Test::Class Best Practices

xUnit style testing for Perl

by

Curtis "Ovid" Poe

Test::Class Best Practices

xUnit style testing for Perl

by

Curtis "Ovid" Poe

ABSTRACT

When working with large test suites, using procedural tests for object-oriented code becomes clumsy after a while. This is where `Test::Class` really shines. Unfortunately, many programmers struggle to learn this module or don't utilize its full power.

Please note that article assumes a basic familiarity with object-oriented Perl and testing. Also, some of these classes are not "proper" by the standards of many OO programmers (your author included), but have been written for clarity rather than purity.

MODULES AND THEIR VERSIONS

This article was written with the following modules and versions:

- `Test::Class` version 0.31
- `Test::Most` version 0.21
- `Test::Harness` version 3.15
- `Moose` version 0.7
- `Class::Data::Inheritable` version 0.08

You may use lower versions of these modules (and write the OO by hand instead of using `Moose`), but be aware that you may see slightly different behavior.

Notes about the code

Note that `Moose` packages should *generally* end with this:

```
__PACKAGE__->meta->make_immutable;
no Moose;
```

We're omitting this from our examples. We're also omitting `use strict` and `use warnings`, but assume they are there (they're automatically used when you `use Moose`). The code will, however, run just fine without this. We do this merely to focus on the core features of the code in question.

Of course, you may need to adjust the shebang line (`#!/usr/bin/env perl -T`) for your system.

EVOLUTION OF A PERL PROGRAMMER

There are many paths programmers take in their development, but a typical one seems to be this:

- Start writing simple procedural programs.
- Start writing modules when they find they need to reuse code.
- Start using objects when they need more powerful abstractions.
- Start writing tests.

While it would be nice if people started writing tests from day 1, the reality is most programmers don't. But when they do, what do those tests look like? Well, they're often straight-forward procedural tests like this:

```
#!/usr/bin/env perl -T

use strict;
use warnings;

use Test::More tests => 3;

use_ok 'List::Util', 'sum' or die;

ok defined &sum, 'sum() should be exported to our namespace';
is sum(1,2,3), 6, '... and it should sum lists correctly';
```

Now there's nothing wrong with procedural tests and they're great for non-OO code. For most projects, they handle everything you need to do and if you download most modules off the CPAN you'll generally find their tests -- if they have them -- procedural in style. However, when you start to work with larger code bases, merely have a `t/` directory with 317 test scripts starts to get a bit tedious. Where is the test you need? Trying to memorize all of your test names and grepping through your tests to find out which ones test the code you're working with becomes tedious. That's where Adrian Howard's `Test::Class`, can help.

USING TEST::CLASS

Creating a simple test class

Now let's start digging into `Test::Class`. I'm a huge "dive right in" fan, so we'll now skip a lot of the theory and just see how things work. Though I often use test-driven development (TDD), I'll reverse the process here so you can see explicitly what we're testing. Also, `Test::Class` has quite a number of different features, not all of which I'm going to explain here. See the documentation for more information.

First, we'll create a very simple `Person` class. Because I don't like writing out simple methods over and over, we'll use `Moose` to automate a lot of the grunt work for us.

```
package Person;

use Moose;

has first_name => ( is => 'rw', isa => 'Str' );
has last_name  => ( is => 'rw', isa => 'Str' );

sub full_name {
    my $self = shift;
    return $self->first_name . ' ' . $self->last_name;
}

1;
```

This gives us the constructor and `first_name`, `last_name` and `full_name` methods.

Now let's write a simple `Test::Class` program for it. In order to do this, we need a place to put the tests. Further, to avoid namespace collisions, we need should choose our package name carefully. I like prepending my test classes with `MyTest::` to ensure that we have no ambiguity. In this case, I'll put my `Test::Class` tests in `t/tests/` and our first class will be named `MyTest::Person`. We'll assume the following directory structure:

```
lib/
lib/Person.pm
t/
t/tests/
t/tests/MyTest
t/tests/MyTest/Person.pm
```

Tip: though it might seem nice to put your tests in a `Test::` namespace, don't do that. You might accidentally clash with a testing module on the CPAN.

