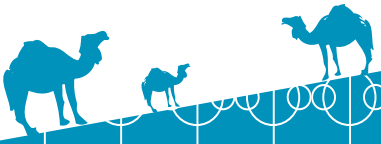
Ausgabe 3/2009 – www.foo-magazin.de - 6,00 €

$foo

PERL MAGAZIN



App::Asciio
ASCII-Art mit Perl

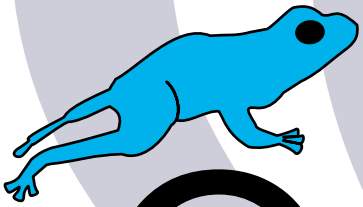
Time::y2038 - Das Jahr-2038-Problem
Keine Angst vor dem Schwarzen Dienstag 2038

Restricted Hashes
Zuverlässige Hashes erzeugen

Nr 11

Modern Perl

Perl @ FrOSCon



22.08.09

St. Augustin

Im Rahmen der 4. FrOSCon

VORWORT

Modern Perl

In letzter Zeit wird es immer mehr zum Thema: "Modern Perl". chromatic hat ein eigenes Blog zu dem Thema und hat das erste Bundle auf CPAN gestellt, das Features des modernen Perls aktiviert. Michael G Schwern geht mit seinem "perl5i" sogar noch einen Schritt weiter und verwendet noch mehr Module, die er zum "modernen Perl" zählt.

Auch ich beschäftige mich mehr und mehr damit, was eigentlich modernes Perl ist. Im Blog des O'Reilly-Verlags habe ich eine kleine Antwort darauf gegeben. Das erhebt aber nicht den Anspruch der Vollständigkeit. Es gibt sicherlich verschiedene Meinungen dazu, was man als "modernes Perl" bezeichnet, aber einige Punkte werden überall gleich sein.

Damit das auch der breiten Öffentlichkeit gezeigt wird, werde ich auf der diesjährigen FrOSCon mit ein paar Anderen einen Perl-Developer Raum haben, in dem wir verschiedene Vorträge halten wollen. Ein paar davon werden auch die "Evolution" der Perl-Programmierung aufzeigen oder zumindest andeuten.

Wie auch immer: Die Öffentlichkeit sollte erfahren, dass Perl-Programme heute oft anders aussehen als noch vor 10 Jahren.

Auch in dieser Ausgabe gibt es wieder einige Artikel zu Themen und Modulen, die oft zum "modernen Perl" gezählt werden: So zeigt Wolfgang Kinkeldei in seinem Artikel etwas zu seinen Best Practices beim Einsatz von Catalyst und auch DBIx::Class ist wieder Thema. Dass Perl auch für die Zukunft gerüstet ist, zeigt Thomas Fahle in seinem "HowTo"-Artikel.

Viel Spaß beim Lesen,

Renée Bäcker

Die Codebeispiele können mit dem Code

43ndj2

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/issue11.

IMPRESSUM

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

INHALTSVERZEICHNIS

ALLGEMEINES

PERL

MODULE

ANWENDUNGEN

TIPPS & TRICKS

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, als Organisator des kleinen Frankfurt Perl-Community Workshops und und als Grant-Manager bei der Perl Foundation.

Ferry Bolhár-Nordenkampf

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

Herbert Breunung

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

Thomas Fahle

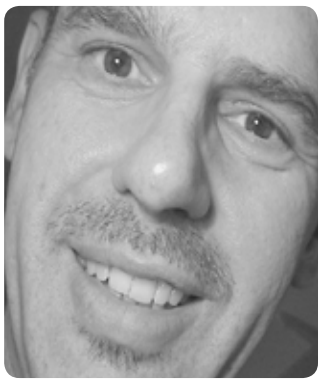
Thomas Fahle, Perl-Programmierer und Sysadmin seit 1996.
Websites:

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

Wolfgang Kinkeldei

Wolfgang Kinkeldei arbeitet als Software-Entwickler bei einem mittelständischen Mediendienstleister in Nürnberg. Zu seinen Hauptaufgaben zählen die Automatisierung von Arbeitsabläufen in der Druckvorstufe sowie die Erstellung von Web-basierten Läsungen. Die meisten seiner Projekte werden mit Perl gelöst.

Frank Seitz



Frank Seitz hat Informatik studiert und arbeitet als freiberuflicher Web-, Unix- und Datenbankentwickler. Er nutzt Perl seit Perl 4 und liebt diese Sprache für ihre Flexibilität, ihre Ausdrucksstärke und ihren Facettenreichtum. Seit Jahren lässt er seine Ideen in seine persönliche Perl-Klassenbibliothek einfließen, die eine wichtige Grundlage seiner Arbeit darstellt.

Vererbung in Perl

Im Unterschied zu Java besteht in Perl die Möglichkeit der Mehrfachvererbung. Von welchen Klassen eine Subklasse erbt, wird mit dem Array `@ISA` ("is a" ausgesprochen) festgelegt. Damit erbt die Subklasse alle Methoden der Superklasse.

```
package T1;
sub test{ print "test\n" };

package T2;
our @ISA = qw(T1);
sub new{ return bless {}, shift; }

package main;
T2->test;
```

Das einfache Beispiel, das die Vererbung demonstrieren soll, gibt einfach "test" aus, obwohl in der Klasse `T2` keine Methode `test` definiert hat. Hier sucht Perl automatisch nach eine Methode `test` in den Klassen, von denen `T2` erbt. Perl verwendet für diese Suche das Array `@ISA`. Für jede Klasse in der Vererbungshierarchie wird dieses Array nach den Regeln "Tiefensuche" und dann von "links nach rechts" durchsucht.

Vererbungsbaum

Mit dem `@ISA`-Array wird auch gleichzeitig ein Vererbungsbaum festgelegt, der durchsucht wird, wenn in der Klasse eine Methode nicht gefunden wird. Wenn eine Klasse von mehreren anderen Klassen erbt (Listing 1), kann auch eine Diamantstruktur wie in Abbildung 1 herauskommen.

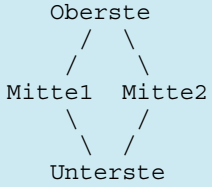


Abb.2: Diamantenstruktur

```
package Oberste;
sub hallo { print "Hallo \$foo-Leser\n" };

package Mitte1;
our @ISA = qw( S1 );

package Mitte2;
our @ISA = qw( S1 );
sub hallo { print "Die Mitte2 gruesst\n" };

package Unterste;
our @ISA = qw( Mitte1 Mitte2 );

package main;
Unterste->hallo();
```

Listing 1

An dieser Struktur lässt sich auch gut erklären, wie Perl das Array `@ISA` durchsucht. Wenn für ein Objekt der Klasse `Unterste` eine Methode aufgerufen wird, und diese in der Klasse nicht gefunden wird, dann schaut Perl erst in `Mitte1` (weil es nach oben geht (Tiefensuche) und "ganz links" in `@ISA` in `Unterste` steht), dann in `Oberste` weil es von `Mitte1` "nach oben" geht. Wenn dann die Methode noch nicht gefunden wurde, wird die Klasse `Mitte2` durchsucht.

Mit einer gepatchten Version des Tk-Moduls `Tk::PerlInheritanceTree` – das hoffentlich bald auf CPAN erscheint – lässt sich dieser Vererbungsbaum auch schön darstellen (siehe Listing 2).

Der Vererbungsbaum (siehe Abbildung 2) kann sogar während der Laufzeit verändern. Man kann Superklassen für eine Klasse hinzufügen oder einfernen (siehe Listing 3).

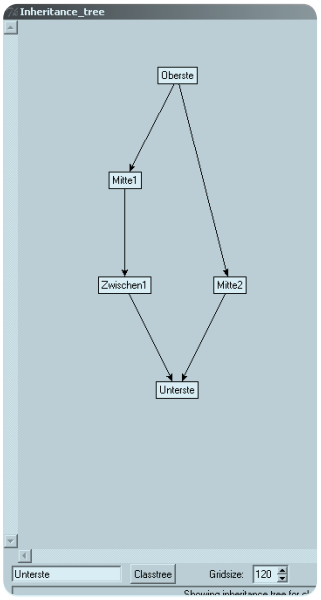


Abb. 2: Vererbungsbaum

```
#!/usr/bin/perl

use strict;
use warnings;

use lib qw(Kurs);

use Tk;
use Tk::PerlInheritanceTree;

my $mw = tkinit;
my $tree = $mw->PerlInheritanceTree(
                  -maxdepth => 10 )->pack;

$tree->classname( 'Unterste' );

MainLoop;
```

Listing 2

Hier wird einfach ein leeres Package erzeugt, für das die Vererbung während der Laufzeit verändert werden soll. Der Aufruf von `header` wird in ein Block-`eval` gepackt, damit das Programm nicht bei einem Fehler gleich abbricht.

Nach dem ersten `eval` wird das Modul `CGI` geladen und das `@ISA`-Array des Packages `OOPerl` wird verändert. Danach wird der gleiche Aufruf gemacht. Die Ausgabe sieht dann so aus:

```
C:\Kurs>perl vererbung.pl
1: Can't locate object method "header" \
   via package "OOPerl" \
   at vererbung.pl line 12.
2: Content-Type: text/html
```

Wie man sieht, findet Perl beim zweiten Aufruf von `header` die Methode.

Die Suche von einem anderen Punkt aus starten

Nehmen wir an, dass zwischen der Klasse `Unterste` und `Mitte1` noch ein paar Klassen in der Vererbungshierarchie sind, dann möchte man vielleicht nicht, dass das komplette Array `@ISA` durchsucht wird, sondern direkt bei `Mitte1` gestartet wird. In diesem Fall hat man folgende Möglichkeit:

```
package Unterste;

# Zwischen1 ist eine Subklasse von Mitte1
our @ISA = qw(Zwischen1);

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

    $class->Mitte1::test();
}
```

In diesem Fall wird nicht erst `Zwischen1` nach der test-Methode durchsucht, sondern gleich `Mitte1`.

```
#!/usr/bin/perl

{
    package OOPerl;
}

eval{ print "1: ",OOPerl->header; }
       or print "1: $@";

require CGI;
@OOPerl::ISA = qw'CGI';

eval{ print "2: ",OOPerl->header; }
       or print "2: $@";
```

Listing 3

Fallen bei der Mehrfachvererbung

Die Mehrfachvererbung ist aber auch Quelle einer ganz gemeinen Falle, auf die Curtis 'Ovid' Poe in seinem use.perl.org-Journal hinweist: Je nach Reihenfolge der Superklassen im `@ISA`-Array sind manche Methoden nie erreichbar. Z.B. bei diesem Code:

```
{
    package Platypus;
    our @ISA = qw<SpareParts Duck>;
    package Duck;
    our @ISA = 'SpareParts';
    sub quack {}
    package SpareParts;
    our @ISA = 'Animal';
    sub quack {}
    package Animal;
}
```

ergibt sich dieser Vererbungsbaum

```
        Animal
          |
     SpareParts
      |     |
      |   Duck
      |     |
     Platypus
```

Durch den Suchalgorithmus, der oben beschrieben wurde, wird die Methode `quack` aus der Klasse `Duck` nie gefunden. Um auf diese unerreichbaren Methoden aufmerksam zu werden, kann man das Modul `Class::Sniff` verwenden.

```
my $platypus = Class::Sniff->new({
    class => 'Platypus',
});
explain $platypus->unreachable;
```

Die Ausgabe zeigt an, dass genau diese Falle zugeschnappt hat.

Renée Bäcker

Perl Scopes Tutorial - Teil 1

Dieser Bericht beschäftigt sich mit *Scopes*, d.h. mit Bereichen innerhalb eines Programms, die sich durch Sichtbarkeit und Gültigkeitsdauer der in ihnen enthaltenen Objekte unterscheiden.

Scopes

Scopes definieren Sichtbarkeits- und Gültigkeitsbereiche in einem Programm; sie stellen eine bestimmte Umgebung für Code bereit. Unter *Scoping* versteht man das Anlegen und Verwalten solcher Umgebungen, entweder bereits beim Kompilieren eines Skripts (*statisches Scoping*), oder zur Laufzeit (*dynamisches Scoping*).

Perl stellt – wie denn auch nicht – beide Varianten des Scopings zur Verfügung; sie können sogar innerhalb eines Programms abwechselnd verwendet werden. Genau daraus entstehen aber oft Irrtümer, wenn die Auswirkungen der beiden Scoping-Arten miteinander verwechselt werden. Dieser Beitrag soll hier Klarheit schaffen.

Da Perl keine lokalen Funktionen kennt [1], existieren Scopes dort hauptsächlich, um den Geltungsbereich von **Variablen und Werten** festzulegen bzw. einzuschränken. Beim Betrachten von Scopes wird es in diesem Bericht daher vorwiegend um die Ansprechbarkeit und Lebensdauer von Variablen gehen. Desweiteren sind Scopes beim Kompilieren (Parsen) von Code wichtig, weil das Verhalten des Compilers für jeden Scope individuell festgelegt werden kann.

Das klassische Beispiel für die Verwendung von Scopes sind Funktionen – wird mit

```
sub test {
  ...
}
```

eine Funktion definiert, so wird für sie ein Scope eingerichtet, der lexikalisch durch die beiden geschwungenen Klammern begrenzt ist. Wird eine lokale Variable innerhalb dieses Scopes deklariert, so kann sie auch nur darin (von Code im Funktionskörper) angesprochen werden. Dadurch lassen sich Variable, die nur innerhalb der Funktion benötigt werden, sicher benutzen, weil ein unbeabsichtigtes Überschreiben einer außerhalb der Funktion verwendeten gleichnamigen Variable nicht mehr möglich ist..

Scopes werden in Perl durch **Blöcke**, d.h. durch in geschwungene Klammern { ... } eingefasste Codeteile repräsentiert. Meistens ist der Umfang eines Blocks und der Geltungsbereich des zugehörigen Scopes identisch, doch gibt es hierzu vor allem in Schleifenkonstrukten Ausnahmen (siehe dazu auch perlsyn). In Perl repräsentiert ein Block eine *Compilation Unit*, d.h. das kleinste Stück Code, das vom Compiler als Einheit betrachtet wird.

Die folgenden Anweisungen gestatten oder erfordern die Verwendung von Blöcken und damit die Definition von Scopes:

• `for`, `foreach` (als Befehl, nicht als Modifier [2]). Der Schleifenkopf zählt zum Scope der Schleife, eine darin deklarierte Variable, z.B.

```
for (my $i = 0; $i < 10; $i++)
{
  print $i; # $i im Schleifenkopf deklariert
}
```

kann im durch den Block begrenzten Schleifenkörper angesprochen werden. Das gilt sinngemäß auch für einen `continue` Block, falls vorhanden. Im Zusammenhang mit `foreach` gibt es darüber hinaus eine Besonderheit, auf die in der nächsten Ausgabe des $foo-Magazins näher eingegangen wird.

• `while`, `until` (als Befehl, nicht als Modifier). Der Schleifenkopf zählt zum Scope der Schleife, eine darin deklarierte Variable, z.B.

```
while (my $input = <STDIN>)
{
  # $input im Schleifenkopf deklariert
  print "Eingabe: $input\n";
}
```

kann im durch den Block begrenzten Schleifenkörper angesprochen werden. Das gilt sinngemäß auch für einen `continue` Block, falls vorhanden.

• `if`, `unless` (als Befehl, nicht als Modifier)
• `else`, `elsif`
• `eval`, auch String-`eval`s oder `eval`s des `(?{...})` Konstrukts in regulären Ausdrücken sowie in Substitutionsanweisungen mit dem `e` Modifier:

```
eval "$code";
m/(?{$code})/;
s/Regexp/$code/e;
```

In allen Fällen wird für die Ausführung von `$code` ein temporärer Scope eingerichtet.

• `do` (mit Codeblock)
• `sort`, `map`, `grep` (wenn als erstes Argument ein Codeblock angegeben ist)
• `BEGIN`, `CHECK`, `UNITCHECK`, `INIT`, `END` Blöcke
• Inline-Blöcke, wie z.B.

```
my $y = 2;
print ${my $y = 1; \$y}, $y;
# Gibt 1 2 aus
```

Der Code innerhalb der geschwungenen Klammern, der (hier) eine Skalarreferenz zurückliefert (die dann dereferenziert wird), wird in einem eigenen Scope ausgeführt.

• Jede Funktion (benannt oder anonym)

Darüber hinaus kann man auch jeden anderen Code in einen eigenen Scope einbetten, wenn man ihn als Block schreibt. Beachte, dass ein solcher Block eine Schleife, die genau einmal ausgeführt wird, darstellt. Daher sind darin auch Anweisungen wie `last` oder `redo` erlaubt.

Aber auch jede Skriptdatei selbst repräsentiert einen – den **äußersten** – Scope [3], der solange aktiv ist, bis ein anderer eingerichtet wird. Daraus folgt, dass die Wirkungsweise von Direktiven und Anweisungen, die für Scoping verwendet werden, *über Dateigrenzen nicht hinausreicht*.

Weiters folgt, dass Scopes *verschachtelt* werden können, d.h. ein Scope kann weitere Scopes enthalten, die ihrerseits Scopes enthalten können. Die Verschachtelungstiefe ist nur durch den vorhandenen Rechnerspeicher begrenzt. Ein innerer Scope erbt zunächst die Umgebung des ihn umgebenden Scopes. Diese kann aber auf die Bedürfnisse des Codes, der im Scope ausgeführt wird, angepasst werden. Enthält der Scope weitere Scopes (*Subscopes*), so wird die angepasste Umgebung an diese weitergereicht, doch sind auch in den Subscopes Änderungen möglich. Jeder Scope übernimmt die Umgebung des äußeren Scopes und gibt sie – bei Bedarf entsprechend angepasst – an seine Subscopes weiter.

Schließlich folgt daraus, dass Variable, die in einer Datei als lokal (lexikalisch) deklariert werden, aus anderen Dateien heraus nicht ansprechbar sind (obwohl sie dennoch weiter existieren können) – ein häufig durchaus erwünschter Effekt.

Statisches Scoping

Beim *statischen Scoping* wird die Umgebung durch *Direktiven* festgelegt und vom Compiler eingerichtet. Eine Änderung dieser Umgebung zur Laufzeit ist nicht mehr möglich. Die durch eine Direktive eingerichtete Umgebung kann textmäßig (*lexikalisch*) nur von Code **unterhalb** dieser Direktive angesprochen werden. Die Wirkungsweise von statischem Scoping ist leicht nachvollziehbar, da es mit Laufzeitverhalten ("was wird wann ausgeführt?") nichts zu tun hat. Man spricht daher auch oft von *lexikalischem Scoping*.

Um dessen Wirkungsweise zu verstehen, ist es hilfreich, sich vorzustellen, was beim Kompilieren von Code (also bevor das Skript losläuft) passiert. Der Compiler liest ein Skript Zeile für Zeile, von oben nach unten ein. Stößt er dabei auf eine *Direktive*, d.h. auf eine Anweisung, die sein Verhalten steuert [4], wird diese Direktive sofort verarbeitet. Handelt es sich dabei um eine Direktive für das Scoping, wirkt sich diese immer auf den *gerade aktuellen* Scope aus:

```
{
  my $x = 1;
  print $x; # lokales (lexikalisches) $x
}
print $x; # globales $x
```

Hier laufen folgende Vorgänge ab:

• Es wird die öffnende, geschwungene Klammer entdeckt und daraufhin ein neuer Scope mit einem leeren *Scratchpad* eingerichtet. Scratchpads sind scope-eigene Speicherbereiche und werden später noch genauer beschrieben.

• Es wird die `my` Direktive verarbeitet und im Scratchpad des gerade neu eingerichteten Scopes für die Variable `$x` ein Eintrag angelegt. Die Zuweisung des Wertes an die Variable passiert hier noch nicht; sie findet erst zur Laufzeit statt.

• In der `print`-Anweisung wird neuerlich `$x` angesprochen. Der Compiler sieht im Scratchpad des aktuellen Scopes nach. Da es dort einen Eintrag mit diesem Namen gibt, generiert er für die `print`-Anweisung den Code für den Zugriff auf das lexikalische `$x`.

• Es wird die schließende geschwungene Klammer entdeckt. Der aktuelle Scope ist damit beendet und es wird wieder der vorige Scope (Dateiscope) verwendet.

• Der Compiler stößt neuerlich auf eine Variable `$x` und sieht im Scratchpad des nun aktuellen Scopes nach. Da es darin noch keine solche Variable gibt, wird `$x` jetzt als global angesehen und der entsprechende Code zum Zugriff auf eine globale Variable generiert.

Daher wird jede Variable als global angesehen, wenn sie nicht vorher ausdrücklich anders deklariert wurde. Die Quintessenz daraus: nur Code, der in einem Scope textmäßig (*lexikalisch*) **unterhalb** einer Deklaration – bis zum Blockende (oder Dateiende) – aufscheint, kann sich auf diese Deklaration beziehen.

Wenngleich hier offensichtlich, gibt es Fälle, bei denen die Wirkungsweise statischen Scopings nicht ganz so klar zu Tage tritt. In den folgenden Abschnitten werden hierfür Beispiele gezeigt.

Die Direktiven, die für statisches Scoping verwendet werden, sind `package`, `my`, `state` und `our`. Erstere legt einen Namensraum fest, der für *globale* Variable verwendet wird, wenn deren Namen unqualifiziert (ohne explizite Angabe eines Namensraumes) aufscheinen. Die anderen Direktiven dienen zur *Deklaration* von Variablen und deren Benutzung. Alle diese Direktiven werden im Folgenden noch genau beschrieben.

