Eine Shell für Perl
Devel::REPL

Oft möchte man eine Kleinigkeit in Perl ausprobieren, einfach mal testen, ob der Befehl überhaupt das gewünschte Ergebnis bringt oder nicht. Und dann jedesmal einen Einzeiler mit "perl -wle '...'" daraus machen ist zu viel Tipparbeit. Hier wäre eine Perl-Shell ganz nützlich. Ich möchte hier mal ein paar Möglichkeiten aufzeigen, wobei ich den Schwerpunkt auf `Devel::REPL` legen werde.

Das, den Modulen/Skripten zugrunde liegende, Prinzip nennt sich "REPL" und bedeutet "read-eval-print-loop". Diese lange Bezeichnung sagt schon ganz gut aus, was das Prinzip macht: Lese die Benutzereingaben, evaluiere sie, gebe das Ergebnis aus und mache das ganze immer wieder (in einer Schleife).

Für Perl gibt es - wie es für Perl typisch ist - mehrere Module/Skripte, die das leisten:

- PerlConsole
- ein selbst geschriebenes Skript wie unter http://sedition.com/perl/perl-shell.html
- der Perl-Debugger
- Devel::REPL

Vermutlich gibt es noch mehr Lösungen, aber ich möchte hier nur auf die letzten drei Sachen eingehen.

Ein selbstgeschriebenes Skript hat den Charme, dass man es selbst gemacht hat, ist also gut für das persönliche Empfinden. Ansonsten sollte man sich aber an andere Lösungen halten. Denn selbstgeschriebene Lösungen müssen selbst erweitert und gewartet werden.

Perl-Debugger

Der Perl-Debugger ist ganz nützlich für das Debugging von Programmen. In der Ausgabe 7 von $foo gab es auch schon eine Einführung von Thomas Fahle. Diese REPL kann man ganz einfach mit `perl -d 1` starten. Allerdings sollte man dabei auch beachten, dass der Debugger sich in einigen Punkten anders verhält als der normale Perl-Interpreter. Das sind zum Teil Bugs, zum Teil Features. Der geneigte Leser kann sich die Bug-Liste unter http://rt.perl.org anschauen. Für einfache Sachen ist der Debugger aber durchaus zu gebrauchen - neben den typischen Debugging-Aufgaben. So kann man einfache Befehle einfach mal testen:

```
perl -d -e 0

  DB<1> print "Hallo Foo-Magazin"
Hallo Foo-Magazin

  DB<2> print 6 * 7; print "\n"; print 6 x 7
42
6666666
```

Das Zeilenende wird als Anweisungsende aufgefasst, das übliche Semikolon am Zeilenende kann daher entfallen. Mehrere Anweisungen in einer Zeile müssen allerdings durch ein Semikolon getrennt werden.

Näher will ich hier gar nicht auf den Debugger eingehen.

Devel::REPL

Ich persönlich bevorzuge `Devel::REPL`, weil es die weitestgehenden Möglichkeiten bietet. So ist es auch möglich, eigene Plugins zu schreiben, um das Verhalten des Moduls zu verändern. Unter ActivePerl war die Installation bei meinem letzten Versuch nicht wirklich erfolgreich, weil ein paar Mo-

dule nicht in den PPM-Repositories zu finden waren. Das Modul verwendet `Moose`, so dass die Liste an Abhängigkeiten nicht gerade kurz ist.

Nach der Installation macht das Modul aber Spaß. Der einfachste Start ist der Aufruf von `re.pl`, das bei der Installation mitgeliefert wird.

```
C:\>perl re.pl

$ my $a = 3;
3
$ $a + 2;
5
$
```

Das Semikolon kann hier auch ausgelassen werden, da `Devel::REPL` die Befehle in einem Block ausführt. Also ist auch das hier gültig:

```
$ $a + 3
6
```

Das Modul kann aber noch mehr. `Devel::REPL` verfügt über ein Plugin-System, mit dem man das Verhalten der "Shell" noch erweitern kann. Ohne Plugins ist es nicht möglich, Befehle über mehrere Zeilen zu schreiben:

```
$ sub test {
Compile error: Missing right curly or square
bracket at (eval 234) line 5, at end of line
syntax error at (eval 234) line 5, at EOF

$
```

