Perl und Win32

ActivePerl - Chocolate - Vanilla

Datenbanken

Perl und Datenbankverbindungen

Perl 5.10

Neue Features in der neuen Perl-Version.

Testen mit Perl

Neunter Deutscher

Perl

21.02.- 23.02.2007

www.perl-workshop.de

Workshop

Fachhochschule München

Vorwort

Hallo Perl-Interessierte,

nun ist es endlich soweit - das erste deutschsprachige Perl-Magazin ist erschienen! Viele Programmiersprachen haben ihre Fachzeitschrift und können ihre Möglichkeiten dadurch auch gut in die "Öffentlichkeit" tragen.

Im Internet tauchen immer wieder Meinungen wie "Perl ist tot" auf. Dass dies nicht stimmt und dass Perl viel mächtiger als andere Programmiersprachen ist, soll mit dieser und den zukünftigen Ausgaben von "$foo - das Perl-Magazin" bewiesen werden.

Eine Zeitschrift wie diese schwebt mir schon seit längerem vor. Durch ein paar Ausgaben von "The Perl Review" inspiriert habe ich mir Gedanken über den Aufbau einer deutschsprachigen Zeitschrift gemacht - und jetzt halten Sie das Ergebnis in den Händen.

"$foo - das Perl-Magazin" wird immer in verschiedene Bereiche eingeteilt sein. Für Windows-Nutzer gibt es eine extra "Win32-Ecke" und für die Leseratten wird in jeder Ausgabe ein Buch vorgestellt. Ein "Workshop"-Teil soll einzelne Themen genauer beleuchten und zum Nachmachen animieren. Dieser Teil wird relativ ausgedehnt sein. In dieser Ausgabe wird "Testen mit Perl" den Anfang in dieser "Workshop"-Reihe machen.

Neben dem "Workshop" gibt es immer mehrere allgemeine Themen, die nicht ganz so ausführlich wie das "Workshop"-Thema behandelt werden. Neuigkeiten aus der Perl-Foundation, den Perl-Monger-Gruppen und Termine von Veranstaltungen dürfen natürlich nicht fehlen.

In den einzelnen Ausgaben werden auch verschiedene Projekte vorgestellt. So werden in den nächsten Ausgaben das Poard-Projekt (Forumssoftware der Perl-Community.de) und das Übersetzungs Projekt (von Joergen W. Lang) vorgestellt.

So ein Magazin lebt auch von den Lesern. Ich bitte um Feedback und wer mir Artikel für die zukünftigen Ausgaben schicken will, kann dies gerne tun.

Viel Spaß beim Lesen.

Ihr

Renée Bäcker

Titelseite: abgeändertes Ortsschild der Gemeinde Perl an der Mosel

Inhaltsverzeichnis

Impressum

Herausgeber: Smart-Websolutions Windolph und Bäcker GbR,
Maria-Montessori-Str. 13,
64584 Biebesheim
Tel.: 01803 - 278 25 51 98 (0,09 € / Minute)
E-Mail: redaktion@foo-magazin.de
Redaktion: Renée Bäcker, André Windolph
Anzeigen: Renée Bäcker
Auflage: 500 Exemplare
Druck: Spengler's Druckwerkstatt,
Im Pfützgarten 7,
64572 Büttelborn

Perl ist tot, lang lebe Perl!

Von vielen Seiten bekomme ich immer gesagt, dass Perl doch schon längst "tot" sei und nichts mehr damit passiert. Da kann ich immer nur müde lächeln, denn Perl lebt. Nicht nur die Entwicklung von Perl6 schreitet voran, auch an Perl5 wird noch weitergearbeitet. Die Version 5.10 steht kurz vor der Veröffentlichung und diese Version bietet einige neue Features.

Rafael Garcia-Suarez ist der Pumpkin der Version 5.10 von Perl. Er bereitet die neue Version von Perl mit den Versionen 5.9.x vor. Die Features, die ich hier vorstelle, sind auch schon in den späten Versionen von 5.9 vorhanden.
Dieser Text bezieht sich auf die Version 5.9.4. Bis zu 5.9.5 werden vermutlich noch einige Verbesserungen kommen. 5.9.5 soll dann schon alle Features haben, die dann auch bei Perl 5.10 zu finden sein sollen.

feature

Um die neuen Features verwenden zu können, wurde das Pragma `feature` eingeführt. Alle neuen Features, die über das Pragma `feature` eingeführt werden, haben einen lexikalischen Gültigkeits-bereich, das heißt, man kann sie für einzelne Code-Blöcke "einschalten".

defined-or

In vielen Modulen und Programmen sieht man Zeilen wie diese hier:

```
my $bar = $var || $vb;
# oder
$foo ||= 10;
```

Gerade bei Perl-Einsteigern, kommt es vor, dass diese Zeilen nicht das tun, was der Programmierer will. Auch ich bin anfangs in diese Falle getappt. Dann wenn ich eigentlich sagen wollte "wenn $a keinen Wert hat, dann soll in $foo der Wert von $b stehen". Die zwei Zeilen Code beachten aber nicht, ob $a keinen Wert hat, sondern nur, ob $a "false" ist. In Perl kann aber vieles "false" sein: 0, "0", "", undef,...
Gerade wenn man Default-Werte setzt, will man überprüfen, ob $a nicht definiert ist (auch wenn $a einen "false"-Wert hat). In Perl 5.10 gibt es dann das "defined-or": `//`

```
$var // $vb;
```

ist das gleiche wie

```
defined $var ? $var : $vb
```

Im Code sieht das dann wie folgt aus:

```
my $bar = $var // $vb;
$bar //= 10;
```

In Perl gibt es für solche Operatoren auch gleichbedeutende Operatoren mit niedrigerer Priorität. Für `||` ist es `or`, und für das `//` ist es `err`.

```
use feature qw(err);
my $bar = ($var err $another_var);
```

Da `err` eine niedrige Priorität hat, ist die Verwendung der Klammern notwendig.

say

`say` erlöst die Programmierer vom lästigen `"\n"`-tippen. Die neue `say`-Funktion fügt der Ausgabe ein `"\n"` an - entspricht also einem

```
sub say {
    print @_,"\n";
}
```

Hat man bisher immer so etwas geschrieben:

```
print "dies und das\n";
```

reicht jetzt ein

```
say "dies und das";
```

`say` verkürzt den Code nicht nur durch das mögliche Weglassen von `\n`, sondern auch durch den kürzeren Funktionsnamen. Durch diese Funktion wird auch Zeit - durch weniger Tippaufwand - gespart wird.

Zusätzlich verringert es die Fehlerwahrscheinlichkeit. Hier zwei Beispiele, die "Eifer des Gefechts" (schnelles tippen) manchmal passieren und durch `say` verhindert werden:

```
print 'Hello World\n';
print "Hello World/n";
```

Smart-match

Mit `use feature qw(~~)` wird die Überprüfung, ob ein Element z.B. in einem Array ist, wesentlich vereinfacht. Aber nicht nur Arrays kann man damit überprüfen, sondern alle möglichen Dinge. Man kann sogar überprüfen, ob eine Subroutine bei einem bestimmten Argu-ment "true" zurückliefert.

Ein paar Beispiele (von *Paul Fenwick*):

```
use feature qw(~~ say);
if($x ~~ @array)  {say "$x exists" }
if($x ~~ /ninja/) {
     say "Ninja in String" }
if(@x ~~ /ninja/) {
     say "Ninja in array"  }
if($key ~~ %hash) {
     say "$key exists"     }
```

Bei Subroutinen sieht das Ganze so aus:

```
#!/usr/bin/perl

use strict;
use warnings;
use feature qw(~~ say);

my $sub = sub { return shift == 2 ?
                  1 : 0 };

if($sub ~~ 1){ say "1: true" }
if($sub ~~ 2){ say "2: true" }
```

Und die Ausgabe sieht folgendermaßen aus:

```
[reneeb@localhost 5.9.4_Tests]$ perl
   smart_match2.pl
2: true
```

Die beiden Argumente können auch nach Belieben vertauscht werden. Das heißt, dass `if(@array ~~ 1)` das gleiche liefert wie `if(1 ~~ @array)`.
Der Smart-Match Operator kann wie jeder andere Operator auch überladen werden. Wenn der Code aus Listing 1 ausgeführt wird, erhält man

```
[reneeb@localhost]$ perl smart_match.t
ausserhalb
innerhalb
innerhalb
ausserhalb
[reneeb@localhost$
```

given-when

Das given-when ist das Perl-Pendant zu switch-case in einigen anderen Program-miersprachen.
Es löst Switch.pm ab, das "nur" ein Source-Filter und sehr Fehlerhaft war.
So sieht ein Beispielcode aus:

```
#!/usr/bin/perl

use strict;
use warnings;
use feature qw(say switch);

chomp(my $var = <STDIN>);
given($var){
    when(/\D/)  {
   say "Du musst eine Zahl eingeben!"}
    when($_ == 1) {
   say "Du hast 1 gesagt" }
    when($_ == 2) {
   say "2 ist auch ne schoene Zahl!"}
    default       {
   say "Du hast $_ gesagt" }
}
```

Wie man sieht, wird hier die Variable `$_` für den `given`-Block automatisch gesetzt, so dass die Vergleiche einfacher werden. Nach dem Ausführen des `when`- beziehungsweise `default`-Blocks, wird automatisch `next` aufgerufen.

Wenn man das Beispielprogramm laufen lässt, sieht das so aus:

```
[reneeb@localhost]$ perl given_when.pl
1
Du hast 1 gesagt
[reneeb@localhost]$ perl given_when.pl
d
Du musst eine Zahl eingeben!
[reneeb@localhost]$
```

```
#!/usr/bin/perl

package FooMagazin;

use overload '~~' => sub{
    my ($obj,$check) = @_;
    my ($low,$high)  = @$obj;
  return ($check > $low and $check < $high);
};

sub new{
  my ($class,$low,$high) = @_;
  return bless [$low,$high],$class;
}

package main;

use strict;
use warnings;
use feature qw(switch say ~~);

my $obj   = FooMagazin->new(3,10);
my @array = qw(1 3 8 11);

foreach( @array ){
  when( $obj )  { say "innerhalb";  }
  when(not $_ ~~ $obj ) { say "ausserhalb" }
}
```

Listing 1

Bei der `when`-Bedingung wird auto-matisch der `smart-match`-Operator ge-nommen, so dass man auch sehr einfach eine Überprüfung machen kann, ob eine eingegebene Zahl erlaubt ist:

```
#!/usr/bin/perl
use strict;
use warnings;
use feature qw(say switch);

my @invalid = (10..12);
chomp(my $var = <STDIN>);
given($var){
    when(@invalid) {
         say "ungültig!"}
    default { say "ok!" }
}
```

foreach/when

Zusätzlich zu dem given-when, gibt es auch das foreach-when. Das sieht ziemlich gleich aus:

```
my @array = qw(Dies ist
               ein Test);
for(@array){
    when(/Test/) {  say "Gude" }
}
```

eigene lexikalische Pragmas

Es gibt schon lange lexikalische Pragmas. Zum Beispiel

```
{
    use strict;
    # weiterer Code;
}
```

Mit Perl 5.10 kann man seine eigenen lexikalischen Pragmas schreiben (`feature` wurde so implementiert).
`%^H` erlaubt es, diese "Hin-weise" an den optree zu hängen.
Hier ein kurzes Beispiel, in dem das Pragma `foomagazin` mit der Funktion `hello_world` "Hallo Welt" ausgibt. Wenn man das "Deutsch" ausschaltet, kommt das englische "Hello World" raus.

Das Beispiel zeigt, wie man das "Deutsch" für einen einzelnen Block ausschaltet.
In Listing 2 ist das Pragma zu sehen, in Listing 3 dann das Programm, das von dem
Pragma Gebrauch macht und die folgende Augabe erzeugt.

```
[reneeb@localhost]$ perl lex_test.pl
Hallo Welt!
Hello World!
Hallo Welt!
[reneeb@localhost 5.9.4_Tests]$
```

State-Variablen

State-Variablen speichern einen Zustand. Es sind Variablen mit lexikalischem Gültigkeitsbereich und darauf kann "von außen" nicht zugegriffen werden. Ein Beispiel - wie man es bisher machen würde - sieht so aus:

```
#!/usr/bin/perl

use strict;
use warnings;

{
    my $i;
    sub incrementor{
        return $i++;
    }
}

print incrementor(),"\n",
      incrementor(),"\n";
```

Und erzeugt folgende Ausgabe:

```
[reneeb@localhost]$ state_vars.pl
0
1
```

Der Wert bleibt also persistent erhalten, obwohl man nicht direkt darauf zugreifen kann. Das hängt damit zusammen, dass Perl's Garbage Collector mit einem Referenzzähler arbeitet. Da immer eine Referenz auf `$i` existiert, wird die Variable nie "gelöscht". Würde die Referenz nicht existieren, würde die Variable am Ende des Blockes "gelöscht" werden.
Mit Perl 5.9.4 wurde die Funktion `state` eingeführt. Es funktioniert ähnlich wie die `my`-Variable, allerdings ist die `state`-Variable nicht `undef`.
Bei Perl 5.9.4 sieht die obige Funktion so aus:

```
#!/usr/local/bin/perl
use strict;
use warnings;
use feature qw(state say);
sub incrementor{
    state $foo = 5;
    return $foo++;
}

say incrementor(),"\n",incrementor();
```

```
package foomagazin;

sub import{
    $^H{hello_world} = 1;
}

sub unimport{
    $^H{hello_world} = 0;
}
sub hello_world{
    my $hash = (caller(0))[10];
    if($hash->{hello_world}){
        print "Hallo Welt!\n";
    }
    else{
        print "Hello World!\n";
    }
}


1;
```

Listing 2

Hier wird die die `state`-Variable mit 5 initialisiert - und das nur beim ersten Mal. Danach behält `$foo` den veränderten Wert. Die Ausgabe sieht dann wie folgt aus:

```
[reneeb@localhost reneeb]$ ./state.pl
5
6
```

sonstige Erweiterungen

Hier werden ein paar ausgesuchte Erweiterungen nur ganz kurz angeschnitten.

Einzeiler

Um die Features von Perl 5.10 auch bei Einzeilern nutzen zu können, kann man den umständlichen Weg über

```
perl -e 'use feature qw(say);
      say "feature genutzt"'
```

oder den neuen Schalter -E nehmen, der das äquivalent zu -e ist, nur dass alle neuen Features aktiviert sind. Also

```
perl -E 'say "feature genutzt"'
```

"Constant folding"

Perl verarbeitet Konstanten zur Compile-Zeit. Bei Perl 5.8 und kleiner gibt folgendes Beispiel eine Fehlermeldung `Illegal division by zero at script.pl line 11`:

```
#!/usr/bin/perl

use strict;
use warnings;

use constant NENNER => 10;
use constant ZAEHLER => 0;

my $quot;
if( ZAEHLER ){
    $quot = NENNER / ZAEHLER;
}
```

Ab Perl 5.10 werden solche Fehler erst zur Laufzeit ausgegeben. Der Compiler wird das obige Beispiel wegoptimieren.

Wenn ein etwas abgewandeltes Skript ausgeführt wird, wird wieder die `Illegal division by zero`-Fehlermeldung ausgegeben:

```
#!/usr/bin/perl

use strict;
use warnings;

use constant NENNER => 10;
use constant ZAEHLER => 0;

my $quot;
if( ZAEHLER ){
    $quot = NENNER / ZAEHLER;
}
else{
    $quot = 100 / ZAEHLER;
}
```

UNIVERSAL::DOES

Mit `UNIVERSAL::DOES` kann man über eine Klasse oder ein Objekt herausfinden, welche Rollen es einnimmt. Dabei wird ausgegangen, dass eine Klasse immer die eigene Rolle einnimmt. Eine Rolle ist eine Gruppierung von speziellen Verhaltensweisen (meistens irgendwelche Methodennamen).

`DOES` ist ähnlich wie `isa`, da man weiß - wenn es "true" liefert -, dass ein Objekt oder eine Klasse ein bestimmtes Verhalten kennt. `isa` betrachtet jedoch nur den Vererbungsbaum, während `DOES` sich nicht darum kümmert *wie* dieses Verhalten erzeugt wird.

Man kann die `DOES`-Methode auch überschreiben.

```
#!/usr/bin/perl

use strict;
use warnings;
use foomagazin 'german';

foomagazin::hello_world();

{
   no foomagazin;
    foomagazin::hello_world();
}

foomagazin::hello_world();
```

Listing 3

Hier noch eine kurze (unvollständige) Liste mit weiteren Erweiterungen:

- Field Hashes
- Assertions
- `$AUTOLOAD` ist "tainted" wenn die Methode im Taint-Modus aufgerufen wird
- Source Filter können auf `@INC` angewandt werden
- Verbesserungen bei den Threads
- Bessere UTF-8 Unterstützung
- Schnellere Reguläre Audrücke
- Einige Verbesserungen für Windows
- Mehr Dokumentation
- Geringerer Speicherverbrauch

Es gibt noch weitere Entwicklungen in Perl 5.10, die sehr speziell sind. Diese können in den perl59xdelta.pod Dateien nachgeschlagen werden.

Yves Orton hat die RegEx-engine verändert und einige neue Features eingefügt. Wie diese aussehen, wird in der nächsten Ausgabe erläutert.

Perl und Datenbanken

Perl muss in vielen Fällen mit Datenbanken arbeiten. Datenbanken werden für dynamische Webseiten benötigt oder in den meisten Fällen, in denen riesige Datenmengen verarbeitet werden müssen.

Perl bietet - wie so häufig - verschiedene Wege, um mit einer Datenbank arbeiten zu können. In diesem Artikel wird das Beispielhaft mit MySQL gemacht, wobei es für (fast) alle Datenbanken einen entsprechenden Treiber gibt. Das Zauberwort für die Arbeit mit Datenbanken heißt `DBI`.

`DBI` bedeutet *DataBase Independent Interface* und ist genau das - eine Schnittstelle unabhängig vom dahinterliegenden Datenbanksystem (*DBMS*). Man kann sagen, dass `DBI` als Proxy dient. Es leitet die Anfragen beziehungsweise die Funktionsaufrufe direkt an den Datenbanktreiber weiter wie in Abbildung 1 zu sehen ist.
In `DBI` ist auch festgelegt, welche Funktionen ein solcher Treiber implementieren muss.

