This article is provided courtesy of *STQE,* the software testing and quality engineering magazine.

A graphical user interface is not designed for automation. Here's how to use a scripting interface to test the software.

BY BRIAN MARICK

Bypassing the GUI

GRAPHICAL USER INTERFACES MAKE TEST AUTOMATION HARD. THE problems are well known. You need specialized tools to drive the GUI. Those tools can be confused by the common programming practice of inventing custom user interface controls. When they are used in the simplest way, the tools lead to fragile tests that tend to break en masse when the GUI changes. Making the tests resistant to change requires elaborate and sometimes awkward testing frameworks.

Sometimes these frameworks are the only solution. Other times, there are alternatives. This article is about one of them: testing through the program's scripting interface. My example will be testing Microsoft Word through its COM (Component Object Model) interface. That's the interface that allows you to write a Visual Basic or Perl program that uses Word to generate form letters. Since automated tests are just programs, the same interface is dandy for testing.

This approach can be used with interfaces other than COM. For example, Web services

QUICK LOOK

- Testing using the Component Object Model
- The Ruby scripting language
- Combining exploration and automation

are becoming popular. These applications can be accessed from anywhere over the Internet, these days often using an interface called SOAP. Tests of such an application would look quite like what you'll see in this article.

This article will give you four things:

1. A feel for what a scripting interface looks like.

2. An example of how to write tests against one.

3. Some exposure to the scripting language I think is best for testers, Ruby. I assume no programming background.

4. An appreciation of the benefits of bypassing the GUI.

The Application

In addition to exposing its workings through its GUI, Word also exposes them through an *object model.* You may not ever test Word, but understanding its object model will help you understand the object models of other programs.

The easiest way to describe an object model is to show it in use. To do that, I'll use Ruby. Like many scripting languages, Ruby is *interpreted.* That means it lets you type instructions one at a time. As you type each one, Ruby performs it. This feature is enormously valuable. When I started this article, I knew practically nothing about controlling Word programmatically. Rather than writing a test, seeing the whole thing fail, trying to figure out what had gone wrong, fixing the test, seeing it fail again…I instead took baby steps, getting each step of the test to work before moving on.

So let's use Ruby to control Word. If you want to follow along, the StickyNotes at the end of the article tell you how to download Ruby.

Here are the first two instructions. What I type is in bold font; what Ruby types is in normal font.

```
> require 'win32ole'
true
> Word = WIN32OLE.new('Word.Application')
#<WIN32OLE:0xa0ad168>
```

The first line performs some behind-the-scenes magic that allows Ruby to talk to applications like Word. The **true** merely means that the magic worked. The second line starts Word and creates an object, **Word**, with which to control it. You can read that line as "Please, Mr. **WIN32OLE**, create a **new** instance of the **Word application** and stash the result in **Word**." The weird text in angle brackets is merely the printable representation of **Word**.

Now we can do things to Word. The first thing we can do is to make it visible. By default, Word doesn't open a window on the screen when it's started this way. To make it open a window, we do this:

```
> Word.visible = true
```

What's happening here? The object **Word** has a number of *attributes* attached to it. The notation *.attribute* asks for the value of an attribute, and the notation *.attribute=* sets the value. When we make the **visible** attribute **true**, Word shows itself.

If we look at the Word window, we see no open documents—we're not editing anything yet. We can "see" the same thing programmatically. One of **Word**'s attributes is **documents**: the list of open documents.

```
> Word.documents
#<WIN32OLE:0xa020110>
```

That by itself is not very informative. It's another weird object that prints out with angle brackets. But that weird object itself has attributes:

```
> Word.documents.count
0
```

Yes, indeed, we have no open documents. What we see here is that an object model consists of a collection of objects that are chained together. **Word** is connected to its **documents**. The **documents** attribute will be connected to individual documents (once we create them), and those individual documents will be connected to all sorts of objects representing the text within the document. Our tests will create those objects and navigate among them, *just as users do when they use the GUI.* That is, the GUI and the programmatic interface are both "skins" over the same underlying "guts." Since most of the bugs lie in the guts, we can find them either through the GUI or via Ruby.

