Ausgabe 4/2012 – www.foo-magazin.de - 6,00 €

$foo

PERL MAGAZIN

Compile- vs. Runtime
Alles eine Frage der Phase...

XML::Compile
und SOAP

Mojolicious Tutorial
Teil 1

Nr. 24

LONDON PERL 2012 WORKSHOP

- # Free Conference
- # Free Training
- # Free Coffee and Cakes
- # Free Evening Social
  (with limited free drinks, and food)
- # International Speakers

Wholly Supported by Sponsors

www.londonperlworkshop.org.uk

The London Perl Workshop is a free, one day, conference in central London, UK. LPW is the UK's Premier Perl Event and is also known as the United Kingdom Perl Workshop (UKPW). In 2012 it will be held on Saturday 24th November at Westminster University's New Cavendish Campus. We welcome people from any language, background, or circumstance to attend and join in with the event.
Please register your attendance on the above website, this will enable us to accurately judge room sizes and free food/drink numbers.

25 Years of Perl

antibody lux
Booking.com
:BYTEMARK HOSTING
eligo
EVOZON software development
EXOnetric
{MAGNUM SOLUTIONS LIMITED}
motortrak driving digital retail
NET-A-PORTER
nestoria
OPUS VL
PetaMem
SHADOWCAT SYSTEMS Supporting CPAN
University of Westminster

VORWORT

Perl in der Gesellschaft

Die Gesellschaft ist vielschichtig - genau wie Perl. Wir hatten auf der Webseite den Satz "DAS Magazin für den Perl-Programmierer", dass wir nur einen Teil unserer Gesellschaft ansprechen - nämlich den männlichen Teil - war nicht beabsichtigt. Darauf aufmerksam wurde ich durch einen Thread auf Perl-Community.de. Dort wurde auch angemerkt, dass es in den Artikeln ebenfalls nur "Programmierer" heißt.

Da wir aber alle Teile der Gesellschaft ansprechen wollen, werden wir in Zukunft sowohl von Programmierern als auch Programmiererinnen schreiben.

Ebenfalls ein wichtiger Teil Gesellschaft sind Wissenschaftler - auch diese Personen verwenden Perl. Unter http://perl4science.github.com/ gibt es eine Seite, die die Verwendung von Perl in der Wissenschaft voranbringen will. Einer der Hauptinitiatoren ist dabei Joel Berger, der unter http://blogs.perl.org/users/joel_berger/2012/07/response-to-scientific-papers-and-softwares.html ein wenig mehr über die Motivation dahinter erzählt.

Zum Abschluss noch ein kleiner Aufruf: Wir brauchen Autoren und Artikelideen. Leider können wir nicht wirklich Autorenhonorar bieten, weil wir die "Gewinne" in Sponsoring für Veranstaltungen wie den Deutschen Perl-Workshop und die YAPC::Europe stecken. Wenn jemand Interesse hat, Artikel zu schreiben, bitte unter info@perl-magazin.de melden.

Jetzt wünschen wir aber viel Spaß mit der neuen Ausgabe von $foo.

Renée Bäcker

Die Codebeispiele können mit dem Code

2mcbv39

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

Alle weiterführenden Links werden auf del.icio.us gesammelt. Für diese Ausgabe: http://del.icio.us/foo_magazin/issue24

IMPRESSUM

Herausgeber:	Perl-Services.de Renée Bäcker Bergfeldstr. 23 D - 64560 Riedstadt
Redaktion:	Renée Bäcker
Anzeigen:	Renée Bäcker
Layout:	//SEIBERT/MEDIA
Auflage:	500 Exemplare
Druck:	powerdruck Druck- & VerlagsgesmbH Wienerstraße 116 A-2483 Ebreichsdorf
ISSN Print:	1864-7537
ISSN Online:	1864-7545
Feedback:	feedback@perl-magazin.de

INHALTSVERZEICHNIS

ALLGEMEINES

PERL

MODULE

NEWS

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, Organisator des kleinen Frankfurt Perl-Community Workshops, Grant Manager bei der TPF und bei der Perl-Marketing-Gruppe.

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.

Mark Overmeer

Mark Overmeer ist ein fleißiger Programmierer - mit einigen großen Perl Projekten. Auf CPAN findet man Mail::Box --for automatic email processing-- und XML::Compile --XML processing--. Außerdem arbeitet er an CPAN6, dem Nachfolger des CPAN Archivs. Mark hat einen Abschluss als Master in Computing Science und arbeitet zur Zeit als selbständiger Programmierer, Systemadministrator und Trainer. Er ist Geschäftsführer von NLUUG (früher bekannt als die Niederländische UNIX User Group), SPPN (Niederländische Perl Promotion Foundation) und Oophaga (CAcert Equipment). Und das neben vielen anderen Aktivitäten. Mehr Informationen unter *http://solutions.overmeer.net*.

Renée Bäcker

Subroutinen Prototypen

Viele Perl-Programmierer verwenden Subroutinen-Prototypen ohne dass sie verstehen wofür diese Prototypen da sind. Eine Subroutine mit Prototyp sieht wie folgt aus:

```
sub hello ($) {
    my $name = shift;
    print "hello $name\n";
}
```

`$` ist hier der Prototyp. Doch was sagt dieser Prototyp aus? In anderen Sprachen werden solche Prototypen/Signaturen dafür verwendet, um herauszufinden welche Subroutine aufgerufen wird (falls mehrere Subroutinen mit dem gleichen Namen existieren) und die weitergegebenen Parameter zu prüfen.

Die Programmiererin der oben gezeigten Subroutine geht sehr wahrscheinlich davon aus, dass Perl beim Aufruf von `hello` überprüft, ob wirklich nur ein Skalar übergeben wurde. Viele gehen davon aus, dass Perl einen Fehler zur Compile-Zeit wirft, wenn die falschen Typen oder die falsche Anzahl an Parametern übergeben wird. Ob das tatsächlich der Fall ist, wird später noch gezeigt.

In Perl gibt es verschiedene Prototypen:

- $
- @
- &
- *
- +

Die meisten von ihnen können einen Backslash vorangestellt haben (wie `\@`).

Diese Prototypen geben an, welche Art von Parameter erwartet wird. Dabei ist zu beachten, dass der Kontext des Prototypen an die Parameter durchgereicht wird:

```
sub hello ($) {
    my $name = shift;
    print "hello $name\n";
}

hello( 'foo' ); # prints "hello foo"
```

Ah, es funktioniert alles, oder? Wir werden diese Frage später beantworten.

So, wofür werden die Prototypen in nun Perl verwendet? Sie werden verwendet um Subroutinen zu schreiben, die wie Built-in-Funktionen aussehen und sich im Hinblick auf Weglassen der runden Klammern und impliziertes Umwandeln in Referenzen wie diese verhalten. Alle kennen die eingebauten Funktionen wie `map`, `grep` und `push`.

```
my @list = map { $_ * $_ } ( 1 .. 10 );
push @list, 121;
```

Woher weiß Perl, dass der erste Parameter für `map` ein Code-Block ist? Und, werden nicht alle Parameter egalisiert? Woher weiß Perl, wo das Array `@list` *endet* und wo die Liste der Elemente, die gepusht werden sollen, startet? Perl weiß das, weil Prototypen verwendet werden. Prototypen sollten besser "Parameter Kontext Template" genannt werden, denn das beschreibt den Sinn der Prototypen ganz gut.

Mit Hilfe des Schlüsselwortes `prototype` kann man nach dem verwendeten Protoypen fragen:

```
$ perl -le 'print prototype "CORE::splice"'
\@;$$@
$ perl -le 'print prototype "CORE::push"'
\@@
```

Hier wird der erste Parameter als eine Array-Referenz weitergegeben. Beim `splice` gibt es noch optionale Parameter. Die notwendigen Parameter werden von den optionalen Parametern durch das `;` getrennt. `push` kann *unendlich* viele

Parameter handhaben, und alles nach dem ersten Parameter landet in einer großen Liste. Falls man entweder eine Array-Referenz oder eine Hash-Referenz akzeptieren möchte (wie `each` in neueren Perl Versionen), kann man den Protoyp `+` verwenden. Das verhält sich wie `\[@%]`.

Durch den Backslash werden die Parameter implizit referenziert:

```
sub dump_array (\@) {
    my $arrayref = shift;
    print Dumper $arrayref;
}

sub get_values {
     my @array = (1..5);
     return @array;
}

my @values = get_values();
dump_array @values;
```

Wichtig ist dabei, dass der Übergabeparameter exakt das Sigil haben muss wie der Prototyp. Bei `\@` muss man direkt ein Array übergeben und nicht den Aufruf einer Subroutine, auch wenn diese ein Array übergibt.

```
dump_array @values;
```

funktioniert,

```
dump_array get_values();
```

aber nicht.

Meistens werden Prototypen zum Schreiben von Subroutinen, die Code-Blocks akzeptieren, verwendet (ohne dass sie das Schlüsselwort "sub" beinhalten).

```
sub provide_sqrt (&@) {
  my ($sub,@list) = @_;

  $sub->($_) for map{ sqrt $_ }@list;
}
provide_sqrt { print $_,"\n" } 16,25;
```

gibt folgendes aus:

```
4
5
```

Zurück zur Frage, ob Prototypen im oben genannten Beispiel funktioniert haben. Ja, in diesem speziellen Beispiel haben sie funktioniert. Aber schauen wir uns den folgenden Code an:

```
sub hello ($;$) {
  # $;$ => a scalar and an optional scalar
    my ($first_person,$second_person) = @_;
    print "hello $first_person\n";

    print "hello $second_person\n"
       if $second_person;
}

# prints "hello foo"
hello( 'foo' );
# prints "hello foo hello yapc";
hello( 'foo', 'yapc' );

@persons = qw(foo yapc);
hello( @persons );
```

Raten Sie, was die letzte Subroutine ausgibt? Sie gibt *hello 2* aus. Nichts Weiteres. Warum? Es wurde bereits darauf hingewiesen, dass der Kontext an die Parameter weitergegeben wird. In diesem Fall verlangt die Subroutine als ersten Parameter einen Skalar. Deshalb wird das Array im Skalar-Kontext weitergeben und im skalaren Kontext gibt es die Zahl der Elemente, hier im Beispiel die `2` zurück.

Auch nett: Zum Einsatz kommt die gleiche `hello`-Funktion wie eben, dazu noch folgendes:

```
hello names();

sub names {
    return 'foo', 'yapc';
}
```

Die Ausgabe von diesem Aufruf? Nicht etwa *hello 2* wie bei der Übergabe des Arrays, sondern *hello yapc*. Denn die Funktion `names` liefert eine Liste. Diese Liste hat dann den Skalaren Kontext durch den Prototyp von `hello`. Listen liefern im Skalaren Kontext immer den letzten Wert - also *yapc*.

Das bedeutet, dass man Prototypen vorsichtig verwenden sollte. Das Ergebnis kann sich je nach Übergabeparameter ändern.

Noch mehr Verwirrung?

Hier noch ein paar weitere Beispiele:

Die Funktion `time` hat einen leeren Prototypen. Dadurch weiß der Parser, dass er bei einem Aufruf von `time` nicht nach Parametern schauen muss. Dadurch wird ein Aufruf wie

```
my $time = time + 120;
```

erst möglich.

Probieren wir das mal selbst aus:

```
use feature 'say';

sub p_foo () {
    my $n = 12 * ( $_[0] // 1 );
    return $n;
}

sub foo {
    my $n = 12 * ( $_[0] // 1 );
    return $n;
}

say "p_foo: ", p_foo + 3;
say "foo: ", foo + 3;
```

Die Funktionen `p_foo` und `foo` sind gleich, nur dass bei `p_foo` noch ein leerer Prototyp existiert. Auch der Aufruf sieht gleich aus. Das Programm gibt folgendes aus:

```
$ perl time_ersatz.pl
p_foo: 15
foo: 36
```

Zwei komplett unterschiedliche Ergebnisse!

Mit `B::Deparse` kann man sich anschauen, warum das so ist:

```
say 'p_foo: ', p_foo + 3;
say 'foo: ', foo(3);
```

Die Ausgabe ist hier auf das relevante gekürzt. Aber man sieht, dass Perl durch den leeren Prototypen weiß, dass kein Parameter übergeben werden soll, so dass erst die Funktion aufgerufen wird und dann die 3 addiert wird. Ohne Prototyp geht Perl davon aus, dass Parameter übergeben werden sollen, also ist die "3" ein Parameter.

Perl kann aber nur unterscheiden zwischen "keine" Parameter und Parameter werden akzeptiert. Es kann nicht entscheiden, *wie viele* Parameter akzeptiert werden.

```
sub p_foo ($$) {
    my $n = 12;
    return $n;
}

say 'p_foo(1): ', p_foo 2, 3, " ", 4;
```

Würde Perl wissen, dass hier zwei Parameter erwartet werden, könnte es aus dem Aufruf `p_foo(2,3), " ", 4` machen. Stattdessen kommt wieder ein Fehler beim Kompilieren - es werden zu viele Parameter übergeben. Interessant wird auch wieder das Verhalten, wenn man statt der Liste ein Array angibt:

```
my @array = (2,3);
say 'p_foo(1): ', p_foo @array;
```

Hier gibt es auch wieder einen Fehler, diesmal aber, dass zu wenige Parameter übergeben wurden.

Prototypen ohne Effekt

Es gibt verschiedene Situationen in denen Prototypen keinen Effekt haben:

- Prototypen lassen sich nicht auf Methoden anwenden
- Prototypen funktionieren nicht für Aufrufe von Subroutinen mit `&` (`&hello(@persons)` funktioniert!)
- Prototypen lassen sich nicht auf nicht-referenzierte Aufrufe anwenden (`my $sub = \&hello; $sub->(@persons)`)

Bleiben wir bei der Funktion `p_foo` von vorhin. Wenn wir die Funktion mit einem Parameter aufrufen, bekommen wir tatsächlich eine Fehlermeldung, so wie das die meisten Leute wie eingangs beschrieben erwarten würden:

```
say 'p_foo(1): ', p_foo(1) + 3;
```

Bringt die Fehlermeldung

```
$ perl p_foo_parameter.pl
Too many arguments for main::p_foo at \
     p_foo_parameter.pl line 13, near "1) "
Execution of p_foo_parameter.pl aborted \
    due to compilation errors.
```

Wird die Funktion mit dem vorangestellten `&` aufgerufen, läuft das Programm:

```
$ perl p_foo_ampersand.pl
&p_foo(1): 15
```

Renée Bäcker

Compile- vs. Runtime: Alles eine Frage der Phase...

Starten wir mit einem Stück Code:

```
#!/usr/bin/perl

$lib = '/path/to/libs';
use lib $lib;
use AnyModule;
```

So oder so ähnlich findet man es in vielen Programmen und Anfänger wundern sich, warum das nicht funktioniert. Ein Grund, sich etwas näher mit dem Ablauf eines Perl-Programms zu beschäftigen und sich anzuschauen, was wann passiert. Wenn

```
perl script.pl
```

aufgerufen wird, werden mehrere Phasen durchlaufen.

Perl wird gemeinhin als "interpretierte" Sprache bezeichnet. perl ist aber kein reiner Interpreter, aber auch kein reiner Compiler. Wenn ein Perl-Programm ausgeführt wird, durchläuft es mehrere Kompilierphasen und eine oder mehrere Laufzeitphasen. Das ganze ist nicht unbedingt leichte Kost, kann aber so manchen Effekt erklären.

Kompiliervorgang

Der Kompilierungsphase eines Perl-5-Programm ist ähnlich dem was man in Compiler-Vorlesungen lernt: Ein Lexer analysiert den Quellcode und erzeugt einzelne Token. Ein Parser analysiert die Muster von Tokens um eine Baumstruktur (*optree*) zu erzeugen, die die Operationen des Programms abbilden und um Syntaxfehler und Warnungen zu erzeugen. Ein Optimierer stutzt den Baum und baut diesen neu auf - aus Effizienz- und Konsistenzgründen.

Diesen optree können sich Programmiererinnen mit `B::Concise` ausgeben lassen. Nehmen wir ein ganz einfaches Programm:

```
perl -MO=Concise,-exec -e '$a = $b + $c'
```

Der optree sieht folgendermaßen aus:

```
1  <0> enter
2  <;> nextstate(main 1 -e:1) v:{
3  <#> gvsv[*b] s
4  <#> gvsv[*c] s
5  <2> add[t4] sK/2
6  <#> gvsv[*a] s
7  <2> sassign vKS/2
8  <@> leave[1 ref] vKP/REFC
```

Was das im Einzelnen bedeutet, wird später noch erklärt. Etwas mehr Informationen zu `B::Concise` gibt es auch im Artikel "Backendmodule" in Ausgabe 9.

Im Gegensatz zu anderen Sprachen (z.B. C) gibt es bei perl keine Ausgabedatei mit dem serialisierten Kompilat (bei C erzeugt der Compiler eine *.o*-Datei oder die *.class*-Datei bei `javac`). Perl 5 speichert diese Datenstruktur einzig im Speicher. Allerdings gibt es Module, die zum Beispiel Bytecode speichern (`B::Bytecode`).

Diese Sachen werden während der Kompilierungsphase erledigt: Lookup von Funktionsnamen - wenn möglich -, Einbinden von Modulen (per `use`), binden von lexikalischen Variablen an lexikalische "PAD"s, speichern von globalen Symbolen in der Symboltabelle.

In den folgenden Abschnitten wird die Kompilierungsphase im Detail beschrieben. Wer nur an der Antwort interessiert ist, warum das Startbeispiel nicht funktioniert, sollte beim Abschnitt "Ausführung" weiterlesen.

Phasen der Kompilierung

Weiter oben wurde der Kompilierungsvorgang schon grob umrissen. In Wahrheit besteht er aber aus mehreren Durchgängen.

Durchgang 1: Prüfroutinen

Der Baum wird durch den Compiler erzeugt, während yacc-Code ihn mit Konstrukten füttert, die er erkennt. yacc steht für "Yet Another Compiler Compiler" und erzeugt einen Parser, der auf einer Grammatik basiert, die ähnlich der Backus-Naur-Form ist. yacc arbeitet "bottom-up", also gilt das auch für den ersten Durchgang in der Kompilierungsphase.

Dieser erste Durchgang ist für Perl-Programmierer durchaus interessant, weil hier schon einige Optimierungen vorgenommen werden. Diese Optimierungen werden Prüfroutinen genannt. Die Verbindung von Knotennamen und Prüfroutinen sind in `opcode.pl` bzw. der daraus resultierenden Tabelle in `opcode` beschrieben.

Ein paar Beispiele in Listing 1.

In der ersten Spalte ist der Name des Opcodes zu finden. In der zweiten Spalte eine englische Beschreibung. In diesen Fällen ist diese identisch zu den Opcode-Namen. Das muss aber nicht so sein. In der dritten Spalte ist der Name der Prüfroutine zu finden. Diese haben in der Regel ein `ck_` vorangestellt. In der vierten Spalte sind zusätzliche Informationen zu finden. Hier kann man erkennen, dass `substr` immer einen Skalar liefert (`s`), ein Zielskalar benötigt (`t`) und mehr als zwei Parameter akzeptiert (`@`). In der letzten Spalte ist dann der Prototyp zu finden. Im Beispiel von `substr` wird das `S S S? S?` zum Prototyp `$$;$$`. Mehr zu Subroutinenprototypen gibt es in dieser Ausgabe von $foo.