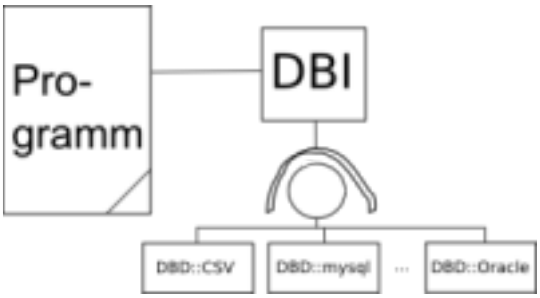


Abbildung 1: DBI als "Proxy" für Datenbanktreiber

Datenbanktreiber

Für die gängigen Datenbanksysteme gibt es einen Treiber, der mit `DBI` verwendet werden kann. Diese Treiber können auf `CPAN` gefunden werden. Einige ausgewählte Treiber sind *DBD::mysql*, *DBD::Oracle*, *DBD::ODBC* und *DBD::CSV*. Selbst auf CSV-Dateien und Excel lässt sich über `DBI` zugreifen.
Die Treiber sind für den DBMS-spezifischen Teil zuständig.

Arbeit mit der Datenbank

Nach den theoretischen Grundlagen geht es jetzt in die praktische Arbeit mit Datenbanken. In den Beispielen wird mit MySQL und dem entsprechenden Treiber DBD::mysql gearbeitet.

Verbindung herstellen

In Listing 1 wird gezeigt, wie die Verbindung zu einer MySQL-Datenbank aussehen würde.

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use DBI;
 6
 7 my $db   = 'NameDerDatenbank';
 8 my $user = 'Datenbank_User';
 9 my $pass = 'Passwort';
10 my $host = 'localhost';
11 my $dsn  = "DBI:mysql:$db:$host";
11
12 my $dbh  = DBI->connect($dsn,
13                $user,$host)
14                or die $DBI::errstr;
15
16 print "Verbindung hergestellt\n";
```

Listing 1: Verbindung zur Datenbank

Die Zeile 12 ist auch die einzige, die bei Verwendung einer anderen Datenbank angepasst werden muss. So kann zum Beispiel die Verbindung zu einer SQLite-Datenbank hergestellt werden - wenn die Zeile 12 durch folgende Zeile ersetzt wird

```
my $dbh = DBI->connect("DBI:SQLite...")
            or die $DBI::errstr;
```

Wenn die Verbindung zur Datenbank hergestellt ist, können verschiedene SQL[2]-Abfragen an die Datenbank gestellt werden. Wie dies aussieht, wird in den folgenden Absätzen erläutert.

Tabellen erzeugen

In diesem Beispiel sollen zwei Tabellen angelegt werden. In der Grafik 2 ist zu sehen, wie die Tabellen aussehen. Die dazugehörigen *CREATE*-Statements wurden mit Hilfe des Moduls `FabForce::DBDesigner4` ermittelt. Um die Tabellen anlegen zu können, wird zuerst wieder

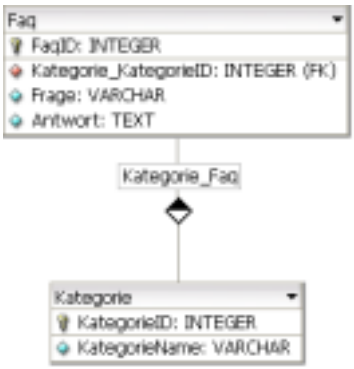


Abbildung 2: Beispieltabellen

eine Verbindung zur Datenbank aufgebaut. Danach wird die der SQL-Befehl "vorbereitet" und ausgeführt. Listing 2 zeigt, wie das Skript dann aussieht.

Durch die `do`-Methode von `DBI` wird der SQL-Befehl ausgeführt und die Tabellen werden erstellt.

Tabelle auslesen

In den meisten Fällen sollen Daten aus der Datenbank ausgelesen werden. Für diese Fälle werden *SELECT*-Befehle benötigt. Vom Prinzip her wird jetzt das gleiche gemacht wie in den anderen Beispielskripten auch. Jetzt werden die Ergebnisse allerdings abgefragt und ausgegeben.

Dazu gibt es verschiedene `fetch*`-Methoden. Im Listing 3 werden die `fetchrow_array`- und die `fetchrow_hashref`-Methode gezeigt.

andere Befehle

Genauso einfach wie die bisher gezeigten Beispiele, können Datensätze eingetragen oder aktualisiert werden, Tabellen geändert oder gelöscht werden. Für Funktionen, die Datenbank-spezifisch sind - wie zum Beispiel Stored Procedures - kann die DBI-Methode C<func> verwendet werden.
Ein einfaches Beispiel für C<INSERT>s sieht folgendermaßen aus:

```
1 my $insert = q~INSERT INTO
2             Kategorie VALUES(?,?)~;
3 $dbh->do($insert,undef,1,
4             'Neue Kategorie');
```

Sicherheit

Gerade bei Webanwendungen spielt die Sicherheit eine sehr große Rolle. Hacker versuchen mit sogenannten SQL-Injections Informationen über die Datenbank und die Inhalte zu bekommen. `DBI` bietet aber ein Feature, mit dem die Sicherheit der

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use DBI;
 6
 7 my $db_name = 'NameDerDatenbank';
 8 my $db_user = 'Datenbank_User';
 9 my $db_pass = 'Datenbank_Passwort';
10 my $db_host = 'localhost';
11
12 my $dbh     = DBI->connect("DBI:mysql:$db_name:$db_host",
13                            $db_user,$db_host)
14                            or die $DBI::errstr;
15
16 my $faq     = qq~CREATE TABLE Faq(
17           FaqID       INT          NOT NULL PRIMARY KEY,
18           Frage       VARCHAR(255) NOT NULL,
19           Antwort     TEXT         NOT NULL,
20           KategorieID INT          NOT NULL,)~;
21
22 my $cat     = qq~CREATE TABLE Kategorie(
23           KategorieID   INT          NOT NULL PRIMARY KEY,
24           KategorieName VARCHAR(100) NOT NULL,)~;
25
26 for($faq,$cat){
27     my $sth     = $dbh->do($_) or die $dbh->errstr();
28 }
```

Listing 2: Anlegen von Tabellen mit DBI

```
 1 # fetchrow_array Beispiel
 2
 3 my $stmt = "SELECT * FROM persons";
 4 my $sth  = $dbh->prepare($stmt);
 5 $sth->execute();
 6
 7 while( my @row = $sth->fetchrow_array() ){
 8     print join(";",@row);
 9 }
10
11 # fetchrow_hashref Beispiel
12
13 my $stmt2 = "SELECT * FROM Faq"
14 my $sth2  = $dbh->prepare($stmt2);
15 $sth2->execute();
16
17 while( my $hashref = $sth2->fetchrow_hashref() ){
18     print "Datensatz:\n";
19     while( my ($key,$value) = each %$hashref ){
20         print "Spalte $key: $value\n";
21     }
22 }
```

Listing 3: SELECT-Statements

Perlskripte stark erhöht werden kann: die ?-Notation. Die gleiche Wirkung kann auch mit der DBI-Methode quote erreicht werden, aber die ?-Schreibweise ist kürzer und kann für Optimierungen verwendet werden.
Das folgende Codefragment zeigt die ?-Schreibweise für ein *SELECT*-Statement.

```
1 my $vorn  = "Tim";
2 my $nachn = "O'Reilly"
3 my $stmt  = q~SELECT * FROM
4     persons WHERE firstname = ?
5     AND name = ?~;
6 my $sth = $dbh->prepare($stmt);
7 $sth->execute($vorn,$nachn);
```

Durch das quote oder die ?-Notation werden Sonderzeichen gequoted damit es bei der SQL-Abfrage keine Probleme gibt.
Zum Beispiel der Nachname "O'Reilly" macht Probleme wenn man String-Werte im SQL einfach mit ' umgibt wie in

```
my $stmt = q~SELECT * FROM persons
     WHERE nachname = '$nachname'~;
```

Wenn dieses Statement ausgeführt wird, tritt ein Fehler auf, weil das ' in "O'Reilly" nicht geschützt ist. Um diesen Fehler zu vermeiden gibt das quote und die ?-Schreibweise. Diese zwei Möglichkeiten sind in Listing 4 gezeigt.

```
 1 # mit quote
 2 my $nachname = "O'Reilly";
 3 $nachname    = $dbh->quote($nachname);
 4 my $stmt     = qq~SELECT * FROM persons
 5       WHERE nachname = '$nachname'~;
 6 my $sth      = $dbh->prepare($stmt);
 7 $sth->execute();
 8
 9 # mit ?
10 my $lastname = "O'Reilly";
11 my $stmt2    = q~SELECT * FROM persons
12        WHERE nachname = ?~;
13 my $sth      = $dbh->prepare($stmt2);
14 $sth->execute($lastname);
```

Listing 4: ?-Notation

Zurück zu den SQL-Injections. Als Beispiel, wo solche Angriffe auftauchen können ist der Login zu einem Admin-Bereich einer Webseite. Viele solcher Login-Skripte im Internet verwenden einen SQL-Befehl wie

```
SELECT count(*) FROM
myusers WHERE id='$name'
AND password='$password'
```

und wenn count() mehr als 1 Element zurückliefert, ist der Nutzer eingeloggt.
Ein Angreifer könnte als ID

```
' OR '1' = '1
```

und das gleiche als Passwort eingeben. Zusammengesetzt sieht dann die SQL-Abfrage so aus:

```
SELECT count(*) FROM myusers WHERE
id='' OR '1'='1' AND password='' OR
'1'='1'
```

Und schon ist der Angreifer im Admin-Bereich. Mit der ?-Notation wäre das nicht passiert, weil dann die ' in den Werten, die vom Angreifer eingegeben wurden, geschützt sind.

Optimierung

Die ?-Schreibweise erhöht nicht nur die Sicherheit, sondern kann auch zur Perfomance-Steigerung verwendet werden.
Viele Skripte sehen wahrscheinlich so aus wie der folgende Code.

```
1 my @daten = qw(dies ist ein Test);
2 for my $value(@daten){
3   my $sth = $dbh->prepare("INSERT
     INTO tabelle VALUES('$value')");
4     $sth->execute();
5 }
```

Das kostet Zeit, weil bei jedem Schleifendurchlauf die SQL-Abfrage "vorbereitet" wird und dann erst ausgeführt wird.
Mit der ?-Schreibweise kann man das beschleunigen, da die SQL-Abfrage immer gleich ist (bis auf den einzufügenden Wert). Diese Beschleunigung funktioniert nicht nur mit *INSERT*s, sondern mit allen Abfragen. Optimiert sieht der Codeausschnitt folgendermaßen aus.

```
1 my @daten = qw(dies ist ein Test);
2 my $stmt  = "INSERT INTO tabelle
                  VALUES(?)"
2 my $sth   = $dbh->prepare($stmt);
3 for my $value(@daten){
4    $sth->execute($value);
5 }
```

Die Fehlerabfrage ist aus Gründen der Übersichtlichkeit weggelassen.

Fallen

Hier werden zwei "Fallen" dargestellt, in die man leicht tappen kann und bei denen die Fehlerfindung nicht ganz einfach ist. Bei diesen Fallen, bleibt der Code nahezu unverändert nur den String bei der Herstellung der Verbindung muss angepasst werden.

MS-Access und .mdw-Dateien

Viele Access-Datenbanken sind durch eine Passwort-Datei geschützt. Diese Dateien enden mit .mdw. Wenn Access diese Datei nicht genannt bekommt, kann man keine Verbindung zu der Datenbank herstellen.
Dazu wird eine Workgroup-Datei anglegt - die .mdw-Datei. Darin werden die Gruppen und die Berechtigungen gespeichert. Mit einer "normalen" Verbindung wird das ganze jetzt fehlschlagen und man wird gebeten, den Administrator der Datenbank um die Rechte zu bitten. Eine kleine Änderung im DSN[1]-String bewirkt aber, dass der Verbindungsaufbau funktioniert.
Im DSN-String muss noch der Key "SystemDB" auftauchen mit dem Pfad zur .mdw-Datei:

```
1 my $dsn  ='driver=Microsoft Access-
  Driver (*.mdb); dbq=c:\database.mdb;
  SystemDB=c:\sicherheit.mdw';
2 my $dbh = DBI->connect(
        "DBI:ODBC:$dsn", $user,$pass)
        or die $DBI::errstr;
```

Oracle ohne Umgebungsvariablen

In manchen Fällen sind die Umgebungsvariablen für Oracle nicht gesetzt - dann findet der Datenbanktreiber die Datenbank nicht und eine Verbindung kann nicht hergestellt werden. Auch in diesem Fall muss einfach der DSN-String angepasst werden:

```
1 my $dsn ='host=myhost.com;sid=ORCL';
2 my $dbh = DBI->connect(
        "DBI:Oracle:$dsn",$user,$pass)
        or die $DBI::errstr;
```

In diesem Fall erzeugt `DBD::Oracle` den vollen Descriptor-String und benötigt nicht tsnames.ora.

Lösung **ohne** DBI

`DBI` ist die praktischste Lösung, um mit Datenbanken zu arbeiten. Es gibt allerdings auch Module, die neben `DBI` existieren. Eine häufig genutzte Variante ist das Arbeiten mit `Win32::ODBC`. Dennoch ist `DBI` in mindestens 95% aller Fälle die bessere Wahl.

Referenzen

http://perl.renee-baecker.de/perl_datenbanken.pdf
http://search.cpan.org/dist/DBI/
http://www.perl.com/pub/1999/10/DBI.html
http://perloo.de/DBI/
http://www.northbound-train.com/perl/article/Article.html
http://aktuell.de.selfhtml.org/artikel/cgiperl/odbc/
http://de.wikipedia.org/wiki/SQL_Injection

[1] Data Source Name
[2] Structured Query Language

Datenbanken ohne SQL

In Zeiten von Abstraktion und Objektorientierung darf auch die Arbeit mit Datenbanken wirken als ob die Zeit stehen geblieben wäre. Die Zeiten, in denen gute SQL-Kentnisse nötig waren, um mit einer Datenbank zu arbeiten, sind vorbei - Dank DBIx::Class.

`DBIx::Class` ist ein Modul für das sogenannte Object-Relational-Mapping (*ORM*). Damit ist es möglich, auf Datenbanken in Objektorientierter Weise zuzugreifen - ohne dass SQL benötigt wird.

Man muss sich zwar weiterhin mit Datenbanken auskennen, aber Kentnisse über Datenbanksystem-spezifische Syntax sind unnötig geworden. `DBIx::Class` kümmert sich um alles.
Wer mit `Catalyst` arbeitet, wird vielleicht schon mit `DBIx::Class` in Berührung gekommen sein.

Abb 1: Aufbau der Bundesliga-Tabelle

In diesem Artikel zeige ich, wie man mit `DBIx::Class` arbeitet - vom einfachsten Beispiel bis hin zu einer Datenbank mit mehreren Tabellen, die zueinander in Verbindung stehen.

Der Einstieg

Zum Einstieg zeige ich eine einfache Datenbank, die nur eine einzige Tabelle hat. Das könnte zum Beispiel eine einfache Bundesligatabelle sein, deren Struktur in Abbildung 1 zu sehen ist.

```
1 package My::DB;
2
3 use strict;
4 use warnings;
5 use base qw(DBIx::Class::Schema);
6
7 __PACKAGE__->load_classes(qw/liste
        der modulnamen/);
8
9 1;
```

Listing 1

Die Arbeit beginnt...

Für die Arbeit mit `DBIx::Class` schreiben wir eine Basisklasse (Listing 1), die alle Module, die für die einzelnen Tabellen benötigt werden, lädt.
Für unseren einfachen Fall, müssen wir nur eine Klasse laden. Die Zeile 7 heißt in unserem Bundesligabeispiel einfach

```
__PACKAGE__->load_classes(qw/Bundesliga/);
```

Jetzt fehlt nur noch das Modul für die Bundesligatabelle. Der Code des Moduls ist in Listing 2 zu sehen.