Dynamisches Scoping

Dynamisches Scoping wird durch Anweisungen zur Laufzeit veranlasst. Je nachdem, ob eine solche Anweisung ausgeführt wird oder nicht, kann derselbe Code von mal zu mal in einer anderen Umgebung ablaufen. Die Umgebung ist somit vom Lauf des jeweiligen Programmes abhängig und daher lässt sich nicht immer vorherbestimmen, wie sie zu einem bestimmten Zeitpunkt aussehen wird.

Dynamisches Scoping wird häufig verwendet, wenn man die Umgebung, in der Code abläuft, *vorübergehend ändern* möchte. Indem man die gewünschten Änderungen innerhalb eines eigenen Scopes durchführt und den davon betroffenen Code in diesen Scope packt, stellt man sicher, dass der Code in der angepassten Umgebung abläuft, aber danach – d.h. nach Verlassen des Scopes – wieder genau die Umgebung existiert, die vor dem Betreten des Scopes vorhanden war.

Oft sind es Perl's spezielle Variable, deren Werte man auf diese Art *temporär* ändert, um eine bestimmte, durch den Wert kontrollierte Funktionalität für einen begrenzten Bereich wirksam werden zu lassen. Ein recht bekanntes Beispiel dafür ist:

```
{
  # $/ enthaelt jetzt "undef" (keinen Wert)
  local $/;
  # Lies gesamtes <INPUT> in $data
  $data = <INPUT>;
 } # $/ enthaelt wieder vorigen Wert
```

Hier wird in einem eigenen Scope die Variable, die das Satztrennzeichen beim Einlesen (*Input Record Separator*) enthält, auf `undef` gesetzt, d.h gelöscht. Bei der folgenden Leseoperation wird daher der Inhalt der gesamten Datei auf einmal in die Variable `$data` eingelesen (*slurp mode*).

Ohne dynamisches Scoping und die `local`-Anweisung müsste man den ursprünglichen Wert von `$/` vor der Änderung selber sichern und nach dem Einlesen wieder herstellen, da die durch die Änderung eingetretene Wirkung sonst auch bei allen weiteren Leseoperationen erhalten bliebe. Das wäre ziemlich umständlich und fehleranfällig. Durch das Verwenden von dynamischem Scoping hingegen ist sichergestellt, dass alle auf diese Art geänderten Variable nach dem Verlassen des Blocks automatisch wieder auf den vorigen Wert gesetzt werden.

Beachte, dass es hierbei um das vorübergehende Austauschen *eines Wertes* **einer bereits vorhandenen Variable** *zur Laufzeit* geht, im Unterschied zu den beim statischen Scoping verwendeten Direktiven, die meist **das Anlegen neuer Variable** *beim Kompilieren* zur Folge haben.

Wie in obigem Beispiel bereits gezeigt wurde, wird für das dynamische Scoping der Befehl `local` verwendet, der in Ausgabe 13 noch detailliert beschrieben wird.

Variable

Perl kennt zwei Arten von Variablen: globale und lokale. Beide Arten haben miteinander nichts zu tun und sind auch völlig unterschiedlich implementiert. Die folgenden Abschnitte beschreiben beide Arten, wobei den lokalen Variablen, da hauptsächlich sie mit Scoping in Zusammenhang stehen, naturgemäß wesentlich mehr Augenmerk gewidmet wird.

Globale Variable

Globale Variable gibt es, seit es Perl gibt – wann immer eine Variable ohne vorherige Deklaration verwendet wird, wird sie als global eingerichtet. "Global" bedeutet, dass die Variable von überall sichtbar ist, sogar von Code aus anderen Dateien. Umgekehrt kann man auch auf globale Variable aus hinzugeladenem Code zugreifen – einige ältere Module, vor allem solche, die nicht objektorientiert arbeiten, verwenden diesen Mechanismus ganz bewusst zur Konfiguration: durch das Setzen oder Löschen solcher Variable wird die Arbeitsweise des Moduls beeinflusst.

Globale Variable gehören immer einem Namensraum (*Package*) an, daher spricht man auch von *Paketvariablen*. Der Name des Packages kann dem Variablenamen vorangestellt und von diesem durch zwei Doppelpunkte getrennt werden; dadurch ist die Variable eindeutig bezeichnet. Wird der Paketname weggelassen, wird stattdessen der Name des gerade eingestellten Packages verwendet. Dieser wiederum wird durch die `package` Direktive festgelegt; wurde eine solche nicht angegeben, wird der Namensraum `main` verwendet.

Einige Variablenamen werden *immer* im Package `main` angenommen, auch wenn mittels `package` gerade ein anderer Namensraum ausgewählt ist: dazu zählen alle Interpunktionsvariable (also solche, deren Namen keine alphanumerischen Zeichen enthalten oder damit beginnen), sowie die speziellen *Namen* `ARGV`, `ARGVOUT`, `ENV`, `INC`, `SIG`, `STDERR`, `STDIN` und `STDOUT`.

Globale Variable werden in Symboltabellen (*Symbol Tables*) gesammelt und sind mittels Stashes (*Symbol Table Hashes*) und Typeglobs (jene Konstrukte, deren Namen mit einem `*` beginnen) implementiert. Der Artikel "Typeglobs" ($foo Ausgaben 7 bis 9) beschreibt diese Implementierung detailliert.

Für Scoping haben diese Variable, da sie global sind, keine Bedeutung. Allerdings kann man seit Perl 5.6 mittels der `our` Direktive auch die Benutzung globaler Variable innerhalb von Scopes explizit deklarieren. Darauf wird in der Beschreibung von `our` eingegangen.

Die "package" Direktive

Die `package` Direktive veranlasst den Compiler, vom Zeitpunkt des Auffindens an alle Variable, die ohne expliziten Packagenamen aufscheinen und nicht ausdrücklich als lokal (lexikalisch) deklariert wurden, in den als Parameter angegebenen Namensraum zu stellen. Da es sich um eine Compilerdirektive handelt, kann der Name nur als *Bareword* (d.h. als Stringliteral ohne Anführungszeichen) angegeben werden.

`package` wird vor allem von Modulen (zuladbarer Code) verwendet. Der Name des verwendeten Packages ist dabei identisch mit dem Modulnamen. Da jedes Modul einen anderen Namen hat, ist sichergestellt, dass immer ein eigener Namensraum verwendet wird und somit nie zwei globale Variable aus unterschiedlichen Modulen gleich benannt sind, selbst wenn zweimal derselbe Bezeichner verwendet werden sollte.

Die Wirkung von `package` ist auf Scopegrenzen beschränkt – eine derartige Deklaration hat daher nur innerhalb des Scopes, in dem sie vorkommt (typischerweise der Dateiscope), Gültigkeit. In den meisten Moduldateien scheint daher die `package` Direktive als erste Anweisung auf und bleibt bis zum Dateiende (oder einem anderen `package`) gültig. Danach wird wieder der vorherige Scope (und damit auch der vorher eingestellte Namensraum) aktiv. Weitere globale Variable werden daher wieder in diesem Namensraum, und nicht mehr im Namensraum des Moduls gesucht bzw. angelegt.

Mehr über Module, Packages und die `package` Direktive findet man in perlmod. An dieser Stelle soll nur noch einmal festgehalten werden, dass auch `package` an Scopegrenzen gebunden ist (oder anders gesagt, **dass auch der voreingestellte Namensraum globaler Variable Teil der durch Scopes bereitgestellten Umgebung ist**).

Der zweite Teil des Berichtes wird lexikalische Variable, ihren Einsatz in Schleifen und in Funktionen behandeln.

Ferry Bolhár-Nordenkampf

[1] Zur Zeit werden noch keine Funktionen mit eingeschränktem Geltungsbereich (*lexikalische Funktionen*) unterstützt, obwohl der Parser seit Version 5.6 in der Lage ist, Code wie

```
my sub test {
  ...
}
```

sinngemäß zu interpretieren. Allerdings wird die Verarbeitung mit der Meldung

```
"my sub" not yet implemented
```

abgebrochen. Das "not yet" im Meldungstext lässt hoffen, dass dies eines Tages doch möglich sein wird.

Es gibt jedoch ein CPAN-Modul, das mit Hilfe von Source-Filtern eine ähnliche Möglichkeit bereits jetzt zur Verfügung stellt (`Sub::Lexical`).

[2] Unter *Modifier* versteht man die Verwendung des jeweiligen Befehls als Ergänzung zu einem anderen, wie z.B. in diesen Konstrukten:

```
print "$_\n" foreach @INC;

print "$input\n" while $input = <STDIN>;
```

Diese Konstrukte stellen zwar auch Schleifen dar, aber die Schleifen besitzen keinen eigenen Scope.

Das gilt sinngemäß auch für `if` und `unless`, wenn diese Befehle als Modifier verwendet werden.

[3] Es kann helfen, sich den Inhalt eines Perl-Skripts (einer Datei) von { ... } umschlossen vorzustellen.

[4] Die derzeit vorhandenen Compiler-Direktiven sind

```
my    our    state    package
use   no     format   sub
$[
```

wobei `my`, `our`, `state` und `package` einen scope-weiten Geltungsbereich besitzen, `sub` und `format` hingegen einen globalen. Bei `use` richtet sich der Geltungsbereich nach dem Pragma oder Modul, das eingebunden wird; bei Pragmas, die z.B. die Hint-Variablen (`$^H` bzw. `%^H`) beeinflussen (`strict`, `integer`, siehe auch Abschnitt über lexikalische Pragmas), ist der Bereich ebenfalls auf Scopes begrenzt, bei solchen, die z.B. auf Symboltabellen Einfluss nehmen (`vars`, `subs`), ist er global. `no` wird ebenfalls meistens mit lexikalischen Pragmas verwendet, um deren Wirkung temporär umzukehren oder zu ändern. In der Beschreibung des Pragmas oder Moduls ist sein Geltungsbereich, sofern dieser eine Rolle spielt, jeweils angeführt.

Erwähnenswert ist vielleicht die letzte "Direktive" – hinter der scheinbaren Zuweisung zur Laufzeit

```
$[ = 1;
```

steckt in Wirklichkeit die Aufforderung an den Compiler, den Index für das erste Element eines Arrays bzw. das erste Zeichen in einem Substring ab nun mit dem angegebenen Wert (statt mit 0) zu nummerieren. Daher darf auch nur eine numerische Konstante, die der Compiler verarbeiten kann, zugewiesen werden, jede andere Zuweisung an `$[` führt zur Ausgabe der

```
That use of $[ is unsupported
```

Meldung. Der Grund für diese ungewöhnliche Direktive liegt in der Rückwärtskompatibilität zu Perl 4, wo Zuweisungen an `$[` tatsächlich erst zur Laufzeit ausgeführt wurden. Da man von der Möglichkeit, den Basisindex von Arrays bzw. die Zeichennummerierung in Strings zu ändern, wenig Gebrauch macht, hat man auf das Schreiben eines eigenen Pragmas hierfür bisher verzichtet.

Beachte, dass der Compiler `$[` als *lexikalisches Pragma* betrachtet, d.h., Änderungen an dieser Variable sind an Scopegrenzen gebunden:

```
{                   # Neuer Scope
  $[ = 1;           # Array-Basis jetzt 1
     $a[1] = 'Hello';
}                   # Endes des Scopes
print $a[0];        # Gibt "Hello" aus
```

Zuverlässige Objekte durch Restricted Hashes

Perl ist - unter anderem - eine sehr leistungsfähige und flexible Objektorientierte Programmiersprache. Eine Klasse zu definieren ist nicht schwer, aber tausend Wege führen nach Rom. Und es gibt Detail-Aspekte, die auch einem erfahrenen Perl-Programmierer einiges Nachdenken abfordern.

Wir wollen in diesem Artikel Klassen betrachten, die einen Hash als fundamentale Datenstruktur benutzen. Das ist sicherlich die am häufigsten anzutreffende Objekt-Struktur. Ein Hash speichert bekanntlich Attribut/Wert-Paare und das ist genau das, was wir meistens für unsere Objekte brauchen.

Eine einfache Basisklasse

Da wir unsere Klassen nicht immer "from scratch" schreiben wollen, definieren wir uns zunächst eine Basisklasse, die die grundlegende Funktionalität unserer Hash-basierten Klassen bereitstellt. Wir nennen die Klasse wie die Datenstruktur, die sie repräsentiert: "Hash".

```
package Hash;

sub new {
    my $class = shift;

    my $self = bless {},$class;
    $self->set(@_);

    return $self;
}

sub set {
    my $self = shift;

    while (@_) {
        my $key = shift;
        $self->{$key} = shift;
    }

    return;
}
```

Listing 1

Wir geben ihr zunächst einen Konstruktor `new()`, mit dem wir ein Objekt erzeugen und gleichzeitig seine Attribute setzen können. Das Setzen der Attribute delegiert der Konstruktor an die Methode `set()` (siehe Listing 1).

Eine einfache Anwendungs-Klasse

Mittels der Klasse "Hash" definieren wir uns durch Ableitung nun unsere Anwendungs-Klassen mit den dazugehörigen Attributen. Zum Beispiel eine Klasse "Person" mit den drei Attributen "vorname", "nachname" und "geschlecht" und der speziellen Methode `anrede()`, die eine Briefanrede für die betreffende Person generiert:

```
package Person;
use base qw/Hash/;

sub anrede {
    my $self = shift;

    if ($self->{'geschlecht'} eq 'w') {
        return "Sehr geehrte Frau
                $self ->{'nachname'}";
    }
    return "Sehr geehrter Herr
            $self->{'nachname'}";
  }
```

In unserem Programm nutzen wir die Klasse wie folgt:

```
my $per = Person->new(
    vorname=>'Elli',
    nachname=>'Pirelli',
    geschlecht=>'w',
);

print $per->anrede,"\n";

__END__
Sehr geehrte Frau Pirelli
```

Das ist ein geradliniger Ansatz. Er hat aber eine große Schwäche: Es ist nirgends verbindlich festgelegt, welche Attribute die Anwendungs-Klasse "Person" besitzt.

Da der dem Objekt zugrunde liegende Hash *jeden* Attributnamen akzeptiert, sind sowohl die Entwickler als auch die Nutzer der Klasse "Person" vor Schreibfehlern und Irrtümern nicht geschützt:

• Die Klasse kann *intern* versehentlich unkorrekte Attributnamen verwenden - z.B. beim internen Hash-Zugriff, der in der Methode `anrede()` vorkommt. Solche Fehler können leicht beim Refactoring passieren oder schlicht durch Typos.

• Einem Nutzer der Klasse Person kann *extern* das gleiche passieren - z.B. beim Konstruktor-Aufruf.

Derartige Fehler werden durch den Code nicht erkannt. Sie machen sich irgendwann im Programmverlauf in falschem Objektverhalten bemerkbar und sind in größeren Programmen unter Umständen schwer zu finden:

```
my $per = Person->new(
    vorname=>'Elli',
    nachname=>'Pirelli',
    geschlcht=>'w',
);

print $per->anrede,"\n";

__END__
Use of uninitialized value in string eq at
  ./test.pl line 34.
Sehr geehrter Herr Pirelli
```

Was kann man tun?

Die Situation ist vergleichbar mit der Situation eines Perl-Programms, bei dem auf `use strict` verzichtet wurde. Der Perl-Interpreter unterlässt es dann bekanntlich, im Quelltext auftretende Variablen darauf hin zu überprüfen, ob sie zuvor deklariert wurden. Auf diese Prüfung zu verzichten, wird allgemein als grundlegender methodischer Fehler angesehen.

Erfreulicherweise gibt es so etwas Ähnliches wie `use strict` auch für Hashes. Dieses Feature erlaubt es uns, den Hash-Zugriff einer lückenlosen Kontrolle zu unterwerfen, ohne die Kontrolle selbst ausprogrammieren zu müssen. Das Konzept nennt sich "Restricted Hash". Dieses können wir uns bei der OO-Programmierung zunutze machen.

Restricted Hashes

Ein Restricted Hash ist ein Hash, dessen Schlüssel und/oder Werte eingeschränkt manipuliert werden dürfen. Die Beschränkungen werden für den einzelnen Hash durch Funktionsaufrufe - mit dem Hash als Argument - festgelegt, nicht etwa durch Deklaration eines Hash-Typs, was wesentlich unflexibler wäre. Die Einhaltung der Beschränkungen wird bei Zugriff auf den Hash von Perl überprüft. Tritt eine Verletzung auf, wird eine Exception generiert, d.h. das Programm wird mit einer Fehlermeldung abgebrochen.

Die Funktionalität der Restricted Hashes ist seit Perl 5.8.0 in dem Core-Modul `Hash::Util` enthalten, d.h. sie ist bei jedem neueren Perl von Hause aus vorhanden.

Folgende Möglichkeiten werden von Hash::Util geboten, Hash-Zugriffe zu beschränken:

• Die Menge der zulässigen Schlüssel kann festgelegt werden (Funktionen: `lock_keys`, `lock_keys_plus`). D.h. weitere Schlüssel können zu dem Hash nicht hinzugefügt werden. Zulässige Schlüssel müssen im Hash aber nicht vorhanden sein, können also durchaus gelöscht werden.

• Ein einzelner Wert des Hashs kann eingefroren werden (`lock_value`). Der entsprechende Wert ist danach nicht mehr änderbar.

• Der Hash insgesamt kann eingefroren werden (`lock_hash`). Anschließend kann kein Wert verändert und kein Schlüssel hinzugefügt oder gelöscht werden. Dies geht auch rekursiv (`lock_hash_recurse`).

• Alle Beschränkungen können einzeln oder insgesamt wieder aufgehoben werden (`unlock_keys`, `unlock_value`, `unlock_hash`, `unlock_hash_recurse`).

• Es kann geprüft werden, ob ein Hash ein normaler, also ein "Unrestricted Hash" ist (`hash_unlocked`).

• Es kann die Liste der zulässigen Schlüssel und die der aktuell nicht im Hash vorhandenen zulässigen Schlüssel abgefragt werden (`legal_keys`, `hidden_keys`).

Attributzugriff beschränken

Die gesamten Möglichkeiten der Restricted Hash Programmierschnittstelle in `Hash::Util` benötigen wir nicht, um den Attribut-Zugriff bei Hash-Objekten sicher zu machen. Uns reicht es, die Menge der erlaubten Hash-Schlüssel in Stein zu meißeln. Hierzu erweitern wir unsere Basisklasse um die Methode `lockKeys()`, die die aktuell vorhandenen Objektattribute als die einzig zulässigen festschreibt:

```
package Hash;

use Hash::Util;

sub lockKeys {
    my $self = shift;
    Hash::Util::lock_keys(%$self);
    return;
}
```

Warum schreiben wir den Aufruf von `lock_keys()` mit Package-Präfix, also `Hash::Util::lock_keys`, statt `lock_keys()` zu importieren und den Package-Präfix wegzulassen? Der Grund ist: Bei der Implementierung von Klassen sollte man auf den Import von Subroutinen grundsätzlich verzichten, da dies den Namespace der Klasse verunreinigt. Die Subroutine `lock_keys()` ist keine Methode der Klasse "Hash" und hat daher in deren Namespace nichts zu suchen. Hash::Util verhält sich vorbildlich und exportiert per Default keine Bezeichner.

Jetzt stellt sich noch die Frage, an welcher Stelle wir dem Objekt-Hash die zulässigen Schlüssel, sprich: Attribute, verpassen, bevor wir sie festschreiben. Dies geschieht natürlich im Konstruktor, den wir nun für unsere Anwendungs-Klasse definieren müssen:

```
package Person;
use base qw/Hash/;

sub new {
    my $class = shift;

    my $self = $class->SUPER::new;
    $self->set(
        vorname=>undef,
        nachname=>undef,
        geschlecht=>undef,
    );
    $self->lockKeys;
    $self->set(@_);

    return $self;
}
```

Listing 2

Innerhalb des Konstruktors passiert folgendes (siehe Listing 2):

1. Zunächst instantiieren wir das Objekt durch Aufruf des Basisklassen-Konstruktors.
2. Dann fügen wir die Klassen-spezifischen Attribute hinzu. An dieser Stelle könnten wir Defaultwerte für die Attribute vorgeben. Da wir bei unserer Beispielklasse keine sinnvollen Defaultwerte haben, initialisieren wir hier alle Attribute auf `undef`.
3. Wir locken die Schlüssel des Objekt-Hashs. Von nun an können keine neuen Schlüssel hinzugefügt werden.
4. Wir setzen die beim Konstruktoraufruf angegebenen Attribute mit `set()`. Alle Attributnamen müssen nun korrekt sein, sonst knallt es!
5. Wir liefern die Objektreferenz zurück.

Was haben wir erreicht?

Durch die Initialisierung des Objekt-Hash im Konstruktor und den anschließenden Aufruf von `lockKeys()` haben wir die beruhigende Sicherheit erlangt, dass im gesamten Programm kein fehlerhafter Lese- oder Schreibzugriff auf den Objekt-Hash stattfinden kann - weder von innerhalb noch von außerhalb der betreffenden Klasse - ohne dass wir es bemerken.

Der fehlerhafte Code von oben (fehlerhaft wegen des falschgeschriebenen "geschlecht"):

```
$per = Person->new(
    vorname=>'Elli',
    nachname=>'Pirelli',
    geschlcht=>'w',
);

print $per->anrede,"\n";
```

führt unverändert in der Methode `set()` der Basisklasse zu dem unzulässigen Schreibzugriff:

```
   $self->{'geschlcht'} = 'w'
```

Jetzt bricht das Programm jedoch augenblicklich mit der Exception `Attempt to access disallowed key 'geschlcht' in a restricted hash at ./test.pl line 24.` ab. D.h. der Programmierfehler schwelt nicht länger dahin, sondern wird explizit sichtbar. Bei einem fehlerhaften Lesezugriff passiert genau dasselbe.

