Test::Tutorial

chromatic and Michael G Schwern
Testing? Why do I care?
What Is Testing?

- Check that your code does what it's supposed to do.
- At varying levels of granularity.
- In varying environments.
- At various points in its development.
What Is Automated Testing?

- Programs to check other programs.
- As with any rote task, you let the computer do it.
  - Humans will forget rote tests, computers will not
- Press a button and walk away.
  - No human required (very important)
- Manually running tests is a waste of your time.
- Tests should run as close to instantaneous as possible
  - so you won't have an excuse not to run them
  - so you'll run them as often as possible
  - Instant feedback
Testing Promotes Automation

- Testable code is decoupled
- Testable code is scriptable
Why Test?

● no missing functionality
● no accidental functionality
● when your tests pass, you're done
More informative bug reports

- Better to get the diagnostic output of your tests than "It doesn't work"
- Easily generated by end users
  - "Run the tests and send me the output"
- Helps IMMENSELY with porting (this is why my code works on VMS)
  - You can often port code without ever using the machine you're porting to
More More Reasons

- Most of the time spent on a project is debugging and bug fixing.
  - Worse, it often comes at the end (hidden cost)
  - "Oh, I'm 99% done, I just need to do some testing"

- Testing as you go will increase your development time, but reduce debugging time.
  - It will let you estimate more realistically
  - Increased project visibility
  - Reduced debug time once you get used to it
The Real Reason For Writing Tests

● Confidence.
  ♦ No fear of change
  ♦ No fear of lurking bugs
  ♦ No fear of breaking old things
  ♦ No fear that new things don't work
    □ Knowing when things don't work.

● So you can play and experiment without worry.

● Enable refactoring
Testing Is Laziness

- Take an O(n) amount of work and make it O(1)
  - Instead of walking through the code by hand at each change
  - Teach the computer to do that.
What to test
Textbook Testing

- Traditional testing philosophy says things like:
  
  Test all subroutines
  Test all branches of all conditions
  Test the boundary conditions of all inputs
  ...

- This is just big and scary and too much.

- We're lazy, and we think we can still be effective with much less work.
XP Testing

- XP says to write your tests before you write your code.
  - It's hard enough to get people to write tests at all.
  - Changing their coding philosophy at the same time is worse.
- If you can do Test First, excellent.
- If you're not already testing, this is a chance to start some new habits...
On Test-First Programming

- Think of it as coding to teeny, tiny, mini-iterations.
- Break each task into boolean expressions.
- Ask "What feature do I need next?"
  - Test the smallest and most immediate element of the overall task.
  - Take small steps!
The two test-first questions

- "How can I prove that this feature works?"
  - Write the simplest test that will fail unless the feature works.
  - The test must fail.

- "What is the least amount of code I can write to pass the test?"
  - The simpler the test, the simpler the code you need.
  - The test must now pass.

- This produces known good code and a comprehensive test suite.

- Be sure to run the entire test suite after you implement a task.

- Don't be afraid of baby steps. That's the point.
Test Bugs

- Another good philosophy is to test bugs and new features.
- Every time you find a bug, write a test for it.
- Every time you add a new feature, write a test for it.
- In the process, you might test a few other related things.
- This is the simplest way to retrofit tests onto existing code.
Effects Of Testing Bugs

● This has pleasant effects:
  ◆ Slowly grows your test suite
  ◆ Focuses tests on the parts of the code which have the most bugs

● You're allowed to make mistakes, but ONLY ONCE. Never twice.

● A disproportionate amount of bugs use the same logic.
  ◆ One test can catch lots of bugs
Knowing You're Done

t/Your-Code.t......ok
All tests successful.

● For once, your computer is telling you something good.
● Instant, positive feedback
There Is No Magic

- You may have seen this from h2xs.

```
### We start with some black magic to print on failure.

# Change 1..1 below to 1..last_test_to_print.
# (It may become useful if the test is moved to ./t subdirectory.)
BEGIN { $| = 1; print "1..1\n"; }
END { print "not ok 1\n" unless $loaded; }
use Foo;
$loaded = 1;
print "ok 1\n";

### End of black magic.

- Testing really isn't this frightening.
```
And Now For Something Completely Different
The Most Basic Perl Test Program

#!/usr/bin/perl -w

print "1..1\n";

print 1 + 1 == 2 ? "ok 1\n" : "not ok 1\n";

* Since 1 + 1 is 2, this prints:

1..1
   ok 1

* "1..1" I'm going to run one test.
* "ok 1" The first test passed.
Perl's Testing Protocol

- There are two parts to running a test in Perl.
  - Your test
  - Test::Harness
- The output of your test is piped to Test::Harness.
- Test::Harness interprets your output and reports.

```
$ perl -MTest::Harness -wle 'runtests @ARGV' contrived.t
contrived....ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs ( 0.02 cusr +  0.02 csys)
```
There's TMTOWTDI and there's this...

