Ausgabe 4/2007 – www.foo-magazin.de – 6,00 €

$foo

PERL MAGAZIN

Moose
Objektorientierung mit Moose in Perl

Caching in Webapplikationen
Das merk' ich mir...

VOIP mit Net::Sip
Internettelefonie mit Perl

Neu, schön und grün:
$foo im neuen Design – was ab sofort für immer und nur heute anders ist.

Nr 04

Sichern Sie Ihren nächsten Schritt in die Zukunft

Astaro steht für benutzerfreundliche und kosteneffiziente Netzwerksicherheitslösungen. Heute sind wir eines der führenden Unternehmen im Bereich der **Internet Security** mit einem weltweiten Partnernetzwerk und Büros in Karlsruhe, Boston und Hongkong. Eine Schlüsselrolle im Hinblick auf unseren Erfolg spielen unsere Mitarbeiter und hoffentlich demnächst auch Sie! Astaro bietet Ihnen mit einer unkomplizierten, kreativen Arbeitsumgebung und einem dynamischen Team beste Voraussetzungen für Ihre berufliche Karriere in einem interessanten, internationalen Umfeld.

Zur Verstärkung unseres Teams in Karlsruhe suchen wir zum nächstmöglichen Eintritt:

Perl Backend Developer (m/w)

Ihre Aufgaben sind:

- Entwicklung und Pflege von Software-Applikationen
- Durchführung eigenständiger Programmieraufgaben
- Optimierung unserer Entwicklungs-, Test- und Produktsysteme
- Tatkräftige Unterstützung beim Aufbau und der Pflege des internen technischen Know-hows

Unsere Anforderungen an Sie sind:

- Fundierte Kenntnisse in der Programmiersprache Perl, weitere Kenntnisse in anderen Programmier- oder Script-Sprachen wären von Vorteil
- Selbstständiges Planen, Arbeiten und Reporten
- Fließende Deutsch- und Englischkenntnisse

Software Developer (m/w)

Ihre Aufgaben sind:

- Entwicklung und Pflege von Software-Applikationen
- Durchführung eigenständiger Programmieraufgaben
- Optimierung unserer Entwicklungs-, Test- und Produktsysteme
- Tatkräftige Unterstützung beim Aufbau und der Pflege des internen technischen Know-hows

Unsere Anforderungen an Sie sind:

- Kenntnisse in den Programmiersprachen Perl, C und/oder C++ unter Linux, weitere Kenntnisse in anderen Programmier- oder Script-Sprachen wären von Vorteil
- Kompetenz in den Bereichen von Internet Core Protokollen wie SMTP, FTP, POP3 und HTTP
- Selbstständiges Planen, Arbeiten und Reporten
- Fließende Deutsch- und Englischkenntnisse

Astaro befindet sich in starkem Wachstum und ist gut positioniert um in den Märkten für IT-Sicherheit und Linux-Adaption auch langfristig ein führendes Unternehmen zu sein. In einer unkomplizierten und kreativen Arbeitsumgebung finden Sie bei uns sehr gute Entwicklungsmöglichkeiten und spannende Herausforderungen. Wir bieten Ihnen ein leistungsorientiertes Gehalt, freundliche Büroräume und subventionierte Sportangebote. Und nicht zuletzt offeriert der Standort Karlsruhe eine hohe Lebensqualität mit vielen Möglichkeiten zur Freizeitgestaltung in einer der sonnigsten Gegenden Deutschlands.

Interessiert?
Dann schicken Sie bitte Ihre vollständigen Unterlagen mit Angabe Ihrer Gehaltsvorstellung an careers@astaro.com. Detaillierte Informationen zu den hier beschriebenen Stellenangeboten und **weitere interessante Positionen** finden Sie unter www.astaro.de. Wir freuen uns darauf, Sie kennen zu lernen!

Astaro AG
Amalienbadstr. 36 • D-76227 Karlsruhe
Monika Heidrich • Tel.: 0721 25516 0

www.astaro.de

e-mail | web | net | security

astaro internet security

VORWORT

Perl-Programmierer (homo perligata)

Die letzten Wochen und Monate waren geprägt von Perlmonger-Treffen, YAPC, berufliche Treffen und anderen Begegnungen mit anderen Perl-Programmierern. Dabei fällt mir auf, dass der Perl-Programmierer im Allgemeinen eine besondere Art von Mensch ist – darum taufe ich diese Sorte von Mensch mal „homo perligata“.

Egal wohin man kommt und wie gut man die Anderen bei einem Treffen kennt, man wird sofort freundlich aufgenommen - so als würde man schon seit etlichen Jahren dazugehören. Ich hoffe, die YAPC-Neulinge empfanden das genauso. Die Perl-„Gurus“ sind absolut normale Typen (ok, manche sind schon etwas abgedreht aber auf eine nette Art und Weise) mit denen man sich unterhalten kann wie mit jedem Anderen.

À propos YAPC... Ein wenig verrückt muss man wohl als „homo perligata“ sein, denn das Orga-Team aus Wien ist mit orangenen Haaren rumgelaufen und die nächsten Organisatoren werden die YAPC::Europe 2008 mit einer schwedischen Flagge auf der Stirn rumlaufen – und das obwohl die YAPC in Kopenhagen stattfindet.

Das ist komischerweise bei anderen Programmiersprachen nicht so ausgeprägt bis gar nicht vorhanden. Das habe ich auf der letzten EuroOSCON gemerkt und bekomme das von einigen Java-Entwicklern immer wieder zu hören.
Der Perl-Programmierer ist ein geselliger Mensch und ist gerne in der „Community“. Außenstehende können das nicht immer begreifen, aber ist man einmal „drin“, ist es absolut toll.

Es scheint aber auch Institutionen zu geben, die dem „homo perligata“ nicht sehr wohl gesonnen sind. Wie ist es sonst zu erklären, dass die gelben „Flitzer“ der Schneckenpost das eine Mal einen $foo-Leser finden und mal soll es den Leser nicht geben. So gibt es wohl in Erlangen ein schwarzes Loch für $foo-Ausgaben. Dort verschwindet das Lesematerial des „homo perligata“ auf unerklärliche Weise – oder ist der Briefträger ein heimlicher Perl-Programmierer? In Wien, Luxemburg und anderen Bereichen von Europa gibt es hingegen eine Raum-Zeit-Verschiebung, die nach zwei erfolgreichen Zustellungen, eine dritte Zustellung des Magazins verhindert.

Aber bei allen die diese Ausgabe in den Händen halten, scheint alles geklappt zu haben. Dann wünsche ich allen „homo perligata“ viel Spaß mit der vierten Ausgabe von $foo.

Die Codebeispiele können mit dem Code
X1MJjA
von der Webseite heruntergeladen werden.

Viel Spaß beim Lesen!
Renée Bäcker

IMPRESSUM

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

INHALTSVERZEICHNIS

Über die Autoren

Renée Bäcker

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

Herbert Breunung

Ein perlbegeisteter Programmierer aus dem wilden Osten, der eine Zeit lang auch Computervisualistik studiert hat, aber schon vorher ganz passabel programmieren konnte. Er ist vor allem geistigem Wissen, den schönen Künsten und elektronischer sowie handgemachter Tanzmusik zugetan. Seit einigen Jahren schreibt er einen Texteditor in Perl für den noch dringend ein Name gesucht wird und betätigt sich auch aktiv aus perl-community.de und der Wikipedia wo er fast nur noch die Kategorie Programmiersprache Perl betreut.

Simon Betrang

Simon Bertrang ist als Entwickler für einen der großen Streaming-Provider tätig. Er verwendet Perl seit über 10 Jahren für einen nicht unbeträchtlichen Teil seiner Projekte und Arbeiten.

Als OpenBSD-Entwickler ist er unter anderem bemüht die Perl-Pakete auf dem neusten Stand zu halten und das Repertoire zu erweitern.

Martin Fabiani

Martin Fabiani http://www.fabiani.net/ (33 Jahre alt) kommt aus Nordtirol, lebt und arbeitet aber seit 1998 in Deutschland. Seit 1999 ist er freiberuflich als Perl-Entwickler und -Trainer tätig, und hat im Jahr 2000 über Perl das spannende Thema Metadirectories und Identity Management entdeckt, wo man Perl hervorragend für Datensynchronisationen verwenden kann, vor allem bei größeren Datenmengen und komplexeren Synchronisationsalgorithmen, und somit auch für die teilautomatisierte Administration verbundener Systeme. In seiner Freizeit leitet er das Forum auf http://www.perl-community.de unter dem Pseudonym Strat und versucht, ein Mega-Synchronisationsframework für Perl mit Schnittstellen zu einer Menge verschiedener Datenhaltungssystemen zu entwickeln.

Tina Müller

Tina Müller hat 1998 in einem Studentenjob Perl kennengelernt. Nachdem sie an der Universität Miranda, Modula 2 und Java beigebracht bekam, war dies also ihre erste Sprache, die sie dank der Ausdrucksstärke und dem Do-What-I-Mean-Konzept begeistern konnte. Nach und nach lernte sie die Perl-Community kennen, also das Usenet, Webforen, die Konferenzen und Workshops, die Perl-Mongers, Perl-Golf usw. Während für sie so in privaten Projekten Perl nicht mehr wegzudenken ist, hat sie das Glück, auch im Beruf mit Perl zu programmieren, vornehmlich Backends für kleine und große Webseiten.

Ronnie Neumann

Ronnie Neumann nutzt Perl seit seiner Zeit als System- und Netzwerkadministrator. Seit dieser Zeit ist er als User im Forum http://board.perl-comunity.de aktiv. Ronnie nutzt Perl für administrative Tasks und Web-Anwendungsentwicklung. Seit einiger Zeit ist er als Lehrer für arbeitstechnische Fächer an einer Berufsschule in Frankfurt tätig und hat dort leider zu wenig Einsatzmöglichkeiten für Perl.

Steffen Ullrich

Steffen Ullrich <Steffen_Ullrich@genua.de> hat 18 Jahre Programmiererfahrung, davon die letzten 11 Jahre fast ausschließlich mit Perl. Er arbeitet zur Zeit bei der Firma GeNUA mbh (http://www.genua.de) was neben einer angenehmen, familienfreundlichen Arbeitsatmosphäre auch ein Paradies für Perl-Entwickler ist. Er ist Autor und Maintainer von `Net::SIP` und `Devel::TrackObjects` sowie aktueller Maintainer von `IO::Socket::SSL`.

//SEIBERT/MEDIA lässt das $foo-Gewand glänzen

In der Tat war der Auftrag etwas außergewöhnlich: Europas einziges Magazin in Sachen Perl-Programmierung sollte ein neues Design erhalten. Diesmal also kein Relaunch für eine Website, sondern für ein noch junges, wachsendes und hochkarätiges Printprodukt – die Design-Abteilung der Wiesbadener Internet- und Werbeagentur //SEIBERT/MEDIA war begeistert. „Solche Aufträge sind für einen Grafiker natürlich eine wunderbare Sache, bei der man seine ganze Kreativität ins Spiel bringen kann“, freute sich Abteilungsleiter Martin Riekert über die Aufgabe für sein Team, das aus 6 Mitarbeitern besteht. Neben den bisherigen Kernbereichen Web-Technologie, Online-Marketing und Consulting entwickeln sich die Designer im seit 1996 bestehenden Hause Seibert, das immerhin rund 60 Mitarbeiter beschäftigt, zu einem gefragten Partner. Ob großflächige Plakataktionen, die Gestaltung von langfristig angelegten Anzeigenkampagnen, komplette Produkteinführungen oder gerade aktuell der Entwurf eines umfangreichen Wintersportkatalogs in zehntausendfacher Auflage – längst ist es nicht mehr allein der Internetauftritt und die Web-Applikation mit der Perl-Programmierung im Backend, die erfolgreich realisiert werden.

Vom Selfmade-Layout zum professionellen Kult-Magazin

Letztendlich war es exakt jene Entwicklung, die das „$foo Perl-Magazin“ eine Anfrage Richtung Landeshauptstadt schicken ließ. Redakteur Renée Bäcker:

„Wir brauchen eine Leitidee, ein Kommunikationskonzept und eine grafische Realisierung, in der sich Perl und unsere Zeitschrift mit den entsprechenden Eigenschaften und Charakteren optimal wiederfinden. Macht doch mal.“

Nachdem ja bereits einige Ausgaben existierten, waren die ersten Schritte zwar logisch, „lasst uns die Exemplare mal richtig auseinandernehmen“, mitunter aber auch ziemlich verstörend: „Uiuiuiui, wie sieht das denn aus?“, erinnert sich Martin Riekert an die Momente, in denen der Handlungsbedarf besonders deutlich wurde. Das war’s dann allerdings schon mit dem Lamentieren, die Aufgabenstellung wurde schließlich klar und deutlich formuliert:

Folgende Werte sind zu berücksichtigen: vielfältig – einfach, grenzenlos – individuell sowie modular – praktisch; dazu Perl-Gemeinde, Kult-Magazin, Programmierung und Freiheit. Fünf Kategorien ergaben sich aus diesen Vorlagen, wir brauchen

1. ein modular erweiterbares Layoutsystem,
2. die Berücksichtigung der Einfachheit und des Kultcharakters der Skriptsprache,
3. eine junge, dynamische und humorvolle Bildsprache,
4. ein funktionales und modernes Typokonzept,
5. sowie eine individuelle Farbgestaltung je Magazinausgabe.

Kurz: Jede Ausgabe soll einzigartig sein!

Da Perl (die größtmögliche Freiheit für Programmierer!) eine der ältesten etablierten Programmiersprachen überhaupt ist, wurde die Leitidee „Kult-Magazin“ für den Relaunch als sehr passend empfunden. Der Kult-Gedanke sollte sich im gesamten Erscheinungsbild widerspiegeln: Flächen mit Rapport aus Kreisen wie auf 70er-Jahre-Tapeten wurden eingesetzt, ebenso prägnante Balkenelemente, abgerundete Ecken sowie schräge Winkel verwendet. Der Charakter des Spontanen und Einzigartigen wird auf diese Art und Weise bei jeder Ausgabe neu erlebt – inklusive einer ständig wechselnden frischen Farbgebung und dem Dromedar. Natürlich. Um die Sammelleidenschaft der Perl-Gemeinde auf ein Neues zu entfachen, erhält jede Ausgabe übrigens ihre ganz persönliche Farbnote. In dieser Ausgabe ist es passenderweise die Hausfarbe des „Malers“.

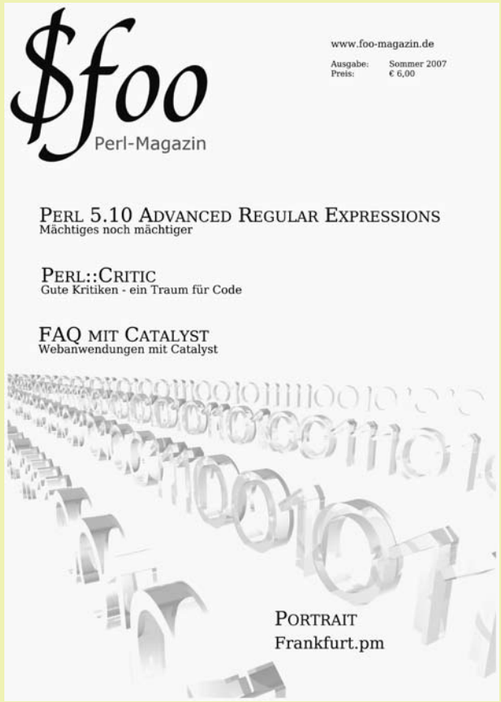
Perl ist Kult. Und $foo das dazugehörende Magazin.

Es grüßen, die Design-Abteilung von //SEIBERT/MEDIA und ein paar stolze Perl-Programmierer.

// SEIBERT / MEDIA

//SEIBERT/MEDIA GmbH
Rheingau Palais
Söhnleinstraße 8, 65201 Wiesbaden
www.design.seibert-media.net

Consulting / Technologies / Design



01 Vorher

02 Nachher

***01** $foo-Magazin (Ausgabe 2, 2007) im alten Layout / funktional und puristisch*
***02** $foo-Magazin (Ausgabe 4, 2007) nach dem Layout-Relaunch / individuell, jung, dynamisch, kultig*

Perl + Excel + Kommandozeile = ExcelPerl

Motivation

Auf dem Frankfurter Perlmongers-Stammtisch hörte ich von XLSperl (http://perl.jonallen.info/projects/xlstools), mit dem man Excel-Dateien mit Perl-Einzeilern lesen kann. Diese Idee gefiel mir, weil es einen Arbeitsschritt erspart, nämlich das Speichern von Excel-Dateien im CSV-Format, bevor man Perl-Einzeiler darauf los lässt, und durch Verwendung des Moduls `Spreadsheet::ParseExcel` ist es ziemlich plattformunabhängig und benötigt auch kein installiertes Excel.

Ich verwende Perl-Einzeiler und Excel häufig zur Analyse von irgendwelchen Daten oder zur Generierung von Testdaten. Bei XLSperl störte mich jedoch, dass Excel-Dateien nur gelesen werden können, nicht aber verändert. Und bei dem Zwischenschritt CSV-Datei gehen eventuelle Formatierungen und Formeln verloren. Man kann zwar eine zweite Excel-Datei via Spreadsheet::WriteExcel dranhängen, in die man die Datei Zelle für Zelle kopiert, aber das erschien mir zu unhandlich und anfällig für verschiedene Verluste (die Spreadsheet-Excel-Module decken ja den Funktionsumfang von Excel nicht 100% ab).

Ein weiterer möglicher Weg wäre, auf Excel-Dateien via ODBC (z.B. via DBI und DBD::ODBC) zuzugreifen. Aber dieser Weg passte weniger zu meinen Vorstellungen, weil ich mehr Kontrolle über Excel haben wollte als nur über die reinen Daten. Da ich mit Excel-OLE-Automationen schon ein wenig Erfahrung hatte, beschloss ich, mein eigenes XLSperl zu schreiben, das auf Win32::OLE basiert, und nannte es ExcelPerl.

Vorteile

- Direkte Änderungen an der Datei sind möglich (Werte, Formate, ...), ohne dass Informationen verloren gehen.
- Zugriff auf die komplette Funktionalität der installierten Excelversion.
- Der Benutzer hat die Wahl, ob und unter welchem Namen er die Änderungen speichern will.
- Ich kann weitere benötigte Features einbauen, die in XLS perl (noch?) nicht vorhanden sind.

Nachteile

- Läuft nur noch unter MS-Windows-Betriebssystemen
- Benötigt Excel

Grundsätzliche Ideen

Die Verwendung sollte analog Perl-Einzeilern mit -ane funktionieren, die Werte einer Zeile in einer Liste @F darstellen, und auch die Variablen $_ (= gesamte Zeile, gejoint durch ein Trennzeichen) und $. (=aktuelle Zeilennummer) sollen vorhanden sein.

Ich will nicht nur an die Werte, sondern auch auf die Zellenobjekte direkt zugreifen. Ich habe mich dafür für ein Array namens @C entschieden.

Man sollte angeben können, welches Worksheet man bearbeiten will, und ab welcher Zeile man mit der Bearbeitung beginnen will, um so eventuelle Überschriften überspringen zu können.


```
Programmargumente parsen
Neues ExcelPerl-Objekt erstellen und Excel-Datei öffnen
Code aus dem optionalen Begin-Block ausführen
Solange eine Zeile aus Excel gelesen werden kann:
    Tu das und stelle die Variablen @F, @C, $_ und $. bereit
    Führe den im Argument -ane angegebenen Perl-Code auf die Zeile aus
    Code aus dem optionalen End-Block ausführen
Fertig
```

Listing 1

Man soll Code vor und nach der Schleife ausführen können. Dafür sehe ich zwei weitere Programmparameter namens -begin und -end vor (ab V0.12).

Ich will nicht nur an die Werte, sondern auch auf die Zellenobjekte direkt zugreifen. Ich habe mich dafür für ein Array namens @C entschieden, das die Zellen-Objekte der aktuellen Zeile enthält.

Die Funktionalität soll ein OOP-Modul werden, das von einem kleinen Hauptprogramm aufgerufen wird; vielleicht hat jemand mal Verwendung für das Modul.

Wenn @F verändert wird, soll diese Änderung auch gleich in die Excel-Datei geschrieben werden. Dafür bot sich entweder eine Art Cleanup-Handler an, der mit dem Vergleichen und aktualisieren loslegt, wenn @F aus dem Scope fällt, oder ein Tie. Ich entschied mich für den Tie, weil ich so den größten Teil der Komplexität aus dem Hauptprogramm heraushalten kann.

Planung

Mein Hauptprogramm soll in etwa aussehen wie in Listing 1 dargesetellt.