Einen wichtigen Unterschied zur Variablen-Überprüfung durch `use strict` gibt es allerdings: Die Hash-Zugriffe werden zur *Laufzeit* geprüft, nicht zur Compile-Zeit. Entsprechende Fehler lassen sich also nur durch Tests aufdecken, nicht schon durch Laden des Programmcode mittels `use` oder `require`.

Ableiten

Schränken wir unsere Objekte nicht übermäßig ein, wenn wir kategorisch verhindern, dass Schlüssel hinzugefügt werden können? Sind wir auf das Hinzufügen nicht beim Ableiten angewiesen oder wenn unsere Objekte eine variablen Attributmenge haben sollen? Die Antwort lautet: nein.

Das praktische an den Restricted Hashes ist, dass wir die eingeführten Restriktionen jederzeit wieder aufheben können – die Sache unterliegt unserer freien Kontrolle.

Für unsere Zwecke erweitern wir die Basisklasse "Hash" um eine Methode `unlockKeys()`, die den Objekt-Hash freischaltet, falls er gesperrt ist:

```
sub unlockKeys {
    my $self = shift;
    Hash::Util::unlock_keys(%$self);
    return;
}
```

Mittels dieser Methode können wir nun ohne Schwierigkeiten von einer restriktiven Basisklasse ableiten. Wir geben einfach nach Aufruf des Basisklassen-Konstruktors den Objekt-Hash temporär frei, um ihn nach dem Hinzufügen der Attribute der Subklasse gleich wieder zu sperren. Der resultierende Konstruktor sieht so aus:

```
package Kunde;
use base qw/Person/;

sub new {
    my $class = shift;

    my $self = $class->SUPER::new;
    $self->unlockKeys;
    $self->set(
        kundenNr=>undef,
    );
    $self->lockKeys;
    $self->set(@_);

    return $self;
}
```

Die Ergänzung besteht in dem eingefügten Aufruf `$self->unlockKeys`. Nach diesem Prinzip können wir eine beliebig tiefe Vererbungshierarchie aus Klassen mit sicheren Objekten aufbauen.

Soll eine andere Methode als der Konstruktor Attribute hinzufügen, kann diese Methode sich selbstverständlich der gleichen Mittel bedienen.

Was der Spaß kostet

Um wieviel sind Restricted Hashes langsamer? Die Antwort hierauf ermitteln wir durch drei Benchmark-Tests, die mittels des Core-Moduls "Benchmark" leicht erstellt sind (den Code geben wir hier nicht wieder).

Objekt instantiieren: Im ersten Test instantiieren wir ein Kundenobjekt und setzen dabei alle vier Attribute "kundenNr", "vorname", "nachname" und "geschlecht". Dies machen wir für eine Kunden-Klasse mit Restricted Hash (RestrHash) und eine Kunden-Klasse ohne Restricted Hash (Hash). Im Falle des Restricted Hash werden, wie oben beschrieben, im Konstruktor der Klasse "Person" alle Hash-Schlüssel gesperrt, im Konstruktor der Subklasse "Kunde" temporär entsperrt und am Ende erneut gesperrt:

```
.              Rate RestrHash       Hash
RestrHash   32733/s         --       -33%
Hash        48641/s        49%         --
```

Die Implementierung mit Restricted Hash ist um etwa ein Drittel langsamer gegenüber der Implementierung mit unbeschränktem Hash. In absoluten Zahlen ausgedrückt können wir statt ca. 49.000 Kunden-Objekten "nur" ca. 33.000 pro CPU-Sekunde instantiieren (1.8 GHz Notebook).

Attribut lesen: Der zweite Test liest ein einzelnes Attribut des Kunden-Objekts per Hash-Lookup:

```
.              Rate RestrHash       Hash
RestrHash 6244051/s         --       -1%
Hash      6330614/s         1%         --
```

Der Unterschied ist minimal. Das ist sehr günstig, da Objekt-Attribute ja am häufigsten gelesen werden.

Attribut schreiben: im dritten Test schreiben wir ein einzelnes Attribut des Kunden-Objekts durch Zuweisung an die betreffende Hash-Komponente:

```
.                Rate      Hash RestrHash
RestrHash 3020573/s        2%         --
Hash      2959690/s        --        -2%
```

Der Unterschied beim Schreiben ist ebenfalls zu vernachlässigen.

Alternative und Fazit

Eine andere Möglichkeit, fehlerhafte Objekt-Zugriffe abzuwehren, bieten Inside-Out Objekte. Unter diesem Paradigma werden Typos sogar zur Compile-Zeit erkannt. Klassen mit Inside-Out Objekten haben noch andere vorteilhafte Eigenschaften. Dies mag der Grund sein, warum Damian Conway in seinen "Best Practices" von der Nutzung von Restricted Hashes abrät - er will genau einen "besten" Weg zur Implementierung von Klassen aufzeigen. Und das sind für ihn Inside-Out Klassen.

Andererseits sind Inside-Out Klassen deutlich schwieriger zu implementieren. Wer diese Komplikationen generell oder im Einzelfall nicht auf sich nehmen will und lieber mit klassischen Hash-basierten Objekten arbeitet, bekommt mit Restricted Hashes sehr gute Möglichkeiten an die Hand.

Da Restricted Hashes normale Hashes sind, geht bei Schreib/Lese-Operationen kaum Performance verloren. Das Sperren und Entsperren in den Konstruktoren hat einen gewissen Preis, der angesichts der gewonnen Sicherheit jedoch erträglich erscheint.

Frank Seitz

Perl 6 Tutorial - Teil 8: Introspektion und Metaprogrammierung

Herzlich willkommen auch zum achten und letzten Teil dieses tiefschürfenden Tutoriums für werdende Perl 6-Programmierer. Diesmal wird es sogar noch eine Ebene tiefer gehen, zu den Interna der Sprache und wie sie befragt und verändert wird. Da diese Bereiche stärker theorielastig sind und von Pugs und Rakudo bisher kaum implementiert werden, besteht diese Folge aus mehr Text und weniger Beispielen als üblich.

Wozu Meta-Programmierung?

Von Anfang an ging es bei diesem Projekt nicht nur darum vergangene Irrtümer auszubügeln und den aktuellen Stand dynamischer Sprachen in vielem wieder ein- und überholen. Perl 6 sollte auch nach längerer Reifung ebenfalls lange halten. Was bedeutet, dass die Sprache fähig sein muss, künftige Anpassungen zu unterstützen, ohne dass die Implementation des Interpreters angefasst werden wird. Einfacher gesagt: Perl 6 sollte bessere Möglichkeiten der Meta-Programmierung erhalten, als die derzeit berüchtigten *Sourcefilter*, die abgeschafft wurden, da sie viel zu langsam und fehleranfällig sind, um in echtem Produktionscode eingesetzt zu werden.

Das ganze hat aber noch einen anderen Grund. Viele der weit verbreiteten Sprachen haben etliche Varianten oder auch Ableger, da verschiedene Fachbereiche oder Anwendergruppen verschiedene Betrachtungsweisen kennen oder bevorzugen. Um die Aufsplitterung von Perl in Derivate wie es z.B. mit PHP geschah zu verhindern, muss Perl noch wandlungsfähiger werden.

Und nur eine gemeinsame Basis macht eine Infrastruktur und damit alle Beteiligten wirklich mächtig (siehe CPAN). Gerade in den letzten Jahren wird das mit dem Modewort *domain specific language* (DSL) zusammengefasst, wenn Sprachen einer Aufgabenstellung angepasst werden können, um diese effektiv und für die mit der Materie Vertrauten verständlich zu lösen. Flexibilität (TIMTOWTDI) hatte sich Perl schon immer auf die Fahnen geschrieben und dies wird mit Perl 6 nur noch etwas weiter getrieben. Perl 6 ist in Wirklichkeit eine Meta-Programmiersprache, die sich zur Laufzeit in fast alles Denkbare verwandeln kann. Dies ist in Ordnung so, mein Larry Wall, da alles erlaubt sein soll, wenn man es vorher explizit deklariert. Und Lernende werden mit einem ausgewogenem Satz vordefinierter Befehle zu einem "guten Stil" angeleitet, bis sie fähig werden, alle Regeln zu verändern. Und je mehr sie lernen diese Regeln zu ihrem Vorteil zu ändern, umso kürzer, effektiver und oft auch lesbarer werden ihre Programme.

Dieser Ansatz bringt nicht nur Freiheit für den Programmierer sondern erlaubt die immer komplexer werdenden Softwaresysteme kompakter und beherrschbarer zu schreiben. Ein Blick in die Java-Welt zeigt die Nachteile hierarchisch-logischer Strukturen, die oft zu unbeweglich und überladen und damit schwer lesbar und veränderbar sind. Aber selbst in der Welt von Perl und Ruby lernte man in den letzten Jahren, dass die zuerst gepriesenen Web-Frameworks schnell sehr umfangreich werden. Jedes ist eine Welt für sich, die erlernt sein will, obwohl sich viel Funktionalität ähnelt. Deshalb entstanden in letzter Zeit mit Rack, Mojo und anderen Unterfangen Software, die sich auf einzelne Probleme konzentriert, um diese in logischen Einheiten mit einer möglichst einfachen API handhabbar zu halten. Die Idee dahinter ist nicht neu, denn Perl 1.0 brachte Shellbefehle in die C-Syntax, um komplexere Probleme kurz und und dem menschlichen Denken nah zu lösen. Später erfüllten in Perl 5.x gute Module den gleichen Zweck, nur das sie bereits von Programmierern der Sprache zugefügt werden konnten, ohne den C-Quellcode von Perl anzufassen. Das neue Perl 6 ist eine logische Fortset-

zung dieser Entwicklung zu einer Sprache die quasi "flüssig" ist, und in verschiedene Situationen die Syntax annehmen kann, der dem Ideal des Benutzers entspricht. Dadurch fallen Barrieren, die bisher durch Programmiersprachgrenzen aufgestellt wurden. Große Systeme bestehen zunehmend aus Teilen die in verschiedenen Sprachen und Versionen geschrieben wurden, die sich nur in Details unterscheiden können, aber dennoch von zusätzlicher *Middleware* "zusammengehalten" werden müssen. Diese Teilsysteme werden oft kaum mehr angefasst, da sie ausgereift, zuverlässig aber nicht immer leicht wartbar und auch nicht leicht erneuerbar sind. Deshalb wachsen diese Flickenteppiche zu ungeahnten Komplexitäten und es gibt bereits Lehrstühle für Professoren, die Menschen beibringen wollen mit solchen Ungetümen umzugehen. Besser wäre es doch, wenn solche Software gleich in Perl 6, vielleicht auch Ruby oder Lisp geplant wird, um diese Komplexität nicht entstehen zu lassen. Und selbst wenn "unberührbare" Software Teil einer Anwendung ist, kann eine sehr anpassungsfähige Sprache Kommunikation mit wenig Programmieraufwand ermöglichen, und Perl bleibt was es immer war: eine *glue-language*.

Wo beginnt Metaprogrammierung ?

Wenn eine kleine *sub* den verfügbaren Wortschatz erweitert, ist das bereits Metaprogrammierung? Das kommt auf den Fall an. Subroutinen können dazu verwandt werden etwas Ordnung in Spaghetticode zu bringen, was unter das Schlagwort "strukturierte Programmierung" fällt. Da die Errungenschaften von `if`, `while` und Routinen seit den 70er-Jahren Allgemeingut wurden, fällt der Begriff heute selten. Wesentlich häufiger hört man dieser Tage dagegen das Wort "funktionale Programmierung". So heißt ein Programmierstil der sehr abstrakt ist und daher in die Metaprogrammierung hineinragt. Mark Jason Dominus beschreibt in "Higher Order Perl" genauer wie das in Perl aussieht (Buchrezension in $foo Winter 2008). Das Buch ist wohl zum Teil nach den "Higher Order Functions" benannt. Das sind Routinen die eine Aufgabe sehr abstrakt lösen und deshalb sehr vielseitig einsetzbar sind. Viele Perlianer verwenden bestimmt solche "Higher Order Functions", ohne den Begriff zu kennen. Z.B. `map` und `grep` fallen in diese Kategorie. Aber mit Perl 5.0 konnte man auch bereits selber solche Funktionen schreiben, da schachtelbare Namensräume für Variablen und Codereferenzen als Parameter anwendbar waren. Ähnlich der OOP war in Perl 5 fast alles möglich, nur vieles jetzt einfacher. Eine wichtige Technik in der funktionalen Programmierung ist z.B. das *Currying*. Auf Deutsch: das Erstellen einer Codereferenz auf eine Funktion höherer Ordnung (einen Alias), bei der bestimmte Parameter festgelegte Werte haben und nur die restlichen Parameter bestimmt werden können. Also wenn eine Funktion `potenz` beliebige Potenzen berechnet (ja es ginge auch mit `**`, aber folgender Code entspricht einem funktionalem Lösungsansatz):

```
subset Num+ of Num where { $_  > 0 };
subset Num- of Num where { $_  < 0 };
subset Zero of Num where { $_ == 0 };

multi sub potenz (Num :$basis!,
                  Zero :$exponent!) {
    return 1;
}

multi sub potenz (Num :$basis!,
                  Num- :$exponent!) {
    return 1 / potenz( $basis, -$exponent);
}

multi sub potenz (Num :$basis!,
                  Num+ :$exponent!) {
    return $exponent * potenz(
                  $basis, $exponent - 1 );
}
```

Ein Funktion `quadrier` würde ich nun explizit wie folgt erhalten:

```
&quadrier := &potenz.assuming(
                        exponent => 2 );
# alternative Schreibweise:
&quadrier := &potenz.assuming(
                        :exponent(2) );
```

Die lässt sich wie erwartet aufrufen:

```
say quadrier(5);  # ergibt 25
```

`.assuming` (englisch für angenommen) drückt in einer Alltagssprache genau aus worum es hier geht: `quadrier` entspricht `potenz`, unter der Annahme, dass der Exponent 2 ist. Das `:=` ist keine Zuweisung sondern ein *binding*. P6 kennt weder direkten Manipulation der Symboltabelle mit Typeglobs noch Referenzen (der Schrägstrich (*backslash*) erzeugt jetzt *captures*, siehe Teil 6). Um einen Alias auf den Inhalt einer Variable zu erhalten, bindet man diese Variable mit `:=` an eine Andere.

```
my $planeten = 7;
my $planety := $planeten;
$planeten = 9;
say $planety;     # gibt 9
say "yes" if $planety =:= $planeten;
```

Das letzte Beispiel führt die Ausgabe aus, da die Variablen an das gleiche Objekt gebunden sind. Ein Binden zur Kompilierungszeit mit der Schreibweise `::=` mag selten gebraucht werden, aber das Ausführen von Routinen zum frühestmöglichen Zeitpunkt wesentlich öfter. In Perl 5 schrieb man dazu den Code in einen BEGIN-Block, was weiterhin möglich ist, aber Perl 6 kennt noch ein anderes Konzept, das dem ähnelt, aber weitaus mächtiger ist. Manche behaupten sogar, dass vor allem diese Macros LISP mächtiger machen als alle anderen Sprachen, was in den 60er und 70er Jahren auch gestimmt haben mag.

Die wunderbare Welt der Macros

Macros sind (wie angedeutet) Routinen, die beim Kompilieren ausgeführt werden. Im Gegensatz zu einer `sub`, ändern sie die Sprache, da sie statt eines Wertes, einen AST (Baumstruktur, die als Zwischenform beim Kompilieren entsteht) liefern, der beim Kompilieren anstelle jedes Macro-Aufrufes in den AST des Programms eingefügt wird, bevor die Ausführung beginnt.

In C ist das bei weitem einfacher. Hier sind Macros lediglich Textbausteine die an die mit dem Makronamen markierten Stellen im Quellcode eingefügt werden, bevor der Compiler sein Werk beginnt. Die alten Perl 5-Sourcefilter taten ihr Unwesen nach dem gleichen Prinzip, nur dass hier die volle Kraft der P5-Regex am Werke war. Das kann subtile, schwer zu entdeckende Fehler erzeugen, da jeder Macroprozessor blind für die Bedeutung der Sprachsyntax ist und der von ihm erstellte Quelltext nirgends einsehbar ist. Perl 6-Macros haben standardmäßig (wie alle Blöcke) keine Seiteneffekte auf die lexikalische Struktur der umgebenden Quellen. So etwas nennt die Fachwelt: hygienische Macros. Folgende Makros sind hygienisch:

```
macro summe {  3 + 4  }
macro summe { '3 + 4' }
macro summe is parsed { 3 + 4 }
```

`parsed` ist ein Trait von Routinen das nur Macros wirklich benötigen, aber da es default ist, kann es auch weggelassen werden. Und auch die Verwendung der Macros ist denkbar einfach.

```
say 2 *  summe;     # 14
say 2 *  summe();   # 14
say 2 * &summe();   # 14
```

Auch wenn alle Aufrufe `14` ergeben, so sind sie nicht identisch, da der dritte erst zur Laufzeit aufgelöst wird. Doch manchmal sind dreckige Macros genau was man möchte und dann schreibt man.

```
macro summe is reparsed {  3 + 4  }
```

Wird dieses Macro im vorigen Beispiel aufgelöst, hieße das Ergebnis `10`. Denn dieses Macro gibt nur einen String zurück, der zusammen mit dem umgebenden Quellcode compiliert wird. Dabei gilt die alte Regel: Punkt- vor Strichrechnung. Da diese Funktionsweise derartige Probleme provoziert, hat sie die beschriebene Kindersicherung und es wird standardmäßig von Macros ein AST zurückgegeben. Aber auch außerhalb von Macros kann dies getan werden. *quoting* mit dem Adverb `:code` (siehe letzte Folge) und ein `quasi` vor geschweiften Klammern oder Anführungszeichen kann das (*quasiquoting*) bewirken.

```
return quasi { say "foo" };
return Q :code / say "foo" /;
```

Beide Zeilen liefern keine Referenz auf eine Routine sondern ein kompiliertes Stück Programm. Folglich ist ein Macro eine zu `BEGIN` ausgeführte Routine deren Rückgabewert derart `quasi` kommentiert (*gequoted*) ist. Es ist eine sehr elegante Eigenschaft von Perl 6, dass jedes Sprachelement mit den Bordmitteln der Sprache beschrieben werden kann. Deswegen kann die Syntax, so reichhaltig sie auch scheinen mag, auf einen wesentlich kleineren Kern reduziert werden, aus dem sich beliebige Sprachen aufbauen lassen. Deshalb gilt für Perl 6 was John Foderaro einst über Lisp sagte: "es ist eine programmierbare Programmiersprache". Weil Perl syntaktisch viel reichhaltiger als Lisp ist, das nur Funktionsnamen, Wertelisten und runde Klammern kennt, müssen Perl-Macros wesentlich mehr können, um wirklich alle Sprachbestandteile erweitern zu können. Ein neuer Operator (z.B. für die mathematische Fakultät-Funktion) wird so erzeugt:

```
macro postfix:<!>   { [*] 1..$^n }
macro postfix:('!') { [*] 1..$^n }
```

Das gewählte Operatorsymbol kann auch in andere Klammern als den Spitzen gehüllt sein, aber der Vorsatz "postfix:" ist entscheidend, da wir einen Postfix-Operator, also einen nachgestellten Operator (wie in `$p++`) definieren wollen. Um noch zu bestimmen, wo der neue Operator seinen Platz in der Vorrangtabelle hat, könnte man hinzufügen:

```
macro postfix:<!> is equiv(&postfix:<++>) {
    ... }
```

Das könnte man auch mit `is tighter` oder `is looser` definieren. Dann bekäme der Operator eine eigene Spalte in der Vorrangtabelle die jeweils über oder unter dem angegebenen Operator liegt.

Diesem Schema entsprechend gibt es eine lange Reihe von Schlüsselwörtern, die jede Art von Operator oder Schlüsselwort bezeichnen. Es lassen sich eigne Arten des *Quoting*, Regex-Befehle, Spezialvariablen oder neue sekundäre Sigils einführen. Wenn jemand XML-Kommentare in seinem Perl haben möchte so reicht ein:

```
macro circumfix:«<!--
    -->» ($text) is parsed / .*? / { "" }
```

`$text` ist der Parameter, der den Text zwischen den Kommentarzeichen beinhaltet. Nach dem `is parsed` steht die Regex mit der geparsed werden soll.

Da entlang Alice

Aber die Manipulation der Sprache kennt noch eine Ebene. Es ist sogar möglich die Regeln zu ändern mit denen der Interpreter den Quellcode einliest. Seine Arbeitsweise wird in der STD.pm mit der Hilfe von P6-Regex-Grammatiken definiert. Wie diese formuliert werden und aufgebaut sind, behandelte die letzte Folge.

Intern wird die Sprache in mehrere Teilsprachen ("slangs") gegliedert. Die Kernsprache ist z.B. eine Grammatikobjekt, auf das mit `$~MAIN` zugegriffen werden kann. Die Regeln für das Kommentieren stehen unter `$~Q`. Und die Regex die bestimmen wie Regex einzulesen sind, finden sich unter `$~Regex`. `~` ist die Twigil dieser Sondervariablen. Und da Grammatiken nach außen normale Objekte sind, können sie abgeleitet und verändert werden wie jedes andere Objekt auch. Sämtliche Änderungen gelten nur für den aktuellen Namensraum (*scope*) und die *defaults* können jederzeit wiederhergestellt werden, da sie unter `%?LANG` gespeichert sind.

Zeig mir dein Inneres!

Objekte können in Perl 6 sehr stark durchleuchtet und verändert werden. Jede Elternklasse, jede Methode und jede Signatur (mit `obj.methode.signature`) kann abgefragt oder verschiedenst geprüft werden. `$obj.WHERE` nennt die Speicheradresse, `.WHAT` den Typ, was in etwa dem Befehl ref in Perl 5 entspricht.