- Here's some of the many ways people write their tests:
  - t/op/sysio.t

    ```perl
    print 'not ' unless (syswrite(O, $a, 2) == 2);
    print "ok 20\n";
    ```

  - ext/Cwd/t/cwd.t

    ```perl
    print +(getcwd eq $start ? "" : "not "), "ok 4\n";
    ```

  - t/pod/plainer.t

    ```perl
    unless( $returned eq $expected ) {
      print map { s/^/\#/mg; $_; } 
      map {+$_} # to avoid readonly values
        "EXPECTED:\n", $expected, "GOT:\n", $returned;
      print "not ";
    }
    printf "ok %d\n", ++$test;
    ```

- Maintenance nightmare.
I'm ok, you're ok

#!/usr/bin/perl -w

use Test::Simple tests => 1;

ok( 1 + 1 == 2 );

- "ok" is the backbone of Perl testing.
  - If the expression is true, the test pass.
  - False, it fails.
- Every conceivable test can be performed just using ok().
YOU FAILED!!!

```perl
#!/usr/bin/perl -w

use Test::Simple tests => 2;
ok( 1 + 1 == 2 );
ok( 2 + 2 == 5 );

● from that comes:

1..2
ok 1
not ok 2
#     Failed test (contrived.t at line 5)
#     Looks like you failed 1 tests of 2.

◆ "1..2" I'm going to run two tests.
◆ "ok 1" The first test passed.
◆ "not ok 2" The second test failed.
◆ Some helpful commentary from Test::Simple
We're going to write some tests for Date::ICal.
  ● It's real code.
  ● It's sufficiently complex.
  ● Everyone understands dates.
    □ Some people even have them.
Where To Start?

- This is the hardest part.
- Retrofitting a test suite onto old code sucks.
  - Marching through the testing swamps.
- Write tests from the start and your life will be easier.
- In any event, begin at the beginning.
new()

- Since Date::ICal is OO, the beginning is when you make an object.
  - (white-lie: the beginning is when you load the module)

```perl
#!/usr/bin/perl -w
use Test::Simple tests => 2;
use Date::ICal;
my $ical = Date::ICal->new;       # make an object
ok( defined $ical );              # check we got something
ok( $ical->isa('Date::ICal') );   # and it's the right class
```

- This produces:

```
1..2
ok 1
ok 2
```

- This is your first useful test.
Names

● "ok 2" isn't terribly descriptive.
  ☐ what if you have 102 tests, what did #64 do?
● Each test can be given a little description.

```perl
ok( defined $ical, 'new() returned something' );
ok( $ical->isa('Date::ICal'), " and it's the right class" );
```

● This outputs

```
1..2
ok 1 - new() returned something
ok 2 - and it's the right class
```
What's In A Name

- Two views on names.
  - A name is a descriptive tag so you can track the test output back to the code which produced it. (the original purpose)
  - A name is a short description of what was tested.

- There's a subtle difference.

- Don't pull your hair out over it.
  - More importantly, don't pull other people's hair out over it.
Test The Manual

● Simplest way to build up a test suite is to just test what the manual says it does.
  ◆ Also a good way to find mistakes/omissions in the docs.
  ◆ You can take this five steps further and put the tests IN the manual. Test::Inline, later.

● If the docs are well written, they should cover usage of your code.
  ◆ You do have docs, right?
SYNOPSIS

● A good place to start.
  ◆ A broad overview of the whole system

● Here's a piece of Date::ICal's SYNOPSIS.

SYNOPSIS

use Date::ICal;

$ical = Date::ICal->new( year => 1964, month => 10, day =
                      hour => 16, min => 12, sec => 47, tz => '0530' );

$hour = $ical->hour;
$year = $ical->year;

● Oddly enough, there is a bug in this.
SYNOPSIS test

use Test::Simple tests => 8;
use Date::ICal;

$ical = Date::ICal->new(
    year => 1964, month => 10, day => 16, hour => 16,
    min => 12, sec => 47, tz => '0530' );

ok( defined $ical, 'new() returned something' );
ok( $ical->isa('Date::ICal'), "and it's the right class" );

ok( $ical->sec == 47, 'sec()' );
ok( $ical->min == 42, 'min()' );
ok( $ical->hour == 10, 'hour()' );
ok( $ical->day == 16, 'day()' );
ok( $ical->month == 10, 'month()' );
ok( $ical->year == 1964, 'year()' );
SYNOPSIS results

1..8
ok 1 - new() returned something
ok 2 - and it's the right class
ok 3 - sec()
not ok 4 - min()
# Failed test (ical.t at line 14)
not ok 5 - hour()
# Failed test (ical.t at line 15)
ok 6 - day()
ok 7 - month()
ok 8 - year()
# Looks like you failed 2 tests of 8.

● Whoops, failures!
● We know what and where it failed, but not much else.
● How do you find out more?
  ◆ Throw in print statements
  ◆ Run in the debugger.
● That sounds like work.
Test::More

- Test::Simple is deliberately limited to one function.
- Test::More does everything Test::Simple does.
  - You can literally s/use Test::Simple/use Test::More/
- It provides more informative ways to say "ok".
Test::More's `is()` function:

- declares that something is supposed to be something else
- "Is this, that?"

