A Development Tool for Improving Library Code

James E Keenan

Yet Another Perl Conference::North America::2015
Salt Lake City, Utah
Monday, June 8 2015

Today I'm going to present a simple development tool that enabled me to rapidly improve the existence of library code I wrote many years ago. The code I will present is probably the simplest code anyone will present at this year's YAPC -- so if you want complex stuff, leave now for one of the other sessions happening now. But because this tool is easy to use and has few prerequisites, you will be able to shape this tool to meet your development needs and improve your code as well.

I'll talk first about the library code and the situation that led me to want to improve its performance. Then, in stages, I'll build up a Perl program I used while refactoring that code. Finally, I'll present the development tool by which I measured performance improvements.

Library Code

When I use the term library code in the context of Perl, I'm referring to something which, for short, "looks like a CPAN distribution." That is, it consists of:

1 Perl modules -- *.pm files -- which provide functions or methods available for use in other Perl programs.

2 Test programs arranged in a test suite.

3 A build tool such as Makefile.PL which, in tandem with a library like ExtUtils-MakeMaker, provides functionality for compiling, testing and installing the Perl modules.

```
$> perl Makefile.PL
```

```
$> make
$> make test
$> make install
```

For the purpose of today's discussion, I'm going to assume that when you run your test suite -- that is, when you call "make test" -- you exercise a very high proportion of the code in your Perl modules. This means that you have very high test **coverage** of your source code. We call measuring the degree to which your test suite exercises your source code **coverage analysis**. You can perform coverage analysis by using Paul Johnson's Devel-Cover library, available on CPAN.

List-Compare

List-Compare <http://search.cpan.org/dist/List-Compare/> is a Perl library which I wrote and first uploaded to CPAN in 2002. It provides basic set relationships like intersection, union, subset, and so forth, among two or more lists. It started life as a wrapper around certain recipes in the second edition of the Perl Cookbook. It has both object-oriented and functional interfaces, and from the start I was writing tests for both interfaces.

I knew nothing about coverage analysis until I heard Paul Johnson speak about Devel-Cover at YAPC::EU in Paris in 2003. And I didn't start using Devel-Cover myself until Andy Lester gave a talk about it at YAPC::NA in Buffalo in 2004. But once I learned how to do it, I became a coverage fanatic. I wrote more tests for List-Compare until I knew my test coverage of its statements was more than 99%.

I put List-Compare under version control using Subversion in 2005 -- and then left it largely untouched for the next nine years. When I started my current job at MediaMath in 2012, I was pleasantly surprised to find that List-Compare was already being used in production. Better still, List-Compare is used by about 25 other CPAN libraries <https://metacpan.org/requires/distribution/List-Compare>.

In 2014 I moved List-Compare's version control from Subversion to Git and placed a repository for it on git repository on github.com <https://github.com/jkeenan/list-compare.git>. But I still made very few changes to the source code.

Set-Intersection

In February of this year, however, I somehow became aware of the existence of another CPAN library called Set-Intersection <http://search.cpan.org/dist/Set-Intersection/>, written by a Japanese Perl hacker named Katou Akira who goes by the handle of **turugina**. Set-Intersection follows the Unix adage of **"do one thing and do it well"**. It focuses strictly on computing the intersection of sets; you can't use it for any other set relationships. But it computes intersections **much** faster than List-Compare does. I was flattered to see that Set-Intersection's documentation made reference to List-Compare. I decided to repay the flattery by sending turugina a small patch for documentation and test improvements; that patch was accepted.

I then decided to see if I could speed up the way List-Compare computes intersections by "borrowing" turugina's method. Of course, to see whether that borrowing actually worked, I would have to start benchmarking List-Compare -- and I hadn't done any serious benchmarking of List-Compare in thirteen years. I would also have to make sure that I didn't introduce any bugs during the refactoring -- but I could trust that the testing and coverage analysis I had done more than a decade earlier would quickly alert me to such bugs.