Es lassen sich mit Roles beliebige Methoden zu Laufzeit in ein Objekt einfügen, aber der direkteste Weg dafür ist wohl:

```
augment slang Regex {
    token regex_metachar:<^> { ... }
}
```

`augment` (englisch für einblenden) fügt nur Regeln (Methoden) in die Grammatik (Klasse) ein, soll eine gesamte Slangdefinition ausgetauscht werden, dann ist `supersede` (englisch für überlagern) das Mittel der Wahl.

Es gibt noch viele weitere Sondervariablen, wie für den umgebenden Block (`&?BLOCK`) oder die umgebende Routine (`&?ROUTINE`) mit denen sich weit mehr tun lässt als noch in Perl 5, z.B. lassen sich alle Sprungmarken im aktuellen Block mit `&?BLOCK.labels` auflisten. Auch `&?ROUTINE.name` ist praktisch, wenn man nicht mehr weiß wo sich die Ausführung gerade befindet. Doch dies sieht man alles in den Tabellen im Anhang B meines Wiki-Kompendiums, siehe Artikelende.

Was sich sonst noch tat

Zu Beginn dieses Tutorials kündigte ich an, mit jedem Teil auch alle Änderungen bereits erwähnter Syntax zu dokumentieren. Doch zum Glück waren es weit weniger, als angenommen, sodass eine kleine Aktualisierung am Ende des letzten Teils genügt. Gleich im ersten Teil wies ich auf eine potentielle Stolperfalle:

```
# Zuweisung einer Zeile aus einem Datenstrom
$b = =$a;
# numerischer Vergleich
$b == $a;
```

Diese ist mittlerweile behoben, denn der Prefix-Operator `=` wurde ersatzlos gestrichen. Stattdessen sollte man die Methoden `.lines` und `.get` verwenden. `.lines` liefert einen Iterator und `.get` tatsächlich Textzeilen.

```
my $name = "artikel.txt";
my $handle = $name.open err die
             "Kann '$name' nicht öffnen: $!";
my $ganzer_inhalt = $handle.slurp;
my $ganzer_inhalt = slurp "artikel.txt";

for $handle.lines -> $zeile { ... }
while $handle.get -> $zeile { ... }
$zeile = $handle.get;
$handle.close;
```

Da der Kopf einer while-Schleife in den Skalarkontext und `eager` evaluiert wird, würde `.lines` dort einen Array mit allen Zeilen liefern. Da jedoch `for` den Arraykontext forciert, der standardmäßig `lazy` ist, wird dort bei jeder Iteration jeweils eine Zeile überwiesen. Im Skalarkontext werden alle Zeilen mit ~ verbunden geliefert.

Der neue Metaoperator `R` kam hinzu. Er vertauscht lediglich die Operanden, R steht für *reverse* (englisch rückwärts).

```
say 3 R- 4;     # sagt 1
say 2 R** 3;    # sagt 9
```

Der Kreuz-Metaoperator wurde vereinfacht. Von nun an reicht es dem Operator ein `X` voranzustellen, vorher hieß das noch z.B. `X~X`.

```
<a b> X~ <1 2>  # <a1 a2 b1 b2>
```

Der Hyper-Metaoperator lässt sich aber nicht so vereinfachen, da die Pfeile (<< und >>) bestimmen, wie der Operator auf Dimensionalität der jeweiligen Seite reagieren soll.

Neben automatisch generierten positionalen Parametern (erkennbar an der twigil ^), gibt es nun automatisch generierte benannte Parameter, deren Twigil : sich nahtlos in die Syntax zur Deklarierung benannter Parameter einfügt.

Selbstverständlich gab es noch weit mehr Änderungen, doch diese sind oft subtiler oder liegen außerhalb der behandelten Themen. Einen tieferen Einblick gewährt das Kompendium in der Wiki der deutschen Perl-Community unter *wiki.perl-community.de/cgi-bin/foswiki/view/Wissensbasis/PerlTafel*, oder einfacher zu tippen: *de.perl.org* und dann auf Wiki > Wissensbasis > Tutorials klicken. Viele hilfreiche Texte beinhaltet auch die Seiten von Moritz Lenz unter *perl-6.de*.

Dieses Tutorial wurde jetzt in eine Wiki online gestellt unter: *http://wiki.perl-community.de/cgi-bin/foswiki/view/Wissenbasis/Perl6Tutorial*.
Es kann nun von jedem kommentiert und verbessert werden. Ich habe vor es weiter zu verbessern und auch in Englisch zu übersetzen, wo es in der Wiki der TPF unter *http://www.perlfoundation.org/perl6/index.cgi?perl_tablets* zu finden sein wird.

Herbert Breunung

Perl 6 - Der Himmel für Programmierer - Update 2

In dieser $foo-Ausgabe wird das Perl 6-Tutorial abgeschlossen, ein geeigneter Anlass, die bemerkenswertesten Änderungen und Neuerungen seit dem blauen Heft (2/2008) zu untersuchen. Ein Hauptaugenmerk möchte ich dabei auf das im ersten Update vorgestellte SMOP legen, da es viel zu wenig beachtet wird, aber wesentliche Anregungen derzeit von dort ausgehen.

Die Kurzübersicht

Zu Pugs gibt es kaum etwas zu sagen. Er kompiliert wieder, bewegt sich aber ansonsten kaum. Deshalb wurde er im letzten Jahr endgültig von Rakudo eingeholt und teilweise schon überholt. Deswegen ist Rakudo eindeutig der zu empfehlende Perl 6-Interpreter, auch wenn es zum Glück mit STD_red/elf und mildew/SMOP Alternativen gibt. Ebenfalls nennenswert ist "November", die erste brauchbare, in Perl 6 geschriebene Software und die Seite *perl6-projects.org*, selbstredend eine Übersicht zu fast allen Perl 6-Projekten.

Im November an Weihnachten denken

Letzten August saß ich mit etwa 200 weiteren Perl-Enthusiasten im Hörsaal einer Kopenhagener Wirtschaftsschule, und verfolgte wie sich Johan Viklund und Carl Mäsak gegenseitig das Mikrophon reichten. Abwechselnd berichteten sie über ihr Vorhaben, in Perl 6 eine Wiki-Software zu schreiben. Dafür gab es für sie 3 wichtige Gründe: der Spaß an Rakudo zu hacken, der Entwicklung von Perl 6 zu helfen und die 1000$ die Conrad Schneiker (*www.athenalab.com*) dafür ausgelobt hat und Anfang diesen Jahres auszahlte. Der leicht schwermütige Name des Projektes leitet sich nicht von der nordischen Herkunft der Programmierer ab, sondern von der Vorfreude, dass es von einer funktionsfähigen Wiki bis zur Herausgabe von Perl 6.0, das bekanntermaßen zu Weihnachten erscheint, nicht mehr lang dauern kann.

Unter *http://november-wiki.org/* kann sich jeder das Ergebnis ansehen, das ein auf Rakudo laufendes, aktuelles November über CGI an einen Apache liefert. Auch wenn auf der Seite derzeit kaum Inhalte zu finden sind, so halfen *masak* und *viklund* mit ihrer Arbeit (Quellen unter *github.com/viklund/november*) Rakudo sehr. Sie fanden immer neue Schwachstellen und Fehler (allein *masak* gab über 300 Fehlerberichte in das Rakudo-RT), da eine Anwendung wie November eine ganz andere Herausforderung ist als die bisherigen Beispielskripte, die sich zumeist im einstelligen Kilobyte-Bereich bewegen.

Auch schreiben November-Autoren erste, wichtige Perl 6-Bibliotheken, wie etwa eine Übersetzung von `HTML::Template`. Derzeit erarbeiten sie die `Web.pm`, (zu finden unter *github.com/rakudo/*) einer Übertragung der Rubybibliothek *Rack* (*rack.rubyforge.org*), welche dem Ansatz des Perl 5-Meta-Rahmenwerks Mojo entspricht.

Weiter dem Himmelsweg folgend

Während *masak* voriges Jahr Frustration und Freude an der Herausforderung fand, um Rakudos technische Begrenzungen "herumzuprogrammieren", so musste er in den letzten Monaten zunehmend darauf verzichten, da viele der gefundenen Schwächen (besonders die harten, die zu Abstürzen führten) behoben wurden.

Rakudo hat sich wirklich gemausert. Rein äußerlich wurde es ein eigenes Projekt, das zu Anfang des Jahres aus Parrot's Quellcodearchiv in ein Eigenes umzog (*github.com/rakudo*). Mit "rakudo.org" bekam es dazu eine eigene Netzseite und es wird nun auch einige Tage nach einer Parrotversion veröffentlicht, da weitere Verschiebungen der Entwicklungszyklen erwartet werden.

Aber auch intern haben Patrick Michaud, Jonathan Worthington, Stephen Weeks und weitere viel getan. Mit dem pragmatischen Ansatz: möglichst schnell die wichtigsten Sprachelemente umzusetzen und dann immer dort weiter zu programmieren wo es gerade am meisten drückt, arbeiten sie sich durch die Synopsen. Deshalb ist es nicht einfach eine genaue Liste der Fortschritte aufzuzählen, auch wenn hervorhebenswert sind: die MMD (*multi method dispatch* - welche Multimethode soll wann aufgerufen werden, siehe Perl 6 Tutorial Teil 6), *roles* mit Parametern und die OOP, die bald zu 90% abgedeckt sein wird.

Ein halbwegs brauchbares Maß für den erreichten Fortschritt sind die Tests. Es werden bereits ca. 11.300 von ungefähr 16.700 existierenden Softwaretests absolviert und allein im April wurden mehr Tests erstmals erfüllt als je zuvor. Selbstverständlich hat bereits im vorigen Jahr Rakudo Pugs als treibenden Motor für die Weiterentwicklung der offiziellen Testsuite (spectest) abgelöst. Dennoch befinden sich die Tests aus historischen Gründen weiterhin im Git-Archiv von Pugs (*svn.pugscode.org/pugs/* - SVN-Spiegel). Diesem Ansatz folgend decken die Tests natürlich nicht sämtliche Regeln der Synopsen ab, sondern vor allem was Pugs und Rakudo bisher implementiert haben.

Basis des Himmelsweges

So wie November mit seinen praktischen Bedürfnissen Rakudo vorantreibt, so hilft auch Rakudo wiederum Parrot, da es von allen Compilern bei weitem der aktivste ist. Aktuell weiterentwickelt werden aber auch befunge, cardinal (Ruby 1.9), ECMAscript, fun (joy), jvm (Java VM bytecode Übersetzer), lua (Lua 5.1), matrixy (Octave), markdown, NQP, Perk (Java), Pipp (PHP), POD (perl POD), primitivearc (Arc), Porcupine (Pascal), pynie (Python), shakespeare-parrot (Shakespeare), WMLScript und XML (an SAX angelehnt). An der bereits vorgestellten Sprache NQP (not quit Perl 6) wird allerdings wenig getan, da sie das angestrebte Ziel eines Hochsprachenersatzes für PIR gut erfüllt. Innerhalb Parrot wurden vor allem 2 Subsysteme spezifiziert und generalüberholt: IO (Synopse 16) und "Concurrency" (Synopse 17 - gleichzeitige Ausführung mehrerer Programmteile). Dies konnte einige Abstürze und Abweichungen von der Perl Spezifikation in Rakudo beheben. Auch erwähnenswert ist das neue Handbuch, das unter `docs/book` in den Parrot-Quellen liegt, aber auch unter *http://docs.parrot.org/parrot/latest/html/* gelesen werden kann.

Parrot ist wie Rakudo ebenfalls organisatorisch selbständiger geworden. Am 8. Juni 2008 wurde die *Parrot Foundation* gegründet, welche nun die Rechte hält und Aktionen koordiniert. Leiterin der neuen Organisation ist Allison Randall, die als ehemalige Chefin der *Perl Foundation* die dafür nötige Erfahrung besitzt. Als neue Internetadresse war *parrot.org* noch erhältlich. Diese Abnabelung von der TPF macht organisatorisch Sinn, da Parrot nicht nur für Perl da ist. So können auch Sponsoren wie ActiveState, NLNet Foundation, Mozilla Foundation und die BBC Parrot direkter fördern.

Ein großes Datum für Parrot war auch der 17. März 2009, als die mythische Version 1.0 mit dem Namen "Haru Tatsu" vom Stapel gelassen wurde. Dies soll allerdings nicht ausdrücken dass Parrot nun vollständig ist, sondern dass die Werkzeuge zum erstellen der Sprach-Compiler nun gut genug sind und sich deren API bis Version 2.0 nicht ändern wird.

Interpreter im Vergleich

Die Idee hinter Parrot ist eine schnelle, stabile, produktionsreife VM, auf der sich alle Sprachen gegenseitig aufrufen und ihre Bibliotheken ausleihen können. So wichtig das Gespann Rakudo/Parrot und faszinierend dieser Ansatz ist, die anderen Interpreter haben auch ihre Berechtigung und sind ebenfalls wichtig für die Entstehung des Perl 6-Ökosystems. SMOP konzentriert sich z.B. darauf Perl 6 auszuführen und dabei möglichst alle Parser zu unterstützen. Vielleicht könnte es zu der Lösung werden, mit der Perl 5-Module auch unter Perl 6 benutzt werden können. Der dritte im Bunde ist das bisher kaum erwähnte elf. Dahinter steckt mehr die Pugs-Philosophie der Freude und des freien Experimentierens mit unterschiedlichen Ansätzen, Front- und Backends. Elf richtet sich damit direkt an Perl-Programmierer, die es

für Perl 6 begeistern möchte, da es nicht in Haskell sondern hauptsächlich in Perl 6 geschrieben ist.

Die Ziele einer Elfe

Elf möchte ein in Perl 6 geschriebener Perl 6 Interpreter werden. Seine Stärke liegt derzeit darin, dass es auch große Programme ausführen kann. Allerdings verwendet es derzeit einen in Ruby geschrienen Parser der auf einer älteren Version von Larry's STD.pm basiert, weswegen er *STD_red* heißt. Er soll aber bald gegen *STD_green* ersetzt werden, einen Rahmen aus Perl 6, der eine aktuelle *STD.pm* verwendet. Da *STD_green* sich derzeit noch nicht selbst kompilieren kann, benötigt es derzeit noch *STD_blue*, eine in Perl 5 geschriebene Variante die ebenfalls eine aktuelle der *STD.pm* und *gimme5* (Larry's Perl 5 Emitter der STD.pm) sowie einige kleine unsaubere Tricks verwendet.

Die einfache Sache der Programmierung

Nun endlich zu SMOP (u.a. "simple matter of programming"), dass sich sehr stark gewandelt hat. Die alte Architektur mit einer kleinen Kernsprache (smop) und einer komplexeren Hochsprache (s1p – "swamp"), wurde zugunsten eines RI (request interface) aufgegeben. Schwach angelehnt an Class::MOP (Grundlage von Moose) und anderen Quellen implementiert SMOP ein Meta-Objektprotokoll und alles (jede Variable, jedes Codestück, ja der Interpreter selbst) ist ein Objekt das unabhängig von seiner Implementation über das RI abgefragt und damit auch serialisiert werden kann. Das erlaubt es nicht nur ein Programm anzuhalten und später in einer anderen Umgebung fortzusetzen, sondern in Verbindung mit der Fähigkeit eines "Polymorphic Eval" kann SMOP die Ausführung der Interpreter zur Laufzeit wechseln und Programme, die von unabhängig entwickelten Perl 6-Kompilern ausgeführt werden, können sich gegenseitig Routinen aufrufen. Da das Konzept des RI bereits sehr ausgefeilt ist, übernimmt Jonathan derzeit diese API auch für Rakudo.

Das ist nur eines von vielen Beispielen wie sich die verschiedenen Implementationen gegenseitig unterstützen. Den Tunnel von mehreren Stellen gleichzeitig zu graben ist auch der einzige Weg, in absehbarer Zeit zu einem Ergebnis zu kommen.

Da kp6 aufgegeben wurde, besitzt SMOP jetzt mit mildew einen neuen Parser, der eine fast aktuelle *STD.pm* verwendet und wesentlich schneller als kp6 ist. Neben mildew gibt es aber noch andere Interpreter wie mold und slime. Die Anbindung zu elf (misc/elfish/elfX im Pugsarchiv) ruht zur Zeit. Die Namensgebung der SMOP-Teile (slime, mold, mildew, swamp) lässt ahnen, dass hier noch viel am Gären ist und nicht zuletzt die Förderung im letzten Jahr durch die Perl Foundation unterstreicht, dass SMOP einen wichtigen Beitrag leistet. Besonders das Objektsystem von Perl 6 wurde in vielen Details klarer, dank SMOP.

Weitere Vorhaben

Aber nicht nur die TPF unterstützt Perl 6-Projekte, auch beim diesjährigen "Google Summer of Code" ist P6 mit 3 Anträgen vertreten. Kevin Tew soll einen "low level" JIT-Kern für Parrot schreiben, Pawel Murias wird SMOP mit Multimethoden versorgen und Hinrik Sigurdsson hat sich bereit erklärt die Endbenutzerdokumentation für Perl 6 und einige dafür notwendige Werkzeuge zu schreiben. Das kann nützliche Ergebnisse hervorbringen, denn immerhin hat eines der 3 vorjährigen GSoC-Projekte die Perl 6 betreffen ("Ausbau der Test-Suit" unter $foo-Autor Moritz Lenz), zu wichtigen, praktischen Fortschritten geführt.

Ich selbst habe in den letzten Wochen damit begonnen, mit der Hilfe der freundlichen Bewohner von #perl6 (auf freenode) die November-Wiki mit mehr Inhalten zu füllen, um auch nicht technisch versierten Perl 6-Anfängern einen leichten, schnellen und aktuellen Einblick in die Welt um Perl 6 zu ermöglichen.

Herbert Breunung

Smart-Websolutions

Windolph und Bäcker GbR

Perl-Programmierung

info@smart-websolutions.de

MODULE

112% DBIx::Class

Im dritten Teil von "110% DBIx::Class" zeige ich wieder drei Dinge, die nicht immer so offensichtlich sind. Diese Themen sind teilweise von Beiträgen auf Perl-Community.de inspiriert.

Nach MAX() oder COUNT() Spalten sortieren

Als Ausgangslage für diese Frage wurde ein Forum genommen, in dem es zwei Tabellen gibt: "Threads" und "Posts". In der Tabelle "Threads" werden Informationen wie "Betreff" oder "Forum" gespeichert. In "Posts" sind die Beiträge zu dem Thread gespeichert. Wie bei den meisten Foren soll es eine Übersichtsseite geben, in der die Threads und ein paar Informationen über den letzten Beitrag eines jeden Threads gezeigt werden sollen. Die Threads sollen so sortiert werden, dass der Thread mit der letzten Antwort ganz oben steht; darunter dann der Thread mit der vorletzten Änderung.

```
+-----------+-------------------+
| Titel     |  letzte Antwort   |
+-----------+-------------------+
| Thread1   | 15.01.2009 20:09  |
+-----------+-------------------+
| Thread2   | 15.01.2009 15:15  |
...
```

Die Frage war, wie man diese Übersicht hinbekommt.

Die Abfrage soll alle Infos über die Threads und zusätzlich noch die Zeitstempel-Information des Posts liefern. Das an sich ist relativ leicht mit `DBIx::Class` zu lösen (Listing 1)

Das daraus resultierende SQL-Statement ist in Listing 2 dargestellt.

```
SELECT me.testid, me.name, Ps.timestamp
  FROM T me LEFT JOIN P Ps
    ON ( Ps.T_testid = me.testid )
  GROUP BY testid
  ORDER BY Ps.timestamp DESC:
```

Listing 2

```
#!/usr/bin/perl

 use strict;
 use warnings;
 use Local::DBIC_Schema;

 my $schema = Local::DBIC_Schema->connect( 'DBI:SQLite:board' );
 $schema->storage->debug(1);

 # Tabelle 'T' enthält die Informationen über die Threads und hat
 # eine has_many-Relation mit dem Namen Ps auf die Posts-Tabelle
 my $thread = $schema->resultset( 'T' )->search(
     undef,
     {
         join      => 'Ps',
         '+select' => 'Ps.timestamp',
         '+as'     => 'timestamp',
         order_by  => 'Ps.timestamp DESC',
         group_by  => 'testid',
     }
 );
```

Listing 1

Hierbei ist aber nicht garantiert, dass man die aktuellsten Posts aus dem Thread erwischt. Man muss also nach dem `MAX(timestamp)` sortieren. Die erste Idee sieht so aus:

```
'+select' => [
    {max => 'timestamp'}
],
'+as'     => ['maxtime'],
order_by  => 'maxtime',
```

Das funktioniert aber nicht, da `DBIx::Class` die Spaltennamen, die man mit dem `+as` vergibt nicht in das SQL-Statement übernimmt. Aus diesem Grund muss man hier mit SQL-Basics ran:

```
'+select' => \'MAX(timestamp) as tmax',
order_by  => 'tmax DESC',
```

`DBIx::Class` behandelt Skalarreferenzen als wirkliches SQL. Bei bestimmten Befehlen muss man dann aber aufpassen, wenn man von einem Datenbanksystem zu einem anderen System wechseln will. Eventuell muss man dann einige Befehle anpassen.

Mit den oben gezeigten Änderungen sieht dann das SQL so aus:

```
SELECT me.testid, me.name, MAX(Ps.timestamp)
       as tmax
  FROM T me LEFT JOIN P Ps
    ON ( Ps.T_testid = me.testid )
 GROUP BY testid
 ORDER BY tmax DESC:
```

Und das liefert die gewünschte Forenübersicht.