Bei der Methode `table` muss der Name der Tabelle in der Datenbank übergeben werden, für die das Modul "zuständig" sein soll. Und bei `add_columns` müssen die Namen der Spalten angegeben werden. Man kann auch eigene Accessor-Namen vergeben, wenn die Spalten-namen zu umständlich sind. Wenn zum Beispiel auf `Platz` im Perl-Programm mit der Methode `Pos` zugegriffen werden soll, müsste man das so angeben:

```
__PACKAGE__->add_columns(
        Platz => {accessor => 'Pos'});
```

Als Beispieldaten lade ich drei Vereine in die Tabelle:

```
my @tabelle = (
          [1,'Frankfurt',31,'20:10'],
          [2,'Darsmtadt',27,'19:13'],
          [3,'Mainz'    ,26,'12:12'],);
```

Und schon kann es mit einer ersten kleinen Anwendung losgehen...

```
 1 package My::DB::Bundesliga;
 2
 3 use strict;
 4 use warnings;
 5 use base qw(DBIx::Class);
 6
 7 __PACKAGE__->load_components(qw/PK::Auto Core/);
 8 __PACKAGE__->table('BundesligaTabelle');
 9 __PACKAGE__->add_columns(qw/Platz Verein Punkte Tore/);
10 __PACKAGE__->set_primary_key('Platz');
11
12 1;
```

Listing 2

In Zeile 5 von Listing 3 wird das Modul für das Schema geladen - die Basisklasse. In Zeile 9 werden alle Einträge der Tabelle Bundesliga geholt. Hier wird der Name der Klasse angegeben, die für die Tabelle "zuständig" ist - und nicht der Tabellenname in der Datenbank.
Es wird ein Array zurückgeliefert, in dem die

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use My::DB;
 6
 7 my $schema  = My::DB->connect("DBI:SQLite:dbname=Buli");
 8
 9 my @entries = $schema->resultset('Bundesliga')->all;
10 for my $entry(@entries){
11     print sprintf("%2d %-12s %2d %6s\n",$entry->Platz,
12                                        $entry->Verein,
13                                        $entry->Punkte,
14                                        $entry->Tore);
15 }
```

Listing 3

Objekte vom Typ `My::DB::Bundesliga` gespeichert sind. Über dieses Array kann man jetzt iterieren und die Informationen ausgeben.
Man sieht sehr schnell, dass die Methodennamen, den Spaltennamen entsprechen. Wenn man nicht in dem Modul - wie oben gezeigt - den Accessor-Namen selbst festlegt.
Wenn das Beispielskript ausgeführt wird, bekommt man als Ausgabe

```
~/dbix_test 359> perl buli_test.pl
 1 Frankfurt    31  20:10
 2 Darmstadt    27  19:13
 3 Mainz        26  12:12
```

In der Dokumentation zu `DBIx::Class` ist auch beschrieben, wie man nicht gleich alle Datensätze auslesen muss, sondern wie man mit einem Iterator arbeiten kann. Ich verwende das allerdings nicht, da es damit zu Problemen kommen kann. Zumindest bei `SQLite` stößt man dann schnell an Grenzen - und bekommt `table locked`-Meldungen.

Wer also auf der sicheren Seite sein will, sollte mit `all` alle Datensätze holen und dann über das Array iterieren.

doch wer spielt mit?

Zu der Bundesligatabelle kommt jetzt noch eine weitere Tabelle. Diese soll Spielernamen enthalten und bei welchem Verein diese spielen. Mit der zusätzlichen Tabelle sieht das Datenbankschema aus wie in Abbildung 2 gezeigt.

Folgende Spieler werden in die Tabelle Spieler eingetragen:

```
my @spieler = (
      [1,'Meier',      'Frankfurt'],
      [2,'Takahara',   'Frankfurt'],
      [3,'Amanatidis','Frankfurt'],
      [4,'Cha',        'Mainz'    ],
      [5,'Zidan',      'Mainz'    ],
      [6,'Beierle',    'Darmstadt']);
```

Da jetzt eine weitere Tabelle verwendet werden soll, muss die entsprechende Klasse geschrieben werden, die die Spieler-Tabelle umsetzt.

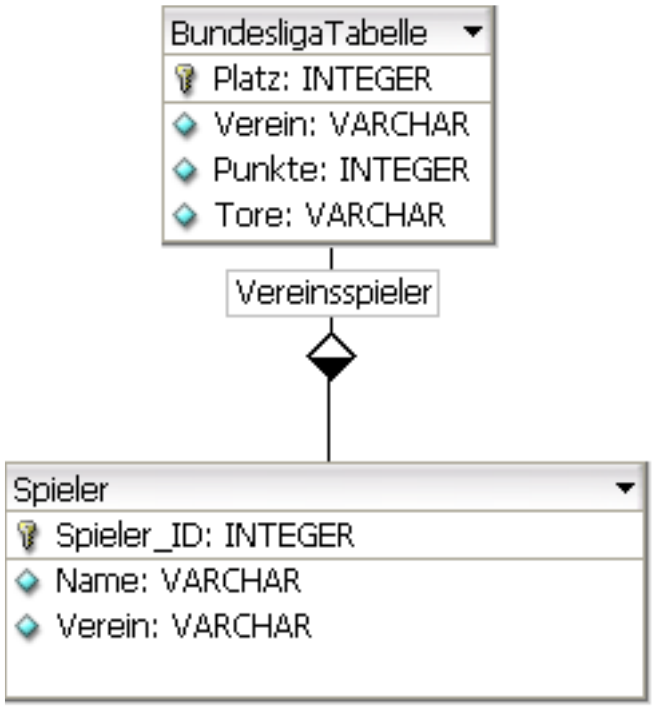


Abb 2: Die Spieler der Vereine

In Zeile 10 sage ich, dass die Tabelle einen zusammengesetzten Primärschlüssel hat, der aus den Spalten `Name` und `Verein` besteht. Mit dem `belongs_to` stelle ich eine Verbindung zwischen der "Bundesliga"-Tabelle und dieser Tabelle her. Da ein `JOIN` aber nicht mit dem Primarschlüssel der "Bundesliga"-Tabelle gemacht werden soll, muss hier in einer Hash-Referenz noch mitgeteilt werden, mit welchen Spalten das gemacht werden soll. Hier kann man noch verschiedene Angaben zu einem `JOIN` machen, zum Beispiel den Typ des `JOIN`s.

Da es aber auch umgekehrt eine Verbindung von der "Bundesliga"-Tabelle zur "Spieler"-Tabelle geben soll, muss die "Bundesliga"-Klasse noch angepasst werden.

```
 1 package My::DB::Player;
 2
 3 use strict;
 4 use warnings;
 5 use base qw(DBIx::Class);
 6
 7 __PACKAGE__->load_components(qw/PK::Auto Core/);
 8 __PACKAGE__->table('Spieler');
 9 __PACKAGE__->add_columns(qw/Spieler_ID Name Verein/);
10 __PACKAGE__->set_primary_key('Spieler_ID');
11
12 __PACKAGE__->belongs_to('Bundesliga' => 'My::DB::Bundesliga',
13                         {'foreign.Verein' => 'self.Verein');
14
15 1;
```

Listing 4

Wir fügen einfach eine Zeile mit

```
__PACKAGE__->has_many(
  'Spieler' => 'My::DB::Player',
 {'foreign.Verein' => 'self.Verein'});
```

ein.
Und damit `DBIx::Class` auch noch Bescheid weiß, dass es eine neue Tabelle gibt, muss das der Basisklasse auch noch mitgeteilt werden:

```
__PACKAGE__->load_classes(
              qw/Bundesliga Player/);
```

Hier wurde einfach noch die "Player"-Klasse mit angegeben.
Und schon kann man sich die Tabellensituation des Ersten ausgeben lassen und sich die dazugehörigen Spieler anzeigen lassen (Listing 5).

Die Zeilen 1-5 bauen die Query auf. Die erste Hash-Referenz in der `search`-Methode enthält die Informationen für die `WHERE`-Klausel. Hier werden also die Datensätze gesucht, bei denen die Spalte `Platz` den Wert `1` hat. Die nächste Hash-Referenz bestimmt dann noch, mit welcher Tabelle der `JOIN` gemacht werden soll. Hier ist der Alias anzugeben, den man in der Klasse angegeben hat (siehe das `has_many`). Mit dem `prefetch` sage ich, dass auch alle Spalten der "Spieler"-Tabelle mit abgefragt werden sollen.

In Zeile 11 rufe ich die Spieler ab. Der Methodenname ergibt sich aus dem Alias, der in der "Bundesliga"-Klasse angegeben wurde.

```
 1 my @joined = $schema->resultset('Bundesliga')
 2                     ->search({Platz    => 1,},
 3                              {join     => ['Spieler'],
 4                               prefetch => qw/Spieler/,},)
 5                     ->all;
 6 for my $info(@joined){
 7     print sprintf("%2d %-12s %2d %6s\n",$info->Pos,
 8                                         $info->Verein,
 9                                         $info->Punkte,
10                                         $info->Tore);
11     my @spieler = $info->Spieler;
12     for my $spiele(@spieler){
13         print sprintf("  %-15s %-12s\n",$spiele->Name,
14                                         $spiele->Verein);
15     }
16 }
```

Listing 5

Die Ausgabe des Testskripts ist diese:

```
1 Frankfurt     31  20:10
  Meier            Frankfurt
  Takahara         Frankfurt
  Amanatidis       Frankfurt
```

Normalisierung

Da diese Datenbank die Information des Vereinsnamens mehrfach speichert, ist einiges an Redundanz da. Deshalb erstelle ich eine neue Tabelle, die nur für den Vereinsnamen da ist. In den beiden anderen Tabellen wird der Vereinsname durch einen Verweis auf diese neue Tabelle ersetzt.

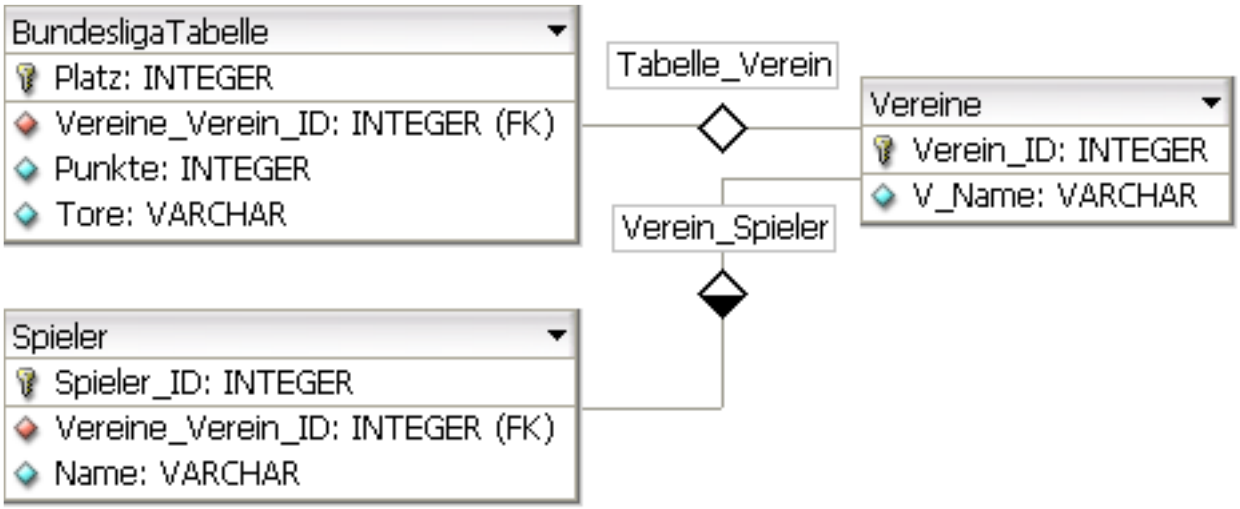


Abb 3: Die Datenbank mit Informatinen über Tabelle, Spieler und Vereine

Dadurch gibt es jetzt zwei `1:n`-Beziehungen, die mit `DBIx::Class` mit `has_many` dargestellt werden.

Die Vereine die eingetragen werden, sind

```
my @vereine =
([1,'Frankfurt'],
 [2,'Darmstadt'],
 [3,'Mainz']);
```

Die Basisklasse muss angepasst werden - die Vereinsklasse muss hinzugefügt werden.

Die "Verein"-Klasse sieht aus wie in Listing 6 gezeigt.

Und in der "Spieler"-Klasse muss die Zeile mit `belongs_to` angepasst werden. Sie verweist jetzt nicht mehr auf die "Bundesliga"-Klasse, sonderen auf die "Verein"-Klasse. In der Zeile steht jetzt

```
__PACKAGE__->belongs_to(
    'Verein' => 'My::DB::Bundesliga',
    'Verein');
```

Auch die "Bundesliga"-Klasse muss verändert werden. Da die Tabelle jetzt einen Fremdschlüssel hat, muss aus der `has_many`-Beziehung ein `belongs_to` werden. So sieht die Zeile jetzt folgendermaßen aus:

```
__PACKAGE__->belongs_to('Vereine' =>
      'My::DB::Verein','Verein');
```

```
 1 package My::DB::Verein;
 2
 3 use strict;
 4 use warnings;
 5 use base qw(DBIx::Class);
 6
 7 __PACKAGE__->load_components(qw/PK::Auto Core/);
 8 __PACKAGE__->table('Vereine');
 9 __PACKAGE__->add_columns(qw/Verein_ID V_Name/);
10 __PACKAGE__->set_primary_key('Verein_ID');
11
12 __PACKAGE__->has_many('Bundesliga' => 'My::DB::Bundesliga',
13                  {'foreign.Verein' => 'self.Verein_ID'});
14 __PACKAGE__->has_many('Spieler'    => 'My::DB::Player'    ,
15                  {'foreign.Verein' => 'self.Verein_ID'});
16
17 1;
```

Listing 6

Wenn ich die Tabelle wie oben haben will, muss mein Skript so aussehen:

```
 1 my @tabelle = $schem->resultset(
        'Bundesliga')
 2                  ->search({},
 3       {join     => [qw/Vereine/],
 4        prefetch => qw/Vereine/}
 5                           )->all;
 6
 7 for my $pos(@tabelle){
 8     print sprintf(
     "%2d %-12s %2d %6s\n",$pos->Pos,
 9              $pos->Vereine->V_Name,
10              $pos->Punkte,
11              $pos->Tore);
12 }
```

Die erste Hash-Referenz bei der C<search>-Methode ist leer, weil wir keine einschränkende `WHERE`-Klausel haben wollen.
Die Informationen zum Tabellenersten und den Spielern zu bekommen, ist etwas schwieriger, weil dann mehrfach ge`JOIN`ed werden muss.
Durch das `prefetch` wird festgelegt, von welchen Tabellen die Spalten zusätzlich abefragt werden sollen. Wenn als *Value* eine Hash-Referenz übergeben wird, macht `DBIx::Class` automatisch einen weiteren `JOIN` daraus.

Die SQL-Abfrage, die aus den `resultset`-Angaben ensteht ist folgende:

```
SELECT
  me.Platz, me.Verein, me.Punkte,
  me.Tore, Vereine.Verein_ID,
  Vereine.V_Name, Spieler.Name,
  Spieler.Verein
FROM
  Bundesliga me
JOIN
  Vereine Vereine
ON
  ( Vereine.Verein_ID = me.Verein )
LEFT JOIN
  Spieler Spieler
ON
 (Spieler.Verein = Vereine.Verein_ID)
WHERE
  ( Platz = ? )
ORDER BY
  Spieler.Verein
```

Geschichte zählt

Um dieses Beispiel noch komplizierter werden. Zu jedem Spieler soll gespeichert werden, bei welchen anderen Vereinen der Spieler schon gespielt hat. Da ein Spieler bei mehreren Vereinen gespielt haben kann und ein Verein mehrere Spieler hat(te), ergibt sich daraus eine `m:n`-Beziehung,
die bei `DBIx::Class` mit `many_to_many` abgebildet wird.

```
 1 my @joined = $schema->resultset('Bundesliga')
 2                     ->search({Platz    => 1,},
 3                              {prefetch => {
 4                                   Vereine => 'Spieler'
 5                                              },
 6                              },
 7                             )
 8                     ->all;
 9 for my $info(@joined){
10     my @vereine = $info->Vereine;
11     print sprintf("%2d %-12s %2d %6s\n",$info->Pos,
12                                        $vereine[0]->V_Name,
13                                        $info->Punkte,
14                                        $info->Tore);
15     my @spieler = $vereine[0]->Spieler;
16     for my $spiele(@spieler){
17         print sprintf("    %-15s %-12s\n",$spiele->Name,
18                                           $spiele->Verein->Vereine->V_Name);
19     }
20 }
```

Listing 6

Um diese Informationen zu speichern, muss eine zusätzliche Tabelle in die Datenbank eingefügt werden. Diese hat in diesem Fall nur zwei Spalten, da der Zeitraum der Vereinszugehörigkeit in diesem Beispiel egal sein soll. In der Tabelle wird die ID des Spielers und die ID des Vereins gespeichert. Das neue Datenbankschema ist in Abbildung 4 dargestellt.

Wie schon bei den Erweiterungen zuvor, muss die Basisklasse erweitert werden. Die neue Klasse muss ebenfalls geladen werden.
In der Tabelle sind für jeden Verein die aktuellen und die ehemaligen Spieler eingetragen.
Die Klasse für die Verbindung steht in Listing 7.

Da diese Tabelle zwei Fremdschlüssel hat, müssen auch zwei `belongs_to`- definiert werden.
Auch die "Spieler"-Klasse und die "Verein"-Klasse müssen an die neue Situation angepasst werden. Es muss jeweils noch eine `has_many`-Beziehung zu der Verbindungstabelle definiert werden.

In der "Spieler"-Klasse werden diese zwei Zeilen hinzugefügt:

```
__PACKAGE__->has_many('Historie' =>
   'My::DB::SpielerVereine',
   'Spieler_ID');
__PACKAGE__->many_to_many(
   'Vereine' => 'Historie','Vereine');
```

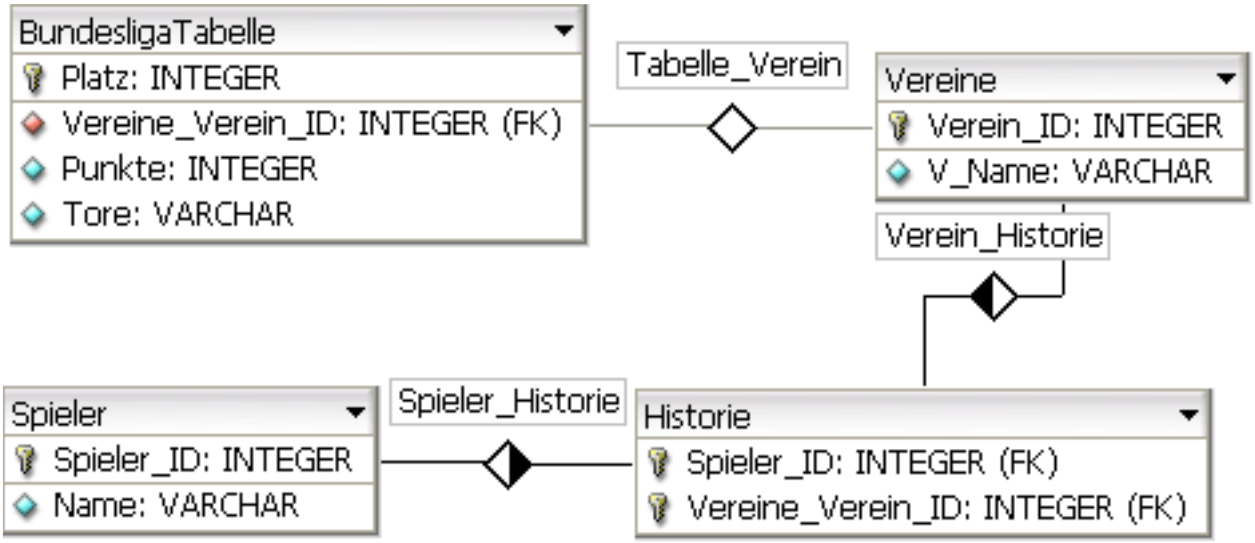


Abb 4: Datenbankschema inklusive Historie der Vereinszugehörigkeit

```
 1 package My::DB::SpielerVereine;
 2
 3 use strict;
 4 use warnings;
 5 use base qw(DBIx::Class);
 6
 7 __PACKAGE__->load_components(qw/PK::Auto Core/);
 8 __PACKAGE__->table('SpielerVereine');
 9 __PACKAGE__->add_columns(qw/Verein_ID Spieler_ID/);
10 __PACKAGE__->set_primary_key('Verein_ID','Spieler_ID');
11
12 __PACKAGE__->belongs_to('Vereine' => 'My::DB::Verein',
13              {'foreign.Verein_ID' => 'self.Verein_ID'});
14 __PACKAGE__->belongs_to('Spieler' => 'My::DB::Player'    ,
15             {'foreign.Spieler_ID' => 'self.Spieler_ID'});
16
17 1;
```

Listing 7

und in der "Verein"-Klasse

```
__PACKAGE__->has_many('Historie' =>
  'My::DB::SpielerVereine',
  'Verein_ID');
__PACKAGE__->many_to_many(
  'Spielers' => 'Historie','Spieler');
```

Die `has_many`-Beziehungen sind notwendig, weil diese Klassen den Fremdschlüssel für die `m:n`-Beziehung stellen. Über `many_to_many`-Beziehungen wird definiert, wie man von der einen Tabelle zur anderen kommt, dabei muss mit angegeben werden, wie der Alias in der Verbindungsklasse heißt.
Jetzt sollen alle aktuellen Frankfurter Spieler herausgesucht werden. Dazu muss das Skript so aussehen:

```
1 my @historie = $schema
      ->resultset('Verein')
2     ->search(
           {V_Name => 'Frankfurt'},)
3     ->all;
4
5 print "ehem. und jetzige
       Frankfurter Spieler:\n"
6 my @spieler_hist =
       $historie[0]->Spielers;
7 for my $spieler(@spieler_hist){
8     print $spieler->Name,"\n";
9 }
```

Damit sieht die Ausgabe so aus:

```
C:\>perl buli_test.pl
ehem. und jetzige Frankfurter Spieler:
Meier
Takahara
Amanatidis
Cha
Beierle
```

weitere Beziehungen

`DBIx::Class` stellt weitere Beziehungen zur Verfügung, die genauso deklariert werden wie die bisher gezeigten Beziehungen.

- 1:1-Bezieung (`has_one`)
- optionale 1:1-Beziehung (`might_have`)

INSERT, UPDATE, ...

Nicht nur `SELECT`s können mit `DBIx::Class` realisiert werden, sondern auch alle anderen Datenbank-Operationen.

Datensatz einfügen

Beim Datensatz einfügen, legt man ein neues `resultset` an und füllt dieses `resultset` mit den gewünschten Daten.

Ich möchte jetzt einen weiteren Verein einfügen, weil die Bundesliga aufgestockt werden soll...