And the actual test class might start out looking like this:

```
package MyTest::Person;

use Test::Most;
use parent 'Test::Class';

sub class { 'Person' }

sub startup : Tests(startup => 1) {
    my $test = shift;
    use_ok $test->class;
}

sub constructor : Tests(3) {
    my $test  = shift;
    my $class = $test->class;
    can_ok $class, 'new';
    ok my $person = $class->new,
        '... and the constructor should succeed';
    isa_ok $person, $class, '... and the object it returns';
}

1;
```

Note: we’re using `Test::Most` instead of `Test::More`. We’ll be taking advantage of `Test::Most` features later. Also, those methods should really be ’ro’ (read-only) because now we can leave the object in an inconsistent state. This is part of what I meant about "proper" OO code, but again, this is written for illustration purposes only.

Before we get into what all of that means, let’s jump ahead and run this. To do that, in our `t/` directory, include the following program as `run.t`.

```
#!/usr/bin/env perl -T

use lib 't/tests';
use MyTest::Person;

Test::Class->runtests;
```

This little program sets the path to our test classes, loads them and runs the tests. Now you can run that with the `prove` utility:

```
prove -lv --merge t/run.t
```

`Tip:` The `--merge` tells `prove` to merge `STDOUT` and `STDERR`. This avoids synchronization problems that happen when `STDERR` is not always output in synchronization with `STDOUT`. It’s recommended that you do *not* use this unless you’re running your tests in verbose mode. This is because failure diagnostic will then be sent to `STDOUT` and `TAP::Harness` discards `STDOUT` lines beginning with ’#’ if not running in verbose mode.

And we get the output similar to the following:

```
t/run.t ..
1..4
ok 1 - use Person;
#
# MyTest::Person->constructor
ok 2 - Person->can('new')
ok 3 - ... and the constructor should succeed
ok 4 - ... and the object it returns isa Person
ok
All tests successful.
Files=1, Tests=4,  0 wallclock secs
Result: PASS
```

(For layout reasons, detailed timing information after "wallclock secs" is omitted).

You'll note that the test output (named the "Test Anything Protocol", or "TAP", if you're curious) for the `constructor` method begins with the following diagnostic line:

```
# MyTest::Person->constructor
```

That occurs before every test method's output and makes it very easy to find which tests failed.

Now let's take a closer look at our tests and see what's going on.

```
package MyTest::Person;

use Test::Most;
use parent 'Test::Class';

sub class { 'Person' }

sub startup : Tests(startup => 1) {
    my $test = shift;
    use_ok $test->class;
}

sub constructor : Tests(3) {
```

```
14:     my $test  = shift;
15:     my $class = $test->class;
16:     can_ok $class, 'new';
17:     ok my $person = $class->new,
18:         '... and the constructor should succeed';
19:     isa_ok $person, $class, '... and the object it returns';
20: }
21:
22: 1;
```

Lines 1 through 4 are fairly straightforward. Line 4 has us inheriting from `Test::Class` and that's what makes all of this work. Line 6 defines a `class` method which our tests will use to know which class they're testing. It's very important to do this rather than hard-coding the class name in our test methods. That's good OO practice in general and later we'll see how this helps us.

The `startup` method has an attribute, 'Tests', which has the arguments `startup` and `1`. Any method labeled as a `startup` method will run once before any of the other methods run. The `1` (one) in the attribute says "we're also going to run one test in this method". If you don't run any tests in your `startup` method, omit this number:

```
sub load_db : Tests(startup) {
    my $test = shift;
    $test->_create_database;
}

sub _create_database {
    ...
}
```

Tip: as you can see from the code above, you don't need to name the `startup` method `startup`. I recommend you give it the same name as the attribute for reasons discussed later.

That will be run once and only once for each test class. Because the `_create_database` method does not have any attributes, you may safely call it and `Test::Class` will not try to run it as a test.

Of course, there's a corresponding `shutdown` available:

```
sub shutdown_db : Tests(shutdown) {
    my $test = shift;
    $test->_shutdown_database;
}
```

This allows you to set up and tear down a pristine testing environment for every test class without worrying that other test classes will interfere with the current tests. Of course, this means that tests may not be able to run in parallel and there are ways around that, but it's beyond the scope of this article.

As mentioned, our `startup` method has a second argument which tells `Test::Class` that we're going to run one test in this `startup` method. This is strictly optional. Here we use it to safely test that we can load our `Person` class. As an added feature, if `Test::Class` detects that the `startup` test failed or an exception is thrown, it assumes that there's no point in running the rest of the tests, so it skips the remaining tests for the class.

Tip: Don't run tests in your startup method. We'll explain why in a bit. For now, it's better to do this:

```
sub startup : Tests(startup) {
    my $test  = shift;
    my $class = $test->class;
    eval “use $class”;
    die $@ if $@;
}
```

However, we'll keep the test in the `startup` method for a while longer, just so you can see how it works.

