Ausgabe 2/2008 – www.foo-magazin.de - 6,00 €

$foo

PERL MAGAZIN

Threads
Abläufe parallelisieren

TWiki
eines der führenden Open-Source-Wikis für Unternehmen

SANE Backend
für Canon USB Scanner

Nr. 06

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

astaro
internet security

e-mail | web | net | security

VORWORT

Hallo, Grüezi, Salut, Hola, Hoi...

Das deutschsprachige $foo-Magazin ist mittlerweile nicht nur bei Perl-Programmierern in Deutschland bekannt, sondern auch in verschiedenen anderen Ländern. Deshalb möchten wir an dieser Stelle unsere Leser aus Österreich, der Schweiz, Luxemburg, Israel, Dubai, den USA und Japan ganz herzlich begrüßen!

Das zeigt uns, dass wir mit dem Magazin auf dem richtigen Weg sind. Wir freuen uns über jeden neuen Leser - vielleicht können wir zukünftig auch noch Leser in Afrika und/oder Australien an dieser Stelle begrüßen.

Internationale Neuigkeiten rund um Perl sind seit ein paar Monaten im Internet zu finden: http://perl-nachrichten.de.

An dieser Stelle möchten wir auch auf die folgenden Veranstaltungen hinweisen, die in den nächsten Monaten weltweit stattfinden:

- YAPC::Asia in Tokyo (Japan)
- YAPC::NA in Chicago (USA)
- FrOSCon in Bonn (Deutschland)
- Nordischer Perl-Workshop in Stockholm (Schweden)
- Portugiesischer Perl-Workshop in Braga
- Russischer Perl-Workshop in Moskau
- Italienischer Perl-Workshop in Pisa
- Französicher Perl-Workshop in Albi

...

Die genauen Termine für die nächsten 3 Monate sind auf Seite 57 in dieser Ausgabe zu finden. Weitere Termine werden in der nächsten Ausgabe von $foo veröffentlicht. Wir wünschen allen Teilnehmern viel Spaß!

Sollte jemand von den Teilnehmern Interesse daran haben, über eine dieser Veranstaltungen einen Artikel für das $foo-Magazin zu schreiben, würden wir uns sehr freuen. Bitte einfach eine EMail an redaktion@foo-magazin.de senden.

Viel Spaß mit der 6. Ausgabe des $foo-Magazins!

Katrin Blechschmidt

Die Codebeispiele können mit dem Code

GHU34

von der Webseite www.foo-magazin.de 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

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

Renée Bäcker

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

Herbert Breunung

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

Jürgen Ernst

Seit 1999 ist er selbständiger Hardware- und Software-Entwickler (http://www.juergen-ernst.de/) und betreut als Spezialist für das Internet derzeit ca. 300 Internetpräsenzen. Perl setzt er bei fast allen Projekten ein, da sich so die Komplexität einer Anwendung deutlich reduzieren lässt. Auch kommen Projekte in der Open-Source-Szene nicht zu kurz. So hat er für Mozilla Firefox und Mozilla Thunderbird einige Erweiterungen geschrieben und arbeitet derzeit an einem Scanner-Backend für SANE.

Martin Fabiani

Martin Fabiani http://www.fabiani.net/ (34 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.

Andreas Romeyke

Andreas Romeyke studierte Telekommunikationsinformatik und arbeitet als Softwareentwickler am Max-Planck-Institut für Neuro- und Kognitionswissenschaften Leipzig. Er arbeitet seit über 10 Jahren mit Perl und zählt ‚Higher Order Perl' von Dominus zu den wichtigsten Perl-Büchern der letzten Jahre. Nebenbei ist Andreas Gründungsmitglied der Leipziger Linux User Group, sowie der Gesellschaft für die Anwendung offener Systeme e.V. Zu erreichen ist Andreas unter art1@andreas-romeyke.de.

Martin Seibert

Martin Seibert ist Geschäftsführer der //SEIBERT/MEDIA GmbH aus Wiesbaden. Die Multimedia-Agentur arbeitet seit 1996 mit heute knapp 60 Mitarbeitern. Sie ist eine der professionellen und erfahrenen Web-Agenturen in Deutschland und hat vier Bereiche Consulting (Strategie, Konzepte, Usability), Design (Webdesign, Printdesign, Corporate Design), Technologies (Frontend, Backend inkl. Perl und TWiki-Anpassungen) and Systems (Webhosting, Web Security). Martin Seibert ist zertifizierter TWiki-Berater und bietet mit //SEIBERT/MEDIA umfangreiche Dienstleistungen rund um TWiki von individuellen Hosting-Angeboten über Full-Service-Implementierungen inklusive Strategie, Konzeption, Design, Implementierung, Betrieb und Schulung. Wir setzen TWiki in unserem eigenen Intranet und bei einigen Kunden ein. Wir kennen die Erweiterungen und können mit unseren Perl-Programmierern Ergänzungen und Erweiterungen vornehmen.

Keine Angst vor Ties

Warum Ties?

In Perl kann man das Verhalten der meisten Standardvariablen überschreiben, indem man eine solche Variable an eine Klasse bindet (=Tie). Diese Variable kann man ganz normal verwenden, nur hinter den Kulissen passiert irgendwelche dunkle Magie, die auf Veränderungen dieser Variable reagiert und dann Aktionen veranlasst, die diese Variable normalerweise nicht beherrschen.

Bei Excel Perl (siehe Herbstausgabe des foo-Magazins) wird ein tied Array namens @F verwendet, das den Werten einer Zeile in einer Excel-Datei entspricht. Wenn man Elemente dieses Arrays modifiziert, modifiziert man damit auch den dazugehörigen Zellwert, z.B. setzt folgender Einzeiler die erste Spalte von datei.xls auf den Wert 42:

```
excelPerl.pl -ane " $F[0] = 42 " datei.xls
```

Auch viele CPAN-Modulautoren verwenden Ties, um Komplexität vom Anwender zu verbergen, z.B:

- *Tie::File* - bindet ein Array an eine Datei, wobei ein Element einer Zeile (konfigurierbar) entspricht. Wenn das Array verändert wird, werden diese Änderungen auch in der Datei durchgeführt.
- *DB File* - einfacher Zugriff auf Berkeley DBs.
- *Tie::Hash::Sorted* - speichert einen Hash in sortierter Reihenfolge.
- *Tie::Array::Sorted* - speichert eine Array in sortierter Reihenfolge.
- *Tie::Cycle* - bei jedem Aufruf einer skalaren Variablen wird der nächste Wert eines Zyklus zurückgegeben.
- *Tie::Registry* - einfacher Zugriff auf die Windows Registry

... und viele, viele mehr ...

Wie die Dokumentation zu diesen Modulen zeigt, ist die Verwendung von Ties meist sehr einfach und komfortabel. Aber auch die Implementierung von Ties ist nicht schwierig.

Scalar binden = Tie::Scalar

In länger laufenden Scripten muss man manchmal die aktuelle Uhrzeit ausgeben (z.B. für Logeinträge). Eigentlich könnte man sich eine Subroutine schreiben, die die aktuelle Zeit zurückgibt, und die dann jedesmal aufrufen. Man kann dies jedoch auch recht einfach mit Ties realisieren, wo bei Lesezugriffen auf die Variable (nennen wir sie `$actualTime`) die Zeit zurückgegeben wird und vielleicht bei Schreibzugriffen der zugewiesene Wert als neues Format für die zurückgegebene Zeit gesetzt wird (siehe POSIX -> strftime), z.B. Listing 1.

Die Implementierung dieser Klasse könnte folgendermaßen aussehen: siehe Listing 2.

(Ich empfehle, zum Spielen in jeder Methode ein paar prints reinzuschreiben, um so ein besseres Gefühl dafür zu bekommen, was da eigentlich passiert.)

Die wichtigen Methoden lauten also: TIESCALAR, STORE und FETCH.

Da in diesem Beispiel nur ein Wert im Objekt gespeichert wird (nämlich das Format der Uhrzeit), könnte man als Objekt auch problemlos eine Scalarreferenz anstelle der anonymen Hashreferenz verwenden und würde dabei ein klein wenig RAM sparen.

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

use TieMyTime; # Klasse mit der dunklen Magie laden

# $actualTime an diese Klasse binden und als Format HH:MM:SS mitgeben
tie( my $actualTime, 'TieMyTime', '%H:%M:%S' );

print "$actualTime: irgendeine Ausgabe\n";
sleep(3);
print "$actualTime: irgendeine andere Ausgabe\n";
$actualTime = '%Y.%m.%d %H:%M:%S';
print "$actualTime: Ausgabe mit anderem Format\n";

# Ab sofort brauchen wir keine Zeit mehr
untie( $actualTime );
```

Listing 1

```
package TieMyTime;
use warnings;
use strict;

use POSIX (); # wird zum Formatieren der Uhrzeit verwendet (oder Datum?)

# TIESCALAR wird aufgerufen, wenn eine Variable an 'TieMyTime' gebunden wird.
sub TIESCALAR {
    my( $class, $format ) = @_;

    $format = '%H:%M:%S' unless defined $format; # falls leer, Standard

    # Hashreferenz als Objekt erstellen, und darin das Format speichern
    my $self = bless( { format => $format }, $class );

    # und zurückgeben
    return $self;
}

sub FETCH {
    my( $self ) = @_;

    # einfach aktuelle Uhrzeit ermitteln, formatieren und zurueckgeben
    return POSIX::strftime( $self->{format}, localtime(time) );
}

sub STORE {
    # $newFormat bekommt den zugewiesenen Wert
    my( $self, $newFormat ) = @_;

    # neues Format setzen
    $self->{format} = $newFormat;

    return;
}

# UNTIE wird bei untie( $actualTime) aufgerufen:
sub UNTIE {
    my( $self ) = @_;
    print "Untie aufgerufen\n";
}

# Wenn es am Ende nochwas aufzuraeumen gibt, kann man dafuer auch
# DESTROY implementieren. Meist ist es jedoch besser, in UNTIE aufzuraeumen.
sub DESTROY {
    my( $self ) = @_;
    print "Ende der Zeit\n";
}

1;
```

Listing 2

```
package TieMyHashCI;
use warnings;
use strict;
use Carp qw(croak);

sub TIEHASH {
    my( $class ) = @_;
    return bless( {}, $class );
}

sub STORE {
    my( $self, $key, $value ) = @_;
    $self->{ lc $key } = { originalKey => $key, value => $value };
}

sub FETCH {
    my( $self, $key ) = @_;

    # Fehler bei Lesezugriff auf nicht existierenden Key?
    croak( "Fehler: key '$key' existiert nicht" )
        unless exists $self->{ lc $key };
    return $self->{ lc $key }->{value};
}

sub CLEAR { # wenn hash leere Liste zugewiesen wird
    my $self = shift;
    $self = {};
}

# die folgenden Methoden sind 1:1 mappings, nur mit lowercase
sub DELETE {
    my( $self, $key ) = @_;
    return delete $self->{lc $key};
}
sub EXISTS {
    my( $self, $key ) = @_;
    return exists $self->{ lc $key };
}
sub SCALAR {
    my $self = shift;
    return scalar %{ $self} ;
}

# wenn man mit each oder keys über einen Hash iteriert, wird bei jedem Aufruf von each das
# nächste Schlüssel-/Wertpaar zurückgegeben. Beim ersten Mal wird FIRSTKEY aufgerufen, da
# nach NEXTKEY, solange Werte vorhanden sind.
sub FIRSTKEY {
    my $self = shift;
    my $a = keys %{ $self }; # resette each iterator
    return $self->_mapEachKey2OriginalKey();
}

sub NEXTKEY {
    # $lastKey ist der letzte Key, ueber den iteriert wurde
    my( $self, $lastKey ) = @_;
    return $self->_mapEachKey2OriginalKey();
}

sub _mapEachKey2OriginalKey {
    my( $self ) = @_;

    my $keyLc = each %{ $self };
    ( defined $keyLc )
        ? return $self->{$keyLc}->{originalKey}
        : return;
}

sub UNTIE {
    my $self = shift;
    undef $self; # ist hier eigentlich ueberfluessig
}

1;
```

Listing 3

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

use TieMyHashCI;
tie( my %hash, 'TieMyHashCI' );

%hash = ( 'PeRl' => 'Mein Perl',
          'C'    => 'Mein C',
          );
$hash{TCL} = 'Mein Tcl';

print "C: $hash{c}\n";
print "Perl: $hash{Perl}\n\n";
foreach my $lang ( sort keys %hash ) {
    print "FOR: $lang: $hash{$lang}\n";
}

untie( %hash );
```

Listing 4

Arrays binden = Tie::Array

Genauso einfach geht es, Arrays zu binden. Dabei sind die folgenden Methoden wichtig:

- TIEARRAY (Klasse, weitere Parameter) - Bindet ein Array (ähnlich wie TIESCALAR)
- FETCH (Objekt, Index) - Holt Wert von Index
- STORE (Objekt, Index, Wert) - Speichert Wert an Index
- FETCHSIZE (Objekt) - gibt die Anzahl der Arrayelemente zurück (nicht den letzten Index)
- STORESIZE (Objekt, Anzahl) - setzt die neue Anzahl der Arrayelemente

Weitere optionale Methoden lauten: EXISTS, DELETE, CLEAR, PUSH, POP, SHIFT, UNSHIFT, SPLICE (implementieren die entsprechenden Perl-Befehle), EXTEND (um wieviel soll die Speicherstruktur, auf der das tied Array abgebildet wird, wachsen?), UNTIE, DESTROY

Codebeispiel: Siehe das CPAN-Modul Tie::Cycle

Hash binden = Tie::Hash

Einen Hash zu binden ist ein klein wenig anspruchsvoller. Hier ein Codebeispiel, um einen Hash zu bekommen, dessen Schlüssel unabhängig von Groß-/Kleinschreibung sind. Sowas verwende ich gerne, um Konfigurationsdaten zur Verfügung zu stellen. Dann brauche ich nicht andauernd nachschauen, wie denn jetzt die genaue Schreibweise in der Konfigurationsdatei nochmal war. Dabei verwende ich als Speicherstruktur gerne einen zweidimensionalen Hash, der in etwa folgendermaßen aussieht:

```
$self = {
   lc($key1) => {
      originalKey => $key1, value => $value1
   },
   lc($key2) => {
      originalKey => $key2, value => $value2
   },
};
```

So kann ich die Keys in Originalschreibweise zurückgeben, wenn nötig (siehe Listing 3). Und im Hauptprogramm kann man ihn folgendermaßen verwenden (siehe Listing 4).