```
1 my $new_club = $schema
       ->resultset('Verein')
2      ->new({Verein_ID => 4});
3 $new_club->V_Name('Bremen');
4 $new_club->insert();
```

In Zeile 1 und 2 wird der neue Datensatz erzeugt. Der `new`-Methode kann man dabei schon Werte für die einzelnen Spalten übergeben. Wie Zeile 3 aber zeigt, kann man auch die Accessoren verwenden, um Werte zu setzen.
Bis zu diesem Punkt ist zwar ein neuer Datensatz generiert worden, aber so steht er noch nicht in der Datenbank. Erst das `insert` aus Zeile 4 speichert den Datensatz in der Datenbank.
Datensatz aktualisieren
Ähnlich funktioniert es mit dem aktualisieren von Datensätzen. Ich habe mich umentschieden, nicht Bremen soll in die Bundesliga aufgenommen werden sondern der 1. FC Hintertupfingen.
Da wir einfach nur den Namen "Bremen" durch "Hintertupfingen" ersetzen wollen, machen wir ein `update`.
Dazu suchen wir zuerst den "Bremen"-Eintrag und ändern dann den Namen.

```
  1 my ($club) = $schema-
>resultset('Verein')
  2                    -
>search({V_Name => 'Bremen'})
  3                    ->all;
  4 $club->V_Name('Hintertupfingen');
  5 $club->update;
```

DBIx::Class beobachten

Wenn man mal Probleme hat, sollte man sich die Abfragen, die von `DBIx::Class` zusammengebaut werden, anschauen. Um zu verfolgen was das Modul so macht, kann man ganz einfach mit `$schema->storage->debug(1)` den Debug-Modus einschalten.
So konnte ich auch einige meiner Probleme lösen.

Links

- http://de.wikipedia.org/wiki/ORM
- http://search.cpan.org/dist/DBIx-Class/

Jochen Lillich: "Radio Perl"

FM: Jochen, erzähl doch zunächst mal etwas über dich und deinen Hintergrund im Perl-Umfeld.

JL: Ich bin 37 Jahre alt, von denen ich mich schon mehr als 12 mit freier Software beschäftige. Das hat während meines Informatik-Studiums angefangen und ist bis heute auch beruflich mein zentrales Thema. Meine Brötchen verdiene ich bei 1&1 als IT-Teamleiter. Mein Einstieg in Perl begann 1996 mit dem Bau meines ersten Webservers. Damals war Perl das übliche Werkzeug für dynamische Websites, von PHP war noch nicht viel zu hören. Ich muss zugeben, dass sich PHP später zunächst eine Weile lang in den Vordergrund drängte, aber ich bin schnell und reumütig zu Perl zurückgekehrt, weil es nicht nur für Websites, sondern auch für Admin-Zwecke so mächtig ist.

FM: Seit 2006 produzierst du jetzt “Radio Perl”. Wie kam es dazu?

JL: Ich bin im letzten Jahr auf das Phänomen der Podcasts gestoßen und habe schnell eine lange Liste von Abonnements aufgebaut. Das Themenspektrum, das mich auf meinen täglichen Zugfahrten zwischen Freiburg und der Firma in Karlsruhe unterhält und weiterbildet, reicht dabei von IT-Management bis “World of Warcraft”. Und weil ich nicht nur ein begeisterter Podcast-Hörer, sondern auch ein experimentierfreudiger Mensch bin, habe ich mir überlegt, selbst einen Podcast zu produzieren. Perl ist eines der Themen, mit denen ich mich stark beschäftige und identifiziere, und so musste ich mir nur noch einen Namen und eine grobe Struktur ausdenken.

FM: Wie sieht diese Struktur aus?

JL: Jede Episode von “Radio Perl” hat einen Themenschwerpunkt, der besonders ausführlich behandelt wird. In der ersten Folge ging es um Perls Zukunftsaussichten und Episode 2 stellte das Web-Anwendungs-Framework Catalyst vor. Ein weiterer fester Bestandteil sind zudem aktuelle Neuigkeiten aus der Entwicklerszene. Und weil Perl einen großen Teil seiner Attraktivität, Flexibilität und Leistungsfähigkeit aus dem CPAN bezieht, stelle ich in jeder Folge Perl-Module aus diesem Netzwerk vor, die mir besonders interessant erscheinen.

FM: “Radio Perl” ist ja noch sehr jung. Wie waren bisher die Reaktionen auf deinen Podcast?

JL: Durch die Bank positiv! Es ist so toll, wenn kurze Zeit nach Veröffentlichung einer neuen Folge die E-Mails und die Kommentare im Blog auftauchen und man mit “Super, weiter so!” oder “Interessant und gute Stimme!” ermutigt wird. Und das ist ja erst der Anfang.

FM: Was hast du denn für die Zukunft geplant?

JL: Ich will auf jeden Fall noch Interviews einstreuen. Es gibt so viele deutschsprachige Entwickler, die interessante Projekte in Perl umsetzen. Die möchte ich gern in meinem Podcast vorstellen. Es schweben mir noch ein paar interessante Dinge vor, die ich aber noch nicht bekannt machen möchte. Bei allen Ideen, die ich habe, bin ich aber auch immer auf die Vorschläge meiner Hörer angewiesen. Ich sammle im Blog zum Podcast die Vorschläge und stelle daraus die Inhalte neuer Folgen zusammen. Die Vorschläge, die bisher reinkamen, reichen für die nächsten paar Folgen, aber ich sammle natürlich schon jetzt neue Themen, die für Perl-Entwickler interessant sein könnten. Wer also Vorschläge für Themen oder Verbesserungen hat, kann mir mit einer E-Mail eine Freude machen.

FM: Ist es viel Aufwand für dich, “Radio Perl” zu produzieren?

JL: Großer Aufwand steckt in der Vorbereitung. Für 30 Minuten Podcast brauche ich etwa 2 Stunden Planungszeit, in der ich ein Thema recherchiere und mir Stichworte zum jeweiligen Thema aufschreibe. So entsteht ein kompletter

Ablaufplan, nach dem ich den Podcast aufnehme. Aus dem Ablaufplan jeder neuen Folge wird dann ein Blogeintrag, in dem vor allem die Links zu den angesprochenen Websites und Perl-Modulen hinterlegt sind. Technisch hält sich der Aufwand dafür stark in Grenzen. Wobei ich sagen muss, dass ich zwar Musiker bin, aber bisher mit Audiorecording nur wenig Erfahrung hatte. Ich hatte darüber gelesen, wie toll die Apple-Rechner für Multimedia-Anwendungen geeignet sind, und hab mir daraufhin einen Mac Mini zugelegt. Mit dabei war die Software-Sammlung "iLive" und darunter auch "Garageband", eine Software für Audioaufnahmen. Mit der ist die Produktion eines Podcasts denkbar einfach: ich nehme den Inhalt auf, schneide ihn und versehe ihn mit Hintergrundmusik. Ich benutze ein Studio-mikrofon, dessen Signale über ein kleines Mischpult an den Mac gehen und mit GarageBand aufgenommen werden. Dann exportiere ich die Aufnahme als MP3-Datei und übertrage sie auf meinen Server, wo sie dann zum Download bereitliegt.

FM: Von welcher Website kann der Podcast heruntergeladen werden?

JL: Die URL der Webseite lautet ganz einfach http://www.radioperl.de. Von dort wird man auf meinen Blog "IT-Dojo" weitergeleitet, wo ich neben Perl-Programmierung in Zukunft auch andere IT-Themen behandeln will.

FM: Gibt es noch etwas, auf was du hinweisen möchtest?

JL: Ja, auf den German Perl Workshop im Februar, den ich besuchen will. Wer ebenfalls dort ist und mich in Person kennenlernen m?chte, kann mir gern eine E-Mail an jochen@lillich.info schicken, um ein Treffen auszumachen. Ich freue mich drauf!

FM: Vielen Dank Jochen, dass du uns von deinem Projekt "Radio Perl" erzählt hast. Wir wünschen dir noch viel Spaß und viel Erfolg damit.

Test make things better

... und Perl ist mittendrin!

Wer kennt sie nicht - die berühmt berüchtigten Bluescreens oder andere Fehlermeldungen? So ziemlich jeder dürfte schonmal vor dem Computer gehockt und eben diesen verflucht haben. Warum passiert so etwas? Und warum ausgerechnet dann wenn ich es überhaupt nicht gebrauchen kann?
Wir sollten unseren Kunden und uns selbst so etwas nicht zumuten. Ein einfaches - aber auch ungeliebtes - Mittel sind Tests. Die Tests können zwar keine 100%ige Sicherheit auf Fehlerfreiheit geben, aber sie erhöhen die Wahrscheinlichkeit, dass das Programm richtig funktioniert.
Bei den Modulen zum Testen von Modulen und der Dokumentation wird auf jede einzelne Methode eingegangen, da diese Methoden in Testskripten sehr häufig verwendet werden. Bei den Modulen zum Testen von Web-Applikationen und Konsolen-Programmen wird nur eine allgemeine Erläuterung gegeben.
In diesem Artikel werden einige Module aus dem Test::*-Namespace vorgestellt und es wird zum Schluss an Hand einer beispielhaften Modulentwicklung gezeigt, wie man von Anfang an mit Tests umgeht. Bevor es aber losgeht, werden noch ein paar allgemeine Dinge zu Tests genannt.

Warum Tests?

Tests haben einen "schlechten Ruf", weil es kein produktiver Code ist. Was aber häufig nicht gesehen wird, ist die Tatsache, das Tests den produktiv-Code noch produktiver macht. Kaum ein Entwickler schreibt gerne Tests, aber wenn man klein anfängt, ist das alles gar nicht mehr so schlimm.

"Ich brauche keine Tests!"

... waren vielleicht die letzten Worte des Ingenieurs, der die Software für die Ariane 5-Rakete einsetzte.
1996 stürzte eine unbenannte Ariane-5 Rakete ab, weil veraltete Software eingesetzt wurde, die noch aus Ariane 4-Zeiten stammte und nicht an die Ariane 5-Begebenheiten angepasst wurden.
Viele "herrliche" Bugs und ihre Auswirkungen sind unter [1] zu finden.
Diese Beispiele zeigen, dass Tests nicht nur "Schönheitsfehler" finden, sondern auch erhebliche Kosten einsparen können.
Programmierer sind Menschen und Menschen machen Fehler. Das ist ja auch (meistens) nicht schlimm, aber man sollte doch versuchen, die Auswirkungen zu begrenzen. Diese Fehler sollten auch nicht unbedingt zu Kunden gelangen.

"Ich weiß was mein Skript macht"

Das will auch niemand anzweifeln. Doch manchmal kommt der Chef am Freitag nachmittag rein und möchte noch etwas am Programm geändert haben - bis zum Wochenende. Da kommt Hektik und Unruhe auf. "Da muss ich nur an dieser Stelle etwas ändern ...". Sicher, dass es nur diese eine Stelle war? In der Hektik oder wenn man sein Skript mal zwei, drei Monate aus den Augen gelassen hat, übersieht man doch mal was.
Wenn man nach einer Änderung die Tests laufen lässt, kann man gleich sehen, ob man nicht doch aus Versehen einen neuen Bug eingebaut hat.

Test-driven Development

Es gibt in der Softwareentwicklung auch den Ansatz, dass die Tests vor dem eigentlichen Code existieren. Wie später gezeigt wird (im Abschnitt `Test::More`), bietet auch Perl eine geeignete Möglichkeit, dies umzusetzen.

Philosophie der Testautomatisierung

Bei der Testautomatisierung gibt es ein paar Dinge, die man beachten sollte. Allerdings ist das Schreiben von Tests etwas "lockerer" als das Schreiben von Produktivcode.
Generell gilt, dass ein Test besser ist als gar keiner. Man kann also mit einem kleinen Satz an Testskripten starten und dann immer weiter aufbauen. Es sollte für jeden gefundenen Bug ein neuer Test geschrieben werden, der auch

[1] http://www5.informatik.tu-muenchen.de/~huckle/bugs.html

überprüft, dass dieser Bug tatsächlich behoben wurde.
Damit stellt man auch gleichzeitig sicher, dass dieser Bug nicht mehr im Produktivcode auftaucht.
Wenn es tatsächlich ein Bug war, sollte der neue Code des Moduls die neuen Tests bestehen, aber die alten Tests nicht. Wenn der neue Code des Moduls die alten Tests besteht, hat man den Bug nicht behoben.
In den folgenden Abschnitten werden noch ein paar Dinge genannt, die man bei der Erstellung einer Testsuite beachten sollte. Dies ist kein Muss, aber es hilft, die Tests auch richtig zu schreiben.

Testfälle generieren

Zu diesem Punkt sollte man gleich sagen, dass dies nicht immer möglich ist. In einigen Fällen muss man auf bestimmte Testfälle zurückgreifen und kann keine Testfälle automatisch generieren. Doch wo es geht, sollten die Testfälle generiert werden, um ein größeres Spektrum zu testen.
Als Beispiel soll hier mal folgendes Modul dienen:

```
 1 package TestPackage;
 2
 3 use strict;
 4 use warnings;
 5
 6 sub summe{
 7     return 12;
 8 }
 9
10 1;
```

In diesem einfachen Beispiel sieht man sofort, dass in der Funktion `summe` nicht wirklich eine Summe gebildet wird. Aber nicht alle Module sind so übersichtlich und Fehler fallen sofort auf.
Manchmal sind es hunderte Zeilen von Code, die einen Fehler verursachen können.
Nun aber zurück zu dem Thema, warum Testfälle generiert werden sollen. Wenn nur hartcodierte Testfälle berücksichtigt werden, können Tests erfolgreich sein, obwohl das Modul eigentlich fehlerhaft ist.
Das folgende Testskript läuft ohne Fehler durch, obwohl das oben gezeigte Modul nicht korrekt arbeitet:

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use TestPackage;
 6 use Test::More tests => 4;
 7
 8 my @testvalues =
          ([1,11],[2,10]);
 9 for my $ref (@testvalues){
10     is(TestPackage::summe(
           @$ref),12);
11 }
12
13 my @test2      = ([6,6],[5,7]);
14 for my $arref (@test2){
15     my ($sum1,$sum2) = @$arref;
16     is($sum1 + $sum2 -
    TestPackage::summe(@$arref),0);
17 }
```

Die Ausgabe ist absolut super! 0 Fehler! Das Modul kann ja nur korrekt arbeiten.

```
C:\programs>perl testskript.pl
1..4
ok 1
ok 2
ok 3
ok 4
```

Wenn man hier Testfälle generiert, dann fällt schnell auf, dass das Modul vielleicht doch nicht so korrekt arbeitet.

