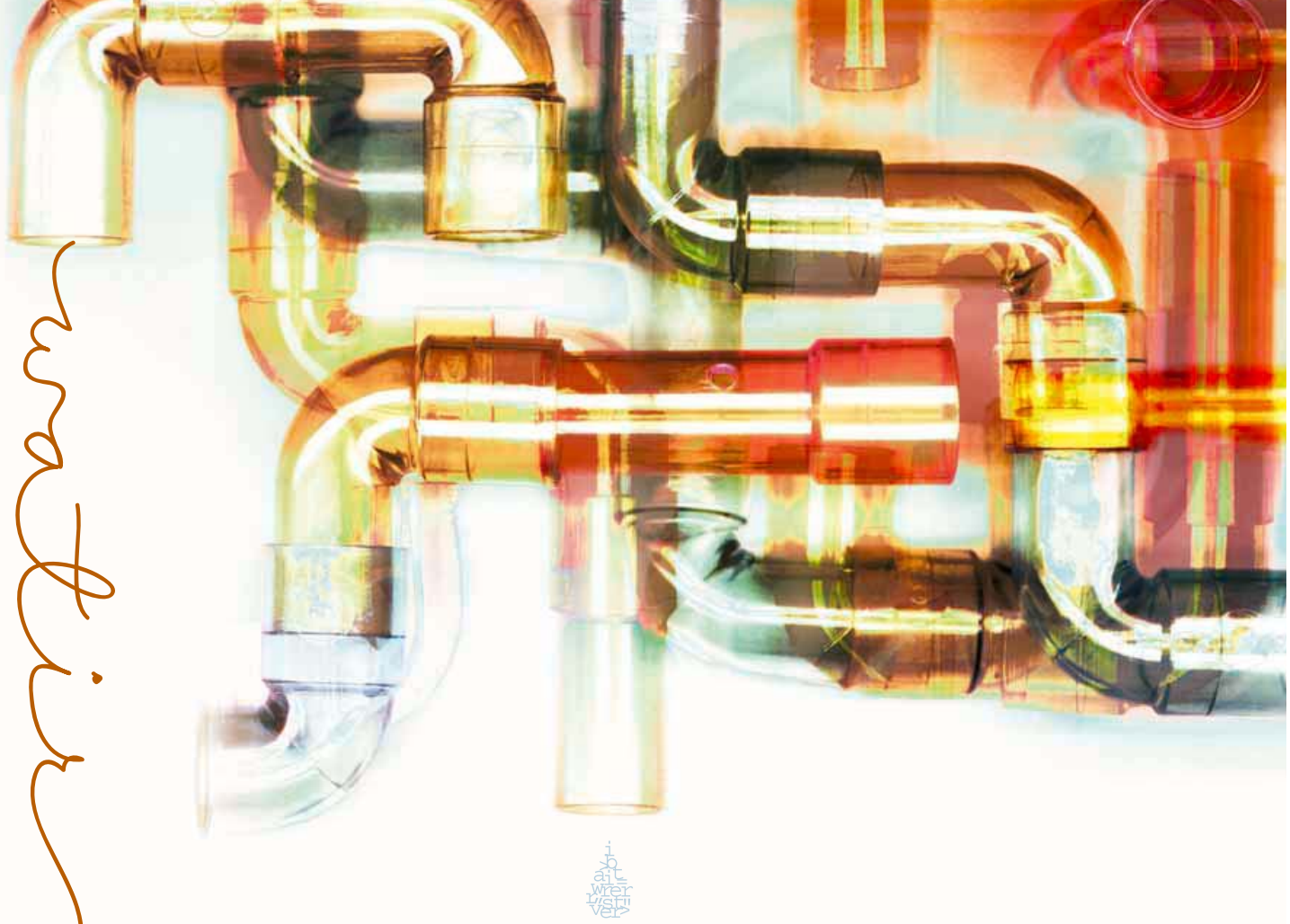

watirmarks

Test automation of Web applications can be done more effectively by accessing the plumbing within the user interface. Here is a detailed walk-through of Watir, a tool many are using to check the pipes.

by Jonathan Kohl and Paul Rogers

As Web applications become more popular and increasingly more complex, the need for test automation of these applications grows. Automating tests through a GUI is difficult and can be costly. The most popular methods of automating using a GUI are to use window coordinates to click items or to create and use references to window objects. The first method relies on locations of objects not changing on a page; the second usually relies on a type of proprietary object mapping format where all Web page objects must be captured and saved prior to scripting. Another approach is to seek out and use testable interfaces that are built into Web browser applications and provide the ability to directly access the objects on the page as they appear in the Web browser.

