Perl Beginners' Site

Perl - because programming should be fun.

Resources

Platforms

Common Uses

"Perl for Newbies" - Part 5 - The Perl Beginners' Site

Learn Perl Now!
And get a job doing Perl.

Show Navigation Controls

"Perl for Perl Newbies" - Part 5 - Good Programming Practices ¶

Contents ¶

Licence ¶

1. Introduction ¶

We've already introduced some good software engineering practices in the previous lectures, but this lecture is going to contain a concentrated discussion of them. We will explain the motivation for their use, and show how to implement them, while giving some “hands-on” demonstrations.

The mission of the Perl for Perl Newbies talks has been continued in our work on The Perl Beginners' site, which aims to be the premier web site for finding resources to learn about Perl.

2. Automated Testing ¶

Automated testing is a software engineering method in which one writes pieces of code, which in turn help us ascertain that the production code itself is functioning correctly. This section provides an introduction to automated software testing in Perl.

2.1. Motivation for Testing ¶

So why do we want to perform automated software testing? The first reason is to prevent bugs. By writing tests before we write the production code itself (so-called **Test-First Development**) we ascertain that the production code behaves according to the specification given in the tests. That way, bugs that could occur, if the code was deployed right away, or tested only manually, would be prevented.

Another reason is to make sure that bugs and regressions are not reintroduced in the code-base. Say we have a bug, and we write a meaningful test that fails when the bug is still in the code, and only then fix the bug. In that case, we can re-use the test in the future to make sure the bug is not present in the current version of the code. If the bug re-surfaces in a certain variation, then it will likely be caught by the test.

Finally, by writing tests we provide specifications to the code and even some form of API documentation, as well as examples of what we want the code to achieve. This causes less duplication than writing separate specification documents and examples, and, furthermore, is validated to be functional.

2.2. Demo ¶

Let's suppose we want to test a function that adds two numbers. (This is a classic example.) We have the following function in the module Add1.pm:

```
package Add1;

use strict;
use warnings;

use vars qw(@EXPORT_OK @ISA);

use Exporter;

@ISA = (qw(Exporter));

@EXPORT_OK = (qw(add));

sub add
{
    my $x = shift;
    my $y = shift;

    return 4;
}

1;
```

One way to write a rudimentary script to test it, would be the following:

```
#!/usr/bin/env perl

use strict;
```

```
use warnings;

use Add1 (qw(add));

if (!(add(2,2) == 4))
{
    die "add(2,2) failed";
}
exit(0);
```

This script will die with an ugly exception if adding 2 and 2 failed and quietly exit with a success code if everything is OK. Let's run it:

```
$ perl add1-test.pl
$
```

Everything is OK. Now let's write another test:

```
#!/usr/bin/env perl

use strict;
use warnings;

use Add1 (qw(add));

if (!(add(2,2) == 4))
{
    die "add(2,2) failed";
}

{
    my $result = add(1,1);

    if ($result != 2)
    {
        die "add(1,1) resulted in '$result' instead of 2."
    }
}

exit(0);
```

This time the test fails:

```
$ perl add1-test-2.pl
add(1,1) resulted in '4' instead of 2. at add1-test-2.pl line 18.
$
```

As a result, we need to fix the production code:

```
package Add2;

use strict;
use warnings;

use vars qw(@EXPORT_OK @ISA);
```

```
use Exporter;

@ISA = (qw(Exporter));

@EXPORT_OK = (qw(add));

sub add
{
    my $x = shift;
    my $y = shift;

    return $x+$y;
}

1;
```

And the equivalent test script is successful:

```
#!/usr/bin/env perl

use strict;
use warnings;

use Add2 (qw(add));

if (!(add(2,2) == 4))
{
    die "add(2,2) failed";
}

{
    my $result = add(1,1);

    if ($result != 2)
    {
        die "add(1,1) resulted in '$result' instead of 2."
    }
}

exit(0);
```

Now we can continue writing more tests, and see that they passed.

2.2.1. Test::More ¶

Perl ships with a module called Test::More (which is part of the Test-Simple CPAN distribution, which may be more up-to-date there), that allows one to write and run tests using convenient functions. Here's an example for a test script:

```
#!/usr/bin/env perl

use strict;
use warnings;
```

```
use Test::More tests => 7;

use Add2 (qw(add));

# TEST
is (add(0, 0),
    0,
    "0+0 == 0",
);

# TEST
is (add(2, 2),
    4,
    "2+2 == 4",
);

# TEST
is (add(4, 20),
    24,
    "4+20 == 24",
);

# TEST
is (add(20, 4),
    24,
    "20+4 == 24",
);

# TEST
is (add(-2, 8),
    6,
    "(-2)+8 == 6",
);

# TEST
is (add(4, 3.5),
    7.5,
    "4+3.5 == 7.5",
);

# TEST
is (add(3.5, 3.5),
    7,
    "3.5+3.5 == 7"
);
```

`is()` is a Test-More built-in that compares a received result ("have") to an expected result ("want") for exact equivalence. There are also `ok()`, which just tests for truth-hood, `is_deeply()` which performs a deep comparison of nested data structures, and others.

You may also notice the `# TEST` comments - these are **Test::Count** annotations that allow us to keep track of the number of test assertions that we have declared and update it.

Now, the output of this would be:

```
1..7
ok 1 - 0+0 == 0
ok 2 - 2+2 == 4
ok 3 - 4+20 == 24
ok 4 - 20+4 == 24
ok 5 - (-2)+8 == 6
```

```
ok 6 - 4+3.5 == 7.5
ok 7 - 3.5+3.5 == 7
```

This is in an output format called [TAP - The Test Anything Protocol](). There are several TAP parsers, which analyse the output and present a human-friendly summary. For example, we can run the test script above using the `prove` command-line utility that ships with perl 5:

```
$ prove Test-More-1.t
Test-More-1.t .. ok
All tests successful.
Files=1, Tests=7,  0 wallclock secs ( 0.06 usr  0.01 sys +  0.06 cusr  0.01 csys =  0.14 CPU)
Result: PASS
```

For more information refer to the following sources:

1. [Test::Tutorial on the CPAN]()
2. ["Testing with Perl" by Gabor Szabo]() - comprehensive material of a talk about Perl and testing.
3. [Test::Count]() - allows one to keep track of the number of assertions in the test file.
4. [Test-Run]() - an alternative test harness under development (with output in colour and other enhancements).
5. [The Perl Quality Assurance (QA) Project]().

2.2.2. ./Build test ¶

Standard CPAN and CPAN-like Perl packages contain their tests as a group of `*.t` under the sub-directory `t/`, and allow running them by invoking the `make test` or `./Build test` commands.

Using the CPAN package [Module-Starter]() one can generate a skeleton for one's own CPAN-like package, which can also afterwards contain tests. Keeping your code organised in such packages, allows one to make use of a convenient build-system such as [Module-Build](). It also allows one to package it as operating-system-wide packages, which can be removed easily using the system's package manager. Finally, these packages can later be uploaded to CPAN for sharing with other users and developers.

