Ausgabe 4/2009 – www.foo-magazin.de - 6,00 €

$foo

PERL MAGAZIN

DPath ist nicht XPath!
Suche in komplexen Datenstrukturen

Methoden-Signaturen
Methoden-Deklarationen mit Signaturen und ohne Source-Filter

WxPerl-Tutorial
GUI-Entwicklung mit Perl

Nr 12

Integralis ist in Europa führender Anbieter von IT-Sicherheitslösungen mit einem Netzwerk von 19 Niederlassungen in 9 Ländern weltweit. Mit rund 500 Mitarbeitern erzielten wir im Jahr 2008 einen Umsatz von 168,9 Mio. EUR. Wir unterstützen namhafte, auch international tätige Unternehmen bei der Planung, der Umsetzung und dem Betrieb von IT-Sicherheitsmaßnahmen. Integralis ist ein Tochterunternehmen der börsennotierten Integralis AG **www.integralis.com**

Zur Verstärkung unseres Entwickler-Teams mit Standort in **Ismaning** bei München suchen wir ab sofort auf längerfristiger freiberuflicher Basis eine/n

Softwareentwickler/in (Perl, Linux) für Managed Security Services

Ihre Aufgaben:

- Mitarbeit im Bereich der Integralis Managed Security Services zur Überwachung, Logfileanalyse und Fernwartung von Sicherheitssystemen in Kundennetzwerken
- Softwareentwicklung in Perl, Shell, C++ und PHP für unsere Security Service Appliance unter Linux
- Weiterentwicklung unserer Agent-Software für Linux, Solaris und Windows
- Ausbau der Infrastruktur unserer Security Management Center
- Implementierung von Services für bestehende und neue Sicherheitsprodukte aus den Bereichen Perimeter Security, Content Security, Threat and Vulnerability Management, Identity and Access Management
- 3rd Level Support für unsere Security Management Center weltweit
- Entwicklung und Durchführung von QA-Testverfahren sowie Dokumentation
- Eigene Projektverantwortung nach erfolgreicher Einarbeitung

Ihr Profil:

- Informatikstudium oder vergleichbare Ausbildung
- Erfahrung im Bereich Softwareentwicklung, insbesondere Skript-Programmierung (gute Kenntnisse in Shell, Perl)
- Fundierte Kenntnisse im Bereich TCP/IP-Netzwerke, idealerweise mit Security-Erfahrung
- Erfahrung in der Systemverwaltung von Unix/Linux-Systemen (idealerweise Red Hat)
- Oracle, Remedy und weitere Programmiersprachen wären von Vorteil
- Eigenverantwortliche und systematische Arbeitsweise
- Sehr gute Englischkenntnisse in Wort und Schrift
- Ausgeprägte Kommunikationsfähigkeit

Ihre Perspektiven:

- Mitarbeit in einem innovativen und abwechslungsreichen Umfeld mit einer sehr positiven Arbeitsatmosphäre in einem kleinen, stabilen Team
- Möglichkeit von Auslandsaufenthalten in Großbritannien und den USA
- Flexible Arbeitszeiten

Sind Sie interessiert? Dann freuen wir uns auf die Zusendung Ihrer vollständigen Bewerbungsunterlagen unter Angabe des nächstmöglichen Eintrittstermins und Ihren Gehaltsvorstellungen.

Integralis Services GmbH
c/o Integralis Deutschland GmbH
Birgit Fritz/Maria Zwierzyna
Human Resources
Ferdinand-Braun-Str. 1
D-74074 Heilbronn

Tel.: 0049/7131/799-117 oder -101
Fax: 0049/7131/799-120
personalabteilung.hn@integralis.com
www.integralis.com

Von Perl und Pumpkings

Es muss immer einen geben, der für die Perl-Releases verantwortlich ist. Das sind die sogenannten "Pumpkings". Der Name Pumpking kommt von einem Kürbis (engl. Pumpkin), der als "Token" gedient hat. Ohne diesen Kürbis durfte man keine Patches einspielen.

Jede Perl-Reihe bekommt einen eigenen Pumpking. Für Perl 5.8.x war es Nicholas Clark, für Perl 5.10.0 war Rafael Garcia-Suarez und für Perl 5.10.1 Dave Mitchell zuständig. Im Juli dieses Jahres hat Rafael den Job als Pumpking aufgegeben. Seitdem wird nach einem Pumpking gesucht.

Anfang Oktober wurde dann Perl 5.11.0 veröffentlicht. Das hatte einige Änderungen im ganzen Release-Prozess zur Folge: Zum einen hat Jesse Vincent einen neuen Release-Zyklus begonnen und zum anderen gibt es für die unterschiedlichen Releases immer einen neuen Pumpking. In Zukunft wird an jedem 20. eines Monats ein neues Perl-Release erstellt.

Als Pumpking sollte man sich einigermaßen mit Perl und auch etwas mit C auskennen. Aber sonst kann sich eigentlich jeder für diesen Job melden. Bis Anfang 2010 sind die Pumpkings festgelegt, aber auch danach wird es ja noch Perl-Releases geben. Jesse Vincent wird den Pumpking-Neulingen auch mit Ratschlägen zur Seite stehen, wenn es um die Release-Erstellung geht.

Die ganzen Vorgänge zeigen doch eins:

Perl: Alive and Kicking and Stronger than ever!

Aber jetzt wünsche ich viel Spaß beim Lesen der 12. Ausgabe von $foo.

Renée Bäcker

Die Codebeispiele können mit dem Code

95m?2j

von der Webseite www.foo-magazin.de heruntergeladen werden!

The use of the camel image in association with the Perl language is a trademark of O'Reilly & Associates, Inc. Used with permission.

Alle weiterführenden Links werden auf del.icio.us gesammelt. Für diese Ausgabe: http://del.icio.us/foo_magazin/issue12.

IMPRESSUM

Herausgeber:	Smart Websolutions Windolph und Bäcker GbR Maria-Montessori-Str. 13 D-64584 Biebesheim
Redaktion:	Renée Bäcker, Katrin Bäcker, André Windolph
Anzeigen:	Katrin Bäcker
Layout:	//SEIBERT/MEDIA
Auflage:	500 Exemplare
Druck:	powerdruck Druck- & VerlagsgesmbH Wienerstraße 116 A-2483 Ebreichsdorf
ISSN Print:	1864-7537
ISSN Online:	1864-7545

INHALTSVERZEICHNIS

ALLGEMEINES

6 Über die Autoren
8 Methoden-Signaturen
58 Rezension - Automating System Administration with Perl

MODULE

12 Modern Art des Profilens
17 Elchtest (Moose)
21 Perl und XML
29 DPath ist nicht XPath!
37 Neue Syntax dank Devel::Declare
41 WxPerl Tutorial - Teil 1

PERL

46 Perl-Scopes Tutorial - Teil 2

TIPPS & TRICKS

56 HTML::Display

NEWS

55 Win32 Tipps & Tricks
60 "Merkwürdigkeiten"
61 CPAN News
63 Neues von TPF
65 Termine

66 LINKS

Hier werden kurz die Autoren vorgestellt, die zu dieser Ausgabe beigetragen haben.

Renée Bäcker

Seit 2002 begeisterter Perl-Programmierer und seit 2003 selbständig. Auf der Suche nach einem Perl-Magazin ist er nicht fündig geworden und hat so diese Zeitschrift herausgebracht. In der Perl-Community ist Renée recht aktiv – als Moderator bei Perl-Community.de, als Organisator des kleinen Frankfurt Perl-Community Workshops und und als Grant-Manager bei der Perl Foundation.

Ferry Bolhár-Nordenkampf

Ferry kennt Perl seit 1994, als sich sein Dienstgeber, der Wiener Magistrat, näher mit Internet-Technologien auseinanderzusetzen begann und er in die Tiefen der CGI-Programmierung eintauchte. Seither verwendet er – neben clientseitigem Javascript – Perl für so ziemlich alles, was es zu programmieren gibt; auf C weicht er nur mehr aus, wenn es gar nicht anders geht – und dann häufig auch nur, um XS-Module für Perl schreiben. Wenn er nicht gerade in Perl-Sourcen herumstöbert, schwingt er das gerne das Tanzbein oder den Tennisschläger.

Herbert Breunung

Ein perlbegeisteter Programmierer aus dem ruhigen Osten, der eine Zeit lang auch Computervisualistik studiert hat, aber auch schon vorher ganz passabel programmieren konnte. Er ist vor allem geistigem Wissen, den schönen Künsten, sowie elektronischer und handgemachter Tanzmusik zugetan. Seit einigen Jahren schreibt er an Kephra, einem Texteditor in Perl. Er war auch am Aufbau der Wikipedia-Kategorie: "Programmiersprache Perl" beteiligt, versucht aber derzeit eher ein umfassendes Perl 6-Tutorial in diesem Stil zu schaffen.

Thomas Fahle

Thomas Fahle, Perl-Programmierer und Sysadmin seit 1996.
Websites:

- http://www.thomas-fahle.de
- http://Perl-Suchmaschine.de
- http://thomas-fahle.blogspot.com
- http://Perl-HowTo.de

Rolf Schaufelberger

Rolf Schaufelberger ist geschäftsführender Gesellschafter der plusW GmbH. Seit Mitte der 90er programmiert er mit Perl und seit seiner Selbständigkeit 2001 ist er ganz auf Perl spezialisiert. Seine Firma entwickelt Webportale für die Druckindustrie und betreibt selbst die Website www.mypixler.com. Alle Websites basieren auf einem selbst entwickelten Framework auf der Basis Mason, DBIx::Class und neuerdings auch Moose. Daneben ist er Mitglied der Deutschen Perl-Workshop Gb R und bei Stuttgart.pm und ist dort zur Zeit mit den Planungen zum Perl-Workshop 2010 beschäftigt.

Steffen Schwigon

Steffen Schwigon wurde 1972 geboren und hat kurz danach, ok, 1987, angefangen zu programmieren. Atari Basic, Pascal, C, C++, Java, Perl - in der Reihenfolge. Debian GNU/Linux ist sein Betriebssystem, X Emacs sein Editor. In Perl programmiert er seit 2002. Er ist aktives Mitglied der Dresden Perl Mongers und bei den Perl-Workshop-Orgas.

Methoden-Signaturen

Java-Programmierer kennen sie von Anfang an und auch immer Perl-Programmierer wollen sie - die Signaturen bei Methoden. Aber was sind Signaturen?

Was sind Signaturen

Die Signatur einer Methode besteht aus Name der Methode und Parameter. In anderen Sprachen - wie z.B. Java - gehören auch Sichtbarkeit und Rückgabetype zur Signatur. Diese zwei Sachen gibt es bei den Signaturen in Perl aber nicht. Die Signatur einer Methode muss dabei eindeutig sein.

So sehen z.B. Signaturen in Java aus

```
public void methodenname ( int number ){
    // Code
}
```

Ein großer Unterschied zwischen der Java- und der Perl-Implementierung von Signaturen ist, dass bei Java mehrere Methoden den gleichen Namen haben können, die sich dann aber durch die Parameter unterscheiden:

```
public void gruss ( String Name ){
    // Code
}

public void gruss (
     String Grussformel, String Name ){
    // Code
}
```

java weiß dann durch die Anzahl und Art der Parameter, welche Methode gemeint ist. In Perl geht das nicht. Mit den Modulen, die ich weiter unten vorstelle, dürfen die Methodennamen nur einmal vorkommen. Die beiden gezeigten Java-Methoden müssten also in einer Perl-Methode dargestellt werden, was man durch "optionale" Parameter in der Signatur erreichen kann:

```
method gruss ( $name, $grussformel? ){
    # Code
}
```

Im Gegensatz zu Java und anderen Sprachen werden beim Perl-Code die übergebenen Parameter nicht validiert. Das kann man aber mit anderen Modulen erreichen (siehe Abschnitt "MooseX::Method::Signatures").

Michael G Schwern hat bei den Perl 5 Porters die Diskussion begonnen, ob man nicht ab Perl 5.12 (für das es schon erste Vorbereitungen gibt aber noch einige Zeit bis zur Veröffentlichung vergehen wird) Signaturen "automatisch" unterstützen soll, in dem man entsprechende Module in den Kern packt und mit einem `use 5.012` einbindet. Aber die Entscheidung ist noch nicht gefallen.

In Perl 6 gibt es aber auf jeden Fall Signaturen und folgende Module versuchen möglichst nah (soweit es Perl 5 zulässt) an der Spezifikation zu bleiben.

Module für Signaturen

Auf CPAN gibt es mehrere Module, die die Verwendung von Signaturen ermöglichen. Bei älteren Modulen ist Vorsicht geboten, weil diese meist als sogenannten Source-Filter implementiert sind. Solche Source-Filter sind zwar in manchen Fällen sehr praktisch, aber häufig sind sie Fehlerbehaftet. Im Falle von Source-Filtern gilt noch "Only perl can parse Perl". Mehr zu Source-Filtern wird es in der nächsten Ausgabe von $foo geben.

In den folgenden Abschnitten möchte ich zwei Module vorstellen, die nicht als Source-Filter umgesetzt sind, sondern den den Op-Tree von Perl eingreifen, bevor das Perl-Programm ausgeführt wird. Wie das genau gemacht wird, wird ebenfalls in einem extra Artikel in einer der nächsten Ausgaben von $foo erläutert.

Die Module, die hier vorgestellt werden, sind `Method::Signatures` und `MooseX::Method::Signatures`. Es gibt noch das einfachere `signatures` von Florian Ragwitz, das bietet aber ohne große eigene Anstrengungen nur eine sehr einfache Signaturen-Implementierung. Die beiden vorgenannten Module bieten dagegen viel mehr Möglichkeiten.

Method::Signatures

Wenn man `Method::Signatures` in seinen Code einbindet, steht das Schlüsselwort `method` zur Verfügung. `method`en sind normale Subroutinen, bei denen `$self` automatisch aus `@_` geholt wird. Dieses Modul eignet sich daher hervorragend für Objektorientierten Code. Das hier ist somit gültiges Perl:

```
package Foo;

use Method::Signatures;

method gruss {
    print "Objekt: $self";
}
```

Parameter werden - ähnlich wie Prototypen - in runden Klammern hinter dem Methodennamen angegeben:

```
method gruss ( $name ){
    print "Hallo $name";
}
```

Dieses Beispiel ist äquivalent zu

```
sub gruss {
    my $self = shift;
    my ($name) = @_;
    print "Hallo $name";
}
```

Allerdings bleibt bei der Umsetzung der Signaturen die Original-Zeilennummerierung erhalten, so dass ein Fehler auch wirklich an der Stelle zu suchen ist, die Perl angibt.

Schon an diesem kleinen Beispiel sieht man, dass Signaturen Schreibarbeit sparen und die Übersichtlichkeit erhöhen.

Wie das Beispiel schon andeutet, stehen in `@_` alle Parameter - nur der erste Parameter wird automatisch aus dem Array geholt. Der Rest ist genauso wie bei "herkömmlichen" Subroutinen.

Das Modul ist zwar noch als "Alpha" in der Dokumentation gekennzeichnet, aber da es immer mehr im Einsatz ist und die Module dahinter ständig verbessert werden, wird sich an der API voraussichtlich nicht mehr viel ändern.

Was das Modul so alles kann:

Das Modul kennt positionale und benannte Parameter. In Perl wird das mit einem Array bzw. mit einem Hash abgebildet. Das obige Beispiel zeigt positionale Parameter im Einsatz. Bei der Methode `gruss` landet der erste Parameter immer in `$name`. Bei benannten Parametern kann die Reihenfolge geändert werden, hier kommt es nur auf den Namen an. Ein Beispiel ist in Listing 1 zu sehen.

```
use strict;
use warnings;

{
  package Magazin;
  use Method::Signatures;

  method new {
      return bless {}, $self;
  }

  method magazin( :$name, :$genre ){
      print "Name: $name -> Genre: $genre\n";
    }
}

Magazin->new->magazin(
  name  => '$foo',
  genre => 'IT',
);

Magazin->new->magazin(
  genre => 'feuerwehr',
  name  => '112',
);
```

Listing 1

Zu beachten ist, dass der Bezeichner der Variablen als Hash-Schlüssel genommen wird. Mit `Method::Signatures` kann man diese beiden Arten von Parametern auch mischen. Allerdings müssen die benannten Parameter zum Schluss kommen - sowohl beim Aufruf der Methode als auch bei der Methodendefinition (Listing 2).

```
use strict;
use warnings;

{
    package MethodTest;

    use Method::Signatures;

    method new { bless {}, $self }
    method test($name, :$opt){
        print $name, " -> ", $opt;
    }
}

# alles ok
MethodTest->new->test(
  'hallo',
  opt => 1,
);

# wirft Fehler
MethodTest->new->test(
  opt => 1,
  'hallo',
);
```

Listing 2

Wie im Eingangstext bereits erwähnt, gibt es auch optionale Parameter, die mit einem '?' nach dem Variablennamen gekennzeichnet sind. Optionale Parameter können **nicht** mit benannten Parametern gemischt werden. Also der folgende Code ist **nicht** möglich:

```
method gruss( $grussformel?, :$name )
{
  ...
}
```

Ein weiterer wichtiger Punkt ist, dass man mit dem Modul auch Standard-Werte setzen kann. Damit wird die lästige Abfrage im Code, ob der Paramater gesetzt ist, überflüssig. Listing 3 zeigt ein Beispiel und verdeutlicht auch, wie man auch bei Parametern mit Standard-Wert ein `undef` hinbekommt.

```
use strict;
use warnings;

{
    package DefaultValue;
    use Method::Signatures;

    method new{ bless{}, $self }
    method test( $name = 'hallo'){
        print $name,"\n" if defined $name;
    }
}

DefaultValue->new->test;
DefaultValue->new->test( 'zweiter Aufruf' );
DefaultValue->new->test( undef );
```

Listing 3

Es gibt noch einige weitere Sachen, die man mit `Method::Signatures` machen kann, die ich hier aber nicht weiter ausführen werde. Das waren die wichtigsten Punkte, die einem das Leben stark vereinfachen können.

MooseX::Method::Signatures

Wer Perl "modern" programmieren will, benutzt in der Objektorientierung das Modul `Moose` (siehe auch den Artikel "Elchtest" in dieser Ausgabe). Auch für `Moose` gibt es Erweiterungen, mit der sich Methodensignaturen einsetzen lassen.

Die meisten Features sind sehr ähnlich zu `Method::Signatures`, allerdings hat dieses Modul noch ein paar Vorteile: Man kann den Typ der Parameter angeben und man kann mehr mit "Constraints" machen. Gerade die Typüberprüfung finde ich sehr angenehm, da durch diese Möglichkeit viel Code wegfällt. In dem Code müsste man selbst erst überprüfen, ob überhaupt ein Objekt vorliegt und ob das Objekt vom richtigen Typ ist.

Im Listing 4 ist ein ganz einfaches Beispiel zu sehen, wie eine Methode aussieht, die als Parameter ein Objekt vom Typ "Magazin" erwartet

```
use Moose;
use MooseX::Method::Signatures;

method test ( Magazin $foo ) {
    print "Ausgabe: ", $foo->ausgabe, "\n",
          "Seiten: ",  $foo->seiten, "\n";
}
```

Listing 4

Das sieht doch viel besser aus als der Code, der in Listing 5 zu sehen ist.

```
use Scalar::Util qw( blessed );

sub test {
    my ($self, $foo) = @_;

    unless ( blessed $foo and $foo->isa(
                                'Magazin' ) )
    {
        die '$foo is not of type "Magazin"';
    }

    print "Ausgabe: ", $foo->ausgabe, "\n",
          "Seiten: ",  $foo->seiten, "\n";
}
```

Listing 5

Man kann auch mehrere Typen aufzählen, die ein Objekt sein darf. Darf das Objekt entweder vom Typ "Magazin" oder vom Typ "Buch" sein, dann sieht die Methoden-Signatur folgendermaßen aus:

```
method text ( Magazin|Buch $foo ) { ... }
```

Ein weiteres wichtiges Feature sind anderweitige Einschränkungen für Parameter. Darf ein Parameter nur eine Zahl zwischen 0 und 255 sein - z.B. weil sie für RGB-Werte in Bildern benötigt wird -, dann sieht es wie folgt aus:

```
use Moose;
use MooseX::Method::Signatures;

method red ( $r where {
    $_ >= 0 and $_ <= 255
} ) { ... }
```

Der Block hinter dem `where` muss gültigen Perl-Code enthalten.

Ein weiteres nützliches Feature sind Labels. Wie im Teil über `Method::Signatures` beschrieben, gibt es auch bei dem Moose-Modul die benannten Parameter. Normalerweise ist es so, dass der Name des Parameters (z.B. `$foo`) als Schlüsselname genommen wird (z.B. foo). Der Parameter wird also mit

```
test( foo => 'wert' );
```

übergeben. Es kann aber vorkommen, dass man einen anderen Schlüsselnamen verwenden will, weil dieser unter Umständen "sprechender" ist. Und möchte man im Code dann aber einen einfachen Variablennamen benutzen, kommen die Labels ins Spiel.

```
method test( :schluessel($foo) ) { ... }
```

sorgt dafür, dass der Parameter dann so übergeben wird:

```
test( schluessel => 'wert' );
```

Natürlich bietet auch `MooseX::Method::Signatures` die Möglichkeiten mit, positionalen und benannten Parametern, optionalen und notwendigen Parametern sowie Standardwerten. Da die Syntax aber identisch zu `Method::Signatures` ist, habe ich hier auf Beispiele verzichtet.

Das etwas bekanntere Modul `MooseX::Declare` verwendet `MooseX::Method::Signatures` und bietet deklarative Programmierung für `Moose` an. Auf das Modul werde ich an dieser Stelle jedoch nicht eingehen.

Fazit

Ein großer Vorteil bei der Verwendung von Signaturen ist, dass der Code wartbarer wird, da weniger Code geschrieben werden muss. Und nicht geschriebener Code ist fehlerfreier Code.

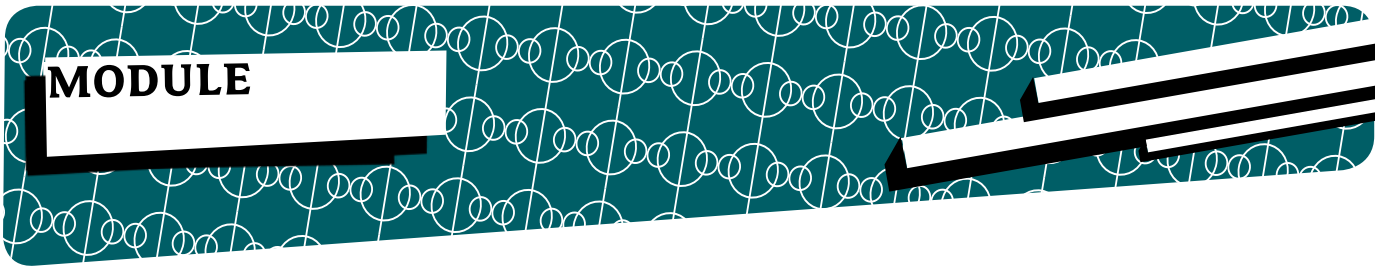
Zusätzlich kann man sehr einfach Default-Werte für Variablen angeben, was das Leben in vielen Fällen einfacher macht.

Einen Geschwindigkeitsnachteil gibt es so gut wie gar nicht, da die Module nicht mit Source-Filtern arbeiten. Michael G Schwern hat für sein Modul `Method::Signatures` einen Geschwindigkeitsverlust von 1% festgestellt.

Bei den Modulen gibt es meistens ein paar kleinere Einschränkungen, die sich aber auf den "Durchschnittscode" nicht auswirken. Welche Einschränkungen es gibt, ist jeweils auch in der Dokumentation der Module ausführlich dargelegt.

Man sollte auch bedenken, dass in Perl die Überprüfung der Parameter erst zur Laufzeit möglich ist - und nicht wie bei Java schon zur Compile-Zeit.

Renée Bäcker



Modern Art des Profilens

In Ausgabe 3 von $foo hat Mike Schilli schon einen kurzen Artikel über das Profilen von Perl-Code geschrieben. Damals wurden die Module `Devel::DProf` und `Devel::SmallProf` kurz angerissen. Ich möchte in dieser Ausgabe ein anderes Modul vorstellen und tiefer in die Thematik einsteigen. Das Modul heißt `Devel::NYTProf` und wurde bei der "New York Times" entwickelt. Mittlerweile hat Tim Bunce (der DBI-Autor) das Modul übernommen.

Da `Devel::NYTProf` in C geschrieben ist, ist das Modul sehr schnell.

Arten des Profilens

Das Profiling in Perl kann auf drei Arten gemacht werden: Befehlsorientiert, Unterprogrammorientiert und bestimmte Anwendungsbereiche. Die bisherigen Profiler konnten immer nur eins.

Befehlsorientiert

Beim Befehlsorientierten profilen wird die Zeit gemessen, die zwischen der Ausführung des einen Befehls und dem nächsten Befehl vergeht. Sobald ein neuer Befehl erreicht wird, wird die Zeit berechnet, die seit dem Beginn des vorherigen Befehls vergangen ist. Diese wird dann der Code-Zeile zugeordnet, in der der vorherige Befehl begann.

Standardmäßig ermittelt der befehlsorientierte Profiler die erste Zeile des aktuellen Blocks und die erste Zeile des aktuellen Befehls und summiert deren Zeiten. `Devel::NYTProf` ist der einzige Profiler für Perl, der auch auf Block-Basis profilen kann.

