Table of Contents

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Chapter 7. Developer Tricks

Hacks 60-69

Surviving software development and enjoying it are two very different things. Do you know what your code is doing? Can you look at any piece and know where it belongs and what it means? Do you trust your code? Do you trust your coworkers? What can you do to take back control of your projects, code-wise?

Obviously reducing the friction of writing code will make your life easier, but what about the friction of *designing* and *maintaining* code? Comprehensive testing and collective code standards help. Here are a few ideas to bring up in your next developer meeting that will make you a hero.

Hack 60. Rebuild Your Distributions

Rebuild your distributions with ease.

If you work with Perl modules built in the standard CPAN format (and you should, as the many available tools make your life easier this way), you generally will have a *Makefile.PL* or *Build.PL* file, *lib/* and *t/* directories, manifests, and so on. If the module uses `ExtUtils::MakeMaker`, you change your tests, update the module and rebuild the distribution again with a command such as:

```
$ make realclean && perl Makefile.PL && make && make test
```

Modules that use `Module::Build` require instead:

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
$ ./Build realclean && perl Build.PL && perl ./Build && perl ./Build test
```

It gets annoying typing this over and over again. Worse, if you do this for patches you send to others, you might forget and assume you have a *Makefile* when using `Module::Build` or a *Build* file when using `ExtUtils::MakeMaker`. This is tiresome.

The Hack

Instead, put this *rebuild* script in your path and set the appropriate permissions:

```
#!/bin/bash

if [ -f Build.PL ]; then
    makeprog=Build
    makecommand="perl ./Build"
elif [ -f Makefile.PL ]; then
    makeprog=Makefile
    makecommand=make
else
    echo Nothing to reload!
    exit 1
fi

if [ -f $makeprog ]; then
    $makecommand realclean
fi
perl $makeprog.PL && $makecommand && $makecommand test
```

Running the Hack

Whenever you want to rebuild your project, type `rebuild` at the command line in the parent directory of the project and don't worry about whether you're using `Module::Build` or `ExtUtils::MakeMaker`.

Hacking the Hack

If you really want to get carried away, bash scripts put their command line arguments in variables named `$1`, `$2`, and so on. It's trivial to add extra commands to build your distribution, your manifest, or whatever else you like:

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
if [ "$1" = dist ]; then
    $makecommand dist
fi
```

Hack 61. Test with Specifications

Let the computer write your tests.

Writing tests is a great way to gain confidence in your code. Each test you write makes a tiny claim about what your code ought to do. When it passes, you have clear evidence to support the claim. If you write enough tests to make a cohesive suite, the tiny claims within the suite combine to imply a general claim that your code works properly.

There are times, however, when writing a suite of tests is the hard way to make a general claim. Sometimes, the claim you want to make seems so simple, yet the tests you have to write seem so voluminous. For these times, it would be nice to be able to turn the process around. Instead of writing tests to make a claim, why not make the claim outright and let the computer write the tests for you? That's the idea behind specification-based testing. The `Test::LectroTest` family of modules brings this idea to Perl.

The Hack

To make claims about your code, you define *properties* that say that your code must behave in particular ways for a general spectrum of conditions. Then LectroTest automatically gathers evidence to support or refute your claims by executing large, random test suites that it generates on the fly. When it finishes, it prints the results in standard TAP format, just as the other Perl testing tools do.

Suppose you need to test Perl's `sqrt` function, which you expect to compute the square root of its argument. The first step is to figure out what `sqrt` ought to do. From your school days, you recall that the square root of a number *x* is the number that when multiplied by itself (that is, *squared*) gives you back *x*. For example, the square root of 9 is 3 because 3 times 3 is 9.

The square root function undoes the effect of squaring. You could consider this the defining property of the function. Putting it more formally, you might say: "For all numbers *x*, the

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

square root of *xx x* should equal *x*." To test this claim, all you need to do is restate it as a LectroTest property:

```
# loads and activates LectroTest
use Test::LectroTest;

Property
{
    ##[ x <- Float ]##                  # first  part
    sqrt( $x * $x ) == $x;              # second part
}, name => "sqrt is inverse of square";  # third  part
```

This is a complete Perl program that you can run. It tests a single property, specified in three parts. The first part (in the funny brackets) specifies the domain over which the property's claim should hold. Read it as saying, "For all floating-point numbers *x*..."

LectroTest actually offers a tiny language for declaring more complex domains, but it's not necessary here.

The second part is a snippet of Perl code that checks the property's claim for a given value of *x*. If the claim holds, the code must return a true value; otherwise, it must return a false value. In this property, read the code as saying, "...the square root of *xx x* should equal *x*." For convenience, LectroTest makes the variables mentioned in the first part of the property available in the second part as lexical variables—`$x`, here.

Because of the imperfections of floating-point arithmetic, a more robust way of testing this claim would be to check whether the difference between `$x` and `sqrt($x * $x)` is within an acceptably small range. For simplicity, however, I've used a straight equality test, which could result in a false negative test result.

The third part gives the property a name. It's optional but adds a lot of documentation value, so give your properties meaningful names.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Running the Hack

To test whether your claims hold, just execute the program that contains your properties. In this case, you have only one property, so the program's output might look like:

```
1..1
not ok 1 - 'sqrt is inverse of square' falsified in 3 attempts
# Counterexample:
# $x = "-1.61625080606365";
```

Here, LectroTest says that, for some value of *x*, it was able to prove your property's claim false. It provides the troublesome value of *x* as a counterexample that you can use to figure out what went wrong.