Here's an example of testing a CPAN distribution from CPAN using `./Build test`:

```
shlomi:~/TEMP$ ls
shlomi:~/TEMP$ mv ~/Test-Count-0.0500.tar.gz .
shlomi:~/TEMP$ ls
Test-Count-0.0500.tar.gz
shlomi:~/TEMP$ ls -l
total 16
-rw-r--r-- 1 shlomi shlomi 12933 2009-08-02 20:52 Test-Count-0.0500.tar.gz
shlomi:~/TEMP$ tar -xvf Test-Count-0.0500.tar.gz
Test-Count-0.0500
Test-Count-0.0500/Changes
Test-Count-0.0500/MANIFEST
Test-Count-0.0500/META.yml
Test-Count-0.0500/Build.PL
Test-Count-0.0500/Makefile.PL
Test-Count-0.0500/README
Test-Count-0.0500/t
Test-Count-0.0500/t/boilerplate.t
Test-Count-0.0500/t/03-filter.t
Test-Count-0.0500/t/01-parser.t
Test-Count-0.0500/t/pod-coverage.t
Test-Count-0.0500/t/02-main.t
Test-Count-0.0500/t/00-load.t
Test-Count-0.0500/t/pod.t
Test-Count-0.0500/t/sample-data
```

```
Test-Count-0.0500/t/sample-data/test-scripts
Test-Count-0.0500/t/sample-data/test-scripts/arithmetics.t
Test-Count-0.0500/t/sample-data/test-scripts/01-parser.t
Test-Count-0.0500/t/sample-data/test-scripts/basic.arc
Test-Count-0.0500/examples
Test-Count-0.0500/examples/perl-test-manage-helper.pl
Test-Count-0.0500/examples/perl-test-manage.vim
Test-Count-0.0500/lib
Test-Count-0.0500/lib/Test
Test-Count-0.0500/lib/Test/Count.pm
Test-Count-0.0500/lib/Test/Count
Test-Count-0.0500/lib/Test/Count/Base.pm
Test-Count-0.0500/lib/Test/Count/Parser.pm
Test-Count-0.0500/lib/Test/Count/Filter.pm
Test-Count-0.0500/lib/Test/Count/Filter
Test-Count-0.0500/lib/Test/Count/Filter/ByFileType
Test-Count-0.0500/lib/Test/Count/Filter/ByFileType/App.pm
Test-Count-0.0500/inc
Test-Count-0.0500/inc/Test
Test-Count-0.0500/inc/Test/Run
Test-Count-0.0500/inc/Test/Run/Builder.pm
shlomi:~/TEMP$ cd Test
Test-Count-0.0500/        Test-Count-0.0500.tar.gz
shlomi:~/TEMP$ cd Test-Count-0.0500
shlomi:~/TEMP/Test-Count-0.0500$ ls
Build.PL  examples  lib          MANIFEST  README
Changes   inc       Makefile.PL  META.yml  t
shlomi:~/TEMP/Test-Count-0.0500$ perl Build.PL
Checking whether your kit is complete...
Looks good

Checking prerequisites...
Looks good

Creating new 'Build' script for 'Test-Count' version '0.0500'
shlomi:~/TEMP/Test-Count-0.0500$ ./Build
Copying lib/Test/Count/Filter/ByFileType/App.pm -> blib/lib/Test/Count/Filter/ByFileType/App.pm
Copying lib/Test/Count/Base.pm -> blib/lib/Test/Count/Base.pm
Copying lib/Test/Count/Filter.pm -> blib/lib/Test/Count/Filter.pm
Copying lib/Test/Count/Parser.pm -> blib/lib/Test/Count/Parser.pm
Copying lib/Test/Count.pm -> blib/lib/Test/Count.pm
Manifying blib/lib/Test/Count/Parser.pm -> blib/libdoc/Test::Count::Parser.3pm
Manifying blib/lib/Test/Count/Base.pm -> blib/libdoc/Test::Count::Base.3pm
Manifying blib/lib/Test/Count.pm -> blib/libdoc/Test::Count.3pm
Manifying blib/lib/Test/Count/Filter/ByFileType/App.pm -> blib/libdoc/Test::Count::Filter::ByFileType::App.3pm
Manifying blib/lib/Test/Count/Filter.pm -> blib/libdoc/Test::Count::Filter.3pm
shlomi:~/TEMP/Test-Count-0.0500$ ./Build test
t/00-load.t ....... 1/3 # Testing Test::Count 0.0500, Perl 5.010000, /usr/bin/perl5.10.0
t/00-load.t ....... ok
t/01-parser.t ..... ok
t/02-main.t ....... ok
t/03-filter.t ..... ok
t/boilerplate.t ... ok
t/pod-coverage.t .. ok
t/pod.t ........... ok
All tests successful.
Files=7, Tests=30,  4 wallclock secs ( 0.12 usr  0.03 sys +  2.59 cusr  0.19 csys =  2.93 CPU)
Result: PASS
shlomi:~/TEMP/Test-Count-0.0500$
```

2.3. Types of Tests: Unit Tests, Integration Tests, System Tests ¶

Software design methodologists distinguish between several types of automated tests. First of all, **unit tests** (also see the Wikipedia article) test only a single "unit" of the code (say a module or a class), to see if it behaves as expected. They generally make sure that the behaviour of the module is sane and desirable, while not trying to see if it works as part of the larger scheme.

On the other hand, **system tests** test the entire system. For example, if we're writing code to generate a web-site, we could test that the various pages of the resultant site contain some of the qualities that we expect. System tests tests the system as a whole, to see if there's a bug somewhere.

Between unit tests and system tests there could be several intermediate layers of tests, normally called **integration tests** .

You can write all these tests using TAP, Test::More and other testing modules on the CPAN, but it's important to be aware of the distinction.

Smoke Tests ¶

"Smoke tests" is a term referring to a subset of the tests used to see if the software application performs its very basic operation well enough to give way for further testing. It is akin to plugging in an Electronics device and making sure it doesn't raise smoke from mis-operation. As a result, if the entire tests suite is time consuming, the smoke testing should take a short time to perform.

Using Perl for Testing Code in Other Programming Languages ¶

You can use Perl to test software written in many other programming languages:

- If you want to perform system tests of foreign applications, you can look at the various way for Perl to invoke other command-line programs, and for its sockets and networking capabilities.

  For GUI (= Graphical User-Interface) tests, you can look at Win32-GuiTest and X11-GUITest.

- If you want to write unit-tests for these applications in Perl, you should look at the "Inline" family of modules that allow you to write native subroutines in Perl.

  Also of interest is the Ctypes for Perl project (which is currently under development.).

2.4. Mocking ¶

When testing certain parts of the application, it is sometimes desirable to mimic the functionality of different parts, so the testing will be isolated. For example, if we're testing a server-side script (such as a CGI script), we may wish to provide a server-emulating object that's completely under our control and that inputs the script with our own parameters. This is called **mocking** (see the Wikipedia article about Mock objects), and there are several mechanisms for doing so for Perl facilities:

- Test-MockObject
- Test-MockModule
- DBD-Mock - mock databases for testing.