Also begann ich mit dem OOP-Framework (siehe die Datei ExcelPerl.pm). Die wichtigsten Teile sind die folgenden Methoden:

- new erzeugt ein neues Objekt
- openöffnet die Excel-Datei via Win32::OLE
- getNextRow liefert die nächste Zeile als tied Array @F und die Objekte als @C zurück
- die folgenden Getter/Setter-Methoden:
  - currentRow: ist Basis für $.
  - changeCount hier wird festgehalten, ob Änderungen an der Datei vorgenommen wurden
  - und weitere

Da Tie mit einem eigenen Namensraum glücklicher ist als mit einem geteilten, habe ich dafür ExcelPerl::RowArray verwendet. Ich fügte diesen Namensraum ans Ende der Datei ExcelPerl.pm ein, weil mir dafür eine weitere Datei nicht gerechtfertigt schien.

Aus dem Tie sind die folgenden Methoden interessant:

- TIEARRAY der Konstruktor, der das Objekt als Hashreferenz initialisiert
- STORE bekommt folgende Parameter: das Objekt, den Index der Änderung und den neuen Wert. Wenn es sich um eine Änderung handelt, wird diese an Excel weitergereicht und der Änderungszähler via C<$excelObj-E #!/usr/bin/ erhöht

Probleme und Lösungen

Ein erster Prototyp war nach etwa zwei Stunden fertig. Er funktionierte, war aber so häßliche, dass ich nach besseren Wegen suchte. Ich fragte auf http://board.perl-community.de und http://www.perlmonks.org/ um Rat, wenn mir kein sauberer Lösungsweg einfiel.

Vor allem pkai aus Darmstadt hatte ein paar tolle Verbesserungsvorschläge (Nochmals ein großes Dankeschön!).

Verwendeten Datenbereich erkennen

Da ich keine brauchbare Excel-Methode fand, die mir den verwendeten Bereich des Spreadsheets gab, ich aber nicht immer über alle möglichen Zeilen iterieren wollte, war mein erster Versuch, den verwendeten Bereich des Spreadsheets über Leerzeilen zu erkennen (wenn mehr als n Leerzeilen in Folge kommen, muss dies das Ende sein). Hat zwar einigermaßen funktioniert, war aber suboptimal.

pkai fand die folgende Lösung (siehe Listing 2):

```
$bottomRight = (
 split( /:/,
     $worksheet->UsedRange->address(0,0) )
 )[-1];
```
Listing 2

Das half mir schon eine Menge weiter, und machte das Programm um einiges robuster und nebenbei auch schneller.

Tied Array aus Subroutine zurückgeben

Wenn man was aus einer Subroutine zurückgibt, wird dies zuerst in eine interne Liste gespeichert (hier ist mal wieder der Unterschied zwischen Array und Liste deutlich zu sehen), und dadurch verliert das Array natürlich alle Magie und somit auch den Tie. Über eine globale Variable wollte ich es nicht lösen, weil da ja wieder der Variablenname sowohl im Hauptprogramm als auch in ->getNextRow bekannt sein muss. Man kann @F zwar als Referenz zurückgeben, aber dann könnte ich nicht mehr mit $F[0] arbeiten, sondern mit $F->[0], und ich wollte aber das Interface so weit wie möglich identisch zu perl -ane haben. Ich dachte über viele komplizierte Sachen nach, fand heraus, dass sie nicht funktionieren würden, und überlegte mir noch kompliziertere Sachen. Erst nobull von den PerlMonks, die ich um Hilfe fragte, sagte mir, wenn ich es nicht rausbekommen könne, dann muss ich es halt reinbekommen, und zwar mit Call-By-Reference (peinlich, daß es so einfach gehen kann und es mir nicht eingefallen ist):

```
my @F = ();
while( my @C = $excelPerl->getNextRow( \@F ) {

und in ExcelPerl.pm dann:
sub getNextRow {
    # $F ist Referenz auf @F
    my( $self, $F ) = @_;
    …
    tie( @$F, 'ExcelPerl::RowArray', $self );
    …
    return @C;
} # getNextRow
```
Listing 3

Excel braucht absoluten Dateinamen

`$excel->Workbooks->Open( $file )` kann die Datei nur finden, wenn man sie mit einem absoluten Pfad angibt. Komfortabler ist jedoch, wenn der Dateiname beim Aufruf von ExcelPerl auch relativ angegeben werden kann, und dann on-the-fly in einen absoluten verwandelt werden kann. Gottseidank gibt es aus dem Modul Win32 die Funktion GetFullPathName für derartige Probleme, also (siehe Listing 4):

```
# wird bei Activestate automatisch geladen
use Win32;
# Fehlerbehandlung
defined( $file ) or die "Usage\n";
# Unwandlung
$file = Win32::GetFullPathName( $file );
```
Listing 4

Perl-Code wird für jede Zeile erneut evaluiert

```
while( my @C = $excelPerl->getNextRow( \@F ) {
    # ...
    eval $perlCode;
    die $@ if $@;
} # while
```
Listing 5

Hier schug pkai vor, das `eval` vor die Schleife zu stellen und eine anonyme Subroutine daraus zu machen (so wie bei XLSperl), und ich entwickelte (mit Vereinfachung von tinita) das folgende Konstrukt (gekürzt):

```
my $aneCode = eval <<'EOM' . $perlCode . '};';

sub {
    my $xlsPerl = shift;
    my $C = shift;
    my @C = @$C;
    # prepare some other helpful variables
    local $. = $xlsPerl->currentRow();
    no warnings 'uninitialized';
    local $_ = join(
        $xlsPerl->colSeparator, @F
    ) . „\n";
EOM
    die $@ if $@;
```
Listing 6

So kann die Schleife jetzt ganz einfach aussehen, und läuft nebenbei auch schneller:

```
while( my @cells =
        $xlsPerl->getNextRow( \@F ) ) {
     $aneCode->( $xlsPerl, \@cells );
 } # while
```
Listing 7

BEGIN- und END-Blöcke wären hilfreich

In der Version 0.12 habe ich zwei weitere Argumente hinzugefügt mit Namen -begin und -end, mit der zwei optionale Codestrings übergeben werden können, die dann vor bzw. nach der while-Schleife ausgeführt werden, sodass jetzt auch folgende Konstrukte möglich sind (siehe Listing 8 und Listing 9).


```
excelPerl.pl                                    ^
    -ane "$x{$F[0]}++"                          ^
    -end "use Data::Dumper; print Dumper \%x"   ^
    -f test.xls
```
Listing 8

```
excelPerl.pl                                                    ^
    -begin "use Text::CSV_XS; $csv = Text::CSV_XS->new( { binary => 1} )"   ^
    -ane "$csv->combine(@F); print $csv->string, $/"            ^
    -f file.xls
```
Listing 9

Spart manchmal ein klein wenig an Tipperei...

Weitere Verwendungsbeispiele erhält man, wenn man `perldoc excelPerl.pl` eingibt.

Download von ExcelPerl

http://www.fabiani.net/ -> Perl -> Downloads

Martin Fabiani

PERL-BOOTCAMP 2008

In der Zeit vom 28. April bis zum 02. Mai 2008 findet im Kloster Eberbach (in der Nähe von Wiesbaden) ein Perl-Bootcamp statt. Organisiert wird der Kurs von Big Nerd Ranch Europe. Als Trainer wurde brian d foy angeheuert, der an mehreren Perl-Büchern mitgeschrieben hat und vor kurzem sein neuestes Werk „Mastering Perl" herausgebracht hat. Somit dürfte der Kurs auch etwas für erfahrenere Perl-Programmierer ganz interessant sein – wenn auch nicht ganz billig.

Bei dem Bootcamp werden folgende Themen behandelt:

- Implementierung Objekt-Orientierter Konzepte in Perl
- Packages und Namensräume verwenden
- Referenzen und Gültigkeitsbereiche
- Komplexe Datenstrukturen verändern
- Erstellen von Modulen für den eigenen Bedarf
- Perl-Code testen und debuggen
- zu CPAN beitragen

Weitere Informationen unter:
http://bignerdranch.com/classes/perl.shtml

Renée Bäcker

MODULE

RegEx Module – Zahme Regenechsen...

Reguläre Ausdrücke (RegEx) zählen eindeutig zu den Stärken von Perl und man landet doch recht häufig bei diesem Mittel. Die RegEx können dabei recht verwirrend sein.

In diesem Artikel werden einige Module im Zusammenhang mit Regulären Ausdrücken gezeigt. Das geht von Modulen, die dem Programmierer die Zusammenstellung der RegEx abnehmen bis hin zu Modulen, die bei der „Analyse“ von Regulären Ausdrücken helfen.

Heinzelmännchen für Programmierer

Regexp::Common

Eine große Sammlung von „alltäglichen“ Regulären Ausdrücken wie IP-Adressen, Integer-Zahlen und anderes ist in Regexp::Common vereint. Durch einen einfachen Mechanismus können auch Plugins recht schnell geschrieben werden. Der Vorteil der Sammlung gegenüber selbstgeschriebenen Ausdrücken liegt darin, dass `Regexp::Common` schon viel getestet wurde.

Die Verwendung von `Regexp::Common` ist sehr einfach. Die Regulären Ausdrücke sind „hierarchisch“ in einem Hash abgelegt.

So sind unterhalb von `$RE{num}` alle Ausdrücke, die Zahlen behandeln, zu finden. Mit `$RE{num}{int}` bekommt man den Regulären Ausdruck für Integerzahlen.

Ein kleines Beispiel, in dem Dezimalzahlen gesucht werden, ist in Listing 1 zu sehen.

```
#!/usr/bin/perl

use strict;
use warnings;
use Regexp::Common;

my @zahlen = qw(1.000 0.5 3 15 15e21);

for my $val ( @zahlen ){
    if( $val =~ /^$RE{num}{int}$/ ){
        print $val, ": yes\n";
    }
}
```

Listing 1

Regexp::Assemble

1000 einzelne Regexes, doch wie kann man das zusammenfassen? Die Lösung heißt `Regexp::Assemble`. Mit diesem Modul können mehrere einzelne Reguläre Ausdrück mit “ODER‘ verknüpft werden. Das Modul macht dabei keine sture Aneinanderreihung der Teile, sondern versucht „intelligent“ Schnittmengen zu finden und das Resultat als kompakten Ausdruck darzustellen.

Wenn in einem String überprüft werden soll, ob er die Worte „Perl“ oder „Pelle“ enthält, dann sieht das Skript so aus (Listing 2):

```
#!/usr/bin/perl

use Regexp::Assemble;

my @words = qw(Perl Pelle);
my $string = "Perl ist toll";

my $re = Regexp::Assemble->new;
$re->add($ _ ) for @words;

print "yes\n" if $string =~ /$re/;
```

Listing 2

Der Reguläre Ausdruck, der von `Regexp::Assemble` erzeugt wird, sieht so aus: `(?-xism:Pe(?:lle|rl))`

Leider kann das Modul keine ‘UND‘ Verknüpfung, aber es ist trotzdem sehr hilfreich. Auch Buchstaben-Bereiche werden von dem Modul nicht erkannt. Füttert man Regexp::Assemble einzeln mit den Ziffern 1 bis 9, so macht es daraus den RegEx `(?-xism:[123456789])` statt `(?-xism:[1-9])`.

Ein weiteres Modul in diese Richtung ist `Regex::PreSuf`

Regexp::MatchContext

Gern verwendete Variablen bei Regulären Ausdrücke sind `$``, `$&` und `$'` beziehungsweise deren langnamigen Pendants `$PREMATCH`, `$MATCH` und `$POSTMATCH`. Diese Variablen sind hilfreich, aber sie haben einen ganz großen Nachteil: Die Regulären Ausdrücke werden extrem langsam. Und das wirkt sich nicht nur auf den Regulären Ausdruck aus, in dem diese Variablen verwendet werden, sondern auf **alle** RegEx im Programmlauf.

Tina Müller hat mal mit einem ganz einfachen Skript die Zeitunterschiede verdeutlicht (Listing 3), danach wurde die Zeile `my $x = $&;` auskommentiert und die Aufrufe wiederholt (siehe Listing 4).

```
$ cat regex.pl
#!/usr/bin/perl
use strict;
use warnings;
my $x = $&;
my $string = "123" x 1000;
for (0..100000) {
    if ($string =~ m/3/) {
        my $x = "matched";
    }
}
$ time perl regex.pl

real    0m0.108s
user    0m0.100s
sys     0m0.008s
$ time perl regex.pl

real    0m0.107s
user    0m0.104s
sys     0m0.000s
```

Listing 3

```
$ time perl regex.pl

real    0m0.056s
user    0m0.052s
sys     0m0.004s
$ time perl regex.pl

real    0m0.055s
user    0m0.052s
sys     0m0.004s
```

Listing 4

So richtig benchmarken kann man das Problem nicht, da die Auswirkungen auf die Laufzeit sehr stark vom Programm abhängen. Ein Programm mit sehr vielen Regulären Ausdrücken hat natürlich wesentlich größere Laufzeitnachteile als ein Programm mit sehr wenigen RegEx.

Damian Conway hat ein Modul geschrieben, das diesen Geschwindigkeitsnachteil zumindest auf den einen Regulären Ausdruck begrenzt. Dazu muss in dem Regulären Ausdruck der Modifier (?p) verwendet werden. Ein Beispiel dazu ist in Listing 5 zu sehen. Mit der Benutzung des Moduls gibt es erst einen Zeitvorteil wenn mehrere Reguläre Ausdrücke verwendet werden, von denen nicht alle diese „Kontext“-Variablen verwenden.

In Perl 5.10 braucht man das zusätzliche Modul nicht mehr. Yves Orton hat dort den p-Modifier in die RegEx-Engine eingebaut, die ähnlich wie Conways Modul bestimmte Variablen nur für den Regulären Ausdruck setzt, der das ausdrücklich anfordert (siehe auch $foo „Sommer 2007“).

Was passiert da eigentlich?

YAPE::Regexp::Explain

Reguläre Ausdrück sind teilweise verdammt schwer zu verstehen. Gerade Perl-Einsteiger schauen häufiger mal mit drei Fragezeichen im Gesicht auf ein Perl-Programm. „Was macht denn das da? Das ist ja total kryptisch.“ bekommt man dann zu hören. In so einem Fall hilft YAPE::Regexp::Explain. Es nimmt die RegEx auseinander und erläutert jeden Teil ausführlich. So bekommt man einen Eindruck davon, was der Reguläre Ausdruck überhaupt macht.

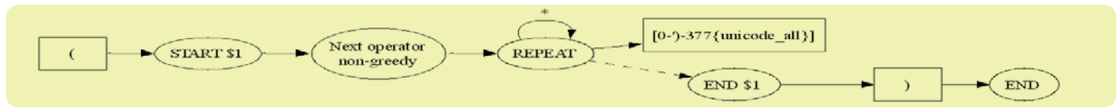
```
#!/usr/bin/perl

use strict;
use warnings;
use Regexp::MatchContext -vars;

my $string = 'Dies ist ein langer Text';
$string =~ /(?p)lang/;

print qq~
PREMATCH:   $PREMATCH
MATCH:      $MATCH
POSTMATCH:  $POSTMATCH
~;
```

Listing 5



Aber das Modul hilft natürlich nicht nur den Einsteigern sondern auch „alten Hasen“. Ein einfaches Beispielprogramm ist in Listing 6 dargestellt

In der Ausgabe des Programms (siehe Listing 7) erkennt man, wie das Modul die einzelnen Teile des Regulären Ausdrucks erklärt. Dadurch ist es auch sehr gut dazu geeignet, die verschiedenen Konstrukte wie `(?:)` kennenzulernen.
Durch die Einrückung der Erklärungen ist gut ersichtlich, zu welchem Klammerpaar ein Teil des Ausdrucks gehört.

GraphViz::Regex

Eine graphische Ausgabe kann mit dem Modul `GraphViz::Regex` von Leon Brocard erzeugt werden. Das ist natürlich vor allem für Leute interessant, die in der Schule oder sonstwo schon etwas von „Automaten“ gehört haben.

```
#!/usr/bin/perl

use strict;
use YAPE::Regex::Explain;

my $re = '\(([^\(]*?)\)';
print YAPE::Regex::Explain->new($re)->explain;
```

Listing 6

Das Modul versucht nämlich aus dem Regulären Ausdruck einen Automaten zu erzeugen. Somit kann man auch „per Hand“ versuchen, zu einem gegebenen String nachzuvollziehen warum der RegEx einen Treffer erzeugt oder eben auch nicht. So kann man auch Personen, die mit Regulären Ausdrücken nicht unbedingt etwas zu tun haben, die Ergebnisse näherbringen.

Wer sich gut mit Interna auskennt, kann auch den in Perl eingebauten RegEx-Debugger verwenden. Doch sehr vielen wird die Ausgabe nur wenig sagen. Der RegEx-Debugger ist wieder ein Thema für sich.

Renée Bäcker

```
#!/usr/bin/perl

use strict;
use warnings;
use GraphViz::Regex;

my $re = '\(([^\(]*?)\)';
my $graph = GraphViz::Regex->new( $re );

my $output = 'regex.png';
open my $fh, '>', $output or die $!;
binmode $fh;
print $fh $graph->as_png;
close $fh;
```

Listing 8

```
C:\Perl\FooMagazin>explain.pl
The regular expression:

(?-imsx:\(([^\(]*?)\))

matches as follows:

NODE             EXPLANATION
----------------------------------------------------------------
(?-imsx:         group, but do not capture (case-sensitive)
                  (with ^ and $ matching normally) (with . not
                  matching \n) (matching whitespace and #
                  normally):
----------------------------------------------------------------
  \(               '('
----------------------------------------------------------------
   (                group and capture to \1:
----------------------------------------------------------------
    [^\(]*?          any character except: '\(' (0 or more
                      times (matching the least amount
                      possible))
----------------------------------------------------------------
  )                end of \1
----------------------------------------------------------------
  \)               ')'
----------------------------------------------------------------
  )                end of grouping
  ----------------------------------------------------------------
```

Listing 7

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

Smart-Websolutions
Windolph und Bäcker GbR
Perl-Programmierung
info@smart-websolutions.de

MODULE

Objektorientierung in Perl mit Moose

Eine typische Problemstellung – klassisch gelöst

Objektorientierung ist eine Technik, die jeder Programmierer in seinem Werkzeugkasten haben sollte. Nicht immer ist ein objektorientierter Ansatz aber auch der beste oder schnellste Weg ein Problem zu lösen.
In diesem Artikel, in dem es um die Möglichkeiten von OOP in Perl gehen soll, möchte ich mit einem Beispiel beginnen, das sehr gut ohne Objektorientierung zu lösen ist.

Ich bin im Netz, auf der Webseite codekata.com, auf die Aufgabenstellung gestoßen auf der Dave verschiedenste Programmieraufgaben präsentiert.

Es handelt sich um die Aufgabe, aus einer Textdatei mit den Wetterdaten eines Monats, den Tag zu suchen an dem die Temperaturabweichung zwischen maximaler und minimaler Tagestemperatur am geringsten ist. Hier jetzt eine perltypische Lösung, die ich für die Aufgabe entwickelt habe (siehe Listing 1)

```
#!/usr/bin/perl
use strict;
use warnings;
my %msdy = (
   'day' => 0,
   'max' => 1000,
   'min' => 0
);

while( <DATA> ){
   next unless /^\s*\d+/;

   my ($day, $max , $min ) = /^\s*(\d+)\s*(\d+)\*?\s*(\d+)\*?/ ;
   @msdy{'day', 'max', 'min'} = ( $day, $max, $min ) if
       $max - $min < $msdy{'max'} - $msdy{ 'min' };
}

print $msdy{'day'} . "\n";

__DATA__

MMU June 2002

Dy MxT   MnT   AvT
1  88    59    74
2  79    63    71
3  77    55    66
4  77    59    68
5  90    66    78
6  81    61    71
7  73    57    65
8  75    54    65
9  86    32*   59
mo  82.9  60.5  71.7
```

Listing 1

Die Wetterdaten habe ich in die __DATA__ Sektion gepackt. Dies gilt auch für die nachfolgenden Skripte, auch wenn es nicht jedesmal abgebildet ist.

Ähnliche Lösungen haben ich und viele andere Administratoren und Programmierer, schon dutzendfach entwickelt, um z.B. Log-Dateien zu parsen. Zu Beginn wird ein Hash definiert, wobei der Wert für die maximale Temperatur so gewählt ist, dass er weit außerhalb der zu erwartenden Messwerte liegt. Die `while`-Schleife iteriert über die einzelnen Zeilen. Mit einem regulären Ausdruck wird überprüft, ob es sich um eine Zeile mit Messwerten handelt. Dies wird angenommen, wenn die Zeile, nach einigen Leerzeichen, mit einer Ziffern-Folge beginnt. Ist dies gegeben, wird über einen weiteren regulären Ausdruck der Inhalt der Zeile auf die drei Variablen `$day`, `$max`, und `$min` verteilt. Wenn die Temperaturabweichung kleiner ist, als die in dem zu Beginn des Skriptes definierten Hashs, werden dessen Werte überschrieben. Nach der Schleife wird der (erste) Tag aus dem Hash ausgegeben, der im Monat die kleinste Differenz zwischen maximaler und minimaler Temperatur aufweist.

Wie bereits zu Beginn des Artikels aufgeführt, ist diese Lösung ohne Objektorientierung sehr kompakt und in wenigen Minuten entwickelt. Wieso also sollte man eine objektorientierte Lösung anstreben?