In the case of Internet Explorer, there is a published, maintained testable interface in the form of the Component Object Model (COM) interface. This interface provides almost full control of the Internet Explorer Web browser and allows access to the objects in Web pages presented in the Web browser. Most scripting languages have extensions that allow them to "talk to" Internet Explorer. If this sounds like a lot of work, don't worry. Other people have developed toolkits for you.

The Web Application Testing in Ruby (Watir) project is one such toolkit. Watir is built on the object-oriented scripting language Ruby. Its developers have used it for large-scale system testing, functional testing, and for automating user acceptance tests. Watir uses a real programming language, is a free open source product, and allows direct control over objects such as HTML and JavaScript in a Web page. There are no layers between the HTML and your test scripts—you can create your own if you want—and no license fees. At the time of writing, the Watir tool only supports Internet Explorer, but there is work underway to support other browsers.

Enough Background. I Want To Test!

Let's start with an example. You'll need to have Ruby and the Watir toolkit installed on your machine. (See the Sticky Notes for instructions. We used Watir version 1.0.3 for this article.) You don't need to know how to program in Ruby to start learning, but it is a good idea to have a basic understanding of HTML.

Ruby provides a command interpreter called the Interactive Ruby Shell. We will use this tool to test-drive our examples. To begin, open a Windows command prompt, type: **irb**, and hit Enter. This will start up the Interactive Ruby Shell. You will see a command prompt something like this:

```
irb(main):001:0>
```

This is the prompt for the IRB program. We'll abbreviate the prompt to **irb>** in this article. We can type Ruby scripting code here, and the interpreter does its job and prints out a value for us.

To add the numbers 2 and 2, we enter this in IRB and hit Enter. Note that what the user types is in bold type:

```
irb> 2 + 2
=> 4
```

IRB evaluated the expression and printed the result.

In Ruby, everything is an object. This is important because to automate Web applications with Watir, all we have to do is identify objects and send them messages.

Here's an example using a string object:

```
irb> "watir".reverse
=> "ritaw"
```

Ruby syntax uses a "dot" to separate objects and messages. In this example, we created a string object "watir", and we sent it a message "reverse". The Ruby interpreter evaluated this for us and printed the result, "ritaw". Watir also uses this notation to interact with objects on a Web page. (See the StickyNotes for more on objects and messages).

A Google Web Search

If we open our Internet Explorer Web browser and navigate to the Google home page, we can start designing a simple test. We manually use Google to search for documents on the Web, and our example test case will follow what we do as users. In this test, we will search for a Ruby development book known as the "pickaxe." Run through this test case manually first:

Steps:

1. Go to the Google home page in the Internet Explorer browser.
2. Enter "pickaxe" in the search text field.
3. Click the Google Search button.

Expected Result:

A Google page with search results and a link to the book *Programming Ruby, 2nd Ed.* should be shown.

Now that we understand what we want to automate, we can start using Watir. To start exploring our test case, we need to tell IRB to use the Watir library:

```
irb> require 'watir'
=> true
```

IRB returns "true" to tell us that it has successfully loaded the Watir library into memory.

Now let's start up Internet Explorer using Watir:

```
irb> ie = IE.new
=> #<IE:0x2df6358 @form=nil, @defaultSleepTime=0.1,
@ presetFrame="",
@ie=#<WIN32OLE: 0x2df6310>
@activeObjectHighLightColor="yellow", @frame="",
@logger=nil, @typingspeed=0.08>
```

By sending the "new" message to the IE object, we create a new instance of Internet Explorer, which we now can refer to in our IRB session as **ie**. A new instance means that a new Internet Explorer browser opens and is ready to accept commands. If we send messages to **ie**, Watir sends the corresponding commands to Internet Explorer. What IRB returned to us in the **ie** assignment above isn't important right now; it simply tells us the attributes of our new object.