```perl
is( $this, $that );

# From
ok( $ical->day   == 16,       '  day()'   );

# To
is( $ical->day,     16,       '  day()'   );
```
Here's the test with ok() replaced with is() appropriately.

```perl
use Test::More tests => 8;
use Date::ICal;

$ical = Date::ICal->new(
    year => 1964, month => 10, day => 16, hour => 16,
    min  => 12, sec     => 47, tz  => '+0530' );

ok( defined $ical,            'new() returned something' );
ok( $ical->isa('Date::ICal'), "  and it's the right class" );
is( $ical->sec,     47,       '  sec()'   );
is( $ical->min,     42,       '  min()'   );
is( $ical->hour,    10,       '  hour()'  );
is( $ical->day,     16,       '  day()'   );
is( $ical->month,   10,       '  month()' );
is( $ical->year,    1964,     '  year()'  );

"Is $ical->sec, 47?"
"Is $ical->min, 12?"
```
Diagostic Output

1..8
ok 1 - new() returned something
ok 2 - and it's the right class
ok 3 - sec()
not ok 4 - min()
# Failed test (- at line 13)
#    got: '12'
#    expected: '42'
not ok 5 - hour()
# Failed test (- at line 14)
#    got: '21'
#    expected: '10'
ok 6 - day()
ok 7 - month()
ok 8 - year()
# Looks like you failed 2 tests of 8.

- $ical->min returned 12 instead of 42.
- $ical->hour returned 21 instead of 10.
Interpreting The Results

- Turns out, there is no 'tz' argument to new()!
  - And it didn't warn us about bad arguments
- The real argument is 'offset'
  - So the synopsis is wrong.
  - This is a real bug I found while writing this
- Damn those tests.
When to use is()

- Use instead of ok() when you're testing "this equals that".
  - Yes, there is an isnt() and isn't().

- is() does a string comparison which 99.99% of the time comes out right.
  - cmp_ok() exists to test with specific comparison operators
Tests Are Sometimes Wrong

● The previous example was supposed to be highly contrived to illustrate that tests are sometimes wrong.

● When investigating a test failure, look at both the code and the test.

● There's a fine line of trusting your testing code.
  ♦ Too much trust, and you'll be chasing phantoms.
  ♦ Too little trust, and you'll be changing your tests to cover up bugs.
How Can I Be Sure The Test Is Right?

- Write the test
- Run it and make sure the new test fails
- Add the new feature / fix the bug
- Run the test and make sure the new test passes.
- Some development systems, such as Aegis, can enforce this process.
- It's difficult to do this when writing tests for existing code.
  - Another reason to test as you go
Version Control and Testing

- VC & testing work well.
  - Run the tests, make sure they pass
  - Make sure everything is checked in.
  - Write tests for the bug / feature.
    - Make sure they fail.
  - Fix your bug / write your feature
  - Run the tests.
    - If they pass, commit. You're done.
    - If they fail, look at the diff. The problem is revealed by that change.

- The smaller the change, the better this works.

- You are using version control, right?
Testing vs Brooks's Law

- Tests catch damage done by a new programmer immediately
- Easier for other developers to help you
  - They can pre-test their patches.
  - Even if you write perfect code, the rest of us don't.
Testing Lots Of Values

- Date handling code is notorious for magic dates that cause problems
  - 1970, 2038, 1904, 10,000. Leap years. Daylight savings.
- So we want to repeat sets of tests with different values.
use Test::More tests => 32;
use Date::ICal;

my %ICal_Dates = (
    '19971024T120000' => # from the docs.
        [ 1997, 10, 24, 12, 0, 0 ],
    '20390123T232832' => # after the Unix epoch
        [ 2039, 1, 23, 23, 28, 32 ],
    '19671225T000000' => # before the Unix epoch
        [ 1967, 12, 25, 0, 0, 0 ],
    '18990505T232323' => # before the MacOS epoch
        [ 1899, 5, 5, 23, 23, 23 ],
);

while( my($ical_str, $expect) = each %ICal_Dates ) {
    my $ical = Date::ICal->new( ical => $ical_str, offset => 0 );
    ok( defined $ical, "new(ical => '$ical_str')" );
    ok( $ical->isa('Date::ICal'), "  and it's the right class" );
    is( $ical->year,    $expect->[0],   '  year()'  );
    is( $ical->month,   $expect->[1],   '  month()' );
    is( $ical->day,     $expect->[2],   '  day()'   );
    is( $ical->hour,    $expect->[3],   '  hour()'  );
    is( $ical->min,     $expect->[4],   '  min()'   );
    is( $ical->sec,     $expect->[5],   '  sec()'   );
}
The Good News

- If you can write good code, you can learn to write good tests.
- Just a while loop.
- Easy to throw in more dates.
The Bad News

- You have to keep adjusting the # of tests when you add a date.
  - use Test::More tests => ##;
- There are some tricks:

  # For each date, we run 8 tests.
  use Test::More tests => keys %ICal_Dates * 8;
- There's also 'no_plan':

  use Test::More 'no_plan';
Plan? There Ain't No Plan!

- The plan exists for protection against:
  - The test dying
  - Accidentally not printing tests to STDOUT
  - Exiting early
- The first two have other protections, and the third will shortly.
  - So the plan isn't as useful as it used to be
- Newer versions of Test::Harness allow the plan to be at the end:
  
  ```
  ok 1
  ok 2
  ok 3
  1..3
  ```
- This allows Test::More to count your tests for you.
  - You have to upgrade Test::Harness for this to work.
Boundary tests

- Almost bad input
- Bad input
- No input
- Lots of input
- Input that revealed a bug
Bad Input Can Do Bad Things

- Garbage in / Error out
  - graceful exceptions, not perl errors
  - helpful warnings, not uninitialized value warnings
- Make sure bad input causes predictable, graceful failure.
Basic bad input example

use Test::More tests => 2;

local $!;
ok( !open(FILE, "I_dont_exist"), 'non-existent file' );
isnt( $!, 0, ' $! set' );

- Note, the exact value of $! is unpredictable.
Tests with warnings

● Test::More used to have a problem testing undefined values