Eine objektorientierte Lösung

Die obige Lösung ist straight-forward - sie löst das Problem in wenigen Zeilen Perl. Sie kann aber auch für sonst nichts verwendet werden. Oft ist das so völlig in Ordnung, wenn es um eine Einmal-Lösung – ein Wegwerf-Skript – geht. Wenn wir über Objektorientierung sprechen, dann geht es primär auch um die Wiederverwendbarkeit von Code. Es macht also Sinn objektorientierte Lösungen zu entwickeln, wenn es absehbar ist, dass uns die Problemstellung weiterhin aus verschiedenen Perspektiven beschäftigen wird. Bevor wir aber diesen nächsten Schritt tun, möchte ich erstmal eine objektorientierte Lösung der bekannten Problemstellung aufzeigen.

Wenn wir den Datensatz betrachten, können wir – der menschliche Betrachter – erkennen, dass es um eine Liste/Tabelle von Temperatur-Messwerten bzw. Messungen handelt. Genau diesen Sachverhalt können wir objektorientiert abbilden. Wir bilden also zwei Klassen: eine für Messwerte und eine für die Liste mit den Messwerten:

Die Klasse `TemperatureProbe` beinhaltet Attribute für Tag, maximale Temperatur, minimale Temperatur und die durchschnittliche Temperatur, die wir ebenfalls der Textdatei entnehmen können – bis jetzt aber keine Verwendung dafür hatten. Ein weiterer Aspekt der Objektorientierung ist, das Methoden mit den Objekten, bzw. der Klasse verbunden werden können. Wir benötigen den Konstruktor `new`, sowie Methoden, die uns den Zugriff auf die Attribute erlauben, also `day`, `max`, `min` und `avg`. In Perl existiert zwar keine strenge Kapselung, die uns zwingt auf die Attribute über Methoden zuzugreifen, aber es ist eine sinnvolle Vorgehensweise, die es uns z.B. erlauben würde, notwendige Daten-Validierungen vorzunehmen, worauf ich aber erstmal verzichten möchte.

Die Klasse `TemperatureTable` benötigt erstmal nur ein Attribut `table`, das eine Liste von Messwerten, der uns bekannten Objekte vom Typ `TemperatureProbe` beinhaltet. Wir benötigen im ersten Schritt drei Methoden, den Konstruktor `new`, sowie eine Methode zum parsen einer Datei `parse_file`, die als Argument ein filehandle erwartet, sowie die Methode `get_min_spread_day`, die uns den gesuchten Tag mit der geringsten Temperaturdifferenz liefert.

Mit typischer Perl5 Objektorientierung sieht das Ganze dann z.B. so aus (siehe Listing 2).

Wenn man die Klassen außer Acht lässt, reduziert sich unser Programm jetzt auf drei Zeilen:

```
my $table = TemperatureTable->new;
$table->parse_file(*DATA);
print $table->get_min_spread_day . "\n";
```

Listing 3

Diese Betrachtungsweise ist aber nur bedingt legitim, weil die Klassen deutlich mehr Zeilen benötigen, als unser ursprüngliches Skript. Was haben wir also gewonnen? Wir haben jetzt eine vollständige Repräsentation der Tabelle als Objekt zur Verfügung. Wir können also problemlos weitere Methoden der Klasse `TemperatureTable` hinzufügen. Wenn wir z.B. eine Methode `get_max_spread_day` benötigen, können wir sie einfach der Klasse hinzufügen.

```
#!/usr/bin/perl

use strict;
use warnings;

use Data::Dumper;

package TemperatureProbe;

sub new {
    my $class = shift;
    my @probe = @_;

    my $self = {
        day => $probe[0],
        max => $probe[1],
        min => $probe[2],
        avg => $probe[3],
    };
    return bless $self, $class;
}

sub day {
    my $self = shift;
    $self->{'day'} = shift if @_;
    return $self->{'day'}
}

sub max {
    my $self = shift;
    $self->{'max'} = shift if @_;
    return $self->{'max'}
}

sub min {
    my $self = shift;
    $self->{'min'} = shift if @_;
    return $self->{'min'}
}

sub avg {
    my $self = shift;
    $self->{'avg'} = shift if @_;
    return $self->{'avg'}
}

package TemperatureTable;

sub new {
    my $class = shift;
    my $self = { table => [] };
    return bless $self, $class;
}

sub parse_file {
    my $self = shift;
    my $fh = shift or die "parse_file() needs a filehandle to parse on.";
    while (<$fh>) {
        next unless /^\s*\d+/;
        my ($day, $max, $min, $avg) = /^\s*(\d+)\s*(\d+)\*?\s*(\d+)\*?\s*(\d+)/;
        my $current = TemperatureProbe->new($day, $max, $min, $avg);
        push @{$self->{table}}, $current;
    }
}

sub get_min_spread_day {
    my $self = shift;
    my %msdy = ( 'day' => 0, 'max' => 1000, 'min' => 0 );
    for my $probe (@{$self->{table}}) {
```

Listing 2

```
        if ($probe->max-$probe->min < $msdy{'max'} - $msdy{'min'}) {
            @msdy{'day', 'max', 'min'} = ($probe->day, $probe->max, $probe->min)
        }
    }
    return $msdy{'day'};
}

package main;

my $table = TemperatureTable->new;
$table->parse_file(*DATA);

print $table->get_min_spread_day . "\n";
print Dumper $table;
```

Fortsetzung Listing 2

```
#!/usr/bin/perl

use strict;
use warnings;

package TemperatureProbe;
use Moose;

has 'day'   =>  ( isa => 'Num', is => 'rw' );
has 'max'   =>  ( isa => 'Num', is => 'rw' );
has 'min'   =>  ( isa => 'Num', is => 'rw' );
has 'avg'   =>  ( isa => 'Num', is => 'rw' );

package TemperatureTable;
use Moose;

has 'table' =>  ( isa => 'ArrayRef', is => 'rw', default => sub { [] } );

sub parse_file {
  my $self = shift;
  my $fh = shift or die "parse_file() needs a filehandle to parse on.";
  while (<$fh>) {
      next unless /^\s*\d+/;
      my ($day, $max, $min, $avg) = /^\s*(\d+)\s*(\d+)\*?\s*(\d+)\*?\s*(\d+)/;
      my $current = TemperatureProbe->new(
          day => $day,
          max => $max,
          min => $min,
          avg => $avg
      );
      push @{$self->table}, $current;
  }
}

sub get_min_spread_day {
  my $self = shift;
  my %msdy = ( 'day' => 0, 'max' => 1000, 'min' => 0 );
  for my $probe (@{$self->{table}}) {
      if ($probe->max - $probe->min < $msdy{'max'} - $msdy{'min'}) {
          @msdy{'day', 'max', 'min'} = ($probe->day, $probe->max, $probe->min)
      }
  }
  return $msdy{'day'};
}

package main;

my $table = TemperatureTable->new;
$table->parse_file(*DATA);

print $table->get_min_spread_day . "\n";
```

Listing 4

Unser erstes Skript hätten wir dazu komplett umschreiben müssen. Bevor wir uns aber um weitere Methoden Gedanken machen, stellt sich die Frage: Geht das alles auch mit weniger Tipp-Aufwand für Konstruktoren und Accessoren?

Der Ritt auf dem Elch!

Im CPAN finden sich mittlerweile einige Module, die dem geneigten Programmierer die OO-Programmierung in Perl vereinfachen können. Ich möchte in diesem Artikel auf Moose eingehen, das vieles vom kommenden Perl 6 bereits heute in Perl 5 ermöglicht. Bevor ich die einzelnen Features diskutiere, möchte ich erstmal die an Moose angepasste Variante des obigen Skriptes präsentieren (siehe Listing 4).

Änderungen an TemperatureProbe

Die Klasse `TemperatureProbe` ist auf fünf Zeilen reduziert worden, bei der Klasse `TemperatureTable` ist der Konstruktor verschwunden. Konzentrieren wir uns zuerst auf die Änderungen an `TemperatureProbe`:

In der ersten Zeile unter der `package`-Deklaration wird Moose mit `use` eingebunden. Danach werden nur noch die benötigten Attribute definiert:

```
has 'day'   =>  ( isa => 'Num', is => 'rw' );
has 'max'   =>  ( isa => 'Num', is => 'rw' );
has 'min'   =>  ( isa => 'Num', is => 'rw' );
has 'avg'   =>  ( isa => 'Num', is => 'rw' );
```

Listing 5

mit `has 'attribut'` wird das jeweilige Attribut des Objektes bezeichnet, mit `=> ( isa => 'Num', is => 'rw' );` welche Eigenschaften das Objekt besitzt. Das tolle an Moose ist, dass mit dieser Definition das wesentliche erledigt ist. Moose baut für uns den Konstruktor `new`, den wir wie gewohnt verwenden können, sowie auch Accessor-Methoden, die uns den Zugriff auf die Objekteigenschaften erlauben.

Bei der Definition der Attribute würde im einfachsten Fall `is => 'rw'` genügen. Wenn wir sicher sind, dass wir die Attribute eines Objektes nach der Erzeugung nicht mehr verändern wollen, dann wäre auch `is => 'ro'` möglich gewesen, wobei readonly-Attribute erzeugt würden.
Wir haben aber auch die Möglichkeit gleich einen Typ für das Objektattribut anzugeben. Das mag dem einen oder anderen Leser merkwürdig vorkommen, da Perl 5 gar keine Typisierung kennt. Aber Moose kennt Typisierung, die sich an dem orientiert was mit Perl 6 auf uns zukommt. In einem nachfolgenden Beispiel werde ich noch auf die Möglichkeit der Subtypisierung eingehen. Im obigen Beispiel ist als Typ für alle Attribute `isa => 'Num'` angegeben, d.h. das nur numerische Werte für die Attribute akzeptiert werden.

Änderungen an TemperatureTable

Auch in der Klasse `TemperatureTable` verwenden wir die Attributsdefinition von Moose via

```
has 'table' =>  ( isa => 'ArrayRef',
      is =>  'rw', default => sub { [] } );
```

Als Typ geben wir eine Referenz auf ein anonymes Array an. Hier sehen wir auch ein weiteres Feature von Moose, für die Definition von Attributen: Wir können einen default-Wert angeben. In diesem speziellen Fall benötigen wir dazu eine anonyme Subroutine. Wenn es sich um einen skalaren Wert handelt, kann dieser direkt per `default` übergeben werden. Ansonsten ist es notwendig, den default-Wert über eine Codereferenz zu erzeugen.

Eine weitere Änderung ergibt sich in der Methode `parse_file` bei der Erstellung der `TemperatureProbe`-Objekte:

```
my $current = TemperatureProbe->new(
    day => $day,
    max => $max,
    min => $min,
    avg => $avg
);
```

Listing 6

Der selbsterstellte Konstruktor im ersten OOP-Beispiel erwartete eine geordnete Liste mit den Werten für die vier Attribute. Der von Moose für uns erzeugte Konstruktor erwartet eine Liste mit Schlüssel-/Wert-Paaren.

Die notwendigen Anpassungen in der Methode `parse_file` sind minimal und erhöhen darüberhinaus die Lesbarkeit des Programmes.

Rollenspiele

Wir haben jetzt die Möglichkeit die Klassen um beliebige Methoden zu erweitern, je nachdem welche Funktionalität wir benötigen. Wir könnten z.B. eine Klasse von `TemperatureTable` erben lassen, die mit der europäischen Celcius-Skala arbeitet o.ä., wobei wir uns im bekannten Rahmen des objektorientierten Paradigmas bewegen würden.

Häufig benötigen wir aber wiederkehrende Funktionalität in unseren Programmen, wie z.B. die Ausgabe von Daten als HTML-Tabelle, oder als ASCII-Tabelle, oder als (...). An diesen Stellen greifen wir natürlich auf CPAN-Module zurück, die uns die Arbeit erleichtern, trotzdem schreiben wir jede Menge Methoden, die immer wieder fast das selbe tun. Alternativ können wir eine Klasse schreiben, von der wir die benötigten Methoden erben würden. Mit Moose und dem kommenden Perl 6 steht uns aber noch eine andere Möglichkeit zur Verfügung: Roles.

Roles erweitern Klassen auf Basis einer „was kannst du“-Frage, nicht der in der klassischen Objektorientierung üblichen „wer bist du“-Frage. Die Kenner von Ruby kennen dieses Konzept bereits von den dort existierenden Mixins. Klassen werden um Methoden erweitert, in dem sie auf bestimmte Methoden-Aufrufe reagieren können müssen. Ich möchte das am Beispiel konkretisieren (siehe Listing 7).

Als ein Beispiel für eine solche Role ist die obige `Role_HTML_Table` zu sehen. Sie benötigt in jeder Klasse die sie einbinden möchte, die Methode `_as_aoa()`. Dies wird durch die Zeile `requires '_as_aoa';` mit dem Keyword requires definiert. Danach folgt die konkrete Methode `to_html()`, um die jede Klasse erweitert wird, die über die geforderte Methode ein Array_of_Arrays zur Verfügung stellt, sprich eine simple Tabelle. Die Methode `to_html()` ist dabei nur ein simpler Wrapper für das CPAN-Modul HTML::Table. Genauso wäre es aber auch möglich eine Role zu definieren die Text::ASCIITable verwendet, oder eines der CSV-Module etc. - wobei jedes als Role in jede beliebige Klasse eingebunden werden könnte, die die Methode `_as_aoa()` zur Verfügung stellt. Wir erreichen so also eine deutlich höhere Wiederverwendbarkeit von Programmcode.

```
package Role_HTML_Table;

use Moose::Role;
use HTML::Table;

requires '_as_aoa';

sub to_html {
    my $self = shift;
    my $table = HTML::Table->new(
        -class=>'emb_table',
        -evenrowclass=>'even',
        -oddrowclass=>'odd',
        -data => $self->_as_aoa
    );
    return $table->getTable;
}
```

Listing 7

Hier möchte ich noch eine klare Abgrenzung zu den in Java verwendeten Interfaces machen. Interfaces sind rein abstrakt und erzwingen die konkrete Implementierung der gewünschten Methode in jeder einzelnen Klasse. Zumal sie ein Kunstgriff sind, da Java keine Mehrfachvererbung kennt. Roles hingegen beinhalten die konkrete Implementierung der gewünschten Methode, erwarten also nur das ihnen auf einem vereinbarten Weg, in unserem Fall die Methode `_as_aoa()`, die benötigten Daten zur Verfügung gestellt werden

Die Implementierung in die ursprünglische Klasse wollen wir uns abschließend betrachten (siehe Listing 8).

Zu Beginn der Klasse `TemperatureTable` erfolgt die Einbindung der gewünschten Roles mit dem Keyword with:

```
with 'Role_HTML_Table';
with 'Role_ASCII_Table';
```

Gegen Ende der Klasse sehen wir die geforderte Methode `_as_aoa()`, die wir als Vorraussetzung für die Einbindung der Role definiert haben. Sie macht wirklich nichts anderes als eine einfache Tabelle als Array_of_Arrays zu erstellen und zurückzugeben.

Wenn wir uns nun das Ende des Skriptes betrachten, sehen wir, dass der Aufruf der Methode `to_html()` genauso erfolgt, als wäre die Methode in der Klasse selbst definiert, oder geerbt:

```
print $table->to_html;
```

```
#!/usr/bin/perl

use strict;
use warnings;

package TemperatureProbe;
use Moose;
use Moose::Util::TypeConstraints;

subtype Day
    => as Num
    => where { $_ >= 1 and $_ <= 31 };

has 'day'    =>  ( isa => 'Day', is => 'rw' );
has 'max'    =>  ( isa => 'Num', is => 'rw' );
has 'min'    =>  ( isa => 'Num', is => 'rw' );
has 'avg'    =>  ( isa => 'Num', is => 'rw' );

package TemperatureTable;
use Moose;
use Moose::Util::TypeConstraints;

with 'Role_HTML_Table';
with 'Role_ASCII_Table';

subtype ProbeList => as ArrayRef => where {
        ( blessed($_) && $_->isa('TemperatureProbe') || return ) for @$_ ; 1
};

has 'table' =>  ( isa => 'ProbeList', is => 'rw', default => sub { [] } );

sub parse_file {
    my $self = shift;
    my $fh = shift or die "parse_file() needs a filehandle to parse on.";
    while (<$fh>) {
        next unless /^\s*\d+/;
        my ($day, $max, $min, $avg) = /^\s*(\d+)\s*(\d+)\*?\s*(\d+)\*?\s*(\d+)/;
        my $current = TemperatureProbe->new(
            day => $day,
            max => $max,
            min => $min,
            avg => $avg
        );
        push @{$self->table}, $current;
    }
}

sub get_min_spread_day {
    my $self = shift;
    my %msdy = ( 'day' => 0, 'max' => 1000, 'min' => 0 );
    for my $probe (@{$self->{table}}) {
        if ($probe->max - $probe->min < $msdy{'max'} - $msdy{'min'}) {
            @msdy{'day', 'max', 'min'} = ($probe->day, $probe->max, $probe->min)
        }
    }
    return $msdy{'day'};
}

sub _as_aoa {
    my $self = shift;
    return [ map { [ $_->day, $_->max, $_->min, $_->avg ] } @{$self->table} ];
}

package main;

my $table = TemperatureTable->new;
$table->parse_file(*DATA);

print $table->get_min_spread_day . "\n";
# print $table->to_html;
print $table->to_ascii_table;
```

Listing 8

Zu guter Letzt: Einschränkungen

Ich sehe schon das erste Stirnrunzeln bei dieser Überschrift. Ja, es gibt auch Einschränkungen. Und zur Abwechslung mal im besten Sinne. Denn Moose erlaubt die Verwendung von Constraints, die es uns erlauben sehr genau zu sagen was ein Attribut sein kann – und was nicht. Im obigen Skript habe ich dieses Feature bereits stellenweise verwendet, wie dem aufmerksamen Leser nicht entgangen sein dürfte.

Im Vergleich zur strengen Typisierung erlaubt uns Moose via `use Moose::Util::TypeConstraints;` sehr genau zu definieren, wie wir uns einen `subtype` vorstellen. In der Klasse `TemperaturProbe` sehen wir ein recht einfaches Beispiel, wo ein Subtype für das Attribut 'day' definiert wird:

```
subtype Day
     => as Num
     => where { $_ >= 1 and $_ <= 31 };
has 'day'    =>  ( isa => 'Day', is => 'rw' );
```

Der Subtype 'Day' wird als numerischer Wert definiert, mit der Einschränkung, dass der Wert größer/gleich 1 sein muss und kleiner/gleich 31 ist. Entsprechend könnten wir auch einen Subtype für die Temperaturen definieren, die sich auch in einem bestimmten Bereich bewegen dürfen etc.

Eine weitere Verwendung von `Moose::Util::TypeConstraints` findet sich in der Klasse `TemperatureTable` (siehe Listing 9).

Hier wird ein Subtype ProbeList definiert, bei dem es sich um eine Referenz auf ein Array handelt, mit der Einschränkung, dass jedes Element der Liste ein Objekt vom Typ `TemperatureProbe` sein muss. Wenn dies nicht zutrifft, bricht die anonyme Subroutine einfach ab. Läuft die Schleife komplett durch, wird 1 zurückgegeben, was den Test als erfolgreich verlaufen rückmeldet.

Zu guter Letzt bleibt mir nur den Autoren des Moduls Moose zu danken, die Perl5 schon heute ein wenig näher an die Möglichkeiten von Perl6 bringen und uns Programmierern ein Werkzeug an die Hand geben, das die Arbeit mit objektorientiertem Perl deutlich angenehmer und effizienter gestaltet.

Moose ist darüber hinaus hervorragend im CPAN dokumentiert – insbesondere existiert ein ausführliches Kochbuch unter Moose::Cookbook, dass in mehreren Schritten die Möglichkeiten von Moose beleuchtet und dabei über die Möglichkeiten dieses Artikel weit hinausgeht.

Ronnie Neumann

```
subtype ProbeList => as ArrayRef => where {
        ( blessed($_) && $_->isa('TemperatureProbe') || return ) for @$_ ; 1
};
has 'table' =>  ( isa => 'ProbeList', is => 'rw', default => sub { [] } );
```

Listing 9

TPF-TICKER

PM News: Ruhr.pm startet Jobbörse

Die Perlmonger-Gruppe von der Ruhr hat eine Jobbörse auf ihrer Homepage veröffentlicht. Darin ist eine Umkreissuche auf Basis der eingegebenen Postleitzahl enthalten. Zur Zeit ist der Schwerpunkt der Jobbörse noch das Ruhrgebiet, aber das könnte sich bald ändern.

Grant für MediaWiki-Parser

Shlomi Fish hat erneut einen Grant von der Perl Foundation erhalten. Die Förderung ist für ein Modul, mit dem Texte in MediaWiki-Syntax geparst werden können. Weiterhin muss Fish ein Kwiki-Plugin schreiben, mit dem Kwiki-Seiten auch mit MediaWiki-Syntax geschrieben werden können.

