Chapter 3

Test Now, Test Forever (Diagnosis)

"A crash is when your competitor's program dies. When your program dies, it is an 'idiosyncrasy'. Frequently, crashes are followed with a message like 'ID 02'. 'ID' is an abbreviation for idiosyncrasy and the number that follows indicates how many more months of testing the product should have had."

— Guy Kawasaki

This chapter might appear at first blush to be out of sequence. We're steadily getting more specific in the details of taking over code, and here near the beginning is a chapter on how to do testing. Shouldn't it be near the end?

In a word, no. I am going to describe a philosophy of testing that will revolutionize your development practices if you have not already encountered it. I will show you how to implement it for Perl code and give a detailed example. It is so pivotal to the development process that I want to make sure you see it as soon as possible. So yes, it really should come before everything else.

3.1 Testing Your Patience

Here's the hard part. Creating tests while you're writing the code for the first time is far, far easier than adding them later on. I know it looks like it should be exactly the same amount of work, but the issue is motivation. When robots are invented that can create code, they won't have this problem, and the rest of us can mull over this injustice while we're collecting unemployment pay (except for the guy who invented the robot, who'll be sipping margaritas on a beach somewhere, counting his royalties and hoping that none of the other programmers recognize him).

But we humans don't like creating tests because it's not in our nature; we became programmers to exercise our creativity, but "testing" conjures up images of slack-jawed drones looking for defects in bolts passing by them on a conveyor belt.

The good news is that the Test:: modules make it easy enough to overcome this natural aversion to writing tests at the time you're developing code. The point at which you've just finished a new function is when your antitesting hormones are at their lowest ebb because you want to know whether or not it works. Instead of running a test that gets thrown away, or just staring at the code long enough to convince yourself that it *must* work, you can instead write a real test for it, because it may not require much more effort than typing:

```
is(some_func("some", "inputs"), qr/some outputs/,
   "some_func works");
```

The bad news is that retrofitting tests onto an already complete application requires much more discipline. And if anything could be worse than that, it would be retrofitting tests onto an already complete application that you didn't write.

There's no magic bullet that'll make this problem disappear. The only course of action that'll take more time in the long run than writing tests for your inherited code is not writing them. If you've already discovered the benefits of creating automated tests while writing an application from scratch then at least you're aware of how much they can benefit you. I'll explore one way to make the test writing more palatable in the next chapter.

3.2 Extreme Testing

This testing philosophy is best articulated by the Extreme Programming (XP) methodology, wherein it is fundamental (see [BECK00]). On the subject of testing, XP says:

- Development of tests should precede development of code.
- All requirements should be turned into tests.
- All tests should be automated.
- The software should pass all its tests at the end of every day.
- All bugs should get turned into tests.

If you've not yet applied these principles to the development of a new project, you're in for a life-altering experience when you first give them an honest try. Because your development speed will take off like a termite in a lumberyard.

Perl wholeheartedly embraces this philosophy, thanks largely to the efforts in recent years of a group of people including Michael Schwern, chromatic, and others. Because of their enthusiasm and commitment to the testing process, the number of tests that are run when you build Perl from the source and type "make test" has increased from 5,000 in Perl 5.004_04 (1997) to 70,000 in the current development version of Perl 5.9.0 (2004). That's right, a 14-fold increase.

True to the Perl philosophy, these developers exercised extreme laziness in adding those thousands of tests (see sidebar). To make it easier to create tests for Perl, they created a number of modules that can in fact be used to test anything. We'll take a look at them shortly.

What is it about this technology that brings such joy to the developer's heart? It provides a safety net, that's what. Instead of perennially wondering whether you've accidentally broken some code while working on an unrelated piece, you can make certain at any time. If you want to make a radical change to some interface, you can be sure that you've fixed all the dependencies because every scenario that you care about will have been captured in a test case, and running all the tests is as simple as typing "make test". One month into creating a new system that comprised more than a dozen modules and as many programs, I had built up a test suite that ran nearly 600 tests with that one command, all by adding the tests as I created the code they tested. When I made a radical change to convert one interface from functional to object-oriented, it took only a couple of hours because the tests told me when I was done.

This technique has been around for many years, but under the label *regression testing*, which sounds boring to anyone who can even figure out what it means.[1] However, using that label can be your entrance ticket to respectability when trying to convince managers of large projects that you know what you're talking about.

1. It's called regression testing because its purpose is to ensure that no change has caused any part of the program to *regress* back to an earlier, buggier stage of development.

What's this about laziness? Isn't that a pejorative way to describe luminaries of the Perl universe?

Actually, no; they'd take it as a compliment. Larry Wall enumerated three principal virtues of Perl programmers:

1. *Laziness:* "Hard work" sounds, well, hard. If you're faced with a mindless, repetitive task—such as running for public office—then laziness will make you balk at doing the same thing over and over again. Instead of stifling your creative spirit, you'll cultivate it by inventing a process that automates the repetitive task. If the Karate Kid had been a Perl programmer, he'd have abstracted the common factor from "wax on" and "wax off" shortly before fetching an orbital buffer. (Only to get, er, waxed, in the tournament from being out of shape. But I digress.)
2. *Impatience:* There's more than enough work to do in this business. By being impatient to get to the next thing quickly, you'll not spend unnecessary time on the task you're doing; you'll find ways to make it as efficient as possible.
3. *Hubris:* It's not good enough to be lazy and impatient if you're going to take them as an excuse to do lousy work. You need an unreasonable amount of pride in your abilities to carry you past the many causes for discouragement. If you didn't, and you thought about all the things that could go wrong with your code, you'd either never get out of bed in the morning, or just quit and take up potato farming.

So what are these magic modules that facilitate testing?

3.2.1 The Test Module

Test.pm was added in version 5.004 of Perl. By the time Perl 5.6.1 was released it was superseded by the Test::Simple module, which was published to CPAN and included in the Perl 5.8.0 core. Use Test::Simple instead.