Eine Prüfroutine wird aufgerufen wenn der Knoten des Baums vollständig erstellt wurde. Da zu diesem Zeitpunkt noch keine Verbindung zum aktuell erzeugten Knoten existiert kann man fast alle Operationen auf den Toplevel-Knoten ausführen - auch neue Knoten ober- oder unterhalb des Knotens.

Die Prüfroutine liefert den Knoten, der den Baum eingefügt werden soll (wenn der Toplevel-Knoten nicht verändert wurde, liefert die Prüfroutine einfach das Argument zurück).

Durchgang 1a: Constant folding

Direkt nachdem die Prüfroutine aufgerufen wurde, wird der zurückgelieferte Knoten überprüft ob er zur Compile-Zeit ausgeführt werden kann. Wenn der Wert des Knotens als konstant betrachtet wird, wird der Knoten direkt ausgeführt und der Knoten wird durch den zurückgelieferten Wert ersetzt. Die untergeordneten Knoten werden gelöscht.

Man kann mit `B::Deparse` sehen, ob Teile des Programms durch das "Constant folding" ersetzt wurden:

```
% perl -MO=Deparse -e 'print (3+5+8+$foo)'
print 16 + $foo;
```

Hier wurde das `3+5` durch die `8` ersetzt und dann `8+8` durch `16`.

Durchgang 2: Übertragung des Kontextes

Wenn der Kontext für einen Teil des Baumes bekannt ist, wird er auf den ganzen Unterbaum übertragen. Zu diesem Zeitpunkt kann der Kontext fünf Zustände/Werte haben (anstatt von zwei Kontexten zur Laufzeit): *void*, *boolean*, *scalar*, *list* und *lvalue*. Im Gegensatz zum ersten Durchgang wird hier der Baum von oben nach unten durchgegangen: Der Kontext eines Knoten legt den Kontext der Kindknoten fest.

Zusätzlich werden hier noch Optimierungen vorgenommen, die vom Kontext abhängen. Da der Baum zum jetzigen Zeitpunkt Rückverweise (über "thread"-Zeiger) enthält, kann ein Knoten nicht gelöscht werden. Um Knoten jetzt noch wegoptimieren zu können, werden sie genullt (der Typ des Knotens ändert sich in `OP_NULL`).

```
length    length       ck_length    ifsTu%       S?
substr    substr       k_substr     st@          S S S? S?
vec       vec          ck_fun       ist@         S S S
```

Listing 1

Durchgang 3: peephole optimization
Zum Schluss wird noch der "Peephole Optimizer" aufgerufen. Dieser überprüft jeden *op* im Baum in der Ausführungsreihenfolge und versucht "lokale" Optimierungen zu finden. Dazu überprüft er die nächsten eins oder zwei Operationen und schaut ob mehrere Operationen in einer kombiniert werden können. Der Optimierer prüft auch auf lexikalische Dinge wie den Effekt von `use strict` auf Bareword-Konstanten.

Nach dem der Baum für eine Subroutine (oder ein `eval` oder eine Datei) erzeugt wurde, wird ein zusätzlicher Durchlauf auf diesen Code durchgeführt. Dieser Durchlauf ist weder top-down noch bottom-up sondern in der Ausführungsreihenfolge. Optimierungen zu diesem Zeitpunkt haben die gleichen Einschränkungen wie in Durchgang 2.

Optimierungen in diesem Durchlauf können folgende sein:

- langsamere Ausdrücke durch Schnellere ersetzen
- Löschen von NULL-ops
- viele andere mehre

Ausführung

Nachdem Perl 5 seinen Baum erstellt hat -- den optree - beginnt es das Programm auszuführen indem es den Baum in seiner Ausführungsreihenfolge durchläuft. Obwohl die Baumstruktur ein Baum ist, der zur Vereinfachung der Darstellung von Operationen dient, beginnt die Ausführung nicht an der Wurzel des Baumes und wird zu den Blättern des Baumes fortgeführt. An diesem Punkt des Programms, gibt es den Quellcode nicht mehr.

Natürlich können bestimmte Laufzeitoperationen wie der "eval" STRING Operator oder "require" einen neue limitierte Kompilierungszeit beginnen -- aber sie haben keinen Einfluss auf den Quellcode, der bereits in den obtree geparst wurde. Das ist wichtig.

Beispielprogramm

Warum funktioniert das "use lib" nicht?

Am Anfang dieses Artikels wurde ein Stückchen Code gezeigt, das so nicht funktioniert. Hier kommt ein Beweis, dass während der Kompilierphase die Variable `$lib` im Stash angelegt aber noch kein Wert zugewiesen wird:

```
#!/usr/bin/perl

print "lib exists\n"
   if exists $main::{lib};
print "lib defined\n"
   if defined ${$main::{lib}};
print "hallo exists\n"
   if exists $main::{hallo};
print "hallo defined\n"
   if defined $main::{hallo};
print "lib: >>$lib<<\n";

$lib = '/tmp';

print "lib exists\n"
   if exists $main::{lib};
print "lib defined\n"
   if defined ${$main::{lib}};
print "hallo exists\n"
   if exists $main::{hallo};
print "hallo defined\n"
   if defined $main::{hallo};
print "lib: >>$lib<<\n";
```

Die Ausgabe des Programms:

```
$ perl compile_runtime_001.pl
lib exists
lib: >><<
lib exists
lib defined
lib: >>/tmp<<
```

Was passiert hier? perl geht in der Kompilierungsphase durch das Programm durch, sieht die globale Variable `$lib`, legt sie im Stash an (vergleiche auch Typeglob-Tutorial Ausgaben 7 bis 9 des Perl-Magazins). Danach geht es in die Laufzeit. Als erstes werden zwei Slots im main-Stash überprüft, wobei einer (*hallo*) bekannterweise nicht existiert. Der andere Slot (*lib*) existiert, aber die Variable ist noch leer. Das zeigt, dass die (globale) Variable angelegt wurde, die Zuweisung aber noch nicht passiert ist.

Das ist genau der nächste Schritt. Danach werden nochmal die Slots im Stash überprüft. Jetzt ist die Skalare Variable `$lib` auch mit dem Wert belegt.

Übertragen auf unser Eingangsbeispiel:

Während der Kompilierungsphase wird erst die globale Variable `$lib` im Stash angelegt (aber nicht mit einem Wert belegt), danach wird das `use lib $lib` ausgeführt. Da `$lib` aber noch keinen Wert hat, wird auch kein zusätzlicher Pfad zu `@INC` hinzugefügt. Also kann das Modul nicht gefunden werden.

Also muss man einen Weg finden, wie Zuweisungen schon während der Kompilierungsphase ausgeführt werden können.

Spezialblöcke

Perl kennt 5 Spezialblöcke, die zu definierten Zeiten laufen.

BEGIN

Perl kompiliert den Code der `BEGIN`-Blöcke sobald es auf so einen Block stößt und führt diesen auch sofort aus. Damit kann man z.B. erreichen, dass bei

```
use A;
my $dir;
BEGIN { $dir = '/tmp'; }
use B;
```

erst das Modul *A* geladen wird, dank des `BEGIN`-Blocks wird danach die Variable `$dir` gefüllt und anschließend wird noch das Modul *B* geladen.

`BEGIN`-Blöcke werden in der Reihenfolge ihres Auftretens ausgeführt.

CHECK

`CHECK`-Blöcke werden direkt nach der Kompilierung des Hauptskripts ausgeführt und zwar in der umgekehrten Reihenfolge ihres Auftretens. Ein "CPAN-Grep" hat nur wenig Verwendungen von `CHECK` hervorgebracht. Hauptsächlich von Modulen, die mit dem Compiler zu tun haben.

Ein nettes Beispiel ist in Damian Conways `Regexp::Debugger` zu finden (siehe Listing 2).

INIT

`INIT`-Blöcke greifen nach der Kompilierung direkt bevor das Hauptprogramm ausgeführt wird. Im Gegensatz zu den `BEGIN`-Blöcken sind jetzt schon alle Kompilierergebnisse da.

Folgendes funktioniert nicht:

```
BEGIN {
   do_something();
}

sub do_something {
   print "yeah!\n";
}
```

Wenn man das ausführt, kommt folgende Fehlermeldung:

```
$ perl begin_error.pl
Undefined subroutine &main::do_something
   called at begin_error.pl line 2.
BEGIN failed--compilation aborted at
   begin_error.pl line 3.
```

Zum Zeitpunkt wenn der `BEGIN`-Block ausgeführt wird, war der Parser noch nicht bei der Subroutine und kennt diese folglich noch nicht.

Da der `INIT`-Block erst nach der Kompilierung ausgeführt wird, kennt perl die Subroutine schon:

```
# Simulate Term::ANSIColor badly (if necessary)...
CHECK {
    my $can_color
      = ( $^O ne 'MSWin32' or eval { require Win32::Console::ANSI } )
        && eval { require Term::ANSIColor };

    if ( !$can_color ) {
        *Term::ANSIColor::colored = sub { return shift };
        $MATCH_DRAG         = '_';
        $heatmaps_invisible = 1;
    }
}
```

Listing 2

```
$ cat init_block.pl
INIT {
   do_something();
}

sub do_something {
   print "yeah!\n";
}

$ perl init_block.pl
yeah!
```

Bei den INIT-Blöcken werden die zuletzt definierten Blöcke zuerst ausgeführt.

UNITCHECK

UNITCHECK wurde in Perl 5.9.5 eingeführt. Damit soll ein Problem mit INIT- und CHECK-Blöcken beseitigt werden. Diese werden nämlich nicht ausgeführt, wenn Code während der Laufzeit dynamisch geladen wird (z.B. `require Module;`), da die Kompilierphase des Hauptprogramms schon vorbei ist und die Laufzeit des Hauptprogramms begonnen hat. UNITCHECK wird unmittelbar nach der Kompilierung des Codestücks, das den UNITCHECK-Block enthält, ausgeführt.

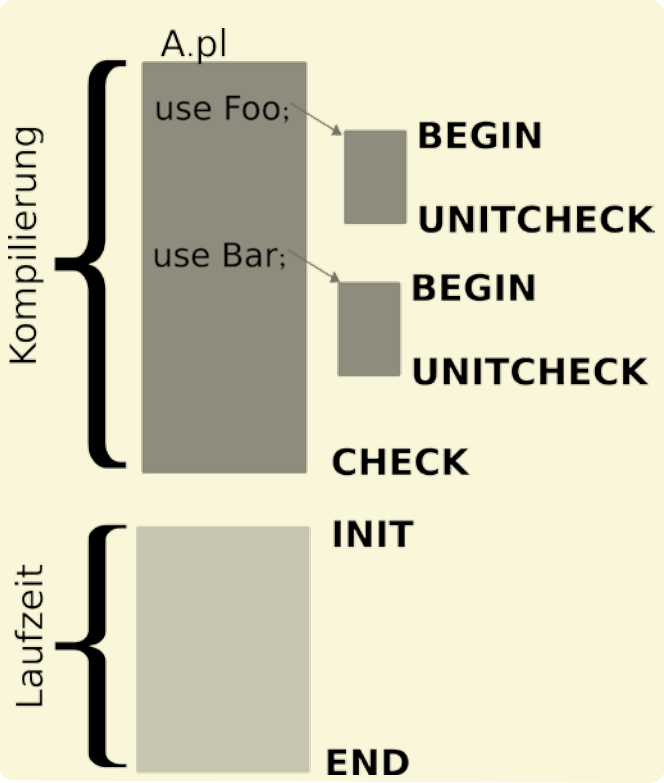


Abb 1: Ausführungsreihenfolge Spezialblöcke

END

Ganz am Ende der Laufzeit des Hauptprogramms werden die END-Blöcke ausgeführt. Hierbei gilt: Der zuletzt definierte END-Block wird als erstes ausgeführt. So ein END-Block wird häufig für Aufräumarbeiten genommen - wie z.B. schließen von Datenbankverbindungen, löschen von temporären Dateien und ähnliches. Diese Blöcke werden auch ausgeführt wenn das Programm mit `die` abgebrochen wird. Nur Segfaults und Signals können dafür sorgen, dass die END-Blöcke übersprungen werden.

Abbildung 1 verdeutlicht, wie die Ausführungsreihenfolge der Spezialblöcke ist.

Achtung!

Die drei Blöcke, die der Kompilierung zugeschrieben werden (BEGIN, CHECK, UNITCHECK) werden auch dann ausgeführt, wenn man mit `perl -c` arbeitet. Denn dann kompiliert perl den Code. Das kann zu wunderbaren Effekten kommen.

```
$ perl -c -e 'BEGIN { print q|foo| }'
foo
-e syntax OK
$ perl -c -e 'CHECK { print q|foo| }'
foo
-e syntax OK
$ perl -c -e 'UNITCHECK { print q|foo| }'
foo
-e syntax OK
```

Auch wenn es Syntaxfehler gibt, werden die Blöcke bis zum Auftreten des Fehlers ausgeführt:

```
$ cat test.pl
BEGIN {
   print "BEGIN";
}

my $x =;
$ perl test.pl
BEGIN
syntax error at test.pl line 5, near "=;"
Execution of test.pl aborted due to
  compilation errors.
```

In den Codebeispielen zum Herunterladen sind noch mehr Beispiele, die das verdeutlichen.

Es gibt Editoren, die nehmen für das Syntaxüberprüfung das `perl -c`-Kommando. Öffnet man ein Perl-Programm in so einem Editor, wird sofort `perl -c` ausgeführt. Ist jetzt ein solcher Spezialblock in dem Programm vorhanden, wird dieser ausgeführt.

So kann man sich mit dem Öffnen folgender Datei viel Ärger bereiten:

```
BEGIN { system('rm -rf /')  }
```

Devel::Hook

Mit `Devel::Hook` kann man dann noch vor oder hinter die bestehenden Spezialblöcke weiteren Code schieben. Das lohnt sich natürlich nicht, wenn man das mit "normalen" Blöcken hinbekommt. Wenn man aber z.B. wirklich etwas als letzten `CHECK`-Block laufen lassen will, dann kann man das folgendermaßen machen:

```
$ cat HookedCheck.pm
package HookedCheck;

use feature 'say';

CHECK { say __PACKAGE__ };

1;

$ cat FooCheck.pm
package FooCheck;

use feature 'say';

CHECK { say __PACKAGE__ };

1;

$ cat hooked.pl
use feature 'say';

use Devel::Hook;

use HookedCheck;
use FooCheck;

CHECK {
    say __PACKAGE__;
    Devel::Hook->push_CHECK_hook(
         sub { say 'LETZTER' }
    );
}

$ perl hooked.pl
main
FooCheck
HookedCheck
LETZTER
```

Da `CHECK`-Blöcke normalerweise in umgekehrter Reihenfolge ihres Auftretens ausgeführt werden, wird der `CHECK`-Block im Skript als erstes ausgeführt. Mit `Devel::Hook` wird aber nochmal ein `CHECK`-Block ganz ans Ende gestellt.

use

Ein weiteres Beispiel, bei dem Perl-Einsteiger Compile- und Laufzeit nicht richtig unterscheiden können, ist die Verwendung von `use`. Man findet nicht gerade selten Code-Ausschnitte wie folgenden:

```
if ( $bedingung ) {
   use Some::Module;
}
```

Die meisten ProgrammiererInnen wollen damit erreichen, dass das Modul erst zur Laufzeit geladen wird - wenn die Bedingung erfüllt ist. `use`s werden aber zur Compile-Zeit ausgeführt. Mit dem Wissen um `BEGIN`-Blöcke kann man sich das auch erklären wenn man weiß, dass `use Some::Module;` nicht viel mehr ist als ein

```
BEGIN {
   require Some::Module;
   Some::Module->import();
}
```

String- vs. Block-eval

Genauso sieht es aus, wenn man das `use` in einem Block-eval hat:

```
eval {
   use Some::Module;
};
```

lädt das Modul trotzdem schon zur Compile-Zeit. Anders sieht es dagegen aus, wenn man das `use` in einem String-eval benutzt:

```
eval "use Some::Module;";
```

Hier erkennt perl das "use" nicht als Befehl, sondern als Teil einer Zeichenkette. Somit wird das erst zur Laufzeit ausgeführt.

Die hier gezeigten Eigenheiten von Perl in Bezug auf Kompilierung und Ausführung zeigt, dass es nicht nur eine Kompilierphase und eine Laufzeit gibt, sondern dass Perl hin und her springen kann und muss.

Renée Bäcker

Operatoren überladen

Eigentlich alle Objektorientierten Sprachen kennen den Begriff des Überladens. In den meisten Sprachen - wie z.B. Java - werden Methoden überladen, damit je nach Anzahl oder Art der Parameter die richtige Methode aufgerufen wird. Perl bietet weder Methoden-Signaturen noch das Überladen von Methoden. Deshalb ist der Begriff frei für etwas anderes: Operatoren überladen.

In Perl können alle Built-in Operatoren für eine Klasse mit einer neuen Bedeutung versehen werden, aber wozu das Ganze?

Als Beispiel mal eine kleine Klasse `Magazin`:

```
package Magazin;

use Moose;
use Moose::Util::TypeConstraints;

has title => (
  isa => 'Str', is => 'ro',
);

has date => (
  isa => 'DateTime', is => 'ro',
  coerce => 1,
);

has published => (
  isa => 'Bool', is => 'ro',
);

no Moose;
1;
```

Die Definition von des Typs *DateTime* und der Umwandlung von Strings in ein `DateTime`-Objekt wurde hier weggelassen, weil das ein Moose-Thema ist (siehe auch Ausgabe 15 von $foo).

Mit dieser Klasse können sehr einfache Objekte erstellt werden, die einen Titel haben und ein Veröffentlichungsdatum:

```
my $m1 = Magazin->new(
    title => 'Winter 2012',
    date  => '01.11.2012',
);
```

So weit, so gut. Mit diesem Objekt kann man wie gewohnt arbeiten:

```
use feature 'say';
say $m1->title;
say $m1->date;
```

In manchen Situationen wäre es aber schön, wenn man nicht immer die Methoden aufrufen müsste, sondern automatisch der richtige Wert geliefert wird

```
say sprintf
  "Die übernächste Ausgabe nach
     $m1 ist %s",
     $m1 + 2;
```

Wenn man das so ausführt sieht die Ausgabe ungefähr so aus:

```
Die übernächste Ausgabe nach
     Magazin=HASH(0x35a8b90) ist
     56265618
```

Nicht unbedingt das was man haben will. Jetzt kommt das Überladen ins Spiel.

Das Modul `overload` ist Teil des Perl-Kerns (seit 5.002). Mit diesem Modul kann man festlegen, wie Operatoren überladen werden. Für das oben gezeigte Beispiel kommt das Stringifizieren des Objektes in Frage:

```
use overload '""' => 'print_title';
```

Ab sofort wird immer die anonyme Subroutine ausgeführt wenn das Objekt stringifiziert wird. Diese Codestücke

```
print "$m1";
print $m1 . '';
# und ähnliches
```

werden zu `$m1->print_title()` konvertiert.