Um mehrzeilige Befehle zu ermöglichen, gibt es das Plugin `Devel::REPL::Plugin::MultiLine::PPI`. In der Standard `re.pl` wird das Plugin automatisch geladen. Schreibt man sich eine eigene REPL mit `Devel::REPL`, so kann man das Plugin laden:

```
use Devel::REPL;
my $repl = Devel::REPL->new;
$repl->load_plugin( 'MultiLine::PPI' );
$repl->run;
```

Wenn jetzt das Skript gestartet wird, kann man auch mehrzeilige Befehle eingeben:

```
C:\>perl repl.pl

$ sub test {

>  print "perl-magazin.de"

> }

$ test()
1
$ perl-magazin.de
```

Es werden schon etliche Plugins mit der Distribution mitgeliefert. Zwei für mich sehr interessante Plugins sind `Devel::REPL::Plugin::History`, mit dem man auf Befehle zurückgreifen kann, die man innerhalb der Session (dem aktuellen Programmlauf) ausgeführt hat.

```
C:\>perl repl.pl

$ my $a = 'test';
test
$ !1
my $a = 'test';
test
```

Und `Devel::REPL::Plugin::Nopaste`, mit dem der Inhalt der aktuellen Sitzung in einem Nopaste-Bereich der Öffentlichkeit zugänglich gemacht wird. Trifft man auf Probleme oder Fragen während man einige Sachen austestet, kann man mit einem einfachen Befehl die eigenen Tests veröffentlichen und in Foren und/oder IRC einfach auf die Nopaste-Seite verweisen. Die Veröffentlichung kann man mit dem Befehl `#nopaste` anstoßen (siehe Listing 1).

Trotz der Fehlermeldung ist der Code jetzt auf pastie.org zu finden.

Profile und RC-Dateien

Bei `Devel::REPL` kann man auch mit Profilen arbeiten. In diesen Profilen kann gespeichert werden, welche Plugins geladen werden sollen. Die Profile werden dann als Perl-Modul abgespeichert. Ein Beispiel-Profil ist in Listing 2 zu sehen.

```
C:\>perl repl.pl

$ my $a = 3
3
$ my $a = 3
3
$ #nopaste
There is no form numbered 2 at C:/strawberry/perl/site/lib/App/Nopaste/Service/P
astie.pm line 13
http://pastie.org/836669
$
```

Listing 1

```
package Devel::REPL::Profile::FooMagazin;

use Moose;
use namespace::clean -except => [ 'meta' ];

with Devel::REPL::Profile;

sub plugins {
    return qw/MultiLine::PPI Done RegexExplain/;
}

sub apply_profile {
    my ($self,$repl) = @_;

    $repl->load_plugin( $_ ) for $self->plugins;
}

1;
```

Listing 2

Das Profil wird dann mittels

```
C:\>perl re.pl --profile FooMagazin
```

eingebunden.

Die RC-Dateien (*Run Control*) werden üblicherweise dafür benutzt, um noch mehr Vorbereitungsarbeit zu erledigen. So kann man darin Beispielsweise Datenbank-Verbindungen aufbauen oder schon bestimmte Module oder auch REPL-Plugins laden. Die RC-Dateien werden standardmäßig in `$HOME/.re.pl/repl.rc` gesucht. Aber man kann eine solche Datei auch beim Start der REPL angeben:

```
C:\>perl re.pl --rcfile projektname.rc
```

Wie so eine Datei aussehen kann, wird hier gezeigt.

```
use CGI;
use DBI;

sub dbi_connect {
 my $dbh = DBI->connect( 'DBI:SQLite:test' )
 or die $DBI::errstr; $dbh
}
```

Und warum gibt es RC-Dateien und Profile? Mit den RC-Dateien kann man sehr gut auf Projektebene arbeiten. So kann man für jedes Perl-Projekt eine eigene RC-Datei anlegen, in der dann die spezifischen Module geladen werden. Beim Start der REPL gibt man dann nur noch an, welche RC-Datei geladen werden soll und schon hat man eine funktionierende Umgebung für das Projekt zur Verfügung. In den Profilen werden dann die persönlichen Einstellungen für die REPL gespeichert. So kann man für unterschiedliche Entwickler die gleiche RC-Datei verwenden, jeder hat aber sein eigenes Profil.

Ein eigenes Plugin schreiben

Weiter oben bin ich schon auf Plugins eingegangen. In diesem Abschnitt möchte ich zeigen, dass es sehr einfach ist, eigene Plugins zu schreiben. Allerdings sollte man schonmal etwas von `Moose` gehört haben. `Devel::REPL` ist mit `Moose` implementiert und dort gibt es die Methoden-Modifikatoren `before`, `around` und `after`. Damit ist es möglich, mit einfachen Mitteln vor und/oder nach einer Methode noch weiteren Code auszuführen.

