Testing and Documenting Perl Code

Scott Wiersdorf
NTT/Verio

NTT/VERIO

Why Developers Should Test and Document Their Code

or "Why Something is Better Than Nothing"

NTT/VERIO

Assertion

I want to convince you that just as building a house without plans is cobbling, not engineering, writing software without knowing in advance how the software should work is also cobbling and not software engineering.

This is not to say that you must know all of the methods or even how the software will precisely work at the beginning of the project. Because of the complex nature of software writing, we do it non-linearly, which allows us to discover new things and make adjustments as we go.

Writing tests and documentation (which is what this presentation is about) are essential to the discovery process. By writing tests and documentation before you begin coding, you are creating plans for your code. Tests and documentation are the architectural plans for a piece of software, without which the engineering process is simply an exercise in trial-and-error and the software itself is highly fragile and likely poorly-designed. Testing and documenting before coding allows you to achieve a more succinct, stable, and readable API than might otherwise be realized.

In addition to providing plans for new code, tests and documentation provide a reference for existing code. Regression tests give you as the programmer a tool that can tell you almost instantly if any refactoring you've done has broken something. This lets you go back and clean up that section of code you've got marked as 'FIXME' that you've been meaning to go back and fix for a year but haven't had time. Without regression tests, the FIXME's stay forever.

Roadmap: Testing and Documenting Code

- About Me (briefly)
- Guilt Trip: Why Not Test/Document
- A Testing Example
- Fleshing Out Our Example
  - Tests
  - Documentation
  - Writing Code
  - Future Improvements
- The Take-away

What you should know about me

- I am not a testing expert!
- I am a programmer
- I do not always write tests or documentation for all my code
- I *do* write tests and documentation for *nearly* all of my code
- I have (good) intentions to go back when it's time to refactor old code and write tests and documentation for them, too

Why Not Testing

Roadmap: Why Not Testing

- Ignorance
- Someone Else's Problem
- Looming deadline/short on time
- Extra work
- Testing is for weenies!

Why Not Test: Ignorance

- I'm supposed to test?
  - Yes!
  - Software engineering has come a long way since "top-down modular design" was innovative
  - Testing is less effort than debugging
  - Learning to test your code will make you a more portable programmer
  - You'll have more friends if you test your code
- I don't make mistakes (denial)
  - Yeah, right.
  - You write your programs once...and then write them all over again when a change needs to be made (you're a real hard worker)
  - Meskimen's Law: There's never time to do it right, but there's always time to do it over.
- Work smarter, not harder: regression tests are your friends

Why Not Test: Ignorance (cont'd)

- I think someone else in our department tests (see next slide)
  - Could be. After all, what are users for?
  - The best unit tests are done by the developer who wrote the unit
- I don't know how to set up tests
  - That's why you're paying careful attention!
  - We'll cover that later

Why Not Test: Someone Else's Problem (SEP)

- We already have a department that does that
- Ok, that's a valid point
- But here are a few more valid points:
  - It's also your problem because it's your code
  - No one knows the code like you do
  - No one cares about your code like you do
    - If you don't care about your code as you write it, you are a curse to your employer
    - You won't care much about any of your code after a couple of weeks anyway
- Point: the best time to write tests is before and while you are coding, not after (though after is better than nothing). You're the best person to do that

Why Not Test: Looming Deadline

- "We try to solve the problem by rushing through the design process so that enough time will be left at the end of the project to uncover errors that were made because we rushed through the design process." --Glenford Myers
- This also applies to testing
- The best time to write tests is at the beginning of a project
- You'll save time either at the beginning of the project (by not writing tests) or at the end of the project (less debugging)
- With regression tests, you will save time at the end of and throughout the project
- You will have **more** provably correct code sooner if you test
- You will save oodles of time in the future with regression tests
- There really aren't many good time-based arguments for not writing tests

Why Not Test: Extra Work

- You will always be afraid of touching a finished, working piece of code until you have tests
- Fourth Law of Code: Code that has no regression or unit tests is either dead code or very fragile code that breaks at inconvenient times (doubly so for production code)
- Corollary: Working code with a complete suite of regression and unit tests is living, growing code and will be useful to many people over a longer period of time and will tolerate a high degree of change and internal reworking
- You will never have real confidence in your code until you have tests
- Your software will never be world-class quality until you have tests (almost all CPAN modules have tests and many CPAN modules have very thorough tests)

Why Not Test: Only Weenies Test!

- Some weenies:
  - Larry Wall
  - and a cast of thousands, including
  - Tom Christiansen
  - Gurusamy Sarathy
  - Damian Conway
  - Simon Cozens
  - and more: consider CPAN
- Don't you wish you were a weenie?