With regard to mocking modules, one may opt to simulate loading a module using the Perl %INC variable (see perlvar) by doing something like:

```
use strict;
use warnings;

package CGI;

# .
# .
# .
```

```
BEGIN
{
    $INC{'CGI.pm'} = "/usr/lib/perl5/site_perl/5.10.0/CGI.pm";
}

1;
```

After doing this, the tested code can do `use CGI;` and still think it loaded the original module, while actually it is using our own mocked version.

3. Version Control ¶

Version control systems are also known as “revision control systems”, and “source control systems”. Version control is considered part of “software configuration management” (SCM) and there are also some more comprehensive SCM systems. Version control programs allow one to maintain various historical versions of one's data, retrieve earlier versions, and do other operations like branching or tagging.

This section will give the motivation for why you should start using version control for your software development, and will give a short demonstration using the Mercurial version control system. Feel free to skip this section if you're already drinking the version control kool-aid.

3.1. Motivation for Version Control ¶

Using version control gives several important advantages over the alternative of not using any version control system at all:

- You **won't lose** your code by accident. Having a version control system, preferably with a remote service, will mean you're going to have another place where your code is stored. If several developers are working on the code simultaneously, then each one of them will have a copy of the entire code (or, in some cases, even the entire history).
- It allows you to **keep historical versions** of the code, for easy reverting, comparison and investigation.

  Let's say you introduced a bug. With a version control system you can easily revert to a previous version of the code where the bug was not present to verify that it did not exist there. Then you can diff the results, or even bisect the history to find the exact check-in that introduced this bug.
- It allows one to maintain several simultaneous lines of code (normally called **"branches"**) and to easily compare between them and merge them.

Finally, you'll find using a modern and high-quality version control system a more convenient and more robust solution than using archives (such as .zip files) and patches. There are plenty of open-source and gratis version control systems, some of which are highly mature and esteemed and you shouldn't have a problem finding something that suits you.

3.2. Demo of Mercurial ¶

This section will demonstrate basic version control usage using the Mercurial version control system.

Please note: by choosing Mercurial I do not mean to imply that it is the best VCS out there or that you should necessarily use it. By all means, it is likely that there are other VCSes which are better in many respects. However, I'm familiar with Mercurial, and I think it is suitable for the demonstration here.

If you're interested in choosing a version control system, you can refer to these resources:

- The Better SCM Site
- The Free Version Control Systems appendix of “Producing Open Source Software” by Karl Fogel.
- The Wikipedia list of version control systems

The Demo ¶

First of all, install Mercurial using your operating system's package manager, or by downloading an installer from the Mercurial site.

Then create a new empty directory and run `hg init .`:

```
$p4n/5/merc-test$ hg init .
```

Now let's add some files. Start your favourite text editor and put these contents in the file `MyModule.pm`:

```
# This is MyModule.pm
package MyModule;

use strict;
use warnings;

sub add
{
    my ($x, $y) = @_;

    return $x+$y*2;
}

1;
```

Now let's put it under version control:

```
$p4n/5/merc-test$ mkdir MyModule
$p4n/5/merc-test$ cd MyModule/
$p4n/5/merc-test/MyModule$ gvim MyModule.pm # Edit it.
$p4n/5/merc-test/MyModule$ ls
MyModule.pm
$p4n/5/merc-test/MyModule$ hg status
? MyModule/MyModule.pm
```

As we can see from the output of `hg status`, the file is not tracked. Let's add it:

```
$p4n/5/merc-test/MyModule$ hg add MyModule.pm
$p4n/5/merc-test/MyModule$ hg status
A MyModule/MyModule.pm
$p4n/5/merc-test/MyModule$
```

Now the file is scheduled to be committed (note the `A`). Let's commit it:

```
$p4n/5/merc-test/MyModule$ hg commit -m "Added MyModule.pm"
$p4n/5/merc-test/MyModule$ hg status
$p4n/5/merc-test/MyModule$
```

We can see it in the output of the version control command hg log, which, as it name implies, gives a log of what has been done in the past:

```
$p4n/5/merc-test/MyModule$ hg log
changeset:   0:7dec17ed3e88
tag:         tip
user:        Shlomi Fish <shlomif@ELIDED>
date:        Fri Jan 14 18:07:32 2011 +0200
summary:     Added MyModule.pm
```

Now let's add a test:

```
$p4n/5/merc-test/MyModule$ gvim mytest.t # Test
$p4n/5/merc-test/MyModule$ cat mytest.t

use strict;
use warnings;

use Test::More tests => 1;

use MyModule;

is (MyModule::add(0, 0), 0, "0+0 is 0.");
shlomif[homepage]:$p4n/5/merc-test/MyModule$ prove mytest.t
mytest.t .. ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs ( 0.03 usr  0.01 sys +  0.02 cusr  0.00 csys =  0.06 CPU)
Result: PASS
$p4n/5/merc-test/MyModule$ hg status
? MyModule/mytest.t
$p4n/5/merc-test/MyModule$ hg add mytest.t
$p4n/5/merc-test/MyModule$
```

And let's commit it as well by using hg commit.

```
$p4n/5/merc-test/MyModule$ hg commit -m "Added the test."
```

Now let's add another test assertion:

```
shlomif[homepage]:$p4n/5/merc-test/MyModule$ cat mytest.t
\#!/usr/bin/perl

use strict;
use warnings;

use Test::More tests => 2;

use MyModule;

\# TEST
is (MyModule::add(0, 0), 0, "0+0 is 0.");

\# TEST
is (MyModule::add(2, 0), 2, "2+0 is 2.");
```

```
$p4n/5/merc-test/MyModule$ prove mytest.t
mytest.t .. ok
All tests successful.
Files=1, Tests=2,  0 wallclock secs ( 0.03 usr  0.00 sys +  0.02 cusr  0.00 csys =  0.05 CPU)
Result: PASS
```

However, before we commit let's see which changes have been made:

```
shlomif[homepage]:$p4n/5/merc-test/MyModule$ hg diff
diff -r e2b34f948dcd MyModule/mytest.t
--- a/MyModule/mytest.t Fri Jan 14 18:14:05 2011 +0200
+++ b/MyModule/mytest.t Fri Jan 14 18:18:57 2011 +0200
@@ -3,9 +3,13 @@
 use strict;
 use warnings;

-use Test::More tests => 1;
+use Test::More tests => 2;

 use MyModule;

 \# TEST
 is (MyModule::add(0, 0), 0, "0+0 is 0.");
+
+# TEST
+is (MyModule::add(2, 0), 2, "2+0 is 2.");
+
```

This displays the differences from the working copy to the pristine version in the repository.

```
$p4n/5/merc-test/MyModule$ hg status
M MyModule/mytest.t
$p4n/5/merc-test/MyModule$ hg commit -m "Add another assertion"
$p4n/5/merc-test/MyModule$ hg status
```

And it's committed.