Refining your claims

In this case, what went wrong is that the property made an overly broad claim. The square root function only applies to non-negative numbers (ignore imaginary numbers for this hack), and yet the property made its claim about *all* floating-point numbers, which includes those less than zero.

This illustrates an important benefit of the specification-based approach to testing: because it forces you to make your claims explicit, it can reveal hidden assumptions and holes in your thinking. Now you must consider what `sqrt` *should* do when given a negative number. For Perl, it probably ought to result in an error, so you might revise your property:

```
# helper:  returns true if calling the given function
# results in an error; returns false otherwise

sub gives_error(&)
{
    ! eval { shift->( ) } and $@ ne "";
}

Property
{
    ##[ x <- Float ]##
    $x < 0 ? gives_error { sqrt( $x ) }
           : sqrt( $x * $x ) = = $x
}, name => "sqrt is inverse of square and dies on negatives";
```

You also could make `sqrt`'s split personality more obvious by writing two properties that together define its behavior:

```
Property
{
    ##[ x <- Float ]##
    $tcon->retry( ) if $x < 0;      # only test non-negatives
    sqrt( $x * $x ) = = $x;
}, name => "sqrt is inverse of square";

Property
{
    ##[ x <- Float ]##
    $tcon->retry( ) unless $x < 0;  # only test negatives
    gives_error { sqrt( $x ) };
}, name => "sqrt of negative numbers gives error";
```

Calling `$tcon->retry( )` tells LectroTest to retry a test case you don't like, starting over with a brand new, random case. Use this call in your properties to subject only sensible cases to your tests. In the first property, for instance, the conditional call to `$tcon->retry( )` ensures that LectroTest subjects only non-negative values of *x* to the `sqrt`-is-the-inverse-of-squaring test.

The LectroTest-provided `$tcon` object lets you talk to the underlying test controller to do all sorts of interesting things. See the LectroTest documentation to learn more.

You now have two properties. The first claims, "For all non-negative floating-point numbers *x*, the square root of *xx x* should equal *x*." The second claims, "For all negative floating-point numbers *x*, attempting to take the square root of *x* should result in an error." With practice, it becomes easy to convert LectroTest property specifications into written claims and vice versa. These two claims seem to cover all of the bases, and so it's time to put them to the test.

Interpreting test output

When you run the two-property suite, you get the results:

```
1..2
ok 1 - 'sqrt is inverse of square' (1000 attempts)
ok 2 - 'sqrt of negative numbers gives error' (1000 attempts)
```

Good news! LectroTest subjected each claim to 1,000 random test cases and was unable to find any problems. Still, this happy outcome doesn't *prove* that `sqrt` works as expected. It merely gives you evidence to support that conclusion. Certainly, the evidence is persuasive—you would think that 2,000 tests ought to be enough to flush out any obvious problems—but it's important not to lose sight of the limitations of testing.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

In light of the evidence, though, there is probably no need to test further. The existing results argue persuasively in favor of `sqrt`, and there's no reason to think there are special circumstances that might make the current degree of testing inadequate. The only corner case in sight is the negative-number case, and you have that covered. At this point, you can probably rest satisfied that `sqrt` does the right thing.

Taking advantage of specification-based testing

With the specification-based approach, a little testing code goes a long way. It only took about fifteen lines of code to test `sqrt` fairly rigorously.

Another strength of the approach is that the claims embodied in your testing code are easy to see—just read the properties. These claims are useful beyond their testing value, serving as formal documentation of what you expect your software to do.

Because of these strengths, specification-based tests make a great complement to hand-written tests. When one approach to testing seems difficult, the other is often easy, and combining the approaches makes many complicated testing problems easier. For this reason, you ought to have both approaches in your toolbox.

Specification-based testing is a deep and interesting topic, and this hack only scratches its surface. To learn more, including how to use it with more-traditional Perl testing tools, the documentation for `Test::LectroTest` is a good starting point.

Hack 62. Segregate Developer and User Tests

Run only the tests you need when you need them.

In general, the more tests you have for a system the better. In specific, the more tests you have, the more time it takes to test your code. In very specific, some tests are more valuable than others. You may reach 95% confidence by running a few test files (*user tests*) and that may be good enough for day-to-day operations. You may also have a few deeper tests (*developer tests*) that use external resources or take a long time to explore all of the potential

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

possibilities for failure—and you don't necessarily want to make everyone run them all at once.

If your project uses Perl's standard module-building tools (at least, `Module::Build`, which comes very highly recommended), you can segregate developer and user tests *very* easily, running the time-consuming tests only when you need to, perhaps right before a release.

The Hack

To customize `Module::Build`'s behavior, almost all you have to know is to define your own subclass and override the appropriate `ACTION_*` method. How about running only the user tests with the normal `perl ./Build test` invocation and all tests with `perl ./Build disttest`?

It's helpful to skim the source of Module::Build::Base [Hack #2] when overriding an action. That's where you can learn that the next method to override is `find_test_files( )`. The `ACTION_test( )` method calls this to figure out which tests to run. Override the test finder method to filter out developer tests. Easy!