If you inherit regression tests written to use Test.pm, it is still included in the Perl core for backward compatibility. You should be able to replace its use with Test::Simple if you want to start modernizing the tests.

3.2.2 The Test::Simple Module

When I say "simple," I mean *simple*. Test::Simple exports precisely one function, `ok()`. It takes one mandatory argument, and one optional argument. If its first argument evaluates to true, it prints "ok"; otherwise it prints "not ok". In each case it adds a number that starts at one and increases by one for each call to `ok()`. If a second argument is given, `ok()` then prints a dash and that argument, which is just a way of annotating a test.

Doesn't exactly sound like rocket science, does it? But on such a humble foundation is the entire Perl regression test suite built. The only other requirement is that we know how many tests we expected to run so we can tell if something caused them to terminate prematurely. That is done by an argument to the `use` statement:

```
use Test::Simple tests => 5;
```

The output from a test run therefore looks like:

```
1..5
ok 1 - Can make a frobnitz
ok 2 - Can fliggle the frobnitz
not ok 3 - Can grikkle the frobnitz
ok 4 - Can delete the frobnitz
ok 5 - Can't use a deleted frobnitz
```

Note that the first line says how many tests are expected to follow. That makes life easier for code like Test::Harness (see Section 3.2.9) that reads this output in order to summarize it.

3.2.3 The Test::More Module

You knew there couldn't be a module called Test::Simple unless there was something more complicated, right? Here it is. This is the module you'll use for virtually all your testing. It exports many useful functions aside from the same `ok()` as Test::Simple. Some of the most useful ones are:

`is($expression, $value, $description)`
: Same as `ok($expression eq $value, $description)`. So why bother? Because `is()` can give you better diagnostics when it fails.

`like($attribute, qr/regex/, $description)`
: Tests whether `$attribute` matches the given regular expression.

`is_deeply($struct1, $struct2, $description)`
: Tests whether data structures match. Follows references in each and prints out the first discrepancy it finds, if any. Note that it does not compare the packages that any components may be blessed into.

`isa_ok($object, $class)`
: Tests whether an object is a member of, or inherits from, a particular class.

`can_ok($object_or_class, @methods)`
: Tests whether an object or a class can perform each of the methods listed.

`use_ok($module, @imports)`
: Tests whether a module can be loaded (if it contains a syntax error, for instance, this will fail). Wrap this test in a `BEGIN` block to ensure it is run at compile time, viz: `BEGIN {use_ok("My::Module")}`

There's much more. See the Test::More documentation. I won't be using any other functions in this chapter, though.

Caveat: I don't know why you might do this, but if you `fork()` inside the test script, don't run tests from child processes. They won't be recognized by the parent process where the test analyzer is running.

3.2.4 The Test::Exception Module

No, there's no module called Test::EvenMore.[2] But there is a module you'll have to get from CPAN that can test for whether code lives or dies: Test::Exception. It exports these handy functions:

`lives_ok()`
: Passes if code does not die. The first argument is the block of code, the second is an optional tag string. Note there is *no comma* between those arguments (this is a feature of Perl's prototyping mechanism when a code block is the first argument to a subroutine). For example:

```
lives_ok { risky_function() } "risky_function lives!";
```

`dies_ok()`
: Passes if the code *does* die. Use this to check that error-checking code is operating properly. For example:

```
dies_ok { $] / 0 } "division by zero dies!";
```

`throws_ok()`
: For when you want to check the actual text of the exception. For example:

```
throws_ok { some_web_function() } qr/URL not found/,
          "Nonexistent page get fails";
```

The second argument is a regular expression that the exception thrown by the code block in the first argument is tested against. If the match succeeds, so does the test. The optional third argument is the comment tag for the test. Note that there *is* a comma between the second and third arguments.

2. Yet. I once promised Mike Schwern a beer if he could come up with an excuse to combine the UNIVERSAL class and an export functionality into UNIVERSAL::exports as a covert tribute to James Bond. He did it. Schwern, I still owe you that beer . . . .

3.2.5 The Test::Builder Module

Did you spot that all these modules have a lot in common? Did you wonder how you'd add a Test:: module of your own, if you wanted to write one?

Then you're already thinking lazily, and the testing guys are ahead of you. That common functionality lives in a superclass module called Test::Builder, seldom seen, but used to take the drudgery out of creating new test modules.

Suppose we want to write a module that checks whether mail messages conform to RFC 822 syntax.[3] We'll call it Test::MailMessage, and it will export a basic function, `msg_ok()`, that determines whether a message consists of an optional set of header lines, optionally followed by a blank line and any number of lines of text. (Yes, an empty message is legal according to this syntax. Unfortunately, too few people who have nothing to say avail themselves of this option.) Here's the module:

Example 3.1 Using Test::Builder to Create Test::MailMessage

```
1  package Test::MailMessage;
2  use strict;
3  use warnings;
4  use Carp;
5  use Test::Builder;
6  use base qw(Exporter);
7  our @EXPORT = qw(msg_ok);
8
9  my $test = Test::Builder->new;
10
11 sub import
12 {
13   my $self = shift;
14   my $pack = caller;
15
16   $test->exported_to($pack);
17   $test->plan(@_);
```