```perl
use Test::More tests => 1;
is( undef, undef, 'undef is undef' );
```

● The test will pass, but there would be warnings.
  - The user will see them, but the test will not.

● There's a whole bunch of these in Test-Simple/t/undef.t
Catching Warnings

- Use $SIG{__WARN__}.

```perl
my $warnings = ''; local $SIG{__WARN__} = sub { $warnings . join '', @_ }; use Test::More tests => 2; is( undef, undef, 'undef is undef' ); is( $warnings, '', ' no warnings' );
```

- Use the same technique to check for expected warnings.
Dealing With Death

● Use eval BLOCK.

```perl
local $@;
eval {
    croak "Wibble";
};
like( $@, qr/^Wibble/ );
```

● Use the same technique to check that things didn't die.
  ♦ Useful for past bugs where certain inputs would cause a fatal error.
Acceptance, Regression, Unit, Functional...

- Same thing, just a matter of timing.
- Unit: Detailed tests of individual parts
  - Unit tests are easy(er)
  - So we think of the rest in terms of unit tests
- Functional: Tests of your API
  - Blackbox unit tests
- Integration: Testing that the pieces work together
  - Just unit testing bigger units
- Acceptance: Tests defining your requirements
  - Customer driven unit tests
- Regression: Tests for backwards compatibility
  - Old tests never die, they just become regression tests
- All can be done with the same techniques
Blackbox vs Glassbox

- No, they're not window managers.
- Blackbox tests use only the public, documented API.
  - No cheating
  - You have to forget how the code is implemented
  - More closely approximates real world usage
  - Immune from internal changes
  - Often forces you to make the public API more flexible
- Glassbox tests can use whatever you want.
  - Cheat, steal, lie, violate encapsulation
  - Often necessary to test certain 'untestable' parts
  - May be broken by internal changes, undermines the test suite.
- Blackbox is preferred where possible, glassbox is sometimes necessary.
  - Sometimes you can just peek inside the box.
Test::More toys

- Test::More has 13 ways to say ok.
  - It also has a wonderful man page.
- Here's some of the most common.
like()

- Next to is() and ok(), you'll be using like() the most.

  ```perl
  like( $this, qr/that/ );
  ```

- This is the same as:

  ```perl
  ok( $this =~ /that/ );
  ```

- It has nicer diagnostics:

  ```perl
  not ok 1
  #     Failed test (contrived.t at line 2)
  #           'wibble'
  #     doesn't match '(?-xism:woof)'
  ```

- Because qr// was added in 5.005, it understands a string that looks like a regex for older perls.

  ```perl
  like( $this, '/that/' );
  ```

- There is an unlike() which is the !~ version.
isa_ok()

- We've been doing this a lot.

```perl
ok( defined $ical, "new(ical => '$ical_str')" );
ok( $ical->isa('Date::ICal'), " and it's the right class" );
```

- You do this so much in OO code, there's a special function.

```perl
isa_ok( $ical, 'Date::ICal' );
```

- It works on references, too.

```perl
isa_ok( $foo, 'ARRAY' );  # is $foo an array ref?
```

- It also has nice diagnostics.

