Software Testing for Sysadmin Programs

ADAM MOSKOWITZ

Adam Moskowitz is a Senior Tools Engineer at MathWorks and was the program chair of the LISA 2009 conference. Since entering the field in 1978, he has been a computer operator (what sysadmins were called in The Good Old Days), an application developer, a system administrator, and a teacher of all of those things. At MathWorks he is building a system to manage VMs in both production and ephemeral testing environments (and quietly take over the world). When he's not writing code, you will find him judging barbecue contests or riding a tandem bicycle with his wife. adamm@menlo.com

Testing is a common practice in modern software development, but the mere mention of it tends to raise hackles in the sysadmin community. To me, this is disappointing because testing produces better code and makes it easier to safely implement changes in your code. Too many articles about testing just argue its merits or excoriate people who don't use it, so instead I'm going to show you how easy it is to start testing your code and introduce you to a simple testing framework that is suitable for use with the kinds of programs sysadmins tend to write.

Before I go any further, let me explain exactly what I mean by testing. Good programs include defensive code—things like validating input, checking for errors, etc.; that is not testing. Rather, testing is writing a separate program that exercises your "real" code and proves it actually works. Typical test programs run your real program with a representative sample of valid and invalid inputs and verify that the responses from your program are correct. By having a separate program, you can make changes to your real code, and if all the tests still pass, it's likely that you didn't break anything.

System Administration Meets Software Development

Infrastructure as Code. You've all heard it before, and you probably have an idea of what it means. To me, one aspect of Infrastructure as Code is bringing the discipline of software engineering to bear on system administration. Software engineering covers a lot of ground, so let's focus on just three practices used by software engineers: version control, code reviews, and testing.

Pretty much everyone agrees version control is a good idea. You can use Git, Subversion, Perforce, or something else—just pick one and use it; all of them are better than not using any of them.

If you're doing code reviews, great; if not, they're trivial to start when you decide you're ready. I'm betting you can figure out how to do them without much help. You don't need a tool to do code reviews but using one can make the job easier. I like the tool ReviewBoard [4] but, again, there are plenty of other good tools.

Testing is where it gets interesting. Some of you may be testing your Puppet modules and Chef recipes, but what about all those other programs you wind up writing; do you write tests for those? To quote Yosemite Sam, "Dem's fight'n words!" Over the years, I've seen more and more sysadmins embrace the idea of using version control, and some are slowly embracing code reviews, but testing still seems to meet plenty of resistance.

As a software developer, my day looks something like this: Pick a task off the backlog (typically a task that needs to be automated), write some code and some tests (in either order or in parallel), keep working at it until the task is completed and all the tests pass, then submit a code review. Assuming my reviewer(s) liked what I wrote, I commit the code into our version control system. For the past year I've been writing most of my code in Groovy [2], using

Maven [3] to build my projects, and testing everything with Spock [5]; these were chosen as standards for my group before I was hired, and so far I haven't had any reason to change. All that was fine until one day I had to write a very "sysadmin-y" program in Bash, and the person reviewing my code said, "Where are your tests?"

Now, at my company, code reviews are pretty much required, but as the developer I have the option of choosing which changes requested by the reviewer I implement, including not implementing any at all. I would have been on solid ground to have said something like "No one writes unit tests for shell programs" or "It's too hard to write tests for shell," but that just didn't feel right to me. "Would it really be that hard to come up with some meaningful tests for this shell script?" I asked myself. I couldn't shake the feeling that I could come up with something that wouldn't feel like Rube Goldberg invented it, so I decided to spend a few hours on the problem.

By the end of the day I had enough of a skeleton developed that I believed I had found a viable solution; by the end of the next day I had a fully fleshed-out system, and the next morning (48 hours after the first code review), I submitted a second review with the comment, "Full tests for the Bash code are now included." Because I'm lucky enough to have time to take small detours when I find something that may be useful for my colleagues, I took a third day to build an example Bash program and set of tests to see just how far I could reduce the framework code. In the end I got it down to 57 lines and a very simple directory hierarchy. I'll show it to you in just a moment.

Arguments Against Software Testing

The most common reasons I've heard for not doing software testing fall into one of the following categories:

- Test harnesses are too hard to understand and too difficult to set up.
- It's too hard to test the kind of programs I write.
- It takes too much time.

Rather than get all preachy about it, I'm going to take the rest of this article to show you a technique I've been using for the past year that I believe will address at least the first two objections; I'll deal with the third objection later. Let's get to it, shall we?

Introducing a Testing Framework

There are more unit test frameworks than I can count. The Wikipedia article lists over 400 of them; Java alone has 35 different frameworks; Perl, Python, Ruby, and even Shell each have around eight. No wonder folks don't know where to start. As I wrote above, my solution is based on Groovy (which requires the Java JDK), Maven, and Spock; all are quite powerful and, in their full depth, are somewhat complicated, but I'm going to stick to a very small subset to keep things simple.