Don't celebrate just yet, though: `ACTION_disttest( )` launches another Perl process to run *Build.PL* and, eventually, `perl ./Build test`. Because this is another process, there's no easy way to set a flag or a Perl variable to tell the *second* invocation of `ACTION_test ( )` to run all tests. Fortunately, you *can* set an environment variable, perhaps `PERL_RUN_ALL_TESTS`, that both the parent and child can see.

All that's left is to decide where the developer tests are; how about in *t/developer/*.t*? That's enough to write a `Module::Build` subclass:

```
package Module::Build::FilterTests;

use base 'Module::Build';

use SUPER;
use File::Spec::Functions;

sub ACTION_disttest
{
    local $ENV{PERL_RUN_ALL_TESTS} = 1;
    super( );
}

sub find_test_files
{
    my $self  = shift;
    my $tests = super( );
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
    return $tests unless $ENV{PERL_RUN_ALL_TESTS};

    my $test_pattern = catfile(qw( t developer *.t ) );
    push @$tests, <$test_pattern>;
    return $tests;
}

1;
```

The `SUPER` module makes calling the parent implementations of overridden methods a little cleaner syntactically. The only other notable feature of the test is the use of the `glob` operator to find all tests in the *t/developer/* directory.

Running the Hack

In your own *Build.PL* file, load and instantiate a `Module::Build::FilterTests` object instead of a `Module::Build` object. Everything else should proceed as normal.

The easiest way to distribute a custom `Module::Build` subclass is to distribute it by storing it in *build_lib/* or another directory and to use it from *Build.PL* with a `use lib` line.

```
$ perl Build.PL
Checking whether your kit is complete...
Looks good
Deleting Build
Removed previous script 'Build'
Creating new 'Build' script for 'SomeModule ' version '1.28'
$ perl ./Build
$ perl ./Build test
... user tests run ...
```

To run the developer tests, either set the `PERL_RUN_ALL_TESTS` environment variable before running `perl ./Build test` or run the distribution tests with `perl ./Build disttests`.

Hacking the Hack

Could you do the same by checking for the presence of the environment variable in each test file? Absolutely—but consider that adding a new developer test is as easy as putting it

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

in the *t/developer/* directory without adding any extra magic to the test file. Also you get the `test` and `disttest` targets to behave correctly almost for free.

You can further customize the tests, running specific subsets for different test targets. `perl ./Build networktests` could try to connect to a server, for example. See the `Module::Build` documentation for more information.

The author particularly recommends that CPAN authors who want to raise their Kwalitee scores skip the POD tests for non-developers.

In theory, you can achieve the same effect with `ExtUtils::MakeMaker`. Yet every time the author spelunks into the module's documentation, or worse—code, he wakes up fully clothed and shivering in the shower several hours later.

Hack 63. Run Tests Automatically

See what's broken right after you break it!

The idea behind test-driven development is that rapid feedback based on comprehensive test coverage helps you write sane, clean code that never regresses. Once you know *how* to write tests, the next step is getting sufficient feedback as soon as possible.

What if you could run your tests immediately after you write a file? That's fast feedback that can help you see if you've broken anything. Best of all, you don't have to switch from your most beloved editor!

The Hack

The heart of this hack is the `onchange` program:

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
#!/usr/bin/perl

# onchange file1 file2 ... fileN command

use strict;
use warnings;

use File::Find;
use Digest::MD5;

my $Command     = pop @ARGV;
my $Files       = [@ARGV];
my $Last_digest = '';

sub has_changed
{
    my $files = shift;
    my $ctx   = Digest::MD5->new( );

    find( sub { $ctx->add( $File::Find::name, ( stat($_) )[9] ) },
        grep { -e $_ } @$files );

    my $digest      = $ctx->digest( );
    my $has_changed = $digest ne $Last_digest;
    $Last_digest    = $digest;

    return $has_changed;
}

while (1)
{
    system( $Command ) if has_changed( $Files );
    sleep 1;
}
```

This takes a list of files or directories to monitor and a command to run when they change. Of course, using `File::Find` means that this processes directories recursively.

Running the Hack

For a Perl application that uses the standard CPAN module structure (modules in *lib/*, tests in *t/*, and either a *Makefile.PL* or *Build.PL* to control everything), running is easy. Open a new terminal, change to your build directory, and run the command:

```
$ onchange Build.PL lib t 'clear; ./Build test'
```

Then every time either a test or a module file changes such that its MD5 signature changes, `onchange` will rebuild the project and run its tests again.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Hacking the Hack

MD5 signatures aren't the best way to tell if a file has changed. Even changing whitespace can make files look different (though sometimes it can be significant). You could use `PPI::Signature`, at least on pure-Perl code, to examine files and see if there are any syntactically significant changes.

You don't have to run *all* of the tests every time a module changes; it might be sufficient to run the user tests and the module-specific unit tests—but `onchange` needs to change to make this happen. If you had some way of correlating module files to their tests, you could even do this automatically.

Hack 64. See Test Failure Diagnostics — in Color!

Highlight the unexpected.

If you follow good development practices and write comprehensive unit tests for your programs, you'll be able to develop faster and more reliably. You'll also eventually run into the problem of too many successes hiding the failures—that is, if you keep your tests always succeeding, you only need to know about the tests that fail.

Why not make them stand out?

Perl's standard testing harness, `Test::Harness` is actually a nice wrapper around `Test::Harness::Straps`, which is a parser for the TAP format that standard tests follow. If and when the report that `Test::Harness` writes isn't sufficient, use `Test::Harness::Straps` to write your own.

The Hack

There are two barriers to this approach. First, the default behavior of Perl's standard testing tools is to write diagnostic output to STDERR. `Test::Harness` doesn't capture this. Second,

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

`Test::Harness` goes to a bit of trouble to set up the appropriate command line paths to run tests appropriately.

The first problem is tractable, at least if you can use a module such as `IPC::Open3` to capture STDERR and STDOUT. The second problem is a little trickier. The current version of `Test::Harness::Straps`, which ships with `Test::Harness` 2.48, doesn't *quite* provide everything publicly that you need to run tests well. Hopefully a future version will correct this, but for now, the hack is to use a private method, `_command_line( )`, to generate the appropriate command for running the test.

Adding color is very easy, at least on platforms where `Term::ANSIColor` works. The code can be as simple as:

