An Introduction to Perl Modules

Writing, Documenting and Testing Object-Oriented Perl Modules

Author:

Madison Kelly, mkelly@alteeve.com

Date:

October 29, 2009

Shameless Plug:

http://alteeve.com

Table of Contents

1. Welcome!

This talk is essentially a condensed, hopefully more approachable version of 'perlmod', 'perlobj' and 'perltoot'.

For years, Perl Modules seemed like a mysterious, inaccessible realm best left to perl gurus. This kept me from even considering writing them. It wasn't until I was had to read through the source of a set of modules that I realize how, well, accessible they are.

I want to present this talk to help other perl developers see how powerful perl modules are, and how easy it is to write them.

For reference, the modules I will introduce here are Object Oriented. Modules do not need to be so, however.

2. So Then, What is a Perl Module, Anyway?

A perl module is, at it's most basic, a collection of methods stored in a file that can be loaded by a program to provide a certain set of functions. It is very similar to how we would load a library of functions. There is a little magic in perl modules, but nothing crazy.

2.1 That's Nice, So Why not just use a library?

Perl modules are designed to be very portable blocks of code. In truth, a well written library is not really different from a module, but modules are generally designed to stand alone from the programs that use them.

2.2 So What is the Difference?

Three main things:

- Modules have their scope defined via the 'package NAME' command. Any variables and subroutines following the 'package' operator are considered to be in the package's name space. The scope refers to code in the same block, same file or same eval, same as how the 'my' operator defines a variable's scope.
- Modules generally have a “constructor method”, usually a special subroutine called “new”. This sets up the module for use by a program and returns a handle to the module, through which its methods are accessed.
- Module constructor methods receive, as their first argument, the name of the package. This is also the package's “name space”. Secondly, they usually 'bless' a variable, which simply marks that reference as being inside the name space of the package.

If this seems overwhelming just now, please don't worry. It will make more sense once the examples start flowing.

3. Let's Get Started

Let's start with a simple module that does simple math.

3.1 Name Space

First up, I need a name for my modules. I don't want to risk stepping on any existing module names, so I hopped over to '`http://cpan.org`' and did a search for names that interest me. In my case, I like the acronym 'AN' for my personal projects, so I would like to name my first module '`AN::Tut::Sample1`'. I don't see anything using the '`AN::`' root, so I will use it.

There are no special rules for naming your modules. Use whatever appeals to you and fits your project. However, if you want your module to eventually be hosted on CPAN, you may need to think twice about your module name. Specifically, the “top level”. CPAN is reluctant to give out new top-level name space and would rather you use an existing top-level name space like '`App::*`', '`Net::*`' or similar. My choice of '`AN::*`' is, admittedly, a little risky and could well bite me one day.

This is a good time to mention the double-colon you see in module names. This is nothing more than a directory deliminator. It's meant to be more portable between differing operating systems. So, to use my '`AN::Tut::Sample1`' module name as an example, this would translate to:

```
AN/Tut/Sample1.pm
```

On *nix systems or:

```
AN\Tut\Sample1.pm
```

On Microsoft systems.

When you use your module, you do not need to specify the '`.pm`' at the end of the actual module name. This is because perl knows to look for module files with this extension in its '`@INC`' variable.

3.2 Directory and Files

Before we begin, we need to decide where to store our files. Perl maintains an internal variable called '`@INC`' which contains an array of directories. When you use the '`use ...`' command to load a module, perl steps through this array looking for the module or library you are asking for. We will use:

```
/usr/share/perl5/
```

Any new directories will be under this one. So our first example will be in the directory:

```
/usr/share/perl5/AN/Tut/Sample1.pm
```

If you want to store your module in a directory not in '`@INC`', you can do so. Simply have your calling script push the directory containing your module into the '`@INC`' array in a '`BEGIN {}`' block so that it can be found **before** the interpreter goes looking for it.

3.3 Some Lingo

There are a few terms that are worth covering now.

- Reference: A reference is simply a pointer, stored in a string variable, to another string variable, an array, a hash or a code block. See '*perldoc perlref*'.
- Function: A function is another name for a subroutine, generally used for perl's various built-in functions. I often use the term function for my own subroutines out of (bad?) habit. See '*perldoc perlref*'.
- Library: This is simply a standard file with a collection of sub routines. Generally their extension is '`.lib`' and must end with '1;'.
- Module: A module is a special type of library. Exactly how it is special will be explained below. See '*perldoc perlmod*'.
- Package: A package is, essentially, a block of code with a specified name space. The scope of a package's name space is the same as the scope used by 'my ...'. See '*perldoc -f package*'.
- Method: A method is simply a sub routine in a module. The only difference is that a method expects an object reference or a package name as it's first argument. Which it gets depends on how it was called. See the first part of '*perldoc perlobj*'.
- Class: A class is simply a package with a set of methods in it's scope.
- Object: An object is simply a reference with the class it belongs to prefixed onto the front of the reference. See '*perldoc perlobj*'.
- bless: This is actually a function whose sole purpose is to associate a reference with the package it is in. What it actually does is take the package name and prefixes it to the reference. See '*perldoc -f bless*'.

If this seems a little vague just now, don't worry. Each item will be shown below, one step at a time.

4. Our First Module; AN::Tut::Sample1

Let's start off with a very simple module that provides two methods:

- A constructor method.
- A method that takes two numbers, adds them and returns the result.

4.1 Our First Script; sample1.pl

We need a simple script to load and call our method. Let me show you a completed script and the completed module, then we will step through them to show how they work.

It would probably help to copy these into two files on your own system so that you can play along.

This is the normal perl script that we will use to call our module.

/usr/share/perl5/AN/Tut/sample1.pl

```
#!/usr/bin/perl

# This just sets perl to be strict about how it runs.
use strict;
use warnings;

# Load my module.
use AN::Tut::Sample1;

# Call my constructor method and show the user what is happening.
print "Here is what happens when I call the 'new' constructor method.\n";
my $an=AN::Tut::Sample1->new();
print "Now, this is what my 'an' object looks like: [$an]\n";

# Call the method 'add' using my 'an' object.
my $added_1=$an->add(2, 2);
print "2 + 2 = [$added_1]\n";

# Call the method 'add' directly via the module.
my $added_2=AN::Tut::Sample1->add(2, 2);
print "2 + 2 = [$added_2]\n";

exit 0;
```

4.2 Our First Module; Sample1.pm

This is the actual module. It contains only two methods; the 'new' constructor method and the 'add' method.

/usr/share/perl5/AN/Tut/Sample1.pm

```
package AN::Tut::Sample1;

# This just sets perl to be strict about how it runs and to die in a way
# more compatible with the caller.
use strict;
use warnings;
use Carp;

# My constructor method
sub new
{
	# gets the Package name.
	my $class=shift;
	print "The first argument passed into my constructor method is the 'class': [$class]\n";

	# Create an anonymous hash reference for later use.
	my $self={};
	print "This is what the simple hash reference 'self' looks like at first: [$self]\n";

	bless ($self, $class);# Associate 'self' as an object in 'class'.
	print "This is what the hash ref. 'self' looks like after being 'bless'ed into this class: [$self]\n";

	return ($self);
}

# My addition method
sub add
{
	# I expect this to be called via the object returned by the constructor
	# method.
	my $self=shift;
	print "The first argument passed into my 'add' method: [$self]\n";

	# Pick up my two numbers.
```

```
	my $num_a=shift;
	my $num_b=shift;

	# Just a little sanity check.
	if (($num_a !~ /(^-?)\d+(\.\d+)?/) || ($num_b !~ /(^-?)\d+(\.\d+)?/))
	{
		croak "The method 'AN::Tut::Sample1->add' needs to be passed two numbers.\n";
	}

	# Do the math.
	my $result=$num_a + $num_b;

	# Return the results.
	return ($result);
}

1;
```

4.3 Step Through the Module; 'AN::Tut::Sample1'

Let's start by stepping through the 'AN::Tut::Sample1' file and talk about each section.

4.3.1 Function: package

The first line is the 'package' function.

```
package AN::Tut::Sample1;
```

This function sets the scope of the 'AN::Tut::Sample1' package. It is traditionally at the top of the module file, but doesn't strictly need to be. Anything within the scope of the 'package' call will be loaded by the calling script. For example, let's say I had something like this:

```
sub blah
{
	print “Interesting things!\n”;
}

package AN::Tut::Sample1;
...
```

The subroutine 'blah' would *not* be a method available in the 'AN::Tut::Sample1' package as it is outside the package scope.

4.3.2 The 'BEGIN' Block

At this time, this isn't really needed, but I like to be in the habit of setting versions in my modules. This will be useful later if you want to ensure that a script is using a particular version of a module (or higher) later.

```
# This sets the version of this file. It will be useful later.
BEGIN
{
	our $VERSION="0.1.001";
}
```

4.3.3 Setting up the Environment

The only thing of note here is the 'use Carp' module. This allows me to have the module 'die' (croak) or 'warn' (carp) in a way friendlier for the calling program. Please see 'perldoc Carp' for details.

```
# This just sets perl to be strict about how it runs and to die in a way
# more compatible with the caller.
use strict;
use warnings;
use Carp;
```

4.3.4 The Constructor Method

The constructor method is useful for storing values, setting options and so on within a given invocation of the module. I'll go into this more in the next example. We will see in the next example how we could use the constructor to record how many times a method is called and store the last arguments sent to a method. For now though, let's keep it simple.

