Enterprise Perl

Jacinta Richardson
Paul Fenwick

Enterprise Perl

by Jacinta Richardson and Paul Fenwick

Conventions used throughout this text are based upon the conventions used in the Netizen training manuals by Kirrily Robert, and found at http://sourceforge.net/projects/spork

These notes represent a work-in-progress for Perl Training Australia's upcoming "Enterprise Perl" course. The OSCON tutorial, "Doing Perl Right", is intended as a heavily condensed version of this course for a conference setting.

As these notes are a work-in-progress, the authors apologise for any omissions, mistakes, and completely missing sections that may exist.

Updated copies of these notes will become available for free on Perl Training Australia's course notes page, at http://www.perltraining.com.au/notes.html.

This is revision 0.1-alpha-oscon of Perl Training Australia's "Enterprise Perl" training manual.

Table of Contents

List of Tables

Chapter 1. About Perl Training Australia

Training

Perl Training Australia (http://www.perltraining.com.au) offers quality training in all aspects of the Perl programming language. We operate throughout Australia and the Asia-Pacific region. Our trainers are active Perl developers who take a personal interest in Perl's growth and improvement. Our trainers can regularly be found frequenting online communities such as Perl Monks (http://www.perlmonks.org/) and answering questions and providing feedback for Perl users of all experience levels.

Our primary trainer, Paul Fenwick, is a leading Perl expert in Australia and believes in making Perl a fun language to learn and use. Paul Fenwick has been working with Perl for over 10 years, and is an active developer who has written articles for *The Perl Journal* and other publications.

Consulting

In addition to our training courses, Perl Training Australia also offers a variety of consulting services. We cover all stages of the software development life cycle, from requirements analysis to testing and maintenance.

Our expert consultants are both flexible and reliable, and are available to help meet your needs, however large or small. Our expertise ranges beyond that of just Perl, and includes Unix system administration, security auditing, database design, and of course software development.

Contact us

If you have any project development needs or wish to learn to use Perl to take advantage of its quick development time, fast performance and amazing versatility; don't hesitate to contact us.

Table 1-1. Perl Training Australia's contact details

Phone:	+61 3 9354 6001
Fax:	+61 3 9354 2681
Email:	contact@perltraining.com.au
Webpage:	http://perltraining.com.au/
Address:	104 Elizabeth Street, Coburg VIC, 3058 AUSTRALIA

Chapter 2. Rationale

The moving target of "best practice"

No matter what the field of endeavour, so long as it's still in use, "best practice" is a moving target. "Best practice" in car safety ten years ago is very different to best practice today. Even sanitation engineering, a field which has existed for millennia, changes with new technologies, research and advances in other fields. This moving target is especially true in the very young fields of Computer Science and Software Engineering.

The consequences of evolving best practice is that practitioners should always endeavour to stay abreast of changes in their field. For programmers and developers this may require:

- Attending programming-related conferences (especially those which cover your languages of choice).
- Participating in related user groups or mailing lists about your languages of choice.
- Paying attention to new books released regarding your languages of choice. Not all books will be good, but knowing what's being written about can give you hints as to which new technologies are worth paying attention to.
- Learning the basics of a new computer language regularly.

In particular, Perl programmers should be involved in their local Perl Mongers group (http://www.pm.org/).

"Perl Best Practices" by Dr Damian Conway

In 2005, O'Reilly Press published the "Perl Best Practices" book written by Dr Damian Conway. This book sets aside some very solid advice on coding standards, smart ways to code in Perl and should be mandatory reading by everyone who does any programming in Perl. However, not all the things it recommends are "best practice" now. For example light-weight objects are being discarded in favour of the Moose object framework. As such, it's important to keep paying attention to the changes in the language as time goes on.

Focus of this course

This course is designed to bring the fluent Perl programmer up to date in the current state of best practice with Perl. As discussed above, this is a moving target, so we expect that the material covered will change over time.

The concepts in this course are essential to any Perl project spanning several modules or even several hundred lines. You will learn what you need to do to best leverage existing tools, modules and ideas to make your code cleaner, easier to write, and importantly - easier to maintain.

Intended audience

Our target market for this course are those who are fluent in Perl but who either missed learning many of the topics in this course, or who learned these topics some time ago and who wants to learn about the improvements made since.

Chapter 3. Starting a Project

Things to consider

Before starting any Perl project it's important to stop and consider a few essential points:

1. Is starting a new project an efficient use of your time? Is there an alternative application that already fulfils your goals?
2. Are there modules on the CPAN that you can use to reduce the amount of work you have to do? (Even if you have to patch, subclass, or otherwise improve those modules in order to make them suitable.)
3. What systems do you have in place to support the application? For example you'll want (at the least): a revision control system; development, test and production environments; and sound policies ensuring correct handling of these environments.
4. How are you going to build the application? Ideally you should have a good idea of the requirements and be able to put together at least a top level design which specifies what modules you'll need and roughly how they'll interact.

After you've considered these things, it's a good idea to flesh out your design a little further. Ideally you'll be able to break the application into segments each with clear responsibilities. For example, you may have a system which displays data from a database in some format. Using `DBIx::Class` (or any other Object Relational Model) will allow you to abstract away the need to write SQL in your programs; while using `Template::Toolkit` means you don't have to embed display logic within your code and allows non-Perl programmers to design and edit the displayed content.

As systems get more complex, there's a greater chance that you'll find yourself taking an Object Oriented (OO) approach. While OO is not the only viable paradigm for large projects, it is a very popular, well-understood, and well-documented approach. Spending some time thinking about these ideas at the start of your code can make your programming faster and easier and will hopefully also make the end code much easier to maintain.

We will cover more on these ideas throughout the course.

Perl's modules

Perl's libraries are called modules and end with `.pm`. In addition to the Perl modules you may also find compiled library files (ending with `.so`), module fragments for autoload (ending with `.al`) and documentation (ending with `.pod`).

There are a few specialised modules called pragmas such as `strict`, `warnings` and `autodie`. Pragmas are denoted by their filename being all in lowercase. These tend to affect the compilation of our program. For example in the case of `strict`, undeclared variables are compilation errors. Many of these are lexically scoped which means you can turn their effects on and off for different blocks in your code.

All non-pragma (standard) modules should start with a capital letter to distinguish them from pragmas.

A Perl module is a file which contains a package of the same name. So `autodie.pm` (a pragma) has a package line inside which says:

```
package autodie;
```

Where we have namespace separators in a module name such as `File::Copy` then those represent directory separators on the file system. So the code for `File::Copy` is found in the `File/Copy.pm` file. Inside is the package line:

```
package File::Copy;
```

That the `Copy.pm` file is in the `File` namespace does not suggest more than a conceptual relationship with the other modules also in the `File` namespace. For example `File::Copy` and `File::Spec` are both file system related, but the former allows you to copy and rename files while the latter allows you to perform operations on file names in a portable manner.

Module locations

Where Perl stores modules depends on where Perl is installed on your system. You can find out these locations by typing `perl -V` and looking at the contents of `@INC`. For example this might be:

```
@INC:
  /etc/perl
  /usr/local/lib/perl/5.8.8
  /usr/local/share/perl/5.8.8
  /usr/lib/perl5
  /usr/share/perl5
  /usr/lib/perl/5.8
  /usr/share/perl/5.8
  /usr/local/lib/site_perl
  /usr/local/lib/perl/5.8.4
  /usr/local/share/perl/5.8.4
  .
```

When Perl searches for a module, it starts at the top of this list and looks for the first match. Thus if we had two `File::Copy` modules installed, one in `/usr/local/lib/perl5/5.8.8/File/Copy.pm` and one in `/usr/share/perl/5.8.4/File/Copy.pm` the only the former would be used.

We can change `@INC` by using the `lib` pragma. Thus if we wanted to tell Perl to prefer older versions of modules including `File::Copy` we might write:

```
use lib '/usr/share/perl/5.8.4/';
use File::Copy;
```

The same works for telling Perl to use our own versions:

```
use lib '/home/sue/Perl/';
use File::Copy;
```

and if there is a `/home/sue/Perl/File/Copy.pm` that is the module which will be used.

Building a module with Module::Starter

Starting a new module can be a lot of work. A good module should have a build system, documentation, a test suite, and numerous other bits and pieces to assist in its easy packaging and development. These are useful even if we never release our module to CPAN.

Setting this up can be a lot of work, especially if you've never done it before. Yet, we want to spend our time writing code not trying to get the set-up perfect. That's where `Module::Starter` comes in handy. It provides a simple, command-line tool to create a skeleton module quickly and easily.

Using module-starter

Before we can build our module, we need to install `Module::Starter` from the CPAN. `Module::Starter` allows us to choose from a variety of build frameworks, from the aging `ExtUtils::MakeMaker` to `Module::Install` and `Module::Build`. While `ExtUtils::MakeMaker` comes standard with Perl, you may need to install the other build frameworks, although `Module::Build` comes standard with Perl 5.10.0 and above. At Perl Training Australia we generally use `Module::Install`.

Creating a module with `Module::Starter` couldn't be easier. On the command line we simply write:

```
    module-starter --module=My::Module --author="Jane Smith"
--email=jane.smith@example.com --builder=Module::Install
```

The module name, author, and e-mail switches are all required. We've used the optional `--builder` switch to specify we want to use `Module::Install` as our build-system, instead of the default `ExtUtils::MakeMaker`.

Once this is done, you should have a `My-Module` directory with a skeleton module inside.

Configuration

`module-starter` will look for a configuration file first by checking the `MODULE_STARTER_DIR` environment variable (to find a directory containing a file called `config`), then by looking in `$HOME/.module-starter/config`. This can be used to provide the required arguments to make module creating even easier.

The configuration file is just a list of names and values. Lists are space separated. For example we might have:

```
author:  Jane Smith
email:   jane.smith@example.com
builder: Module::Install
```

With this configuration file, the above call to `module-starter` becomes:

```
    module-starter --module=My::Module
```

A skeleton tour

If you've never created a module before, or you've been making them by hand, then it's nice to take a look at what you get for your `Module::Starter` skeleton.

```
    $ ls -la

    total 8
    drwxr-xr-x   4 pjf pjf    0 Jul  4 16:59 .
    drwxr-xr-x  51 pjf pjf    0 Jul  4 16:59 ..
    -rw-r--r--   1 pjf pjf   96 Jul  4 16:59 .cvsignore
    -rw-r--r--   1 pjf pjf  109 Jul  4 16:59 Changes
    -rw-r--r--   1 pjf pjf   90 Jul  4 16:59 MANIFEST
```

```
-rw-r--r--   1 pjf pjf  183 Jul  4 16:59 Makefile.PL
-rw-r--r--   1 pjf pjf 1378 Jul  4 16:59 README
drwxr-xr-x   3 pjf pjf    0 Jul  4 16:59 lib
drwxr-xr-x   2 pjf pjf    0 Jul  4 16:59 t
```

Let's look at each of these files in turn:

.cvsignore

`Module::Starter` assumes you'll be using CVS for revision control, and provides a `.cvsignore` file with the names of files that are auto-generated and not to be tracked with revision control. At Perl Training Australia we use git for new projects, and so we rename this to `.gitignore`.

Changes

This is a human-readable file tracking module revisions and changes. If you're going to release your code to the CPAN, it's essential for your users to know what has changed in each release. Even if you're only using your code internally, this is a good place to document the history of your project.

MANIFEST

The `MANIFEST` file tracks all the files that should be packaged when you run a `make tardist` to distribute your module. Normally it includes your source code, any file needed for the build system, a `META.yml` that contains module meta-data (usually auto-generated by your build system), tests, documentation, and anything else that you want your end-user to have.

If you don't want to manually worry about adding entries to the `MANIFEST` file yourself, most build systems (including `Module::Install`) allow you to write `make manifest` to auto-generate it. For this to work, you'll want to make a `MANIFEST.skip` file which contains filenames and regular expressions that match files which should be excluded from the `MANIFEST`.

Makefile.PL

This is the front-end onto our build system. When we wish to build, test, or install our module, we'll always invoke `Makefile.PL` first:

```
perl Makefile.PL
make
make test
make install
```

Most build systems will provide a `make tardist` target for building a tarball of all the files in our `MANIFEST`, a `make disttest` for making sure our tests work with only the `MANIFEST` listed files, and `make clean` and `make distclean` targets for clearing up auto-generated files, including those from the build system itself if a `make distclean` is run.