Probleme und Lösungsideen bei Ties

Variablen verlieren beim tie ihre Inhalte

In diesem Fall kann man beim Tie noch weitere Parameter mitgeben und die dann in TIESCALAR, TIEARRAY oder TIEHASH setzen lassen, z.B.

```
my @array = 1..100;
tie( @array, 'TieMyArray', \@array );
```

und dann in TIEARRAY sowas wie das folgende schreiben:

```
my( $class, $values ) = @_;
# werte kopieren
return bless(
   { values => @{ $values } }, $class
);
```

Methoden direkt ausführen

Manchmal reichen die automatischen Methoden nicht aus, sondern man benötigt weitere, die auch von außen aufgerufen werden können. Hier kann man die Syntax `tied( %hash )->methodenName(...);` verwenden, z.B. `tied(%hash)->reset;`

Rückgabe von tied Variablen aus Funktionen

Wenn man tied Variablen aus einer Funktion ohne Referenzierung zurückgibt, verlieren sie alle Magie, weil die Rückgabewerte in eine Liste gepresst werden (z.B. return @tiedArray), und dort die Werte einfach reinkopiert werden. Deshalb die immer als Referenz zurückgeben.

Mehrdimensionale Ties

Es ist leider nicht so ohne weiteres möglich, mehrdimensionale Datenstrukturen zu binden. Für mein oben erwähntes Konfigurations-Darstellungsproblem habe ich häufig zweidimensionale Hashes. Da ich die auf einen Rutsch einlese (z.B. in den zweidimensionalen Hash %config), gehe ich meist in der Sub, die die Config einliest, folgendermaßen vor (siehe Listing 5).

```
tie( my %configHash, 'TieMyHashCI' );          # 1. Dimension tien
foreach my $level1Key ( %config ) {            # ueber 1. Dim iterieren
  tie( my %level2Hash, 'TieHashCI' );          # 2. Dim tien
  %level2Hash = %{ $config{$level1Key} };      # kopieren
  $configHash{$level1Key} = \%level2Hash;      # in 1. Dim speichern
}
return \%configHash;                           # als Referenz zurückgeben
```

Listing 5

Wann machen Ties keinen Sinn?

Ties sind eine sehr mächtige Waffe, die vom Anwender sehr einfach verwendet werden kann. Da jedoch eine Variable an eine Klasse gebunden wird, wird bei jedem Zugriff darauf eine Methode der Klasse aufgerufen, was langsamer ist als mit einem ungebundenen Hash zu arbeiten. Wenn man sehr schnell laufenden Code benötigt, ist es häufig nicht sinnvoll, mit TieMyHashCI zu arbeiten, sondern es ist schneller, wenn man selbst die Schlüssel auf Groß- oder Kleinschreibung umstellt.

Weitere Informationen

siehe:

- perldoc perltie
- Programmieren mit Perl
- CPAN-Module, die Ties verwenden

Fazit

Die Technik der Ties ist nicht besonders komplex. Die Verwendung eines Ties ist meist sehr einfach. Deshalb sollte meiner Meinung nach die Fähigkeit, Variablen zu binden, im Werkzeugkasten jedes fortgeschrittenen Perl-Programmierers vorhanden sein.

Martin Fabiani

TPF-TICKER

Veränderungen im „Grants Committee“

Das „Grants Committee“ hat sich in letzter Zeit etwas verändert: Hugo van der Sanden und Stas Bekman haben das Gremium verlassen und Will „Coke“ Coleda sowie Perrin Harkins haben die freien Plätze eingenommen.

TPF wählt neue Mitglieder

Die Perl Foundation hat drei neue Mitglieder gewählt.

* Karen Pauley, Steering Committee Chair
* Josh McAdams, Public Relations
* Jeremy Fluhmann, Conferences Committee Chair

Andy Lester, der die Position des PR-Managers aufgibt, wird weiterhin für PR und Buzz via Perlbuzz sorgen.

Threads - Abläufe parallelisieren

Es läuft und läuft und läuft... Ein Blick auf die Uhr verrät, dass das Programm schon eine halbe Ewigkeit läuft. Naja, wenigstens meint man das und in manchen Fällen sind schon Minuten zu lang für einen Programmdurchlauf. In solchen Situationen wünscht man sich, die Abläufe zu parallelisieren. In Perl gibt es mehrere Möglichkeiten dazu, aber jede hat ihre Vor- und Nachteile.

In diesem Artikel möchte ich zeigen, wie man seinem Ziel - die Parallelisierung von Abläufen - etwas näher kommt und dabei auf zweieinhalb Lösungen eingehen. Zweieinhalb? Ja, zweieinhalb. Die letzte Lösung ist keine komplett neue Lösung, sondern eine Mischung aus den beiden ersten Möglichkeiten.

Begriffe

Bevor wir uns hier darum kümmern wie man die Abläufe parallelisieren kann, müssen ein paar Begriffe geklärt werden, die in diesem Artikel noch häufiger verwendet werden.

Prozess

Der erste Begriff ist der „Prozess“. Als „Prozess“ fasst man den Programmcode und die Daten im Arbeitsspeicher zusammen. Weiterhin gehören noch Register im Prozessor, Stack, Puffer und Filehandles dazu. Die letztgenannten Dinge werden auch als „Kontext“ bezeichnet.

Thread

Ein Thread ist ein Teil eines Prozesses. Ein Prozess kann mehrere Threads haben, die sich einige Ressourcen des Prozesses teilen. Nicht geteilt werden ein eigener Stack und Befehlszähler.

fork

Ein Prozess kann mit dem Systemaufruf fork() geklont werden. Das heißt, es wird eine identische Kopie des ursprünglichen Prozesses erzeugt. fork() erwartet keine Parameter und gibt einen numerischen Wert (Integer) zurück. Das Duplikat, das als Child-Prozess bezeichnet wird, übernimmt die aktuellen Werte aller Variablen und weiterer Datenstrukturen.

Man kann sich das ganze bildlich so vorstellen: Ein Mann lässt sich klonen. Der Klon erwacht im Zimmer nebenan und hat die gleichen Erinnerungen wie das Original; einschließlich dem Betreten des anderen Zimmers, in dem noch das Original ist. Die Kopie denkt dann „Ups, bin ich vorhin nicht in das andere Zimmer gegangen? Und wer ist der nette Herr in dem anderen Zimmer?“.

Original und Kopie arbeiten aber ab dem fork() unabhängig voneinander. Das heißt, die Kopie kann in die Stadt einkaufen gehen, ohne dass das Original etwas davon mitbekommt - und umgekehrt. Ab dieser Stelle hat also jeder der beiden Prozesse alle bisher benutzten Variablen mit ihren aktuellen Belegungen zur Verfügung; Änderungen dieser Belegungen wirken sich aber nur noch innerhalb des eigenen Prozesses aus, der andere Prozess bekommt davon nichts mit. Nur Filehandels werden nicht kopiert - beide Prozesse schreiben und lesen aus den bzw. in die selben Filehandles.

```
Ursprungs-   =   Child-    +   Parent-
prozess          prozess        prozess
```

Um Probleme zu vermeiden, müssen Parent- und Child-Prozess wissen, wer von ihnen wer ist. Dieses Problem wird mit den sogenannten Prozess-IDs (kurz: PID) gelöst. Dies sind eindeutige positive Ganzzahlen. Um die Abarbeitung von

Aufgaben zu beschleunigen bzw. zu parallelisieren, müssen sowohl der Kind- als auch der Vaterprozess das gleiche tun (Listing 1).

Doch welche Ausgabe kommt bei dem Beispiel? Es kommt darauf an... Und genau das ist das „Problem“ hierbei. Da nicht festgelegt ist, wann welcher Prozess CPU-Zeit zugeteilt bekommt, ist auch die Ausgabe nicht vorhersehbar. Aber bei vielen Programmen ist dies auch nicht notwendig. In einem Bio-Informatik-Projekt konnte ich mit fork die Bearbeitungszeit von Berechnungen von 3 auf 1 Tag reduzieren.

Allerdings muss man dabei auch aufpassen, dass man nicht zu viele Kindprozesse erzeugt. Einmal habe ich es mitbekommen, wie ein Python-Programm eines Werkstudenten einen Server nahezu lahmgelegt hat, weil zigtausend Kindprozesse erzeugt wurden. Und da bei fork alles kopiert wird (Speicher, Heap, etc.), steigt der Speicherverbrauch sehr stark an und der Rechner gerät leicht ins Swappen.

Interprozesskommunikation

Diese Unabhängigkeit kann aber auch zu einem Problem werden. Was ist, wenn Original und Kopie miteinander kommunizieren müssen?

Die Kommunikation kann man über verschiedene Wege sicherstellen. Z.B. über Shared Memory, Signale oder Pipes. Pipes kann man sich als „Röhre“ vorstellen, über die in nur eine Richtung kommuniziert werden kann. Dies sieht so aus, wie in Listing 2 dargestellt.

```
my @kid_values    = (1..10);
my @parent_values = (11..19);

my $pid = fork();
if($pid == 0){
  mach_was( @kid_values );
  exit(0);
}
else{
  mach_was( @parent_values );
  wait();
}

sub mach_was{
   print $_,"\n" for @_;
   # irgendwelche Rechenintensiven Sachen
}
```

Listing 1

Ausgabe:

```
~/entwicklung 21> perl fork.pl
Kam vom Kind: Hallo
```

Der Parent-Prozess lauscht so lange an der Pipes, bis der Child-Prozess etwas in die Pipe schreibt.

fork ist sehr gut für die Nebenläufige Programmierung geeignet, aber Programme werden relativ schnell komplex und gerade für Einsteiger ist fork unübersichtlich.

Perl-Threads

Was Threads im Sinne der Informatik sind, wurde schon weiter oben geklärt. Kommen wir jetzt also zu den Threads wie man sie in Perl verwenden kann. Das unterscheidet sich nämlich etwas von der Definition, die oben gegeben wurde.

Das erste Thread-Modell wurde in Perl 5.005 eingeführt (`use Thread`), was aber ziemlich unausgereift war und deshalb wurden mit Perl 5.6 die sogenannten Interpreter-Threads (ithreads - `use threads`) eingeführt. Der größte Unterschied zwischen den ithreads und den 5.005-Threads ist, dass in den ithreads die Daten explizit ge“shared“ werden müssen.
Erst in Perl 5.8 wurde die Thread-Implementierung stabil. In Perl 5.10 wurden die 5.005-Threads komplett entfernt, so dass es nur noch die sogenannten Interpreter-Threads gibt.

Um Threads benutzen zu können, muss Perl schon mit Threadunterstützung kompiliert werden. Ob das eigene Perl mit Threadunterstützung kompiliert wurde, kann man mit `perl -V` überprüfen. Dort wird die Konfiguration von Perl

```
pipe(READER,WRITER);
my $pid = fork();
if($pid == 0){
  close READER;
  print WRITER 'Hallo';
  exit(0);
}
else{
  close WRITER;
  while(my $line = <READER>){
    print "Kam vom Kind: ",$line,"\n";
  }
  wait();
}
```

Listing 2

angezeigt. Unter anderem auch so etwas wie

```
usethreads=define use5005threads=undef \
    useithreads=define usemultiplicity=define
```

Hier sieht man, dass die „alten“ Threads nicht unterstützt werden, dafür aber die Interpreter-Threads.

Innerhalb eines Programms, kann man die Angaben über das Config-Modul von Perl erfragen.

```
use Config;
print "ithreads defined?",
$Config{useithreads} ? "ja" : "nein";
```

Man muss sich allerdings überlegen, ob man die Threadunterstützung benötigt, denn diese kostet für „normale“ Programme etwas Performance. Wie groß dieser Performanceverlust ist, lässt sich im folgenden Code ersehen. Die hier gezeigten Werte sind ein Mittel über 10 Läufe des Programms.

```
reneeb~$ /perl510/bin/perl gen_code.pl
336
reneeb~$ /perl510_thr/bin/perl gen_code.pl
368
```

Man sieht, dass das Perl mit Thread-Unterstützung für die Ausführung deutlich länger benötigt als das Perl ohne Threadunterstützung.

Das Programm für den Benchmark ist in Listing 3 zu sehen. Als Perlversion wurde 5.10.0 genommen, je einmal mit Threads und einmal ohne Threads - sonst jedoch die gleichen Parameter für `Configure`. Wer also voraussichtlich keine Threads benötigt und sich sein Perl selbst kompiliert, kann auch darauf verzichten.

Threads verwenden

Das `threads`-Modul bringt eine einfache API mit, mit der Threads verwendet werden können.

```
for my $name ( @accounts ){
   my $thread = threads->create(
       \&start_thread, $name
   );
   $thread->detach;
}

sub start_thread{
    print "Starte Thread mit Namen $name\n";
    # starte Berechnungen
}
```

Die einzelnen Funktionen können in der Dokumentation von `threads` (`perldoc threads`) nachgelesen werden. In diesem Artikel geht es mehr um die grundsätzlichen Ideen zur Nebenläufigen Programmierung.