CPAN News IV

WebService::Validator::HTML::W3C

Für Webentwickler dürfte dieses Modul ziemlich interessant sein: WebService::Validator::HTML::W3C. Damit ist es möglich, den W3C-Validator in das eigene Perl-Programm einzubauen. So kann bei der Entwicklung von Webanwendungen von Anfang an darauf geachtet werden, dass das HTML W3C-valide ist.

Man kann die Ausgabe an eigene Bedürfnisse anpassen...

Die Option „detailed“ muss auf „true“ gesetzt werden, wenn später im Programm die errors-Funktion verwendet werden soll, weil nur dann die einzelnen Fehlermeldungen gespeichert werden.

```
use WebService::Validator::HTML::W3C;

my $uri       = 'http://foo-magazin.de';
my $validator = WebService::Validator::HTML::
W3C->new(
    detailed  => 1,
);

if( $validator->validate( $uri ) ){
  if( not $validator->is _ valid ){
    my %hash;
    for my $error ( @{$validator->errors} ) {
      $hash{ $error->msg }++;
    }

    for my $key ( sort keys %hash ){
      my $err = $hash{$key};
      print sprintf "%dx %s\n", $err, $key;
    }
  }
}
```

Dir::Self

Wer kennt sie nicht, die Variablen __FILE__ und __LINE__? Dir::Self führt eine weitere solche Variable ein: __DIR__. Damit bekommt man den Pfad zum Verzeichnis, in dem die Sourcen liegen. Der große Unterschied zu FindBin besteht darin, dass Module den Pfad zum eigenen Verzeichnis bekommen, während $FindBin::Bin den Verzeichnisnamen liefert, in dem das aufgerufene Skript liegt.

```
#!/usr/bin/perl

use strict;
use warnings;
use Dir::Self;

print "Hauptskript ist im Verzeichnis ",
_ _ DIR _ _ ,"\n";

require Test::Package;


package Test::Package;

use strict;
use warnings;
use Dir::Self;

print "TestPackage ist im Verzeichnis ",
_ _ DIR _ _ ,"\n";

1;
```

JavaScript::Writer

Mit JavaScript::Writer kann man JavaScript aus einem Perl-Programm heraus erzeugen. Das Modul ist wohl noch ganz am Anfang seiner Entwicklung, denn es hat noch ein paar Defizite. Aber es ist ganz praktisch, wenn JavaScript „on-the-fly“ erzeugt werden soll. Ganz ohne JavaScript-Kenntnisse kommt man in der Regel aber nicht aus, da man Funktionen kennen muss und noch einiges mehr.
Einige Funktionalitäten fehlen noch beziehungsweise müssen in der Dokumentation eingetragen werden. Wenn sich das Modul weiterentwickelt, ist es sicherlich irgendwann sehr gut für Code-Generierung geeignet.

```
use JavaScript::Writer;

my $js = JavaScript::Writer->new;
my $f  = $js->function(\&create _ js _ func-
tion);
print $f->as _ string;

sub create _ js _ function{
    my ($js) = @ _ ;

    $js->var( 'test', '$foo - Perl-Magazin' );
    $js->var( 'i', 10 );
    $js->if( 'i < 20', sub{
       shift->alert('Kleiner als 20') } );
    $js->else(sub{
       shift->alert( 'Groesser als 20') } );
}
```

Acme::Pythonic

Oft wird Perl mit Python verglichen und die Python-Anhänger schwören auf die Möglichkeit, Semikolons und geschweifte Klammern wegzulassen. Wer dies in Perl vermisst und diese Python-Art in seine Programme einfließen lassen will, kann mit Acme::Pythonic arbeiten. Alles was nach der Anweisung use Acme::Pythonic steht, kann ohne Semikolon und Block-Klammern geschrieben werden.
Insgesamt ist das Modul eher als „Spass“ zu sehen, aber es ist mal ganz nett um ein Gefühl für die Pythonschreibweise zu bekommen.

```
use Acme::Pythonic;

my $test = 'hallo welt'
my $i    = 10

if $i < 10:
    print "\$i ist kleiner als 10\n"
print $test,"\n"

hallo( $test )

sub hallo:
    $test =~ s/welt/\$foo/
    print $test,"\n"
```

Geo::GoogleEarth::Document

GoogleEarth bietet – über Dateien im KML-Format – die Möglichkeit, Informationen über bestimmte Orte zu speichern. Diese Dateien können dann in GoogleEarth oder GoogleMaps eingebunden werden. So kann man zum Beispiel eine KML-Datei mit interessanten Punkten vom Urlaub anfertigen und seinen Freunden weitergeben, damit diese wissen, wo sie im nächsten Urlaub hinfahren sollten. Oder man kann die Fussball-Stadien der Umgebung auflisten.

```
use Geo::GoogleEarth::Document;

my %placemarks = (
    'Commerzbank-Arena' => {
        lat  => 50.06874400,
        lon  => 8.64524200
    },
    'Bruchweg-Stadion' => {
        lat  => 50.0009717881533,
        lon  => 8.245587306186593,
    },
);

my $doc = Geo::GoogleEarth::Document->new;

for my $place ( keys %placemarks ){
    $doc->Placemark(
        name => $place,
        %{ $placemarks{$place} },
    );
}

print $doc->render;
```

App::Grepl

In welchem Skript geht es um die „Commerzbank-Arena“? App::Grepl liefert die Möglichkeit, in bestimmten Programmbereichen nach Pattern zu suchen. Dabei kann angegeben werden, welche Bereiche berücksichtigt werden sollen. Möglich ist es, in der Dokumentation, in Heredocs, Strings und Kommentaren zu suchen.
Das Ergebnis ist ganz interessant wenn man das Beispielskript auf den CPAN_News-Ordner der Beispielcodes dieser $foo-Ausgabe anwendet. Im zweiten Durchlauf nicht nur in 'pod', sondern auch in 'quote' suchen lassen...

```
use App::Grepl;
use Dir::Self;

my $grep = App::Grepl->new({
    dir       => __DIR__,
    look_for => [ 'pod' ],
    pattern   => qr/Commerzbank-Arena/,
});

$grep->search;
```


VoIP mit Net::SIP

Was ist SIP

SIP ist der IETF Standard für VoIP, d.h. Telefonie etc über Internet. Daneben gibt es noch den H.323 Standard von der ITU und das proprietäre Skype. SIP ist in Deutschland inzwischen stark verbreitet, meist von den Kunden direkt mittels Routern wie Fritz!BoxFON oder transparent für den Kunden in sogenannten NGN (New Generation Networks).

SIP beschreibt ausschließlich die Verbindung. Der Transport der Nutzdaten (Sprache, Video...) erfolgt i.A. über RTP. Über SIP tauschen die Gegenstellen SDP Pakete aus, um die Endpunkte für den Transport der Nutzdaten festzulegen.

SIP selber ist ein umfangreiches Protokoll. Die aktuelle Version ist definiert in RFC3261, zur Telefonie sind aber noch mindestens RFC2327 und RFC3264 zu verstehen, welche die Herstellung von Datenverbindungen mittels SDP beschreiben. Weiterhin gibt es viele Erweiterungen, Auslegungen etc, die in vielen weiteren RFCs beschrieben sind.

Was ist Net::SIP

Net::SIP ist eine Sammlung von Modulen, um in Perl mit SIP arbeiten zu können. Es läuft auf perl5.8 und perl5.9. Einzige Abhängigkeit außer den CORE Modulen ist Net::DNS.

Net::SIP ist entstanden im Zusammenhang mit der Implementation eines VoIP/SIP Gateways für das GeNUGate Hochsicherheitsfirewall Produkt (siehe http://www.genua.de).

Was kann Net::SIP

Net::SIP implementiert ausschließlich SIP, sowie die für SIP notwendigen Teile von SDP. Es gibt eine minimale Implementation, um RTP Daten zu übermitteln. Es implementiert keine Schnittstelle zu Sprach- oder Videodaten, d.h. mit Net::SIP alleine ist es nicht möglich ein SIP Telefon zu erstellen.

Man kann jedoch Net::SIP nutzen um z.B.:

- Jemanden anrufen und vorgefertige RTP Daten verschicken. Sinnvoll für automatische Ansagen (für die es hoffentlich auch Nutzung außer Werbung gibt).
- Anrufe entgegennehmen und mit vorgefertigten Ansagen beantworten (z.B bei Anruf Podcast), dabei evtl. Sprachdaten der Gegenstelle speichern (Anrufbeantworter).
- Verbindungen mit externen RTP Quellen vermitteln (z.B. bei Anruf wird Verbindung zu Internetradio hergestellt). Hier ist u.U. jedoch ein Programm nötig, welches die Codes angleicht.
- Verbindungen zwischen 2 SIP Parteien aufbauen (3pcc, d.h. 3rd Party Call Control).
- Nachrichten verschicken (SMS), sofern das SIP Endgerät über entsprechende Möglichkeiten verfügt.
- VoIP/SIP über Firewalls hinweg vermitteln durch Umschreiben der SIP/SDP Pakete und Transport der RTP Daten.
- Aufbau einer einfachen SIP Zentrale mit Registrar und Proxy, an die die SIP Telefone angeschlossen werden. Mit Perl kann man dann komfortabel (zumindest das, was Perl Programmierer als komfortabel empfinden) Policies z.B. basie-

rend auf der Nummer der Gegenstelle implementieren, d.h. Anrufe blockieren, an mehrere Telefone weiterleiten, auf Anrufbeantworter lenken, mit einer bestimmten Ansage versehen etc. Auch Warteschleifenmusik sollte gehen.

Man kann mit den Mitteln von Net::SIP jedoch keine Umwandlung von Codecs, Mixen von Sprachdaten (für Konferenzanrufe) o.ä. vornehmen, wie es z.B Asterisk kann. Auch die Reaktion auf Usereingaben über die Tastatur (DTMF) bleibt Asterisk vorbehalten. Es ist jedoch möglich, Anrufe nach in Perl implementierten Kriterien an den Asterisk SIP Proxy weiterzuleiten.

Momentan hat Net::SIP Unterstützung für SIP über UDP und IPv4. Unterstützung für SIP über TCP ist praktisch nichts getestet, SIPS (verschlüsselt) ist nicht unterstützt. Nicht unterstützt ist auch IPv6. In der Praxis sollten diese Limitierungen z.Z. kein Problem darstellen, da SIP fast ausschließlich über UDP/IPv4 benutzt wird.

Struktur von Net::SIP

Auf der untersten Ebene sind die Klassen zur Verarbeitung der SIP Pakete, d.h. Net::SIP::Packet und die Spezialisierungen Net::SIP::Request und L<Net::SIP::Response>.

Darauf aufbauend implementiert Net::SIP::Endpoint eine SIP Endstelle, während Net::SIP::Registrar einen einfachen Registrar und Net::SIP::StatelessProxy einen Proxy konfiguriert. Mittels Net::SIP::ReceiveChain können Programme konstruiert werden, die sich je nach SIP Request als Endstelle, Registrar oder Proxy verhalten, bei Bedarf auch mit Authorisierung durch Net::SIP::Authorize.

Der Empfang und das Verschicken von Daten sowie evtl. nötige Wiederholungen beim Verschicken sind in Net::SIP::Dispatcher und Net::SIP::Leg implementiert.

Bis zu dieser Stelle ist der Code so aufgebaut, das er voll asynchron ausgeführt werden kann, was u.a. für den SIP-Firewall bei GeNUA nötig war, da dieser aus Performancegründen in einem Eventloop (d.h. keine Threads und kein Forking) implementiert ist. Allerdings ist eine eventbasierte Arbeitsweise für einfache Aufgaben zu umständlich.
Daher gibt es als oberste Ebene Net::SIP::Simple, welches ein einfaches Interface für Standardaufgaben (anrufen, Anruf annehmen, Registrar bauen...) bereitstellt und auch über eine einfache Implementation zum Versenden und Empfangen von RTP Daten verfügt. Damit lassen sich prima Tests schreiben (was der primäre Zweck war), es ist jedoch auch leistungsfähig genug für komplexere Applikationen.

Beispiele

Die *.raw Dateien in den Beispielen sind mit dem PCMU/8000 Codec codierte Dateien - das ist das einzige Format was die simple RTP Implementation zur Zeit empfangen und verschicken kann.Wandeln kann man solche Dateien mit sox, z.B.siehe Listing 1.

Ein Proxy mit Registrarfunktion

Dieses Beispiel ist ein Proxy, bei dem man sich auch registrieren kann. Wenn ein neuer Anruf reinkommt, wird zuerst geschaut, ob die Zieladresse lokal registriert ist und dann zu dieser vermittelt, ansonsten der Anruf nach außen weitervermittelt (siehe Listing 2).

Genau einen Anruf entgegennehmen

Das Programm in Listing 3 registriert sich an dem obigen Registrar/Proxy als ‘sip:00@me.local‘. Dann wartet es auf einen Anruf, spielt diesem die Message aus welcome.raw vor und speichert gleichzeitig die Sprachdaten vom Anrufer in data.raw. Außerdem läßt es nur bestimmte Anrufer zu (z.B. ‘sip:31@me.local‘). Nachdem eine Nachricht empfangen wurde beendet es sich.

Anrufen

Das Programm in Listing 4 benutzt als Proxy den obigen Registrar/Proxy und ruft ‘00@me.local‘, also den simplen Anrufbeantworter aus dem letzten Beispiel, an. Dort hinterläßt es die Message aus message.raw und beendet dann die Verbindung.

```
sox input.ogg -t raw -b -U -c 1 -r 8000 output.raw
sox -t raw -b -U -c 1 -r 8000  input.raw output.ogg
```

Listing 1

Wie kann ich helfen

Die beste Hilfe ist momentan, Net::SIP zu nutzen und dadurch Schwachstellen der Implementierung bzw. Dokumentation oder Probleme bei der Zusammenarbeit mit bestimmten SIP Gegenstellen aufzudecken.

Ansonsten freue ich mich über sinnvolle Beispiele und auch über Links zu Projekten, die Net::SIP nutzen. Beides kann ich dann zum Projekt hinzufügen, um damit die Dokumentation und Motivation insbesondere für Neueinsteiger zu erhöhen.
Und außerdem würde ich sehen, was ich auf keinen Fall ändern darf, um Euren Code nicht zu brechen.

Weitere Informationen

Ich habe versucht, die Dokumentation zum Module ausführlich zu machen (wobei man wohl nie genug Dokumentation haben kann). Weiterhin sind in der Distribution Beispiele und funktionierende Applikationen für Anrufen, Anrufe entgegennehmen, 3rd Party Call Control, Anrufbeantworter und ein Proxy, der SIP über einen Firewall transportiert. Diese Beispiele werden bei der Installation des Modules nicht mit installiert, sind aber z.B. sichtbar auf (http://search.cpan.org/src/SULLR/Net-SIP-0.37/)
Weiterhin stehe ich gerne für Nachfragen, Problem- und Bugreports zu Verfügung und beantworte entsprechende Mails normalerweise innerhalb von wenigen Tagen.

Steffen Ullrich

```
use strict;
use Net::SIP;
my $ua = Net::SIP::Simple->new( leg => '192.168.178.2:5060' );
$ua->create_chain([
    $ua->create_registrar,
    $ua->create_stateless_proxy,
]);
$ua->loop;
```

Listing 2

```
use strict;
use Net::SIP;
my $ua = Net::SIP::Simple->new(
    from => 'sip:00@me.local',
    leg => '192.168.178.3:5060',
    registrar => '192.168.178.2:5060',
);
$ua->register;
my ($done,$call_created);
$ua->listen(
    cb_create => \$call_created,
    filter => sub {
        $call_created && return; # habe schon einen Anruf laufen
        return shift =~m/31\@/; # check gegen Absender
    },
    init_media => $ua->rtp( 'send_recv', 'welcome.raw','data.raw' ),
        recv_bye => \$done,
);
$ua->loop(\$done);
```

Listing 3

```
use strict;
  use Net::SIP;
  my $ua = Net::SIP::Simple->new(
      from => 'sip:31@fasel.com',
      outgoing_proxy => '192.168.178.2',
  );
  my $done;
  my $call = $ua->invite( 'sip:00@me.local',
      init_media => $ua->rtp( 'send_recv','message.raw' ),
      cb_rtp_done => \$done,
  );
  $call->loop( \$done );
  $call->bye;
```

Listing 4

Perl 6 Tutorial – Teil 1 – Allgemeine Grundlagen

Motive für ein neues Tutorial

Perl 6 materialisiert sich langsam vor unseren staunenden Augen und Neugierige, die bereits heute mit Hilfe des Pugs-Interpreters und der Dokumentation (Synopsen) ihre ersten Programme darin schreiben, stehen vor echten Hürden. Die Sprache verändert sich in Nuancen fast jede Woche, Pugs kennt noch nicht alle Befehle, und die Synopsen sind wahrlich keine leichte Bettlektüre. Sie sind umfangreich, mit Fachslang angereichert und eher darauf ausgerichtet komplizierte Sonderfälle auszuleuchten, als Anfängern die ersten Schritte zu erleichtern. Dafür bräuchte es eher ein Tutorial, dass seine Leser behutsam in die neue Sprache einführt und nur so wenig Wissen wie sinnvoll voraussetzt, damit so viele Interessierte wie möglich, sich die begeisternden Verbesserungen und Erweiterungen gegenüber Perl 5 aneignen können.

Dieser Artikel soll ein solches Tutorial sein und er wird mit jeder Folge ein kleines Teilgebiet mit Beispielen und in Einzelheiten dem geneigten Perl 6-Anfänger näher bringen. Denn Perl war immer darauf aus, dem erfahrenen Codepoeten die Arbeit zu leichtern und ist deswegen nicht an einem Tag vollständig erlernbar. Auch darin ist Perl 6 eine Steigerung von Perl 5. Um diesem Kurs einen größeren Nutzen zu entnehmen, empfehle ich es, beim Lesen die Beispiele, oder eigene Abwandlungen davon, mit Pugs auszuprobieren. Pugs findet ihr auf pugscode.org oder über perl.org. Hilfreich ist es auch, vorher den zweiteiligen Perl 6-Artikel gelesen zu haben, der in den vorigen beiden Ausgaben abgedruckt war. Dort wurde von der Entwicklung des Projektes und den linguistischen Konzepten der Sprache berichtet.

Hallo Perl 6

Fast jede Einführung in eine Programmiersprache beginnt mit einem „Hello World“. Auch wenn ich mir eigentlich was originelleres ausdenken wollte, bleibt es doch das kleinste Programm, dass etwas erfreulich sichtbares tut, also starten wir auf ausgetretenen?, nein bewährten Pfaden:

Perl 6:

```
say "Seid gegrüßt, ihr Perl 6 Programmierer.";
```
Listing 1

Dieses erste Programm gibt doch tatsächlich den genannten Text aus. Der Befehl ‘say‘ entspricht dem bekannten ‘print‘, fügt aber noch ans Ende der Ausgabe einen Zeilenumbruch an, der dem Standard des aktuellen Betriebssystems entspricht. Klar gibt es ‘print‘ weiterhin, aber zur Gewöhnung wird ab jetzt ‘say‘ verwendet. Ich kann aber bereits die Einwände der erfahreneren Perlschreiber hören: Ein gutes Skript beginnt mit strict und warnings. Und recht haben sie. In Perl 5 müsste das kleine Programm lauten:

Perl 5:

```
use strict;
use warnings;

print
 "Seid gegrüßt, ihr Perl 6 Programmierer.\n";
```
Listing 2

Aber in Perl 6 entfällt das, weil beide Pragmas jetzt standardmäßig aktiviert sind. Das spart nicht nur die 2 Zeilen Code, die in fast allen meiner Programme sind, sondern es hilft jenen, die nichts darüber Wissen, potenzielle Probleme eher zu sehen. Bei Bedarf kann ja beides, wie bekannt, mit ‘no strict;‘ und ‘no warnings;‘ abschaltet werden.

Runde und geschweifte Klammern

Der nächste Schritt ist in Tutorials oftmals die Einführung von Variablen, denen ein Wert zugewiesen wird, der gleich wieder ausgegeben wird.

Perl 6:

```
my $a = 3;
say "Ich jongliere mit $a Bällen.";
```
Listing 3

Da Skalare immer noch Variablen sind, die einen einzelnen Wert speichern (egal welchen Datentyps) und immer noch mit der Sigil ‘$‘ anfangen, gibt es hier nichts weiter zu erklären. Die Variable ist natürlich mit ‘my‘ als lexikalisch lokal deklariert, da ‘use strict‘ gilt. Vielleicht könnte ich ein ‘if‘ einfügen, da ich Artistik mit ein oder zwei Bällen nicht jonglieren nennen würde.

Perl 6:

```
my $a = 3;
if $a > 2 {
       say "Ich jongliere mit $a Bällen."
}
```
Listing 4

Und hier sehen wir schon einen Unterschied zu Perl 5, der angenehm auffällt. Der dem ‘if‘ folgende Term kann, muss aber nicht mehr, in runde Klammern gesetzt werden. Runde Klammern dienen in Perl 6 nur noch dem Gruppieren. Hat man jedoch keine verschachtelte Struktur, sondern nur eine einzelne Anweisung oder eine einfache Folge davon, werden sie nicht benötigt. Dies gilt überall. Auch ein Array, eine Variable die mehrere Werte speichert, kann nun ohne runde Klammern gefüttert werden.