Ein weiteres Problem von vielen Befehlsorientierten Profilern wird in `NYTProf` gelöst: Wenn ein Statement eine Subroutine aufruft und dann etwas anderes ausführt, das keinen neuen Befehl ausführt, z.B.

```
funktion() + mkdir('./verzeichnis');
```

wird die Zeit, die für das Erstellen des Verzeichnisses benötigt wird, auf den letzten Befehl von `funktion` addiert. Das kann die Zeiten des letzten Befehls von `funktion` enorm verändern. Dauert es in diesem Beispiel lange, bis das Verzeichnis erstellt ist, erkennt man das nicht bei den meisten Befehlsorientierten Profilern. Stattdessen wundert man sich, warum der letzte Befehl in `funktion` so viel Zeit benötigt und eventuell fängt man dann an der falschen Stelle mit dem Optimieren an.

`Devel::NYTProf` umgeht dieses Problem, in dem es die opcodes abfängt - die anzeigen, dass die Kontrolle an ein anderes Statement ging - und das Profiling dementsprechend anpasst.

Subroutinen-orientiert

Profiler, die Subroutinen-orientiert arbeiten, messen die Zeit zwischen dem "Betreten" und dem "Verlassen" der Subroutine. Dabei wird der Zähler für Aufrufe inkrementiert und die Zeit addiert. Für jede aufgerufene Subroutine wird der Zähler und die Dauer extra gespeichert - und das für jede Stelle, an der die Subroutine aufgerufen wird.

Das "Betreten" der Subroutine wird durch das Abfangen des `entersub` opcodes erkannt; das Verlassen über perl's eigenen Stack. Das Ergebnis ist schnell und sehr robust.

Allerdings gibt es bei den meisten Subroutinen-orientierten Profilern ein Problem: Man kann sie durch Sprungbefehle wie `goto` oder `next` durcheinander bringen.

Anwendungsbereich

Im großen Gegensatz zu den zwei zuvor genannten Arten steht das Profiling eines bestimmten Anwendungsbereichs. Damit ist zum Beispiel die Untersuchung nur der Datenbank-Abfragen gemeint. In Ausgabe 10 von $foo wurde im Artikel "111% DBIx::Class" gezeigt, wie man Datenbankabfragen mit `DBIx::Class` profilen kann. Hier wird dann nicht der Perl-Code betrachtet, sondern nur die Datenbankabfragen an sich.

Das kann dann ganz sinnvoll sein, wenn man bei der Untersuchung des Perl-Codes festgestellt hat, dass eine bestimmte Subroutine sehr viel Zeit benötigt. Wenn diese Subroutine Datenbank-Abfragen beinhaltet, kann man diese dann untersuchen um festzustellen, ob die Datenbank der Flaschenhals ist.

`Devel::NYTProf` kann sowohl zeilenorientiert als auch unterprogrammorientiert profilen, was neben anderen Features ein großer Pluspunkt ist. So muss man seine Programme nicht doppelt profilen.

CPU-Zeit vs. Real Time

Beim Schreiben eines Profilers stellt sich auch die Frage, ob der Zeitaufwand in CPU-Zeit oder in Realer Zeit gemessen werden soll. Das kann unter Umständen sehr unterschiedliche Ergebnisse hervorbringen. Der Vorteil der Messung in CPU-Zeit ist, dass die Auslastung des Rechners quasi keine Rolle spielt. Aber die Zeiten werden maximal in 0,01-Sekunden-Schritten gemessen. Die Messung in realer Zeit hat da eine viel bessere Auflösung (0,000001 Sekunden). Zusätzlich wird beim Profilen mit realer Zeit auch die non-CPU-Wartezeit (z.B. warten auf die Datenbank oder Zugriffe auf die Festplatte) mit ausgegeben.

Ein Programm profilen

Aber wie untersucht man jetzt das Programm? Die Einbindung von `Devel::NYTProf` ist einfach. Genau wie bei anderen `Devel::`-Modulen kann man beim Aufruf von Perl mit dem `-d`-Parameter das Modul einbinden:

```
perl -d:NYTProf anwendung.pl
```

Alternativ kann auch die Umgebungsvariable `PERL5DB` gesetzt werden:

```
PERL5DB='use Devel::NYTProf'
```

Das Beispielprogramm für diesen Artikel ist in Listing 1 dargestellt:

```
#!/usr/bin/perl

use strict;
use warnings;

use B ();

test();

B::walkoptree( B::main_root(), 'debug' );

sub test {
    my $a = 3;
}

sub UNIVERSAL::debug {
    print $_[0]->name,"\n";
}
```

Listing 1

Ausgabeformate

Die Profiling-Ergebnisse können von Haus aus bei der Auswertung in zwei Formate umgewandelt werden: CSV oder HTML. Die CSV-Daten können dann z.B. leicht in eine Datenbank importiert werden, was sinnvoll sein kann, wenn man die Performanz einer Anwendung über einen längeren Zeitraum beobachten will.

Für die Auswertung der Daten gibt es entsprechend den Formaten die Programme `nytprofcsv` und `nytprofhtml`, die mit dem Modul mitgeliefert werden.

Die Profiling-Daten werden mit folgendem Befehl in CSV-Dateien umgewandelt:

```
nytprofcsv -f nytprof.out
```

Damit wird ein Verzeichnis `nytprof` erzeugt und darin die CSV-Dateien abgelegt - für jedes Modul/Skript drei Dateien: `<Modul>-line.csv`, `<Modul>-block.csv` und `<Modul>-sub.csv`.

Für den normalen Gebrauch ist das HTML-Format sehr gut geeignet. Ähnlich wie bei der Generierung von CSV-Dateien werden mit `nytprofhtml` HTML-Dateien erstellt. Zusätzlich noch ein paar CSS- und JavaScript-Dateien. In Abbildung 1 ist die Startseite der generierten HTML-Dateien zu sehen.

Diese Startseite ist eine Zusammenfassung der Ergebnisse. In der oberen Tabelle sind die Subroutinen aufgeführt, die am meisten Zeit verbrauchen. Darunter sind die "langsamsten" Dateien aufgelistet. Die Werte sind durch Farbe hinterlegt, so dass man schon auf den ersten Blick die "kritischen" Daten erfassen kann. Die "Grenzen" für die einzelnen Farben werden für jeden Programmlauf basierend auf den jeweiligen statistischen Daten neu ermittelt. Man kann also nicht sagen, dass ein "200ms-Befehl" immer rot ist. In einem allgemein langsamen Skript kann das durchaus zu den schnellsten Befehlen gehören. Dann wird die Zeit grün hinterlegt.
Von dieser Startseite aus, kann man sich durch die Profiling-Ergebnisse navigieren. Insgesamt kann man sagen, dass die Dateien untereinander sehr gut verlinkt sind.

Wie bei den CSV-Dateien gibt es auch bei der HTML-Auswertung verschiedene Dateien pro Modul/Skript. Bei jeder Ansicht (line/block/sub) gibt es ganz oben eine kurze Zusammenfassung, um welche Datei es sich handelt, wie viele Statements in dieser Datei ausgeführt wurden und wie viel Zeit insgesamt darauf verwendet wurde (siehe Abbildung 2).

Unter der Übersicht, gibt es eine Liste der Subroutinen mit den dazugehörigen Profiling-Daten. Diese Spalten gibt es dabei:

• Calls : Wie oft wurde diese Subroutine aufgerufen

• P: Von wievielen Stellen im Code wurde diese Subroutine aufgerufen

• F : Aus wievielen Dateien heraus wurde diese Subroutine aufgerufen

• Exclusive Time: Zeitverbrauch der Subroutine, ohne die Subroutinen, die darin aufgerufen wurden

• Inclusive Time: Zeitverbrauch der Subroutine, inklusive der Subroutinen, die darin aufgerufen wurden

In der Tabelle mit dem Quellcode werden dann noch Angaben darüber gemacht, wieviele Statements in dieser Zeile abgearbeitet werden und wieviel Zeit dafür verbraucht wurde. Im Quellcode sind auch die Kommentare sehr nützlich, da dort noch erweiterte Informationen zu finden sind. Z.B. Angaben darüber, wieviel Zeit für eine gewisse Anzahl von Aufrufen einer anderen Subroutine verbraucht wurde. Der Kommentar sieht dann beispielsweise so aus:

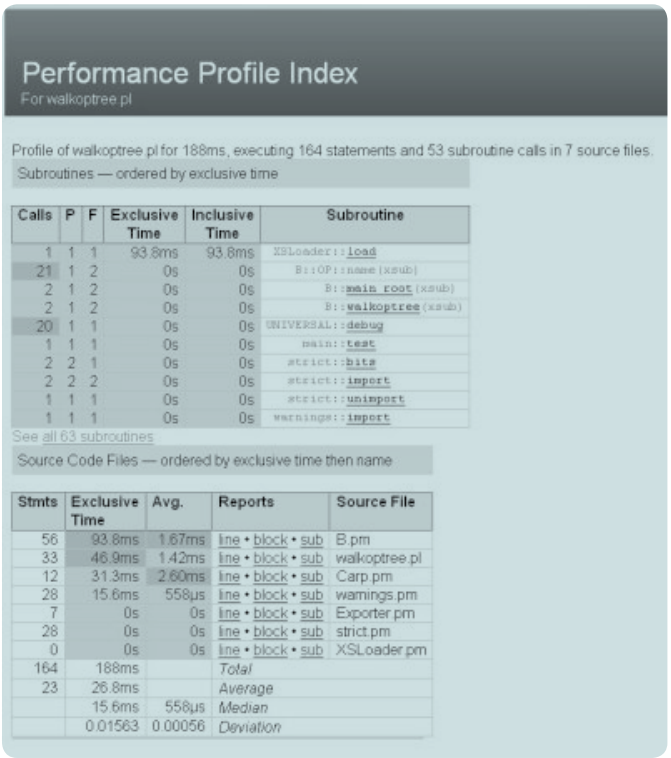


Abbildung 1

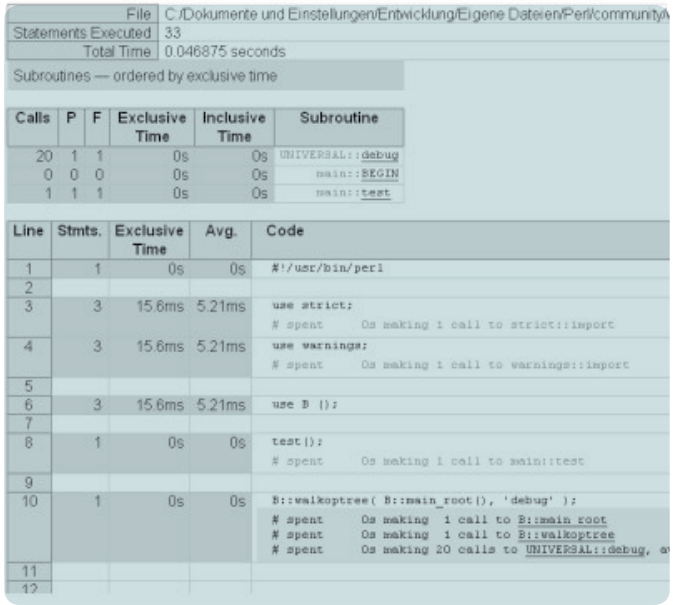


Abbildung 2

```
# spent 0s making 20 calls to
   UNIVERSAL::debug, avg 0s/call
```

Die Subroutinen-Namen sind auch verlinkt, so dass man gleich zu der anderen Stelle springen kann. Umgekehrt gibt es auch Links, zu den Stellen, von denen aus die Subroutine aufgerufen wurde.

Ein Vorteil von `Devel::NYTProf` ist auch, dass man zwischen den Verschiedenen Darstellungen (line/block/sub) switchen kann. So kann man sich immer die gewünschten Daten anschauen. Nicht wundern: Die Zeiten für eine Subroutine werden immer bei der ersten Zeile der Subroutine zusammengefasst und bei der Block-Auswertung bei der ersten Zeile des Blocks (siehe Abbildung 3).

Devel::NYTProf und Webanwendungen

Das bisher gezeigte Beispiel behandelt ein einfaches Skript, eine Webanwendung sollte bei Perfomance-Problemen ebenfalls untersucht werden. Aus diesem Grund kann man `Devel::NYTProf` auch in den Apachen einbinden. Damit kann man die Zeitfresser bei allen möglichen Nutzer-Interaktionen herausfinden. Wer jetzt noch eine große Testsuite aus z.B. Selenium-Tests hat, kann jetzt unkompliziert die Anwendung profilen.

Man muss nur noch das Modul einbinden. Das geht über die Shebang:

```
#/usr/bin/perl -d:NYTProf
```

Beim Aufruf der Anwendung wird jetzt das Modul geladen und das Profiling wird wie oben beschrieben durchgeführt.

Aber auch mit mod_perl-Anwendungen kommt das Modul zurecht. Zuvor muss das Modul aber noch in den Apachen eingebunden werden. Für die Interaktion mit Apache gibt es das Modul `Devel::NYTProf::Apache`, das mit der Apache-Direktive `PerlModule` geladen wird:

```
PerlModule Devel::NYTProf::Apache
```

Damit wird bei jedem Request die aufgerufene Anwendung untersucht. Die Daten landen dann in `/tmp/nytprof.$$.out`. `$$` steht dabei für die ID des Prozesses.

`Devel::NYTProf` erkennt auch, wann die Kontrolle von perl an mod_perl übergeht (wenn der Request abgearbeitet wurde). Dadurch wird verhindert, dass der zuletzt ausgeführte Befehl die Zeit von mod_perl (z.B. Wartezeit auf den nächsten Request) angerechnet bekommt. Das würde sonst die Ergebnisse verfälschen.

Catalyst

Möchte man Catalyst-Anwendungen untersuchen, kann man so vorgehen:

				sub find {			
365	39004	0.14204	4e-06	my $self = shift;	365	2744927	9.28967
366				my $wanted = $self->_	366		
				# spent 0.37123s maki			
367					367		
368				# Use a queue based s	368		
369				my @found = ();	369		
370				my @queue = $self->ch	370		
				# spent 0.04848s maki			
371	5572	0.01181	2e-06	eval {	371		
372	2539326	7.23647	3e-06	while (my $Element =	372		
373	1	0.84905	0.84905	my $rv = &$wanted($s	373		

Abbildung 3

```
$ perl -d:NYTProf script/yourapp.pl
$ ab -n 10000 http://localhost:3000/path/to/
                    slow/page
$ GET http://localhost:3000/quit
$ nytprofhtml
```

Der dritte Schritt ist notwendig, damit `Devel::NYTProf` sauber beendet wird. `quit` sollte dabei so aussehen:

```
sub quit :Global { exit(0) }
```

Devel::NYTProf und ge'fork'te Anwendungen

`Devel::NYTProf` kann auch Anwendungen profilen, die mit `fork` arbeiten. Dabei leidet die auch nicht Performanz darunter. Das Modul erkennt den fork und beginnt eine neue Datei mit den Profiling-Daten. Die Prozess-ID wird dabei an den Dateinamen angehängt.

Renée Bäcker

„Eine Investition in Wissen bringt noch immer die besten Zinsen.“

(Benjamin Franklin, 1706-1790)

Aktuelle Schulungsthemen für Software-Entwickler sind: AJAX - interaktives Web * Apache * C * Grails * Groovy * Java agile Entwicklung * Java Programmierung * Java Web App Security * JavaScript * LAMP * OSGi * Perl * PHP – Sicherheit * PHP5 * Python * R - statistische Analysen * Ruby Programmierung * Shell Programmierung * SQL * Struts * Tomcat * UML/Objektorientierung * XML. Daneben gibt es die vielen Schulungen für Administratoren, für die wir seit 10 Jahren bekannt sind.

Siehe linuxhotel.de

Elchtest

Ein gebrochenes Schlüsselbein bescherte mir über Ostern genügend freie Zeit, um Moose einem persönlichen Elchtest zu unterziehen. Nach dem Artikel hier in $foo und etlicher weiterer Lektüre wollte ich Moose einmal an einer konkreten Anwendung ausprobieren. Dazu wählte ich ein von mir entwickeltes Programm, welches aus Kundenbestellungen eines Webshops PDF-Dokumente erstellt, hauptsächlich Kalender, aber auch Postkarten, Poster sowie Rechnungs- und Lieferscheinformulare. Besonders bei den Kalendern sorgen die verschiedenen Ausführungsvarianten und Kundenanforderungen, welche immer unter Zeitdruck implementiert werden mussten, im Laufe der Zeit zu einer recht unübersichtlichen Ansammlung von Klassen, bei denen trotz Mehrfachvererbung das Auftauchen gleicher Codesegmente in verschiedenen Klassen nicht verhindert und die gewünschte Flexibilität nicht erreicht wurde. Schon länger geisterte die Vorstellung einer Art Baukasten in meinem Kopf herum, welcher es mir erlaubt, eine Seite eines Dokumentes aus einzelnen Elementen zusammenzusetzen. Je nach Art des PDF, welches erstellt werden muss, sind bestimmte Elemente (z.B. ein Kalenderbild) immer vorhanden, andere jedoch nur bei bestimmten Kundenoptionen (z.B. Logos oder Zusatztexte). Ich hatte mir schon verschiedene Lösungsansätze mit Plugins überlegt, aber keiner konnte mich so richtig begeistern. Bis Moose mit seinen Rollen die Bühne betrat.

Moose und Rollen wurden im zuvor erwähnten Artikel schon beschrieben, deshalb hier nur eine kurze Einführung. Eine Rolle in Moose erweitert das Prinzip von Interface Klassen wie sie z.B. Java kennt, indem eine Rolle die Implementierung der Funktion enthält. Eine Klasse kann dann diese Rolle dann verwenden (in Moose Termininologie: A class does a role) - siehe Listing 1.

Anstatt also eine Unterklasse zu erstellen, welche das Loggingverhalten von einer Logging Klasse erbt, wird einfach das Verhalten per Rolle hinzugefügt. Jede Klasse, die die Rolle `MyApp::Logger` "verwendet", kann die Methode `log` wie eine eigene Methode aufrufen.

Zurück zu meinen Kalendern. Am Anfang meiner Überlegungen stand die Frage, welches Verhalten in Rollen, und welches direkt als Methode in verschiedenen Unterklassen implementiert werden sollte. Da Rollen ein sehr neues Konzept darstellen, gibt es wenig Literatur die einem hierbei Tipps oder Erfahrungswerte liefern kann. Im konkreten Fall kamen noch die Besonderheiten der PDF Generierung dazu. Wir verwenden hierzu die kommerzielle PDFlib, das Grundprinzip gilt sicher aber ähnlich für andere PDF-Bibliotheken wie z.B. PDF::API2. Ein PDF-Dokument wird erstellt, indem Seite für Seite die Inhalte geschrieben werden. Man kann nicht in mehrere Seiten gleichzeitig schreiben, jede Seite wird für sich vollständig erzeugt und dann geschlossen. Man kann aber entweder vorhandene PDF Dokumente einlesen oder virtuelle Dokumente erzeugen und deren Seiten in die gerade offene PDF-Seite einfügen. Die Reihenfolge, in welcher Inhalte in eine Seite eingefügt werden, spielt meist keine Rolle, es sei denn, dass z.B. ein Text über ein Bild geschrieben werden soll. Auf Basis dieser Überlegungen entstand ein erster Entwurf einer Klassenhierarchie:

```
package MyApp::Logger;
use Moose::Role;

sub log {
    my $self = shift;
    print "Logger ", shift,"\n";
}

1;

package MyApp::User;
use Moose;

with ( 'MyApp::Logger');

sub  foo {
      my $self = shift;
      $self->log(' User::foo' );
}

1;
```

Listing 1

Eine Klasse `Document` mit Unterklassen für die verschiedenen Dokumenttypen (Kalender, Postkarten, ...) und eine Klasse `Page` für die eigentlichen Seiten, ebenfalls mit Unterklassen, da eine Kalenderseite doch etwas anderes darstellt, als eine Postkartenseite. Die einzelnen Inhalte (Bilder, Logos, Texte, Kalendarium etc) sollen dann über Rollen eingefügt werden. Wie oben gesehen werden Rollen mit `with` eingebunden, das würde aber bedeuten, dass ich entweder für jede mögliche Kombination an Merkmalen eine Subklasse benötige, oder jede Rolle selbst prüfen muss, ob sie überhaupt zur Abwendung kommt oder nicht. Beide Szenarien sind weder praktikabel noch zufrieden stellend. Glücklicherweise gibt es die Möglichkeit, einer Objektinstanz Rollen dynamisch hinzuzufügen.

Das Beispiel in Listing 2 zeigt, wie der obigen Klasse User eine Rolle `PerlProgrammer` mit der Methode `what_is_hot` hinzugefügt wird.

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

use strict;
use MyApp::User;
use MyApp::PerlProgrammer;

my $user = MyApp::User->new();
$user->foo();

# Rolle zu Objekt-Instanz hinzufuegen
MyApp::PerlProgrammer->meta->apply($user);
$user->what_is_hot();
```

Listing 2

Damit war das Problem, Funktionalität abhängig von den Bestelldaten hinzuzufügen, aber nur zum Teil gelöst. Die Idee war, eine Klasse für das jeweilige PDF-Dokument zu haben, welches sich das benötigte Seitenobjekt je nach Anforderung zusammenbaut, und dieses Objekt dann die Seite selbständig generiert.

Wenn dem Seiten-Objekt aber Rollen dynamisch zugewiesen werden, woher weiß die Seite dann, welche Methoden (der Rollen) es aufrufen muss? Auch dafür hat Moose eine Lösung parat, die so genannten Method-Modifier. Ein Method-Modifier kann man sich ähnlich wie einen Trigger in einer Datenbank vorstellen. Ein Trigger kann dort angeben, dass er "BEFORE" oder "AFTER" nach einem "INSERT", "UPDATE" etc aufgerufen wird. Genauso verhält es sich mit dem Method-Modifiern: Man hat hier neben "before" und "after" noch "around", und bei Klassen noch "augment" zur Auswahl. Die Syntax sieht dann folgendermaßen aus:

```
before 'methodenname' => sub {...};
```

Ein 'around' -modifier kann - im Gegensatz zu before und after - die original übergebenen Argumente manipulieren und auch den Aufruf der eigentlichen Funktion unterlassen.

Mit den Modifier before und after bot sich eine sehr elegante Möglichkeit, mein "PDF-Baukasten" zu erreichen. Die einzelnen Rollen brauchten so nur angeben, dass sie nach dem Öffnen oder vor dem Schließen einer Seite ausgeführt werden sollen. Die Klasse `Page` benötigt dazu nur die beiden Methoden `open` und `close`, diese werden von `buildPDF` aufgerufen. Die Inhalte der Seite kommen dann über die fest oder dynamisch zugewiesenen Rollen automatisch hinzu. Im konkreten Fall muss als erstes ein Bild eingefügt werde, dies erfolgt via `after 'open'`, die anderen Inhalte kommen dann mit einem `before 'close'` Modifier aus, da deren Ausführungsreihenfolge egal ist. In den nachfolgenden Beispielen geben die Klassen und Rollen jeweils nur ein `print` aus, das genügt aber um die Funktionsweise zu demonstrieren. In den folgenden Beispielen verwenden wir eine Klasse `Calendar`, ein Klasse `page` mit zwei Subklassen `Startpage` für das Deckblatt und `Calenderpage` für die 12 Monatsblätter. Da alle Kalenderseiten ein Bild besitzen, kann die Rolle `Image`, welche das Bild einfügt, mit der `with` Anweisung fest in die Klasse eingebunden werden, ebenso die Rolle `Text` in der Klasse `Startpage`. Damit erhalten wir folgenden Code:

Die beiden Rollen:

```
package PW::PDF::Roles::Page::Image;

use Moose::Role;

after 'open' => sub {
    print "\tImage\n";
};

1;

package PW::PDF::Roles::Page::Text;

use Moose::Role;

before 'close' => sub {
    print "\tText\n";
};

1;
```

Listing 3

Die Klasse für die Kalenderseiten:

```
package PW::PDF::Page::CalendarPage;

use Moose;
extends 'PW::PDF::Page';

__PACKAGE__->meta->make_immutable;

sub buildPDF {
    my $self = shift;
    my $pageno = shift;

    $self->open();
    print "\tCalendar Page $pageno\n";
    $self->close();
}

1;
```

Listing 4

und für das Deckblatt:

```
package PW::PDF::Page::StartPage;

use Moose;

extends 'PW::PDF::Page';
with 'PW::PDF::Roles::Page::Text';

 __PACKAGE__->meta->make_immutable;

sub buildPDF {
    my $self = shift;

     $self->open();
     print "\tStartpage\n";
     $self->close();
}

1;
```

Listing 5

und zuletzt die Kalenderklasse:

```
package PW::PDF::Document::Calendar;

use Moose;

extends 'PW::PDF';
with  'MooseX::Log::Log4perl' ;

use PW::PDF::Page::StartPage;
use PW::PDF::Page::CalendarPage;

sub buildPDF {
    my $self = shift;

    my $startpage =
PW::PDF::Page::StartPage->new();
    my $calpage =
PW::PDF::Page::CalendarPage->new();

    $startpage->buildPDF();
    $calpage->buildPDF($_) for (1..2);
}

1;
```

Listing 6

Und das Script dazu:

```
#!/usr/bin/perl -w
use lib 'src/lib1';

use strict;
use PW::PDF::Document::Calendar;

my $cal =
PW::PDF::Document::Calendar->new();
$cal->buildPDF;
```

Listing 7

ergibt als Ausgabe:

```
open Page
        Image
        Startpage
        Text
close Page
open Page
        Image
        Calendar Page 1
close Page
open Page
        Image
        Calendar Page 2
close Page
```

Listing 8

Das entspricht genau dem gewünschten Ergebnis. Jetzt nehmen wir an, dass in alle Seiten noch ein Logo eingefügt werden soll. Statt nun weitere Unterklassen von `StartPage` bzw. `CalendarPage` zu bilden und dort jeweils(!) zu prüfen, ob ein Logo vorhanden ist und wenn ja diesen einzufügen, erstellen wir einfach eine neue Rolle `Logo` und weisen diese Rolle den beiden Klasen dynamisch zu. Die Rolle Logo ist hier im Beispiel weitgehend identisch mit Text, die neue Kalenderklasse sieht nun so aus - siehe Listing 7.