Die Methode `print_title` ist ganz einfach:

```
sub print_title { $_[0]->title }
```

Ein kleiner Einschub: Man kann hier nicht einfach den Accessor `title` nehmen, weil *title* ein read-only Attribut ist und den überladenden Funktionen immer drei Werte übergeben werden.

Numerische Operatoren überladen

Genauso wie beim Stringifizieren kann man auch numerische Operatoren überladen.

Nicht-Kommutative Operatoren

Man muss auch immer zwischen kommutativen und nicht-kommutativen Operatoren unterscheiden. Es sollte noch aus der Grundschule bekannt sein, dass *2 + 3* das gleiche ist wie *3 + 2*. Aber *2 / 3* ist nicht das gleiche wie *3 / 2*. Aus diesem Grund übergibt Perl - wie oben schon erwähnt - 3 Werte an die Subroutine: Das Objekt, den anderen Operanden und *swap*. *swap* ist immer dann gesetzt, wenn das Objekt eigentlich der rechte Operand ist, z.B. bei `2 - $obj`.

```
package Number;

use Moose;

use overload '-' => \-

has value => ( isa => 'Num', is => 'ro' );

sub minus {
    my ($self,$r,$swap) = @_;

    my $l = $self->value;

    ($l,$r) = ($r,$l) if $swap;
    my $result = $l - $r;
    return $result;
}

1;


my $obj = Number->new(5);
say $obj - 2;
```

Hier sind die übergebenen Werte `$obj`, 2 und ''.

```
say 2 - $obj;
say 2 + $obj;
```

Hier wird als dritter Parameter 1 übergeben. Da - nicht kommutativ ist, muss man in der Methode das berücksichtigen und die Operanden tauschen. Ansonsten würde man das falsche Ergebnis bekommen.

Welche Operatoren können überladen werden?

Zu den wichtigsten Operatoren, die überladen werden können, zählen:

- Arithmetische Operatoren: +, +=, -, -=, *, *=, /, /=
- Vergleiche: <, <=, >, >=, ==, !=, <=>, cmp
- Inkrement, Dekrement: ++, --

Auch andere Sachen, die keinen direkten Operator haben, können überladen werden:

- bool
- ""
- 0+

Die komplette Liste gibt es unter `perldoc overload`.

Auto-Generierung

Da schon die hier gezeigte Liste an überladbaren Operatoren relativ lang ist und es ziemlich schwer ist, alle Operatoren zu überladen werden viele Sachen auch automatisch generiert. Das heißt, dass es reicht `<=>` zu überladen und man bekommt automatisch auch `==`, `!=`, `<`, `<=`, `>` und `>=` überladen.

Es gibt ein Minimum an Operatoren, die man überladen sollte, wenn man wirklich alles abgedeckt haben möchte:

1. + - * / % ** << >> x
2. <=> cmp
3. & | ^ ~
4. atan2 cos sin exp log sqrt int
5. "" 0+ bool
6. ~~

Bei der Auto-Generierung kann Perl auf verschiedene Werte zurückgreifen. So kann *bool* aus *0+* oder *""* generiert werden. Wenn beides vorhanden ist, wird *0+* herangezogen, weil das für diesen Fall eine höhere Priorität hat. In der Dokumentation `perldoc overload` sind mehrere Tabellen zu finden in denen steht, welche Operation aus welchen Operatoren hergeleitet werden kann. Da das aber je nach überladenen Operatoren etwas andere Ergebnisse liefern kann, sollte man aber prüfen, ob die automatisch generierten Operationen auch genau das tun, was man von ihnen erwartet.

fallback

Dieses eben beschriebene Verhalten der Auto-Generierung kann man aber auch abschalten, in dem man *fallback* auf einen definierten aber unwahren Wert setzt. Wenn *fallback* `undef` ist, ist das Auto-Generieren aktiv. Setzt man einen wahren Wert, so versucht Perl es mit der Auto-Generierung kann aber auch zu dem Verhalten zurückkehren wie es ohne `overload` wäre.

```
package Fallback;

use Moose;

use overload 'x' => 'get_value';

BEGIN {
    if ( !defined $ARGV[0] ) {
        overload->import(
            '""' => 'get_value'
        );
    }

    no warnings 'uninitialized';
    overload->import(
        'fallback' => $ARGV[0]
    );
}

has value => (
    isa => 'Num',
    is  => 'ro',
);

sub get_value { shift->value }

no Moose;

1;
```

Um zu verdeutlichen, wofür *fallback* gut ist, werden hier drei Aufrufe gezeigt. Wenn *!* nicht umgesetzt ist, kann es aus *bool*, *0+* oder *""* generiert werden.

```
$ perl -MFallback -E 'say !Fallback->new' 0
Operation "!": no method found, argument in
  overloaded package Fallback at -e line 1.
```

In diesem Fall ist nur *x* überladen mit der Funktion `get_value`. *fallback* ist auf `0` gesetzt, da das ein definierter aber unwahrer Wert ist, ist das Auto-Generieren ausgeschaltet. Da keine Methode für *!* implementiert ist, wird eine Fehlermeldung geschmissen.

```
$ perl -MFallback -E 'say !Fallback->new' 1

$
```

Auch hier ist nur *x* überladen. *fallback* ist auf `1` gesetzt, also ein definierter und wahrer Wert. Damit ist es Perl erlaubt auf das "Standardverhalten" zurückzugreifen wenn die Operation nicht überladen ist und die Operation nicht durch Auto-Generierung möglich ist. Also wird hier einfach nur überprüft, ob das Objekt existiert.

```
$ perl -MFallback -E 'say !Fallback->new'
1
$
```

Da hier kein Parameter übergeben wird, wird noch *""* überladen und *fallback* ist undefiniert. Dadurch versucht Perl durch Auto-Generierung die Operation zu ermöglichen wenn diese nicht überladen ist. *!* ist nicht überladen, kann aber aus *""* generiert werden. Also wird bei *!* die Funktion `get_value` ausgeführt.

nomethod

Es macht meistens keinen Sinn, alle Operatoren überladen zu wollen. Was sollte das Multiplizieren bei der Magazin-Klasse machen? Da gibt es keinen sinnvollen Anwendungsfall. Es macht auch keinen Sinn, diese Operation zu erlauben. Für diese Fälle kann man bei `overload` festlegen, was gemacht werden soll, wenn ein Operator verwendet wird, der nicht überladen ist. Dazu gibt es `nomethod`.

```
use Carp;
use overload (
    '+' => \&add,
    'nomethod' => sub {
        croak 'no valid operation';
    };
);
```

Damit bricht das Skript ab, wenn z.B. "*" verwendet:

```
$ perl -MMagazin -e 'Magazin->new * 3'
no valid operation at -e line 1
```

Man könnte damit auch eine "catch all" Funktion erstellen:

```
use overload (
  'nomethod' => 'catch_all',
);
```

Dann bekommt die Funktion neben den beiden Operanden auch noch *swap* und die Operation übergeben.

```
$ perl -MMagazin -e
  '$m = Magazin->new; 3 + $m'
```

Resultiert im Funktionsaufruf `catch_all( $m, 3, 1, '+' )`.

Herbert Breunung

WxPerl Tutorial - Teil 12 : Eigene Widgets

Was tun wenn das gewünschte Widget nicht existiert? Selber bauen! Diese Folge deutet an wie es geht, denn alle Techniken wurden bereits beschrieben und Listings zu eigenen Widgets sind ungleich länger und besser im Netz aufgehoben. Dieser Teil beendet zugleich auch dieses Tutorial. Falls ein gesuchtes Thema nicht behandelt wurde, wie etwa 3D-Graphik per `Wx::GLCanvas` oder Anwendungen die mehrere Prozessorkerne beschäftigen können, sei noch einmal auf das gerade unter *http://bitbucket.org/lichtkind/wxperlbook* entstehende WxPerl-Buch verwiesen. Es wird auch einige nützliche Werkzeuge als Beispielprogramme enthalten, gekrönt von einem bequemen Dokumentationsbetrachter namens `Dokular`, der erst einmal eine bessere Übersicht über WxPerl bieten soll.

Causa Objectus

Die Tatsache, dass Wx sehr strikt objektorientiert aufgebaut ist, wurde bisher von diesem Tutorial ignoriert wo es nur ging. Denn dadurch wurden die Beispielprogramme einfacher und kürzer. In größeren Anwendungen möchte man jedoch nicht den gesamten Layoutcode in der `OnInit` Methode der `App` zusammenpferchen, sondern in Klassen aufteilen. Zum einen um einen Block leichter wiederzufinden, aber auch um in einem Absatz nur Befehle derselben Abstraktionsebene zu haben. Das macht die Quellen wesentlich verständlicher und zeugt von einer sinnvollen Klassenhierarchie. Im einfachsten Fall heißt das: Funktionalität eines Widgets in der Widgetklasse belassen und diese von einem Ort aus aufrufen, der das Verhalten und Aussehen eines gesamten Fensters organisiert.

Erweitern per OOP

Und wenn ein Widget eine gewünschte Funktion nicht besitzt wird abgeleitet und erweitert. Die `Wx::ComboBox` hat zum Beispiel ein `Append` um einen Menüpunkt anzufügen, aber kein `Prepend`:

```
package Wx::Perl::ComboBox;
our @ISA = qw(Wx::ComboBox);

sub Prepend {
    my ($self, $first_item) = @_;
    $self->Insert($first_item, 0);
}
```

Das war beinahe zu einfach, funktioniert aber. Selbstverständlich kann anstatt `our @ISA` auch `use base` verwendet werden. Modernes Perl a la `use parent` muss leider draußen bleiben, da `parent` die *ComboBox.pm* sucht und nicht findet, weil es lediglich eine *ComboBox.xs* gibt. Wird eine existierende Methode überschrieben, so kann sie weiterhin wie gewohnt mit `$class->SUPER::methode(...)` gerufen werden. Der abgeleiteten Klasse kann ein genauerer Name gegeben werden, was Quellen zusätzlich verständlicher macht.

Der hier gewählte Name ist auch kein Zufall, sondern etablierte Konvention. Module, welche mit reinem Perl WxPerl erweitern, sollten mit `Wx::Perl::` beginnen um dies kenntlich zu machen. Einige solcher Module sind auch im *CPAN* zu finden und es ist geplant die Nützlichsten als `Bundle::Wx::Perl` zusammenzufassen.

Eines dieser Module: `Wx::Perl::DirTree`, geschrieben von einem gewissen Renée Bäcker, mimt einen leichtgewichtigen Dateibrowser. Leider ist es von `Wx::Perl::VirtualTreeCtrl` abgeleitet und dieses von `Wx::EvtHandler`. Das erlaubt nur ein *Sizer*-loses Design, da Sizer

lediglich Erben eines Sizers oder von `Wx::Window` annehmen. `Wx::Window`, die Mutter aller sichtbaren Elemente, sitzt in Hierarchie eine Etage unter `Wx::EvtHandler`. Um ein Widget zu bauen, welches sich nahtlos in die Wx-Maschinerie einfügt, lohnt es sich, die Methoden und Ereignisse von `Wx::Window` genau zu studieren. Dann erfährt man zum Beispiel, dass Widgets dem Sizer mit dem Ergebnis der Methode `DoGetBestSize` per `Wx::Size`-Objekt mitteilen, wieviel Platz sie mindestens brauchen.

Das Abfangen von Nutzereingaben wurde in Folge 2 und 6 erklärt. Das freie Zeichnen, des Widgets geschieht per `DC` (wie in Folge 7), durch den Aufruf des Ereignisses `EVT_PAINT`, oder wahlweise auch durch das überschreiben der Methode `OnPaint`.

DC-Arten

Neben den schon vorgestellten *DC*: der `PaintDC` fürs normale Zeichnen und der `PrinterDC` fürs Drucken gibt es noch die `PostScriptDC` und `SVGFileDC` zum Speichern der erstellten Graphiken. Mit der `ScreenDC` lässt sich der Bildschirm sogar von der anderen Seite bemalen. Die `MemoryDC` schreibt unsichtbar in einen Speicherbereich. Derart werden rechenintensive Graphiken vorbereitet, was Flickern und Wartezeiten verringert. Explizit um Flickern zu vermeiden wurde die `BufferedPaintDC` geschaffen. Die ist für die eigenen Widgets zu empfehlen. Eine Alternative wäre der Aufruf der Methode `Freeze`. Damit ist jede optische Aktualisierung des Widgets bis zum zugehörigen Thaw unterbunden..

Das `DCOverlay` zeichnet auf einen `Wx::Overlay`. Wie der Name andeutet ist es eine Zeichenfläche für temporäre Überlagerungen eines Widgets oder einer anderen `DC`. Etwa die *Cursor*-Spur bei Mausgesten lässt sich so darstellen und sehr schnell wieder löschen:

```
$overlay->Reset;
```

ohne die "Unterlage" zu beeinträchtigen. Zuvor musste natürlich das Ganze eingerichtet werden:

```
my $overlay = Wx::Overlay->new;
my $dc = Wx::ClientDC->new( $widget );
my $overlaydc =
   Wx::DCOverlay->new($overlay, $dc);
$dc->Clear;
$dc-> ...
```

Die Möglichkeiten solch gezeichneter Widgets kennen kaum Grenzen. Das bekannte Klangbearbeitungsprogramm *Audacity* verwendet zum Beispiel Wx und ein Großteil der Oberfläche besteht aus Widgets Marke Eigenbau. Auch wenn es kein Perl ist, so ist der Inhalt des Verzeichnisses "/src/widgets" ihres *SVN*-Archives unter *http://code.google.com/p/audacity/source/browse/audacity-src/trunk#trunk%2Fsrc%2Fwidgets* sehr aufschlussreich. Auch das Gespräch mit einem Autor in der "Floss Weekly"-Folge 42, geführt von Perl-Legende Randal L. Schwartz (*http://twit.tv/show/floss-weekly/42*), hilft Fehler zu vermeiden.

Es muss aber nicht immer der volle Aufwand betrieben werden, da es auch halbfertige Widgets wie die `ComboCtrl` gibt. Eine `ComboBox` ist eigentlich nichts weiter als eine `ComboCtrl`, also eine aufgebohrte `TextCtrl`, deren Methode `ShowPopup` ein Kontextmenü befiehlt. Es könnte aber genauso gut ein selbst gestaltetes `PopupWindow` sein, siehe Folge 9.

Auch die einen `Wx::Scrollbar` gibt es als einzelnes Widget, womit sich sehr ausgefallene Ideen umsetzen lassen. Doch es gibt eine Grenze, auch für dieses bisher längste Tutorial im Perl-Magazin.

Ende

Vielen Dank für das Lesen, die Fragen und Kommentare. Bitte schreibt weiterhin, es ist eine wichtige Quelle für das Tutorial, die WxPerlTafel, Buch und die anderen Dinge in Planung. Es werden sich wahrscheinlich noch Helfermodule aus *Kephra* verselbständigen, welche das Programmieren mit WxPerl leichter machen sollen. Wenn dies geschieht, wird sich dazu auch eine Meldung in den *CPAN News* finden.

Mark Overmeer

XML::Compile und SOAP

In der Ausgabe 03/2012 dieses Magazins habe ich die Grundlagen der Handhabung von XML in Perl erklärt. Verwendet habe ich hierzu das Modul `XML::Compile`. Dieses Modul erstellt strikte Konverter von XML-Nachrichten in komplexe Perl-Datenstrukturen und umgekehrt.

Die meisten Perl Module versuchen nach dem DWIM (Do What I Mean) - Prinzip vorzugehen um den Aufwand für den Programmierer so gering wie möglich zu halten: der Code vermutet, wie er sich zu verhalten hat. Wenn die XML Nachricht durch ein "schema" - der formalen Spezifikationssprache von Nachrichtenstrukturen - definiert ist, ist DWIMing gefährlich: die Prüfung auf Richtigkeit der Struktur ist standardmäßig notwendig. DWIMing könnte falsch liegen.

Wenn eine Bibliothek Schemata versteht, kann es dem Programmierer helfen eine gültige Nachricht zu erzeugen. Jedoch ist zu beachten, dass die meisten XML-Module, die man auf CPAM findet Schemata nicht verstehen. Dies wird zu Problemen führen. Beispielsweise besagt das Schema: "xyz ist vom Typ Integer", aber Rundungsfehler in der Berechnung im Perlskript ergeben einen Wert von 1.9999. Wenn die Bibliothek hier nicht hilft, muss der Programmierer um das Problem "herumprogrammieren". Wenn der Programmierer aber faul (oder unwissend) ist, wird eine Zahl des Typs float an die Position eines Integers gesetzt, was Prüfungsfehler auf dem Remote Server zur Folge hat. Die Schnittstelle wird instabil.

`XML::Compile` verfolgt basierend auf diesem Schema bei XML Nachrichten strenges Vorgehen. Alle Schemaeigenschaften werden zur Verfügung gestellt, obwohl einige etwas schwieriger sind als die meisten anderen. In diesem Artikel werde ich erklären, wie die Bibliothek arbeitet und wie man "*xsi:type*" und "*any*" Komplikationen löst. Am Ende des Artikels werden SOAP Erweiterungen erklärt.

Elemente erstellen mit SOAP::Lite

Schauen wir uns an, wie ein XML Element mit Hilfe von `SOAP::Lite` von Hand erstellt wird. Dies ist die gängigste Methode in Perl:

```
use SOAP::Lite;
my $elem = SOAP::Data->type('integer')
                     ->name(xyz => 1.999);

my $ser  = SOAP::Serializer->new;
print $ser->serialize($elem);
```

Das Ergebnis (verschönert) ist in Listing 1 zu sehen.

Dies zeigt eine ganze Menge von Namensraum-Deklarationen für SOAP, die wir hier nicht benötigen. Außerdem zeigt ein "xsi:type"-Attribut, dass der Wert ein Integer-Wert sein sollte. Schemabasierende XML-Protokolle wissen bereits den Typ des Wertes, dieses Attribut ist ein Relikt. Und, wie man hier sehen kann, wird der Wert nicht gerundet um zum Typ Integer zu werden.

```
<?xml version="1.0" encoding="UTF-8"?>
<xyz xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
     xsi:type="xsd:integer"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  1.999
</xyz>
```

Listing 1

Natürlich erkennt man die Notwendigkeit der Rundungstricks erst nachdem die Anwendung produktiv gegangen ist und man reale Daten, und keine Testdaten, verwendet. Dann kommt man zu dem Entschluss, dass es sinnvoll wäre, Hilfsfunktionen für `SOAP::Lite` zu erstellen, wie beispielsweise:

```
sub integer($$)
{    my ($name, $value) = @_;
     SOAP::Data->name(
            $name, int $value + 0.5);
}
```

So, wir benötigen 47 dieser Hilfsroutinen für die Basistypen die in der Schemataspezifikationen definiert sind und 14 weitere für Facetten (die weitere Restriktionen darstellen). Dann müssen wir das Schema manuell in die richtige Aufrufreihenfolge der Bausteine bringen.

Moderne Schemata sind groß und komplex. Sie sind ohne teuere Tools nicht zu lesen. Ein Beispiel: GML (*Geographic Markup Language*) hat mehr als 500 Strukturen! Alle professionellen Schemata werden größer und viel zu komplex um sie von Hand zu implementieren. Es wird immer unpraktischer `SOAP::Lite` zu verwenden.