Now let's take a closer look at the `constructor` method.

```
sub constructor : Tests(3) {
    my $test  = shift;
    my $class = $test->class;
    can_ok $class, ’new’;
    ok my $person = $class->new,
        ’... and the constructor should succeed’;
    isa_ok $person, $class, ’... and the object it returns’;
}
```

Tip: We did not name the constructor tests `new` because that's a `Test::Class` method and overriding it will cause our tests to break.

Our `Tests` attribute lists the number of tests as '3', but if we don't know how many tests we're going to have, we can still use `no_plan`.

```
    sub constructor : Tests(no_plan) { ... }
```

As a short-cut, omitting arguments to the attribute will also mean `no_plan`:

```
    sub constructor : Tests { ... }
```

The `my $test = shift` line is equivalent to `my $self = shift`. I've like to rename `$self` to `$test` in my test classes, but that's merely a matter of personal preference.

Also, the `$test` object is an empty hashref. This allows you to stash data there, if needed. For example:

```
    sub startup : Tests(startup) {
        my $test = shift;
        my $pid  = $test->_start_process or die "Could not start process: $?";
        $test->{pid} = $pid;
    }

    sub run : Tests(no_plan) {
        my $test = shift;
        my $process = $test->_get_process($test->{pid});
        ...
    }
```

The rest of the test method is self-explanatory if you're familiar with `Test::More`.

Of course, we also had `first_name`, `last_name` and `full_name`, so let's write those tests. Because we're in "development mode", we'll leave these tests as `no_plan`, but don't forget to set the number of tests when you're done.

```
sub first_name : Tests {
    my $test   = shift;
    my $person = $test->class->new;
    can_ok $person, 'first_name';
    ok !defined $person->first_name,
      '... and first_name should start out undefined';
    $person->first_name('John');
    is $person->first_name, 'John',
      '... and setting its value should succeed';
}

sub last_name : Tests {
    my $test   = shift;
    my $person = $test->class->new;
    can_ok $person, 'last_name';
    ok !defined $person->last_name,
      '... and last_name should start out undefined';
    $person->last_name('Public');
    is $person->last_name, 'Public',
      '... and setting its value should succeed';
}

sub full_name : Tests {
    my $test   = shift;
    my $person = $test->class->new;
    can_ok $person, 'full_name';
    ok !defined $person->full_name,
      '... and full_name should start out undefined';
    $person->first_name('John');
    $person->last_name('Public');
    is $person->full_name, 'John Public',
      '... and setting its value should succeed';
}
```

Tip: when possible, name your test methods after the method they're testing. This makes finding them much easier. You can even write editor tools to automatically jump to them. Of course, not all test methods will fit this pattern, but many will.

The `first_name` and `last_name` tests can probably have common elements factored out, but for now they're fine. Let's see what happens when we run this (warnings omitted):

```
t/run.t ..
ok 1 - use Person;
#
# MyTest::Person->constructor
ok 2 - Person->can('new')
ok 3 - ... and the constructor should succeed
ok 4 - ... and the object it returns isa Person
#
# MyTest::Person->first_name
ok 5 - Person->can('first_name')
ok 6 - ... and first_name should start out undefined
ok 7 - ... and setting its value should succeed
#
# MyTest::Person->full_name
ok 8 - Person->can('full_name')
not ok 9 - ... and full_name should start out undefined

#   Failed test '... and full_name should start out undefined'
#   at t/tests/Test/Person.pm line 48.
#   (in MyTest::Person->full_name)
ok 10 - ... and setting its value should succeed
#
# MyTest::Person->last_name
ok 11 - Person->can('last_name')
ok 12 - ... and last_name should start out undefined
ok 13 - ... and setting its value should succeed
1..13
# Looks like you failed 1 test of 13.
Dubious, test returned 1 (wstat 256, 0x100)
Failed 1/13 subtests

Test Summary Report
-------------------
t/run.t (Wstat: 256 Tests: 13 Failed: 1)
  Failed test:  9
  Non-zero exit status: 1
Files=1, Tests=13,  0 wallclock secs
Result: FAIL
```

Uh oh. We can see that `full_name` isn't behaving the way we expected it to. Let's assume that we want to `croak` if either the first or last name is not set. To keep this simple, we'll just assume that neither `first_name` nor `last_name` may be set to a false value.

```
sub full_name {
    my $self = shift;

    unless ( $self->first_name && $self->last_name ) {
        Carp::croak("Both first and last names must be set");
    }

    return $self->first_name . ' ' . $self->last_name;
}
```