So let's create a document:

```
> Word.documents.add
#<WIN32OLE:0xa09cc80>
> Word.documents.count
1
```

This looks as if we're asking for the **add** attribute of **Word.documents**. Instead, we're asking **Word.documents** to add a new document. (This is often called "sending the object a message," "invoking an operation," or "calling a function.") If you've programmed before, you might be used to languages that require parentheses when calling a function—**Word.documents.add()**—but Ruby is happy for you to leave them off.

How do we know what attributes and operations are available? We can look in the documentation—no, don't laugh! Remember, the scripting interface isn't intended for us testers. It's designed to let end users write form letter generators—and they'll have to have documentation to do that.

Of course, that documentation might well come too late in the project, in which case we testers will have to ask the programmers. That means we might build tests on a changing scripting interface, but it almost certainly won't change nearly as much as the GUI.

(Not knowing any Word programmers, I used three sources of information when working up this example. First, I found the text of the *Microsoft Office 97 Visual Basic Programmer's Guide* on the Web. It was easy to translate code snippets into Ruby, even though I've never written a line of Visual Basic. But that book doesn't describe all of Word's operations and attributes. When I needed to do something the book didn't cover, I did that thing through the GUI while recording a Word macro. Then I looked at the macro's Visual Basic code to find the operation name I needed. Finally, when I needed an attribute name, I used a handy feature of WIN32OLE: it lets you ask for the names of the object's operations and attributes. It's easy to guess what an attribute like **count** must mean for **documents**.)

Let's try one more thing before building the test. At any given moment, Word maintains a **selection** object that describes where new text should go. (In the GUI, this is represented by either the "I-beam" cursor or a stretch of highlighted text.) So we can use that to type text into the new document:

```
> Word.selection.typetext('hello, world')
nil
```

The text magically appears in the new document. (The **nil** is what Ruby says when it doesn't have anything special to say.)

Building a Test

Now we're ready to test. Let's suppose we're testing Word's Find and Replace feature. I'll actually write only one test, but let's set things up as if I were more ambitious.

Among programmers who are fond of testing, the "xUnit" series of testing frameworks has become quite popular. The "x" in "xUnit" stands for a particular programming language: jUnit is for tests written in Java, cppUnit for tests written in C++, and so forth. There are two Ruby xUnit frameworks. I'm using the one called Test::Unit. Figure 1 shows a simple template for Word tests.

The **require** statements set up both Test::Unit and the Word interface. The **class** definition groups some tests together. Each **def** is a test for Test::Unit to run. It's our job to fill in the blanks with Ruby code that will find Word bugs. After the end of the **class**, we see lines that start Word, add a document, run all the tests, close the document (without saving), and exit Word.

Notice that I'm setting things up so that multiple tests all use the same document. That's because programs like Word often suffer from "creeping crud," where something goes wrong in Word's internal state, the wrongness builds up for a while, then reaches the point where it causes a crash. Although it's not my goal to find such bugs, I'll be happy if I do. So I don't restart Word between tests, nor do I use different documents for each test.

However, I don't want test interdependence, such as a test that won't work unless another test runs first. That's a maintenance nightmare. So I'll want to make sure to clear the document before each test. I can do that by adding the following to the **class**:

```
def set_up
  Word.selection.wholestory
  Word.selection.delete
end
```

The first line selects the entire document (control-A in the GUI), and the second deletes the selection. Test::Unit knows to run **set_up** before each test.

Let's suppose we want to write a test that checks whether Word's Find and Replace feature works in some boundary cases: a match at the beginning of the document, at the beginning of an internal line, at the end of an internal line, and at the end of the document. So we'd like to work with a document containing text like this:

```
MATCH at beginning of file
MATCH at beginning of interior line
at end of interior line, we find a MATCH
at end of file, we find a MATCH
```

My strategy is to put that text into the document, then search for "MATCH" four times. Each time, I'll check that the search was successful and that the text matched was indeed the right "MATCH."

