December, 2008

printed in Germany

testing experience

The Magazine for Professional Testers

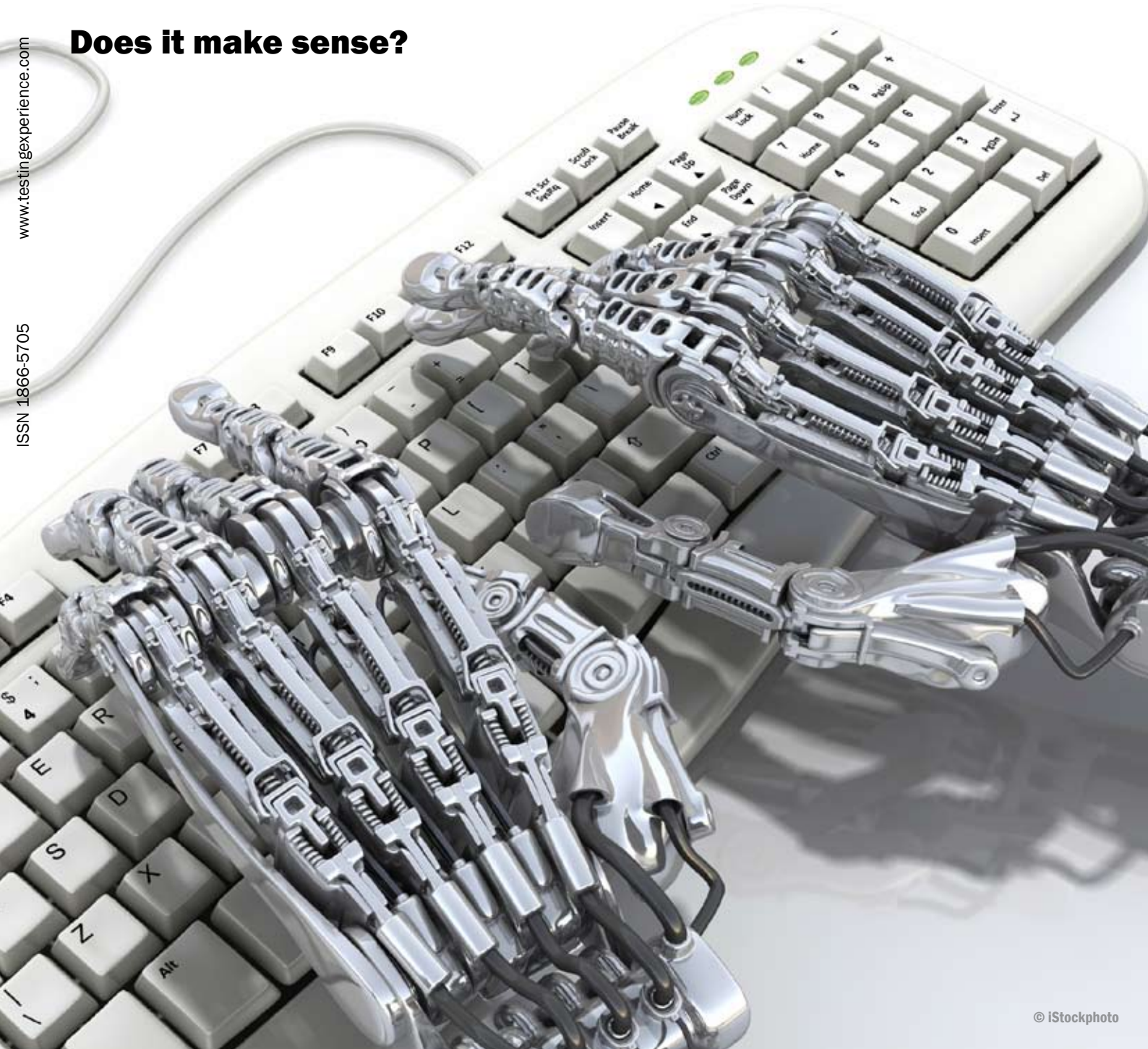
free exemplar

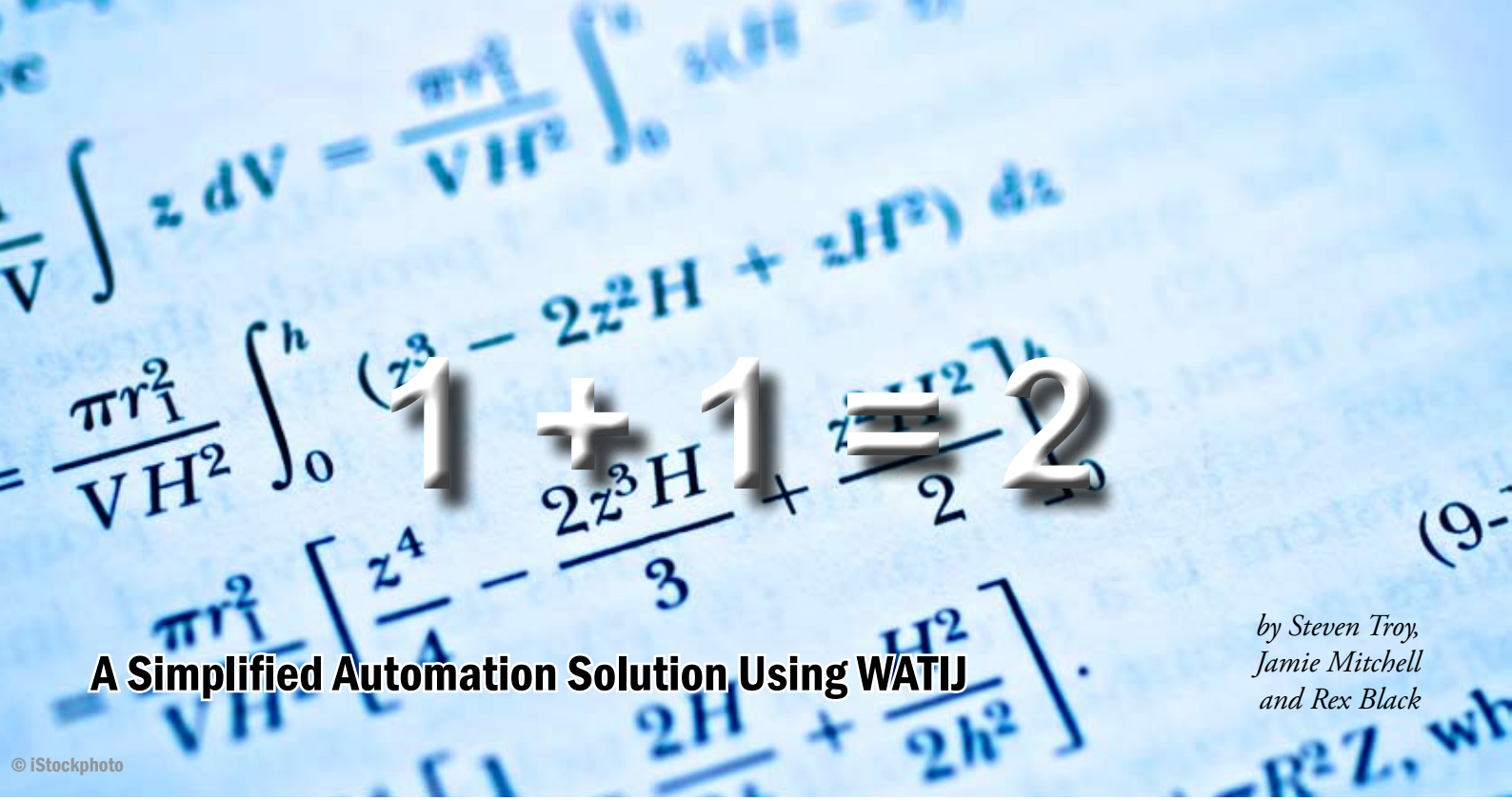
www.testingexperience.com

ISSN 1866-5705

Test Automation -

Does it make sense?





A Simplified Automation Solution Using WATIJ

by Steven Troy, Jamie Mitchell and Rex Black

Regression is a serious risk for software. Unlike physical structures such as bridges, skyscrapers, and submarines, software is quite sensitive to small changes, which can result in serious, unpredictable side-effects, often functionally distant from the change that was made. Because of this, test teams use regression testing to mitigate the risk.

Unfortunately, the scope of regression testing grows in a non-linear fashion over the life of an application. While the scope of a maintenance programming activity—say, to release a few bug fixes and features for a successful existing application—is linear in relation to the number of bug fixes and features, the scope of the regression testing includes all the bug fixes and features ever done.

This leads many organizations to try to automate regression testing. Sadly, many organizations fail in these attempts, either because of high initial costs or because of maintainability problems with their regression tests. Lack of good design for the automated test frameworks and bad expectations on the part of people involved often contribute to these failures.