Perl 6:

```
my @primzahlen = 2,3,5,7;
```
Listing 5

Dies ist auch möglich, weil das Komma jetzt ein listenerzeugender Operator ist. Die zweite, nicht ganz so offensichtliche Neuheit in Perl 6, die das Listing 4 zeigt, ist das Fehlen eines abschließenden Semikolons. Seit Perl 1.0 trennt das Semikolon einzelne Befehle. Lediglich der letzte Befehl innerhalb geschweifter Klammern (in einem Block), muss nicht mit einem Semikolon abgeschlossen werden. Auch das sieht man in Listing 4. Um sich nicht mehr merken zu müssen, wann nach einer schließenden geschweiften Klammer ein Semikolon zu stehen hat und wann nicht (diese Regeln wurden weitestgehend aus C übernommen), hat Larry beschloßen, dass jedes Semikolon nach geschweiften Klammern ab jetzt freiwillig (optional) ist. Bei ‘if‘-Blöcken gilt das auch in Perl 5, aber nicht z.B. bei ‘eval‘-Blöcken.

Perl 5:

```
eval {
   print "Finden sie alle 3 Unterschiede !\n"
};
```
Listing 6

Perl 6:

```
try {
   say "Finden sie alle 3 Unterschiede !"
}
```
Listing 7

Der dritte Unterschied ergab sich, weil ‘eval‘ jetzt nur noch Strings auswertet und für das Ausführen von Blöcken probehalber, nun ‘try‘ statt ‘eval‘ zu verwenden ist.

Ein nicht mehr ganz so freies Format

In Sachen Formatierung ist Perl allerdings etwas strenger geworden. Leerzeichen können nicht mehr überall nach belieben eingefügt werden. Natürlich ist die Breite der Einrückungen, sowie generell die Anzahl der Leerzeichen und Zeilenumbrüche zwischen 2 Bestandteilen eines Ausdrucks unerheblich.

Perl 6:

```
# Zuweisung eines Wertes
$b    = $a ;
# jetzt ich hab noch eine Kopie des Wertes
   $C = $a;
$d
=
$a
;            # und noch eine
```
Listing 8

Aber Leerzeichen und Zeilenumbrüche haben schon eine trennende Wirkung. Gerade bei Operatoren sollte man jetzt stärker darauf achten. Besonders der neue Präfix- Operator (‘=‘), der teilweise die beliebte Funktion des Diamantoperators (‘<>‘) übernimmt, kann leicht für eine Zuweisung gehalten werden. Oder auch folgende 2 Anweisungen laden zu unterhaltsamen Verwechslungen ein:

Perl 6:

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

Listing 9

Das gleiche Zeichen kann jetzt mehrere Operatoren darstellen, je nachdem, wie es von Leerzeichen umgeben ist:

Perl 6 (siehe Listing 10)

Diese Unterscheidungen sind nicht gänzlich neu, da beide Arten des Autoinkrementes schon in C bekannt sind. Unerwartete Probleme kann es für Umsteiger aber auch beim Schreiben von Arrays, Hashes und von Objekten geben. Folgende Schreibweisen werden nicht mehr erkannt:

Perl 5:

```
$array    [$i];
$hash     {$k};
$obj   -> methode ();
```

Listing 11

Wer seinen Quellcode so formatieren möchte, dass bestimmte Teilausdrücke oder die Zeilenenden genau untereinander stehen, braucht in Perl 6 die „long dot“ genannte Schreibweise:

Perl 6:

```
# ein Wert aus einem Array
@array\       .[$i];
# ein Wert aus einem Hash
%hash\      .{ $k };
$obj\   # hier kann ich Kommentare einfügen
     .methode ( ) ;       # Methodenaufruf
```

Listing 12

Blitzmerker haben aus diesem Code bereits geschlossen, dass Arrays nun immer mit ‘@‘ anfangen, Hashes nun immer mit ‘%‘ und das der Punkt das Objekt und den Namen der Methode trennt, wozu früher ein Pfeil (‘->‘) diente (z.B.:‘$obj.methode()‘). Dass der Backslash (‘\‘) die nachfolgenden Leerzeichen und Zeilenumbrüche „quotet“, also ihrer normalen Bedeutung enthebt, ist auch nichts wirklich Ungewöhnliches.

Das Stirnrunzeln beim Betrachter dieser Zeilen erwarte ich erst, sobald sein Hirn diese beiden Gedanken verbindet und die Frage auftaucht: Der Sliceoperator (die Klammern hinter einer Variable, mit deren Hilfe wir einzelne Werte, also Teile (slices), eines Arrays oder Hashes geliefert bekommen) ist doch nicht ernsthaft eine Methode des Objektes vom Typ Array oder Hash?

Doch, ist es. Perl 6 ist so objektorientiert wie Ruby, auch wenn es sein möglichstes tut, dass vor dem Benutzer zu verbergen. Zum Beispiel dadurch, in dem es oftmals den Punkt optional läßt. Statt dem auch möglichen ‘@array.[3]‘ reicht wie gewohnt ein ‘@array[3]‘. Nur in diesem Sonderfall, wenn man zwischen Objekt und Methode Leerzeichen einfügen will, muss der Punkt verwendet werden, um Doppeldeutigkeiten zu vermeiden. Der Backslash darf auch nicht fehlen, denn ein ‘.say‘ ist in Perl 6 eine andere Schreibweise für ‘$_.say‘ oder auch ‘say $_‘. Steht also ein Punkt vor einem Befehl, sucht Perl kein vorher erwähntes Objekt, dessen Methode hier aufgerufen werden soll, sondern interpretiert diesen Befehl als Methode der aktuellen Kontextvariable, die immernoch, wie in Perl 5, ‘$_‘ heißt.

```
++$b      # Präfix-Operator,   Autoinkrement vor Evaluierung
$b++      # Postfix-Operator,  Autoinkrement nach Evaluierung
+$a       # Präfix-Operator,   das selbe wie num $a, ähnlich int $a
$a + $b; # Infix-Operator,    Addition
=$b       # Präfix-Operator,   Zeile aus einem Datenstrom
$a = $b; # Infix-Operator,    Zuweisung
[$a]      # Circumfix-Opertor, Teilarray oder Reduktionsoperator
< $a>     # Circumfix-Opertor, qw()
<< $a >> # Circumfix-Opertor, interpolierendes qw()
```

Listing 10

```
my $a #( = 6; ) = 5;  #  es 5 wird zugewiesen
say "$a #(????) $a";  #( Ausgabe von '5 #(????) 5'
                          keine Kommentare innerhalb von Strings )

$affe#{{.schreit()}}.guckt()  # der schreit nicht, der guckt nur
@zahlen#({ ?? ).reverse;        reingefallen alles noch Kommentar
             }).sort;         # gibt sortierte Version des Arrays zurück
```

Listing 13

Noch Kommentare?

In allen bisherigen Beispielen war es zu sehen: Wie aus Perl 5, anderen Sprachen und Unix-Shells bekannt, leitet auch in Perl 6 die Raute (‘#‘) einen Kommentar ein, der mit der selben Zeile endet. Neu in Perl sind jedoch Kommentare, deren Ende sich bestimmen läßt. Diese können mitten in eine Zeile eingefügt werden oder sich über mehrere Zeilen erstrecken. Solche Kommentare beginnen auch mit einer Raute, der jedoch unmittelbar eine Klammer folgt. Als Klammern gelten die Zeichen ‘()‘, ‘[]‘,‘{}‘ und ‘<>‘, wobei natürlich die zugehörige schließende Klammer den Kommentar beendet. Dabei kann man stets die Klammern wählen, die sich optisch vom Quellcode abheben. Dies kennt man in Perl 5 schon von den Regulären Ausdrücken und vom Quoting, aber zur Verdeutlichung einige Beispiele (Perl 6 siehe Listing 13)

Die letzten beiden Beispiele zeigen, dass mehrere Klammern benutzt werden dürfen. Dabei hat man nur darauf zu achten, beim Abschließen des Kommentars genau die passende Art, Anzahl und Reihenfolge der Klammern zu verwenden. Manche mögen darin wieder neue Wege sehen, unleserlichen Quelltext zu schreiben, aber hinter dem Vorgestelltem steht vor allem eine neue Arbeitslogik des Perlinterpreters, die einige aufdringliche und einige leicht zu übersehende Problemfälle bereinigt.

Setz es in „Anführungszeichen“

Mit der Arbeitslogik meine ich das „one-pass parsing“, zu deutsch: ein Interpreter kann in einem Durchgang den Code vollständig erfassen. Mit Perl 5 ist das leider anders. Hier schaut der aufgebohrte Bison-Parser öfters vor und zurück, um zu erkennen, was der Programmierer grad von ihm möchte und wechselt dabei zwischen verschiedenen Zuständen, wie etwa: „Dies ist Kommentar“, oder „Dies ist gerade eine Anweisung“. Perl 6 kennt nicht die Sonderregeln und Doppeldeutigkeiten, welche dies notwendig machten. Das macht die Interpreter nicht nur schlanker und schneller, sondern auch narrensicherer. Beim Einlesen kann ein Perl 6-Interpreter die Quellen in immer kleinere Bedeutungseinheiten aufspalten. Selbst wenn eine dieser Bedeutungseinheiten keinen Sinn ergibt (eine Katze über die Tastatur lief), beeinträchtigt dies nicht den Rest des Programmes. Ein Perl 5-Beispiel mit einer Regex, die mit Kommentaren versehen ist zeigt, welche hinterhältigen Fehler die alte Lesart bisher erlaubte:

Perl 5:

```
/ (      # begin der capture
    \w+ # alphanumerische Zeichen
  |     # nächste Alternative
...
  ) /x;
```

Listing 14

Der Slash (‘/‘) im Kommentar läßt perl vermuten, hier sei die Regex zu Ende und wird vom Nachfolgendem sehr sehr verwirrt. In Perl 6 haben Kommentare keinen Einfluß mehr auf Anweisungen und auch Dokumentation (POD) kann überall eingefügt werden, ohne das es mißverstanden wird. Auch öffnende Klammern denen ihr Gegenstück fehlt, werden in Perl nun wohlwollend übergangen. Allerdings ist im täglichen Gebrauch vor allem folgende Konsequenz dieser Technik wirklich praktisch:

Perl 6:

```
say "Und der Gewinner ist %gewinner{"name"}.";
```

Listing 15

Mit Perl 5 ginge das nicht ansatzweise, weil perl die zweiten Anführungsstriche als Textende ansieht, als Ende des Zustandes: „Ich lese gerade einen Text von links nach rechts.“. In Perl 6 werden jedoch zuerst die äußeren Anführungsstriche als Begrenzung des auszugebenden Textes erkannt und danach die darin zu interpolierende Hashvariable %gewinner, von der nur der Wert mit dem Schlüssel „name“ gebraucht wird.

Dies soll für einen ersten Rundgang genügen, durch die allgemeinen Grundlagen von Perl 6. In der nächsten Folge werden Opertoren zur Veränderung und dem Vergleich von Skalarwerten das Hauptthema sein.

Herbert Breunung

Daten Scheibchenweise … (Slices)

Ein nützliches Feature von Perl ist es, sogenannte „Slices“ verwenden zu können. Wörtlich übersetzt heißt es „Scheiben“. Diese „Scheiben“ sind bei Listen/Arrays häufig gewünscht.

Slices im Alltag

Folgende Situation: Eine Person A war beim Arzt und benötigt Medikamente. A geht zur Apotheke X, um diese Sachen zu besorgen. X ist wie ein großes Array (1. Element: Nasenspray, 2. Element: Aspirin,...). A möchte natürlich nicht die ganze Apotheke (nicht das ganze Array) haben, sondern nur einzelne Sachen daraus. A hat jetzt zwei Möglichkeiten:

1. A sagt dem Apotheker ein Medikament, lässt den Apotheker das Medikament holen, nennt danach das nächste Element und so weiter bis A alle Sachen bekommen hat.

2. A gibt dem Apotheker eine Liste mit den Arzneinamen und lässt den Apotheker alle Medikamente holen. Der Apotheker kommt mit einem ganzen Korb voll mit den Arzneipäckchen.

Slices in Perl

So etwas kann ganz einfach in Perl nachgebildet werden, wie die Listings 1 bis 5 zeigen.

Beim Heraussuchen der Slices (@apotheke[@rezept]:) muss das „@“ vor apotheke stehen, weil diese Operation eine Liste zurückliefert (im Gegensatz zu dem Zugriff auf einzelne Elemente – $apotheke[$index]).

Wichtig ist auch, dass die Elemente in der gleichen Reihenfolge zurückgeliefert werden, wie die Indexliste ist, in diesem Fall also „Nasenspray“ und dann „Hustensaft“. Wäre das Rezept anders herum (my @rezept = (2,0)), wäre zuerst der Hustensaft im Korb und dann das Nasenspray.

Hashslices

Analog zu den Arrayslices oben gibt es in Perl auch noch die Hashslices, die aber im Prinzip genauso funktionieren. Hier eine etwas andere Situation:

```
# die Apotheke
my @apotheke = (‘Nasenspray‘, ‘Aspirin‘, ‘Hustensaft‘, ‘Allergietabletten‘);

# A benötigt folgende Medikamente
# hier sind aber die Arzneinamen schon
# durch den Index in der Apotheke ersetzt
my @rezept = (0,2);

# A gibt dem Apotheker und bekommt den Korb voll mit Medikamenten
my @korb = @apotheke[ @rezept ];

print $ _ ,"\n" for @korb;
```

Listing 1

Ein Rechtsanwalt kommt in das Gerichtsarchiv und möchte die Unterlagen zu mehreren Fällen einsehen. Dazu hat er sich in seiner Kanzlei schon eine Liste mit Aktenzeichen angefertigt. Auch hier hat der Anwalt wieder die zwei Möglichkeiten, den Archivar einzeln losschicken oder dem Archivar einfach die Liste geben und der Archivar kommt mit einer ganzen Ladung an Aktenordnern zurück. Die erste Möglichkeit würden den Archivar wahrscheinlich ärgern, die zweite Möglichkeit ist aber optimal.

Das ganze in Perl:

```
my %archiv = (
    az0815 => ‘Akte zum Aktenzeichen 0815‘,
    az2373 => ‘Akte zum Aktenzeichen 2373‘,
    az2371 => ‘Akte zum Aktenzeichen 2371‘,
);

my @liste = qw(az0815 az2371);

my @akten = @archive{ @liste }
```

Listing 2

Slices als lvalue

Die Slices kann man nicht nur für das Auslesen von Daten verwenden, sondern auch für Zuweisungen. Soll zum Beispiel ein Hash verwendet werden, der später für eine reine Existenzüberprüfung dient, kann man das so machen:

```
my @keys = qw(key1 key2 key3);
my %hash;

@hash{ @keys } = (1) x scalar @keys;
```

Listing 3

Jetzt wird für jeden Schlüssel ein Eintrag im Hash erstellt. Jeder Schlüssel soll den Wert „1“ erhalten. Deswegen muss eine Liste mit „1“en zugewiesen werden.

Ein häufiger Fehler, der gemacht wird, ist folgender:

```
@hash{ @keys } = 1;
```

Listing 4

Hier wird aber nicht für jeden Schlüssel ein „1“ gespeichert, sondern nur für den ersten Schlüssel. Die restlichen Schlüssel erhalten den Wert „undef“. Mit dem „(1) x scalar @keys“ wird eine Liste erzeugt, die genauso lang ist wie die Liste der Schlüssel.

Analog funktioniert es für Arrayslices.

```
my @array = (1..10);
@array[2,4,6] = (99,98,97);
```

Listing 5

Hier werden in dem Array die Elemente mit den Indizes 2,4,6 durch die Zahlen in der zugewiesenen Liste ersetzt. $array[2] ist dann 99, $array[4] ist dann 98 und so weiter.

Renée Bäcker

TPF-TICKER

TPF: Vorstand wurde neu gewählt

Im Vorstand der Perl-Foundation gab es einige Änderungen. Bei den Wahlen im August 2007 gab es ein paar Veränderungen: Richard Dice übernahm das Amt des Präsidenten von Bill Odom, der zum neuen Vorsitzenden gewählt wurde. Jim Brandt wird neuer Vize-Präsident. Im Amt bestätigt wurden Kurt DeMaagd als Schatzmeister und Allison Randal sowie Nat Torkington als weitere Vorstandsmitglieder.

Richard Dice bleibt weiterhin Vorsitzender des „Steering Committees“ und Jim Brandt im „Conference Committee“ - bis zu den nächsten Wahlen in den Kommitees.

Kevin Lenzo, der bisher der Vorsitzende war, bleibt als Nicht-Stimmberechtigtes Mitglied dem Vorstand erhalten und trägt jetzt den Titel „Director Eremitus“.

Design von Code trennen – HTML::Template::Compiled

Zu den guten Sitten bei der Entwicklung von Webapplikationen gehört es, HTML-Code von Applikations-Code zu trennen. Neben „übermächtigen“ Templating-Systemen wie `Mason` [1] oder `Template-Toolkit` [2] gibt es auch „leichtgewichtige“ Systeme wie `HTML::Template::Compiled` [3]. Hier wird gezeigt, wie man das Modul verwendet.

Sehr einfacher HTML-Text wird häufig mit dem Modul `CGI.pm` erzeugt, aber sobald das komplexer wird, hat man viele verschachtelte Funktionsaufrufe. Übersichtlicher wird das durch „Here-Dokumente“ - aber auch das ist keine richtige Lösung, wenn es um komplexeren HTML-Code geht.

Wenn Applikations-Code mit HTML-Code vermischt wird, geht die Übersichtlichkeit schnell verloren. Es gibt einen richtigen Bruch im Quelltext. Da macht das Warten der Anwendung keinen Spaß.

Und soll jede Seite anders aussehen, passiert es schonmal, dass für zwei sehr ähnliche Sachen der gleiche Code geschrieben wird - bis auf den HTML-Teil. So etwas kann leicht durch das Verwenden eines Templating-Systems vermieden werden.

Ich persönlich habe mich für `HTML::Template::Compiled` (HTC) entschieden, weil es ziemlich schnell ist, nicht zu mächtig und für meine Bedürfnisse genau richtig ist. Es gibt aber nicht **das** Template-System. Es muss jeder für sich entscheiden, mit welchem System er/sie zurecht kommt. Bei Apache findet man einen kleinen Vergleich mit verschiedenen Systemen [4].

Die Trennung von HTML und Applikations-Code hat auch den Vorteil, dass ein Designer sich um das Aussehen kümmern kann und der Programmierer davon nicht betroffen ist. Es muss also nicht befürchtet werden, dass nach einer Design-Veränderung das Programm nicht mehr funktioniert.

Was steckt hinter HTC?

Das Modul wurde von Tina Müller entwickelt und beinhaltet einige Ideen von `HTML::Template`, das von Sam Tregar geschrieben wurde. Allerdings wurden einige Features weggelassen, dafür sind viele wichtige Features dazugekommen.

HTC compiliert das Template in Perl-Code. Dadurch wird das ganze System flexibler. In HTC ist auch ein Plugin-Mechanismus eingebaut, so dass es möglich ist, Plugins wie zum Beispiel für Inline-Images zu verwenden.

Die Parameter werden erst beim Aufruf von `output` ersetzt.

Wie wird das Modul verwendet?

Das Modul ist objektorientiert und wird auch so verwendet. Als erstes wird ein Objekt der Klasse `HTML::Template::Compiled` erzeugt, dem der Pfad zum Template übergeben wird.

```
use HTML::Template::Compiled;
my $obj = HTML::Template::Compiled->new(
        filename => '/path/to/template.tmpl'
);
```

Die Methode, mit der am meisten gearbeitet wird, ist `param`. Damit übergibt man dem Objekt die Werte für die Platzhalter.

```
$obj->param(PLATZHALTER => 'wert');
```

Für die Ausgabe wird dann nur noch `$obj->output()` aufgerufen, das die Ausgabe als String zurückliefert. Erst bei diesem Aufruf werden die Platzhalter tatsächlich ersetzt.

Tags im Template

Jetzt kommt der wichtigste Teil: Wie markiert man im Template, was wohin soll? Und wie kennzeichne ich, dass hier Mehrfach das „gleiche“ stehen soll? Für diese verschiedenen Sachen gibt es verschiedene Tags.

Da nicht jeder den gleichen Geschmack beim Aussehen der Tags hat, können die Tagstyles gewählt werden. In diesem Artikel verwende ich das Aussehen von HTML-Kommentaren.

Variable

Das einfachste sind Variablen. Hier wird einfach der Wert ersetzt. Ein ganz einfaches Template wäre

```
<!-- TMPL_VAR NAME=PLATZHALTER -->
```

Und um hier einen Wert einzufügen, reicht ein

```
$obj->param(PLATZHALTER => 'wert');
```

Schleifen

Natürlich gibt es auch bei `HTML::Template::Compiled` Schleifen. Man kann ja nicht immer wissen, wie viele Werte angezeigt werden. Zum Beispiel wenn Daten eines Gästebuchs angezeigt werden sollen. Da sich die Zahl der Einträge ständig ändert, kann man ja nicht jedesmal das Template anpassen.

