Unit Testing with Perl

Testing effectively using Perl and the Test Anything Protocol

- Daniel LeWarne
  - Perl Developer at 

    - We're hiring
  - CPAN: POSSUM
- Qualifications
  - I write a lot of tests
    - In Perl

What is Testing?

What is testing?

Three Types of Testing

→ Quality Assurance Testing (QA)

→ Integration Testing

→ Unit Testing

What is testing?

Why Test?

Why NOT?

Why Not?

→ My code is perfect

→ It's legacy code

→ This code will never change

→ It only affects a small part of the system

→ It's QA's job!

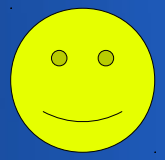
→ Testing is too hard :(

Why Not?

→ My code
→ It's legacy co
→ This code will nev
→ It only affects
→ It's QA's
→ Testi

Why Test?

- Confirm code works
- Aid in future changes
- Insure against unintended changes
- Handle edge cases (unit tests)
- You can use test cases as a plan (TDD)
- Testing is Easy!
- You're probably doing it already



You're already doing it

The Hard

```
warn "SUCCES          if $          1 "Joe";
if ($x->some_met
   print "ok, loc                  .
} else {
   print "                  .\n";
}

                e_function() ? "☃" : "☹
```

Test Anything Protocol

*"**TAP**, the **Test Anything Protocol**, is a simple text-based interface between testing modules in a test harness."*

Source: <http://testanything.org/>

Test Anything Protocol

```
1..3
ok 1
not ok 2 - Test 2 Description
ok 3 - Test 3 Description
```

Test Anything Protocol

So, TAP consists of two main parts:

- The Plan
- The Body
  - ok
  - not ok

* Parsed by Test::Harness::TAP

CPAN to the Rescue

- Test::Simple
- Test::More

Our First Test

- The Plan
  - How many tests will you run?
  - Declare this when you load the module

Our First Test

```
#!/usr/bin/perl
use strict;
use warnings;

# Load Test::Simple and *plan*
use Test::Simple tests => 1;
```

Our First Test

```
sub return_true {
  return 1;
}

ok return_true(), “Perl is predictable”;

# And that's it!
```

Our First Test

- Test::Simple
  - Simplifies writing TAP output
  - Provides a planning mechanism
  - Provides the “ok” command

Our First Test

- So what's Test::More for?

```
# Load Test::More and *plan*
use Test::More tests => 1;
```

Our First Test

```
sub return_a_string {
  return “My String”;
}

ok return_a_string() eq “My String”,“Strings match.”;

is return_a_string(), “My String”,“Strings match.”;
```

Our First Test

- Test::More

`is`	`is $x, $y`	Compares 2 values for equality
`isnt`	`isnt $x, $y`	Compares 2 values for inequality
`like`	`like $foo, qr/^foo$/`	Uses a regex
`is_deeply`	`is_deeply $refa, $refb`	Compares 2 references deeply
`diag`	`diag $comment`	Makes a nicely formatted TAP comment

And More!

Our First Test

- Running the test

```
$ perl my_test.pl
1..1
ok 1 - Perl is predictable
ok 2 - strings match.
ok 3 - strings match.
```

Our First Test

- Running the test
- The **prove** command

```
$ prove -v my_test.pl
1..1
ok 1 – Perl is predictable
ok 2 – strings match.
ok 3 – strings match.
```

(Remember to update the plan when adding tests!)