```
package PW::PDF::Document::Calendar;

use Moose;

extends 'PW::PDF';
with  'MooseX::Log::Log4perl' ;

use PW::PDF::Page::StartPage;
use PW::PDF::Page::CalendarPage;

use PW::PDF::Roles::Page::Logo;

sub buildPDF {
    my $self = shift;

    my $startpage =
PW::PDF::Page::StartPage->new();
    my $calpage =
PW::PDF::Page::CalendarPage->new();


PW::PDF::Roles::Page::Logo->meta->apply
   ($startpage);
```

Listing 9 (Fortsetzung Seite 20)

```
PW::PDF::Roles::Page::Logo->meta->apply($cal
page);

    $startpage->buildPDF();
    $calpage->buildPDF($_) for (1..2);
}

1;
```

Listing 9 (Fortsetzung von Seite 19)

und das Ergebnis ist dann :

```
open Page
        Image
        Startpage
        Logo
        Text
close Page
open Page
        Image
        Calendar Page 1
        Logo
close Page
open Page
        Image
        Calendar Page 2
        Logo
close Page
```

Das dynamische Hinzufügen von Rollen zu einem Objekt fanden wohl auch andere als derart nützlich, dass es mit `MooseX::Object::Pluggable` ein Modul gibt, welches diese Funktionalität mit etwas mehr Komfort und Möglichkeiten bietet. Das Modul selbst ist nichts anderes als eine Rolle welche in die Klasse, die erweitert werden soll, eingebunden wird (hier `PW::PDF::Page`). Dann sucht es Rollen in einem Ordner `Plugin`. Verschieben wir also unsere Rollen dahin, binden das Modul mit `with 'MooseX::Object::Pluggable'` in unsere Page-Klasse ein und ändern unsere Kalenderklase wie folgt ab - siehe Listing 10.

```
package PW::PDF::Document::Calendar;

 use Moose;

 extends 'PW::PDF';
 with  'MooseX::Log::Log4perl';

 use PW::PDF::Page::StartPage;
 use PW::PDF::Page::CalendarPage;

 sub buildPDF {
     my $self = shift;

     my $startpage =
 PW::PDF::Page::StartPage->new();
     my $calpage   =
 PW::PDF::Page::CalendarPage->new();

     $startpage->load_plugin('Logo');
     $calpage->load_plugin('Logo');

     $startpage->buildPDF();
     $calpage->buildPDF($_) for (1..2);
 }

 1;
```

Listing 10

Und wir erhalten das gleiche Resultat wie zuvor. Man kann `load_plugin` entweder einen absoluten Pfad für die Rolle angeben, oder aber das Plugin suchen lassen, entweder mit den Default Suchpfaden oder mit einer Liste von angegebenen Suchpfaden. Soll die Rolle `Logo` z.B. für `CalenderPage` eine andere sein als normal genügt es, Logo.pm in `CalendarPage/Plugin/Logo.pm` zu implementieren und wir erhalten als Resultat (ohne Änderung in Calendar.pm bzw. CalendarPage.pm):

```
open Page
        Image
        Startpage
        Logo
        Text
close Page
open Page
        Image
        Calendar Page 1
        CalendarPage Logo
close Page
open Page
        Image
        Calendar Page 2
        CalendarPage Logo
close Page
```

Man versuche, das mal in "traditionellem' OO-Perl so einfach zu implementieren. Mein persönliches Fazit nach den ersten Versuchen ist durchweg positiv: Die Idee eines Baukastens konnte realisiert werden, das System ist inzwischen im produktiven Einsatz. Ein neues Projekt, bei welchem ich ebenfalls regen Gebrauch von Rollen und Method-Modifiern mache, ist in Arbeit. Dazu macht es Spaß - man entdeckt mit wachsendem Verständnis immer neue und weitere Möglichkeiten.

Rolf Schaufelberger

Perl und XML

"Viele Leute glauben XML könne sogar Kaffee kochen oder den Familienhund waschen"

- Zitat aus XML IN A NUTSHELL (O'Reilly)

Die "eXtensible Markup Language" (*XML*) ist eine Auszeichnungssprache, um Informationen in einer strukturierten Form darzustellen und wird sehr häufig beim Datenaustausch zwischen zwei Schnittstellen verwendet. Bekannte Anwendungen von XML sind im Internet zu finden: RSS-Feeds sind in XML dargestellt genauso wie die "Scalable Vector Graphics" (*SVG*).

Gibt es Einschränkungen im Format, werden diese in Schemas wie "Document Type Definition" (*DTD*) oder XML-Schema definiert und man kann ein XML-Dokument gegen diese Schemas validieren.

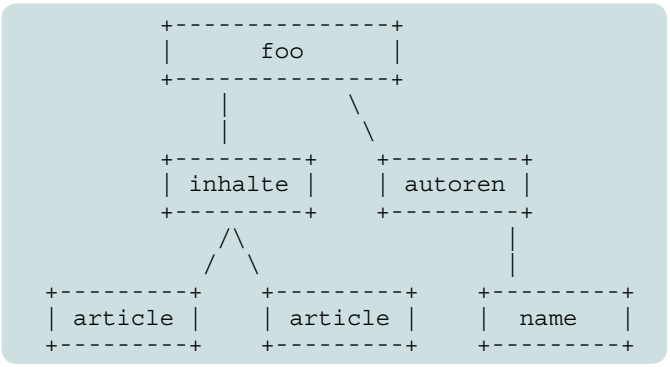
Dieser Artikel soll drei unterschiedliche Module zur XML-Verarbeitung (und ggf. Manipulation) zeigen und wie man XML-Dokumente gegen Schemas validiert. Weiterhin wird die Transformation mittels XSLT ein Thema sein.

Alle Beispiele in diesem Artikel werden mit dem folgenden XML-Dokument arbeiten, das sich jeweils im `DATA`-Bereich des Programms befindet:

```
<?xml version="1.0" encoding="UTF-8"
   standalone="yes"?>
<foo ausgabeId="12">
  <inhalte>
    <artikel>XML und Perl</artikel>
    <artikel>Tipps und Tricks</artikel>
  </inhalte>
  <autoren>
    <name>Renee Baecker</name>
  </autoren>
</foo>
```

Solche XML-Dokumente können beliebig komplex werden. Und je komplexer das XML werden kann, umso genauer sollte man sich überlegen, welches Modul man einsetzt.

Prinzipiell gibt es zwei unterschiedliche Methoden, wie ein XML-Dokument geparst werden kann: Erstens Event-basiert und zweitens durch Erstellen eines Baums. Ich werde in diesem Artikel nur auf Module eingehen, die einen Baum aus dem XML erzeugen. Für das obige Beispiel würde der Baum so aussehen:



Auf CPAN finden sich - wie bei einigen anderen Themen - nahezu unendlich viele Module für die Verarbeitung von XML-Dokumenten. Jedes hat seine Stärken und Schwächen. Die drei Module, die ich hier zeigen werde, dürften die am meisten benutzten Module in diesem Bereich sein.

Auf XPath gehe ich nicht näher ein, sondern zeige bei den einzelnen Modulen, wie man damit arbeiten kann - wenn die Module XPath unterstützen.

XML::Simple

Wie der Name schon andeutet, ist dieses Modul für den "einfachen" Gebrauch gedacht. Es gibt nicht viele Funktionen,

die man kennen muss: Mit `XMLin` wird das XML geparst und mit `XMLout` kann man XML-Dateien schreiben.

Ein kleines Programm, das die Titel der Artikel aus dem XML-Dokument ausgibt sieht so aus:

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

use strict;
use warnings;
use XML::Simple;

my $xml = do{ local $/; <DATA> };
my $tree = XMLin( $xml );

print $_ for
   @{ $tree->{inhalte}->{artikel}  };
```

Wie man sieht, ist die Verwendung ziemlich einfach. Und für einfache XML-Dokumente, bei denen man den Inhalt gut im Griff hat, ist das Modul auch gut geeignet. Das Modul hat aber den Nachteil, dass man bei komplexeren XLM-Daten mit Parametern spielen muss. Z.B. werden die Artikel in dem Beispiel-XML in einer Arrayreferenz gespeichert, die Autoren - da es nur einer ist - als String. Würde ich einen Artikel aus dem XML streichen, dann würde auch der verbliebene Artikel nicht in einer Arrayreferenz gespeichert werden, sondern als String - und der Code würde den Fehler `Can't use string ("XML und Perl") as an ARRAY ref while "strict refs" in use ...` ausgeben.

Man kann dann aber mit `XMLin( $xml, ForceArray => 1 )` die Arrayreferenzen erzwingen. Dann wird aber wirklich alles in Arrayreferenzen gespeichert. Das kann unter Umständen dazu führen, dass man viel Code ändern muss.

Bei dem Modul gibt es noch viele andere Parameter, mit denen man das Verhalten von `XML::Simple` beeinflussen kann. Gerade wenn man in der Hashreferenz etwas ändert und die Änderungen mit `XMLout` in eine Datei schreiben will, muss man viel testen bis man das gleiche Format wie die Eingabedatei hat. Für das Beispiel sähe das so aus:

```
: XMLout(
  $tree, XMLDecl => 1, RootName => 'foo' )
```

`XML::Simple` übergeht den Namen des Root-Elements, außer man schaltet das explizit mittels `KeepRoot => 1` ab. Dann braucht man bei `XMLout` auch den `RootName` nicht anzugeben. Allein diese Beispiele zeigen, dass man teilweise etwas mit den verschiedenen Einstellungen spielen muss.

Insgesamt ist das Modul trotzdem für einfache XML-Dokumente gut geeignet. Vor allem wenn es um das Parsen der Dokumente geht. Die Dokumentation ist auch recht ausführlich und mit einigen Beispielen gespickt, so dass man sich dort auch vieles anschauen kann.

XML::Twig

Ein weiteres wichtiges Modul für das XML-Parsing ist `XML::Twig` - mit dem man auch komplexe XML-Dokumente parsen kann. Ein sehr großer Vorteil dieses Moduls ist, dass man auch sehr große XML-Dokumente damit parsen kann, ohne das komplette Dokument in den Speicher zu laden. Man muss bei großen Dokumenten immer im Hinterkopf haben, dass sie im Speicher ein Vielfaches von dem Platz benötigen, den sie auf der Platte einnehmen.

Auch wenn das Beispiel für die Möglichkeiten von `XML::Twig` etwas unterdimensioniert ist, werde ich hier darstellen, wie mit dem Modul große Dateien Stück für Stück bearbeitet werden können.

Als erstes zeige ich jedoch, wie so ein kleines XML-Dokument als Baum in den Speicher geladen wird:

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

use strict;
use warnings;
use XML::Twig;

my $xml    = do{ local $/; <DATA> };
my $parser = XML::Twig->new;
my $tree   = $parser->parse( $xml );

my $root         = $tree->root;
my $authors_node = $root->first_child(
                              'autoren' );
my @authors      = $authors_node->children;

for my $author ( @authors ){
    print "Autor: ", $author->text
}
```

Man bekommt bei `XML::Twig` keine Hashreferenz wie bei `XML::Simple`, sondern ein Objekt, in dem alle möglichen Informationen zu dem Baum gespeichert sind. Dadurch ist es möglich, sich durch den Baum zu bewegen, z.B. mit den Methoden `first_child` und `parent`. Als erstes muss aber der root-Knoten geholt werden und von dort aus, kann man sich überall hinbewegen. Bei den meisten Methoden zum

Auffinden von anderen Knoten kann man optional eine Bedingung mit übergeben, wie das Beispiel mit `first_child( 'autoren' )` zeigt. Ohne diese Bedingung würde `first_child` den `inhalte`-Knoten liefern.

Für kleine Dateien sieht das ganze recht komplex aus, aber man lernt es schnell zu schätzen wenn man sich durch den Baum bewegen kann. Damit wird die Anwendung auch robuster und kann kleinere Änderungen am XML leichter verkraften.

Jetzt aber noch ein Beispiel, wie man mit `XML::Twig` nur Teilbereiche einer (großen) Datei bearbeitet. Bei unserem Beispiel-XML interessieren mich wieder nur die Autoren. Den Rest möchte ich gar nicht betrachten. Also kann ich `XML::Twig` mitteilen, dass nur der `autoren`-Knoten interessant ist:

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

use strict;
use warnings;
use XML::Twig;

my $xml    = do{ local $/; <DATA> };
my @authors;

my $parser = XML::Twig->new(
    twig_handlers => {
        autoren => sub{ filter_authors(
                    \@authors, @_ )
                     },
    },
);
my $tree   = $parser->parse( $xml );

for my $author ( @authors ){
    print "Autor: ", $author;
}

sub filter_authors {
    my ($arref, $twig, $author) = @_;

    my @authors      = $author->children;
    for my $single_author ( @authors ){
        push @$arref, $single_author->text;
    }
}
```

Mit den `twig_handler`, die man dem Konstruktor übergeben kann, legt man die Tagnamen fest, bei denen eine bestimmt Subroutine ausgeführt werden soll. Hier kann man beliebig viele Handler angeben. Ich arbeite meistens mit anonymen Subroutinen, da ich bei Codereferenzen zu viel mit globalen Variablen arbeiten müsste und die Informationen zu speichern (wie in diesem Fall das Array `@authors`).

Die Subroutine, die dann ausgeführt wird, bekommt als ersten Parameter das `XML::Twig`-Objekt übergeben und als zweiten Parameter das Objekt des Knotens. Auch hier kann ich mich wieder mit den Methoden von `XML::Twig` durch den Baum arbeiten.

Die `twig_handler` sind wirklich sehr praktisch, wenn man nur bestimmte Teile eines XML-Dokuments bearbeiten möchte. Man kann nicht nur einfache Tagnamen übergeben, sondern auch XPath-ähnliche Angaben, so dass man den Bereich sehr genau bestimmen kann.

Apropos XPath: Man kann die gewünschten Knoten auch per XPath finden:

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

use strict;
use warnings;
use XML::Twig;

my $xml    = do{ local $/; <DATA> };

my $parser = XML::Twig->new();
my $tree   = $parser->parse( $xml );

my @authors = $parser->get_xpath(
    '//autoren/name'
);

for my $author ( @authors ){
    print "Autor: ", $author->text;
}
```

Dieses Modul bietet so viele Möglichkeiten, die in diesem Artikel nicht alle genannt werden können, da das zu weit führen würde. Insgesamt lässt sich aber sagen, dass das Modul sehr gut und mächtig ist. Ein Blick in die Doku wird das bestätigen.

XML::LibXML

Ein Modul, das in "harter" Konkurrenz zu `XML::Twig` steht, ist `XML::LibXML`. Beide Module haben ihre Vor- und Nachteile. Ein Vorteil von `XML::Twig` gegenüber diesem Modul ist, dass große XML-Dokumente Stück für Stück bearbeitet werden können, während `XML::LibXML` das komplette Dokument einliest. Aber so lange man nicht mit Dokumenten arbeitet die mehrere hundert Megabyte haben, sollte `XML::LibXML` gut und zuverlässig arbeiten. Ein Vorteil von `XML::`

LibXML gegenüber XML::Twig ist, dass der Speicherverbrauch bei gleichen XML-Dokumenten geringer ist. Außerdem bietet XML::LibXML die Möglichkeit, XML-Dokumente zu validieren (das wird in einem späteren Abschnitt behandelt).

Häufig wird das persönliche Empfinden entscheiden, welches der XML-Module man einsetzt.

Doch kommen wir jetzt zum Einsatz von XML::LibXML.

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

use strict;
use warnings;
use XML::LibXML;

my $xml     = do{ local $/; <DATA> };
my $parser  = XML::LibXML->new;
$parser->keep_blanks(0);

my $doc     = $parser->parse_string( $xml );

my $root    = $doc->getDocumentElement;
my $authors = $root->getElementsByTagName(
    'autoren'
);
my ($node)  = $authors->get_nodelist;

for my $author ( $node->getChildnodes ){
    print "Autor: ",
            $author->getFirstChild->getData;
}
```

Nachdem ein Objekt von XML::LibXML erzeugt wurde, muss noch mit keep_blanks(0) eingestellt werden, dass die Leerzeichen zwischen den Knoten entfernt werden sollen. Das spielt später noch eine Rolle.

XML::LibXML bietet genau wie die anderen Module die Möglichkeit, die Daten aus verschiedenen Quellen zu parsen. Wie in diesem Beispiel aus einem String, aber auch Daten direkt aus einem Filehandle oder einer Datei funktionieren.

Mit getDocumentElement wird dann das root-Element des XML-Dokuments geholt. Danach werden noch alle Elemente mit dem Tagnamen autoren geholt. Als Ergebnis erhält man ein NodeList-Objekt, dessen Elemente mit get_nodelist geholt werden können. Da es in unserem Beispiel nur einen Knoten mit dem Tagnamen autoren gibt, kann man den entsprechenden Knoten direkt einem Skalar zuweisen.

Und jetzt kommt die Stelle, an der es wichtig ist, dass vorher keep_blanks(0) gemacht wurde: Behält man das Standardverhalten von XML::LibXML bei, hat der Knoten 'autoren' mehrere Kinder. Zum einen die Leerzeichen nach dem Tag, dann der 'name'-Knoten und dann wieder die Leerzeichen zwischen 'name' und dem schließenden autoren. Uns interessieren aber nur die Namen der Autoren.

Man muss dann noch mit getFirstChild arbeiten, weil der Text als Kind-Knoten zu name angesehen wird.

Das ganze jetzt noch mit XPath:

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

use strict;
use warnings;
use XML::LibXML;

my $xml     = do{ local $/; <DATA> };
my $parser  = XML::LibXML->new;
my $doc     = $parser->parse_string( $xml );

my $root    = $doc->getDocumentElement;
my @authors = $root->findvalue(
   'autoren/name'
);

for my $author ( @authors ){
    print "Autor: ", $author;
}
```

Wie man sieht, kann man mit XPath und XML::LibXML sehr bequem an die gewünschten Daten kommen. Bei XML::LibXML ist XPath 1.0 erlaubt. Durch diese Möglichkeiten und den Validierungsfunktionen benutze ich mittlerweile in den meisten Fällen dieses Modul.

Allerdings muss man sich am Anfang etwas durch die Dokumentation kämpfen. Hier hilft häufig die "Reference Card" und der Artikel auf xml.com. Die Links sind unter *http://del.icio.us/foo_magazin* zu finden.

Validieren von XML-Dokumenten

Zu Beginn des Artikels habe ich schon auf XML-Schema und DTD aufmerksam gemacht. Gerade wenn Daten zwischen zwei Organisationen ausgetauscht werden, muss sichergestellt werden, dass die Daten valide sind. Diese Aufgabe kann mit XML::LibXML::Schema gelöst werden.

```
my $schema = XML::LibXML::Schema->new(
    location => $xsd_file
);
my $parser = XML::LibXML->new;
my $tree   = $parser->parse_file(
    $xml_file
);
eval{
    $schema->validate( $tree );
    1;
} or die "Can't validate $xml_file: $@";
```

Hier wurde das Block-`eval` um die Instanzierung von `XML::LibXML::Schema` weggelassen, da wir von einer validen XSD-Datei ausgehen. XSD steht für *XML-Schema-Definition* und ist ein konkretes XML-Schema. Sollte die Datei nicht valide sein, bricht das Programm schon bei der Instanzierung ab. Für Ungeübte ist die Fehlermeldung aber nicht immer aussagekräftig.

Danach wird der Baum mittels `XML::LibXML` erzeugt. Für die Validierung wird dem Schema-Objekt einfach der XML-Baum übergeben. Das ganze ist in ein Block-`eval` gepackt, weil sonst das Skript abbrechen würde wenn der XML-Baum nicht valide ist. Über dieses "try-catch"-Konstrukt mittels `eval` habe ich ausführlich in der "Frühjahr 2009"-Ausgabe von $foo berichtet.

Ein XML-Schema, das das Beispiel validieren würde, sieht so aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="autoren">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="name"
 type="xsd:string" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="inhalte">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="artikel"
 type="xsd:string" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="foo">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="inhalte" />
        <xsd:element ref="autoren" />
      </xsd:sequence>
      <xsd:attribute name="ausgabeId"
           type="xsd:integer" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Man sieht schon, dass das XML-Schema sehr ausführlich ist. In den folgenden Abschnitten wird auch noch die Validierung mit der DTD besprochen und ich kann jetzt schon vorwegnehmen, dass die DTD viel kürzer ist. Dafür kann man mit XML-Schema sehr viel genauer beschreiben wie das XML-Dokument aussehen muss.

In der Beispiel-XSD werden drei Elemente definiert, die die drei "komplexen" Elemente des XML-Dokuments repräsentieren. Arbeiten wir uns mal von unten nach oben durch das Schema. "foo" ist das `root`-Element des XML-Dokuments und ist ein "komplexes" Element (`complexType`). Mit Der folgenden Auflistung werden die Kind-Knoten im Baum definiert. Durch eine klare Einrückung von XML-Code lässt sich das Schema einfacher erstellen, da man so relativ leicht die Kind-Knoten identifizieren kann. Das `ref` gibt an, auf welche Element-Definition in der XSD der Kind-Knoten verweist. Mit `xsd:attribute` kann man noch die Attribute eines Elements definieren.

Was auf jeden Fall noch erwähnenswert ist, ist die Definition der "Blätter" des Baums. Z.B. das Element "artikel". In der XSD wird gesagt, dass der Knoten vom Typ "string" ist und unbegrenzt häufig unterhalb von "inhalte" vorkommen darf.

XML-Schema definiert einige simple Datentypen wie "string", "integer" und andere. Man kann dann eigene komplexe Datentype darauf aufbauen und Einschränkungen einbauen:

```
<xsd:simpleType name="monatInt">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="1"/>
    <xsd:maxInclusive value="12"/>
  </xsd:restriction>
</xsd:simpleType>
```

Wie schon angedeutet, ist die DTD viel kürzer, hat aber auch weniger Möglichkeiten. Wir speichern die Definitionen in einer externen Datei und binden diese dann in der XML-Datei ein:

```
<?xml version="1.0" encoding="UTF-8"
      standalone="yes"?>
<!DOCTYPE foo SYSTEM "test.dtd">
<foo ausgabeId="12">
  <!-- weiteres XML wie oben -->
</foo>
```

Dann sieht die dazugehörige DTD wie folgt aus:

```
<!ELEMENT foo (inhalte,autoren)>
<!ELEMENT inhalte (artikel)+>
<!ELEMENT autoren (name)+>
<!ELEMENT artikel (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ATTLIST foo
    ausgabeId CDATA #REQUIRED
>
```

Hier wird festgelegt, dass das Element "foo" Unterknoten "inhalte" und "autoren" hat. Die folgenden Definitionen beschreiben, wie diese Unterknoten genau aussehen: Inhalte besteht aus mindestens einem "artikel". Das "+" und andere Mengenangaben dürfte von den Regulären Ausdrücken in Perl bekannt sein. Mit `#PCDATA` wird ausgedrückt, das der Knoten jeglichen Text enthalten darf. Genauer gehe ich aber auf das Format der DTD nicht ein, da das den Rahmen sprengen würde.

Bei der Validierung muss man aber etwas aufpassen:

```
my $schema = XML::LibXML::Dtd->new(
    "", $dtd_file
);
my $parser = XML::LibXML->new;
$parser->keep_blanks(0);

my $tree   = $parser->parse_file(
    $xml_file
);

eval{
    print $tree->validate( $schema );
    1;
} or die "Can't validate $xml_file: $@";
```

Zuerst muss ähnlich wie bei der Validierung mit Schema ein Objekt für die Definition erstellt werden. Da wir nur ein System-File verwenden, übergeben wir als ersten Parameter einen leeren String und als zweiten Parameter den Pfad zur DTD-Datei. Danach wird eine neue Instanz von `XML::LibXML` erzeugt. Die Zeile mit `$parser->keep_blanks(0)` ist sehr wichtig, weil `libxml2` die Leerzeichen aus dem XML-Dokument beibehält und bei der Validierung eine Fehlermeldung mit `standalone: foo declared in the external subset contains white spaces` erscheint.

In dem Block-`eval` wird dann das XML validiert. Anders als bei der Validierung mit dem XML-Schema, wird hier die Funktion `validate` des Tree-Objekts benutzt und nicht vom DTD.

Transformation mittels XSLT

Wenn XML-Dateien nicht als Konfigurations-Dateien gebraucht werden, müssen sie häufig in andere Formate übertragen werden. Diesen Vorgang nennt man Transformation. Auch für diese Aufgabe gibt es verschiedene Perl-Module. Ich werde hier die meisten Beispiele hier mit `XML::XSLT` und mit `XML::LibXML` in Verbindung mit `XML::LibXSLT` zeigen.

Für alle Wege, die ich hier zeige, werde ich das Stylesheet aus Listing 1 verwenden.

Damit wird das Beispiel XML-Dokument in ein HTML-Format überführt, in dem es eine Überschrift mit der Nummer der $foo-Ausgabe gibt und alle Autoren in einer Liste aufzählt:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:element name="h1">foo - Ausgabe: <xsl:value-of select="/foo/@ausgabeId" />
    </xsl:element>
    <xsl:apply-templates select="/foo/autoren" />
  </xsl:template>

  <xsl:template match="autoren">
    Autoren dieser Ausgabe: <xsl:element name="br" />
    <xsl:element name="ul">
    <xsl:apply-templates select="./name" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="name">
    <xsl:element name="li"><xsl:value-of select="." /></xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

Listing 1

```
<?xml version="1.0"?>
<h1>foo - Ausgabe: 12</h1>
    Autoren dieser Ausgabe:
<br/><ul><li>Renee Baecker</li></ul>
```

XML::XSLT

`XML::XSLT` ist ein PurePerl-Modul, das schon seit einigen Jahren existiert. Die Transformation mit diesem Modul durchzuführen ist ziemlich simple:

```
#!/usr/bin/perl

use strict;
use warnings;
use XML::XSLT;

my $xslfile = "test.xsl";
my $xmlfile = "test.xml";

my $xslt = XML::XSLT->new( $xslfile );
print $xslt->serve( $xmlfile );
$xslt->dispose();
```

Dem Konstruktor von `XML::XSLT` muss man mindestens den Pfad zur XSLT-Datei übergeben. Die Transformation selbst findet erst beim Aufruf von `serve` statt, dem man noch den Pfad zur XML-Datei übergeben muss. Die Methode `dispose` sollte aufgerufen werden wenn die Transformation nicht am Ende eines Programms durchgeführt wird, es sorgt dafür, dass Speicher an den Interpreter gegeben wird.

Wer sich dafür interessiert, was im Hintergrund abgeht, kann dem Konstruktor die Option `debug => 1` mitgeben und man bekommt mitgeteilt, was der Parser so alles macht.