Interpreter und Compiler

Ein *Schema* ist ein XML-Nachricht-Konstruktion und -Destruktionsprotokoll. Es beschreibt im Detail was erlaubt ist. Jemand oder etwas muss dem folgen.

```
    Data -----> processor -----> XML
                    ^
                 schema
```

Der Prozessor hat die Aufgabe dieses richtig zusammenzusetzen. Wir wollen den Prozessor nicht mehr von Hand schreiben, wie in `SOAP::Lite`. Deshalb schauen wir uns automatische Ansätze an.

Ein Schema ist so ähnlich wie ein Programm: ein Set von Regeln muss befolgt werden. Wie Programme kann der Prozessor in drei verschiedenen Wegen implementiert werden:

1. als ein Interpreter, der das Schema fortlaufend während Verarbeitung liest.
2. als ein Compiler, der das Schema in puren Code weit weg von der Originalquelle übersetzt.
3. irgendwo zwischen den oben genannten Optionen, indem die Quelle in ein Schema übersetzt wird, das die Aufgabe effizienter verarbeitet als der pure Interpreter.

Die Firmen, die XML groß gemacht haben, sind die Firmen die auch Java groß machten. Deshalb sind viele Schemabibliotheken in Java geschrieben. Als eine Typ-basierende Sprache, kann das Schematypsystem einfach in das Javatypsystem eingebunden werden. Diese Java-basierenden Bibliotheken übersetzen Schemaelemente in Objekte und Strukturen in Klassen und Methoden. Das Erstellen einer Nachricht wird eine Art von Serialisierung, das Auflösen von XML wird zur Objektgenerierung.

Für Perl hat dieser Ansatz einige Nachteile. Der wichtigste: ohne strengen Typisierungssupport in der Sprache, muss man diese Abstraktionsschicht selbst implementieren. Entweder half-harted, wie `SOAP::Lite` -- dem Erstellen von leichten Miniobjekten für jeden Typ -- oder wie `SOAP::WSDL2`, welches ein Paket für jeden Typ erstellt. In beiden Fällen sind für die Verarbeitung viele Objekte und Methodenaufrufe notwendig.

`XML::Compile` hat die Herangehensweise eines vollständigen Compilers: Er übersetzt das Schema nicht in Klassen und Methoden sondern in Subroutinen, die die Daten reorganisieren. `XML::LibXML` (*gnomes libxml2 libary*) wird verwendet um XML abzubilden. Skalare und verschachtelte `HASH`es und `ARRAY`s werden auf der Perl-Seite verwendet.

Sagen wir, man möchte einen einfachen kurzen Wert in eine XML Struktur schreiben. `XML::Compile` übersetzt diese Zeile des Schemas:

```
<element name="xyz" type="short">
```

wie in Listing 2 dargestellt (sehr vereinfacht)

Diese Subroutine gibt einen anonymen Closure zurück (es hat den Namen des Knoten in "$name" erkannt). Dieses Closure wird später so aufgerufen:

```
my $call = create_writer_short('xyz');
my $doc  =
XML::LibXML::Document->new('1.0', 'UTF-8');

my %data = (xyz => 'foo');
my $xml  = $call->($doc, \%data);
```

```
sub create_writer_short($)    # Compiler style
{   my $name = shift;
    sub ($$)
    {   my ($doc, $perl_hash) = @_;
        my $i = $perl_hash->{$name}
            or die "no data for $name";
        $i =~ m/^\s*[+-]?[0-9]+\s*$/ && $i >= -32768 && $i <= 32767
            or die "invalid value for short $name: $i";
        my $node = $doc->createElement($name);
        $node->appendText($i);
        $node;
    }
}
```

Listing 2

```
package Element;    # Interpreter style
sub process($$)
{   my ($self, $name, $data) = @_;
    my $value = $data->{$name} // $self->defaultValue;
    $self->checkValue($value)            or die;
    !$self->isRequired || defined $value or die;
    my $fixed = $self->fixedValue;
    !defined $fixed || $value eq $fixed  or die;
    ... etc...
}
```

Listing 3

Der oben genannte Code ist eine Vereinfachung: einige Dinge sind in dem Beispiel kombiniert: Wertprüfung, Wertparsen, Elementerstellung und Verfügbarkeit (ist notwendig). Es gibt sogar noch mehr Sachen an einem einfachen Schemaelement, beispielsweise Wiederholungshäufigkeit, Standardwert, ob es verpflichtend ist, kann es *NULL* sein und die Attribute. es ist nicht ausreichend einfachen Code für alle möglichen Kombinationen herzustellen.

Jetzt beginnt man den Unterschied zwischen (halben und ganzen) Interpretern und Compilern zu spüren. Interpreter werden auf einem Verständnislevel beim Start des Programms stehen bleiben und dann mehr Arbeit während der Laufzeit erledigen. Im oben genannten Beispiel würde der Runtime-Code für den Interpreter vermutlich etwa so etwas wie in Listing 3 ausführen.

Obwohl das Schema uns sagt, dass es keinen standardmäßigen oder festgelegten Wert für dieses spezielle Element gibt, wird der Code für jedes abgearbeitete Element immer wieder ausgeführt. Es ist schwierig dieses Laufzeitverhalten zu optimieren.

Erstellen eines Compilers

Mit einem Compiler wie `XML::Compile` gehen wir weiter zum nächsten Verständnislevel. Es ist zwar mehr Aufwand während der Vorbereitung notwendig, aber es vermindert die Last während der Laufzeit. In Perl --langsamer als C und Java-- ist es notwendig, die Performanz im Blick zu halten. Der `XML::Compile` Übersetzer erstellt Code, der logisch dem folgenden gleich ist:

```
my $node = XML::LibXML::Element->...;
my $process
   = $check_avail->(
       $decode_element->(
         $validate_element->(
            $take_element->($node)));
```

Wenn die Prüfung ausgeschaltet ist, wird der `$validate_element` Aufruf nicht in den `$process`-Wasserfall eingebunden. Falls das Element nicht benötigt wird, wird $check_avail nicht vorhanden sein. Und so weiter. Einfache Element werden in einem kleinen Wasserfall abgearbeitet, komplexe Elemente werden in einem größeren Baum von Aufrufen abgearbeitet. Code-Referenzen rufen Code-Referenzen auf, die wiederum Code-Referenzen aufrufen... unzählige kleine Bausteine.

Das oben gezeigte ist aufgebaut, wie es in Listing 4 dargestellt wird.

Schemata sind mit Hilfe von Interpretern klar aufgebaut. Speziell die "xsi:type" und "any" Kontrukte (beide sollten sterben") können nicht in der Kompilierungszeit vorausgesagt werden. Ein beiden Fällen muss man `XML::Compile` mit Informationen aus der schriftlichen Dokumentation helfen...

```
my $call  = make_process;
my $node  = XML::LibXML::Element->...;
print "VALUE = ", $call->($node), "\n";

sub make_process()
{   my $step1 = make_take_element;
    my $step2 = make_validate_element($step1);
    my $step3 = make_decode_element($step2);
    my $step4 = make_check_avail($step3);
    $step4;
}

sub make_take_element()          # run compile-time
{   # This sub returns a subroutine to do the work later
    # the first paramater $_[0] will be the $node
    sub { my $node = shift;
          $node ? $node->textContent : undef;
        };
}

sub make_validate_element($)     # correct value?
{   my $first_do = shift;        # trap code-ref in closure
    sub { my $value = $first_do->($_[0]);
          defined $value && $value =~ m/^\s*[+-]?[0-9]+\s*$/
              or die "not a number";
          $value;
        };
}

sub make_decode_element($)       # make value Perl native
{   my $first_do = shift;
    sub { my $value = $first_do->($_[0]);
          defined $value ? $value + 0 : undef;
        };
}

sub make_check_avail($)
{   my $first_do = shift;
    sub { my $value = $first_do->($_[0]);
          defined $value or die "required is missing";
          $value;
        };
}
```

Listing 4

Verwendung von xsi:type

Eines der Komplikationen ist die Unterstützung von "xsi:type". Dieses Attribut wird von XML-RPC verwendet: Schema-lose Remote-Procedure-Calls mit Hilfe von XML. Der Server stellt einige Funktionen zur Verfügung, die Parameter werden über das Netzwerk in eine XML kodierte Struktur geschickt. Es gibt dafür kein Schema, deshalb muss der Client dem Server mitteilen, um welche Typen von Datenelementen es sich handelt. Schemata unterstützen diese Funktion.

Ich selbst mag diese Arbeitsart nicht. Clients und Server werden oft von verschiedenen Leuten/Firmen geschrieben und gewartet. Deshalb muss die Kommunikation eine gut geplanten Schnittstelle haben: eine dokumentierte Spezifikation, die mehrere Jahre und Generationen von Client- und Serversoftware überleben kann. Wenn einer der Seiten den Freiraum bekommt eine eigene Interpretation des Protokolls zu haben, wird das sicher zu einem Chaos führen. Die Praxis hat mir viele dieser Horrorszenarien gezeigt, meist in kommerziellen Anwendungen. "xsi" sollte sterben"!

Es sollte verschwinden, aber man trifft es immer wieder an. Deshalb müssen wir es unterstützen. Schauen wir uns an, wie man es zum Laufen bekommt. Wir beginnen mit einem leerem Schema. Wir laden es mit

```
my $schema =
     XML::Compile::Cache->new($schema);
```

Die `::Cache` Klasse erweitert die `::Schema` Klasse mit einer einfacheren Prefix-Handhabung und dem Caching von kompilierten Steuerungen. Wenn man eine WSDL-Datei hat, beginnt man mit

```
my $wsdl = XML::Compile::WSDL11->new($wsdl);
```

Die `::WSDL11` Klasse wiederum erweitert `::Cache`. Auf jeden Fall kann man seine eigenen Prefixes für den Namensraum, den man verwendet, hinzufügen. Standardmäßig werden Prefixes von schema/wsdl-Dateien genommen, aber man sollte besser nicht annehmen, dass diese Namensraumabkürzungen über die Zeit gleich bleiben. Eindeutig bedeutet das:

```
my $exns = 'http://example.com/ns1';
$schema->prefixes(ex => $exns);   # or
$wsdl->prefixes(ex => $exns);
```

Jetzt sehen wir uns vier Wege an, um den gleichen "Handler" zu erstellen - siehe Listing 5.

Jetzt, wenn die Nachricht Elemente mit "xsi:type" verwendet, muss man das dem Compiler zuvor mitteilen, bevor man `compileClient()` oder `reader()` aufruft.

```
$schema->xsiType('ex:basetype'
   => [ qw/ex:type1 ex:type2/ ]);
```

Wenn der Basistyp komplex genug ist, kann man

```
$schema->xsiType('ex:basetype' => 'AUTO');
```

verwenden.

Man sollte aber nicht dieses 'AUTO' für Typen so einfach wie 'long' verwenden: die Ergebnisliste wird groß sein und *alle* Verwendungen von longs werden langsamer werden.

Verwendung von ANY Elementen

Ein anderes unangenehmes Überbleibsel von XML's Vergangenheit ist "any". Diese "any" Element geben an, wo der Schemadesigner aufgegeben hat anzugeben wie die Nachricht strukturiert ist. Mit ein bisschen Glück findet man ein Blatt Papier (nicht zu sehr veraltet) welches einem sagt was man an der Stelle in den Nachrichten erwartet kann. Diese "any" Elemente sind typischerweise wie eine Seuche: wenn der Designer eines entdeckt hat kann er faul werden. Er neigt dazu es überall zu (miss-)brauchen.

Die Verwendung und die Definition sind nur in der schriftlichen Dokumentation miteinander verknüpft. Schauen wir uns das nächste Beispiel an: Erwartet der Server ein "message" Objekt an der "any" stelle oder wird er abstürzen?

```
<element name="unclear-content">
  <sequence>
    <element name="some-data" type="int" />
    <any minOccurs="0" />
  </sequence>
</element>

<element name="message" type="ex:message" />
```

Es gibt eine saubere Alternative in Form von `substitutionGroup`'s. Die Syntax ist lang, aber sehr eindeutig (siehe Listing 6).

```
use XML::Compile:Util qw/pack_type/;
my $type = pack_type $exns, 'sometype'; # eq "{$exns}sometype"

# pass $h around yourself
my $h = $schema->compileClient(READER => $type);          #1
my $h = $schema->compileClient(READER => 'ex:sometype');  #2

# or, compiled once, then cached
my $h = $schema->reader($type);                           #3
my $h = $schema->reader('ex:sometype');                   #4
```

Listing 5

```
<element name="clear-content">
  <sequence>
    <element ref="any-message" minOccurs="0" />
  </sequence>
</element>

<element name="any-message" type="ex:any-msg-type"
   abstract="true" />

<element name="message" type="ex:message"
   substitutionGroup="ex:any-message" />

<complexType name="message">
  <complexContent>
    <extension base="ex:any-msg-type" />
  </complexContent>
</complexType>
```

Listing 6

Im obigen Beispiel erstellen wir ein Basiselement (welches optional in "clear-content" eingebunden ist). Dieses "message" Element ist eine Substitutionsgruppe für any-Nachrichten, deshalb kann sie an jeder Stelle, an der eine any-Nachricht spezifiziert wurde, ersetzt werden. Hier eine Beispielnachricht:

```
<clear-content>
  <message>
     ...
  </message>
</clear-content>
```

Man kann mehrstufige Ersatzgruppen erstellen. Jedes Element, welches als "abstract" deklariert wurde, kann nicht instanziiert werden. Der Basistyp muss nicht abstrakt sein. Diese Substitutionsgruppen sind spezifisch und leistungsfähig. Wie kann man solche armen "any" Element in XML:: Compile verwenden? Für den Reader gibt es hier einen einfachen Trick:

```
my $r = $schema->compileClient(
       $type, any_element => 'ATTEMPT');
```

Dies wird einen Reader für ein ungekanntes Element während der Laufzeit kompilieren - aber nur einmal, weil es gecacht wird.

Für den Writer ist es ein bisschen komplizierter. Jeder Writer für die "any" Komponenten muss manuell erstellt werden und separat aufgerufen werden.

```
my $anyt = 'ex:mytype';
# or '{$exns}mytype'
my $doc  =
 XML::LibXML::Document->new('1.0', 'UTF-8');
# sub-tree
my $w1   = $schema->writer($anyt);
# top-tree
my $w2   = $schema->writer($topt};

my $node = $w1->($doc, $the_any_data);
my $top_data = { ... somewhere => {
                          $anyt => $node }};
my $root = $w2->($doc, $top_data);
# do not forget this
$doc->setDocumentElement($root);
print $doc->toString(1);
```

Die `XML::LibXML` Bibliothek verlangt, dass alle Komponenten des Baumes vom selben `::Document` Element erstellt werden. Man erstellt einen Unterbaum für das "any" Element, setzt es in den Datenbaum für das ganze Dokument und erstellt abschließend die Nachricht. Das "any" Element hat seinen Typ als Schlüssel. Natürlich sollte man ein schöne Unterroutine erstellen um diese Prozesserschwerung zu verstecken.

SOAP

Machen wir einen Schritt in die Welt von SOAP. Im Jahr 2000 haben einige bedeutende Firmen eine Spezifikation von SOAP (Version 1.1) zusammengestellt, welche vom W3C als "NOTE" akzeptiert wurde. Diese Klassifikation zeigt die schwache Qualität dieses Teils: Auf nur 40 Seiten werden sechs verschiedene Techniken beschrieben, die in diesen Firmen verwendet werden. Dies ist viel zu wenig Information um eine neue Implementation darauf zu stützen. Glücklicherweise waren nicht viele meiner Vermutungen falsch.

Die Struktur von SOAP Nachrichten (Anfragen und Antworten) ist

```
<SOAP-ENV:Envelope>
  <SOAP-ENV:Header />
  <SOAP-ENV:Body />
</SOAP-ENV:Envelope>
```

Die im Body gekapselten Daten können ausgetauscht werden. Der Header enthält Meta-Daten, Authentifizierung, Routing-Information und weiteres.

SOAP definiert drei Arten von Operationen:
1. rpc-encoded
2. rpc-literal
3. document-literal

Das erste ist eine Art von XML-RPC: der Server spezifiziert Arbeitsprozesse aber nicht die Typen. Die Prozesse bilden direkt die Subroutinen, die auf dem Server implementiert sind, ab. Der aufrufende Client kennt einen Trick um die Daten, welche als Parameter der Funktion übergeben werden, zu entschlüsseln, aber nichts ist in der Spezifikation beschrieben. Das funktioniert vielleicht gut wenn Client und Server die selbe Bibliothek verwenden. `XML::Compile` mag die Unvorhersehbarkeit von Nachrichten nicht, und unterstützt deshalb 'rpc-encoded' im Moment nicht. Die SOAP 1.2 Spezifikation (eine gute Spezifkation, die fast keiner verwendet) rät davon ab die Operation von Typ 1 zu verwenden.

Der zweite Typ ist 'rpc-literal'. In diesem Fall sind die Parameter der Funktion, die aufgerufen wird, gut beschrieben. Der größte Unterschied zum dritten Modus ist, dass der Body klar erkennbar den Namen der Operation (Funktion) und seiner Parameter, enthält. Das ist eine kleine Abstraktion.

Abstrakter ist der 'document-literal' soap-Stil. In diesem Fall werden Operationen anhand des Types des Elements im Body erkannt. Das ist ein bisschen komisch... war es zu viel Arbeit den Namen der Operation zur Referenz in der erstellten Anfrage zu implementieren? Mit ein wenig Aufwand kann man jedoch den Elemententyp direkt zum Operationsnamen hinzuordnen. Der Client schickt ein Dokument, der Server antwortet mit einem Dokument. Dies ist abstrakter als Funktionsnamen (zumindest in der Theorie). Es könnte mehrere Reimplementationen der Serversoftware überleben.

WSDL

Was benötigt man um SOAP zu verwenden? Man benötigt
• Details über die Operationen (was ist im Header und im Body für Anfragen und Antworten),
• ein Schema welches Typen und Elemente definiert (zur Verwendung im Header und im Body), und
• eine Serveradresse zur Verbindung.