Step 1

To satisfy our first step, we must direct our Web browser instance to the Google home page:

```
irb> ie.goto("http://www.google.com")
=>1
```

This sends the "goto" message to the **ie** object with the URL as a parameter. The Web browser should now be on the Google home page.

Exploring Objects

To search on Google, we need to enter a search term in the text field and then click the Google Search button. Let's repeat our manual search for the Programming Ruby book, this time using IRB. To do this, we'll need to enter text in a text field and push a button.

To manipulate the objects on the Google page, we need to learn something about them. One way is to right click and View Source to see the page source (JavaScript, HTML, etc.). This can be hard to read, so Watir provides a way to view these objects.

We can use a Watir method called `showAllObjects` to display all the objects on the Web page. We're only interested in the objects we need to use for our test, so this is a good way to find them.

```
irb> ie.showAllObjects
---------- Objects in page -------------
text    name=q    id=  value=                  alt=  src=
submit  name=btnG id=  value=Google Search     alt=  src=
submit  name=btnI id=  value=I'm Feeling Lucky alt=  src=

=> nil
```

The `showAllObjects` method returns all the objects on the Web page. The first column shows what kind of object it is; the "`name`", "`id`", "`value`", "`alt`", and "`src`" column prints out the attributes of that object. These attribute values are defined in the HTML page source. For example, the definition of the text field looks like the following:

```
<input maxLength=256 size=55 name=q value="">
```

You can see where the name attribute comes from and that `showAllObjects` doesn't show all attributes (only the ones that are likely to be useful in tests).

Notice some of the entries don't have values. That's because the HTML didn't mention them (or in the case of **value**, gives it a value that means "no text"). To save space, we are only showing the objects that we know we're interested in: text fields ("`text`" in the first column) and buttons ("`submit`" in the first column). You will see more rows describing all of the page objects displayed on your screen.

When there are several objects of the type you're looking for, it can be hard to be sure you've picked the right one. A Watir message called "flash" will cause its recipient to be highlighted in yellow and blink ten times. If we find multiple objects that look like good candidates, we can flash each one of them and pick the one we want before writing our test.

There is only one text item, so that's the one we'll use. To be absolutely sure we've identified the right object, let's send a flash method to it. Watch what happens in your Web browser:

```
irb> ie.textField(:name, "q").flash
=> 10
```

The text field on the Google home page flashed ten times. Now we know we have the correct text object, and we can successfully send messages to that object using Watir. We can try the same thing with a button object that looks like a likely candidate:

```
irb> ie.button(:value, "Google Search").flash
=> 10
```

The Google Search button should flash yellow ten times. If we look at the page source, the HTML tag looks like this:

```
<input type=submit value="Google Search" name="btnG">
```

We could use either the value attribute or the name attribute. For readability, we'll use the value attribute in this test case. For non-English versions of Google, using the name attribute would be a better option because the value will be different.

Now we are ready to proceed. Both of the objects we need for the second and third steps in our test case are available to us using Watir. The flash method has worked on the attributes we have used to uniquely identify the objects we need. If we had tried to access an object with the wrong attribute, we would get an error. For example, if we mistyped an attribute:

```
ie.button(:value, "Gogle Search").click
```

we would see an error like this:

```
UnknownObjectException: Unable to locate object, using
value and Gogle Search
```

Step 2

To satisfy the second step, we can enter our search parameter into the text field we identified using the Watir "`set`" method:

```
irb> ie.textField(:name, "q").set("pickaxe")
=> "pickaxe"
```

The text field will now contain the word "pickaxe". We identified the text field on the Google home page by using the name attribute in the HTML for the text field. Google calls this field "q", so we tell Watir to find the text field with a name attribute "q" and to set the text field to the value "pickaxe".

Step 3

To satisfy the third step, we need to click the Search button:

```
irb> ie.button(:value, "Google Search").click
=> nil
```

A Google Search page should now be returned with "Programming Ruby, 2nd Ed." high up on the results list. This is the link to the site with the second edition of the excellent book *Programming Ruby*.