Das Testskript angepasst sieht dann so aus:

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use TestPackage;
 6 use Test::More tests => 5;
 7
 8 for my $counter (0..4){
 9     my ($sum1,$sum2) = (int
    rand 100, int rand 40);
10     is(TestPackage::summe(
    $sum1,$sum2),$sum1 + $sum2);
11 }
```

und schon sieht das Testergebnis gar nicht mehr so toll aus (Werte können variieren):

```
C:\programs>perl testskript.pl
1..5
not ok 1
#   Failed test in testskript.pl
    at line 10.
#          got: '12'
#     expected: '55'
not ok 2
#   Failed test in testskript.pl
    at line 10.
#          got: '12'
#     expected: '100'
not ok 3
#   Failed test in testskript.pl
    at line 10.
#          got: '12'
#     expected: '22'
not ok 4
#   Failed test in testskript.pl
    at line 10.
#          got: '12'
#     expected: '98'
not ok 5
#   Failed test in testskript.pl
    at line 10.
#          got: '12'
#     expected: '62'
# Looks like you failed 5 tests of
    5.
```

Wenn das Modul dann korrigiert ist

```
 1 package TestPackage;
 2
 3 use strict;
 4 use warnings;
 5
 6 sub summe{
 7     my ($sum1,$sum2) = @_;
 8     return $sum1 + $sum2;
 9 }
10
11 1;
```

dann läuft auch der Test durch:

```
C:\programs>perl testskript.pl
1..5
ok 1
ok 2
ok 3
ok 4
ok 5
```

False Dilemma

Unter dem "False Dilemma" versteht man den Fall, dass fehlerhafter Code mit einem falschen Test zu einem scheinbaren Erfolg führt.
Wenn ein Skript zum Beispiel das Quadrat einer Zahl berechnen soll und der Code so aussieht

```
1 package Quadrat;
2
3 sub quadrat{
4     my ($zahl) = @_;
5     return $zahl + $zahl;
6 }
```

Wenn ich einen Test schreibe, der so aussieht:

```
1 #!/usr/bin/perl
2
3 use strict;
4 use warnings;
5 use Test::More tests => 2;
6
7 use_ok("Test::More");
8 is(Quadrat::quadrat(2),4);
```

dann läuft der Test durch. Ist ja auch scheinbar richtig, da `4` das Quadrat von `2` ist. Aus diesem Grund sollte man viele Tests machen.

Testreihenfolge

Bei Tests sollte man auch einen Blick auf die Reihenfolge der Tests werfen. Tests sollten immer wieder in anderen Reihenfolgen durchgeführt werden. Auch Tests, die eigentlich andere Tests vorher benötigen, die Variablen setzen, sollten durchgemischt werden. Dann so kann man überprüfen, ob das Programm auch wirklichdie fehlenden Variablen anmeckert. Durch eine feste Testreihenfolge können Testergebnisse verfälscht werden.
Als Beispiel dient das folgende Programm, das zwei Zahlen, die nacheinander eingegeben werden, auf "Perfektheit" überprüft:

```
1 #!/usr/bin/perl
2 use strict;
3 use warnings;
4
5 my $tmp;
6
7 for(0..1){
```

```
 8      print 'Zahl eingeben: ';
 9      my $number = <STDIN>;
10
11      for my $f(1..$number-1){
12          next unless(
  ($number/$f) == int($number/$f));
13          $tmp += $f;
14      }
15
16      print "Perfekte Zahl ? ",
   $tmp == $number ? "Ja" : "Nein";
17      print "\n";
18 }
```

Wenn erst die `6` und dann die `8` eingeben wird, erhält man folgendes Ergebnis:

```
~/entwicklung 296> perl cgi_test.pl
Zahl eingeben: 6
Perfekte Zahl ? Ja
Zahl eingeben: 8
Perfekte Zahl ? Nein
```

Sieht ja echt gut aus. Mein Programm funktioniert, zumindest *scheint* es zu funktionieren. Tauscht man die Zahlen, bekommt man folgendes Ergebnis:

```
~/entwicklung 298> perl cgi_test.pl
Zahl eingeben: 8
Perfekte Zahl ? Nein
Zahl eingeben: 6
Perfekte Zahl ? Nein
```

Uuups, funktioniert doch nicht. Dieses kleine Beispiel soll zeigen, dass unterschiedliche Testreihenfolgen sehr wichtig sein können.

Zwischenfazit

Die einleitenden Abschnitte sollten zeigen wie wichtig Tests sind. Es möchte sich ja auch niemand in ein Auto setzen, wenn vorher keine Sicherheitstests durchgeführt wurden. Die folgenden Punkte sollen einen kleinen Überblick geben, warum Tests wichtig sind.

* Bugs kosten Zeit
* Bugs kosten (viiiiieeel) Geld
* Wir sind Menschen und Menschen machen Fehler
* neues Feature -> neuer Test
* Bugs nerven - Kunden und Programmierer
* Tests erhöhen die Qualität des Programms
* Tests sichern die Funktionalität des Programms

Alle sprechen die gleiche Sprache - TAP

Bei Perl hat sich mittlerweile ein Protokoll etabliert, das bei (nahezu) allen Test-Modulen verwendet wird - TAP.
TAP steht hierbei für `Test Anything Protocol`. Damit wird sichergestellt, dass die Ergebnisse von Tests immer gleich dargestellt werden.
Eine Beipielausgabe eines Tests sieht so aus:

```
~/EigeneModule/My-Module 89> perl -
  Ilib t/test.t
1..373
ok 1 - use My::Module;
ok 2 - The object isa My::Module
ok 3 - My::Module->can(...)
ok 4
ok 5
[..]
ok 17
ok 18
ok 19 - F checked hydro
ok 20 - F checked waal
ok 21 - F checked iso
ok 22 - A checked hydro
ok 23 - A checked waal
ok 24 - A checked iso
```

In der ersten Zeile ist der *Plan* aufgeführt. Was der *Plan* ist, wird bei `Test::Simple` genauer erläutert. In den darauf folgenden Zeilen steht für jeden einzelnen Test, der in test.t gemacht wird, das Ergebnis. Mit `ok` wird gezeigt, dass der Test erfolgreich war und mit `not ok` dass der Test fehlgeschlagen ist. Die Zahl nach `ok` beziehungsweise `not ok` zeigt, welcher Test gelaufen ist.
Bei den meisten Tests kann man noch einen Namen oder eine Nachricht angeben. Damit kann der Test leichter identifiziert
werden wenn etwas schiefgelaufen ist. Wenn zum Beipiel Test 3 fehlschlägt, kann man anhand der Nachricht ("*My::Module->can(...)*") schnell feststellen, dass bei dem `can_ok`-Test (siehe `Test::More`) etwas schiefgelaufen ist und kann sich bei der Fehlersuche auf diesen einen Test beschränken.
Der ganze Aufbau von TAP kann unter `perldoc Test::Harness` eingesehen werden.

Das Test Anything Protocol wird nicht nur bei Perl verwendet. Es gibt einige Implementierungen für andere Sprachen wie Python, PHP und C++.

Parsen des Protokolls

Seit 2006 gibt es auch einen Parser, der dieses Protokoll parsen kann: TAPx::Parser. Das Modul wurde von Curtis 'Ovid' Poe geschrieben und ermöglicht es, die Ausgaben der Testsuite zu parsen und gegebenenfalls weiterzuverarbeiten. Denkbar ist es, die Test-Ergebnisse für eine Webseite aufzubereiten.

Testen von Modulen

Die in diesem Abschnitt beschriebenen Module werden meistens für Tests von anderen Modulen eingesetzt. Sie können aber auch bei Tests von Applikationen eingesetzt.

Test::Harness

Bei Software-Tests ist test harness eine Sammlung von Programmen und Testdaten um ein Programm zu testen. Dies geschieht unter wechselnden Bedingungen und dabei werden die Ergebnisse kontrolliert. Das Perl-Modul Test::Harness übernimmt genau diese Aufgaben: Es startet die Testprogramme und erstellt eine Statistik für die Ergebnisse.
Das Modul bietet auch nur zwei Funktionen:

* runtests
* execute_tests

runtests startet automatisch alle Testskripte, die der Funktion übergeben werden. Ein Beispiel:

```
1 #!/usr/bin/perl
2 use Test::Harness;
3
4 my $file = 'example.t';
5 runtests($file);
```

Wenn example.t so aussieht:

```
1 #!/usr/bin/perl
2 use Test::More tests => 1;
3 ok(1 + 1 == 2);
```

Erhält man folgende Ausgabe:

```
C:\community>test_harness.pl
example....ok
All tests successful
Files=1, Tests=1,  0 wallclock
   secs ( 0.00 cusr +  0.00 csys =
   0.00 CPU)
```

Nach dem Start des Programms wird eine Liste ausgegeben, in der für jedes gestartete Testskript angezeigt wird, ob die Tests erfolgreich waren oder nicht. Nach dieser Liste wird die Statistik ausgegeben. In diesem Fall waren alle Tests erfolgreich. Hier wurde 1 Programm gestartet, das einen Test enthielt.
Ein zweites Testskript wird der Testsuite hinzugefügt. Dieses Skript enthält auch wieder einen einfachen Test.

```
1 #!/usr/bin/perl
2 use Test::More tests => 1;
3 ok(1 + 1 == 3);
```

Dann sollen beide Testskripte gestartet werden:

```
1 #!/usr/bin/perl
2 use Test::Harness;
3
4 my @files = qw(harness_bsp.pl
5                harness_bsp2.pl);
6 runtests(@files);
```

Es ist leicht zu erkennen, dass der Test im zweiten Skript fehlschlagen wird. Dieser Test wurde eingefügt, um die Statistik zu zeigen wenn ein Test fehlschlägt.

Die Ausgabe bei einem Testlauf sieht dann wie in Listing 1 gezeigt aus.

Schon während die Skripte laufen, werden die Tests angezeigt, die fehlschlagen. Hier wird durch das `NOK 1` angezeigt, dass in dem Skript `harness_bsp2` der Test Nummer 1 fehlschlägt. Ganz unten erscheint dann die Aufstellung, welche Tests in welchen Skripten fehlgeschlagen sind. Hier wird `harness_bsp2.pl` aufgeführt und am Ende wird auch eine Liste der fehlgeschlagenen Tests ausgegeben.
Zum Schluss wird eine Zusammenfassung ausgegeben, in der aufgeführt wird, dass 1 von 2 Skripten fehlgeschlagen sind und dass 1 von 2 Subtests nicht erfolgreich waren.
Die Funktion `execute_tests` eignet sich, wenn man eine eigene Statistik erstellen will. Sie liefert zwei Hashreferenzen. Im ersten Hash sind alle Daten zu allen Tests: Wie viele Dateien enthalten sind, wie viele Tests geplant sind, wie viele davon erfolgreich oder fehlgeschlagen sind und noch weitere Informationen. Im zweiten Hash befinden sich die Informationen zu den fehlgeschlagenen Tests. Nach Testskript aufgeschlüsselt finden sich Informationen wie: Welche Tests sind fehlgeschlagen, wie viele Tests geplant waren und noch mehr.
Allerdings ist diese Funktion erst ab Version 2.57 implementiert und mit Perl wird erst aber der Version 5.9.4 mit einer Version > 2.57 ausgeliefert.

Weiterhin gibt es noch drei Parameter zur Konfiguration des Moduls:
Wenn `$Test::Harness::verbose` auf einen *"wahren"* Wert gesetzt wird, wird die Ausgabe der Testskripte mit ausgegeben. Ansonsten werden nur Daten von `STDERR` ausgegeben.
Mit `$Test::Harness::switches` kann man Parameter für den Perl-Interpreter setzen. Standardmäßig wird `-w` gesetzt.
Wenn `$Test::Harness::Timer` gesetzt wird und `Time::HiRes` installiert ist, wird nach jedem Testskript die Laufzeit ausgegeben.

Test::Simple

Der Name ist Programm: *Simple*. Das Modul ist sehr übersichtlich und bietet nur zwei Funktionalitäten, die für einfache Tests ausreichend sind. Dieses Modul beschränkt sich auf zwei grundlegende Funktionen, die auch in `Test::More` umgesetzt sind:
Es muss ein Plan angegeben werden, der angibt wie viele Test gemacht werden und ist somit ein Test in sich.

Listing 1

```
~/entwicklung 33> perl harness_test.pl
harness_bsp.....ok
harness_bsp2....
#   Failed test in harness_bsp2.pl at line 3.
harness_bsp2....NOK 1# Looks like you failed 1 test of 1.
harness_bsp2....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 1
        Failed 1/1 tests, 0.00% okay
Failed Test     Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------------
harness_bsp2.pl    1   256     1    1 100.00%  1
Failed 1/2 test scripts, 50.00% okay. 1/2 subtests failed, 50.00% okay.
```

Sonst gibt es nur noch die Funktion `ok`, mit der einfache Tests gemacht werden können. Als ersten Parameter erwartet die Funktion einen *boolschen* Wert. Ist der Wert *wahr*, wird `ok` ausgegeben, andernfalls ein `not ok`. Als zweiten Parameter kann ein Name für den Test angegeben werden.

Beispiel:

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use Test::Simple tests => 5;
 6
 7 ok(1,'erfolgreicher Test');
 8 ok(0,'nicht erfolgreich');
 9 ok(1 == 1,'1 gleich 1');
10 ok(get_number() == 23,
     'get_number() liefert 23');
11 ok(get_string() eq get_hello(),
     'get_string() equals
      get_hello()');
12
13 sub get_number{
14     23;
15 }
16
17 sub get_string{
18     'hello world!'
19 }
20
21 sub get_hello{
22     'hello world!'
23 }
```

Ausgabe:

```
~/entwicklung 234> perl test_foo.pl
1..5
ok 1 - erfolgreicher Test
not ok 2 - nicht erfolgreicher Test
#   Failed test 'nicht erfolgreich'
#   in test_foo.pl at line 8.
ok 3 - 1 gleich 1
ok 4 - get_number() liefert 23
ok 5 - get_string() equals
  get_hello()
# Looks like you failed 1 test of
  5.
```

Test::More

`Test::More` ist so ziemlich das wichtigste Modul bei Tests. Es bietet einige Funktionen, die Tests erleichtern.

`Test::More` erspart umständliche Aufrufe der `ok`-Funktion wie es in `Test::Simple` notwendig ist. Die `ok`-Funktion existiert zwar weiterhin, aber in den meisten Fällen ist eine andere Funktion besser.

Wichtig - wie bei `Test::Simple` - ist es, einen *Plan* festzulegen. Über den Plan teilt man `Test::More` mit, wie viele Tests gemacht werden sollen. Das ist ein Test in sich, da der Tester eine Meldung bekommt, wenn in dem Testskript mehr oder weniger Tests als angegeben durchgeführt wurden. Das kann darauf hindeuten, dass einige Tests vergessen wurden oder dass mehr Tests als nötig gemacht wurden.

Bei jedem Test, der geschrieben wird, muss also der Plan angepasst werden. In den nachfolgenden Abschnitten werden einige Funktionen von `Test::More` besprochen. Bei allen Tests, die bei `Test::More` gemacht werden, testen das folgende Modul:

```
 1 package FooBar;
 2
 3 sub return_42{
 4     return 42;
 5 }
 6
 7 sub echo{
 8     my ($echo) = @_;
 9     return $echo;
10 }
11
12 sub random{
13     my $text = 'Zufallszahl: '.
    rand($$);
14     return $text;
15 }
16
17 sub lower_case{
18     my ($echo) = @_;
19     $echo = lc $echo;
20     return $echo;
21 }
22
23 sub get_title{
24     my ($html) = @_;
25     my ($title) = $html =~
    m!<title>(.*?)</title>!;
26     return $title;
27 }
28
29 sub todo{
30 }
31
32 1;
```

Es ist natürlich möglich, die `ok`-Funktion zu verwenden, wie sie aus `Test::Simple` bekannt ist. In den meisten Fällen soll aber vermutlich etwas verglichen werden; entweder zwei Strings oder zwei Zahlen. Die erste Methode, die das Leben einfacher macht, ist `cmp_ok`. Diese Funktion benötigt drei Parameter plus den optionalen Test-Namen.
Noch einfacher ist es allerdings mit der `is`-Funktion. Diese Funktion erkennt automatisch, ob Zahlen oder Strings verglichen werden sollen und macht den nötigen Abgleich.
Die drei folgenden Tests sind also äquivalent:

```
1 #!/usr/bin/perl
2
3 use Test::More tests => 3;
4 use FooBar;
5
6 my $number = 42;
7 ok(FooBar::return_42() == 42);
      # ok
8 cmp_ok(FooBar::return_42(),
   '==', 42); # ok
9 is(FooBar::return_42(),42);
         # ok
```

Genauso wie:

```
1 #!/usr/bin/perl
2
3 use Test::More tests => 3;
4 use FooBar;
5
6 my $string = 'Test';
7 ok(FooBar::echo($string) eq
   $string);          # ok
8 cmp_ok(FooBar::echo($string),
   'eq', $string);    # ok
9 is(FooBar::echo($string),
   $string);          # ok
```

Für die Funktion `is` gibt es auch noch das Gegenteil und heißt sinnigerweise `isnt`. Damit können Vergleiche auf Ungleichheit gemacht werden:

```
1 my $string = 'Test';
2 isnt(FooBar::echo($string),
    'test'); # ok
3 is(FooBar::echo($string),
    'test'); # not ok
```

Die eben genannten Funktionen prüfen auf Gleichheit. In einigen Fällen ist es aber so, dass man nicht genau weiß, wie etwas zurückgeliefert wird oder der Zufallszahlengenerator ist mit beteiligt - wie bei

```
1 sub random{
2     my $text = 'Zufallszahl:'.
   rand($$);
3     return $text;
4 }
```

Es ist aber ein Teil des Rückgabewerts bekannt. Im normalen Programm würde man gleich mit Regulären Ausdrücken anfangen. Auch `Test::More` bietet Funktionen, die mit Regulären Ausdrücken umgehen können: `like` und das Gegenteil dazu `unlike`.
Damit kann man auch Funktionen wie das oben beschriebene `random` testen:

```
 1 #!/usr/bin/perl
 2
 3 use Test::More tests => 3;
 4 use FooBar;
 5
 6 my $test     = 'Zufallszahl:';
 7 my $test_zwo = 'ein anderer
     Test ';
 8
 9 like(FooBar::random(),
    qr/$test/);       # ok
10 unlike(FooBar::random(),
    qr/$test/);       # not ok
11
12 like(FooBar::random(),
    qr/$test_zwo/);   # not ok
13 unlike(FooBar::random(),
    qr/$test_zwo/);   # ok
```

Bisher wurden nur einfache Tests gemacht: Strings und Zahlen verglichen. In Perl gibt es aber noch andere Datentypen: Arrays und Hashes. Auch die sollen verglichen werden wenn sie der Rückgabewert einer Funktion sind.
Ein einfacher Vergleich wie dieser:

```
1 my @array_eins = qw(1 2 3);
2 my @array_zwo  = qw(1 2 3);
3 is(\@array_eins,\@array_zwo);
```

geht schief.

```
not ok 1
#   Failed test in test_foo.pl at
      line 9.
#          got: 'ARRAY(0x209624)'
#     expected: 'ARRAY(0x164e74)'
```

Da muss also etwas anderes her. `Test::More` hat auch hier die entsprechende Funktion: is_deeply.
Diese Funktion kann aber nur einfache Hashes beziehungsweise Arrays vergleichen. Für komplexe Datenstrukturen sind die Module `Test::Differences` und `Test::Deep` geeignet. Aber für Arrays und Hashes ist die `is_deeply`-Funktion von `Test::More` sehr gut geeignet:

```
1 #!/usr/bin/perl
2
3 use Test::More tests => 2;
4
5 my @array_1 = qw(1 2 3);
6 my @array_2  = qw(1 2 3);
7 my @array_3 = qw(1 2 4);
8 is_deeply(\@array_1,\@array_2);
9 is_deeply(\@array_1,\@array_3);
```

Ergibt folgende Ausgabe:

```
~/entwicklung 127> perl test_foo.pl
1..2
ok 1
not ok 2
#   Failed test in test_foo.pl at
        line 9.
#     Structures begin differing
         at:
#           $got->[2] = '3'
#     $expected->[2] = '4'
# Looks like you failed 1 test of
         2.
```

Es wird also auch gut aufgezeigt, an welcher Stelle etwas schief gelaufen ist.