Diese Fakten sind alle in einer WSDL Datei zusammengefasst, ein anderer Standard im Bereich der XML "Technologie". Wenn man WSDL hat, wird das Leben einfach. Eine komplette Client-Implementation sieht wie folgt aus:

```
# During the program's initiation
my $wsdl = XML::Compile::WSDL11->new(
                            $wsdl_filename);
# I like that:
$wsdl->addKeyRewrite('UNDERSCORES');

# for each operation you use
my $call = $wsdl->compileClient($op_name);

# At run-time, as often as you want
my ($answer, $trace) = $call->(%data);

# Or, if you do not want a trace
my $answer = $call->(%data);

# Data can also be passed by reference
my $answer = $call->(\%data);

# Now, Data::Dumper is your friend
$Data::Dumper::Indent    = 1;
$Data::Dumper::Quotekeys = 0;
print Dumper $answer;

# $trace is an
# XML::Compile::SOAP::Trace object
$trace->printTimings;
$trace->printRequest(pretty_print => 1);
```

Einfacher: alle möglichen Aufrufe werden während der Initierung kompiliert und gecacht.

```
# During initiation
$wsdl->compileCalls;

# At run-time
my ($answer, $trace) =
             $wsdl->call($op_name, %data);
my $answer = $wsdl->call($op_name, %data);
```

Ja, das ist die Logik die man benötigt. Aber was machen wir mit dem Bedarf von %data? Beginnen wir mit dem Verständnis der Ausgabe von

```
print $wsdl->explain(
               $op_name, PERL => 'INPUT');
print $wsdl->explain(
               $op_name, PERL => 'OUTPUT');

my $operation = $wsdl->operation($op_name);
print $operation->explain(
     $wsdl, PERL => 'INPUT', recurse => 1);

# do not forget this helper
print $wsdl->template(PERL => $type);
```

`INPUT` und `OUTPUT` sind WSDL Begriffe: jeweils Input für den Server und Output vom Server. Also, Anfragen und Antworten. 'PERL' meint hier, dass in diesem Beispiel Perlsyntax erstellt wird.

Der Header und der Body haben keine oder mehr Elemente. Dasselbe Element kann mehrfach erscheinen. Das Element kann möglicherweise nur durch seinen Typ spezifiziert werden. Das einzige, was wir verwenden können, um ein Element eindeutig zu identifizieren, ist "part-name". Hier sieht man einen Header:

```
...
<soap:header message=
   "msgs:auth" part="h_one" use="literal" />
...
<wsdl:message name="auth">
  <wsdl:part name="
      h_one" element="tns:Auth" />
</wsdl:message>
```

Dasselbe gilt für Body-, Fault- und Headerfault- komponenten: sie sind von Nachrichtenteilen erstellt. In einer komplexeren Form, wird %data wiefolgt weitergereicht:

```
%data = ( Header => { h_one => $Auth_data }
        , Body   => { b_one => $data1,
                      b_two => $data2 }
        );
```

Der aufgerufene Handler ist aber flexibel: falls die 'part names' im Header und im Body sich unterscheiden, kann man auch sagen:

```
%data = (h_one => $Auth_data,
         b_one => $data1,
         b_two => $data2);
```

Noch besser: meistens gibt es üblicherweise nur einen Body, der auf ein höheres Level gehoben wird:

```
%data = (h_one => $Auth_data, %$data1);
```

Beispielsweise sind diese drei gleichwertig falls der einzige Bodyteil 'params' genannt wurde:

```
my $set    = $wsdl->compileClient(
                            'setCounter');
my $answer = $set->(new_value => 42);
my $answer = $set->({new_value => 42});
my $answer = $set->(params =>
                       {new_value => 42});
my $answer = $set->({params =>
                      {new_value => 42}});
my $answer = $set->(Body =>
           {params => {new_value => 42}});
my $answer = $set->({Body
       => {params => {new_value => 42}}});
```

Die $answer wird in der vierten Form entschlüsselt.

Ein SOAP Server

Jetzt, wo wissen wir, wie ein Client erstellt wird, wäre es schön auch einen Server erstellen zu können. Beispielsweise sollte man, wenn man einen Client für einen Service, den man nicht selbst verwaltet, entwickeln muss, einen Testserver mit Hilfe von ein paar Zeilen Code aufsetzen können.

Ein Server ist ein Daemon, ein Prozess der darauf wartet, dass sich Clients verbinden und Fragen stellen. Man nimmt seine Daemon-Implementation von

- Any::Daemon ::Daemon::AnyDaemon
- CGI / mod_perl ::Daemon::CGI
- Net::Server ::Daemon::NetServer
- PSGI / Plack ::Daemon::PSGI
- NGiNX ::Daemon::NGiNX

Jedes hat `XML::Compile::SOAP::Daemon` seine eigene Laufzeit (und Dokumentation). Der Code, um einen Server zu starten, sieht (üblicherweise) so aus:

```
use XML::Compile::SOAP::Daemon::AnyDaemon;
my $daemon  =
  XML::Compile::SOAP::Daemon::AnyDaemon->new
  ( pid_file => ..., user => ..., ... );

# load the WSDL
my $wsdl =
         XML::Compile::WSDL11->new($wsdlfn);

# compile the operation handlers
$daemon->operationsFromWSDL(
   $wsdl, callbacks => \%callbacks);

# Response to ?WSDL requests
$daemon->setWsdlResponse($wsdlfn);

# Start the daemon
$daemon->run(max_childs => 10);
exit 0;
```

Der Trick liegt in `%callbacks`: ein Mapping von Operationsnamen zu seinen Handlern, die Code-Referenzen sind. Jeder Handler wird wiefolgt aufgerufen:

```
sub my_callback($$$)
{   my ($soap, $data_in, $request) = @_;
    ... processing ...
    return $data_out;
}
```

Der `$soap` Parameter ist das `::Daemon::` Objekt, `$data_in` ist die Perlstruktur verschlüsselt in der XML Nachricht, und `$request` ist die erhaltene Anfrage. Der wirkliche Typ von `$request` ist vom Backend abhängig, aber in den meisten Fällen ist es ein `HTTP::Request` Objekt. Rate, was `$data_out` ist.

Fehlende Rückfragen werden sauber einen Fehler an den Client zurückgeben. Es ist ein Tracing möglich. Daemons zu debuggen ist kniffelig.

Letzte Worte

XML zu verwenden ist eine Dose voller Würmer, genauso wie das Verwenden einer anderen Bibliothek oder Anwendung. Wie immer, wird man Bugs finden und limitiertes Verständnis seiner Konzepte durch den Entwickler. Man sollte sich also die Zeit nehmen um sich selbst damit vertraut zu machen (um smartere Fehler zu machen)!

Renée Bäcker

Mojolicious-Tutorial: Teil 1

In der letzten Ausgabe wurde bereits eine Mini-Einführung in `Mojolicious` gegeben. Ab dieser Ausgabe soll ein ausführlicher Einstieg gezeigt werden. Der Einstieg sollte sehr leicht werden, weil das Framework außer Perl 5.10.x keine Voraussetzungen hat. Alle notwendigen Module werden mitgeliefert. Im Laufe des Tutorials soll eine Webanwendung programmiert und erweitert werden, bei der Läufer ihre täglichen Zeiten und Kilometer eintragen können.

`Mojolicious` wurde von Sebastian Riedel begonnen, der auch schon `Catalyst` gestartet hat. Er selbst sagt, dass er aus den Erfahrungen von Catalyst heraus einiges anders und besser machen wollte. Dabei muss er auch Kritik einstecken, weil Riedel einige Räder neu erfunden hat; so gibt es einen Parser für HTML, eine UserAgent-Klasse, eine eigene Template-Engine und vieles mehr, wofür es auf CPAN bereits erprobte Module gibt. Der Vorteil von Sebastian Riedels Ansatz ist aber, dass `Mojolicious` keine Abhängigkeiten zu anderen CPAN-Modulen hat. Die gibt es erst, wenn man Plugins verwendet. Doch dazu kommen wir erst in eine der kommenden Ausgaben.

Der Vorteil des Ansatzes ohne Abhängigkeiten liegt darin, dass man Mojolicious sehr schnell installiert hat - auch auf einfachen Webhosting-Paketen.

Aber bevor wir loslegen, schauen wir uns mal an, was `Mojolicious` noch so alles mitbringt.

Skripte

`Mojolicious` liefert auch gleich ein paar Skripte mit, die bei der Entwicklung der Anwendung unterstützen. Zum einen *morbo* und *hypnotoad* - beides Webserver - und zum anderen *mojo*.

morbo und hypnotoad

Der Unterschied zwischen *morbo* und *hypnotoad* liegt in der Optimierung für spezielle Einsatzszenarien. *morbo* ist eher für die Entwicklungsumgebung geeignet, weil es auf Änderungen am Skript, den Modulen und den Templates lauscht und dann selbständig neu lädt.

morbo unterstützt alle möglichen Techniken wie IPv6, TLS und Websockets. Diese Themen werden in späteren Folgen des Tutorials näher betrachtet. *morbo* startet nur einen einzelnen Prozess, was für die Entwicklung auch ausreichend ist.

hypnotoad ist dagegen für den Produktivbetrieb gedacht. Es werden mehrere Prozesse gestartet. Es werden - wenn möglich - mehrere CPU-Kerne und copy-on-write verwendet. Genau wie bei *morbo* werden auch hier IPv6, TLS und Websockets unterstützt. Da der Server Signale für das Prozessmanagement verwendet, sollte man in seiner Anwendung auf ein eigenes Signalhandling verzichten.

mojo

Ein weiteres Skript ist `mojo`. Mit diesem Skript kann man die Arbeiten an der Webanwendung beginnen und noch viele weitere Sachen machen, die in den kommenden Abschnitten noch näher betrachtet werden.

Unsere Anwendung

Wie bereits oben erwähnt, sollen bei der Anwendung die Läufer ihre Kilometer und Laufzeiten eintragen. Da nicht jeder alle Daten sehen darf, muss eine Benutzerverwaltung und entsprechend Authentifizierung dabei sein. Der User hat ein einfaches Formular bei dem Streckenlänge, Datum und benötigte Zeit eingetragen werden kann. Schließlich kann

der Läufer auch eine kleine Auswertung seiner Laufleistung bekommen.

Nachdem jetzt klar ist, welche Hilfsmittel `Mojolicious` so alles mitbringt und was unsere Anwendung können soll, kann mit der Erstellung der Anwendung begonnen werden.

Erstellen des Grundgerüsts

Damit kommen wir zur ersten Funktion von *mojo*: *generate*. Mit diesem Kommando können verschiedene Dinge erstellt werden. Zum Erstellen des Grundgerüsts muss das Sub-Kommando `app` verwendet werden. Als Parameter wird noch der Name der Anwendung übergeben (siehe Listing 1).

Damit wird also eine Struktur aufgebaut, die einer CPAN-Distribution sehr ähnlich ist. So ist es zu einem späteren Zeitpunkt auch relativ einfach, die Anwendung auf CPAN zu veröffentlichen. Schauen wir uns die Dateien aber mal an.

Unter `script/` wird ein Skript erstellt, dass die Anwendung an sich startet:

```
use FindBin;
use lib "$FindBin::Bin/../lib";
# Start commands for application
require Mojolicious::Commands;
Mojolicious::Commands->
   start_app('TrackRuns');
```

Mehr ist erstmal auch nicht notwendig.

Die *Controller* und anderen Module liegen unter `lib/`. mojo erstellt dort zwei Module: `TrackRuns.pm`, in dem die Anwendung an sich eingerichtet wird: Plugins werden geladen, Routen werden konfiguriert und vieles mehr.

```
package TrackRuns;
use Mojo::Base 'Mojolicious';

# This method will run once at server start
sub startup {
  my $self = shift;

  # Documentation browser under "/perldoc"
  $self->plugin('PODRenderer');

  # Router
  my $r = $self->routes;

  # Normal route to controller
  $r->get('/')->to('example#welcome');
}

1;
```

Das zweite Modul ist `TrackRuns::Example`. Dort ist der Code zu finden, der hinter den Routen steckt. Was es mit den Routen auf sich hat, wird später noch genauer erläutert.

```
package TrackRuns::Example;
use Mojo::Base 'Mojolicious::Controller';

# This action will render a template
sub welcome {
  my $self = shift;
    # Render template
    # "example/welcome.html.ep" with message
  $self->render(
    message => 'Welcome to the Mojolicious
                real-time web framework!');
}

1;
```

Mit `Mojo` ist es einfach, Webanwendungen zu testen. Auch in der neu angelegten Anwendung wird ein Basistest erzeugt. Dieser liegt unter `t/`.

```
$ mojo generate app TrackRuns
    [mkdir] /track_runs/script
    [write] /track_runs/script/track_runs
    [chmod] track_runs/script/track_runs 744
    [mkdir] /track_runs/lib
    [write] /track_runs/lib/TrackRuns.pm
    [mkdir] /track_runs/lib/TrackRuns
    [write] /track_runs/lib/TrackRuns/Example.pm
    [mkdir] /track_runs/t
    [write] /track_runs/t/basic.t
    [mkdir] /track_runs/log
    [mkdir] /track_runs/public
    [write] /track_runs/public/index.html
    [mkdir] /track_runs/templates/layouts
    [write] /track_runs/templates/layouts/default.html.ep
    [mkdir] /track_runs/templates/example
    [write] /track_runs/templates/example/welcome.html.ep
```

Listing 1

Nicht alles an einer Webanwendung ist dynamisch. Einige Dateien sind statisch - z.B. CSS- oder JavaScript-Dateien, Bilder etc. Diese statischen Dateien liegen standardmäßig unter `public/`. Diesen Pfad kann man auch im Starter-Modul in der Methode `startup` anpassen:

```
sub startup {

    # neuen Pfad statischer Inhalte
    $self->static->paths = [
      '/home/mojo/static'
    ];

    # zusaetzlicher Pfad
    push @{ $self->static->paths },
      '/var/www/web111';
}
```

Unter Templates werden - welche Überraschung - die Templates gespeichert. Mojolicious erfindet auch in Sachen Templates das Rad neu und führt eine neue Template-Engine ein. Die soll für den Anfang auch zum Einsatz kommen.

Starten der Anwendung

mojo erstellt eine voll lauffähige Anwendung und die wollen wir jetzt starten (siehe Abbildung 1).

```
$ cd track_runs/script/
$ morbo track_runs
Server available at http://127.0.0.1:3000.
```

Routen

Direkt nach dem Anlegen der Anwendung existiert nur eine einzelne Route: `/`. Die Webanwendung soll aber noch mehr können als statische Inhalte und diese eine Seite anzuzeigen. Es gibt mehrere URLs, unter denen etwas zu sehen ist.

Für die Anwendung, die hier entwickelt werden soll, gibt es etliche URLs:

```
/login
/logout
/register
```

sind URLs, die auch Gäste aufrufen können. Es sollte klar sein, was hinter diesen URLs steckt.

```
/run/add
/run/edit
/run/delete
/runs
/runs/statistics
```

sind URLs für registrierte und eingeloggte Besucher. Hier können Daten zu einzelnen Läufen eingegeben und bearbeitet werden, man kann sich aber auch alle Läufe auflisten lassen und sich Statistiken generieren lassen.

```
/user
/user/edit
/user/delete
```

sind nur für den Administrator erreichbar. Hier werden einfach Benutzer verwaltet.

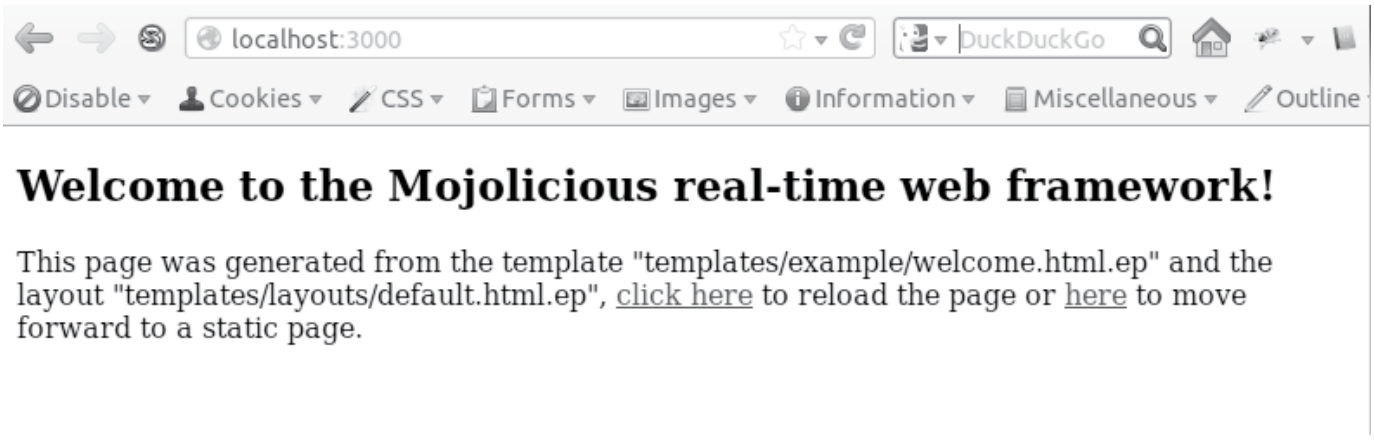


Abb. 1: Diese Startseite wird direkt erzeugt

Mojolicious muss jetzt noch wissen, wie man von der URL zum Perl-Code kommt. Im Starter-Modul ist folgendes zu finden:

```
my $r = $self->routes;

# Normal route to controller
$r->get('/')->to('example#welcome');
```

In `$self->routes` steckt ein Router-Objekt. Diesem Router-Objekt wird die Route `/` bekannt gemacht. In diesem Fall ist das sogar auf GET-Requests beschränkt. Wenn man auf die Kurzform verzichten möchte, kann man auch die verketteten Methoden `route` und `via` verwenden. Diese Möglichkeit hat ihre Stärken, wenn man sich an anderer Stelle das Leben leicht machen möchte. Mit den beiden Methoden sieht das folgendermaßen aus:

```
$r->route( '/' )
  ->via( 'get' )
  ->to( 'example#welcome' );
```

Das ist aber viel mehr Schreibarbeit! Was aber, wenn man eine Route für mehrere Request-Methoden verwenden möchte?

```
$r->get('/')->to('example#welcome');
$r->post('/')->to('example#welcome');
```

ist zu viel Redundanz! `any` statt `get` und `post` ist schon wieder zu lax, denn dann wäre z.B. auch *PUT* erlaubt. Bei `via` kann man eine Liste an erlaubten Methoden angeben, also könnte man

```
$r->route( '/' )
  ->via( qw|GET POST| )
  ->to( 'example#welcome' );
```

schreiben um die Route für *GET*- und *POST*-Requests einzurichten.

```
$r->any('/statisch');
```

fügt eine Route hinzu, die einfach nur das Template ausgibt. Das eignet sich besonders dann, wenn man Template-Features braucht (z.B. Einbinden von anderen HTML-Snippets), aber sonst keine dynamischen Werte ausliefert. Wenn nur eine solche Route angegeben ist, gibt Mojolicious das Template direkt im Ordner *templates* aus. In diesem Fall wäre das *statisch.html.ep*. Mehr zu der Benamung der Templates folgt im Abschnitt *Templates*.

Soll etwas Dynamisches erzeugt werden, muss etwas programmiert werden und dann muss eine Verknüpfung von Code zur Route existieren. Die Verknüpfung mit dem Code wird über die `to`-Methode erreicht. Am einfachsten geht es, indem man der Methode einen String übergibt mit dem Muster *Controller#Action*. In diesem Fall wäre `TrackRuns::Example` der Controller und *welcome* die Action. Das `TrackRuns::` muss hier nicht angegeben werden, weil Mojolicious automatisch den Namen der Anwendung voranstellt.