That should be pretty clear, now let's look at the new test. We'll use the `throws_ok` test from `Test::Exception` to test the `Carp::croak()`. Because we're using `Test::Most` instead of `Test::More`, we can use this test function without specifically using `Test::Exception`.

```
sub full_name : Tests(no_plan) {
    my $test   = shift;
    my $person = $test->class->new;
    can_ok $person, 'full_name';

    throws_ok { $person->full_name }
        qr/^Both first and last names must be set/,
        '... and full_name() should croak() if the either name is not set';

    $person->first_name('John');

    throws_ok { $person->full_name }
        qr/^Both first and last names must be set/,
        '... and full_name() should croak() if the either name is not set';

    $person->last_name('Public');
    is $person->full_name, 'John Public',
      '... and setting its value should succeed';
}
```

And now our tests all pass and we can go back and set our test plan numbers, if desired:

```
All tests successful.
Files=1, Tests=14,  0 wallclock secs
Result: PASS
```

Inheriting tests

By now you're probably looking at that and saying "that's a heck of a lot of work just for testing a class" and if this was all there is to it, you'd be perfectly justified in forgetting about `Test::Class`. However, let's see how `Test::Class` really shines by writing a subclass of `Person` named `Person::Employee`. We'll keep it simple by only providing an `employee_number` method, but you'll quickly understand the benefits.

```
package Person::Employee;

use Moose;
extends 'Person';

has employee_number => ( is => 'rw', isa => 'Int' );

1;
```

And the test class for it:

```
package MyTest::Person::Employee;

use Test::Most;
use parent 'MyTest::Person';

sub class {'Person::Employee'}

sub employee_number : Tests(3) {
    my $test = shift;
    my $employee = $test->class->new;
    can_ok $employee, 'employee_number';
    ok !defined $employee->employee_number,
        '... and employee_number should not start out defined';
    $employee->employee_number(4);
    is $employee->employee_number, 4,
        '... but we should be able to set its value';
}

1;
```

Notice that instead of inheriting from `Test::Class`, we've inherited from `MyTest::Person`, just like out `Person::Employee` class inherited from `Person`. Also, we have overridden the `class` method to ensure that tests know which class they're using.

At this time, we also need to add `MyTest::Person::Employee` to `t/run.t`:

```
    #!/usr/bin/env perl -T

    use lib 't/tests';

    use MyTest::Person;
    use MyTest::Person::Employee;

    Test::Class->runtests;
```

And when we run it `t/run.t`:

```
    All tests successful.
    Files=1, Tests=31,  1 wallclock secs
```

Whoa! Wait a minute. We only added three tests. We started with 14, how come we now have 31?

Because `MyTest::Person::Employee` *inherited* the tests from `MyTest::Person`. That means that the 14 original tests plus the 14 inherited tests and the 3 added tests give us 31 tests! But these aren't frivolous tests, either. Look at the new test's output:

```
# MyTest::Person::Employee->constructor
ok 16 - Person::Employee->can('new')
ok 17 - ... and the constructor should succeed
ok 18 - ... and the object it returns isa Person::Employee
#
# MyTest::Person::Employee->employee_number
ok 19 - Person::Employee->can('employee_number')
ok 20 - ... and employee_number should not start out defined
ok 21 - ... but we should be able to set its value
#
# MyTest::Person::Employee->first_name
ok 22 - Person::Employee->can('first_name')
ok 23 - ... and first_name should start out undefined
ok 24 - ... and setting its value should succeed
#
# MyTest::Person::Employee->full_name
ok 25 - Person::Employee->can('full_name')
ok 26 - ... and full_name() should croak() if the either name is not set
ok 27 - ... and full_name() should croak() if the either name is not set
ok 28 - ... and setting its value should succeed
#
# MyTest::Person::Employee->last_name
ok 29 - Person::Employee->can('last_name')
ok 30 - ... and last_name should start out undefined
ok 31 - ... and setting its value should succeed
```

Because we didn't explicitly hard-code the class name in our tests and because `MyTest::Person::Employee` had overridden the `class` method, these new tests are being run against instances of `Person::Employee`, not `Person`. This allows us to know that we did not break any of our inherited behavior! However, if we do need to alter the behavior of one of those methods, as we might expect with object-oriented code, all you need to do is override the corresponding test method. For example, what if employees must have their full names listed in the format "last name, first name"?

```
sub full_name {
    my $self = shift;

    unless ( $self->first_name && $self->last_name ) {
        Carp::croak("Both first and last names must be set");
    }
```

```
        return $self->last_name . ', ' . $self->first_name;
    }
```