Bei der Verwendung von Threads ist zu beachten, dass durch das Kopieren der gesamten Daten des Hauptthreads der Speicherverbrauch und die Performance eher schlechter wird. Deshalb sollten Threads erzeugt werden, so lange der Hauptthread noch nicht übermäßig viele Daten gesammelt hat. Auch die Anzahl der Threads sollte gut bedacht gewählt werden.

Eine Variable für alle

Wie auch bei den forks können die Threads standardmäßig untereinander keine Variablen teilen. Da dies aber häufig notwendig ist, gibt es das Modul `threads::shared`. Damit können Variablen mit dem Attribut `:shared` versehen werden. Bei komplexeren Datenstrukturen ist aber zu bedenken, dass jede „Ebene“ einzelnen geteilt werden müssen.

```
my %progressor = ();
my $account    = Email::Config->new( $yaml );
my @accounts   = $account->names;

for my $name ( @accounts ){
    # alle Threads sollen auf den Hashwert
    # von jedem Account zugreifen können
    $progressor{$name} = &share([]);
}
```

Wenn man Variablen `shared`, muss dabei auch bedacht werden, dass die Reihenfolge, in der Threads auf diese Variablen zugreifen, nicht vorhersagbar ist. So kann es passieren, dass ein Thread in der einen Code-Zeile den Wert verändert, sich auf diese Änderung in der nächsten Code-Zeile verlässt, ein anderer Thread zwischen der Ausführung dieser beiden Codezeilen auch auf die Variable zugegriffen hat und diese verändert.

Um solch ein Verhalten zu vermeiden, können die Variablen `lock`t werden. Hierbei sichert sich ein Thread das exklusive Zugriffsrecht, solange der aktuelle Block nicht verlassen wird (siehe Listing 4).

Thread-Modelle

Es gibt verschiedene Modelle, wie man Threads einsetzen kann und es hängt vom Zweck des Programms ab, welches Modell eingesetzt werden soll. Das „Boss/Worker“-Modell eignet sich vor allem bei GUIs und bei Servern, bei denen ein

```
#!/usr/bin/perl

use YAML::Tiny;

my $string = do{ local $/; <DATA> };
for( 1..1_000_000 ){
  my $yaml   = YAML::Tiny->read_string( $string );
  my $config = $yaml->[0];

  my $code = '';
  variable_declaration( \$code, $config->{root} );
}

print time - $^T,"\n";

sub variable_declaration{
    my ($coderef,$config) = @_;

    for my $name ( keys %$config ){
        if( ref($config->{$name}) eq 'HASH' ){
            declare_hash( $coderef, $config->{$name}, $name );
        }
    }
}

sub declare_hash{
    my ($coderef,$config,$name) = @_;

    $$coderef .= 'my %' . $name . " = (\n";

    for my $key ( keys %$config ){
        my $val = $config->{$key}->{value};
           $val = $config->{$key}->{type} eq 'qr' ? "qr/$val/" : $val;
        $$coderef  .= "$key => $val,\n"
    }

    $$coderef .= ");\n";
}

__DATA__
---
root:
  regex:
    nameCheck:
      type: qr
      value: tr0nix
    nrCheck:
      type: qr
      value: ^\d$
    emptyCheck:
      type: qr
      value: ^$
```

Listing 3

Thread auf neue Aufgaben wartet und diese dann auf die sogenannten „Worker“-Threads verteilt. Die „Working Crew“ kann man gut verwenden, wenn eine größere Anzahl von Threads mehr oder weniger die gleichen Aufgaben erledigen sollen, nur mit etwas anderen Daten. Ein Beispiel wäre die Bearbeitung aller Dateien eines Verzeichnisses. Ein Thread ist für alle Dateien, die mit ‚A‘ anfangen, zuständig - ein anderer Thread für alle Dateien mit dem Anfangsbuchstaben ‚B‘ und so weiter. Bei der „Pipeline“ ist jeder Thread für einen gewissen Schritt in einem Workflow zuständig. Ist der Schritt abgearbeitet, wird das Ergebnis an den Thread weitergereicht, der für den nächsten Schritt zuständig ist.

Fallen

Thread-sichere Module

Einige auf CPAN - und selbst ein paar Standardmodule - sind nicht Thread-sicher. In einigen Modulen steht schon in der Dokumentation der Hinweis, ob das Modul „Thread-safe“ ist oder nicht. Im Zweifelsfall, muss man es einfach ausprobieren. Generell sollte man ersteinmal davon ausgehen, dass ein Modul nicht Thread-sicher ist - wenn es nicht anders dokumentiert ist.

```
my $variable : shared;
sub test{
   lock( $variable );
   sleep 3;
   $variable++;
} # aktueller Block wird verlassen,
  #lock wird aufgelöst

for (1..3){
   threads->create( \&test );
}
```

Listing 4

```
for my $name ( @accounts ){
   my $thread = threads->create(
       \&start_thread, $name
   );
   $thread->detach;
}

create_gui();

sub start_thread{
   print "Starte Thread mit Namen $name\n";
     # starte Berechnungen
}

sub create_gui{
   require Tk;
   require Tk::Progressbar;
   my $mw = Tk::tkinit();
   $mw->resizable( 0, 0 );
   $mw->configure( -title => 'Test' );
   $mw->Label( -text => 'Testlabel' )
      ->pack;
   Tk::MainLoop();
}
```

Listing 5

```
use threads;

for my $name ( @accounts ){
  my $thread = threads->create(
      \&start_thread, $name
  );
  $thread->detach;
}

sub start_thread{
  print "Starte Thread mit Namen $name\n";
      # starte Berechnungen
}
```

Listing 6

```
use forks;

for my $name ( @accounts ){
  my $thread = threads->create(
      \&start_thread, $name
  );
  $thread->detach;
}

sub start_thread{
  print "Starte Thread mit Namen $name\n";
  # starte Berechnungen
}
```

Listing 7

Tk und Threads

GUIs haben das Problem, dass sie „einfrieren“, wenn eine länger andauernde Aufgabe erledigt wird. Ich hatte dieses Problem bei dem „SWS Website Backups“-Programm. In dem Programm werden von (mehreren) Webseiten Backups über Confixx und FTP erzeugt. Bei größeren Webseiten kann das lange dauern und wenn viele Webseiten so gesichert werden sollen, dann legt das unter Umständen die GUI für mehrere Stunden lahm. Hier wurden dann Threads eingesetzt.

Im Zusammenspiel Tk <-> Threads gibt es aber das Problem, dass das Programm nicht funktioniert, wenn die Threads erst erzeugt werden wenn die GUI schon erstellt wurde. Wer Threads und Perl/Tk zusammenbringen möchte, muss erst die Threads erzeugen und dann die GUI erstellen (siehe Listing 5). Zu beachten ist dabei, dass `require Tk` an Stelle von `use TK` verwendet wird.

weitere Fallen

Es gibt noch mehr Fallstricke mit Threads, die hier aber nicht eingehender betrachtet werden sollen.

Änderungen für alle Threads

Auch wenn Threads für sich alleine agieren, gibt es ein paar Befehle, die sich auf alle Threads auswirken. Diese sollten nur mit äußerster Vorsicht angewendet werden. Wird das Verzeichnis in einem Thread mit `chdir` gewechselt, so gilt diese Änderung genauso für alle anderen Threads. Weitere Befehle dieser Art sind im Threads-Tutorial (`perldoc perlthrtut`) aufgeführt.

use forks;

Dies ist die „zweieinhalbte“ Lösung. Sie übernimmt die bessere Syntax ohne die Nachteile der Perl-Threads zu verwenden. Im Hintergrund läuft, wie der Name schon sagt, ein fork. So kann mit nur wenigen Codeänderungen der Code von ithreads auf fork umgestellt werden.

Der alte ithreads-Code ist in Listing 6 zu sehen im Vergleich dazu der Code mit `use fork` in Listing 7.

Renée Bäcker

Perl Snippets

Vorwort

In dieser Reihe sollen zukünftig Perl-Schnipsel vorgestellt werden, die auf den ersten Blick völlig unscheinbar daherkommen und sich dann als wahre Superhelden entpuppen.

Die Herausforderung

Ein Auswertescript sollte in einer Liste jedes Vorkommen eines bestimmten Elementes durch ein oder mehrere andere Elemente ersetzen.
Zum Beispiel werden an Stelle von ‚1' die Elemente ‚o','n','e', an Stelle von ‚4' die Elemente ‚f','o','u','r' und statt ‚9', die Elemente ‚n', ‚i', ‚n', ‚e' eingefügt.

```
#!/usr/bin/perl -w
use strict;
# Liste
my @array = (1, 2, 3, 4, 5, 6 ,7 ,8, 9);
# Elemente die ersetzt werden sollen
my %hash  = (
   1 => ['o','n','e'],
   4 => ['f','o','u','r'],
   9 => ['n','i','n','e']
);
# Ersetzung
my @tmparray;
foreach $_ (@array) {
   exists $hash{$_} ?
     push @tmparray, @{$hash{$_}} :
     push @tmparray,$_
}
@array=@tmparray;
print join (" ",@array), "\n";
```

Das Script schreibt auf die Ausgabe daher

```
o n e 2 3 f o u r 5 6 7 8 n i n e
```

und hat damit seine Funktionalität unter Beweis gestellt.

Die Stärke von map

Das `push` auf ein Array kostet Zeit. Daher ist zu erwarten, dass `map` die Aufgabe schneller lösen kann. Man beachte, dass Perls `map` im Gegensatz zu anderen funktionalen Sprachen auch Teillisten statt einzelner Listenelemente zurückgeben kann. Daher kann das Script abgewandelt werden zu:

```
#!/usr/bin/perl -w
use strict;
# Liste
my @array = (1, 2, 3, 4, 5, 6 ,7 ,8, 9);
# Elemente die ersetzt werden sollen
my %hash  = (
   1 => ['o','n','e'],
   4 => ['f','o','u','r'],
   9 => ['n','i','n','e']
);
# Ersetzung
@array = map {
   exists $hash{$_} ? @{$hash{$_}} : $_
}  @array;
print join (" ",@array), "\n";
```

Es ist zwar bekannt, dass `map` schneller als eine `foreach`-Schleife arbeitet. Überraschend ist aber, dass `map` mehr als das 4-fache herausholt. In der Originalversion schaffte `map` die Listenverarbeitung um 10-er Potenzen schneller. Daher müsste dieser Schnipsel eigentlich unter dem Motto „The power of ‚map'" stehen.

Der Vorteil von `map` kommt insbesondere dann zum Tragen, wenn in einer Liste Elemente mehrfach nach einem Muster ersetzt werden sollen.

Der folgende Benchmark verdeutlicht dies:

```
#!/usr/bin/perl -w
use strict;
use Data::Dumper;
use Benchmark qw(:all) ;
# Liste
my @array = (1, 2, 3, 4, 5, 6 ,7 ,8, 9)x10;
# Elemente die ersetzt werden sollen
my %hash   = (
   1 => [`o`,`n`,`e`],
   4 => [`f`,`o`,`u`,`r`],
   9 => [`n`,`i`,`n`,`e`]
);
my $count=50000;
my $s_foreach = sub {
    my @tmparray;
    foreach $_ (@array) {
        exists $hash{$_} ?
            push @tmparray, @{$hash{$_}} :
            push @tmparray,$_
    }
};
my $s_map = sub {
    map { exists $hash{$_} ?
              @{$hash{$_}} : $_}  @array;
};
cmpthese(
    $count, {
        `foreach` => $s_foreach,
              `map` => $s_map,
          }
);
```

Als Ergebnis spuckt das Benchmark-Modul folgende Zeilen aus:

```
           Rate     foreach      map
foreach  6435/s          --    -69%
map     20492/s       218%      --
```

Wenn ihr ähnliche Beispiele für Perl-Schnipsel kennt, die sich überraschend als wahre Superhelden entpuppen, schreibt mir bitte per Email an: foo@andreas-romeyke.de.

Andreas Romeyke

TPF-TICKER

TPF für Artistic License

Die Perl-Foundation unterstützt ein Java-Projekt, das unter der Artistic License veröffentlich ist und jetzt in einer Lizenzstreitigkeit vor Gericht ist. Das Urteil könnte Auswirkungen auf Copyright-Rechte von Open Source Projekten haben.

Parrot Grant Update Januar/Februar

Seit Parrot 0.5.1 im Dezember wurde wieder einiges an Parrot geändert. So wurde viel Code aufgeräumt und die Perl6-Implementierung für Parrot heißt jetzt „Rakudo“. Mit Parrot 0.5.3 wurden auch einige Meilensteine erreicht, die dazu führen, dass Parrot-Entwickler Geld von der Perl-Foundation erhalten.

Bessere Performance von RT

Das Ticketsystem, das für die CPAN-Module unter http://rt.cpan.org läuft, ist jetzt schneller. Best Practical hat einige Codeveränderungen vorgenommen, die das träge System beschleunigen. In einigen Tagen will Best Practical die Änderungen allen RT-Nutzern zugänglich machen.

Perl 6 - Der Himmel für Programmierer - Update 1

In der zweiten $foo-Ausgabe, im Sommer 2007, gab es einen langen Artikel zu lesen, der die glorreiche und manchmal eigenwillig scheinende Entwicklung des Projektes „Perl 6“ beschrieb. Darin wurde gleichfalls erklärt was sich hinter Schlagworten wie Pugs, Parrot und Ponie verbirgt und somit sämtliche Verwirrung diesbezüglich aufgelöst. Doch die Zeit schritt weiter und heutzutage kursieren Begriffe wie NQP, Rakudo, kp6 und SMOP. Dieses scheinbare Durcheinander soll nun die folgende Aktualisierung aufzulösen.

Die Kurzübersicht

Es hat sich viel getan im letzten Jahr. Neue Ansätze wie etwa Rakudo, SMOP oder die STD.pm sind entstanden, Parrot und seine Werkzeuge (PCT aka Parrot Compiler Tools) haben sich gut weiterentwickelt und um Pugs ist es in den letzten Monaten ruhig geworden.