Modules do not need this, per-se, as we will see when we call the 'add' method later.

```
# My constructor method
sub new
{
	# gets the Package name.
	my $class=shift;
	print "The first argument passed into my constructor method is the 'class': [$class]\n";

	# Create an anonymous hash reference for later use.
	my $self={};
	print "This is what the simple hash reference 'self' looks like at first: [$self]\n";

	bless ($self, $class);# Associate 'self' as an object in 'class'.
	print "This is what the hash reference 'self' looks like after being 'bless'ed into this class:
[$self]\n";

	return ($self);
}
```

Let's look at a few things now:

4.3.4.1 sub new

The name 'new' itself is nothing special. I could just as easily have called this method 'constructor', 'glarb' or 'wow'... Perl doesn't care. The only thing to be said for the word 'new' is that it is probably the most often used name for constructor methods, so it might make the most sense to users of your module.

4.3.4.2 my $class=shift;

When a method is called directly, the first argument passed into it is the name of the package. In our case, this is 'AN::Tut::Sample1'. This is the module's "class", or name space. This will be used in a moment by 'bless'.

4.3.4.3 my $self={};

This is a string variable that stores a pointer to an anonymous hash. In the real world, and as we will see in the next example, this would be where we'd store module-wide data. For now though, to keep things simple, we'll leave it empty and focus on what it looks like from invocation and through 'bless'ing.

If the syntax seems a bit off, I could have written:

```
my %self;
my $self=\%self;
```

This is exactly the same as above. If this still doesn't seem clear, please look at 'perldoc perlref'.

4.3.4.4 The 'add' Method

The biggest thing to note here is the the first argument passed into the function. I've called the variable that will pick up this argument 'self' because I anticipate that this method will be called using the object returned by my 'new' constructor method described above. This would look like:

```
# Call 'add' using the object returned by my constructor method.
my $an=AN::Tut::Sample1->new();
$an->add(2, 2);
```

In this case, the first argument passed in is the 'bless'ed hash reference.

However:

If the user calls this method directly, the first argument passed in will be the module's class, it's name space. Specifically, the name following the 'package' function. If the user does this, nothing stored in the 'bless'ed '$self' hash reference will be available. In this case, that is fine as the constructor does nothing yet. As we'll see shortly, the 'new' method will work just fine in either case.

```
# My addition method
sub add
{
	# I expect this to be called via the object returned by the constructor
	# method.
	my $self=shift;
	print "The first argument passed into my 'add' method: [$self]\n";

	# Pick up my two numbers.
	my $num_a=shift;
	my $num_b=shift;

	# Just a little sanity check.
	if (($num_a !~ /(^-?)\d+(\.\d+)?/) || ($num_b !~ /(^-?)\d+(\.\d+)?/))
	{
```

```
                croak "The method 'AN::Tut::Sample1->add' needs to be passed two numbers.\n";
        }

        # Do the math.
        my $result=$num_a + $num_b;

        # Return the results.
        return ($result);
}
```

Other than the first passed argument, the rest of this method works like any old subroutine.

4.3.4.5 Closing the Module

If you are familiar with writing libraries then this will be familiar.

When the module is loaded, it must end with '`1;`' so that the '`use AN::Tut::Sample1`' returns a success.

```
1;
```

4.4 The '*sample1.pl*' script

Before we step through the '`sample1.pl`' script itself, lets take a look at the output printed when we run it. This way, as we step through it, we can talk about the relevant output at the same time.

4.4.1 Output From Running 'sample1.pl'

This shows the output printed to the shell when the '`sample1.pl`' script is called. Notice the difference of what it passed to the 'add' method when it's called via the 'new' constructor versus being called directly? At this point, it makes no difference at all, but very shortly it will.

`/usr/share/perl5/AN/Tut/sample1.pl`

```
Here is what happens when I call the 'new' constructor method.
The first argument passed into my constructor method is the 'class': [AN::Tut::Sample1]
This is what the simple hash reference 'self' looks like at first: [HASH(0x945d880)]
This is what the hash reference 'self' looks like after being 'bless'ed into this class:
[AN::Tut::Sample1=HASH(0x945d880)]
Now, this is what my 'an' object looks like: [AN::Tut::Sample1=HASH(0x945d880)]
The first argument passed into my 'add' method: [AN::Tut::Sample1=HASH(0x945d880)]
2 + 2 = [4]
The first argument passed into my 'add' method: [AN::Tut::Sample1]
2 + 2 = [4]
```

In brief; we call the 'new' constructor method which in turn shows what happens in that method. Then I call the 'add' method using the object returned by 'new' and print the results. Secondly, I call 'add' again with the same arguments to compare what happens differently in the 'add' method when called directly.

4.4.2 Loading Modules

The first two modules are the usual, built-in modules that tell perl to be picky about what it fails on.

```
#!/usr/bin/perl

# This just sets perl to be strict about how it runs.
use strict;
use warnings;
```

Then we load our module.

```
# Load my module.
use AN::Tut::Sample1;
```

What we've done here is tell perl to look for the module 'AN::Tut::Sample1' and load anything in it's name space, specifically, anything in the scope of the 'package' function.

You probably noticed that the 'AN' sub directory was not added to the '@INC' array, but the module can still be found. You will remember that perl looks at the module name and converts the double-colons '::' into directory delimiters. Therefore, perl actually looks for 'AN/Sample1.pm' within the directories in '@INC', not simply 'Sample1.pm'.

In our module, we set a version in the 'BEGIN' block. You likely also noticed that the variable was in all capital letters. This is a special variable that we can use to ensure in our calling script that the module is of a certain version or newer. We don't actually do this here, to keep things simple, but you can modify the 'use' line above to specify a certain version. To do so, change the line to:

```
# Load my module.
use AN::Tut::Sample1 0.1.001;
```

In this case, perl will fail at compile time if the module version it finds is too old. For example, if you instead changed the version to, say, '0.1.002' and tried to run the script, you would see the error:

```
AN::Tut::Sample1 version v0.1.2 required--this is only version v0.1.1 at ./sample1.pl line 14.
BEGIN failed--compilation aborted at ./sample1.pl line 14.
```

We won't explore this further in this paper, but it was worth mentioning that we could see how this works.

4.4.3 Creating an Object to Access our Module's Methods

We want to show what happens when we call the 'add' method using the object returned by our constructor method versus calling the method directly. To do this, we must first call the constructor and store the returned object in a string variable. Once we have this, we will print the contents of the object.

```
# Call my constructor method and show the user what is happening.
print "Here is what happens when I call the 'new' constructor method.\n";
my $an=AN::Tut::Sample1->new();
print "Now, this is what my 'an' object looks like: [$an]\n";
```

You will notice once we run this script that the contents of '$an' will match the 'bless'ed version of the module's internal '$self' hash reference. The code above generates this output:

```
Here is what happens when I call the 'new' constructor method.
The first argument passed into my constructor method is the 'class': [AN::Tut::Sample1]
This is what the simple hash reference 'self' looks like at first: [HASH(0x9271880)]
This is what the hash reference 'self' looks like after being 'bless'ed into this class:
[AN::Tut::Sample1=HASH(0x9271880)]
Now, this is what my 'an' object looks like: [AN::Tut::Sample1=HASH(0x9271880)]
```

We can see that the first and last lines come from the 'sample1.pl' script and the rest of the lines are printed by 'AN::Tut::Sample1's 'new' constructor. Pay attention to how the module's class is the argument passed in, what the '$self' hash reference looks like at first and how the module's class get prefixed to this hash reference by the 'bless' function. This is how the object returned to the caller is constructed and will become the instance of this module.

4.4.4 Call the 'add' Method Via the Module's Object

First I will call the 'add' method using the object, then, we will call it again directly. Thanks to the 'print' statement in the 'add' method, you will see that the first argument passed into the method differs in either case. When called using '$an', the first argument it receives is the module's object, which again is the 'bless'ed '$self' hash reference. When called directly, you will see that the first argument is the module's class, which is the name set by the 'package' function.

4.4.4.1 Calling 'add' Via the Module's Object

Watch when this portion of 'sample1.pl' runs.

```
# Call the method 'add' using my 'an' object.
my $added_1=$an->add(2, 2);
print "2 + 2 = [$added_1]\n";
```

You will see a combination of the 'print' from the 'add' method followed by the simple 'print' above.

```
The first argument passed into my 'add' method: [AN::Tut::Sample1=HASH(0x9271880)]
2 + 2 = [4]
```

Take notice of how the argument passed into the method is the object used to call it.

4.4.4.2 Calling 'add' Directly

This time we call the 'add' method directly.

```
# Call the method 'add' directly via the module.
my $added_2=AN::Tut::Sample1->add(2, 2);
print "2 + 2 = [$added_2]\n";
```

This time, the 'add' method prints a line showing that the first argument passed in is the module's class.

```
The first argument passed into my 'add' method: [AN::Tut::Sample1]
2 + 2 = [4]
```

5. Our Second Module; AN::Tut::Sample2

This module is simply an expansion of our first module. The difference is that now we will make use of the module's object. We add five methods to this module. Two are meant to be used by the user and three are "*private methods*".

5.1 The New Module; Sample2.pm

This is the new module, 'AN::Tut::Sample2', with the five new methods and an expanded constructor.