Understanding Our Regression Testing Problem

In this article, we'll talk about how we implemented automated regression testing in a maintainable, low-cost way for a widely-used product called CA Service Desk Manager. This popular, versatile, and comprehensive IT support system is browser-based. Therefore, we crafted an extensible automation architecture for CA Service Desk Manager that we can use in a wide variety of our future quality assurance needs. We are still developing this test system, but we already have over 2000 test cases running dependably and we are adding more daily. We call our solution the Service Desk Manager Automation Framework Code.

CA Service Desk Manager is a business critical, high-availability product. Customers and users insist on the utmost reliability. Downtime costs money. Our QA group, responsible for regression testing this system, previously had relied on over 13,000 manual test cases to ensure high quality during maintenance testing. We needed a new approach to test this application, since each maintenance release and patch required a tremendous manual test effort. As mentioned above, we knew the amount of regression testing effort would only increase.

While it was clear that test automation would need to be part of the solution, we knew that automation raised a number of problems on its own. We'd had a lot of past experience with automation, mostly unsatisfactory. The supposedly simple tools were not simple! The record/playback tools that supposedly allowed testers without programming skills to easily create, run and maintain libraries of automated test cases didn't deliver on that promise.

Our challenge was to come up with a solution which we could support, extend and utilize without an exorbitant amount of effort. In addition, many of our testers do not have formal programming experience; they are domain experts who came from the business side of the organization. Replacing established testers with programming-skilled tester who could directly script tests was not an option, nor was trying to morph all of our testers into programmers.

We understood that we would still need automators - developers who were experienced with programming languages and associated tool sets. We decided to utilize our automators to create a meta-language, to use an advanced tool set to build a simpler tool set that our testers could use directly. Done right, a few automators should be able to support an unlimited number of testers. Since the testers would build the needed assets, the test cases, we saw this as a way to give our solution scalability. The growth of our test cases would not be limited by the number of automators but by the number of testers; this mirrors manual testing. Essentially, we wanted to use our automators to add complexity to the heart of our test system to make the interface of it simpler. In other words, we would design our test system for usability by the manual testers, following the Keep It Simple, Stupid (KISS) principle.

Selecting the Right Tool

This principle led us to the following mandates for our test framework:

- Make it relatively easy to develop, since we don't want to build a more complex test framework than the system under test (SUT).
- Make it easy to maintain.
- Make it easy to use by any tester. (We joked that it needed to be easy enough for a manager to use.)
- Make it powerful and efficient enough to automate a large number of test cases within a reasonable time.

For the heart of our system we needed a toolset for the automators to use. We considered both commercial record/playback tools with framework tools that sit on top of them and open source scripting and automation test tools. We quickly came to prefer a tool called Web Application Testing In Java (WATIJ.) This tool is an application programming interface (API) wrapped around the Java language, expressly created for the automated testing of web applications through Internet Explorer (IE.)

Since this tool would put Java at our automators' fingertips, we would have access to an enormous number of existing libraries and tools that our automators could use. A large benefit of this synergy would be that our test system would not be restricted to only testing

web applications; we could use it in the future to expand to related areas that need testing. For example, we would be able to write a test that accesses a web service directly, calling on it to do some processing. We could then use WATIJ to bring up CA Service Desk Manager to validate that the processing behaved as expected. In addition, since parts of our system under test are written in Java, WATIJ would fit in nicely to our existing environment.

We have to perform much of our testing at the graphical user interface (GUI) level, so our framework needed to interact directly with the CA Service Desk Manager interface. Since WATIJ only supports IE, this limited our automated testing. We decided, however, since most of our testing was already done in IE, we could live with this constraint. There is reason to believe that, in the future, WATIJ will also directly interact with Firefox and other popular browsers. Until then, we could automate the majority of our tests using IE and then use a subset of our manual tests to validate CA Service Desk Manager on other browsers.

Another limitation that initially concerned us was the lack of record/playback capability in WATIJ. Upon reflection, we realized that as long as our automators could supply abstract interfaces (essentially keywords) to our testers, this constraint would not reduce the benefits we would obtain.

Using WATIJ, we would be able to simulate human user interaction with the SUT. This includes simple tasks that any automation tool could do, including page navigation, clicking on links, filling out forms and validating form content. An advantage of WATIJ is that it also facilitates much more complex actions, such as file uploading and downloading, handling popup windows, dialog manipulation, and screen captures.