The setup is trivially easy: download the Java JDK gzipped tar file, unpack it somewhere (I like `/opt/<thing>-<version>` but you can do this all in `$HOME` if you prefer), make an optional symbolic link, and add one environment variable to your preferred shell's start-up file. Repeat for Groovy (a zip file) and Maven (requires two environment variables). When you're done, update your path. That's it: no installing dozens of packages, no dependency hell, no spending hours getting lots of little pieces all in the right places. The first time you test a program, Maven will automatically download Spock for you. A full set of detailed instructions can be found on my Web site [1].

The next step is to lay out your program source code, your test code, and tie it all together with a Maven `pom.xml` file. There's a link to these steps at [1], so I'll just show you what files go where:

```
-rw-r--r--  example/pom.xml
-rwxr-xr-x  example/src/main/bash/example
drwxr-xr-x  example/src/main/resources
-rw-r--r--  example/src/test/groovy/Test_example.groovy
drwxr-xr-x  example/src/test/resources
```

The file `example/src/main/bash/example` contains the program to be tested, and `example/src/test/groovy/Test_example.groovy` contains the test code; the former is written in Bash, the latter in Groovy. The two resource directories are there to keep Maven from complaining that they don't exist.

"Wait, Groovy?!?! I have to learn yet another programming language? Is this guy for real?" Please, there's no need to shout. No, you don't have to learn a new language, just a few constructs, and most of those are identical to Perl (which you probably already know); I'll give you enough examples of the new bits that I'm betting you'll be able to pick it up with very little work.

Here's our example program:

```
#!/bin/bash
if [ $1 = yes ] ; then
   echo hello, world
   exit 0
else
   echo goodbye, cruel world 1>&2
   exit 1
fi
```

As you can see it doesn't do anything useful, but it does just enough to let me demonstrate the three most basic Spock tests: testing the exit value of a program, examining standard out, and examining standard error. Here's the file `example/src/test/groovy/Test_example.groovy`:

```
1 package com.menlo.example
2
3 import spock.lang.*
4
5 class Test_example extends Specification {
6     static String here = System.getProperty("user.dir")
7     static String prog = "${here}/src/main/bash/example"
8
9     def "exits with 0"() {
10        when:
11            Process p = "${prog} yes".execute()
12            p.waitFor()
13
14        then:
15            p.exitValue() == 0
16            p.text.contains("hello, world")
17    }
18
19    def "exits with 1"() {
20        when:
21            Process p = "${prog} no".execute()
22            p.waitFor()
23
24        then:
25            p.exitValue() == 1
26            p.err.text.contains("goodbye, cruel world")
27    }
28 } // Test_example
```

For now, skip over lines 1–8; I'll cover them in the next paragraph. The first test is lines 9–17: line 10 defines the "stimulus" block, and line 14 defines the "response" block; that is, "given certain actions, confirm that certain results are true." In this case, run our program with the argument "yes" (line 11), then check that the exit value is zero (line 15) and that we get "hello, world" on standard out (line 16); you can ignore line 12 even though it's required. The second test (lines 19–27) is nearly identical to the first, the main difference being that we examine standard error instead of standard out (line 26).

For our purposes, that is, when testing programs written in anything other than Java or Groovy, lines 1–5 don't matter as long as they exist; it won't hurt anything if they're identical for every test file you ever write. Lines 6 and 7 define two variables that let us find our Bash program in a portable way; in other test files the only thing you'll need to change is `example` in line 7 to the name of the program being tested. (Obviously, if you're testing a Perl program, you'll also need to change `bash` to `perl`.)

To run the tests, just type `mvn test` and watch a few hundred lines of output go scrolling by; don't try to read it all, the important bits will be at the very end. You should see lines that look like this:

```
Running com.menlo.example.Test_example
. . .
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
. . .
[INFO] BUILD SUCCESS
```

What I haven't shown you, and what I'm going to ask you to take on faith, is how Maven knows what to do. It's all contained in the file `pom.xml`, and the example you can download from my Web site has everything you need; there's no need to modify the file or even look inside it; just put a copy in the top directory of each Maven project and you'll be fine. My Web site also contains a small shell script, `new-testing-project`, that will create new Maven project directories for you, populate them with skeleton files, and drop a fully formed `pom.xml` into place.

Testing Scripts

For this next bit, let's agree to call the programs that sysadmins write "scripts," only because it will provide a convenient shorthand; also, let's agree to call things like `mount` and `ifconfig` "system programs," again, for convenience.

One of the challenges with testing scripts is they often rely heavily on system programs; a typical script takes an argument or two from the command line, uses it to call a system program, captures the output of that program, manipulates it, then passes it to a second system program. Most of the tricky code in the script is dedicated to parsing the output of the system programs, trying to extract the desired pieces or detect an error. In many cases, actually running the system programs is destructive or produces an undesirable result, or there's no easy way to cause the system program to fail (so the script's error-checking code can be, um, checked). Fortunately, there's a technique common in modern software development that can be used, after a fashion, for testing scripts as well; this technique is often known as "mocks" or "stubs."