/usr/share/perl5/AN/Tut/Sample2.pm

```
package AN::Tut::Sample2;

# This sets the version of this file. It will be useful later.
BEGIN
{
	our $VERSION="0.1.001";
}

# This just sets perl to be strict about how it runs and to die in a way
# more compatible with the caller.
use strict;
use warnings;
use Carp;

# My constructor method
sub new
{
	# gets the Package name.
	my $class=shift;

	# Now this hash reference will be used to store a counter of how many
	# times the module is called and how many times each method is called.
	my $self={
		CALL_COUNT	=>	0,
		CALLED		=>	{
			ADD		=>	0,
			SUBTRACT	=>	0,
		},
	};

	bless ($self, $class);# Associate 'self' as an object in 'class'.

	return ($self);
}

# My addition method
sub add
{
	# I expect this to be called via the object returned by the constructor
	# method. The next two arguments are the two numbers to sum up.
	my $self=shift;
	my $num_a=shift;
	my $num_b=shift;

	# Make sure that this method is called via the module's object.
	croak "The method 'add' must be called via the object returned by 'new'.\n" if not ref($self);

	# Count this call.
	$self->_count_module;
	$self->_count_method_add;
```

```
	# Just a little sanity check.
	if (($num_a !~ /(^-?)\d+(\.\d+)?/) || ($num_b !~ /(^-?)\d+(\.\d+)?/))
	{
		croak "The method 'AN::Tut::Sample2->add' needs to be passed two numbers.\n";
	}

	# Do the math.
	my $result=$num_a + $num_b;

	# Return the results.
	return ($result);
}

# My subtraction method
sub subtract
{
	# I expect this to be called via the object returned by the constructor
	# method. Then I expect a number followed by the number to subtract
	# from it.
	my $self=shift;
	my $num_a=shift;
	my $num_b=shift;

	# Make sure that this method is called via the module's object.
	croak "The method 'subtract' must be called via the object returned by 'new'.\n" if not ref($self);

	# Count this call.
	$self->_count_module;
	$self->_count_method_subtract;

	# Just a little sanity check.
	if (($num_a !~ /(^-?)\d+(\.\d+)?/) || ($num_b !~ /(^-?)\d+(\.\d+)?/))
	{
		croak "The method 'AN::Tut::Sample2->subtract' needs to be passed two numbers.\n";
	}

	# Do the math.
	my $result=$num_a - $num_b;

	# Return the results.
	return ($result);
}

# This simply returns how many times things have been called.
sub get_counts
{
	my $self=shift;
	croak "The method 'get_counts' must be called via the object returned by 'new'.\n" if not ref($self);

	# I don't actually do anything here, I just return value. The one thing
	# to note though is how I call the internal method '_count_module',
	# which will increment the overall module call to account for this
	# call, and then return the values from the 'self' hash directly so
	# that they don't increment.
	return ($self->_count_module, $self->{CALLED}{ADD}, $self->{CALLED}{SUBTRACT});
}

# My internal method to count calls to this module. Returns the current count.
sub _count_module
{
	my $self=shift;

	# Increment by one.
	$self->{CALL_COUNT}++;

	return ($self->{CALL_COUNT});
}

# My internal method to count calls to the 'add' method.
sub _count_method_add
{
	my $self=shift;
```

```
	# Increment by one.
	$self->{CALLED}{ADD}++;

	return ($self->{CALLED}{ADD});
}

# My internal method to count calls to the 'add' method.
sub _count_method_subtract
{
	my $self=shift;

	# Increment by one.
	$self->{CALLED}{SUBTRACT}++;

	return ($self->{CALLED}{SUBTRACT});
}

1;
```

5.2 Public Versus Private Methods

> "*Perl doesn't have an infatuation with enforced privacy. It would prefer that you stayed out of its living room because you weren't invited, not because it has a shotgun*" - Larry Wall

In truth, there is no difference between "public" and "private" methods. This is entirely a "by convention" concept. If you wish, you can call a private method, though you do so at your own risk.

5.2.1 Public Methods

Public methods are the "normal" methods in that they are well documented. Good module programmers do all that they can to not change the arguments or returned values of a method so that programs that use them won't break as the method changes over time.

In this module, there are four public methods.

- 'new' The constructor method.
- 'add' The addition method that takes two numbers and returns the sum of those numbers.
- 'subtract' The subtraction method that takes two numbers and returns the subtracted value.
- 'get_counts' The method that takes no arguments and returns how many times the module has been called and how many times the 'add' and 'subtract' methods where called.

5.2.2 Private Methods

Private methods are intended to be used internally within the module itself. By convention, private methods begin with an underscore, though this is not a requirement. There is rarely public documentation and little effort is made towards backwards compatibility.

In this module, there are three private methods.

- '_count_module' This method counts each time the module is called.
- '_count_method_add' This counts each time the 'add' method is called.
- '_count_method_subtract' This counts each time the 'subtract' method is called.

5.2.3 The 'bless'ed '$self' Hash Reference, redux

Throughout this document, we will use '$self' as our object and it will be a hash reference. Note though that you could use any reference; An array reference, a simple string reference, etc. Likewise, you can use a name other than '$self'. All that really matters is that it is 'bless'ed into your package.

When the object is used to by the user to call a method, the object itself becomes the first argument passed into the called method. In this module, we are creating a somewhat artificial use for the object's hash reference; We use it to store integer values representing how many times different methods are called.

You will notice that in any method called using the object, the '$self' hash keys in the constructor can be directly accessed. Doing so is considered bad form though. So right off the bat we will create private methods to manipulate the contents of this hash. The reasoning for this extra overhead is protection of the data contained within the hash and to create a single point of manipulation for any given key. This becomes particularly important in complex methods where the values stored in the hash are used to control how things work.

5.3 Stepping Through the New Module and Script

There isn't much changed from the user's point of view, so we will just look at the important bits.

In the real world, this would be the next version of the same module. For this reason, we will be careful to not change what arguments are accepted or what values are returned. That is exactly the goal of modules. Portable, backwards compatible code!

5.3.1 The Constructor Method

The first real difference with the new module is the '$self' hash reference. Now there are keys and values where the first module's '$self' hash reference was empty.

```
	my $self={
		CALL_COUNT	=>	0,
		CALLED		=>	{
			ADD		=>	0,
			SUBTRACT	=>	0,
		},
	};
```

You might have noticed that these keys are in all capital letters. There is no requirement for this, but it is recommended to make it easier to see at a glance that you are accessing keys in the 'bless'ed hash.

These keys will be used to keep a running count of how many times the module is called. Every call to any method will be recorded in '$self->{CALL_COUNT}', how many times the 'add' and 'subtract' methods are called will be stored in '$self->{CALLED}{ADD}' and '$self->{CALLED}{SUBTRACT}' respectively.

5.3.2 The Updated 'add' Method

From a functional point of view, 'add' still does exactly the same thing it did before. This way, if our user upgrades to the new version of this module, their old calls to this method will still work.

There are three key internal differences:

- In the first module, we recommended that users access our method using the module's object returned by the constructor. Now we require it and will return an error if it is called directly.
- We call the '_count_module' private method which will record the call to the module.
- We call the '_count_method_add' private method which will record the call to the 'add' method.

5.3.2.1 Requiring Calls be Via the Module's Object

We want to make sure that every call to our module and the 'add' method are counted. This can only occur if the module is called using the module's '$an' object as this is how we get access to our 'bless'ed hash reference.

```
	# Make sure that this method is called via the module's object.
	croak "The method 'add' must be called via the object returned by 'new'.\n" if not ref($self);
```

If the user tries to directly access this method, they will see this error:

```
The method 'add' must be called via the object returned by the 'new' constructor method.
 at ./sample2.pl line 36
```

This works because the 'ref()' function will return the module's class if '$self' is indeed our hash reference. Otherwise '$self' contains the module's class name directly so 'ref()' returns nothing, triggering the 'croak'.

5.3.2.2 Automatically Counting the Method and Module Call

Before we get to work on the numbers, we count the call to the module and this method by making two calls to the relevant private methods.

```
	# Count this call.
	$self->_count_module;
	$self->_count_method_add;
```

The rest of this method is the same as before.

5.3.3 The New 'subtract' Public Method

This nearly identical to the 'add' method. The only difference is that is subtracts the second number from the first.

5.3.4 The New '_count_module' Private Method

This private method is called automatically by all three public methods. This way we can keep a count of how many times the module was called in any way.

```
sub _count_module
{
	my $self=shift;

	# Increment by one.
	$self->{CALL_COUNT}++;

	return ($self->{CALL_COUNT});
}
```

It simply increments the object's 'CALL_COUNT' hash key and the returns the same.

5.3.5 The New '_count_method_add' Private Method

This private method is called automatically when the 'add' method is called.

```
sub _count_method_add
{
	my $self=shift;

	# Increment by one.
	$self->{CALLED}{ADD}++;

	return ($self->{CALLED}{ADD});
}
```

Like in the '_count_module' above, this simply increments the object's '{CALLED}{ADD}' hash key and then returns it.

5.3.6 The New '_count_method_subtract' Private Method

This private method is almost identical the '_count_method_add' except that it is called by the 'subtract' public method. It increments and returns the object's '{CALLED}{SUBTRACT}' hash key

5.3.7 The New 'get_counts' Public Method

This method returns three values;

- How many times the module has been called, including the current call. It does this by calling '_count_module' in the 'return ()'.
- How many times the 'add' method was called. In this case, we directly call and return the object's hash key because calling '_count_method_add' would increment the value improperly.
- How many times the 'add' method was called in the same way as above.

5.4 The New Script; Sample2.pl

This is the updated script that will demonstrate the new methods in this module.

