59.3.2 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.

59.3.3 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.

59.4 DESIGNING AND WRITING YOUR MODULE

Considerations for module design and coding:

59.4.1 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.

According to Damian Conway, you should consider using OO:

- When the system is large or likely to become so
- When the data is aggregated in obvious structures that will become objects
- When the types of data form a natural hierarchy that can make use of inheritance
- When operations on data vary according to data type (making polymorphic invocation of methods feasible)
- When it is likely that new data types may be later introduced into the system, and will need to be handled by existing code