NAME

perlmodstyle - Perl module style guide

INTRODUCTION

This document attempts to describe the Perl Community's "best practice" for writing Perl modules. It extends the recommendations found in *perlstyle* , which should be considered required reading before reading this document.

While this document is intended to be useful to all module authors, it is particularly aimed at authors who wish to publish their modules on CPAN.

The focus is on elements of style which are visible to the users of a module, rather than those parts which are only seen by the module's developers. However, many of the guidelines presented in this document can be extrapolated and applied successfully to a module's internals.

This document differs from *perlnewmod* in that it is a style guide rather than a tutorial on creating CPAN modules. It provides a checklist against which modules can be compared to determine whether they conform to best practice, without necessarily describing in detail how to achieve this.

All the advice contained in this document has been gleaned from extensive conversations with experienced CPAN authors and users. Every piece of advice given here is the result of previous mistakes. This information is here to help you avoid the same mistakes and the extra work that would inevitably be required to fix them.

The first section of this document provides an itemized checklist; subsequent sections provide a more detailed discussion of the items on the list. The final section, "Common Pitfalls", describes some of the most popular mistakes made by CPAN authors.

QUICK CHECKLIST

For more detail on each item in this checklist, see below.

Before you start

- Don't re-invent the wheel
- Patch, extend or subclass an existing module where possible
- Do one thing and do it well
- Choose an appropriate name

The API

- API should be understandable by the average programmer
- Simple methods for simple tasks
- Separate functionality from output
- Consistent naming of subroutines or methods
- Use named parameters (a hash or hashref) when there are more than two parameters

Stability

- Ensure your module works under `use strict` and `-w`
- Stable modules should maintain backwards compatibility

Documentation

- Write documentation in POD
- Document purpose, scope and target applications

- Document each publically accessible method or subroutine, including params and return values
- Give examples of use in your documentation
- Provide a README file and perhaps also release notes, changelog, etc
- Provide links to further information (URL, email)

Release considerations

- Specify pre-requisites in Makefile.PL or Build.PL
- Specify Perl version requirements with `use`
- Include tests with your module
- Choose a sensible and consistent version numbering scheme (X.YY is the common Perl module numbering scheme)
- Increment the version number for every change, no matter how small
- Package the module using "make dist"
- Choose an appropriate license (GPL/Artistic is a good default)

BEFORE YOU START WRITING A MODULE

Try not to launch headlong into developing your module without spending some time thinking first. A little forethought may save you a vast amount of effort later on.

Has it been done before?

You may not even need to write the module. Check whether it's already been done in Perl, and avoid re-inventing the wheel unless you have a good reason.

Good places to look for pre-existing modules include http://search.cpan.org/ and asking on modules@perl.org

If an existing module **almost** does what you want, consider writing a patch, writing a subclass, or otherwise extending the existing module rather than rewriting it.

Do one thing and do it well

At the risk of stating the obvious, modules are intended to be modular. A Perl developer should be able to use modules to put together the building blocks of their application. However, it's important that the blocks are the right shape, and that the developer shouldn't have to use a big block when all they need is a small one.

Your module should have a clearly defined scope which is no longer than a single sentence. Can your module be broken down into a family of related modules?

Bad example:

"FooBar.pm provides an implementation of the FOO protocol and the related BAR standard."

Good example:

"Foo.pm provides an implementation of the FOO protocol. Bar.pm implements the related BAR protocol."

This means that if a developer only needs a module for the BAR standard, they should not be forced to install libraries for FOO as well.

What's in a name?

Make sure you choose an appropriate name for your module early on. This will help people find and remember your module, and make programming with your module more intuitive.

When naming your module, consider the following:

- Be descriptive (i.e. accurately describes the purpose of the module).
- Be consistent with existing modules.
- Reflect the functionality of the module, not the implementation.
- Avoid starting a new top-level hierarchy, especially if a suitable hierarchy already exists under which you could place your module.