Refactoring Workflow

I adopted the following workflow.

* In List-Compare's interface, I tackled one, externally visible function at a time, creating one new git branch at a time for work on each function.

```
$> git checkout -b speedup_get_intersection
```

* List-Compare's external functions tend to call internal functions to accomplish their work. Back in February the "get_intersection()" function called a number of internal functions which were defined in package List::Compare::Base::_Auxiliary. So that's where I began to hack:

```
$> vi lib/List/Compare/Base/_Auxiliary.pm
```

* One of the internal functions "get_intersection()" called was "_calculate_hash_intersection()". This is how it was defined:

```
sub _calculate_hash_intersection {
  my $xintersectionref = shift;
  my @xkeys = keys %{$xintersectionref};
  my %intersection = %{${$xintersectionref}{$xkeys[0]}};
  for (my $m = 1; $m <= $#xkeys; $m++) {
    my %compare = %{${$xintersectionref}{$xkeys[$m]}};
    my %result = ();
    foreach (keys %compare) {
      $result{$_}++ if (exists $intersection{$_});
    }
    %intersection = %result;
  }
  return \%intersection;
}
```

In the subroutine definition above, you will notice that at two locations I assign one hash to another hash. When I wrote this code thirteen years ago, I did so to improve the code's readability and decrease the line noise. But in the second case, when I was assigning to %compare, I used %compare only once thereafter. It turns out that you can get measurable performance improvements just by eliminating unnecessary hash-to-hash assignments.

* I stored my revisions in the repository.

```
$> git add lib/List/Compare/Base/_Auxiliary.pm
$> git commit -m "Eliminate hash-to-hash assignment in
_calculate_hash_intersection()."
```

* I ran my test suite to make sure tests were still passing.

```
$> perl Makefile.PL
$> make
$> make test
```

So far, so good. Nothing really new so far. But here's what I did to incorporate benchmarking into this workflow.

The "Direct" Program

I selected a few representative test cases from my test suite and wrote a new program holding them. That meant importing Test::More into my program.

```
# get_1_intersection.pl
use strict;
use warnings;
use 5.10.1;
use Test::More qw( no_plan );
use lib '/home/jkeenan/gitwork/list-compare/blib/lib';
use List::Compare::Functional qw( get_intersection );

my $medium  = [ 1001 ..   4000 ];
my $small   = [  991 ..   1010 ];

my ($args, $expect, @int, $seen);

say "List::Compare::Functional version: ",
    sprintf("%.5f" => $List::Compare::Functional::VERSION);

$args   = [ $medium, $small ];
$expect = { map { $_ => 1 } ( 1001 .. 1010 ) };
@int    = get_intersection( $args);

$seen   = { map { $_ => 1} @int };

is_deeply($seen, $expect, "Got expected intersection");
```

I then reworked the test functions I was calling to be callable for benchmarking. That meant importing Benchmark into my program.

```
# get_2_intersection.pl
use strict;
```

```
use warnings;
use 5.10.1;
use Test::More qw( no_plan );
use Benchmark qw( timethis );
use lib '/home/jkeenan/gitwork/list-compare/blib/lib';
use List::Compare::Functional qw( get_intersection );

my $medium  = [ 1001 ..   4000 ];
my $small   = [  991 ..   1010 ];

my ($args, $expect, @int, $seen);

say "List::Compare::Functional version: ",
    sprintf("%.5f" => $List::Compare::Functional::VERSION);

$args   = [ $medium, $small ];
$expect = { map { $_ => 1 } ( 1001 .. 1010 ) };

@int    = get_intersection( $args);
$seen   = { map { $_ => 1} @int };

is_deeply($seen, $expect, "Got expected intersection");

timethis( 1_000, sub { get_intersection($args) } );
```