We can now continue doing commits, adding more tests and fixing bugs as we go. For example, let's add another test:

```
$ gvim mytest.t # Edit
$ hg diff
@@ -3,7 +3,7 @@
 use strict;
 use warnings;

-use Test::More tests => 2;
+use Test::More tests => 3;

 use MyModule;

@@ -13,3 +13,6 @@
 \# TEST
 is (MyModule::add(2, 0), 2, "2+0 is 2.");

+# TEST
+is (MyModule::add(1, 1), 2, "1+1 is 2.");
```

```
+
$ prove mytest.t
mytest.t .. 1/3
mytest.t .. Dubious, test returned 1 (wstat 256, 0x100)
Failed 1/3 subtests

Test Summary Report
-------------------
mytest.t (Wstat: 256 Tests: 3 Failed: 1)
  Failed test:  3
  Non-zero exit status: 1
Files=1, Tests=3,  0 wallclock secs ( 0.03 usr  0.01 sys +  0.03 cusr  0.00 csys =  0.07 CPU)
Result: FAIL
```

Oops! The test has failed, now we need to fix a bug. With every commit, it is important that all tests will pass (unless perhaps we are working on a branch.). Let's correct it:

```
$ gvim MyModule.pm
$ hg diff MyModule.pm
diff -r ebc249691c24 MyModule/MyModule.pm
--- a/MyModule/MyModule.pm      Fri Jan 14 18:20:29 2011 +0200
+++ b/MyModule/MyModule.pm      Sat Jan 15 10:43:16 2011 +0200
@@ -8,7 +8,7 @@
 {
     my ($x, $y) = @_;

-    return $x+$y*2;
+    return $x+$y;
 }

 1;
```

Corrected, and now the test passes. Let's see which files changed:

```
$ hg status .
M MyModule.pm
M mytest.t
```

Two files are changed in the working copy. We can now put them in the repository using `hg commit`:

```
$ hg commit -m "Fixed a bug - we did x+y*2 instead of x+y"
```

Now let's suppose we broke something and the change is too big to fix, and we wish to revert to the pristine version. Our version control system allows us to do that:

```
$ hg diff
diff -r a7599e97a8d8 MyModule/MyModule.pm
--- a/MyModule/MyModule.pm      Sat Jan 15 10:46:24 2011 +0200
+++ b/MyModule/MyModule.pm      Sat Jan 15 10:48:04 2011 +0200
@@ -8,7 +8,7 @@
 {
     my ($x, $y) = @_;

-    return $x+$y;
```

```
+    return $x*100+$y;
 }

 1;
$ prove mytest.t
mytest.t .. 1/3
\#   Failed test '2+0 is 2.'
\#   at mytest.t line 14.
\#          got: '200'
\#     expected: '2'

\#   Failed test '1+1 is 2.'
\#   at mytest.t line 17.
\#          got: '101'
\#     expected: '2'
\# Looks like you failed 2 tests of 3.
mytest.t .. Dubious, test returned 2 (wstat 512, 0x200)
Failed 2/3 subtests

Test Summary Report
-------------------
mytest.t (Wstat: 512 Tests: 3 Failed: 2)
  Failed tests:  2-3
  Non-zero exit status: 2
Files=1, Tests=3,  0 wallclock secs ( 0.03 usr  0.01 sys +  0.02 cusr  0.00 csys =  0.06 CPU)
Result: FAIL
$ hg status .
M MyModule.pm
$ hg revert My
MyModule.pm   MyModule.pm~
$ hg revert MyModule.pm
$ hg status .
? MyModule.pm.orig
$ prove mytest.t
mytest.t .. ok
All tests successful.
Files=1, Tests=3,  0 wallclock secs ( 0.04 usr  0.00 sys +  0.02 cusr  0.00 csys =  0.06 CPU)
Result: PASS
$
```

Now that it's working we can perform more changes, and continue to commit them. We can see the log of all our changes:

```
$ hg update
0 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ hg log
changeset:   3:a7599e97a8d8
tag:         tip
user:        Shlomi Fish <shlomif@ELIDED>
date:        Sat Jan 15 10:46:24 2011 +0200
summary:     Fixed a bug - we did x+y*2 instead of x+y

changeset:   2:ebc249691c24
user:        Shlomi Fish <shlomif@ELIDED>
date:        Fri Jan 14 18:20:29 2011 +0200
summary:     Add another assertion

changeset:   1:e2b34f948dcd
user:        Shlomi Fish <shlomif@ELIDED>
date:        Fri Jan 14 18:14:05 2011 +0200
summary:     Added mytest.t.
```

```
changeset:   0:7dec17ed3e88
user:        Shlomi Fish <shlomif@ELIDED>
date:        Fri Jan 14 18:07:32 2011 +0200
summary:     Added MyModule.pm
```

4. Class Accessors ¶

Object accessors are a way to abstract access to an object's member variables (also known as “properties”, “attributes”, “fields”, “slots”, etc.) behind method calls. For example we can use `$person->age()` to get the age of `$person` and `$person->age(21)` or `$person->set_age(21)` to set their age to 21.

Accessors provide several important advantages over accessing the properties of objects directly and this section will serve as an introduction to them.

4.1. Example ¶

Here's an example class with some accessors and a script that uses it:

```
package Person;

use strict;
use warnings;

sub new
{
    my $class = shift;

    my $self = {};
    bless $self, $class;

    $self->_init(@_);

    return $self;
}

sub _init
{
    my $self = shift;
    my $args = shift;

    $self->_first_name($args->{'first_name'});
    $self->_last_name($args->{'last_name'});

    $self->_age($args->{'age'});

    return;
}

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

```
    if (@_)
    {
        my $new_first_name = shift;
        $self->{'_first_name'} = $new_first_name;
    }

    return $self->{'_first_name'};
}

sub _last_name
{
    my $self = shift;

    if (@_)
    {
        my $new_last_name = shift;
        $self->{'_last_name'} = $new_last_name;
    }

    return $self->{'_last_name'};
}

sub _age
{
    my $self = shift;

    if (@_)
    {
        my $new_age = shift;
        $self->{'_age'} = $new_age;
    }

    return $self->{'_age'};
}

sub greet
{
    my $self = shift;

    print "Hello ", $self->_first_name(), " ", $self->_last_name(), "\n";

    return;
}

sub increment_age
{
    my $self = shift;

    $self->_age($self->_age()+1);

    return;
}

sub get_age
{
    my $self = shift;

    return $self->_age();
}

1;
```

```
#!/usr/bin/env perl

use strict;
use warnings;

use Person;

my $shlomif =
    Person->new(
        {
            first_name => "Shlomi",
            last_name => "Fish",
            age => 32,
        }
    );

$shlomif->greet();
$shlomif->increment_age();

print "Happy Birthday, Shlomi, your age is now ", $shlomif->get_age(), ".\n";

my $newton =
    Person->new(
        {
            first_name => "Isaac",
            last_name => "Newton",
            age => 366,
        }
    );