The appropriate test method in `MyTest::Person::Employee` might look like this:

```
    sub full_name : Tests(no_plan) {
        my $test   = shift;
        my $person = $test->class->new;
        can_ok $person, 'full_name';

        throws_ok { $person->full_name }
        qr/^Both first and last names must be set/,
          '... and full_name() should croak() if the either name is not set';

        $person->first_name('John');

        throws_ok { $person->full_name }
        qr/^Both first and last names must be set/,
          '... and full_name() should croak() if the either name is not set';

        $person->last_name('Public');
        is $person->full_name, 'Public, John',
          '... and setting its value should succeed';
    }
```

Make those changes and all tests will pass. `MyTest::Person::Employee` will call its own `full_name` test method and not that of its parent class.

Refactoring test classes

Refactoring with methods

There's a lot of duplication in the `full_name` test which you should factor out into common code. In our `MyTest::Person` class, one way to do this might be:

```
sub full_name : Tests(no_plan)
    my $test   = shift;
    $test->_full_name_validation;
    my $person = $test->class->new(
        first_name => 'John',
        last_name  => 'Public',
    );
    is $person->full_name, 'John Public',
      'The name of a person should render correctly';
}

sub _full_name_validation {
    my ( $test, $person ) = @_;
    my $person = $test->class->new;
    can_ok $person, 'full_name';

    throws_ok { $person->full_name }
        qr/^Both first and last names must be set/,
        '... and full_name() should croak() if the either name is not set';

    $person->first_name('John');

    throws_ok { $person->full_name }
        qr/^Both first and last names must be set/,
        '... and full_name() should croak() if the either name is not set';
}
```

And in `MyTest::Person::Employee`:

```
sub full_name : Tests(no_plan)
    my $test   = shift;
    $test->_full_name_validation;
    my $person = $test->class->new(
        first_name => 'Mary',
        last_name  => 'Jones',
    );
    is $person->full_name, 'Jones, Mary',
      'The employee name should render correctly';
}
```

Just like with any other OO code, we inherit the `_full_name_validation` method and can share it with our subclass.

Refactoring with fixtures

When writing test classes, the `startup` and `shutdown` methods are very handy, but those run only at the beginning and end of your test class. Sometimes you need code to run before the beginning and end of every test method. For example, in our code above, many of the test methods had the following line of code:

```
    my $person = $test->class->new;
```

Now you really may not want to duplicate that every time, so you can use what's known as a *fixture*. A fixture is "fixed state" for you tests to run against. These allow you to remove a lot of duplicated code from your tests and to have a controlled environment. You could do something like this:

```
    sub setup : Tests(setup) {
        my $test    = shift;
        my $class   = $test->class;
        $test->{person} = $class->new;
    }
```

Or if you want to start with a known set of data:

```
    sub setup : Tests(setup) {
        my $test    = shift;
        my $class   = $test->class;
        $test->{person} = $class->new(
            first_name => 'John',
            last_name  => 'Public',
        );
    }
```

Now, all of your test methods can simply use `$test->{person}` (you can make that a method if you prefer) to access a new instance of the class you're testing without having to constantly duplicate that code.

Now, all of your test methods can simply use `$test->{person}` (you can make that a method if you prefer) to access a new instance of the class you're testing without having

to constantly duplicate that code.

The corresponding `teardown` method is useful if you need to clean up on a per test basis. We'll cover more of these methods later.

MAKING OUR TESTING LIVES EASIER

Auto-discovering your test classes

By this time, you're probably beginning to understand how `Test::Class` can make managing large codebases a bit easier, but what about making `Test::Class` tests easier? The first problem is our helper script, `t/run.t`:

```
#!/usr/bin/env perl -T

use lib 't/tests';

use MyTest::Person;
use MyTest::Person::Employee;

Test::Class->runtests;
```

Right now, this doesn't look so bad, but as we start to add more classes, this gets to be unwieldy. What if you forget to add a test class? Your class might be broken, but since the test class is not run, how will you know? So let's fix this to 'auto-discover' our tests.

```
#!/usr/bin/env perl -T

use Test::Class::Load qw(t/tests);
Test::Class->runtests;
```

Just tell `Test::Class::Load` (bundled with `Test::Class`) which directories your test classes are in and it will find them for you. It does this by loading attempting to load all files with a `.pm` extension, so if you have "helper" test modules which are not `Test::Class` tests, keep them in a separate directory.