You'll almost certainly wish to customise your `Makefile.PL` a little, especially if your module has dependencies. You'll want to consult your build system documentation for what options you can uses. For `Module::Install` this documentation can be found at http://search.cpan.org/perldoc?Module::Install.

README

The `README` file should contain basic information for someone thinking of installing your module. Mentioning dependencies, how to build, and how to find/report bugs are all good things to mention in the `README` file. Some systems (including the CPAN) will extract the `README` and make it available separate from the main distribution.

lib/

The `lib/` directory will contain your skeleton module, and is where you'll be doing much of your work. `Module::Starter` will have already added some skeleton documentation, a version number, and some skeleton functions.

You can add more modules to the `lib/` directory if you wish. Splitting a very large module into smaller, logical pieces can significantly improve maintainability.

t/

The `t/` directory contains all the tests that will be executed when you run a `make test`. By default, `Module::Starter` will provide some simple tests to ensure that your module compiles, that you'll filled in relevant sections of the boilerplate documentation, and that your documentation covers all your subroutines and doesn't contain any syntax errors.

If you're new to testing in Perl, then you should start by reading the `Test::Tutorial` at http://search.cpan.org/perldoc?Test::Tutorial.

At Perl Training Australia, we usually add a test based on `Test::Perl::Critic` http://search.cpan.org/perldoc?Test::Perl::Critic to encourage good coding practices, and `Test::Kwalitee` http://search.cpan.org/perldoc?Test::Kwalittee to catch common mistakes that are made in distributions.

Ideally when developing your module, the more tests you have, the better. If you already have a test suite and you're wondering which parts of your code are not being tested, you can use the excellent `Devel::Cover` tool from http://search.cpan.org/perldoc?Devel::Cover.

Encapsulation and big systems

Using `Module::Starter` should encourage you to consider how to break up your modules into small, independent parts. Just as `File::Copy` only provides code to `copy` and `move` files, so should each of your modules have a specific purpose and interface.

Good encapsulation is all about ensuring that your modules only interact through clearly defined interfaces. Code outside a module should not be calling its private functions directly, nor should it access object attributes directly if you're using hash-based object oriented Perl. The advantages are that the internals of your module can change dramatically without affecting external code (so long as you leave your interface alone). The main disadvantage crops up when a module's interface doesn't allow you access to an internal you want access to. Fortunately if you're the module maintainer, this is easy to fix.

Clearly your module design may change while you're writing it, but it's a good idea to spend some time up front scoping out what the public interface will be. What functions will you export, what arguments will they take and how will they interact? Would your purpose be best served just exporting those functions, or would an object oriented approach be more appropriate?

Once you start writing your module, you may find that it needs some helper modules as well. For example consider the `File::Spec` module. `File::Spec` allows you to deal with file systems in an operating system agnostic fashion so that you can use the temporary directory or back track up a directory tree without relying on the semantics of your system to the exclusion of others. The code in `File::Spec` is simple because all of the system-specific code is in its helper modules `File::Spec::Cygwin`, `File::Spec::Mac`. `File::Spec::Unix`, `File::Spec::Win32` etc. These helper modules are not designed to be used directly.

Class exercise

Your task is to rebuild a micro-chip animal registration system. Most of the data we already need has been stored in a spreadsheet without good data constraints. Duplicates abound and the data should be normalised.

In the end system animal details will be entered by a veterinary nurse and may need to be updated in the future. Each animal will have a unique ID provided by the embedded microchip and which will be also be provided to the owner on a card (as a 3d bar-code). Your program is responsible for generating the bar-code image for printing on the card.

The bar-code should also provide other information about the animal that a vet might want (such as the animal's birth year, pedigree, breed(s)) so that they can access that even if they are unable to access our database. If necessary, we can provide the vets with software to read the bar-code, but if there is a standard format that any reader can process, use that. If the vet can access our database they should be able to add health records of the animals they've seen.

We also want to be able to generate a bunch of related reports. Everything from how many animals are being registered per week, through to the average number of vet visits per animal per year, and the number of different vet clinics an animal goes to. Vets might be interested to know how many animals from our system they see each week.

1. What are the major components in this system?
2. Can we break down those components into separate parts?
3. Are any of these parts, things we may be able to use CPAN to supply?
4. Once we've leveraged CPAN as much as we can, what modules might we still need to write?
5. What, if any, parts of this system could be relegated to a "wish list" status?

Further information

If you're thinking of releasing your code to the CPAN, or just want more information about preparing a module for distribution, you can learn more by reading the `perlnewmod` page with `perldoc perlnewmod` or on-line at http://search.cpan.org/perldoc?perlnewmod.

Chapter 4. Accessing Perl's documentation with perldoc

perldoc

While opinions vary about the elegance and beauty of Perl, many people agree that Perl is one of the most well documented programming language in use today. Better still, much of that documentation is freely available and comes standard with Perl.

Most of Perl's documentation is accessible through the perldoc utility. Perldoc reads POD (plain old documentation) out of .pod files or from within the source of modules and programs themselves. This Perl tip is going to discuss some of the documents and perldoc switches that we find invaluable.

To learn more about how to write your own POD documentation read `perldoc perlpod`.

Module documentation

Individual module documentation

To read the documentation from any installed module we can write:

```
perldoc <module_name>
```

for example:

```
perldoc CGI
```

For multi-part module names, such as `Scalar::Util`, we can type either of the following:

```
perldoc Scalar::Util;
perldoc Scalar/Util;
```

although the first form is more regularly seen.

Module source

Sometimes we want to browse the source of a module without having to find out where the module is installed. To do this, we can use the *-m* switch to Perldoc:

```
perldoc -m CGI
```

Be aware that under older versions of perldoc this will display the source of the file containing the POD. For modules that separate code and pod, `perldoc -m` may not do what you expect.

Standard installed modules

To find out which modules were installed with Perl by default read:

```
perldoc perlmodlib
```

Non-standard installed modules

To see a list of the modules you've installed since Perl was installed read: perldoc perllocal

Note that this page may not always be accurate. Modules installed into user directories are not typically added to the perllocal list. Further, the `perllocal` list may not writable by the user at the time of module installation. Finally, modules installed using your operating system's packaging system (eg, RedHat RPM files, or Debian .deb files) rarely appear in `perllocal`.

Function documentation

Perhaps the most used switch to Perldoc is *-f*. This provides a full description of any Perl function. Thus, to find out all we could ever want to know about print, we'd type:

```
perldoc -f print
```

These pages provide information about what arguments our functions take, what they return, example uses, caveats and more.

To find the documentation for the file test operations type:

```
perldoc -f -x
```

The *-f* switch pulls its data from the `perlfunc` page. Thus if you wish to browse Perl's functions you may wish to start there.

Questions and answers

Perl has a number of FAQ files. These range from general questions through to quite specific topics.

Frequently asked questions

To find out all the topics covered in the FAQs you can read:

```
perldoc perlfaq
```

Searching the FAQs

Perl has a great focus on laziness. As a result you can search the FAQs with a regular expression, from the command line. To do this, provide a regular expression (or a string of interest) to perldoc with the *-q* flag:

```
perldoc -q 'CGI script'
perldoc -q 'CGI script.*browser'
```

This then searches the headings (questions) of the entries in the FAQs for your matches. You can also pass in keywords which interest you:

```
perldoc -q CGI script
```

which returns more results than the previous search.

General pages

Well beyond the few uses above, Perl has a great many more pages. To find a list of all of these read:

```
perldoc perl
```

or (to find these broken into more meaningful sections):

```
perldoc perltoc
```

Tutorials

In particular, the tutorials are well written and a great place to start to learn about new topics. Popular tutorials include:

```
perldoc perlreftut          (References)
perldoc perlopentut         (Open)
perldoc perldebtut          (Debugging)
perldoc perlretut           (Regular expressions)
perldoc perlboot            (OO for beginners)
perldoc perltoot            (OO part 1)
perldoc perltootc           (OO part 2)
perldoc perlthrtut          (Threads)
perldoc perlxstut           (XS)
```

Writing your own POD (Plain Old Documentation)

(See also progperl/pod.sgml)

Most of the help pages `perldoc` accesses are written in POD (plain old documentation). In this tip we're going to cover the basics of how to write POD.

POD is designed to be a simple, clean documentation language. It's goals are to be:

- Easy to parse
- Easy to convert to other formats (such as HTML and text)
- Easy to write in, even with the addition of sample code
- Easy to read in the native format.

Almost all Perl modules include their documentation in POD format. Although this is occasionally done as `.pod` files it's much more common to include the POD directly into the code. This has the advantage that it becomes much harder to accidently separate the code from its documentation.

Paragraphs

POD uses three kinds of paragraphs (text preceded by and followed by a blank line). These are:

Text paragraphs

Plain text paragraphs which the formatter can reformat if required.

Code paragraphs

Text paragraphs which the formatter must not reformat. These paragraphs are typically used for code. For example:

```
use strict;

my $test = 1;
```

You signal to the formatter that some text is a code paragraph by starting each line in the paragraph with at least one space (or tab) character.

Command paragraphs

Unlike the previous two paragraph styles, the content of these paragraphs are not seen by the user. These exist to signal to the formatter which lines are for it to parse, and to provide some indication of how you want your text to appear. POD command start with the equals character (=) in the first column and are listed below.

These are often called command *lines* however it should be recognised that the specification for these *lines* is that they appear on a line preceded and followed by blank lines. Thus a paragraph.

`=head1, =head2 ...`

These denote headings for first level, second level and so on. For example:

```
=head1 Plain Old Documentation
```

`=pod`

Tells the POD formatter that this is the start of a section of POD and thus to start formatting. This lasts until the `=cut` command is seen. This will cause the Perl interpreter to skip this section.

`=cut`

Tells the POD formatter that this is the end of a section of POD and thus to stop. This will cause the Perl interpreter to resume looking for Perl code.

`=over n`

This moves the indent level over by n spaces. This is used for creating itemised and descriptive lists.

`=item string`

This creates a list item with the descriptive term set to `string`. Use an `*` (asterisk) to represent itemised lists.

`=back`

Moves the indent level back to the level prior to the last `=over`. This is used for closing the current list level.

`=for format, =begin format, =end`

POD allows you to write different content for different formats. Thus you might have a table created by spaces for text based content, a html table and a LaTeX table each specified where required. These commands provide this functionality. eg:

```
=for html
        <table>
                <tr><td>1</td><td>Rhymes with sun</td></tr>
                <tr><td>2</td><td>Rhymes with shoe</td></tr>
        </table>

=for text
        1       Rhymes with sun
        2       Rhymes with shoe
```

`=for` applies only to the next paragraph whereas `=begin` will run until the next `=end` or `=cut`.

That's almost it to writing POD. There are also few formatting codes to allow you to specify **bold**, *italicised*, `code` style and similar effects. These are below and you can read more about these in `perldoc perlpodspec`.

```
B<bold>         # bold text
I<italicised>   # italicised text
C<code>         # fixed-width "code" text
```

An example

```
#!/usr/bin/perl -w
use strict;

# ...

# POD traditionally goes entirely at the top or entirely at the
# bottom of the file.

=pod

=head1 LoremIpsum.pm

=head2 Synopsis

        use LoremIpsum;

        aenean($dignissim);
        vivamus($accumsan, $semper, $elit, $eget, $odio);
        lacinia($nonummym $congue);

=head2 Description

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Mauris
nec wisi. Cras vitae justo. Nullam congue auctor libero.
Suspendisse ut libero et ante condimentum pellentesque. Donec
tincidunt.

Aenean dignissim. Proin elit nibh, placerat nec, lacinia
at, malesuada sit amet, nisl. In egestas pulvinar arcu. Praesent in
neque nec elit gravida dictum. Aenean et pede nec purus convallis
accumsan. Proin dolor. Donec id orci. Vivamus non augue feugiat
dolor viverra vehicula. Mauris vel wisi. Fusce in leo. Sed non
nulla. Fusce orci.

=over 4

=item C<aenean>

Aenean ultricies, ligula vel feugiat lobortis, arcu tortor
fermentum metus, ac rhoncus turpis lectus ac orci. Sed posuere
tellus in lectus eleifend rhoncus. Nam id massa.

=item C<vivamus>

Donec aliquam justo ut ante. Morbi tincidunt, sem a auctor
faucibus, felis leo blandit justo, in vestibulum felis dolor at
lorem.

=item C<lacinia>

Suspendisse lacinia sem eu ligula. Donec blandit molestie nulla.
Fusce a augue.  Vestibulum ante ipsum primis in faucibus orci
```

```
luctus et ultrices posuere cubilia Curae.

=back

=cut

1;
```

Chapter 5. Testing

Oh no!

```
Is there any sexier topic in software development than software
testing?  That is, besides game programming, 3D graphics, audio,
high-performance clustering, cool websites, et cetera? Okay, so
software testing is low on the list.

        -- Perl Testing: A Developer's Notebook
```

POD Testing

When software developers are rushed, both testing and documentation tends to suffer. However these two areas are of critical importance when it comes to maintaining and improving the software. Wouldn't it be great if there was an easy way for developers start their test suites and ensure they have good documentation at the same time?