Pugs und Konsorten

Manchen ist das selbstverständlich Anlass genug, den oft angekündigten Untergang der Perlwelt zu datieren. Denn die einzige, halbwegs vollständige Implememention wurde krankheitsbedingt von ihrer Hauptentwicklerin verlassen. Andererseits hat Perl 6 Pugs sehr viel zu verdanken und es hat auch immer psychologische Ursachen, welchen Tatsachen ein Mensch mehr Beachtung schenkt. Das greifbarste Beispiel für Pugs Erfolg ist seine riesige, in Perl 6 geschriebene Testsuite. Sie besteht aus 19000 Tests in über 700 Dateien und wurde zur Offiziellen gekürt, an der sich jeder kommende Perl 6-Interpreter messen lassen muß. Weniger greifbar, aber auch sehr wertvoll waren die von Pugs-Entwicklern angeregten Verbesserungen im Syntax. Denn während der Implementation wurden regelmäßig offen gelassene Grenzfälle und Widersprüche der Spezifikation gefunden. Auch das erste Ausprobieren der Sprache mit Pugs brachte Konsequenzen der Grundregeln und deren Nachbesserungen zu tage, die auf dem Papier kaum überschaubar waren. Letztlich hat auch die Beschäftigung mit Haskell Perl 6 um einige schöne und trickreiche Funktionen bereichert (mehr dazu im dritten Teil des Perl 6-Tutorials, in dieser Ausgabe). Die durch Pugs gesammelte Erfahrung half darüber hinaus einzelne Funktionen nach Perl 5 in Form von Perl6::* Modulen zu portieren (siehe Bundle::Perl6 aber auch weitere wie Moose). Sogar die Programmierung der neuen (an Perl 6 angelehnten) Funktionen in Perl 5.10 war dank Pugs leichter.

Gibt es also einen echten Grund zu trauern? Eigentlich nicht, da die Situation heute eine völlig andere ist als im Februar 2005. Die damalige Motivationsflaute hat Pugs deutlich vertrieben, in dem es mit zwanglosen Spieltrieb, ungewöhnlichen Methoden und sichtbaren Ergebnissen Aumerksamkeit und etliche Folgeaktivitäten provozierte. Damit lenkte es auch eine Zeit lang von Parrot ab, daß einige interne Probleme hatte. Während die PCT (Parrot Compiler Tools-letztens noch als Partridge vorgestellt) sich durch die stetige Arbeit von Patrick Michaud und Allison Randal weiterentwickelten, war die Entwicklung der eigentlichen VM sporadischer. Nach dem Weggang von Leopold Tötsch brauchte es auch Zeit, bis das Projekt zu einer Arbeit fand, die auf genügend Schultern verteilt ist.

Auch viele Unsicherheiten, die in den ersten Jahren noch Thema waren, wie „Was wird mit Perl 5?“ oder „Werden sich die verschiedenen Projekte nicht behindern?“ usw. sind heutzutage (in Kreisen die mit Perl 6 zu tun haben) weitestgehend überwunden. Folglich kann Pugs, daß eh niemals als produktionsreifer Kompiler geplant war in Frieden gehen. Es wird auch von Rakudo abgelöst, daß einmal der maßgebende Perl 6-Interpreter werden soll. Mehr dazu in einem späteren Abschnitt.

Aus dem letztens vorgestellten Mini-Perl entwickelte sich innerhalb des Projektes Pugs sogar mit „kp6“ ein vollständiger Kompiler, der eine Untermenge von Perl 6 (kp6 - kind‘a (of) Perl 6) vom Perl 5-Interpreter ausführen lässt. Flavio S. Glock begann mit „v6“ überdies einen Sourcefilter, der gleichfalls (wenn auch auf eine andere Art) Perl 6-Programmierung mit dem alten Interpreter ermöglichen sollte. Innerhalb beider Vorhaben geschieht allerdings zur Zeit nicht viel.

In der Reihe derzeit (größtenteils) inaktiver Projekte steht auch MAD, der Perl 5-zu-Perl 6-Übersetzer. MAD steht für Misc. Attribute Decorator, was sicher nur eine Ausrede des Herren Wall ist, dem ganzen einen witzigen Namen zu geben. Sage LaTorra schrieb eine frühe MAD-Version während des „Summer of Code“ 2006. Er wurde dafür von Larry angeleitet und von Google bezahlt. Seit dem rührte sich aber auch dort kaum etwas.

Was macht Larry eigentlich gerade?

Und das ist auch nicht schlimm, da Larry Wall, der MAD zu seinem persönlichem Vorhaben erklärte, weil es (Zitat:) „irrsinnig gut“ werden soll, derzeit weit wichtigereres tut. Etwas das dazu beiträgt, einen produktionsreifen Interpreter so schnell wie möglich zu erzeugen. Parallel zu den Synopsen pflegt er nämlich seit einigen Monaten noch eine Datei, die im Pugs SVN unter „src/perl6/STD.pm“ zu finden ist und bereits auf über 100kB angewachsen ist. Diese STD.pm definiert, nach welchen Regeln ein Parser Perl 6-Quellcode in seine Grundbausteine zerlegen soll und noch ein paar zugehörige Dinge wie etwa die Vorrangwerte für Operatoren. Diese Syntaxdefinition ist sinnigerweise auch in Perl 6-rules geschrieben, die mit ihren mächtigen und ableitbaren Grammatiken perfekt dafür geeignet sind. Perl hat dadurch auch zum ersten mal in seiner Geschichte eine eindeutige und sauber lesbare Spezifikation, besser noch: eine Spezifikation, die funktionaler Teil des Parsers ist. Dies soll helfen einen korrekten und leicht änderbaren Interpreter zu schreiben und zu pflegen. Bleibt nur noch das kleine Henne und Ei Problem, daß es einen Perl 6-Interpreter braucht, um solch einen zu erzeugen. Genau genommen braucht es aber „nur“ einen Compiler, der „rules“ in etwas Ausführbares umwandelt. Und so etwas gibt es schon mit der PGE. Jedoch sind PGE und die STD.pm noch nicht so weit gediegen, um miteinander zu arbeiten. Deshalb bemühen sich derzeit einige der geschäftigsten Unternehmungen die STD.pm auf anderen Wegen, wie auf pugs, perl, ruby oder was sich sonst anbietet, auszuführen. Hierzu ließen sich viele Codenamen (wie STD_red, gimme5, metholate etc.) aufzählen und erklären, aber all dies sind bisher Versuche und vieles davon im Aufbau oder Wandlung. Wenn sich daraus etwas entscheidendes ergibt, wird $foo darüber berichten. Kommen wir deshalb zum langlebigsten aller Perl 6-Projete, daß wichtige Fortschritte zu melden hat.

Die neuen Kunststücke des Papageis

Vor einem Jahr war das nichtvorhandene OOP-Modul (wie berichtet) noch ein großes Hindernis für Parrot‘s Fortkommen. Deshalb war die Erleichterung gewaltig, als auch dieser Abschnitt spezifiziert und programmiert wurde. Dies ist sicher noch nicht die optimale und endgültige Implementation, aber eine funktionierende Objektorientierung ermöglichte die Enstehung von NQP (Not Quit Perl 6), einem Parrot-Parser für eine Untermenge von Perl 6, der weitestgehend fertiggestellt ist und nur bei Bedarf noch stellenweise erweitert wird. Der Sinn von NQP ist nämlich nicht noch einen halbfertigen Perl 6-Interpreter zu bauen, sondern eine Alternative für PIR zu schaffen, die z.B. durch eine kompaktere Art der Parameterübergabe (Signaturen) glänzt, wie sie aus Hochsprachen vertraut ist. Somit ist NQP auf der einen Seite ein Parser für eine eigene Sprache, andererseits auch ein Werkzeug für Parrot-Programmierer, um noch schneller zu Ergebnissen zu kommen.

Es stellte sich zum Beispiel heraus, daß Umwandlungen von Baumstrukturen mit NQP einfacher programmierbar sind, als mit der TGE. Während Parrot-Parser Quellcode in Bytecode umwandeln, durchlaufen diese Daten mehrere baumförmige Zwischenstufen. Die Tree Grammar Engine suchte dabei Pfade ab und wandelte die Inhalte der Knoten nach Regeln um. Implementiert man dagegen die Knoten in NQP als Objekte und ihre Transformation mit Multimethoden dieser Objekte, kann man die Unterscheidungen je Datenyp des Knoteninhaltes und Pflege der Datenstruktur an die Programmiersprache delegieren, was auf jeden Fall kürzer und robuster ist. Daß NQP die TGE ersetzt hat überdies den positiven organisatorischen Nebeneffekt, daß TGE-Entwicklerin Allison Randal somit Parrot‘s Chefarchitektin werden konnte. Dies ist derzeit die beste Wahl, da sie nicht nur die dienstälteste aktive Entwicklerin ist und sehr motiviert obendrein, sondern weil sie sich in den letzten Jahren auch das nötige

Fachwissen angeeignet hat. Es wird nämlich oft übersehen, daß die JavaVM und .Net zwar von großen Firmen verbreitet werden, aber rein technisch ca. den Stand der frühen 80er Jahre repräsentieren. In Parrot hingegen sind auch Ergebnise neuerer Forschung geflossen und Allison wirft regelmäßig ein Auge darauf, daß es so bleibt. Sie wertet es teilweise auch als Reaktion auf Parrot, wenn Sun und Microsoft in jüngster Zeit damit Werbung machen, daß auf ihrer VM auch dynamische Sprachen (python, ruby, tcl etc.) laufen.

Raku-Do

Parrots Hauptaufgabe (Perl 6 auszuführen) wurde nie vernachläßigt. Dennoch war es so still um Parrots Perl 6-Parser, daß viele nicht wußten, daß Parrot seit Anfang 2006 sehr wohl etwas Perl 6 versteht. Sicher war dies bisher deutlich weniger als Pugs vermochte, aber das wird sich hoffentlich bald ändern. Dank NQP und anderer Fortschritte beschleunigte sich die Entwicklung dieses Parsers im letzten Jahr. Um das hervorzuheben, gab ihm Patrick Michaud Januar 2008 den Namen Rakudo, im Quellcode heißt er allerdings weiterhin funktional „perl6“. Rakudo ist japanisch und kann ins deutsche mit „Weg des Kamels“ oder „Himmel“ übersetzt werden. „Weg des Kamels“ ist auch sehr zutreffend, soll doch Rakudo der Standardweg werden Perl 6 zu kompilieren und auszuführen. Und das Perl 6 der Himmel für Programmierer ist, hat ja dieser Artikel schon ein Jahr vor dieser Namensgebung verkündet.

Rakudo selbst wird derzeit in drei verschiedenen Sprachen geschrieben. Zuallererst hat es mit der in „Perl 6-rules“ geschriebenen „grammar.pg“ eine stark reduzierten Version der STD.pm die von der PGE tatsächlich in einen „parse tree“ kompiliert werden kann. Zwar waren es frühe Versionen dieses Teils des Perl 6-Parsers, die zu der groben Struktur der STD.pm inspiriert haben, aber schon wie beschrieben, ist die Entwicklung der STD.pm derzeit weitestgehend unabhängig von Rakudo, auch wenn eine Vereinigung natürlich angestrebt wird. Der „Rest“ von Rakudo war einst in PIR geschrieben, das jedoch zunehmend durch NQP ersetzt wird, weil das nicht nur effektiver ist, sondern weil Quellen, die fast wie Perl aussehen, freiwillige Helfer mehr anziehen, als der assemblerartige Syntax der PIR. Die Entwicklung von Parrot und Rakudo steht auch finanziell auf gesunden Beinen, da der NLNet-Grant eine kleine finanzielle Grundsicherung bis Parrot 1.0 gewährt und Patrick nun von der Mozilla Foundation gefördert wird.

SMOP

Zudem gibt es noch interessante Neuheiten ausserhalb von Parrot. Daniel Ruoso hat unter dem eigenwilligen Namen SMOP (Simple Meta Object Programming) eine „runtime engine“ begonnen. Diese kann zwar nur ausführen, was Parser für sie aufbereiten, ist bafür aber schneller als Pugs und unkomplizierter als Parrot. Im Moment wird auf ein Zusammenwirken mit kp6 als Parser hingearbeitet, aber andere Kombinationen sind denkbar, da SMOP eine sehr flexible Architektur hat. Beweggrund für SMOP‘s Entstehung waren ja Probleme, Pugs oder Parrot als „backend“ für kp6 verwenden zu wollen. Daniel‘s Ansatzpunkt ist dabei ein Meta-Objektsystem, welches Objekte mit frei definierbarem Verhalten ausführen kann. Das ist sinnvoll, da Perl 6 intern voll objektorientiert ist. Man könnte auch scherzhaft sagen, daß SMOP geschrieben wurde um das Objektsystem zu prüfen, Parrot für die „rules“ (PGE) und Pugs um die „junctions“ zu testen (das waren tatsächlich eine von Pugs ersten Fähigkeiten). Doch im ernst, SMOP ist bereits weit mehr als ein kleiner Test. Seine Architektur enthält mehrere fortschrittliche Konzepte. Selbst wenn kp6 nicht wieder aufwacht, ist ein schneller und vollständiger Perl 6-Interpreter der auf SMOP aufbaut möglich und wäre eine interessante Alternative zu Parrot.

TIMTOI

Auf jeden Fall sollten Perl-Freunde sich schon jetzt an den Gedanken gewöhnen, daß Perl 6 mehr als einen Interpreter haben wird und daß jede Software, welche die Testsuite und damit die Spezifikation erfüllt, sich auch Perl 6 nennen darf. Das ist Larry‘s ausdrückliche Haltung. Somit entfällt die Unterscheidung der Schreibweisen Perl und perl. Auch wenn „perl“ sicher auf vielen Rechnern als symbolischer Link auf Rakudo oder anderes ein wichtiger Befehl sein kann, wird man mit Perl nur noch eine großartige Sprache, aber nicht mehr den Interpreter bezeichnen.