```perl
not ok 1 - The object isa Date::ICal
#     Failed test (- at line 2)
#     The object isn't a 'Date::ICal' it's a 'ARRAY'
```
can_ok()

- A test for $obj->can($some_method)

  ```perl
  ok( $obj->can('foo'), 'foo() method inherited' );
  ```

- Simple but useful test can be like:

  ```perl
  # Does the Foo class have these methods?
  can_ok( 'Foo', qw(this that whatever wibble) );
  ```

- Might seem silly, but can catch stupid mistakes like forgetting a "=cut"

- Takes an object or a class.

- Also useful for checking your functions are exported

  ```perl
  use Text::Soundex;
  can_ok(__PACKAGE__, 'soundex');
  ```
use_ok()

- The real first thing you test is if the module loaded.

```
use Test::More tests => 1;
BEGIN { use_ok( 'Date::ICal' ); }
```

- Has to be inside a BEGIN block to act like a real 'use'.
- Remember the black magic? That's what it was doing.
is_deeply()

- For comparing complex data structures
  - Hashes, lists, hash of lists of hashes of lists of scalar references...

    ```perl
    my %expect = ( this => 42, that => [qw(1 2 3)] );
    my %got = some_function();
    is_deeply( \%got, \%expect );
    ```

- Will show you where the two structures start to diverge

    ```perl
    not ok 1
    #     Failed test (- at line 2)
    #     Structures begin differing at:
    #     $got->{that}[2] = '3'
    #     $expected->{that}[2] = Does not exist
    ```

- In CS this is really a "shallow comparison" and is() is "deep".
  - So the name is wrong because Schwern failed CS.

- A stopgap measure
  - Currently doesn't handle circular structures (patches welcome)

- Waiting for someone to step up to the plate and write Test::Set
diag()

- Test::More's functions are pretty good about providing diagnostics.
- Sometimes you need more...
- diag() lets you display whatever diagnostic information you want.
  - Guaranteed not to interfere with Test::Harness
  - Not a test function
  - Will not display inside a TODO block
- Useful for giving suggestions about tricky failures
Odd User Reactions

● Sometimes users react rather oddly to tests.
  ✦ won't report failures
  ✦ will react to failures as if the test caused the bug!
  ✦ will report "the tests failed" and leave off all the diagnostics
  ✦ won't run the tests at all
Getting People to RUN Your Tests

- Once you've gotten people writing tests...
- ...your next problem is getting them to RUN them
Make It Simple

- Preferably ONE command.
  - no user interaction (or smart defaults)
  - 'make test'
  - 'quicktest' CVS integration.
Test On Commit

- Make running the tests part of your commit policy
  - Automate with CVS commit actions (CVSROOT/modules)
  - Use a system such as Aegis
Daily Smoke Test

- Run the whole battery of tests against the latest code every day, automatically
  - CPAN::Smoke is one example
Test Before Release

- Automatically run tests as part of your release process.
  - 'make disttest'
  - your release process is automated, right?
Testing Is Eating Your Own Dog Food

- It forces you to use your own API.
- Code that's hard to test may be hard to use.
- This often makes your API more flexible.
  - Tends to get rid of constants and assumptions
'make test'

schwern@blackrider:~/src/devel/File-chdir$ make test
PERL_DL_NONLAZY=1 /usr/local/bin/perl5.6.1 -Iblib/arch
-iblib/lib -I/usr/local/perl5.6.1/lib/5.6.1/ppc-linux-64int
-ibusr/local/perl5.6.1/lib/5.6.1 -e
'use Test::Harness qw(&runtests $verbose); $verbose=0;
runtests @ARGV;' t/*.t

t/array.............ok
t/chdir...............ok
t/var..................ok
All tests successful.
Files=3, Tests=48, 2 wallclock secs
( 1.71 cusr +  0.38 csys =  2.09 CPU)

- When you run 'make test' on a CPAN module, you're using:

  ExtUtils::MakeMaker
  Test::Harness
  your test
What in the hell is all that mess?

PERL_DL_NONLAZY=1

- magic to force XS code to strictly check shared libraries
  
  -Iblib/lib  -Iblib/lib

- Changes @INC to use the module you're about to install
  
  -I/usr/local/perl5.6.1/lib/5.6.1/ppc-linux-64int ...

- Mistake. Code specific for testing Perl itself that leaked out.

- Causes problems with core modules on CPAN.

- Fixed in latest versions of MakeMaker.
The mess continued...

- e 'use Test::Harness qw(&runtests $verbose);

- import runtests and $verbose
  
  $verbose=0

- This is really $verbose=$(TEST_VERBOSE)
  
  runtests @ARGV;' t/*.t

- Pass in all your tests to Test::Harness::runtests()
Still more mess...