Why Not Documentation

Roadmap: Why Not Documentation

- Same reasons as testing
  - Ignorance
  - Someone Else's Problem (documentation team)
  - Looming deadline/short on time
  - Extra work
  - Documenting is for weenies!
- We'll go through this quickly

Why Not Document: Ignorance

- What good is an undocumented module?
- No one can use it except you
- Do you really want to own this module for eternity?
- By documenting your module, you free the module to be owned by someone else
- You also free yourself from having to explain to others how it works

Why Not Document: Someone Else's Problem

- Isn't this why we have a documentation team?
- Can they understand your code?
- (No, they can't. For the love of Pete, have a heart and document your code)
- (And another thing... if documenting your code is not part of your job description, it should be. Generally speaking, tech writers are not programmers. To think they should document your code is evidence of misunderstanding their function. -B. Heaton)
- You really are the only one who knows how this code works--if you document it, then someone else will understand it too (and you will be free!)
- If you're really a poor writer, do the best you can and let the doc team clean it up later
- The doc team **loves it** when developers document their code

Why Not Document: Looming Deadline

- I'm short on time--I'm not going to document
- Good point: given a choice between documenting and coding, choose coding
- This is why we document before we code!
- It will shorten the amount of time you spend coding by more than the time spent documenting
- You will be more productive (only anecdotal evidence to support this, but I believe in it)

Why Not Document: Extra Work

- You have fallen victim to *false laziness*
- False laziness says that less work up front yields less total work in the long run
- True laziness says that more work testing and documenting up front yields less total work in the long run
- If you document your code, you'll spend less time in the following areas:
  - explaining how your code works to others (this lasts as long as the module and is the #1 benefit in my book)
  - rewriting code to get the API right
  - "We try to solve the problem by rushing through the design process so that enough time will be left at the end of the project to uncover errors that were made because we rushed through the design process." --Glenford Myers
  - time spent documenting is time spent designing and is well spent mental exertion

Why Not Document: Extra Work (cont'd)

- Substanitated Rumor: People who document well also think well
- Writing solidifies nebulous thoughts: "How can I know what I think till I see what I say?" --E. M. Forster
- If you take the time to document, you'll resolve most of the really hard logic problems and interface issues early, saving you labor later

Why Not Document: Only Weenies Document

- Some more weenies:
  - Larry Wall
  - and a cast of thousands
  - and more: consider CPAN

A Testing Example

Roadmap: An Example of Writing Tests

- Creating a module
- Editing test.pl
- Running tests
- Fixing mistakes

Creating a Module

- Many ways to create a module
- Some more popular than others
- We're going to use 'h2xs' because it's easy and found everywhere

Creating a Module (cont'd)

- create a module:

```
h2xs -AX Foo::Bar
```

- This will create the following hierarchy:

```
Foo/Bar/Bar.pm
Foo/Bar/Changes
Foo/Bar/MANIFEST
Foo/Bar/Makefile.PL
Foo/Bar/test.pl
```

- Notice the 'test.pl' file

Example 1: Create a module

- Goal: a module that can parse an Apache log file and give summary information in the form of an object
- Start with h2xs:

```
h2xs -AX Log::Apache::Object
Writing Log/Apache/Object/Object.pm
Writing Log/Apache/Object/Makefile.PL
Writing Log/Apache/Object/test.pl
Writing Log/Apache/Object/Changes
Writing Log/Apache/Object/MANIFEST
```

- The first thing we do is edit 'test.pl'
- Depending on your version of Perl, you'll see varying amounts of stuff cluttering your file
- We'll cover what's in 'test.pl' next

Example 1: sample test.pl

- A sample test.pl: this much will be created for you

```
############################

# change 'tests => 1' to 'tests => last_test_to_print';

use Test;
BEGIN { plan tests => 1 };
use Log::Apache::Object;
ok(1); # If we made it this far, we're ok.

############################
```

Example 1: Edit test.pl

- now we hack on test.pl
- we write one or two tests for each "feature" we're about to add

```
use Test;
BEGIN { plan tests => 1 };
use Log::Apache::Object;
ok(1); # If we made it this far, we're ok.

## first test
my $obj;
ok( $obj = new Log::Apache::Object );
```

Example 1: Running test.pl

- tests are run automatically when we type 'make test'

```
% perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Log::Apache::Object
% make
cp Object.pm blib/lib/Log/Apache/Object.pm
Manifying blib/man3/Log::Apache::Object.3pm
% make test
PERL_DL_NONLAZY=1 /usr/bin/perl -Iblib/arch -Iblib/lib \
-I/usr/local/lib/perl5/5.6.1/i386-freebsd \
-I/usr/local/lib/perl5/5.6.1 test.pl
1..1
ok 1
Can't locate object method "new" via package \
"Log::Apache::Object" (perhaps you forgot to load \
"Log::Apache::Object"?) at test.pl line 7.
*** Error code 255

Stop in /usr/home/scottw/testing/Log/Apache/Object.
```

