

My Own Test Module

by Jonas B. Nielsen
jonasbn@cpan.org



Only very late in my career I learned the meaning of the term *syntactic sugar*. Now I am writing an article where the main thread is about writing a Perl test module and this is actually about syntactic sugar.

So what is syntactic sugar? Syntactic sugar is computer language constructs that present a certain syntax in a more readable and understandable manner but really adds nothing anything new to the language.

Most statements and programs can be written in numerous ways, some more obscure than others. Some programming languages then offer constructs to increase readability. This is often referred to as syntactic sugar.

Perl can be extended with modules, many of these are available with the core distribution and others as CPAN distributions. These extend the capabilities of Perl to solve all sort of problems. Most of the modules on CPAN do as such not solve a problem, which cannot be solved using the basic language constructs of Perl and many of the test modules fall in this category.

These modules work in a way by extending and specializing the actual language in such way that cumbersome bodies of code or obscure snippets can be written out in a simple manner, sweetening the experience for the programmer, and enhancing the Perl code with syntactic sugar. Complex tasks become simple subroutine calls.

■ Test Modules -----

The focus of this article is on test modules and writing my own. I base the examples in this article on my own fairly new module, `Test::Timer`, which tests that a certain operation does not take over a specified amount of time. The development of this module sparked the idea of writing this article based on my experiences with writing a test module for use with Perl.

The Perl test modules all work in a similar manner because most of them rely on the `Test::Builder` module to handle most of the hard work. Of existing test modules I can mention three which are useful for my task:

- `Test::Simple`
- `Test::More`
- `Test::Benchmark`

There are plenty other `Test::*` modules, but perhaps you work with a special domain or the general test modules does not address your particular problem area so you want to create your own.

■ Test::Timer -----

I created `Test::Timer` to scratch an itch I had with some work done with a client using Perl to develop a web-based front-end utilizing a web-service-like layer of XML-over-HTTP to talk to some diverse telco and back-office back-ends.

These back-ends, however, did not always respond within the expected time frame, and, instead of spending time in a terminal timing tests or small scripts, I implemented `Test::Timer`. The goal was to simply define some response times and if the back-ends did not respond within these, the test would fail.

I had a look at a module, `Test::Benchmark`, which looked a lot like what I wanted to accomplish. I wanted to use the same scheme as `Test::Benchmark` to accomplish my goal, but with a slight difference: the module should still benchmark the code, but use the result of the benchmark in comparison to the threshold times the test author specifies.

As do many of the other test modules, I also need to be able to invert the tests. Since these are simple assertions, sometimes

Code Listing 1: Timing code

```
use Benchmark;
use Test::More tests => 1;

#Testing that we at-most take 3 seconds
{
    my $t0 = new Benchmark;

    #my code sleeping here
    sleep(2);

    my $t1 = new Benchmark;
    my $td = timediff($t1, $t0);
    print timestr($td)."\n";
    my ($time) = timestr($td) =~ m/^(\\d+)/;

    ok($time < 3);
}
```