You should contact modules@perl.org to ask them about your module name before publishing your module. You should also try to ask people who are already familiar with the module's application domain and the CPAN naming system. Authors of similar modules, or modules with similar names, may be a good place to start.

DESIGNING AND WRITING YOUR MODULE

Considerations for module design and coding:

To OO or not to OO?

Your module may be object oriented (OO) or not, or it may have both kinds of interfaces available. There are pros and cons of each technique, which should be considered when you design your API.

In *Perl Best Practices* (copyright 2004, Published by O'Reilly Media, Inc.), Damian Conway provides a list of criteria to use when deciding if OO is the right fit for your problem:

- The system being designed is large, or is likely to become large.
- The data can be aggregated into obvious structures, especially if there's a large amount of data in each aggregate.
- The various types of data aggregate form a natural hierarchy that facilitates the use of inheritance and polymorphism.
- You have a piece of data on which many different operations are applied.
- You need to perform the same general operations on related types of data, but with slight variations depending on the specific type of data the operations are applied to.
- It's likely you'll have to add new data types later.
- The typical interactions between pieces of data are best represented by operators.
- The implementation of individual components of the system is likely to change over time.
- The system design is already object-oriented.
- Large numbers of other programmers will be using your code modules.

Think carefully about whether OO is appropriate for your module. Gratuitous object orientation results in complex APIs which are difficult for the average module user to understand or use.

Designing your API

Your interfaces should be understandable by an average Perl programmer. The following guidelines may help you judge whether your API is sufficiently straightforward:

Write simple routines to do simple things.

It's better to have numerous simple routines than a few monolithic ones. If your routine

changes its behaviour significantly based on its arguments, it's a sign that you should have two (or more) separate routines.

Separate functionality from output.

Return your results in the most generic form possible and allow the user to choose how to use them. The most generic form possible is usually a Perl data structure which can then be used to generate a text report, HTML, XML, a database query, or whatever else your users require.

If your routine iterates through some kind of list (such as a list of files, or records in a database) you may consider providing a callback so that users can manipulate each element of the list in turn. File::Find provides an example of this with its `find(\&wanted, $dir)` syntax.

Provide sensible shortcuts and defaults.

Don't require every module user to jump through the same hoops to achieve a simple result. You can always include optional parameters or routines for more complex or non-standard behaviour. If most of your users have to type a few almost identical lines of code when they start using your module, it's a sign that you should have made that behaviour a default. Another good indicator that you should use defaults is if most of your users call your routines with the same arguments.

Naming conventions

Your naming should be consistent. For instance, it's better to have:

```
display_day();
display_week();
display_year();
```

than

```
display_day();
week_display();
show_year();
```

This applies equally to method names, parameter names, and anything else which is visible to the user (and most things that aren't!)

Parameter passing

Use named parameters. It's easier to use a hash like this:

```
$obj->do_something(
 name => "wibble",
 type => "text",
 size => 1024,
);
```

... than to have a long list of unnamed parameters like this:

```
$obj->do_something("wibble", "text", 1024);
```

While the list of arguments might work fine for one, two or even three arguments, any more arguments become hard for the module user to remember, and hard for the module author to manage. If you want to add a new parameter you will have to add it to the end of the list for backward compatibility, and this will probably make your list order unintuitive. Also, if many elements may be undefined you may see the following unattractive method calls:

```
$obj->do_something(undef, undef, undef, undef, undef, undef,
1024);
```

Provide sensible defaults for parameters which have them. Don't make your users specify

parameters which will almost always be the same.

The issue of whether to pass the arguments in a hash or a hashref is largely a matter of personal style.

The use of hash keys starting with a hyphen (`-name`) or entirely in upper case (`NAME`) is a relic of older versions of Perl in which ordinary lower case strings were not handled correctly by the => operator. While some modules retain uppercase or hyphenated argument keys for historical reasons or as a matter of personal style, most new modules should use simple lower case keys. Whatever you choose, be consistent!