I then enabled the program to run either the tests, or the benchmarks, or both by putting the tests and the benchmarks in different blocks which would run -- or not run -- depending on what switches I passed to the program on its command-line. That meant importing Getopt::Long -- another library distributed with the Perl 5 core -- into my program.

```
# get_3_intersection.pl
use strict;
use warnings;
use 5.10.1;
use Test::More qw( no_plan );
use Benchmark qw( timethis );
use Getopt::Long;
use lib '/home/jkeenan/gitwork/list-compare/blib/lib';
use List::Compare::Functional qw( get_intersection );

my ($tests_only, $benchmarks_only);

GetOptions(
    "tests-only"        => \$tests_only,
    "benchmarks-only"   => \$benchmarks_only,
) or die("Error in command line arguments");

die("Select either 'tests-only' or 'benchmarks-only' -- but not both!")
```

```
    if ($tests_only && $benchmarks_only);
  if ($benchmarks_only) { pass("Running benchmarks only") };
  if ($tests_only)      { pass("Running tests only") };

  my $medium  = [ 1001 ..   4000 ];
  my $small   = [  991 ..   1010 ];

  my ($args, $expect, @int, $seen);

  say "List::Compare::Functional version: ",
      sprintf("%.5f" => $List::Compare::Functional::VERSION);

  $args   = [ $medium, $small ];
  $expect = { map { $_ => 1 } ( 1001 .. 1010 ) };

  unless ($benchmarks_only) {
    @int    = get_intersection( $args);
    $seen   = { map { $_ => 1} @int };
    is_deeply($seen, $expect, "Got expected intersection");
  }

  unless ($tests_only) {
    timethis( 1_000, sub { get_intersection($args) } );
  }
```

I called the program I just wrote the direct program since it directly calls the tests and the benchmarks. I could run the direct program from the command-line and get results like this:

```
  $> perl get_3_intersection.pl --tests-only
  ok 1 - Running tests only
  List::Compare::Functional version: 0.38000
  ok 2 - Got expected intersection
  1..2

  $> perl get_3_intersection.pl --benchmarks-only
  ok 1 - Running benchmarks only
  List::Compare::Functional version: 0.38000
  timethis 1000:  1 wallclock secs ( 1.32 usr +  0.00 sys =  1.32 CPU) @
757.58/s (n=1000)
  1..1

  $> perl get_3_intersection.pl
  List::Compare::Functional version: 0.38000
  ok 1 - Got expected intersection
  timethis 1000:  1 wallclock secs ( 1.35 usr +  0.00 sys =  1.35 CPU) @
740.74/s (n=1000)
  1..1
```

