

# Guidelines for Scripting languages in Test Automation

**Mary R. Sweeney**  
**Exceed Technical Training**

It's no secret to many test teams that the use of scripting languages such as Perl, Ruby, Tcl, or VBScript can enhance their test automation projects. In fact, many teams have taken it to the next level and have built their own sophisticated automated test tools making extensive use of scripting languages. In this paper we'll explore some guidelines

## What is a scripting language?\*

“...a program or sequence of instructions that is interpreted or carried out by another program rather than by the computer processor (as a compiled program is)...

In general, script languages are easier and faster to code in than the more structured and compiled languages. A script takes longer to run than a compiled program since each instruction is being handled by another program first (requiring additional instructions) rather than directly by the basic instruction processor. “

–Tech target

([www.whatis.com](http://www.whatis.com))

\*Although this is a straightforward definition it is fairly simplistic and somewhat incomplete. For a better, and longer, one see “Open testware Reviews Scripting Language Survey” available from Tejas Software Consulting [tejasconsulting.com/open-testware](http://tejasconsulting.com/open-testware)

for using scripting languages on a test project: Why and when to use scripting languages, what possible benefits can be achieved and the common pitfalls teams can fall into that can sink an automated test project that incorporates scripting languages.

## Potential uses for scripting languages in an automated test project

Now that we know what they are – kind of (see sidebar)– why use scripting languages for testing? This paper's focus is on guidelines for integrating scripting languages into a test project, however, it's important to get an idea of why you'd ever want to do so in the first place. So I'll outline a few ways to use them and also point you to some resources where you can find out more.

One of the most important things you can do easily with a scripting language is bypass the GUI (graphical user interface) and attach to a COM<sup>1</sup> or middle-ware component. Isolating a component in this way allows you to verify it's critical functions. The following simple code uses VBScript to attach to Microsoft Excel's library.

```
On error Resume Next
Dim oApp
Dim oWorkbook
Set oApp=GetObject(, "Excel.application") `look for a running copy
If err.number <> 0 then `if excel is not running then run it
    Set oApp = CreateObject("Excel.application") `run it
```

<sup>1</sup> COM Component Object Model; refers to the object model for components developed by Microsoft Corp.

```

End if
Err.clear `clear errors
Set oWorkbook = oApp.Workbooks.open("C:\mytestspreadsheet.xls")
OApp.visible = true
Msgbox "opened Excel successfully"

```

*Listing 1. Accessing a library*

The code in Listing 1 declares a couple of variables to set up for accessing the Excel library (note the lack of data types for the variables). Then the Excel application is opened. If Excel is already running, the code attaches to it. If it's not, it launches an instance of Excel. Once we have launched Excel we can exercise specific functions within it's library. In the case of the code above we open an existing spreadsheet and then display it. Here's where you would add code specific to the test you were trying to accomplish. In this case, we could be exercising the code within the library to verify it works as designed or we could simply be verifying the existence of the spreadsheet. We can add further tests here to add valid and invalid data that tests specific functions of the library.

Why would you want to do this? Of course, you don't need to test Excel! What this code demonstrates is the simplicity with which you can open and exercise a COM library with a language such as VBScript. It's essentially the beginnings of a simple test harness for a COM library.<sup>2</sup> Again, attaching to these libraries and bypassing the GUI allows you to test critical library functions and identify them as the source of problem bugs or eliminate them.<sup>3</sup>

I've done a lot of work lately testing applications at the database layer. You can easily use a scripting language to attach to a database and verify test data that you have entered at the GUI to ensure that it has been inserted into the database properly. You can also use a scripting language to enter test data into a database for use in load testing or to verify the application's functionality in its return trip from the database to the GUI. The following code, in Listing 2, uses Perl to attach to an Oracle database.

```

<Job ID="PerlTest1">
<script language=PerlScript runat=server>
my $conn =
    $Wscript->CreateObject('ADODB.Connection');
    $conn->Open('OracleDSN');
    if($conn->{State} == 1) {
        $Wscript->Echo("Connection Successful!");}
    else {$Wscript->Echo("Connection Failed");}
my $adOpenKeySet_CursorType = 1;

```

<sup>2</sup> This is, of course, very Microsoft-centric here and I confess my bias. Most of my examples will be for Windows since that's where my personal experience lies. However, based on experiences by colleagues I know that most of the points for using scripting languages for test automation are true for other OS' in similar, if not exact, ways. I've had a lot colleagues review this document and their experiences and opinions are included throughout.