- we forgot to update the number of tests
- we haven't written a "new" method yet

Example 1: Fixing Mistakes

- Let's update the tests:

```
BEGIN { plan tests => 2 };
```

- and write a 'new' method:

```
sub new {
  my $self  = { };
  my $proto = shift;
  my $class = ref($proto) || $proto;
  bless $self, $class;

  return $self;
}
```

- and run the tests again

```
% make test
cp Object.pm blib/lib/Log/Apache/Object.pm
PERL_DL_NONLAZY=1 /usr/bin/perl -Iblib/arch -Iblib/lib test.pl
1..2
ok 1
ok 2
```

- Success!

Wasn't That Satisfying?

- We now have a module that works and we can prove it
- We can get instant feedback when we make changes
- We'll never have to write a test for the constructor again
- ...unless we add/change functionality in the constructor
- We can be **bold** in making wide-impact changes such as refactoring entire methods or changing global defaults, etc.

Fleshing Out Our Example

Roadmap: More Tests, Features and Documentation

- Documenting What We're About To Do
- Adding New Functionality
- Writing Code
- What We Just Did
- Future Improvements

Documenting What We're About To Do

- What good is a module with no documentation?
- Documenting is easy with POD (it's already there)
- Documenting before coding also helps solidify the API in the same way testing does, but even better because putting abstract concepts into concrete language (e.g., English) forces your brain to conceptualize it; you quickly find ambiguities and contradictions

Documenting: SYNOPSIS

```
=head1 SYNOPSIS

  use Log::Apache::Object;
  my $obj =
    new Log::Apache::Object(file => '/www/logs/access_log');

  $obj->parse();
  print "I have had " . $obj->count(status => 404) .
    " errors since the log was rotated\n";
```

Documenting: DESCRIPTION

```
=head1 DESCRIPTION

B<Log::Apache::Object> is an OO class for parsing Apache log
files.  The following methods are available:

=over 4

=item B<new>

Creates a new B<Log::Apache::Object>. Valid arguments are
key/value pairs:

    file => "filename.log"

=item B<init>

Initializes the object; this is called when the object is
instantiated, but may be called any time thereafter to
re-initialize the object (e.g., with new data). Takes the same
arguments as B<new>.
```

Documenting: DESCRIPTION (cont'd)

```
=item B<file>

Sets/returns the current log file to process.

Example:

    $obj = new Log::Apache::Object;
    $obj->file('/www/logs/access_log');

    print "Working with log " . $obj->log . "\n";
```

Documenting: DESCRIPTION (cont'd)

```
=item B<parse>

Parses the log file and stores the log data internally to the
object.  Future method calls will access this storage so the
log will not have to be re-parsed.

Example:

    $obj->parse();

=item B<count([$field [, $value]])>

If no argument is specified, returns the number of lines in
the log.  If B<$field> is specified, the total
number of unique keys in that field will be returned (e.g.,
'host' will return unique hosts in the log). If B<$value>
is specified, the number of entries in B<$field> for
B<$value> will be returned.

Example:

    print "I have " . $obj->count . " lines\n";
    print "I have " . $obj->count('host') . " unique hosts\n";
    print "10.0.11.1 made " . $obj->count('host' => '10.0.11.1')
      " requests\n";
=back
```

Documenting: BUGS

- BUGS gives us a public "to do" list (nobody wants to admit to having bugs)

```
=head1 BUGS

=over 4

=item *

Only parses Apache combined log format

=item *

Stores the entire file in memory

=item *

Does not cache data outside of the object (re-parse each time)

=back
```

What We Just Did: Documentation

- We wrote the documentation before the code
  - This helped us solidify the API before we wrote it
  - Now our module is completely documented (to date)
- When we do 'make install', it will "manify" the POD and make a real manpage
- We did a good enough job in the docs for somebody not familiar with the code to use it
  - This is the Perl Virtue of *laziness*
  - We don't have to answer any questions about our code
  - We just say, "RTFM"

Admission of Guilt

- I didn't write all of the documentation before I began coding
- 'init' and 'file' were after-thoughts
- But I documented them after realizing I needed them
- Documenting, testing, coding is not linear but recursive and sometimes random: that's ok (as long as it gets done)