First, I'll write the code that puts the text into the document:

```
def test_find_boundaries
  original =
    'MATCH at beginning of file
     MATCH at beginning of interior line
     at end of interior line, we find a MATCH
     at end of file, we find a MATCH'
  Word.selection.typetext(original)
```

Unfortunately, the text is wrong. The second line doesn't really have a "MATCH" at the beginning of the line. While the first line has no spaces before the "MATCH" (because it's just after the opening apostrophe), the second, third, and fourth lines have leading blanks between the left margin and the text. That's because I was tidy and lined everything up. I could indent those lines to the left margin, but that would look ugly and make my test less readable. Since readable tests are important, I add another line:

```
def test_find_boundaries
  original =
    'MATCH at beginning of file
     MATCH at beginning of interior line
     at end of interior line, we find a MATCH
     at end of file, we find a MATCH'
  original = original.without_left_whitespace
  Word.selection.typetext(original)
```

This illustrates the importance of using a language like Ruby. Because it's a full-featured programming language, you can ex-

```
require 'test/unit'
require 'test/unit/ui/console/testrunner'

require 'win32ole'

class TestWord < Test::Unit::TestCase

  def test_XXX
    # add test
  end

  def test_YYY
    # add test
  end
end

Word = WIN32OLE.new 'Word.Application'
Word.documents.add
Test::Unit::UI::Console::TestRunner.run(TestWord.suite)
Word.documents.close(false)
Word.quit
```

Figure 1: A simple template for Word tests

This article is provided courtesy of *STQE,* the software testing and quality engineering magazine.

tend it to do things you need. (Or, if you're not a hotshot programmer, have someone else extend it for you.) In this case, I added a new operation to strings, one that removes leading white space from each line. That required one line of code. (You can find it, and other code not shown here, in the StickyNotes at the end of the article.)

Typing text puts the cursor at the end of the document. I need to go back to the beginning, where I'll start my search:

```
Word.selection.start=0
Word.selection.end=0
```

Setting the **start** position to 0 makes the selection begin before the first character. (Position 1 is between the first and second characters.) Setting the end position to 0 makes the selection start and end at the same place, which turns the selection into the I-beam cursor.

Now I tell Word what to search for and instruct it to execute the search. (This is like typing the search string into the Find and Replace popup and hitting OK.)

```
Word.selection.find.text='MATCH'
result = Word.selection.find.execute
```

Did the search succeed? (It should have.) I can discover that by checking the **result** against an expected value.

```
assert_equal(true, result)
```

My test asserts that the **result** should be **true**. If it's not, Test::Unit will tell me of a test failure.

When Word finds something, it's supposed to set the selection to the matched text. Here, I check whether the selection starts and ends in the right place (the first five characters in the file), and contains the right text:

```
assert_equal(0, Word.selection.start)
assert_equal(5, Word.selection.end)
assert_equal('MATCH', Word.selection.text)
```

I could repeat the same search three more times, first calculating the expected **start** and **end** positions for each match. But that kind of mindless repetitive behavior is what computers are for, not testers. So I'll make Ruby do it for me.

There is already a Ruby string operation, **index**, that returns the position of a match within a string. Using it, I quickly write an **indices** operation, which returns a list of positions of multiple matches. After writing it, I try it out in the interpreter.

```
> original.indices('MATCH')
[0, 27, 98, 130]
```

Now I can easily finish my test because the end of the match is always five past the start. In Figure 2, I've marked the changes in bold. The Ruby code indented under the **for** is executed four times, one for each number in the list. The first time, **start** has the value 0, the second time 27, and so on.

That's pretty much it. (Word passes the test, by the way.) I could continue testing Word's searching for a long time. If I did, I'd have to learn more details of its object model. (I've only written two programs to drive Word, both of them quite simple, so you now know almost as much as I do.) But I'm confident that there would be no great stumbling blocks in the way.

Where Do We Stand?

How does this approach let us do our job better?

1. A GUI is not designed for automation. A GUI control that can't be automated is an inconvenience to testers, but not users. So fixing the problem is low priority.

In contrast, a scripting interface is designed to let users automate. A scriptable object that doesn't script is entirely pointless. Here, the interests of users and testers are aligned, so the problem is much more likely to be fixed.

As a consequence, when we bypass the GUI, we're much less likely to find ourselves unable to finish automating.