Using a common base class

Naturally, because this is programming, we want to be able to factor out common code. We've done a little bit of this already, but there's room for improvement. You'll notice that both test classes have a method for returning the name of the class being tested. But since we can *calculate* the name of this class, so why not push this into a base class? We'll put this in `t/tests/My/Test/Class.pm`.

```
package My::Test::Class;

use Test::Most;
use parent qw(Test::Class Class::Data::Inheritable);

BEGIN {
    __PACKAGE__->mk_classdata('class');
}

sub startup : Tests( startup => 1 ) {
    my $test = shift;
    ( my $class = ref $test ) =~ s/^MyTest:://;
    return ok 1, "$class loaded" if $class eq __PACKAGE__;
    use_ok $class or die;
    $test->class($class);
}

1;
```

For `Person::Employee`, we merely need to delete the `class` method. For `Person`, we delete the `class` method, delete the `startup` method and have it inherit from `My::Test::Class` instead of `Test::Class`. Now. `class` will always return the current class we're testing and it's guaranteed to be loaded by the time the test class has run. Here's what the new `MyTest::Person` class looks like:

```
package MyTest::Person;

use Test::Most;
use parent 'My::Test::Class';

sub constructor : Tests(3) {
    my $test   = shift;
```

```
        my $class = $test->class;
        can_ok $class, 'new';
        ok my $person = $class->new, '... and the constructor should succeed';
        isa_ok $person, $class, '... and the object it returns';
    }

    sub first_name : Tests(3) {
        my $test   = shift;
        my $person = $test->class->new;
        can_ok $person, 'first_name';
        ok !defined $person->first_name,
          '... and first_name should start out undefined';
        $person->first_name('John');
        is $person->first_name, 'John', '... and setting its value should succeed';
    }

    sub last_name : Tests(3) {
        my $test   = shift;
        my $person = $test->class->new;
        can_ok $person, 'last_name';
        ok !defined $person->last_name,
          '... and last_name should start out undefined';
        $person->last_name('Public');
        is $person->last_name, 'Public', '... and setting its value should succeed';
    }

    sub full_name : Tests(4) {
        my $test   = shift;
        $test->_full_name_validation;
        my $person = $test->class->new(
            first_name => 'John',
            last_name  => 'Public',
        );
        is $person->full_name, 'John Public',
          '... and setting its value should succeed';
    }

    sub _full_name_validation {
        my ( $test, $person ) = @_;
        my $person = $test->class->new;
        can_ok $person, 'full_name';

        throws_ok { $person->full_name }
            qr/^Both first and last names must be set/,
            '... and full_name() should croak() if the either name is not set';

        $person->first_name('John');

        throws_ok { $person->full_name }
            qr/^Both first and last names must be set/,
            '... and full_name() should croak() if the either name is not set';
    }

    1;
```

And the test results for `MyTest::Person::Employee`:

```
    All tests successful.
    Files=1, Tests=32,  1 wallclock secs
```

Now we have an extra test, but that's because of the `ok 1` found in the `My::Test::Class::startup` method. It gets called an extra time for the loading of `My::Test::Class`.

Tip: If your class must be loaded at `BEGIN` time, override this `startup` method in your test class but be sure to provide a `class` method.

Run individual test classes

When I'm running tests, I hate to leave my editor merely to run tests from the command line. To avoid this, I have something similar to following mapping in my `.vimrc` file:

```
     noremap ,t :!prove --merge -lv %<CR>
```

Then, when I'm writing tests, I merely hit `,t` and my test runs. However, doing this in a test class doesn't work. The class gets loaded, but the tests do not run. I could simply add a new mapping:

```
     noremap ,T  :!prove -lv --merge t/run.t<CR>
```

The problem is that this runs *all* of my test classes. If I have several hundred tests, I don't want to hunt back through all of the test output to see which tests failed. Instead, I want to run a single test class. To do this, I first alter my mapping to include the path to my test classes.

```
noremap ,t  :!prove -lv --merge -It/tests %<CR>
```

I also remove the `Test::Class->runtests` line from `t/run.t` (or else I'll have my tests run twice if I run the full test suite). Instead, now that I have a common base class, I add the following line to `My::Test::Class`:

```
INIT { Test::Class->runtests }
```

Now, regardless of whether or not I'm in a standard `Test::Most` test program or one of my new test classes, I can type `,t` and run just the tests in the file I'm editing.

If you run the tests for `MyTest::Person::Employee`, you'll see the full run of 32 tests because `Test::Class` will run the tests for the current class and all classes which it inherits from. If you run the tests for `MyTest::Person`, you'll only see 15 tests run, which is the behavior we wanted.