```
#!/usr/bin/perl

use strict;
use warnings;

use IPC::Open3;
use Term::ANSIColor;
use Test::Harness::Straps;

my $strap = Test::Harness::Straps->new( );

for my $file (@ARGV)
{
    next unless -f $file;

    my $output;

    my $command = $strap->_command_line( $file );
    my $pid     = open3( undef, $output, $output, $command );
    my %results = $strap->analyze( $file, $output );

    print $_->{output} for @{ process_results( $file, \\%results ) };
}

sub process_results
{
    my ( $file, $results ) = @_;
    my $count              = 0;

    my @results;
    for my $test ( @{ $results->{details} } )
    {
        $count++;
        next if $test->{ok};

        push @results =>
        {
            test   => $test,
            output => create_test_result(
                $test->{ok}, $count, @{$test}{qw( name reason diagnostics )}
            )
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
            };
        }

        return \\@results;
    }

    sub create_test_result
    {
        my ( $ok, $number, $name, $reason, $diag ) = @_;

        $ok       = $ok ? 'ok' : 'not ok';
        $reason ||= '';
        $reason   = " ($reason)" if $reason;
        $diag   ||= '';

        return color( 'bold red' ) .
               sprintf "%6s %4d %s%s\\n%s\\n", $ok, $number, $name, $reason,
               color( 'clear yellow' ) . $diag . color( 'reset' );
    }
```

The code loops through every test file given on the command line, running it through `IPC::Open3::open3( )` to collect the output from both STDERR and STDOUT into `$output`. It uses `Test::Harness::Straps`'s `analyze( )` method to turn the TAP output into a data structure representing the tests, and then processes each result.

Passing tests are uninteresting; only failures with diagnostics are useful, so the `process_results( )` function filters out everything else. Test numbers and names print out in bold red and test diagnostics print in clear yellow.

Hacking the Hack

`Test::Harness::Straps` actually provides much more information, such as the number of total tests expected and actually run, as well as special information about TODO and SKIP tests. It's easy to provide a `Test::Harness`-style summary of each test file run as well as the total tests.

It's also possible to write a harness that collects output over a network or from other sources. *Perl Testing: A Developer's Notebook* (O'Reilly, 2005) has other examples and suggestions.

Hack 65. Test Live Code

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Verify your code against actual use...with little penalty.

Perl culture widely acknowledges automated testing as one step in verifying quality. Most CPAN modules and many large applications, not to mention Perl distributions themselves, have comprehensive test suites that run before installation to show what works and, occasionally, what doesn't work.

In theory, that's enough. In practice, it can be difficult to predict exactly how your code will react to production systems, live customers, and their actual data. Testing against this scenario would be incredibly valuable, but it's much more complex—not to mention probably much slower. If your automated tests are effective, they'll match the behavior of most customer requests.

Fortunately, it's possible to embed tests in production code that test against live data, non-destructively, but that don't generate too much data nor slow down your system.

The Hack

Imagine you have a web application that allows employees to manage their user data stored in a backend LDAP database. Abstraction is the key to a good system. You've created a `User` object and you've tested that the system vets and verifies all sorts of names and addresses that you could think of. You don't know how the system will react to the messy real world, though.

Instead of hard-coding the creation of the `User` object, create a factory object that returns a `User` object or equivalent.

```
package UserFactory;

use User;
use UserProxy;

my $count    = 0;

sub create
{
   my $self  = shift;
   my $class = $count++ % 100 ? 'User' : 'UserProxy';
   return $class->new( id => $count, @_ );
}

1;
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Every hundred requests, the factory will create a `UserProxy` object instead of a `User` object. As long as `UserProxy` implements the same interface as `User` (and actually behaves similarly), the rest of the code should see no difference. `UserProxy` is a bit more complex:

```
package UserProxy;

use strict;
use warnings;

use User;
use Test::Builder;

sub new
{
    my ($class, %args) = @_;
    my $proxied        = User->new( %args );
    my $Test           = Test::Builder->create( );
    $Test->output( time( ) . '_' . $proxied->id( ) . '.tap' );
    $Test->plan( 'no_plan' );
    bless { proxied => $proxied, test => $Test }, $class;
}

sub proxied
{
    my $self = shift;
    return $self->{proxied};
}

sub test
{
    my $self = shift;
    return $self->{test};
}

sub can
{
    my ($self, $method) = @_;
    my $proxied         = $self->proxied( );
    return $proxied->can( $method );
}

sub verify_name
{
    my ($self, $name)   = @_;
    my $proxied         = $self->proxied( ):
    my $test            = $self->test( );
    $test->ok( $proxied->verify_name( $name ), "verify_name( ) for '$name'" )
        || $test->diag( $proxied->verification_error( ) );
}

# ...

1;
```

When `UserFactory` creates a `UserProxy`, the proxy class creates an actual `User` object with the same arguments. It also wraps some calls to normal `User` methods with its own methods to run tests. The example `verify_name( )` method wraps the call to `User-`

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

`>verify_name( )` in a test case, expecting it to pass but logging it with a test diagnostic containing the error if it does not.