<sup>3</sup> For more information on the benefits of bypassing the GUI see an excellent article by Brian Marick "Bypassing the GUI" in which he describes doing just that using Ruby<sup>3</sup>. The writings of Bret Pettichord also contain examples of scripting languages and reasons for bypassing the GUI during a test project ([www.pettichord.com](http://www.pettichord.com)).

```
my $rst = $WScript->CreateObject('ADODB.Recordset');
my $rst2 = $WScript->CreateObject('ADODB.Recordset');
$rst->Open('SELECT * FROM TestData', $conn,
$adOpenKeyset_cursorType);
$WScript->Echo("There are ".$rst->{RecordCount}." records in the
Recordset");
```

*Listing 2. Perl script opening a SQL Server database*

The code in listing 2 is short and sweet and shows how few lines of code you can use to attach to a database. The first few lines declare the script language being used as PerlScript. Then the code uses Microsoft's ADO libraries to open a connection to a database using a pre-set Data Source Name (DSN) that contains the login information required for the Oracle database, including servername, password, and username. The code then checks the state of the connection to determine if it's successfully opened the database. If that's true then a Recordset object is created to hold the return of a SQL Select statement that retrieves a set of records from a table in the system called "TestData". Then the code reports the number of current records in this table. This code can be useful for verifying the correct input of the amounts of data after inputting test information from the application's GUI. Not too many more lines of code and you can add checks to determine whether a specific data row was input successfully and to insert test data from here.

How about some other uses? Danny Faught, Tejas Software Consulting, wrote a great paper "A Lesson in Scripting"<sup>4</sup> in which he describes the use of Perl script for creating test data and monitoring the health of an application during tests. Many of his examples demonstrate the use of Perl with both Unix systems and Windows systems as well. Most any scripting language you choose can do similar tasks.

You can also use scripting languages for other testing-related tasks like determining important system metrics prior to and after a test. Additional metrics and tasks might include:

- Determining memory state (for uncovering system memory leaks),
- Determining user and domain name,
- Uncovering important file locations and content,
- Determining and using system registry settings,
- Performing data driven testing,
- Monitoring system logging,
- Performing test logging,
- Reading or setting environment variable values
- Performing platform administration.

---

<sup>4</sup> Software Test Automation Conference, Mar/Apr 2002. Copyright Tejas Software Consulting. [www.tejasconsulting.com](http://www.tejasconsulting.com)

Not all of these are tasks you'll want to perform with a script on every test project, but at some point all of them have been necessary on one project or another on which I've worked.

The list for using scripting language solutions really is fairly endless and will depend on the test requirements of your particular project, of course. Just about every project I've ever been on, I find a different use for a quick and simple script, but often they are very specific to that particular project and wouldn't necessarily be useful elsewhere. Actually, that's one of the great things about using a quick-to-use scripting language, I find: You can meet specific needs that the major tools can't provide for a very narrowly focused test requirement. You also don't feel too badly about dumping your scripting language tests when they're obsolete because it hasn't usually taken a tremendous amount of time to produce them.

What can happen to get this all started on a project is that the test team is trying to find a way to satisfy a specific requirement and somebody with a little programming experience says "hey, you know I could write a simple script to do that" and then away you go. Of course if you don't have anyone around with that kind of skill, then you can't do that. But more about personnel requirements later. First let's discuss some of the benefits and liabilities, the good and the bad, of using scripting languages for test automation.

## **Benefits and liabilities of using scripting languages in an automated test project**

Once you have decided you can use a scripting language for some testing task, what are some of the plusses and minuses you can expect? We've gotten to the point these days where enough of us have used them sufficiently to be able to point to some real advantages and also some liabilities for using scripting languages on a test project. In the next sections I'll try to steer you away from some common problems in employing scripting languages in your test solutions and show you some of the good things about using them as well.

### **Benefits**

**Free!** Free is good, right? Well, usually, and that *is* the case here. Most of these tools are open source, if they aren't then they are very low cost. In fact every one of them in my list in the Comparison section (below) has free implementations available.