Strictness and warnings

Your module should run successfully under the strict pragma and should run without generating any warnings. Your module should also handle taint-checking where appropriate, though this can cause difficulties in many cases.

Backwards compatibility

Modules which are "stable" should not break backwards compatibility without at least a long transition phase and a major change in version number.

Error handling and messages

When your module encounters an error it should do one or more of:

- Return an undefined value.
- set `$Module::errstr` or similar (`errstr` is a common name used by DBI and other popular modules; if you choose something else, be sure to document it clearly).
- `warn()` or `carp()` a message to STDERR.
- `croak()` only when your module absolutely cannot figure out what to do. (`croak()` is a better version of `die()` for use within modules, which reports its errors from the perspective of the caller. See *Carp* for details of `croak()`, `carp()` and other useful routines.)
- As an alternative to the above, you may prefer to throw exceptions using the Error module.

Configurable error handling can be very useful to your users. Consider offering a choice of levels for warning and debug messages, an option to send messages to a separate file, a way to specify an error-handling routine, or other such features. Be sure to default all these options to the commonest use.

DOCUMENTING YOUR MODULE

POD

Your module should include documentation aimed at Perl developers. You should use Perl's "plain old documentation" (POD) for your general technical documentation, though you may wish to write additional documentation (white papers, tutorials, etc) in some other format. You need to cover the following subjects:

- A synopsis of the common uses of the module
- The purpose, scope and target applications of your module
- Use of each publically accessible method or subroutine, including parameters and return values
- Examples of use
- Sources of further information
- A contact email address for the author/maintainer

The level of detail in Perl module documentation generally goes from less detailed to more detailed. Your SYNOPSIS section should contain a minimal example of use (perhaps as little as one line of code; skip the unusual use cases or anything not needed by most users); the DESCRIPTION should describe your module in broad terms, generally in just a few paragraphs; more detail of the module's routines or methods, lengthy code examples, or other in-depth material should be given in subsequent sections.

Ideally, someone who's slightly familiar with your module should be able to refresh their memory without hitting "page down". As your reader continues through the document, they should receive a progressively greater amount of knowledge.

The recommended order of sections in Perl module documentation is:

- NAME
- SYNOPSIS
- DESCRIPTION
- One or more sections or subsections giving greater detail of available methods and routines and any other relevant information.
- BUGS/CAVEATS/etc
- AUTHOR
- SEE ALSO
- COPYRIGHT and LICENSE

Keep your documentation near the code it documents ("inline" documentation). Include POD for a given method right above that method's subroutine. This makes it easier to keep the documentation up to date, and avoids having to document each piece of code twice (once in POD and once in comments).

README, INSTALL, release notes, changelogs

Your module should also include a README file describing the module and giving pointers to further information (website, author email).

An INSTALL file should be included, and should contain simple installation instructions. When using ExtUtils::MakeMaker this will usually be:

perl Makefile.PL

make

make test

make install

When using Module::Build, this will usually be:

perl Build.PL

perl Build

perl Build test

perl Build install

Release notes or changelogs should be produced for each release of your software describing user-visible changes to your module, in terms relevant to the user.

RELEASE CONSIDERATIONS

Version numbering

Version numbers should indicate at least major and minor releases, and possibly sub-minor releases. A major release is one in which most of the functionality has changed, or in which major new functionality is added. A minor release is one in which a small amount of functionality has been added or changed. Sub-minor version numbers are usually used for changes which do not affect functionality, such as documentation patches.