3. http://www.faqs.org/rfcs/rfc822.html

```
18
19    $self->export_to_level(1, $self, 'msg_ok');
20  }
21
22  sub msg_ok
23  {
24    my $arg = shift;
25    my $tester = _new();
26    eval
27    {
28      if (defined(fileno($arg)))
29      {
30        while (<$arg>)
31        {
32          $tester->_validate($_);
33        }
34      }
35      elsif (ref $arg)
36      {
37        $tester->_validate($_) for @$arg;
38      }
39      else
40      {
41        for ($arg =~ /(.*\n)/g)
42        {
43          $tester->_validate($_);
44        }
45      }
46    };
47    $test->ok(!$@, shift);
48    $test->diag($@) if $@;
49  }
50
51  sub _new
52  {
53    return bless { expect => "header" };
54  }
55
56  sub _validate
57  {
58    my ($self, $line) = @_;
59    return if $self->{expect} eq "body";
```

```
60    if ($self->{expect} eq "header/continuation")
61    {
62      /^\s+\S/ and return;
63    }
64    $self->{expect} = "body", return if /^$/;
65    /^\S+:/ or croak "Invalid header";
66    $self->{expect} = "header/continuation";
67  }
68
69  1;
```

In line 1 we put this module into its own package, and in lines 2 and 3 we set warnings and strictness to help development go smoothly. In lines 4 and 5 we load the Carp module so we can call `croak()`, and the Test::Builder module so we can create an instance of it. In lines 6 and 7 we declare this to be a subclass of the Exporter module, exporting to the caller the subroutine `msg_ok()`. (Note that this is *not* a subclass of Test::Builder.)

In line 9 we create a Test::Builder object that will do the boring part of testing for us. Lines 11 through 20 are copied right out of the Test::Builder documentation; the `import()` routine is what allows us to say how many tests we're going to run when we `use` the module.

Lines 22 through 49 define the `msg_ok()` function itself. Its single argument specifies the mail message, either via a scalar containing the message, a reference to an array of lines in the message, or a filehandle from which the message can be read. Rather than read all of the lines from that filehandle into memory, we're going to operate on them one at a time because it's not necessary to have the whole message in memory. That's why we create the object `$tester` in line 25 to handle each line: it will contain a memory of its current state.

Then we call the `_validate()` method of `$tester` with each line of the message. Because that method will `croak()` if the message is in error, we wrap those loops in an `eval` block. This allows us easily to skip superfluous scanning of a message after detecting an error.

Finally, we see whether an error occurred; if an exception was thrown by `croak()` inside the `eval` block, `$@` will contain its text; otherwise `$@` will be empty. The `ok()` method of the Test::Builder object we created is the same function we're used to using in Test::Simple; it takes a true or false value, and an optional tag string, which we pass from our caller. If we had an exception, we pass its text to Test::Builder's `diag()` method, which causes it to be output as a comment during testing.

The `_new()` method in lines 50–53 is not called `new()` because it's not really a proper constructor; it's really just creating a state object, which is why we didn't bother to make it inheritable. It starts out in life expecting to see a mail header.

Lines 56–70 validate a line of a message. Because anything goes in a message body, if that's what we're expecting we have nothing to do. Otherwise, if we're expecting a header or header continuation line, then first we check for a continuation line (which starts with white space; this is how a long message header "overflows"). If we have a blank line (line 67), that separates the header from the body, so we switch to expecting body text.

Finally, we must at this point be expecting a header line, and one of those starts with non-white-space characters followed by a colon. If we don't have that, the message is bogus; but if we do, the next line could be either a header line or a continuation of the current header (or the blank line separating headers from the body).

Here's a simple test of the Test::MailMessage module:

```
1  #!/usr/bin/perl
2  use strict;
3  use warnings;
4
5  use lib qw(..);
6  use Test::MailMessage tests => 2;
7
8  msg_ok(<<EOM, "okay");
9  from: ok
```

```
10 subject: whatever
11
12 body
13 EOM
14 msg_ok(\*DATA, "bogus");
15
16 __END__
17 bogus mail
18 message
```

The result of running this is:

```
1..2
ok 1 - okay
not ok 2 - bogus
#     Failed test (./test at line 14)
# Invalid header at ./test line 14
# Looks like you failed 1 tests of 2.
```

Although we only used one Test:: module, we could have used others, for example:

```
use Test::MailMessage tests z=> 2;
use Test::Exception;
use Test::More;
```

Only one of the `use` statements for Test::Modules should give the number of tests to be run. Do not think that each `use` statement is supposed to number the tests run by functions of that module; instead, one `use` statement gives the total number of tests to be run.

`brian d foy`[4] used Test::Builder to create Test::Pod,[5] which is also worth covering.

4. That's not a typo; he likes his name to be spelled, er, rendered that way, thus going one step farther than bell hooks.
5. Now maintained by Andy Lester.

3.2.6 The Test::Pod Module

Documentation in Perl need not be entirely unstructured. The Plain Old Documentation (POD) format for storing documentation in the Perl source code (see the *perlpod* manual page) is a markup language and therefore it is possible to commit syntax errors. So rather than wait until your users try to look at your documentation (okay, play along with me here—imagine that you *have* users who want to read your documentation), and get errors from their POD viewer, you can make sure in advance that the POD is good.

Test::Pod exports a single function, `pod_ok()`, which checks the POD in the file named by its argument. I'll show an example of its use later in this chapter.

3.2.7 Test::Inline

If you're thinking that tests deserve to be inside the code they're testing just as much as documentation does, then you want Test::Inline. This module by Michael Schwern enables you to embed tests in code just like POD, because, in fact, it uses POD for that embedding.

3.2.8 Test::NoWarnings

Fergal Daly's Test::NoWarnings (formerly Test::Warn::None) lets you verify that your code is free of warnings. In its simplest usage, you just use the module, and increment the number of tests you're running, because Test::NoWarnings adds one more. So if your test starts:

```
use Test::More tests => 17;
```

then change it to:

```
use Test::NoWarnings;
use Test::More tests => 18;
```

and the final test will be that no warnings were generated in the running of the other tests.