While there aren't any Perl modules that can ensure that your documentation is correct, there are modules that exist to make sure that you have at least documented your interface, and that your documentation syntax is correct. In this section we will examine these modules and how they can be used to write a basic test suite for your documentation.

Test::Pod

`Test::Pod` lets you check the validity of a POD file. While perldoc and many of the other POD tools will be fairly forgiving when it comes to reading your POD, you're much more likely to avoid strange results if you do things correctly.

Test::Pod reports its results in the standard `Test::Simple` fashion using TAP output. This means that it fits in nicely with any other tests your test suite contains. To use it, add the following test to your test suite, and it'll ensure that the pod in your Perl files is valid. You can either pass the `all_pod_files_ok()` a list of directories to search through, or have it default to `blib/`.

```
#!/usr/bin/perl
use Test::Pod 1.00;
all_pod_files_ok();
__END__
```

A *Perl file* is considered to be any file which ends in `.PL`, `.pl`, `.pm`, `.pod`, or `.t`, or a file which starts with a shebang (`#!`) and includes the word `perl`.

Test::Pod::Coverage

`Test::Pod::Coverage` checks that your module documentation is comprehensive. It makes sure that there's either a `=headX` or an `=item` block documenting each public subroutine. If it cannot find documentation for a given subroutine, then it will fail.

Like `Test::Pod`, `Test::Pod::Coverage` should fit straight into your test suite. To use it, add the following test to your test suite:

```
#!/usr/bin/perl
use Test::Pod::Coverage;
all_pod_coverage_ok;
__END__
```

Private subroutines (those which start with an underscore) are not tested for documentation as they are not considered to be part of the module's application interface.

Trying it out

Create two test files in the `t/` directory with the above examples in them. For example `pod_valid.t` and `pod_coverage.t`.

Build your module and run the tests:

```
perl Makefile.PL
make
make test
```

This should result in output like:

```
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e"
"test_harness(0, 'blib/lib', 'blib/arch')" t/*.t
t/Acme-Example....ok
t/pod_coverage....ok
t/pod_valid.......ok
All tests successful.
Files=3, Tests=3,  1 wallclock secs ( 0.28 cusr +  0.01 csys =  0.29 CPU)
```

Add a test subroutine in your module (eg lib/Acme/Example.pm) without documentation, and re-run the tests, to see the pod coverage fail:

```
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e" "test_harness(0, 'blib/lib', 'blib/a
t/Acme-Example....ok
t/pod_coverage....NOK 1#     Failed test (/usr/share/perl5/Test/Pod/Coverage.pm at line 112)
# Coverage for Acme::Example is 0.0%, with 1 naked subroutine:
#       undocumented_sub
# Looks like you failed 1 tests of 1.
t/pod_coverage....dubious
Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 1
Failed 1/1 tests, 0.00% okay
t/pod_valid.......ok
Failed Test      Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------------
t/pod_coverage.t    1   256     1    1 100.00%  1
Failed 1/3 test scripts, 66.67% okay. 1/3 subtests failed, 66.67% okay.
make: *** [test_dynamic] Error 255
```

Testing Strategies

One important part of ensuring a program operates as expected is *testing*. This can be anything from a casual use of the code to a complete set of regression, unit, and user tests.

Black box testing

In black box testing you pretend that you don't know anything about the internals of your code, but instead test to the requirements specification you have. Thus, if you have a date field, you might test a range of things which look like dates and things which don't look like dates. If the requirement specification provides minimum and maximum values for a box, you test either side of those, as well as in the middle.

Because your tests do not require specialist knowledge of the program's implementation, the internals can change without invalidating the tests. In all well designed projects, interfaces between components should be well defined and fairly static.

White box testing

White box testing relies on intimate knowledge of the code internals. For example if your code creates arbitrary restrictions on things, such as the allowed elements in a list, then white box testing involves testing outside that limit. If the code dies under certain conditions, it involves testing those conditions.

White box tests can be invalidated by changes to your code and therefore may involve extra work to keep up to date. At the same time, it is always good practice to ensure that while your code expects no more than 30 elements in this list, here, it doesn't allow 31 elements in the code over there.

Code coverage testing is a white box testing methodology and will be covered later in this chapter.

Testing your code

Testing cannot prove the absence of bugs, although it can help identify them for elimination. It's impossible to write enough tests to prove that your program works completely. However, a comprehensive test-plan with an appropriate arsenal of tests can assist you in your goal of making your program defect free.

The modules discussed below are useful in both black and white box testing.

Test::Harness

The `Test::Harness` module is part of the standard Perl distribution and is ideal for testing modules. If you've ever installed a Perl module from source, then you've probably made use of `Test::Harness` in the **make test** portion.

Essentially, `Test::Harness` expects tests to print either "`ok`" or "`not ok`". In addition to this, we need to tell `Test::Harness` how many tests we're going to run in this file, so it knows if our testing program has halted prematurely. We do this at the start with a line which prints `1..M`, where `M` is the final test number. A trivial example is as follows:

```
print "1..1\n";

if (1) {
        print "ok\n";
} else {
        print "not ok\n";
}
```

If Perl is behaving properly, this test should always pass.

By convention, test files are regular perl Programs with a `.t` extension. In our case we may call our program `pass.t`. Then to run it, we can either use the command line:

```
perl -MTest::Harness -e "runtests 'pass.t'";
```

or the program:

```
#!/usr/bin/perl -w
# Run our simple tests
use strict;
use Test::Harness;

runtests 'pass.t';
```

If we wish to run tests in many files, we pass the list of filenames to the `runtests` function. Perl modules that use `ExtUtils::MakeMaker` will run all the tests that match the filename pattern `t/*.t`, and this is one of the reasons for the `.t` convention.

Running more than one test in a file is perfectly allowable. To make it easy to determine which tests succeeded and which tests failed (which is important when you have 100 tests), `Test::Harness` accepts the addition of a number to the test output. For example:

```
print "1..2\n";

if (1) {
        print "ok 1\n";
} else {
        print "not ok 1\n";
}

if ("also true") {
        print "ok 2\n";
} else {
        print "not ok 2\n";
}
```

Good programmers will note that keeping track of the test number (and fixing the later values when you want to add a test into the middle) is more difficult that it should be. This brings us nicely to `Test::More`.

Test::More

`Test::More` removes some of the hassle from writing your test scripts. It introduces a number of functions to make your tests even easier to write. For example, `is`, `isnt`, `like` and `unlike`:

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

# Test if simple addition is working.
is( (1 + 5), 6);

# Let's not compare apples to oranges.
isnt( "Malus domestica", "Citrus sinensis" );

# Does my name match this regular expression?
like ( $self->name(), qr/Jacinta/ );

# We can also provide the regular expression in a string form:
like ( $self->hobby(), '/Perl/i' );
```

```
# Make sure my favourite foods don't contain broad beans.
unlike ( $self->favourite_foods(), qr/broad beans/i );
```

`Test::More` automatically numbers your tests for you. This makes debugging easier and saves you from having to keep track of the numbers yourself. To tell `Test::More` how many tests you are expecting you add this information in the `use` line:

```
use Test::More tests => 10;
```

If you're just starting your test suite, or adding an unknown number of extra tests you can tell `Test::More` that you don't *know* how many tests there will be in total:

```
use Test::More 'no_plan';
```

Of course, once your test suite has become more stable, you should always provide a number of tests. In this way `Test::Harness` and any other testing modules can tell if your tests have stopped prematurely (which usually indicates something has gone wrong).

`Test::More` has a lot more tests than the four covered before. We'll illustrate some of these in the following example:

```
#!/usr/bin/perl -T
use strict;
use warnings;
use Test::More tests => 5;

# Load in my module, fail if it won't load.
use_ok 'Local::People';

# Test that we can create a new person
my $person;
eval { $person = Local::People->new(cache_id => 1234) };
like( $@, undef );

# Test that the age is 26
is( $person->age(), 26 );

# Test that the status is not "error"
isnt( $person->status(), "error");

# Test that languages matches the expected list:
my @languages = qw/Perl C PHP C++ Bash/;
eq_set( $person->languages(), \@languages );
```

`Test::More` does a lot more than this. To find out more about its functionality refer to its documentation. Many articles on testing in Perl and specifically `Test::More` have been written and we recommend these two in particular: Building Testing Libraries (http://www.perl.com/pub/a/2004/05/07/testing.html) and An Introduction to Testing (http://www.perl.com/pub/a/2001/12/04/testing.html?page=1).

Coverage testing and Devel::Cover

When testing code coverage there are several different metrics that can be used. These include:

- **Statement coverage** A statement is covered if it is executed. A statement is not necessarily the same as a line of code. A statement may also be a block of code or a segment. This is the weakest form of coverage.
- **Branch coverage** A branch is covered if it is tested in with both true and false outcomes. A good testing suite will ensure that a program will jump to all possible destinations at least once.

  Branch coverage helps you spot missing else cases and cases where your least commonly executed blocks have invalid code in them. 100% branch coverage implies 100% statement coverage.
- **Path coverage** Path coverage ensures that all paths through a program are taken. However this can lead to a great number of paths which are impossible to test. Often the path is restricted to paths in subroutines or consecutive branches.

  100% path coverage implies 100% branch coverage, but this is hard.
- **Condition coverage** Condition coverage requires testing all terms in each boolean expression. So `if($x || $y)` must be tested with each of the four possible values of `$x` and `$y`.

  100% condition coverage implies 100% branch coverage.

Testing the code coverage of your test suite is bound to show up areas of your code that are never executed. Sometimes these areas will not be reachable. If this is the case then you've spotted an error in your implementation, design or specification. This is a good thing to spot.

`Devel::Cover` provides statement, branch, condition, subroutine, pod and time coverage. Using `Devel::Cover` is very easy:

```
perl -MDevel::Cover=-db,cover_db,-coverage,statement,time yourprog args
cover
```

The `cover` program generates coverage reports.

Putting it all together, an example

Our module

```
package WalkMiddleEarth;
use strict;
use DBI;

our @ISA = qw/Exporter/;
our @EXPORT = qw/calculate_distance get_tofrom show_distance/;

=head2 calculate_distance

This expects a number of steps, and a step length in centimetres.

=cut

sub calculate_distance {
        my ($steps, $step_length) = @_;

        unless($steps and $step_length) {
                die "It appears that you've given me the wrong values";
        }
        if($steps <= 0) {
                die "Surely you've taken more than $steps steps!";
```

```
        }

        if($step_length <= 0) {
                die "Did you walk backwards by any chance?  ".
                    "$step_length should be positive";
        }
        my $distance = $step_length*$steps/100_000;     # Kilometres.

        return $distance;
}

=head2

This finds the first distance that is greater than or equal to the
distance provided (in kilometres).

=cut

sub get_tofrom {
        my ($distance) = @_;

        my ($to, $from, $delta) = _dbh()->selectrow_array("
                SELECT source, destination, distance
                FROM distance
                WHERE distance <= ?
                ORDER BY distance DESC
                LIMIT 1", undef, $distance);

        return ($to, $from, $delta);
}

=head2 show_distance

This expects a number of steps, and a step length (in cm) and returns
a message that indicates how far the user has walked.

=cut

sub show_distance {

        my $distance = calculate_distance(@_);
        my ($to, $from, $delta) = get_tofrom($distance);

        if( $to ) {
                return "You have walked $distance kilometres, ".
                       "which is more than the distance from $from ".
                       "to $to ( " . sprintf("%0.2f km",$delta) . ")."
        }
        else {
                return "You haven't walked very far, even by hobbit ".
                       "distances!"
        }
}

=head2 dist_between

Given two locations in Middle Earth, look-up and return the distance
between them.

