## May 27, 2011

Spent several hours exploring existing Getopt modules and argument validation modules on CPAN.

Organized my planning notes.

Wrote up general goals.

Wrote up a SYNOPSIS-like demonstration of intended uses.

My initial thinking had focused on providing methods to allow users to control help text. However, after looking at Pod::Usage, I realized that the module needs to coordinate well with POD. In fact, in any large project -- especially those headed for CPAN -- users will likely want their script usage text to be generated by pod2usage. For substantial scripts, a POD-based approach has the added benefit of providing man pages and other help text in a familiar Unix style.

I also realized that the other extreme is also a critical use scenario -- namely, quick scripts intended for developer use or some other very small audience. Such scripts don't need elaborate help text. Instead, what's needed is the ability to generate the help text from the option spec itself.

I initialized the project with module-starter and git init.

## May 28, 2011

Developed several usage scenarios, both how the caller would use the module and what the generated help text would look like.

Identified major components of the work and ordered them.

Scoped out iteration 1: nothing more than the OO front-end for Getopt::Long.

## May 30, 2011

Defined iteration 1 more narrowly: let the user pass in the Getopt::Long spec. Iteration 1 is a very thin OO wrapper around Getopt::Long.

Identified the methods needs in the minimal class.

## June 3, 2011

Finished Iteration 1.

Overview:

# A simple container for an option spec.

Getopt::Fullserv::Opt

getopt\_long\_spec # foo|f=i

name # foo

default => undef # User-supplied default value for the option.

# The option/arg parser.

Getopt::Fullserv;

original\_args => [] # eg @ARGV before parsing

unparsed\_args => [] # eg @ARGV after parsing

options => {} # see process\_raw\_specs()

# Convert Getopt::Long spec supplied by user into a hash

# of Getopt::Fullserv::Opt objects, keyed by the option name.

process\_raw\_specs()

# Method to retrieve one Opt object, using `name`.

# Currently used only for unit testing.

opt()

# Invokes GetOptionsFromArray, handles the setting

# of the original\_args and unparsed\_args parameters,

# and sets the user-supplied default values. Uses @ARGV

# if the user does not supply an array ref.

parse()

I did not use TDD fully. A very rough draft of the code was written during a day-long meeting. I took those notes and tried to use TDD. For a time I got bogged down in over-optimizing the testability of the code. After a few missteps I started to get into a true TDD cycle: write failing test, write code, etc. As with anything, some balance and common sense are useful. At times it probably makes sense to push the code slightly ahead of the tests, especially as you are feeling your way among various approaches. Nonetheless, I think I perceived a couple of times where the TDD rhythm kept the work on a good path.

Iteration 2 will deal with very basic support for the --help and --man options, based on pod2man.

## June 6, 2011

Made a concrete plan for Iteration 2. Implemented the following using TDD for the most part:

• Allow user to set a usage string. If not provided, use generic text

• Allow user to set help text.

• Intercept warnings from Getopt::Long

Still need to write some more tests to provide better coverage for the new code.

Considered switching to the OO interface for Getopt::Long, but it's lame and does not support GetOptionsFromArray().

## June 7, 2011

Mostly tests and info collecting:

• Took notes on Conway's best practices regarding command-line args/opts.

• Took notes on Pod::Usage.

• Added ability to set man\_text.

• Tests for cleaned\_warning.

• Tests for parse\_succeeded.

• Tests when there are parse errors.

## June 9, 2011

Started work on the help options:

• Add auto\_help attribute.

• Allow user to disable/enable auto\_help.

• Added ability to define parser parameters in the constructor.

• Added the help options: usage, help, man.

Going to encounter some complications in allowing user to control auto\_help after the constructor.

## June 10, 2011

Wrapped up work on help options:

• Finished work supporting auto\_help features

• Allow the user to call usage(), help(), man(), warn(), quit().

• Allow user to control auto\_help precisely after the constructor -- ie, specify a particular set of meta options (--usage, --help, --man).

Big departure from TDD. It felt like a hindrance for this phase of the work, because the implementation particulars were in so much flux. Did I get in a rush and start coding before planning? Or maybe TDD is of dubious value for some types of work?

I'm not sure that this outcome was due to insufficient planning. My ultimate user interface was pretty much exactly what I had planned for:

$gf->auto\_help(0);

$gf->auto\_help(1);

$gf->set\_auto\_help\_options('usage', 'help');

$gf->warn(@msg);

$gf->quit(@msg);

$gf->usage(@msg);

$gf->help(@msg);

$gf->man(@msg);

But the underlying code to achieve that outcome went through several quick rounds of refactoring during the day. I could have written tests for those user-facing methods, and that might have been good -- or at least fine. Two things kept me away from TDD.

(1) Several of the user-facing methods print output, which poses some additional testing challenges. I opted to ignore those issues until I had a clearer vision of my implementation approach.

(2) The intra-module approach to providing those user-facing methods changed rapidly through the day, so I was glad not to have committed to any unit tests for those private methods during the rough drafting phase.

One core dilemma of TDD is that the act of writing tests doubly commits you to a design choice. Both your module and your tests become bound to the design. That commitment might be worthwhile for user-facing methods, because it forces you to be very clear about goals and so forth. However, all of the small, private, intra-module methods that support the user-facing functionality seem like another story. During development, those frequently change. Writing unit tests in advance for such methods seems like a potential hindrance.

## June 11, 2011

Simple code reorg (putting related methods next to each other, and adding code headings). Also fleshed out the code comments.

## June 13, 2011

Identified code needed unit tests, and learned how to unit test code that prints: use Capture::Tiny or write yourself a method that will temporarily send STDOUT to a scalar.