3.2.9 The Test::Harness Module

Test::Harness is how you combine multiple tests. It predates every other Test:: module, and you'll find it in every version of Perl 5. Test::Harness exports a function, `runtests()`, which runs all the test files whose names are passed to it as arguments and summarizes their results. You won't see one line printed per test; `runtests()` intercepts those lines of output. Rather you'll see one line printed per test *file*, followed by a summary of the results of the tests in that file. Then it prints a global summary line. Here's an example of the output:

```
t/01load....ok
t/02tie.....ok
t/03use.....ok
t/04pod.....ok
All tests successful.
Files=4, Tests=24,  2 wallclock secs ( 1.51 cusr +  0.31 csys =
1.82 CPU)
```

As it runs, before printing "ok" on each line, you'll see a count of the tests being run updating in place, finally to be overwritten by "ok". If any fail, you'll see something appropriate instead of "ok".

You can use Test::Harness quite easily, for instance:

```
% perl -MTest::Harness -e 'runtests(glob "*.t")'
```

but it's seldom necessary even to do that, because a standard Perl module makefile will do it for you. I'll show you how shortly.

Test::Harness turns your regression tests into a full-fledged deliverable. Managers just love to watch the numbers whizzing around.

3.3 An Example Using Test:: Modules

Let's put what we've learned to use in developing an actual application. Say that we want to create a module that can limit the possible indices of an array, a bounds checker if you will. Perl's arrays won't normally do that,[6] so we need a mechanism that intercepts the day-to-day activities of an array and checks the indices being used, throwing an exception if they're outside a specified range. Fortunately, such a mechanism exists in Perl; it's called *tieing,* and pretty powerful it is too.

Because our module will work by letting us tie an array to it, we'll call it Tie::Array::Bounded. We start by letting h2xs do the rote work of creating a new module:

```
% h2xs -AXn Tie::Array::Bounded
Writing Tie/Array/Bounded/Bounded.pm
Writing Tie/Array/Bounded/Makefile.PL
Writing Tie/Array/Bounded/README
Writing Tie/Array/Bounded/test.pl
Writing Tie/Array/Bounded/Changes
Writing Tie/Array/Bounded/MANIFEST
```

That saved a lot of time! h2xs comes with perl, so you already have it. Don't be put off by the name: h2xs was originally intended for creating perl extensions from C header files, a more or less obsolete purpose now, but by dint of copious interface extension, h2xs now enjoys a new lease on life for creating modules. (In Section 8.2.4, I'll look at a more modern alternative to h2xs.)

Don't be confused by the fact that the file Bounded.pm is in the directory Tie/Array/Bounded. It may look like there's an extra directory in there but the hierarchy that h2xs created is really just to help keep your sources straight. Everything you create will be in the bottom directory, so we could *cd* there. For instant gratification we can create a Makefile the way we would with any CPAN module:

6. If you're smart enough to bring up `$ [`, then you're also smart enough to know that you shouldn't be using it.

```
% cd Tie/Array/Bounded
% perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Tie::Array::Bounded
```

and now we can even run a test:

```
% make test
cp Bounded.pm blib/lib/Tie/Array/Bounded.pm
PERL_DL_NONLAZY=1 /usr/local/bin/perl -Iblib/arch -Iblib/lib -
I/usr/lib/perl5/5.6.1/i386-linux -I/usr/lib/perl5/5.6.1 test.pl
1..1
ok 1
```

It even passes! This is courtesy of the file test.pl that h2xs created for us, which contains a basic test that the module skeleton created by h2xs passes. This is very good for building our confidence. Unfortunately, test.pl is not the best way to create tests. We'll see why when we improve on it by moving test.pl into a subdirectory called "t" and rebuilding the Makefile before rerunning "make test":

```
% mkdir t
% mv test.pl t/01load.t
% perl Makefile.PL
Writing Makefile for Tie::Array::Bounded
% make test
PERL_DL_NONLAZY=1 /usr/local/bin/perl -Iblib/arch -Iblib/lib -
I/usr/lib/perl5/5.6.1/i386-linux -I/usr/lib/perl5/5.6.1 -e 'use
Test::Harness qw(&runtests $verbose); $verbose=0; runtests
@ARGV;' t/*.t
t/01load....ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs ( 0.30 cusr +  0.05 csys =
0.35 CPU)
```

The big difference: "make test" knows that it should run Test::Harness over the .t files in the t subdirectory, thereby giving us a summary of the results.

There's only one file in there at the moment, but we can create more if we want instead of having to pack every test into test.pl.

At this point you might want to update the MANIFEST file to remove the line for test.pl now that we have removed that file.

If you're using Perl 5.8.0 or later, then your h2xs has been modernized to create the test in t/1.t; furthermore, it will use Test::More.[7] But if you have a prior version of Perl, you'll find the test.pl file we just moved uses the deprecated Test module, so let's start from scratch and replace the contents of t/01load.t as follows:

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

use Test::More tests => 1;
use blib;
BEGIN { use_ok("Tie::Array::Bounded") }
```

The `use blib` statement causes Perl to search in parent directories for a blib directory that contains the Tie/Array/Bounded.pm module created by make. Although we'll usually run our tests by typing "make test" in the parent directory, this structure for a .t file allows us to run tests individually, which will be helpful when isolating failures.

Running this test either stand-alone ("./01load.t") or with "make test" produces the same output as before (plus a note from `use blib` about where it found the blib directory), so let's move on and add some code to Bounded.pm. First, delete some code that h2xs put there; we're not going to export anything, and our code will work on earlier versions of Perl 5, so remove code until the executable part of Bounded.pm looks like this:

7. I name my tests with two leading digits so that they will sort properly; I want to run them in a predictable order, and if I have more than nine tests, test 10.t would be run before test 2.t, because of the lexicographic sorting used by the `glob()` function called by "make test". Having done that, I can then add text after the digits so that I can also see what the tests are meant for, winding up with test names such as 01load.t.