An der Testausgabe sieht man, dass es auch Zusatzinformationen zu einem Test gibt. In diesem Fall wird angegeben, welche Elemente der Arrays ungleich sind. Solche zusätzlichen Ausgaben - die `Test::Harness` übrigens ignoriert werden - können auch selbst ausgegeben werden.

Dafür gibt die Funktion `diag`:

```
1 #!/usr/bin/perl
2
3 use Test::More tests => 1;
4
5 diag('Diagnosemeldung: Test1');
6 is(42,42);
```

Und die Ausgabe:

```
~/entwicklung 128> perl test_foo.pl
1..1
# Diagnosemeldung: Test1
ok 1
```

Für die Überprüfung, ob alle gewünschten Funktionen in einem Modul implementiert sind, gibt es die Funktion `can_ok`. Die Funktion akzeptiert sowohl ein Objekt als auch ein Modulname für den Test:

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use Test::More tests => 1;
 6 use FooBar;
 7
 8 my @methods = qw(echo return_42
 9              lower_case random
10              get_title todo);
11
12 can_ok('FooBar',@methods);
```

Da alle diese Methoden in dem Modul implementiert sind, bekommt erscheint diese Ausgabe:

```
~/entwicklung 129> perl test_foo.pl
1..1
ok 1 - FooBar->can(...)
```

Für Objektorientierte Programmierung gibt es noch die Funktion isa_ok, die ein Objekt überprüft, ob es zu einer gegebenen Klasse gehört:

```
1 #!/usr/bin/perl
2
3 use Test::More tests => 1;
4 use CGI;
5
6 my $cgi = CGI->new();
7 isa_ok($cgi,'CGI');
```

Da `$cgi` ein Objekt con `CGI` ist, erhält man folgende Ausgabe:

```
~/entwicklung 130> perl test_foo.pl
1..1
ok 1 - The object isa CGI
```

Soll das Laden von Modulen getestet werden, stehen zwei Funktionen aus `Test::More` zur Verfügung:
`use_ok` und `require_ok`. Die Namen sind dabei Programm: `use_ok` bindet ein Modul über `use` ein und das `require_ok` benutzt das `require`.

Mit folgendem Skript wird getestet ob zwei bestimmte Module installiert sind:

```
1 #!/usr/bin/perl
2
3 use strict;
4 use warnings;
5 use Test::More tests => 2;
6
7 use_ok('CGI');
8 use_ok('NonExModule');
```

Da das zweite Modul nicht installiert ist, erscheint folgende Ausgabe:

```
~/entwicklung 154> perl test_foo.pl
1..2
ok 1 - use CGI;
not ok 2 - use NonExistentModule;
#   Failed test 'use NonExModule;'
#   in test_foo.pl at line 8.
#     Tried to use 'NonExModule'.
#     Error:  Can't locate
  NonExModule.pm
  in @INC (@INC contains: ... .)
  at (eval 4) line 2.
# BEGIN failed--compilation
  aborted at test_foo.pl line 8.
# Looks like you failed 1 test of
  2.
```

Die letzten beiden Punkte von `Test::More` sind der `SKIP`- und der `TODO`-Block. Mit dem `SKIP`-Block kann man einen Teil des Codes auslassen wenn eine bestimmte Bedingung erfüllt ist. Dies ist sehr nützlich, wenn für den Test eine Internetverbindung benötigt wird, bei einer nicht vorhandenen Internetverbindung nicht alle Tests fehlschlagen sollen.
Sonst begibt man sich auf Fehlersuche und dabei lag es nur an der fehlenden Verbindung.
Ein weiterer - noch häufigerer - Anwendungsfall ist das Abfragen von Modulen. Wenn für Tests bestimmte Module benötigt werden, die auf dem Rechner vielleicht nicht installiert sind, macht es keinen Sinn, den Test weiter auszuführen.
Die folgenden zwei `SKIP`-Blöcke sollen dies demonstrieren:

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use Test::More tests => 2;
 6 use LWP::Simple;
 7
 8 SKIP:{
 9     eval "use NonExMod";
10     skip "NonExMod not
          installed",1 if $@;
11
12     NonExMod::subroutine();
13 }
14
15 SKIP:{
16    my $url = 'http://www.1.de/';
17    my $res = get();
18     skip "Keine Verbindung",1
    unless $res;
19
20     require Foo;
21     is(Foo::get_title($res),
         'Test');
22 }
```

Da beides nicht funktioniert, wird folgende Meldung ausgegeben:

```
  ~/entwicklung 161> perl
test_foo.pl
  1..2
  ok 1 # skip NonExistentMod not
installed
  ok 2 # skip Keine Verbindung
```

Mit dem `TODO`-Block kann angegeben werden, welche Tests fehlschlagen werden, weil die Funktionalität noch nicht implementiert ist. In dem Beispielmodul soll die Funktion `todo` in Zukunft mal den Satz `"Hallo Welt!"` zurückliefern. Der Funktionsrahmen ist zwar schon aufgeschrieben, die Funktion ist aber noch nicht mit Leben gefüllt.

So könnte also ein Test aussehen:

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use Test::More tests => 1;
 6 use FooBar;
 7
 8 TODO:{
 9     local $TODO = 'not yet
       implemented';
10     is(FooBar::todo(),'Hallo
       Welt!','teste todo()');
11 }
```

Die Testausgabe sieht dann so aus:

```
~/entwicklung 243> perl test_foo.pl
1..1
not ok 1 - teste todo() # TODO not
  yet implemented
#   Failed (TODO) test 'teste
  todo()'
#   in test_foo.pl at line 10.
#          got: undef
#     expected: 'Hallo Welt!'
```

Wenn eine Testsuite mit `Test::Harness` gestartet wird, werden diese Tests im `TODO`-Block nicht als fehlgeschlagen gewertet.

Test::Exception

Mit `Test::Exception` können Module auf Warnungen und Fehler hin überprüft werden. Damit kann zum Beispiel überprüft werden, ob das Modul tatsächlich abbricht wenn eine nicht-existente Datei geöffnet werden soll.

Das Modul stellt die folgenden vier Methoden zur Verfügung.

- dies_ok
- throws_ok
- lives_ok
- lives_and

Mit `dies_ok` überprüft man, ob eine Funktion wie gewünscht abbricht. Programme liefern häufig falsche Ergebnisse wenn bei bestimmten Ereignissen nicht abgebrochen wird. Deshalb sollte ein Programm mit wissentlich falschen Daten gefüttert werden, um zu überprüfen, ob das Programm richtig darauf reagiert.

`throws_ok` überprüft auch, ob die Fehlermeldung durch einen bestimmten Regulären Ausdruck gematcht wird. Dies ist sinnvoll, wenn nicht nur überprüft werden soll, ob das Programm abbricht, sondern ob der "richtige" Fehler ausgegeben wird.

Eine Methode, die auf jeden Fall durchläuft, kann mit `lives_ok` getestet werden. Hierbei ist es völlig egal, welchen Rückgabewert die Methode hat. Nicht egal ist dies wiederum bei `lives_and`. Hier wird noch ein zusätzlicher Test auf den Rückgabewert gemacht.

Dieses Modul soll getestet werden:

```
 1 package FooBar;
 2
 3 sub dies{
 4     my $file = '/test.upc';
 5     open my $fh,'<',$file or
             die $!;
 6     close $fh;
 7 }
 8
 9 sub throws{
10     my ($nenner) = @_;
11     my $zahl = 19 / $nenner;
12 }
13
14 sub lives{
15     1;
16 }
17
18 sub lives_42{
19     42;
20 }
21
22 1;
```

Folgendes Testskript benutzt die vier Methoden von `Test::Exception` um den Einsatz des Moduls zu verdeutlichen:

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use Test::More tests => 4;
 6 use Test::Exception;
 7 use FooBar;
 8
 9 dies_ok {FooBar::dies() }
10         'died as expected';
11 throws_ok {FooBar::throws(0) }
12    qr/division by zero/i,
13    'Division by 0 not allowed';
14 lives_ok {FooBar::lives() }
15    'no matter what is returned';
16 lives_and {is
      FooBar::lives_42(), 42}
17   'lives_42 returns 42';
```

Die Ausgabe des Tests sieht wie folgt aus:

```
  ~/entwicklung 79> perl test_foo.pl
  1..4
  ok 1 - died as expected
  ok 2 - Division by 0 not allowed
  ok 3 - it does not matter what
lives() returns
  ok 4 - lives_42 returns 42
```

Test::TestCoverage

Es wurde von mehreren Personen angemerkt, dass `Test::TestCoverage` sehr dem Modul `Devel::Cover` ähnelt. Es gibt Ähnlichkeiten im Sinn und Zweck der Module, aber `Test::TestCoverage` verfolgt eine etwas andere Strategie.

Test::Coverage vs Devel::Cover

Mit `Devel::Cover` kann überprüft werden, wie häufig bestimmt Code-Stücke aufgerufen wurde. So kann man feststellen, ob unnützer Code geschrieben wurde. `Test::TestCoverage` interessiert sich nicht dafür, *wie oft* ein Code-Stück aufgerufen wurde und erzeugt auch nicht den Overhead an Statistiken.
`Test::TestCoverage` interessiert sich nur dafür, ob auch tatsächlich alle "`public`"-Methoden im Testskript aufgerufen wurden.

Nachfolgend werden die Funktionen von Test::TestCoverage erläutert.

- test_coverage
- ok_test_coverage
- reset_test_coverage
- reset_all_test_coverage

Mit test_coverage wird ein Modul zur Überprüfung "angemeldet". Wenn das Modul noch nicht geladen ist, dann wird es automatisch geladen. Dies ist auch notwendig, damit `Test::TestCoverage` die Subroutinen des Moduls herausfindet.
Der eigentliche Test ist `ok_test_coverage`. Standardmäßig wird dabei das zuletzt angemeldete Modul überprüft. Soll ein anderes Modul getestet werden, muss der Name als Parameter übergeben werden.
Wurden mehrere Module "angemeldet", kann man die Tests auch vereinfachen und über `all_test_coverage_ok` alle Module auf einmal testen.

Als Beispielmodul wird folgendes Modul genommen:

```
1 package FooBar;
2
3 sub new{ bless {},shift }
4
5 sub echo{
6     my ($self,$echo) = @_;
7     print $echo,"\n";
8 }
9 1;
```

Dieses Modul soll getestet werden und bei dem Test soll darauf geachtet werden, dass alle public-Methode verwendet werden (hier: `new` und `echo`). Das Testskript sieht dann wie folgt aus:

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use Test::More tests=>1;
 6 use Test::TestCoverage;
 7
 8 test_coverage('FooBar');
 9
10 my $foo = FooBar->new();
11
12 ok_test_coverage();
```

Wenn der Test so läuft, erhält man folgende Ausgabe:

```
~/entwicklung 67> perl test_foo.pl
1..1
not ok 1 - Test test-coverage echo
   are missing
#   Failed test 'Test test-
   coverage echo  are missing'
#   in test_foo.pl at line 12.
#          got: 0
#     expected: 1
# Looks like you failed 1 test of
   1.
```

Damit wird angezeigt, dass die Methode `echo` nicht aufgerufen wird. Fügt man einen Aufruf der Methode ein, so sieht das Testskript folgendermaßen aus:

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use Test::More tests=>1;
 6 use Test::TestCoverage;
 7
 8 test_coverage('FooBar');
 9
10 my $foo = FooBar->new();
11 $foo->echo('Hallo Welt');
12
13 ok_test_coverage();
```

Jetzt läuft der Test fehlerfrei durch:

```
~/entwicklung 69> perl test_foo.pl
1..1
Hallo Welt
ok 1 - Test test-coverage
```

Das Modul hat noch einige Schwächen, da zum Beispiel exportierte Methoden noch nicht überprüft werden können. Da ist `Devel::Cover` schon einiges weiter.

Test::CheckManifest

Dieses Modul ist eigentlich kein Test-Modul um Funktionalität zu sichern, sondern unterstützt den Programmierer bei der Einhaltung von CPAN-Konformität. In der `MANIFEST`-Datei sind alle Dateien aufgeführt, die zu einer Distribution gehören - zumindest sollte es so sein.
Ob dies auch tatsächlich der Fall ist, kann sehr einfach mit `Test::CheckManifest` überprüft werden. Ein

```
1 use Test::More;
2
3 eval "use Test::CheckManifest
    1.0";
4 plan skip_all =>
    "Test::CheckManifest 1.0
    required" if $@;
5
6 ok_manifest();
```

ist ausreichend.

Die Funktion `ok_manifest` ist die einzige Methode, die es in dem Modul gibt. Der Funktion kann man noch über Filter und Pfadangaben mitteilen, welche Dateien und Ordner nicht zur Distribution gehören und keine Fehlermeldung erzeugen sollen.
Dies ist besonders sinnvoll, wenn man mit einem Versionierungstool wie SVN oder CVS arbeitet. Über die Pfadangaben, können die Ordner des Tools von der Überprüfung ausschließen.

Links

- http://qa.perl.com/test-modules
- http://www.petdance.com/perl/automated-testing/

Perl auf Windows

- und sie passen doch zusammen...

Unter Perl-Programmierern hat man sehr lange Zeit kaum einen Windows-An-wender gefunden. Das lag mit Sicherheit auch daran, dass Perl bei (fast) jedem Linux/Unix-System dabei ist, aber unter Windows nicht zu finden ist.
Perl unter Windows zu installieren ist auch auch nicht immer der größte Spaß. Zum Glück gibt es mittlerweile einige verschiedene Wege, wie man seinem Windows-Rechner Perl beibringen kann. Die bekannteste Perl-Distribution für Windows ist wohl ActivePerl von ActiveState.
Seit 2006 gibt es unter `http://win32.perl.org` ein Portal für alle Perl-Programmierer, die mit Windows arbeiten.

ActivePerl

`ActivePerl` ist der bekannteste Ableger von Perl für die Windows-Welt. Die Perl-Distribution ist kann kostenlos von der Webseite von ActiveState heruntergeladen werden. Für Unternehmen, die es sich leisten wollen, gibt es auch kosten-pflichtigen Support von ActiveState.

PPM

Über den Perl Package Manager (*PPM*) lassen sich Module sehr leicht nachträglich installieren. Abhängigkeiten löst PPM selbständig. Die Module, die als PPD - dem Format für den PPM - vorliegen, sind schon für vorkompiliert.
Allerdings gibt es nicht alle Module in der PPM-Version. Vor allem ActiveState erneuert die Module nur sehr langsam und auf den Build-Status kann man sich nicht unbedingt verlassen.
Wird ein Modul unbedingt benötigt, kann man bei Randy Kobes nachfragen, ob er das Modul in der PPM-Version bereitstellt.

cygwin

Cygwin ist für die *nix-Freunde unter den Windows-Nutzern sehr gut geeignet. Das Paket emuliert ein Linux unter Windows und liefert neben Perl noch weitere *nix-typische Tools wie `grep`, `diff` und `awk`.
Das Paket ist ziemlich groß, aber für *nix-Liebhaber ist dieses Paket ein Muss.

PXPerl

`PXPerl` ist vor allem durch `Pugs` - einer Perl6-Implementierung - bekannt geworden. Allerdings wird `PXPerl` nicht mehr weiterentwickelt und es findet sich nur noch auf ein paar Seiten, die freie Downloads anbieten. Hier ist kein Support mehr zu erwarten.
Diese Distribution ist nicht mehr zu empfehlen außer man will mit Parrot und Pugs spielen. Mittlerweile gibt es jedoch schon andere Installer für Windows, die Pugs beinhalten.

CamelPack

Das `CamelPack` wurde innerhalb von zwei Tagen erschaffen. Adam Kennedy hat in einem Wettbewerb dazu aufgerufen, möglichst schnell eine Perl-Distribution zu erschaffen, die auf Windows läuft und alles wichtige - wie einen C++-Compiler mitbringt.

VanillaPerl / StrawberryPerl

Die "Geschmacks"-Perl-Distributionen sollen in Zukunft einen Ersatz für ActivePerl - in Bezug auf Komfort bei der Modul-Installation und so weiter - darstellen. Diese Pakete befinden sich noch in der Entwicklung. Ziel ist es, einen Compiler für die XS- und Inline-Module zu bieten und dem Nutzer einen größtmöglichen Komfort zu bieten.

sonstige Distributionen

Es gibt noch einige weitere Perl-Distributionen für Windows, die aber eher unbekannt sind. So gibt es zum Beispiel mit `SiePer` ein Perl, das von Siemens bereitgestellt wird.