Sollen die Controller in einem anderen Namespace liegen, kann man diesen mit `namespace` anpassen:

```
$r->namespace( 'Controller' );
$r->get('/')->to('example#welcomer');
```

Die einfachste Variante, auf den Controller und die entsprechende Action zu verweisen, wurde schon gezeigt. Es besteht aber auch die Möglichkeit, der `to`-Methode einen Hash zu übergeben:

```
$r->get('/')->to(
  controller => 'example',
  action     => 'welcome',
);
```

Bei Mojolicious besteht die Möglichkeit, noch während der Laufzeit weitere Routen zu definieren. Das muss nicht in der `startup`-Methode stattfinden. So könnte in der Methode `welcome` folgendes zu finden sein:

```
if ( -e '/tmp/welcome.tmp' ) {
    $self->routes->route( '/hello' )->to(
      'example#hello'
    );
}
```

Selten hat eine Anwendung nur solche fixen URLs. Es müssen noch weitere Informationen übergeben werden, z.B. welche Benutzerdaten bearbeitet werden sollen. Man könnte jetzt mit GET-Parametern arbeiten und die URL `/user/edit?id=1234` aufrufen. Aber heute ist es üblich, solche Informationen im Pfad der URL zu übermitteln, also `/user/edit/1234`. Jetzt für alle Benutzer dynamisch solche Routen hinzuzufügen wäre unsinnig. Aus diesem Grund gibt es Platzhalter. Mit einem Platzhalter sieht das Routing beispielsweise so aus:

```
$r->route( '/user/edit/:user_id' )->to(
    'user#edit'
);
```

Werden solche Platzhalter benutzt und man möchte Defaultwerte haben, wird es mit der Hash-Variante übersichtlicher:

```
$r->route( '/user/edit/:user_id' )->to(
    controller => 'user',
    action     => 'edit',
    user_id    => 1,
);
```

Das gute an den Routen in Mojolicious ist, dass man den Routen auch Namen geben kann. Dadurch kann man in Templates und Redirects Code schreiben, der erhalten bleibt auch wenn sich die Route (sprich: URL) ändert.

```
$r->get('/test')
  ->to('example#welcome')
  ->name( 'start' );

$r->get('/redirect')
  ->to( 'example#name_test' );
```

Letztere Route ist nur für diesen Test da. Die Methode `name_test` macht nur ein Redirect auf die Willkommensseite:

```
sub name_test {
  my $self = shift;
  $self->redirect_to(
    $self->url_for('/test')
  );
}
```

Wenn jetzt die Route für `/test` geändert werden soll - weil z.B. die Anwendung umzieht, müsste auch `name_test` angepasst werden. Hier kommen die Namen für Routen ins Spiel, denn die Methode kann auch folgendermaßen aussehen:

```
sub name_test {
  my $self = shift;
  $self->redirect_to( 'start' );
}
```

Ein weiteres nützliches Feature ist der Umgang mit Dateiendungen. Mojolicious erkennt die Dateiendungen automatisch und stellt die Information im Stash (siehe Abschnitt "Stash") zur Verfügung. Rufen wir doch einfach mal die URL `http://localhost:3000/test.html` auf, wobei die Methode `example` - auf die URL verweist - mittlerweile folgendermaßen aussieht:

```
sub welcome {
  my $self = shift;
  $self->render(
    message =>
      'Format: ' . $self->stash('format'),
  );
}
```

Die jetzt erscheinende Startseite ist in Abbildung 2 zu sehen. Wie man sieht, gilt weiterhin die Route für `/test`, aber zusätzlich wurde als Format *html* erkannt.

Soll die Format-Erkennung ausgeschaltet werden, muss das bei der Route angegeben werden:

```
$r->get('/test', format => 0)
  ->to('example#welcome')
  ->name( 'start' );
```

Sollen nur bestimmte Formate erkannt werden, kann man `format` auch eine Arrayreferenz übergeben:

```
$r->get('/test', format => [qw|xml json|])
  ->to('example#welcome')
  ->name( 'start' );
```

Wenn man die Routen erstellt, muss man allerdings auf die Reihenfolge achten, denn je nach Routen, werden manche Routen nie erreicht. Ein Beispiel:

```
$r->route('/user/:id')->to('user#show');
$r->route('/user/edit')->to('user#edit');
```

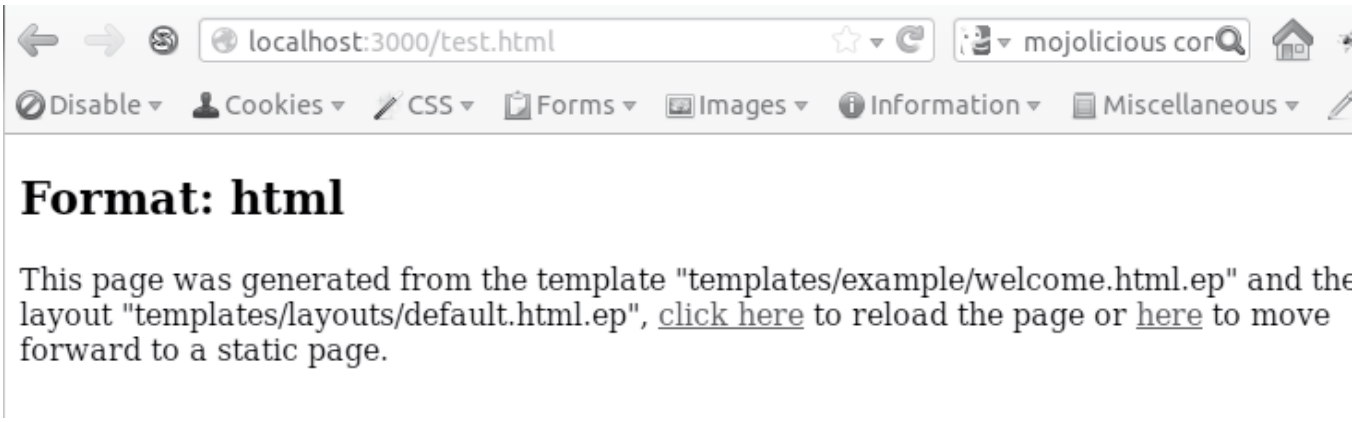


Abb. 2: Das format "html" wird verlangt

In diesem Fall wird die *edit*-Aktion nie erreicht, weil beim Aufruf der URL `http://localhost:3000/user/edit` die erste Route schon greift und das "edit" als "id" betrachtet wird. Beim Erstellen der Route sollte man von den spezifischen URLs zu den unspezifischen URLs gehen.

Platzhalter

Im vorigen Abschnitt wurde schon ein Beispiel gezeigt, wie Routen dynamisch sein können - mit Platzhaltern. In Mojolicious gibt es drei verschiedene Arten von Platzhaltern: Generische, Relaxed und Wildcard Platzhalter. Sie unterscheiden sich eigentlich nur in Kleinigkeiten, aber diese können durchaus wichtig werden.

Generische Platzhalter

Generische Platzhalter matchen alle Zeichen außer `/` und `..`

```
$r->route('/user/:id');

URL                       id
/user/vorname.nachname    Keine Route matcht
/user/vorname             vorname
/user/vorname/            Keine Route matcht
/user/                    Keine Route matcht
/user/vorname nachname    vorname nachname
```

Relaxed Platzhalter

Relaxed Platzhalter matchen alle Zeichen außer `/`.

```
$r->route('/user/#id');

URL                       id
/user/vorname.nachname    vorname.nachname
/user/vorname             vorname
/user/vorname/            Keine Route matcht
/user/                    Keine Route matcht
/user/vorname nachname    vorname nachname
```

Wildcard Platzhalter

Relaxed Platzhalter matchen alle Zeichen - auch `/` und `..`

```
$r->route('/user/*id');

URL                       id
/user/vorname.nachname    vorname.nachname
/user/vorname             vorname
/user/vorname/            vorname
/user/                    Keine Route matcht
/user/vorname nachname    vorname nachname
```

Templates

Schon beim Erstellen der Anwendung mit *mojo* wird ein erstes Template angelegt. Der Speicherort hängt dabei auch vom Codepfad der Aktion ab: Der Code ist in `Example::welcome()`, also liegt das Template unter `example/welcome.html.ep`. Der Dateiname sagt dabei noch mehr aus. *html* steht für das Format, das ausgegeben wird, und das *ep* ist die Dateiendung der Template-Engine.

Wie man Templates für andere Formate und andere Template-Engines verwendet, wird in den kommenden Ausgaben noch Thema sein. Standardmäßig wird aber Mojolicious' eigene Template-Engine verwendet und in dieser Anwendung wollen wir erstmal nur HTML ausgeben.

Wird in der Aktion nicht angegeben, welches Template herangezogen werden soll, wird nach *controller/aktion.html.ep* geschaut.

```
# Im Controller TrackRuns::Example
sub template {
  my $self = shift;
  $self->stash( message => 'hallo' );
}
```

Wird diese Aktion ausgeführt, wird das Template *example/template.html.ep* verwendet. Möchte man sich von diesem Schema entfernen und ein anderes Template verwenden, kann man der Methode `render` den Namen des Templates angeben:

```
# Im Controller TrackRuns::Example
sub template {
  my $self = shift;
  $self->stash( message => 'hallo' );
  $self->render('test/nachricht');
}
```

Jetzt wird das Template *test/nachricht.html.ep* verwendet. Findet Mojolicious dieses Template nicht, sucht es wieder nach *example/template.html.ep*.

Schauen wir uns aber erstmal ein solches Template an:

```
% layout 'default';
% title 'Welcome';
<h2><%= $message %></h2>
This page was generated from the template
"templates/example/welcome.html.ep"
 <a href="<%== url_for %>">click here</a>
 to reload the page.
```

Hier sind gleich mehrere Sachen, die man sich näher anschauen kann. Die ersten beiden Zeilen enthalten Perl-Code - erkennbar daran, dass sie mit `%` anfangen. `layout` und `title` sind sogenannte *Helper*, kleine Hilfsfunktionen von denen Mojolicious schon einige vordefiniert. Dazu zählen eben `layout` und `title`, die beide die übergebenen Werte auf den Stash legen.

Das Layout ist ein HTML-Gerüst, in das man das aktuelle Template einbetten kann. Die Verwendung des Layouts kann entweder in jedem Template passieren, man kann das Template aber auch in der Aktion der `render`-Methode übergeben oder schon in `startup` das "globale" Layout festlegen:

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

  # Default layout
  $self->defaults(layout => 'mylayout');
}
```

Die Layouts werden im *templates/*-Ordner im Unterorder *layout* abgelegt.

Folgt dem `%` ein `=`, so wird die Zeile Perl-Code durch das Ergebnis ersetzt. Findet man ein `%= 8 + 5`, erscheint in der Ausgabe das Ergebnis der Operation. Sollen in dem Ergebnis XML-Entities ersetzt werden, so kann man `%==` verwenden.

```
% for my $i ( 1..3 ) {
%= $i + 5;
% }
```

Die erste und die dritte Zeile sind normaler Perl-Code, der kein Ergebnis liefert, deshalb hier die einfache Form `%`. Das Ergebnis der Addition soll aber in der Ausgabe erscheinen, deshalb hier die Form `%=`.

Aber nicht immer benutzt man eine ganze Zeile für den Perl-Code, sondern hat auch noch HTML oder anderen Text in der Zeile. Hierfür muss dann der Perl-Code in `<>` eingefasst werden:

```
% for my $i ( 1..3 ) {
<%= $i + 5; %><br />
% }
```

Werden Variablen im Template verwendet (z.B. `<%= $message %>`, müssen diese im Stash gesetzt sein, sonst gibt es eine Fehlermeldung:

```
Global symbol "$messages" requires
  explicit package ...
```

Das kommt uns doch bekannt vor: das sieht doch aus wie bei normalem Perl-Code. Das hängt damit zusammen, dass die Templates zu Perl-Subroutinen kompiliert werden.

Stash

Im ganzen Text war immer wieder vom "Stash" die Rede. Der Stash wird benutzt, um Daten in die Templates zu bekommen. Dazu gibt es einfach die Methode `stash`:

```
# Im Controller TrackRuns::Example
sub template {
  my $self = shift;
  $self->stash( message => 'hallo' );
  $self->render('test/nachricht');
}
```

Dabei können beliebige Werte im Stash gesetzt werden. Allerdings gibt es ein paar Standard-Stash-Werte, die von anderer Stelle schon gesetzt werden. Da sollte man ein wenig aufpassen, dass man sich damit nicht das Verhalten kaputt macht. Erinnern wir uns an die Routen:

```
$r->route( '/user/edit/:user_id' )->to(
    controller => 'user',
    action     => 'edit',
    user_id    => 1,
);
```

Diese drei Werte (controller, action und id) werden automatisch im Stash abgelegt, genauso wie das Format (zur Erinnerung: Aufruf von `http://localhost:3000/test.html`). So können wir ohne weiteres folgendes im Template schreiben:

```
Info:<br />
Controller: <%= $controller %><br />
Aktion: <%= $action %><br />
Typ: <%= eval '$format' %><br />
```

ohne dass wir etwas extra im Stash setzen müssen. In der letzten Zeile ist das `eval` notwendig, weil sonst ein Fehler kommt, wenn kein Format erkannt wurde. Es gibt eine Reihe von "besonderen" Namen, die man nach Möglichkeit nicht für die eigenen Werte nutzen sollte: action, app, cb, controller, data, extends, format, handler, json, layout, namespace, partial, path, status, template und text.

Die `stash`-Methode kann beliebig oft aufgerufen werden und der Stash existiert auch so lange, bis eine Antwort generiert wurde.

Um an die Werte von Namen zu kommen, muss man einfach nur den Namen übergeben:

```
my $controller =
  $self->stash('controller');
```

Fazit und Ausblick

In dieser Ausgabe wurde gezeigt, wie man eine Mojolicious-Anwendung beginnt und wie man das Routing einrichtet. Außerdem wurde gezeigt, wie man mit Platzhaltern diese Routen dynamisch hält. Templates waren ein weiteres Thema: Wo liegen die Templates, wie sieht die Templatesprache bei Mojolicious aus. Als letztes wurde erklärt was der Stash ist und wie man diesen nutzt.

In der nächsten Ausgabe wird besprochen, wie man die Anwendung konfiguriert und wie man an die gesendeten Daten etc. kommt. Weitere Themen werden Mojolicious-Sessions, Logging und unterschiedliches Verhalten von verschiedenen Umgebungen sein. Auch das Thema Rendering wird intensiver besprochen.

EVENT

15. Deutscher Perl-Workshop: Call for Papers

Vom 13.03.2013 bis 15.03.2013 (Mittwoch bis Freitag) findet der 15. Deutsche Perl-Workshop im Betahaus in Berlin statt. Zielgruppe des Workshops sind alle ernsthaften Perl-Anwender und die, die es werden wollen.

Unser Workshop steht und fällt mit den Vorträgen. Üblicherweise sind Vorträge 5, 20 oder 40 Minuten lang. Alle Themen, die in irgendeiner Weise mit Perl oder dem Perl-Umfeld zu tun haben, können als Vorträge für den Workshop interessant sein.

Deinen Vorschlag reichst Du bitte bis spätestens Donnerstag, den 13.12.2012 als Abstract auf der Webseite h*ttp://act.yapc.eu/gpw2013* ein. Du kannst natürlich gerne auch mehrere Themen vorschlagen. Dein Abstract sollte in rund 2000 Zeichen (das sind ca. 30 Zeilen a 72 Spalten) das Thema beschreiben, was besonders an Deinem Ansatz ist und weshalb Perl als Sprache in diesem Fall besonders nützlich ist.

Solltest Du noch Fragen haben oder Anregungen für Vortragsthemen suchen, schaue bitte in den FAQ nach.

Der Deutsche Perl-Workshop wird seit 1999 jährlich ausgerichtet und wandert seit 2004 durch Deutschland. Durchschnittlich 100 Personen nehmen am Workshop teil.

In diesem Jahr wird der Workshop von den Berliner Perlmongers organisiert. Die Perlmongergruppe gehört zu den größten und aktivsten in Deutschland. An jedem letzten Mittwoch des Monats treffen wir uns an wechselnden Orten. Mehr Infos gibt es auf Twitter unter *https://twitter.com/BerlinPM* und unter *http://www.perlmongers.de/?BerlinPM*.

Herbert Breunung

Rezension - Arbeitsgrundlage

Peteris Krumins
Perl One-Liners Explained
Eigenverlag, Februar 2012
111 Seiten, PDF (englisch)
catonmat.net
$ 9,95

Curtis 'Ovid' Poe
Beginning Perl
Wrox, Sept. 2012
744 Seiten, Tachenbuch (englisch)
ISBN 978-1-1180-1384-7
$ 39,99

Mathias Geirhos
IT-Projektmanagement
Galileo, 2011
192 Seiten, brosch.
ISBN 978-3-8362-1773-6
€ 12,90

Zurück zu den Wurzeln. Dieses Mal geht es um Wissen, dass man für seine professionelle Programmierarbeit unbedingt besitzen sollte. Das neue *Beginning Perl* will die berechtigte Frage beantworten: "Womit muss man vertraut sein, um reale Aufträge selbständig lösen (... und sich die restlichen Details im Netz selbst heraussuchen) zu können?". Im zweiten Buch von Mathias Geirhos geht es darum, wie das Projekt geplant und durchführt wird, ohne das man selbst oder das Vorhaben darunter leidet. Doch zum Einstieg soll es um winzige Perlskripte gehen, welche bei den kleinen, nebenbei anfallenden, Arbeiten nützlich sind. Es sei auch nachgereicht, dass die im Frühjahr (1/2001) rezensierte 5. Auflage der Taschenreferenz inzwischen von Peter Klicman übersetzt wurde. Als `ISBN 978-3-86899-187-1` ist es gedruckt für €9,90 und unter `ISBN 978-3-86899-188-8` ist das PDF für €7,90 zu bekommen.

Perl One-Liners Explained

Während das letztens vorgestellte *eBook Zukunft.pl* sich aus einem Lehrgang entwickelte und sein Autor es mit Blogbeiträgen (*http://www.hidemail.de/blog/*) umrahmte, ging Peteris Kruminis den umgekehrten Weg. Manch einer mag bereits einhaken: "Peteris wer?". Der Name ist in Perlkreisen kein bekannter und tatsächlich scheint der junge Mann kein ausgewiesener Experte zu sein. In diesem Fall stört das jedoch nicht wirklich. Denn was er schreibt stimmt auf jeder Zeile und gute, leichte Literatur für aufstrebende Perlentwickler zu schreiben ist kein Verbrechen. Im Gegenteil: das sorgfältig erstellte kleine Buch bietet für mittleres Geld eine bisher nicht publizierte Art, Perl in kleinen, leicht verdaulichen Portionen zu lernen. Anhand von Einzeilern, die sogar öfters nützliche Aufgaben vollführen, lernt der Leser die wichtigsten Befehle, Spezialvariablen, Kommandozeilenparameter, Regex-Spezialzeichen und eine Hand voll Module kennen. All dies ist gut indiziert und übersichtlich in Kapitel eingeordnet, deren Übersicht auch von Lesegeräten als solche erkannt wird (was leider nicht immer so ist).

Für Unschlüssige gibt es die angesprochenen Blogartikel *http://www.catonmat.net/blog/perl-one-liners-explained-part-one/*, aus denen das Buch entstand, nach wie vor im Netz. Der Ursprung liegt jedoch bei Eric Pement (*http://www.pement.org*). Dessen Sammlungen von Einzeilern für die Arbeit in der Kommandozeile awk1line.txt und sed1line.txt sind seit Jahren ein beliebter und frei verfügbarer Einstieg in *awk* und *sed*. Peteris schrieb bereits zwei Bücher, welche genauer aber bündig erläutern, warum jeder Einzeiler tut was er verspricht. Diese Liste übersetzte er dann nach Perl als *http://www.catonmat.net/download/perl1line.txt* (freier Download) und schrieb dazu ebenfalls kurze Erklärungen. Deshalb ist es auch eine gute Gelegenheit für Perlprogrammierer, anhand von Beispielen, in `sed` oder `awk` einzutauchen und umgekehrt.

Nur Abigails Primzahlendetektor wird in mehr als fünf beschwingten aber sachlichen Sätzen erklärt. Alles dreht sich um einfaches Perl für den Gebrauch in der Befehlszeile - ohne ausgefeilte Tricks. Auch für Einzeiler prädestinierte Module wie `IO::All` gehen bereits über den niedrig abgesteckten Horizont hinaus. Wirklich anspruchsvolles Golfen gibt es auf *http://deoxy.org/meme/Perl/OneLiners* oder *http://terje2.frox25.no-ip.org/*. Und wer diese Materie mag, kann sich sich sicher auch für Philippe "BooK" Bruhats (der kleine Franzose mit dem rosa T-Shirt) Übersicht geheimer Perl-Operatoren *http://github.com/book/perlsecret/blob/master/lib/perlsecret.pod* begeistern.

Beginning Perl

Ebenso von einem kleinem Beispie zum Nächsten schwingt sich auch Ovids Buch. Doch Curtis Poe, Koautor der 2/2012 rezensierten *Perl Hacks*, fährt auf diesen Seiten in weit tiefere Gewässer. *Beginning Perl* klingt zwar nach einem Anfängerbuch, es ist auch eines - aber für Berufsanfänger. Daher auch der inoffizielle Untertitel: "get a job, hippy". Es ist für jemand, der beim Bewerbungsgespräch gepokert hat ("Ich kann C, Java und Ruby, da kann Perl ja nicht so schwer sein."), und nun sich vor der Aufgabe wiederfindet, in Perl aus einer Datenbank Informationen zu stochern, um aus ihnen *Templates* für eine *Catalyst*-Anwendung zu erstellen: unicodesicher, geschwindigkeitsoptimiert, mit Tests und bis morgen Mittag. Nun ist glücklich, neben einem gesprächsbereitem Experten sitzt oder sich wenigstens mit dieser Schwarte in eine stille Ecke verziehen kann, was ersteren Fall erfolgreich simuliert. Ein Computer mit Perl um die Beispiele nachzuvollziehen und die Aufgaben zu lösen wäre praktisch, aber erfreulicherweise achtete der Schreiber darauf, dass selbst an einem stillen Ort höchstens eine zusätzliche Rolle Zellulose benötigt wird, aber keine weitere Dokumentation.

Denn mit wohltuend wenigen Worten kommt Meister Ovid zum Wesentlichen und belässt es dabei auch. Angefangen wird am Anfang. Wer nicht träge im Kopf ist, könnte hiermit auch Programmieren lernen. Das täte er auf sehr lockere Art, da Curtis den Leser in einer leicht geglätteten Alltagssprache als gleichrangig anspricht. Stellenweise ragen sogar sarkastische Spitzen hervor, mit denen Curtis Poe sein Verständnis für die überraschenden Momente zeigt, die Perl einem Anfänger oder Umsteiger bereiten kann. Diese lernen hier manchmal auch modernes Perl (cromatic war Lektor), guten Stil und strategisches Denken - hauptsächlich aber das was der Autor während der letzten zehn Jahre in etlichen Firmen vorfand: Perl 5.8 und 5.10, `DBI`, `Template::Toolkit`, *Test*, *CSV*, *CLI* usw., aktuellere Sachen wie `Catalyst`, `DBIx::Class` und `Moose` werden nur kurz überflogen. Der Aufbau der 18 Kapitel ist vorbildlich. Die Titel sind kapp und gut gewählt, die einleitenden Listen geben einen schnellen Überblick, und der Inhalt ist logisch in Unterkapitel aufgeteilt , wovon die letzten beiden immer eine kurze Wiederholung und Lösungen der eingestreuten Übungsaufgaben beinhalten. In der freien Online-Version im O'Reilly OFPS unter *http://ofps.oreilly.com/titles/9781118013847/* findet man darüber hinaus Kommentare der Leser.

Das Druckwerk erreichte im September 2012 das Tageslicht. Jeder der etwas anderes behauptet, bezieht sich wahrscheinlich auf das gleichnamige Buch von Simon Cozens aus dem Jahr 2000, welches auf *http://www.perl.org/books/beginning-perl/* gelesen werden kann und welches 2004 von James Lee für den Apress-Verlag aktualisiert wurde.

IT-Projektmanagement

Matthias Gerhos war sich des Umstands voll bewusst, dass er über ein gern gemiedenes Thema vorträgt, über das schon viel dahergeschwafelt wurde. Deswegen komprimierte er seine Erfahrung auf 190 DIN A5 Seiten und tat sehr gut daran. Denn bei Lichte betrachtet findet sich viel, dass so klingt, als wenn man darauf auch selber gekommen wäre. Unterhaltsam geschrieben ist es trotzdem.

Der Wert des Buches erwächst aber aus der langjährigen Erfahrung des Verfassers, und der Tatsache, dass er sich selbst und die eigene Rollen als Projektleiter und Autor nicht bitterernst nimmt. Lakonisch und in schnell wechselnden Bildern beschreibt er realistisch das Ringen um Ziele, die Erfolge und das Scheitern von Vorgaben - letzteres niemals ohne das passende Gegenmittel zu erwähnen. Dadurch hat der Leser in jeder Situation eine bewährte Prüfliste bei der Hand und Anleitung zu Selbstreflektion. Denn eine zuversichtliche aber disziplinierte Grundhaltung, feinfühliges und psychologisches vorausschauendes Handeln und vor allem der Fokus auf das Gesamtbild scheint einen erfolgreichen Projektmanager auszumachen. Um diese drei Eigenschaften zu verin-

nerlichen und im Projektalltag behaupten zu können, liest man hauptsächlich diese Seiten, nicht um Unerwartetes zu hören. Praxisnahe Diagramme, Übersichten und Fallbeispiele helfen dabei.

Ein Höhepunkt ist zweifellos die Kurzeinführung in die Scrum-Methode im vorletzten Kapitel. Wer sich vor Augen hält, wie viele dicke Bücher allein darüber geschrieben wurden, erkennt hier eine Qualität des Verfassers, die Kernpunkte auf neun kleinen Seiten aufzählen zu können. Es zeugt auch von Charakterstärke nicht dem Massenwahnsinn zu verfallen und auch die Schattenseiten der allseits hochgelobten agilen (beweglichen) Methodik sehen zu können. Die abschließende Fabel zeugt ebenfalls davon, und betont noch einmal die Grundhaltung und Tugenden eines erfolgreichen Projektmanagers.

Ausblick

Anfang 2013 wird voraussichtlich angesehen:

- *Perl-Bundle* des Linux-Magazin vom Perlmeisters Michael Schilli
- *Perl* von Udo Müller
- *Der leidenschaftliche Programmierer* von Chad Fowler

TPF News

Grant: Spanische Übersetzung der Perl-Dokumentation

Unter https://github.com/zipf/perldoc-es sind die Arbeiten an der spanischen Übersetzung der Perl-Dokumentation einsehbar. Neben der Übersetzung neuer Dokumente wurde auch darauf geachtet, schon übersetzte Texte an Änderungen in Perl 5.16.0 anzupassen.

Aktuell ist rund 1/3 der Dokumentation übersetzt. Insgesamt gilt es 168 Dokumente mit knapp 1 Million Worte zu übersetzen.

Während der Übersetzung prüft das Team auch, ob die Code-Beispiele funktionieren und mit der Dokumentation übereinstimmen.

Grant: Fixing Perl5-Bugs

Die Änderungen an Code-Blöcken in Regulären Ausdrücken (`(?{}>`) sind in Perl blead eingeflossen und werden in Perl 5.18.x enthalten sein. Code-Blöcke in Suchmustern werden im gleichen Durchgang geparst wie der normale Code. Damit sind auch Konstrukte wie `(?{ $x = '{' })` möglich, die bisher Fehlermeldungen (`Sequence (?{...}) not terminated or not {}-balanced in regex`) geworfen haben.

Weiterhin werden literale Code-Blöcke nur einmal kompiliert:

```
for my $p (...) {
    # this 'FOO' block of code is
    # compiled once, at the same time as
    # the surrounding 'for' loop
    /$p{(?{FOO;})/;
}
```

Leider passiert bei solch großen Änderungen auch der ein oder andere Fehler. Im Juli und August hat Dave Mitchell diese Fehler beseitigt.

Ebenfalls im August hat Mitchell sich angeschaut, warum

```
$&;
$_ = 'x' x 1_000_000;
1 while /(.)/g;
```

mehrere Minuten läuft und ohne das `$&` sehr schnell ist. Perl kopiert den String-Puffer nicht wenn `/g` verwendet wird - auch dann nicht wenn es eine Capture-Group gibt. Dave Mitchell hat die Regex-Engine so angepasst, dass bei Verwendung von `$&` und ähnlichem nicht mehr der ganze String-Puffer kopiert wird, sondern nur noch der interessante Bereich des Strings. Dadurch hat die Verwendung von `$&` nicht mehr die Performanz-Nachteile wie bisher.

Grants verlängert

Die Grants von Nicholas Clark und Dave Mitchell wurden verlängert, so dass sie jeweils weitere 400 Stunden in ihren Grants arbeiten können.

Grant: Cross-Compilation

Wie schon berichtet, möchte Jess Robinson die Cross-Compilation für Perl verbessern und konzentriert sich dabei auf Android. Bisher wurde viel an der Konfiguration von Perl gearbeitet.

```
$ cat /tmp/foo.pm
print STDERR __FILE__ . " was not the file you expected to get loaded?\n";
$ ~/Sandpit/5000/bin/perl -e 'require ::tmp::foo;'
/tmp/foo.pm was not the file you expected to get loaded?
$ echo $?
0
```

Listing 1

Grant: Improving Perl 5

Auch in den letzten Monaten hat Nicholas Clark intensiv an Perl 5 gearbeitet. Unter anderem hat er an sicherheitsrelevanten Dingen bzgl. `require` gearbeitet. Dieses Sicherheitsleck betraf diejenigen, die Dateien von der Platte geladen haben und Benutzereingaben nicht validiert haben. So wurde mit `require ::foo` die Datei "/foo.pm" geladen. Nach den Änderungen von Nick wurde nicht die komplette Platte durchsucht, sondern nur die Verzeichnisse in `@INC` (siehe Listing 1).

Auch XS-Code ist anfällig wenn man die Funktionen `Perl_load_module()` oder `Perl_vload_module()` verwendet. Diese Funktionen erwarten einen Package-Namen und validieren nicht die übergebenen Werte.

Nebenbei hat Clark Probleme in `File::stat` festgestellt: `-x` und `-X` lieferten falsche Ergebnisse wenn das Skript unter *root* gelaufen ist. Die Built-Ins `-x` und `-X` haben diesen Fehler nicht gehabt.

Weitere Punkte die bearbeitet wurden:

• Unnötiger Code in `Class::Struct`
"Unnötiger" Code - benötigt für Kompatibilität mit Perl 5.002 und Perl 5.003 - in `Class::Struct` hat Auswirkungen auf Fehlermeldungen. Im Zuge des Aufräumens wurden die Tests aufgemöbelt - von 26 Tests auf 69 Tests.

• Config::Perl::V
`Config::Perl::V` ist ein Modul von H. Merijn Brand zum strukturierten Zugriff auf Informationen von `perl -v`. Nick Clark hat an dem Modul gearbeitet, so dass es die neueren Methoden von `Config` benutzt.

Neuer Verantwortlicher für "Konferenzen"

Im Gegensatz zur YAPC::Europe wird die YAPC::NA von der Perl Foundation organisiert. Seit August 2012 ist Heath Bair der neue Verantwortliche für Konferenzen innerhalb der Perl Foundation.

Grant: Cooking Perl with Chef

Chef ist ein Open Source Configuration Management Tool und David Golden hat im Rahmen seines Grants Tools entwickelt, mit denen Perl-Programmiererinnen ihre Anwendungen verlässlich und einfach auf verschiedenen Maschinen installieren können. Als Ergebnis sind drei "Kochbücher" entstanden:

- Perl Interpreter Deployment
  http://community.opscode.com/cookbooks/perlbrew
- CPAN Modul Deployment
  http://community.opscode.com/cookbooks/perlbrew
- Plack Anwendungen deployen
  http://community.opscode.com/cookbooks/carton

Zusätzlich wurde eine Webseite für Perl + Chef eingerichtet: http://perlchef.com/. Auf dieser Webseite sind auch Unterlagen zur Einführung in Perl und Chef zu finden.

Auf der Mailingliste für DevOps (http://lists.perl.org/list/perl-devops.html) können Interessierte über Tools und Erfahrungen zu Deployment Tools diskutieren.

Mit Pantry (https://metacpan.org/module/Pantry) existiert ein Tool, das bei der Erstellung von Konfiguration helfen soll.

Grant: Refactoring Perl Debugger

Einer großen Aufgabe hat sich Shlomi Fish zugewendet: Er möchte den Perl-Debugger refactorn. In den ersten Schritten werden erstmal Tests hinzugefügt. Der Debugger ist nicht ganz einfach und es gibt mehrere Module, die etwas Ähnliches machen wie der Debugger. Auch Rocky Bernstein hat mit Devel::Trepan einen Debugger-Ersatz angefangen.

Grant: Devel::Cover

`Devel::Cover` hat einige Updates erfahren. Dabei hat Paul Johnson einen Perl-Bug entdeckt (https://rt.perl.org/rt3//Public/Bug/Display.html?id=113464), bei dem `B::Deparse` eine Warnung ausgibt. Johnson hat sowohl einen Patch für `B::Deparse` eingereicht als auch `Devel::Cover` so geändert, dass Warnungen im Zusammenhang mit `B::Deparse` abgeschaltet werden.

Auch an *cpancover* hat Johnson gearbeitet.

Im Juli hat Paul Johnson die Installation von `Devel::Cover` gefixt: Es gab Probleme wenn Devel::Cover in Verzeichnissen mit Leerzeichen gebaut wurde. unter MacOS X kam das wohl häufiger vor.

In neueren Devel::Cover-Versionen holt sich das Modul zur Installationszeit die @INC-Verzeichnisse von Perl. Diese wurden in einer Variablen innerhalb von Devel::Cover gespeichert. Das kann aber zu Problemen führen, wenn sich die Umgebung in der Devel::Cover gebaut wird von der Umgebung unterscheidet, in der das Modul später verwendet wird. Johnson möchte diese Informationen in Zukunft von Config::Perl::V auslesen.

In seinem August-Bericht hebt Paul Johnson auch hervor, dass es ein großer Vorteil des Grants ist, dass er sich die Ergebnisse von CPANTester und Durchläufen von Devel::Cover genauer anschauen kann und sich auch näher mit Feedback aus der Community auseinandersetzen kann.

Auf dem QA Hackathon in Paris hat Johnson ein vim Plugin geschrieben, dass coverage-Informationen in vim anzeigt.

Grant: Lists, Iterators, and Parcels

Patrick Michaud hat einen Zwischenbericht zu seinem Grant "Lists, Iterators, and Parcels" abgegeben. Er schreibt, dass sich seit Annahme des Grants einiges verändert hat, so dass die ursprünglich angedachten Arbeiten teilweise überflüssig wurden bzw. geändert werden mussten. So hat sich die Performanz von Listen und Iteratoren deutlich verbessert und in Rakudo sind diese nicht mehr in PIR sondern in Perl 6 umgesetzt. In diesem Zusammenhang wurde auch die Synopsis 7 (Listen und Iteratoren) aktualisiert.

Michaud hat über seine Arbeiten auch Vorträge auf der YAPC::EU 2011 und der YAPC::NA 2012 gehalten.

Grant: Alien::Base

Joel Berger hat zusammen mit David Mertens an Problemen mit Alien::Base auf MacOS gearbeitet. Jetzt laufen die Tests auf dem Mac durch. Als "realen" Test seiner Arbeiten hat Berger das Modul ACME::Alien::DontPanic erstellt. Während der Arbeiten hat Joel Berger sich mit autoconf/automake und libtool beschäftigt und schätzt jeden glücklich, der sich damit nicht auseinandersetzen muss.

YAPC::NA 2013 in Austin

Als Austragungsort für die nordamerikanische Perl-Konferenz wurde Austin/Texas ausgewählt. Ort und Datum werden noch bekannt gegeben. Vortragsvorschläge werden schon angenommen.

CPAN News XXIV

Calendar::Slots

Termine, Termine, Termine... heutzutage geht fast nichts mehr ohne Kalender. Von einem Meeting zum Anderen, weggehen mit Freunden und vieles mehr, aber wann hat man noch Zeit? Mit `Calendar::Slots` kann man sich einen ganz einfachen Kalender bauen.

Der Kalender ist wirklich ganz einfach, da kein iCal-Import/Export möglich ist und weil bei Terminkollisionen kein Hinweis kommt - der letzte Termin gewinnt.

Trotz dieser Einschränkungen ist `Calendar::Slots` ein ganz nettes Modul...

```
use Calendar::Slots;

my $cal = new Calendar::Slots;

$cal->slot(
    date  => '2012-10-11',
    start => '10:30',
    end   => '11:30',
    name  => 'write article for $foo',
);

my $slot = $cal->find(
    date => '2012-10-11',
    time => '11:00',
);

print $slot->name;
# prints "write article for $foo"
```

Gibt es beispielsweise wöchentliche Termine, kann man statt mit einem Datum auch direkt mit dem Wochentag arbeiten:

```
$cal->slot(
    weekday => 1,
    start   => '07:30',
    end     => '08:00',
    name    => 'Eine neue Woche beginnt',
);
```

Täglich wiederkehrende Termine können für einen bestimmten Zeitraum mittels `start_date` und `end_date` statt des `weekday` angegeben werden. Vielmehr Kombinationsmöglichkeiten bietet das Modul allerdings nicht.

Hat man alle Termine beisammen, so kann man sich diese auch ganz nett auf der Konsole anzeigen lassen:

```
# define the time period we want to show
my $this_week = $cal->materialize(
    2012_10_08,
    2012_10_14,
);
print $this_week->as_table;
```

Die Ausgabe ist in Listing 1 zu sehen.