**Pre-installed.** Some of these scripting languages even come pre-installed on your operating system so you may already have one or more. For example, if you have a recent Linux OS version you already have Perl available as well as Python, Ruby and Tcl. The same is essentially true for a recent Mac OS.

If you have a recent Windows operating system (95 and above), you already have VBScript.

**Simplified structure (weak typing).** Scripting languages were created to address basic, specific, tasks and not initially intended for full application development (although you can certainly do that now with some of the more mature scripting languages) therefore they aren't, in general, as bloated with language constructs and strong typing. In other

words, scripting languages generally have a simplified taxonomy within the language itself. In general these languages are weakly-typed which means few or no datatypes in the language. Some, such as VBScript, are much smaller subsets of their compiled counterparts, which leads to the next benefit.

**Easy to learn and write (mostly).** This is actually debatable because it depends on your background and the maturity of the language you choose. If you're already a moderately experienced programmer any scripting language is really pretty easy to pick up. Their more stream-lined taxonomy means they are, in general, less bloated with complexity and that gives you a less complicated set of things to learn. Some of the languages, including Ruby, Python, and Tcl have line interpreters which give you feedback with each line of code that you type. This enhances learning the language. Of course, so does any good IDE. I don't really find that scripting languages are necessarily any easier for a beginner to pick up, any more than any other language. The more mature scripting languages are well-documented and have decent IDE's available. The less mature languages have less open-source or low-cost, user-friendly IDE's available to make coding easier to learn and less useful documentation for learning them as well.

**Allow you to satisfy requirements you couldn't easily do otherwise.** This actually is the real reason to use these languages. Some test requirements are difficult to satisfy using the larger vendor test tools. They can't build their tools to satisfy every specific need of every specific application and if you try to get them to upgrade their tool for you, you usually have a very long wait, or, more likely, you're out of luck. One company I worked with a few years ago, did have a major tool from Segue, but they hired me to come in to teach them how to write a harness for a middle-ware object that their tool couldn't reach.

**Faster from requirement to result.** The scripting languages, which are for the most part freely downloadable and, as I mentioned, frequently already installed, can be the fastest option compared to a compiled language if you don't have a compiled language already installed and ready to go. For example, I don't always want to take the time to install Visual Studio on a test platform, if it's not already there when I can write code I need in a scripting language that *is* there!

Once you successfully satisfy a test requirement with a simple script written in a scripting language, you'll be hooked. But watch out for the liabilities we discuss in the next section.

## Liabilities

**Smaller experience pool.** Even if you find staff who have enough experience with programming to use these languages effectively, using them for testing purposes is still pretty new. I've found plenty of experienced programmers scratching their heads wondering how to employ them on a test project. Using scripting languages for testing requires a context-shift that not everyone picks up on right away. Once I show them a few examples, their reaction frequently is "oh, okay, that works" and then they're good to go. However, I'm not confident there's enough experienced tester-programmers out there to make that context shift, which leads to my next liability.

**Not ready for primetime employees.** A little bit of knowledge can be dangerous. It's difficult to tell with confidence in an interview or on the job how proficient someone really is with programming much less programming in a testing environment. I've seen just a few too many situations where an employee with a single course in a language is identified as the Test Automation lead on a project using that language because they are deemed to be the "expert" by a non-technical-savvy manager. Do that and you're dooming your project to failure. Similarly, I've also seen students in some of my classes hired fresh out of a 3-day class to lead an automated test project because the test team hiring them didn't have anyone proficient enough in programming to do the interview properly. To solve this issue, you can employ members of the development team to determine if a prospective tester actually has any real programming skills. Make them write some test code and ask the developer to review it. Try to pick a developer with some test background.

**Can be a budget-waster.** Writing code can take more time than you think and writing scripting code *is* writing code. You can end up chasing around a simple bug in your test software and spend a whole day or more doing it. Anyone who's written more than a little code knows you never say "I can write that in an hour" even if you think you can. When I worked years ago for the now-defunct Software Test Labs we had a tester volunteer to write some DOS code to perform a fairly easy function. It wasn't as easy a task as he had first expected and then he got invested in getting it to work. No one could stop him until he used up the entire time budget for the project and got kicked off. That was a big lesson learned. You've got to keep tabs on your code-writing testers and ensure they don't get hopelessly bogged down on a single task. As a code-writer, it's just too easy to invest your ego in a programming assignment and vow to get it done or bust, especially for a neophyte. Unfortunately, the task and even the whole project can bust too. Which in turn leads to another corollary:

**Too easy for the whole team to get distracted.**<sup>5</sup> This is closely related to the last item, but is important enough that it deserves it's own bullet. Once you get a few scripts that work, you might decide to write a more sophisticated test driver and test suite, which isn't necessarily a bad idea. However, if you don't watch out you'll soon you might find yourself spending too much time writing test code and not enough testing. Test tool writing can be a business in itself. Some larger companies I've worked with can afford to employ a team to write test code and sophisticated drivers and harnesses but even then it's important to closely manage these projects. Small to mid-sized companies need to be even more careful. There are a lot of good test tools out there and sometimes writing your own is a bad idea, especially late in the project. An excellent article which addresses many of these issues is "Build it or Buy it" by Elizabeth Hendricksen<sup>6</sup>. Both Bret Pettichord and James Bach have pointed out this problem in their writing; I've seen it happen myself, and have to confess that I've been tempted to fall into this hole too.

If using scripting languages for test automation gets a bad name with a group or company, in my experience, it's often because they used it for too much too soon and for too little benefit.

---

<sup>5</sup> Thanks for Bret Pettichord for this item.

<sup>6</sup> published in STQE Magazine, May/June 2000. Download at [www.qualitytree.com](http://www.qualitytree.com)

**Scripting languages are moving targets.** As my colleague, Danny Faught (Tejas Software Consulting) points out, there are constant updates to these languages. Which version should you use? Let me just fix this one for you right now: Download the current scripting language of your choice and then don't change it for upgrades unless you have a real business reason to do so. I'm always amazed at how often the scripting languages seem to come out with a new release. If you constantly upgrade you may have to deal with incompatibility with previous releases. In my experience, new releases tend to be pretty buggy for a while anyway. Get the "last stable release" and stick with it for a good long time.

## **How to interleave scripting languages successfully into an existing test project**

**Start small.** If someone on your test team steps up and says "hey, I could write a script to do that" when trying to satisfy an elusive requirement, try to give them a chance and see how it goes, all the while checking in regularly to determine how much time it's taking and how proficient the person is on this task. If you don't have time to monitor this, then don't take it on but I would encourage you to try it some time, and if it doesn't work out try it again some other time with a different task. You've got to start somewhere and starting small is best.

**Set proper expectations.** Adding any kind of new technology or process to an *existing* project will always add more time to the project. If you can add in a few scripting languages tests right from the beginning it may not add *more* time onto the project but I wouldn't ever regard it as a way to *save* time on a test project, even though, sure, that *could* happen. *Adding scripting tests to your test plan should be considered a way to expand and deepen your test project with new abilities to harness individual library functions and database tests and perhaps add more platform monitoring routines.* Remember that automated test scripting will be just one additional tool to your overall test plan. It may end up only being effective as a very small subset of what you do overall.

Years ago, I taught Microsoft Visual Test (now Rational Visual Test) at a variety of companies. Visual Test is kind of a super-scripting language specifically for testing that is long past it's prime. It had much more programming involved than the higher-end test tools available from the major test tools vendors. I often found that the expectation of the managers of the companies purchasing Visual Test as a new tool was that this new tool would allow them to eliminate testers and speed up the test process. They were usually surprised and disappointed too when I told them it would probably have just the opposite effect. That they would require *more* time to interleave the tool into their testing process and that at least at first they would need more testers with technical programming expertise to use it effectively. (By the way, these more technical personnel usually want a higher rate of pay too.) So they weren't really going to save a lot of money or time at least at first. These benefits may come eventually but the main benefit, again, is the ability to broaden and deepen your test project with tests you wouldn't otherwise be able to perform without the tool..

These same issues I found for integrating Visual Test into a test project are largely true for the use of scripting languages, as well. So don't count on any instant time or budget

savings and please don't set your expectations so high that you think you'll immediately be able to eliminate use of a major vendor tool you're currently using.

I encourage you to think of the use of scripting languages as a great adjunct to add to your existing tools. I don't advocate writing *all* your automated tests yourself using an open source scripting language! (Although in some companies this *might* work if you combine it with the use of some of the new open source test software.) At least initially, this would be an unrealistic expectation and will doom your early attempts to failure. Set low expectations to build expertise within your team before adding more major projects using a scripting language.