Our First Test

- prove
  - Runs the test script
  - Parses TAP output
  - Summarizes tests

Other Test::More Features

- Additional Test Functions
  - isa_ok $object, “Some::Object”;
  - can_ok $object, “some_method”;
  - BEGIN { use_ok(“Some::Module”) }
  - require_ok (“Some::Module”);
  - cmp_ok $x, '>=', $y, “Check x is at least y”;
  - pass ($test_name), fail($test_name)

Other Test::More Features

- Dynamic plans

```
use Test::More;
BEGIN { plan tests =>
    $ENV{RUN_MORE_TESTS} ? 500 : 5 }
```

- Another option is to “plan” at the end

```
use Test::More;
# test test test
done_testing();
```

Other Test::More Features

- Skipping tests

```
SKIP: {
    skip “No network support”, 2
        unless $have_network;
    net_test(“google.com”);
    net_test(“slashdot.com”);
}
```

Other Test::More Features

- Skipping the entire suite

```
use Test::More skip_all => “Some Reason”;
# Or
use Test::More;
BEGIN { if ($ENV{RUN_THIS}) {
    plan tests => 1;
  } else {
    plan skip_all => “Set RUN_THIS”;
} }
```

Other Test::More Features

- TODOs
  - Like skip, but runs tests anyway
  - Tests are expected to fail

```
TODO: {
  local $TODO = “Implement frobnicator”;
  ok ( $frobnicator->frobs );
  isa_ok ( $frobnicated, 'My::Frobnicated' );
}
```

What to Test

- Code that should be refactored
- Business logic
- Edge cases
- Newly reported bugs (every one)
- Any new code

What NOT to Test

Test **Anything** Protocol, not Test **Everything**

- Perl built-in functions
- Anything in third party modules
  - Moose attributes
  - Catalyst dispatchers
  - DBI

Building a Test Suite
Testing the Distribution

- Look for the t/ directory
- Most CPAN modules will have one
- Tools exist to generate your distribution's directory structure
  - ~~h2xs (not really what it's for)~~
  - catalyst.pl (for a Catalyst project)
  - module-starter (Module::Starter)
  - dzil (Dist::Zilla)
  - Module::New

Building a Test Suite
Testing the Distribution

- Test files are run in order of ASCII name, and descend recursively
  - t/100-bar.t runs after t/001-foo.t
  - This should **generally** not be important

Building a Test Suite
Testing the Distribution

- Test files are run in order of ASCII name, and descend recursively
  - t/100-bar.t runs after t/001-foo.t
  - This should **generally** not be important
- Choose a logical naming structure
  - t/calculations/001-riemmans-sums.t
  - t/errors/001-invalid-username.t

Building a Test Suite
Testing the Distribution

- Run your entire test suite after significant changes
  - prove -r
  - make test (ExtUtils::MakeMaker or Module::Install)
  - ./Build test (Module::Build)

Building a Test Suite
Testing the Distribution

- Consider automatic runs
  - Nightly
  - Hooks into your revision control system
    - i.e., a git commit hook, but don't go crazy
- Consider Continuous Integration
  - Such as Jenkins
    - jenkins-ci.org
  - As the name implies, runs the test suite “continuously”

Test Driven Development

- Consider adding tests prior to writing new code.
- This can:
  - Be used as a detailed spec
  - Give a clear roadmap for the code
  - In the case of bugs, help give a clear understanding of what is going on

Ensure Total Coverage

Tools exist to help monitor coverage. These include:

- Test::Pod::Coverage
  - As the name implies, tests that your documentation is complete
- Devel::Cover
  - Generates detailed, in-depth charts on which code needs further tests
  - Helps evaluate edge cases
  - Run with “cover -test” or “./Build testcover”

Testing Strategies

Mocking

- Mock parts of the code that aren't under your control
  - Modules maintained by other teams
  - API calls to external services
- Test::MockObject
  - Pretend like an object out of your control exists
- DBD::Mock
  - Mock database connection
  - This can be finicky

Fixtures

- Wikipedia: “a **test fixture** is a fixed state of the software under test used as a baseline for running tests”
- Typically dummy data that simulates a known state
- Loaded prior to running the tests against the data, either before the test run or as part of each script
  - Possibly torn down after running the tests
  - Could be a fixed data file or SQLite database
- DBIx::Class::Fixtures

xUnit Testing

- Test::Class
- Provides JUnit-like (or xUnit) testing
- Packages tests into methods

```
sub addition : Test(2) {
        is 10 + 20, 30, 'addition works';
        is 20 + 10, 30, 'both ways';
}
```

- Provides for setup and teardown

Test Like A Pro

Many additional testing modules exist on CPAN to help cover some specific areas.