Ein kurzes Beispiel in Listing 3.

```
#!/usr/bin/perl

{
    package Foo;

    use Moose;
    use LWP::Simple;

    my $url = 'http://perl-magazin.de';

    sub url {
        print $url;
    }

    after 'url' => sub {
        print "\n",
        join "\n",
        grep{ defined }head( $url );
    };
}

Foo->url;

C:\>perl after.pl
http://perl-magazin.de
text/html; charset=ISO-8859-1
Apache/2.2.3 (Debian)
C:\>
```

Listing 3

Als Beispiel Plugins werde ich zuerst ein Plugin schreiben, das nach der Ausführung den gerade ausgeführten Befehl ausgibt.

```
#!/usr/bin/perl

package Devel::REPL::Plugin::Done;
use Devel::REPL::Plugin;

use Data::Dumper;

use namespace::clean -except => [ 'meta' ];

my $code;

before 'eval' => sub { $code = $_[1] };

after 'print' => sub {
    my $self = shift;
    my $fh   = $self->out_fh;
    print $fh "\n" . $code;
};

__PACKAGE__
```

Hier sieht man die Anwendung von `before` und `after`. Das `eval` und `print` sind keine Perl-Built-in-Funktionen, sondern Methoden aus `Devel::REPL`. In der Methode Da man bei `print` nicht den Befehl übergeben bekommt, muss der Befehl abgefangen werden, bevor er ausgeführt wird (deswegen das `before 'eval'`). Die Ausgabe des evals wird in der `print`-Methode ausgegeben. Da der Befehl nach der Ausgabe angezeigt werden soll, muss hier das `after 'print'` verwendet werden.

```
C:\>perl repl.pl

$ #explain ^d{4}$
The regular expression:

(?-imsx:^d{4}$)

matches as follows:

NODE         EXPLANATION
-----------------------------------------------
(?-imsx:     group, but do not capture
             (case-sensitive)(with ^ and $
             matching normally) (with . not
             matching \n) (matching white-
             space and # normally):
-----------------------------------------------
  ^          the beginning of the string
-----------------------------------------------
  d{4}       'd' (4 times)
-----------------------------------------------
  $          before an optional \n, and the
             end of the string
-----------------------------------------------
)            end of grouping
-----------------------------------------------

$
```

Listing 4

Bei den ersten eigenen Plugins muss man gegebenenfalls einfach mal ausprobieren vor oder nach welcher Methode der eigene Code ausgeführt werden soll.

Als zweites Beispiel soll ein Plugin erstellt werden, das zu einem Regulären Ausdruck eine Erklärung ausgibt. Die Erläuterung liefert das Modul `YAPE::Regex::Explain`, auf das ich in Ausgabe 4 schonmal genauer eingegangen bin. Das Plugin soll den Befehl `#explain` in der REPL hinzufügen.

```
#!/usr/bin/perl

package Devel::REPL::Plugin::RegexExplain;
use Devel::REPL::Plugin;

use YAPE::Regex::Explain;

use namespace::clean -except => [ 'meta' ];

sub expr_command_explain {
  my ( $self, $eval, $code ) = @_;

  return YAPE::Regex::Explain
                    ->new($code)->explain
}

__PACKAGE__
```

Das Plugin heißt `Devel::REPL::Plugin::RegexExplain` und wird später mit `$repl->load_plugin( 'RegexExplain' );` geladen. Die Verwendung von `namespace::clean` ist nützlich, um den Namensraum des Moduls sauber zu halten. Alle importierten Funktionen - außer die `meta` Methode - werden aus dem Namensraum gelöscht. Die `meta`-Methode wird von MooseX::Object::Pluggable benötigt.

Der Subroutinenname `expr_command_explain` legt fest, dass man später mit `#explain` auf die Funktionalität des Plugins zugreifen kann. Das `expr_command_` ist hier der "Präfix", der `Devel::REPL` mitteilt, dass ein neues Kommando registriert wird. Und das `explain` ist das Kommando.

Mit diesem kurzen Plugin sind Erklärungen zu Regulären Ausdrücken über "explain" möglich - siehe Listing 4.

#Renée Bäcker