**Add a few programming proficient testers to your team.** They can add a lot to a test project. If you don't have them, you'll never get any of the benefits of using scripting languages. But, don't make the mistake of hiring and using only those testers who are programming proficient either. This appears to be a new trend with which I don't agree. It's great to have a few testers who are programmers on a test project but not *all* of them. Why not? People who are experienced in writing code tend to look at everything from a code writer's perspective. This can be very useful in uncovering the kinds of bugs a developer might be prone to. However, since these programming-proficient testers also tend to know more about how applications are supposed to work the danger is that they will ignore some of the more common errors a neophyte user can be prone to making, such as entering invalid data and clicking the wrong things. I think you need both kinds of tester and it's about time we had the more technical white box tester that only a programming-proficient person can be but we shouldn't let the pendulum swing too far in that direction. If we do we'll lose the benefits of the best test technicians out there, just because they can't write code.

When you add some programming expertise to your team be sure to add at least one of that is VERY proficient. We call that a "guru". This has to be the go-to person for others and is someone who is experienced enough to write scripts easily. My gut feeling is that there really should be two gurus if you have a team of any decent size, like 8 or more. Otherwise your guru spends all his or her time helping everyone else and doesn't get a thing done on his own. You can hire a consultant for this at first to get your team started writing scripts, to evaluate the background of your personnel, and to train them, but ultimately you need at least one of your own.

**Investigate training options.** Although there are not as many courses for scripting languages as there are for the regular programming languages, there are some excellent books that can help one pick up the basics and many on-line resources. There are also more and more opportunities for learning scripting languages specifically related to test projects (check out the acknowledgements and resources at the end of this paper). My caution here is not to expect that one short course or one good book will be enough for the beginner to become a scripting language expert. I touched on this earlier but it's worth mentioning again: This is an opportunity for failure that happens more often than it should. You need at least one guru on your staff who is very proficient in programming and then additional training for that person and basic courses for some of the others on the team can really pay off. Otherwise, it's just money down the tubes. One course does not make anyone an expert.

**Allow time to maintain your scripts or be willing to throw them out.** Test code writing is not precisely the same as software development in at least one major way: you don't necessarily need to buy into full code-reuse. You'll find yourself throwing code away as requirements change and your test code is not necessary or useful. You may also find yourself doing shameless copy and pasting between test scripts more often than any good developer would. This is not all bad. You simply won't have time to manage the maintenance for all your simple tests so be willing to throw them away more often. At the same time, recognize that from build-to-build of your software, even your best little test scripts will need a bit of testing, debugging and tweaking. Not allowing sufficient time for this can significantly reduce the benefits of using scripting languages. Code reuse is a good thing in test code writing as long as it doesn't take up too much of your project time and budget. Test code is not the same as production software so don't treat it that way.

Warning: If you buy into building your own sophisticated drivers and test suites, in any language, you will need to allocate as much time to maintenance and code reuse and, yes, testing, as any other software development project!

### **How to select a scripting language**

I wish I could tell you: "Stick with *XYZScript* and nothing else, it's the way to go" but life is never that simple so why should this be? The choice will usually be made for you by the one person you have who is a guru already or one you hire. If that person chooses VBScript instead of Perl is it a mistake? Does it matter? Here are some of the decision factors in choosing a language but ultimately, most any language is a workable choice:

**Do you need cross platform capability?** Most of the languages are able to be used in multiple environments, double-check that before you commit to the first effort though. In a later section I'll tell you where you can get these languages. The platforms that they can be used on are listed right up front on the websites from which they can be downloaded.

**Are there readily available resources such as books, user's groups, training, and free documentation?** The more mature languages such as Perl, Tcl, and VBScript have a lot of this. The more recent languages like Ruby do not have as much but probably will eventually. This leads to the next decision factor.

**How mature is the language?** Why do you care about this? Well, not only will a more mature language have more resources, but it will also have less bugs in the most stable release. And a more mature language will have a larger pool of experienced people to choose from. It will also have more capability to be able to perform a broader range of tasks. Perl, VBScript and Python fall into this category as well as many others.

**How easy-to-learn is the language?** There are some differences in this and it can make a difference as to how proficient your non-guru types can become at doing basic scripting language tasks, like at least being able to write code to check the machine name and operating system version. In general a language that's based on Basic that uses more English-like statements is easier to learn than a language like Perl with it's swirly-braces and powerful, but complex, modules.

