How to improve your testing process by using programming skills

A Lesson in Scripting

BY DANNY R. FAUGHT

Are you a programmer?

The testing profession attracts people from a wide variety of backgrounds, and many testers are not familiar with programming. I want to motivate you to learn a programming language, specifically, a scripting language. I'm going to throw some code at you, and while I don't expect you to understand everything about it, you may be able to start seeing how problems can be solved with a script.

Should you be a programmer?

Sometimes testers have a desire to learn the language that their project developers are using, so they can understand how the product is implemented. But I also think testers should learn programming skills in order to make the testing process more effective. It's hard to get a firm understanding of a programming language unless you have a need to do something useful with it. Writing programs to help with your daily testing tasks can provide this motivation, while learning a language just to read code that someone else wrote isn't nearly as motivating.

QUICK LOOK

- Examples of simple scripts
- Choosing a language

For example, on one project I'm working on right now, I knew I needed to learn how to program on the system under test, but just didn't get around to it. Once I decided to start automating some tests, though, I finally had my motivation. I was able to pick up enough of a new language to complete the first test (and find the first bug!) within an afternoon.

More to it than test execution

When people talk about test automation, they're usually talking about automating test execution. But I want you to think more broadly when you're considering how you can automate the many different things you do as a tester. Sometimes it isn't appropriate to automate your test execution. When this happens, you shouldn't think that nothing can be automated. There are still plenty of opportunities to automate. You can automate the setup of your test machines, including installing the product and configuring it, and you can also automate your test design.

One of my favorite automation examples required only a single line of code. While I was doing exploratory testing on a Web application, I came to a file attachment dialog, where the browser would send a file from the local disk over to the Web server. After trying some simple test cases, I threw some extremes at the application.

I always like to start by sending a million zeros. This combines a data value that the application may not be expecting—a very long line (there are no line feeds in the data) and a large volume of data. But where do you get a file like that? You create it on the spot.

If you know several different programming languages, you know that most or all of them are up to this task. For simple types of test data, you can even use a text editor—type out a few lines, copy and paste, then copy the whole thing and paste again. By repeating this process, you can grow the file exponentially and get as many megabytes of data as you need. (You'll also find out how robust your text editor is.) But I'm lazy, so I prefer an easier approach.

For this particular task, I used a scripting language. Why? Because I can do it as a one-liner. My favorite scripting language happens to be Perl, so here's the Perl code for my one-liner:

```
print "\0" x 1000000
```

Here I used Perl's Print command to print out the data. The "x 1000000" tells Perl to make a million duplicates. The data that we're duplicating is "\0." The backslash ensures that we get a byte value of zero. (Note that this does not print out as the text character "0," which is actually encoded with a byte value of 48.) Why do I take such pains to get real zeros? Because programs don't expect to get that type of data!

But where does the data go? I'm going to enlist the help of a command-line shell. (What's a command line? See the sidebar on page 20.) Now that we have some idea of how the Perl code works, I'll show you the full command line that I use to execute the code:

```
perl –e 'print "\0" x 1000000' > datafile
```

If I type this code into a shell and press Enter, a file named "datafile" will appear on my disk, containing exactly one million zero bytes. Because it's a one-liner, I can just type it in wherever I need it. I don't need to save the code in a file, set up the network to access a code repository, or dig up some sort of removable media to copy the code over.

One thing I need to do before I can use my one-liner is to make sure that the Perl interpreter is installed. On a few systems, you have a good chance of finding Perl already installed. For example, a Linux system is very likely to have Perl installed, but Microsoft Windows systems usually don't have it by default. (You'll find a link to Perl's Web site in the StickyNotes for this article.)

A little help from the shell

The work in the one-liner is split between Perl and the command line interface (CLI). The shell first interprets the command line as a whole, before the Perl interpreter gets to see it. In this case, I used a Unix-style shell. Starting the command line with "perl" tells the shell to look for a program named "perl" in all the places it knows to look for programs and run the program (you'll need to set your PATH environment variable to include the directory where the Perl interpreter is installed).

The first command line argument sent to Perl is "–e", and the code in single quotes is the second and last argument. The –e option tells Perl to execute the code following the –e rather than looking for the code in a file. Finally, "> datafile" is interpreted by the shell. This tells the shell to open a file named "datafile" and direct the output from Perl to the file. We could have done the file handling within the Perl code, but it's easier using the shell's syntax. When we're typing in one-liners, saving keystrokes is an important feature.