The Version Comparison Program: *compare-git-versions*

I then wrote a second program which runs the direct program twice: first, in the version before my refactorings, then in the version after my refactorings. I use git to switch back-and-forth between the two versions. I describe this second program as a **version comparison program** which, as of now, I am calling *compare-git-versions*. How do I control which versions I'm running? By enabling compare-git-versions to take command-line switches as well!

```
   $ perl ~/gitwork/compare-git-versions/compare-git-versions \
   > --workdir=/home/jkeenan/gitwork/list-compare \
   > --program=/home/jkeenan/learn/perl/lc/get_3_intersection.pl \
   > --before=v0.38 \
   > --after=speedup_get_intersection \
   > --tests-only
   Note: checking out 'v0.38'.

   You are in 'detached HEAD' state. You can look around, make experimental
   changes and commit them, and you can discard any commits you make in this
   state without impacting any branches by performing another checkout.

   If you want to create a new branch to retain commits you create, you may
   do so (now or later) by using -b with the checkout command again.
Example:

     git checkout -b new_branch_name

   HEAD is now at daa3559... Update support information.  v0.38.
   Checking if your kit is complete...
   Warning: the following files are missing in your kit:
       META.yml
   Please inform the author.
   Generating a Unix-style Makefile
   Writing Makefile for List::Compare
   Writing MYMETA.yml and MYMETA.json
   cp lib/List/Compare.pm blib/lib/List/Compare.pm
   cp lib/List/Compare/Functional.pm blib/lib/List/Compare/Functional.pm
   cp lib/List/Compare/Base/_Engine.pm blib/lib/List/Compare/Base/_Engine.pm
   cp lib/List/Compare/Base/_Auxiliary.pm
blib/lib/List/Compare/Base/_Auxiliary.pm
   Manifying 4 pod documents
   ok 1 - Running tests only
   List::Compare::Functional version: 0.38000
   ok 2 - Got expected intersection
   1..2
   Previous HEAD position was daa3559... Update support information.  v0.38.
   Switched to branch 'speedup_get_intersection'
   Checking if your kit is complete...
```

```
   Warning: the following files are missing in your kit:
       MYMETA.json
       MYMETA.yml
   Please inform the author.
   Generating a Unix-style Makefile
   Writing Makefile for List::Compare
   Writing MYMETA.yml and MYMETA.json
   cp lib/List/Compare/Base/_Auxiliary.pm
blib/lib/List/Compare/Base/_Auxiliary.pm
   cp lib/List/Compare/Base/_Engine.pm blib/lib/List/Compare/Base/_Engine.pm
   cp lib/List/Compare.pm blib/lib/List/Compare.pm
   cp lib/List/Compare/Functional.pm blib/lib/List/Compare/Functional.pm
   Manifying 4 pod documents
   ok 1 - Running tests only
   List::Compare::Functional version: 0.39000
   ok 2 - Got expected intersection
   1..2
```

That's the output if I run it with the "--tests-only" command-line switch. But what's more interesting is the output I run compare-git-versions with the "--benchmarks-only" switch instead:

```
   $ perl ~/gitwork/compare-git-versions/compare-git-versions \
   > --workdir=/home/jkeenan/gitwork/list-compare \
   > --program=/home/jkeenan/learn/perl/lc/get_3_intersection.pl \
   > --before=v0.38 \
   > --after=speedup_get_intersection \
   > --benchmarks-only
  Note: checking out 'v0.38'.

  You are in 'detached HEAD' state. You can look around, make experimental
  changes and commit them, and you can discard any commits you make in this
  state without impacting any branches by performing another checkout.

  If you want to create a new branch to retain commits you create, you may
  do so (now or later) by using -b with the checkout command again. Example:

    git checkout -b new_branch_name

  HEAD is now at daa3559... Update support information.  v0.38.
  Checking if your kit is complete...
  Warning: the following files are missing in your kit:
      META.yml
  Please inform the author.
  Generating a Unix-style Makefile
  Writing Makefile for List::Compare
  Writing MYMETA.yml and MYMETA.json
  cp lib/List/Compare.pm blib/lib/List/Compare.pm
  cp lib/List/Compare/Base/_Engine.pm blib/lib/List/Compare/Base/_Engine.pm
  cp lib/List/Compare/Functional.pm blib/lib/List/Compare/Functional.pm
```

```
  cp lib/List/Compare/Base/_Auxiliary.pm
blib/lib/List/Compare/Base/_Auxiliary.pm
  Manifying 4 pod documents
  ok 1 - Running benchmarks only
  List::Compare::Functional version: 0.38000
  timethis 1000:  2 wallclock secs ( 1.34 usr +  0.00 sys =  1.34 CPU) @
746.27/s (n=1000)
  1..1
  Previous HEAD position was daa3559... Update support information.  v0.38.
  Switched to branch 'speedup_get_intersection'
  Checking if your kit is complete...
  Warning: the following files are missing in your kit:
      MYMETA.json
      MYMETA.yml
  Please inform the author.
  Generating a Unix-style Makefile
  Writing Makefile for List::Compare
  Writing MYMETA.yml and MYMETA.json
  cp lib/List/Compare/Base/_Engine.pm blib/lib/List/Compare/Base/_Engine.pm
  cp lib/List/Compare/Functional.pm blib/lib/List/Compare/Functional.pm
  cp lib/List/Compare/Base/_Auxiliary.pm
blib/lib/List/Compare/Base/_Auxiliary.pm
  cp lib/List/Compare.pm blib/lib/List/Compare.pm
  Manifying 4 pod documents
  ok 1 - Running benchmarks only
  List::Compare::Functional version: 0.39000
  timethis 1000:  1 wallclock secs ( 1.32 usr +  0.00 sys =  1.32 CPU) @
757.58/s (n=1000)
  1..1
```