$newton->greet();
print "Newton would have been ", $newton->get_age(),
    " years old today if he had been alive.\n"
    ;
```

4.2. Motivation ¶

So why should we use accessors instead of doing a direct `$person->{'age'}` access to the object's property? There are several reasons for that:

1. Writing the property names directly each time is prone to mis-spellings and errors, because they are strings. On the other hand, with method calls, the existence of a particular one is validated at run-time, and will throw an exception if a method was misspelled into a name that is not present.
2. If a property needs to be converted from a first-order property to a calculated value, then one can still use the existing method-based interface to access it, just by changing the implementation of the methods. On the other, this is much more difficult to change with a direct-field access.
3. The external interface provided by methods is cleaner and easier to maintain compatibility with, than a direct class access.
4. There may be other reasons, like better concurrency, persistence, etc.

4.3. Accessor modules on the CPAN ¶

As you may have noticed from our example, writing accessors by hand involves a lot of duplicate code, and can get tedious. One way to overcome it is by using namespace games (e.g: `*Person::${field} = sub { .... }`), but there are many

modules on CPAN that do it all for you. Here's an overview of some of the most prominent ones:

Class-Accessor ¶

Class-Accessor was one of the earliest accessor providing modules and is still pretty popular. It is pure Perl, has no dependencies, and works pretty well. It has many enhancements on CPAN that may work better for you.

Class-XSAccessor ¶

Class-XSAccessor is an accessor generator partially written using C and Perl/XS which is the Perl external subroutine mechanism. As such, it provides an unparalleled speed among the other accessor generators, and is even faster than writing your own accessor methods by hand, like we did in the example.

Moose ¶

While Moose provides accessors, they are only the tip of its iceberg. Moose is in fact a “post-modern” object system for Perl 5 that provides a type system, delegators, meta-classes, wrapping routines, and many other advanced features. As I once said:

> *Every sufficiently complex Class::Accessor program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Moose.*

If you're looking to take your object oriented programming in Perl 5 to new levels - look no further than that. One should be warned that as of this writing (August, 2009), Moose may have a relatively long startup time, although the situation has been improved and is expected to improve further.

5. Useful Features in Recent Perls ¶

This section will cover some new features in recent versions of Perl 5 that may prove useful for robust programming or may be encountered in the wild.

5.1. use base ¶

The use base pragma allows one to conveniently set the base packages of an object, while loading their corresponding modules at the same time. Using it is preferable to fiddling with `@ISA` directly.

Note that if you are using Moose, you should use the `extends()` function instead of `use base`.

The parent pragma forked from base.pm to "remove the cruft that accumulated there". It may be preferable.

5.2. Lexical Filehandles ¶

Traditionally Perl filehandles had been "typeglobs" - global names - normally starting with an uppercase letter that were not scope safe. While they could have been localised using "local", this was still a far cry from true lexical scoping. perl-5.6.x, however, introduced lexical filehandles for both file handles and directory handles.

Here is an example for a program implementing a directory listing.

```
#!/usr/bin/env perl
```

```
use strict;
use warnings;

sub get_entries
{
    my $dir_path = shift;

    opendir my $dir_handle, $dir_path
        or die "Cannot open '$dir_path' as a directory - $!.";

    my @entries = readdir($dir_handle);

    closedir($dir_handle);

    return [ sort { $a cmp $b } @entries ];
}

foreach my $arg (@ARGV)
{
    print "== Listing for $arg ==\n";
    foreach my $entry (@{get_entries($arg)})
    {
        print $entry, "\n";
    }
}
```

And here is an example that copies a file:

```


#!/usr/bin/env perl

# This is just for demonstration. A better way would be to use File::Copy :
#
# http://perldoc.perl.org/File/Copy.html
#

use strict;
use warnings;

my $source_fn = shift(@ARGV);
my $dest_fn = shift(@ARGV);

if ( (!defined($source_fn)) || (!defined($dest_fn)) )
{
    die "You must specify two arguments - source and destination."
}

open my $source_handle, "<", $source_fn
    or die "Could not open '$source_fn' - $!.";
open my $dest_handle, ">", $dest_fn
    or die "Could not open '$dest_fn' - $!.";

while (my $line = <$source_handle>)
{
    print {$dest_handle} $line;
}

close($source_handle);
close($dest_handle);
```

IO::Handle and Friends

Perl provides a set of lexical and object-oriented abstractions for file handles called IO::Handle. Starting from recent versions of Perl, one can use them with the built-in perlfunc mechanisms. You can find more information about them here:

- IO::Handle's Documentation
- IO::File's Documentation
- IO::Socket's Documentation

6. The local keyword ¶

Before Perl 5 came out and Perl got lexical scoping and the `my` keyword, an older `local` keyword was made available for programmers to temporarily "localise" the values of variables (or parts there of) in Perl.

As opposed to `my`, which is lexically scoped, `local` is **dynamically scoped**. What happens when one writes a `local $myvar = NEW_VALUE_EXPR();` (which will work only for package variables) is that perl will store the previous value of the variable somewhere safe, allow the programmer to tamper with it as it pleases, and restore its value to its previous, saved state, when the block exits. As opposed to my, the new localised value will survive function calls in different functions.

So when should local be used?

6.1. Use and Abuse ¶

The rule of the thumb is that for general scoping, `local` should not be used instead of `my`, which is safer and better. You may still encounter some code using local in the wild, but assuming you need to maintain it, this code should be revamped to use `my` instead.

For more information refer to:

- Mark Jason Dominus' "Coping With Scoping" - a general and comprehensive discussion.
- "Seven Useful Uses of local" - also by Mark Jason Dominus.
- A Linux-IL post explaining the difference between my and local - by Shlomi Fish.

7. Using POD for Documentation ¶

POD is short for "Plain Old Documentation", and is a lightweight markup language, which is the de-facto standard for writing documentation for Perl programs, Perl modules and Perl itself.

In the context of Perl modules, POD is primarily used to give API (Application Programmers' Interface) documentation. In the context of Perl programs, POD is primarily used to document the usage of the program and the command line flags it accepts. POD is also used to document the perl core (so-called perldocs).

7.1. POD Demonstration ¶

How to write POD ¶

POD sections start with a single POD directive on a new line and continue up to the next =cut directive also on a line of its own. Here are some POD directives:

Headers ¶

=head1, =head2, =head3, etc. - these are headers. The lower the header number is, the more significant it is and the bigger font will be used for it. Headers are followed by the text of the header. For example:

```
=head1 All you wanted to know about animals.

=head2 Introduction

This document aims to explain about animals.

=head2 Mammals.

=head3 Cats

Cats are awesome. They are useful for keeping the rats' population at
bay.

=head3 Dogs

Dogs have been called Man's best friend.
```

Regular Text ¶

As you can see, a regular paragraph text is a paragraph. Paragraphs are separated by blank lines, and newlines are ignored.

Code Blocks ¶

A **code block** (or verbatim paragraph) can be added by creating a portion of the text that's indented by using whitespace. In code blocks, newlines are not ignored. For example:

```
=head1 All you wanted to know about animals.

=head2 Introduction

This document aims to explain about animals.

=head2 Mammals.

=head3 Cats

Cats are awesome. They are useful for keeping the rats' population at
bay.

=head3 Dogs

Dogs have been called Man's best friend.