If you're running Perl from an MS-DOS shell on Windows, all of the same mechanisms apply for starting Perl and creating the data file. One difference you need to be aware of is something that will cause you fits any time your code passes through more than one interpreter on any system. In the earlier example, the single quotes are interpreted by the shell, and the inner quotes are passed through to Perl. This works because I used two different types of quotes. MS-DOS doesn't like single quotes, so we need to use a somewhat more awkward method:

```
perl –e "print \"\0\" x 1000000" > datafile
```

We can't change to single quotes inside the Perl code, because the magic \0 incantation only works within double quotes. So we embed double quotes within double quotes by escaping the inner quotes with backslashes. It's ugly, but it works.

Now, let's suppose that we try our data file in a test, and much to our disappointment, it doesn't flush out a bug. No problem, we'll try ten million bytes! Just run the one-liner again with one more zero. Maybe we did find a bug, and we want to reproduce it with a smaller data file. Again, a simple change to the Perl code can yield any size file we want, and also with any contents we specify. We should also try a more normal character like "a" since the program we're testing may reject zeros as invalid data.

Shell scripts

We've been using the shell to help our code, so why not do some programming using only the shell? Indeed, you can put shell commands in a file and execute them as a script. Even MS-DOS has a simple batch file mechanism (ever see a file ending in ".bat"?). Unix-style shells include features to facilitate scripting, including looping mechanisms, Boolean and math logic, and even user-defined functions. Unix shells aren't just for Unix users, either—they are available for a wide range of operating systems (usually for free).

Here's a simple script that I recently used to monitor a system while I ran a stress test. You could consider it a heart monitor—giving you a heartbeat so you can watch the basic health of the system.

```
while true
do
    date
    sleep 30
done
```

To run this script on a system that has the bash shell installed, save it in a file named "monitor." Then at a shell prompt, type "bash monitor." You'll see an endless stream of output like this:

```
Sun Sep 30 15:45:47   2001
Sun Sep 30 15:46:17   2001
Sun Sep 30 15:46:48   2001
Sun Sep 30 15:47:19   2001
```

Note that the times are sometimes more than thirty seconds apart. This is because of the time it takes to execute the commands in the loop between the calls to the sleep command. Hold down the Control key and press "c" to stop the script. Unless you do some tricks to run this script in the background, you'll want to run the monitor in one window and the test in another.

Why in the world would you want to check the time every thirty seconds? I don't have an elaborate test harness on the embedded system I'm testing on. In fact, no commercial test tools are available for it. I am running a simple stress test that spawns many simultaneous tasks. I may leave the test running over a weekend. If the system crashes while I'm gone, I want to know when it crashed, and maybe learn some additional details about the health of the system right before the crash. My simple script will give me that.

If the system crashed, I'll know when it happened because the stream of time stamps will have come to an abrupt halt. Sometimes, a crash is preceded by a progressive degradation in performance. I can tell if that happened by looking at the gap between each time stamp. I have seen some pathological cases where most of a system had crashed, and the only thing still running was the monitor script. Therefore, it's good to have at least two different types of monitors running so you can validate the data.

You will probably want to log the output of your monitor to a file. If you cause a system crash, your shell window may go down with the system. Or you may find nothing but a stream of "out of memory" errors filling your screen, and have no idea when they started. So try "bash monitor | tee monitor.out" to run the script. This sends the output from the monitor to the

What's a command line?

I was trying to show a tester how to use Perl once, and I asked her to type something on the command line. Her response really surprised me—"What's a command line?" It hadn't occurred to me that a tester could have made it through several projects without ever escaping the confines of a graphical user interface (GUI). In fact, it's possible to work with a scripting language entirely within the GUI, but I've never bothered to work out how to set it up.

For those who are stranded in the land of the GUI, ask an old-timer what computers were like before the graphical interface came along. The user used a terminal with a text-only display or teletype and a keyboard, and initiated all actions by typing the name of a command along with options to specify what the command should do. This is a command line interface (CLI).

Personally, I prefer to use a hybrid approach. I use a GUI for tasks that it's well suited for, and use one or more CLIs running within the GUI where appropriate. My Perl one-liners are a good example of this. Using the command line instead of the GUI saves me from having to save the code to a file before I run it.