DBIx::Class und Log::Log4perl

Mit `$schema->storage->debug(1)` bekommt man die SQL-Statements und die Parameter auf `STDERR` angezeigt. Das ist nützlich, wenn man sich nicht sicher ist, welche Abfragen tatsächlich gestartet werden. Dazu muss man jedoch eine Kommandozeile zur Verfügung haben oder Zugriff auf die Log-Dateien. Das ist bei Shared Webhosting nicht immer möglich. Und in manchen Fällen sollen diese Ausgaben ins "normale" Logging mit eingebunden werden.

Aus diesem Grund habe ich `DBIx::Class::Log4perl` geschrieben, mit dem man diese Ausgaben mittels `Log::Log4perl` konfigurieren kann. Eine Beispielkonfiguration, die alle Ausgaben in eine Datei namens `test.log` schreibt, ist in Listing 3 zu sehen.

Wie man `Log::Log4perl` allgemein einsetzt, wurde in `$foo` "Winter 2008" vorgestellt.

`DBIx::Class::Log4perl` muss dann im Schema-Modul eingebunden werden. Dort muss dann auch angegeben werden, wo die Konfigurations-Datei liegt.

```
package My::Schema;

use DBIx::Class::Log4perl;
use base qw/DBIx::Class::Schema/;

__PACKAGE__->logger_conf( './test2.conf' );
# load classes
1;
```

Danach muss man nur noch das Logging einschalten.

```
use My::Schema
my $schema =
  My::Schema->connect( 'DBI:SQLite:db' );
$schema->logging(1);

my ($user) = $schema->resultset( 'Test' )
                    ->search({
  id => 1,
});
```

Allerdings benutzt es auch die `storage->debug()`-Funktion, so dass man dann das Debugging nicht mehr anderweitig nutzen kann (siehe $foo-Magazin Ausgabe 10).

dbicadmin

`DBIx::Class` liefert ein Skript mit, mit dem man auf der Kommandozeile mit `DBIx::Class` arbeiten kann: `dbicadmin`. Auch wenn die Dokumentation nicht wirklich optimal ist, kann das Skript hilfreich sein, wenn man mal was schnell auf der Kommandozeile machen möchte.

```
log4perl.logger = DEBUG, Stdout
log4perl.appender.Stdout = Log::Log4perl::Appender::File
log4perl.appender.Stdout.filename = test.log
log4perl.appender.Stdout.layout = PatternLayout
log4perl.appender.Stdout.layout.ConversionPattern = %m%n
```

Listing 3

Um sich alles aus einer Tabelle anzeigen zu lassen, kann man folgendes verwenden:

```
~> dbicadmin --op=select \
--schema=DBIC _Schema --class=T \
--connect=["dbi:SQLite: dbname=test"]
```

"DBIC_Schema" ist die Schema-Klasse, die man für die Anbindung an die Datenbank geschrieben hat. Mit `--class` wird dann gesagt, welche Klasse respektive Tabelle man nehmen möchte. Mit `--op` gibt man dann die Datenbankinteraktion an. Gültig sind `select`, `insert`, `delete` und `update`. Natürlich werden noch die Daten für die Verbindung benötigt. Die werden über `--connect` an das Skript übermittelt.

Unter Windows ist darauf zu achten, dass die Anführungszeichen mit einem Backslash gequotet werden.

Für alle Optionen, die auch "komplexe" Daten erlauben, wird `JSON` verwendet. Soll z.B. eine `where`-Klausel bei der Abfrage angegeben werden, dann sieht das so aus:

```
~> dbicadmin --op=select \
--schema=Local::DBIC_Schema --class=T \
--connect=["dbi:SQLite:dbname=test"] \
--where={"testid":{">":"11370"}}
```

Dieser Befehl sucht alle Einträge aus der Tabelle, deren *testid* größer 11370 ist.

Für komplexe Abfragen ist das Tool nicht so gut geeignet, da es dann schnell unübersichtlich wird, aber für einfache Dinge eignet sich das Skript.

Renée Bäcker

Meine (Lern-) Erfahrungen mit Catalyst

Bevor ich loslege, muss ich mich outen: Ja ich benutze immer noch Embperl. Doch leider halten sich die Aktualisierungen oder Bugfixes gelinde gesagt in Grenzen und die Zeit, die ich benötige, eintreffende Mails der Mailingliste zu lesen geht langsam aber sicher gegen Null. Schade eigentlich, aber im Zuge zunehmender Komplexität unserer Projekte, höheren Sicherheitsanforderungen und der Tatsache, dass viele Projekte von durchaus mehreren Entwicklern durchgeführt werden, muss etwas neues her.

Stellt sich nur die Frage, welches Framework langfristig betrachtet das richtige ist. Verfolgt man die Aktualisierungen im CPAN und den Ratschlägen anderer, stößt man unweigerlich auf `Catalyst` und `DBIx::Class`. Also muss ich mich zwangsläufig damit befassen. Um ein erstes Gefühl zu bekommen, wie man damit umgeht stellte ich mir eine kleine Aufgabe, die ich mit einem Test-Projekt erfüllen wollte. Meine Wünsche waren:

• Die Erstellung des HTML-Markups soll so bequem wie möglich sein, ich will mich nicht selbst um das Escaping und Quoting kümmern müssen. Kompakt sollte die Schreibweise natürlich auch sein. `Template::Toolkit` oder vergleichbares scheidet für mich aus.

• Erst nach einem Login sollte man an Seiteninhalte herankommen, eine Registrierung und die obligatorische "Passwort vergessen" Funktion muss natürlich auch sein. Schön wäre es, wenn bei einem Seitenaufruf eine Umleitung über die Login-Seite funktionieren würde und der Benutzer dann bei der ursprünglich gewünschten Seite landet.

• JavaScript- und CSS-Inhalte sollten mit möglichst kurzen URLs aufrufbar sein, mehrere Dateien zu einer URL kombinieren und möglichst noch Minifiziert ausliefern, um Bandbreite und Zeit zu sparen.

• Ajax sollte problemlos machbar sein.

• Vernünftige Unterstützung gängiger Datenbanken ist Pflicht. Ich möchte mich auch nicht gerne wiederholen. Das Anlegen eines Schemas in der Datenbank sollte genügen. Ich möchte keine Schema-Klassen von Hand editieren.

• Cleverer Umgang mit Formularen wäre nett. Anzeige und Validierung der Formulare sollte in jedem Falle sein.

Installation und das erste Programm

Diesen Teil meiner Erfahrung kann ich mir eigentlich sparen, denn es gab ja bereits Artikel im $foo Magazin und es existieren auch massenweise Tutorials, die genau auf diesen beiden Punkten sehr stark sind. Also fassen wir uns kurz. Nachdem ich das halbe CPAN geladen hatte, konnte ich erfolgreich eine erste Test-Applikation erstellen, den Server starten und die vorbereitete Willkommens-Seite aufrufen. Bisher eine leichte Übung.

Nun ging es also an die Umsetzung meiner Wünsche.

Template::Toolkit mag ich nicht

Zu lange habe ich die *spitzen Klammern* selbst getippt und mich um das Escaping und Quoting von Sonderzeichen gekümmert. Ich mag nicht mehr, Perl soll das für mich übernehmen. Meine Vorstellung von Markup ist, pures Perl dafür zu verwenden:

```
with { id => 'content' }
div {
    with { href => c->uri_for(c
               ->controller('Product')
               ->action_for('list')) }
      a { 'klick mich' };
  };
```

Um HTML etwa dieser Form zu erzeugen:

```
<div id="content">
    <a href="http://localhost:3000/
       product/list">klick mich</a>
</div>
```

Sicher kann man darüber streiten, ob die eine oder die andere Form des Markups besser wäre, ich persönlich bevorzuge es, immer nur eine Syntax in meinem Editor zu bearbeiten. Darüber hinaus ist die von mir bevorzugte Version ein syntaktisch korrektes Perl Programm und durch die Verschachtelung der Code-Blöcke ist es unmöglich, das Schließen von Tags zu vergessen oder Fehler beim Escaping zu machen. Einen ähnlichen Ansatz übrigens geht `Catalyst::View::Template::Declare`, mir persönlich sind einige Dinge daran aber etwas zu unflexibel.

Da ich seit über einem Jahr eine derartige Templating-Engine in der Schublade hatte, konnte ich mich daran versuchen, die Logik in Catalyst zu integrieren. Erstaunlicherweise ist es extrem einfach, eigene View Basisklassen für Catalyst zu erstellen. Der gröbste Teil der Arbeit war binnen weniger Stunden erledigt und ich hatte schon einmal Templates nach meinem Geschmack. So nebenbei entstand dann auch ein dem *WRAPPER* Konzept von `Template::Toolkit` ähnliches Verfahren, das die Inhalte einzelner Seiten in ein entsprechend vorbereitetes Gerüst packt.

Nach Erstellung meiner View Basisklasse (ich nannte sie *Catalyst::View::ByCode*) musste ich also nur noch einen geeigneten View erstellen und meine Applikation konfigurieren. Eine Konfigurations-Zeile in der Applikation genügte dafür -- und schon wurden anstelle der `Template::Toolkit` Templates meine eigenen genommen:

```
package MyApp;
...
__PACKAGE__->config(
    default_view => 'ByCode',
    ...
```

Als Seitengerüst genügte diese simple Datei (wrapper) - siehe Listing 1.

Zwei knappe Dinge zum Verständnis:

`yield`
diese Aufrufe sorgen dafür, dass entsprechend definierte Bereiche in Form von anderen Template-Dateien an den jeweiligen Stellen eingebunden werden. Die Form ohne weitere Argumente bindet den eigentlichen Content des zu rendernden Dokuments ein.
`load`
sorgt je nach Verwendung für die Generierung der HTML-Tags "script" oder "link" um auf möglichst einfache Weise CSS und JavaScript einzubinden. Ja, ich weiß schon, keine Media-Types, keine Browser-Weiche, keine Angabe einer JavaScript-Version, aber irgendeinen Tod muss ich ja sterben...

Du kommst hier nicht rein

Dieser Teil meiner Anforderungen klang zunächst extrem problemlos. Die beiden Plugins `Catalyst::Plugin::Authentication` und `Catalyst::Plugin::`

```
doctype 'xhtml';
html {
  head {
      title { stash->{title} || 'namenlos' };
      load Js  => qw(prototype scriptaculous default.js);
      load Css => qw(site.css);
  };
  body {
      with {id => 'header'}  div { yield 'header'; };
      with {id => 'main', class => 'clearfix'} div {
          with {id => 'leftnav'} div { yield 'leftnav'; };
          with {id => 'content'} div { yield; };
      };
      with {id => 'footer'}  div { yield 'footer'; };
  };
};
```

Listing 1

```
package MyApp::RequireLoginController;
use strict;
use warnings;
use parent 'Catalyst::Controller';

sub auto :Private {
    my $self = shift;
    my $c = shift;

    if (!$c->user_exists) {
        $c->flash->{next_page} = $c->req->uri->as_string;
        $c->response->redirect(
            $c->uri_for($c->controller('Login')
                        ->action_for('index'))
        );
        return;
    }

    return 1;
}

1;
```

Listing 2

`Authorization::Roles` erledigen diese Aufgabe mit Bravour. In Verbindung mit dem ebenfalls als Plugin vorhandenen Session-Management sind die benötigten Funktionalitäten quasi schon "an Bord". Es geht nur noch darum, die notwendigen Formulare und Login-Logik zu erstellen, das aber ist reines Handwerk.

Etwas Kopfzerbrechen bereitete mir Anfangs die Problematik der automatischen Umleitung zur Login-Seite. Da alles Bisherige so einfach zu bewerkstelligen war, sollte sich doch auch für diese Anforderung eine einfache Umsetzung stricken lassen? Klar, wir sind ja objektorientiert, kennen Mehrfachvererbung und Catalyst bietet mit seiner `auto` Methode einen erstklassigen Hook, um solche Funktionalitäten in jeden Controller einzuklinken.

Und tatsächlich: Die Controller, deren URLs erst nach einem Login aufrufbar sein sollen, beerben einfach eine Klasse, deren `auto` Methode entsprechende Prüfungen vornimmt, Umleitungen ausführt und fertig. Effektiver Code: 10 Zeilen plus eine weitere Basisklassen-Angabe pro Controller -- das war ja einfach! (siehe Listing 2)

Die Controller für Seiten, die einen Login erfordern beginnen dann so:

```
package MyApp::Controller::Products;
use strict;
use warnings;
use parent 'MyApp::RequireLoginController';

   # ...
```

Intelligenter Umgang mit Assets

`Catalyst::View::JavaScript::Minifier::XS` schien auf den ersten Blick die passende Lösung zu sein. Leider benötigt diese View-Klasse noch einen zusätzlichen Controller, um richtig zu funktionieren und versteht keinerlei Konfigurations-Direktiven. Der Autor hat auch bis heute in keiner Weise auf einen eingereichten Bug-Report reagiert. Eine Alternative wäre noch `Catalyst::Plugin::Assets` gewesen, dieses Plugin allerdings operiert optimalerweise mit `Template::Toolkit`.

Mein Ansatz hier war es, eine eigene Basisklasse für meine JavaScript- und CSS-Controller zu generieren. Damit komme ich mit der einmaligen Umsetzung der gesamten Logik aus. Aus den Zusatz-Pfaden am Ende der URL wollte ich die auszuliefernden Dateien auslesen, diese kombinieren, minifizieren und anschließend ausliefern. Über ein in Form von Konfigurations-Einstellungen vorzunehmendes Abhängigkeits-Management sollten grundlegende Dinge (z.B. JavaScript Bibliotheken) automatisch mitkommen. Da nach der Kombination der Dateien bereits der komplette auszuliefernde Text vorliegt, entschied ich mich für die Ausgabe direkt im Controller ohne einen eigenen View bemühen zu müssen. Damit war auch die Instantiierung extrem einfach.

Auch hier war ich angenehm erstaunt, dass selbst das "Verbiegen" der üblichen Logik, die die meisten Catalyst-Anwender bevorzugen, extrem leicht möglich ist. Das nenne ich flexibel!

Nicht Meister Proper sondern Ajax

So manch einer mag sich an dieser Stelle fragen: Was ist denn so spannend bei Ajax? Es ist doch auch nur ein Request, der eben nur entsprechende HTML-Fragmente bzw. JSON Resultate ausliefert. Hierbei allerdings wäre mir, wenn gerade die Session abläuft ein Redirect zur Login Seite (die dann möglicherweise innerhalb der zuletzt dargestellten Seite eingebettet würde) gar nicht recht. Eine simple Meldung und der entsprechende HTTP Status würde hier vollkommen reichen. Außerdem soll die per Ajax aufgerufene URL ausschließlich ein HTML-Fragment ausliefern anstelle die bei sonstigen Seiten erwünschten Dekorationen wie Navigation, Fußzeilen etc. Als URL-Schema wäre mir an dieser Stelle *http://host/controller_base/ajax/irgendwas...* recht. Denn damit wäre es ein Leichtes, die notwendige Ajax-Logik in den gleichen Controllern unterzubringen, die auch die regulären Seiten bearbeiten.

Eine primitive Basisklasse konnte hier (fast) alle notwendigen Dinge vorbereiten (siehe Listing 3).

Die tatsächliche Ajax-Aktion braucht dann lediglich die hier definierte Chain zu benutzen und schon passiert das Erwünschte (Vorausgesetzt natürlich die Template-Generierung weiß mit den hier gesetzten stash-Variablen umzugehen).

```
package MyApp::Controller::Products;
use strict;
use warnings;
use parent qw(
              MyApp::RequireLoginController
              MyApp::AjaxController
              );

# URL: /products/ajax/detail/<<product_id>>
sub detail :Chained('ajax') :Args(1) {
    my $self = shift;
    my $c = shift;
    my @args = @_;

    # rest wie üblich...
}

...
```

Ganz zum Nulltarif übrigens kann man bequem URIs für beliebige Aktions-Methoden generieren:

```
# in einem beliebigen Controller:
my $uri = $c->uri_for(
            $c->controller('Products')
              ->action_for('detail'),
                $product_id );

# im Controller 'Products' geht es kürzer:
my $uri = $c->uri_for(
            $c->controller()
              ->action_for('detail'),
                $product_id );
```

Sollte sich an den Attributen der Aktions-Methode etwas ändern, der obige Aufruf bleibt gleich und liefert weiterhin die dann gültige URI, es gibt also keine Notwendigkeit die Logik der URI-Generierung mehrfach zu programmieren oder gar hart codierte URIs zu verwenden.

```
package MyApp::AjaxController;
 use strict;
 use warnings;
 use parent 'Catalyst::Controller';

 sub begin :Private {
     my $self = shift;
     my $c = shift;
     $c->stash->{is_ajax_request} = 0;
 }

 # :PathPrefix === :PathPart('controller_base')
 # :Chained === :Chained('/')
 sub base :PathPrefix :Chained :CaptureArgs(0) {}

 # :PathPart('ajax') ist default...
 # ohne :CaptureArgs(0) wäre /controller_base/ajax als URL aufrufbar
 sub ajax :Chained('base') :CaptureArgs(0) {
     my $self = shift;
     my $c = shift;

     $c->stash->{wrapper} = undef;
     $c->stash->{is_ajax_request} = 1;
 }

 1;
```

Listing 3

Nichts geht ohne Persistenz

Die Einbindung einer Datenbank in Catalyst entpuppt sich als Kinderspiel. Ein beim Anlegen der Applikation erzeugtes Script erledigt (auch wiederholt) die Anlage der notwendigen Klassen, was `DBIx::Class::Schema::Loader` aus den in der angegebenen Datenbank vorhandenen Tabellen erledigt.

```
./script/myapp_create.pl \
model DB \
DBIC::Schema MyApp::Schema \
create=static \
dbi:Pg:dbname=myapp postgres password
```

Nach dem Lauf dieses Scripts werden neben der Model-Klasse auch die Schema-Klassen angelegt. Diese enthalten einen MD5-Schlüssel und erlauben, dass nach einem entsprechenden Kommentar auch eigene Erweiterungen vorgenommen werden dürfen, die durch eine Neu-Generierung per obigem Script dann erhalten werden.

Ob nun Tabellen-Namen im Singular oder Plural verwendet werden und wie man die Benennung der Spalten gestaltet, sollte jeder für sich selbst entscheiden. Ich habe mich letztlich aus Lesbarkeitsgründen für den Singular entschieden und verwende einfache sprechende Spaltennamen. Fremdschlüssel heißen exakt genau so wie die Tabelle, auf die sie referenzieren.

```
package MyApp::Controller::Login;
use strict;
use warnings;
use parent qw(Catalyst::Controller::HTML::FormFu);

# :FormConfig === FormConfig('login/register.yml')
sub register :Local :FormConfig {
    my $self = shift;
    my $c = shift;

    my $form = $c->stash->{form};

    if ($form->submitted_and_valid) {
        #
        # Formulareingabe erfolgreich - ab in die Datenbank
        #
        $form->model->update(
            $c->model('DB::Person')
              ->new_result({})
        );

        # vielleicht noch eine Umleitung zu anderer Seite
        # $c->response->redirect( ... );
    } elsif ($form->has_errors) {
        #
        # Fehler im Formular. Wenn wir hier nichts tun, wird das
        # Formular einfach nochmal mit Fehlern angezeigt.
        #
    }

    $c->stash->{title} = 'Registrierung';

    # im Template:
    #    stash->{form}->render();
    # um das Formular (evtl. mit Fehlern) anzuzeigen
}

# einfache Callback Funktion zur Formularprüfung
# hier: eingegebene eMail Adresse darf noch nicht vergeben sein
#
sub email_check :Private {
    my $value = shift;
    my $result = MyApp->model('DB::Person')
                      ->search({email => $value})
                      ->single();

    return $result ? 0 : 1;
}
```

Listing 4

Formulare und Datenbank: ein Kinderspiel

Mittlerweile wird von den üblichen Tutorials sehr stark `Catalyst::Controller::HTML::FormFu` propagiert. Und das ganz zu Recht. Formularsysteme gibt es ja reichlich im CPAN, was aber dieses Modul besonders stark macht, ist gerade die harmonische Integration von Formularen zu den Schema-Klassen, also unserer Datenbank.

Besonders nett ist die Tatsache, dass Formulare durch Datenstrukturen bereitgestellt werden. Diese wiederum lassen sich als YAML-Dateien in einem **forms** Verzeichnis unterbringen. Ich empfand es als praktisch, die Verzeichnis-Strukturen für Controller, Templates und Formulare möglichst parallel zu halten. So sind die beteiligten Bestandteile schnell zu finden. Als kleines Beispiel soll das Registrierungs-Formular dienen.

Zunächst die Controller-Logik (siehe Listing 4).

Das Formular könnte dann so aussehen, wie in Listing 5 dargestellt.

Zugegeben, recht viel mehr Tipparbeit ist die HTML-Fassung des Formulars auch nicht. Aber der Nutzen ist die Logik im Verborgenen. Neben der reinen Darstellung des Formulars kümmert sich `HTML::FormFu` auch um die Verarbeitung der Query-Parameter, prüft die Eingaben und unterstützt bei Abfragen und Eintragungen in die Datenbank. Spätestens dann ergibt sich ein erheblicher Unterschied zur händischen Programmierung.

```
---
indicator: Anmelden
auto_fieldset: 0

elements:
  - type: Text
    name: name
    label: Name
    constraint: Required

  - type: Text
    name: email
    label: eMail
    constraints:
      - Required
      - Email
      - type: Callback
        callback: 'MyApp::Controller::Login::email_check'
        message: Wird bereits verwendet

  - type: Text
    name: login
    label: Login
    constraints:
      - Required
      - type: Callback
        callback: 'MyApp::Controller::Login::login_check'
        message: Wird bereits verwendet

  - type: Password
    name: password
    label: Passwort
    constraint: Required

  - type: Password
    name: repeat_password
    label: Passwort Wiederholung
    constraints:
      - Required
      - type: Callback
        callback: 'MyApp::Controller::Login::double_check'
        message: Nicht zweimal gleich eingegeben

  - type: Submit
    name: Anmelden
    value: Anmelden
```

Listing 5

Fazit

Den Köpfen hinter `Catalyst` ist großartiges gelungen. Das auf den ersten Blick fälschlicherweise von mir als schwerfällig beurteilte Design entpuppt sich bei näherer Betrachtung als extrem flexibel und erweiterungsfähig. Lediglich der Funktionsumfang verwirrt anfangs. Es gibt für alle üblichen Anforderungen einen Lösungsvorschlag, wenn der nicht gefällt sind entsprechende Anpassungen (neben der üblichen Lernkurve) mit Leichtigkeit zu erreichen. Für mich war es jedes Mal erschreckend zu sehen, wie klein die notwendigen Module waren.