```bash
  t/array.................ok
  t/chdir................ok
  t/var..................ok
```

- Your tests are all ok

  All tests successful.

- It's Miller Time.

  Files=3, Tests=48, 2 wallclock secs
  ( 1.71 cusr +  0.38 csys =  2.09 CPU)

- Benchmark of how long your tests took to run. May go away.
New MakeMaker Is A Little Different

$ make test
PERL_DL_NONLAZY=1 /usr/local/bin/perl
"-MExtUtils::Command::MM" "-e"
"test_harness(0, 'blib/lib', 'blib/arch')" t/*.
t/array....ok
t/chdir....ok
t/var.......ok
All tests successful.
Files=3, Tests=48, 3 wallclock secs
( 2.27 cusr + 0.48 csys = 2.75 CPU)

- The -I$(PERL_LIB) -I$(PERL_ARCH) mistake is gone
- The hanging Test::Harness wires have been put away
- Mostly done for non-Unix platforms.
Some modules put tests in test.pl.

Do not do that.

'make test' does not parse the output which means...

- 'make test' won't exit with non-zero on failure.
- Things like the CPAN shell won't know there was a failure.
- Historical accident, MakeMaker predates Test::Harness.
Testing and Perl versions

- Test::Simple/More will be in 5.8.0.
- Test.pm was put in 5.4.5.
- Test::Harness has been around so long nobody remembers who wrote it.
  - pre-5.6.1 will not support TODO tests or no_plan.
- They're all available from CPAN.
- They all work back to 5.4.0.
- They all work on every platform.
Some people worry about having too many prereqs on their CPAN modules

- Don't want to add prereqs on testing modules

A prereq of Test::More in turn prereqs & upgrades Test::Harness.

Even though Test::More isn't yet in core, it's already widely installed.

Acme::ComeFrom, Acme::Magpie, Acme::Time::Asparagus, Acme::USIG, Acme::Your, Alzabo, Apache::ConfigParser, Apache::DefaultCharset, Apache::GuessCharset, Apache::RSS, Apache::Session::CacheAny, Apache::Session::Generate::ModUniqueId, Apache::Session::Generate::ModUsertrack, Apache::Session::PHP, Apache::Session::SQLite, Apache::Singleton, Apache::StickyQuery, App::Info, Archive::Any, Astro::Funtools::Parse, Attribute::Profiles, Attribute::Protected, Attribute::Unimplemented, CPAN, Business::Tax::Vat, Cache::Mmap, Carp::Assert, CDDB::File, CGI::Application, CGI::FormMagick, CGI::Untaint, CGI::Untaint::creditcard, CGI::Untaint::email, CGI::Untaint::uk_postcode, Class::DBI,
More modules with Test::Simple/Test::More prerequisites

Class::DBI::FromCGI, Class::DBI::Join, Class::DBI::mysql, Class::DBI::SQLite, Class::Factory, Class::Observable, Class::PseudoHash, Class::Trigger, CompBio, File::Random, Crypt::CAST5_PP, Crypt::OOEnigma, Data::BFDump, Data::BT::PhoneBill, Date::Chinese, Date::DayOfWeek, Date::Discordian, Date::Easter, Date::Passover, Date::ICal, Date::ISO, Date::Japanese, Date::Leapyear, Date::Range, Date::Range::Birth, Date::Roman, Date::Set, Date::SundayLetter, Devel::Caller, Devel::LexAlias, Devel::Profiler, Devel::Tinderbox::Reporter, DNS::Singleton, Email::Find, Email::Valid::Loose, Encode::Punycode, Getopt::ArgvFile, GraphViz::Data::Structure, Hash::Merge, HTML::Calendar::Simple, HTML::DWT, HTML::ERuby, HTML::FromANSI, HTML::LBI, HTML::Lint, HTML::TableParser, HTML::Template::JIT, HTML::TextToHTML, I18N::Charset, IDNA::Punycode, Ima::DBI, Image::DS9, Inline::TT, IO::File::Log, Lingua::Pangram, Lingua::SoundChange, Lingua::Zompist::Barakhinei, Lingua::Zompist::Cadhinor, Lingua::Zompist::Kebreni, Lingua::Zombist::Verdurian, Locale::Maketext::Lexicon, Log::Dispatch::Config, Log::Dispatch::DBI, Mail::Address::MobileJp,
Everybody's Depending on Us!

So the prerequisite will likely already be resolved.

Brought to you by Schwern Of Borg.
t/lib trick

- If you still don't want to have prerequisites on testing modules
  - Copy Test/Builder.pm & Test/More.pm into t/lib/
  - Slap a "use lib 't/lib'" on your tests
  - distribute the whole thing

- Who does this?
  - CGI, CPANPLUS, MakeMaker, parrot, Test::Harness

- Caveats
  - You'll be adding to Test::More's takeover of search.cpan.org
  - Adds 18K to your tarball.
  - Can't use TODO or no_plan.
Make the GUI layer thin

- GUIs, CGI programs, etc... are hard to test.
- Make the problem as small as possible.
  - Separate the form from the functionality.
  - Put as much code into format agnostic libraries as possible
  - Large, stand-alone programs (especially CGIs) ring alarm bells.
- You might wind up with a small amount that still needs to be tested by hand.
  - At least you don't have to test the whole thing by hand.
Testing Web Stuff

● WWW::Automate is your friend.
  ◆ LWP with lots of help.
  ◆ Easily deals with forms
  ◆ "Click" on buttons
  ◆ Follow links
  ◆ Has a "back" button

● Makes simulating a real web site user easier.
Domain Specific Test Libraries

- WWW::Automate
  - Technically not a test library, but sooooo useful
- Test::Exception
- Test::Differences
  - Testing large blocks of text and complicated structures
- Test::Unit
  - Straight XUnit port to Perl
  - Great for those used to JUnit & PyUnit
- Test::Class
  - XUnit, but adapted to Perl
  - Inherited tests
- Test::MockObject
- Test::Inline
  - Embed tests in your documentation
- Test::Mail
  - and many more
Test::Builder

- Usually you want Test::More's general functions + domain specific ones.
  - Unfortunately, sometimes test libraries don't play well together
  - Who owns the test counter?
  - Who prints the plan?

- Test::Builder is a single backend to solve that problem.
  - Singleton object to handle the plan and the counter
  - Test::More-like methods you can write wrappers around

- Test libraries built on Test::Builder will work together.

  Test::Exception, Test::Class, Test::MockObject, Test::Inline, Test::Mail, Test::More, Test::Simple

- Attend "Writing A Test Library" for more information
Passing Tests Should PASS

- One must trust their test suite, else it will be ignored.
- When it fails, it should indicate a **real problem**.
- "Expected failures" sap that trust.
  - "Oh, don't worry, that test always fails on Redhat 6.2"
  - If a failure sometimes isn't really a failure, when do you know a real failure?
- "Expected failures" make test automation impossible.
  - Programs don't know "well, the test failed but it really passed"
  - Joe CPAN module installer also doesn't know that.
- Get your test suite at 100% and **keep it there**.
  - That's worth saying again.
- **STAY AT 100% PASSING!**
Failure Is An Option

● There are three varieties of test failure, and several solutions.
  ◆ A failure indicating a mistake/bad assumption in the test suite.
    □ You fix it.
  ◆ A real failure indicating a bug or missing feature.
    □ You fix it, or...
    □ You put off fixing it and...
    □ comment out the test (blech) or...
    □ declare it "TODO"
  ◆ A failure due to an assumption about the environment.
    □ You can't fix it, so you "skip" it.
It'll Never Work

- Sometimes, a test just doesn't make sense in certain environments.

- Some examples...
  - Features which require a certain version of perl
  - Features which require perl configured a certain way (ex. threads)
  - Features which are platform specific
  - Features which require optional modules
## Skipping Tests

- Let's assume we have a test for an HTML generator.
- Let's also assume that if we have HTML::Lint, we want to lint the generated code.