```
.-------------------------------------------------------------------.
| name                    | start | end  | when     | type    | _weekday |
+-------------------------+-------+------+----------+---------+----------+
| Eine neue Woche beginnt |  0730 | 0800 |        1 | weekday |          |
| write article for $foo  |  1030 | 1130 | 20121011 | date    |        4 |
'-------------------------+-------+------+----------+---------+----------'
```

Listing 1

SVG::Sparkline

Graphen und Grafiken werden in vielerlei Situation herangezogen, um Fakten übersichtlich darzustellen. Daten zu sammeln und in Graphen aufzubereiten, kann einige Zeit in Anspruch nehmen. Da ist es praktisch, wenn man einiges an Arbeit automatisieren kann.

Auf CPAN gibt es etliche Module, wie z.B. `GD::Graph`, die genau zu diesem Zweck programmiert wurden. Um Graphen für verschiedene Zwecke nutzen zu können, eignet sich als Basis das SVG-Format hervorragend, weil man damit die Zeichnungen in den verschiedensten Größen ohne Qualitätsverlust verwenden kann. In der Ausgabe 17 des Perl-Magazins wurde schon ausführlich gezeigt, wie man mit Perl-Programmen SVG-Grafiken erstellen kann.

In diesen CPAN-News wird ein neues Modul vorgestellt: `SVG::Sparkline`.

*Sparkline*s sind kleine Grafiken, die den Verlauf einer Messung wie z.B. Temperaturen oder Festplattennutzung darstellen. Dabei wird häufig auf Achsen und Koordinaten verzichtet.

`SVG::Sparkline` unterstützt zur Zeit sechs Arten von Grafiken:

- Area
- Bar
- Line
- RangeArea
- RangeBar
- Whisker

Da an dieser Stelle das Modul nur kurz vorgestellt werden soll, wird hier nur an Hand von *Line*-Grafiken gezeigt werden, was man mit dem Modul machen kann. Die anderen Graph-Typen funktionieren analog.

Als erstes soll eine *Line*-Grafik die Auslastung des Laptops, wofür die Load-Avg-Werte herangezogen werden (siehe Listing 2).

Das daraus resultierende SVG ist in Abbildung 1 zu sehen

Interessant wird es auch, wenn bestimmte Werte farbig markiert werden. So lässt sich der maximale Wert in der Grafik rot hinterlegen. Dazu muss man nur noch den Parameter `mark` angegeben werden:

```
my $svg = SVG::Sparkline->new(
    Line => {
        values => \@values,
        color  => 'black',
        height => 12,
        mark   => [
           high => 'red',
        ],
    }
);
```

Ergebnis siehe Abbildung 2.

Neben `high`, `low`, `first` und `last` kann man auch einen Index angeben für den Punkt der markiert werden soll. Auch die Angabe mehrerer Markierungen ist möglich:

```
my $svg = SVG::Sparkline->new(
    Line => {
        values => \@values,
        color  => 'black',
        height => 12,
        mark   => [
            3 => '#cdcdcd',
            6 => '#0f0f0f',
        ],
    }
);
```

Ergebnis siehe Abbildung 3.

Abb. 1: Die Load-Werte des Laptops über 5 Minuten

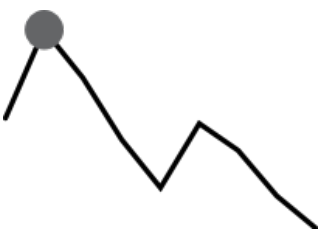


Abb. 2: Markierter Spitzenwert

Abb. 3: Mehrere markierte Werte

```
use IO::File;
use SVG::Sparkline;
use Sys::Statistics::Linux::LoadAVG;

my @values;
my $counter = 1;

my $stats = Sys::Statistics::Linux::LoadAVG->new;
while ( 1 ) {
     my $load = $stats->get;
     push @values, [ $counter, $load->{avg_1} ];

     $counter++;
     last if $counter == 10;
     sleep 30;
}


my $svg = SVG::Sparkline->new(
    Line => {
        values => \@values,
        color  => 'black',
        height => 12,
    }
);

my $fh = IO::File->new( 'test.svg', 'w' );
$fh->print( $svg->to_string );
$fh->close;
```

Listing 2

App::Cerberus

Woher kommen die Webseitenbesucher, welcher Browser wird verwendet? Diese und noch mehr Fragen stellen sich, wenn man Webseiten betreibt. Diese Informationen in jeder Anwendung herauszufinden ist mühsam. `App::Cerberus` ist eine fertige Anwendung, die man in den einzelnen Anwendungen einbinden kann. Es ist über Plugins möglich, weitere Informationen zur Verfügung zu stellen. In den einzelnen Webanwendungen muss man nur noch einen HTTP GET-Request an Cerberus abschicken und man bekommt die ganzen Informationen als JSON zurückgeliefert.

Nach der Installation von `App::Cerberus` muss man nur `cerberus.pl` starten und schon kann man mit den Abfragen beginnen. Konfiguriert wird Cerberus über YAML-Dateien.

Folgende Plugins werden gleich mitgeliefert:

- App::Cerberus::Plugin::Throttle
- App::Cerberus::Plugin::BrowserDetect
- App::Cerberus::Plugin::TimeZone
- App::Cerberus::Plugin::GeoIP

Mit `App::Cerberus::client` gibt es auch schon einen passenden Client

```
use App::Cerberus::Client;

 my $client = App::Cerberus::Client->new(
     servers => 'http://localhost:5000',
 );

 my $info = $client->request(
     ip => '80.1.2.3,
     ua => 'Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)'
 );
```

X11::WindowHierarchy

Wer Informationen zur Fenster Hierarchie haben möchte, sollte sich `X11::WindowHierarchy` anschauen.

```
use X11::WindowHierarchy;

my @windows = x11_filter_hierarchy(
   filter => qr/\w/
);
printf "window [%s] (id %d)%s\n",
    $_->{title},
    $_->{id},
    $_->{pid} ? ' pid ' . $_->{pid} : ''
        for @windows;

# Dump all information we have about all windows on display :1
use Data::TreeDumper;
print DumpTree(x11_hierarchy(display => ':1'));


~/Magazine/Foo/Issue24/code/cpan_news$ perl x11_window_hierarchy.pl
Found window [gnome-session] (id 10485761) pid 1679
Found window [gnome-settings-daemon] (id 12582913) pid 1742
Found window [compiz] (id 14680065) pid 1755
Found window [gnome-screensaver] (id 16777217) pid 1754
Found window [gnome-screensaver] (id 16777219) pid 1754
```

WWW::Testafy

Testafy ist ein Tool für automatisierte Webseiten-Tests. Im Gegensatz zu den bisher existierenden Tools in der Perl-World können die Testanforderungen hier in "normalem" Englisch verfasst werden. Was man benötigt ist `WWW::Testafy` und einen Account bei *http://www.testafy.com*.

Ein Beispiel für einen Test:

```
For the url http://www.perlybook.org/perltuts
Given a test delay of 2 seconds
When the CPAN link is clicked
Then take a screenshot
```

Einer großen Erklärung bedarf das eher nicht, weil die Testbeschreibung aussagekräftig ist.

Im Perl-Programm sieht das wie folgt aus.

```
use WWW::Testafy;

my $te = new WWW::Testafy;

my $id = $te->run_test(
    pbehave  => qq{
        For the url http://www.perlybook.org/perltuts
        Given a test delay of 2 seconds
         When the CPAN link is clicked
        Then take a screenshot
    };
);

my $passed  = $te->test_passed($id);
my $planned = $te->test_planned($id);
print "Passed $passed tests out of $planned\n";
print $te->test_results_as_string($id);
```

Während der Beta-Phase ist der Dienst noch kostenlos, über die Preise danach ist noch nichts bekannt.

App::Birthday

Nie mehr vor Freunden blamiert dastehen, nur weil man mal den Geburtstag vergessen hat. Wie das geht? Mit `App::Birthday`! Das Modul liefert gleich das Programm `birthday` mit. Das Programm muss mit zwei JSON-Dateien konfiguriert werden: Mit einer JSON-Datei für die Geburtstage (Standardname ist *birthday.json*) und einer Datei für die Mailkonfiguration (Standard *config.json*).

Die *birthday.json* sieht so aus:

```
{
    "carol" : {
        "date" : "16.9",
        "friends" : {
            "names" : ["ted"],
            "subject" : "Carol has today birthday!"
        },
        "email" : "carol@somewhere.com",
        "subject" : "Happy Birthday Carol!",
        "text" : "Happy Birthday"
    },
    "ted" : {
        "date" : "25.11",
        "friends" : {
            "names" : ["carol"],
            "subject" : "Ted has today birthday!"
        },
        "email" : "ted@another.com",
        "subject" : "*** Happy Birthday! ***",
        "text" : "Happy Birthday"
    },
}
```

Über "friends" kann man auch steuern, welche Freunde noch an den Geburtstag des Geburtstagskinds erinnert werden soll.

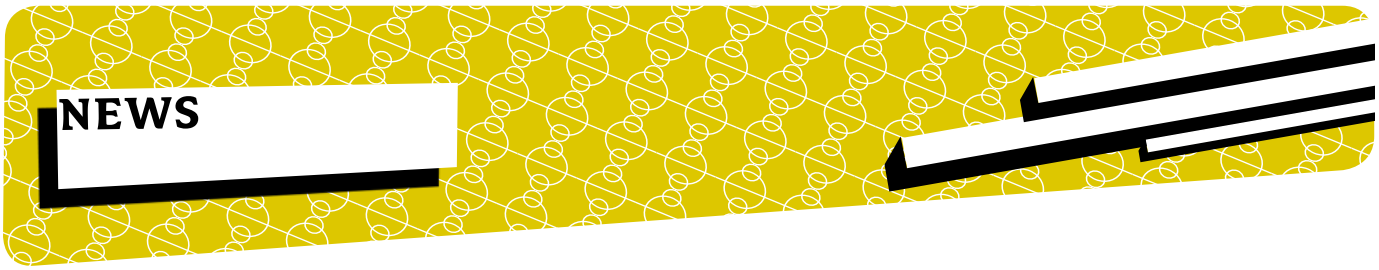
Wenn man das Modul selbst verwenden möchte, benötigt man nur die Funktion `send_mails`:

```
use App::Birthday;

my %friends = (
  Hugo => {
      email   => 'hugo@foo-magazin.de',
      date     => '24.12',
      subject => 'Happy Birthday altes Haus',
      text      => q~Hallo Hugo,
wir wünschen Dir alles Gute zum Geburtstag

Foo-Magazin~,
   },
);

my %config =  (
      "from" : "bob@foo-magazin.de",
      "maintainer" : "bob@foo-magazin.de",
      "transport" : {
          "host" : "mail.foo-magazin.de",
          "port" : "25"
      }
  );
send_mails( 'Hugo', \%friends, \%config );
```



Perl5-Porters-News

Ab dieser Ausgabe nehmen wir eine neue Kategorie in das Magazin auf - die Perl5-Porters-News. Hier werden wir immer über aktuelle Diskussionen auf der Perl5-Porters-Mailingliste berichten. Denn viele Diskussionen sind auch für Nicht-Kern-Entwickler interessant.

Die Perl5 Porters sind eine Gruppe von Personen, die sich um die Weiterentwicklung der Sprache Perl kümmert.

Sicherheitssachen in Perl 5.16.x

Eine lange und ausgedehnte Diskussion drehte sich um ein potentielles Sicherheitsloch in Perl 5.16.x. Reini Urban wies darauf hin, dass Perl Null-Bytes (`\0`) bei allen möglichen Eingaben erlaubt - auch bei Systemaufrufen und in verschiedenen Datenstrukturen. Er meinte, dass ein Angreifer damit Shell-Code in den Speicher laden könnte.

Die Perl 5 Porters konnten aber keinen realen Angriffsvektor finden. Dennoch sind viele der Meinung, dass am Null-Byte-Handling etwas geändert werden sollte. Weil es z.B. beim Öffnen von Dateien zu Problemen kommen kann:

```
open my $fh, '>', "test.pl\0hallo";
```

öffnet `test.pl`.

Fließkommazahlen runden

Ein Bugreport meldete, dass nach einem Umzug von i386 nach x64_64 unter CentOS 6 das Rundungsergebnis einer Fließkommazahl unterschiedliche Ergebnisse liefert:

Auf x86:

```
# perl -e 'printf("%.2f", 6.685);'
6.68
```

Auf x64:

```
# perl -e 'printf("%.2f", 6.685);'
6.69
```

Als Lösung für gcc 4.5+ wurde folgendes präsentiert: Man kann Perl mit dem Flag `-fexcess-precision=standard` kompilieren.

CPANPLUS aus dem Kern entfernen

SawyerX hat vorgeschlagen, CPANPLUS aus dem Kern zu entfernen. Craig Berry hat dem widersprochen, weil CPAN.pm (und andere CPAN-Clients) unter VMS nicht funktioniert. Im Laufe der Diskussion wurden einige mögliche Fehlerquellen in CPAN.pm gefunden. So wurden Pfade nicht richtig zusammengesetzt und Listen-Zuordnungen (z.B. `local %ENV = ( 1 => 2 )` funktionieren unter VMS nicht. Wenn CPAN.pm unter VMS läuft, wird CPANPLUS wahrscheinlich aus dem Kern entfernt.

Davon betroffen wären auch einige Kernmodule, die nur für CPANPLUS im Kern sind.

http://www.nntp.perl.org/group/perl.perl5.porters/2012/09/msg193264.html

a2p / s2p

Auf der Mailingliste der Perl 5 Porters wurde gefragt, ob die Tools `a2p` und `s2p` noch von Personen verwendet werden. `a2p` und `s2p` übersetzen *awk* bzw. *sed* Skripte nach Perl. Wer die Tools noch verwendet, sollte sich bei den Perl 5 Porters melden, nicht dass dann auf einmal die Tools verschwunden sind.

Bessere Fehlermeldung, wenn Modul nicht gefunden wurde

Wer in häufig in Foren unterwegs ist, wird oft die Fehlermeldung `Can't locate Stuff/Of/Dreams.pm in @INC (@INC contains: ...)` zu sehen bekommen. Für Anfänger ist diese Nachricht nicht aussagekräftig genug. Paul Johnson hat einen Patch eingereicht, der diese Fehlermeldung erweitert:

```
Can't locate Stuff/Of/Dreams.pm in
 @INC (did you install the Stuff::Of::Dreams
  module?) (@INC contains: ...)
```

Refactoring von Tests

Colin Kuskie hat einige Tests umgeschrieben, so dass das TAP nicht mehr händisch ausgegeben wird, sondern von test.pl. Diese Aufgabe war im *perltodo* Repository von Ricardo Signes aufgeführt: *https://github.com/rjbs/perltodo*.

Filter::Simple soll aus dem Kern entfernt werden

Dieser Vorschlag kam von James E Keenan und er traf auf breite Zustimmung.

Schlüsselwörter oder Funktionen für neue Features?

Yves Orton hat einige Ideen für neue Features. Bevor diese in den Kern fließen, möchte er wissen, ob die neuen Features über Subroutinen oder über Schlüsselwörter eingebracht werden sollen. Er ist der Meinung, dass Schlüsselwörter nur für Features verwendet werden sollten, die das Parsen des Quellcodes verändern (z.B. ein neuer Infix-Operator). Über Funktionen, die auch per Default geladen werden können wie z.B. `utf8::encode`, soll alles andere erledigt werden.

Benchmark-Programme gesucht

Nicholas Clark ist auf der Suche nach Benchmark-Programmen, die mehr als nur die allereinfachsten Operationen ausführen. Es sollen Programme sein, die aus dem typischen Arbeitsalltag kommen und ohne Abhängigkeiten funktionieren. Mit diesen Benchmark-Programmen möchte Clark neue Features wie lexikalische Subroutinen etc. testen.

Subroutinen-Signaturen

Peter Martini arbeitet schon einige Zeit an Subroutinen-Signaturen. Was er vorhat, hat er in einem Blogpost beschrieben: *http://blogs.perl.org/users/peter_martini/2012/09/subroutine-signatures---the-plan-v1.html* .

Unschöner Perl-Code als Zeichen für Optimierung

Als Aufhänger für diese Diskussion diente die Frage nach Benchmark-Programmen. David Golden ist der Meinung, dass die Stellen an denen man unschönen Perl-Code aus Performanz-Gründen schreibt, genau die Stellen sind, an denen man bei Perl Hand anlegen sollte. Einige Mitglieder haben zu einer Liste beigetragen, in der solche Codebeispiele gesammelt werden. Die Diskussion ist unter http://www.nntp.perl.org/group/perl.perl5.porters/2012/09/msg192748.html zu finden.

given/when - eine unendliche Geschichte

Schon einmal wurde für ein Release `given/when` in nicht-kompatibler Weise geändert. Aber auch die aktuelle Implementierung hat ihre Macken. Deshalb diskutieren die Perl 5 Porters, wie `given/when` endgültigt gefixt werden kann oder ob das Konstrukt wieder aus Perl verschwinden soll. Ricardo Signes, aktueller Pumpking hat folgende Tabelle gepostet:

```
$x ~~ undef
$x ~~ $overloaded_object
$x ~~ sub {}

$x ~~ regex
...or fail


...with when:

when ("foo") # str eq
when (12345) # num ==
when ($x)    # ~~
when { ... } # block evaluates true

when breaks the enclosing topicalizer
```

Mal sehen wie die Geschichte weitergeht.

:utf8 status

Der IO-Layer `:utf8` ist problematisch, weil er mehr als nur UTF-8 erlaubt. Leon Timmermans und Christian Hansen arbeiten daran, die IO-Layer zu fixen. Aktuell können die Arbeiten noch nicht abgeschlossen werden, weil noch zwei Bugs im Wege stehen. Der problematische Bug ist, dass `:stdio` plus einem anderen Layer hängt, es aber genügend Code gibt, der auf diesem Verhalten von `:stdio` aufsetzt und `:stdio` damit nicht entfernt werden kann.

Lexikalische Subroutinen

Father Chrysostomos hat in einem Branch seine Arbeiten an lexikalischen Subroutinen beendet. Wer interessiert ist und damit herumspielen will, kann sich den Branch unter *http://perl5.git.perl.org/perl.git/shortlog/refs/heads/smoke-me/lex-sub* anschauen.

Lexikalische Subroutinen können wie folgt verwendet werden:

```
{
    my sub bar { say "hoge" };

    bar();
}

# can't call bar() here; doesn't exist
```

Termine

November 2012

01. Treffen Dresden.pm
03./04. Frankfurter Perl-Community Workshop
06. Treffen Stuttgart.pm
Treffen Frankfurt.pm
09./10. Österreichischer Perl-Workshop
12. Treffen Ruhr.pm
14. Treffen Niederrhein.pm
19. Treffen Erlangen.pm
21. Treffen Darmstadt.pm
24. Londoner Perl-Workshop
27. Treffen Bielefeld.pm
28. Treffen Berlin.pm

Dezember 2012

04. Treffen Stuttgart.pm
Treffen Frankfurt.pm
07.-09. Quack and Hack Europe 2012
10. Treffen Ruhr.pm
12. Treffen Niederrhein.pm
17. Treffen Erlangen.pm
18. Treffen Bielefeld.pm
19. Treffen Darmstadt.pm
Treffen Berlin.pm

Januar 2013

01. Treffen Frankfurt.pm
Treffen Stuttgart.pm
03. Treffen Dresden.pm
09. Treffen Niederrhein.pm
14. Treffen Ruhr.pm
16. Treffen Darmstadt.pm
21. Trefen Erlangen.pm
29. Treffen Bielefeld.pm
30. 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.perlfoundation.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.

Perl-Services.de

Programmierung - Schulung - Perl-Magazin

info@perl-services.de