```
creating parser object:
 opening xsl...
 opening document
 Opening URL
 preprocessing stylesheet...
# ein paar Sachen ausgelassen
processing project:
 processing project:
  matching template for "/" with count 1
        and path "":
    #6 "name" did not match
    #5 "autoren" did not match
    found #4 with "/" in "/"
  Source

  evaluating template content with current
  path "":
   XML::DOM::Text
    adding node (non-deep)..
```

So kann man sich auf die Suche nach Fehlern machen, wenn das Stylesheet nicht zu den gewünschten Ergebnissen führt.

`XML::XSLT` liefert standardmäßig das Resultat mit einem HTTP-Header und der XML-Deklaration aus:

```
Content-Type: text/xml
Content-Length: 130

<?xml version="1.0" encoding="UTF-8"?>
```

In einigen Fällen ist dies sicherlich nicht gewollt. Dann muss man bei der `serve`-Methode die Optionen `http_headers` und `xml_declaration` auf "0" setzen.

XML::LibXSLT + XML::LibXML

Wenn man sich für das Parsen von XML, dem Schreiben von XML, das Validieren von XML und für die Transformation nicht mit mehreren verschiedenen Modulen beschäftigen möchte, dann ist dieser Abschnitt sicherlich sehr interessant. Mit dem zusätzlichen Modul `XML::LibXSLT` ist es auch mit `XML::LibXML` möglich, XML-Dokumente zu transformieren. Ich verwende wieder das gleiche Stylesheet wie im Beispiel zuvor.

```
#!/usr/bin/perl

use strict;
use warnings;
use XML::LibXSLT;
use XML::LibXML;

my $xml_file  = 'test.xml';
my $xslt_file = 'test.xsl';

my $libxml  = XML::LibXML->new();
my $libxslt = XML::LibXSLT->new();

my $parsed_xml = $libxml->parse_file(
    $xml_file
);

my $stylesheet =
  $libxslt->parse_stylesheet_file(
    $xslt_file
);

my $result = $stylesheet->transform(
    $parsed_xml
);

print $stylesheet->output_string( $result );
```

Zuerst müssen sowohl die XML-Datei als auch das Stylesheet mit `XML::LibXML` geparst werden. Als nächstes muss das Stylesheet mit `XML::LibXSLT` geparst werden (`parse_stylesheet_file`). Dann wird das XML mit der Methode `transform` transformiert. Mit `output_string` kann man sich das Ergebnis holen.

```
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:foo="urn:foo">
  <xsl:template match="/">
    Heute ist der <xsl:value-of select="foo:print_time()" />
    <!-- hier der Rest von Stylesheet -->
```

Listing 2

Mit `XML::LibXSLT` kann man auch eigene Funktionen registrieren und bei bestimmten Knoten dann diese Callbacks ausführen lassen. Soll z.B. am Anfang des HTML-Dokuments noch das aktuelle Datum ausgegeben werden, fügt man den Namespace "foo" und einen Funktionsaufruf im Stylesheet hinzu (siehe Listing 2). und man registriert für die URN "foo" einfach die Funktion `print_time`.

```
XML::LibXSLT->
   register_function("urn:foo", "print_time",
     sub { scalar localtime });
```

Das Ergebnis sieht dann so aus:

```
<?xml version="1.0"?>

Heute ist der Fri Mar  6 09:55:36 2009
<h1>foo - Ausgabe: 12</h1>
    Autoren dieser Ausgabe:
<br/><ul><li>Renee Baecker</li></ul>
```

Renée Bäcker

DPath ist nicht XPath!

Intro

Zeigen Sie anhand selbstgewählter Beispiele den Einsatz einer XPath-ähnlichen Abfragesprache auf Datenstrukturen, erläutern Sie an gegenwärtig aktuellen Fragen Aspekte ihrer Wirkungsweise und diskutieren Sie bestehende Alternativen.

Abstract

Ich sammle beruflich Testergebnisse in einer Datenbank. Tests produzieren das in der Perl-Testwelt übliche TAP (Test Anything Protocol).

Neben einer Web-Oberfläche, welche die Testergebnisse mittels `TAP::Parser` und Freundesmodulen präsentiert, stelle ich auch eine Schnittstelle zum skriptfähigen Zugriff auf die Daten zur Verfügung.

Das Gesamtkonzept habe ich auf der YAPC::Europe 2009 präsentiert, hier möchte ich auf die beiden Kernmodule eingehen, die das ganze treiben: `Data::DPath` und `TAP::DOM`.

TAP am Beispiel

Tests produzieren als Ausgabe TAP. Ich nutze insbesondere eingebettete YAML-Blöcke, um Daten zu transportieren, z.B. Benchmark-Ergebnisse oder Meta-Daten der Test-Umgebung. Das sieht etwa so aus:

```
TAP Version 13
1..3
ok 1 - Kernel has feature X
not ok 2 - hot stuff # TODO unfinished
ok 3 - benchmark results
  ---
  timing:
     count: 12654
     throughput: 12.924
  metadata:
     kernel: 2.6.32
     ram: 8192
  ...
```

Listing 1

Dieser Test-Report besagt:

- verwende Features der TAP-Protokoll-Version 13 (das eingebettete YAML)
- der Report umfasst 3 Einzeltests
- der erste Test mit Beschreibung *Kernel has feature X* war erfolgreich
- der zweite Test *hot new stuff* war nicht erfolgreich, aber er ist als TODO (einschließlich Erklärung) markiert, so dass ein Fehlschlagen erlaubt ist und den Gesamterfolg nicht beeinträchtigt
- der dritte Test *benchmark results* war erfolgreich und zu ihm gehört ein Block aus Daten, die in (einem Subset von) YAML beschrieben sind.

In der Perl-Welt produziert man solche Reports üblicherweise mit `Test::More`, man kann es aber auch manuell mit `print` und `echo` aus C oder Shell heraus tun.

Wenn man einen solchen TAP-Report via `TAP::Parser` bzw. dessen Frontend-Tool **prove** auswertet, bekommt man im wesentlichen genau eine vorbereitete Gesamtauswertung, mit einer Statistik, wieviele Tests fehlgeschlagen sind, welche Nummern das waren, etc.

Mir reicht das nicht.

Big Picture

Mein eigentlicher Anwendungsfall, den ich bei der YAPC::Europe 2009 präsentiert habe, ist eine Testdatenbank, auf die mit einer Query-Language zugegriffen wird. Dort war allerdings zusätzlich noch der Zugriff auf die Historie von Reports möglich und alles kombiniert in einer Remote-API.

Little Picture

In diesem Artikel erläutere ich statt der großen Anwendung die Einzelmodule `Data::DPath` und `TAP::DOM` und wie man sie mit der Kommandozeilenapplikation **dpath** als Frontend kombinieren kann.

Das alles stellt nur einen Beispiel-Use-Case "Auswertung von Tests" dar. Weitere Use-Cases kann man sich selber konstruieren: mit DPath andere große Datenstrukturen, wie z.B. riesige DBIx::Class-Objekte untersuchen oder TAP::DOM anders verwenden, etc.

Data::DPath

`Data::DPath` erlaubt den Zugriff auf Datenstrukturen mit einer Pfad-Syntax. Das Vorbild ist XPath.

Warum nicht richtiges XPath?

Die auf CPAN vorhandenen XPath-Engines können nur eingeschränkt auf Perl-Datenstrukturen angewendet werden. Das liegt am Unterschied zwischen Perl-Datenstrukturen und XML-Dokumenten.

- XPath ist für XML; DPath für Datenstrukturen (Hashes, Arrays), mit stärkerem Perl-Fokus.
- Obwohl auch XML-Dokumente als Datenstruktur dargestellt werden können, sind sie speziell.
- Elemente in XML haben immer eine Ordnung/Reihenfolge, im Gegensatz zu Hash-Elementen in Perl.
- XML-Element-Namen auf dem gleichen Level können wiederholt vorkommen, nicht so Hash-Keys.
- XML-Element-Namen sind eingeschränkter als beliebig schräge Hash-Keys.
- XML-Elemente können Attribute haben und diese können in XPath adressiert werden; Perl Datenstrukturen brauchen das nicht. Umgekehrt können Datenstrukturen in Perl g*eblesste* Elemente enthalten, die DPath adressieren kann.
- XML-Dokumente verwenden evtl. Namespaces; Datenstrukturen nicht.
- Arrays mit Index 1 wie in XPath zu starten wäre in Perl-Datenstrukturen verwirrend.
- DPath erlaubt Filter-Ausdrücke, die beliebige Perl-Ausdrücke und somit entsprechend mächtig sind, es ist keine eigene Sub-Language wie in XPath.

Synopsis

Die Verwendung von Data::DPath sieht man in Listing 2.

Es matcht eine Datenstruktur `$data` gegen einen DPath-Pfad, der besagt: ausgehend vom Wurzelelement "/" folgt erst ein Key "AAA", dann "*irgendwas*" ("*"), dann ein Key CCC.

Auf diesen Pfad passen diese beiden Zugriffe:

```
$data->{AAA}{BBB}{CCC}
$data->{AAA}{RRR}{CCC}
```

Um die passenden Elemente zu finden, arbeitet `Data::DPath` einen Pfad Schritt für Schritt ab. Ein Schritt ist der Teil bis zum nächsten "/". Es matcht diesen Schritt gegen die bisher gefundenen Elemente, was anfangs immer das Wurzelelement ist. Es ermittelt in jedem Schritt alle matchenden Unterelemente, geht zum nächsten Schritt und vergleicht erneut.

```
use Data::DPath 'dpath';
my $data  = {
             AAA  => { BBB => { CCC  => [ qw/ XXX YYY ZZZ / ] },
                       RRR => { CCC  => [ qw/ RR1 RR2 RR3 / ] },
                       DDD => { EEE  => [ qw/ uuu vvv www / ] },
                     },
           };
$resultlist = $data ~~ dpath '/AAA/*/CCC';

# [ ['XXX', 'YYY', 'ZZZ'], [ 'RR1', 'RR2', 'RR3' ] ]
```

Listing 2

Spezial-Pfadschritte

Die Mächtigkeit ergibt sich aus den Spezial-Schritten "*", ". .", ".", "//" und zusätzlichen Filterausdrücken, die man in eckigen Klammern ("[]") an einen Schritt anhängen kann. Wenn die Hashkeys schwierig aussehen, kann man sie mit Hochkommas (" oder ') quoten.

Dabei bedeutet

• // Finde jeden Punkt in der Datenstruktur unterhalb der aktuell passenden.
Es wird oft am Anfang benutzt, um irgendwo mittendrin in der Datenstruktur zu beginnen:

```
//Zoo/Käfig/Säugetiere/Verwandte/Affe
```

oder um unbekannte Zwischenschritte automatisch finden zu lassen:

```
//Käfig//Affe
```

(Natürlich sollten Affen auch im Zoo lieber auf Bäumen leben. Sie sind ja doch recht lässige Gesellen.)

• * Passt auf alle Elemente an dieser Stelle, macht praktisch "*irgendeinen*" Schritt. Das sind üblicherweise alle Hashkeys oder alle Element in einem Array, relativ zum aktuellen Element.

• . . Matcht das direkte Elternelement relativ zum bisherigen Punkt.

• . Der "*Kein-Schritt*". Dieser Schritt bleibt passiv am aktuellen Punkt, geht also nicht weiter in die Tiefe und erlaubt damit das in Reihe schalten ("chaining") von Filterregeln, um die Einzelausdrücke einfach zu halten, oder auch das Filtern von schwierig erreichbaren Punkten, wie z.B. dem Root-Element der Datenstruktur.

Das sieht z.B. so aus:

```
/.[ FILTER ]
/aaa/bbb/.[FILTER1]/.[FILTER2]/.[FILTER3]
```

Siehe auch Abschnitt über *Filter* weiter unten.

• Werden die genannten Spezial-Elemente als Hashkeys benötigt oder tauchen Slashes "/" in Hashkeys auf, hilft Quoting:

```
/".."/
/"."/
/"//"/
/"*"/
/"*"[ filter ]/
/"AAA/BBB"/
```

Damit sehen weitere Pfadbeispiele aus, wie in Listing 3 dargestellt.

Filter

Filter können in eckigen Klammern an einem Pfadschritt angegeben werden. Sie enthalten eine Bedingung, die auf jedes einzelne der in diesem Schritt gefundenen Elemente angewendet wird und erfüllt sein muss, um das Element mit in die Ergebnismenge des aktuellen Schrittes aufzunehmen.

Es funktioniert ähnlich wie ein `grep{}` in Perl, zusätzlich mit ein paar speziellen Funktionen und syntaktischem Zucker.

Nachfolgend ein paar Beispiele.

`/AFFE/*[2]/`

Ein Integerwert (auch negativ) bedeutet, wähle das Element mit diesem Index aus dem Array. Im Beispiel findet "*" alle Unterelemente des Schrittes AFFE und der Filter "[2]" reduziert diese Menge auf das dritte Element (Arrayindex bei 0 beginnend)..

```
$data ~~ dpath '/AAA/*/CCC'
$data ~~ dpath '/AAA/BBB/CCC/../..'    # parents  (..)
$data ~~ dpath '//AAA'                 # anywhere (//)
$data ~~ dpath '//AAA/*'               # anywhere + anystep
$data ~~ dpath '//AAA/*[size == 3]'    # filter by arrays/hash size
$data ~~ dpath '//AAA/*[size != 3]'    # filter by arrays/hash size
$data ~~ dpath '/"EE/E"/CCC'           # quote strange keys
$data ~~ dpath '/AAA/BBB/CCC/*[1]'     # filter by array index
$data ~~ dpath '/AAA/BBB/CCC/*[ idx == 1 ]' # same, filter by array index
$data ~~ dpath '//AAA/BBB/*[key eq "CCC"]'  # filter by exact keys
$data ~~ dpath '//AAA/*[ key =~ /CC/ ]'     # filter by regex matching keys
$data ~~ dpath '//CCC/*[ value eq "RR2" ]'  # filter by values of hashes
```

Listing 3

```
/AFFE/*[ idx == 2 ]/
```
Die ausführliche Form des vorherigen Beispiels.
Das "*" matcht wieder alle Elemente unterhalb von `AFFE` und der Filter wählt nur die Elemente, deren Index gleich 2 ist.

```
/AFFE/*[key eq "CCC"]
```
In allen Elementen unterhalb von `AFFE` matcht es nur jene, deren Key "CCC" ist. Diesmal wird also "*" als Hash und jedes Element darin als Key/Value-Paar interpretiert.

```
/AFFE/*[key =~ /CCC/ ]
```
Die Regex-Variante des vorherigen Beispiels. In allen Elementen unterhalb von `AFFE` matcht es nur jene, deren Keys auf den Regex "/CCC/" matchen, also z.B. auch Keys "fooCCCbar".

```
//AFFE/*[value eq "RR2"]
```
Finde Elemente unterhalb von `AFFE`, deren Wert gleich `RR2` ist. Oft kombiniert man dies mit dem Zurückgehen zum Elternelement, um Nachbarelemente zu finden, wie im folgenden:

```
//AFFE/*[value eq "RR2"]/..
```
Findet einen Punkt in der Struktur, unterhalb dem es ein Key/Value-Paar gibt, dessen Value gleich "RR2" gibt.

```
//AFFE[size >= 3]
```
Findet `AFFE`-Elemente, sofern sie Arrays oder Hashes mit mindestens 3 Elementen sind.

Es gibt im Vorbild "XPath" noch ein paar weitere Möglichkeiten, die manches vereinfachen würden. Dort gibt es das Konzept der "Achsen", von denen in DPath insbesondere noch die Eltern-Achse fehlt, um über Arrayzugriffe auf weiter oben stehende Elemente zuzugreifen, was man in DPath derzeit noch so macht:

```
//AFFE/../../../../
```

So etwas könnte mit expliziten Achsen leichter gehen:

```
//AFFE/::parents[3]/
```

Das geht momentan noch nicht.

Es sollte aber trotzdem alles ausdrückbar sein. Emailt mir Beispiele, wenn ihr ein Element nicht mit DPath erreicht, aber mit XPath.

Worauf nun diese Pfadsprache anwenden? Auf große Datenstrukturen. Mein Beispiel sind Testberichte in TAP, da sind wir beim nächsten Kapitel.

TAP::DOM

Das Modul `TAP::DOM` macht etwas ganz einfaches: es parst einen TAP-Bericht mit `TAP::Parser` und konvertiert das Ergebnis in eine große Datenstruktur aus Arrays und Hashes.

Der Aufbau der Struktur ist recht kanonisch: Die Namen der Hash-Keys richten sich nach den Namen der Callbacks in `TAP::Parser`, Einzelzeilen werden in einem Array aufgehoben und Diagnostics-Zeilen und YAML-Blöcke werden als Kindelemente unterhalb der vorherigen Testzeile eingeordnet.

Es gibt damit nur ein einziges Problem: die Datenstruktur ist groß. Daher werden wir sie nachher mit `Data::DPath` auswerten.

Beispiel DOM

Einen leicht kommentierten und für bessere Lesbarkeit manuell nachbearbeiteten Ausschnitt eines TAP-DOMs sieht man in Listing 4 (Seite 16 bis 18).

Viel mehr gibt es über `TAP::DOM` nicht zu sagen.

Es enthält den Inhalt eines Testruns in einer großen Datenstruktur. Und ist damit prädestiniert, mit `Data::DPath` ausgewertet zu werden. Beide Module zusammenzubringen, könnte in Pseudocode so aussehen:

1. use Data::DPath;
2. use TAP::DOM;
3. $raw_tap = read_from_file_or_stdin();
5. $data = TAP::DOM->new({ tap => $raw_tap });
6. $result = $data ~~ dpath 'pfadstring';
7. $result ausgeben, z.B. als YAML oder mit Data::Dumper

Das alles und noch viel mehr macht `App::DPath` (weiter auf Seite 35).

```
bless( {
 'version'       => 13,
 'plan'          => '1..6',
 'tests_planned' => 6
 'tests_run'     => 8,
 'is_good_plan'  => 0,
 'has_problems'  => 2,
 'skip_all'      => undef,
 'parse_errors'  => [
                     'Bad plan.  You planned 6 tests but ran 8.'
                    ],
 'pragmas'       => [
                     'strict'
                    ],
 'exit'          => 0,
 'start_time'    => '1236463400.25151',
 'end_time'      => '1236463400.25468',
 # für die Bedeutung des "summary" siehe auch TAP::Parser::Aggregator.
 'summary' => {
               'status'          => 'FAIL',
               'total'           => 8,
               'passed'          => 6,
               'failed'          => 2,
               'all_passed'      => 0,
               'skipped'         => 1,
               'todo'            => 4,
               'todo_passed'     => 2,
               'parse_errors'    => 1,
               'has_errors'      => 1,
               'has_problems'    => 1,
               'exit'            => 0,
               'wait'            => 0
               'elapsed'         => bless( [
                                            0,
                                            '0',
                                            0,
                                            0,
                                            0,
                                            0
                                           ], 'Benchmark' ),
               'elapsed_timestr' => ' 0 wallclock secs (
                                      0.00 usr +  0.00 sys =  0.00 CPU)',
             },
 'lines' => [
             {
              'is_actual_ok' => 0,
              'is_bailout'   => 0,
              'is_comment'   => 0,
              'is_plan'      => 0,
              'is_pragma'    => 0,
              'is_test'      => 0,
              'is_unknown'   => 0,
              'is_version'   => 1,                    # <---
              'is_yaml'      => 0,
              'has_skip'     => 0,
              'has_todo'     => 0,
              'raw'          => 'TAP version 13'
              'as_string'    => 'TAP version 13',
             },
             {
               'is_actual_ok' => 0,
               'is_bailout'   => 0,
               'is_comment'   => 0,
               'is_plan'      => 1,                    # <---
               'is_pragma'    => 0,
               'is_test'      => 0,
               'is_unknown'   => 0,
               'is_version'   => 0,
               'is_yaml'      => 0,
               'has_skip'     => 0,
               'has_todo'     => 0,
               'raw'          => '1..6'
               'as_string'    => '1..6',
             },
```

Listing 4 (Fortsetzung auf Seite 34)

```
            {
              'is_actual_ok' => 0,
              'is_bailout'   => 0,
              'is_comment'   => 0,
              'is_ok'        => 1,                    # <---
              'is_plan'      => 0,
              'is_pragma'    => 0,
              'is_test'      => 1,                    # <---
              'is_unknown'   => 0,
              'is_unplanned' => 0,
              'is_version'   => 0,
              'is_yaml'      => 0,
              'has_skip'     => 0,
              'has_todo'     => 0,
              'number'       => '1',                  # <---
              'type'         => 'test',
              'raw'          => 'ok 1 - use Data::DPath;'
              'as_string'    => 'ok 1 - use Data::DPath;',
              'description'  => '- use Data::DPath;',
              'directive'    => '',
              'explanation'  => '',
              '_children'    => [
                                  # ----- "_children" sind nachfolgende
                                  #       Kommentar/YAML-Zeilen -----
                                  {
                                    'is_actual_ok' => 0,
                                    'is_unknown'   => 0,
                                    'has_todo'     => 0,
                                    'is_bailout'   => 0,
                                    'is_pragma'    => 0,
                                    'is_version'   => 0,
                                    'is_comment'   => 0,
                                    'has_skip'     => 0,
                                    'is_test'      => 0,
                                    'is_yaml'      => 1,              # <---
                                    'is_plan'      => 0,
                                    'raw'          => '   ---
  - name: \'Hash one\'
    value: 1
  - name: \'Hash two\'
    value: 2
...'
                                    'as_string'    => '   ---
  - name: \'Hash one\'
    value: 1
  - name: \'Hash two\'
    value: 2
...',
                                    'data'         => [
                                                        {
                                                          'value' => '1',
                                                          'name' => 'Hash one'
                                                        },
                                                        {
                                                          'value' => '2',
                                                          'name' => 'Hash two'
                                                        }
                                                      ],
                                  }
                                ],
            },
            {
              'is_actual_ok' => 0,
              'is_bailout'   => 0,
              'is_comment'   => 0,
              'is_ok'        => 1,                    # <---
              'is_plan'      => 0,
              'is_pragma'    => 0,
              'is_test'      => 1,                    # <---
              'is_unknown'   => 0,
              'is_unplanned' => 0,
              'is_version'   => 0,
              'is_yaml'      => 0,
```

Listing 4 (Fortsetzung von Seite 33)

```
                'has_skip'      => 0,
                'has_todo'      => 0,
                'explanation'   => '',
                'number'        => '2',                    # <---
                'type'          => 'test',
                'description'   => '- KEYs + PARENT',
                'directive'     => '',
                'raw'           => 'ok 2 - KEYs + PARENT'
                'as_string'     => 'ok 2 - KEYs + PARENT',
              },
              # etc., Rest siehe TAP-DOM/t/some_tap.dom
            ],
}, 'TAP::DOM')                                             # blessed
```

Listing 4 (Fortsetzung von Seite 34)

App:DPath

Mit `App::DPath` wird ein Kommandozeilentool **dpath** als Frontend bereitgestellt, was die üblichen Vor- und Nacharbeiten um die Benutzung von `Data::DPath` kapselt:

- Input von Datei oder STDIN lesen
- Input in Datenstruktur wandeln
- dazu verschiedene Input-Formate verstehen (YAML, JSON, XML, TAP) und automatisch die entsprechenden CPAN-Module anwenden, um die jeweilige Datenstruktur zu erhalten
- einen gegebenen Pfad gegen diese Struktur matchen
- die Ergebnismenge ausgeben, optional in verschiedenen Ausgabeformaten: YAML, JSON, Data::Dumper

Wie machen wir das?

TAP besorgen

TAP erhalten wir aus dem Ausführen von Testscripts:

```
perl t/affe.t
```

Leider können wir nicht **prove** verwenden. Obwohl das fast wieder das gleiche TAP ausgibt, das es geparst hat, fügt es doch selbst (imho ungeschickterweise) Zeilen hinzu, die selbst TAP sind, konkret ein abschließendes einzelnes "ok", und erzeugt damit neuen Inhalt, der nicht mehr auf den "plan" passt. Schade. Schön wäre es, wenn **prove** idempotent wäre, naja.

Obige Zeile ist etwas idealisiert, ausführlich müssen wir paar Dinge selber machen, die sonst **prove** mit wenigen Optionen für uns macht:

```
perl -Ilib t/affe.t 2>&1
```

So verwendet es unser lokales `./lib` Verzeichnis und mischt STDIN mit STDERR zusammen. Letzteres ist wichtig, anderenfalls kommen die diagnostics-Zeilen asynchron mit den ok-/not_ok-Zeilen; wir brauchen diesen Zusammenhang aber, damit es im TAP::DOM die richtigen "children"-Strukturen aufbaut.

dpath verwenden

Jetzt kommt die Essenz des ganzen Artikels.

Wir pipen einfach den TAP-Strom in **dpath** und sagen noch, dass es TAP ist, damit es die Datenstruktur mittels `TAP::DOM` aufbaut. Als DPath geben wir einfach mal das Wurzelelement "/" an:

```
perl t/affe.t | dpath -i tap '/'
```

Das default-Ausgabeformat von `dpath` ist YAML. Obige Zeile gibt uns also den kompletten TAP-DOM als YAML aus.

Jetzt interessieren uns davon Detailergebnisse.

Gib `FAIL` oder `PASSED` aus:

```
perl t/affe.t |
  dpath -i tap '/summary/status'
```

Gib Anzahl der ausgeführten Tests:

```
perl t/affe.t | dpath -i tap '/summary/total'
```

Gib Anzahl der erfolgreichen Tests:

```
perl t/affe.t |
  dpath -i tap '/summary/passed'
```

Gib Anzahl der fehlgeschlagenen Tests:

```
 perl t/affe.t |
   dpath -i tap '/summary/failed'
```

Und so weiter. Bis dahin noch recht öde.

Jetzt wollen wir das Ergebnis aller Tests wissen, deren Beschreibung irgendwas mit "AFFE" enthält:

```
perl t/affe.t | dpath -i tap '/
  lines/*/description[value =~ /AFFE/]/..'
```

oder noch etwas schwammiger:

```
perl t/affe.t | dpath -i tap
  '//description[value =~ /AFFE/]/..'
```

Wir steigen also hinab bis zur passenden Beschreibung und wieder eins hoch, weil wir den ganzen Datensatz sehen wollen.

Wenn wir an die eingebetteten YAML-Daten im TAP-Beispiel am Anfang des Artikels herankommen möchten, z.B. den "throughput"-Wert, dann so (Zeilenumbrüche ignorieren):

```
perl t/affe.t |
dpath -i tap \
  '//description[value =~ /benchmark/]/
  ../_children//data/throughput'
```

oder, wenn wir sicher sind, dass die Keys nirgendwo anders vorkommen:

```
perl t/affe.t |
   dpath -i tap '//data/throughput'
```

Jetzt wollen wir alle Tests sehen, die als TODO markiert sind, aber trotzdem ok geliefert haben, also unerwartet erfüllt wurden:

```
perl foo.t |
   dpath -i tap
   '//has_todo[value==1]/
   ../is_actual_ok[value==1]/..'
```

Das gibt dann sowas aus:

```
---
- as_string:     "ok 149 - ANYWHERE + NOSTEP #
                  TODO deferred"
  description:   "- ANYWHERE + NOSTEP"
  directive:     TODO
  explanation:   deferred
  has_todo:      1
  is_actual_ok:  1
  is_ok:         1
  is_test:       1
  number:        149
  type:          test
```

Wenn die Testmenge nur unübersichtlich genug ist, ist das ein nützlicher Fortschritt.

Möchten wir lieber `Data::Dumper` als Ausgabeformat, dann einfach noch die Option "`-o dumper`" hinzufügen.

Implementierungsdetails

Die folgenden Abschnitte zeigen ein paar Details, wie Data::`DPath` umgesetzt wurde.

Text::Balanced

Ohne es im Detail zu erklären, kann ich sehr `Text::Balanced` empfehlen, auf dem die Pfad-Parse-Mechanik in `Data::DPath` basiert. Es ist ein guter Mittelweg, wenn Regexe nicht mehr reichen und eine komplette Grammatik Overkill scheint, weil das Problem gefühlt "irgendwie einfacher" lösbar sein muss.

Im Kern ist das DPath-Parsen nur ein schrittweises Zerlegen von balancierten Ausdrücken, entweder zwischen zwei Slashes "/" oder zwischen eckigen Klammern "[]". Plus paar Sonderlocken, um z.B. mit Slashes innerhalb der eckigen Klammern klarzukommen.

Modern Perl

Good vs. evil, god vs. the devil, what side are you on? Ich jedenfalls habe `Data::DPath` mit new school dependencies geschrieben, konkret mit Moose, MooseX::Declare und Perl 5.10. Das gibt mir neben diesem Implementierungsstil:

```
use MooseX::Declare;
use 5.010;

class Data::DPath is dirty {
    clean;
    method match (Any $data, Str $path) {
        given ($step->kind) {
            when ('ANYWHERE') {
                # ...
            }
        }
    }
}
```

auch noch den Smartmatch-Operator `~~`, der zum Überladen für diesen Anwendungsfall, dass Daten gegen etwas "matchen", wie geschaffen war.

Damit gebe ich zurück ins Studio.

Steffen Schwigon

Neue Syntax dank Devel::Declare

Ein paar Seiten zuvor wurde schon gezeigt, wie man mit bestimmten Modulen Methoden-Signaturen in seinen Perl-Anwendungen verwenden kann. `Method::Signatures` verwendet dabei `Devel::Declare`, um das Schlüsselwort "method" zu implementieren. In diesem Artikel soll beschrieben werden, wie man selbst die Syntax von Perl erweitern kann. Hier soll beispielhaft der Kontext von Subroutinen erzwungen werden und dafür die Schlüsselwörter "void", "list" und "single" bereitgestellt werden. Hier wird "single" genommen, weil `scalar` schon eine Built-In-Funktion von Perl ist.

Am Ende soll in einem Skript oder Modul das hier möglich sein:

```
void test { print "test" };

list names {
    my @names = qw/llama camel/;
    return @names;
}

test();
my @array = names();
print $array[0];

my $scalar = test(); # wirft Fehler;
```

Das Modul, das eigene Schlüsselwörter ermöglicht, ist `Devel::Declare`. Um es anfangs etwas leichter zu machen, verwenden wir `Devel::Declare::MethodInstaller::Simple`.

Erst mit dem Hilfsmodul

Mit `Devel::Declare::MethodInstaller::Simple` ist es sehr einfach, ein neues Schlüsselwort hinzuzufügen (Listing 1)

```
package CheckContextSimple;

use strict;
use warnings;
use parent
  'Devel::Declare::MethodInstaller::Simple';

sub import {
    my ( $class ) = @_;

    my $caller    = caller;

    $class->install_methodhandler(
        into            => $caller,
        name            => 'void',
    );
}

1;
```

Listing 1

Wer es noch nicht kennt: `parent` ist ein leichtgewichtiger Ersatz für `base` (was weiterhin existiert). Mit `use parent 'Modul'` erbt man also von dem angegebenen Modul.

In `import` wird dann `install_methodhandler` aufgerufen, das von dem `Devel::Declare`-Hilfsmodul geerbt wird. Man muss dieser Methode noch mitgeben, in welches Package die Subroutine "installiert" werden soll und wie das Schlüsselwort heißt.

Jetzt läuft das Skript

```
#!/usr/bin/perl

use strict;
use warnings;
use CheckContextSimple;

void test { print "test" };
test();
```

schon durch. Allerdings findet im Moment noch kein Check statt, ob `test` wirklich im `void`-Kontext aufgerufen wird. Dazu muss man noch eine Methode aus dem `Devel::Declare`-Hilfsmodul überschreiben - `code_for`. Hier findet

dann die "Installation" der Subroutine im aufrufenden Package statt.

Diese `code_for`-Methode ist in Listing 2 dargestellt, wobei hier erstmal nur "void" berücksichtigt wird. Für die anderen Schlüsselwörter muss `%subs` entsprechend erweitert werden (der vollständige Code ist in den Code-Beispielen zu finden).

Mit `$self->declarator` bekommt man das Schlüsselwort, für das das `code_for` ausgeführt wird - in diesem Fall 'void'. Der anschließend deklarierte Hash dient als "Dispatcher" für die unterschiedlichen Schlüsselwörter. Aus Gründen der Übersicht, wird hier nur die Subroutinen-Referenz für "void"-Funktionen aufgeführt.

Falls ein Name existiert, wird dann die Subroutine erzeugt. Der Name ist im Beispielskript "test". Das wird dann auch der Subroutinenname. Als erstes wird der Subroutinenname bestimmt und danach in `voidsub` (Listing 3) eine anonyme Subroutine erstellt. Mit dem Ausdruck `subname $name => $sub` wird dank `Sub::Name` der aufgerufene Subroutinenname für `caller` sichtbar und heißt nicht nur `__ANON__`.

`Method::Signatures` zeigt, dass man mit `Devel::Declare` noch viel mehr machen kann. Möchte man neben dem neuen Schlüsselwort das Attribut-Handling erweitern, so muss man nicht (nur) `code_for` überschreiben, sondern

```
sub code_for {
    my ($self,$name) = @_;

    my $decl = $self->declarator;

    my %subs = (
        void   => \&voidsub,
    );

    if( defined $name ) {
        my $pkg = $self->get_curstash_name;
        $name   = join( '::', $pkg, $name )
            if index( $name, '::' ) == -1;
        return sub (&) {
            my $code = shift;
            my $sub  =
             $subs{$decl}->($name,$code,@_);
            no strict 'refs';
            *{$name} =
             subname $name => $sub;
            return;
        }
    }
    else {
        return sub (&) { shift };
    }
}
```

Listing 2

```
sub voidsub {
    my ($name,$code) = (shift,shift);

    my $sub  = sub {
        if( defined wantarray ) {
            die "sub $name must be" .
                "called in void context";
        }

        $code->(@_);
    };
}
```

Listing 3

auch `parse_proto`. Je mehr man Perl's Syntax erweitern will, desto weniger hilft das Hilfsmodul und man kann sich direkt mit `Devel::Declare` beschäftigen.

und jetzt mit Devel::Declare...

Die folgenden Abschnitte werden verdeutlichen, wie viel Arbeit das Hilfsmodul erspart. Als erstes brauchen wir wieder ein Modul in dem die Schlüsselwörter definiert werden sollen. Diesmal erben wir nicht von dem Hilfsmodul, sondern binden direkt `Devel::Declare` ein.

```
package CheckContext;

use Devel::Declare;
```

Die Schlüsselwörter müssen ja schon zur Compile-Zeit des Skripts zur Verfügung stehen, deshalb wird die ganze Definition in der `import`-Methode gemacht. Wer nicht weiß, warum da die `import`-Methode genommen wird, sollte sich die `perldoc`-Seite zu `use` anschauen.

```
sub import {
    my ( $class ) = @_;

    my $caller    = caller;

    Devel::Declare->setup_for(
        $caller,
        {
            schluesselwort =>
                    { const => \&parser },
        }
    );

    no strict 'refs';
    *{ $caller . '::schluesselwort' } =
        sub (&) {};
}
```

Listing 4

Was passiert in Listing 4? Da die Schlüsselwörter ja in dem Skript oder Modul vorhanden sein soll, in dem es eingebun-

den wird, benötigen wir den `caller`. Da `caller` in skalarem Kontext aufgerufen wird, wird nur der Package-Namen des Skripts/Moduls zurückgegeben. Danach wird die `setup_for`-Methode aus `Devel::Declare` aufgerufen. Dieser muss als erstes der Package-Name übergeben werden, in den die Schlüsselwörter "installiert" werden sollen. Als nächstes folgt ein anonymer Hash, in dem alle neuen Schlüsselwörter aufgeführt werden sollen.

In unserem Fall sieht das dann so aus:

```
{
     void   => { const => \&parser },
     list   => { const => \&parser },
     single => { const => \&parser },
}
```

Der anonyme Hash, der als Wert zu jedem Schlüsselwort übergeben wird, gibt einen eigenen Parser an - und auf welchem optype das ganze gilt. Wie der Parser aussieht, wird in einem späteren Abschnitt noch gezeigt.

Zum Schluss muss für jedes Schlüsselwort eine leere Subroutine im Package erzeugt werden, die als Platzhalter dient. Das ist notwendig, damit es zu keinem Fehler kommt, wenn der Platzhalter nicht ersetzt wird.

Im Moment ist das eine leere Subroutine, die nichts macht. Aber man beachte den Prototypen `&`. Dadurch sind Aufrufe wie

```
void { print "test" };
```

möglich und der Block hinter "void" wird durch den Prototypen automatisch als Code-Block betrachtet. Prototypen in Perl sind etwas Eigenes und machen nicht unbedingt das was die meisten von Prototypen erwarten. chromatic hat im August einen interessanten Blogpost darüber geschrieben. Der Link ist im delicious-Account von $foo zu finden.

Das `void { print "test" };` ist aber noch nicht das Ergebnis, das wir haben wollen. Am Ende soll ja wie oben beschrieben auch ein Name hinter "void" stehen können. Dazu muss jetzt der eigene Parser geschrieben werden; dieser wird dann zur Compile-Zeit ausgeführt.

Im ganzen Modul verteilt werden zwei Variablen benötigt, zum einen eine Variable in der gespeichert wird, wo sich der Parser gerade in der neuen Syntax befindet. Zum andern die Variable, in der das neue Schlüsselwort gespeichert wird. Deshalb führen wir die globalen Variablen `$offset` und `$declarator` ein.

```
our ($offset,$declarator);
```

Zur Verdeutlichung was passiert, nehmen war als "Syntaxstring" einfach `void test { print "hallo" }` an.

Der Parser muss dann diese Variablen `local`isieren. Danach geht es damit weiter, dass wir uns in der neuen Syntax quasi hinter das neue Schlüsselwort "bewegen". Dazu rufen wir eine Subroutine `skip_declarator` auf, die wie folgt aussieht:

```
 sub skip_declarator {
     $offset +=
Devel::Declare::toke_move_past_token($offset);
 }
```

Anschließend wird der Name der Method aus dem Syntaxstring mit `strip_name` gelöscht. Damit sieht der Syntaxstring so aus: `void { print "hallo" }`. Das ist ja schon der Aufruf der Funktion `void` mit einem Block als Parameter. Jetzt soll aber noch die Überprüfung stattfinden, ob die Funktion im richtigen Kontext aufgerufen wird. Je nachdem was gewünscht wird ("void", "single", "list"), muss der Block etwas abgeändert werden. Das was in den Block eingefügt werden soll, wird in `inject_code` ermittelt und danach mittels `inject_if_block` eingefügt. Danach sieht der Syntaxstring für "void" so aus:

```
void  { if( defined wantarray ){ ... }
print "hallo" }
```

Wichtig: Der String, der eingefügt wird (hier das "`if( defined wantarray ) { ... }`"), darf keine Zeilenumbrüche haben.

Zum Schluss wird noch für den Namen - in unserem Beispiel 'test' - ein Eintrag im GLOB-Slot gemacht, damit man im Programm dann auch wirklich `test()` aufrufen kann. Zu dem Thema mit den Typeglobs gibt es in $foo Nr. 7 - 9 ein dreiteiliges Tutorial von Ferry Bolhár-Nordenkampf.

Die endgültige `parser`-Funktion ist in Listing 5 zu sehen:

```
sub parser {
    local ( $declarator, $offset ) = @_;

    skip_declarator();    # step past 'void'

    # strip out the name 'test', if present
    my $name   = strip_name();
    my $inject = inject_code(
        $declarator, $name
    );

    inject_if_block($inject);

    if (defined $name) {
        $name = join('::',
        Devel::Declare::get_curstash_name(),
        $name)
          unless ($name =~ /::/);
        shadow(sub (&) {
            no strict 'refs';
            *{$name} = shift;
        });
    } else {
        shadow(sub (&) { shift });
    }
}
```

Listing 5

Mit dem Modul `B::Deparse` (siehe auch $foo Nr. 9), kann man sich dann anschauen, was aus dem ursprünglichen Syntaxstring geworden ist (siehe Listing 6).

```
$ perl -MO=Deparse context.pl
use CheckContext;
use warnings;
use strict 'refs';
void sub {
    if (defined wantarray) {
        die 'sub test must be called' .
            'in void context';
    }
    print 'test';
}
;
test();
context.pl syntax OK
```

Listing 6

Die allgemeinen Methoden wie `strip_name`, `skipspace` etc. sind auch im Modul `Devel::Declare::Context::Simple` zu finden. Durch dessen Verwendung lässt sich wieder Arbeit einsparen.

Zusammenfassend kann man sagen, dass `Devel::Declare` ein sehr nützliches Modul ist und z.B. Methodensignaturen ermöglicht hat. In den Codebeispielen, die zum Download bereitstehen, ist das komplette Beispiel aus diesem Artikel zu finden. Die Beispiele aus dem CPAN und aus diesem Artikel zeigen, dass man sich durch `Devel::Declare` Perl selbst gut erweitern kann.

Zu beachten ist allerdings, dass das `die` nicht zur Compile-Zeit ausgeführt wird, sondern erst zur Laufzeit.

Renée Bäcker

WxPerl Tutorial - Teil 1 : Die Grundlagen

Einleitung und Übersicht

Perl hält sich oft im Hintergrund. Bescheiden, zuverlässig und oft unerkannt bereiten Perl-Programme weltweit Daten auf, erstellen Berichte, Makefiles, HTML, u.v.m.. Doch mit vielen Benutzern ist Kommunikation nur über die graphische Oberfläche möglich. Und eigentlich eignet sich eine flexible und vielseitige Sprache wie Perl sehr gut um kleine aber auch große Applikationen zu schreiben.

Dieser Text leitet ein mehrteiliges Tutorium ein, welches mit WxPerl einen Weg zeigt, beides zu tun. Es benötigt neben einem installierten WxPerl etwas Grundkenntnisse über Referenzen, Module und Objekte. Damit es auch nachträglich als Nachschlagewerk benutzbar bleibt, ist der Inhalt thematisch geordnet. Dieser erste Teil deckt Installation, etwas Hintergrundwissen und generelle API-Benutzung ab und wird nur bis zum *"Hallo Welt"* führen. Ein Tutorium das ähnliche Inhalte wiedergibt, aber nicht identisch ist, entsteht seit längerem unter: *http://wiki.perl-community.de/cgi-bin/foswiki/view/Wissensbasis/WxPerlTutorial*. Eine Mitarbeit dort ist stets willkommen.

Wann ist Wx die beste Wahl?

Neben WxPerl gibt es noch `Tk`, `Qt`, `Gtk`, `FLTK`, `Prima` und vieles mehr. Wenn eine *ASCI*-Ausgabe ausreicht, kann `Curses` oder `Text::Graphics` eine pragmatische Lösung sein, aber wenn Pixelland betreten werden soll, scheiden diese Alternativen aus.

`Win32::GUI`, `X11::Xlib`, `Mac::Carbon` und `Mac::CocoaDialog` sind ideal wenn fest steht, dass die Grenzen einer Plattform nicht überschritten werden und/oder Details der plattformspezifischen API benötigt werden. Alle diese Pakete sind kleiner (in MB) als WxPerl, aber `Wx` unterstützt ebenfalls viele plattformspezifische Funktionen, zumindest ist mir nichts in `Mac::CocoaDialog` aufgefallen, was `Wx` nicht auch könnte. In dieser Liste fehlt `Gtk` absichtlich, da es verlassen und veraltet ist (letztes Release 2003, bündelt eine Version von 2001). Das aber glücklicherweise gut gepflegte `Gtk2` gehört auch nicht ganz in diese Kategorie, da es zwar nativ auf *nixen ist, aber auch andernorts Knöpfe, Leisten und Fenster (was mit *Widgets* zusammengefasst wird) zur Schau stellen kann. Womit wir nun wirklich bei den Modulen wären, die `Wx` ähneln, da sie *Widgets* betriebssystemunabhängig und ohne Hilfe eines Browsers zeichnen können. Auch hier verlassen Kandidaten (`Qt`) das Feld bereits bei Start, da sich dort seit 6 Jahren nichts bewegt, im CPAN sogar seit 12. `FLTK` ist ebenfalls nicht installierbar und ruht bereits seit 9 Jahren, weswegen nur ein genauerer Vergleich mit `Tk`, `Gtk2` und `Prima` lohnt.

`Tk` war bisher die mit Abstand meistverwendete GUI-Bibliothek für Perl, auch wenn sie ursprünglich für TCL entwickelt wurde. Sie zeichnet sich daher durch Stabilität, gute Dokumentation, viele Erweiterungen und eine einfache API aus. Um kleineren Scripten, die keine Threads beinhalten, schnell ein Gewand zu geben ist es empfehlenswert. Nachteile gegenüber WxPerl liegen im kleineren Funktionsumfang, schlechtere Handhabung komplexer Strukturen, größerem Ressourcenverbrauch, konstant eigenwilligem Aussehen und dass das Projekt seit dem Abschied von Nick Ing-Simmons im Bugfixmodus läuft.

Eine recht schöne API und (fast grüne) Testmatrix hat `Gtk2` (Binding für *Gtk+*). Die Entwicklung ist sehr aktiv und unter Unixderivaten besteht optisch kein Unterschied zu `Wx`, das hier meist *Gtk+* verwendet (Es gibt auch Ports für *X11* und *Motif*). `Wx` tut sich dadurch hervor, auf jeder Plattform

nativ (wie für dieses System entwickelt) auszusehen. Denn Wx-Programme sind tatsächlich nativ (mit einer kleinen Zwischenschicht), da `Wx` seine Befehle an die jeweils native Schnittstelle delegiert. Damit kann es relativ schnell und sparsam und solide sein. Andere Bibliotheken wie `Swing` oder `Tk`, die mit *Skins* das Aussehen eines Betriebssystems nachahmen, sind dort weniger sparsam. Höchstens `SWT` geht einen vergleichbaren Weg, ist aber im Gegensatz zu Swing für Perl derzeit unerreichbar. Jedenfalls ist auch `Gtk2` empfehlenswert, wenn es nicht auch unbedingt nach *Win* oder *Mac* aussehen muss, oder einzelne Fähigkeiten dringend gebraucht werden, die *Gtk+* übersteigen, wie spezielle, komplexe Widgets.

Interessant ist auch `Prima`, welches (beinah) nur in Perl verfasst ist und daher dem Nutzer neben einem sehr schlichten Aussehen einige Freiheiten mehr bietet, als andere GUI-Bibliotheken. Natürlich benötigt es auch einen kleinen kompilierten Kern, aber im Gegensatz zu `Wx` und auch teilweise `Tk` hat man das Gefühl, dass die API von einem Perl-Programmierer (Dmitry Karasik) entwickelt wurde. Wen spartanische Graphik nicht stört, sondern dem simple APIs und einfache Installierbarkeit wichtig sind, sollte einen Blick auf `Prima` richten.

Es war einmal Wx

Die Entstehung von `Wx`, genauer gesagt *WxWidgets*, hat gar nichts mit Perl zu tun. 1992 war Julian Smart im schottischen Edinborough damit beschäftigt für das AI-Institut der Universität ein C++-Programm zu schreiben, das sowohl unter einem X-basierenden Unix als auch Windows laufen sollte, genauer XView und MFC 1.0. Aus dem W wie Windows und dem X entstand der Name der heute eher als W(idgets) X(cross-platform) gelesen werden möchte (Julians Favorit nach einem Wettbewerb). Diesen Wettbewerb gab es kurz nach der Umbenennung von WxWindows in WxWidgets, zu der 2004 ein freundlicher Brief von Microsoft mit beiliegendem Check geführt hat. Über die Jahre bekam WxWidgets viele weitere Ports (`Gtk/Motif/Gtk+/Mac(Carbon und Cocoa)/OS2/MGL/WinCE`) und Wrapper (`Basic/C/D/Eiffel/Haskell/Java/Javascript/Perl/Python/PHP/Ruby/.NET`), für Programmierer die von einer anderen Sprache aus WxWidgets benutzen möchten. Die Wx-Lizenz ist eine leicht geänderte LGPL, womit *Wx* problemlos und ohne Gebühren für freie und kommerzielle Projekte verwendet werden kann. Aktuell ist die die stabile Version 2.8.10, aber auch die am 8. September erschienene "Entwicklerversion" 2.9.0 ist nach 6 Releasekandidaten recht ausgereift. Aber die vielleicht wichtigste "Eigenschaft" von WxWidgets ist die hilfsbereite Gemeinschaft der Entwickler und Benutzer, die unter anderem sehr aktive und freundliche Mailinglisten wie die *wx-users@googlegroups.com* unterhält.

Diesseits des XS

Auf der der Perl-Seite dagegen ist Mattia Barbon seit November 2000 fast allein damit beschäftigt eine Verbindung zu *WxWidgets* zu knüpfen. Er ist definitiv einer der zu wenig besungenen Helden der Szene und das Perl-Modul `Wx` ist seit der Version 0.57 wirklich nur ein XS-Binding zu einer Sammlung kompilierter Wx-C++-Bibliotheken, die selbst während der Laufzeit gewechselt werden können. Sind solche nicht auf dem Rechner vorhanden, so wird das von `Alien::wxWidgets` während der Installation von `Wx` nachgeholt. `Alien::wxWidgets` ist nicht mit den "normalen" C++-WxWidgets identisch, da es auch viele Zusätze aus dem Verzeichnis "contribs" (Erweiterungen wie z.B. Scintilla aka Wx::STC) beinhaltet. Auch wurde auf der Wx-Seite die API an vielen Stellen vereinfacht, z.B. mit Arrays anstatt Objekten als Parameter. Oder es wurde für eine Carp-Funktion gesorgt mit der Programme ihren letzten Krächzer in einem WxFenster abgeben können. Viele WxWidgets-Bibliotheken fehlen in WxPerl da die Klassen für Array, String, UTF und Regex keine Verbesserung darstellen und auch für FTP, TCP, Zip und DateTime kennt das CPAN mächtigere Lösungen. Bildbearbeitung und Logging sind auch Fälle in denen CPAN mehr zu bieten hat, aber wenn man eh einmal `Wx` benutzt, könnte man auch auf Abhängigkeiten wie `GD`, `Log::Log4Perl` oder einem XML-Parser verzichten. Auch wenn Wx umfangreich, stabil und ausgereift ist, wird es erstaunlich selten verwendet, was vielleicht auch daran liegt dass es immer wieder ernsthafte Probleme mit der Installation dieser komplexen Software gibt.

Installation

Im Allgemeinen reicht der bekannte Spruch