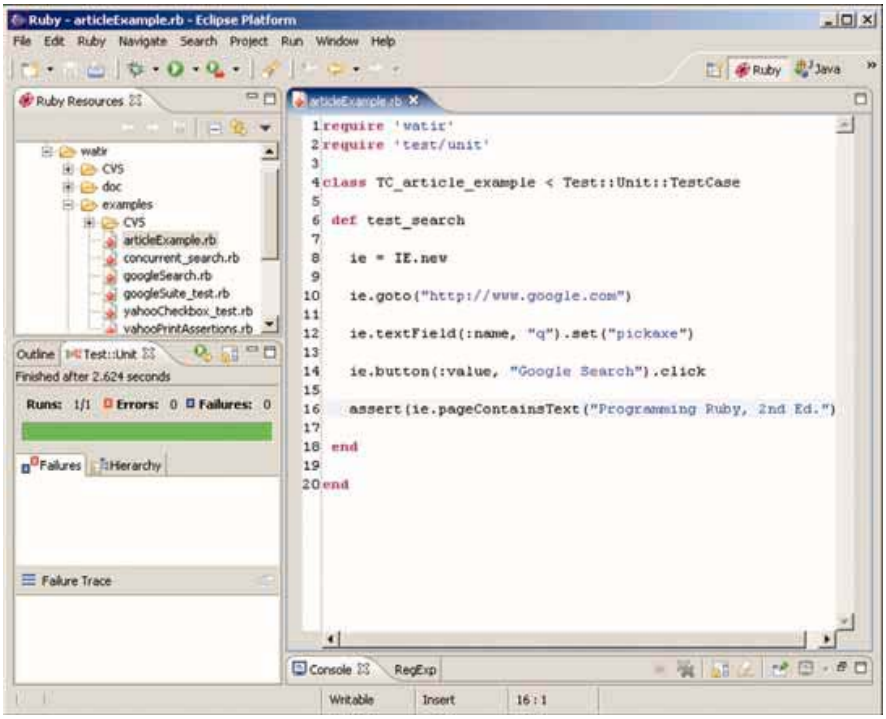

Figure 1: A complete test case in a development environment

We have successfully controlled Internet Explorer using Watir to drive a Web application. Now that we've seen it in action, how does it work?

Watir and Objects and Messages

Let's review carefully how the commands to Internet Explorer work, using this one as an example:

```
ie.button(:value, "Google Search").click
```

Read the command from left to right. **ie** is an object that accepts messages. The **button** message picks one of the buttons on a page. (If you're having trouble with the jargon like "object" and "message", see the Sticky Notes.) The arguments to **button** narrow the choices to a particular button: in this case, it's the one whose HTML tag had 'value = "Google Search" in it. That button is an object that accepts messages. In particular, it accepts the **click** message, which tells Internet Explorer to click that button.

Every object with which we can interact on a Web page is accessed this way with Watir. For full instructions on how to interact with objects in Web applications, see the Watir User Guide.

Verifying Results

So far we've learned how Watir interacts with a Web page, and we've tried a simple test case example. Just interacting with a Web application from one page to the next doesn't make much of a test case though. We need to verify that Google found the page we want. And, since this is supposed to be an automated test, we want to do that without typing into IRB. IRB is good for exploring and getting the test written, but we don't necessarily want to do it more than once.

One simple way to create test cases is to use test::unit, a testing framework that is packaged with Ruby. Test::unit allows us to use assertions to verify that the actual results match our expected results. In this case, we assert that the result page should contain the text "Programming Ruby, 2nd Ed.". If it does, the test will pass. If not, test::unit will flag the failure.

Figure 1 is what the complete test case might look like in a development environment. The assertion is the highlighted line.

Most of what we see in this test case should look exactly the same as what we did previously using IRB. To make the example into a Watir test script, in our Ruby file, we again declare the Watir library at the beginning with the **require 'watir'** statement. To also use test::unit, we employ a **require 'test/unit'** statement. Test::unit requires us to use the format in Figure 1. Each **def** is a new test case, beginning with the word *test*.