## Comparisons

I can't possibly do a comparison of all languages for one very good reason: I'm not at expert level at all of them so it wouldn't be fair! However I do know a few, and I can give you some basic pros and cons derived from those experiences and those of my colleagues that will help get your own process started. I'm sure some will disagree but that can be a good thing, let the debate begin:

### Perl

#### Pros

- Cross platform utilization: Unix, Linux, Windows systems, OS/2, NetWare, z/OS, etc.
- Mature language
- Very powerful; lots of documentation
- Lots of experienced people available

#### Cons

- has a cryptic, confusing syntax; lots of {} (swirly braces)<sup>7</sup>

### Ruby

#### Pros

- Fewer libraries to learn to use; less documentation to wade through
- Easier to learn and use
- Windows, Unix/Linux
- Lots of interest in the Agile community

#### Cons

- Fewer libraries to learn to use; less documentation to wade through<sup>8</sup>
- Relatively immature language, which means it has a few more bugs

### Python

#### Pros

- Easier to learn and use; comes with a user friendly IDE.
- Mature, popular language
- Windows, many Unix/Linux platforms, OS390

---

<sup>7</sup> Yes, I'm aware some of you won't acknowledge this is true, especially if you've coded primarily with a swirly brace language! I don't consider it particularly tough to pick up but I've noticed that some find the use of braces distracting and have a hard time picking it up.

<sup>8</sup> Yes this is both a plus and a minus!

Cons

- Haven't heard any! (But that doesn't mean there isn't any)

## JavaScript

Pros

- Mature language; most of the same benefits as Perl: documentation, resources
- Cross-platform utilization
- Many experienced people out there

Cons

- Also uses some relatively cryptic language syntax
- Almost always limited to running with a browser<sup>9</sup>

## VBScript <sup>10</sup>

Pros

- Best choice for Windows systems (my opinion)
- Mature language; easiest syntax
- Same syntax as many macro languages and vendor tools
- Lots of experienced people available

Cons

- Can't really leverage cross-platform capability
- Windows only

## Tcl

Pros

- Smallest language and is often chosen to be ported to embedded operating systems
- Mature, popular language
- Windows, many Unix/Linux platforms

Cons

---

<sup>9</sup> I'm not so sure there isn't a version on JavaScript that runs outside of a browser but I couldn't find a reference to one that does. I'm sure I'll find out once this paper is published!

<sup>10</sup> I think it's the best choice for Windows because it's already pre-installed and works easily with the Windows Script Host and other scripting tools to do pretty much anything you want on a Windows platform. As a Microsoft product it has similarities to the applications developed in Windows so you can leverage your knowledge in that way as well. Plus it's the same essential language syntax used by many other Microsoft products such as VBA (Visual Basic for Applications).

- Haven't heard any! (But that doesn't mean there isn't any)

## Rexx

### Pros

- Mature, popular language developed originally for IBM systems
- Object Rexx: Windows (not free), some Linux platforms, OS/2, Solaris/Sparc, AIX
- Net Rexx: runs on a JVM (Java Virtual Machine) 1.2+ for any platform presumably

### Cons

- Haven't heard any! (But that doesn't mean there isn't any)

Scripting language implementations and versions are a moving target, so this list is probably not very complete and will soon be out of date also. This should, however, give you a place to start.

## Scripting languages vs. Compiled languages?

Are scripting languages better for testing than compiled languages? I'm not arguing that. In fact, I use and teach compiled languages for testing including Visual Basic 6 and the .Net platform languages. I find scripting languages useful for some things and compiled languages useful for others. Usually I choose a scripting language for something relatively quick and straightforward where I don't want to go to the bother of writing and compiling an executable and then porting it over. I can jump into a scripting language to retrieve system information quickly, generate quick test data or perhaps attach to a database.

I'll choose the compiled language when I'm thinking of a longer term test utility or suite or driver. Additionally, you may choose to use the same compiled language, such as C++, C#, C, Visual Basic .Net, Java, etc. that the developer's have used to write the application. This can be an advantage for white box testing because it aids in understanding possible code errors and, of course, you already have it available. Most compiled languages have a great tool set and IDE and a wealth of information available. Remember though that it will either need to be installed on all machines which can be a lengthy process, or you will have to compile your test software on one machine and port it to the test platform.