/usr/share/perl5/AN/Tut/sample2.pl

```
#!/usr/bin/perl

# This just sets perl to be strict about how it runs.
use strict;
use warnings;

# Load my module.
use AN::Tut::Sample2;

# Call my constructor method and show the user what is happening.
print "Starting 'sample2.pl'\n";
my $an=AN::Tut::Sample2->new();

# Call the method 'add' using my 'an' object.
my $added_1=$an->add(2, 2);
print "2 + 2 = [$added_1]\n";

# Call the method 'add' using my 'an' object.
my $subtract_1=$an->subtract(9, 4);
print "9 - 4 = [$subtract_1]\n";

# Now show the counts.
my ($module_count, $add_count, $subtract_count)=$an->get_counts();
print "Method call counts; All: [$module_count], add: [$add_count], subtract: [$subtract_count]\n";

# Change this to '1' to see how calling this method directly will trigger an
# error.
if (0)
{
        my $added_2=AN::Tut::Sample2->add(2, 2);
        print "2 + 2 = [$added_2]\n";
}

exit 0;
```

This is pretty similar to 'sample1.pl'. We see that the 'add' method is still called the same way and we've got a call to the new 'subtract' method.

The first real difference is the call to the 'get_counts' method to show how many times we called the module and it's public methods. The second is the disabled, illegal call to 'add'. If you change 'if (0)' to 'if (1)' and try running the script you will see how the 'add' method will 'croak' not having been called via the module's object.

5.5 Running 'sample2.pl'

This is what we see when we run 'sample2.pl'.

/usr/share/perl5/AN/Tut/sample2.pl

```
Starting 'sample2.pl'
The first argument passed into my 'add' method: [AN::Tut::Sample2=HASH(0x92e3f20)]
2 + 2 = [4]
9 - 4 = [5]
Method call counts; All: [3], add: [1], subtract: [1]
```

It's a lot cleaner than 'sample1.pl' and closer to what your user might actually expect to see.

6. Using Sibling Modules

At some point you may decide that you have too many methods, or you will have methods that are too dissimilar to keep them in the same module. At this point you will want to break up your methods into a suite of modules in a common name space. These modules are then referred to as “sibling” modules.

This begs the question; How do you put methods in different modules while still allowing those methods to call one another? There are many ways to split up your modules, and no one way is particularly better than another. We will see here one method that is some what object oriented.

6.1 Overview

This section will use almost all new methods and a new file layout. Before we jump into it, I would like to give a brief overview of how the script and modules will work to help you follow along easier.

Access to all modules and their methods will be done via a single “parent” module. This is in fact just another sibling module, and is being called a “parent” only to help you visualize it's role.

When the user calls the parent's constructor method, it automatically calls the constructor methods of all the sibling modules and then stores the returned objects as internal variables. Then the constructor calls the method '_parent' in each sibling, passing into the child module the handle to itself. Finally, the parent returns it's own blessed reference, it's object, to the user.

The parent module offers a method for each sibling module with a matching name whose sole purpose is to return to the user it's handle on the child module's object. It is through these methods that the user will access the methods in each siblings modules. This is also how each child module will access it's sibling's methods.

This will all make more sense once we step through the new script and modules.

6.1.1 The New Suite

This example suite uses many files. They are:

`sample3.pl`
: This is the new sample script that will use the new module suite.

`AN::Tut::Tools`
: This is the “parent” method. Access to all child modules will be through this module.

`AN::Tut::Tools::Math`
: This is a module that provides mathematical methods. It provides the same methods as before plus one new method.

`AN::Tut::Tools::Say`
: This is a module that provides a localization method.

`Tools.pod`
: This is the POD documentation for the `AN::Tut::Tools` module. We will cover documentation in the next section.

`Math.pod`
: This is the POD documentation for `AN::Tut::Tools::Math` module.

`Say.pod`
: This is the POD documentation for `AN::Tut::Tools::Say` module.

`test.pl`
: This is a script used for testing our modules. We will cover tests more in the last section.

`t/Math.t`
: This is the test script for testing the `AN::Tut::Tools::Math` methods.

`t/Say.t`
: This is the test script for testing the `AN::Tut::Tools::Say` methods.

6.1.2 The Parent Module; AN::Tut::Tools

Let's start by looking at the full “parent” module:

/usr/share/perl5/AN/Tut/Tools.pm

```
package AN::Tut::Tools;

# This sets the version of this file. It will be useful later.
BEGIN
{
	our $VERSION="0.1.001";
}

# This just sets perl to be strict about how it runs and to die in a way
# more compatible with the caller.
use strict;
use warnings;
```

```
use AN::Tut::Tools::Math;
use AN::Tut::Tools::Say;

# My constructor method
sub new
{
	# gets the Package name.
	my $class=shift;

	# Now this hash reference will be used to store a counter of how many
	# times the module is called and how many times each method is called.
	my $self={
		HANDLE  =>	{
			MATH	=>	"",
			SAY	=>	"",
		},
	};

	# Associate 'self' as an object in 'class'.
	bless ($self, $class);

	# Get a handle on the other two modules.
	$self->Math(AN::Tut::Tools::Math->new());
	$self->Say(AN::Tut::Tools::Say->new());

	# Pass along my handle to the sibling modules so that they can talk to
	# this module and their other sibling.
	$self->Math->_parent($self);
	$self->Say->_parent($self);

	return ($self);
}

# This is a the public access method to the internal 'AN::Tut::Tools::Math'
# object.
sub Math
{
	my $self=shift;

	$self->{HANDLE}{MATH}=shift if defined $_[0];

	return ($self->{HANDLE}{MATH});
}

# This is a the public access method to the internal 'AN::Tut::Tools::Say'
# object.
sub Say
{
	my $self=shift;

	$self->{HANDLE}{SAY}=shift if defined $_[0];

	return ($self->{HANDLE}{SAY});
}

1;
```

This is the “main” module that your users will load to get access to all the modules and method in your new suite. It provides only three methods, 'new', the usual constructor, 'Math', which sets and/or returns a handle to 'AN::Tut::Tools::Math' and 'Say', which sets and/or returns a handle to 'AN::Tut::Tools::Say'.

6.1.2.1 Loading the Siblings

The first real difference in this module is that it loads the sibling modules itself.

```
use AN::Tut::Tools::Math;
use AN::Tut::Tools::Say;
```

By doing this, the user will not need to load all of the modules in our suite themselves. By loading just 'AN::Tut::Tools', they will have access to all the methods in all our modules.

6.1.2.2 Getting a Handle on our Siblings

Simply loading our siblings isn't enough because we are building object oriented style methods. We will need to call each sibling module's constructor method and store the returned object.

In this case, we will store these objects in the hash keys below:

```
	my $self={
		HANDLE =>	{
			MATH	=>	"",
			SAY	=>	"",
		},
	};
```

The name of the keys is not important. The only thing that matters is that the objects are stored in the 'bless'ed hash reference. For this reason, we call their constructors *after* we bless '$self'.

```
	# Get a handle on the other two modules.
	$self->Math(AN::Tut::Tools::Math->new());
	$self->Say(AN::Tut::Tools::Say->new());
```

Notice that the call to each sibling's constructor is contained withing the argument list for the 'Math' and 'Say' methods? This is just a lazy way to get the module's object and to pass it to the relevant method in one go.

6.1.2.3 Passing on our 'bless'ed Reference to our Siblings

The last thing our constructor method does is call a special, internal methods called '_parent' for each of our sibling modules.

```
	# Pass along my handle to the sibling modules so that they can talk to
	# this module and their other sibling.
	$self->Math->_parent($self);
	$self->Say->_parent($self);
```

As we will see later, each sibling will store the parent's object and us it to call the parent and it's siblings' methods.

6.1.2.4 Getting a Handle on AN::Tut::Tools::Math

The 'Math' method does only two things; It returns whatever is stored in '$self->{HANDLE}{MATH}' and, if something was passed into it, it saved that argument.

```
sub Math
{
	my $self=shift;

	$self->{HANDLE}{MATH}=shift if defined $_[0];

	return ($self->{HANDLE}{MATH});
}
```

When our constructor method called 'AN::Tut::Tools::Math's 'new' constructor within the call to 'Math', the returned object got passed to this method and stored in '$self->{HANDLE}{MATH}'. Now, whenever 'Math' gets called, the object referencing 'AN::Tut::Tools::Math' gets returned.

We do it this way for the same reasons we discussed in the section; "**The 'bless'ed '$self' Hash Reference, redux**". That is, we never want to directly access or alter the values stored in '$self'. It is always better in the long run to have a single method to do this manipulation so that any future changes, tweaks or edits can be done in one place only.

6.1.2.5 Getting a Handle on AN::Tut::Tools::Say

This works exactly the same as the 'Math' call above.

```
sub Say
{
	my $self=shift;

	$self->{HANDLE}{SAY}=shift if defined $_[0];

	return ($self->{HANDLE}{SAY});
}
```

6.1.3 The 'Say' Sibling; AN::Tut::Tools::Say

This new module provides a method that can return a language-specific string suitable for display to the end user. We will use this to create English or French strings, depending on the user's preference, describing the results of a calls to 'AN::Tut::Tools::Math'.