Um mit Schleifen arbeiten zu können, sollte man sich etwas mit Referenzen auskennen. HTC benötigt für die Schleifen nämlich eine Arrayreferenz, in der für jeden Eintrag eine Hashreferenz vorhanden ist.

Loop

Die einfachste Form der Schleife ist die `LOOP`. Das entspricht einer `for`-Schleife in Perl.

```
<!-- TMPL_LOOP NAME=TESTLOOP -->
Wert: <!-- TMPL_VAR NAME=VALUE -->
<!-- /TMPL_LOOP -->
```

Wie man sieht gibt es für diesen Tag – ähnlich wie bei HTML – ein schließendes Tag.

Im Skript sieht die Ersetzung dann so aus:

```
$obj->param(TESTLOOP => [
{VALUE => 'Wert1',
{VALUE => 'Wert2',
  ]);
```

While

Die while-Schleife eignet sich ganz gut dazu, Ergebnisse aus Datenbank-Abfragen darzustellen.

Bedingungen

Die Bedingungen sind sehr nützlich, wenn je nachdem was passiert eine andere Information auftauchen soll. Allerdings sollte man aufpassen, dass man nicht wieder zu sehr Darstellung und Logik miteinander vermischt.

Man sollte also immer abwägen, ob man eine Bedingung ins Template schreibt oder doch lieber im Perl-Code integriert.

If

Das `if`-Tag setze ich sehr häufig ein. So kann ich zum Beispiel bestimmte Bereich nur einblenden, wenn eine bestimmte Bedingung erfüllt ist. So zum Beispiel, wenn auf einer Login-Seite eine Fehlermeldung und ein Link kommen soll - aber nur wenn es eine Fehlermeldung gibt. Das sieht dann so aus:

```
<!-- TMPL_IF NAME=ERROR -->
Es ist ein Fehler aufgetreten. Wenn Sie Ihr
Passwort vergessen haben, klicken Sie bitte
<a href="passwort_vergessen.html">hier</a>
<!-- /TMPL_IF -->
```

So kann ich die Anzeige „ausschalten“, indem ich für `ERROR` einfach keinen Parameter übergebe und wenn irgendwo ein Fehler aufgetreten ist, mache ich einfach

```
$obj->param(ERROR => 1);
```

Wie sollte es auch anders sein - es gibt natürlich auch einen `else`-Tag, damit Alternativen gezeigt werden können. Dabei ist zu beachten, dass der `else`-Tag zwischen den `if`-Tag und em schließenden `if`-Tag kommt, also

```
<!-- TMPL_IF NAME=MYIF -->
zeig was
<!-- TMPL_ELSE -->
zeig was anderes
<!-- /TMPL_IF -->
```

Ausserdem gibt es einen `unless`- und einen `elsif`-Tag.

Switch-Case

Ein weiterer Vorteil von HTC gegenüber `HTML::Template` ist das Switch. Ich persönlich setze es nur bei kleinen Fallunterscheidungen ein, ansonsten verlagere ich es in den Perl-Code.

Das folgende Beispiel soll zeigen, wie man sehr einfach einen Deutschen mit „Hallo“ und einen Italiener mit „Ciao“ begrüßen kann:

```
<!-- TMPL_SWITCH NAME=WELCOME -->
<!-- TMPL_CASE deutsch -->Hallo
<!-- TMPL_CASE italiano -->Ciao
<!-- /TMPL_SWITCH -->
```

Und im Perl-Skript würde das folgendermaßen aussehen:
So etwas macht natürlich nur Sinn, wenn man sich Perl-Code

```
$obj->param(WELCOME => 'italiano');
```

sparen kann. Zum Beispiel wenn das „italiano“ von einer select-Box kommt oder aus einer Datenbank. Wenn man sich das „italiano“ selbst erst noch holen muss, dann könnte man sich das Switch sparen, einen einfachen Platzhalter machen und das „Hallo“ beziehungsweise „Ciao“ direkt einsetzen.

Include

Die Includes sind wichtig, wenn man zum Beispiel ein Basislayout hat und das Menü in einer anderen Datei hat und in das Basislayout einbinden will. Auch die eingebundene Datei kann ein Template sein. Dann werden die Platzhalter „übernommen“. Das heißt, dass die Werte für die Platzhalter auch an das große Objekt übergeben werden müssen.

Ein kleines Beispiel:

Basis.tmpl:

```
<html>
<body>
   <!-- TMPL_INCLUDE NAME=menue.tmpl -->
</body>
</html>
```

menue.tmpl:

```
<a href="item1.html">Menüpunkt1</a><br />
<a href="item2.html">Menüpunkt2</a><br />
```

Wenn dieses Template in einem Skript verwendet wird, sieht die Ausgabe so aus:

```
<html>
<body>
 <a href="item1.html">Menüpunkt1</a><br />
 <a href="item2.html">Menüpunkt2</a><br />
</body>
</html>
```

Dynamische Includes

Die dynamischen Includes gibt es nicht unter `HTML::Template`. Dies ist jedoch ein überaus nützliches Feature. So kann man eine Webseite erstellen, die in mehrere Bereiche eingeteilt wird, für jeden Bereich ein eigenes Menü hat, aber überall ein gleiches Grundlayout.

Basis.tmpl:

```
<html>
<body>
  <!-- TMPL_INCLUDE_VAR NAME=MYMENU -->
</body>
</html>
```

menue.tmpl:

```
<a href="item1.html">Menüpunkt1</a><br />
<a href="item2.html">Menüpunkt2</a><br />
```

Wenn dieses Template in einem Skript verwendet wird, sieht die Ausgabe so aus:

```
use HTML::Template::Compiled;
my $obj = HTML::Template::Compiled->new(
       filename => '/path/to/Basis.tmpl'
);
$obj->param(MYMENU => 'menue.tmpl');
```

bringt dann als Ausgabe:

```
<html>
<body>
  <a href="item1.html">Menüpunkt1</a><br />
  <a href="item2.html">Menüpunkt2</a><br />
</body>
</html>
```

Perl-Code

Bei `HTML::Template::Compiled` besteht auch die Möglichkeit, Perl-Code in das Template einzubetten - allerdings erst seit der Version 0.82. Ich persönlich nutze diesen Tag überhaupt nicht, da es für mich eine zu starke Vermischung von HTML und Perl bedeutet.

Aber da es viele Entwickler gibt, die solche Features gerne nutzen - und dafür auch ihre guten Gründe haben - stelle ich dieses Feature hier vor.

Standardmäßig kann man keinen Perl-Code im Template einbetten. Will man diese Eigenschaft nutzen, so muss das bei der Objekterzeugung mitgeteilt werden:

```
my $obj = HTML::Template::Compiled->new(
       filename    => $file,
       use_perl => 1
);
```

Jetzt kann im Template auch Perl verwendet werden, zum Beispiel so:

```
<!-- TMPL_PERL if(4 > 3){ -->
4 ist größer als 3!
<!-- TMPL_PERL } -->
```

Zum Testen habe ich ein kleines Skript dazu geschrieben (siehe Listing 1).

Und man erhält folgende Ausgabe:

```
~/FooMagazin 17> perl htc_test.pl

4 ist größer als 3!
```

Allerdings muss man mit dem Tagstyle etwas aufpassen. Die klassische Darstellungsweise mit `<TMPL_PERL if(1 > 0)>` funktioniert nicht Hier muss ein anderer Tagstyle - wie zum Beispiel der ASP-Style - verwendet werden.

`HTML::Template::Compiled` ist **kein** Perl-Parser. Das bedeutet, dass man bei der Syntax selbst aufpassen muss. Ansonsten kommt eine Fehlermeldung, die aber nicht immer so eindeutig ist.

Caching

`HTML::Template::Compiled` ermöglicht das Caching. Mit `precompile` können Templates schon in Perl-Code übersetzt werden und in einem Verzeichnis abgelegt werden. Wenn bei der Objekterzeugung die `cache_dir`-Angabe gesetzt ist, wird das Template nicht geparst, sondern es wird der generierte Perl-Code geladen. Das Template wird nur geparst, wenn es sich geändert hat.

Referenzen

[1] http://www.masonhq.com/
[2] http://www.template-toolkit.org/
[3] http://search.cpan.org/dist/HTML-Template-Compiled/
[4] http://perl.apache.org/docs/tutorials/tmpl/comparison/comparison.html

Renée Bäcker

```
1  #!/usr/bin/perl
2
3  use strict;
4  use warnings;
5  use HTML::Template::Compiled;
6
7  my $template = q~ <!-- TMPL_PERL if (4 > 3) { -->
8  4 ist größer als 3!
9  <!-- TMPL_PERL } -->~;
10
11 my $obj = HTML::Template::Compiled->new(scalarref => \$template,
12 use_perl => 1,);
13 print $obj->output();
```

Listing 1

Das merk‘ ich mir – Caching in Webapplikationen

Bei größeren Webapplikationen muss man sich verstärkt darum kümmern, die Datenbank vor zu vielen Zugriffen zu schützen. Eine Methode ist Caching. Wenn man richtig cacht, skaliert die Applikation besser. Denn während man durch Vervielfachung der Webserver zwar die Applikation selbst einfach loadbalancen kann, ist eine Vervielfachung der Datenbanken nicht so leicht, da die Masterdatenbank auch dadurch belastet wird, dass sie die neuen Daten auch an alle Replikanten verteilen muss. Mehrere Ebenen von Replikation führen dagegen dazu, dass die Daten langsamer bei allen Replikanten ankommen.

Dieser Artikel soll anhand zweier Beispiele zeigen, welche Daten man cachen kann. Damit soll verdeutlicht werden, wie man sich eine leicht anwendbare, generische API für das Caching erstellt.

Natürlich sollte man die Effizienz der Datenbankabfragen nicht vernachlässigen. Gerade bei spezialisierten Anwendungen kann man, indem man die Anwendungsfälle analysiert, sehr viel mehr optimieren.

Community-Framework

Als Anwendungsfall stellen wir uns eine Art Framework vor, das mehrere „Module“ beinhaltet, wie ein Forum, eine Galerie, ein Blog und ein integriertes User-Rollen-System. In all diesen Modulen werden bei jedem Request Daten benötigt, die sich aber häufig nicht ändern. Ein Cachen der gesamten HTML-Ausgabe verbietet sich jedoch, da sowohl für jeden eingeloggten Benutzer als auch für Besucher Teile der Seite dynamisch sind, beispielsweise die Anzeige der momentan eingeloggten Benutzer. Wir müssen also Datenstrukturen cachen, die dann dynamisch zum fertigen HTML zusammengebaut werden.

Beispiel 1: Rollen-System

Die Applikation hat ein Rollen-System, welches pro Benutzer mehrere Rollen hat, und jeder Rolle ist eine Liste von erlaubten Aktionen zugeordnet. Nicht eingeloggte Benutzer bekommen z.B. die Rolle ‘gast‘, welche das Forum anzeigen kann, die Galerie und anonyme Artikel schreiben kann.

Anhand dieser Informationen werden Aktionen erlaubt/verboten und Buttons/Links im HTML dargestellt oder nicht.
Schneller für das Auslesen aus der Datenbank wäre eine direkte Zuordnung von Aktionen zum Benutzer, jedoch hätte man damit eine nicht-normalisierte Datenbankstruktur, mehr Daten und mehr Aufwand als Administrator beim Zuweisen der Aktionen.

Bei jedem Request wird nun aus der Datenbank gelesen, um welchen Benutzer es sich handelt. Dann werden seine ihm zugeordneten Rollen geholt, und dazu wiederum die erlaubten Aktionen. Da es sich hier um zwei 1:n- Beziehungen handelt, ist das ein nicht zu vernachlässigender Aufwand für die Datenbank. Im Einzelnen ist das nicht tragisch, aber es gibt in einer Applikation mehrere solcher Stellen, die eine Seite zusammengenommen langsam machen und die Datenbank belasten können.

An dieser Stelle kann man aber die Applikation optimieren. Die zugeordneten Rollen ändern sich ja in der Regel sehr selten, besonders für die Rolle ‘gast‘.

Nun kann man für jeden Nutzer, sobald er sich einloggt, also den ersten Request macht, seine ihm zugeordneten Rollen/Aktionen in eine Perl-Datenstruktur speichern und cachen. Da in der Regel immer nur ein Bruchteil der gesamten Benutzer gleichzeitig eingeloggt ist, sollte sich der Speicherplatz hier in Grenzen halten.

Die Stelle, an der die Rollen aus der Datenbank geholt werden, ist in Listing 1 gezeigt.

Listing 1 zeigt ungefähr, was alles aus der Datenbank geholt werden muss. Mit entsprechenden Optionen kann man DBIx::Class auch dazu bringen, die Daten in weniger Statements zu holen, es bleibt jedoch halbwegs aufwändig.

An diese Stelle kommt nun der Code, wie er in Listing 2 gezeigt ist. Man holt sich die Daten, die man braucht, aus dem Cache, und falls dieser nichts zurückliefert, greift man auf die bisherige Methode zurück. Wie der Cache intern arbeitet, wird später gezeigt.

```
sub fetch_roles_and_actions {
    my ($f) = @_;
    # Datenbankabfrage, wie sie z.B. mit DBIx::Class aussehen würde
    # select role_id from user_role where user_id = ?
    # select * from role where role_id in (...)
    my @roles = $user->roles;
    # Weitere Datenbankabfragen
    # select * from role_action where role_id in (...)
    my @actions = map { $_->actions } @roles;

    # In eine für die Benutzung/fürs Template
    # optimierte Form bringen.
    $roles_and_actions = {
        roles => [map { $_->view } @roles],
        actions => [map { $_->view } @actions],
    };
    return $roles_and_actions;
}
```

Listing 1

```
# alt
my $roles_and_actions = $f->fetch_roles_and_actions();
#
# neu
# als Cache-Key z.B. system/roles/123
my $roles_and_actions = $f->from_cache(‘system/roles/‘ . $user->id);
unless ($roles_and_actions) {
    # nichts im Cache oder veralteter Cache
    # also doch Datenbank befragen
    $roles_and_actions = $f->fetch_roles_and_actions();

     $f->store_cache(
        ‘system/roles/‘ . $user->id,    # Key
        $roles_and_actions,             # Daten
        # Sekunden, bis der Eintrag automatisch verfällt
        60,
    );
}
# noch etwas bequemer
my $roles_and_actions = $f->cache(
    ‘system/roles/‘ . $user->id,
    sub { $f->fetch_roles_and_actions() },
    60,
);
```

Listing 2

Damit hat man sich um alles nötige beim Auslesen gekümmert. Das Framework, hier in der Variable `$f`, wandelt den Hash mittels eines Serialisierungs-Moduls von Perl-Daten in einen String um und umgekehrt. Das Caching-Modul verwaltet also einen einfachen String.

Zusätzlich sollte man dafür sorgen, dass die Cache-Einträge auch irgendwann wieder verfallen; hier wurden 60 Sekunden angegeben. Memcached oder Cache::FileCache (siehe unten) bieten entsprechende Optionen dafür.

Um explizit eine Änderung im Rollensystem sofort sichtbar zu machen, muss man den Cache löschen. Also muss man

in der Methode, die die Rollenänderung des Users durchführt, folgenden Code hinzufügen:

```
$f->delete_cache('system/roles/' . $user->id);
```

Ändert man eine Rolle und löscht beispielsweise eine zugeordnete Aktion, müsste man alle User mit dieser Rolle ausfindig machen und deren Rollen-Cache löschen; man kann aber auch das automatische Verfallsdatum relativ niedrig setzen. Das hat den Nachteil, dass sich Rollenänderungen erst nach maximal dieser Zeit auswirken. Das muss von Fall zu Fall entschieden werden.

Den Zeitraum, bis ein Cache-Eintrag verfällt, muss man generell nicht sehr hoch bemessen. Eine Minute ist ausreichend. Es geht beim Caching um Skalierung. Es soll in erster Linie die Datenbank bei vielen Requests vor zu hoher Belastung schützen. D.h. bei einem mäßig besuchten Forum wird es öfter vorkommen, dass die Daten direkt aus der Datenbank gelesen werden. Die Datenbank sollte also durchaus so aufgebaut sein, dass der Benutzer nicht ewig warten muss. Das Caching wird erst dann richtig nützlich, wenn viele gleichzeitige Requests abgesetzt werden.

Beispiel 2: Thread-Darstellung

Als weiteres Beispiel soll die Anzeige eines Threads im Forum dienen. Jede Seite eines Threads zeigt 30 Artikel an. Im Rohtext können sich Tags befinden, z.B. Links auf andere Threads, etwa `[thread]1234[/thread]`, oder Links auf Benutzerprofile. In der gerenderten Form werden hier Links angezeigt der Form

```
<a href="http://example.com/forum?thread=1234">
  Thread-Titel
</a>
```

Somit müssen hier weitere Datenbankzugriffe erfolgen, um den Thread-Titel oder den Benutzernamen auszulesen. Ausserdem erscheint neben jedem Artikel das Profil des Autors.

Das heisst, es müssen 30 Thread-Artikel ausgelesen werden, die zugehörigen Profile der Autoren, und zu den Artikeltexten danach die verlinkten Thread-Titel und Benutzernamen.

Man könnte mit entsprechend viel Aufwand optimierte Datenbankabfragen machen, aber gerade die im Artikel enthaltenen Links machen das ganze doch etwas umständlich. Man müsste alle Artikeltexte nach Tags durchsuchen, die verlinkten IDs gesammelt auslesen und wieder den Artikeln zuordnen.

Hier kann man es sich nun einfacher machen und einen Cache-Eintrag für den Thread machen, der alle Artikel-IDs enthält, und für jeden Artikel einen extra Cache-Eintrag. Das resultiert dann in unserem Fall in 31 Datei- oder Cache-Zugriffen (bei Bedarf kann man auch immer 30 Artikel ein einem Cache-Eintrag speichern). Die gecachten Datenstrukturen müssen nun lediglich an das Templating-System geliefert werden. Im Idealfall hat man so z.B. bei einem Zugriff eines Gast-Benutzers keine einzige Datenbankabfrage.

Gegebenenfalls muss man aber auch die gecachte Datenstruktur noch verändern und z.B. Teile herauslöschen. Dies sollte jedoch in der Regel kaum Zeit kosten.

Beispiel: Die Übersichtsseite einer Forum-Webseite zeigt die Liste der Foren und den letzten Artikeltitel des Forums. Manche Foren sind nur von Moderatoren lesbar. Cachen muss man alle, aber für normale Benutzer und Gäste muss man dann die Moderatoren-Foren noch aussortieren.

Welche Datenstrukturen werden gecacht?

Es empfiehlt sich nicht, die Daten zu cachen, die z.B. direkt von DBIx::Class stammen. In jedem Objekt stehen nämlich eine ganze Menge Meta-Informationen zur Datenbank, die man für die Anzeige gar nicht braucht. Deshalb empfiehlt es sich, eine View-Klasse zu erstellen; das kann ein einfacher Hash sein oder aber ein simpel gehaltenes Objekt.

Caching im Dateisystem, im Speicher oder zentral?

Bei einer kleinen Webseite kann man getrost im Dateisystem cachen. Sobald die Seite aber so häufig besucht wird, dass man Loadbalancer einsetzt, sollte man zentral cachen, und das geht mit einem Server wie memcached. Im Prozess selbst zu speichern macht den Prozess sehr gross, und weder mit FastCGI noch mit mod_perl kann man Speicher einfach so untereinander teilen.

Aus eigener Erfahrung kann ich empfehlen, in der Applikation einen Wrapper um das Caching-System zu basteln. Somit ist man immer flexibel und kann bei Bedarf umschalten.

Die Caching-APIs an sich sind eigentlich recht einfach. Man gibt beim Speichern einen eindeutigen Schlüssel an, die Datenstruktur, und das Verfallsdatum bzw. -Zeit. Der schwierige Teil ist, sich zu überlegen, was man cacht, wie lange, und an welchen Stellen man bestimmte Cache-Einträge löschen muss.

memcached, Cache::Memcached

memcached [1] ist ein in C geschriebener Caching-Server. Für die meisten Linux-Distributionen sollte es fertige Pakete geben. memcached lässt sich sehr einfach starten. Zum ersten Testen reicht ein simples:

```
$ memcached
```

Der Default-Port ist 11211, man kann ihn mit -p verändern. Mit der Option -d schickt man den Prozess in den Hintergrund.

Ein Beispiel für die Benutzung mit Cache::Memcached [2] ist in Listing 3 beschrieben.

Man sieht also, das Modul kümmert sich auch um die Serialisierung von Datenstrukturen. Der Server kümmert sich um das Verfallsdatum.

```
use Cache::Memcached;

my $memd = new Cache::Memcached {
  'servers' => [ "127.0.0.1:11211" ],
};

$memd->set('foo' => "Some value", 60);

$memd->set("system/roles/$id" => {
  complex => [ "object", 2, 4 ]
});

my $val = $memd->get("foo");
my $val = $memd->get("system/roles/$id");
if ($val) { print $val->{complex}->[2]; }
$memd->delete("foo");
```

Listing 3

Ist der Cache einmal voll, also hat der memcached-Server seine Grenze erreicht und es befinden sich nur Einträge im Cache, die noch nicht abgelaufen sind, muss trotzdem etwas gelöscht werden. memcached wirft an dieser Stelle den Eintrag raus, der am längsten nicht abgerufen wurde („LRU“).

