developer.*
A Web Magazine for Software Developers

Diving in Test-First

by Danny R. Faught

Do you test your code before you send it downstream? Come on, tell the truth! How much confidence do you have that another code change won't break something? Some of us take testing to an extreme. I want to show you a real-world example that will give you a brief glimpse of how test-first development can greatly increase your confidence in your code.

What is test-first?

When I first heard about the idea of writing tests before writing the code, I thought it was crazy, because I thought it referred to independent testers, i.e., people who focus on testing other people's code. Developers always change the spec while they're coding, so why bother to code the tests to a spec that will be out of date in nanoseconds? But test-first applies to testing done by developers, and in that context it makes great sense. The test development is tightly intermingled with the coding. *Really tightly*.

Kent Beck describes a five-step cycle for test-driven development [Beck, 2002, p 7]. The steps are:

1. Add a little test.
2. Run all tests and fail.
3. Make a little change.
4. Run the tests and succeed.
5. Refactor to remove duplication.

Test-driven development (I often use "test-first" to mean the same thing) is one of the core practices in Extreme Programming (XP). I've felt that teams who don't adopt XP whole-hog can still benefit greatly from using some of the practices. I'm very happy to see test-first catching on more widely now.

XP practitioners almost always speak of their programming practices with great passion. Once you buy into test-driven development, you'll probably never turn back.

In this article, I'm going to illustrate the test-first cycle. But I'm going to really put it to the test. I'm not going to use a toy program developed yesterday using the latest programming techniques; I'm going to use a legacy program that I wrote ten years ago. Can we find the passion using a real-world programming challenge? Let's find out.

An ugly real-world example

In 1993, I wrote the `stress_driver`, a generic stress test program implemented in Perl and designed to mercilessly flog an operating system. Somehow I've found myself testing operating systems again now, and as luck would have it, my former employer has open-sourced the code. I've started using test-first practices when I tinker with the `stress_driver` code, though the tests I have so far only cover a small fraction of the code.

I teach tutorials for testers who want to learn how to use Perl for test automation, and I use the `stress_driver` as an example in my tutorials. We found a bug in the program during one tutorial, and I fixed the bug on the spot in order to demonstrate the Perl debugger. None of the existing tests found the bug. If I had been thinking test-first, I would have written a test before trying to fix the bug.

I never got around to folding the fix back into my main source base. So let's take a step back and look at the way the process should have gone, while we apply the fix back at the home office.

I'll do step one of the test-driven cycle and "Add a little test." It will be an automated test, of course, but first, I'll run a test by hand to demonstrate the failure. I give the `stress_driver` something to occupy itself with, then after a few seconds, I press `CTRL-C`:

```
> stress_driver -- 20 /bin/sleep
starting stress_driver, logging to 'stress_driver.log.4273'
stress_driver: aborted by SIGEvent::Event=SCALAR(0x82a0c58)->data()
```

It's not really important for this example how the `stress_driver` works, except for this bizarre output: `SIGEvent::Event=SCALAR(0x82a0c58)->data()`. I remember programming `stress_driver` to print out the signal name if it gets killed by a signal, in this case, "SIGINT". But instead, I'm getting gobbledygook. Here's the `stress_driver` code that's supposed to handle this:

```
sub abort_handler {
    # data passed in is the signal that killed stress_driver
    my ($event) = @_;
    print STDERR "stress_driver: aborted by SIG$event->data()\n";
    &log ("stress_driver: aborted by SIG$event->data()");
    exit (&cleanup);
}
```

I'd really like to write a proper unit test here. A unit test would directly call the subroutine I want to test, and it would rely as little as possible on other subroutines in the program. So I experiment with Perl's `Test::Unit` module, and while the module works fine, I pull my hair out trying to stub out the functions that `abort_handler` calls. Perhaps only purists insist on stubbing out every function call so we're testing in true isolation from the other code in the program. But most troublesome is the fact that this subroutine exits the program—not a nice thing to happen in the middle of a unit test run! The test also needs to stub out Perl's built-in exit function.

So I punt, and use the same black-box test framework that I've been using so far, that calls `stress_driver` from the command line and uses the `Test::More` module as the test framework. The result is shown below, the "abort" test:

```
#!/usr/bin/perl

use warnings;
use strict;
use Test::More tests => 1;
use sdsetup;

our ($sdsleep);
my ($outfile) = "sd_sigint.out";
my ($pid);
unlink($outfile);
if (0 == ($pid = fork())) {
    open(STDOUT, ">$outfile");
    open(STDERR, ">&STDOUT");
    exec "$ENV{testprog} -- 10000 $sdsleep";
    die "exec error: $!";
}
elsif ($pid < 0) {
    die "fork error: $!";
}
sleep (3);  #hack warning - race condition
die "kill error" unless kill('SIGINT', $pid) == 1;
wait;
die "error running stress_driver" if $?;

like `grep 'aborted by' $outfile`, qr/aborted by SIGINT/,
    "checking stress_driver output after SIGINT";
exit 0;
```

It's far more complex than a unit test would be. It's even worse because I had trouble when I used `bash` to run `stress_driver` and redirect the output, so I had to do it the hard way with fork/exec. And it has a race condition—if `stress_driver` takes more than three seconds to initialize, I may get a false failure.