Our automation framework needed to be able to find disparate HTML elements on a page. Many of the tools that we investigated did not handle this sometimes cumbersome task well. We were delighted to find that WATIJ included a powerful element finding capability that can find, access and control any HTML element easily. It even supports XPath expressions that are lightning-fast.

A final concern was the programming environment for our automators. Because it is configured as a standalone API, WATIJ programmers are not forced into using a proprietary testing framework or special integrated development environment (IDE.) Instead, we could integrate WATIJ into almost any popular testing framework we wanted, including JUnit or TestNG. As WATIJ works like any other Java library, it could also be imported into various popular IDEs, including Eclipse and IntelliJ. This easy integration should ensure high productivity immediately without special training or ramp-up time for our automators.

We finally decided to use WATIJ rather than other open-source or vendor tools based on the following considerations:

- WATIJ is open-source with a robust community of users.
- WATIJ is extremely fast to write as well as execute.
- WATIJ provides automatic synchronization, since WATIJ waits until a page download is complete rather than trying to manipulate partially downloaded pages.
- WATIJ is easy to use.
- WATIJ is Java-based, giving us the integration benefits noted above.

At this point, we believed WATIJ would provide the right capabilities for our test framework.

Implementing the Framework

Having picked our tool, we understood that we were only half-way there. We had a solution for our automators, but nothing yet for our testers to use. Our next step was to determine which low-level GUI interactions were the most common. By understanding the GUI domain, we could prioritize the logical automation capabilities we needed to build into our framework.

We split our GUI domain into two separate sets of tasks. First, we isolated general low-level control interaction tasks that testers would need in any browser application. These included:

- Click on a link
- Click on a button
- Set a text field
- Set/unset a radio button
- Check/uncheck a check box
- Select a value from a drop-down combo or list

We decided that these tasks could essentially be distilled down into two abstract functions:

- setItem()
- clickItem()

When called, these two functions would "do the right thing" based on the control passed to it. Remember, we were creating a simplified language (called the Service Desk Manager Framework Code) that non-technical testers would use, directly creating automated tests. Wherever possible, we wanted to avoid any low-level commands in this language, but we knew we would need them sometimes.

A common task that needs to be performed is logging into a web site. As an example, consider the Yahoo login shown in Figure 1.

Figure 1: Web site login

The manual tasks that a user needs to perform to login are:

1. Bring up a browser window
2. Set the correct URL and open that page
3. Wait for the page to load and stabilize
4. Type the user name into the ID field
5. Type in the password to the password field
6. Click on the Sign In button

The following programmer-written WATIJ code directly performs these tasks:

```
IE ie = new IE();
ie.start("www.mail.yahoo.com");
ie.textField(SymbolFactory.id,
"username").set("Harry_Potter");
ie.textField(SymbolFactory.id,
"passwd").set("Alohomora");
ie.button(SymbolFactory.id,
".save").click();
```

Here are the simplified commands that the tester would write:

```
start("www.mail.yahoo.com");
setItem("Textfield", "username",
"Harry_Potter");
setItem("TextField", "passwd",
"Alohomora");
clickItem("Button", ".save");
```

Because the system automatically handles synchronization, the tester need not worry about it.

However, the above example does not really show the simplification that we get from this framework. We get much more leverage when dealing with more complex tasks that are specific to the SUT, CA Service Desk Manager. This is the second set of tasks that we needed to perform. Each task is encapsulated in a keyword-like method written as part of the Service Desk Manager Framework code. Examples include the following:

- clickNode() // Click on Any Node in the User Interface
- login() // Login to Service Desk Manager
- logout() // Log off of Service Desk Manager
- alertMessage // retrieve a message from alertmessage
- isLookup // checks whether a link is a lookup or not

Let's look at an example of how this works. Figure 2 shows the screen for logging into the CA Service Desk Manager.