Herbert Breunung

Perl6 Tutorial - Teil 3: Operatoren für Arrays

Gar keine Einleitung

Herzlich willkommen zum dritten Teil dieses Perl 6-Tutorials. Wie immer sind ein Editor und Pugs dazu hilfreich, und wir setzen die Besichtigung genau da fort, wo wir letztens inne hielten: bei den Operatoren.

Arrays in Perl

Und wie angekündigt geht es in dieser Folge um Operatoren, die den Umgang mit Arrays erleichtern. Hier hat Perl 6 sehr zugelegt. Zwar zeichnet sich Perl 5 durch sehr einfach und vielseitig verwendbare Arrays aus, die zudem mit ‚map', ‚grep' und ‚for' sehr flexibel bearbeitet werden können (das gibt es natürlich auch alles weiterhin), aber besonders funktionale Sprachen wie Haskell haben in den letzten Jahren gezeigt, daß auf dem Gebiet weit mehr möglich ist. Und es war auch die durch Pugs herbeigeführte, enge Zusammenarbeit von Haskell-Programmierern und dem Design-Team, die auf diesem Gebiet deutliche Spuren im Perl 6-Syntax hinterlassen hat. Darüber hinaus hat sich ebenso grundlegendes in der Schreibweise der Arrays geändert, das damit nichts zu tun hat.

Als Perl noch klein war und auf Larrys Wickeltisch lag, gab es nur einzelne Werte und einfache Listen oder Hashes als Spielzeug. Um sie gut unterscheiden zu können, bekamen sie als Erkennungssymbol die Sigils $, @ uns %. Da aber Arrays und Hashes auch einzelne Werte abliefern können und Hashes sogar Arrays von Werten, entschied Larry, es sei besser, in diesen Fällen die Sigil zu ändern, um immer zu sehen, ob man es gerade mit einen oder mehrere Werten zu tun hat. (Hashes sind ja in Perl 5 bloß Arrays mit gerader Länge.) Mit 7 Jahren erhielt Perl jedoch zur Einschulung Referenzen. Das war gut, denn nun konnte es komplexe Datenstrukturen erzeugen, was für größere Schulaufgaben wichtig ist. Aber damit verloren die veränderlichen Sigils ihren ursprünglichen Sinn, denn „$fibo[7]" konnte nun auch (eine Referenz auf) einen Array oder einen Hash zurückgeben. Und auch das Dereferenzieren mit geschweiften Klammern wie bei `@a = @{$fibo[7]}` kann zu komplexen Ausdrücken führen, die Anfänger verwirren. Deshalb haben Variablen in Perl 6 unveränderliche Sigils. Zu deutsch: eine Arrayvariable beginnt immer mit einem '@'. Damit bleibt wenigstens der Typ der Variable sichtbar. Zusätzlich vereinfacht das Dinge wie:

Perl 5:

```
@gerade = @{$zahlen[4]};
$zahl   = $halde->[3][4][5];
```

Perl 6:

```
@gerade = @zahlen[4];
$zahl   = $halde[3;4;5];
```

Diese Beispiele zeigen: den Kontext bestimmt jetzt vor allem der Empfänger der Daten. Erhält ein Array eine Arrayref, dereferenziert Perl 6 automatisch. Weist man einem Array einen Skalar zu, hat er nur noch einen (neuen) Wert. Wird einem Skalar ein Array zugewiesen, so wird er zur Arrayref (ich mein natürlich Capture), die behandelt werden kann wie ein Array, auch wenn sie die Sigil $ hat. (Fast wie ein Array in PHP.)

In Teil 2 wurde vorgestellt, daß ebenso Operatoren den Kontext bestimmen können. Auch hier erzwingt „~" den Stringkontext, was der Aneinanderreihung der Inhalte aller Elemente entspricht (selbstverständlich durch Leerzeichen getrennt). Und das „+" fordert den numerischen Kontext, in dem ein Array die Anzahl seiner Elemente liefert, in gleicher Weise wie Perl 5-Arrays im skalaren Kontext. Dafür gibt es noch einen objektorientierten, expliziteren Syntax:

Perl 5:
```
scalar @gerade;
scalar @gerade;
$#rray;
```

Perl 6:
```
+@gerade;       # Elementenanzahl
@gerade.elems; # geht auch
@rray.end;     # letzter Index
```

Doch auch für das „ ~@a“ wurde noch eine Schreibweise geschaffen, da innerhalb von doppelten Anführungsstrichen Arrays und Hashes nicht automatisch expandieren. Dies soll eine einfache Ausgabe von Mailadressen und Prozentangaben ermöglichen.

Perl 6:
```
say @namen;
say "meine Adresse: ich@namen.de";
```

Möchte man die Ausgabe der oberen Zeile innerhalb einer Angabe, geht das nun auch so:

Perl 6:
```
say "Die taoistischen Unsterblichen sind:
   @namen[]";
```

Auch die Ausgabe einzelner Arraywerte kann Umsteiger anfangs ratlos machen, denn z.B. das vorletzte Element fordert man jetzt mit „@a[*-2]“ an. Der Stern bedeutet jetzt in fast jedem Kontext so etwas wie „alles“, „unendlich“, „irgendetwas“ oder „Grenzwert“. Damit möchte ich aber die Einweisung in Perl 6-Arrays beenden, da eigentlich die Operatoren das aktuelle Thema sind.

Füttern der Arrays

Das einfache füllen eines Arrays geht wie gewohnt:

Perl 6:
```
my @noten = (
 'Do', 'Re', 'Mi', 'Fa', 'Sol', 'La', 'Si'
);
```

Die runden Klammern können weggelassen werden, da das Komma jetzt arrayerzeugender Operator ist. Sonst würde im nächsten Beispiel gerade mal „Si“ in „@noten[0]“ landen.

Perl 6:
```
my @noten = 'Re', 'Mi', 'Fa', 'Sol', 'La',
    'Do', 'Si';
```

Aber so würde es wohl auch kein geübter Perlautor schreiben, denn es gibt ja qw(). Gibt es das noch? So etwas praktisches und nützliches wurde nicht nur behalten, sondern auch in der Schreibweise vereinfacht.

Perl 6:
```
my @noten = <Re Mi Fa Sol La Do Si>;
my @tiere = << $baer $hase $igel >>;
```

Im vorigen Teil hatte ich ja schon erwähnt das es von „<>“ noch eine Version gibt, die interpoliert. Die Namensgebung lehnt an ‚ und „ an, wo auch die „doppelte“ Version interpoliert (Variablennamen mit ihren Inhalt ersetzt). Diese Schreibweise ist auch praktisch, wenn dem Array Werte eines Hashes zugewiesen werden.

Perl 6:
```
my @ausgaben = %rechnung< alfred hein peter >;
my @ausgaben = %rechnung<< @namen[] >>;
```

Hashschlüssel wurden zwar vorher auch oft nicht gequotet, aber sowas tun nur Rüpel, da Barewords bekanntermaßen böse und hinterhältig sind und liebe Programmierer immer strict verwenden. In Perl 6 hat man hier keine Wahl, da strictness default ist, und es keine Bareword-Stringliterale mehr gibt.

Auch die altbekannten Rangeoperatoren kann man immer noch zum Befüllen von Arrays verwenden:

Perl 6:
```
my @ziffern = 0 .. 9;  # 0,1,2,3,4,5,6,7,8,9
my @buchst  ='a'..'z'; # <a b c d e f ... y z>
```

Ersteres kann man aber jetzt noch kürzer schreiben:

Perl 6:
```
my @einer = ^10;        # 0,1,2,3,4,5,6,7,8,9
```

Ja genau, das Dach schließt das ihm Folgende als Grenzwert aus, und die 0 wird standardmäßig als untere Schranke angenommen. Dieses Konstrukt hat vor allem zwei praktische Anwendungen. Im Skalarkontext kann man damit Bereiche definieren, deren Schranken nicht zur Menge gehören, aber auch einfache for-Schleifen können so noch um ein paar Anschläge kürzer geschrieben werden:

Perl 6:
```
3 ~~ 3^.. 7;          # gibt Bool::False
for ^5 { ... }        # iteriert 5 mal
```

Und da wir grad bei Schleifen sind: Wie oft wollten Perl 5-Programmierer die Iteratorvariable in Sprüngen oder rückwärts zählen lassen. Dies geht nun alles sehr einfach:

Perl 6:
```
for  ^10:by( 2) {...}; # 0,2,4,6,8
for 4..1:by(-1) {...}; # 4,3,2,1
```

Selbstverständlich ist „by“ kein Spezialbefehl für Schleifen, sondern kann immer für Bereiche (Ranges) eingesetzt werden. Wem es überhaupt nicht gefällt, kann im zweiten Beispiel auch auf reverse zurückgreifen, denn ‚4..1‘ evaluiert zu einer leeren Menge oder einem leeren Zahlenbereich.

Perl 6:
```
for reverse 1..4 {...}; # 4,3,2,1
```

Es lebe die Faulheit

Die Menge die ein Perl 6-Array speichert, kann aber nicht nur leer sein, sondern auch unendlich groß. Dafür braucht es nicht einmal unendlich viel RAM. Es reicht vollkommen aus, ihn wie folgt als unendlich zu bestimmen:

Perl 6:
```
@zahlen = 5 .. Inf;   # von 5 bis Unendlich
@zahlen = 5 .. *;     # das Selbe
@zahlen = * .. *;     # aka -Inf .. +Inf
```

Inf (gibt es in den Varianten +Inf und -Inf) oder * steht hier für Unendlich, und es ist auch ein mathematisch korrektes Inf, das Perl 6 zurückgibt, wenn man eine Division duch 0 versucht. Aber wie funktionieren die letztgenannten Beispiele? Perl 5 hätte hier unsanft angemeldet „Range iterator outside integer range at ...“, Perl 6 expandiert jedoch den Array erst, wenn er tatsächlich gebraucht wird, und dann auch nur bis zu dem benötigten Element. Diese „lazy evaluation“ gehört zu den Einflüssen funktionaler Sprachen wie Haskell in Perl 6, die ich zu Beginn ansprach. Sie bewirkt, daß eine „for“-Schleife, die über „@zahlen“ iteriert, so lange läuft, bis ein Sprungbefehl das Verlassen der Schleife erzwingt oder das Betriebsystem aufhört zu arbeiten. Doch im Gegensatz zu Haskell gilt für Perl immer noch TIMTOWTDI. Wer seine Listen sofort expandiert haben möchte, kann dies mit „eager“ erzwingen.

Perl 6:
```
# jetzt hängt das programm wirklich
@zahlen = eager 5..*;
$zahl   = lazy  "$text:" ~ berechne();
```

Im Gegenzug ist es auch möglich, mit „lazy“ die „lazy evaluation“ zu erzwingen, wo Perl per default „eager“ evaluiert, wie bei einer Zuweisung in den Skalarkontext. Es empfiehlt sich jedoch dabei aufzupassen, da die Variablen oder Routinen in dem Beispiel zu einem späteren Zeitpunkt andere Werte liefern können. Überhaupt erfordert die „lazy evaluation“ etwas mehr Aufmerksamkeit, um genau zu erkennen, wann etwas ausgeführt wird. Das folgende Beispiel filtert die numerisch geraden Zahlen aus einem Array und überweist sie in einen zweiten Array. Der verwendete feed-Operator dient der Weiterreichung von Arrayelementen, um etwa Schwartz‘sche Transformationen und ähnliches explizit zu formulieren. Neu daran ist, daß die Werte einzeln „@out“ verlassen um sogleich das ganze Fließband an Befehlen zu passieren und im „@in“ zu verschwinden. Im dritten Beispiel würde die gerade Zahl also zuerst halbiert und in „@in“ gespeichert, bevor der nächste Wert aus „@out“ entnommen wird. Diese häppchenweise Arbeitsweise gab den feed-ops ihren Namen. Das vierte Beispiel demonstriert, daß sie von links nach rechts zuweisen können und daß sich die Taktstraße auch aus mehreren Quellen speisen kann. Wegen der Schreibweise „==>>“ folgt dem letzten Element aus „@out1“ das erste Element aus „@out2“.

Perl 6:
```
@in = grep { $_%2 } <== @out;
# funktionsgleich
@in = grep { $_%2 }, @out;
@in = map {$_ / 2} <== grep { $_%2 }
                   <== @out;
@out1 ==>> @out2 ==>> grep { /\s/ } ==> @in;
```

Selbstverständlich unterbricht ein „eager“ oder „sort“ die „lazy evaluation“ in solchen Befehlsfolgen.

Befehle wie push, pop, shift und splice, die auch dafür geeignet sind, Arrays zu füttern, werden bereits genügend in der perldoc beschrieben. Bis auf Nuancen hat sich für diese Befehle nichts geändert, weswegen sie hier nicht erklärt werden.

Die Raubtierfütterung geht weiter

Wenn einem gerade nichts einfällt, kann ein Array natürlich auch mit Wiederholungen eines Wertes gefüllt werden:

Perl 6:
```
@affen = "gibbon" xx 5;
# mir ist doch noch was eingefallen
@affen = <gibbon makake> xx 3;
```

Anders als in Perl 5.10 unterscheidet „x“ hier keinen Kontext. Es wiederholt, wie in der vorigen Folge gezeigt, immer einen String und erzeugt einen meist längeren String. „xx“ wiederholt immer eine Liste, selbst wenn sie nur ein Element hat. Die Ergebnisse werden dann auch, wie anfangs beschrieben, an den Kontext des Empfängers angepasst. Was Perl 5 leider auch nicht kennt, ist das große „X“, der Kreuzoperator. Ohne Schleifen und Zusatzmodule können damit alle Kombinationen der Elemente mehrerer Arrays erzeugt werden. Wichtigste Regel ist hier dabei, daß der linke Array Vorrang hat, also zuerst alle Paare mit dem ersten linken Element gebildet werden, dann mit dem zweiten usw.