Wolfgang Kinkeldei

Link zur "github-Spielwiese" des Autors (mit komplettem Code): *http://github.com/wki*

REST-APIs in Perl-Anwendungen

Web 2.0 ist ein Schlagwort, das schon seit Jahren nicht mehr wegzudenken ist. Einer der wichtigsten Web 2.0-Aspekte ist, dass Nutzer etwas zum Inhalt der Seite beitragen. Dann stellt sich irgendwann auch die Frage, auf welchem Weg der Nutzer seinen Beitrag leisten kann. Meistens geht das über Formulare auf der Webseite wie zum Beispiel in Foren.

Aber die Interaktion mit Nutzern und/oder anderen Webseiten muss häufig noch auf anderem Wege geschehen um Inhalte automatisch eintragen oder auslesen zu können. Man kann zwar auch Formulare automatisch ausfüllen und abschicken - dafür gibt es z.B. `WWW::Mechanize` auf CPAN - aber das ist sehr aufwändig und die Auswertung der Ergebnisse ist nicht immer einfach.

So wurden früher meist RPC (*Remote Procedure Call*) und SOAP (*Simple Object Access Protocol*) für die automatische Interaktion verwendet. In letzter Zeit wird aber das so genannte REST immer beliebter. Ich habe so eine API jetzt für `http://perl-nachrichten.de` umgesetzt, damit Firmen Stellenausschreibungen automatisch posten können.

Was ist REST

Die Abkürzung REST steht für *Representational State Transfer* und der Begriff wurde von Roy Fielding in seiner Dissertation geprägt. REST ist eine Architektur für verteilte Anwendungen, bei der jede Ressource eine eigene URI hat. In der URI ist nur die Ressource (Daten) "kodiert" und nicht, was mit dieser Information gemacht werden soll.

Eine Beispielhafte URI sieht beim Beispiel von Jobs auf Perl-Nachrichten.de so aus:

http://perl-nachrichten.de/rest.cgi/=/jobs/314

Das = wird hier als Abgrenzung der REST-API zu den "normalen" URIs im alltäglichen Surfen verwendet. Die gezeigte URI beschreibt nur die Ressource "Job-Angebot mit der ID 314".

Die URIs werden in verschiedenen Quellen zu REST als "*Nomen*" bezeichnet.

Wenn jemand dieser URI sieht, kann er/sie noch nicht sagen, was für eine Aktion mit dieser URI ausgeführt wird.

Die Aktion, die gemacht werden soll, wird dabei von der Request-Methode bestimmt. Dabei werden die Daten via HTTP übertragen, ohne eine eigene Transportschicht wie etwa SOAP zu haben. REST zwingt uns HTTP zwar nicht auf, aber das wird vermutlich das Protokoll sein, das für eine REST-API am häufigsten eingesetzt wird. Deshalb beschränke ich mich in meiner Darstellung auf die Umsetzung der API mit HTTP.

Die typischen Aufgaben von CRUD (Create, Read, Update, Delete) werden dabei auf die Methoden des HTTP-Requests gemappt:

- PUT: Mit *PUT* werden neue Informationen auf dem Server abgelegt.
- GET: Werden Informationen vom Server benötigt, muss man *GET* verwenden.
- POST: Ein Update der Informationen erfolgt bei *POST*-Requests.
- DELETE: Wie der Name schon sagt, werden mit *DELETE* Informationen vom Server gelöscht.
- HEAD: Sind Metainformationen zu einer Ressource auf dem Server abgelegt, kann man diese mit *HEAD* anfragen.

Diese Aufgaben werden als "*Verben*" bezeichnet.

Wie die Ergebnisse dieser Anfragen aussehen, ist bei REST nicht spezifiziert. Man kann also XML-Daten oder JSON oder ein anderes Format ausliefern. Ich habe bei Perl-Nachrichten.de die Möglichkeit vorgesehen, verschiedene Formate anzufordern, damit jeder Client das bekommt, was er am besten kann.

Aber nicht nur die Ergebnisse können unterschiedliche Formate haben, auch die Daten für den Webservice können in verschiedenen Formaten vorliegen. Der Client und der Server müssen sich nur einig darüber sein, was angeliefert wird. Dies wird über die Header-Daten festgelegt. Bei *PUT* und *POST* wird mit der `Content-Type`-Angabe mitgeteilt, welche Art von Daten vom Client an den Server geschickt werden. Mit `Accept` teilt der Client dem Server mit, in welchem Format die Rückgabe sein soll.

Bei den Client-Beispielen ist zu sehen, wie man das mit Perl machen kann.

Diese Formate sind die dritte Säule bei REST - der "*Content*".

Thomas Beyer nennt auf http://www.oio.de/public/xml/rest-webservices.htm die folgenden Merkmale einer REST-Anwendung:

- Die Kommunikation erfolgt auf Abruf. Der Client ist aktiv und fordert vom passiven Server eine Repräsentation an, bzw. modifiziert eine Ressource.
- Ressourcen, die Objekte der Anwendung, besitzen eine ihnen zugeordnete URI, mit der sie adressiert werden können.
- Die Representation einer Ressource kann als Dokument vom Client angefordert werden.
- Repräsentationen können auf weitere Ressourcen verweisen, die ihrerseits wieder Repräsentationen liefern, die wiederum auf Ressourcen verweisen können.
- Der Server verfolgt keinen Clientstatus. Jede Anfrage an den Server muss alle Informationen beinhalten, die zum Interpretieren der Anfrage notwendig sind.
- Caches werden unterstützt. Der Server kann seine Antwort als Cache fähig oder nicht Cache fähig kennzeichnen.

Warum sollte man jetzt eine solche REST-API zur Verfügung stellen? Mit so einer API wird das Arbeiten mit der Webanwendung einfacher und flexibler. So kann man unter Umständen komplett eigene Clients für eine Webanwendung schreiben und muss für immer wiederkehrende Aufgaben den Browser öffnen. Oder man kann die Anwendung in eigene Anwendungen integrieren.

In den folgenden Abschnitten zeige ich, wie man so eine REST-API relativ einfach umsetzen kann.

Einfaches Beispiel mit CGI.pm

Mit `CGI.pm` ist es ziemlich einfach, eine REST-API umzusetzen. Da bei REST die Art der Interaktion durch die Art des Requests bestimmt wird, ist die Methode `request_method` wichtig. Wie oben beschrieben, wird mit weiteren Header-Daten festgelegt, mit welchen Datenarten Client und Server kommunizieren. Auch dafür bietet `CGI.pm` Methoden an.

Mit einem Hash als Dispatcher wird bestimmt, welche Subroutine letztlich aufgerufen wird. Dabei dienen die Requesttypen als Schlüssel und als Werte nehme ich die Coderefernzen. Der Einfachheit halber wird in dem Beispiel davon ausgegangen, dass nur XML-Daten gesendet werden (siehe Listing 1).

In der Subroutine `create_entry` wird der "Content-Type" überprüft. Sollen mehrere Formate erlaubt sein, muss hier das Parsen der Daten in Abhängigkeit zu der Angabe im "Content-Type" geschehen. Ähnlich sieht es in `get_entry` aus: Je nach Format der Daten muss eine andere Art der Serialisierung der Daten genommen werden.

REST-APIs mit Frameworks

In dem Beispiel mit `CGI.pm` sieht man, dass für eine komplexe REST-API viel Arbeit notwendig ist: Man muss sich über akzeptierte Formate Gedanken machen und wie man die Daten für den Client aufbereitet. Wenn man Catalyst verwendet kann man sich das Leben recht einfach machen, da man dann `Catalyst::Controller::REST` verwenden kann. Das Modul hat Voreinstellungen zum Parsen und Erzeugen der verschiedenen Formate.

Der Übersichtlichkeit halber verwende ich für jede Entität, die per REST-API abgebildet werden soll, eine extra Klasse. Die eigene Klasse muss von `Catalyst::Controller:REST`

```
#!/usr/bin/perl

 use strict;
 use warnings;
 use CGI;
 use DBI;
 use XML::Simple;

 my $cgi = CGI->new;
 my $dbh = DBI->connect( 'DBI:....','user','pass' ) or die $DBI::errstr;

 my $method = uc( $cgi->request_method );
 my %hash   = (
     PUT  => \&create_entry,
     GET  => \&get_entry,
 );

 my $sub = $hash{GET};
 if( exists $hash{$method} ){
     $sub = $hash{$method};
 }
 else {
     die "Method not supported";
 }

 $sub->( $cgi );

 sub create_entry {
     my ($cgi) = @_;

     die "no xml data" unless $cgi->content_type ne 'text/xml';

     my $data = XMLin( $cgi->param( 'PUTDATA' ) );

     $dbh->do(qq~
       INSERT INTO tabelle VALUES( $data->{node} )
     ~);

     print $cgi->header( -status => 201 )
 }

 sub get_entry {
     my ($cgi) = @_;

     my @accepts = $cgi->Accept();

     die "you don't accept XML" unless grep{ $_ eq 'text/xml' }@accepts;

     my $sth = $dbh->prepare( qq~SELECT * FROM tabelle~ );
     $sth->execute;

     my $xml = "<root>";
     while( my $hashref = $sth->fetchrow_hashref ){
         $xml .= qq~<item>
           <id>$hashref->{id}</id>
           <node>$hashref->{node}</node>
         </item>~;
     }
     $xml .= "</root>";

   print $cgi->header, $xml;
}
```

Listing 1

erben. Für die Entität - hier "point" - wird eine Methode ohne Ihnhalt angelegt. Diese muss die "ActionClass" mit dem Parameter 'REST' haben.

Mit ActionClass kann man angeben, welche `Catalyst::Action::*`-Klasse (hier: `C::A::REST`) verwendet werden soll. Das soll hier aber nicht Thema sein.

Als nächstes muss man dann noch für die einzelnen HTTP-Request-Arten eine Methode schreiben, wobei das Format so aussieht: `<Entitaet>_<Request>`, also z.B. `point_GET`. Damit wird jeder *GET*-Request auf *point* von der Methode `point_GET` abgearbeitet.

Das Modul parst auch gleich den Inhalt – je nach `Content-type` – und ermöglicht den Zugriff auf die geparsten Daten über `$c->req->data`.

Zum Schluss muss man dann nur noch den Status setzen und die Entität übergeben. Die Serialisierung der Daten übernimmt dann wieder das Modul. In den Codebeispielen dieser Ausgabe ist die komplette Beispielanwendung zu finden (Die benötigten Module werden in einer README-Datei genannt).

```
package GPS::Controller::REST;

use base 'Catalyst::Controller::REST';

sub point : Local : ActionClass('REST') { }

# Answer GET requests to "thing"
sub point_GET {
    my ( $self, $c ) = @_;

    my @points = $c->model('Point')
                   ->search->all;

    my @found;
    for my $point ( @points ){
        push @found, {
            Name => $point->Name,
            Lat  => $point->Lat,
            Lon  => $point->Lon,
        };
    }

    $c->stash->{template} = 'rest/point';
    $c->stash->{points}   = \@found;
}

1;
```

Clients mit Perl schreiben

Grundsätzlich ist es ziemlich einfach, mit Perl Clients für REST-APIs zu schreiben. Das Bundle `libwww-perl` von Gisle Aas beinhaltet viele Module - unter anderem `LWP::UserAgent` und `HTTP::Request::Common`, die für diese Aufgabe benötigt werden. Für die API von Perl-Nachrichten.de sind die Clients weniger als 20 Zeilen lang und dabei wurde nicht auf Platzsparenden Code geachtet.

Für *GET*, *POST* und *HEAD* gibt es Methoden in `LWP::UserAgent`, für *PUT* und *DELETE* muss noch das Modul `HTTP::Request::Common` verwendet werden.

Hier sollen beispielhaft die Clients für *GET*- und *PUT*-Anfragen gezeigt werden.

Der Client in Listing 3 erstellt einen neuen Eintrag. Der API-Key ist hier notwendig, damit keine Spam-Einträge auf der Plattform landen. Da hier XML-Daten als Quelle geschickt werden, muss als `Content-Type` noch "text/xml" angegeben werden.

Der zweite Client holt sich das Jobangebot mit der ID 333. Der Server liefert dabei eine XML-Struktur zurück, da mittels `Accept` festgelegt wurde, dass der Rückgabewert vom Typ "text/xml" sein soll.

Andere Clients

Wie so häufig findet man auf CPAN auch schon vorgefertigte Clients, mit denen im Allgemeinen Clients noch schneller entwickelt werden können. Das ist vor allem dann sinnvoll,

```
#!/usr/bin/perl

use strict;
use warnings;
use LWP::UserAgent;

my $ua = LWP::UserAgent->new;
my $response = $ua->post(
  'http://perl-nachrichten.de/rest.cgi/=/jobs',
  Content => qq~<job>
      <api_key>XXXXXXXXXX</api_key>
      <subject>Perl programmer for OTRS</subject>
      <description>We are searching for a talented Perl programmer
      who wants to work for OTRS the great ticketing system.</description>
  </job>~
);
print $response->content;
```

Listing 2

```
#!/usr/bin/perl

use strict;
use warnings;
use LWP::UserAgent;

my $ua = LWP::UserAgent->new;
my $response = $ua->get(
  'http://perl-nachrichten.de/rest.cgi/=/jobs/333',
  Accept => 'text/xml'
);
print $response->content;
```

Listing 3

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

use Error qw(:try);
use RT::Client::REST;
use RT::Client::REST::Queue;

my $rt = RT::Client::REST->new(
    server  => 'http://rt.cpan.org',
);

$rt->login(
    username=> 'yyyy',
    password=> 'xxxx',
);

my $queue = RT::Client::REST::Queue->new(
  rt => $rt,
  id => 'Test-TestCoverage',
);

my $results;
try {
    $results = $queue->tickets;
} catch Exception::Class::Base with {
    my $e = shift;
    die ref($e), ": ", $e->message;
};

my $count = $results->count;
print "There are $count tickets\n";