Eine weitere interessante Perl-Distribution ist `niPerl`, das auch mit einem kleinen Editor ausgeliefert wird. Mit `niPerl` kann mit wenigen Klicks eine `.exe` für Windows erstellt werden.

kombinierte Distributionen

Unter kombinierten Distributionen sind Pakete zu verstehen, die neben Perl noch andere Programme mitliefern. Das bekannteste Beispiel ist `XAMPP` von Apachefriends.org, das den *Apache*-Webserver plus *MySQL* plus *Perl* ausliefert. Dies ist ganz praktisch wenn eine Testumgebung für *CGI*-Skripte aufgebaut werden soll.

win32.perl.org

Unter http://win32.perl.org ist ein Wiki zu dem Thema *"Perl und Windows"* erreichbar. Dort werden Windows-spezifische Perl-Module vorgestellt und Links zu Beispielen gesammelt. Weiterhin werden auf Probleme mit Modulen hingewiesen. Einige Module sind nicht für Windows gedacht, so dass sich in dem Wiki Hinweise finden, warum diese Module nicht für Win32 gedacht sind.
Auch die `ActivePerl`-Alternativen `VanillaPerl`, `StrawberryPerl` und `ChocolatPerl` werden hier vorgestellt. Die Entwickler dieser drei Perl-Distributionen sind in dem Wiki sehr aktiv.
In der Übersicht über Perl-Distributionen für Windows finden sich auch zusätzliche Informationen darüber, in welcher Form der Installer vorliegt, welche Perl-Version in der aktuellen Distribution verwendet wird und noch weitere Angaben.
Auch Diskussionen zu den verschiedenen Themen gibt es. Die Initiatoren von win32.perl.org freuen sich über jeden Freiwilligen, der hilft, Informationen zu sammeln und zu verbreiten.

Name	Perl	Installer	ActivePerl	PPM	CPAN	PAR	Maturity	Support
Strawberry Perl	5.8.8	exe	no	no	yes	???	alpha	community
Vanilla Perl	5.8.8	exe	no	no	yes	no	experimental	none
ActivePerl	5.8.8	msi	yes	yes	no	yes	stable	commercial
CamelPack	5.8.7	exe	yes	yes	yes	???	stable	author
niPerl	5.8.7	msi	???	no	no	yes	stable	community
PXPerl	5.8.7	exe	yes	no	yes	no	abandoned	none
DeveloperSite.Net	5.8.7	exe	no	no	no	no	stable	community
Apache Perl	5.8.7	exe	yes	yes	no	yes	stable	author
Randy Kobes	5.8.7	exe	yes	yes	no	yes	stable	author
Cygwin Perl	5.8.7	bundled	no	no	yes	no	stable	community
Indigo Perl	5.8.6	zip	no	yes	no	???	stable	author
SiePerl	5.8.0	exe	no	no	???	???	stable	???
XAMPP + Perl	???	???	???	???	???	???	???	???
nsPerl	5.005_03	zip	no	no	no	no	abandoned	none
Chocolate Perl	TBA	exe	maybe	no	yes	yes	concept	n/a

Fazit

Auch unter Windows lässt sich hervorragend mit Perl arbeiten. Die meisten wichtigen Perl-Module können auch unter Windows problemlos installiert werden und über den Perl Package Manager ist das auch sehr komfortabel.
Es gibt ein paar kleine Fallen, die man bei der Arbeit unter Windows beachten muss. So verlangen manche Funktionen in Perl die Windows-Pfadtrenner und manche können mit dem *nix-typischen '/' umgehen. Wie man mit Perl auch in die Tiefen von Windows eintauchen kann - wie man zum Beispiel Word und Excel automatisieren kann oder wie die Registry ausgelesen werden kann - wird in den kommenden Ausgaben von *"$foo - Perl-Magazin"* gezeigt.

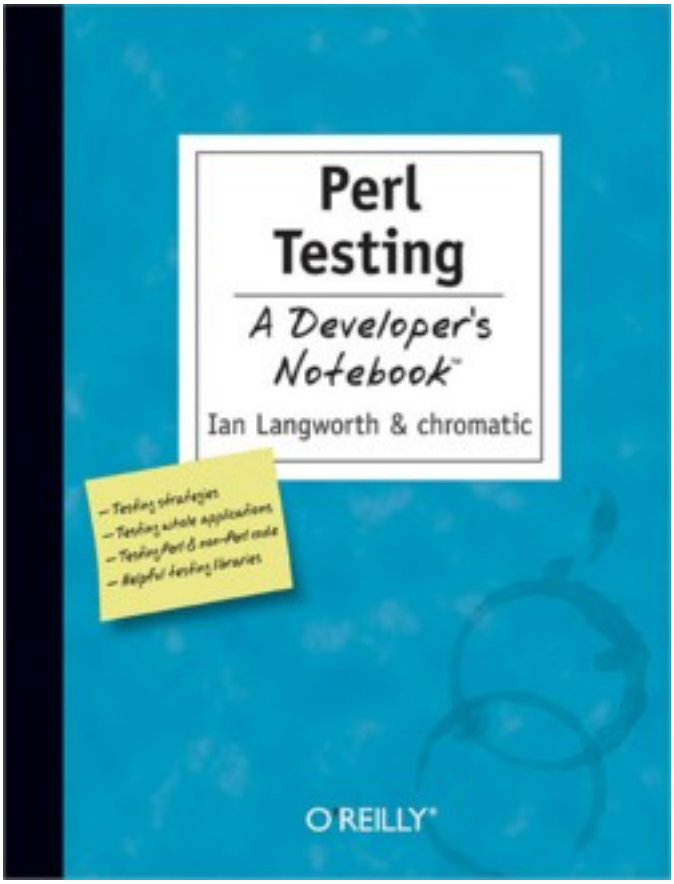
Referenzen

- http://win32.perl.org - Informationsquelle für Perl unter Windows
- http://www.activestate.com - ActiveState.com
- http://www.numeninest.com/Perl/ - niPerl
- http://www.cpan.org/authors/id/G/GR/GRAHAMC/ - SiePerl

Perl-Testing Buch

Ian Langworth & chromatic
O'Reilly
ISBN 0596100922
englischsprachig

Tests von Programmen, Webseiten, Datenbanken,... sind sehr wichtig. Auch wenndas Schreiben von Tests keinen Spaß machen, sind sie doch wichtig, um die Qualität und Funktionalität von Programmen zu sichern. Das Buch "Perl Testing - a Developer's Notebook" bietet jede Menge Tipps rund um das Thema "Testen mit Perl".



Ich habe eine recht "trockene" Materie erwartet, als ich das Buch zum Lesen bekommen habe. Da das Buch aber sehr praxisnah geschrieben ist, kommt man schnell durch.

Jeder Teil des Buches beginnt mit einer kurzen Einleitung und danach kommt die Erklärung mit "How do I do that?". In diesem Teil ist dann immer recht genau beschrieben wie das Problem gelöst werden kann.

Der "What just happened?"-Teil erläutert dann, was passiert ist und warum das Ergebnis so ausfällt, wie es kommt. Ein sehr interessanter Teil ist dann "What about...", der in Frage-Antwort-Stil einige Randthemen anspricht und einzelne Punkte genauer beleuchtet.

Das Buch fängt mit einer allgemeinen Einführung in die "Test"-Thematik an und geht danach detaillierter auf einzelne Gebiete ein. In dem Buch wird so gezeigt, wie Module und Programme getestet werden können. Außerdem wird gezeigt, wie Tests für Webseiten und Datenbanken aussehen.

Unit-Tests sind hauptsächlich aus dem Java-Bereich bekannt. In diesem Buch wird auch beschrieben wie Unit-Tests mit Perl aussehen. Im Abschließenden Kapitel zeigen die Autoren, wie Programme und Module getestet werden können, die nicht in Perl geschrieben sind.

Das Buch ist sehr gut dazu geeignet, beim Lesen direkt am Computer zu sitzen, da die Beispiele sehr einfach sind. Das lädt dazu ein, mit den Beispielen zu spielen.

Zusammenfassend kann man sagen, dass das Buch nicht nur für "Test"-Einsteiger geeignet ist, sondern auch schon für erfahrenere Tester.

Bei mir persönlich hat das Buch zu einem neuen Test-Verhalten geführt: Ich schreibe jetzt noch mehr Tests und teste auch Teile von Applikationen, die ich vorher nicht getestet habe.

Ich kann das Buch also nur empfehlen.

-- Renée Bäcker

Es läuft und läuft und...

läuft und läuft. - Das war der Werbespruch von VW für den Käfer. Aber das gleiche könnte man auch über Perl sagen.
Stattdessen findet man im Internet einige Leute, die sagen, Perl würde sterben. Dies zeigt vor allem eins: Perl hat eine schlechtere Außendarstellung als viele andere Sprachen.

Die Perl-Gemeinschaft ist ziemlich aktiv - allerdings agiert sie sehr häufig nur im Inneren mit sich selbst und trägt kaum etwas nach außen. Andere Sprachen werden dagegen sehr "aggressiv" gefördert - dort wird wegen jeder kleinen Neuerung oder jedem kleinen neuen Programm ein Artikel in einer Zeitschrift geschrieben.
Meiner Meinung nach braucht Perl auch mehr Außendarstellung um zu zeigen, dass Perl noch lange nicht tot ist und dass es mehr bietet als andere Sprachen. "Entscheidern" sollte gezeigt werden, dass Perl für viele Aufgaben sehr gut geeignet ist - häufig besser als die aktuell eingesetzten Sprachen.

Im amerikanischen Raum gibt es einige Dinge, die im Grunde einen gute Basis haben, aber auch nur innerhalb der Perl-Gemeinde bekannt sind. Oder kenn viele außerhalb der Perl-Szene den "Perlcast" oder die Zeitschrift "The Perl-Review"?

Doch wie kann mehr Aufmerksamkeit erreicht werden? - Meiner Meinung nach gibt es dazu mehrere Möglichkeiten, die ich in den nächsten Abschnitten erläutern will.

Perlmongers

Perlmongers haben - meiner Meinung nach - von Grund her eine andere Einstellung zu Perl als die meisten Anderen. Diese Gruppen haben auch die besten Möglichkeiten, etwas für die Bekanntheit von Perl zu tun. Dresden.pm ist hier ganz gut! Teilnahmen am Linux-Info-Tag in Dresden und auch andere Veranstaltungen werden besucht.
Durch Vorträge kann die Leistungsfähigkeit von Perl demonstriert werden. Ich würde mir von einigen anderen Perlmonger-Gruppen wünschen, dass zum Beispiel in Unis oder bei anderen IT-Stammtischen irgendwie auf die Perlmongers aufmerksam gemacht wird...

Zeitschriften

Michael Schillis Artikel im Linux-Magazin zeigen regelmäßig sehr schön die Möglichkeiten von Perl. Das ist schonmal ein Anfang. Doch auch in anderen Zeitschriften könnten solche Artikel stehen.
Auch diese Zeitschrift hat das Ziel, verschiedene Facetten von Perl zu zeigen und auch dem erfahrenen Perl-Programmierer mal was Neues zu zeigen.
Gerade die Zeitschriften mit großer Auflage sollten hin und wieder von Perl berichten - aber dafür braucht es auch Autoren. Hier liegt as jedem Einzelnen, mal einen Artikel zu schreiben und mit den Verlagen zu kommunizieren, ob sie nicht an so einem Artikel interessiert sind.

Net-/Podcast

Das Internet ist ein wichtiges Medium in der heutigen Zeit und in Zeiten der weißen Ohrstöpsel spielen sogenannte Net- oder Podcasts eine wichtige Rolle. Es gibt zwar das englisch-sprachige "Perlcast", aber viele Programmierer können oder möchten sich keine englisch-sprachigen Podcasts anhören.
Jochen Lillich hat das Projekt "Radio Perl" gestartet. Die ersten Folgen seines Netcasts sind online. Er ist an Themenvorschlägen interessiert. Dieses Projekt ist absolut unterstützenswert.

Veranstaltungen

Der Deutsche Perl-Workshop möchte relativ klein bleiben - auch um den Workshop-Charakter zu erhalten.
Trotzdem ist es unerlässlich, dass über den

Workshop "berichtet" wird. Sei es im eigenen Blog, in einer Zeitschrift, in Foren, ... es gibt hunderte von Möglichkeiten, wo so ein Bericht angebracht ist.
Davon haben alle etwas: Man selbst zeigt, dass man an der Sprache interessiert ist und auf dem Laufenden bleibt, der Workshop indem es mehr Interessierte und vielleicht auch weiterhin viele gute Vorträge gibt und Perl weil es seine Lebendigkeit beweisen kann.

Die meisten Veranstaltungen zum Thema Perl bleiben "Untergrundveranstaltungen". Es mag viele geben, die sich für eine Teilnahme interessieren würden, aber von der Veranstaltung nichts erfahren. Deshalb sollten alle Veranstaltungen irgendwo angekündigt werden. Die Liste ist die selbe wie für die Berichte.

Wichtig im Zusammenhang mit Veranstaltungen sind Vorträge! Nicht nur die reinen Perl-Veranstaltungen brauchen Vorträge, auch auf Konferenzen wie die EuroOSCON kann und soll über Perl gesprochen werden.

Firmenintern

Die einfachste "Werbung", die jeder für Perl machen kann, ist sein eigenes Arbeitsumfeld von den Fähigkeiten von Perl zu überzeugen. Zeigen, wie schnell und effektiv mit Perl programmiert werden kann. Natürlich sollte man auch immer im Auge haben, was im Arbeitsumfeld gebraucht wird. Es macht keinen Sinn, mit Perl Echtzeitsysteme oder 3D-Grafiksachen programmieren zu wollen. Aber es gibt viele Felder, für die Perl geeignet ist - und das kann man zeigen.

Es gibt viel was man für Perl tun kann. Eine große Entwicklergemeinde kommt auch Perl zu Gute. Es geht mir nicht darum, dass jeder Perl programmieren "muss" oder die ganzen "Skript-Kiddies" zu Perl zu holen. Mir geht es darum, dass die Wahrnehmung von Perl korrigiert werden sollte. Perl ist sicherlich nicht für alle Aufgaben geeignet - aber für sehr viele.

Links

http://www.linux-magazin.de/Artikel/Perl
http://foo-magazin.de
http://www.perlmongers.de
http://www.it-dojo.de
http://www.perlcast.com

Es läuft und läuft und...

Hier werden 6 Module vorgestellt, die neu auf CPAN sind oder bei denen neue Versionen online sind.
Es sind nur ganz kurze Vorstellungen und sollen zum Ausprobieren anregen.

WWW::Babelfish

Mit `WWW::Babelfish` kann man die Dienste von Babelfish, Google oder Yahoo! in Anspruch nehmen, um sich Texte übersetzen zu lassen. Das eignet sich gut, wenn man on-the-fly einfache Texte übersetzen lassen will.

```
my $babel = WWW::Babelfish->new();
print $babel->translate(
    text        => 'Hallo Welt!',
    source      => 'German',
    destination => 'English');
```

Das wird zwar nicht so schnell einen Übersetzer ersetzen, aber das Modul ist für kleinere Sachen ganz gut zu gebrauchen. Für überlange Texte - die in der Web-Version von Babelfish abgeschnitten werden - hat das Modul eine Lösung parat - es zerlegt den Text in mehrere Teile...

Es sind viele Sprachkombinationen möglich, nicht nur Englisch-Deutsch und Deutsch-Englisch.

LaTeX::Pod

`LaTeX::Pod` ist zum Beispiel für dieses Magazin sehr nützlich: Wenn Artikel im LaTeX-Format ankommen, wird die Datei mit dem Modul ins POD-Format übersetzt und dann mit einem Perl-Programm halbautomatisiert in einen Artikel für dieses Magazin übersetzt.

```
#!/usr/bin/perl

use strict;
use warnings;
use LaTeX::Pod;

my $file = 'test.tex';
my $obj  = LaTeX::Pod->new($file);
print $obj->convert();
```

Das Modul kann noch nicht mit allen Anweisungen etwas anfangen und bei Umlauten hat es Probleme, aber es ist noch jung und entwickelt sich prächtig.

For::Else

Ein Modul für die kleinen Dinge des Lebens... Wer hat die Situation nicht schonmal erlebt: Ich möchte etwas in einer Schleife abarbeiten und wenn das Array leer ist, muss ich eine andere Aktion machen. Bisher sah das immer so (ungefähr) aus:

```
for(@array){
    print;
}

if(!@array){
    print "Array ist leer!";
}
```

Mit `For::Else` kann man das in `if-else`-Manier lösen:

```
use For::Else;

my @array = ();
for(@array){
    print;
}
else{
    print "Array ist leer!";
}
```

Regexp::Assemble

Dieses Module ist ganz brauchbar, wenn man Reguläre Ausdrücke zusammenfassen will. Allerdings können nur Reguläre Ausdrücke, die mit "ODER" verknüpft sind, damit zusammenfassen.

Soll zum Beispiel überprüft werden, ob ein String mindestens eines der folgenden Pattern matcht:

- `Dies`
- `ist`
- `ein`
- `Test`

kann das Modul sehr nützlich sein.

```
use Regexp::Assemble;

my @regexes = qw(Dies ist ein Test);
my $ra = Regexp::Assemble->new();
$ra->add($_) for @regexes;
print $ra->re;
```

Heraus kommt ein RegEx, der alles vereint:

```
(?-xism:(?:(?:Te|i)st|Dies|ein)).
```

Die letzten beiden Module verraten einiges über das System...

http://search.cpan.org

Sys::Statistics::Linux

Das Modul sammelt die Informationen aus den Dateien, die unter `/proc` liegen. Zu den Informationen, die mit diesem Modul ausgelesen werden können, zählen, Angaben zu der Prozessorauslastung, laufende Prozesse, Speicherstatistiken und mehr.
Man kann einzelne Bereiche ein- und ausschalten. Dies hat Geschwindigkeits-vorteile und bei manchen Bereichen muss man ein `sleep` einfügen - weil Differenzen berechnet werden - und bei manchen Bereichen entfällt die.

Ein Beispiel:

```
#!/usr/bin/perl

use strict;
use warnings;
use Data::Dumper;
use Sys::Statistics::Linux;

my $lxs = Sys::Statistics::  \
    Linux->new(Processes => 1);
sleep 1;
my $stat = $lxs->find('Processes',
    cmd => qr/\(su\)/);
print Dumper($stat);
```

Damit werden alle Prozesse gefiltert, bei denen im Wert von `cmd` ein `(su)` vorkommt.