Here is an example program to name your dog:

    #!/usr/bin/perl

    use strict;
    use warnings;
```

```
    my @dog_names = (qw(Rex George Beethoven Max Rocky Lucky Cody));

    print "Name your dog " . $dog_names[rand(@dog_names)] . "!\n";

Put it in a file and run it.
```

Formatting Codes ¶

One can use some formatting codes:

- I<text> - for italic text.
- B<text> - for bold text.
- C<text> - for code (monospace) text.
- L<text> - hyperlinks - see **perldoc perlpod** for more information.
- E<lt> , E<gt> , E<htmlname>, etc. are escapes.

One should note that one can combine several styles at once using BI< ... > notation. Furthermore, one can enclose text with special characters (such as < and >) using several <<< and trailing >>> characters.

Lists ¶

One can use lists in POD by writing =over 4 (or some other value of indent-level instead of "4"), and then several =item's and finally =back. An item can be =item * for a bullet, =item 1. to produce numbered lists or =item title to produce a definition list.

For example:

```
=head1 All you wanted to know about animals.

=head2 Introduction

This document aims to explain about animals.

=head2 Mammals.

=head3 Cats

Cats are awesome. They are useful for keeping the rats' population at
bay.

=head3 Dogs

Dogs have been called Man's best friend.

Here is an example program to name your dog:

    #!/usr/bin/perl

    use strict;
    use warnings;

    my @dog_names = (qw(Rex George Beethoven Max Rocky Lucky Cody));

    print "Name your dog " . $dog_names[rand(@dog_names)] . "!\n";

Put it in a file and run it. This program will generate one of the following
names:

=over 4

=item * Rex
```

```
Rex like the dinosaur.

=item * George

Like George Washington.

=item * Beethoven

Last name of the famous composer.

=item * Max

Short for Maximilian.

=item * Rocky

Like the film.

=item * Lucky

A lucky dog.

=item * Cody

For good coding.

=back
```

For More Information ¶

POD has some other directives. For more information refer to perldoc perlpod, and to the Wikipedia page about POD.

7.2. Testing and Verifying POD ¶

The CPAN module Test-Pod allows one to check for POD errors in files. Its use is recommended to avoid writing POD with errors.

The CPAN module Test-Pod-Coverage tries to make sure that all the public API functions in one's Perl modules have corresponding POD documentation. Its use is also recommended.

Generally, Module-Starter and similar modules will generate such tests for you automatically as part of the skeleton of your CPAN distribution.

7.3. Literate Programming ¶

Literate Programming is a method of writing code that allows one to intermingle code with documentation, re-order the sections of the code in relevance to their intention, and create an entire document typeset that is explaining the code, with full cross-references and interlinks. As Mark Jason Dominus explains POD is not Literate Programming.

Traditionally, Literate Programming systems have generated TeX/LaTeX output, but more recently there have been ones that could output DocBook/XML.

I am personally not writing my code in a Literate Programming style, because I feel that:

1. It will require much more effort to create code that will only be marginally easier to understand.
2. The documentation will need to be maintained along with the code and may become out-of-date. Even inline comments suffer from this symptom, and external documentation much more so.

3. The code should be structured to be as self-documenting as possible. For example, instead of documenting what a block of code is doing, one should extract a subroutine with a name that conveys the intention.

However, I'm mentioning Literate Programming here for completeness sake, should you choose to follow this route.

7.4. POD Extensions ¶

POD has some extended dialects with more features and options:

- PseudoPod -

> *PseudoPod is an extended set of Pod tags used for book manuscripts. Standard Pod doesn't have all the markup options you need to mark up files for publishing production. PseudoPod adds a few extra tags for footnotes, tables, sidebars, etc.*

- MJD's POD variant called MOD - this was used to write the book "Higher Order Perl", and one can download the markup for the book and for source code for the MOD package from the book's site
- Perldoc.pm was an attempt to modernise POD by Ingy döt Net, and incorporating some markup from his Kwiki wiki engine.

Aside from that, Wikipedia has a list of other Lightweight markup languages, and some of them also have implementations in Perl.

8. Module-Build and Module-Starter ¶

Now let's tie everything together. When you download a Perl package from CPAN, there's a standard way to build and install it - `perl Makefile.PL`, `make`, `make test` and `make install` (or alternatively a similar process with `perl Build.PL` and `./Build`).

When creating packages of Perl code, it is preferable to make them capable of being built this way, even if they are intended for internal use. That is because packaging them this way gives you many advantages, among them the ability to specify CPAN (and in-house) dependencies, integrity tests, configurability in building and installation, and simplification of the preparation of system packages (such as `.rpm`s or `.deb`s).

In this section we'll learn how to prepare your own CPAN-like package of Perl 5 code using module-starter and Module-Build. There are some variations on this theme, but it should get you started.

8.1. The Module-Starter Invocation Command ¶

After you install Module-Starter, you can type `module-starter --help` to get the help for it, and get a result such as the following:

```
Usage:
    module-starter [options]

    Options:

        --module=module  Module name (required, repeatable)
        --distro=name    Distribution name (optional)
```

```
        --dir=dirname    Directory name to create new module in (optional)

        --builder=module Build with 'ExtUtils::MakeMaker' or 'Module::Build'
        --eumm           Same as --builder=ExtUtils::MakeMaker
        --mb             Same as --builder=Module::Build
        --mi             Same as --builder=Module::Install

        --author=name    Author's name (required)
        --email=email    Author's email (required)
        --license=type   License under which the module will be distributed
                         (default is the same license as perl)

        --verbose        Print progress messages while working
        --force          Delete pre-existing files if needed

        --help           Show this message

    Available Licenses: perl, bsd, gpl, lgpl, mit

    Example:

        module-starter --module=Foo::Bar,Foo::Bat \
            --author="Andy Lester" --email=andy@petdance.com
```

Let's show an example invocation for our own module called `MyMath::Ops` which will contain some silly mathematical routines:

```
module-starter --distro=MyMath::Ops \
    --dir=MyMath-Ops \
    --module=MyMath::Ops \
    --module=MyMath::Ops::Add \
    --module=MyMath::Ops::Multiply \
    --module=MyMath::Ops::Subtract \
    --module=MyMath::Ops::Divide \
    --mb \
    --author="Perl Newbie" \
    --email='perl-newbie@perl-begin.org' \
    --verbose
```

8.2. Module-Build commands ¶

The first thing we should do is change the directory to the directory that Module-Starter created and run `perl Build.PL`. We get some output like the following:

```
shlomi[homepage]:$p4n/5/src/module-build-and-starter$ cd MyMath-Ops/
shlomi[homepage]:$p4n/5/src/module-build-and-starter/MyMath-Ops$ perl Build.PL
Checking whether your kit is complete...
Looks good

Checking prerequisites...
Looks good

Deleting Build
Removed previous script 'Build'
```

```
Creating new 'Build' script for 'MyMath-Ops' version '0.01'
shlomi[homepage]:$p4n/5/src/module-build-and-starter/MyMath-Ops$
```

What the `perl Build.PL` command does is generate the `Build` script in the current directory that can be used to perform such operations as building, testing, packaging, and installing of the distribution. Sometimes we need to re-run `perl Build.PL` if we modified the configuration.

Now let's run `./Build` and `./Build test`.