/usr/share/perl5/AN/Tut/Tools/Say.pm

```
package AN::Tut::Tools::Say;

# This sets the version of this file. It will be useful later.
BEGIN
{
	our $VERSION="0.1.001";
}

# This just sets perl to be strict about how it runs and to die in a way
# more compatible with the caller.
use strict;
```

```
use warnings;
use Carp;


# The constructor method.
sub new
{
	my $class=shift;

	my $self={
		HANDLE_TUT_TOOLS	=>	""
	};

	bless ($self, $class);

	return ($self);
}

# Get a handle on the AN::Tut::Tools object. I know that technically that is a
# sibling module, but it makes more sense in this case to think of it as a
# parent.
sub _parent
{
	my $self=shift;
	my $parent=shift;

	$self->{HANDLE_TUT_TOOLS}=$parent if $parent;

	return ($self->{HANDLE_TUT_TOOLS});
}

# This method takes the values given to one of the Math methods, the result and
# the method called plus a language code to create a string to show the user.
sub math
{
	my $self=shift;
	my $param=shift;

	# I support two ways to pick up variables, so first I create the
	# empty or default variables here.
	my $task;
	my $num1;
	my $num2;
	my $result;
	my $lang="en";

	# Pick out the passed in parameters or switch to reading by array.
	if (ref($param) eq "HASH")
	{
		# Read in variables from the 'param' hash reference.
		$task=$param->{task};
		$num1=$param->{num1};
		$num2=$param->{num2};
		$result=$param->{result};
		$lang=$param->{lang} if exists $param->{lang};
	}
	else
	{
		# 'param' is not a hash reference, so switch to array-mode.
		$task=$param;
		$num1=shift;
		$num2=shift;
		$result=shift;
		$lang=shift if defined $_[0];
	}

	# Create the 'say' variable.
	my $say="";

	# Now choose the task
	if ($task eq "add")
	{
		# Adding, now choose the language.
```

```
		if ($lang eq "en")
		{
			# English
			$say="I added: [$num1] and: [$num2] and got: [$result].";
		}
		elsif ($lang eq "fr")
		{
			# French
			$say="J'ai additionné: [$num1] et: [$num2] et la somme est: [$result].";
		}
	}
	elsif ($task eq "sub")
	{
		# Subtracting, now choose the language.
		if ($lang eq "en")
		{
			# English
			$say="I subtracted: [$num1] from [$num2] and got: [$result].";
		}
		elsif ($lang eq "fr")
		{
			# French (thanks to Sonja Elen Kisa for correcting my French!)
			$say="J'ai soustrait: [$num1] de: [$num2] et la différence est: [$result].";
		}
	}
	else
	{
		croak "Invalid task: [$task] passed to '\$an->Say->math'. Valid task arguments are 'add' and 'sub'.\n";
	}

	return ($say);
}

1;
```

6.1.3.1 Getting a Handle on our Parent; The '_parent' Method

In this module, we will use the hash key 'HANDLE_TUT_TOOLS' in our blessed reference to store the handle to the “parent” 'AN::Tut::Tools' module. The difference this time is that this module can't go out and get the handle. If it tried to, it would get a new object that wouldn't
match the handle the user got. This is why 'AN::Tut::Tools' called this module's '_parent' method and passed in it's own blessed hash reference. The '_parent' module takes that reference and stores it. Thus, any further call to '_parent' will return the 'AN::Tut::Tools' object.

6.1.4 The 'math' Method

This method is pretty straight forward from a module point of view. There is a little trick at the start that lets this method pick up arguments via an array or a hash. This trick exists so that the testing done in the last section will be more interesting. It checks to see if the first argument passed in my the user is a hash reference. If it is, it begins looking for variables by name. If it isn't, it switches to array-type reading of arguments and copies '$param' to '$task'.

6.1.5 The 'Math' Sibling; AN::Tut::Tools::Math

Let's take a look at the whole module before looking at the new parts:

/usr/share/perl5/AN/Tut/Tools/Math.pm

```
package AN::Tut::Tools::Math;

# This sets the version of this file. It will be useful later.
BEGIN
{
	our $VERSION="0.1.001";
}

# This just sets perl to be strict about how it runs and to die in a way
# more compatible with the caller.
use strict;
use warnings;
use Carp;

# The constructor method.
sub new
{
	my $class=shift;

	my $self={
		HANDLE_TUT_TOOLS	=>	""
	};

	bless ($self, $class);

	return ($self);
}

# Get a handle on the AN::Tut::Tools object. I know that technically that is a
# sibling module, but it makes more sense in this case to think of it as a
# parent.
sub _parent
{
	my $self=shift;
	my $parent=shift;

	$self->{HANDLE_TUT_TOOLS}=$parent if $parent;

	return ($self->{HANDLE_TUT_TOOLS});
}

# My addition method
sub add
{
	# I expect this to be called via the object returned by the constructor
	# method. The next two arguments are the two numbers to sum up.
	my $self=shift;
	my $num1=shift;
	my $num2=shift;

	# Make sure that this method is called via the module's object.
	croak "The method 'add' must be called via the object returned by 'new'.\n" if not ref($self);

	# Just a little sanity check.
	if (($num1 !~ /(^-?)\d+(\.\d+)?/) || ($num2 !~ /(^-?)\d+(\.\d+)?/))
	{
		croak "The method 'AN::Tut::Sample2->add' needs to be passed two numbers.\n";
	}

	# Do the math.
	my $result=$num1 + $num2;

	# Return the results.
	return ($result);
```

```
}

# My subtraction method
sub subtract
{
	# I expect this to be called via the object returned by the constructor
	# method. Then I expect a number followed by the number to subtract
	# from it.
	my $self=shift;
	my $num1=shift;
	my $num2=shift;

	# Make sure that this method is called via the module's object.
	croak "The method 'subtract' must be called via the object returned by 'new'.\n" if not ref($self);

	# Just a little sanity check.
	if (($num1 !~ /(^-?)\d+(\.\d+)?/) || ($num2 !~ /(^-?)\d+(\.\d+)?/))
	{
		croak "The method 'AN::Tut::Sample2->subtract' needs to be passed two numbers.\n";
	}

	# Do the math.
	my $result=$num1 - $num2;

	# Return the results.
	return ($result);
}

# My addition method that calls say on it's own.
sub add_and_say
{
	# I expect this to be called via the object returned by the constructor
	# method. The next two arguments are the two numbers to sum up.
	my $self=shift;
	my $num1=shift;
	my $num2=shift;
	my $lang=defined $_[0] ? shift : "en";

	# Make sure that this method is called via the module's object.
	croak "The method 'add' must be called via the object returned by 'new'.\n" if not ref($self);

	# I use this object to get access to my sibling module's methods.
	my $an=$self->_parent;

	# Call a method in this module, but use the object from 'parent'
	# instead of 'self'.
	my $result=$an->Math->add($num1, $num2);

	# Call $an->Say to print the result.
	my $say_result=$an->Say->math({
		lang   =>     "$lang",
		task   =>     "add",
		num1   =>     $num1,
		num2   =>     $num2,
		result =>     $result
	});

	# Return the results.
	return ($say_result);
}

1;
```

This module is fairly similar to what we saw earlier in 'Sample1.pm' and the 'Sample2.pm' modules. The main difference is that the counting methods are gone and a new 'add_and_say' method has been added.

6.1.5.1 Using the Parent Handle in 'add_and_say'

The new 'add_and_say' method is actually a wrapper for two methods; The 'add' method in this module and the 'math' module in the AN::Tut::Tools::Say sibling module.

```
sub add_and_say
{
	# I expect this to be called via the object returned by the constructor
	# method. The next two arguments are the two numbers to sum up.
	my $self=shift;
	my $num1=shift;
	my $num2=shift;
	my $lang=defined $_[0] ? shift : "en";

	# Make sure that this method is called via the module's object.
	croak "The method 'add' must be called via the object returned by 'new'.\n" if not ref($self);

	# I use this object to get access to my sibling module's methods.
	my $an=$self->_parent;

	# Call a method in this module, but use the object from 'parent'
	# instead of 'self'.
	my $result=$an->Math->add($num1, $num2);

	# Call $an->Say to print the result.
	my $say_result=$an->Say->math({
		lang	=>	"$lang",
		task	=>	"add",
		num1	=>	$num1,
		num2	=>	$num2,
		result	=>	$result
	});

	# Return the results.
	return ($say_result);
}
```

The first interesting bit is:

```
	my $an=$self->_parent;
```

This calls the private method '_parent' and stores the returned object in '$an'. By doing this, we now have the object to 'AN::Tut::Tools' that matches the object returned to the user. This is particularly important in more complex programs that make use of shared variable. This object is how we will access methods in the 'AN::Tut::Tools::Math' module in the next two steps.

```
	my $result=$an->Math->add($num1, $num2);
```

We could have used 'my $result=$self->add' just effectively. The only reason I don't is that using the '$an' object allows me to move a method from one module to another module without changing any code; It makes this method more portable.

Now the magic!

```
	my $say_result=$an->Say->math({
		lang	=>	"$lang",
		task	=>	"add",
		num1	=>	$num1,
		num2	=>	$num2,
		result	=>	$result
	});
```

By using the 'Say' method in the parent module 'AN::Tut::Tools' via the '$an' object, I can call the 'math' method in 'AN::Tut::Tools::Say' from this module! This is how we can create a method that interacts with it's siblings, allowing the user to make a single method call the provides functions spanning multiple modules and methods!