The method of getting to a CLI differs from one system to another. Unix-like operating systems generally offer a wide variety of "shells," many of which have some very useful features such as command line recall and editing, and job control for managing multiple concurrent tasks. A shell window is only a click or two away on most Unix graphical desktops.

For Windows, the simplest CLI is the "Run" dialog on the Start menu, which lets you run one command at a time. This doesn't always work well with non-GUI programs, though. There's the good old MS-DOS command prompt that will even recall and edit previous command lines in some versions. In addition, there are a number of alternative CLIs available on Windows, including ports of several Unix shells. Installing the free Cygwin environment is a good way to get access to several full-featured shells on Windows.

"tee" program, and tee sends it to both the terminal and the file. You get the benefit of being able to check the system's heartbeat at a glance, and you'll also be keeping a record of that heartbeat in a safe place.

I have it easy, stress testing an operating system. If you're testing an application running on top of the operating system, you'll want to monitor the status of the application by performing some simple operation on a regular basis. If you don't have an easy way to do this via a programmable interface such as a command line, programming library (API), etc., then you would be justified in asking the developers to provide such an interface in order to improve the testability of the application.

Drop me a line

Here's another shell script example written for the bash shell. Let's say we already have a script that generates a report from your bug tracking system, or we're generating reports from a test execution tool. We want to email the report to someone automatically. And let's add one additional twist—we need to accomplish this task on several different operating systems, including several versions of Windows running a shell under the Cygwin environment (see www.cygwin.com) and also several different Unix variants. So we'll use a portable mail utility that we can call from another script.

If we name the script "shmail," we can call it like this, specifying the subject of the mail message, whom to send it to, and a plain text file to send: "bash shmail "bug trend report" manager@foo.com reportfile". The script "shmail" looks like this:

```
os=$(uname –s)
subject=$1
to=$2
file=$3
if [ ${os:0:6} = CYGWIN ]
then
    blat $file –s "$subject" –t "$to"
else
    mailx –s "$subject" "$to" < $file
fi
```

The key to the script's portability is the call to the external "uname" utility to determine what operating system we're running on. The next three lines assign meaningful names to the three parameters that we pass in. In the "if" statement, we compare the first six characters of the operating system name to the string "CYGWIN." This string varies depending on which version of Windows we're running the Cygwin environment on, but we'll catch all of them by just looking at the first six characters. If we are running on Windows, we use the open source "blat" program to send the mail. Otherwise, we assume we're on a Unix variant that has "mailx" installed. Of course, additional variations can be added. For both mail programs, we expect that the system knows how to locate them on the hard drive, since we didn't specify the full filename path for the programs.

I think it's very powerful to automate a process and have the results magically show up in my email box. But now that I've introduced this tool, I want to point out that it's very easy to abuse it. I've seen people's mailboxes fill up with automated email that they never look at, because they get so much of it. So please put some careful thought into how much email your scripts are sending, and consider that the needs for automated reporting may change over time—don't just leave your scripts on autopilot without periodically reevaluating the process.

What else can you do with a script?

There are many tasks that scripts can help with. I've used scripts to summarize test results and produce a report. I've used scripts to automate nightly builds. I've used a complex parallelizing test harness that was implemented with scripts. And I've written a reusable stress test driver. Oh, and one other thing—I've implemented many automated test cases, using a wide variety of scripting languages.

For instance, Expect is a special-purpose, public-domain scripting language that is very useful if you're testing a command line interface. I have implemented a reliability test based on user scenarios that I automated using Expect. The computer never knew that there weren't actually hundreds of users all interacting with the system—I even had the script insert random typing delays.

Most commercial test execution tools have a proprietary built-in language, or "vendorscript," to use a word coined by Bret Pettichord. These languages are usually vaguely similar to a popular language like C or Visual Basic, though they generally only include a subset of the features that the real languages support. Depending on how the tool works and how you're using it, you may need to stick with this specialized language for your scripting. Although you will need to learn the nuances of the language, that knowledge won't be applicable to any task other than working with that one tool.

A few tools use a popular language as their native tongue, so while you can't choose the language, at least your knowledge about the language (and perhaps even

some of your code) will be transferable. Some test tools allow you to call an external program. This allows you to use any language that you choose, to the extent that you can do what you need to do outside the direct environment of the test tool.

To compile or not?

Why am I discussing scripting languages and not compiled languages like C++, Ada, or Java? I think scripting languages are generally a better tool for a tester. First, a bit of definition. Code written in a *compiled* language must be processed by a compiler before you can run a program written in that language. A *scripting* language is generally *interpreted,* which means that you can execute the code directly. There are many situations where these definitions get blurred, but let's try to keep it simple.