```
package Tie::Array::Bounded;
use strict;
use warnings;
our $VERSION = '0.01';
1;
```

Now it's time to add subroutines to implement tieing. `tie` is how to make possessed variables with Perl: Literally anything can happen behind the scenes when the user does the most innocuous thing. A simple expression like `$world_peace++` could end up launching a wave of nuclear missiles, if `$world_peace` happens to be tied to Mutually::Assured::Destruction. (See, you can even use Perl to make covert political statements.)

We need a `TIEARRAY` subroutine; *perltie* tells us so. So let's add an empty one to Bounded.pm:

```
sub TIEARRAY
{
}
```

and add a test to look for it in 01load.t:

```
use Test::More tests => 2;
use blib;
BEGIN { use_ok("Tie::Array::Bounded") }

can_ok("Tie::Array::Bounded", "TIEARRAY");
```

Running "make test" copies the new Bounded.pm into blib and produces:

```
% make test
cp Bounded.pm blib/lib/Tie/Array/Bounded.pm
PERL_DL_NONLAZY=1 /usr/local/bin/perl -Iblib/arch -Iblib/lib -
I/usr/lib/perl5/5.6.1/i386-linux -I/usr/lib/perl5/5.6.1 -e 'use
Test::Harness qw(&runtests $verbose); $verbose=0; runtests
@ARGV;' t/*.t
t/01load....Using /home/peter/perl_Medic/Tie/Array/Bounded/blib
t/01load....ok
All tests successful.
```

```
Files=1, Tests=2,  0 wallclock secs ( 0.29 cusr +  0.02 csys =
0.31 CPU)
```

We have just doubled our number of regression tests!

It may seem as though we're taking ridiculously small steps here. A subroutine that doesn't do anything? What's the point in testing for that? Actually, the first time I ran that test, it failed: I had inadvertently gone into overwrite mode in the editor and made a typo in the routine name. The point in testing every little thing is to build your confidence in the code and catch even the dumbest errors right away.

So let's continue. We should decide on an interface for this module; let's say that when we tie an array we must specify an upper bound for the array indices, and optionally a lower bound. If the user employs an index out of this range, the program will die. For the sake of having small test files, we'll create a new one for this test and call it 02tie.t:

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

use Test::More tests => 1;
use blib;
use Tie::Array::Bounded;

my $obj = tie my @array, "Tie::Array::Bounded";
isa_ok($obj, "Tie::Array::Bounded");
```

So far, this just tests that the underlying object from the `tie` is or inherits from Tie::Array::Bounded. Run this test *before* you even add any code to `TIEARRAY` to make sure that it does indeed *fail:*

```
% ./02tie.t
1..1
Using /home/peter/perl_Medic/Tie/Array/Bounded/t/../blib
not ok 1 - The object isa Tie::Array::Bounded
#     Failed test (./02tie.t at line 10)
```

```
#     The object isn't defined
# Looks like you failed 1 tests of 1.
```

We're not checking that the module can be used or that it has a TIEARRAY method; we already did those things in 01load.t. Now we know that the test routine is working properly. Let's make a near-minimal version of TIEARRAY that will satisfy this test:

```
sub TIEARRAY
{
  my $class = shift;
  my ($upper, $lower);
  return bless { upper => $upper,
                 lower => $lower,
                 array => []
               }, $class;
}
```

Now the test passes. Should we test that the object is a hashref with keys upper, lower, and so on? No—that's part of the private implementation of the object and users, including tests, have no right peeking in there.

Well, it doesn't really do to have a bounded array type if the user doesn't specify any bounds. A default lower bound of 0 is obvious because most bounded arrays will start from there anyway and be limited in how many elements they can contain. It doesn't make sense to have a default upper bound because no guess could be better than any other. We want this module to die if the user doesn't specify an upper bound (italicized code):

```
sub TIEARRAY
{
  my ($class, %arg) = @_;
  my ($upper, $lower) = @arg{qw(upper lower)};
  $lower ||= 0;
  croak "No upper bound for array" unless $upper;
  return bless { upper => $upper,
                 lower => $lower,
                 array => []
               }, $class;
}
```

Note that when we want to die in a module, the proper routine to use is `croak()`. This results in an error message that identifies the calling line of the code, and not the current line, as the source of the error. This allows the user to locate the place in their program where they made a mistake. `croak()` comes from the Carp Module, so we added a `use Carp` statement to Bounded.pm (not shown).

Note also that we set the lower bound to a default of 0. True, if the user didn't specify a lower bound, `$lower` would be undefined and hence evaluate to 0 in a numeric context. But it's wise to expose our defaults explicitly, and this also avoids warnings about using an uninitialized value. Modify 02tie.t to say:

```
use Test::More tests => 1;
use Test::Exception;
use blib;
use Tie::Array::Bounded;

dies_ok { tie my @array, "Tie::Array::Bounded" }
        "Croak with no bound specified";
```

If you're running 02tie.t as a stand-alone test, remember to run *make* in the parent directory after modifying Bounded.pm so that Bounded.pm gets copied into the blib tree.

Great! Now let's add back in the test that we can create a real object when we `tie` with the proper calling sequence:

```
my $obj;
lives_ok { $obj = tie my @array, "Tie::Array::Bounded",
           upper => 42
         } "Tied array okay";
isa_ok($obj, "Tie::Array::Bounded");
```

and increase the number of tests to 3. (Notice that there is no comma after the block of code that's the first argument to `dies_ok` and `lives_ok`.)

All this testing has gotten us in a pedantic frame of mind. The user shouldn't be allowed to specify an array bound that is negative or not an integer. Let's add a statement to TIEARRAY (in italics):

```
sub TIEARRAY
{
  my ($class, %arg) = @_;
  my ($upper, $lower) = @arg{qw(upper lower)};
  $lower ||= 0;
  croak "No upper bound for array" unless $upper;
  /\D/ and croak "Array bound must be integer"
    for ($upper, $lower);
  return bless { upper => $upper,
                 lower => $lower,
                 array => []
               }, $class;
}
```

and, of course, test it:

```
throws_ok { tie my @array, "Tie::Array::Bounded", upper => -1 }
          qr/must be integer/, "Non-integral bound fails";
```

Now we're not only checking that the code dies, but that it dies with a message matching a particular pattern.