Apart from the factory/proxy design, the other part that makes this hack work is the use of `Test::Builder` in `UserProxy->new( )`. Each `UserProxy` object contains its own `Test::Builder` object and sends its output to a unique file with a name based on the current time and the number of the request. From there, use a cron job to run `prove` or some other `Test::Harness`-related program to analyze the tests and notify the proper people if things fail.

Be sure to use Test::Builder 0.60 or newer to have access to `create ( )`.

Hacking the Hack

The more data you can keep about failing requests, the better—you can use this data in your automated tests as you add test cases and fix the bugs.

`Test::Builder` provides only a few methods. You may want to write your own test library atop `Test::Builder` based on your needs. An alternate approach is to use `Test::Class` within your proxy. Though you need to find some way to manage the test output and counter on a per-object basis, the module will handle much of the setup code for you. It will also be very valuable if you want to test multiple types of objects that inherit from common ancestors.

Sampling one out of every hundred requests may be the wrong frequency. There's a whole field of statistical analysis devoted to sample and defect rates. This is a decent place to start, and it's likely an improvement over only automated testing, but it's not perfect for every need.

It may be worth serializing the proxied object and storing it somewhere useful in case of failure. Whether you use `Storable` or `YAML` or just save the relevant data somewhere else, having the exact information available to recreate the appropriate customer request will aid debugging.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Hack 66. Cheat on Benchmarks

Add optimizations where they really matter.

A well-known fact among programmers is that the true sign of superiority of a language is its performance executing various meaningless and artificial benchmarks. One such impractical benchmark is the Ackermann function, which really exercises an implementation's speed of recursion. It's easy to write the function but it's difficult for the computer to calculate and optimize.

Of course, benchmarks are rarely useful. Yet sometimes they can teach you about good optimization techniques.

If you love Perl, cheat. It's easy.

A fairly fast but maintainable Perl 5 implementation of this function is:

```
use strict;
use warnings;

no warnings 'recursion';

sub ackermann
{
    my ($m, $n) = @_;
    return            $n + 1      if $m = = 0;
    return ackermann( $m - 1, 1 ) if $n = = 0;
    return ackermann( $m - 1, ackermann( $m, $n - 1 ) );
}

print ackermann( 3, 10 ), "\\n";
```

Analyzing the function reveals that it takes a long, long time to calculate the value for any interesting positive integers. That's why the code disables the warning for deep recursion.

So, cheat. Add two lines of code to the program before calling `ackermann( )` with the seed values to speed it up substantially:

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
use Memoize;
memoize( 'ackermann' );
```

Don't run this with arguments greater than (4, 2) if you plan to use your computer before it finishes computing.

Calculating for (3, 10) with the memoized version took just under 1.4 seconds on the author's computer. The author interrupted the unmemoized version after a minute, then felt bad and restarted. It ran to completion in just over five minutes. These are not scientific results, but the difference in timing is dramatic.

Is this really cheating? A hypothetically complex Perl compiler could notice that `ackermann ( )` has no side effects and mathematically must return the same output for any two given inputs, so it could perform this optimization itself. You're just helping it along with a core module.

See the `Memoize` documentation for information on how this works and legitimate uses of memoization. See the http://en.wikipedia.org/wiki/Ackermann_function Wikipedia entry for more about the Ackermann function.

Hack 67. Build Your Own Perl

Compile Perl just the way you like it.

Perl has so many features that no single binary can do everything everyone wants. If you're debugging XS code, you might want to enable debugging. If you like to experiment, you might want to enable threads. If you need to run Perl on an odd platform where memory or disk space are low, you might want to disable certain features and core modules. You might even want an experimental patch that adds type information (`autobox` on the CPAN) or the defined-or operator (`dor` on the CPAN). You might also want to patch Perl yourself or help test out a development release.

Whatever the case, building your own Perl is reasonably easy.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

The Hack

Before you start, you need a working C development environment with a compiler, system headers, and a Make utility.

First, download Perl. The latest version is always available from the CPAN at http://www.cpan.org/src/. Stable versions have even minor version numbers (Perl 5.*6*.x, Perl 5.*8*.x, Perl 5.*10*.x) while development versions have odd minor numbers (Perl 5.*7*.x, Perl 5.*9*.x). Unless you are ready to report and possibly debug bugs, choose a stable version.

After you have downloaded and unpacked the distribution, change to the new directory. To configure the default build, simply run the *Configure* file:

```
$ sh Configure -de
```

You don't *have* to use the `-de` flag, but the configuration will prompt you for multitudinous options that few people care about and fewer still all understand. However, some options *are* useful.

To install Perl to a different root directory, use the `-Dprefix` option. For example, if you want to test Perl 5.9.3 and install it to */usr/local/bleadperl-5.9.3*, use the flag `-Dprefix=/usr/local/bleadperl-5.9.3`.

If you already have a system Perl installed, some of the OS utilities might rely on it. In this case, build and install a parallel Perl rather than overwrite an installed version.

To build a development release, pass the `-Dusedevel flag`.

To enable threads, use the `-Dusethreads` flag.

To enable debugging of the `perl` binary itself, pass the `-Doptimize='g'` flag.

To see the other configuration options, run:

```
$ sh Configure -h
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Running the Hack

Now you have Perl configured and can build it. Building is as simple as running your Make utility, usually `make` but sometimes `gmake`:

```
$ make
```

If everything goes well, you will have a `perl` binary in a few minutes, along with compiled versions of the necessary core libraries in *lib/*. Then, run the core tests:

```
$ make test
```

Everything *should* pass. If not, check the appropriate *README.** file for your platform to see if there are any expected failures.

If there are failures, or if you're building a development release, consider using the `perlbug` utility built with this Perl to report the failures:

```
$ ./perlbug -nok
```

Use the `-ok` flag instead to report success of a development release.

If you can't send mail from this system, send the output to a file with the `-f` switch:

```
$ ./perlbug -nok -f build.failure
```

Then mail the file to `perlbug@perl.org`.

If everything is to your expectation, install your new Perl with:

```
$ make install
```

Switch to the root user or use `sudo` as necessary, depending on your installation prefix.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Hacking the Hack

The *INSTALL* file in the source distribution contains the most information about building and installing Perl, as well as reporting any bugs to the Perl 5 Porters. If you run into trouble, read that file for answers. The numerous *README.** files contain platform-specific information that affects how Perl builds and runs (read them with `perldoc` or your favorite pager).

To hack on Perl itself, start by reading *pod/perlhack.pod* in the source directory. This also gives the `rsync` command used to access the very latest sources, as well as instructions on sending patches.

Hack 68. Run Test Suites Persistently

Speed up your tests.

Large Perl applications with many interconnected modules can take a long time to start up. Perl needs to load, compile, and initialize all of the modules before it can start running your application.

Tests for a large system can be particularly slow. A test suite typically contains lots of small short-lived scripts, each of which pulls in lots of module code at start up. A few seconds of delay per script can add up to a lot of time spent waiting for your test suite to finish.

The cure for long startup times within web-based applications is to run under a persistent environment such as mod_perl or `PersistentPerl`. `PersistentPerl` works for command-line programs as well. It's usually as simple as changing the shebang line from `#!/usr/bin/perl` to `#!/usr/bin/perperl`.

Running your test suite persistently is slightly more complicated, and doesn't work for every test, but the benefit is a huge speed increase for most of your tests. Running your test suite persistently can speed up your tests by a factor of five on a slow machine.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

The Hack

The first step of the hack is to make `Test::Builder`-based scripts compatible with PersistentPerl. There are several parts to this:

- The script has to reset the `Test::Builder` counter on startup.
- The script needs to prevent `Test::Builder` from duplicating `STDOUT` and `STDERR`, as this seems to be incompatible with `PersistentPerl`.
- Scripts with `no_plan` have to register a `PersistentPerl` cleanup handler to display the final `1..X` line.

`Test::PerPerlHelper` does all of this for you:

```
package Test::PerPerlHelper;

use strict;
use warnings;

use base 'Test::Builder';

require Test::More;

sub import
{
    my $class = shift;

    if (eval {require PersistentPerl} && PersistentPerl->i_am_perperl( ))
    {
        # rebless the Test::Builder singleton into our class
        # so that we can override the plan and _dup_stdhandles methods
        my $Test = Test::Builder->new( );
        bless $Test, __PACKAGE__;
    }

    $class->plan(@_);
}

sub plan
{
    my $class = shift;
    return unless @_;

    my $Test  = Test::Builder->new( );

    if (eval {require PersistentPerl} && PersistentPerl->i_am_perperl( ))
    {
        $Test->reset( );

        Test::Builder::_autoflush(*STDOUT);
        Test::Builder::_autoflush(*STDERR);

        $Test->output(*STDOUT);
        $Test->failure_output(*STDERR);
        $Test->todo_output(*STDOUT);
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
        $Test->no_ending(1);
        my $pp   = PersistentPerl->new( );
        $pp->register_cleanup(sub { $Test->_ending });
    }
    $Test->SUPER::plan(@_);
}

# Duplicating STDERR and STDOUT doesn't work under perperl
# so override it with a no-op
sub _dup_stdhandles {  }

1;
```

Under the hood, `Test::Builder` uses a singleton `$Test` object to maintain state. No matter how many times you call `Test::Builder->new( )`, it always returns a reference to the same `$Test` object. It does this so that all the various CPAN test modules can all share the same test state (especially the test counter).

`Test::PerPerlHelper` makes itself a subclass of `Test::Builder`, and then sneakily reblesses the `Test::Builder` singleton so that it is a `Test::PerPerlHelper` instead of a `Test::Builder`. In this way `Test::PerPerlHelper` can make itself compatible with all of the CPAN `Test::*` modules by customizing the singleton `$Test` object.