Perl 6:
```
1,2 X 3,4       # (1,3),(1,4),(2,3),(2,4)
```

Wie das Resultat aufgelöst wird, hängt wieder vom Kontext ab, wobei es an der Zeit ist, einen neuen, interessanten Kontext vorzustellen. Der „slice“-Kontext, erkennbar an seiner Sigil „@@“, verweist auf Arrays, die wiederum Arrayreferenzen enthalten. Slice bedeutet soviel wie Arraystück oder Scheibe. Der einfache Arraykontext würde einen flachen Array erzeugen. Oft möchte man das gerade nicht.

Perl 6:
```
# [\(1,3),\(1,4),\(2,3),\(2,4)]
$ 1,2 X 3,4
# 1,3,1,4,2,3,2,4
@ 1,2 X 3,4
# [1,3],[1,4],[2,3],[2,4]
@@ 1,2 X 3,4
```

Genauso kontextsensitiv ist auch der „zip“ Operator, der ebenfalls aus bestehenden Arrays neue zusammenstellen kann. Er wird „zip“ oder „Z“ geschrieben, und aus der zweiten Schreibweise läßt sich auch sein Vorgehen ableiten. (Man nehme zuerst ein Element aus dem linken Array, dann das erste aus dem rechten. Dann wieder zum ersten und zurück ...). Zip heißt zu deutsch Reißverschluss, was den Mechanismus auch gut beschreibt.

Perl 6:
```
 $ 1,2 Z 3,4    # [\(1,3),\(2,4)]
 @ 1,2 Z 3,4    # 1,3,2,4
@@ 1,2 Z 3,4    # [1,3],[2,4]
 % 1,2 Z 3,4    # { 1 => 3, 2 => 4 }
```

Besonders für die parallele Verarbeitung mehrerer Arrays in einer Schleife ist das sehr praktisch. Dies zeigt allerdings die nächste Folge des Tutorials, weil es noch Wissen zu den inneren Verhaltensweisen anonymer Blöcke verlangt, was weit außerhalb des heutigen Themas liegt.

Sowohl der Kreuz- als auch der Reißverschlussoperator kann mehrfach verkettet werden, die Ergebnisse werden lediglich entsprechend komplexer.

Perl 6:
```
1,2 X 3,4 X 5,6 # 1,3,5),(1,3,6) ...
```

Giga, Hyper, Meta ...

Mit dem großen Kreuz lässt sich sogar noch weit mehr anstellen, wenn es circumfix (umschließend wie Klammern) um einen anderen Operator gesetzt wird. Dies eröffnet das völlig neue Feld der Metaoperatoren für Arrays. Operatoren für Skalare können somit auf Listen angewendet werden, und für die Fülle der neuen Möglichkeiten muß niemand lange Listen an neue Befehlen lernen. Es reicht völlig, die Arbeitsweise der drei Metaoperatoren zu verstehen.

Der Kreuz-Metaoperator funktioniert auch fast wie der einfache Kreuzoperator. Auf jedes Kombinationspaar wird lediglich der innere Operator angewendet. Am Beispiel ist das leicht nachvollziehbar:

Perl 6:
```
<a b>   X~X <1 2>   # <a1 a2 b1 b2>
<1 2 3> X*X <2 3>   # <2 3 4 6 6 9>
```

Der praktische Nutzen von solchem Code ist nicht nur, daß er mindestens 5 Zeilen Perl 5 ersetzt, sondern daß er dem Interpreter auch erlaubt, die Arbeitsschritte auf mehrere Prozessoren(kerne) zu verteilen, da es auch nicht darauf ankommt, in welcher Reihenfolge die Werte berechnet werden,

solang sie in der erwarteten Reihenfolge ankommen. Genau diese Parallelisierung der Datenverarbeitung erlaubt auch der nächste Metaop: der Hyperoperator. Er besteht aus zwei „>“ oder „<“, kann aber auch mit den UTF-Symbolen namens Chevron geschrieben werden, welche genauso aussehen. Wie zu erwarten, war das Gegenstand kräftiger Diskussionen, während der Larry Wall seinen Standpunkt verteidigte: „UTF ist schon lange Standard und man solle die Verwendung von UTF wenigstens ermöglichen. Alle „UTF-Operatoren“ sind ohnehin optional, und Perl 6-Quellcode wird intern sowieso immer als Unicode angesehen.“

Der klassische Anwendungsfall für Hyperoperatoren besteht darin, die ersten Werte zweier Arrays als Operanden zu verwenden und das Resultat als ersten Wert im Ergebnisarray zu speichern. Nächste Operanden sind die jeweils zweiten Werte usw. Sind beide Arrays unterschiedlich lang, wird mit leeren Werten aufgefüllt.

Perl 6:
```
@reste = <5 3 7> >>%<< <2 7 5>; # <1 3 2>
@summen = <2 3 4> >>+<< 1;      # <3 3 4>
```

Möchte man einen einzelnen Wert auf eine Liste anwenden, so reicht es, die Richtung des Hyperoperators umzukehren.

Perl 6:
```
<5 3 7> >>+>> 1;  # <6 4 8>
```

Das breite Ende der Pfeile zeigt an, welche Seite die Dimension des Ergebnisses bestimmt. Hätte das letzte Beispiel Variablen benutzt, ließe es sich aber noch elegant verkürzen.

Perl 6:
```
@a = @a >>+>> 1;
@a >>+=>> 1;      # dito
@a >>++;          # noch kürzer
```

Sehr nützlich ist auch, daß Hyperoperatoren sehr gut mit mehrdimensionalen Arrays umgehen können.

Perl 6:
```
# [[-1, -2], -3]
-« [[1, 2], 3]
# [[5, 7], 9]
[[1, 2], 3] »+» [4, 5, 6]
[[1, 2], 3] «+» [4, [5, 6]]
# == [[1,2] «+» 4, 3 «+» [5, 6]]
# == [[5, 6], [8, 9]]
```

Im zweiten Beispiel bestimmt die linke Seite, wegen der Richtung der Operatoren, die Dimensionalität. Beim dritten ist es keine Seite, und so wird auch jedes Slice als Element angesehen und entsprechend „ausmultipliziert“.

Die dritte Klasse an Metaoperatoren sind die Reduktionsoperatoren. Ihr Name ist redlich verdient, da sie die Dimension ihres Operanden reduzieren, oder zu deutsch: Sie machen aus einem Array einen Skalar, indem sie alle Arrayelemente miteinander anwenden. Reduktionsoperatoren werden in eckigen Klammern geschrieben.

Perl 6:
```
[+] 1..4      # 10 = 1+2+3+4
[~] 'a'..'f' # 'abcdef'
```

Trickreich und effektiv lassen sie sich auch mit Vergleichs- und Auswahloperatoren kombinieren.

Perl 6:
```
# wahr wenn sortiert
$sorted    = [<] @a
# erster nicht leerer Wert im Array
$wert      = [||] @a
# @zahlen.min ginge auch
$min       = [min] @zahlen
$min, $max = [minmax] @zahlen
```

Aber auch Reihen, wie man sie beim Mathe-Wettbewerb „Euler“ häufig braucht, lassen sich mit einer weiteren Schreibweise der Reduktionsoperatoren sehr knapp formulieren.

Perl 6:
```
[\*] 1..3   # 1, 2 = 1*2, 6 = 2*3
[\+] ^10    # 0,1,3,6,10,15,21,28,36,45
```

Diese Form liefert die Aufzählung aller Zwischenergebnisse eines Reduktionsops. Formal hieße sich das ungefähr als „@ergebnis[$n] = @ergebnis[$n-1] op @input[$n]“ ausdrücken. Und auch an dieser Schreibweise erkennt man die Regel einer Huffman-Kodierung (Je seltener etwas benötigt wird, desto länger (seltsamer) ist sein Name).

Und noch ein Blick in die Quantenwelt

Neben Larry Wall ist Damian Conway einer der Hauptgestalter der Perl 6-Syntax. Nicht nur die ableitbaren Grammatiken der „rules“ basieren zu einem guten Teil auf seinem „Parse::

RecDescent“, auch sein Modul „Quantum::Superpositions“ ließ viele Perl-Programmierer erste Bekanntschaft mit Junctions schließen. Wie vieles, wofür sich Damian begeistert, können auch Junctions als abgehoben und schwer zugänglich erscheinen. In funktionalen Sprachen wie Haskell erwiesen sie sich jedoch als hilfreich, um komplexe logische Verknüpfungen einfacher zu formulieren. Würde man in Perl 5 z.B. noch schreiben:

Perl 5:
```
if ($farbe eq "rot" or $farbe eq "blau"
        or $farbe eq "grün") {
    print "Sie wählten eine Grundfarbe,
        interessant.\n"
}
```

So geht das in Perl 6 auch junktiv.

Perl 6:
```
if $farbe eq "rot" | "blau" | "grün" {
  say "Sie wählten eine Grundfarbe,
     interessant."
}
```

Aus Perl 5 ist bekannt, daß „|“ für ein logisches „oder“(or) steht, „&“ für „und“ (and) und „^“ für „entweder oder“ (xor) steht. Neu ist, daß man den Sachverhalt: „rot oder blau oder grün“ in einer Variable speichern kann. Das letzte Beispiel ginge damit so:

Perl 6:
```
my $gfarben = "rot" | "blau" | "grün";
if $farbe eq $gfarben ...
```

Wem das nicht ganz geheuer kann auch einen vertrauten Array verwenden und dessen Elemente junktiv verknüpfen.

```
my @gfarben = <rot blau grün>;
if $farbe eq any(@gfarben) ...
```

Sicher ließe sich anstatt „any(@gfarben)“ auch die reduktive Variante „[|] @gfarben“ verwenden, allerdings ist ersteres weit eingängiger (und aus Quantum::Superpositions bekannt). In gleicher Weise entspricht das „all“ einem „[&]“ (and), „one“ dem „[^]“ (xor) und „none“ einem „not“.

Perl 6:
```
if all(@tiere) ~~ all(@affen)
    { say "nur affen hier ...
if all(@tiere) ~~ all(@affen) |  $faultier
    { ...
if schau() | vergleich() eq any(@gfarben)
    { ...
```

Selbstverständlich lassen sich Junctions auch schachteln. Mit ihnen kann man auch Operationen auf mehrere Werte parallel anwenden, ähnlich den Hyperoperatoren.

Perl 6:
```
my $gfarben = "rot" | "blau" | "grün";
# "rote" | "blaue" | "grüne";
$gfarben ~= 'e';
# (4|5) & (5|6)
(1|2) + (3&4);
```

Am schönsten daran ist wohl, daß man so komplexe logische Regelwerke kombinieren kann und Perl alle Details überläßt.

Danke

Über Arrays in Perl 6 lassen sich noch viele weitere Neuerungen berichten. Dieses mal ging es vor allem um die damit zusammenhängenden Operatoren. In der nächsten Folge lautet das Thema Kontrollstrukturen. Es wird daran anknüpfen, da Schleifen oft für die Bearbeitung von Arrays und Hashes eingesetzt werden.

Herbert Breunung

TWiki - eines der führenden Open-Source-Wikis für Unternehmen

Mehr als zwei Millionen User weltweit arbeiten mit dem in Perl programmierten Wiki-System TWiki. Damit ist das systemorientierte TWiki neben dem artikelzentrierten MediaWiki das erfolgreichste und zudem das wohl ausgereifteste Open-Source-Wiki für den Einsatz im Unternehmens-Intranet. Dennoch fristet TWiki in den größeren Medien immer noch ein Schattendasein und viele Manager haben das Potenzial dieser mächtigen Applikation noch nicht erkannt.

Firmen-Wiki ist kein Web-Lexikon

Die Software von Wikipedia ist MediaWiki und das ist das Killer-Argument für diese Applikation. Manager sehen den Erfolg des Web-Lexikons und schlussfolgern: Wenn Wikipedia so erfolgreich ist, müssen wir nur diese Software verwenden, um ebenfalls Erfolg zu haben. Aber das ist Denkfehler. MediaWiki wurde exklusiv für Wikipedia entwickelt und ist auf dessen Bedürfnisse ausgerichtet. Die Anforderungen einer Web-Enzyklopädie sind artikelzentriert.

Ein Unternehmen benötigt kein Lexikon, sondern eine Ergänzung zum Intranet, die auf einem Wiki basiert. Ein Wiki sollte speziell auf die individuellen Bedürfnisse eines Unternehmens abgestimmt sein.

Dennoch sind auch TWiki-Dokumente sehr einfach veränderbar. Wer bereits einmal einen Wikipedia-Artikel modifiziert hat, ist problemlos in der Lage, mit TWiki zu arbeiten. Ein langer Lernprozess ist nicht nötig.

Ein weiteres Argument für MediaWiki ist dessen Vandalismus-Sicherheit. In der Tat können Änderungen rasch rückgängig gemacht und verschiedene Versionen von Dokumenten komfortabel miteinander verglichen werden. Doch die Vandalismus-Bedrohung ist ein Scheinargument. Schutzmechanismen mögen für ein Web-Lexikon sinnvoll sein, in einem Unternehmens-Wiki sind sie schlicht überflüssig.

Bei jeder Änderung wird die IP des Rechners gespeichert, von dem aus die Modifikation erfolgt. Die meisten Mitarbeiter sind zudem am Intranet angemeldet. Jede IP kann von der IT-Abteilung personalisiert und jede Änderung zurückverfolgt werden. Deshalb wird es keinen Mitarbeiter geben, der böswillig Dokumente manipuliert. Die Vandalismus-Frage hat im Firmen-Wiki keine Relevanz und faktisch tritt Vandalismus nicht auf.