6.2 Putting it All Together; sample3.pl

/usr/share/perl5/AN/Tut/sample3.pl

```
#!/usr/bin/perl

# This just sets perl to be strict about how it runs.
use strict;
use warnings;

# Load my module.
use AN::Tut::Tools 0.0.001;

# Call my constructor method.
my $an=AN::Tut::Tools->new();

# Call the 'add' method via the 'Math' method.
my $num1=10;
my $num2=12;
my $result=$an->Math->add($num1, $num2);

### Here are the different ways this method can be called.
# Using the default language 'en', English.
print $an->Say->math("add", $num1, $num2, $result), "\n";
# Specifying the language 'fr', French
print $an->Say->math("add", $num1, $num2, $result, "fr"), "\n";
# Array-type; allows for better self-documenting code and flexible argument
# order.
print $an->Say->math({
       task    =>      "add",
       num1    =>      $num1,
       num2    =>      $num2,
       result  =>      $result,
       lang    =>      "fr"
}), "\n";

# This time I do the print and math at the same time. Ya, it's ugly. :)
$num1=40;
$num2=12;
print $an->Say->math({
       task    =>      "sub",
       num1    =>      $num1,
       num2    =>      $num2,
       result  =>      $an->Math->subtract($num1, $num2),
       lang    =>      "fr"
}), "\n";

# Finally, let's call 'add_and_say' to show how one module's method can call a
# sibling module's method.
print $an->Math->add_and_say(15, 20), "\n";
# Again, but specifying a language.
print $an->Math->add_and_say(2, 18, "fr"), "\n";

exit 0;
```

We accessed both the 'Math' and 'Say' modules via the '$an' object. We only needed to load the parent module 'AN::Tut::Tools' and then use it's pointer methods. This lets us expand our module in future releases, adding entirely new modules to our suite and the user never has to worry about it. They can simply start using the new module's methods via new pointer methods we would add.

The rest of this sample script shows examples of different ways the methods can be called. These are needlessly complex, but will show different ways of writing tests in the last section of this paper.

Here's what it looks like when we run it.

```
/usr/share/perl5/AN/Tut/sample3.pl
```

```
I added: [10] and: [12] and got: [22].
J'ai additionné: [10] et: [12] et la somme est: [22].
J'ai additionné: [10] et: [12] et la somme est: [22].
J'ai soustrait: [40] de: [12] et la différence est: [28].
I added: [15] and: [20] and got: [35].
J'ai additionné: [2] et: [18] et la somme est: [20].
```

Pretty neat, eh?

7. Documentation: PODs (Plain-Old Document)

Documenting your modules can be done in a few ways. As good programmers, we already liberally sprinkle our code with comments in the program itself. We also use clear variable names to make the code as self-documenting as possible.

What about our users though? If we're writing a module, we can't expect a user to ever look at our module's source code, so we need another mechanism to make our documentation available to them. Perl has this ability through 'POD's; Plain Old Documentation.

POD documentation can exist in-line in our module or exist in a dedicated '`Module.pod`' file sitting beside our '`Module.pm`' module file. In either case, the user can then read our file by using a tool like the Linux command line '`perldoc Your::Module`'.

The syntax for PODs is a simple markup language.

7.1 In-line POD Documentation

Before we get into the actual syntax of POD, I wanted to point out the difference between in-line POD and a dedicated POD file.

When writing your POD in-line, you simply wrap the docs in '=pod' and '=cut'. Anything between these will be ignored by the perl interpreter at run time. All POD command syntax must start at the beginning of a newline with a blank line above and below it to be parsed properly.

Generally, when doing in-line documentation, a POD section will precede the method or function it relates to. This is by convention only however.

You do not need to wrap the text inside PODs unless you specifically want a certain indentation. Instead, just write each paragraph on a single line and the POD interpreted will handle line wrapping for you.

7.2 POD Markup Syntax

It is a simple markup syntax that POD compatible readers can interpret and format for display to the user. The most common interpreter is '`perldoc`', though many other POD interpreters exist to translate POD documentation into web pages and other formats.

POD syntax is pretty straight-forward. Here is a quick overview of how the syntax would be ordered, with the actual documentation still missing. This is pretty much copypasta of the '`perldoc perlpod`' documentation.

```
=pod
=encoding type
=head1 Heading Text
=head2 Heading Text
=head3 Heading Text
=head4 Heading Text
=over #
=item stuff
=back
=begin format
=end format
=for format text...
=cut
```

All command syntax requires a blank line preceding it and for the command to be at the start of the new line.

7.2.1 =pod

This starts the POD documentation. If you are using a dedicated '`something.pod`' file, this will be the first line, or at least the start of what you want the user to read. When using in-line documentation, this will end the compilation of your code until the first '`=cut`' is seen.

7.2.2 =cut

This ends a block of POD documentation. When in-line with a program, normal compilation of the script will resume after this line.

7.2.3 =head# (1 - 4)

This precedes the heading text. Any text following a '=head#' will be given a prominent font style to bring attention to the user. There are four heading levels supported, with 3 and 4 being recent additions not supported by older versions of perldoc.

Under all four heading styles, text following the heading is indented eight spaces.

7.2.3.1 =head1

Heading 1 bolds the text and does not indent the string following it. An example:

```
=head1 METHODS

This module provides methods that do foo...
```

7.2.3.2 =head2

Heading 2 bolds the text and indents the following text four spaces. For example:

```
=head 2 method_name

This method provides bar...
```

7.2.3.3 =head3

Heading 3 underlines the text and indents the following text eight spaces.

7.2.3.4 =head4

Heading 4 does not format the following text and indents it eight space.

7.2.4 =over #, =item, =back

The '=over' command starts a list of '=item's, ending with the closing '=back' command. This is useful for creating a list, for example, of parameters a method takes.

The '#' following the '=over' is an optional number of 'ems' (elements) to indent the paragraph(s) describing each item in the list. The default is 4 ems when no number is given. At the command line, this translates to the description text being incremented four spaces. By contrast, '=over 2' will increment the description text by 2 elements, or spaces when interpreted by perldoc.

An example:

```
=over

=item

foo

Foo is the first argument that 'some_method' takes and can be ...

=item

bar

Bar is the second argument that 'some_method' takes and is used to...
```

```
=item

baz

Baz is the last argument taken and is optional. It controls...

=back
```

7.2.4.1 Some notes on '=item'

If you put one bare word following '=item', the description text will start in-line after item's text. If you put a space in front of the first line of description text though, the description text will start on an indented new line with the first line of the paragraph being indented one extra space.

Alternatively, you can put a star '*' between the item command and it's text to create a bullet list (=item * foo). Likewise, if the first line in the description is a single bare word, perldoc will create a bulleted entry. Last, you can put a number followed by a period after the item and before it's text to create a numbered list (=item 1. foo). A bare word followed by '()' (=item foo()) will be underlined and the description text will start on a new line. Whichever you use, just be sure to be consistent.

Lastly, you can not use '=head#' inside an '=over' - '=back' block.

7.2.5 Code Blocks

To prevent a line from being parsed in any way, simply put a space or tab at the beginning of the line. This is an effective way to show code.

7.2.6 Other Syntax

This covers just the basics of documenting your perl module. There are many other commands and formatting options well worth reviewing in 'perldoc perlpod'! There are ways to specify the encoding used in your POD, a method for embedding other bits of data for compatible interpreters like HTML, manual formatting and embedding links.

7.3 Dedicated POD File Examples

Lets see some real-world POD files created for the our most recent suite of modules.

7.3.1 AN::Tut::Tools.pod

/usr/share/perl5/AN/Tut/Tools.pod

```
=pod

=encoding utf8

=head1 NAME

AN::Tools::Tut

This module provides access to the the other C<AN::Tut::Tools> module's methods.

=head1 SYNOPSIS

  use AN::Tut::Tools;

  # Get a common object handle on all AN::Tut::Tools::* modules.
  my $an=AN::Tut::Tools->new();

=head1 DESCRIPTION

This module provides access to C<AN::Tut::Tools::Math> and C<AN::Tut::Tools::Say>. It provides no usable methods
other than an object to access these other modules.

=head1 NOTES

All AN::Tut::Tools::* modules expects the data they receive to be in UTF-8 encoded format. Likewise, they return
UTF-8 encoded strings. If you are getting weird output or are seeing a "wide character in print" error, check that
you are not getting double-encoded UTF-8 strings by casting your data as UTF-8 in the first place.

=head1 METHODS

Below are the detailed usage instructions for the methods provided by this module.

=head2 Math

=head3 Example

  # Call 'add' in 'AN::Tut::Tools::Math'.
  my $num1=10;
  my $num2=12;
  my $result=$an->Math->add($num1, $num2);

=head3 Details

This method simply provides access to the methods provided by AN::Tut::Tools::Math. Please see that module's POD
for mode information.

=head2 Say

=head3 Example

  # Call 'math' in 'AN::Tut::Tools::Say' using the data from the 'Math' call
  # above.
  print $an->Say->math("add", $num1, $num2, $result), "\n";

=head3 Details

This method simply provides access to the methods provided by AN::Tut::Tools::Say. Please see that module's POD
for mode information.

=head1 SEE ALSO

Other modules in the AN::Tools suite:

=over

=item AN::Tut::Tools::Math

=item AN::Tut::Tools::Say

=back
```

```
=head1 LICENSE

 Copyright (c) 2009 Alteeve's Niche!. All rights reserved.
 This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

=cut
```