Let's trim that output down:

```
   $ perl ~/gitwork/compare-git-versions/compare-git-versions \
   > --workdir=/home/jkeenan/gitwork/list-compare \
   > --program=/home/jkeenan/learn/perl/lc/get_3_intersection.pl \
   > --before=v0.38 \
   > --after=speedup_get_intersection \
   > --benchmarks-only

  List::Compare::Functional version: 0.38000
  timethis 1000:  2 wallclock secs ( 1.34 usr +  0.00 sys =  1.34 CPU) @
746.27/s (n=1000)

  Switched to branch 'speedup_get_intersection'

  List::Compare::Functional version: 0.39000
  timethis 1000:  1 wallclock secs ( 1.32 usr +  0.00 sys =  1.32 CPU) @
757.58/s (n=1000)
```

Inspecting the above we see that in List::Compare::Functional version 0.38,

"get_intersection()" ran at 746 times per second. By eliminating one unnecessary hash-to-hash assignment, the performance improved to 757 times per second. Not much, but that was just the first step. I performed several other refactorings -- and eventually was able to eliminate that internal function "_calculate_hash_intersection()" entirely. By List::Compare::Functional version 0.50, "get_intersection()" was able to run the same arguments approximately 950 times per second. So the refactoring sped the function up.

Recording Progress

The actual performance improvements were gratifying. But what I really want to emphasize today is how rapidly this simple developer's tool gave me feedback on my refactorings. Once a given set of refactorings were stored in a git version, I could rapidly switch back and forth between git versions for both testing and benchmarking.

As I worked my way through List-Compare's various interfaces and functions, I used compare-git-versions to benchmark refactorings on each function in term. I took note of the benchmark results and placed edited versions of the output in my git commit messages.

```
    commit 01581e8ea48c1ce23648386850475d88e4c9f4b0
    Author: James E Keenan <jkeenan@cpan.org>
    Date:   Sun Feb 8 20:24:04 2015 -0500

    Revise List::Compare::Functional::_intersection_engine().

    Considerable performance improvement.

    LCF version: 0.39000

    I. 5 lists of strings, each small
    timethis 100000:  8 wallclock secs ( 7.33 usr +  0.00 sys =  7.33 CPU) @
13642.56/s (n=100000)

    II. 5 lists of strings, each small, plus 1 large list
    timethis 50: 26 wallclock secs (26.49 usr +  0.00 sys = 26.49 CPU) @
1.89/s
    (n=50)

    LCF version: 0.41000
```

```
    I. 5 lists of strings, each small
    timethis 100000:  2 wallclock secs ( 1.89 usr +  0.00 sys =  1.89 CPU) @
52910.05/s (n=100000)

    II. 5 lists of strings, each small, plus 1 large list
    timethis 50:  7 wallclock secs ( 6.63 usr +  0.00 sys =  6.63 CPU) @
7.54/s
    (n=50)
```

This lets me easily review speed improvements by reviewing the git log.

```
  $> git log --format=fuller --reverse v0.38..HEAD
```

I've placed compare-git-versions on github <https://github.com/jkeenan/compare-git-versions>. It's not polished, nor do I expect to polish it into a definitive version. I want **you** to play with it, adapt it to your individual needs, then submit patches or pull requests so that collectively we polish it.

It's a developer's tool that will help you in the following situation: You have well tested library code, kept under version control, whose performance you wish to improve.

Thank you very much.