The most obvious benefit of using a scripting language is that you don't have to bother with compiling your code before you run it. This makes it easy to experiment, since you have an "edit-run" cycle rather than an "edit-compile-run" cycle every time you modify the code. Testing often involves experimentation. Think of the Perl one-liner example— with a single command line, we can write the code and run it. If you have a decent shell, you can run the code, recall the previous command line, edit the code to change your test data, and then press Enter to execute the new code.

When learning a scripting language, you can do useful things with the language more easily than with a compiled language. Compiled languages generally require some setup code even for simple programs such as loading header files, building data structures, declaring variables, etc. Simple programs written in a scripting language almost always require less code and less knowledge about the programming environment than programs for a compiled language. So you'll be running programs saying "Hello, world" with less frustration in Perl, for example, than in C or Java.

I also like Perl better than C because Perl code tends to be very compact. I can do a tremendous amount of work in only a few lines of Perl code. If I translate that code to C, I could easily end up with three times as much code or more. So I can write programs more quickly since I don't have to write as much code. This effect may not be quite as pronounced if we compare it to a more modern compiled language, but I doubt that I'll find a compiled language that rivals Perl in compactness.

Fans of compiled languages point out that the loose rules of a scripting language will make scripts prone to errors. The extra code that you write in a compiled language to declare your variables and load function prototypes helps the compiler point out errors that you might otherwise have missed. You should be aware that bugs are more likely to lurk in your scripts if you take advantage of all of the programming shortcuts that are available. You should use some prudence when deciding to use these shortcuts. Because larger and more complex programs are harder to create and thus more buggy than small programs, you should use fewer programming shortcuts on larger programs. For Perl scripts longer than a few lines, I always turn on Perl's options to be pickier when parsing the code, and I take more care to write bulletproof code.

The size of the code is another relevant factor here. Testers generally write small test cases, well under 1,000 lines of code each. Test drivers and other support code are likely to be less than 10,000 lines of code. Scripting languages are well suited for small programs, and for an experienced programmer, they can also be a good choice for medium-sized programs. For large programs, you're better off with a compiled language.

Choosing a language

So which scripting language should you use? There are dozens of options, with more appearing every year. Many of them are distributed with an open source license, so they won't cost you anything. To make an informed choice, you have to do some research.

First, find out what operating systems your scripts will need to run on. Will your scripts need to be portable to more than one operating system? Narrow your list of options by eliminating any languages that aren't supported on all the operating systems you're using.

Also find out what languages the people in your organization already know. They should be involved in the choice of language, because they may someday need to read or modify your code. Also it can be a huge advantage if you have a local guru who can help you learn the language.

Some of the more common scripting languages that I run across lately are: the various shells, Perl, Tcl, Python, and recently, Ruby. Your search shouldn't necessarily be limited to these choices, but you can take this list as a suggestion for where to start.

A beginning programmer may have trouble writing a moderately complex shell script because of the typically lean documentation, and the fact that the programmer would need to learn how to use several external and sometimes esoteric utility programs. Shell scripts are best for very simple kinds of processing. Though our Perl example was only a one-liner, languages like Perl that were designed specifically to be scripting languages are better than the shell for complex programs.

When choosing among scripting languages, you'll notice that older languages such as Perl and Tcl are a bit clumsier with respect to newer programming paradigms such as object-oriented programming. But the newer languages will have a much smaller and less robust body of prewritten code available to handle common programming tasks, and they'll have a smaller user base that can help you with any questions you have. The choice of a programming language can reach a religious fervor in some groups. So in the end, it's up to you to decide which one you have more faith in.

Currently, I'm using an embedded operating system that even Perl hasn't been ported to. I'm using a compiled language for the stress tests that I'm writing, and the system's limited shell language for the test harness. But I'm also looking for opportunities to use a more widely supported operating system for some other types of test automation, such as processing reports from the bug tracking system.

To be most effective, you'll eventually end up learning more than one language. But the most important thing is that you take that first step of learning your first scripting language. You'll find more uses for it than you can imagine. STQE

Danny R. Faught has been a script jockey ever since he started doing software testing. He is now an independent software quality consultant and proprietor of Tejas Software Consulting (www.tejasconsulting.com). *Contact him at* faught@tejasconsulting.com.

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