7.3.2 AN::Tut::Tools::Say.pod

/usr/share/perl5/AN/Tut/Tools/Say.pod

```
=pod

=encoding utf8

=head1 NAME

AN::Tut::Tools::Say

This module provides access to the translation methods.

=head1 SYNOPSIS

  use AN::Tut::Tools;

  # Get a common object handle on all AN::Tut::Tools::* modules.
  my $an=AN::Tut::Tools->new();

=head1 DESCRIPTION

This module provides methods that take arguments and variables from other module's methods and generates a string
in a given language suitable for reading by the user.

=head1 NOTES

All AN::Tut::Tools::* modules expects the data they receive to be in UTF-8 encoded format. Likewise, they return
UTF-8 encoded strings. If you are getting weird output or are seeing a "wide character in print" error, check that
you are not getting double-encoded UTF-8 strings by casting your data as UTF-8 in the first place.

=head1 METHODS

Below are the detailed usage instructions for the methods provided by this module.

=head2 math

=head3 Example

  # Do a simple addition using '$an->Math->add()' so that there is something to
  # talk about.
  my $num1=10;
  my $num2=12;
  my $result=$an->Math->add($num1, $num2);

  # Now generate and print a string to show the user using the default
  # language.
  print $an->Say->math("add", $num1, $num2, $result), "\n";

  # This time, generate a French string.
  print $an->Say->math("add", $num1, $num2, $result, "fr"), "\n";

  # Now, generate a French string using a hash to pass in arguments.
  print $an->Say->math({
        task   =>     "add",
        num1   =>     $num1,
        num2   =>     $num2,
        result =>     $result,
        lang   =>     "fr"
```

```
  }), "\n";

=head3 Details

This method takes arguments passed into and the resulting output from calls to C<AN::Tut::Tools::Math>'s C<add>
and C<subtract> methods. It then produces a string suitable for reading by the user in either English (default) or
French.

=head3 Arguments

This method can either take arguments as an array or via a hash reference. The former being shorter and easier to
type, the latter allowing for a more self-documenting format.

When call as an array, arguments must be passed in the order: C<task>, C<num1>, C<num2>, C<result> and,
optionally, C<lang>. When C<lang> is not specified, C<en> (English) is used.

When call using a hash reference the arguments need not be in any particular order. The arguments below are to be
used:

=head4 C<task>

This is either C<add> or C<sub> with no default being set. This is used as a key to tell this method what string
type to generate; "Addition" or "Subtraction".

=head4 C<num1>

This is the first number used in the equation being discussed.

=head4 C<num2>

This is the second number used in the equation being discussed.

=head4 C<result>

This is the resulting number generated by the equation.

=head4 C<lang>

Default: C<en>

This is the language code used to tell this method what language string to create. Valid arguments are:

=over

=item C<en>

English

=item C<fr>

French

=back

=head1 SEE ALSO

Other modules in the AN::Tut::Tools suite:

=over

=item AN::Tut::Tools

=item AN::Tut::Tools::Math

=back

=head1 LICENSE

 Copyright (c) 2009 Alteeve's Niche!. All rights reserved.
 This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

=cut
```

7.3.3 AN::Tut::Tools::Math.pod

/usr/share/perl5/AN/Tut/Tools/Math.pod

```
=pod

=encoding utf8

=head1 NAME

AN::Tools::Tut::Math

This module provides access to the mathematical methods.

=head1 SYNOPSIS

  use AN::Tut::Tools;

  # Get a common object handle on all AN::Tut::Tools::* modules.
  my $an=AN::Tut::Tools->new();

=head1 DESCRIPTION

This module provides methods for performing basic math.

=head1 NOTES

All AN::Tut::Tools::* modules expects the data they receive to be in UTF-8 encoded format. Likewise, they return
UTF-8 encoded strings. If you are getting weird output or are seeing a "wide character in print" error, check that
you are not getting double-encoded UTF-8 strings by casting your data as UTF-8 in the first place.

=head1 METHODS

Below are the detailed usage instructions for the methods provided by this module.

=head2 add

=head3 Example

  # Do a simple addition.
  my $num1=10;
  my $num2=12;
  my $result=$an->Math->add($num1, $num2);
  # $result is '22'.

=head3 Details

This method takes two numbers and returns the resulting addition of those two numbers. The numbers may be signed
whole integers or floating points (real) numbers.

=head2 subtract

=head3 Example

  # Do a simple subtraction.
  # talk about.
  my $num1=10;
  my $num2=12;
  my $result=$an->Math->subtract($num1, $num2);
  # $result is '-2'.

=head3 Details

This method takes two numbers and returns the result of subtracting the second number from the first. The numbers
may be signed whole integers or floating points (real) numbers.

=head2 add_and_say
```

```
=head3 Example

  # Do a simple addition, but get back the processed string in French.
  $an->Math->add_and_say(2, 18, "fr");
  # $result is 'J'ai additionné: [2] et: [18] et la somme est: [20].'.

=head3 Details

This method takes two numbers and returns a string in the specified language with the answer (English, if not
specified).

=head1 SEE ALSO

Other modules in the AN::Tools suite:

=over

=item AN::Tut::Tools

=item AN::Tut::Tools::Say

=back

=head1 LICENSE

 Copyright (c) 2009 Alteeve's Niche!. All rights reserved.
 This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

=cut
```

8. The Importance of Testing

> "*Just as there's no such thing as a bug-free program*
> *there's no program that can't be debugged. Am I wrong?*" - Ghost in the Shell

All module's and their methods need to be tested. This should be done after every change to the program. This can be extremely tedious without some testing program. Thankfully, several tools already exist to help you write a test script for your modules.

To start, read the PODs for '`Test::Tutorial`', then the '`Test::Simple`' and lastly '`Test::More`'. This section of the talk is an overview of those modules' documentation. The main difference being that I assume you understand why testing is important so I will leave the sarcasm and brow-beating behind.

8.1 Reasoning

The main reason for creating a comprehensive testing module is to help catch unanticipated consequences of changes to your modules. Ideally, your module will be used by many different people and/or in many different ways. Further, a proper test suite is required to get your modules onto CPAN.

A well designed test program calls your various module's and methods in all the possible ways you can think of. As you can guess, this means you have to write your test program to be very thorough. If you decide to leave the testing program until after you have finished your modules, you will be faced with a daunting task that you may simply never do. Like documentation, it is best to write your test suite along side of writing your modules themselves.

8.2 The Test::More module

It is generally recommended to start your first test script using the 'Test::Simple' module. However, I disagree as 'Test::More' is a drop-in replacement for 'Test::Simple' that offers a wider range of tests. You may wish to start with 'Test::Simple' and then switch when you run into it's limitations.

The two main reasons I prefer starting with 'Test::More' are;

- It supports a string describing each test.
- It has methods that go well beyond the simple 'does A match B' test. It allows for regular expression tests, 'unlike' tests and so forth.

8.3 Overview

Each test is ultimately the same; Does 'A' [equal|not equal|match|not match] 'B'. If so, the test passes. If not, the test fails. If one or more tests fail, the entire test suite fails. Later, if your build a Makefile for your modules, a failed test will prevent your module suite from installing on a user's machine.

The core of the 'Test::*' suite is the 'Test::Harness' module. All other 'Test::*' modules print output in a format that 'Test::Harness' can parse. In turn, 'Test::Harness' ensures that all planned tests are run and it handles how the test results and errors are displayed to the user.

8.4 Example; test.pl

Let's take a look at a test suite for our most recent modules.

We'll start with a normal perl script that will load 'Test::More', test against the parent module and then call test files for each of the sibling modules.

/usr/share/perl5/AN/Tut/test.pl

```
#!/usr/bin/perl -Tw

use strict;
use warnings;
use POSIX;

# Be nice and set a version number.
our $VERSION="0.0.001";

# Call in the test module
use Test::More tests => 28;
# use Test::More 'no_plan';

# Load my module via 'use_ok' test.
BEGIN
{
	print "Will now test AN::Tut::Tools on $^O.\n";
	use_ok('AN::Tut::Tools', 0.0.001);
}

# Get a handle on the main module and test it.
my $an=AN::Tut::Tools->new();
like($an, qr/^AN::Tut::Tools=HASH\(0x\w+\)$/, "AN::Tut::Tools object appears valid.");
```

```
# Build an array of methods that 'AN::Tut::Tools' should provide.
my @methods=("Math", "Say");
can_ok("AN::Tut::Tools", @methods);

### test AN::Tut::Tools::Math
print "Testing AN::Tut::Tools::Math\n";
require_ok("AN/Tut/t/Math.t");

### test AN::Tut::Tools::Math
print "Testing AN::Tut::Tools::Say\n";
require_ok("AN/Tut/t/Say.t");

exit;
```

8.4.1 Breaking it Down

Let's take a closer look at the '`test.pl`' script.

8.4.1.1 Loading Test::More

The first interesting part is:

```
# Call in the test module
use Test::More tests => 28;
# use Test::More 'no_plan';
```

When you load '`Test::More`' you can invoke it in one of two ways. Both are shown here, with the second one being commented out.

In the first case, I tell '`Test::More`', which in turn tells '`Test::Harness`', that I plan to run 28 tests. This is important because a given test could fail so badly that the test script could exit without the bad test itself returning a fail. Without a predetermined number of tests, '`Test::Harness`' will see all **run** tests as having passed and declare the tests a success. Conversely, by declaring that we will run 28 tests, and 12 run successfully before the 13th kills the script, '`Test::Harness`' will know that something went wrong and print an overall failure.

In the second commented out case, I tell '`Test::More`' that I don't have a plan. This is useful while you write your test suite. It would be very annoying to have to go back and edit that line every time you add or remove a test. Be sure to always switch back to the planned number of tests when you finish working on your test suite.

8.4.1.2 The 'use_ok()' Method

The first step in our test is to load our '`AN::Tools`' module.

```
BEGIN
{
	print "Will now test AN::Tut::Tools on $^O.\n";
	use_ok('AN::Tut::Tools', 0.0.001);
}
```

This is done in a 'BEGIN' block, as recommended by 'Test::More', so that functions are exported at compile time and that prototypes are properly honoured.