```
cpan> install Wx
```

aber was tun, wenn es Fehlermeldungen gibt oder die CPAN-shell nicht benutzt werden kann/darf?

Windows-Benutzer sollten sowieso das StrawberryPerl bevorzugen, das nicht nur bereits `Wx` beinhaltet sondern auch meist problemlos kompilieren kann. Und wenn es dennoch hakt so hilft oft der der Zyklus: `look Alien::wxWidgets`, `perl Makefile.PL`, `dmake`, `dmake test`, `dmake install`, `exit` und dito für `Wx`. Wer unbedingt ActivPerl benutzen will oder muss, hilft sich in der ppm-Shell mit

```
PPM> rep add wxperl \
     http://www.wxperl.co.uk/repository
PPM> install Alien-wxWidgets
PPM> install Wx
```

wobei seit ppm4 das Repository auch per Dialog (Edit -> Preferences) zugefügt werden kann und die Installation per Knopfdruck passiert.

Unter Debian kann es helfen alle Wx und WxPerl Pakete zu deinstallieren und dafür die "-dev"-Pakete der Version 2.8.8 zu nehmen. Bei Ubuntu

```
 $ apt-get install libwxgtk2.8-0 \
   libwxgtk2.8-dev libwxbase2.8-dev \
   libwxbase2.8-0
$ apt-get install libmodule-build-perl \
   libextutils-cbuilder-perl \
   libmodule-pluggable-perl g++
```

Ist es älter als 8.10 füge zu:

```
$ apt-get install libgtk2.0-dev
```

Dann müsste letztlich folgendes funktionieren:

```
% sudo cpan ExtUtils::CBuilder \
 Alien::wxWidgets Wx
```

wobei hier bei älteren Versionen

```
Do you want to fetch and build wxWidgets
   from sources? [no ]
```

unbedingt mit *yes* beantworten. In besonders hartnäckigen Fällen muss `libextutils-cbuilder-perl` mit Debian synchronisiert werden.

Für Fedora Core 8 und 9 reicht:

```
# rpm -ivh  perl-Wx perl-Alien-wxWidgets
```

Auch Mandriva kennt damit wenig Probleme:

```
# urpmi perl-Alien-wxWidgets
# urpmi perl-Wx
```

In openSuSE muss das *packman* Repository zugefügt werden und von diesem das Paket *wxWidgets-devel* bezogen werden. Unter Gentoo kann helfen:

```
emerge dev-perl/Alien-wxWidgets \
  dev-perl/wxperl
```

Wie bei älteren Ubuntus, so sollte man unter OSX auch `Alien::wxWidgets` immer von den Quellen aus installieren. OSX 10.4 und 10.5 besitzen bereits WxPerl, aber sehr veraltete Versionen. Doch zum Glück kennt man dort geeignete Bordmittel, die beidermaßen funktionieren:

```
fink install wxmac28
port install wxWidgets
```

Wenn all dies nicht hilft, ist es ein Fehler im Modul, den Mattia meist so schnell er kann behebt, wenn man es unter *wxperl-users@perl.org* meldet. Fehler treten immer noch auf, da die API riesig ist, Mattia aber nur eine Person ist. Auch ändern sich manchmal Details, jedoch Teile des Wx-Quellcodes selber werden per Script erzeugt.

Dokumentation

Die Dokumentation von WxPerl ist recht gut. Ich empfehle die offizielle WxWidget-Seite unter *http://wxwidgets.org/docs/* zu besuchen um die Docs dort als HTML zu lesen oder sie sich herunterzuladen. Sie beinhaltet alphabetische und thematische Übersichten und alle Abweichungen zwischen *WxPerl* und *WxWidgets* sind in den Methodenbeschreibungen eingetragen. Diese und sämtliche weitere Dokumentationen sind unter *http://wxperl.sf.net/documentation.html* aufgelistet. All dies ist selbstverständlich in englischer Sprache verfasst. Die einzige mir bekannte deutsche Quelle ist das bereits erwähnte Tutorial unter: *http://wiki.perl-community.de/cgi-bin/foswiki/view/Wissensbasis/WxPerlTutorial*.

Die Wx-Regeln

Bevor man beginnt mit Wx zu programmieren, sollte man zur besseren Orientierung einige Grundregeln der API kennen. Ein wenig C++-Kenntnis schadet dabei nicht.

Erste Regel: Fast alles ist ein Objekt. Mit wenigen Ausnahmen wie Benachrichtigungsboxen oder der Zwischenablage wird jedes Widget mit `Wx::Ding->new('para', 'meter');` erstellt und eine Wx-Applikation ist immer auch eine Hierarchie von Objekten. Diese Hierarchie gut zu kennen ist entscheidend um zu verstehen wie genau *Wx* bei Layoutfragen oder Ereignissen vorgeht.

Zweite Regel: Methoden haben positionale Parameter. Im Unterschied zu `Tk`, `Gtk` oder `Prima`, wo Daten als benannte Parameter in beliebiger Reihenfolge übergeben werden, (wie in einem Hash) erwarten in *Wx* die Methoden ihre Parameter in einer vorgegebenen Reihenfolge, was mehr oder weniger dem Schema: Elternwidget, ID, Input, Position, Stil, etc. folgt. Die Reihenfolge entspricht der Wichtigkeit und nicht benötigte Parameter können weggelassen werden. Genauer: Die ersten 1 bis 3 (variiert) Parameter müssen immer angegeben werden und nach dem letzten, wichtigen Parameter kann aufgehört werden. Wenn ein Parameter zwischendurch uninteressant ist, kann er mit "-1" auf default gesetzt werden.

Auch wichtig: Eigenschaften eines Widgets werden mit Konstanten gesetzt. Sie entsprechen alle einem Zahlenwert der auch eingefügt werden könnte. Aber es ist sauberer, alle benötigten Konstanten mit dem `use Wx qw(.....);` anzumelden und im "Stil"(style)-Parameter zu verwenden. Sollen für ein Widget mehrere Eigenschaften gesetzt werden, so können sie mit einem *bitweisen und* verknüpft werden wie z.B. `wxCAPTION | wxCLOSE_BOX`, was ein Hauptfenster mit Titelleiste und Knopf zum Schließen erzeugen würde. Konstanten können auch in Gruppen angemeldet werden wie `use Wx qw(:window :textctrl :image);`. Während ein Programm im Aufbau ist empfiehlt sich die Gruppe ":everything" zu verwenden. Damit erübrigt sich die Sorge, dass eine fehlende Konstante einen Error erzeugt. Wenn das Programm fertig ist, spart es Arbeitsspeicher ":everything" gegen die eingesetzten Konstanten auszutauschen. Ereignisse (Events) sind auch Konstanten, die aber über `Wx::Event qw(...);` angemeldet werden.

Wichtig zu beachten sind auch die Rückgabewerte mancher Routinen, da sie darüber entscheiden, ob der gewünschte Effekt überhaupt ausgelöst wird. Entsprechend gibt es auch nur selten eine Methode `$widget->hide;`, aber immer ein `$widget->show(0);`, welches das Widget unsichtbar macht.

Ereignisbasierende Programmierung

Eine letzte Sache, die sich vor allem Programmierer vergegenwärtigen sollten, die bisher nie eine GUI eingesetzt haben: Ihr gebt zu einem guten Teil die Kontrolle darüber ab, was geschehen wird. Das Programm beschreibt was geschieht wenn jener Knopf gedrückt wird, aber wann das geschieht liegt in der Hand des Benutzers. Die Zeile `NameDerHauptklasse->new->MainLoop;` startet eine Maschinerie. Der Quelltext endet an dieser Stelle und was bei Abbruch geschieht kann irgendwo tief in einer Unterklasse versteckt sein oder erst während des Ablaufs festgelegt werden. Gerade die Benutzung von `eval` braucht hier eine klare Sicht auf mögliche [nicht-]Inhalte von Variablen zu bestimmten Ausführungszeiten.

Olah Kosmos

Dies wird aber im Einzelnen Teil 2 dieses Tutoriums zeigen, wenn es um Dialoge und Ereignisse geht. Für dieses Mal soll ein kleines Beispielprogramm reichen. `strict` und `warnings` sollten klar sein und `use Wx;` auch, da wir tatsächlich WxPerl benutzen wollen. Das Paket *ServusWelt* wird durch den Befehl `use base qw(Wx::App);` zum Hauptprogramm, da es dadurch von `Wx::App` alle Methoden erbt um ein vollständiges WxPerl-Programm zu sein. Lediglich die Methode `OnInit` definieren wir hier neu, um zu bestimmen was bei Programmstart ausgeführt wird. Da das Objekt des Hauptprogramms der erste Parameter ist, mit dem vom "WxGetriebe" `OnInit` aufgerufen wird, kann sie in Zeile 9 mit `shift` bezogen werden. In Zeile 10 wird das Hauptfenster erstellt. Da es kein übergeordnetes Widget hat, bleibt für den ersten Parameter nur `undef`, ID's sind jetzt auch noch unwichtig also wird der zweite Parameter auf "-1" gesetzt. Parameter 3 ist die Überschrift des Fensters, Nr. 4 die Position die per default mittig ist und

Parameter 5 die Größe. Nach dem gleichen Schema erzeugt die nächste Zeile ein Panel, eine Fläche auf der man Widgets platzieren kann. Der nackte Hintergrund eines Wx::Frame ist absichtlich optisch nicht dazu geeignet, darauf Knöpfe zu legen, da deren Hintergrundfarbe transparent ist und damit das dunkle Grau des Hauptfensters annehmen würde. Da das Panel weder Position noch Größe kennt, nimmt es automatisch den ganzen Hintergrund des Hauptfensters ein. Durch Zeile 10 wird auf das Panel noch ein Text gelegt. `$frame->Show(1);` lässt das Hauptfenster sichtbar werden und `$app->SetTopWindow($frame);` stellt sicher, dass es nicht verdeckt wird. Das abschließende `1;` ist der erwähnte Rückgabewert, der uns versichert das die Applikation wirklich gestartet wird. Das Prinzip kennt jeder der ein Perlmodul geschrieben hat, welches auch immer mit einem positiven Wert endet, wofür natürlich auch phantasievollere Inhalte möglich sind. Die letzte Zeile startet unser bis dahin beschriebenes Wx-Programm.

```
use strict;
use warnings;

package ServusWelt;
use Wx;
use base qw(Wx::App);

sub OnInit {
   my $app = shift;
   my $frame = Wx::Frame->new(
       undef, -1, 'Mein erstes Programm',
       [-1,-1], [250,100],
   );
   my $panel = Wx::Panel->new( $frame, -1);
   my $label = Wx::StaticText->new(
       $panel,      -1,
       'Grreeeting Earthlings!',
       [65,20]
   );

   # Fenster zeichnen
   $frame->Show(1);
   # Fenster als oberstes bestimmen
   $app->SetTopWindow($frame);
   # eins nicht vergessen:
   1;
}

package main;
# Programminstanz erzeugen und starten
ServusWelt->new->MainLoop;
```

Herbert Breunung

Hier könnte Ihre Werbung stehen!

Interesse?

Email: werbung@foo-magazin.de
Internet: http://www.foo-magazin.de (hier finden Sie die aktuellen Mediadaten)

Perl Scopes Tutorial - Teil 2

Im letzten Artikel wurde statisches und dynamisches Scoping beschrieben. Heute sehen wir uns an, welche Auswirkungen Scoping (speziell statisches Scoping) auf lexikalische Variable und deren Geltungsbereiche hat.

Lokale (lexikalische) Variable

Lokale Variable werden innerhalb von Scopes deklariert und sind auch nur darin ansprechbar - können aber durchaus auch außerhalb davon existieren, wie im Abschnitt über persistente Variable noch gezeigt wird.

Lokale Variable werden häufig auch als *lexikalisch* oder *privat* bezeichnet. Der Grund für diese Wortklauberei liegt darin, dass man eine Verwechslung mit dem `local` Befehl (der seinerseits *globale* Variable anspricht und nur deren *Werte* lokalisiert) vermeiden möchte. Daher wird auch in diesem Text ab jetzt nur mehr von **lexikalischen Variablen** (als Gegenteil zu globalen Variablen) gesprochen.

Lexikalische Variable müssen *deklariert* werden; ohne Deklaration nimmt der Compiler beim ersten Antreffen eines neuen Variablenamens an, dass eine globale Variable angesprochen werden soll. Die Deklaration geschieht mit dem Schlüsselwort `my`:

```
my $hugo;
```

Optional kann man der Variable beim Deklarieren auch einen Wert zuweisen. Diese Zuweisung ist aber eine *Anweisung* und wird daher erst zur Laufzeit ausgeführt.

Im obigen Beispiel ist die Variable `$hugo` nun für diesen Scope (bzw. für die gesamte Datei, falls kein anderer Scope aktiv ist) deklariert. Da Scopes auch verschachtelt werden können, wird folgendes Beispiel funktionieren:

```
# Aeussere lexikalische Variable
my $hugo = 1;
{
  print $hugo;     # Gibt 1 aus
  # Innere lexikalische Variable
  my $hugo = 2;
  print $hugo;     # Gibt 2 aus
}
print $hugo;       # Gibt 1 aus
```

Hier wird zunächst eine Variable deklariert, dann wird ein neuer Scope angelegt und in diesem Scope der Wert der Variable ausgegeben. Da der Compiler, der ja Zeile für Zeile von oben nach unten einliest, zu diesem Zeitpunkt nur die äußere Variable kennt, wird sich der erste `print` Befehl genau auf diese Variable beziehen. Danach wird innerhalb des Scopes eine zweite, gleichnamige Variable deklariert. Ab diesem Zeitpunkt beziehen sich alle Angaben von `$hugo` nur mehr auf die innere Variable; die äußere ist in diesem Scope *nicht mehr ansprechbar*. Sie existiert aber natürlich noch weiterhin, denn sonst könnte, wie im Beispiel gezeigt, nach dem Verlassen des Scopes deren Wert ja nicht mehr ausgegeben werden.

Beachte, dass das äußere `$hugo` tatsächlich innerhalb des Scopes nicht mehr ansprechbar ist. Im Gegensatz zu einer gleichnamigen globalen Variable, die man auch im Scope durch Vorsetzen des Packagenamens ansprechen könnte, ist dies hier nicht mehr möglich, auch wenn man z.B. versuchen würde, die innere Variable auf `undef` zu setzen. Das würde nur ihren Wert löschen, aber die äußere Variable bliebe dennoch weiterhin verborgen: die Laufzeitanweisung hätte auf das statische Scoping, das ja schon beim Kompilieren abgelaufen ist, keinen Einfluss mehr. Scopes werden immer von innen nach außen durchsucht; wird der Compiler in einem Scope fündig, wird in den äußeren Scopes nicht mehr weitergesucht.

Aus diesem Beispiel lassen sich zwei Erkenntnisse herauslesen:

• Ein innerer Scope "erbt" die Umgebung eines äußeren, d.h. ihn einschließenden Scopes. Alle lexikalischen Variable, die im äußeren Scope deklariert wurden, sind auch im inneren Scope sichtbar. Daher sind alle lexikalischen Variable, die am Beginn einer Datei deklariert werden, innerhalb der gesamten Datei (und somit auch innerhalb von in dieser Datei definierten Funktionen) ansprechbar.

• Wird aber einmal in einem inneren Scope eine lexikalische Variable deklariert, so kann die gleichnamige Variable eines äußeren Scopes nicht mehr angesprochen werden [1].

Lexikalische Variable in Schleifen

Eine Deklaration wie

```
my $test;
```

wird vom Compiler als "lege eine neue lexikalische Variable an" interpretiert. Während dies im Allgemeinen genau einmal beim Kompilieren geschieht, gibt es Fälle, wo eine solche Deklaration auch zur Laufzeit Auswirkung hat: wenn sie in einer Schleife (oder im Schleifenkopf) aufscheint. In folgendem Code

```
my $x;
for (1..5) {
  $x++;
  print $x; # Gibt 1 2 3 4 5 aus
}
```

wird eine lexikalische Variable deklariert und innerhalb der Schleife inkrementiert. Da die Variable außerhalb der Schleife deklariert wurde, wird innerhalb des Scopes immer *dieselbe Variable* angesprochen und diese daher fünfmal inkrementiert. Hingegen wird in

```
for (1..5) {
  my $x++;
  print $x;  # Gibt 1 1 1 1 1 aus
}
```

die Variable `$x` innerhalb der Schleife immer wieder *neu angelegt* (und mit Null initialisiert), wodurch sie nur jeweils einmal inkrementiert und daher fünfmal dasselbe Resultat ausgegeben wird. Das führt manchmal zu Missverständnissen, besonders, wenn innerhalb und außerhalb der Schleife zweimal dieselbe gleichnamige Variable verwendet wird:

```
my $x = 'x';
while ($j++ < 5)
{
  # Kopiert aeusseres $x in inneres $x
  my $x = $x;
  $x .= 'y'; # Inneres $x
  print $x;  # Gibt fuenfmal 'xy' aus
}
```

Hier wird nicht etwa `xy`, `xyy`, `xyyy`, usw. ausgegeben, sondern fünfmal hintereinander `xy`. Der Grund ist derselbe wie im vorigen Bespiel: die innere Variable `$x` wird innerhalb der `while`-Schleife deklariert und daher bei jedem Schleifendurchlauf neu angelegt - tatsächlich wird nicht eine Variable fünfmal, sondern es werden **fünf neu angelegte Variable** - eine pro Durchlauf - verwendet. Daher ergibt sich nach der Verknüpfung mit dem Stringliteral jedesmal derselbe Wert.

Beachte, dass der Compiler auf eine neu deklarierte Variable erst in der der Deklaration folgenden Anweisung zugreift. In

```
my $x = $x;
```

ist daher die Variable auf der rechten Seite (wie auch in obigem Beispiel) noch die im äußeren Scope deklarierte (oder die globale Paketvariable). Erst in der nächsten Anweisung wird das neu deklarierte lexikalische `$x` angesprochen. Daher werden Konstrukte wie

```
print my $x while $x = <STDIN>;
print $x while my $x = <STDIN>;
```

nicht wie erwartet funktionieren - das `$x` ohne `my` wird noch als global angenommen, die Befehle sprechen daher zwei verschiedene `$x` an (bei der Verwendung von `strict 'vars'` merkt man das aber schnell).

Es ist wichtig, sich vor Augen zu halten, dass `my` - ansonsten eine reine Compilerdirektive - in Schleifen und Funktionen auch einen Laufzeiteffekt hat: es bewirkt bei jedem Schleifendurchlauf oder Funktionsaufruf das **Neuanlegen** der dort deklarierten Variablen. Oft ist dies erst auf den zweiten Blick ersichtlich:

```
my $str .= "$_ = $ENV{$_}\n"
  foreach keys %ENV;
print $str;
```

Dieser Code liest die Namen und Werte aller Environment-Variable - um anschließend *nichts* auszugeben. Warum? Weil `$str` für jeden Schleifendurchlauf neu angelegt wird, *bevor*

mit der Abarbeitung der Schleife begonnen bzw. festgestellt wird, dass alle Elemente von `%ENV` durchlaufen wurden und die Schleife beendet werden kann. Das von der `print`-Anweisung ausgegebene `$str` ist somit die letzte, unmittelbar vor dem Schleifenabbruch angelegte Version, die noch keinen Inhalt besitzt. Ähnlich verhält es sich mit Code wie

```
push my @incs, $_ while <INPUT>;
```

Auch hier wird am Beginn jedes Durchlaufes der `while` Schleife das Array `@incs` neu angelegt. Daher ist das Array nach Erreichen des Schleifenendes leer.

Hier müssen also die Variablen lexikalisch *vor* der Schleife deklariert oder es sollte - ab Perl 5.10 - besser `state` verwendet werden.

Lexikalische Variable in foreach Schleifen

Die Handhabung lexikalischer Variable im Kopf von `foreach`-Schleifen verdient besondere Erwähnung. Gegeben sei folgender Code:

```
my $found;
foreach $found (@args) {
  last if $found eq 'XXX';
}
print '"XXX" '
  . ($found eq 'XXX' ? 'wurde' : 'nicht')
  . " gefunden!\n";
```

Dieser Code sucht in einem Array nach Elementen mit dem String `XXX` und bricht ab, sobald er ein solches Element gefunden hat. Ist die in der Schleife zum Vergleich verwendete Variable nach Abarbeitung der Schleife entsprechend gesetzt, wurde der String gefunden, ansonsten nicht. Das wird nun abgefragt und es wird eine entsprechende Meldung ausgegeben. Soviel zur Theorie.

Die Praxis sieht anders aus: im obigen Code wird *immer* `"XXX" wurde nicht gefunden!` ausgegeben, auch wenn ein Element den gesuchten String enthält! Warum?

Der Grund hierfür liegt in der Arbeitsweise von `foreach`: die angegebene Schleifenvariable (oder `$_`, falls keine angegeben wurde), wird zu einem **Alias** auf das jeweilige Arrayelement; d.h. die Schleifenvariable und das gerade aktuelle Arrayelement zeigen während der Abarbeitung des Codes im Schleifenkörper auf *denselben* Wert. Dadurch wird es möglich, innerhalb der Schleife Arrayelemente durch Modifizieren der Schleifenvariable zu ändern - eine der Stärken von `foreach` [2].

Damit dieses Aliasing möglich wird, wird der Wert der Schleifenvariable vor dem Abarbeiten der Schleife *lokalisiert* und nach dem Schleifenende wieder hergestellt. Daher hat die Variable hinterher keinen Bezug mehr zu den Werten, die sie innerhalb der Schleife hatte. Hinter den Kulissen läuft also ein mit

```
my local $found;
foreach $found (@args)
```

vergleichbarer Code ab. Diese Lokalisierung wird jedoch - und das ist das Bemerkenswerte - **auch mit lexikalischen Variablen** durchgeführt; etwas, das in Perl ansonsten nicht vorgesehen ist und vom Compiler auch, wie in diesem Code

```
my $x = 1;
{
  local $x = 2;
}
```

durch die Ausgabe

```
Can't localize lexical variable $x
```

kategorisch abgelehnt wird. In `foreach`-Schleifen wird jedoch auch mit *lexikalischen* Schleifenvariablen implizit *dynamisches* Scoping durchgeführt.

Abhilfe lässt sich dadurch schaffen, dass man die Schleifenvariable in eine andere Variable kopiert, die keinem solchen dynamischen Scoping unterworfen ist:

```
my $found1;
foreach my $found (@args) {
  $found1 = $found, last if $found eq 'XXX';
}
print '"XXX "'
  . ($found1 eq 'XXX' ? 'wurde' : 'nicht')
  . " gefunden!\n";
```

In diesem Beispiel wird auch die Schleifenvariable dezidiert als lexikalisch deklariert, um optisch hervorzuheben, dass ihr Geltungsbereich auf die Schleife beschränkt ist. Dadurch erübrigt sich jedes dynamische Scoping, da die Variable außerhalb der Schleife ohnehin nicht ansprechbar ist.

Lexikalische Variable in verschachtelten Funktionen

Einige bemerkenswerte Situationen können entstehen, wenn statisches Scoping auf Funktionsdeklarationen angewendet wird, d.h. Funktionen innerhalb von verschachtelten Scopes definiert werden, die auf lexikalische Variable aus anderen Scopes zugreifen. Das führt dann gelegentlich zu so interessanten Hinweisen wie

```
Variable "$hugo" will not stay shared

Variable "$hugo" may be unavailable
```

deren Zustandekommen nun näher beschrieben wird.

Weiter vorne wurde gesagt, dass eine außerhalb einer Funktion deklarierte lexikalische Variable innerhalb dieser Funktion nur sichtbar ist, wenn die Funktion und die Variable im selben Scope deklariert wurden. Folgender Code wäre daher denkbar:

```
use warnings;
use strict;

sub alpha {
  while (my $input = <STDIN>) {
    sub beta {
      return $input * $input;
    }
    print beta() . "\n";
  }
}
alpha();
```

Da die Funktion `beta()` im Scope der `alpha()`-Funktion (bzw. im Scope der in dieser Funktion enthaltenen `while`-Schleife) definiert wurde, sollte man annehmen, dass sie die in `alpha()` deklarierte lexikalische Variable `$input` ansprechen kann. Dabei wird aber ein wesentlicher Punkt übersehen.

Lässt man diesen Code laufen, erhält man zunächst die erste der obigen Warnungsmeldungen angezeigt. Danach wird man mit zunehmendem Frust feststellen, dass nur das erste Mal ein richtiges Ergebnis ausgewiesen wird (das Quadrat des eingegebenen numerischen Wertes), danach wird stets derselbe, erste Wert zurückgeliefert, unabhängig von der Eingabe. Bricht man das Skript ab und startet es erneut, passiert wieder dasselbe: der erste Wert wird richtig berechnet, danach wird nur mehr dieser erste Wert ausgegeben. Warum?

Wie im Abschnitt über statisches Scoping erläutert, legt der Compiler beim Antreffen der `my` Direktive im Scratchpad des gerade aktiven Scopes - hier des Scopes der `while`-Schleife - einen Eintrag für die Variable `$input` an. Da die Funktion `beta()` ebenfalls in diesem Scope deklariert wird, wird auch dort die Variable beim Kompilieren von `beta()` gefunden und daher der Code für den Zugriff auf das `$input` von `alpha()` generiert - `beta()` wird also an das zum Zeitpunkt des Kompilierens vorhandene `$input` *gebunden* (statisches Scoping). Zur Laufzeit wird ein Wert in die Variable eingelesen und anschließend die Funktion `beta()` aufgerufen, die auf die Variable zugreift und deren Quadrat zurückgibt, welches vom `print` Befehl ausgegeben wird. Danach beginnt die Ausführung des Schleifencodes erneut.

Nun kommt es jedoch zu dem weiter oben im Abschnitt über lexikalische Variable in Schleifen beschriebenen Verhalten: eine innerhalb einer Schleife als lexikalisch deklarierte Variable wird bei jedem Schleifendurchlauf *neu angelegt*. Das `$input` des vorigen Schleifendurchlaufes wird also verworfen und es wird ein neues `$input` eingerichtet. Was aber passiert mit der Variable in der Funktion `beta()`? Diese ist durch die Regeln des statischen Scopings an die beim Kompilieren angelegte Version von `$input` gebunden; sie kann auf die zur Laufzeit neu angelegte Variable nicht zugreifen. Die Folge davon ist, dass es ab nun **zwei** gleichnamige, lexikalische Variable `$input` gibt: die bei jedem Schleifendurchlauf neu angelegte Version und die statisch an die Funktion `beta()` gebundene Version - und genauso ist die `"will not stay shared"` Meldung zu interpretieren: die beim Kompilieren noch von beiden Codeteilen ansprechbare Variable wird zur Laufzeit *nicht mehr gemeinsam* verwendet werden können.

Da das `$input` innerhalb von `beta()` durch die Funktion selbst nicht geändert wird, behält es den Wert, den es nach dem ersten Schleifendurchlauf hatte - daher wird immer dasselbe Resultat zurückgegeben [3].

Zur Lösung dieses Problems gibt es - neben der Übergabe von `$input` als Argument zu `beta()` - zwei Ansätze:

• Auf die Definition benannter Funktionen innerhalb anderer Funktionen zu verzichten. Da in Perl Funktionen *immer global* und somit ihre Definitionen von Scopes unabhängig sind, macht es keinen Sinn (und wirkt eher verwirrend), sie zu verschachteln. Das obige Beispiel kann man besser schreiben, wie in Listing 1 dargestellt. Natürlich muss nun die lexikalische Variable außerhalb von `alpha()` (im übergeordneten Scope) deklariert werden, damit beide Funktionen darauf zugreifen können.

• Eine unbenannte (*anonyme*) Funktion zu verwenden. Im Unterschied zu benannten Funktionen, die mit `sub` *definiert* werden, werden anonyme Funktionen zur Laufzeit mit `sub`

```
use warnings;
use strict;

my $input;

sub beta {
  return $input * $input;
}
sub alpha {
  while ($input = STDIN>) {
    print beta() . "\n";
  }
}
alpha();
```

Listing 1

als Operator *zugewiesen*. Durch diese Zuweisung wird die Funktion auch erst zur Laufzeit kompiliert und daher gibt es vorher noch keine globale Funktion, an die eine Variable gebunden werden könnte, wodurch das Problem der Variablenteilung erst gar nicht entsteht. Mit dieser Technik lässt sich der Code wie folgt umschreiben:

```
use warnings;
use strict;

sub alpha
{
  while (my $input = <STDIN>) {
    my $coderef = sub {
      return $input * $input;
    };
    print &$coderef . "\n";
    # oder $coderef->()
  }
}
alpha();
```

`sub` wird hier als *Operator* verwendet (beachte den Strichpunkt nach der den Funktionskörper abschließenden Klammer) - es liefert eine Referenz auf den folgenden Funktionscode zurück, die in der Variable `$coderef` abgelegt wird. Später kann dann mittels Dereferenzierung diese Funktion über `$coderef` aufgerufen werden [4].

Tatsächlich wird in diesem Beispiel die anonyme Funktion bei jedem Schleifendurchlauf mit dem aktellen `$input` neu kompiliert [5] und eine Referenz darauf an `$coderef` zugewiesen. Bei der Funktion in diesem Beispiel ist das kein Problem, bei größeren Funktionen hingegen kann ein solches Vorgehen - das Neukompilieren von Codeteilen in Schleifen - rasch uneffizient werden. In solchen Fällen wird man dann besser nach dem ersten Ansatz - dem Deklarieren von Variablen in Scopes, die den Funktionen, die sie verwenden, übergeordnet sind - vorgehen.

Die zweite Warnungsmeldung (`"Variable may be unavailable"`) kommt auf ähnliche Art zustande. Gegeben sei dieser Code:

```
sub alpha {
  my $var = shift;

  sub beta {
    return sub {
      return $var;
    }
  }
  print "In alpha(): " . beta()->() . "\n";
  # oder &beta->()
}
alpha(1);
print "Hauptprog:  " . beta()->() . "\n";
alpha(2);
print "Hauptprog:  " . beta()->() . "\n";
alpha(3);
print "Hauptprog:  " . beta()->() . "\n";
```

Hier liefert `beta()` eine Codereferenz auf eine anonyme Funktion, die den durch `alpha()` gesetzten Wert der lexikalischen Variable `$var` zurückgibt. Das wird durch das Aufrufen von `alpha()` mit drei Werten getestet, wobei `beta()` sowohl von außerhalb als auch innerhalb von `alpha()` aufgerufen wird.

Lässt man diesen Code laufen, so erhält man - neben der oben gezeigten Warnung - die Ausgabe:

```
In alpha(): 1
Hauptprog:  1
In alpha(): 2
Hauptprog:  1
In alpha(): 3
Hauptprog:  1
```

Wird `beta()` *innerhalb* des Scopes von `alpha()` aufgerufen, funktioniert alles wie erwartet. *Außerhalb* von `alpha()` hingegen tritt dasselbe Phänomen wie im vorigen Beispiel auf: es wird dann immer nur der Wert des ersten Aufrufes zurückgeliefert.

Das Hauptproblem bei diesem und ähnlichem Code ist, dass lexikalisches Scoping einerseits und globale Funktionen andererseits miteinander vermischt werden. Perl kennt keine lexikalischen Funktionen; eine benannte Funktion ist von *überall* aufrufbar, vollkommen losgelöst von den Scopes, in die sie möglicherweise eingebettet wurde. Greift eine solche Funktion dann auf lexikalische Variable zu (die Scoping unterworfen sind), kann es bei unsauberer Verschachtelung zu dem gezeigten Verhalten kommen.

In diesem Code geschieht Folgendes: `beta()` wurde im Scope von `alpha()` definiert und kann daher auch auf dessen lexikalische Variable zugreifen. Die von `beta()` zurückgelieferte anonyme Funktion ist in dessen Scope definiert und hat daher Zugriff auf alle Variablen von `beta()` und somit auch auf das `$var` von `alpha()`.

Wird `beta()` aus `alpha()` heraus aufgerufen, so setzt dieses zuerst die Variable `$var` und ruft anschließend `beta()` auf. Dieses kompiliert die anonyme Funktion mit dem aktuellen Wert von `$var` und liefert eine Referenz auf diesen Code zurück. Daher wird in diesem Fall das erwartete Resultat zurückgegeben.

Wird `beta()` jedoch **außerhalb** des Scopes, in dem es kompiliert wurde, aufgerufen, so existiert keine Variable `$var`, da `alpha()`, in dessen Scope die Variable angelegt ist, ja nicht aufgerufen wurde - es kommt zu einer Verletzung des durch statisches Scoping vorgegebenen *Prinzips der Kapselung*. Dieses Prinzip besagt, dass auf eine lexikalische Variable nur aus dem Scope, in dem sie deklariert wurde, und aus untergeordneten Scopes zugegriffen werden darf, nicht aber durch anderen Code. Durch den Aufruf von `beta()` außerhalb von `alpha()` ist diese Forderung nicht mehr erfüllt und daher kommt es nun erneut zu dem bereits vorher erwähnten Vorgang der *Variablenteilung* - um die anonyme Funktion kompilieren zu können, muss der Compiler `beta()` auf seine eigene Version von `$var` zugreifen lassen, die zu diesem Zweck aus dem `$var` von `alpha()` kopiert wird und somit immer dessen ersten Wert enthält. Das `$var` von `alpha()` ist für `beta()` nicht mehr ansprechbar, worauf der Compiler mit der `"may be unavailable"` Meldung hinweist.

Ein Ausweg wäre hier, die statische Definition von `beta()` durch Zuweisung einer Codereferenz zur Laufzeit zu ersetzen. Verwendet man dabei einen Typeglob, so muss im gesamten Code (fast) nur eine Zeile geändert werden (siehe Listing 2).

Da aus der Definition nun eine Anweisung geworden ist, ist diese klarerweise mit einem Strichpunkt zu beenden. Man wird nun erfreut feststellen, dass der Code wunschgemäß arbeitet.

Die Quintessenz daraus: wann immer der Compiler eine dieser beiden Meldungen von sich gibt, sollte der betroffene Code auf verschachtelte Scopes mit darin enthaltenen Funktionsdefinitionen, die auf äußere lexikalische Variable zugreifen, durchsucht werden. Diese Funktionsdefinitionen sollte man dann durch Zuweisungen von Codereferenzen an Typeglobs ersetzen oder durch umstrukturieren des Programms auf verschachtelte Funktionen ganz verzichten [6].

Auf eine subtile Eigenschaft des Compilers sei abschließend noch hingewiesen: im ersten Codebeispiel wurde eine äußere Funktion `alpha` definiert, um das Problem besser zu veranschaulichen. Doch tatsächlich wäre das gar nicht notwendig - das Problem tritt auch ohne äußere Funktion auf:

```
use strict;
use warnings;

while (my $input = <STDIN>) {
  sub beta {
    return $input * $input;
  }
  print beta() . "\n";
}
```

Hier wird jedoch vom Compiler **keine** Warnungsmeldung ausgegeben, weil die Gefahr, das Prinzip der Kapselung zu verletzen, kleiner ist: da es keine äußere Funktion gibt, gibt es scheinbar auch keinen solchen Scope, der durch direktes Aufrufen von `beta()` durchbrochen werden könnte.

Dennoch ist Vorsicht geboten: auch die `while`-Schleife wird letztlich in einem Scope - dem *Dateiscope* - abgearbeitet, und die Funktion `beta()` könnte auch von Code aus anderen Dateien (und damit aus anderen Scopes!) aufgerufen werden. Damit hätte man dasselbe Problem - nur dass man dann von Perl nicht mehr darauf hingewiesen wird!

```
sub alpha {
   my $var = shift;

   # Zuweisung an Typeglob statt Definition
   *beta = sub {
     return sub {
       $var;
     }
   }; # Strichpunkt hier nicht vergessen
   print "In alpha(): " . beta()->() . "\n";
}
alpha(1);
print "Hauptprog:  " . beta()->() . "\n";
alpha(2);
print "Hauptprog:  " . beta()->() . "\n";
alpha(3);
print "Hauptprog:  " . beta()->() . "\n";
```

Listing 2

Persistente lexikalische Variable

Wie bereits früher erwähnt, müssen Geltungsbereich und Lebensdauer einer lexikalischen Variable nicht immer innerhalb derselben Grenzen liegen. Deklariert man eine Variable *außerhalb* der Funktion, die sie anspricht, in einem übergeordneten Scope:

```
{
  my $x;
  sub inc_x {
    $x++;
  }
}
print inc_x();        # Gibt 1 aus
print inc_x();        # Gibt 2 aus
```

so macht man sich dabei die Tatsache zunutze, dass in Perl Funktionen immer global sind - der Compiler erkennt, dass er am Ende des äußeren Scopes die lexikalische Variable `$x` nicht freigeben kann, weil durch die globale Funktion `inc_x()` immer noch darauf zugegriffen wird. Das Prinzip der Kapselung besagt jedoch, dass auf eine lexikalische Variable, die in einem Scope deklariert wurde, auch nur von diesem Scope (und untergeordneten Scopes) zugegriffen werden kann. Daher ist die Variable nach dem Erreichen des Blockendes nicht mehr ansprechbar, aber dennoch innerhalb des globalen `inc_x()` vorhanden. Anders gesagt: ein Zugriff ist nicht mehr direkt über den Namen, sondern nur noch über die Funktion möglich. Und da die Variable ja außerhalb der Funktion deklariert wurde, wird sie nicht bei jedem Aufruf neu angelegt, sondern behält zwischen Funktionsaufrufen immer den Wert, den sie beim Verlassen der Funktion hatte.

Obwohl also die Variable nur innerhalb des Scopes (und damit nur über die Funktion) ansprechbar ist, bleibt sie während der gesamten Lebensdauer des Perl-Scripts bestehen (weil auch die Funktion, die auf sie zugreift, solange bestehen bleibt [7]). Daher bezeichnet man solche Variable als *persistent*.

Mittlerweile sollte klar sein, dass eine persistente Variable lexikalisch *vor* der Funktion, die auf sie zugreift, deklariert werden muss.

Grundsätzlich gibt es folgende Möglichkeiten, persistente Variable zu deklarieren und zu verwenden:

• Am Beginn einer Datei - dann kommt der Dateiscope zum Tragen, d.h. die Variable ist innerhalb der gesamten Datei ansprechbar, aber nicht durch Code von außerhalb. Durch das Referenzieren in Funktionen kann sie jedoch auch von außen zugänglich gemacht werden, wobei es dann an der Funktion liegt, ob und wie sie den Zugriff auf die Variable gestattet.

Diese Technik wird gerne von *Modulen* (zuladbarem Code) verwendet, die innerhalb des Modulcodes eigene Variable von überall ansprechen, aber gleichzeitig den Zugriff von außerhalb zuverlässig unterbinden oder nur kontrolliert zulassen möchten. Solcher Code könnte z.B. wie folgt aussehen:

```
package MyModule;

my $level;        # Modul-weit ansprechbar

sub set_level {
  my $val = shift;
  die "Wert $val nicht erlaubt!\n"
     if $val < 1 || $val > 5;
  $level = $val;
}

sub get_level {
  warn "Level noch nicht gesetzt!\n"
     unless $level;
  $level;
}
```

Dieser Einsatz persistenter lexikalischer Variable zur *Modul-Konfiguration* ist sehr beliebt. Obiger Code verwendet - z.B. als Abstufung zur Ausgabe von Logdaten - eine Variable `$level`. Diverse Funktionen des Moduls (die hier nicht gezeigt sind) verwenden diesen Wert, um festzustellen, wieviel Informationen sie ausgeben sollen - je höher der numerische Wert, desto mehr soll ausgegeben werden. Anstatt eine Package-Variable zu verwenden, die von überall unkontrolliert (und auch mit ungültigen Werten) beschrieben werden kann, wird eine persistente lexikalische Variable definiert, die beim Kompilieren des Modulcodes eingerichtet wird.

Die Variable ist nur innerhalb der Datei ansprechbar, von außerhalb wird der Zugriff ausschließlich über Funktionen ermöglicht. Die `set`-Funktion prüft dabei vor dem eigentlichen Setzen, ob der gewünschte Wert im erlaubten Bereich liegt und bricht mit einem Fehler ab, falls das nicht der Fall ist. Da keine andere Möglichkeit existiert, die Variable von außerhalb zu setzen, können die übrigen Funktionen später dem gesetzten Wert "blind" vertrauen, es müssen keine Prüfungen mehr damit durchgeführt werden.

Tatsächlich bedarf es keiner sonstigen Maßnahmen mehr, um die Variable "sicher" zu machen - das durch statisches Scoping bereitgestellte Prinzip der Kapselung schließt zuverlässig jede weitere Möglichkeit des Zugriffs aus.

• Die zweite Einsatzmöglichkeit erlaubt das Kapseln von Variablen innerhalb einer Datei. Während im vorigen Fall implizit der Dateiscope verwendet wurde, muss hier ein eigener Scope angelegt werden, um eine solche Kapselung zu ermöglichen. Im Beispiel

```
{
  my $x;
  sub inc_x {
    $x++;
  }
}
```

geschieht dies durch Einfassen der Variablen- und Funktionsdeklaration in denselben Block. Die Variable wird auf diese Art in der Funktion "konserviert", und tatsächlich ist es genau dieser Trick, auf dem *Closures* aufsetzen.

• Ab Perl 5.10 gibt es mittels des neuen Schlüsselwortes `state` die Möglichkeit, lexikalische Variable explizit als persistent zu deklarieren. Damit kann man das obige Beispiel kürzer

```
sub inc_x {
  state $x++;
}
```

schreiben.

Im nächsten Teil dieses Tutorials werden noch Closures beschrieben. Danach sehen wir uns die Sprachelemente von Perl, mit denen Scoping eingerichtet und verwaltet wird, genauer an.

Ferry Bolhár-Nordenkampf

[1] In diesem Zusammenhang soll auf zwei Ungenauigkeiten in perlsub hingewiesen werden, wo es u.a. heißt:

> Unlike dynamic variables created by the "local" operator, lexical variables declared with "my" are totally hidden from the outside world, including any called subroutines.

Die lexikalische Variable in

```
my $x = 'Hallo';

sub test {
  print "$x\n";
}

test();    # Gibt 'Hallo' aus
```

ist natürlich auch innerhalb der Funktion `test()` sichtbar. Anders hingegen wird das Scoping in folgendem Code aussehen:

```
sub alpha {
  my $x  = 'Hallo';
  beta();
}
sub beta {
  print "$x\n";   # globales $x ($main::x)
}
alpha();
```

Nun sollte schon klar sein, wie die Dinge hier liegen: *innerhalb* der Funktion `alpha()` wird eine lexikalische Variable deklariert. Sie ist daher nur innerhalb des Scopes dieser Funktion sichtbar. In der Funktion `beta()`, die ihren eigenen Scope hat, ist die lexikalische Variable von `alpha()` nicht sichtbar (und daher wird `$x` hier als globale Variable interpretiert). Daran ändert auch der Umstand nichts, dass `beta()` aus `alpha()` heraus aufgerufen wird, denn dieser Aufruf erfolgt erst zur Laufzeit und hat daher auf das beim Kompilieren ablaufende statische Scoping keinen Einfluss mehr. Hätte man hingegen `$x` *außerhalb* von `alpha()` deklariert (also in jenem Scope, der beiden Funktionen übergeordnet ist), wäre die Variable auch innerhalb von `beta()` sichtbar gewesen.

Obiges Zitat wäre daher sinngemäß zu ergänzen:

> Unlike dynamic variables created by the "local" operator, lexical variables declared with "my" are totally hidden from the outside world, including any called subroutines >>> unless the lexical variables were declared in an outer scope. <<<

Ferner sei zu obigem Zitat aus *perlsub* noch festgestellt, dass `local` keine Variable "erzeugt" (*Unlike dynamic variables* ***created***...), sondern es ersetzt nur die Werte bereits bestehender Variable. Der Begriff *dynamic variable* existiert in Perl nicht.

[2] Manche Programmierer empfinden das als Schwäche.

[3] Ein Seiteneffekt der Variablenteilung ist, dass das `$input` innerhalb von `beta()` zu einer *`persistenten`* lexikalischen Variable wird, die ihren Wert über Funktionsaufrufe hinweg behält (siehe hierzu auch den Abschnitt über persistente lexikalische Variable).

[4] Eine Alternative wäre die Zuweisung der Codereferenz an einen Typeglob, was einer Funktionsdefinition zur Laufzeit entspricht. Eine derart benannte Funktion kann dann genauso wie jede definierte Funktion angesprochen werden.

[5] Theoretisch. Praktisch tut sich Perl das nicht an, sondern speichert den kompilierten Code in einem Cache. Trotzdem sollte man auf die Zuweisung von Codereferenzen in Schleifen aus Effizienzgründen verzichten, wenn sie nicht unbedingt notwendig ist.

[6] Beim Arbeiten in persistenten Umgebungen, z.B. `PersistentPerl` oder `mod_perl`, erhält man manchmal solche Meldungen, obwohl man scheinbar keine Funktionen verschachtelt hat. In diesen Umgebungen wird der Code des Hauptprogramms intern oft selbst als Funktion abgebildet, wodurch sich ein gleichzeitiges Cachen mehrerer Scripts und deren wiederholtes Ausführen leicht implementieren lässt. Ist man mit dieser Situation konfrontiert, sollte in der Dokumentation der jeweiligen Software nachgeschlagen werden, um festzustellen, wie diese intern das Programm abbildet. Danach ist zu prüfen, ob dadurch nicht eine der geschilderten Situationen vorliegt.

[7] Im Falle von Closures kann die Lebensdauer auch wesentlich kürzer sein, wenn die Closure nur temporär angelegt und verwendet wird (mehr dazu im Abschnitt Closures).

Werden Sie selbst zum Autor...
... wir freuen uns über Ihren Beitrag!

info@foo-magazin.de

Win32 Tipps & Tricks

In der heutigen Ausgabe gibt es wieder drei Tricks, die Windows-spezifisch sind. Wenn es noch Themenvorschläge und/oder Feedback gibt, dann reicht eine kurze Mail an *feedback@foo-magazin.de*.

Symlinks unter Win32

Unter Linux sind sie bekannt und nützlich - die symbolischen Links. Perl stellt auch zwei Funktionen zur Verfügung, mit denen sehr einfach solche Links erstellt werden können. Unter Windows ist das nicht so einfach möglich. Aber wenn das System im NTFS-Format vorliegt, kann man `Win32::Symlink` verwenden:

```
use Win32::Symlink;

mkdir 'D:\from';
symlink( 'D:\from' => 'D:\to' );
print readlink( 'D:\to' ); # 'D:\from'
rmdir 'D:\from', 'D:\to';
```

Hat sich eine Datei geändert?

Ich habe ein Programm, das auf Änderungen in einer Datei reagieren soll, diese Änderungen bedeuten aber nicht immer dass eine Zeile ans Ende gestellt wurde wie bei Logdateien oder ähnliches üblich. Deshalb kann ich nicht mit `File::Tail` arbeiten. Um benachrichtigt zu werden, ob eine Datei geändert wurde, nehme ich `Win32::FileNotify`:

```
use Win32::FileNotify;

my $file = './test.txt';
my $notify = Win32::FileNotify->new(
                                  $file );
$notify->wait;

 print $file, " wurde veraendert\n";
```

Standardanwendung für eine Datei

Im Windows-Explorer kann man schon an den Icons erkennen, mit welcher Anwendung eine Datei geöffnet wird. Diese Zuordnung von Dateiendung zu Anwendung wird in der Registry festgehalten. Manchmal möchte man aus einem Programm heraus eine Datei mit der "dazugehörigen" Windows-Anwendung öffnen. Doch wie bekommt man am besten heraus, mit welcher Anwendung die Datei geöffnet werden soll? `Win32::FetchCommand` bietet ein sehr übersichtliches Interface zu der Registry-Suche. Das Modul ist hauptsächlich für die Verwendung im Zusammenspiel mit `Win32::Process` gedacht, aber der Pfad zu der richtigen Anwendung kann auch in anderen Fällen sehr nützlich sein.

```
#!/usr/bin/perl

use strict;
use warnings;

use Win32::FetchCommand;

my @Command  = FetchCommand ('.pl');
print "@Command\n";

my ($Exe, @CmdLine) =
 FetchCommand ('file.txt');
print $Exe,"\n";
```

Renée Bäcker

HTML::Display - HTML zum lokalen Browser senden

Dank HTML::Display von Max Maischein (corion) lassen sich einzelne HTML-Seiten ohne Web-Server ganz einfach von Perl aus zum Browser schicken und in diesem anzeigen.

Extrem praktisch, wenn man z.B. einen dynamischen generierten Report als HTML-Datei speichern und diesen gleich im lokalen Browser anzeigen lassen möchte. Fast schon eine Fernsteuerung für Browser.

HTML::Display funktioniert auf den gängigen Betriebssystemem mit den meisten Browsern, wie z.B. Internet Explorer oder Firefox. Bisher habe ich es allerdings nur mit Windows XP SP3 und Ubuntu 8.04 ausprobiert.

Beispiel: lokale HTML-Datei zum Browser des Users senden

```
#!/usr/bin/perl
use strict;
use warnings;
use HTML::Display;  # importiert display()

my $file = "index.html";

display( file => $file ) ;
```

Die dazugehörige HTML-Datei `index.html` im selben Verzeichnis (siehe Listing 1).

Beispiel: Webseite zum Browser des Users senden

Dazu wird zunächst die Website mit LWP::Simple geholt und dann an HTML::Display weitergereicht.