We're really on a roll here! Why don't we batten down the hatches on this interface and let the user know if they gave us an argument we're *not* expecting:

```
sub TIEARRAY
{
  my ($class, %arg) = @_;
  my ($upper, $lower) = delete @arg{qw(upper lower)};
  croak "Illegal arguments in tie" if %arg;
  croak "No upper bound for array" unless $upper;
  $lower ||= 0;
  /\D/ and croak "Array bound must be integer"
    for ($upper, $lower);
```

```
  return bless { upper => $upper,
                 lower => $lower,
                 array => []
               }, $class;
}
```

and the test:

```
throws_ok { tie my @array, "Tie::Array::Bounded", frogs => 10 }
          qr/Illegal arguments/, "Illegal argument fails";
```

The succinctness of our approach depends on the underappreciated *hash slice* and the `delete()` function. Hash slices [GUTTMAN98] are a way to get multiple elements from a hash with a single expression, and the `delete()` function removes those elements while returning their values. Therefore, anything left in the hash must be illegal.

We're nearly done with the pickiness. There's one final test we should apply. Have you guessed what it is? We should make sure that the user doesn't enter a lower bound that's higher than the upper one. Can you imagine what the implementation of bounded arrays would do if we didn't check for this? I can't, because I haven't written it yet, but it might be ugly. Let's head that off at the pass right now:

```
sub TIEARRAY
{
  my ($class, %arg) = @_;
  my ($upper, $lower) = delete @arg{qw(upper lower)};
  croak "Illegal arguments in tie" if %arg;
  $lower ||= 0;
  croak "No upper bound for array" unless $upper;
  /\D/ and croak "Array bound must be integer"
    for ($upper, $lower);
  croak "Upper bound < lower bound" if $upper < $lower;
  return bless { upper => $upper,
                 lower => $lower,
                 array => []
               }, $class;
}
```

and the new test goes at the end of 02tie.t (italicized):

Example 3.2 Final Version of 02tie.t

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

use Test::More tests => 6;
use Test::Exception;
use blib;
use Tie::Array::Bounded;

dies_ok { tie my @array, "Tie::Array::Bounded" }
        "Croak with no bound specified";

my $obj;
lives_ok { $obj = tie my @array, "Tie::Array::Bounded",
         upper => 42 }
         "Tied array okay";

isa_ok($obj, "Tie::Array::Bounded");

throws_ok { tie my @array, "Tie::Array::Bounded", upper => -1 }
          qr/must be integer/, "Non-integral bound fails";

throws_ok { tie my @array, "Tie::Array::Bounded", frogs => 10 }
          qr/Illegal arguments/, "Illegal argument fails";

throws_ok { tie my @array, "Tie::Array::Bounded",
            lower => 2, upper => 1 }
          qr/Upper bound < lower/, "Wrong bound order fails";
```

Whoopee! We're nearly there. Now we need to make the tied array behave properly, so let's start a new test file for that, called 03use.t:

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

```
use Test::More tests => 1;
use Test::Exception;
use blib;
use Tie::Array::Bounded;

my @array;
tie @array, "Tie::Array::Bounded", upper => 5;

lives_ok { $array[0] = 42 } "Store works";
```

As before, let's ensure that the test fails before we add the code to implement it:

```
% t/03use.t
1..1
Using /home/peter/perl_Medic/Tie/Array/Bounded/blib
not ok 1 - Store works
#     Failed test (t/03use.t at line 13)
# died: Can't locate object method "STORE" via package
"Tie::Array::Bounded" (perhaps you forgot to load
"Tie::Array::Bounded"?) at t/03use.t line 13.
# Looks like you failed 1 tests of 1.
```

How about that. The test even told us what routine we need to write. *perltie* tells us what it should do. So let's add to Bounded.pm:

```
sub STORE
{
  my ($self, $index, $value) = @_;
  $self->_bound_check($index);
  $self->{array}[$index] = $value;
}

sub _bound_check
{
  my ($self, $index) = @_;
  my ($upper, $lower) = @{$self}{qw(upper lower)};
  croak "Index $index out of range [$lower, $upper]"
    if $index < $lower || $index > $upper;
}
```

We've abstracted the bounds checking into a method of its own in anticipation of needing it again. Now 03use.t passes, and we can add another test to make sure that the value we stored in the array can be retrieved:

```
is($array[0], 42, "Fetch works");
```

You might think this would fail for want of the FETCH method, but in fact:

```
ok 1 - Store works
Can't locate object method "FETCHSIZE" via package
"Tie::Array::Bounded" (perhaps you forgot to load
"Tie::Array::Bounded"?) at t/03use.t line 14.
# Looks like you planned 2 tests but only ran 1.
# Looks like your test died just after 1.
```

Back to *perltie* to find out what FETCHSIZE is supposed to do: return the size of the array. Easy enough:

```
sub FETCHSIZE
{
  my $self = shift;
  scalar @{$self->{array}};
}
```

Now the test does indeed fail for want of FETCH, so we'll add that:

```
sub FETCH
{
  my ($self, $index) = @_;
  $self->_bound_check($index);
  $self->{array}[$index];
}
```

Finally we are back in the anodyne land of complete test success. Time to add more tests:

```
throws_ok { $array[6] = "dog" } qr/out of range/,
          "Bounds exception";
is_deeply(\@array, [ 42 ], "Array contents correct");
```

These work immediately. But an ugly truth emerges when we try another simple array operation:

```
lives_ok { push @array, 17 } "Push works";
```

This results in:

```
not ok 5 - Push works
#     Failed test (t/03use.t at line 19)
# died: Can't locate object method "PUSH" via package
"Tie::Array::Bounded" (perhaps you forgot to load
"Tie::Array::Bounded"?) at t/03use.t line 19.
# Looks like you failed 1 tests of 5.
```

Inspecting *perltie* reveals that `PUSH` is one of several methods it looks like we're going to have to write. Do we really have to write them all? Can't we be lazier than that?

Yes, we can.[8] The Tie::Array core module defines `PUSH` and friends in terms of a handful of methods we have to write: `FETCH`, `STORE`, `FETCHSIZE`, and `STORESIZE`. The only one we haven't done yet is `STORESIZE`:

```
sub STORESIZE
{
  my ($self, $size) = @_;
  $self->_bound_check($size-1);
  $#{$self->{array}} = $size - 1;
}
```

We need to add near the top of Bounded.pm:

8. Remember, if you find yourself doing something too rote or boring, look for a way to get the computer to make it easier for you. Top of the list of those ways would be finding code someone else already wrote to solve the problem.

```
use base qw(Tie::Array);
```

to inherit all that array method goodness.

This is a big step to take, and if we didn't have canned tests, we might wonder what sort of unknown havoc could be wrought upon our module by a new base class if we misused it. However, our test suite allows us to determine that, in fact, nothing has broken.

Now we can add to 01load.t the methods `FETCH`, `STORE`, `FETCHSIZE`, and `STORESIZE` in the `can_ok` test:

Example 3.3 Final Version of 01load.t

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

use Test::More tests => 2;
use blib;
BEGIN { use_ok("Tie::Array::Bounded") }

can_ok("Tie::Array::Bounded", qw(TIEARRAY STORE FETCH STORESIZE
                                 FETCHSIZE));
```

Because our tests pass, let's add as many more as we can to test all the boundary conditions we can think of, leaving us with a final 03use.t file of:

Example 3.4 Final Version of 03use.t

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

use Test::More tests => 15;
use Test::Exception;
use blib;
use Tie::Array::Bounded;
```

```
my $RANGE_EXCEP = qr/out of range/;

my @array;
tie @array, "Tie::Array::Bounded", upper => 5;
lives_ok { $array[0] = 42 } "Store works";
is($array[0], 42, "Fetch works");

throws_ok { $array[6] = "dog" } $RANGE_EXCEP,
          "Bounds exception";
is_deeply(\@array, [ 42 ], "Array contents correct");

lives_ok { push @array, 17 } "Push works";
is($array[1], 17, "Second array element correct");

lives_ok { push @array, 2, 3 } "Push multiple elements works";
is_deeply(\@array, [ 42, 17, 2, 3 ], "Array contents correct");

lives_ok { splice(@array, 4, 0, qw(apple banana)) }
         "Splice works";
is_deeply(\@array, [ 42, 17, 2, 3, 'apple', 'banana' ],
          "Array contents correct");

throws_ok { push @array, "excessive" } $RANGE_EXCEP,
          "Push bounds exception";
is(scalar @array, 6, "Size of array correct");

tie @array, "Tie::Array::Bounded", lower => 3, upper => 6;

throws_ok { $array[1] = "too small" } $RANGE_EXCEP,
          "Lower bound check failure";

lives_ok { @array[3..6] = 3..6 } "Slice assignment works";
throws_ok { push @array, "too big" } $RANGE_EXCEP,
          "Push bounds exception";
```

Tests are real programs, too. Because we test for the same exception repeatedly, we put its recognition pattern in a variable to be lazy.

Bounded.pm, although not exactly a model of efficiency (our internal array contains unnecessary space allocated to the first `$lower` elements that will never be used), is now due for documenting, and h2xs filled out some POD

stubs already. We'll flesh it out to the final version you can see in the Appendix. I'll go into documentation more in Chapter 10.

Now we create 04pod.t to test that the POD is formatted correctly:

Example 3.5 Final Version of 04pod.t

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

use Test::Pod tests => 1;
use blib;
use Tie::Array::Bounded;
pod_ok($INC{"Tie/Array/Bounded.pm"});
```

There's just a little trick there to allow us to run this test from any directory, since all the others can be run with the current working directory set to either the parent directory or the t directory. We load the module itself and then get Perl to tell us where it found the file by looking it up in the `%INC` hash, which tracks such things (see its entry in *perlvar*).

With a final "make test", we're done:

```
Files=4, Tests=24,  2 wallclock secs ( 1.58 cusr +  0.24 csys =
1.82 CPU)
```

We have a whole 24 tests at our fingertips ready to be repeated any time we want.

You can get more help on how to use these modules from the module Test::Tutorial, which despite the module appellation contains no code, only documentation.

With only a bit more work, this module could have been submitted to CPAN. See [TREGAR02] for full instructions.

3.4 Testing Legacy Code

"This is all well and good," I can hear you say, "but I just inherited a swamp of 27 programs and 14 modules and they have no tests. What do I do?"

By now you've learned that it is far more appealing to write tests as you write the code they test, so if you can possibly rewrite this application, do so. But if you're stuck with having to tweak an existing application, then adopt a top-down approach. Start by testing that the application meets its requirements . . . assuming you were given requirements or can figure out what they were. See what a successful run of the program outputs and how it may have changed its environment, then write tests that look for those effects.