Returns undefined if the distance cannot be found.

=cut

sub dist_between {
```

```
        my ($from, $to) = @_;
        ($from and $to) or die "I need two locations\n";

        # Our $to/$from may appear in either order in the table,
        # so we look for either arrangement.

        my ($distance) = _dbh()->selectrow_array("
                SELECT distance
                FROM distance
                WHERE (source = ? AND destination = ?) OR
                      (source = ? AND destination = ?)
        ",undef,$from,$to,$to,$from);
        return $distance;
}

sub _dbh {
        return DBI->connect("DBI:mysql:database=pjf","pjf.id.au","********",
                { AutoCommit => 1, RaiseError => 1, PrintError => 0 });
}
```

Some tests

Below is a fairly comprehensive set of tests for the `WalkMiddleEarth` module.

```
#!/usr/bin/perl -w
use warnings;
use strict;
use Test::More tests => 15;
use lib '.';

use_ok 'WalkMiddleEarth';

# Tests for WalkMiddleEarth::calculate_distance
# Try no data
eval{ WalkMiddleEarth::calculate_distance() };
ok( $@, "No data to WalkMiddleEarth::calculate_distance should throw an exception" );

# Try one argument
eval{ WalkMiddleEarth::calculate_distance(4) };
ok( $@, "Only one arg to WalkMiddleEarth::calculate_distance should throw an exception" );

# Try the other argument
eval{ WalkMiddleEarth::calculate_distance(undef, 4) };
ok( $@, "Undef steps should throw an exception" );

# Try a negative number of steps
eval{ WalkMiddleEarth::calculate_distance(-2, 4) };
ok( $@, "Negative steps should throw an exception" );

# Try a negative step-length
eval{ WalkMiddleEarth::calculate_distance(2, -4) };
ok( $@, "Negative step length should throw an exception" );

# Try both values negative
eval{ WalkMiddleEarth::calculate_distance(-2, -4) };
ok( $@, "Negative arguments should throw an exception" );

# Check we get the expected value for this data
my $distance = 4 * 70 / 100000;
is( WalkMiddleEarth::calculate_distance( 4, 70 ), $distance );

# Try an obviously wrong step-size
```

```
eval{ WalkMiddleEarth::calculate_distance(4, 300 ) };
ok( $@, "A *huge* step size should throw an exception" );

#######################################

# Tests for WalkMiddleEarth::get_tofrom
# First try no data
eval { WalkMiddleEarth::get_tofrom() };
ok( $@, "No data to WalkMiddleEarth::get_tofrom should throw an exception" );

# Try a zero length distance
eval { WalkMiddleEarth::get_tofrom(0) };
ok( $@, "Zero length distance should throw an exception" );

# Try a negative distance
eval { WalkMiddleEarth::get_tofrom(-400) };
ok($@, "Negative distance should throw an exception" );

# A known value
eq_set( [WalkMiddleEarth::get_tofrom(22.113) ], ["Three Farthing Stone",
        "Hobbiton", 22.113] );

#######################################

# Tests for WalkMiddleEarth::show_distance
# Try no data
eval { WalkMiddleEarth::show_distance() };
ok( $@, "No data to WalkMiddleEarth::show_distance should throw an exception" );

# Try to create a very small distance
like( WalkMiddleEarth::show_distance(1, 1), qr/You haven't walked very far/ );

# Try to create a bigger distance
like( WalkMiddleEarth::show_distance(32000, 70),
      qr/You have walked 22.4.* from Hobbiton to Three Farthing Stone/i );
```

Test results

As you'll see below, we're actually testing for things which *should* occur, but do not. For example, we have a test to include an unreasonably large stride (of 3m). In our tests we've commented that we expect to throw an exception here, but looking at our module we know it doesn't.

This is a case where our sensibly coded up test-suite based on requirements, rather than implementation, shows us a deficiency.

Below are the results for running these tests. We can either run each test file separately (which is easy as we only have one in this case) or we can use test harness to run each and every test.

Results from running `test.t` directly:

```
1..15
ok 1 - use WalkMiddleEarth;
ok 2 - No data to WalkMiddleEarth::calculate_distance should throw an exception
ok 3 - Only one arg to WalkMiddleEarth::calculate_distance should throw an exception
ok 4 - Undef steps should throw an exception
ok 5 - Negative steps should throw an exception
ok 6 - Negative step length should throw an exception
ok 7 - Negative arguments should throw an exception
ok 8
not ok 9 - A *huge* step size should throw an exception
```

```
#     Failed test (tests.t at line 39)
not ok 10 - No data to WalkMiddleEarth::get_tofrom should throw an exception
#     Failed test (tests.t at line 46)
not ok 11 - Zero length distance should throw an exception
#     Failed test (tests.t at line 50)
not ok 12 - Negative distance should throw an exception
#     Failed test (tests.t at line 54)
ok 13 - No data to WalkMiddleEarth::show_distance should throw an exception
ok 14
ok 15
# Looks like you failed 4 tests of 15.
```

Results from running tests using test harness

```
$$ perl -MTest::Harness -e "runtests 'tests.t'";
tests.............NOK 9#      Failed test (tests.t at line 39)
tests.............NOK 10#     Failed test (tests.t at line 46)
tests.............NOK 11#     Failed test (tests.t at line 50)
tests.............NOK 12#     Failed test (tests.t at line 54)
tests.............ok 15/15# Looks like you failed 4 tests of 15.
tests.............dubious
        Test returned status 4 (wstat 1024, 0x400)
DIED. FAILED tests 9-12
        Failed 4/15 tests, 73.33% okay
Failed   Test Status Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------------
tests.t            4  1024    15    4  26.67%  9-12
Failed 1/1 test scripts, 0.00% okay. 4/15 subtests failed, 73.33% okay.
```

Results from running tests compiled with Devel::Cover

```
$$ perl -MDevel::Cover -T tests.t
1..15
Devel::Cover 0.50: Collecting coverage data for branch, condition, statement, subroutine and time.
Selecting packages matching:
Ignoring packages matching:
    /Devel/Cover[./]
Ignoring packages in:
    .
    /usr/lib/perl/5.6.1
    /usr/lib/perl5
    /usr/local/lib/perl/5.6.1
    /usr/local/lib/site_perl
    /usr/local/lib/site_perl/i386-linux
    /usr/local/share/perl/5.6.1
    /usr/share/perl/5.6.1
    /usr/share/perl5
ok 1 - use WalkMiddleEarth;
ok 2 - No data to WalkMiddleEarth::calculate_distance should throw an exception
ok 3 - Only one arg to WalkMiddleEarth::calculate_distance should throw an exception
ok 4 - Undef steps should throw an exception
ok 5 - Negative steps should throw an exception
ok 6 - Negative step length should throw an exception
ok 7 - Negative arguments should throw an exception
ok 8
not ok 9 - A *huge* step size should throw an exception
#     Failed test (tests.t at line 39)
not ok 10 - No data to WalkMiddleEarth::get_tofrom should throw an exception
#     Failed test (tests.t at line 46)
```

```
not ok 11 - Zero length distance should throw an exception
#     Failed test (tests.t at line 50)
not ok 12 - Negative distance should throw an exception
#     Failed test (tests.t at line 54)
ok 13 - No data to WalkMiddleEarth::show_distance should throw an exception
ok 14
ok 15
# Looks like you failed 4 tests of 15.
---------------------------------- ------ ------ ------ ------ ------ ------
File                                 stmt branch   cond    sub   time  total
---------------------------------- ------ ------ ------ ------ ------ ------
WalkMiddleEarth.pm                   81.8   80.0   50.0   80.0   76.4   76.7
tests.t                             100.0    n/a    n/a    n/a   23.6  100.0
Total                                93.4   80.0   50.0   80.0  100.0   87.8
---------------------------------- ------ ------ ------ ------ ------ ------
```

Running **cover** provides us with a set of HTML files of information on which lines executed and how many times, as well as coverage information. However, this material heavily relies on colours of similar intensity, which do not print well. Text only information has been included as the results here.

```
File Coverage

    File: WalkMiddleEarth.pm
Coverage: 76.7%

line stmt branch cond sub time                    code
[...]
8                                 sub calculate_distance {
9    11               11  77          my ($steps, $step_length) = @_;
10
11   11   100    100      161          unless($steps and $step_length) {
12   4                    43                  die "It appears that you've
                                                 given me the wrong values";
13                                     }
14
15   7    100             214          if($steps <= 0) {
16   2                    12                  die "Surely you've taken
                                                 more than $steps steps!";
17                                     }
18
19   5    100             50           if($step_length <= 0) {
20   1                    6                   die "Did you walk backwards
                                                  by any chance? ".
21                                                "$step_length should be
                                                   positive";
22                                     }
23   4                    32           my $distance = $step_length*$steps
                                                    /100_000; # Kilometres.
24
25   4                    32           return $distance;
26                                }
[...]
```

As you can see the output is a bunch of columns of numbers and your code. The column values correspond to:

1. Code line number
2. Number of times this statement was executed
3. Coverage of this branch (where applicable)

4. Coverage of this condition (where there are multiple things to be tested in a condition)
5. Number of times this subroutine was called
6. Time spent in statement
7. Corresponding code

Some of these numbers are hyper-links to other other mini reports. These reports are covered below.

```
Subroutine Coverage

    File: WalkMiddleEarth.pm
Coverage: 80.0%

line     subroutine
15   calculate_distance
42   get_tofrom
63   show_distance
87   dist_between
103  _dbh

---------------------------------------

Branch Coverage

    File: WalkMiddleEarth.pm
Coverage: 80.0%

line  %  coverage              branch
17   100 T   F    unless ($steps and $step_length)
21   100 T   F    if ($steps <= 0)
25   100 T   F    if ($step_length <= 0)
66   100 T   F    if ($to) { }
88   0   T   F    unless $from and $to

---------------------------------------

Condition Coverage

    File: WalkMiddleEarth.pm
Coverage: 50.0%

line  %  coverage        condition
         A B dec
         0 X  0
17   100 1 0  0   $steps and $step_length
         1 1  1

         A B dec
88   0   0 X  0   $from and $to
         1 0  0
         1 1  1
```

In this case we see that our tests have highlighted that there are regions where we are not testing our code. For example, we're not testing anything in the `dist_between` function. We should probably fix this.

Note that adding `if(0) { ... }` will not affect coverage results. This is because the perl compiler is smart enough to optimise such blocks away.

We can add the following tests:

```
# Tests for WalkMiddleEarth::dist_between

# No arguments.
eval { WalkMiddleEarth::dist_between() };
ok($@,"dist_between with no args should throw an exception.");

# Wrong number of arguments
eval { WalkMiddleEarth::dist_between("Hobbiton") };
ok($@,"dist_between with wrong number of args should throw an exception.");

# Locations that don't exist.
is(WalkMiddleEarth::dist_between("Melbourne","Sydney"),
   undef,"Non-existent locations");

# Known distance that does exist.
is(WalkMiddleEarth::dist_between("Three Farthing Stone", "Hobbiton"),
   22.113, "Distances in the Shire");
```

This gives us the following results:

```
---------------------------------- ------ ------ ------ ------ ------ ------
File                                 stmt branch   cond    sub   time  total
---------------------------------- ------ ------ ------ ------ ------ ------
WalkMiddleEarth.pm                  100.0  100.0  100.0  100.0   79.0  100.0
tests2.t                            100.0    n/a    n/a    n/a   21.0  100.0
Total                               100.0  100.0  100.0  100.0  100.0  100.0
---------------------------------- ------ ------ ------ ------ ------ ------

File Coverage

    File: WalkMiddleEarth.pm
Coverage: 100.0%

line stmt branch cond sub time                                    code
[...]
86                              sub dist_between {
87   4                4   33           my ($from, $to) = @_;
88   4    100    100      38           ($from and $to) or die "I need two locations\n";
89
90                                     # Our $to/$from may appear in either order
                                       # in the table
91                                     # so we look for either arrangement.
92
93   2                   19           my ($distance) = _dbh()->selectrow_array(
94                                             SELECT distance
95                                             FROM distance
96                                             WHERE (source = ? AND destination = ?) OR
97                                                   (source = ? AND destination = ?)
98                                     ",undef,$from,$to,$to,$from);
99   2                   497          return $distance;
100                             }
[...]