`Test::PerPerlHelper` only does this and other `PersistentPerl`-related compatibility tricks if the test script is running under `PersistentPerl`, so you can safely run the same test script normally as well.

The only change that you should have to make to your test scripts is make them end in a true value:

```
use Test::More 'no_plan';

ok(1);
ok(2);
ok(3);

1;  # persistent tests need to end in a true value!
```

Creating a wrapper around prove

Next you need to make all of your test scripts run in the same shared `PersistentPerl` interpreter.

Normally when you run a program under `PersistentPerl`, the Perl interpreter stays running in the background after your program terminates. The next time you run the program, `PersistentPerl` will reuse the same backend interpreter. Typically, each program gets its own private interpreter.

However, for test suites, this policy of one interpreter per program causes a problem. When you use `Test::Harness`'s `prove` program to run your tests, you don't want to make `prove` itself persistent; you want to make all of your test scripts persistent—and you want them all to share a single interpreter.

The first step is to create a wrapper script called `perperl-runscript` which you will use to run every test script:

```
#!/usr/bin/perperl -- -M1

use strict;
use Test::PerPerlHelper;

my $script;
while (my $arg = shift)
{
    # if the arg is a -I switch, add the directory to @INC
    # unless it already exists
    if ($arg =~ /^-I(.*)/ and -d $1)
    {
        unshift @INC, $1 unless grep { $_ eq $1 } @INC;
    }
    else
    {
        $script = $arg;
    }
}

do $script or die $@;
```

Place `perperl-runscript` somewhere in your `$PATH`.

Running the Hack

Set the `HARNESS_PERL` environment variable to `perperl-runscript` to cause `Test::Harness` to run every test through this script instead of through Perl. Because the name of this script never changes, `PersistentPerl` will always use the same backend interpreter to run every test. The `-M1` switch on the shebang line tells `perperl` to only spawn one backend interpreter.

You can set `HARNESS_PERL` on the same line as `prove`:

```
$ HARNESS_PERL=perperl-runscript prove -Ilib t/
```

Better still, create a wrapper script around `prove` called `perperl-prove`:

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
#!/bin/sh

export HARNESS_PERL=perperl-runscript

prove $*
```

Now you have the choice of running your test suite persistently or non-persistently:

```
$ perperl-prove -Ilib t/
$ prove -Ilib t/
```

To "restart" `PersistentPerl`, you must kill its backend processes:

```
$ killall perperl_backend
```

You have to restart `PersistentPerl` if any code outside of the test script itself has changed. However you *don't* have to restart `PersistentPerl` if only the test script has changed.

Hacking the Hack

There are some limitations with running tests persistently. In particular:

- Scripts that muck about with `STDIN`, `STDOUT`, or `STDERR` will have problems.
- The usual persistent environment caveats apply: be careful with redefined subs, global variables, and so on; `required` code only gets loaded on the first request, and so forth.
- Test scripts have to end in a true value.

Expect some scripts to cause problems. When you find a script that does not play nicely with `PersistentPerl`, you can configure the script to skip all its tests when run persistently:

```
use Test::More;
use Test::PerPerlHelper;
if (eval { require PersistentPerl } and PersistentPerl->i_am_perperl( ) )
{
    Test::PerPerlHelper->plan(
        'skip_all',
        'Redirecting STDIN doesn't work under perperl' );
}
else
{
    plan "no_plan";
}
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

You can also use the "Reload Modified Modules" [Hack #30] hack to reload modules without restarting `PersistentPerl`.

Add the following lines to your test script:

```
use Module::Reloader;
Module::Reloader::reload( ) if $ENV{'RELOAD_MODULES'};
```

Now you can reload modules by setting the `RELOAD_MODULES` environment variable to a true value:

```
$ RELOAD_MODULES=1 perperl-prove t/
```

If you don't set the environment variable, then the modules will not be reloaded:

```
$ perperl-prove t/
```

Note that for some reason `Module::Reloader` doesn't work on the first run of a script; it only starts working on the second run. The first run fails with an error, `Too late to run INIT block`.

Hack 69. Simulate Hostile Environments in Your Tests

Test devastating failures with aplomb.

When you publish a CPAN module that depends on other modules, you list the prerequisite modules in your *Makefile.PL* or *Build.PL* script.

Using *Build.PL*:

```
my $builder = Module::Build->new(
  # ... other Build.PL options ...
  requires =>
  {
      'Test::More'       => 0,
      'CGI'              => 2.0,
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
  }
);
```

Using *Makefile.PL*:

```
WriteMakefile(
    # ... other Makefile.PL options ...
    'PREREQ_PM' =>
    {
        'Test::More'     => 0,
        'CGI'            => 2.0,
    }
);
```

However, there are a few ways that this standard prerequisite checking can be insufficient. First, you may have optional prerequisites. For instance, your module will use `Foo::Bar` if it happens to be installed, but should fail gracefully when `Foo::Bar` is absent.

Second, if the behavior of a module changed between two versions, you may still want to support both versions. For example, `CGI` changed how it handles `PATH_INFO` in version 3.11. Your `CGI::Super_Path_Info` module probably wants to be compatible with both `CGI` version 3.11 and also with earlier (and later) versions.

Finally, occasionally a user will install your module by hand to bypass the prerequisite check, hoping to use an older version of `Foo::Bar` than the one you require. Sometimes your module works fine (maybe with some feature limitations), but your test suite breaks because your tests assumed the presence of a new feature.