If you prefer Emacs, you can put the following in your `~/.emacs` file.

```
(eval-after-load "cperl-mode"
    '(add-hook 'cperl-mode-hook
        (lambda () (local-set-key "\C-ct" 'cperl-prove))))

(defun cperl-prove ()
    "Run the current test."
    (interactive)
    (shell-command (concat "prove -lv --merge -It/tests "
        (shell-quote_argument (buffer-file-name)))))
```

That will bind this to `C-c t` and you can pretend that you're as cool as `vim` users (just kidding! Stop the hate mail already).

HANDLING STARTUP/SETUP/TEARDOWN/SHUTDOWN METHODS

We often find that when we're running tests, we often need to have special code run at the start and end of a class and at the start and end of every test method. These might be useful to connecting to databases, deleting temp files, setting up test fixtures and so on. `Test::Class` can help us with this.

For simplicity's sake, we'll refer to `Test::Class`'s methods for handling this as *test control* methods.

`Test::Class` provides four such methods.

- `startup` This method is run *once* for each class, before any tests are run.
- `shutdown` This method is run *once* for each class, after all tests have run.
- `setup` This method is run before each test method.
- `teardown` This method is run after each test method.

"startup" and "shutdown"

One common function for the `startup` and `shutdown` methods is to set up and tear down a database:

```
package Tests::My::Resultset::Customer;
use parent 'My::Test::Class';

sub startup : Tests(startup) {
    my $test = shift;
    $test->_connect_to_database;
}

sub shutdown : Tests(shutdown) {
    my $test = shift;
    $test->_disconnect_from_database;
}
```

```
    # ... and so on
```

What happens here is that when the test class is loaded, the first code which gets run is `startup`. At the end of the test, the `shutdown` method is called and we disconnect from the database. Note that if the `startup` method has any tests and one fails, or if it throws any exception, the rest of the tests will not be run, but any tests for parent classes will still be run.

```
    sub startup : Tests(startup) {
        ok 0;    # the test class will abort here
    }
```

If this occurs, the `shutdown` method will *not* be called.

"setup" and "teardown"

Of course, we also might need to run code before and after every test method. Here's how to do that:

```
    sub setup : Tests(setup) {
        my $test = shift;
        $test->_start_db_transaction;
    }

    sub check_priviledges : Tests(no_plan) {
        my $test = shift;
        $test->_load_priviledge_fixture;
        ...
    }

    sub teardown : Tests(teardown) {
        my $test = shift;
        $test->_rollback_db_transaction;
    }
```

The above code let's us start a database transaction before every test method. The `check_priviledges` method loads its own test fixture and the `teardown` method rolls back the transaction, ensuring that the next test will have a pristine database. Note that if the `setup` method fails a test, the `teardown` method will still be called. This is different behavior for the `startup` method because `Test::Class` moves on to the next test and assumes you still want to continue.

Overriding test control methods

Two common problems which occur with users new to `Test::Class` is that they either find that they're running more test control methods than they expected or their test control methods are running in an order they did not expect. For example, let's say we have this in our test base class:

Controlling order of execution

```
sub connect_to_db : Tests(startup) {
    my $test = shift;
    $test->_connect_to_db;
}
```

And in a test subclass:

```
sub assert_db : Tests(startup => 1) {
    my $test = shift;
    ok $test->_is_connected_to_db,
      'We still have a database connection';
}
```

That will probably fail and your tests will not be run. Why? Because `Test::Class` runs tests in alphabetical order in a test class. Because it includes *inherited* tests in your test class, you've inherited `connect_to_db`, but since that sorts *after* `assert_db`, it gets run after it. Thus, you're asserting your database connection *before* you've connected.

The problem here is that this is OO code and you shouldn't be relying on execution order. The fix is simple. Rename both startup methods to `startup` and have the child class call the super class method:

```
sub startup : Tests(startup) {
    my $test = shift;
    $test->SUPER::startup;
    die unless $test->_is_connected_to_db,
      'We still have a database connection';
}
```

This works because `Test::Class` knows you've overridden the method and you can simply call it manually.

Warning: Note that we now `die` in the startup method rather than running a test. This is because `Test::Class` has no way of knowing if you're really doing to call the super class or not. As a result, it has no way of knowing what the real test count is. Thus, we `die` instead of relying on a test failure to halt the `startup` method.

Tip: for reasons mentioned above, don't put tests in your in your test control methods.