Versehentliche oder fehlerhafte Änderungen sind aber auch in TWiki problemlos rückgängig zu machen: TWiki versioniert automatisch alle Informationen und Dokumente und hält jede Modifikation fest. Frühere Versionen sind also jederzeit wieder herstellbar.

Macht durch eine starke Community

Die Stärke von TWiki liegt in der Quantität und vor allem der Qualität seiner Plug-ins. Derzeit gibt es für TWiki mehr als 200 Extensions, die von Unternehmen für Unternehmen entwickelt wurden.

Sicher existieren auch für MediaWiki mehrere Erweiterungen. Diese haben aber längst nicht die Güte und Ausgereiftheit, die für den professionellen Einsatz im Unternehmen vonnöten sind – einige Plug-ins funktionieren erst gar nicht richtig und verlässlich.

Der TWiki-Background ist ungleich mächtiger. Viele Unternehmen wie zum Beispiel Yahoo, Motorola und andere Größen beteiligen sich aktiv an der Weiterentwicklung von TWiki, um der Community etwas zurückzugeben. Daraus re-

sultiert ein enormes Maß an Ausgereiftheit aller verfügbaren Extensions, mit denen zahlreiche Aufgaben abgebildet werden können. Gerade durch diese Offenheit und zusätzliche Funktionalität eignet sich TWiki für den Einsatz im Intranet und steigert dessen Effizienz und Effektivität signifikant.

Besonderheiten von TWiki

Das hervorragend dokumentierte TWiki, dessen parsing engine in Perl programmiert ist, hat einen schlanken Kern, der ohne eigene Datenbank und ohne eigenes Dateiformat auskommt. Zusatzfunktionen werden je nach den individuellen Bedürfnissen nachträglich installiert, wobei die Einbindung von Erweiterungen die Stabilität des Kerns nicht beeinträchtigt.

Die Nutzerverwaltung des Systems ist zu großen Teilen automatisiert und gestattet auch die unkomplizierte Einbindung externer User (z.B. via LDAP).

Alle Daten sind in Webs, Topics und Attachements organisiert. Dabei fungiert ein Web als Themengruppe, in die beliebig viele Topics eingestellt werden können. Bei der Einrichtung legt das System standardmäßig das Main Web, das TWiki Web und die Sandbox, in der sich Einsteiger ausprobieren können, an.

Die Topics sind die einzelnen Dokumente eines Webs. Neben von Mitarbeitern anzulegenden Topics gibt es eine Reihe von Standard-Topics wie zum Beispiel TextFormattingRules (mit Informationen über Formatierungsmöglichkeiten) oder WebChanges (mit Informationen über Topic-Änderungen im Web).

Topics können hierarchisch angeordnet und anderen Topics unter- oder übergeordnet werden. Das ist von wesentlicher Bedeutung für die strukturierte Ablage von Informationen. Ein Topic selbst besteht aus dem Text an sich, den Revisionsdaten, zusätzlichen Metainformationen, in denen Autor und Zeitpunkt der letzten Änderung enthalten sind, und den optionalen Anhängen. Dateianhänge dürfen bis zu 10 MB groß sein, wobei Grafiken, Bilder, Tabellen etc. problemlos auch in den Text selbst einzufügen sind. Anhänge oder Topics werden nicht gelöscht, sondern in den Trash verschoben und sind per Klick wieder herstellbar.

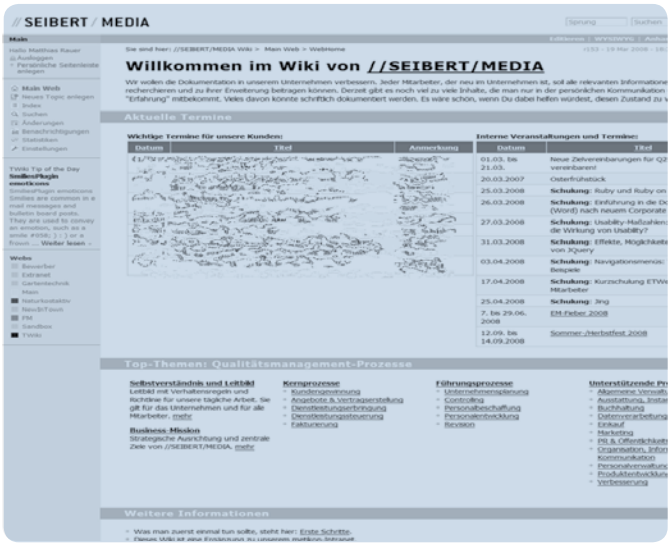
Einsatz in der Praxis

Generell muss die Etablierung eines Wikis durch eine Reihe von Maßnahmen unterstützt werden. Greifen diese Aktivitäten, wird das System, wie ich aus eigener Erfahrung weiß, schnell von Mitarbeitern akzeptiert und angenommen. In unserem Unternehmen //SEIBERT/MEDIA GmbH wird TWiki seit Juni 2007 eingesetzt und umfasst inzwischen acht Webs. Allein das Main Web ist in wenigen Monaten auf fast 1.000 Topics angewachsen.

Wertvoll erweist sich TWiki insbesondere bei der Abbildung von Qualitätsmanagement-Prozessen, die in unserem Intranet-Wiki viel Raum einnehmen, und für die Dokumentation von Projekten in kleinen bis mittleren Arbeitsgruppen, denn es richtet seinen Fokus vor allem auf die Zusammenarbeit. Mitarbeiter nutzen das System aktiv, um gemeinsam interne Prozesse abzubilden, um Neuerungen und Veränderungen in Unternehmensabläufen voranzutreiben und zu diskutieren oder auch um interne Veranstaltungen zu organisieren. Beispielhaft für die aktive Kommunikation und Sammlung von Wissen sind die Dokumente, in denen Learned Lessons und Ergebnisse aus Kundenprojekten, Meetings und internen Wissenszirkeln festgehalten und so allen Kollegen zugänglich gemacht werden.

TWiki erleichtert die Ablage, Bearbeitung und Erweiterung von Vorlagen und Anleitungen. Das ist bekanntlich in statischen Intranets, die nicht selten unter dem ‚One-Administrator-Syndrome' leiden und deshalb nur schleppend wachsen, kaum möglich.

Gewiss birgt ein Wiki immer die Gefahr von Redundanzen. Deshalb ist es wichtig, Inhalte strukturiert und hierarchisch



anzulegen, was durch TWiki leichter gemacht wird. Die Abstellung eines erfahrenen Mitarbeiters, der Administrator gewissermaßen für die strukturelle Pflege zuständig ist, hat sich darüber hinaus als sinnvoll erwiesen.

Bei professioneller und zielgerichteter Handhabung werden mit TWiki die Worthülse „Wissensmanagement“ auch tatsächlich mit Inhalten gefüllt und so nach und nach weite Teile des Unternehmens-Know-hows abgebildet. In der praktischen Anwendung wird deutlich, wodurch sich TWiki als strukturiertes Wiki auszeichnet: Es verbindet die Vorteile eines Wikis mit denen einer Datenbank-Applikation, ohne selbst eine solche zu benötigen. Die relative Offenheit ermöglicht die strukturelle Anpassbarkeit an individuelle Unternehmensbedürfnisse. Das gewährleistet zum einen die problemlose Übertragung von unstrukturiertem Content in strukturierte Inhalte (und umgekehrt), zum anderen eignet sich TWiki dadurch ausgezeichnet zur Entwicklung weiterer Wiki-Applikationen.

Einige Erweiterungen

Wie erwähnt ist TWiki nicht zuletzt dank seiner Erweiterbarkeit und der hohen Ausgereiftheit seiner Plug-ins und Extensions anderen Systemen voraus. Die folgenden beispielhaft angeführten Module verdeutlichen schon die Leistungsfähigkeit und die zusätzliche Funktionalität von TWiki.

Das Workflow-Plug-in macht den aktuellen Status eines jeden Topics kontrollierbar und nachvollziehbar und dokumentiert alle Arbeitsschritte und Veränderungen in einem Topic.
Das Table-Plug-in ermöglicht die mühelose Erstellung und Integration (editierbarer) Tabellen in Dokumente, die mit dem Chart-Plug-in in PNG- oder GIF-Grafiken automatisch in umgewandelt werden. In Verbindung mit dem Spreadsheet-Plug-in wird TWiki so um die wichtigen Grundfunktionen einer Tabellenkalkulation bereichert.

Das WebStatistics-Plug-in erweitert TWiki um eine wertvolle Statistik-Funktion und macht ersichtlich, wie häufig welche Topics aufgerufen werden und welche Mitarbeiter wie oft Änderungen im System vornehmen.

Das Notification-Plug-in eröffnet die Möglichkeit, Mitarbeiter automatisch per E-Mail zu informieren, wenn für sie relevante Topics modifiziert wurden. Das erhöht zum einen die Reaktionsgeschwindigkeit und erspart zum anderen die manuelle Suche nach Änderungen.

Dank des TagMe-Plug-ins können Topics mit Tags gekennzeichnet und so Inhalte über Tagging Words auffindbar gemacht werden.

Erwähnt sein sollte auch das Smilies-Plug-In: Das ist für die Dokumentation von Prozessen und Wissen nun nicht unbedingt relevant, aber es bringt (bei aller Professionalität) ein gewisses Maß an Emotionalität und Lockerheit in das System mit ein, was sich wieder auf die Nutzungshäufigkeit auswirkt.

Das sind nur wenige von mehr als 200 ausgereiften Zusatzmodulen. Das systemorientierte TWiki ist in hohem Maße an individuelle Belange anpassbar und deshalb das Wiki-System, das sich meiner Meinung und auch meiner Erfahrung nach am besten für den Einsatz im Unternehmens-Intranet eignet.

Weitere Informationen finden sich in unserem Weblog und auf der offiziellen TWiki-Website:
• http://www.twiki.org
• http://blog.seibert-media.net/2007/11/arbeitstechniken/warum-in-jedes-intranet-ein-wiki-integriert-sein-sollte
• http://blog.seibert-media.net/2008/01/arbeitstechniken/wikis-sind-der-kitt-der-intranets-zusammenhaelt
• http://blog.seibert-media.net/share/2008-03-30_1851.png

Martin Seibert

Warum man von MediaWiki auf TWiki umsteigen sollte...

Für alle, die in ihrem Unternehmen mit einem MediaWiki arbeiten müssen, hier ein paar Argumente für Twiki:
• Mit MediaWiki kann man die Rechte nicht vernünftig einstellen („Spaces“).
• MediaWiki hat keine vergleichbaren und brauchbaren PlugIns. (z.B. für grafische Auswertungen im Controlling)
• Bei MediaWiki kann man keine Dateien anhängen.
• TWiki hat personalisierbare Seitenleisten.
• TWiki hat WYSIWYG schon nativ integriert.
• Man kann von MediaWiki problemlos mit einem Converter auf TWiki umsteigen.
• MediaWiki ist für Wikipedia. TWiki ist für Unternehmen.

// SEIBERT / MEDIA

BESSERE ATMOSPHÄRE? MEHR FREIRAUM?

Wir suchen erfahrene Perl-Programmierer/innen (Vollzeit)

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

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

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

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

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

SANE Backend für Canon USB Scanner

Kennen Sie das? Sie gehen frohen Mutes in ein Geschäft, erwerben nach langem Überlegen ein neues Gerät für Ihren Computer und zu Hause stellt sich heraus, dass die Hardware nicht mit Ihrem Betriebssystem läuft. Dem Autor dieses Artikels ist es leider passiert. Der folgende Artikel beschreibt nun, wie *Perl* dabei geholfen hat, ein `SANE` Backend zu realisieren [1].

Der Flachbettscanner

Meine Neuerwerbung war der USB Flachbettscanner *CanoScan LiDE 600F*. Dieser ist kompatibel zum Modell *CanoScan LiDE 600*, besitzt aber noch einen steckbaren Aufsatz zum Scannen von Negativen. Die Hardware ist sehr gut, nur an der Software für Linux hapert es. Aber hier kann man als Programmierer nachbessern. Man erinnere sich dabei an das wohlbekannte Zitat aus Wilhelm Tell: „Die Axt im Haus erspart den Zimmermann“.

Das SANE Projekt

Um einen Scanner unter Linux in Betrieb zu nehmen, kommt man um das *SANE* Projekt [2] nicht herum. *SANE* steht für „Scanner Access Now Easy“ und stellt eine einfache und vor allem komfortable API zwischen Scanner und Anwendung dar. Dabei beschränkt sich *SANE* nicht nur auf Scanner, sondern erlaubt es auch, auf Videokameras oder Webcams zuzugreifen.

Auf den Webseiten fand sich dann auch der Hinweis, dass der Scanner *CanoScan LiDE 600F* nicht unterstützt wird. Dass sich noch niemand der Hardware angenommen hat, liegt wohl an dem in diesem Scanner eingesetzten Chip (Philips CP2155BE). Nachfragen in China bzw. Taiwan bei verschiedenen Herstellern lieferte das Ergebnis, dass kein Datenblatt erhältlich ist, weil es sich um einen „custom chip“ handelt.

Keine Hilfe vom Hersteller

Man sollte meinen, dass man vom Hersteller eines Produktes zumindest ein klein wenig Hilfe bekommen könnte. Leider musste ich feststellen, dass die Firma *Canon* überhaupt nicht gewillt ist irgendeine Art von Hilfe oder Dokumentation anzubieten, die es ermöglichen könnte, einen Linux-Treiber zu schreiben.

Am Telefon sagte mir der Mitarbeiter vom Support, dass dies so in Japan beschlossen wurde und es keine Ausnahmen gäbe. Diese Unternehmenspolitik ist für mich nicht nachvollziehbar. Zumal der Linux-Treiber doch kostenfrei erstellt wird und der Hardwarehersteller dadurch problemlos einen weiteren Kundenkreis erschließen könnte.