USING MOCKS IN PLACE OF SYSTEM COMMANDS

The idea behind a mock is simple: rather than run `/bin/mount`, we run our own private imitation of `mount` that emits whatever output we need for our tests but doesn't affect the state of the system. Some scripts call system programs via an explicit path—for example, `/bin/mount -l -t ext3`—instead of relying on `mount` being somewhere in `$PATH`. There's something to be said for that style but it makes testing impossible. While you could depend on `$PATH`, making sure to set it at the top of every script you write, the alternative I prefer is to use variables for each system program I call. By doing it like this:

```
MOUNT=${TESTBIN:-/bin}/mount
```

you're still protected from an incorrectly set `$PATH` while, at the same time, having the flexibility to run a mock during testing.

Here's a second example that uses mocks:

```
-rw-r--r--  example2/pom.xml
-rwxr-xr-x  example2/src/main/bash/example2
drwxr-xr-x  example2/src/main/resources
-rw-r--r--  example2/src/test/groovy/Test_example2.groovy
drwxr-xr-x  example2/src/test/resources
drwxr-xr-x  example2/src/test/resources/bin
-rwxr-xr-x  example2/src/test/resources/bin/mount
drwxr-xr-x  example2/src/test/resources/data
-rw-r--r--  example2/src/test/resources/data/mount.error
-rw-r--r--  example2/src/test/resources/data/mount.single
-rw-r--r--  example2/src/test/resources/data/mount.separate
```

The files `mount.single` and `mount.separate` contain the output from `mount` on systems with everything on a single partition and with /, `/home`, `/tmp`, `/usr`, and `/var` on separate partitions.

Our mock `mount` command is trivially short:

```
#!/bin/bash
cat $MOCK_MOUNT_FILE
exit $MOCK_MOUNT_EXIT
```

Obviously, if your script calls `mount` more than once in a single run, a more sophisticated mock is needed. Finally, here's how we tie all this together inside `Test_example2.groovy`:

```
package com.menlo.example2

import spock.lang.*

class Test_example2 extends Specification {
   static String here    = System.getProperty("user.dir")
   static String bin     = "${here}/src/test/resources/bin"
   static String data    = "${here}/src/test/resources/data"
   static String prog    = "${here}/src/main/bash/example2"
   static String wrapper = "${here}/target/example2"

   def "test all on one partition"() {
   setup:
      File f = new File(wrapper)
      f.delete() // make sure we start with a new file
      f << "#!/bin/bash\n"
      f << "export MOCK_MOUNT_FILE=${data}/mount.single\n"
      f << "export MOCK_MOUNT_EXIT=0\n"
      f << "export TESTBIN=${bin}\n"
      f << "${prog}\n"
      f.setExecutable(true, false)

   when:
      Process p = "${wrapper}".execute()
      p.waitFor()

   then:
      // your tests here

   cleanup:
      assert new File(wrapper).delete()

   }
} // Test_example2
```

Each subsequent test would have to duplicate the `setup` and `cleanup` stanzas, substituting values for `MOCK_MOUNT_FILE` and `MOCK_MOUNT_EXIT` as appropriate.

You may have noticed the directory `target` in the definition of `wrapper`; this is where Maven puts all temporary files. When you're done testing a particular script, run the command `mvn clean` to clean up.

Mocks can be quite complicated, and I could probably fill an entire article on how to get fancy with them; for now I think I've left you with enough to get started.

"It Takes Too Long"

The last reason people give for not writing tests—"it takes too much time"—is often the most difficult to respond to, but I hope by now I've established enough credibility that you'll at least read my argument.

To me, testing is kind of like insurance: if you never need it then the money you spend on it is "wasted," but it takes only one accident (or failure or whatever) for every dollar you've paid in premiums to be returned ten-fold (or more). But software will inevitably fail, and writing software is hard. There are far more variables, edge cases, and unknowns involved in software than any one person can understand, and even the best developers are far from perfect. Put together, it's not a question of whether any given piece of software will fail but, rather, when it will fail and how much damage the failure will cause.

Of course, tests are themselves software and thus will fail, but my 30+ years of experience tells me that the time typically put into writing tests catches at least 80% of the bugs; I also know that writing tests makes you approach software development differently and results in more correct (or, at least, more robust) software. To me, producing better software far outweighs the fact that testing is not a "silver bullet" and that your software may still fail. In other words, apply the Pareto Principle (aka "the 80-20 rule") and avoid letting perfection get in the way of improving your work.

The other big benefit to testing is that it lets you make changes to your software; not only can you tell that the new code works, you can be confident that you haven't broken your old code. For example, if you wrote a program to run under CentOS but now want it to work on Ubuntu as well, you can run the tests to prove it works on CentOS, modify the code, and add tests for Ubuntu. Once the new stuff is working, go back and run the old tests on CentOS to see that your program still works there.

So there you have it: I've shown you how to install and configure a simple test harness, and how to write basic tests; I've given you a brief introduction to mocks; and I've offered an argument to justify spending the extra time to write tests for your programs. Now it's up to you to decide whether to apply what I've shown you the next time you have to write a program as part of your job as a system administrator.

Resources

[1] http://menlo.com/tdd4sa/.

[2] http://groovy.codehaus.org/.

[3] https://maven.apache.org/.

[4] https://www.reviewboard.org/.

[5] https://code.google.com/p/spock/.

XKCD