3.4.1 A Simple Example

You have an inventory control program for an aquarium, and it produces output files called cetaceans.txt, crustaceans.txt, molluscs.txt, pinnipeds.txt, and so on. Capture the output files from a successful run and put them in a subdirectory called success. Then run this test:

Example 3.6 Demonstration of Testing Program Output

```
1  my @Success_files;
2  BEGIN {
3    @Success_files = glob "success/*.txt";
4  }
5
6  use Test::More tests => 1 + 2 * @Success_files;
7
8  is(system("aquarium"), 0, "Program succeeded");
9
10 for my $success (@Success_files)
11 {
12   (my $output = $success) =~ s#.*/##;
13
14   ok(-e $output, "$output present");
15
```

```
16    is(system("cmp $output $success > /dev/null 2>&1"),
17       0, "$output is valid");
18  }
```

First, we capture the names of the output files in the success subdirectory. We do that in a BEGIN block so that the number of names is available in line 6. In line 8 we run the program and check that it has a successful return code. Then for each of the required output files, in line 14 we test that it is present, and in line 16 we use the UNIX *cmp* utility to check that it matches the saved version. If you don't have a *cmp* program, you can write a Perl subroutine to perform the same test: Just read each file and compare chunks of input until finding a mismatch or hitting the ends of file.

3.4.2 Testing Web Applications

A Common Gateway Interface (CGI) program that hasn't been developed with a view toward automated testing may be a solid block of congealed code with pieces of web interface functionality sprinkled throughout it like raisins in a fruit cake. But you don't need to rip it apart to write a test for it; you can verify that it meets its requirements with an end-to-end test. All you need is a program that pretends to be a user at a web browser and checks that the response to input is correct. It doesn't matter how the CGI program is written because all the testing takes place on a different machine from the one the CGI program is stored on.

The WWW::Mechanize module by Andy Lester comes to your rescue here. It allows you to automate web site interaction by pretending to be a web browser, a function ably pulled off by Gisle Aas' LWP::UserAgent module. WWW::Mechanize goes several steps farther, however (in fact, it is a subclass of LWP::UserAgent), enabling cookie handling by default and providing methods for following hyperlinks and submitting forms easily, including transparent handling of hidden fields.[9]

9. If you're thinking, "Hey! I could use this to write an agent that will stuff the ballot box on surveys I want to fix," forget it; it's been done before. Chris Nandor used Perl to cast thousands of votes for his choice for American League All-Star shortstop [GLOBE99]. And this was before WWW::Mechanize was even invented.

Suppose we have an application that provides a login screen. For the usual obscure reasons, the login form, login.html, contains one or more hidden fields in addition to the user-visible input fields, like this:

```
<FORM ACTION="login.cgi" METHOD="POST">
 <INPUT NAME="username" TYPE="text">
 <INPUT NAME="password" TYPE="text">
 <INPUT NAME="fruglido" TYPE="hidden" VALUE="grilku">
 <INPUT TYPE="Submit">
</FORM>
```

On successful login, the response page greets the user with "Welcome, " followed by the user's first name. We can write this test for this login function:

Example 3.7 Using WWW::Mechanize to Test a Web Application

```
1  #!/usr/bin/perl
2  use strict;
3  use warnings;
4
5  use WWW::Mechanize;
6  use Test::More tests => 3;
7
8  my $URL = 'http://localhost/login.html';
9  my $USERNAME = 'peter';
10 my $PASSWORD = 'secret';
11
12 my $ua = WWW::Mechanize->new;
13 ok($ua->get($URL)->is_success, "Got first page")
14   or die $ua->res->message;
15
16 $ua->set_fields(username => $USERNAME,
17                 password => $PASSWORD);
18 ok($ua->submit->is_success, "Submitted form")
19   or die $ua->res->message;
20
21 like($ua->content, qr/Welcome, Peter/, "Logged in okay");
```

In line 12 we create a new WWW::Mechanize user agent to act as a pretend browser, and in line 13 we test to see if it was able to get the login page; the `get()` method returns a HTTP::Response object that has an `is_success()` method. If something went wrong with fetching the page the false value will be passed through the `ok()` function; there's no point in going further so we might as well `die()` (line 14). We can get at the HTTP::Response object again via the `res()` method of the user agent to call its `message()` method, which returns the text of the reason for failure.

In lines 16 and 17 we provide the form inputs by name, and in line 18 the `submit()` method of the user agent submits the form and reads the response, again returning an HTTP::Response object allowing us to verify success as before. Once we have a response page we check to see whether it looks like what we wanted.

Note that WWW::Mechanize can be used to test interaction with any web application, regardless of where that application is running or what it is written in.

3.4.3 What Next?

The kind of end-to-end testing we have been doing is useful and necessary; it is also a lot easier than the next step. To construct comprehensive tests for a large package, we must include unit tests; that means testing each function and method. However, unless we have descriptions of what each subroutine does, we won't know how to test them without investigative work to find out what they are supposed to do. I'll go into those kinds of techniques later.

3.5 A Final Encouragement

In addition to the more obvious benefits, constructing tests before or contemporaneously with code development encourages good program interface design.

Take, for example, a web-based system incorporating CGI programs and an extensive back end. When writing tests for such a beast it will become rapidly apparent that testing it via the CGI interface is tedious at best. You have to wait for the whole server round trip to happen, and most of that time is occupied by the operation of software and networks you may not be responsible for and don't want to test. By cutting out the fat and calling the back end directly you'll eliminate the tedium. However, you don't want to leave out interface code that should be tested. So you make the CGI programs as small as possible: Gather user inputs, pass them on to interface-independent code, take the outputs of that code and format them for the user. That way you have so little code in the CGI programs that testing them will be a snap. A single call to a CGI program itself will accomplish that, and every other test can concentrate on going directly to the back end.

Congratulations. You've just reinvented the Model-View-Controller pattern (see [GAMMA95]), a device generally recognized to be a pretty good thing.

3.5.1 A Final Caveat

Tests can't replace using your brain. They're only as smart as their creator: If there's a bug that isn't tested for, the tests won't find it. You still have to look at what you write and think about it, or it could harbor a bug that you didn't think to test for. Testing just saves you from having to repeat the same train of thought over and over again.