Win32::SystemInfo

`Win32::SystemInfo` ermöglicht es, sehr leicht auf Prozessor-Informationen und Informationen über den Speicher zuzugreifen.

```
use Win32::SystemInfo;
use Data::Dumper;

my %phash;
Win32::SystemInfo::ProcessorInfo \
   (%phash);
print Dumper(\%phash);
```

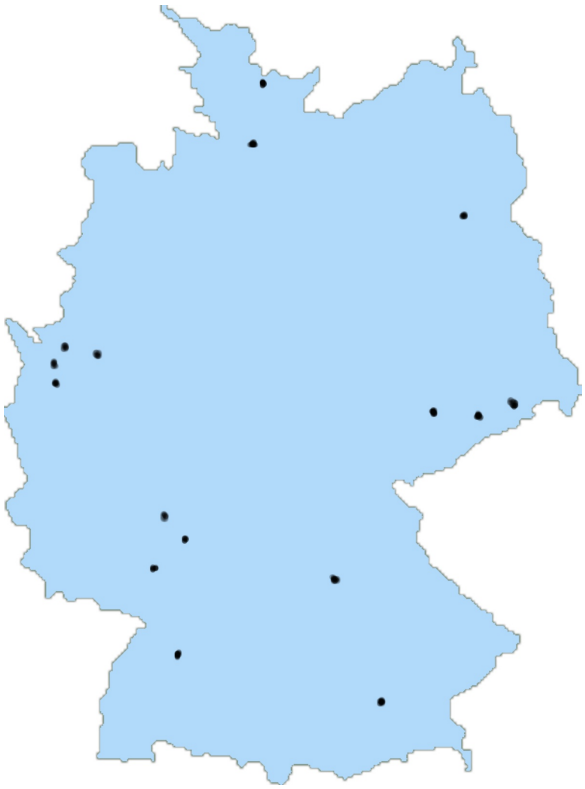
So bekommt man Informationen über die MHz-Anzahl, den Hersteller und den Indentifier für jeden einzelnen Prozessor.

Perlmongers

Vielleicht haben Sie schon den "Perlmongers" gehört, aber wissen nicht so genau, was das eigentlich sein soll?
Dieser Artikel soll einen Einblick in die Gruppe der Perlmongers geben, wie sie entstanden ist, was die Ziele sind und welche Vorteile man daraus ziehen kann.

Entstehung

brian d foy hat 1997 auf der 1. Perl-Konferenz die User-Gruppe New York (NY.pm) gegründet. Die Endung .pm bezieht sich dabei auf die Endung von Perl-Modulen. Der Name "Perlmonger" wird als Backronym für diese Endung verwendet.
Kurz nach der öffentlichen Bekanntmachung dass NY.pm gegründet wurde, hat Chris Nandor in Boston die zweite Perl-User-Grupper gegründet. Auf der zweiten Perl-Konferenz 1998 wurden etliche neue Perlmonger-Gruppen gegründet.
Seit 2000 sind die Perlmongers ein Teil der "Perl-Foundation".
2005 hat Dave Cross die Aktivität der Perlmonger-Gruppen "überprüft". Im Laufe der Zeit wurden einige Gruppen gegründet, die aber nach ein paar Treffen nicht mehr aktiv waren oder die nie wirklich aktiv gewesen sind. Nach der "Bereinigung" der Liste sind mittlerweile 178 aktive Gruppen über den Globus verteilt.



* Berlin.pm
* Bielefeld.pm
* Chemnitz.pm
* Cologne.pm
* Darmstadt.pm
* Dresden.pm
* Erlangen.pm
* Frankfurt.pm
* Hamburg.pm
* Kaiserslautern.pm
* Kiel.pm
* Munich.pm
* Niederrhein.pm
* Parderborn.pm
* Ruhr.pm
* Stuttgart.pm
* Naumburg

Ziele der Perlmonger

Das Hauptziel der Perl-Monger-Gruppen ist der Kontakt zu anderen Perl-Interessierten. Das geht vom Hobby-Programmierer über den Berufs-programmierer bis hin zu den "Cracks" von Perl. Ein wichtiger Aspekt der Treffen ist der Austausch. Man kann Probleme mit Perl ansprechen oder einfach Erfahrungen austauschen. Bei den sogenannten "Social Meetings" sitzen die Gruppen häufig zusammen und reden über Perl und "Gott und die Welt".
Viele Gruppen haben auch sogenannte "Tech-Meetings" bei denen Vorträge gezeigt werden.
Die "Organisation" Perl-Monger stellt auch eine gewisse Infrastruktur für den Betrieb einer Gruppe zur Verfügung. So werden Webspace, Mailinglisten und noch mehr bereitgestellt.

Vorträge

Neben Vorträgen von eigenen Mitgliedern, ist es möglich, dass mal ein "bekanntes Gesicht" der Perlszene auftaucht und einen Vortrag hält. So hat Damian Conway im September 2006 bei Porto.pm einen Vortrag gehalten. Auf der Webseite der Perlmongers

(http://www.pm.org) haben sich einige Personen eingetragen, die bereit sind, einen Vortrag auf einem Treffen zu halten (wenn es sich ergibt).

Vorteile als Perlmonger

Ein großer Vorteil ist natürlich, dass man einige nette Leute kennenlernt, mit denen man sich gut unterhalten kann. Die Gastfreundschaft von Perlmongers zeigt sich auch, wenn man mal in eine fremde Stadt kommt. Da ist schnell mal ein zusätzliches Perlmonger-Treffen angesetzt, oder es bietet sich jemand an, der die Stadt zeigen kann oder Kontakte zu Unterkünften bieten kann. Es ist immer gut wenn man ein Netzwerk an Bekannten aufbaut. Es soll sogar Perlmonger geben, die reisende Perlmonger für die eine oder andere Nacht beherbergen. Wo findet man noch solche Gastfreundschaft?!?

Perlmongers in Deutschland

Gemessen an der Zahl der Perlmonger-Gruppen, ist Deutschland nach den USA die zweitgrößte Perl-Gemeinde der Welt. Mittlerweile gibt es 20 Gruppen in Deutschland. Einige davon sind sehr klein, aber andere sind relativ groß und sehr aktiv.
Wo es Gruppen in Deutschland gibt und wann diese Gruppen sich treffen, erfährt man unter http://www.perlmongers.de.
Es lohnt sich, einfach mal vorbeizuschauen. Da es nur eine lose Gruppierung von Perl-Programmierern ist, gibt es auch so etwas wie "Mitgliedsbeitrag" oder ähnliches nicht.

Darmstadt.pm

Darmstadt.pm ist noch eine sehr kleine Runde. Im März 2006 wurde die Perlmonger-Gruppe in Südhessen von Ronnie Neumann gegründet. Seitdem findet jeden dritten Donnerstag im Monat ein "Social Meeting" im Nachrichtentreff statt. Zur Zeit finden sich immer nur zwei Personen ein, aber das wird sich (hoffentlich) bald ändern.
Einige Treffen finden zusammen mit IT-Stammtisch Darmstadt statt - immer dann wenn interessante Vorträge anstehen.
Darmstadt.pm freut sich über jeden Interessierten, der an den Treffen oder an der Mailingliste teilnimmt.

Darmstadt

Darmstadt ist eine Wissenschaftsstadt in der Mitte von Europa. Darmstadt ist die größte Stadt in Südhessen und liegt verkehrstechnisch ideal: Die Autobahnen A67 und A5 kreuzen sich bei Darmstadt, der Frankfurter Flughafen ist in knapp 30 Minuten mit dem Auto zu erreichen und sowohl S-Bahnen nach Frankfurt als auch der ICE hält in Darmstadt.
In Darmstadt sind bekannte Firmen wie die ESOC - das Kontrollzentrum der ESA -, T-Systems, Merck und viele andere angesiedelt. Die große Dichte an Wissenschaftlern wird durch die Technische Universität, die Fachhochschule und die Evangelische Hochschule geschaffen.
Auch vier Institute der Fraunhofer Gesellschaft befinden sich in Darmstadt.

Nachrichtentreff

Das Nachrichtentreff ist eine Bar und ein Restaurant in der Innenstadt von Darmstadt. Unweit von großen Einkaufsmöglichkeiten und Kinos ist das Nachrichtentreff angesiedelt. Hier gibt es gut bürgerliche Küche zum Essen und allerlei zu Trinken.

Links

http://darmstadt.perlmongers.de
Online-Heimat von Darmstadt.pm
http://mailingliste...
Mailingliste von Darmstadt.pm
http://www.darmstadt.de
Seite der Stadt Darmstadt
http://www.it-stammtisch-darmstadt.de
IT-Stammtisch Darmstadt

Perl-Community.de

Bis August 2003 war die größte deutsche Perl-Gemeinde unter http://www.perl.de zu finden, aber dann hatte der Inhaber der Domain keine Lust mehr und verkaufte die Domain. Martin Fabiani hat dann unter der Domain Perl-Community.de ein neues Forum installiert und die meisten Mitglieder von Perl.de kamen zu Perl-Community.de. Mit der Zeit hat sich die Community immer weiter vergrößert und ist mittlerweile zu der größten deutschsprachigen Perl-Gemeinde geworden.
Neben dem Forum gibt es noch ein Wiki, eine Linkliste und auch Treffen in der realen Welt.

Forum

Das Forum von Perl-Community.de wird sehr rege benutzt und subjektiven Ein-schätzungen nach, ist jede Perl-bezogene Frage im Schnitt nach 30 Minuten (oder schneller) beantwortet.

Wiki

Das Wiki von Perl-Community.de ist eine große Wissenssammlung in Sachen "Perl". Hier werden allgemeine Fragen be-antwortet wie "Wie installiert man ein Modul?" oder "Was bedeutet ein 500er Fehler bei CGI-Skripten?".

Das Wiki ist in vier große Bereiche geteilt:

User

Die Nutzer des Wikis stellen sich auf hier vor - ein Übersicht über alle Nutzer.

Wissensbasis

Unter der Wissensbasis sind die "Frequently Asked Questions" (FAQ) und einige Skript-Beispiele zu finden. Hier sammeln sich die Antworten auf viele Fragen.

Perldoc

Ein paar Mitglieder von Perl-Community.de haben angefangen, die Perl-Dokumentation zu übersetzen. Bisher sind es nur einige Dokumente, aber die wichtigsten sind mit dabei. Es sind aber nicht nur die übersetzten Perldocs im Wiki, sondern auch die englischen Fassungen der übrigen Dokumente.

Community

Hier finden sich die Plaunungen zu den Treffen und die Berichte von Workshops und Treffen wieder. Auch Verweise zu den deutschen Perlmonger-Gruppen sind hier zu finden.

Treffen

Wie eingangs schon erwähnt, treffen sich die Mitglieder von Perl-Community.de auch offline. Neben den Teilnahmen am Deutschen Perl-Workshop oder anderen Perl-Konferenzen, gibt es jährlich min-destens ein Treffen. Früher hieß es "Freak-treffen" und mittlerweile ist es zum "Frank-furter Perl-Community Workshop" um-benannt worden.
Treffen im realen Leben ermöglichen es, den Gegenüber aus dem Internet auch tatsächlich mal zu sehen. Das erhöht die Achtung vor dem Anderen und sorgt für eine angenehme Atmosphäre.

Poard

Da die Forensoftware auf Perl-Community.de so einige Eigenarten hatte und sehr unübersichtlich programmiert ist, haben sich die Mitglieder von Perl-Community.de entschlossen, eine eigene Forensoftware zu schreiben.
Das Ganze begann schon 2004 und da es nur eine "Nebenbeschäftigung" ist, zieht sich die Entwicklung etwas länger hin. Eine Entwicklungsversion kann unter http://develop.perl-community.de betrachtet werden.

Links

http://www.perl-community.de
Startseite Perl-Community.de

TPF-Grants

Die Perl-Foundation (TPF) vergibt in jedem Jahr sogenannte Grants für Perl-Programmierer. Mit diesen Grants versucht die Organisation, führende Perl-Entwickler zu unterstützen und damit auch die Sprache "Perl" weiter zu verbreiten. Mit einem "Grant" ist meistens auch die Aufforderung verbunden, auf einigen Konferenzen über die Arbeiten zu berichten, die unterstützt werden.

Was ist ein "Grant"?

Ein Grant ist ein Zuschuss für Arbeiten, die mit Perl zu tun haben und der Perl-Gemeinde zu Gute kommt. Dieser Zuschuss fällt unterschiedlich hoch aus - je nach Bedeutung und Größe. Allerdings ist die Aussicht darauf, einen Zuschuss von über 5.000 - 6.000 US$ zu bekommen, eher gering.
Nur wirklich große und bekannte Projekte bekommen bis zu 80.000 US$.

Wie bekommt man den Grant?

Um so einen Zuschuss zu bekommen, muss man sich bei der Perl-Foundation bewerben. Das sogennante "Grant-Commitee", das über die Vergabe der Zuschüsse entscheidet, berät einmal im Quartal über die vorliegenden Bewerbungen. Viele Bewerbungen fallen schon bei der ersten Durchsicht wegen formaler Fehler durch.
Man sollte auf jeden Fall deutlich machen, welchen Nutzen von dem Projekt die Perl-Gemeinde hat und wie man versucht, das Ziel zu erreichen.

Wie sieht eine Bewerbung aus?

Genau ist das nicht zu sagen, aber es gibt einige Punkte, wegen denen Bewerbungen nicht akzeptiert werden. Das "Grant-Commitee" veröffentlicht abgelehnte Bewerbungen nicht, um die Antragsteller nicht bloßzustellen. Dennoch wurde eine Liste mit den häufigsten Fehlern erstellt.
Die Bewerbung kann im einfachen Text-Format abgegeben werden. Eine Dummy-Bewerbung mit den notwendigen Fragen ist auf der Homepage der Perl-Foundation einzusehen.

Beispiele für gewährte Zuschüsse

In der Vergangenheit hat die Perl-Foundation immer wieder Zuschüsse für Arbeiten gewährt, bei denen sie überzeugt war von dem Nutzen für die Perl-Gemeinde. Der erste Zuschuss wurde 2001 an Damian Conway vergeben, der damit viele Module neu- und weiterentwickelt hat. Auch für das Perl6-Design war der Zuschuss gedacht.
Adam Kennedy hat für die Entwicklung des Moduls `PPI` 5.000 US$ bekommen. Mit dem Zuschuss wurde das Modul bis zu Verwendbarkeit gefördert. Und Nicholas Clark hat 2006 einen Zuschuss
Neben diesen drei Beispielen findet man auf der Webseite der Perl-Foundation alle bisher vergebenen Zuschüsse.

Links

http://www.perlfoundation.org
Perl-Foundation
http://news.perlfoundation.org/2005/12/
Mehr Informationen über das "Grant-Commitee"

Termine

Termine

Hier werden alle Termine genannt, die mit Perl zu tun haben und uns bekannt sind. Natürlich können sich Termine und Uhrzeiten noch ändern, aber darauf haben wir keinen Einfluss.

Wenn Sie weitere Termine haben, die in diesem Kalender angezeigt werden sollen, dann schicken Sie bitte eine E-Mail an

termine@foo-magazin.de

Dabei sollte beachtet werden, dass zirka 2 Wochen vor Erscheinungstag, der Redaktionsschluss ist.

F e b r u a r

01. Treffen von *Dresden.pm*

06. Treffen *Frankfurt.pm*
Treffen *Stuttgart.pm*

12. Treffen *Ruhr.pm*

14. Treffen *Cologne.pm*
Treffen *Hamburg.pm*

15. Treffen *Darmstadt.pm*
Treffen *Erlangen.pm*

21. 9. Deutscher Perl-Workshop (München)

22. 9. Deutscher Perl-Workshop (München)

23. 9. Deutscher Perl-Workshop (München)

27. Treffen *Bielefeld.pm*

28. Treffen *Berlin.pm*

M ä r z

01. Treffen *Dresden.pm*

06. Treffen *Frankfurt.pm*
Treffen *Stuttgart.pm*

12. Treffen *Ruhr.pm*

14. Treffen *Cologne.pm*
Treffen *Hamburg.pm*

15. Treffen *Darmstadt.pm*
Treffen *Erlangen.pm*

21. Treffen *Munich.pm*

27. Treffen *Bielefeld.pm*

28. Treffen *Berlin.pm*

A p r i l

03. Treffen *Frankfurt.pm*
Treffen *Stuttgart.pm*

04. YAPC::Asia (Tokio)

05. Treffen *Dresden.pm*
YAPC::Asia (Tokio)

07. OSDC Taiwan

11. OSDC Taiwan

09. Treffen *Ruhr.pm*

11. Treffen *Cologne.pm*
Treffen *Hamburg.pm*

19. Treffen *Darmstadt.pm*
Treffen *Erlangen.pm*

24. Treffen *Bielefeld.pm*

25. Treffen *Berlin.pm*

Links

Links

http://www.Perl.org

Auf Perl.org kann man die aktuelle Perl-Version downloaden. Ein Verzeichnis mit allen möglichen Mailinglisten, die mit Perl oder einem Modul zu tun haben, ist dort zu finden. Auch Links zu anderen Perl-bezogenen Seiten sind vorhanden.

http://www.perl-foundation.org

Die Perl-Foundation nimmt eine stuernde Funktion in der Perl-Gemeinde ein: Es werden Zuwendungen für Leistungen zugunsten von Perl gegeben. So wird z.B. die Bezahlung der Perl6-Entwicklung über die Perl-Foundation geleistet. Jedes Jahr werden auch Studenten beim "Google Summer of Code" betreut.

http://www.perl-workshop.de

Der Deutsche Perl-Workshop hat sich zum Ziel gesetzt, den Austausch zwischen Perl-Programmierern zu fördern. Der 9. Deutsche Perl-Workshop findet vom 21.-23. Februar an der Fachhochschule München statt.

http://www.perl-community.de

Das ist eine der größten deutschsprachigen Perl-Foren. Hier ist aber nicht nur ein Forum zu finden, sondern auch ein Wiki, mit vielen Tipps und Tricks. Einige Teile der Perl-Dokumentation wurden ins Deutsche übersetzt. In der Linkliste von Perl-Community.de finden sich viele Verweise auf nützliche Seiten.

http://www.pm.org

Das Online-Zuhause der Perlmongers. Hier findet man eine Übersicht mit allen Perlmonger-Gruppen weltweit. Hier kann man auch eine neue Gruppe gründen und bekommt Hilfe...

perl -e 'for(qw/36 102 111 111
32 45 32 80 101 114 108 45 77
97 103 97 122 105 110/)
{print chr}'

Smart-Websolutions

Windolph und Bäcker GbR

Perl-Programmierung

info@smart-websolutions.de