2. The scripting interface is much more stable than the GUI. If some designer renamed a menu item, users would sigh and cope. If someone changed **selection.start** to **selection.begin**, the users would scream bloody murder as all their scripts broke. So that someone won't do it. We'll spend less time fixing broken tests, more time writing new ones.

3. We've written tests in a standard scripting language rather than what Bret Pettichord calls "vendorscript" (the custom languages built into most GUI testing tools). Standard languages tend to be more mature than vendorscript, simply because their user base is larger. And you can use them to automate some of your other tasks, so learning them improves tester efficiency in many ways. Finally, using standard languages makes it easier to get help, cooperation, and participation from programmers.

4. Because GUI test tools seldom provide free runtime licenses, it is costly for programmers to run the tests. Since scripting lan-

```
def test_find_boundaries
    original =
        'MATCH at beginning of file
         MATCH at beginning of interior line
         at end of interior line, we find a MATCH
         at end of file, we find a MATCH'
    original = original.without_left_whitespace

    Word.selection.typetext(original)
    Word.selection.start=0
    Word.selection.end=0

    Word.selection.find.text='MATCH'

    for start in original.indices('MATCH')
        result = Word.selection.find.execute
        assert_equal(true, result)
        assert_equal(start, Word.selection.start)
        assert_equal(start+5, Word.selection.end)
        assert_equal('MATCH', Word.selection.text)
    end
end
```

Figure 2: A completed test

This article is provided courtesy of *STQE,* the software testing and quality engineering magazine.

guages are free, that barrier goes away. Programmers can find their own bugs, faster.

5. Testers will be early heavy users of the scripting interface. As such, they may spot omissions, inconsistencies, and awkwardness before the users are subjected to them.

Of course, none of these advantages materialize if there isn't a scripting interface. Fortunately, they're becoming more and more popular as software publishers realize that users will get value from scripting almost any product.

What about the GUI?

I believe any well-run project will include exploratory as well as automated testing. Using a product is a different experience than programming it, and that experience activates different sites of creativity in the skilled brain.

Consequently, you should have people banging away at the GUI to find bugs, usability problems, and missing features. They'll find them in both the GUI and the guts of the program, whereas the scripted tests will find them only in the guts. Do the manual testers need to be backed up by automated GUI tests in order to find even more bugs? Or does the exploratory testing give enough coverage of the GUI?

Were I a project manager, I'd want to be sure exploratory testing gave adequate coverage. Suppose we're finding that code changes often cause GUI bugs that aren't being found by exploratory testing. My first reaction wouldn't be to add regression tests for the GUI. It would be to wonder if we weren't skimping on exploratory testing. If our exploration is missing the GUI bugs, might it not also be missing usability problems and omitted features?

My next reaction would be to ask, "Why is the GUI code so complicated that it's a source of bugs?" Consider Word's Find and Replace dialog. As far as I can tell, it doesn't do much. It collects some values, stuffs them in a **Find** object, and calls **execute** when the user presses OK. How much can go wrong? Oh, the programmer might accidentally wire up the OK button to the Cancel action, but how likely is that to escape programmer manual testing and later independent exploratory testing?

Word's Find and Replace dialog appears to be an example of what's often called a "thin GUI," one that does little processing and is cleanly separated from the guts of the program

PERSPECTIVE

A Lesson in Automation

Ten years ago, Bret Pettichord wasn't the founder of Pettichord Consulting, he was part of a software development and test team working on the Interleaf desktop publishing software. One of the team's goals was to find an automated way to test the graphical user interface. "We tried using an analog capture replay facility we had developed in-house. We abandoned it. It had all the traditional problems: synchronization, reliability, readability. It would generate these big scripts, and you couldn't tell what went wrong." Still, the team knew they wanted to find some way to automate their efforts. Meanwhile, an Application Programming Interface (API) for product customization had been developed internally, using the LISP language. "We had to test the API anyway, so we thought, 'Maybe we can use this.' " It was a good idea that never quite made it, and Pettichord thinks he knows why.

The majority of the problems came from the focus of the automation efforts. The team spent most of their time on the framework, trying to make the automated tests replicate interactive testing. When testing interactively, says Pettichord, "testers routinely make decisions on the fly. Testers focus more on one area and less on another because of the problems found in other tests. We were trying to make our automated tests replicate that mindset—true to the way people thought." The dependencies and relationships, while not all that technically difficult to support, were hard to explain and diagram. "Trying to create this artificial intelligence focused us on making these really small tests of low-level events because it was easier to show the relationships."