Adding New Functionality in test.pl

- Let's edit our test.pl file (remember to update test count)

```
## create a file
open TMP, ">.tmp_$$" or die "$!\n";
print TMP <<'_FILE_';
10.1.1.11 - - [24/Mar/2003:08:25:10 -0700] "GET /nonexistent HTTP/1.1" \
404 300 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; (R1 1.3))"
10.1.1.11 - - [24/Mar/2003:08:25:15 -0700] "GET /nonexistentential HTTP/1.1" \
404 300 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; (R1 1.3))"
10.1.1.11 - - [24/Mar/2003:08:25:24 -0700] "GET /foo/bar.pl HTTP/1.1" \
200 9328 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; (R1 1.3))"
_FILE_
close TMP;

## now further tests
undef $obj;
ok( $obj = new Log::Apache::Object(file => ".tmp_$$") );
ok( $obj->parse() );
ok( $obj->count, 3 );
ok( $obj->count('status'), 2 );  ## two 404's and one 200
ok( $obj->count('status' => '404'), 2 );
ok( $obj->count('host'   => '10.1.1.11'), 3 );

## clean up
END { unlink ".tmp_$$" }
```

What We Just Did: Testing

- We wrote the tests before the code
  - This helped us clarify the API before we wrote it
  - Our test interface also serves as a mock-up or reference for development on the code
  - Tests help us gauge our progress on the actual module
- Writing tests is easy and fast!
- ...usually.
- Writing good tests is always satisfying and productive

Writing Code: 'new' and 'init'

- Now let's write some code!

```
our %agg   = ();
our @lines = ();

sub new {
    my $self  = { };
    my $proto = shift;
    my $class = ref($proto) || $proto;
    bless $self, $class;

    $self->init(@_);

    return $self;
}

sub init {
    my $self = shift;
    my %args = @_;

    $self->file($args{'file'});
}
```

Writing Code: 'file' method

- This could also be done with $AUTOLOAD

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

    return $self->{'_file'} = ( defined $file
                                ? $file
                                : $self->{'_file'} );
}
```

Writing Code: 'parse' method

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

    my %data  = ();
    die "Must specify file to parse\n"
        unless $self->file;

    open FILE, $self->file
        or die "Could not open " . $self->file . ": $!\n";

    while( <FILE> ) {
        chomp;
        next unless ($data{host}, $data{user}, $data{timestamp}, $data{method},
                     $data{uri}, $data{protocol}, $data{status}, $data{bytes},
                     $data{referer}, $data{agent}) = m!
             ^(\S+)\s+\S+\s+            ## host, logname (not captured)
             (\S+)\s+\[(.+?)\]\s+       ## user, timestamp
             "(\S+)\s+(\S+)\s+(\S+)"\s+ ## request URI, method, protocol
             (\d+)\s+(\d+)\s+           ## status, bytes sent
             "(.+?)"\s+"(.+?)"$!x;      ## referer, agent

        ## some aggregate data
        $agg{host}->{$data{host}}++;      $agg{user}->{$data{user}}++;
        $agg{uri}->{$data{uri}}++;        $agg{protocol}->{$data{protocol}}++;
        $agg{status}->{$data{status}}++;  $agg{referer}->{$data{referer}}++;
        $agg{agent}->{$data{agent}}++;

        ## store this line
        push @lines, \%data;
    }
    close FILE;
}
```

NTT/VERIO

Writing Code: 'count' method

```
sub count {
    my $self  = shift;
    my $field = shift;
    my $value = shift;

    return scalar @lines unless $field;
    return scalar keys %{$agg{$field}} unless $value;
    return $agg{$field}->{$value};
}
```

Some Future Improvements

- Future improvements:
  - split URI data into uri and arguments
  - split referer data into host and uri
  - split agent data into browser/version/realname/os
  - make 'count' method handle SQL-like syntax
  - write request data to disk as we read it (less memory)
  - store request data in a binary format (less disk space)
- Future benefits of having regression tests:
  - we won't have to change our tests!
  - we'll be able to tell if adding functionality breaks anything immediately and we'll likely know why

Conclusion

Practical Take-away

- Testing is rewarding and gives instant feedback to how you're progressing and how accurate your code is
- Having regression tests enables you to refactor your code whenever you feel like it without fear of breaking something
- Writing documentation forces your brain to move from abstract to concrete; this process uncovers potential logic problems
- Concentrating on the API in testing and documentation helps you create a cleaner API before you begin coding
- You will save more total time by writing tests and documentation up front and you will free your code to be owned by someone else later

Philosophical Take-Away

- The true quality of a module lies not in clever algorithms, speed, or obscure use of the Perl flip-flop operator, but in clean design and in a well-thought-out API
- Testing and documenting before coding gives you a concrete plan you can turn to while coding; you also are forced to face and resolve logic problems and API conflicts long before you even code. This saves the most time possible because you hardly ever have to backward engineer anything
- Testing and documenting before coding provides you with a suite of regression tests that allows you to refactor your code anytime and as often as you want; you get to keep your confidence that the module still works because you have instant feedback when something breaks. This lets you make bold, sweeping changes or allows you to take advantage of new technology confidently.
- You'll have more friends if you test and document your code