my $iterator = $results->get_iterator;
while (my $t = &$iterator) {
    print "Id: ", $t->id, "; Status: ", $t->status,
        "; Subject ", $t->subject, "\n";
}
```

Listing 4

wenn die REST-API komplexer ist als das - zugegebenermaßen sehr einfache - Beispiel von Perl-Nachrichten.de. Ein Beispiel ist `RT::Client::REST`, mit dem man auf die REST-API einer Request Tracker (*RT*) zugreifen kann. Das Modul liefert die ganze "Infrastruktur". So kann man sich als Autor von CPAN-Modulen die Tickets zu einem seiner Module anzeigen lassen (siehe Listing 4).

Die REST-API von RT ist wesentlich mächtiger, da man die komplette Bearbeitung von Tickets mit eigenen Skripten abhandeln kann. Vom Auflisten der Tickets über das Editieren bis hin zum Schließen der Tickets.

Für die Fans der "parallelen Abarbeitung von Aufgaben" gibt es auch `POE::Component::Client::REST`, mit dem sich in einer POE-Umgebung sehr einfach REST-Clients erstellen lassen. Auf POE werde ich in der übernächsten Ausgabe von $foo eingehen.

To PUT or not to PUT?

Gerade bei einigen Shared Webhosting Angeboten dürfte PUT deaktiviert sein (Erkennbar an der "Method not allowed"-Meldung wenn man die Seite mit PUT anfragt), da man damit Sicherheitslöcher öffnen kann. Auf http://www.apacheweek.com/features/put werden diese Punkte angesprochen, die unbedingt beachtet werden sollten:

- Sicherstellen, dass das PUT Skript nur von autorisierten Usern gestartet werden kann
- Sicherstellen, dass das Skript nur Webinhalte aktualisieren kann (und keine Systemdateien)
- Jeder User dürfen nur eigene Seiten aktualisieren und nicht die Seiten anderer Personen auf dem gleichen Server

Um PUT für einige Skripte zu erlauben, muss man das in der Apache-Konfiguration angeben. Mittels `Script PUT /pfad/zu/script` kann man PUT-Anfragen an das Skript weiterleiten.

Bei einigen Webhostern wird man auf Widerstand treffen, wenn es darum geht, PUT-Anfragen zuzulassen. Ich habe für Perl-Nachrichten.de das Erstellen der Jobangebote auch auf POST-Anfragen umgebogen und erlaube das Aktualisieren der Angebote nicht. Das ist zwar nur ein Workaround, für meine Zwecke aber vollkommen ausreichend.

Renée Bäcker

Rezension - TWIKI

In 9 Kapiteln geht es über eine der größten Wiki-Engines: TWiki. Das Buch richtet sich sowohl an Administratoren als auch an Anwender.

In den Kapiteln wird folgendes behandelt:
- Kapitel 1: Installation
- Kapitel 2: Erste Schritte
- Kapitel 3: Struktur des TWiki
- Kapitel 4: Administration
- Kapitel 5: Anpassen
- Kapitel 6: Plugins
- Kapitel 7: Such-Funktion und Datenbanken
- Kapitel 8: Anwendung
- Kapitel 9: Eigene Plugins

Im ersten Kapitel beschreibt Marbach, wie das TWiki zu installieren ist. Dank der CD, die dem Buch beiliegt, kann man das meiste gleich ausprobieren. Beschrieben wird die Installation auf verschiedenen Umgebungen: Webserver mit root-Zugang, Webhost ohne root-Zugang und die Installation unter Windows. Danach wird auf die erste Konfiguration eingegangen.

Die "ersten Schritte" sind für die TWiki-Neulinge sehr interessant, da die wichtigsten Begriffe und der Aufbau ("Was ist ein Web", Menüs) erläutert wird. Ein erfahrener TWiki-User kann das Kapitel getrost überspringen.

Das dritte Kapitel ist quasi Pflicht für jeden, der mehr mit dem TWiki macht - zum Beispiel Formulare oder Webs erstellt. In diesem Teil des Buches werden die wichtigsten TWiki-Variablen erläutert. Das ist der Punkt, an dem ich mir mehr Ausführlichkeit gewünscht hätte. In dem Kapitel sind einige Tabellen als Übersicht gezeigt, aber kein zusammenhängendes TWiki-Dokument, an dem die Variablen mal im Einsatz gezeigt werden. Hier hilft dann wieder die CD mit dem TWiki weiter. Man kann gleich die Variablen ausprobieren und die Bedeutung testen.

Die Administratoren sollten sich das vierte Kapitel ganz genau anschauen. Denn hier wird alles über Zugriffsrechte und andere Konfigurationen (z.B. Anti-Spam-Einstellungen) geschrieben. Auf diese Themen wird recht genau eingegangen, was bei der Wichtigkeit der Themen auch nötig ist. Auch hier gilt wieder: am besten gleich bei dem mitgelieferten TWiki ausprobieren.

Mit dem einfachen Installieren eines Wikis ist es meist nicht getan. Es muss an das "Corporate Design" angepasst werden und wie das geht, wird im fünften Kapitel erklärt.

TWiki ist sehr gut erweiterbar und es existieren etliche Plugins. Einige davon sind schon vorinstalliert. Diese werden im ersten Teil von Kapitel 6 erläutert. Empfehlenswerte Plugins werden dann im zweiten Teil vorgestellt. Wolf Marbach greift sich dabei einen kleinen Teil heraus, der ihm besonders nützlich erscheint. Ob diese für den Leser nützlich sind, zeigt erst der Einsatzzweck des Wikis.

Im siebten Kapitel des Buches geht der Autor auf die Suchfunktion und die Nutzung einer Datenbank mit TWiki ein. Bei der Suche - damit ist die `SEARCH`-Funktion und nicht die Suche über das Suchfeld gemeint - zeigt Marbach, wie komplexe Suchen mit einer SQL-ähnlichen Syntax mit TWiki realisiert werden.

TWiki benötigt grundsätzlich keine Datenbank, da alle Funktionalitäten Dateibasiert sind. Allerdings kann man eine Datenbank anbinden und über sogenannte *Forms* Eingabemasken für die Datenbank erstellen. Dieses Thema hätte der Autor noch etwas ausführlicher beschreiben können.

Für viele Installationen eines Wikis müssen eigene Plugins geschrieben werden, weil die gewünschte Funktionalität noch nicht vorhanden ist. Marbach zeigt im neunten Kapitel, wie man eigene Plugins für TWiki erstellt.

Fazit

Insgesamt ein ganz gutes Buch. An manchen Stellen hätte ich mir die Erläuterungen etwas ausführlicher gewünscht, aber das Buch liefert einen guten Überblick. Außerdem hat das Buch zwei Pluspunkte: Erstens ist eine CD dabei, auf der ein fertiges TWiki mit VMware Player zu finden ist und zweitens ist es ein Buch für zwei Systeme. Zum einen für – wie der Titel es schon andeutet – TWiki und zum anderen für foswiki, einem Fork von TWiki, das ungefähr zwei Wochen bevor das Buch erschienen ist, entstand.

Dank dem TWiki auf der CD kann man viele Sachen auch gleich ausprobieren – auch die Sachen, die vielleicht etwas ausführlicher hätten beschrieben werden können.

Das Buch richtet sich sowohl an Administratoren als auch an Anwender. Allerdings ist es eher weniger für den einfachen Anwender, der hin und wieder mal einen Artikel schreibt oder bearbeitet, geeignet. Das Buch behandelt die tiefer gehenden Themen wie das Einrichten von Webs oder Plugins.

Renée Bäcker

TWiki - installieren, konfigurieren, administrieren
Wolf Marbach
C&L Computer und Literatur
ISBN 978-3936546

App::Asciio - ASCII-Art mit Perl

Das Dokumentieren von Code macht keinen Spaß und wenn in der Pod-Dokumentation auch noch "Zeichnungen" vorkommen sollen, die das ganze System näher erläutern, dann wird es sehr zeitaufwändig.

Gerade die Zeichnungen sind Zeitfresser und wenn sich etwas am System ändert, bedeutet das viel Arbeit um die ASCII-Zeichnung anzupassen.

Diese Arbeit kann jetzt stark vereinfacht werden. Nadim Khemir hat mit `App::Asciio` eine Anwendung auf CPAN gestellt, die genau darauf abzielt: ASCII-Bilder erstellen.

Der Name Asciio erinnert an Visio, das Microsoft-Programm, mit dem Netzwerkpläne und ähnliches erstellt werden.

Die Anwendung

Wenn die Anwendung gestartet wird, erscheint ein einzelnes Fenster. Für viele Benutzer ist die Bedienung etwas gewöhnungsbedürftig, da es kein Menü gibt. Alles geht über Kontextmenü und/oder Tastatur-Shortcuts.

Um sich einen ersten Überblick zu verschaffen, lohnt es sich, die Hilfe – mittels `F1` – aufzurufen. Dort sind dann die wichtigsten Tastaturkürzel zu finden. Mit den in der Hilfe genannten Shortcuts lassen sich schon viele Diagramme erstellen. Wem das aber nicht genug ist, kann mit `k` das Keymapping aufrufen. Hier zeigt sich eine Schwäche von `Asciio`: Die Oberflächen sind häufig nicht sehr Benutzerfreundlich. Aber man findet alle Keymappings von Asciio. Da es keine Menüs gibt, gewöhnt man sich sehr schnell die Tastenkombinationen an. Mir haben einige Kürzel nicht gefallen, also habe ich sie abgeändert. Damit man weiß, wo man etwas ändern muss, kann man sich in den Keymappings auch die Datei anzeigen lassen, in der das Kürzel definiert wird.

Den umgekehrten Weg kann man mit `Ctrl+Shift+k` gehen. In dem Fenster sieht man alle Dateien, in denen Shortcuts definiert werden und man kann sich alle Shortcuts in den einzelnen Dateien anzeigen lassen.

Doch zurück zur Anwendung. Nachdem Asciio gestartet wurde, ist ein leeres Fenster mit dem Gitternetz zu sehen. Die vertikale Linie am rechten Rand markiert die 80-Zeichen-Grenze, da einige Programmierer sich immer noch streng an diese Grenze halten.

Ich starte jetzt mit ein paar einfachen Formen, die in meinen Diagrammen am häufigsten auftauchen: Boxen (Shortcut: `b`), Pfeile (`a`) und Texte (`t`). Mit `b` füge ich eine Box ein. Die Box ist jetzt markiert. Komme ich mit der Maus in die Nähe der Box, erscheinen an vier Stellen "Striche" und die Zelle in der rechten unteren Ecke ist hervorgehoben. Diese Ecke ist der so genannte "Resizehandle". Fasst man die Box hier an, kann man die Größe des Elements verändern. Die vier "Striche" sind die so genannten "Connectoren". In Asciio ist es möglich, Pfeile und andere "Verbinder" an Formen anzudocken.

Bei einem Doppelklick auf die Box sieht man das Attributfenster für die Box. Hier kann man die Box beschriften – dabei auch einen Titel setzen – und bestimmen, ob bestimmte Bereiche des Rahmens ausgeblendet werden sollen.

Nimmt man den Rahmen an allen Seiten weg, erhält man das Textelement.

Ein Diagramm besteht aber nur selten aus einem einzigen Objekt (Form). Sind jetzt zwei Objekte so angeordnet, dass das eine Objekt das Andere (teilweise) bedeckt, kann man

wie in den meisten Zeichenprogrammen die Objekte unterschiedlich anordnen. Mit `Strg+b` landet das markierte Objekt im Hintergrund, mit `Strg+f` im Vordergrund. Diese Aktionen können leider nicht mit dem Kontextmenü gestartet werden, so dass man die Shortcuts kennen muss.

Ein weiteres wichtiges Tastaturkürzel ist `Strg+g`, mit dem verschiedene Formen gruppieren kann. Dazu einfach die zu gruppierenden Formen auswählen und mit `Strg+g` gruppieren. Die Auswahl kann auf mehreren Wegen erfolgen:

1. Linke Maustaste drücken und mit dem Auswahlrahmen die auszuwählenden Formen einschließen
2. Oder die erste Form mit einem Mausklick auswählen und alle weiteren Formen mit gedrückter `Strg`-Taste anklicken

Sind die Formen gruppiert, erhalten sie eine andere Farbe wenn sie markiert werden. Insgesamt sind vier Farben definiert, die als Markierungsfarbe dienen. So kann man verschiedene Gruppierungen unterscheiden.

Ein ganz nützliches Feature bei Asciio ist das schnelle kopieren von (mehreren) Formen. Dazu einfach die zu kopierenden Formen markieren, die `Strg`-Taste drücken und dann in einer markierten Form die linke Maustaste drücken und zu der gewünschten Stelle ziehen. Dort die Maustaste loslassen und schon sind die Formen kopiert.

Ich stelle in einigen Diagrammen Workflows dar, so dass Boxen und Pfeile häufig eingesetzte Formen sind. In Asciio kann man die Pfeile direkt an Formen hängen (an die oben beschriebenen "Connectoren"). Damit man die Pfeile nicht bei jeder Änderung anpassen muss, passen sie sich automatisch an. Dahinter steckt ein "Optimierer". Dieser sorgt dafür, dass die Verbindung immer zwischen den "besten Connectoren" hergestellt wird. In manchen Fällen arbeitet der Optimierer nicht optimal, so dass z.B. die Pfeilspitze nicht in die gewünschte Richtung zeigt (siehe Abbildung 2). Dann kann man den Optimierer anpassen, der in `App/Asciio/setup/hooks/canonize_connections.pl` steckt.

Sollen mehrere Boxen in das Diagramm eingefügt werden, die alle sehr ähnlich sind, dann kann man das in einem Schritt erledigen. Mit `Strg+m` erscheint ein Dialog, in dem man die Textinhalte für die Boxen festlegen kann. Einfach die Standardtexte (die Buchstaben 'A' bis 'E') durch die eigenen Texte ersetzen.

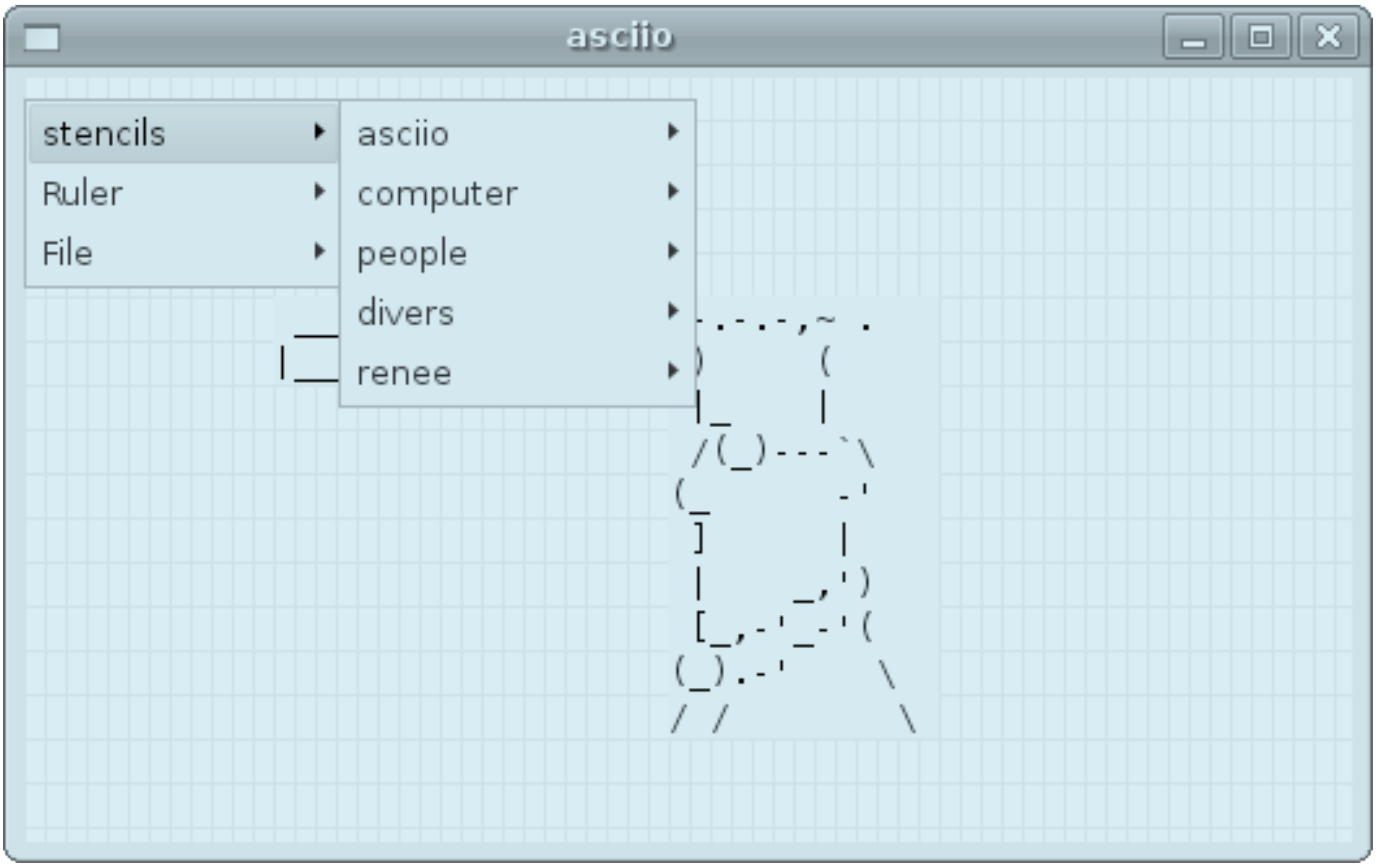


Abb. 1: ASCIIO Hauptfenster

```
$VAR1 = [
        bless( {
                 'HEIGHT' => 1,
                 'TEXT' => '0123456789',
                 'NAME' => 'rulers/0_to_9_horizontal',
                 'WIDTH' => 10,
                 X_OFFSET => 0, Y_OFFSET => 0,
];
```

Listing 1

fertige Formen

Asciio kommt schon mit einigen vorgefertigten Formen, den so genannten `stencils`. Diese sind über das Kontextmenü zu erreichen. In der Standardversion sind dort vier Gruppen zu finden:

- asciio
- computer
- people
- divers

Da wir uns im IT-Umfeld bewegen, dürfte die am häufigsten genutzte Gruppe *computer* sein. Dort sind dann Formen wie die "Internet-Wolke" oder der "Mainframe" zu finden (Abbildung 3).

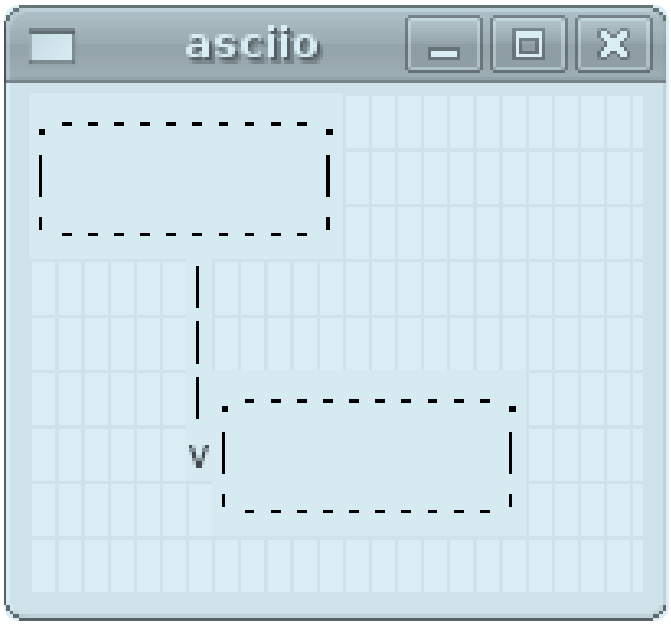


Abb. 2

Eigene Formen

In vielen Projekten werden immer wieder die gleichen Formen benötigt und diese immer zu kopieren, ist zu umständlich. Deshalb kann man eigene Formen bzw. `stencils` erstellen.

Die vorhandenen `stencils` sind unter App/Asciio/setup/stencils zu finden. Dort kann man sich auch anschauen, wie eigene Formen aufgebaut werden müssen. Pro Datei, die in dem Ordner existiert, wird ein Submenü erzeugt. In diesen Dateien ist ein Dump der Objekte (siehe Listing 1), die die Formen darstellen, gespeichert.

Am einfachsten ist es natürlich, eine vorhandene Datei zu kopieren und den Inhalt anzupassen.

Soll eine eigene Form auf einer bestehenden Form basieren, kann man auch wie folgt vorgehen: Bestehende Form zeichnen, an eigene Wünsche anpassen - z.B. Text in einer Box eingeben -, Form markieren und dann im Kontextmenü unter `File -> save  stencil` die Form speichern. Jetzt muss die Datei noch in `App/Asciio/setup/setup.ini` unter `STENCILS` eingetragen werden. Ab diesem Zeitpunkt

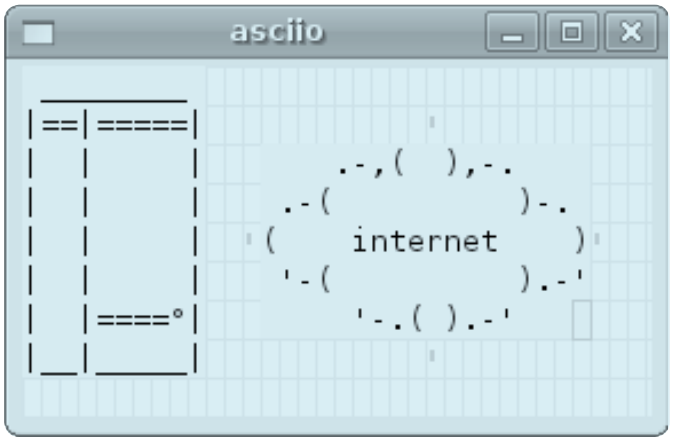


Abb. 3

– wenn Asciio neu gestartet wurde – steht die eigene Form zur Verfügung.

Leider kann man so nur einzelne Formen und keine Gruppierungen erzeugen.

Typen von Formen

Die Formen sind jeweils einem Typ zugeordnet, die in Asciio `element` genannt werden. In der aktuellen Version sind nur ein paar Elemente implementiert. Das sind zur Zeit

- wirl_array
- text und box
- stripes

Der Typ einer Form bestimmt, wie die Form auf das Markieren oder eine Verbindung "reagiert".

Für die Anwendung an sich muss man sich nicht mit den Elementtypen beschäftigen. Erst wenn man eigene Formen erstellt, die nicht in das existierende Schema passen, muss man sich damit auseinandersetzen wie der allgemeine Typ der Form aussehen soll.

Eigene Typen

Wie bei der Erstellung von eigenen Formen muss man sich bei der Erstellung von eigenen Typen an den bestehenden Typen orientieren. Da an keiner Stelle im Asciio-Code beschrieben ist, wie eigene Typ-Klassen auszusehen haben, muss man vieles ausprobieren.

Die mitgelieferten Typen sind unter `App/Asciio/stripes/` finden.

ASCII-Art exportieren

Die Zeichnung kann über das Kontextmenü -> "Files" -> "save as" als ASCII-Art gespeichert werden. Dazu einfach die Endung .txt verwenden.

Eine weitere Möglichkeit ist das Kopieren in die Zwischenablage. Dazu muss dann aber das Tastaturkürzel `Ctrl+e` statt des sonst üblichen `Ctrl+c` verwendet werden.

Speichert man die Zeichnung in einer Datei, die nicht auf `.txt` endet, so wird die Datei im Asciio-eigenen Format gespeichert, womit man keine Zeichnung bekommt, aber damit wird das nachträgliche Bearbeiten möglich.

Soll die aktuelle Ansicht als Bild exportiert werden, so kann man die Dateiendung `.png` verwenden. Dabei ist zu beachten, dass hierbei nur ein Screenshot erzeugt wird, also das Gitternetz zu sehen ist und alle Elemente, die außerhalb des sichtbaren Bereichs liegen nicht zu sehen sind.

ASCII-Art importieren

Leider ist es nicht möglich, ASCII-Art zu importieren, da das Parsen und "aufteilen" in Objekte nahezu unmöglich ist. Man kann nur Asciio-eigene Dateien öffnen.

Asciio skripten

Asciio bietet die Möglichkeit, mit einem eigenen Perl-Skript die Funktionalitäten von Asciio zu nutzen. Nehmen wir als Beispiel mal an, Daten für ein Diagramm kommen als XML-Daten an. In der XML-Datei befinden sich Informationen über die Texte und die Verbindungen von Boxen:

```
<diagram>
  <boxes>
    <box>
      <boxid>1</boxid>
      <title>Leser</title>
    </box>
    <box>
      <boxid>2</boxid>
      <title>foo</title>
    </box>
  </boxes>
  <connections>
    <connection>
      <source>2</source>
      <target>1</target>
    </connection>
  </connections>
</diagram>
```

Dann kann mit folgendem Perl-Skript ein Diagramm erzeugt werden (siehe Listing 2).

```
#!/usr/bin/perl

use strict;
use warnings;
use XML::Simple;
use App::Asciio;
use Getopt::Long;
use Data::Dumper;

use scripting_lib;

GetOptions(
  'xml=s'   => \my $xml_file,
  'setup=s' => \my $setup_path,
);

@ARGV = ('-setup_path', $setup_path, @ARGV);

my $asciio = App::Asciio->new;
my ($parse_ok, $parse_message, $asciio_config)
                    = $asciio->ParseSwitches([@ARGV], 0) ;

$asciio->setup( $asciio_config->{SETUP_INI_FILE}, $asciio_config->{SETUP_PATH}) ;

draw( $asciio, $xml_file );

sub draw {
    my ($asciio, $file) = @_;

    my $xml_ref = XMLin( $file, force_array => ['box','connection'] );
    my %boxes   = draw_boxes( $asciio, $xml_ref->{boxes} );
    draw_connections( $asciio, $xml_ref->{connections}, \%boxes );

    print  $asciio->transform_elements_to_ascii_buffer;
}

sub draw_boxes {
    my ($asciio, $boxes_ref) = @_;

    my %boxes;
    my $x = 0;
    my $y = 2;

    for my $definition( @{ $boxes_ref->{box} } ){
        my $id      = $definition->{boxid};
        my $box     = new_box( TEXT_ONLY => $definition->{title} );
        $boxes{$id} = $box;

        $asciio->add_element_at( $box, $x, $y );

        $x += 10;
        $y += 5;
    }

    %boxes;

}

sub draw_connections {
    my ($ascioo, $conn_ref, $boxes) = @_;

    for my $definition( @{ $conn_ref->{connection} } ){
        my $source = $definition->{source};
        my $target = $definition->{target};
        my $box1   = $boxes->{$source};
        my $box2   = $boxes->{$target};
        add_connection($asciio, $box1, $box2,) ;
    }

    optimize_connections($asciio) ;
```

Listing 2

Damit das ganze lauffähig ist, muss das Modul C<scripting_lib> direkt von CPAN kopiert werden, da das Modul im Installationsprozess "verloren" geht.

Wird das Modul ausgeführt, erhält man folgendes Ergebnis:

```
.-------.
| Leser |
'-------'
    ^
    |
    |     .-----.
    '-----| foo |
          '-----'
```

Installation

Die Anwendung benutzt das GTk-Framework. Auf Windows ist es äußerst schwierig, AsciiO zu installieren. Wer einen Windows-Rechner ohne Perl hat, kann darauf camelbox installieren. camelbox ist eine Perl-Distribution für Windows, die Gtk mitbringt. Ein Nachteil von camelbox ist es allerdings, dass es kein anderes Windows-Perl auf der gleichen Maschine erlaubt.

Auf *NIX-Rechnern gibt es meist schon vorbereitete Pakete, so dass man mittels `apt-get install libgtk2-perl` oder ähnliches schnell Gtk installiert hat. Dann ist auch die Installation von `App::Asciio` mit `CPAN.pm` kein Problem mehr.

Renée Bäcker

Time::y2038 - Keine Angst vor dem Schwarzen Dienstag 2038

Das *Jahr-2038-Problem* kann möglicherweise zu Softwareausfällen im Februar 2038 auf 32-Bit-Systemen führen.

Michael Schwern's `Time::y2038` schafft auf besonders einfache Art und Weise Abhilfe für Legacy Cody.

Für Windows steht `Time::y2038` leider noch nicht zur Verfügung.

Beispiel mit 2038-Bug:

```
#!/usr/bin/perl
use strict;
use warnings;
#
# Based on http://maul.deepsky.com/
#           ~merovech/2038.perl.txt
#

# Zeitzone auf GMT aka UTC setzen
$ENV{'TZ'} = "GMT";

# Epochsekunden kurz vor bzw. nach dem
# kritischen Event
# in einer Schleife durchlaufen und pruefen,
# ob Datum
# und Uhrzeit korrekt ausgegeben werden.
my $clock;
for ($clock = 2147483641;
     $clock < 2147483651; $clock++) {
      print scalar localtime($clock), "\n";
}
```

Ausgabe auf einem 32-Bit Linux:

```
Tue Jan 19 03:14:01 2038
Tue Jan 19 03:14:02 2038
Tue Jan 19 03:14:03 2038
Tue Jan 19 03:14:04 2038
Tue Jan 19 03:14:05 2038
Tue Jan 19 03:14:06 2038
Tue Jan 19 03:14:07 2038
Fri Dec 13 20:45:52 1901
Fri Dec 13 20:45:53 1901
Fri Dec 13 20:45:54 1901
```

Auf 32-Bit Unix/Linux-Systemen wird die Zeit auf den 13. Dezember 1901 gesetzt.

Ausgabe auf einem 32-Bit Windows:

```
Tue Jan 19 04:14:01 2038
Tue Jan 19 04:14:02 2038
Tue Jan 19 04:14:03 2038
Tue Jan 19 04:14:04 2038
Tue Jan 19 04:14:05 2038
Tue Jan 19 04:14:06 2038
Tue Jan 19 04:14:07 2038
```

Das Programm stoppt genau nach der kritischen Sekunde im Jahr 2038.

Beispiel mit Time::y2038

Dank `Time::y2038` muss nur *eine* Zeile Code hinzugefügt werden und das Programm ist Jahr-2038-fest.

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

use Time::y2038;
#
# Based on http://maul.deepsky.com/
#            ~merovech/2038.perl.txt
#

# Zeitzone auf GMT aka UTC setzen
$ENV{'TZ'} = "GMT";

# Epochsekunden kurz vor bzw. nach dem
# kritischen Event
# in einer Schleife durchlaufen und pruefen,
# ob Datum und Uhrzeit korrekt ausgegeben
# werden.
my $clock;
for ($clock = 2147483641;
     $clock < 2147483651; $clock++) {
      print scalar localtime($clock), "\n";
}
```

Ausgabe des Programms (32-Bit Linux)

```
Tue Jan 19 03:14:01 2038
Tue Jan 19 03:14:02 2038
Tue Jan 19 03:14:03 2038
Tue Jan 19 03:14:04 2038
Tue Jan 19 03:14:05 2038
Tue Jan 19 03:14:06 2038
Tue Jan 19 03:14:07 2038
Tue Jan 19 03:14:08 2038
Tue Jan 19 03:14:09 2038
Tue Jan 19 03:14:10 2038
```

Datumsberechnungen (nicht nur über das Jahr 2038 hinaus)

Wer Datumsberechnungen (nicht nur über das Jahr 2038 hinaus) sicher durchführen möchte, sollte sich ohnehin nicht auf `time()` und *Epochen-Sekunden* verlassen, sondern besser ein CPAN-Modul wie `Date::Calc` oder `DateTime` verwenden.