If Pettichord had to do it over again, he says, "First, I would take the actual bugs we'd found manually and see if I could find them through the API. It would make me feel good. Tests that find bugs are good tests. It would also tell me, 'To what degree does the API parallel the GUI?' It's quite possible that what we did in the API would pass, while failing in the GUI. We needed to find out, 'Is this going to work?'"

Second, the team had been testing the product interactively, manually, for ten years. Old habits are hard to change. "We had thirty-some testers on this project. The majority didn't have programming skills, so we were very concerned with how to get people to switch gears and write in LISP." Pettichord wishes that they had focused on getting those nonprogrammers up to speed, so more testers could have been involved: "We tried to sugarcoat it. We were trying to make it appear easy. We should have spent more time training them on the subset of LISP that they needed to understand."

Pettichord concludes, "Automated testing is different from manual testing. You can't try to replicate manual tests. You have to change the rules and the expectations. You have to get tests running. We were selling the vision—we should have been selling the results." —*R.T.*

This article is provided courtesy of *STQE,* the software testing and quality engineering magazine.

(see Figure 3). Such GUIs are inherently more testable, and many believe they make for better overall designs. For example, when your company suddenly realizes that their application could be a Web service as well as a desktop application, that enhancement is much easier if the GUI code isn't all intertwined with the rest of the code.

So I consider the need for GUI regression testing likely a sign of weakness somewhere else in the process, at least given the state of the art of GUI construction technology and GUI testing tools.

What Could Go Wrong?

All this is not to say that testing through the scripting interface is without risks. Here are some that worry me.

- What if the GUI and the scripting interface expose different features? For example, suppose Word's scripting interface provided no way to do what the Find and Replace dialog does. The exploratory manual testers may not know that the automators are not testing searching. They wouldn't then know to lavish extra attention on it. As a result, searching would be undertested.

- It's also possible that searching is available through both interfaces, but is implemented by different code. Then testing scriptable searching wouldn't tell you anything about whether searching with the Find and Replace dialog works.

- Writing tests in a scripting language isn't that hard a programming task, but it's certainly harder than not programming at all. Moreover, some testers have a phobia of programming. They've been misled into thinking all programming is hard, so, for them, all programming *is* hard.

The Stirring Conclusion

Once upon a time, when the world was young, giants walked the earth, and my hair was way down my back…we testers spent less time wrestling with inhospitable and changeable interfaces. We spent more time *testing,* less time maintaining.

The graphical user interface sent us on a long detour. Now that we realize the future is not in monolithic, mostly isolated products, but rather in ones that talk to both people and other programs, we can get back on track. What's good for a flexible, interconnected, networked world is good for testers. All that's required is that we move our hand from the mouse back to the keyboard and start typing scripts. STQE

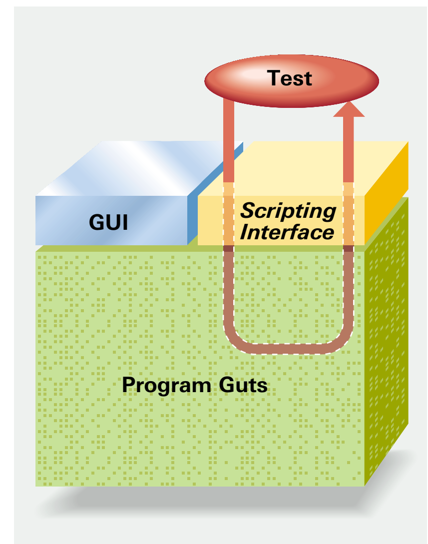

Figure 3: The thin GUI and the scripting interface are both shallow layers on top of the guts of the program.

ANNIE BISSETT

Brian Marick has worked in testing since 1981. He is the author of The Craft of Software Testing, *and is a technical editor for* STQE *magazine. Contact Brian at marick@testing.com. Read other writings at www.testing.com.*

***STQE* magazine is produced by STQE Publishing, a division of Software Quality Engineering.**