The most common CPAN version numbering scheme looks like this:

```
    1.00, 1.10, 1.11, 1.20, 1.30, 1.31, 1.32
```

A correct CPAN version number is a floating point number with at least 2 digits after the decimal. You can test whether it conforms to CPAN by using

```
    perl -MExtUtils::MakeMaker -le 'print MM->parse_version(shift)'
'Foo.pm'
```

If you want to release a 'beta' or 'alpha' version of a module but don't want CPAN.pm to list it as most recent use an '_' after the regular version number followed by at least 2 digits, eg. 1.20_01. If you do this, the following idiom is recommended:

```
  $VERSION = "1.12_01";
  $XS_VERSION = $VERSION; # only needed if you have XS code
  $VERSION = eval $VERSION;
```

With that trick MakeMaker will only read the first line and thus read the underscore, while the perl interpreter will evaluate the $VERSION and convert the string into a number. Later operations that treat $VERSION as a number will then be able to do so without provoking a warning about $VERSION not being a number.

Never release anything (even a one-word documentation patch) without incrementing the number. Even a one-word documentation patch should result in a change in version at the sub-minor level.

Pre-requisites

Module authors should carefully consider whether to rely on other modules, and which modules to rely on.

Most importantly, choose modules which are as stable as possible. In order of preference:

- Core Perl modules
- Stable CPAN modules
- Unstable CPAN modules
- Modules not available from CPAN

Specify version requirements for other Perl modules in the pre-requisites in your Makefile.PL or Build.PL.

Be sure to specify Perl version requirements both in Makefile.PL or Build.PL and with `require 5.6.1` or similar. See the section on `use VERSION` of *"require" in perlfunc* for details.

Testing

All modules should be tested before distribution (using "make disttest"), and the tests should also be available to people installing the modules (using "make test"). For Module::Build you would use the `make test` equivalent `perl Build test`.

The importance of these tests is proportional to the alleged stability of a module. A module which purports to be stable or which hopes to achieve wide use should adhere to as strict a testing regime as possible.

Useful modules to help you write tests (with minimum impact on your development process or your time) include Test::Simple, Carp::Assert and Test::Inline. For more sophisticated test suites there are Test::More and Test::MockObject.

Packaging

Modules should be packaged using one of the standard packaging tools. Currently you have the choice between ExtUtils::MakeMaker and the more platform independent Module::Build, allowing modules to be installed in a consistent manner. When using ExtUtils::MakeMaker, you can use "make dist" to create your package. Tools exist to help you to build your module in a MakeMaker-friendly style. These include ExtUtils::ModuleMaker and h2xs. See also *perlnewmod*.

Licensing

Make sure that your module has a license, and that the full text of it is included in the distribution (unless it's a common one and the terms of the license don't require you to include it).

If you don't know what license to use, dual licensing under the GPL and Artistic licenses (the same as Perl itself) is a good idea. See *perlgpl* and *perlartistic*.

COMMON PITFALLS

Reinventing the wheel

There are certain application spaces which are already very, very well served by CPAN. One example is templating systems, another is date and time modules, and there are many more. While it is a rite of passage to write your own version of these things, please consider carefully whether the Perl world really needs you to publish it.

Trying to do too much

Your module will be part of a developer's toolkit. It will not, in itself, form the **entire** toolkit. It's tempting to add extra features until your code is a monolithic system rather than a set of modular building blocks.

Inappropriate documentation

Don't fall into the trap of writing for the wrong audience. Your primary audience is a reasonably experienced developer with at least a moderate understanding of your module's application domain, who's just downloaded your module and wants to start using it as quickly as possible.

Tutorials, end-user documentation, research papers, FAQs etc are not appropriate in a module's main documentation. If you really want to write these, include them as sub-documents such as `My::Module::Tutorial` or `My::Module::FAQ` and provide a link in the SEE ALSO section of the main documentation.

SEE ALSO

perlstyle

General Perl style guide

perlnewmod

How to create a new module

perlpod

POD documentation

podchecker

Verifies your POD's correctness

Packaging Tools

ExtUtils::MakeMaker, *Module::Build*

Testing tools

Test::Simple, *Test::Inline*, *Carp::Assert*, *Test::More*, *Test::MockObject*

http://pause.perl.org/

Perl Authors Upload Server. Contains links to information for module authors.

Any good book on software engineering

AUTHOR

Kirrily "Skud" Robert <skud@cpan.org>