Figure 2: Logging into the CA Service Desk Manager

Consider performing the following tasks against this interface:

- Open a browser window
- Bring up and initialize the CA Service Desk Manager application
- Log into it
- Move to the Administration tab
- Select a node in the tree

Here is the WATIJ code as created by a programmer:

```
IE ie = new IE();
ie.start("http://servicedesk:8080");
ie.textField(SymbolFactory.name,"USERNAME").set("Harry_Potter");
ie.textField (SymbolFactory.name,"PIN").set("Alohamora");
ie.button(SymbolFactory.name,"imgBtn0_button").click();
ie.frame(SymbolFactory.name,"toolbar").link(SymbolFactory.id,
"tabhref2").click();
ie.frame(SymbolFactory.name,"product").frame(SymbolFactory.
id, "tab_1004").frame(SymbolFactory.id,"role_main").
frame(SymbolFactory.id,"MenuTree").frame(SymbolFactory.id, "frmAd-
mTree").div(SymbolFactory.id,"scrollbarDiv0"). div(SymbolFactory.
id,"divTreeNode1"). link(SymbolFactory.title, "Security and Role
Management").click();
ie.frame(SymbolFactory.name,"product").frame(SymbolFactory.
id, "tab_1004").frame(SymbolFactory.id,"role_main").
frame(SymbolFactory.id,"MenuTree").frame(SymbolFactory.id, "frmAd-
mTree").div(SymbolFactory.id,"scrollbarDiv0"). div(SymbolFactory.
id,"divTreeNode1"). link(SymbolFactory.title, "Contacts").click();
```

And, here is a simple series of commands, using the Service Desk Manager Framework, that a tester would write:

```
Start("http://
servicedesk:8080");
Login("Harry_
Potter","Alohamora");
clickItem("toolbar","Link","tabh
ref2"); //Administration Tab
clickNode("Security and Role
Management->Contacts");
```

Once again, the framework handles the identification of GUI elements and synchronization, freeing the testers from having to deal with that complexity. They simply write short declarations of abstract tasks performed in the same order that they would have performed manually.

A tester can also use the text of the link or button to identify the GUI element as well. In the above example, to open the Administration tab, the tester could have written:

```
clickItem("toolbar","Link","Adm
inistration");   //Administra-
tion Tab
```

Some interactions with the interface are not necessarily tied in with specific GUI elements. In these cases, our framework allows the tester to submit keystrokes the same way a manual tester might do manually. For example, to open the file menu of the browser, a manual tester might simply key in Alt-F. In our framework, we can perform exactly the same action by typing in the following statement:

```
sendKey("%f")
```

Our framework now had everything we needed for a tester to create a test case that initialized the SUT and cause it to perform a series of actions, driving through the test case. What we were still missing, however, was a way to make it a real test. Each test, to be valid, must have a way to compare actual behavior with expected behavior.

To that end, our automators have added to the framework numerous validation methods. For example, suppose we want to check to ensure that a field is set to a specified value after the test is run. The actual WATIJ code we use is:

```
public static String getCell-
Value_id(String _framename,
int _table, String _id) throws
Exception
{
String ret = null;

try{
if(((HtmlElement) ((HtmlElement)
findFrame(ie,_framename).table(_
table)).cell(SymbolFactory.id,_
id)).exists())
{
ret=(String) ((HtmlElement)
findFrame(ie,_framename).table(_
table)).cell(SymbolFactory.id,_
id).text();
}
else
{
_teststatus = "fail";
}
catch(Exception e)
return ret;
}
```

This code is then encapsulated in the Service Desk Manager Framework as follows:

```
Get_value=getCellValue_id("cai_
main", "dtltbl0",df_0_2").trim()
.equals("3"));
```

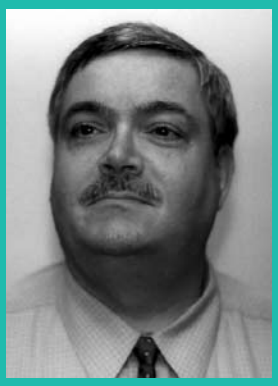
As you can see, this simple command provides our testers with access to a powerful capability without having to master the programming complexity underneath that capability.

Conclusion

While we are still implementing our framework, we can already say that our overall approach to automation using WATIJ as the backbone of our framework has been a great success. We have already automated over 2,000 test cases, fully 15% of the total we will need and a large enough sample to prove the soundness of our approach. We are very pleased with the work of Margie Studer, Chris Whorley, Hari Maddela and Sujit Karri, the architects who made the Service Desk Manager Framework a reality. We are also very happy with the accomplishments made by our entire QA team, who are successfully using framework.

Biography

Steven Troy is a Senior Director of Quality Assurance for CA, Inc. (www.ca.com), the world's leading independent IT management software company.



Jamie Mitchell of Rex Black Consulting Services is a senior consultant specializing in testing, training and automation.

Rex Black is President of RBCS (www.rbcs-us.com), a worldwide leader in testing services, including consulting, outsourcing, assessment, and training.