Weitere ausführliche Infos findet man auf der Webseite von memcached [1].

Cache::FileCache

Cache::FileCache ist ein Modul aus dem Paket Cache::Cache [3]. Dieses Paket bietet mehrere Backends an, unter anderem auch Cache::MemoryCache, welches anders als Cache::Memcached die Daten direkt im Speicher des Prozesses hält. Das kommt für unsere Applikation nicht in Frage, da der Speicher dann viel zu groß werden würde.

Ein Beispiel zur Benutzung von Cache::FileCache sieht man in Listing 4.

Standardmäßig wird der Cache im Verzeichnis `/tmp/FileCache` abgelegt, dieses Verzeichnis kann man jedoch mit der Option `cache_root` verändern.

Mit der Methode `purge` kann man verfallene Einträge löschen lassen (bzw. `Purge`, um alle Namespaces zu reinigen). Entweder man ruft diese Methode in einem Cronjob auf, oder man setzt die Option `auto_purge_interval` auf den Intervall (in Sekunden), in dem `purge` automatisch aufgerufen werden soll. Zu Details sollte man sich die Dokumentation ansehen, zum ersten Testen reichen die in Listing 4 gezeigten Optionen.

```
use Cache::FileCache;

my $cache = new Cache::FileCache({
    namespace          => 'system',
    default_expires_in => 60,
});

my $roles = $cache->get( "roles/$id" );

unless ($roles) {
    $roles = datenbank_abfrage($id);
    $cache->set(
            "roles/$id", $roles, "60 seconds"
            );
    }
```

Listing 4

Im Gegensatz zu Cache::Memcached/memcached muss man sich hier aber um die Größe des Caches kümmern, damit das Dateisystem nicht überläuft. Die Methode `size` (bzw. `Size` für alle Namespaces) liefert die Größe des gesamten Caches. Da in der Dokumentation nichts weiter dazu zu finden ist, muss man also selbst regelmäßig die Größe abfragen und entsprechende Maßnahmen einleiten, wenn `purge` nicht ausreicht.

Im großen und ganzen ist Cache::FileCache auch ein Modul, welches man mit wenig Aufwand für eine erste lauffähige Version eines Programms einsetzen kann.

Einsatz im Framework

Listing 5 zeigt nun, wie die Framework-Funktion `cache` aussehen könnte, die im Listing 2 benutzt wurde. Das Framework (hier in `$f`) weiss, ob es Cache::FileCache oder Cache::Memcached verwenden soll und hat die entsprechenden Objekte schon initialisiert.

Wie man sieht, benutzen die beiden Caching-Module sogar dasselbe Interface, so dass die `cache`-Methode hier im Beispiel sehr viel duplizierten Code enthält. Es sollte jedoch hier auch das Prinzip des Wrappers gezeigt werden. Auf diese Art kann man noch weitere Arten von Caching hinzufügen, die vielleicht ein anderes Interface haben.

```
sub cache {
    my ($f, $key, $expire, $sub) = @_;
    if (my $c = $f->memcached) {
        # konfiguriert mit Cache::Memcached
        # $c enthält also schon ein Objekt dieser Klasse
        my $data = $c->get($key);
        unless (defined $data) {
            # keine Daten im Cache, benutze die übergebene
            # subroutine, um die Daten aus der Datenbank zu holen
            $data = $sub->();
            $c->set($key, $data, $expire);
        }
        return $data;
    }
    elsif (my $c = $f->filecache) {
        # konfiguriert mit Cache::FileCache
        # $c enthält also schon ein Objekt dieser Klasse
        # Der Einfachheit halber benutzen wir einen einzigen
        # Namespace, bei Bedarf kann man auch in Namespaces
        # nach Framework-Modulen unterteilen.
        my $data = $c->get($key);
        unless (defined $data) {
            $data = $sub->();
            # hier bei Bedarf Cache-Größe abfragen, wenn
            # man viel cacht und das Dateisystem nicht
            # überlaufen soll
            $c->set($key, $data, "$expire seconds");
        }
        return $data;
    }
    else {
        # die Applikation läuft ohne Caching
        # also einfach direkt aus der Datenbank holen
        my $data = $sub->();
        return $data;
    }
}
```

Listing 5

Darüber hinaus

Was kann man noch optimieren?

Unter bestimmten Umständen kann es sinnvoll sein, den Mysql-Querycache zu benutzen. Dafür sollte sich eine Tabelle nicht allzu oft ändern, denn dann wird jedesmal der Cache gelöscht.
Hat man Seiten, die keine dynamischen Teile beinhalten, kann man für diese Seiten einen Proxy wie squid [4] vorschalten.

Manchmal ist es auch ratsam, direkt in der Datenbank berechnungsintensive Daten zu speichern. So könnte man beim Beispiel der Thread-Darstellung den gerenderten Text eines Artikels beim Speichern generieren und neben dem Rohtext in die Datenbank schreiben.

Bei den Cache-Lösungen muss man sich immer bewusst sein, dass es „nur“ ein Cache ist, also redundante Daten gespeichert werden. Für Session-Daten ist es daher nicht geeignet.

Referenzen

[1] http://www.danga.com/memcached/
[2] http://search.cpan.org/dist/Cache-Memcached
[3] http://search.cpan.org/dist/Cache-Cache
[4] http://www.squid-cache.org/

Tina Müller

TPF-TICKER

Perl 6 Microgrant

Die Perl Foundation hat einen neuen Perl 6 Microgrant bewilligt. Adriano Ferreira wird einen Artikel über die Operatoren in Perl 6 schreiben und diesen dann auf ONLamp veröffentlichen.

YAPC‘s 2008

Die Austragungsorte für die YAPC::NA und YAPC::Europe im Jahr 2008 wurden bekanntgegeben. Die YAPC::NA wird wieder in Chicago ausgerichtet und die europäische Perl-Konferenz wird in Kopenhagen stattfinden. Genauere Informationen werden bald veröffentlicht.

CPAN-Testers Mailingliste

Es gibt eine neue Mailingliste für die CPAN-Tester. Auf der alten werden weiterhin alle Testreports veröffentlicht, auf der neuen werden die Diskussionen abgehalten.

Um sich dort einzutragen, reicht es, eine leere Mail an cpan-testers-discuss-subscribe@perl.org zu schicken.

Sessions mit Catalyst

Die meisten Web-Anwendungen werden irgendwann zu einem Punkt kommen, an dem es unerlässlich ist, Informationen über mehr als eine einzelne Anfrage zwischenzuspeichern, da HTTP von Haus aus ein zustandsloses Protokoll ist.

Vielen wird dieser Satz vertraut vorkommen, ist es schließlich ein wiederkehrendes Problem bei fast jedem neuen Web-Projekt.

Lösungen dafür gibt es für alle bekannteren Web-Frameworks; hier wird die für Catalyst an einem Beispiel gezeigt.

Ein grundlegendes Session-Framework erhält man mit der Installation von Catalyst::Plugin::Session. Zusätzlich wird noch jeweils ein Modul für die Zustände (State) und eins für die Speicherung (Store) benötigt. Hier findet man auf CPAN inzwischen mehrere Hände voll Module die unterschiedliche Schnittstellen für die Zustände und Speicherung bereitstellen. In diesem Artikel werden zu diesem Zweck „Catalyst::Plugin::Session::State::Cookie“ und „Catalyst::Plugin::Session::Store::File“ eingesetzt. Die Namen der Module sind selbstredend - wir wollen client-seitig ein Cookie für die Session-ID setzen und server-seitig im Dateisystem speichern.

Nennen wir die Beispielanwendung „My::MiniShop“, dann muss „lib/My/MiniShop.pm“ um die Zeilen die mit „use Catalyst“ beginnen um „Session“, „Session::Store::File“ und „Session::State::Cookie“ erweitert werden. In der lib/My/MiniShop.pm sieht das dann zum Beispiel folgendermaßen aus (siehe Listing 1).

```
use Catalyst qw/
    -Debug
    ConfigLoader
    Static::Simple
    Session
    Session::State::Cookie
    Session::Store::File
/;
```

Listing 1

```
sub add : Local {
    my ($self, $c, @name) = @_;

    unless (@name) {
        unless (@name = $c->request->param('name')) {
            $c->response->redirect($c->uri_for('/'));
            return;
        }
    }

    foreach my $name (@name) {
        next unless (exists($product{$name}));

        $c->session->{'products'}{$name}++;
    }

    $c->response->redirect($c->session ?
       $c->session->{'lasturi'} : $c->uri_for('/'));
}
```

Listing 2


```
sub delete : Local {
    my ($self, $c, @name) = @_;

    foreach my $name (@name) {
        if (exists($c->session->{'products'}{$name})) {
            delete($c->session->{'products'}{$name});
        }
    }

    $c->response->redirect($c->session ?
        $c->session->{'lasturi'} : $c->uri_for('/'));
}
```

Listing 3

Wenn die Standardeinstellungen auf dem System nicht von gängigen Umgebungen abweichen, braucht man nichts weiter tun und kann direkt das eigentliche Programm angehen.

In den meisten Fällen kommt man allein mit der zusätzlichen Methode „session“ zurecht; Beispiel der Methode „add“ aus dem MiniShop (siehe Listing 2).

Hier wird der Einfachheit halber der Name des Produktes zur Identifikation genutzt und in der Session um einen Zähler erhöht.

Im Großen und Ganzen ist die Session nichts anderes als eine normale Hash-Referenz mit der man wie gewohnt Arbeiten kann.

Ein weiteres Beispiel dafür ist die Methode „delete“ im MiniShop (siehe Listing 3).

Hier wird mit der Funktion „delete“ der entsprechende Eintrag aus der Session gelöscht, wie man es gewohnt ist.

Damit ist bisher nur die grundlegende Funktionsweise von Sessions in Catalyst beschrieben und es bleibt noch die Dokumentation zu erwähnen in der weitere Methoden zu finden sind, wie beispielsweise „session_expires()“ mit der sich die Ablaufzeit einer Session prüfen, bzw. setzen lässt.

Der für den Artikel geschriebene MiniShop ist zwar äußerst simpel, demonstriert aber auf unkomplizierte Weise die Verwendung des Session-plugins in Catalyst.

Simon Betrang

PODCASTS – TEIL 4

Making your own CPAN

Eine weitere Präsentation von brian d foy, die er bei einem Treffen der L.A. Perlmongers gehalten hat. Diesmal geht es darum, eigene kleine CPAN-„Mirrors“ aufzubauen. Es gibt zwar schon Mini CPAN, aber selbst das ist häufig noch zu groß. brian d foy zeigt, wie man ein eigenes CPAN aufbaut nur mit den Distributionen, die für einen selbst interessant sind.

Ross Turk von Sourceforge.net

Josh McAdams hat sich auf der OSCON 2007 mit Ross Turk - dem „Community Manager“ - über das System von Sourceforge.net unterhalten. Es geht darum, welche Infrastruktur Sourceforge.net für Projekte zur Verfügung stellt und noch viel mehr..

Renée Bäcker

KONFERENZ

YAPC::Europe 2007 in Wien

Der diesjährige europäische Perl-Workshop fand vom 28.-30. August in der österreichischen Hauptstadt Wien statt. Für die drei Tage hatte sich große Prominenz angemeldet: Larry Wall, Damian Conway und Mark Jason Dominus. Am Abend vor der Konferenz gab es ein großes Treffen in einem Wiener Restaurant.

Erster Tag (28.08.2007)

Am 28.08. begann die Konferenz mit der offiziellen Eröffnung durch Thomas Klausner. Bei der Eröffnung wurde auch der Ort der YAPC::Europe 2008 bekanntgegeben - Kopenhagen. Der erste Vortrag wurde von José Castro gehalten. Dieser war hauptsächlich an die „Neulinge" der Konferenz gerichtet. Es ging dabei darum, wie man für sich selbst möglichst viel aus der Veranstaltung mitnehmen kann. Anschließend gab es die erste Keynote – gehalten von Larry Wall. Der Erfinder von Perl ging dabei auf verschiedene Skriptsprachen und deren Design ein.

Nach der Frühstückspause ging es in fünf verschiedenen Räumen weiter, dieser Artikel berichtet aber nur kurz von jeweils einem Vortrag. Der erste reguläre Vortrag, den ich mir angehört habe, wurde von Jan Henning Thorsen über Operator-Overloading gehalten. Hier wurde gezeigt, wie eigene Subroutinen statt der Standardfunktionen bei bestimmten Operatoren ausführen kann. Dazu wird das Modul overload.pm verwendet.

Direkt im Anschluss zeigte Barbie von Birmingham.pm, wie Webanwendungen mit Selenium getestet werden können. Dabei werden Beschränkungen von WWW::Mechanize (zum Beispiel, dass JavaScript nicht berücksichtigt werden kann) aufgehoben.

Beim Vortrag „Evolving architecture – make development easy and your site faster" zeigte Leo Lapworth, wie sich die Umgebung von Webanwendungen im Lauf der Zeit verändert: von einem einzelnen Server, der alles alleine macht bis hin zu einem System mit mehreren Servern, die spezialisiert einzelne Aufgaben wahrnehmen. R. Geoffrey Avery zeigte bei „Manage::Objects" eine Möglichkeit, Objekte in einer Art „Singleton" „dauerhaft" zur Verfügung stellen zu können.

Wie man in 28 Tagen etwas für das tägliche Geschäft lernen kann, zeigte Greg McCarroll in „28 days later".

Der letzte Vortrag vor der Kaffeepause war „Catalyst – refactor large apps with team and have fun!". Dort wurde von Adam Bartosik gezeigt, wie er in einem Projekt von einfachen CGI-Skripten auf die Verwendung von Catalyst kam und welche Vorteile das gebracht hat – aber auch welche Nachteile damit verbunden waren.

Eine sehr interessante Alternative zu HTML::Prototype und CGI::Ajax zeigte Jon Allen in „Lightweight Ajax with OpenThought". OpenThought ist sehr leichtgewichtig und hat viele Möglichkeiten, mit einfachen Mitteln Ajax-Funktionen in Webseiten einzubauen. Ebenfalls um Webanwendungen ging es Leon Brocard „Scaling with memcached". Gerade stark frequentierte Webseiten wie Facebook haben das Problem, dass viele Datenbankabfragen gemacht werden müssen und die Ergebnisse gecachet werden sollen. Dies wird mit memcached und dem Perl-Modul Cache::Memcached erreicht.

Inhalte von Webseiten auslesen war das Thema von Tatsuhiko Miyagawa in „Practical Web scraping with Web::Scraper". Dabei verzichtet er auf Reguläre Ausdrücke und verwendet stattdessen XPath beziehungsweise CSS-Pfade. Der letzte Vortrag am ersten Tag war „Finding Perl's place in the world" von Richard Dice. Er gab einen Einblick in seine Arbeit für die „Perl Foundation" und wie Perl in der Außendarstellung dasteht. Es gibt dabei einige positive Signale, aber auch noch viel Arbeit für alle Aktiven der Perl-Gemeinde.

Zweiter Tag (29.08.2007)

Der zweite Tag begann mit einer Keynote von Damian Conway. Dabei durften die Teilnehmer der YAPC sehen, wie Conway Variablen einführte, die wie Positronen auch zeitlich rückwärts existieren können. Noch vor der Frühstückspause wurde die „Jobmesse" eröffnet. Zehn Unternehmen haben hier ihre Stände aufgebaut, um sich potentiellen Bewerbern zu stellen.

Nach der Frühstückspause zeigte Mark Jason Dominus am Beispiel von Class::Observable, wie man „besseren" Code schreibt und dabei die Fehleranfälligkeit minimiert. Es waren ein paar interessante Punkte dabei, die auf jeden Fall im Hinterkopf verbleiben sollten. „mock" nutzte Google Codesearch um typische Schwachstellen in Perl-Modulen und -Anwendungen zu finden. Zudem stellte er zwei Tools ganz kurz vor, die beim Sicherheitstest von Webanwendungen nützlich sein können. Danach war die Mittagspause angesagt.

Meine Mittagspause ging bis zur Kaffeepause, da mich die Vorträge in der Zwischenzeit nicht so übermäßig interessiert haben. Das lag aber nur daran, dass ich mich mit den Themen nicht beschäftige, und nicht an den Themen selbst.

Nach der Kaffeepause stand der zweite Conway-Vortrag auf dem Programm. Zusammen mit Larry Wall gab es einen Perl6-Update. Es gab viele kritische Fragen aus dem Auditorium, die Conway und Wall gut beantworteten. Direkt im Anschluss wurde der Transport zum „Attendees Dinner" organisiert. Da der Veranstaltungsort nicht alle Teilnehmer bedienen konnte, wurde ein „Alternativ Dinner" organisiert. Beim „Attendees Dinner" gab es neben gutem Essen und Trinken vor allem die Möglichkeit, mit sehr vielen YAPC-Besuchern zu sprechen.

Dritter Tag (30.08.2007)

19 Talks in 90 Minuten – so fing der letzte Tag der YAPC::Europe 2007 an. Die Lightning Talks sind ein gutes Mittel, ein Projekt oder ein Modul in kurzer Zeit vorzustellen. Oder auch einfach etwas allgemeines, lustiges zu zeigen. Gerade für Konferenz-Neulinge ist es ein gutes Mittel, um einen Einstieg in das „Halten von Vorträgen" zu bekommen. Diesmal gingen die Lightning Talks von (Unicode-)Bugs in Perl, über GUI-Testing bis hin zu einem „Lessons learned"-Vortrag.

Danach habe ich erstmal lange Pause gemacht, weil der Vortrag, den ich mir eigentlich anschauen wollte, ausgefallen ist. Vor der Mittagspause habe ich mir noch den MAD-Vortrag von Gerard Goosens angehört. Dort wurde gezeigt, wie man einen Perl5-nach-Perl5-Übersetzer schreibt. Dabei wird das Perl-Programm in eine XML-Struktur übersetzt, die nach Belieben verändert werden kann.

Zwischen Mittagspause und Kaffeepause fanden dann die letzten Vorträge der diesjährigen YAPC::Europe statt. Der erste Vortrag, den ich mir angehört habe, war von Avar Arnfjörd Bjarmason über „5.10 regex pragmata". Das Hauptthema dabei war die Implementierung des POSIX-Plugins für die RegEx-Engine in Perl 5.10. Direkt im Anschluss hat „mock" gezeigt, wie er riesige Datenmengen in eine Datenbank schreibt und dafür verschiedene Techniken wie perlbal, MogileFS und memcached verwendet.

Ein paar „Nachteile" von Perl – wie zum Beispiel fehlende Installationsroutinen von Perl-Anwendungen – zeigte Mark Fowler. Dabei ging er zu jedem vermeintlichen Nachteil auf die Lösung ein. Der letzte Vortrag war eine Präsentation von „Perl Worst Practices" von Marty Pauly. Neben „Variablen" zeigte er auch, warum Reguläre Ausdrück zu „Bad Code" führen.

Wie es Tradition ist, findet zum Abschluss noch eine Auktion statt, mit deren Erlös die kommende YAPC unterstützt werden soll. Dabei werden auch allerlei kuriose Dinge versteigert. Diesmal wurde ein „Trainingslager mit Damian Conway" versteigert, das für 500 Euro (von mehreren Personen bezahlt) an den Auktionator weitergereicht wurde. Die Organisatoren der kommenden YAPC::EU müssen bei der nächsten Konferenz eine schwedische Flagge auf der Stirn tragen – das wurde für 370 Euro versteigert. Damit war die YAPC::Europe 2007 in Wien vorbei!

Renée Bäcker

Rezension – Intermediate Perl

Perl-Programmierer auf dem Weg vom Novizen zum Meister – das ist die Reihe „Learning Perl“, „Intermediate Perl“ und „Mastering Perl“. Diese Rezension macht in der Mitte Halt und beleuchtet das Buch „Intermediate Perl“. Die drei Autoren sind Partner bei dem Schulungsunternehmen „Stonhenge Consulting“, und so verwundert es nicht, dass das Buch leicht verständlich geschrieben ist.

Die erste Auflage von „Intermediate Perl“ wurde im Jahr 2003 von Randal L. Schwartz noch unter dem Namen „Learning Perl Objects, References & Modules“ veröffentlicht. Der alte Titel beschreibt schon, was in dem Buch behandelt wird: das Objektsystem in Perl, wie mit Referenzen umzugehen ist und schließlich den Einsatz sowie das Entwickeln von Modulen.

Die 220 Seiten Inhalt sind in 19 Kapitel aufgeteilt, die jeweils so konzipiert sind, dass man ein Kapitel in rund einer Stunde durcharbeiten kann. Am Schluss jedes Kapitels werden Aufgaben gestellt, die der Leser zur Übung machen kann. Die Lösungen für die Aufgaben sind hinter den 220 Seiten Inhalt angehängt.