Beispiel:

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

use DateTime;

my $dt = DateTime->now();

print $dt->iso8601() , "\n";

$dt->add(years => 200);

print $dt->iso8601() , "\n";
```

Thomas Fahle

Win32 Tipps & Tricks

Da es genügend Dinge gibt, die man als Windows-Anwender und Perl-Programmierer wissen beachten muss, habe ich mich entschlossen die Tipps & Tricks der letzten $foo-Ausgabe in einer extra Mini-Serie fortzusetzen - den "Win32 Tipps & Tricks". In dieser und den nächsten zwei Ausgaben werde ich jeweils drei Tipps & Tricks zeigen. Wenn es darüber hinaus noch Anregungen und/oder Fragen zu "Windows und Perl" gibt, dann einfach an feedback@foo-magazin.de schreiben.

Erstellen von Verzeichnissen mit "non-ASCII"-Zeichen

Nicht in jedem Land oder für jedes Projekt kommt man mit ASCII-Zeichen aus. Dann steht man teilweise vor dem Problem, dass Verzeichnisse mit "non-ASCII"-Zeichen erstellt werden müssen. So ganz einfach ist das nicht. Man kann zwar mit C<mkdir> arbeiten, aber dann wird ein Verzeichnis angelegt, das die "non-ASCII"-Zeichen in einer anderen Kodierung betrachtet und "Zeichensalat" produziert. Will man die korrekten Zeichen haben, dann muss man `CreateDirectory` aus dem Modul `Win32` nehmen.

```
#!/usr/bin/perl

use strict;
use warnings;
use Win32;

my $dirname = "TestDir" . chr(0x010b);

Win32::CreateDirectory( $dirname );
```

Jetzt muss man nur noch aufpassen wenn man den Verzeichnisnamen weiterhin verwenden will (z.B. um eine Datei zu erzeugen). Dann muss man auf weitere Funktionen aus `Win32` benutzen (z.B. `CreateFileW`), die mit diesen Sonderzeichen umgehen können.

Microsoft Access + DBI bei geschützter Datenbank

Access ist in vielen Unternehmen eine beliebte Datenbank-Software. In einigen Fällen ist diese sogar geschützt um für verschiedene Nutzer unterschiedliche Zugriffsrechte einzurichten. Dann braucht man für den Zugriff auf die Datenbank die `.mdw`-Datei (mehr zum Zugriffsschutz bei Access unter http://www.marktscheffel.de/access/accesszugriffsschutz.htm).

Diese `.mdw`-Datei muss dann schon beim Verbinden zur Datenbank mit angegeben werden:

```
my $db    = 'DBName';
my $user  = 'username';
my $pass  = 'password';
my $mdw   = 'C:\Pfad\zur\Datei.mdw';

my $dsn   =
'driver=Microsoft Access-Driver (*.mdb); ';
   $dsn  .= "dbq=$db; SystemDB=$mdw";

my $dbh   = DBI->connect(
    "DBI:ODBC:$dsn",
    $user,
    $pass,
);
```

Wo man noch aufpassen muss ist die Angabe des Treibers. Der Beispielcode kann auf einem Windowssystem eingesetzt werden, bei dem die Treiber einen englischen Namen haben. Bei einem meiner Testsysteme musste ich dagegen `Microsoft Access-Treiber (*.mdb)` angeben, weil der Treiber den deutschen Namen hatte.

Herausfinden, welches Dateisystemformat verwendet wurde

Unter Windows kann man mit verschiedenen Dateisystemen arbeiten: FAT und NTFS. Beide Formate haben ihre Eigenheiten und manches funktioniert bei dem Einen, aber nicht bei dem Anderen. Deshalb kann es ganz praktisch sein wenn man weiß, welches Format aktuell genutzt wird. Dies kann man recht einfach mit dem Modul `Win32`:

```
#!/usr/bin/perl

use strict;
use warnings;
use Win32;

my $fs = Win32::FsType();
print $fs;
```

Renée Bäcker

"Merkwürdigkeiten" - eval und $@

In Ausgabe 9 von $foo habe ich unter "Tipps & Tricks" gezeigt, wie man mit `eval` ein "try-catch" nachbauen kann. Kurz danach, habe ich eine Mail bekommen, dass das nicht gehen würde. Der Code, um den es dabei ging, war folgender:

```
use strict;
use warnings;

eval {
    my $obj = Meine::Klasse->new();
    die "Ein Fehler!\n";
};

if($@) {
    print "Exception caught: $@\n";
}
else {
    print "No exception occurred\n"
}
```

Auf den ersten Blick ist alles klar, oder? Das gibt ganz eindeutig *Exception caught: Ein Fehler!* aus. Das wurde aber nicht ausgegeben, sondern *No exception occured*. Was war passiert? Ein Bug in Perl? Merkwürdig, oder?

Bei der Suche nach dem Fehler zeigte sich schnell, dass das kein Bug in Perl ist, sondern wieder einer der Fallen, mit denen man sich selbst in den Fuß schießen kann. Der Code von `Meine::Klasse` sieht (beispielhaft) so aus:

```
package Local::Class;

use strict;
use warnings;

sub new { bless {}, shift }

sub DESTROY {
    eval{
        my $a = 2;
        # Code zum aufraeumen
    };
}

1;
```

Und Fehler erkannt? Das Problem ist, dass Perls *Garbage Collection* zuschlägt und der Destruktor der Klasse ausgeführt wird. Der Destruktor ist die Subroutine `DESTROY`. Diese wird von Perl automatisch ausgeführt, wenn ein Objekt zerstört wird. Mehr zum Thema "Garbage Collection in Perl" wird es in der nächsten Ausgabe von $foo geben.

Der Ablauf ist somit folgender:
Perl "betritt" den `eval`-Block im Skript. Darin wird ein Objekt der Klasse `Meine::Klasse` erzeugt. Als nächstes wird das `die` ausgeführt und die (globale) Variable `$@` wird gesetzt. Als nächstes wird der `eval`-Block verlassen und Perls Garbage Collection wird angeschmissen. Dabei wird das Objekt von `Meine::Klasse` zerstört, wodurch der Destruktor der Klasse aufgerufen wird. Darin kommt wieder ein C<eval>-Block vor, der aber problemlos durchläuft - also wird die (globale) Variable `$@` auf *undef* gesetzt.

Jetzt kommt der *if-else*-Teil im Hauptskript. Durch den Destruktor ist ja `$@` auf *undef* gesetzt worden, also wird der `else`-Teil ausgeführt.

Wie kann man das Problem lösen?
Sehr einfach: Man muss in dem Destruktor die (globale) Variable `$@` lokalisieren:

```
sub DESTROY {
    local $@;
    eval{
        my $a = 2;
        # Code zum aufraeumen
    };
}
```

Damit gelten die Änderungen, die innerhalb von `DESTROY` an `$@` gemacht werden nur innerhalb dieser Subroutine und den Subroutinen, die in `DESTROY` aufgerufen werden (siehe auch `perldoc -f local`). Verlässt Perl `DESTROY`, wird der Wert von `$@` wieder auf den Wert gesetzt, den die Variable vor der Lokalisierung hatte.

Jetzt wird auch wirklich *Exception caught: Ein Fehler!* ausgegeben.

CPAN News XI

Tree::Family

Ahnenforschung für Perl-Programmierer: Mit `Tree::Family` kann man sehr simpel Familienstammbäume erstellen.

```
#!/usr/bin/perl

use strict;
use warnings;
use Tree::Family;

my $tree = Tree::Family->new(
              filename => './tree.dmp');

my $person = Tree::Family::Person->new(
              name => 'Fred');
my $nother = Tree::Family::Person->new(
              name => 'Wilma');

$person->spouse($nother);

$tree->add_person($person);
$tree->add_person($nother);

for ($tree->people) {
    print $_->name;
}

my $dot_file = $tree->as_dot;
```

Directory::Deploy

Mit `Directory::Deploy` ist es möglich, sehr einfach eine Verzeichnisstruktur auf der Festplatte zu erzeugen. Dabei kann man auch die Rechte setzen und Dateien mit Inhalt füllen. Angaben mit angehängtem Slash bedeuten Verzeichnisse, der Rest wird als Datei behandelt.

```
#!/usr/bin/perl

use strict;
use warnings;

{
    package MyDeploy;
    use strict;
    use warnings;
    use Directory::Deploy::Declare;

    # create directory 'foo' with
    # basic article

    include 'foo/';
    include 'foo/article.pod' => \<<'END';
=head1 ARTIKEL-Titel

=head2 Ueber den Autor
END
    no Directory::Deploy::Declare;
}

my $dpl = MyDeploy->new( base => '.' );
$dpl->deploy;
```

Config::Perl::V

Zwei Rechner, ein Programm und auf einem Rechner läuft das Programm und auf dem Anderen nicht. Wo liegt das Problem? Vielleicht an der Perl-Installation. Wenn man die Konfiguration zweier Perl-Installationen vergleichen will, kann das mit `perl -V` machen oder das Modul `Config::Perl::V`.

```
#!/usr/bin/perl

use strict;
use warnings;
use Config::Perl::V;
use Data::Dumper;

print Dumper Config::Perl::V::myconfig();
```

Text::CSV::Encoded

Ich will den Inhalt von einem Formular in eine CSV Datei schreiben. Wenn dort allerdings ein Umlaut vorhanden ist bekomme ich folgende Fehlermeldung:

```
"combine () failed on argument: ..."
```

In so einem Fall, sollte man sich `Text::CSV::Encoded` anschauen, da man damit CSV-Dateien mit verschiedenen Encodings handlen kann.

```
use Text::CSV::Encoded;

my $csv = Text::CSV::Encoded->new({
    sep_char => ';',
    encoding_in => 'utf-8',
});
my @fields = ( 'äää', 'test' );
$csv->combine( @fields );
print $csv->string,"\n",
     $csv->error_input;
```

Test::NoPlan

Der "plan" bei Tests - also die Anzahl der Tests - ist ein eigener Test. Diese Prüfung kann man aber umgehen, indem man z.B. `use Test::More 'no_plan` schreibt. Um in der Test-Suite so etwas zu vermeiden, kann man `Test::NoPlan` verwenden.

```
#!/usr/bin/perl

use strict;
use warnings;
use Test::NoPlan qw(all_plans_ok);
use Test::More tests => 1;

all_plans_ok();
```

DBIx::Publish

Einen Teil einer beliebigen Datenbank in einer SQLite-Datenbank speichern, das wäre toll. Dann könnte man diese Daten an einen Kollegen geben. Das kann man mit `DBIx::Publish` machen. Die Anwendung des Moduls ist ziemlich einfach (siehe Quellcode).
Das eignet sich auch gut, wenn man nur bestimmte Daten einer Umfrage oder von Statistiken an einen Kunden weitergeben möchte.

```
my $publish = DBIx::Publish->new(
    file   => 'publish.sqlite',
    source => DBI->connect($dsn, $user,
$pass),
);

$publish->table( 'table1',
    'select * from foo where this < 10',
);

$publish->finish;
```

Neues von TPF

Grants im 2. Quartal 2009

In der abgelaufenen Abstimmungsphase hat das Grants Committee der Perl Foundation über vier Grant-Anträge abzustimmen gehabt. Die vier Anträge sind auch im Blog der Perl Foundation zu finden. Am Ende wurde einem Grant-Antrag zugestimmt: *Pod Mangling Utility Improvements von Ricardo Signes*

Hague-Grant für Jonathan Worthington genehmigt

Die Perl Foundation hat einen weiteren Grant für Jonathan Worthington genehmigt. Der Grant läuft unter dem Titel "Traits, Introspection and More Dispatch Work For Rakudo.".

Jonathan Worthingtons Hague-Grant abgeschlossen

Jonathan Worthington hat seinen (ersten) Hague-Grant abgeschlossen. Bei diesem Grant ging es darum, einige OO-Features zu implementieren.

In seinem abschließenden Bericht zeigt Worthington auf, was er alles erreicht hat und dass der Grant noch weitere positive Effekte rund um Rakudo hat.

Die "Ian Hague"-Grants werden für Projekte im Bereich Rakudo und anderen Perl 6 Implementierungen vergeben. Bezahlt werden die Grants aus einer großen Spende von Ian Hague aus dem Jahr 2008. Um einen solchen Grant kann sich jeder bewerben.

9 Projekte für die TPF beim "Google Summer of Code"

Auch in diesem Jahr ist die Perl Foundation wieder beim "Google Summer of Code" dabei. Als Organistor auf Seiten der Perl Foundation arbeitet Jonathan Leto.

Mit 9 Projekten ist die Perl Foundation auch stärker vertreten als 2008. Themen in den Projekten sind Perl 5, Perl 6 und Parrot.

Updates zu laufenden Grants

Port PyYAML to Perl

YAML-Perl macht gute Fortschritte und Ingy hat die Version 0.02 auf CPAN und github geladen. Unter http://tinyurl.com/co6h6h sind Details zum aktuellen Stand und zur TODO-Liste zu finden. Alle Klassen wurden soweit portiert, es fehlen nur noch ein paar Methoden.

Ingy hat auf dem "Nordic Perl-Workshop" in Oslo einen Vortrag gehalten (die Folien werden demnächst unter http://ingydotnet.github.com/yaml-npw2009-talk/ zu finden sein).

Auf dem anschließenden Hackathon will er PyYaml nach Rakudo portieren.

Fixing Bugs in the Archive::Zip Perl Module

Alan hat die Unterstützung für Unicode-Dateinamen in allen Methoden integriert

Extending BSDPAN

Grant wurde wegen fehlender Reaktionen von Colin abgebrochen

Mojo Documentation Project
Sebastian arbeitet nach einer längeren Auszeit am ersten Kapitel, das in Kürze veröffentlicht werden soll.

Moose docs
Dave hat den Grant abgeschlossen und einen Abschlussbericht in seinem Blog gepostet

Improving learn.perl.org - Eric und Tina haben unter http://learnperl.scratchcomputing.com/ die Seite aufgebaut. Noch ist sie quasi eine 1:1-Kopie von http://learn.perl.org. Das wird sich in den nächsten Tagen ändern.

A lightweight web framework for Perl 6 - Ilya, Carl und Stephen haben eine Reihe von Blogeinträgen über ihren Fortschritt veröffentlicht.
- Ilya: http://perl6.ru
- Stephen: http://blogs.gurulabs.com/stephen/
- Carl: http://use.perl.org/~masak/journal/

Integrating Padre with Parrot and Rakudo
Grant durch Gabor abgebrochen. Über die Gründe schreibt er in seinem Blog.

Alberto Simões von der Perl Foundation hebt jedoch hervor, dass das nicht bedeutet, dass die Integration gar nicht gemacht wird oder die Perl Foundation das nicht unterstützt. Im Moment bedeutet das nur, dass Gabor den Grant nicht planmäßig abschließen kann.

Test::Builder 2
Michael hat auf dem QA-Hackathon in Birmingham an Test::Builder 2 gearbeitet. Mit Hilfe von Colin Newell hat er an der Ergebnis- und Ausgabe-Architektur gearbeitet. Mehr über diese Arbeiten gibt es unter http://colinnewell.wordpress.com/2009/03/31/perl-qa-hackathon-testbuilder2/

Embedding perl into C++ applications
Leon hat den Code von Google zu github transferiert. Weiterhin hat er an den Regulären Ausdrücken weitergearbeitet. Die Folien seines Vortrags auf dem Niederländischen Perl-Workshop sind ebenfalls online.

Die Links zu den einzelnen News sind unter *http://delicious.com/foo_magazin* zu finden

Perlcast.com

Neues zu Perl::Critic

Jeff Thalhammer erzählt beim Perlcast Neuigkeiten über `Perl::Critic`. Dabei geht er auf verschiedene Grants ein, die in den letzten beiden Jahren Perl::Critic geholfen haben. Weitere Themen in dem Interview sind die Perl::Critic-Erweiterungen `Perl::Critic::Dynamic` und `Test::Perl::Critic::Progressive`. Mit der "Dynamic"-Erweiterung wird der Perl-Code nicht statisch mit PPI ausgewertet, sondern der Code wird kompiliert und dynamische Regeln werden angewendet.

Mit dem Test-Modul kann man eine Umstellung von altem Code erreichen, da es beim ersten Lauf immer erfolgreich ist und bei jedem weiteren Testlauf muss die Anzahl der "Verstöße" reduziert werden, sonst schlägt der Test fehl.

Termine

August 2009

03.-05. YAPC::EU 2009
03. Treffen Vienna.pm
04. Treffen Frankfurt.pm
06. Treffen Dresden.pm
10. Treffen Ruhr.pm
12. Treffen Hamburg.pm
13. Treffen Cologne.pm
17. Treffen Erlangen.pm
25. Treffen Bielefeld.pm
26. Treffen Berlin.pm

September 2009

07. Treffen Vienna.pm
08. Treffen Frankfurt.pm
10./11. YAPC::Asia in Tokio
14. Treffen Ruhr.pm
16. Treffen Darmstadt.pm
Treffen München.pm
21. Treffen Erlangen.pm
29. Treffen Bielefeld.pm
30. Treffen Berlin.pm

Oktober 2009

01. Treffen Dresden.pm
05. Treffen Vienna.pm
06. Treffen Frankfurt.pm
09-11. WxPerl Workshop
12. Treffen Ruhr.pm
14. Treffen Hamburg.pm
15. Treffen Cologne.pm
19. Treffen Erlangen.pm
21. Treffen Darmstadt.pm
22./23. Italienischer Perl-Workshop in Pisa
27. Treffen Bielefeld.pm
28. Treffen Berlin.pm

Hier finden Sie alle Termine rund um Perl.

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

Uhrzeiten und weiterführende Links zu den einzelnen Veranstaltungen finden Sie unter

http://www.perlmongers.de

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

termine@foo-magazin.de

LINKS

http://www.perl-nachrichten.de

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

http://www.perl-community.de

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

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

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

http://www.perl-workshop.de

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

http://www.perl-foundation.org

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

http://www.Perl.org

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

FrOSCon

Free and Open Source Software Conference - 2009

22.-23. August 2009
Bonn - St. Augustin

- Über 20 Projekte und Aussteller
- Über 60 Vorträge in 5 Hörsälen
- Java-Subkonferenz
- LPI Prüfung
- Hüpfburg

Call for Papers bis 23.5.09
Call for Projects bis 23.6.09

kontakt@froscon.de - www.froscon.de

Hochschule Bonn-Rhein-Sieg, Grantham-Allee 20, 53757 Sankt Augustin

// SEIBERT / MEDIA

BESSERE ATMOSPHÄRE? MEHR FREIRAUM?

Wir suchen erfahrene Perl-Programmierer/innen (Vollzeit)

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

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

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

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

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