Scripting languages, though, as my friends point out to me, can also work well for longer term test utilities or suites or drivers. I find them generally faster to use to get from test requirement to test result. Ultimately, though, using a scripting language is just another option you have that you can employ as you will, but first you have to get the skill set within your test team.

## Scripting languages vs. Vendor macro languages?

What about vendor macro languages? By this I mean the languages often included with tools provided by Rational, Segue, and Mercury. These bear many similarities to the scripting languages as they often are subsets of other languages such as Visual Basic and C with some extensions specific to the tool they're used within. They can have some the same kinds of benefits as scripting languages, such as being quick to use once you've learned them, but in general I've found them to be less powerful and useful<sup>11</sup>. and you can't leverage your knowledge of these languages for other uses since they're usually pretty tool specific.

## Where can you get a scripting language?

All of them can be downloaded off of the web. Here are the websites for some of the ones we've discussed. Usually you can find links to user's groups and other resources on these websites as well.

- Perl: **[www.perl.com](http://www.perl.com)**; Pre-installed on Linux
- Javascript: **[javascript.internet.com](http://javascript.internet.com)**
- Ruby: **[www.ruby-lang.org/en/](http://www.ruby-lang.org/en/)**
- VBScript: **[msdn.microsoft.com](http://msdn.microsoft.com)**; preinstalled on Microsoft platforms. Just create a file with a .vbs extension
- Python: **[www.python.org](http://www.python.org)**
- Tcl: **[www.tcl.tk](http://www.tcl.tk)**
- Rexx: **[www.rexxla.org/Links/links.html](http://www.rexxla.org/Links/links.html)** (Start here to choose the correct platform)

Free online documentation is available for all of the open source languages but the quality varies widely so be sure to do some investigation and planning before making a choice.

## Summary

Scripting languages should be just another tool in the tool belt available to a test team. If you don't have the expertise on your test team, get it soon, it can add more and deeper tests in many areas. Don't expect that it will make your testing times shorter, but you can expect, in the long term, to have the opportunity to create more, complete and perhaps even more flexible tests that bypass the GUI. There are many ways to get started using scripting languages but, please do start small and set realistic expectations.

---

<sup>11</sup> For a good discussion of this see "Hey Vendors Give us Real Scripting Languages" by Bret Pettichord (<http://www.stickyminds.com/sitewide.asp?ObjectId=2326&ObjectType=COL&Function=edetail>)

If you're a test engineer, getting proficient in a language can be a good career move right now as more and more companies are seeing the benefits of having this expertise on their test team.

## **Acknowledgements**

The following people contributed to this article by allowing me to canvass their opinions on this topic. Both gentlemen offer consultation and training on testing using a variety of scripting languages. Special thanks to:

Danny Faught, Tejas Software Consulting ([www.tejasconsulting.com](http://www.tejasconsulting.com))

Bret Pettichord, Pettichord Consulting ([www.pettichord.com](http://www.pettichord.com))

Also, I frequently rely on and am greatly influenced by the writings of my esteemed colleagues in the testing world, Elisabeth Hendricksen ([www.qualitytree.com](http://www.qualitytree.com)) and James Bach ([www.satisfice.com](http://www.satisfice.com)).

And finally I would like to thank my friend Alan Corwin ([www.processbuilder.com](http://www.processbuilder.com)) who always agrees to read through and comment on my articles.

## **Resources**

Training and consultation can be found at the links above and also please watch Exceed Training ([www.exceedtraining.com](http://www.exceedtraining.com)) for upcoming courses. The links for downloading the scripting languages in the preceding section "Where can you get one?" also contain many free on-line resources for documentation and recommend good books for each language.

## **Biography**

Mary Sweeney has been teaching testers how and why to write code since working with Software Test Labs beginning in 1995. She's the author of Visual Basic for Testers (Apress, 2001) and several published articles on test automation topics. Mary is a college professor and also performs independent consultation and training through Exceed Training ([www.ExceedTraining.com](http://www.ExceedTraining.com)). She has degrees in mathematics and computer science from Seattle University and lives and works in the Pacific Northwest. You can contact Mary at [marys@exceedtraining.com](mailto:marys@exceedtraining.com).