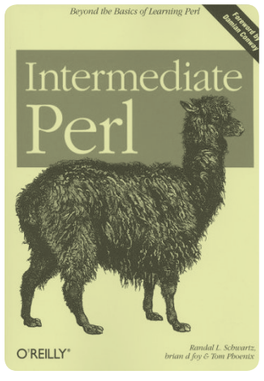
Das erste Kapitel fällt ein wenig aus dem Rahmen, da es einige Perl built-in Funktionen behandelt - map, grep und eval. Danach geht es mit Modulen weiter. Am Anfang geht es um die Verwendung „fremder“ Module, die zum Beispiel von CPAN kommen. Im Anschluss werden Referenzen zum Thema von einfachen Themen wie „Wie erzeugt man Referenzen“ über „Wie lang sind Referenzen gültig“ - hier spielt dann auch Garbage Collection eine kleine Rolle - bis hin zu nützlichen Konstrukten wie die Schwartz‘sche Transformation (benannt nach Randal L. Schwartz).

Nach den „Referenzen“ sind „Objekte“ an der Reihe. Hier wird nach einer Einführung in das Objektsystem von Perl gezeigt, wie Vererbung und das Überschreiben von Funktionen gemacht wird. Der Übergang zum „Module schreiben“ ist dann fließend. Am Anfang wird der Aufbau der Dateien einer typischen CPAN-Distribution erläutert. Nach der Einführung in „Tests“ wird noch gezeigt, wie man seine Module auf CPAN bekommt.

Dieses Buch ist auch für erfahrenere Perl-Programmierer geeignet. Gerade wer seine eigenen CPAN-Module schreiben will und dieses CPAN-konform umsetzen will, sollte sich die letzten Kapitel genauer anschauen. Wer schon länger mit Perl zu tun hat, dem werden die ersten Kapitel über Referenzen nicht weiter bringen. Dort wird Wissen vermittelt, das automatisch kommt, wenn man sich mit komplexen Datenstrukturen beschäftigt.

Insgesamt ist es auf jedenfall ein Buch, das es sich zu lesen lohnt.

Renée Bäcker



Randal L. Schwartz, brian d foy & Tom Phoenix
O‘Reilly Media, 2006
0-596-10206-2

André Mindermann über OTRS

FM: Hallo Herr Mindermann, vielen Dank, dass Sie sich die Zeit für dieses kleine Interview genommen haben. Bevor wir mit den Fragen beginnen, stellen Sie sich und Ihr Unternehmen doch bitte kurz vor.

AM: Mein Name ist André **Mindermann**. Ich bin 41 Jahre alt und habe Betriebswirtschaft, Wirtschaftspädagogik und Philosophie studiert. Ich lebe in Bad Homburg und bin seit 1997 Unternehmer. Im Jahr 2003 habe ich zusammen mit Herrn **Steinbild** (Gründer der SUSE Linux AG) und Herrn **Edenhofer** (Erfinder der Software OTRS) die Firma OTRS GmbH gegründet. OTRS ist heute mit über **49.000 Instalationen in 22 Sprachen das weltweit führende Helpdesk- und Ticketsystem**.

FM: Der Name “((otrs))“ ist Programm. Sie haben viel mit OTRS - einem Ticketsystem geschrieben in Perl - zu tun. Wie sind Ihre Erfahrungen mit Perl in einem größeren System wie OTRS?

AM: **Sehr gut**. Man mag es kaum glauben, aber im Gegensatz zu vielen anderen Programmiersprachen haben wir mit Perl bisher keine Probleme und wir setzen OTRS in vielen Firmen und Konzernen auch in heterogenen Umgebungen ein. Perl ist eine wichtige Schlüssel-Komponente in unserem *Erfolgsrezept*.

FM: OTRS zählt nicht zu den kleinsten Perl-Programmen. Können Sie sagen, worauf Programmierer achten sollen, wenn es um Perl-Programme für die Business-Umgebung geht?

AM: **KIS** (keep it simple as possible). Das zählt nicht nur für die Installation und Benutzung der Software sondern auch für die Architektur und den Quellcode. Je einfacher etwas funktioniert, um so mehr können Sie es **skalieren** und damit **mehr erreichen**. Und wer es bis heute noch nicht verinnerlicht hat, Unit-Tests sind ein **muss**!

FM: Die Installation ist ja häufig ein großes „Manko“ bei Perl-Programmen. Im Gegensatz zu PHP-Skripten muss man bei Perl-Programmen die Skripte mit den richtigen Rechten versehen und eventuell Module vom CPAN installieren. Haben Sie diese Probleme beim Deployment gelöst und wenn ja, **wie** haben Sie das geschafft?

AM: Das ist wahrlich **nicht einfach**. Wir haben für die Perl Pure CPAN Module (auch wenn diese etwas langsamer sind als die in C geschriebenen Brüder) einen eigenen Ordner, welchen wir mit OTRS ausliefern. *So entfällt die Installation einiger dieser CPAN Module* und es müssen nur noch ein paar von CPAN installiert werden (dies erleichtert die Installation sehr stark). Will ein Anwender (der Profi) sein System tunen, dann entfernt er einfach den OTRS-CPAN-Ordner und kann über diesen Weg auch die schnelleren C Module benutzen.

FM: Perl bietet im Bereich Unit-Tests schon eine Vielzahl an Modulen, die für jedermann gedacht sind. Benutzen Sie diese Module oder verwenden Sie ein Framework?
Wie stellen Sie sicher, dass jeder Use-Case in einem Test abgebildet ist. Was können Sie unseren Lesern in Sachen „Test-Strategien“ empfehlen?

AM: Stimmt, es gibt einige Frameworks dafür. Wir hatten *leider* bestimmte Anforderungen, welche uns daran gehindert hatten diese zu nutzen.

Wir haben eine **sehr große** und **stark automatisierte** Unit-Test-Umgebung. Wir haben die Anforderung, auch während der Entwicklung ständig OTRS (den Framework) und die zusätzlichen Applikationen wie unter anderem OTRS::ITSM testen zu lassen. Wir wollen sicherstellen, dass OTRS auf den

verschiedenen Plattformen (x586, X86-64, PPC, ...), Betriebssystemen (Windows, Unix, Linux, Mac, ...), Datenbanken (MySQL, Oracle, Postgres, ...) und Perl-Umgebungen (Perl-Versionen) ohne Fehler läuft. Die jenigen welche diese Konstellationen schon mal gehabt haben, können jetzt _fühlen_ was ich meine. ,-) Somit haben wir eine Umgebung von ca. **50 Maschinen**, welche **ständig** die verschiedensten Möglichkeiten **testen** und das Ergebnis in ein zentrales **Unit-Test-Repository** per SOAP übermitteln. Somit wissen wir **tagesaktuell**, ob es in irgendeiner Konstellation mit einer Änderung Probleme geben würde.

Ich habe Ihnen zur Veranschaulichung ein kleines Test-System unseres Unit-Test-Monitors bereitstellen lassen, um ein wenig zu sehen was ich meine. Unsere Entwickler wissen immer, wenn ein Unit-Test in einer bestimmten Umgebung nicht mehr geht:
http://opms.otrs.com/otrs/public.pl?Action=PublicUnitTest&Subaction=ProductView&Product=OTRS+2.2.x+CVS

PS: Das wichtige bei der Sache ist natürlich auch noch, **kein Bugfix ohne neuen Unit-Test**. Es soll **ein Fehler** ja **nur einmal auftauchen**!

FM: Sie suchen schon seit einiger Zeit nach Perl-Programmierern für Ihr Unternehmen. Woran liegt es, dass Sie anscheinend keinen geeigneten Bewerber finden? Ziehen Sie Konsequenzen aus der langen Suche?

AM: Wir sind ein weltweit *verteiltes Unternehmen* und benötigen Mitarbeiter, die sich in einer dezentralen Unternehmensstruktur wohl fühlen. Zudem ist der Markt an guten Perl-Programmierern leider wirklich dünn gesäht.

FM: Wenn Sie einen Wunsch an die Perl Community in Deutschland frei hätten, was wäre dieser Wunsch?

AM: Mut zur **Lücke** und auch als nicht perfekter Perl-Programmierer bei uns **bewerben** (http://otrs.com/de/job/)! ;)

TPF-TICKER

White Camel Awards

Die White Camel Awards 2007 gehen an:

Allison Randal
Allison hat schon sehr viel für die Perl-Gemeinde getan: Sie war Präsidentin der Perl-Foundation, war und ist sehr aktiv bei der Entwicklung von Perl und Parrot und zuletzt hat sie die Artistic License 2.0 entworfen, die in Zukunft für den Perl-Code gilt.

Tim O'Reilly
Tim O'Reilly ist nicht nur Gründer des O'Reilly-Verlags, bei dem unter anderem das Kamel-Buch herausgekommen ist, Tim ist auch der „Gründer" der Perl-Conference.

Norbert Grüner
Aus deutscher Sicht ist natürlich der dritte Preisträger sehr erfreulich. Norbert Grüner ist Mitbegründer des Deutschen Perl-Workshops und sitzt mittlerweile im Vorstand der YAPC::Europe Foundation.

USER-GRUPPEN

Ruhr.pm – Perl Mongers im Ruhrgebiet

Das Ruhrgebiet ist mit fast 5,4 Millionen Einwohnern der größte Ballungsraum Deutschlands und nach Paris und London der drittgrößte in ganz Europa, doch zwischen zahlreichen konkurrierenden LUGs, Computer- und Elektronik-Treffs fand sich lange Zeit keine einzige Anlaufstelle für Perl-Interessierte und ihren Gedankenaustausch unter Gleichgesinnten.

So entstanden auch die Perl Mongers im Ruhrgebiet Ende 2005 ursprünglich aus der Idee heraus, (noch) eine LUG zu gründen, eine mit Schwerpunkten in der Softwareentwicklung. Nach diversen kleinen Wettstreitigkeiten mit dem Ziel, komplexere Aufgaben mit möglichst wenig Quelltext möglichst schnell zu erledigen, fokussierte sich bald das Interesse mit großem Abstand auf Perl.

Treffen

Seit dem ersten offiziellen Treffen der *Ruhr.pm* [1] am 13. März 2006 kommen wir in der Regel an jedem zweiten Montag des Monats um 19:00 Uhr im *Café Maze* [2] in der Essener Innenstadt zusammen, wo es sich in gemütlicher Atmosphäre abwechslungsreich trinken und ausgezeichnet speisen lässt.

Ein typisches Treffen der Ruhr.pm besteht aus einem ausgedehnten sozialen und einem kürzeren technischen Teil, der meist aus einem etwa 30-minütigen Kurz-Vortrag mit anschließender Diskussion besteht. Aufbauend auf diesen einfachen Kurz-Vorträgen werden bei Interesse oft eingehende Vorträge oder Workshops vorbereitet, die dann entweder im Café Maze, im Konferenzraum der *Keybits OHG* [3] oder in den Wohnungen und Gärten von Teilnehmern abgehalten werden. Diese Gärten sind auch Veranstaltungsort unserer jährlichen Juli-Grillabende.

undef

Zu allen Zusammenkünften begleitet uns undef, unser Maskottchen. undef ist ein in China geborenes Plüsch-Trampeltier mit US-Amerikanischer Staatsbürgerschaft, das erst nach Zahlung eines überhöhten Lösegeldes aus einer Geiselhaft im Amsterdamer Artis-Zoo entlassen wurde.

Teilnehmer

Außer undef nehmen an unseren Treffen je nach Tagesform und Jahreszeit 10 bis 15 Personen verschiedenster Perl-Erfahrungsgrade teil. Unter den regulären Teilnehmern finden sich Schüler, Studenten und Angestellte genauso wie Hobby-Programmierer, Freelancer und Dozenten öffentlicher und privater Bildungseinrichtungen wie Universitäten, der AWO oder des Linuxhotels Villa Vogelsang. Auch Entwickler in großen OSS-Projekten wie OpenOffice.org und Organisatoren und Initiatoren lokaler OSS-Events (u.a. Essener Linuxtage, come2linux) sind stets anzutreffen.

Kommunikation

Neben unseren Treffen sind der *IRC* [4] sowie die quasi obligatorische *Mailingliste* [5] unsere zentralen Kommunikationsmittel, deren Leserschaft neben vielen der regelmäßigen Teilneh-

mer auch eine ganze Reihe weiterer, uns teilweise nur virtuell bekannter Personen umfasst. Hier lassen sich dringende Fragen schnell klären und gemeinsame Projekte auch außerhalb der monatlichen persönlichen Treffen organisieren.

Projekte

Eines unserer aktuell in Entwicklung befindlichen Projekte ist eine nicht-kommerzielle *Jobbörse* [6] für Anstellungen sowie Projekte, die direkt etwas mit Perl zu tun haben. Sie bietet Arbeitgebern und Vermittlern die Möglichkeit, auf ihren Webservern Inserate in einem einfachen *XML-Austauschformat* [7] anzubieten, die mehrmals täglich überprüft und in unsere Datenbank übernommen werden. Diese Datenbank können Interessenten aus ganz Deutschland anhand der Postleitzahl ihres Wohnortes abfragen, eine Umkreissuche informiert sie über Angebote in ihrer Nähe.Unterstützt wird die Suche durch eine grafische Aufbereitung in verschiedenen Karten, aktuell eine für das Ruhrgebiet sowie eine für ganz Deutschland.

Ein anderes Projekt beschäftigt sich mit der Entwicklung des Spiels *Virtual Hacking* [8], das allgemeine Grundlagen der Systemsicherheit und Vernetzung vermitteln soll. Dabei wird ein Netzwerk aus virtuellen Rechnern mit einem fiktiven Betriebssystem abgebildet. Der Zugriff auf die Rechner erfolgt über eine per Telnet exportierte Konsole sowie über eine AJAX-gesteuerte grafische Oberfläche per Webbrowser. Das Ziel des Spiels ist das erfolgreiche Absolvieren von Missionen, für die die Kontrolle über eine große Anzahl virtueller Rechner Voraussetzung ist. Durch aktive Angriffe und Betrug verschafft sich der Spieler Zugang zu virtuellen Rechnern und muss diese gegen andere Spieler absichern und verteidigen. Dabei muss er stets darauf achten, von den eigentlichen Besitzern der virtuellen Rechner unentdeckt zu bleiben sowie durch geschicktes Verwischen von Spuren Verfolger wie Ermittlungsbehörden, Detekteien und Geheimdienste in die Irre zu führen.

Bei unseren Aktivitäten versuchen wir jedoch, uns nicht stur auf Perl zu beschränken. So bemühen wir uns auch um die Lösung der kleineren und größeren Probleme des Alltags, wie z.B. mit unserem Engagement für das noch recht junge *Hermes Project* [9], einen einfach zu konfigurierenden SMTP-Proxy, der transparent für den dahinter liegenden MTA eine ganze Reihe von Anti-Spam-Techniken implementiert. Wir unterstützen das Projekt mit Audits und daraus hervorgehenden Fehlerkorrekturen, Funktionserweiterungen sowie vor allem mit Vorträgen und Informationsständen auf lokalen Veranstaltungen rund um OpenSource.

Vorbeikommen!

Da Du diesen Text bis zum Ende durchgehalten hast und wir Dich nicht verschrecken konnten, bist Du bei der Ruhr.pm sicherlich genau richtig aufgehoben. Wir freuen uns immer über Besuch! Egal ob es sich um einen einmaligen Abstecher aus reinem Interesse handelt oder den Anfang einer regelmäßigen Teilnahme. Und sollte die Ruhr.pm einfach zu weit entfernt sein, besuche uns doch einfach in unserem IRC-Channel und auf unserer Mailingliste. Außerdem findet sich in diesem Fall mit Sicherheit eine andere PM-Gruppe in Deiner Nähe. Eine vollständige Übersicht bietet hier das *Verzeichnis der Perl Mongers* [10].

Veit Wahlich für Ruhr.pm

Links

[1] Perl Mongers im Ruhrgebiet: `http://ruhr.pm.org/`
[2] Café Maze: `http://www.maze-lounge.de/`
[3] Keybits OHG: `http://www.keybits.de/`
[4] IRC-Channel: `http://ruhr.pm.org/chat/>`
[5] Mailingliste: `http://ruhr.pm.org/mailingliste.psp`
[6] Jobbörse: `http://ruhr.pm.org/dev/jobboerse/`
[7] RJXML-Austauschformat: `http://ruhr.pm.org/specs/rjxml1/spezifikation.xml`
[8] Virtual Hacking: `http://virtual-hacking.de/`
[9] Hermes Project: `http://www.hermes-project.com/`
[10] Verzeichnis Europäischer Perl Mongers-Gruppen: `http://www.pm.org/groups/europe.html`

NEWS

Termine

November 2007

01.	Treffen Dresden.pm
03.	Linux-Info-Tag in Dresden
06.	Treffen Frankfurt.pm
	Treffen Stuttgart.pm
12.	Treffen Ruhr.pm
10./11.	Linuxtage in Essen
14.	Treffen Hamburg.pm
	Treffen Cologne.pm
15.	Treffen Darmstadt.pm
	Treffen Erlangen.pm
28.	Treffen Berlin.pm
	Treffen Bielefeld.pm

Dezember 2007

04.	Treffen Frankfurt.pm
	Treffen Stuttgart.pm
06.	Treffen Dresden.pm
10.	Treffen Ruhr.pm
12.	Treffen Hamburg.pm
	Treffen Cologne.pm
20.	Treffen Darmstadt.pm
	Treffen Erlangen.pm
26.	Treffen Berlin.pm
	Treffen Bielefeld.pm

Januar 2008

01.	Treffen Stuttgart.pm
03.	Treffen Dresden.pm
08.	Treffen Frankfurt.pm
09.	Treffen Hamburg.pm
	Treffen Cologne.pm
14.	Treffen Ruhr.pm
17.	Treffen Erlangen.pm
	Treffen Darmstadt.pm
30.	Treffen Berlin.pm
	Treffen Bielefeld.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-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.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. Der 10. Deutsche Perl-Workshop findet im Februar 2008 in Erlangen statt.

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.

// SEIBERT / MEDIA

„Statt mit blumigen Worten umschreiben unsere Programmierer den Job so: Apache, Catalyst, CGI, DBI, JSON, Log::Log4Perl, mod_perl, SOAP::Lite, XML::LibXML, YAML“

Professionell Perl programmieren lernen?

Wir suchen Einsteiger als Perl-Programmierer/innen (Vollzeit/Praktikum)

//SEIBERT/MEDIA besteht aus den drei Kompetenzfeldern Technologie, Consulting und Design und gehört zu den erfahrenen und professionellen Internet-Agenturen in Deutschland. Wir entwickeln seit 1996 mit heute knapp 50 Mitarbeitern Intranets, Extranet-Systeme, Web-Portale aber auch klassische Internet-Seiten. Seit 2005 konzipiert unsere Designabteilung hochwertige Unternehmensauftritte und kommunikative Konzepte. Beratung 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 http://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
HTTP://WWW.SEIBERT-MEDIA.NET/JOBS/

FREAKZEIT im Linuxhotel

DAS Wochenend-Reiseziel für alle, die Spaß an Computern haben

Wo andere nach Hamburg ins Musical oder in den Schwarzwald zur Wellness fahren, treffen sich Computerfreaks nun jedes Wochenende im Linuxhotel, um zusammen mit Gleichgesinnten am PC zu arbeiten.

Es erwartet Sie eine ruhige, konzentrierte Umgebung mit Leihnotebooks zum Testen kritischer Installationen, Seminarraum, Technik, Bibliothek, Probeservern, WLAN, 230V-Steckdosen selbst im 25.000qm großen Privatpark und vielem mehr. Auch Vollpension und Getränke sind enthalten, bei schönem Wetter wird gegrillt, wenn's kalt ist, geht es in die Sauna. Fahrräder stehen bereit, kleine Modellflieger, Fitnessraum, eine Wii, und vor allem all' der Kleinkram, den man für erfolgreiche Computerarbeiten braucht.

Warum nicht 'mal ein Wochenende unter Computerfreaks? Jeder arbeitet an seinem Thema, aber man schaut sich zwanglos über die Schultern und hilft sich gegenseitig!

Die Freakzeit-Wochenenden sind ein Experiment, denn eigentlich lebt das Linuxhotel von sehr zufriedenen Unternehmen und Organisationen, die ihre Mitarbeiter bei uns schulen lassen (wir haben eines der breitesten Kursprogramme in Deutschland, siehe www.Linuxhotel.de).

Doch die IT-Branche ist etwas Besonderes! Viele Profis haben wirklich Spaß an ihrer Arbeit, und es ist nicht einzusehen, warum z.B. Musical-Freunde oft mehrfach pro Jahr von weit her zu ihren Theatern fahren, doch für Computerfreaks gibt es nichts Vergleichbares!

Das Linuxhotel soll der Ort werden, an dem man jedes Wochenende interessante Leute aus unserer Branche zwanglos treffen kann. Machen Sie mit, schnappen Sie sich Ihr Notebook, und melden sich für irgendeines der kommenden Wochenenden an! Siehe www.Linuxhotel.de und auf „Freakzeit“ klicken.