----------------------------------------

Subroutine Coverage

    File: WalkMiddleEarth.pm
Coverage: 100.0%

line     subroutine
15   calculate_distance
42   get_tofrom
63   show_distance
```

```
87   dist_between
103  _dbh

---------------------------------------

Branch Coverage

    File: WalkMiddleEarth.pm
Coverage: 100.0%

line  %  coverage              branch
17   100 T   F    unless ($steps and $step_length)
21   100 T   F    if ($steps <= 0)
25   100 T   F    if ($step_length <= 0)
66   100 T   F    if ($to) { }
88   100 T   F    unless $from and $to

---------------------------------------

Condition Coverage

    File: WalkMiddleEarth.pm
Coverage: 100.0%

line  %  coverage        condition
         A B dec
         0 X  0
17   100 1 0  0   $steps and $step_length
         1 1  1

         A B dec
88   100 0 X  0   $from and $to
         1 0  0
         1 1  1
```

And now we have achieved 100% coverage, even if we have some tests failing.

Good tests

In the above example we demonstrated that 100% coverage is possible even with failing tests. Thus, 100% coverage does not mean that your code is correct, nor that your current tests are any good. It's a good thing to aim for, but it is not the whole solution, or even necessarily a big part of it.

A good test is one which tests a single requirement from the specification. Such as: *the date field only accepts valid dates*. Obviously you'll probably need more than one test for this requirement.

The other kind of good test exists to test a single, known, restriction in the code. For example, the assumption that the name field won't have more than 100 characters in it. This may not be a requirement on your system but could be a consequence of your database design, web form or something else. Once again, you'll probably need more than one test for this restriction.

Tests that exist merely to achieve greater coverage are usually examples of bad tests.

A good test suite has one test for each aspect of each requirement and one test for each aspect of each known code restriction. In the examples above, this would include testing date-like data which can't be correct (such as 31-31-2000) and valid dates. If there are restrictions on the minimum date or maximum date, either through specification and design or through implementation, there should be tests on the either side of the edge cases.

A well designed test suite will already have reasonable code coverage. Coverage testing then highlights the areas in which your test suite needs work. Code coverage can also highlight areas where your specification, design or implementation needs work.

Coverage testing is just one of the many tools in your testing tool-box.

Chapter summary

- Black box testing involves testing the requirements, not the implementation.
- White box testing involves testing the code with knowledge of its internals.
- `Test::Harness` is a module which allows you to manage a test suite.
- `Test::More` provides greater functionality to the `Test::Harness` module by expanding the test functions to include `is`, `isnt`, `like` and `unlike` as well as many others.
- `Devel::Cover` provides coverage testing capabilities for statement, condition, branch, subroutine and time coverage.

Chapter 6. Writing good code

Perl::Critic - automatically review your code

The *best* way to do things in Perl keeps changing, just as the language and the art of programming does. The code you wrote 10 years ago might have been *best practice* back then, but chances are it doesn't look too good by modern standards. Package file handles, obscure special variables, and using `select` to control buffering are still supported by Perl, but they're not things that should exist in modern code.

`Perl::Critic` reads your source code and tells you what you're doing wrong. It uses Damian Conway's book *Perl Best Practices* as a starting point, however it's extensible enough that you can change its policies to enable, disable and customise what it complains about.

Whether you're writing new code, or maintaining old code, `Perl::Critic` can help you avoid bad practices and even teach you new techniques you may not have known existed.

Levels of severity

Chances are, for any serious code, `Perl::Critic` is going to find a lot of things to complain about. To make using it easier, there are five levels of severity that you can pick from. These are: `gentle`, `stern`, `harsh`, `cruel` and `brutal`.

Gentle turns on only the most important and worrisome errors that `Perl::Critic` finds. The stern severity notes all the things that `Perl::Critic` is pretty sure you will want to fix.

The later three levels of severity get more and more picky, down to complaining about your indentation and bracing style, whether you've commented out any code, and the modifiers used on your regular expressions.

It's recommended you start using `Perl::Critic` with gentle feedback, and proceed to the more picky levels in an incremental manner. You may not agree with all of `Perl::Critic`'s feedback, but we'll see in a moment how we can customise it to suit our needs.

Perl::Critic online

The easiest way to use `Perl::Critic` is to play with the web interface at http://perlcritic.com/. Here you can upload your Perl file, select a severity level and see what it complains about. This is a great way to play with `Perl::Critic` but becomes impractical if you have lots of files you want to critique or files containing sensitive information.

perlcritic on the command-line

If you need more power than uploading your script to the web interface, you can install `Perl::Critic` on from CPAN. Then you can use the command-line tool `perlcritic`. This is part of the `Perl::Critic` distribution and can be used as follows:

```
perlcritic --gentle myprogram.pl
```

We can also test all the files in a directory (this time using the stern severity level):

```
perlcritic --stern directory/
```

If you don't want to type full severity levels, you can use numerical shortcuts. The following are equivalent to the full lines above:

```
perlcritic -5 myprogram.pl
perlcritic -4 directory/
```

By default, `perlcritic` assumes a level of `gentle` if no severity level is set.

The `perlcritic` command generates a colour-coded list of things in your code that it has issue with. Example output (without the colour) might look like:

```
Symbols are exported by default at line 40, column 1.
Use '@EXPORT_OK' or '%EXPORT_TAGS' instead.  (Severity: 4)

Always unpack @_ first at line 60, column 1.
See page 178 of PBP.  (Severity: 4)

Subroutine does not end with "return" at line 60, column 1.
See page 197 of PBP.  (Severity: 4)

Expression form of "eval" at line 71, column 3.
See page 161 of PBP.(Severity : 5)

Stricture disabled at line 80, column 3.
See page 429 of PBP.  (Severity: 5)

Expression form of "eval" at line 106, column 4.
See page 161 of PBP.  (Severity: 5)

Mixed high and low-precedence booleans at line 158, column 6.
See page 70 of PBP.  (Severity: 4)

Variable declared in conditional statement at line 223, column 2.
Declare variables outside of the condition.  (Severity: 5)

"return" statement with explicit "undef" at line 227, column 2.
See page 199 of PBP.  (Severity: 5)
```

Understanding the output

Many of the diagnostics `perlcritic` provides by default are designed to only take up one line, although in our output above we've reformatted them to two lines for clarity. The brief format is to prevent the user being overwhelmed by output when they already know what the errors mean. However, when you're just starting, you'll want additional help understanding what you've done wrong - especially if you don't have a copy of *Perl Best Practices* to look up for the page references.

`perlcritic` comes with pre-defined verbosity levels from 1 to 11, with 8 and above providing policy names which can be looked up using `perldoc`, and 10 and above providing full policy information.

```
perlcritic --stern --verbose 10 lib/MyModule.pm

Symbols are exported by default at line 40, column 1.
  Modules::ProhibitAutomaticExportation (Severity: 4)
    When using Exporter, symbols placed in the '@EXPORT' variable are
    automatically exported into the caller's namespace. Although convenient,
    this practice is not polite, and may cause serious problems if the
    caller declares the same symbols. The best practice is to place your
    symbols in '@EXPORT_OK' or '%EXPORT_TAGS' and let the caller choose
    exactly which symbols to export.
```

```
package Foo;

use base qw(Exporter);
our @EXPORT      = qw(&foo &bar);                  # not ok
our @EXPORT_OK   = qw(&foo &bar);                  # ok
our %EXPORT_TAGS = ( all => [ qw(&foo &bar) ] );   # ok
```

It's also possible to configure own format for these diagnostics.

Excluding and including rules

There will be times when you disagree with the rules `Perl::Critic` uses. For example, at Perl Training Australia we often write code which uses the `constant` pragma to allow perl to optimise code away at compile-time. However `Perl::Critic` complains about this on the more strict severity levels.

It's possible to exclude specific lines from criticism by using a special `no critic` comment with a double-hash:

```
use constant WINDOWS => ($^O eq 'MSWin32');  ## no critic
```

However it's also possible to tell `Perl::Critic` at the time of invocation to exclude this rule entirely:

```
perlcritic --gentle \
   --verbose 10 \
   --exclude ProhibitConstantPragma myprogram.pl
```

It's also possible to specifically include rules or groups of rules, even if they're more severe that your chosen severity level. In the following example, we enable all the code layout diagnostics:

```
perlcritic --gentle
   --verbose 10 \
   --exclude ProhibitConstantPragma \
   --include CodeLayout myprogram.pl
```

Profiles

If you find yourself using too many options on the command-line, rather than aliasing it, put the information into a profile. By default, `perlcritic` will look for a `.perlcriticrc` (or `_perlcriticrc` for Windows) file in your current directory first, and then in your home directory. This allows developers to have per-project configuration, as well as their own personal preferences.

If we added the following to our `.perlcriticrc` file:

```
severity = gentle
verbose  = 10
exclude  = ProhibitConstantPragma
include  = CodeLayout
```

then we can condense the previous example back to:

```
perlcritic myprogram.pl
```

Perl::Critic and testing

As all good programmers write code with detailed test suites, you will be glad to know that there's a very easy way to use `Perl::Critic` in your test suite. Just use the `Test::Perl::Critic` module from the CPAN:

```
use Test::Perl::Critic;
use Test::More tests => 1;
critic_ok($file);
```

For CPAN distributions, the recommended usage is a little longer, to ensure that end-users don't have to unnecessarily run what are normally author-only tests. Consult the `Test::Perl::Critic` documentation for more information.

Progressive testing

Adding coding standards to legacy code can be challenging. Rather than trying to make all of your code `Perl::Critic` compliant at once, you may find it useful to use `Test::Perl::Critic::Progressive`. This is designed to prevent further deterioration of your code, and does so by breaking on a test run if there are *more* `Perl::Critic` violations than the last successful test. When run as part of your revision control commit tests, this will ensure that the code in your repository can only get better.

```
use Test::Perl::Critic::Progressive qw( progressive_critic_ok );
progressive_critic_ok();
```

Further reading

- `Perl::Critic` module documentation - http://search.cpan.org/perldoc?Perl::Critic
- `perlcritic` command line tool documentation - http://search.cpan.org/perldoc?perlcritic
- `Test::Perl::Critic` documentation - http://search.cpan.org/perldoc?Test::Perl::Critic
- `Test::Perl::Critic::Progressive` documentation - http://search.cpan.org/perldoc?Test::Perl::Critic::Progressive
- *Perl Best Practices* book information - http://perltraining.com.au/books/bestpractices.html
- Perl Training Australia book offers. Get discount books with your courses, or free books when you book by our *early bird* date:

  http://perltraining.com.au/books/

Chapter 7. autodie - The art of Klingon Programming

Klingon programming proverb

```
bIlujDI' yIchegh()Qo'; yIHegh()!

It is better to die() than to return() in failure.
```

The problem with error handling

One of the first things most people learn when programming in Perl is to always check to see if system calls like `open` are successful. The most common construct seen for such checking is the *do or die* style:

```
open(my $fh, '<', $filename) or die "Can't open $filename - $!";
```

The problem with *do or die* is that for calls to built-ins like `open`, it's very rare that we'd want to do anything *other* than die on failure. Even if we wish to handle the error, the call to `die` provides a convenient way to generate an exception, which we can then trap using an `eval` block.

Unfortunately, the *do or die* construct is tiresome to repeat, easily forgotten, and in many cases takes up more code and visual space than the function it is designed to guard.

Wouldn't it be nice if functions could die automatically on failure?

Fatal

Perl has come bundled with the `Fatal` core module for many years. Put simply, `Fatal` allows a built-in or user-defined subroutine to implement the *do or die* strategy by default. This saves us of the burden of having to remember to write `or die` ourselves, and it means the lazy behaviour of not checking our system calls becomes correct usage.

```
use Fatal qw(open);

# open will die automatically on failure
open(my $fh, '<', $filename);
```

Despite `Fatal`'s long history in the Perl core, it suffers from some serious problems. Since built-ins and subroutines need to be explicitly listed, it's possible to end up with very long and cumbersome use lines:

```
use Fatal qw(open close opendir sysopen fcntl chdir);
```

`Fatal` also tests the return values of subroutines and built-ins using a simple boolean test, but this makes `Fatal` unsuitable for built-ins such as `fork`. The `fork` call will create a child process, and returns the new process ID to the parent, and 0 to the child. On error, it returns undef. Unfortunately, `Fatal` is unable to distinguish between a successful `fork` being returned to the child (returning zero), and a genuine error (returning undef).

Upon some investigation, there are a large number of Perl built-ins that use undef to indicate failure, but can potentially return 0 on success. These include `fileno`, `send`, `recv`, and even `open` when used to create a pipe to a child process. Using `Fatal` with any of these functions has the potential to introduce bugs.

When used, `Fatal` works with package-wide scope, changing all appropriate calls in the current package to the Fatal versions. Unfortunately, for a large package, this can result in difficult action-from-a-distance. The code below contains a bug, as `use_default_config()` is never reached. The `use Fatal` line deep inside the subroutine changes all calls to `open` in the same package.