The test case uses the same statements that we used in IRB, and we use a Watir method **pageContainsText** to find out whether the Google Search page contained the text "Programming Ruby, 2nd Ed.". We wrap this Watir method in an assertion:

```
assert(ie.pageContainsText("Programming Ruby, 2nd Ed."))
```

This means that we check to see if the Google Search page contains the text. If it doesn't, `pageContainsText` returns false. In that case, the **assert** statement fails the test. To see what that looks like, go to the "watir_bonus" folder from your Watir installation. The "examples" directory will contain the file: "articleExample.rb". Open a command prompt and change to the directory to which you downloaded the test case and run the test this way :

```
> ruby articleExample.rb
```

To inspect the test script, you can open it up in your favorite text editor. Ruby comes with the SciTE editor which can be launched by right-clicking the file in Windows Explorer and selecting Edit from the popup menu. Feel free to change what the test searches for and repeat it.

Troubleshooting Using Watir and IRB

So far we've seen that IRB is useful for helping develop test cases, but there are other uses as well. A related use is as a troubleshooting tool. To demonstrate, let's look at an example. Someone from the customer support team has come to talk to you about the Web application on which you work. He says that there have been numerous customer complaints about the application becoming a blank Web page when they are using it. The customer support representative tells you that he has narrowed down the cause to people using the Back button several times in the application. He knows where it is happening, but he can't repeat it. Can you help?

You ask the support representative to pair with you, and you start to work on repeating the case. You soon realize that something as simple as clicking a button, pressing the Back button, and repeating the sequence twice more is difficult to do reliably. Why not have the computer do it? You open up IRB and, at your partner's suggestion, simulate three action and Back button clicks.

Continuing with our Google example, we can alter our test case to be repeated the way a user might do multiple searches using the Back button. This won't cause a failure with Google but works well for an example of how to repeat actions using Ruby and Watir. We can use a loop in Ruby and set the number of times to repeat a search and hit the Back button. We'll start looping the actions three times and watch what happens when the action is played back on our Web browser. If we were to use Google, this is what the test would look like in IRB:

```
irb> ie.goto("http://www.google.com")
=> 0
irb> 3.times do
irb* ie.textField(:name, "q").set("pickaxe")
irb> ie.button(:value, "Google Search").click
irb> ie.back
irb> end
=> 3
```

To close IRB, type "exit" command at the prompt.

Of course, this example doesn't cause a failure with Google. It handles the scenario perfectly. However, there is a problem with the test application. With the first attempt, doing the action and clicking Back together three times does not reproduce the error. So you increase the loop count from three to four. Still no failure. You continue up to seven, and Bang! you see the blank page. You try it at eight and see the blank page again. You retry and see the blank page failure consistently at seven or more iterations. Now that you have a repeatable case, you can make it into a test case and log the bug. You insert the entries from IRB into a test case with an assertion. The developers now have a test case that they can quickly run to repeat the problem and a failing test case to improve the application code to get the test case to pass.

Summing Up

So far we've explored a powerful Web testing tool that uses testable interfaces provided with Web browsers to allow for automated testing. We've seen that we can use the Ruby command interpreter and Watir together as an aid for test case development and as a troubleshooting tool. We've learned a little about Ruby along the way and see the potential to harness a programming language to assist in testing.

As a Watir user, you have access to an active open source community that can lend expertise. If you have questions or need a feature, all you have to do is ask, and the community will help. Because the program is free, you have full access to the source code and can see how it works or make modifications as you see fit. You can even contribute code to the project and share your expertise with the testing community. Welcome to the Watir world. Happy testing. **{end}**

Jonathan Kohl (jonathan@kohl.ca) is a software testing consultant with Kohl Concepts Inc. Based in Calgary, Alberta, Canada, Jonathan is a contributor to the Watir project. Check out other writings at kohl.ca and in the January 2004 and March 2005 issues of Better Software.

Paul Rogers, a senior quality assurance analyst also based in Calgary, Alberta, Canada, has worked in test automation with a variety of languages and tools. Paul is a lead developer of the Watir project. Contact Paul at paul.rogers@shaw.ca.

Sticky Notes

For more on the following topics, go to www.StickyMinds.com/bettersoftware

- How to install Ruby and the Watir toolkit
- More on objects and messages