The 'use_ok' test takes one or two arguments; The first being the module to load and the second, optional argument being the minimum version of the module needed. If anything were to fail when a user ran 'use AN::Tut::Tools 0.0.001', the 'use_ok()' test would catch it.

8.4.1.3 The 'like()' and 'unlike()' Tests

These tests do a regular expression test of the first argument against the second argument.

```
my $an=AN::Tut::Tools->new();
like($an, qr/^AN::Tut::Tools=HASH\(0x\w+\)$/, "AN::Tut::Tools object appears valid.");
```

In this example, I use a regular expression to make sure that the object returned by calling the constructor method is 'bless'ed into my module's class. The 'unlike' test is simply the reverse and would fail is '$an' matched '/^AN::Tut::Tools=HASH\(0x\w+\)$/'.

8.4.1.4 The 'can_ok()' Test

This test takes an array of method names as it's sole argument and fails in any of the method names in the array do not exist.

```
my @methods=("Math", "Say");
can_ok("AN::Tut::Tools", @methods);
```

In this case, I am making sure that both the 'Math' and 'Say' methods exist.

8.4.1.5 Loading Other Test Scripts

It is customary, and expected by perl Makefiles, that test scripts exist in a '/t' subdirectory. Specifically, a 'Module.t' script is expected for each 'Module.pm' module.

```
### test AN::Tut::Tools::Math
print "Testing AN::Tut::Tools::Math\n";
require_ok("AN/Tut/t/Math.t");

### test AN::Tut::Tools::Math
print "Testing AN::Tut::Tools::Say\n";
require_ok("AN/Tut/t/Say.t");
```

The rest of the 'test.pl' script loads these test files, one for each sibling module.

8.4.2 The 'AN::Tut::Tools::Math' Module's 'Math.t' Test Script

The first test script to be run by 'test.pl' is 'Math.t':

/usr/share/perl5/AN/Tut/Tools/Math.t

```
#!/usr/bin/perl -Tw

use AN::Tut::Tools 0.0.001;
my $an=AN::Tut::Tools->new();

# Make sure that $parent matches $an.
my $parent=$an->Math->_parent();
is($an, $parent, "Internal 'parent' method returns same blessed reference as is in \$an.");

# Make sure that all methods are available.
my @methods=("_parent", "add", "subtract", "add_and_say");
can_ok("AN::Tut::Tools::Math", @methods);

# Test adding a few numbers. Start with ints, then try signed and unsigned ints
# and floats.
#  Got                               Expect  Message
is($an->Math->add(4, 8),             12,     "add(); Simple integer addition test.");
is($an->Math->add(4, 8.5),           12.5,   "add(); Integer plus float addition test.");
is($an->Math->add(5.25, 8.5),        13.75,  "add(); Float addition test.");
is($an->Math->add(15, -3),           12,     "add(); Signed integer addition test.");
is($an->Math->add(3, -15),           -12,    "add(); Signed integer addition with negative result test.");
is($an->Math->add(-15, -3),          -18,    "add(); Double-negative integer addition with negative result
test.");
is($an->Math->add(+15, -3),          12,     "add(); Signed integer addition including '+' test.");

# Now test subtracting a few numbers. Same style of tests as above.
#  Got                               Expect  Message
is($an->Math->subtract(15, 3),       12,     "subtract(); Simple integer subtraction test.");
is($an->Math->subtract(15, 2.5),     12.5,   "subtract(); Integer minus float subtraction test.");
is($an->Math->subtract(15.45, 2.5),  12.95,  "subtract(); Float subtraction test.");
is($an->Math->subtract(15, -3),      18,     "subtract(); Signed integer subtraction test.");
is($an->Math->subtract(3, -15),      18,     "subtract(); Signed integer subtraction with positive result test.");
is($an->Math->subtract(-15, -3),     -12,    "subtract(); Double-negative integer subtraction with negative result
test.");
is($an->Math->subtract(+15, -3),     18,     "subtract(); Signed integer subtraction including '+' test.");

# Now test the 'add_and_say' method.
is(
       "I added: [15] and: [20] and got: [35].",
       $an->Math->add_and_say(15, 20),
       "add_and_say(); Test default language processing of a simple integer addition."
);
is(
       "J'ai additionné: [2] et: [18] et la somme est: [20].",
       $an->Math->add_and_say(2, 18, "fr"),
       "add_and_say(); Test specified language processing of a simple integer addition."
);
```

The first part of this test script is pretty normal.

8.4.2.1 The 'is()' and 'isnt()' Tests

The first new test we see is;

```
# Make sure that $parent matches $an.
my $parent=$an->Math->_parent();
is($an, $parent, "Internal 'parent' method returns same blessed reference as is in \$an.");
```

This looks at the '$an' object and compares it with the object returned by the private method '_parent'. If they are the same, the test passes. The 'isnt()' test is simply the opposite test and would fail is the two objects matched. This is an example of how 'is()' and 'isnt()' can match normal strings. There is no separate test between numeric and string matching.

We see this test through the rest of this test files. In all cases, the first argument is what 'Test::More' expects, the second argument is what we're testing and the last argument is a description of the test.

8.4.3 The 'AN::Tut::Tools::Say' Module's 'Say.t' Test Script

The second test script to be run by 'test.pl' is 'Say.t'.

/usr/share/perl5/AN/Tut/Tools/Say.t

```
#!/usr/bin/perl -Tw

use AN::Tut::Tools 0.0.001;
my $an=AN::Tut::Tools->new();

# Make sure that $parent matches $an.
my $parent=$an->Say->_parent();
is($an, $parent, "Internal 'parent' method returns same blessed reference as is in \$an.");

# Make sure that all methods are available.
my @methods=("_parent", "math");
can_ok("AN::Tut::Tools::Say", @methods);

# First, I need something to talk about, so I will use this:
my $num1=10;
my $num2=12;
my $result=$an->Math->add($num1, $num2);

# Now, this module has only one method that takes arguments, but it can receive
# those arguments is several ways. This tests them.
is(
	$an->Say->math("add", $num1, $num2, $result),                                        # Got
	"I added: [$num1] and: [$num2] and got: [$result].",                                 # Expect
	"math(); Test default language processing using the default language, array-type call." # Message
);
is(
	$an->Say->math("add", $num1, $num2, $result, "fr"),                                  # Got
	"J'ai additionné: [$num1] et: [$num2] et la somme est: [$result].",                  # Expect
	"math(); Test processing, specifying French, array-type call."                       # Message
);
is(
	$an->Say->math({task=>"add", num1=>$num1, num2=>$num2, result=>$result, lang=>"fr"}),  # Got
	"J'ai additionné: [$num1] et: [$num2] et la somme est: [$result].",                  # Expect
	"math(); Test processing, specifying French, hash-type call."                        # Message
);
```

This test introduces nothing new. It simply uses 'is()' and 'can_ok()' to do all the testing it needs.

8.4.4 The Majority of the 'Math.t' and 'Say.t' Tests

The majority of the test scripts are a large number of 'is()' tests.

Generally, we start with the simplest tests for each method, then add new tests of growing complexity to test all of the different ways a method can be called. The limit to testing is the limit of your imagination.

This is another good reason to start your test scripts early. When you run into weird problems, often ones found by creative users, you can add tests for those conditions right away. This way, your test scripts grow in effectiveness right along with your modules growing in features.

8.5 Running 'test.pl'

With the test scripts in place, we should see something like this:

/usr/share/perl5/AN/Tut/test.pl

```
1..28
Will now test AN::Tut::Tools on linux.
ok 1 - use AN::Tut::Tools;
ok 2 - AN::Tut::Tools object appears valid.
ok 3 - AN::Tut::Tools->can(...)
Testing AN::Tut::Tools::Math
ok 4 - Internal 'parent' method returns same blessed reference as is in $an.
ok 5 - AN::Tut::Tools::Math->can(...)
ok 6 - add(); Simple integer addition test.
ok 7 - add(); Integer plus float addition test.
ok 8 - add(); Float addition test.
ok 9 - add(); Signed integer addition test.
ok 10 - add(); Signed integer addition with negative result test.
ok 11 - add(); Double-negative integer addition with negative result test.
ok 12 - add(); Signed integer addition including '+' test.
ok 13 - subtract(); Simple integer subtraction test.
ok 14 - subtract(); Integer minus float subtraction test.
ok 15 - subtract(); Float subtraction test.
ok 16 - subtract(); Signed integer subtraction test.
ok 17 - subtract(); Signed integer subtraction with positive result test.
ok 18 - subtract(); Double-negative integer subtraction with negative result test.
ok 19 - subtract(); Signed integer subtraction including '+' test.
ok 20 - add_and_say(); Test default language processing of a simple integer addition.
ok 21 - add_and_say(); Test specified language processing of a simple integer addition.
ok 22 - require 'AN/Tut/t/Math.t';
Testing AN::Tut::Tools::Say
ok 23 - Internal 'parent' method returns same blessed reference as is in $an.
ok 24 - AN::Tut::Tools::Say->can(...)
ok 25 - math(); Test default language processing using the default language, array-type call.
ok 26 - math(); Test processing, specifying French, array-type call.
ok 27 - math(); Test processing, specifying French, hash-type call.
ok 28 - require 'AN/Tut/t/Say.t';
```

The first line shows that there are 28 planned tests. The rest of the output simply shows that the test was “ok”, what the test number was and finally the descriptive text we passed in as the third argument of each test.

Done!

That's it! You now have a documented, testable suite of object oriented perl modules that can talk to each other.

You can find a copy of this paper, along with all source files at:

- http://alteeve.com

Thank you for your time!