```
#!/usr/bin/perl
use strict;
use warnings;

use HTML::Display;
use LWP::Simple;

my $location = "http://perl-howto.de/";

my $html = get $location;

display( html => $html,
         location => $location );
```

Der Parameter `location` setzt die Navigationsleiste des Browsers, während `html` den Quelltext der anzuzeigenden Seite erwartet.

```
<html>
<head>
<title>
HTML-Display rocks
</title>
</head>
<body>
<h1>HTML-Display rocks</h1>
<p>
Lorem ipsum dolor sit amet, consetetur sadipscing elitr,
sed diam nonumy eirmod tempor invidunt ut labore et
dolore magna aliquyam erat, sed diam voluptua.
</p>
<p>
Duis autem vel eum iriure dolor in hendrerit in vulputate velit
esse molestie consequat, vel illum dolore eu feugiat nulla
facilisis at vero eros et accumsan et iusto odio dignissim qui
blandit praesent luptatum zzril delenit augue duis dolore te
feugait nulla facilisi.
</p>
</body>
</html>
```

Listing 1

Beispiel: Datenbankreport auf die Schnelle im Browser anzeigen

Aus einer (sehr einfachen) Adress-Datenbank

```
$ sqlite3 Adressen.db
sqlite> select customer_id,vorname,
          nachname from customer;
1|Hans|Meier
2|Susanne|Schmidt
3|Otto|Muster
```

werden alle Datensätze ermittelt, anschließend mit HTML-Tags versehen und im lokalen Browser angezeigt (siehe Listing 2).

Geht schnell und benötigt keinen Web-Server!

Thomas Fahle

```
#!/usr/bin/perl
use strict;
use warnings;
use DBI;
use Encode;          # importiert decode_utf8
use HTML::Display;         # importiert display()
use CGI;

my $table = retrieve();

my $html = create_html( $table );

display( html => $html ); # legt automatisch eine temp-Datei an

sub create_html {
       my $table = shift @_;
       my $cgi =  new CGI;
       my $content;
       $content = $cgi->start_html( 'Database Report' );
       $content .= $cgi->start_table( { -border => 1 } );

       foreach my $row (@$table) {
              $content .= $cgi->Tr( { -align => 'left', -valign => 'top' },
                                    $cgi->td( $row )
                                  );
       }

       $content .= $cgi->end_table();
       $content .= $cgi->end_html();

       $content = decode_utf8( $content );
       return $content;
}

sub retrieve {

       my $dbh = DBI->connect( "dbi:SQLite:dbname=Adressen.db", "", "",
                                       { RaiseError => 1 } );

       my $sql = qq~select customer_id,vorname,nachname from customer~;

       my $sth = $dbh->prepare( $sql );
       $sth->execute();

       my $tbl_ary_ref = $sth->fetchall_arrayref;

       $sth->finish();
       undef $sth;
       $dbh->disconnect();

       return wantarray ? @$tbl_ary_ref : $tbl_ary_ref;
}
```

Listing 2

Rezension - Automating System Administration with Perl: Tools to Make You More Efficient

David N. Blank-Edelman:
Automating System Administration with Perl:
Tools to Make You More Efficient
ISBN: 059600639X
O'Reilly Media, Auflage: 2nd ed. (15. Juni 2009)

Automate or die

Wie der treffende Titel des Buches bereits aussagt, sind Automatisierung und höhere Effizienz im komplexen Aufgabenfeld Systemadministration Kernthemen. Systemadministratoren (m/w) müssen über viele Kenntnisse und Fähigkeiten aus sehr unterschiedlichen Bereichen verfügen und unter oft hohem Zeitdruck die ständigen wachsenden Anforderungen ihrer Kunden (User) termingerecht und in hoher Qualität erfüllen können.

Für diesen Aufgabenbereich ist die dynamische Programmiersprache Perl in Kombination mit CPAN-Modulen eines der besten Werkzeuge, das am Markt erhältlich ist. David N. Blank-Edelman legt dar, wie man dieses Werkzeug richtig einsetzt.

Die zweite Auflage

Das "Otter Book", ein Klassiker unter den Perl-Büchern von O'Reilly, ist Mitte 2009 in einer vollständig überarbeiteten und erweiterten zweiten Auflage erschienen. Gegenüber der ersten Auflage, die bereits 2000 erschienen ist, hat sich der Seitenumfang von 430 auf 666 Seiten deutlich erhöht, vier Kapitel und zwei Anhänge sind neu dazugekommen.

Inhaltsverzeichnis

• Filesystems
• User Accounts
• User Activity
• TCP/IP Name and Configuration Services
• Working with Configuration Files
• SQL Database Administration
• Email
• Directory Services
• Log Files
• Security
• SNMP
• Network Mapping and Monitoring
• Experiential Learning

Jedes der 14 Kapitel ist praktisch in sich abgeschlossen. Daher kann das Buch in (fast) beliebiger Reihenfolge gelesen bzw. durchgearbeitet werden. Die vorhandenen Verweise auf vorhergehende Kapitel erleichtern die Rosinenpickerei.

Kurz-Tutorien im Anhang

Im Anhang bietet der Autor in kurzen, lesenswerten Tutorien einen Einstieg oder Wiederauffrischung für die folgenden Themen:

• The Eight-Minute XML Tutorial
• The 10-Minute XPath Tutorial
• The 10-Minute LDAP Tutorial
• The 15-Minute SQL Tutorial
• The Five-Minute RCS Tutorial
• The Two-Minute VBScript-to-Perl Tutorial
• The 20-Minute SNMP Tutorial

Die knackig geschriebenen Tutorials sind eine Klasse für sich und rechtfertigen schon allein den Kauf des Buches.

Probleme lösen

Jedes Kapitel beginnt mit einer Beschreibung eines praxisrelevanten Problembereichs und einer Übersicht der Lösungsmöglichkeiten mit Perl-Bordmitteln oder CPAN-Modulen. Grundlegende Perl-Kenntnisse werden vorausgesetzt.

Die vorgestellten Lösungen dienen nicht als Kochbuch, sondern sollen auf dem Weg zur eigenen Lösung helfen. Jede Lösungsmöglichkeit wird so weit behandelt, das der Leser eine informierte Entscheidung treffen kann, welche Methode für dieses oder ein ähnliches Problem eingesetzt werden kann.

David beschreibt auch die Vor- und Nachteile unterschiedlicher CPAN-Module bzw. vergleicht mehrere Module, die für das gleiche Problem zur Verfügung stehen, und verweist ggf. auf weitere Lösungsmöglichkeiten. Das Buch ist daher auch ein Führer durch den CPAN-Dschungel (Stand 2009) und spart erheblich Zeit bei der Suche und Auswahl des passenden CPAN-Moduls.

Am Ende eines jedes Kapitels findet der Leser eine tabellarische Übersicht der beschriebenen CPAN-Module und Verweise zu weiterführenden Ressourcen.

Alle Codebespiele sind nach den Richtlinien der `Perl Best Practices` aufgebaut, mit `PerlTidy` formatiert und verwenden `use strict`. Leider hat der Verlag aus Platzgünden die Zeile `use strict` in den Beispielen wieder gestrichen. Dies ist aber auch der einzige Wermutstropfen.

Der Autor hat Spass am Schreiben, Humor und kann sein umfangreiches, in der Praxis erworbenes, Fachwissen sehr gut **rüberbringen**.

Websites

Zu dem Buch gibt es eine begleitende Website mit Auszügen aus den Buch und einen Twitter-Feed.

http://www.otterbook.com/

http://twitter.com/otterbook

http://oreilly.com/perl/excerpts/system-admin-with-perl/ten-minute-xpath-utorial.html

http://www.oreillynet.com/pub/a/perl/excerpts/system-admin-with-perl/ten-minute-ldap-utorial.html

Thomas Fahle

"Merkwürdigkeiten" - Regex mit /g verhält sich "komisch"

In diesem Teil von "Merkwürdigkeiten in Perl" geht es um ein Verhalten von Regulären Ausdrücken, über das man durchaus stolpern kann wenn man sich mit Regulären Ausdrücken nicht so auskennt.

Reguläre Ausdrücke sind sehr mächtig und sie sind mit ein Grund dafür, dass Perl in der "Textbearbeitung" so stark ist. Und dank den *Perl Compatible Regular Expressions* (PCRE) ist diese mächtige Syntax auch an vielen anderen Stellen einsetzbar (z.B. Apache, PHP und Postfix). Aber je mächtiger ein Tool ist, umso mehr Verwirrung kann das Tool stiften wenn der User sich nicht so sehr gut damit auskennt.

Die Dokumentation der Regulären Ausdrücke in Perl ist sehr umfangreich und ist auf verschiedene Dokumente aufgeteilt, die unterschiedlich tief in das Thema eintauchen. Diese Dokumente sind:

- perldoc perlre
- perldoc perlrequick
- perldoc perlretut

Und da sich in Perl 5.10 einiges bei den Regulären Ausdrücken getan hat, sind dort noch `perldoc perlreguts` und `perldoc perlreapi` hinzugekommen. Diese Dokumentation behandelt aber nur das Backend.

Doch jetzt zum Thema: Bei perl.de ist eine Frage zu Regulären Ausdrücken und der Auswirkung des `/g`-Modifiers aufgekommen. Zu dem Beispiel

```
$text = "foo foo";

for ($i=1;$i<=10;$i++){
   print "$i " if $text =~ /foo/g;
}
```

gibt es folgendes Ergebnis

```
1 2 4 5 7 8 10
```

und der Fragensteller hat sich gewundert, warum das nicht `1 2 3 4 5 6 7 8 9 10` ergibt.

Das ist keine Stolperstelle, wenn man die Dokumentation (`perldoc perlretut`) gelesen hat. Dort steht deutlich, dass `/g` dafür sorgt, dass Perl sich die Position nach einem Match merkt und dann genau an dieser Stelle weitermacht. Die Positionsangabe wird nur bei einem erfolglosen Versuch oder bei Änderungen am zu durchsuchenden String zurückgesetzt.

Was passiert also bei dem oben gezeigtem Code?

$i = 1 : Regex matcht erstes "foo", merkt sich Position 3
$i = 2: Regex matcht zweites "foo", merkt sich Position 7
$i = 3: Regex kann nichts mehr matchen, weil im String nichts mehr steht -> Keine Ausgabe -> jetzt erst wird die Position zurückgesetzt.
$i = 4: jetzt beginnt das ganze wieder von vorne.

Wie kann man das umgehen?
In dem man den Regulären Ausdruck einfach in Listenkontext verwendet, denn damit wird gleich der gesamte String auf alle Vorkommen von "foo" untersucht und die Treffer werden zurückgeliefert. Da nach dem letzten Treffer ein erfolgloser Match kommt, wird die Position zurückgesetzt. Dadurch beginnt die Regex-Engine beim nächsten Schleifendurchlauf wieder von vorne.

```
$text = "foo foo";

for ($i=1;$i<=10;$i++){
   print "$i " if( () = $text =~ /foo/ig );
}
```

Renée Bäcker

CPAN News XII

Text::Diff3

Mit `Text::Diff3` steht ein Perl-Modul zur Verfügung, das das GNU-Tool diff3 implementiert. Damit lassen sich drei Texte Zeile für Zeile vergleichen.

```
use Text::Diff3;

my $f = Text::Diff3::Factory->new;
my $mytext   = $f->create_text(
                    \@zeilen_text1 );
my $original = $f->create_text(
                    \@zeilen_text2 );
my $yourtext = $f->create_text(
                    \@zeilen_text3 );

my $p = $f->create_diff3;
my $diff3 = $p->diff3(
            $mytext, $original, $yourtext);

$diff3->each(sub{
    my($r) = @_;
    print $r->as_string, "\n";
    print $mytext->as_string_range(
                                $r->range0);
});
```

Term::Pulse

Mit Term::Pulse kann man ein Fortschrittsbalken in einem Terminal anzeigen, das ungefähr so aussieht: `Checking...[ = ] (xxx sec elapsed)`. Der Einsatz des Moduls ist sehr einfach, da es die Methoden `pulse_start` und `pulse_stop` exportiert:

```
use Term::Pulse;
pulse_start(
    name => 'Checking',
    rotate => 0,
    time => 1,
);
sleep 3;
pulse_stop();
```

Image::JpegCheck

Dateiendungen geben nicht immer das wieder, was wirklich in den Dateien steckt. Wird ein JPEG-Bild, es liegt aber keins vor, versuchen unbedarfte User einfach ein Bild in einem anderen Format umzubenennen. Prüft man in seiner Anwendung nur die Dateiendung, so kann das zu Problemen führen. Hier kann das Modul `Image::JpegCheck` helfen:

```
use Image::JpegCheck;

# JPEG-Datei
print is_jpeg('foo.jpg'),"\n",
      # PNG-Datei
      is_jpeg('foo_png.jpg'),"\n";

__END__
1
0
```

aliased

Modularisierung ist sehr praktisch, aber je mehr modularisiert wird, umso länger werden die `package`-Namen. Wenn der Name dann häufiger gebraucht wird, dann ist die Schreibarbeit nervig oder man erkennt den Klassennamen nicht gleich. Curtis 'Ovid' Poe hat ein Modul geschrieben, mit dem sich die Schreibarbeit und die Lesbarkeit verbessern lassen: `aliased`.

```
use aliased
  'Ein::langer::Namespace::Name::Klasse';
my $class = Klasse->new;

use aliased
  'Ein::langer::Namespace::Name::Klasse'
    => 'MeineKlasse';
my $my_class = MeineKlasse->new;

# oder
use aliased;

  my $Class  = alias
    "Ein::langer::Namespace::Name::Klasse";
  my $object = $Class->new;
```

WebService::Prowl

Prowl ist ein Push-Client für das iPhone und es gibt mit `WebService::Prowl` ein Perl-Modul, um das ganze bequem nutzen zu können. So kann man sich über Twitter-Meldungen benachrichtigen lassen oder über System-Zustände informieren lassen. Der Einsatz des Perl-Moduls ist sehr einfach:

```
use WebService::Prowl;
my $new_order = check_orders;
if ( $new_order ) {
    my $ws = WevService::Prowl->new(
       apikey => 40byteshexadecimalstring);
    $ws->verify || die $ws->error();
    $ws->add(
        application => "Foo_Order_System",
        event       => "neue Bestellung",
        description =>
               'neue Bestellung fuer $foo',
    );
    }
```

Bisect::Perl::UsingGit

Hin und wieder passiert es, dass an Perl etwas verändert wird und damit auch gleichzeitig ein Fehler eingebaut wird. Bei der Vielzahl von Commits im Perl-Repository ist es schwer, den Commit zu finden, der den Fehler eingebaut hat. `Bisect::Perl::UsingGit` kommt mit einem Tool, dass die Fehlersuche erleichtert: Erstmal die Perl-Sourcen aus dem Repsitory holen, ein Testskript schreiben (das mit der alten Perl-Version funktioniert, mit der neuen aber nicht) und dann das hier - die Perl-Versionen müssen ggf. angepasst werden:

```
git bisect reset
git bisect start
git bisect good perl-5.8.8
git bisect bad perl-5.10.0
# Bisecting: 4041 revisions left to test
# after this
git bisect run bisect_perl_using_git \
    --action perl_fails \
    --filename ~/testcase.pl
# ... after checking out 16 trees and about
one hour, it says:
# fc8cd66c26827f6c2ee1aa00ab2d3b3c320a4a28
is first bad commit
# commit
fc8cd66c26827f6c2ee1aa00ab2d3b3c320a4a28
# Author: Yves Orton <demerphq@gmail.com>
# Date:   Tue Sep 19 03:37:19 2006 +0200
#
#    Re: \N{...} in regular expression
[PATCH]
# ...
git bisect reset
```

Neues von TPF

Grant update - improving learn.perl.org

Ab sofort werden die Grant updates von den einzelnen Grant-Managern durchgeführt.

Als erstes gibt es Neuigkeiten zum learn.perl.org-Grant von Eric Wilhelm:

Eric hat das Modul Combust::Spontaneously geschrieben. Damit bekommt man einen lokalen Webserver (ohne Apache) mit dem perl.org-Template-System innerhalb von 5 Minuten aufzusetzen.

Jetzt beginnt er, einige Inhalte zu übernehmen.

Neues Mitglied im "Board of Directors"

Curtis 'Ovid' Poe ist ab sofort neues Mitglied im "Board of Directors" der Perl Foundation.

Er ist kein Unbekannter in der Perl Foundation, da er schon seit Jahren Mitglied im "Grants Committee" ist.

Personelle Änderungen

Jim Brandt hat im Blog der Perl Foundation ein paar personelle Änderungen bekanntgegeben:

- Josh McAdams macht seinen Platz als PR-Verantwortlicher frei, weil er ab sofort als Vorsitzender des Konferenz Komitees, das für die YAPC::NA verantwortlich ist.
- Casey West übernimmt die Position des Webseiten-Verantwortlichen
- Dan Magnuszewski übernimmt die Rolle des PR-Verantwortlichen von Josh McAdams.

Das Gebiet der PR soll in Zukunft etwas geändert werden. Hier soll ein Komitee ähnlich dem Grants Komitee eingerichtet werden. Dan arbeitet auch an anderen Dingen, um mehr Personen in der PR-Arbeit einzubinden.

Die Perl Foundation bedankt sich bei Jeremy Fluhmann, der mehrere Jahre lang das Konferenz Komitee geführt hat und bei Kirsten Jones, die sich um die Webseite gekümmert hat.

Grants im 3. Quartal 2009

Es wurde über die Grants-Vorschläge abgestimmt wurde, hat das Grants-Komittee die Ergebnisse veröffentlicht.

Folgende Grants wurden angenommen:
- Corporate, Embedded und Multi-user Perl unter Windows - Curtis Jewell. (Dieser Grant wird zum Teil auch von Adam Kennedy finanziell unterstützt)
- wxPerl documentation - Eric J. Roode.
- Ändern des optree-Erzeugungsprozess in Perl 5 in zwei Schritte: Erstellung eins Abstract Syntax Tree und Code-Generierung - Gerard Goossen.

Der letzte Grant wird in zwei kleinere Teile aufteilt.

Veränderungen im TPF-Board

Es gibt ein paar Veränderungen im obersten Gremium der Perl Foundation:

• Nat Torkington hat die Rolle des Chairman von Bill Odom übernommen
• Jim Brandt ist neuer Präsident, da TPF den Grant-Vorschlag von Richard Dice angenommen hat - unter der Voraussetzung, dass er während dieser Zeit kein Mitglied des Boards ist.
• Da Jim Brandt den Posten des Präsidenten übernommen hat, wurde Karen Pauley als Vize-Präsidentin gewählt.

Updates zu laufenden Grants

• Bugfixing im Archive::Zip Modul - Alan Haggai Alavi
Alan Haggai hat die Umstellung auf benannte Paramter abgeschlossen und viele Tests hinzugefügt. Ein neues Release ist im CPAN zu finden. Zwischenzeitlich war Alan einer der "Send-a-newbie"-Gesandten auf der YAPC

• Test::Builder 2 - Michael G Schwern
Michael hat vor zwei Monaten einen neuen Vollzeitjob angefangen und hat nicht an dem Grant weitergearbeitet. Er wird sehr bald wieder mit dem Projekt weitermachen.

• Perl Umfrage - Kieren Diment
Kieren hat das Catalyst-Buch abgeschlossen und arbeitet weiter an der Umfrage. Er hat die Geschichte um "Perl Marketing" verfolgt und wird das später noch weiter analysieren. Kieren hat auch ein paar Open Source Umfragen gefunden, von denen er sich Ideen holen will. Die Perl-Umfrage wird unter http://survey.perlfoundation.org gehostet.

• Embedding perl into C++ applications - Leon Timmermans
Leon hat gerade erst wieder angefangen, daran zu arbeiten. Aber er hat schon mindestens einen Bug gefixt und er erweitert die Tests

• Mojo Documentation Project - Sebastian Riedel
Sebastian kommt mit dem Manual gut voran. Zur Zeit organisiert er das bisher Geschriebene in Kapitel. Sebastian hat zuletzt Schwierigkeiten gehabt über Mojolicious zu schreiben, da die Lernkurve zu steil war. Aber er hat Mojolicious::Lite entwickelt und ein Tutorial darüber in seinem Blog veröffentlicht: http://labs.kraih.com/blog/2009/07/perl-is-awesome.html

• Improving learn.perl.org - Eric und Tina
Eric hat mit mit einigen Themen angefangen aber noch nicht abgeschlossen.

• Pod Mangling Utility Improvements - Ricardo Signes
Ricardo hofft, einige organisatorische Dinge bis zur OSCON geklärt und Grandlagen abgeschlossen zu haben. Wenn nichts dazwischen kommt, wird er kurz nach der OSCON mit der Arbeit anfangen. Ricardo hat hauptsächlich Diagramme und Notizen auf seinem Notebook erstellt. Aber er arbeitet auch an den nächsten Schritten der Programmierung. Hier sind die nächsten Schritte zu finden: http://rjbs.manxome.org/rubric/entry/1792

• A lightweight web framework for Perl 6
Wie immer kann der Fortschritt im GitHub-Projekt unter http://github.com/masak/web und in den Blogposts verfolgt werden.

Termine

November 2009

02. Treffen Vienna.pm
03. Treffen Frankfurt.pm
05. Treffen Dresden.pm
09. Treffen Ruhr.pm
11. Treffen Hamburg.pm
12. Treffen Cologne.pm
16. Treffen Erlangen.pm
18. Treffen Darmstadt.pm
    Treffen München.pm
21. 1. Baltischer Perl-Workshop
24. Treffen Bielefeld.pm
25. Treffen Berlin.pm
28-29. Frankfurter Perl-Community Workshop

Dezember 2009

03. Treffen Dresden.pm
05. Londoner Perl-Workshop
07. Treffen Vienna.pm
08. Treffen Frankfurt.pm
09. Treffen Hamburg.pm
10. Treffen Cologne.pm
14. Treffen Ruhr.pm
16. Treffen Darmstadt.pm
21. Treffen Erlangen.pm
29. Treffen Bielefeld.pm
30. Treffen Berlin.pm

Januar 2010

05. Treffen Frankfurt.pm
07. Treffen Dresden.pm
11. Treffen Ruhr.pm
18. Treffen Erlangen.pm
20. Treffen München.pm
26. Treffen Bielefeld.pm
27. Treffen Berlin.pm

Hier finden Sie alle Termine rund um Perl.

Natürlich kann sich der ein oder andere Termin noch ändern. Darauf haben wir (die Redaktion) jedoch keinen Einfluss.

Uhrzeiten und weiterführende Links zu den einzelnen Veranstaltungen finden Sie unter

http://www.perlmongers.de

Kennen Sie weitere Termine, die in der nächsten Ausgabe von $foo veröffentlicht werden sollen? Dann schicken Sie bitte eine EMail an:

termine@foo-magazin.de

LINKS

http://www.perl-nachrichten.de

Unter Perl-Nachrichten.de sind deutschsprachige News rund um die Programmiersprache Perl zu finden. Jeder ist dazu eingeladen, solche Nachrichten auf der Webseite einzureichen.

http://www.perl-community.de

Perl-Community.de 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. Auf der Linkseite von Perl-Community.de findet man viele Verweise auf nützliche Seiten.

http://www.perlmongers.de/
http://www.pm.org/

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

http://www.perl-workshop.de

Der Deutsche Perl-Workshop hat sich zum Ziel gesetzt, den Austausch zwischen Perl-Programmierern zu fördern.

http://www.perl-foundation.org

Die Perl-Foundation nimmt eine führende 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-Foundationen geleistet. Jedes Jahr werden Studenten beim "Google Summer of Code" betreut.

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.

Smart-Websolutions

Windolph und Bäcker GbR

Perl-Programmierung

info@smart-websolutions.de

// SEIBERT / MEDIA

BESSERE ATMOSPHÄRE? MEHR FREIRAUM?

Wir suchen erfahrene Perl-Programmierer/innen (Vollzeit)

//SEIBERT/MEDIA besteht aus den vier Kompetenzfeldern Consulting, Design, Technologies und Systems und gehört zu den erfahrenen und professionellen Multimedia-Agenturen in Deutschland. Wir entwickeln seit 1996 mit heute knapp 60 Mitarbeitern Intranets, Extranet-Systeme, Web-Portale aber auch klassische Internet-Seiten. Seit 2005 konzipiert unsere Designabteilung hochwertige Unternehmensauftritte. Beratungen im Bereich Online-Marketing und Usability runden das Leistungsportfolio ab.

Ihre Aufgabe wird sein, in unserer Entwicklungsabteilung im Team komplexe E-Business Applikationen zu entwickeln. Dabei ist objektorientiertes Denken genauso wichtig, wie das Auffinden individueller und innovativer Lösungsansätze, die gemeinsam realisiert werden.

Wir freuen uns auf Ihre Bewerbung unter www.seibert-media.net/jobs.
// SEIBERT / MEDIA GmbH, Rheingau Palais, Söhnleinstraße 8, 65201 Wiesbaden
T. +49 611 20570-0 / F. +49 611 20570-70, bewerbung@seibert-media.net

„Statt mit blumigen Worten umschreiben unsere Programmierer den Job so:

Apache, Catalyst, CGI, DBI, JSON, Log::Log4Perl, mod_perl, SOAP::Lite, XML::LibXML, YAML“