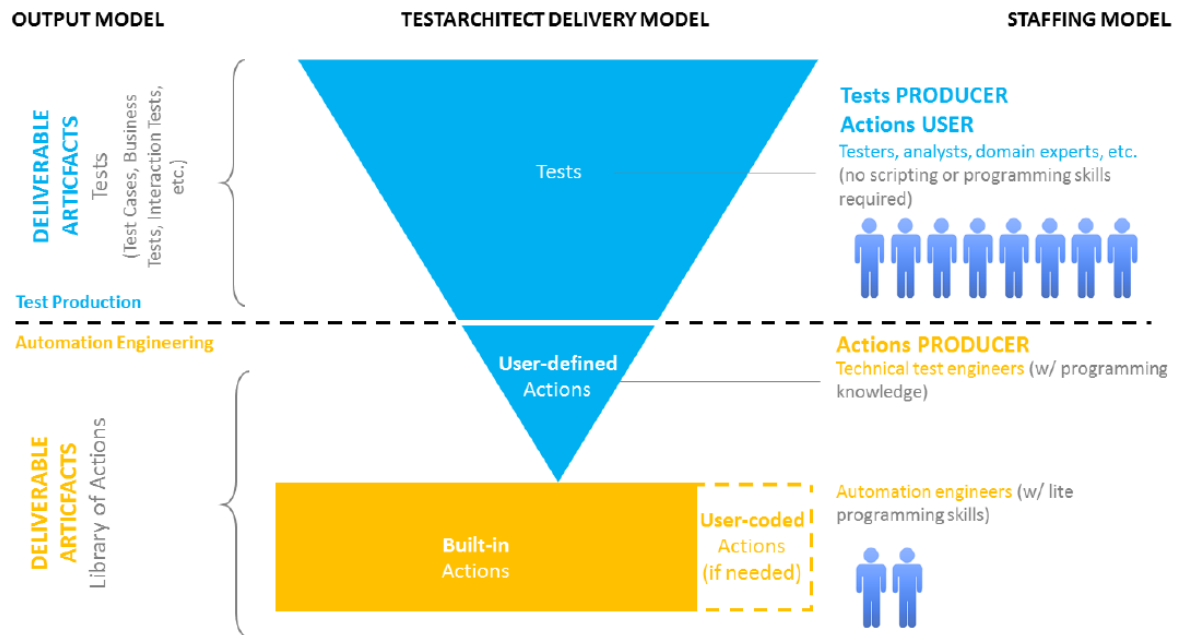
Built-in Actions: These are previously coded keywords. *TestArchitect™* offers an out-of-the-box action library (386 actions and counting) which cover most of day-to-day needs such as GUI interaction, API, database, image-based, etc. These actions were purposefully designed to be used right away without any further coding required.

User-defined Actions: Since the contents of actions are exposed in the spreadsheet format, testers are free to combine built-in or existing user-defined actions to create new user defined actions recursively (Recursive Keyword Combination).

Scripted Actions: When necessary, any team member with some coding background can implement actions in a programming language of choice (C#, Java or Python).

The biggest advantage of *TestArchitect™* comes from Codeless-ness. As you probably notice, a pre-coded action library is out-of-the-box so we mostly don't have to code. More importantly, those keywords are platform-agnostic: a click is just a click no matter if its target is an HTML5 element, an Android / iOS native control, WPF button or Java checkbox.

Codeless-ness is desirable because it enables us to flexibly leverage smaller programming staff to support a much larger non-programming staff. This delivery model can churn out more tests in less time because our team members can all work in parallel.



The combo of Action-Based Testing & TestArchitect™

We have seen how Action-Based Testing along with *TestArchitect™* inherit the KDT legacies we already know and love, such as *reusability*, *maintainability* and *division of labor*. Let's take a look at the how this combo would resolve other shortcomings of old-school KDT frameworks. We're also interested in tackling the large-scale challenges when our project grows.

| Shortcomings of old-school KDT frameworks | Solutions in Action-Based Testing & TestArchitect combo |
|---|--|
| Slow ROI | <p>Thanks to Recursive Keyword Combination, <i>reusability</i> in ABT has been improved significantly compared to old-school KDT frameworks. Business testers can start writing tests early in the sprint based on their business logic knowledge without waiting for the app GUI or the actual action implementation.</p> <p>As a result, test development's speed reaches its peak faster, which means you achieve the break-even point faster.</p> |

| | |
|--|---|
| Big upfront investment in architectural design | <p>In the TestArchitect framework, architectural decisions have already been made with large-scale problems in mind. For instance:</p> <ul style="list-style-type: none"> • Client-server architecture. You can easily share your test assets with other team members through a centralized TestArchitect Repository Server. Test assets are also stored under version control. • Important types of assets are abstracted and decoupled. For instance when we need to modify interface entities, we don't have to update actions. Refactoring capability is built-in. • If the test team is dispersed across multiple geo-locations, we can leverage Repository Replication to increase access speed and ensure service availability (with backup servers). • If many test teams are working on multiple projects, one test team can reuse test assets from another team's project through Project Subscription. |
| Steep learning curve | <p>A test is a sequence of business-readable actions written in spreadsheet format so domain experts can understand it very quickly. No need for any further interpretation.</p> <p>New automation engineers can start automating from Day 1 without too much training required thanks to codeless-ness.</p> |
| Hard to expand to other app platforms | <p>Major app platforms such as native <i>Windows, WPF, Java, HTML5, Android, iOS, CLI, image-based, database, web services, etc.</i> have already been covered. See Datasheet for more information.</p> <p>In case the targeted app platform belongs to a niche market, automation engineers can extend TestArchitect at many levels: <i>Technology, Action, and Harness</i>.</p> |
| Costly to add new test utilities | <p>Common test utilities such as <i>Lab Manager, Dashboard, Reporting, Parallel Execution, Mobile Cloud Execution, etc.</i> are out-of-the-box. See Datasheet.</p> |

Conclusion

As you may have realized, **Keyword-Driven Testing** beats the early-days test automation technique significantly. But modern test automation needs (i.e. Agile) stretch the old-school KDT frameworks past their limits. New challenges expose certain shortcomings of the old-school KDT frameworks. Most significantly, the biggest shortcomings are **Noun-based Keyword** and **Direct Keyword-Code Mapping**. Their consequences are so severe that we need to step back and rethink our approaches. **Verb-based Keyword** and **Recursive Keyword Combination** have emerged as the better alternatives for those shortcomings of old-school KDT frameworks.

Action-Based Testing™ incorporates those great ideas into one methodology. It offers a modern approach to the *Keyword-Driven Testing* ideology which we all know and love. Along with the *TestArchitect™* framework, ABT establishes an empowering platform to close the gap between theory and practice. With proper adoption, you can expect true Test Quality at Speed.

For more information on how to leverage TestArchitect™ as an automation solution, please visit www.testarchitect.com for a [free download](#). For a more in-depth guide on how you may adopt TestArchitect™ and ABT, check out the [TestArchitect Implementation Guide](#).

ABOUT LOGIGEAR

LogiGear is a boutique software testing company and the developer of TestArchitect™. With a codeless approach, TestArchitect™ helps improve automated testing in Agile development by enabling early test development, reducing the time to create and maintain reliable test automation. With more than two decades of serving software businesses and software testing community, LogiGear also offers outsourced software testing services, as well as QA transformation consulting and on-site training to help achieve the most ambitious business goals. To learn more visit www.logigear.com