```
shlomi[homepage]:$p4n/5/src/module-build-and-starter/MyMath-Ops$ ./Build
Copying lib/MyMath/Ops/Subtract.pm -> blib/lib/MyMath/Ops/Subtract.pm
Copying lib/MyMath/Ops/Divide.pm -> blib/lib/MyMath/Ops/Divide.pm
Copying lib/MyMath/Ops/Multiply.pm -> blib/lib/MyMath/Ops/Multiply.pm
Copying lib/MyMath/Ops.pm -> blib/lib/MyMath/Ops.pm
Copying lib/MyMath/Ops/Add.pm -> blib/lib/MyMath/Ops/Add.pm
Manifying blib/lib/MyMath/Ops/Add.pm -> blib/libdoc/MyMath::Ops::Add.3pm
Manifying blib/lib/MyMath/Ops/Multiply.pm -> blib/libdoc/MyMath::Ops::Multiply.3pm
Manifying blib/lib/MyMath/Ops/Subtract.pm -> blib/libdoc/MyMath::Ops::Subtract.3pm
Manifying blib/lib/MyMath/Ops/Divide.pm -> blib/libdoc/MyMath::Ops::Divide.3pm
Manifying blib/lib/MyMath/Ops.pm -> blib/libdoc/MyMath::Ops.3pm
shlomi[homepage]:$p4n/5/src/module-build-and-starter/MyMath-Ops$ ./Build test
t/00-load.t ....... 1/5 # Testing MyMath::Ops 0.01, Perl 5.010001, /usr/bin/perl5.10.1
t/00-load.t ....... ok
t/boilerplate.t ... ok
t/pod-coverage.t .. ok
t/pod.t ........... ok
All tests successful.
Files=4, Tests=22,  1 wallclock secs ( 0.10 usr  0.04 sys +  0.60 cusr  0.12 csys =  0.86 CPU)
Result: PASS
shlomi[homepage]:$p4n/5/src/module-build-and-starter/MyMath-Ops$
```

What happens is that `./Build` copies the files under `blib/` , builds the documentation, and in case we had XS (= "External Subroutine" - perl routines written in a low-level language) it would also build the extensions. This allows us to run tests against the built code, either automated or manual by using the blib module.

After we had ran `./Build`, we ran `./Build test` to run the automated tests that Module-Starter generated for us. As you can see the line says that all tests successful. If they were not, we should fix either the code or the tests, depending on what is wrong.

Now let's move on.

8.3. Adding meaningful code ¶

If we look at the code of the `lib/…*.pm` file, we'll see that there's practically nothing there. So now it's time that we add some meaningful code to the modules. But first we need to add some tests. Let's add this test script under `t/add.t`

```
#!/usr/bin/env perl

use strict;
use warnings;

use Test::More tests => 2;

use MyMath::Ops::Add;

{
    my $adder = MyMath::Ops::Add->new();
```

```
    # TEST
    ok ($adder, "Adder was initialised");

    # TEST
    is ($adder->add(2,3), 5, "2+3 == 5");
}
```

Now we need to add it to the MANIFEST, so it will be included in future versions of Perl. After we did it, let's run ./Build test to see the tests fail:

```
$ perl Build.PL
Creating new 'MYMETA.yml' with configuration results
Creating new 'Build' script for 'MyMath-Ops' version '0.01'
$ ./Build test
t/00-load.t ....... 1/5 # Testing MyMath::Ops 0.01, Perl 5.012003, /usr/bin/perl5.12.3
t/00-load.t ....... ok
t/add.t ........... Can't locate object method "new" via package "MyMath::Ops::Add" at t/add.t line 11.
\# Looks like your test exited with 255 before it could output anything.
t/add.t ........... Dubious, test returned 255 (wstat 65280, 0xff00)
Failed 2/2 subtests
t/boilerplate.t ... ok
t/pod-coverage.t .. ok
t/pod.t ........... ok

Test Summary Report
-------------------
t/add.t         (Wstat: 65280 Tests: 0 Failed: 0)
  Non-zero exit status: 255
  Parse errors: Bad plan.  You planned 2 tests but ran 0.
Files=5, Tests=22,  1 wallclock secs ( 0.14 usr  0.04 sys +  0.56 cusr  0.11 csys =  0.85 CPU)
Result: FAIL
Failed 1/5 test programs. 0/22 subtests failed.
```

So now we need to fix the tests. Open lib/MyMath/Ops/Add.pm and write that:

```
package MyMath::Ops::Add;

use warnings;
use strict;

=head1 NAME

MyMath::Ops::Add - The great new MyMath::Ops::Add!

=head1 VERSION

Version 0.01

=cut

our $VERSION = '0.01';

=head1 SYNOPSIS

Quick summary of what the module does.

Perhaps a little code snippet.
```

```
    use MyMath::Ops::Add;

    my $foo = MyMath::Ops::Add->new();
    ...

=head1 EXPORT

A list of functions that can be exported.  You can delete this section
if you don't export anything, such as for a purely object-oriented module.

=head1 FUNCTIONS

=head2 new

Construct a new object.

=cut

sub new
{
    my $class = shift;

    my $self = bless {}, $class;

    $self->_init(@_);

    return $self;
}

sub _init
{
    my $self = shift;

    return;
}

=head2 $self->add($x, $y)

Adds two numbers.

=cut

sub add
{
    my $self = shift;

    my ($x, $y) = @_;
    return $x+$y;
}

=head2 function1

=cut

sub function1 {
}

=head2 function2

=cut
```

```
sub function2 {
}

=head1 AUTHOR

Perl Newbie, C<< <perl-newbie at perl-begin.org> >>

=head1 BUGS

Please report any bugs or feature requests to C<bug-mymath::ops at rt.cpan.org>, or through
the web interface at L<http://rt.cpan.org/NoAuth/ReportBug.html?Queue=MyMath::Ops>.  I will be notified, and then you'll
automatically be notified of progress on your bug as I make changes.




=head1 SUPPORT

You can find documentation for this module with the perldoc command.

    perldoc MyMath::Ops::Add


You can also look for information at:

=over 4

=item * RT: CPAN's request tracker

L<http://rt.cpan.org/NoAuth/Bugs.html?Dist=MyMath::Ops>

=item * CPAN Ratings

L<http://cpanratings.perl.org/d/MyMath::Ops>

=item * Search CPAN

L<http://metacpan.org/release/MyMath::Ops/>

=back


=head1 ACKNOWLEDGEMENTS


=head1 COPYRIGHT & LICENSE

Copyright 2009 Perl Newbie.

This program is free software; you can redistribute it and/or modify it
under the terms of either: the GNU General Public License as published
by the Free Software Foundation; or the Artistic License.

See http://dev.perl.org/licenses/ for more information.