Controlling what gets executed

Let's say that you've a web page which shows information, but if the user is authenticated, they get extra features. You might test this with the following:

```
sub unauthenticated_startup : Test(startup) {
    my $test = shift;
    $test->_connect_as_unauthenticated;
}
```

And in your "authenticated" subclass:

```
sub authenticated_startup : Test(startup) {
    my $test = shift;
    $test->_connect_as_authenticated;
}
```

Again, your tests will probably fail because `authenticated_startup` is run before `unauthenticated_startup` and you have probably connected as the unauthenticated user in your "authenticated" subclass. However, this time you probably don't even need `unauthenticated_startup` to run. Again, give the tests the same name but *don't* call the parent's method.

```
sub startup : Test(startup) {
    my $test = shift;
    $test->_connect_as_authenticated;
}
```

Again note that we're not running tests in this control method. If the connect fails, throw an exception.

PERFORMANCE

With `Test::Class::Load`, you can run all of your test class tests in one process:

```
use Test::Class::Load qw(path/to/tests);
```

That loads the tests and all modules you're testing *once*. This can be a huge performance boost if you're loading "heavy" modules such as `Catalyst` or `DBIx::Class`. However, be aware that you're now loading all classes in a single process and there are potential drawbacks here. For example, if one of your classes alters a singleton or global variable that another class depends on, you may get unexpected results. Also, many classes load modules which globally alter Perl's behavior. You can grep through your CPAN modules for `UNIVERSAL::` or `CORE::GLOBAL::` to see just how many classes do this.

Bugs involving global state changes can be very hard to track down. You will have to decide for yourself whether the benefits of `Test::Class` outweigh these drawbacks. My experience is that these bugs are usually very painful to resolve, but in finding them, I often find intermittant problems in my code bases that I could not have found any other way. For me, `Test::Class` is a win here, despite occasional frustration.

For those who prefer not to run all of their code in a single process, they often create separate "driver" tests for them:

```
    #!/usr/bin/env perl -T

    use MyTest::Person;
    Test::Class->runtests;
```

And:

```
    #!/usr/bin/env perl -T

    use MyTest::Person::Employee;
    Test::Class->runtests;
```

Of course, you should omit the call to `runtests` if you've included this in your base class `INIT`.

MAKING YOUR CLASSES BEHAVE LIKE XUNIT CLASSES

In xUnit style tests, this is an entire test:

```
    sub first_name : Tests(tests => 3) {
        my $test   = shift;
        my $person = $test->class->new;
        can_ok $person, 'first_name';
        ok !defined $person->first_name,
          '... and first_name should start out undefined';
        $person->first_name('John');
        is $person->first_name, 'John', '... and setting its value should succeed';
    }
```

In the TAP world, we would look at this as three tests, but xUnit says we have three asserts to validate one feature, thus we have one test. Now TAP-based tests have a long way to go before working for xUnit users, but there's one thing we can do. Let's say that you have a test with 30 asserts and the fourth assert fails. Many xUnit programmers argue that once an assert fails, the rest of the information in the test is unreliable. Thus, the tests should be halted. Now regardless of whether or not you agree with this (I hate the fact that, for example, junit requires the test method to stop), you can get this behavior with `Test::Class`. Just use `Test::Most` instead of `Test::More` and put this in your test base class:

```
BEGIN { $ENV{DIE_ON_FAIL} = 1 }
```

Because each test method in `Test::Class` is wrapped in an eval, that test method will stop running, the appropriate `teardown` method (if any) will execute and the tests will resume with the next test method.

I'm not a huge fan of this technique, but your mileage may vary.

CONCLUSION

While many projects are just fine using simple `Test::More` programs, larger projects can wind up with scalability problems. `Test::Class` gives you better opportunities for managing your tests, refactoring common code and having your test code better mirror your production code.

Here’s a quick summary of tips listed above:

- Name your test classes consistently after the classes they’re testing.
- When possible, do the same for your test methods.
- Don’t use a constructor test named `new`.
- Don’t put your tests in the `Test::` namespace.
- Create your own `Test::Class` base class.
- Abstract the the name of the class you’re testing into a `class` method in your base class.
- Name test control methods after their attribute.
- Decide case-by-case whether to call a control method’s parent method.
- Don’t put tests in your test control methods.

ACKNOWLEDGMENTS

Thanks to Adrian Howard for creating the `Test::Class` module and providing me with tips in making it easier to use. Also, David Wheeler provided some useful comments, but that was on a first draft written years ago. I wonder if he remembers? :)