The second step in Beck's cycle is "Run all tests and fail." What that means is that the new test should fail and all others should pass. I have to admit that several of the existing tests were failing the last time I tried them on Linux. I had been developing under Cygwin lately—I tend to migrate from one system to another in my work on `stress_driver`. But I do have what's most important - a test that demonstrates the newly discovered bug:

```
> ./abort
1..1
not ok 1 - checking stress_driver output after SIGINT
#     Failed test (./abort at line 26)
#                    'stress_driver: aborted by
SIGEvent::Event=SCALAR(0x82a12cc)->w->data()
# '
#     doesn't match '(?-xism:aborted by SIGINT)'
```

Now the third step, "Make a little change." I apply the fix that I had written during my tutorial, which involved these two lines of code in `abort_handler` (I changed `$event->data` to `$event->w->data`).

```
print STDERR "stress_driver: aborted by SIG" . $event->w->data() .
"\n";
&log ("stress_driver: aborted by SIG$event->w->data()");
```

Step four is "Run the tests and succeed." I cheat again, only running this one test, but it does succeed.

```
> ./abort
1..1
ok 1 - checking stress_driver output after SIGINT
```

Now step five in the cycle, "Refactor to remove duplication." Clearly, there's duplication in the code I was modifying—the same thing is printed to two different places. But before I refactor, I want to test the output to the log file. I make a change to the `abort` script to add the subtest. First, as much as I don't like to have to count them by hand, I need to tell the harness that there are two tests:

```
use Test::More tests => 2;
```

I specify a log file name rather than letting `stress_driver` generate one based on the process ID:

```
my ($logfile) = "sd_sigint.log";
...
exec "$ENV{testprog} -log $logfile -- 10000 $sdsleep";
```

And I add a subtest, again using a regular expression to match the output:

```
like `grep 'aborted by' $logfile`, qr/aborted by SIGINT/,
    "checking stress_driver log after SIGINT";
```

I expect the test to pass, but instead I get a surprise.

```
> ./abort
1..2
ok 1 - checking stress_driver output after SIGINT
not ok 2 - checking stress_driver log after SIGINT
#     Failed test (./abort at line 30)
#                   '[12/12/102 00:51:40] stress_driver: aborted by
SIGEvent::Event=SCALAR(0x82a1564)->w->data()
# '
#     doesn't match '(?-xism:aborted by SIGINT)'
# Looks like you failed 1 tests of 2.
```

Dang. There were actually two bugs in the original `stress_driver` code. One was that I needed to find my data in the "w" object. The other was that the complex method call was too much for Perl's string interpolation to figure out (plus you'll see a Y2K bug if you look closely). If you go back and look at the two changed lines of code in `abort_handler`, you'll see that I had fixed the interpolation problem on the first line, but I never bothered to check the log file to see that if I had fixed it properly on the second line. I decide to fix the bug by restructuring the code; I suppose I can't call it refactoring now since I'm also changing/fixing the external behavior. I delete the errant code, and add a new function to deal with what's left:

```
&duo_log ("stress_driver: aborted by SIG" . $event->w->data());
...

sub duo_log {
    my ($msg) = @_;
    print STDERR "$msg\n";
    &log ($msg);
}
```

I run the tests again, and do the dance of joy.

```
> ./abort
1..2
ok 1 - checking stress_driver output after SIGINT
ok 2 - checking stress_driver log after SIGINT
```

Now to repeat the cycle back at "Add a little test," or as is often the case with skunkworks projects, I'll get distracted by other things such as finishing writing an article.

Learnings from the deep end of the pool

I learned a lot from this little exercise. Test-first is supposed to give you great confidence in your code because you have a high degree of test coverage, and you can run the tests frequently because they run quickly. However, because I'm adding tests for legacy code a few at a time, I don't have good test coverage yet. Because I tested from the user interface rather than at the unit level, the tests aren't as efficient as I'd like. It takes the better part of a minute to run the tests I have now (including several others besides the abort test). That time will grow quite a bit as I add more tests. I need lightning-fast feedback so that execution time isn't an excuse for not running the tests after every small code change.

I don't like the complexity of the test code. I intend to further investigate a unit test framework so my tests can be greatly simplified. The fact that my code runs on more than one platform was also a deterrent. I'm still trying to develop the discipline of making sure all the tests pass before I move on, but having to run on more than one platform makes this even harder.

So was the experiment a failure? Certainly not. Transitioning to a new technique is rarely easy. I'm very happy that I actually succeeded at using the test-first approach. I ran into a number of difficulties, but none of them were showstoppers. If I didn't do the tests first, I might never get around to writing them at all. My confidence in the code will grow as I get more tests running. Most importantly, I have much more confidence in the code that I happen to be working on, because I'm adding tests for everything I touch. I'm committed to continue to go test-first. I think the passion is contagious.

Thanks to Lisa Crispin, Ron Jeffries, and Brian Marick for their assistance with this article.

Further reading

- For more about test-driven development (for the most part described independently from the rest of XP) see Kent Beck's new book *Test-Driven Development: By Example*, or the article "Learning to Love Unit Testing" (`http://www.pragmaticprogrammer.com/articles/stqe-01-2002.pdf`) by Dave Thomas and Andy Hunt.
- See also my article "Test-First Maintenance: A Diary" (`http://tejasconsulting.com/DFWUUG/test-first-maintenance.html`) to read about my earlier explorations of the test-first concept using `stress_driver`. You'll find the `stress_driver` code I used for this article, plus a pointer to the latest version, at `http://tejasconsulting.com/stress_driver/`.

References

[Beck, 2002] Beck, Kent. *Test Driven Development: By Example*. Boston: Addison-Wesley, 2002.

###

Danny R. Faught is proprietor of Tejas Software Consulting, where he does consulting and training on software quality issues. Danny is the publisher of Open Testware Reviews. You can reach him at `faught@tejasconsulting.com` and `tejasconsulting.com`.