For each of these cases. you can make your module and tests more robust. For example, you can skip tests that are incompatible with an older version of a module:

```
use Test::More;
use CGI;

if ($CGI->VERSION >= 3.11)
{
    plan skip_all => 'skipping compatibility tests for old CGI.pm';
}
else
{
    plan 'tests' => 17;
}
```

You can also skip tests that require the presence of a particular optional module:

```
eval 'require URI;';
if ($@)
{
    plan skip_all => 'optional module URI not installed';
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
}
else
{
    plan 'tests' => 10;
}
```

Now your tests are (hopefully) more robust, but how do you make sure that they will actually work on a system that is missing some modules and has older versions of others?

Ideally, you want to run your test suite against a few different sets of installed modules. Each set will be different from what you have installed in the main Perl `site_lib` of your development machine. It's way too much work to uninstall and reinstall ten different modules every time you make a new CPAN release.

The Hack

There are three possibilities: the user has an old version of the module installed, the user does not have the module installed, and the user has some combination of both for multiple modules.

Simulating old versions of modules

Create custom Perl library directories and include these directories when you use `prove` to run your tests. For instance, to run the tests against an old version of `CGI`:

```
$ mkdir t/prereq_lib
$ mkdir t/prereq_lib/CGI
$ cp CGI-3.10.pm t/prereq_lib/CGI.pm
$ prove -Ilib -It/prereq_lib t/
```

Including *t/prereq_lib* on the command line to `prove` puts at the start of `@INC`, so Perl will load any modules you put in this directory before modules installed in your system's Perl *lib* directories.

Simulating missing modules

That works for older versions of modules, but how do you install the *absence* of a module in a custom library directory so that it takes precedence over a copy already installed on your system?

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

The solution is to create a zero-length file with the same name as the module. This works because in order for a module to load successfully (via `require` or `use`) it has to end in a true value, such as (from actual `CPAN` modules):[1]

[1] The more boring the line of code, the better the opportunity for creativity.

```
1;
666;
"false";
"Steve Peters, Master Of True Value Finding, was here.";
```

A zero-length file doesn't end in a true value, and consequently `require` fails. It doesn't fail with the same error message as a missing module fails with, but it still fails.

For example, to run the tests in an environment missing `URI`:

```
$ mkdir -p t/skip_lib
$ touch t/skip_lib/URI.pm
$ prove -Ilib -It/skip_lib t/
```

Running multiple scenarios

You can create multiple different library directories, each containing a different combination of missing and/or old modules:

```
$ mkdir -p t/prereq_scenarios/missing_uri
$ touch t/prereq_scenarios/missing_uri/URI.pm
$ mkdir -p t/prereq_scenarios/old_cgi
$ cp CGI-3.10.pm t/prereq_scenarios/old_cgi/CGI.pm
$ mkdir -p t/prereq_scenarios/new_cgi
$ cp CGI-3.15.pm t/prereq_scenarios/new_cgi/CGI.pm
```

Then run all of these scenarios at once:

```
$ for lib in t/prereq_scenarios/*; do prove -Ilib -I$lib t/; done
```

However this one-liner stops at the first error and doesn't provide any summary information. Here's a more complete version:

```
#!/usr/bin/perl

use strict;
use File::Find;
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
if (@ARGV < 2)
{
    die "Usage: $0 [prereq_scenarios_dir] [args to prove]\\n";
}

my $scenarios_dir = shift;

my %scenario_modules;
my $errors;

my @scenarios     = grep { -d } <$scenarios_dir/*>;

for my $lib_dir (@scenarios)
{
    unless (-d $lib_dir)
    {
        $errors   = 1;
        warn "lib dir does not exist: $lib_dir\\n";
        next;
    }
    my @modules;

    find(sub
    {
        return unless -f;

        my $dir =  "$File::Find::dir/$_";
        $dir    =~ s/^\\Q$lib_dir\\E//;
        $dir    =~ s/\\.pm$//;
        $dir    =~ s{^/}{ };
        $dir    =~ s{/}{::}g;

        push @modules, $dir;
    }, $lib_dir);

    $scenario_modules{$lib_dir} = \\@modules;
}

die "Terminating." if $errors;

for my $lib_dir (@scenarios)
{
    my $modules   = join ', ', sort @{ $scenario_modules{$lib_dir} };
    $modules    ||= 'none';
    print "\\n" . '#' x 62 . "\\n";
    print "Running tests.  Old (or absent) modules in this scenario:\\n";
    print "$modules\\n";

    my @prove_command = ('prove', "-I$lib_dir", @ARGV);

    system( @prove_command ) && do
    {
        die <<EOF;
##############################################################
One or more tests failed in scenario $lib_dir.
The old or absent modules were:
    $modules

The command was:
    @prove_command
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
Terminating.
############################################################
EOF
    };
}
```

Save this as *prove-prereqs* and run it as:

```
$ prove-prereqs t/prereq_scenarios -Ilib t/
```

Hacking the Hack

PITA, the Perl Image Testing Architecture (http://search.cpan.org/dist/PITA) project, goes much further than this hack does. PITA will be able to test your Perl modules under different versions of Perl and even on different operating systems (possibly running within virtual machines on a single computer). It will allow you to automate the testing process and collect the results generated from several testing environments.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006