```perl
require HTML::Lint;

my $lint = HTML::Lint->new;
isa_ok( $lint, 'HTML::Lint' );

$lint->parse( $some_html );
is( $lint->errors, 0, 'No errors found in HTML' );
```

- Since HTML::Lint is optional, this test will fail if you don't have it.
  - But it's not a real failure, else HTML::Lint isn't really optional.
  - So the user shouldn't hear about it.
# SKIP

- You can explicitly skip a set of tests rather than run them.

```
1..2
ok 1
ok 2 # SKIP no beer
```

- Test #1 passed.
- Test #2 was skipped because there is no beer.
- A skipped test means the test was **never run**.
**SKIP: block**

- Test::More can cause an entire block of code **not to run at all**.

```perl
SKIP: {
    eval { require HTML::Lint };

    skip "HTML::Lint not installed", 2 if $@;

    my $lint = new HTML::Lint;
    isa_ok( $lint, "HTML::Lint" );

    $lint->parse( $html );
    is( $lint->errors, 0, "No errors found in HTML" );
}
```

- if we don't have HTML::Lint, the skip() function is run.
- skip() prevents anything further in the SKIP block to be run.
- the number indicates how many tests you would have run.
- The appropriate number of 'ok's will be output.

```perl
ok 23 # SKIP HTML::Lint not installed
ok 24 # SKIP HTML::Lint not installed
```
skipall

- In some cases you want to skip a whole test file.

  ```perl
  use Test::More;
  if( $^O eq 'MSWin32' ) {
    plan tests => 42;
  } else {
    plan skip_all => 'Win32 specific test';
  }
  ```

- Test::More will exit at the skip_all.
- On non-Win32, the output will be:

  ```
  1..0 # skip Win32 specific test
  ```

- Test::Harness will interpret this as a skipped test.
Procrastination Codified

- It's good to write the test before you add a new feature.
- It's good to write a test as soon as you receive a bug report.
- It's bad to release code with failing tests.
- This would seem to be a contradiction.
  - Either you fix all your bugs and add all your features immediately
  - Or you comment out your failing tests.
- Option #3, for the professionally lazy:
  - Declare your failing tests to be "todo"

This allows one to build a test suite without having to fix all the bugs you find right away.
TODO Test

TODO: {
    local $TODO = 'URI::Geller not quite working';
    
    my $card = 'Eight of clubs';
    is( URI::Geller->your_card, $card, 'Is this your card?' );
    
    my $spoon;
    URI::Geller->bend($spoon);
    is( $spoon, 'bent', 'Spoon bending' );
}

● Output will be something like:

    not ok 23  -  Is this your card
      # TODO URI::Geller not quite working
    not ok 24  -  Spoon bending
      # TODO URI::Geller not quite working
Automated TODO List

● TODO reverses the sense of the test
  ◆ 'not ok' will be treated as a quiet success
  ◆ 'ok' Test::Harness will warn you of an "unexpected success"

● It's a TODO list
  ◆ Write your tests before your feature/bug fix
  ◆ Each 'unexpected success' is an item off your todo list
  ◆ Remove the TODO wrapper

● You can release at any point and not have to cull your test suite

● Keeps users from seeing "expected failures"

● Each open bug can have a test.
  ◆ Sometimes bugs get accidentally fixed
Keep Test Scripts Small

- Many testing questions start with
  - "I've got this test script with 1400 tests..."

- Big tests are
  - Hard to maintain
  - Hard to decouple
  - Hard to read
  - Take a long time to run
  - Have all the same problems as big subroutines

- Keep them small & focused.
  - One function or set of functions per script
  - One aspect per script
  - Put complicated tests in their own script
  - Put slow tests in their own script

- Test::Simple/More's tests are a good example
Big FTP/XML program example

- Common testing problem. You have a big program which...
  - Downloads an XML file via FTP
  - Parses the XML
  - Generates HTML

- How do you test that?
Programs Are Hard, Libraries Are Easy

● The smaller the piece, the better.
● The more flexible the piece, the better.
● The more hooks into the guts, the better.
  ◆ Libraries of functions can have small, flexible pieces.
  ◆ Programs are, by definition, monolithic.

● Extract pieces out of your program and put it into a library
  ◆ Then test the library
  ◆ Side-benefit, you'll have improved your code

● Take the FTP, XML parsing and HTML generation code out of the program.
Separate Form And Functionality

- HTML is hard to test
  - It changes a lot
  - It's hard to parse
- Instead of going from XML straight to HTML
- ...go from XML -> agnostic format -> HTML
  - Test the XML -> agnostic part
  - Test the agnostic -> HTML part
- Much easier to test when only one of the input/output pair is formatted.
- ...and you'll have improved the flexibility of your code.
Mock Code

- Sometimes you just can't run a piece of code in a test
  - Maybe there's no network connection
  - Maybe the test is destructive (system("/sbin/shutdown now"))

- Going to the extreme edge of glassbox testing, replacing code for testing
System call / Power manager example

- Say you have to test a power management daemon
- One of the things it does is puts the computer to sleep
- How do you test that?

```perl
sub should_i_sleep {
    my($power_remaining) = @_;  
    system("/sbin/snooze") if $power_remaining < $Min_Power;
    return 1;
}
```
First, Isolate The Untestable Part

```perl
sub should_i_sleep {
    my($power_remaining) = @_;

    snooze if $power_remaining < $Min_Power;
    return 1;
}

sub snooze {
    system("/sbin/snooze");
}
```

- Test snooze() by hand once.
  - It's small, so you can get away with it
Then, Replace The Untestable Part

```perl
{  
    my @snooze_args = ();
    my $snooze_called = 0;
    local *Power::Manager::snooze = sub {  
        $snooze_called++;
        @snooze_args = @_;  # trap the arguments
        return 0;  # simulate successful system call
    }

    should_i_sleep($Min_Power - 1);
    is( $snooze_called, 1, 'snooze called once' );
    is( @snooze_args, 0, '  called properly'  );
}
```

- Check that it was called.
- Check that it got the right arguments
- By changing the return value to non-zero we can simulate a failure.
- Very, very powerful technique.
Forcing Failure

- How will your program react if, say, the database connection fails?

```perl
use DBI;
{
    local *DBI::connect = sub {
        return 0;
    };

    ...test for graceful failure here...
}

...test for graceful recovery here...
```
He's Your Dog, Charlie Brown

- Don't leave testing for the QA guys
  - too much delay
  - too much animosity

- You know your code, you can test it
  - and you can fix it
  - and you wrote it, so it's your bug :P
Further Reading

- perl-qa@perl.org
- http://archive.develooler.com/perl-qa@perl.org/
- "Perl Debugged"
- "Writing Solid Code"
Thanks

- Norman Nunley
- Andy Lester
- Barrie Slaymaker
- H. Merijn Brand
- Jarkko Hietaniemi
- Tatsuhiko Miyagawa
- Tels
- Rafael Garcia-Suarez
- Abhijit Menon-Sen
- Curtis Poe & OTI
- Beer and Root Beer (fuel of champions)