Test Like A Pro

- Test::Differences
  - eq_or_diff shows `diff`-like output on failure
- Test::LongString
  - Helpful errors on long strings or binary data
- Test::Warn
  - Ensure warnings are what's expected
- Test::Exception
  - dies_ok, lives_ok, throws_ok

Test Like A Pro

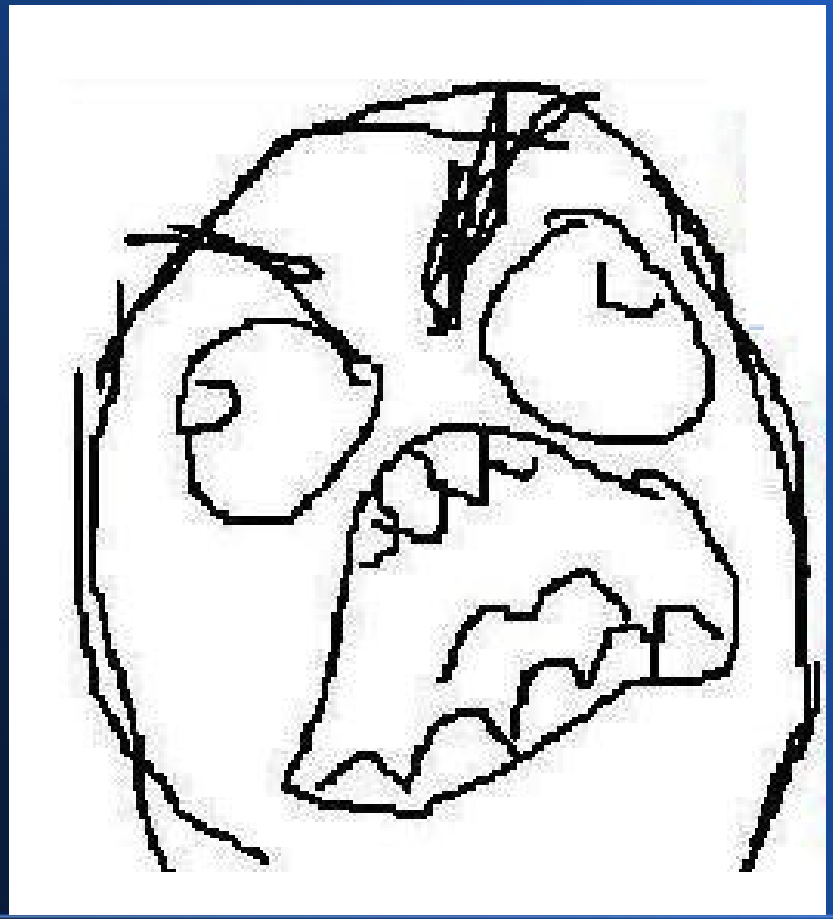
- Domain specific modules
  - Test::WWW::Mechanize
  - Test::Moose
  - Catalyst::Test
    - Test::WWW::Mechanize::Catalyst

Summary

- Test everything in your control
- Consider TDD
- Maintain and run a test suite
- Explore CPAN Test modules

Summary

- Avoid seeing clients like this:



Resources

- testanything.org
- Test::Tutorial
- Perl Testing: A Developer's Notebook (O'Reilly)
- CPAN

Credits

- Slides by Daniel LeWarne
  - Perl Developer at Grant Street Group
  - <http://cpan.org/possum>
- Special thanks to
  - The Perl Foundation
  - Presentation Contributors
  - 

    - grantstreet.com/careers

Unit Testing with Perl

Slides available at

<http://possum.cc/slides/unit_test.pdf>