=cut

1; # End of MyMath::Ops::Add
```

And now let's run "./Build test" again:

```
$ ./Build test
t/00-load.t ....... 1/5 # Testing MyMath::Ops 0.01, Perl 5.014002, /usr/bin/perl5.14.2
t/00-load.t ....... ok
t/add.t ........... ok
t/boilerplate.t ... ok
t/pod-coverage.t .. ok
t/pod.t ........... ok
All tests successful.
```

Since all tests are successful, we can commit the changes to the repository.

Moving on

Now we can continue to add more tests, and then fix the failing ones. If the code becomes too convoluted, due to modifications, we can **refactor it** and improve its modularity. Running the existing automated tests after such a change will better make sure that we didn't break something.

This "write more tests", "get tests to pass", "refactor" is the cycle of development and maintenance, and Perl tools such as Module-Build facilitate it.

8.4. Getting rid of the boilerplate ¶

The skeleton of the distribution generated by Module-Starter contains some boilerplate, which is pre-included text and code, used as placeholders. That should be replaced by more meaningful one by the programmer who is writing the distribution.

Luckily, it also generates a script on t/boilerplate.t that checks for that boilerplate and reports it. However, the tests there are marked as TODO tests, whose failure status is ignored by default. To turn off their TODO status, open t/boilerplate.t in your text editor and remove or comment-out the following line

```
  local $TODO = "Need to replace the boilerplate text";
```

After we do that, we get some test failures when running ./Build test:

```
$ ./Build test
t/00-load.t ....... 1/5 # Testing MyMath::Ops 0.01, Perl 5.014002, /usr/bin/perl5.14.2
t/00-load.t ....... ok
t/add.t ........... ok
t/boilerplate.t ... 1/7
\#   Failed test 'README contains boilerplate text'
\#   at t/boilerplate.t line 23.
\# The README is used... appears on lines 3
\# 'version information here' appears on lines 11

\#   Failed test 'Changes contains boilerplate text'
\#   at t/boilerplate.t line 23.
\# placeholder date/time appears on lines 3

\#   Failed test 'lib/MyMath/Ops.pm contains boilerplate text'
\#   at t/boilerplate.t line 23.
\# stub function definition appears on lines 37 41 44 48
\# boilerplate description appears on lines 21
\# the great new $MODULENAME appears on lines 8

\#   Failed test 'lib/MyMath/Ops/Add.pm contains boilerplate text'
\#   at t/boilerplate.t line 23.
```

```
\# stub function definition appears on lines 74 78 81 85
\# boilerplate description appears on lines 20
\# the great new $MODULENAME appears on lines 8

\#   Failed test 'lib/MyMath/Ops/Multiply.pm contains boilerplate text'
\#   at t/boilerplate.t line 23.
\# stub function definition appears on lines 37 41 44 48
\# boilerplate description appears on lines 21
\# the great new $MODULENAME appears on lines 8

\#   Failed test 'lib/MyMath/Ops/Subtract.pm contains boilerplate text'
\#   at t/boilerplate.t line 23.
\# stub function definition appears on lines 37 41 44 48
\# boilerplate description appears on lines 21
\# the great new $MODULENAME appears on lines 8

\#   Failed test 'lib/MyMath/Ops/Divide.pm contains boilerplate text'
\#   at t/boilerplate.t line 23.
\# stub function definition appears on lines 37 41 44 48
\# boilerplate description appears on lines 21
\# the great new $MODULENAME appears on lines 8
\# Looks like you failed 7 tests of 7.
t/boilerplate.t ... Dubious, test returned 7 (wstat 1792, 0x700)
Failed 7/7 subtests
t/pod-coverage.t .. ok
t/pod.t ........... ok

Test Summary Report
-------------------
t/boilerplate.t (Wstat: 1792 Tests: 7 Failed: 7)
  Failed tests:  1-7
  Non-zero exit status: 7
Files=5, Tests=24,  0 wallclock secs ( 0.03 usr  0.01 sys +  0.15 cusr  0.02 csys =  0.21 CPU)
Result: FAIL
Failed 1/5 test programs. 7/24 subtests failed.
```

Fixing them is left as an exercise for the reader.

8.5. Additional Resources ¶

Here are some additional resources regarding managing a CPAN-like distribution.

1. ExtUtils-MakeMaker is Perl's older and now largely unloved distribution manager, which relies on generating makefiles. It was described by chromatic as “a jumble of Perl which writes cross platform shell scripts to install Perl code, and you customize that by writing a superclass from which platform-specific modules inherit pseudo-methods which use regular expressions to search and replace cross-platform cross-shell code, with all of the cross-platform and cross-shell quoting issues that entails” .

2. Module-Install is a more modern and succinct wrapper around ExtUtils-MakeMaker that has gained some popularity. It ships its code (and the code of its extensions) under an `./inc` directory in the distribution, which has known to cause some bootstrapping issues for co-developers who would like to collaborate on the code from its version control repository. Nevertheless, it may be worth taking a look.

3. *Writing Perl Modules for CPAN* is a book by Sam Tregar, which has a free PDF download. It is somewhat out-of-date (only covering ExtUtils-MakeMaker), but may still be enlightening.

4. Dist::Zilla is a high-level distribution generator, with many available plugins, that abstracts away a lot of the duplication within a module and across modules. It generates fully-functional distributions that can be shipped to CPAN and used normally. As with Module-Install, it may pose a problem to your contributors, especially if they have out-of-date versions of its CPAN modules installed, but it is a useful tool.

5. A Perlmonks.org post titled "RFC: How to Release Modules on CPAN in 2011" goes to more coverage about the issues covered in this section.

6. Jeffrey Thalhammer has prepared a talk titled "CPAN for Private Code" which gives the motivation for packaging Perl code in CPAN-like distributions, even if it is not intended for CPAN.

9. Conclusion ¶

The aim of this presentation was to make your Perl code (and that of other programming languages) less error-prone, easier to understand, and easier to modify. I did not provide a complete coverage of code external quality (which is what the user feels or notices) or internal quality (which is what is also affecting the developers maintaining the code). For a more thorough coverage of those, you are referred to:

1. My essay "What Makes Software High-Quality?".
2. The Wikipedia "Software quality" article, which gives a coverage of the topic with many references.
3. The "Perl Elements to Avoid" Page on the Perl Beginners Site

Happy programming!

9.1. Links ¶

To be done if we see fit (suggestions are welcome).

9.2. Thanks ¶

Here I'd like to thank some people who contributed to this presentation:

- Gabor Szabo - for his constant efforts for promoting Perl 5, Perl 6, and other open-source technologies in Israel and abroad.
- The people who help those who ask questions on Freenode's #perl channel.
- The people who help those who ask questions on beginners@perl.org.
- Damian Conway for writing the excellent book *Perl Best Practices*, which inspired a lot of similar discussion about best practices in Perl.
- chromatic for his "Modern Perl" blog and book (freely available online with sources), which further fuelled interest in a more modern approach to Perl programming.
- Alan Haggai Alavi for providing some help with the Perl Beginners site.

Webmaster: Shlomi Fish (Email - shlomif@shlomifish.org)

Original Design: GoFlexiblePro | Author: G. Wolfgang | W3C XHTML5 | W3C CSS 3

Hosted by: Hexten.net.