```
# Even though it doesn't look like it, the following
# open() will die on failure, meaning use_default_config
# will never get called.

if (open(my $fh, '<', $config)) {
        read_config($fh);
} else {
        use_default_config();
}

# Then, thousands of lines later...

sub open_or_default {
        my ($file) = @_;

        $file ||= 'default_customers.txt';

        # The 'use Fatal' here still changes the open()
        # thousands of lines above!

        use Fatal qw(open);
        open(my $fh, '<', $file);

        return $fh;
}
```

However the most common complaint levelled against `Fatal` is that the generated error messages are ugly. They're *really* ugly. Just look at the code, and the error generated:

```
use Fatal qw(open);

open(my $fh, '<', 'no_such_file');
```

The error:

```
Can't open(GLOB(0x10010dec), <, no_such_file): No such file or directory at (eval 1) line 4
main::__ANON__('GLOB(0x10010dec)', '<', 'no_such_file') called at - line 3
```

While it's certainly possible to determine what went wrong, it's certainly not something that will say "quality software" to your users.

autodie

In Perl, laziness is a virtue. Fatal allows me to write my code in a lazy fashion, without having to manually check every single system call for errors. At least, that's what it's supposed to do; as we've just seen, it comes with its own set of bugs.

Luckily, there's now a replacement for `Fatal`, named `autodie`. Developed by a brilliant young Australian, it fixes not only the multitude of bugs that come with `Fatal`, but provides the laziest possible solution for having all relevant built-ins die on error:

```
    use autodie;        # All relevant built-ins now die on error!
```

The `autodie` pragma has *lexical scope*, meaning it lasts until the end of the current eval, block, or file, in the same way that `strict` and `warnings` work. This means it's perfect for adding to subroutines, without worrying about side-effects elsewhere in your code:

```
sub open_or_default {
        my ($file) = @_;

        $file ||= 'default_customers.txt';

        # The 'use autodie' here only changes the call to open()
        # in this block, and nowhere else.

        use autodie;
        open(my $fh, '<', $file);

        return $fh;
}
```

Just like `Fatal`, it's possible to write `use autodie qw(open)` to enable the do-or-die behaviour for individual functions.

It's also possible to use `no autodie` to disable the `autodie` pragma until the end of the current lexical scope.

Inspecting the error

Errors produced by `autodie` aren't just strings, they're complete objects, and can be inspected to reveal a wealth of information. For example, take the following snippet of code:

```
use POSIX qw(strftime);

eval {
        use autodie;

        # Create a report directory based upon the date.
        my $date = strftime("%Y-%m-%d",localtime);
        mkdir("reports/$date");

        # Create our 'total-report.txt' inside.
        open(my $report_fh, '>', "reports/$date/total-report.txt");

        # Open our index of files to process.
        open(my $index_fh, '<', $INDEX_FILENAME);

        my $sum = 0;

        # Walk through each file in our index...
        while ( my $filename = <$index_fh> ) {

                open(my $fh, '<', $filename);

                # Examine each line in each file.  If we find
                # a "Widgets" line, add it to our total.
```

```
                while ( <$fh> ) {
                        if ( /Widgets: (\d+)/ ) {
                                $sum += $1;
                        }
                }
        }

        # Print our reports and close.
        print {$report_fh} "$sum widgets produced\n";

        close($report_fh);

};
```

There are lots of things that can go wrong with our widget summation code. We're creating directories, opening files all over the place, and even writing to one of them. Even the call to `close` can fail if we can't flush our buffers to disk. But when sometimes does go wrong, how do we know what it is? We *inspect* it:

```
if (my $error = $@) {

        if ($error->matches('open')) {

                # We can access the arguments to our failed function

                my $args = $error->args;

                # Because we're always using the 3-argument open, we can
                # grab the filename easily.  If we were unsure, we could
                # use $args->[-1] instead for the last argument.

                my $filename = $args->[2];

                # We can also inspect other useful information.

                my $file     = $error->file;
                my $function = $error->function;
                my $package  = $error->package;
                my $caller   = $error->caller;
                my $line     = $error->line;

                die "Error occurred opening $filename at $file, $line\n";
        }
        elsif ($error->matches('close')) {

                # Do error recovery for close failure here.

        }
}
```

For more information about inspecting errors, see the `autodie::exception` documentation at http://search.cpan.org/perldoc?autodie::exception.

The problem with system

When looking for the greatest disparity between the ease of calling and the difficulty of checking for errors, Perl's in-built `system` function takes the cake. Using `system` to call a command is simple:

```
system("mount /mnt/backup");
```

However the recommended error checking from `perldoc -f system` is far from trivial:

```
if ($? == -1) {
        print "failed to execute: $!\n";
}
elsif ($? & 127) {
        printf "child died with signal %d, %s coredump\n",
        ($? & 127),  ($? & 128) ? 'with' : 'without';
}
else {
        printf "child exited with value %d\n", $? >> 8;
}
```

The matter becomes even more complicated when one wishes to look up signal names (not just numbers).

Unlike `Fatal`, which doesn't support `system` at all, the new `autodie` pragma supports it provided the optional `IPC::System::Simple` module is installed. Because of the optional dependency, and because enabling the functionality can interfere with the rarely used indirect syntax of `system`, this feature needs to be explicitly enabled:

```
    use autodie qw(system);     # ONLY enable autodie for system

    use autodie qw(:system);    # ONLY enable autodie for system and exec

    use autodie qw(:default :system); # Enable for defaults, and system/exec

    use autodie qw(:all);       # The same, but less typing.
```

This then allows access to the same exception handling methods that we have with the other built-ins. Presently there is no way to interrogate the object for the command, exit status an other specific `system` information.

Hinting interface

`autodie` 2.0 supports a hinting interface for user-defined subroutines. Put simply, if you have a user-defined subroutine that does something funny to signify failure, you can now tell autodie about that. Once it knows, it can Do The Right Thing when checking your subroutine. You can even put the hints into the same file as those subs, and if someone is using autodie 2.00, it will find the hints and use them.

This means that a lot of really ugly error-checking code, both on the CPAN and the DarkPAN, can go away. Lexically.

Let's pretend you're working on a piece of legacy code. For some reason, the people who wrote this code decided the best way to signal errors is by returning the list (undef, "Error message"). I don't know why, but I've seen this anti-pattern emerge independently in three 100k+ line projects I've been involved in.

```
sub some_sub {
        if ( not batteries_full() ) {
                return ( undef, "insufficient energy" );
        }

        if ( not coin_inserted() ) {
                return ( undef, "insufficient credit" );
        }
```

```
        my @results = some_calculation();

        return @results;
}
```

If you want to check to see if some_sub() returns an error, you need to capture its return values, look at the first one to see if it's undefined, and if it's not, use the second one as your error. At least, that's what you're supposed to do.

What actually happens is most developers decide that's way too hard, and don't bother checking for errors. Then one day, the batteries on your doomsday-asteroid-destroying-satellite go flat, nobody notices, and through an ironic twist of fate you're left as the last known human survivor, and there are zombie hordes and walking killer plants outside.

Setting hints

With autodie::hints we can give autodie hints about how the return values are checked. They look like this:

```
use autodie::hints;

autodie::hints->set_hints_for(
        'Some::Package::some_sub' => {
                scalar => sub { 1 },
                list   => sub { @_ == 2 and not defined $_[0] },
        },
);
```

Our hints here are simple subroutines. If they return true, our subroutine has failed. If they return false, it executed successfully.

Notice that our scalar hint always returns true. That's because we consider any call of our subroutine in scalar context to be a mistake. It's returning a list of values, and you should be checking that list. As autodie always checks the return value of a subroutine, no subroutine is ever called in void context. Thus if your code uses `some_sub()` in a void context, autodie will re-cast that call into scalar context, and in this case throw an exception.

Auto-dying based on hints

Once we've set our hints, we can then use autodie to automatically check if we're successful:

```
use Some::Module qw(some_sub);

sub target_asteroid {

        use autodie qw( ! some_sub );

        # autodie has lexical scope, so only calls to some_sub inside
        # the target_asteroid subroutine are affected.

        my @results = some_sub();     # Succeeds or dies
}

sub target_ufo {
        my @results = some_sub();

        # autodie is out of lexical scope, so we have to manually
        # process @results here.
```

```
}
```

Insisting on hints

If you're wondering what that exclamation mark means, it means "insist on hints", and is a new piece of syntax with autodie 2.00. If for any reason autodie can't find the hints for some_sub, our code won't compile. That's a very good thing, and avoids us having a false sense of security if we use autodie on an unhinted sub.

Better error messages

However the error messages from autodie aren't really that useful. They're going to be things like "Can't some_sub() at space_defense.pl line 53". There's a noticeable lack of explanation as to why some_sub() failed.

Luckily, since the way early versions of autodie, we've been able to register message handlers. And with the new features in autodie 2.02, we can produce very rich messages. Let's see how!

```
use autodie::exception;

autodie::exception->register(
        'Some::Module::some_sub' => sub {
                my ($error) = @_;

                if ($error->context eq "scalar") {
                        return "some_sub() can't be called in a scalar context";
                }

                # $error->return gives a list of everything our failed sub
                # returned.  We know this particular sub puts the error
                # message the second argument (index 1).

                my $error_msg = $error->return->[1];

                return "some_sub() failed: $error_msg";
        }
);
```

Now, whenever some_sub() fails, it'll print a genuinely useful message, like "some_sub() failed: Insufficient energy at space_defense.pl line 53". Yes, autodie automatically adds the file and line number for you. Nice!

Manual hints for other modules

But wait, there's more! We don't want to see this sort of code floating around in your programs. You may be dealing with other people's modules that you can't modify, so we can't hide all this configuration in there. So, we can write our own pragma that contains all this info. Here's the full code for a theoretical my::autodie pragma, and is the exact same code used by the t/blog_hints.t file in autodie's test suite.

```
package my::autodie;
use strict;
use warnings;

use base qw(autodie);
```

```
use autodie::exception;
use autodie::hints;

autodie::hints->set_hints_for(
        'Some::Module::some_sub' => {
                scalar => sub { 1 },
                list   => sub { @_ == 2 and not defined $_[0] }
        },
);

autodie::exception->register(
        'Some::Module::some_sub' => sub {
                my ($E) = @_;

                if ($E->context eq "scalar") {
                        return "some_sub() can't be called in scalar context";
                }

                my $error = $E->return->[1];

                return "some_sub() failed: $error";
        }
);

1;
```

It works exactly the same as regular autodie, except it also knows how to handle some_sub(), and display good looking error messages. Here's how we'd use it:

```
use Some::Module qw(some_sub);
use my::autodie qw( ! some_sub );

my @results = some_sub();  # Succeeds or dies with a useful error!
```

Further information

The `autodie` pragma is extremely flexible; it can be sub-classed to add extra features, or change how exceptions are generated. To learn more about `autodie`, see its documentation on the CPAN at http://search.cpan.org/perldoc?autodie.

`autodie` works on Perl 5.8 and 5.10, and is due to become a core module with Perl 5.10.1.

A number of blog posts regarding autodie can be found in Paul Fenwick's blog at http://pjf.id.au/blog/toc.html?tag=autodie .

Chapter 8. Profiling

Why profile?

Profiling is used to determine which parts of your code are taking up the most execution time. This allows us to best decide the which locations to focus our optimising efforts. For example, pretend some code uses two subroutines A and B. Through testing we determine that subroutine A takes 50 time-units to execute whereas subroutine B takes only 2 time-units.

The naive approach would be to assume that our optimising efforts are best spent on speeding up subroutine A. However, if we profile our subroutines when used under normal conditions, we may discover that subroutine A is only being called once per program invocation. On the other hand, subroutine B may be called hundreds of times. With such results, a 25% speed increase to subroutine B is likely to significantly outweigh a 50% speed increase to subroutine A.

Good profiling tools will provide information both on which subroutines take the longest to execute and which subroutines are called most often. Should you need to optimise your code, use these tools to identify potential bottlenecks and focus your efforts where they can do most good.

Profiling Perl

There are a few tools to profile Perl code and which one is *best* depends on your current focus. A commonly used one is `DProf` and its friend `dprofpp` (these both come standard with Perl). To use these type:

```
% perl -d:DProf some_program.pl
% dprofpp
```

This will give you results similar to:

```
Total Elapsed Time = 1.229688 Seconds
  User+System Time = 0.559792 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 85.7   0.480  0.480    100   0.0048 0.0048  main::sub_b
 14.2   0.080  0.560      1   0.0799 0.5599  main::sub_a
 0.00   0.000 -0.000      1   0.0000      -  main::BEGIN
 0.00   0.000 -0.000      1   0.0000      -  strict::import
 0.00   0.000 -0.000      1   0.0000      -  strict::bits
```

The first two lines tell us how many seconds the program took to run. The first line says that the program finished 1.2 seconds after we started it and the second line tells us that had the the code been the only other process on the machine, it would have taken only 0.56 seconds. For the remaining 0.64 seconds the CPU was working on other things.

The first column is the percentage of time of total program invocation that the subroutine took. The exclusive seconds (second column) mark the time that the subroutine used itself without including data from subroutines it called. The cumulative seconds (third column) represent the entire time for the subroutine including further subroutine calls.

By looking at these columns we can determine that our `sub_b` subroutine was called 100 times, took an average of 0.0048 seconds each time for a total of 0.48 seconds. This resulted in it taking up 85.7% of the program execution time.

On the other hand, each call to `sub_a` took 0.08 seconds (exclusively), 16 times longer than `sub_b`. However, as `sub_a` was only called once this resulted in `sub_a` only taking up 14.2% of program execution time.

Removing `sub_a` would yield a 14% improvement. Increasing it's speed by 50% would yield a speed up of 7%. However, improving `sub_b` by only 25% would result in a massive 29% speed up for our program.

These results tell us that although `sub_a` takes up much more time, per call, than `sub_b`, `sub_b` still makes the best first candidate for optimisation.

Further information

For further information about profiling Perl read the perl.com article:
http://www.perl.com/pub/a/2004/06/25/profiling.html

Benchmarking basics

A warning about premature optimisation

Perl is fast. Most of the time the code you write will usually be fast enough. Further, it is possible to waste more time benchmarking a few approaches to a problem than you can ever save in the life of that program.

As a result, it is usually best to just write the code rather than worrying about optimisations. In fact most of the time you can leave thinking about optimisations until you've found that speed is an issue. Once it is an issue though, make sure you use Perl's profiling tools to make sure you're optimising the right code before you start benchmarking. We provide some information on this in our Profiling tip: http://perltraining.com.au/tips/2005-07-04.html

If you have a lot of time to waste however, `Benchmark.pm` is a great way to prove to your colleagues that your approach is faster than theirs.

What is benchmarking?

Benchmarking allows us to measure code execution time. If we measure two or more ways of solving the same problem then we can compare the execution times to determine which one is the faster. The fastest code is also the most efficient.

Why is benchmarking important?

Benchmarking gives us information on how well our code will scale. For example searching for an element in a small list is pretty fast, no matter how we do it. However if we repeat that search hundreds or thousands of times, then we may discover small differences between the execution times of each approach. These small differences combine to be large differences if there are enough executions.

Benchmarking in Perl

The `Benchmark.pm` module comes standard with Perl and provides a great interface for a number of comparison options. This tip does not cover all that `Benchmark` can do.

Benchmark provides two kinds of subroutines. You can use `timethis`, `timeit` and `countit` to examine the speed of a single code snippet. Alternately you can use `timethese` and `cmpthese` to run a number of code snippets. `cmpthese` prints a nice comparison chart at the end.

An example

This example compares using `grep` and `foreach` to find a random item in a list. If the first argument to `cmpthese` is zero then Benchmark will run each test for at least 3 CPU seconds each, this is a good default. We can alternately specify the number of times we want it to run with a positive integer, or a different number of CPU seconds with a negative integer. The second argument in our example is a hash reference of label => subroutine reference pairs.

```
use strict;
use Benchmark qw(cmpthese);

# A big, sorted list, with 10702 elements.
my @list = (1..10000, 'a'..'zz');

cmpthese(0, {
        # Looking for a random item in @list with grep
        grep => sub {
                # A random element in the list
                my $random = $list[rand @list];
                my $found = grep { $_ eq $random } @list;
        },
        # Looking for a random item in @list linearly (with foreach)
        foreach => sub {
                # A random element in the list
                my $random = $list[rand @list];
                my $found;
                foreach (@list) {
                        if($_ eq $random) {
                                $found = 1;
                                last;
                        }
                }
        },
});
```

Running the above code gives us:

```
         Rate    grep foreach
grep    374/s      --    -42%
foreach 641/s     72%      --
```

These results can be interpreted as follows. The `grep` code snippet can be run 374 times a second, while the `foreach` loop ran 641 times per second. `foreach` was 72% faster than `grep` for this kind of data.

If this seems strange, look at the code - you'll see that we're cheating. `grep` has to search the whole of `@list` for each call, because it returns a count of *all* the occurrences for which the search block is true. Our `foreach` loop on the other hand, stops after the item is found; which on average means only searching through half of the loop.

If we change our `foreach` loop to be less efficient by removing the call to `last` we get the following results:

```
         Rate foreach    grep
foreach 305/s      --    -19%
grep    378/s     24%      --
```

which looks more fair.

Using strings instead of code references

Calling a subroutine in Perl is quite costly when compared to more basic operations such as addition and subtraction. This cost can dwarf the time required for tests which don't do much, thus `Benchmark` allows us to pass in strings for our code snippets. In the below code we change our subroutine references to single quoted strings using `q{}`.

```
# WARNING: this snippet is buggy
use strict;
use Benchmark qw(cmpthese);

# A big, sorted list, with 10702 elements.
my @list = (1..10000, 'a'..'zz');

cmpthese(0, {
        # Looking for a random item in @list with grep
        grep => q{
                # A random element in the list
                my $random = $list[rand @list];
                my $found = grep { $_ eq $random } @list;
        },
        # Looking for a random item in @list linearly (with foreach)
        foreach => q{
                # A random element in the list
                my $random = $list[rand @list];
                my $found;
                foreach (@list) {
                        if($_ eq $random) {
                                $found = 1;
                                last;
                        }
                }
        },
});
```

Running the above code gives us:

```
            Rate foreach    grep
foreach 821975/s      --    -13%
grep    945324/s     15%      --
```

Woah! Does this mean that subroutine references were slowing our code down so dramatically? Actually no. What's happened here is quite subtle. When we declared `@list` with `my` we made it a lexical variable. Due to the way `Benchmark` evaluates the code contained in strings our lexical variables are unaccessible to these snippets, instead Benchmark assumes the code snippets refer to package variables such as `@main::list` instead. Unfortunately we don't have a `@main::list` so the code snippets search an empty list many times per second. Declaring `@list` with `our` instead solves the problem:

```
use strict;
```

```
use Benchmark qw(cmpthese);

# A big, sorted list, with 10702 elements.
our @list = (1..10000, 'a'..'zz');

cmpthese(0, {
        # Looking for a random item in @list with grep
        grep => q{
                # A random element in the list
                my $random = $list[rand @list];
                my $found = grep { $_ eq $random } @list;
        },
        # Looking for a random item in @list linearly (with foreach)
        foreach => q{
                # A random element in the list
                my $random = $list[rand @list];
                my $found;
                foreach (@list) {
                        if($_ eq $random) {
                                $found = 1;
                                last;
                        }
                }
        },
});
```

This code yields:

```
         Rate    grep foreach
grep    375/s      --    -40%
foreach 621/s     66%      --
```

which is very comparable to our earlier results.

Interpretation

In all of these examples, we've found that a short-circuiting `foreach` statement is faster than `grep`. Is this meaningful? Can we say that `foreach` is definitely faster than `grep`? The answer is no. The tests above do not consider the behaviour where the element is not in the list. We also don't test other comparison operators, would there be a difference if we were comparing with a regular expression rather than `eq`?

Generalising from the results of a single set of benchmark comparisons is not always possible. The only thing we can assert from these results is that when searching for existing numbers and short strings in a medium length list using `eq`; `foreach` is faster. Thus, when you're working on benchmarks for your own code, make sure you use appropriate data and comparisons for your domain.

Hints

- Check that your code is working the way you expect, particularly if you're providing code snippets in strings. `use strict` will not complain about attempts to use package variables that are undefined. Running `cmpthese` with `1` for its first argument will help you test this faster.

- If the numbers `Benchmark` returns seem too big, something's wrong. Ask yourself whether you think it's likely that Perl really can run `foreach` over such a big list 821,975 times every second.
- If you need to open files, create big lists or hashes, initialise variables etc, do those before you start the comparison. This will simplify your test and make sure that the pre-work doesn't get counted. Of course, don't do this if they are part of your comparison.
- Start simple. Identify the issue you're trying to benchmark and remove as much extraneous code as possible. Add more code to each comparison as you need to once you're comfortable with the results you're getting.
- Differences of less than 5% can usually be ignored. Such differences can change between different runs of the benchmark code as well as between different machine set ups.

Further reading

- http://perlmonks.org/?node_id=8745 *Benchmarking Your Code* by turnstep on PerlMonks
- http://www.perlmonks.org/index.pl?node=393128 *Wasting time thinking about wasted time* by brian_d_foy on PerlMonks
- http://www.oreilly.com/catalog/9780596527242/index.html Chapter 6: Benchmarking Perl. Mastering Perl by brian d foy, O'Reilly Publishers, 2007.
- Beware of Benchmark by Abigail, Open Source Developers' Conference 2004, (link currently not available, will try to add to website version soon).

Big-O notation

Big-O notation is a mathematical construct used to describe algorithmic complexity. In particular it is allows efficiency comparisons of different algorithms for large data sets.

Typical orders are (in order of desirability):

`O(1)` also `C`

Constant time. Computation does not take appreciably longer as the data set grows. For example, a hash lookup is considered to occur in constant time.

`O(log N)`

Logarithmic. Computation time grows as N grows, but this growth is small for large N. For example doing a binary search over a sorted list to find an item.

`O((log N)^c)`

Poly-logarithmic. Computation time grows as N grows, but this extra growth in time is still small for large N.

`O(N)`

Linear. Computation time grows in equal proportion to N's growth. For example, adding a value to each item in a list is linear.

`O(N log N)`

Quasi-linear. Computation time grows only slightly faster than N's growth, for large N.

`O(N^2)` and `O(N^c)`

Quadratic and Polynomial. For `N^2`: doubling `N` will quadruple execution time, `N^3`: doubling `N` increases execution time by a factor of 8. For large N, `O(N^2)` and above rapidly become unusable.

`O(n!)`

Factorial. Almost never a good idea, large for even small N. 5! = 120.

It is an interesting property of Big-O notation that if a function can be written as a sum of other functions, then only the fastest growing function matters. This is because the growth in that function will rapidly dwarf the growth in the others. For example:

```
cN + dN^2       => O(N^2)  for any constants c and d
N^4 + N!        => O(N!)
```

Why Big-O matters

Understanding the basics of how to calculate Big-O notation can help us identify bad choices in our code and spot likely causes of slow execution time. For example consider the following attempt to find all the elements in list1 which are also in list2:

```
my @list1 = (10 .. 100);
my @list2 = (50 .. 200);

# Walk over @list1
foreach my $element (@list1) {

        # Check in @list2 to see if it's there
        foreach my $item (@list2) {

                if($item eq $element) {
                        print "$item is in both lists\n";
                }
        }
}
```

If `@list1` and `@list2` are small, this code works very quickly. However, if we use:

```
my @list1 = (10 .. 10000);
my @list2 = (5000 .. 2000000);
```

then the code takes a very long time. Let's call the size of `@list1` `N` and the size of `@list2` `M` then what we're doing is:

```
for each N
        for each M
                do something
```

You can see here that this means we walk over `@list1` once, **but** for each item we also walk over the whole of `@list2`. This means we're dealing with `O(N M)`. As the smaller list approaches the same size as the larger, this approaches `O(N^2)` which means we should investigate whether a faster possibility exists.

We can rewrite the above code as follows:

```
my @list1 = (10 .. 100);        # length N
my @list2 = (50 .. 200);        # length M

# Create a hash of all the values in @list1: O(N)
my %in_list1;
foreach my $element (@list1) {
        $in_list1{$element} = ();
}

# Walk over each element in @list2 and compare against the hash
# O(M)
foreach my $item (@list2) {
        if(exists $in_list1{$item}) {
                print "$item is in both lists\n";
        }
}
```

Now we can see that the end result is `O(N+M)` - we walk over `@list1` once, and we walk over `@list2` once. This is an `O(N)` solution so we know it's good. Note that we're achieving this trade-off by using extra memory to store our `%in_list1` hash. For very large `N` this needs to be considered.

The above style problem comes up regularly, even though it may not quite look as simple as this. For example, imagine that you have a flat file containing addresses for businesses, and you wish to display a subset of those based on a list of suburbs. You could go through the whole file for each suburb (`O(NM)`) or you could adapt the above solution to store either the addresses or suburbs in a hash (depending on your requirements) and go through each only once. The same style solution can also be used to find the complement of a list.

Ignoring constants

Big-O notation does not predict how fast something will run, but rather gives us information about how well it will scale. For example doubling the size of `N` for a `O(N^2)` problem will *quadruple* the program execution time regardless of whether the function complexity is actually `O(0.5 N^2)` or `O(10000 N^2)`.

Constants only matter when comparing functions of different complexities. For example `O(100*N)` is slower than `O(N^2 /100)` until `N` is greater than 10,000.

See also

Mathematical coverage for Big-O notation can be found on Wikipedia at: http://en.wikipedia.org/wiki/Big_O_notation.

There are also some great notes on Big-O notation on perlmonks.org

- http://www.perlmonks.org/?node_id=573138,
- http://www.perlmonks.org/?node_id=227909 and
- http://www.perlmonks.org/?node_id=25833.

Searching for items in a large list

Often we need to search for one or more items in a larger list. This tip looks at how we can do these can contrasts their time efficiency.

Searching for a single item

In this case we may have a list of usernames and we wish to find out whether the current username is in that list. Or we may have a list of acceptable file extensions and want to know whether this file matches.

In these cases there are a number of methods we can use to find this information. For those from a C background you may think of a solution like this:

```
my $found = 0;
foreach my $username (@usernames) {
        if($username eq $check_name) {
                $found = 1;
                last;
        }
}
```

Those who have used Perl's `grep` function before would probably write:

```
my $found = grep { $_ eq $check_name } @usernames;
```

We could also use `first` from the `Scalar::Util` module (which comes standard with Perl):

```
use Scalar::Util qw(first);

my $found = first { $_ eq $check_name } @usernames;
```

Each of these are fine solutions, with very similar speeds in both the cases that the sought name exists in the list, or does not. If however we are able to store our usernames in a hash, instead of a list; then checking existence becomes much faster:

```
my $found = exists $usernames{$check_name};
```

How much faster? About 100,000 times faster for a list with 10,000 items in it. Of course, this requires that you build a hash instead of an array and this takes time. For two-character long keys, building a hash takes twice the time as building an array, and it's even longer for longer keys. Creating a hash from an existing array just to test for existence is about half as fast as the examples above.

For a single look up from an existing array, the first three options are plenty fast enough.

Searching for multiple items

Let's say we have a list of items and we want to test each one for existence in a much larger list. Again, our naive C-like implementation would look like:

```
foreach my $wanted_book (@books_I_want) {
        foreach my $library_book (@library_books) {
                if($library_book eq $wanted_book) {
                        # yay my book's available
                        order_book($wanted_book);
                        last;
```

```
                }
        }
}
```

We can also do this with `grep`:

```
foreach my $wanted_book (@books_I_want) {
        if(grep { $_ eq $wanted_book } @library_books) {
                # yay my book's available
                order_book($wanted_book);
        }
}
```

Or `first`:

```
use Scalar::Util qw(first);

foreach my $wanted_book (@books_I_want) {
        if(first { $_ eq $wanted_book } @library_books) {
                # yay my book's available
                order_book($wanted_book);
        }
}
```

We can even do this by putting the library books into a hash:

```
# Add library books into a hash
my %library_books;
@library_Books{@library_books} = ();

foreach my $wanted_book (@books_I_want) {
        if(exists $library_books{$wanted_book}) {
                # yay my book's available
                order_book($wanted_book);
        }
}
```

Or alternately, putting our wanted books into a hash (notice the change in the `foreach` loop's list).

```
# Add library books into a hash
my %books_I_want;
@books_I_want{@books_I_want} = ();

foreach my $library_book (@library_books) {
        if(exists $books_I_want{$library_book}) {
                # yay my book's available
                order_book($library_book);

                # We've ordered it, delete from hash
                delete $books_I_want{$library_book};
        }

        # end if we have no more books we want
        last unless keys %books_I_want;
}
```

The efficiency of these solutions are not the same. If we have many library books (for example approximately 10,000) and a medium demand for book orders (approximately 150), one third of which do not exist in the library, then we find that `foreach` is slightly faster (26%) than `grep` which is slightly faster (26%) than `first`. Each of these take between one third and half of a second to run.

However, building the hash of all the library books and then searching is much faster (1928%) than `foreach`, while building the hash of our wanted books is even faster (160%) than that.

If we have a pre-built hash of library books (instead of the list):

```
foreach my $wanted_book (@books_I_want) {
        if(exists $prebuilt_library_books{$wanted_book}) {
                # yay my book's available
                order_book($wanted_book);
        }
}
```

it's blindingly fast at 7846% the speed of the hash of wanted books.

`Benchmark.pm` displays these results as follows:

```
      Rate   first    grep foreach lb-hash  w-hash  pre-built
 first      2.03/s      --    -20%    -37%    -97%    -99%      -100%
 grep       2.55/s     26%      --    -21%    -96%    -98%      -100%
 foreach    3.21/s     58%     26%      --    -95%    -98%      -100%
 lb-hash    65.0/s   3103%   2451%   1928%      --    -62%      -100%
 w-hash      169/s   8238%   6540%   5178%    160%      --       -99%
 pre-built 13444/s 662497% 527577% 419353%  20588%   7846%         --
```

(where lb-hash is building the hash from the library books and w-hash is building the hash from the wanted books). Note that these results will vary slightly from invocation to invocation.

We can tell by the rate that we can run the solution using `first` 2.03 times in a second, while we can use the solution using `foreach` 3.21 times in a second. The results with less library books (about 5000) are much the same.

Conclusion

There are reasons why you may have large lists instead of hashes. For example you may need the elements to be ordered, or an array may make more sense at other points of your program. However, if you are likely to be doing lots of random accesses into a list, including looking for items which do not exist, then it may be more efficient to create a hash. In our situation, even with the cost of creating a hash of all of the library books, we discovered a massive performance increase of over 1900% than using `foreach`. Using a better approach and creating a hash of our wanted books was an improvement of more than 5100% over `foreach`. If we can instead use a hash instead of a list our lookups are even faster.

Chapter 9. Packaging

Building executables with PAR

One of Perl's greatest strengths is the huge number of modules available on the CPAN. Many really tricky problems can be made quite simple with just the use of a few appropriate modules.

Unfortunately, using CPAN modules results in another problem. Using modules makes development easier, but it makes installing code much more difficult. Not only do we have to ensure that the modules we use are installed, but we also need to ensure they modules they depend upon also get installed. Worst of all, we may be writing code for machines that don't even have Perl installed at all, or a version so old that none of our modules work on it.

If this seems familiar, then PAR, the Perl Archiver, may provide relief from your installation woes. PAR allows you to bundle all of your modules, configuration files, and other dependencies into a single compressed archive, and then distribute that along with, or instead of, your code.

Creating a PAR file (simple)

At the most basic level a PAR file is just a regular zip file, but with a .par extension. You can use any zip tool to create them, such as right-clicking a file in Windows XP, and selecting "Send to -> compressed (zipped) folder", by using the 'zip' command under Unix, or by using the Archive::Zip module in Perl.

```
zip example.par Foo.pm Bar/Baz.pm
```

Using a PAR file

Before we show you how to create a PAR file, let's see how to use one. It's as simple as:

```
use PAR "example.par";
use Bar::Baz;           # Loads Bar::Baz from example.par
```

If you have a lot of .par files, you can even specify a file pattern match:

```
use PAR "/home/mylibs/*.par";
use Bar::Baz;
```

The PAR module itself has very few dependencies, and so is *very* easy to install on your target system.

Creating a PAR file (advanced)

Being able to create a .par file using nothing but your system's zip management tools is very convenient, but it still means you have the effort of finding all your dependencies and adding them to the archive. Worse still, if your dependencies change, you have to go through the whole process again.

Fortunately, we can use the PAR Packager, `pp`, to do all the hard work for us. `pp` comes as part of the `Par-Packer` distribution on CPAN, and has quite a few dependencies. However you'll only need `pp` on your build system, not your target system.

Let's start with the situation where you have a program, and you want to find all its dependencies and wrap them up into a .par file. Using pp, we can simply write:

```
pp -p -o archive.par hello.pl
```

The `-o` switch specifies our output file name, and the `-p` switch tells `pp` we're building a <.par> file, rather than anything else (like a stand-alone executable). The resulting `archive.par` contains all the dependencies for our program, and can be loaded with `use PAR "archive.par"`, or opened using a zip tool, just like an archive we built ourselves.

However the resulting `archive.par` built by `pp` also contains our original program. If we're running on a system that already has `PAR::Packer` installed, we can run our program with `parl`:

```
parl archive.par
```

Sometimes `pp` will miss a dependency, perhaps because it's loaded in a strange way, or perhaps we just want to bundle extra modules into our archive, even though our main program doesn't use them. We can always add additional modules with the `-M` switch. For example, the following adds the `IPC::System::Simple` module to our archive, as well as all the modules that `hello.pl` depends upon:

```
pp -p -M IPC::System::Simple -o archive.par hello.pl
```

Creating a stand-alone Perl program

The <pp> program is able to create stand-alone Perl programs, saving us the effort of having to include all our modules into a `.par` file, modifying our program to use it, and then shipping them both together. To do this, we simply use `-P` instead of `-p` when building our archives:

```
pp -P -o portable.pl hello.pl
```

The resulting `portable.pl` file contains `hello.pl`, and all the non-core modules it depends upon. It can be run with:

```
perl portable.pl
```

As of `PAR::Packer` 0.977, the program still depends upon `PAR` having been installed on the target system.

Creating a stand-alone executable

One of the greatest uses of `PAR` is being able to create a stand-alone executable, which will work on systems that don't even have perl installed, or an extremely old version of perl that we'd rather not use.

Building an executable is what `pp` does by default, if not directed otherwise. Of course, it will only run on the same architecture as it was built, so to build a Windows executable you'll need to use `pp` on Windows, or to build a Linux binary you'll need to use `pp` on Linux.

To use our previous examples, we could write:

```
pp -o hello.exe hello.pl          # Windows
pp -o hello hello.pl              # Linux
```

to create a stand-alone `hello.exe` file that can be run without perl, `PAR`, or any additional modules.

You can also use `pp` to include additional shared libraries, or other dependencies that your program needs. For example, on a Unix flavoured system we can include the system `libncurses.so` with:

```
pp -l ncurses -o hello hello.pl
```

For a more lengthy example of building a Windows executable which depends upon numerous `.dll` files, see the `MakeExe.pl` file for `App::SweeperBot` in the further references section below.

Conclusion

By using `PAR`, we can solve the most common problems when trying to distribute dependency-rich perl programs, with a minimum of work needed on the target system. By using `pp`, we can even distribute stand-alone perl executables to systems that don't even have perl installed at all.

`PAR` has many more features, and a very rich API, that have not been covered in this tip. See the further references section below for more information.

Further References

- The `PAR` homepage - http://par.perl.org/
- The `PAR` tutorial - http://search.cpan.org/perldoc?PAR::Tutorial
- The `pp` manual - http://search.cpan.org/perldoc?pp
- The `PAR` manual - http://search.cpan.org/perldoc?PAR
- `MakeExe.pl` for `App::SweeperBot` - http://search.cpan.org/src/PJF/App-SweeperBot-0.02/MakeExe.pl
- SweeperBot, A minesweeper playing bot, written in Perl, and packaged using `PAR` - http://sweeperbot.org/