Vielleicht scheuen sich einige Hersteller davor, den Quellcode der Treiber und der Software freizugeben, weil sie glauben das Risiko, einer abstrusen Patentverletzung bezichtigt zu werden sei dann höher.

USB Daten aufzeichnen

Somit blieb nur noch der Weg selbst „in die Untiefen der Hardware hinabzusteigen“.

Da dieser Scanner ein USB Interface besitzt, bot es sich an ein paar spezifische Scans durchzuführen und die dabei übertragenen Daten mit einem sogenannten USB Sniffer aufzuzeich-

nen. Mit etwas Glück ist man dann in der Lage, das Protokoll zu entschlüsseln. Um die Log-Dateien aufzuzeichnen, verwendete ich *Win XP* und *Benoit's USB Sniffer* [3].

Es ist sehr verwunderlich, wie groß solche Log-Dateien werden können. Allein das Einstöpseln des Scanners dokumentierte das Log mit 7 MB. Der Scan einer Briefmarke in Farbe schlug mit 83 MB zu Buche. Klar, dass diese Dateien erst noch mit *Perl* geparst und verkleinert werden mussten. Ich erspare mir daher den Abdruck dieser Daten und zeige den Inhalt der ersten Zeilen einer geparsten Fassung. Siehe Listing 1.

Da ein USB Host bei einem neuen USB-Gerät nur weiß, dass soeben ein Gerät angeschlossen wurde, aber nicht welche Eigenschaften es besitzt, muss er diese Informationen zuerst abfragen. Dies geschieht mit dem Kommando GET_DESCRIPTOR. Danach wird meist eine bestimmte Konfiguration des Gerätes mit dem Kommando SELECT_CONFIGURATION ausgewählt. Da es für die Dekodierung der Scannerbefehle unrelevant war, habe ich diese Sequenzen in der geparsten Form entfernt und durch Kommentarzeilen ersetzt.

Kommen wir nun zu den Kommandos, die Daten übertragen. Diese werden mit dem Wort BULK (bulk transfer) gekennzeichnet. Mit > und < wird die Übertragungsrichtung zum und vom Scanner angegeben. Dahinter steht in hexadezimaler Schreibweise die Adresse des USB Endpoints. Es folgt die fortlaufende Nummer des URB und der Zeitpunkt seit dem Start des Log in Millisekunden. Die übertragenen Bytes werden als Hexdump in der nachfolgenden Zeile aufgeführt.

```
# GET_DESCRIPTOR
# SELECT_CONFIGURATION
# GET_DESCRIPTOR
BULK > 02 (URB 6) [134 ms]
    00000000: 01 d0 01 00

BULK < 83 (URB 7)
    00000000: 81

BULK > 02 (URB 8) [134 ms]
    00000000: 01 46 01 00

BULK < 83 (URB 9)
    00000000: 00

BULK > 02 (URB 10) [135 ms]
    00000000: 00 72 01 00 1c

BULK > 02 (URB 11) [148 ms]
    00000000: 00 73 01 00 00
```

Listing 1: Geparste Fassung einer USB Log-Datei

Reverse Engineering mit Perl

Nun kann man sich daran machen, die Bedeutung dieser Daten zu entschlüsseln. Dies erweist sich je nach Scanner mehr oder weniger schwer. Glücklicherweise ist es beim *CanoScan LiDE 600F* sehr einfach. Der Abstraktionsgrad des Protokolls ist sehr hoch. Die Sequenzen könnte man als Funktionsaufrufe einer speziellen Scanner-Sprache ansehen. Ich versuche dies kurz zu verdeutlichen.

Es fällt auf, dass die Zugriffe nur auf die Endpoints 02 und 83 gehen, wobei über 02 die Ausgabe und über 83 die Eingabe von Daten erfolgt und nach einer statistischen Analyse über die Häufigkeit einzelner Bytes kommt man der Bedeutung auf die Spur. Die Datensequenz 01 d0 01 00 besteht aus vier Teilen: dem Befehlscode, der Registeradresse und einer Länge als 16Bit-Wert (LSB first).

URB 6 bedeutet somit folgendes: „Frage den Wert des Registers d0 ab“. Der Rückgabewert (URB 7) ist ein einzelnes Byte. Analoges gilt für URB 8 und 9. Für URB 10 bis 11 gibt es keine Rückgabewerte. Diese Befehle setzen ein Register. Die Datensequenz 00 72 01 00 1c von URB 10 bedeutet daher sinngemäß: „Setze das Register 72 auf den Wert 1c“.

Insgesamt lässt sich die in Listing 1 gezeigte Sequenz in einer einfachen Pseudosprache schreiben (siehe Listing 2).

Variablenzuweisungen gibt es hier nicht. Die Werte sind daher als Kommentar angegeben.

Damit ist die Dekodierung des Protokolls schon abgeschlossen. Es gibt außer get und set noch ein paar weitere Befehle, die ich aber wegen der Kürze des Artikels hier nicht aufführen möchte.

An diesem Punkt haben wir z.B. die oben erwähnte 7 MB große Log-Datei auf etwa 7 KB reduziert! Dies entspricht etwa 300 Zeilen in obiger Pseudosprache.

Um nun mit *Perl* den Scanner über USB steuern zu können, müssen wir nur die obigen Pseudobefehle als Basisfunkti-

```
( get d0 )      # -> 81
( get 46 )      # -> 00
( set 72 1c )
( set 73 00 )
( set 41 80 )
...
```

Listing 2: Dekodierte Fassung der USB-Übertragung

onen implementieren. Siehe Listing 4. Baut man sich noch einen Parser für obige Sprache, kann man den Scanner sogar direkt in dieser Sprache programmieren. Dies ist nicht sehr kompliziert. Für Interessierte sei erwähnt, dass obige Syntax an *Lisp* bzw. *Scheme* angelehnt ist.

Ich will nun anhand von zwei Listings zeigen, wie ein Zugriff auf ein USB-Gerät mit *Perl* aussehen kann. Dabei reduziere ich die Befehle auf das Nötigste.

Man braucht zuerst das *Perl*-Paket `Device::USB` [4], um damit auf *libusb* zugreifen zu können. Es kann vom *CPAN* [5] heruntergeladen werden.

Um das USB-Gerät anzusprechen, erzeugt man zuerst sich neues `Device::USB` Objekt. Dann sucht man das Gerät über `find_device` und öffnet es anschließend mit `open`. Ein `close` wird nicht benötigt. Natürlich sollte man in einer robusten Implementation noch Prüfungen einbauen, ob die Befehle erfolgreich waren.

Die weiteren Befehle zeigen, wie man mit *Perl* die USB Konfiguration des geöffneten Gerätes ausliest. Man kann noch mehr auslesen, aber das Beispiel zeigt das Wesentliche.

Die Methode *find_device* sucht hier gezielt nach dem *CanoScan LiDE 600F*. Für andere Geräte muss man die beiden Parameter anpassen. Der erste Parameter ist die *Vendor ID* und der zweite Parameter die *Produkt ID*.

Sollte man diese noch nicht kennen kann man sich z.B. mit dem Befehl `sane-find-scanner -v -v` eine ähnliche Ausgabe anzeigen lassen wie mit obigen Skript. Ansonsten sind die IDs auch in der Ausgabe des Kommandos `cat /proc/bus/usb/devices` zu finden.

```
#!/usr/bin/perl
use Device::USB ;

my $usb = Device::USB -> new() ;
my $dev = $usb -> find_device ( 0x04a9 , 0x2224 ) ;
$dev -> open() ;

display ( 'bcdUSB' , $dev -> bcdUSB() ) ;
display ( 'bDeviceClass' , $dev -> bDeviceClass() ) ;
display ( 'bDeviceSubClass' , $dev -> bDeviceSubClass() ) ;
display ( 'bDeviceProtocol' , $dev -> bDeviceProtocol() ) ;
display ( 'bMaxPacketSize0' , $dev -> bMaxPacketSize0() ) ;
display ( 'idVendor'  , sprintf('0x%04X',$dev -> idVendor()) ) ;
display ( 'idProduct' , sprintf('0x%04X',$dev -> idProduct()) ) ;
display ( 'bcdDevice' , $dev -> bcdDevice() ) ;
display ( 'iManufacturer' , $dev -> iManufacturer() , 1 ) ;
display ( 'iProduct' , $dev -> iProduct() , 1 ) ;
display ( 'iSerialNumber' , $dev -> iSerialNumber() , 1 ) ;
display ( 'bNumConfigurations' , $dev -> bNumConfigurations() ) ;

for ( $i = 0 ; $i < $dev -> bNumConfigurations() ; $i ++ )
{
  display ( ' <configuration ' . $i . '>' ) ;
  $cfg = $dev -> get_configuration ( $i ) ;
  display ( ' wTotalLength' , $cfg -> wTotalLength() ) ;
  display ( ' bNumInterfaces' , $cfg -> bNumInterfaces() ) ;
  display ( ' bConfigurationValue' , $cfg -> bConfigurationValue() ) ;
  display ( ' iConfiguration' , $cfg -> iConfiguration() ) ;
  display ( ' bmAttributes' , $cfg -> bmAttributes() , 'Remote Wakeup' ) ;
  display ( ' MaxPower' , $cfg -> MaxPower() . ' mA' ) ;
}

sub display
{
  my ( $key , $val , $txt ) = @_ ;
  my $pad = ' ' x 22 ;
  $key = substr ( $key . $pad , 0 , 22 ) ;
  if ( $txt == 1 ) { $txt = $dev -> get_string_simple ( $val ) ; }
  if ( $txt ) { $txt = ' (' . $txt . ')' ; }
  print ( $key , $val , $txt , "\n" ) ;
}
```

Listing3: USB Geräte-Informationen ausgeben

```
#!/usr/bin/perl
use Device::USB ;

my $timeout = 500 ;
my $usb = Device::USB -> new() ;
my $dev = $usb -> find_device ( 0x04a9 , 0x2224 ) ;
$dev -> open() ;
print 'libusb:xxx:' , $dev -> filename() , "\n" ;

dispreg ( 'd0' ) ;
dispreg ( '46' ) ;
canon_set ( '02' , '01' ) ;
canon_set ( '02' , '00' ) ;
canon_set ( '01' , '00' ) ;
canon_set ( '01' , '28' ) ;
canon_set ( 'a0' , '04' ) ;
canon_set ( 'a0' , '05' ) ;
canon_set ( '01' , '28' ) ;
canon_set ( '04' , '0c' ) ;
canon_set ( '05' , '00' ) ;
canon_set ( '06' , '00' ) ;
canon_set ( '90' , '27' ) ;
canon_set ( '92' , 'f7' ) ;
canon_set ( '94' , 'f7' ) ;
canon_set ( '93' , '00' ) ;
canon_set ( '91' , '1f' ) ;

while ( 1 )
{
  dispreg ( '91' ) ;
  sleep ( 1 ) ;
}

1 ;

sub canon_set
{
  my ( $reg , $val ) = @_ ;
  my $data = '00 ' . $reg . ' 01 00 ' . $val ;
  my $cnt = $dev -> bulk_write ( hex('02') , hex2bin($data) , $timeout ) ;

  if ( $cnt < 0 )
  {
    die ( join ( ' ' , 'ERROR: canon_set' , $cnt , $reg , '=' , $val ) ) ;
  }

  print 'SET ' , $reg , ' ' , $val , "\n" ;
}

sub canon_get
{
  my ( $reg ) = @_ ;
  my $data = '01 ' . $reg . ' 01 00' ;
  my $cnt = $dev -> bulk_write ( hex('02') , hex2bin($data) , $timeout ) ;

  if ( $cnt < 0 )
  {
    die ( join ( ' ' , 'ERROR: canon_get' , $cnt , $reg ) ) ;
  }

  #--- warten bis Scanner die Daten bereit hat ---
  select ( undef , undef , undef , 0.001 ) ;

  my $data = ' ' ;
  my $cnt = $dev -> bulk_read ( hex('83') , $data , 1 , $timeout ) ;

  if ( $cnt < 0 )
  {
    error ( 'canon_get' , $cnt , $reg , '(bulk_read 0x83)' ) ;
  }

  $data = bin2hex ( $data ) ;
  return ( $data ) ;
}
```

```
sub dispreg
{
  my ( $reg ) = @_ ;
  my $data = canon_get ( $reg ) ;
  print $reg , ' = ' , $data , '   ' ,
        sprintf('%08b',hex($data)) , "\n" ;
}

sub hex2dez
{
  my ( $txt ) = @_ ;
  $txt =~ s![\da-f]+:!!ig ;
  $txt =~ s!\A\s+!! ;
  $txt =~ s!\s+\Z!! ;
  my @data = split ( m!\s+! , $txt ) ;
  map { $_ = hex ( $_ ) } ( @data ) ;
  return ( @data ) ;
}

sub bin2hex
{
  my ( $data ) = @_ ;
  my @data = unpack ( 'C*' , $data ) ;
  my $anz = @data ;
  my $hex = sprintf ( '%02x ' x $anz ,
                      @data ) ;
  return ( $hex ) ;
}

sub hex2bin
{
  my ( $txt ) = @_ ;
  my @data = hex2dez ( $txt ) ;
  my $data = pack ( 'C*' , @data ) ;
  return ( $data ) ;
}
```

Listing4: Lesen/schreiben der Scannerregister

Hier nun noch ein umfangreicheres Beispiel mit dem man die Funktionstasten am Scanner abfragen kann. Wer möchte, kann hiermit leicht ein Skript schreiben, das je nach gedrückter Scannertaste ein bestimmtes Kommando aufruft.

In den Funktionen `canon_set` und `canon_get` sieht man sehr schön, wie die zuvor beschriebenen Datensequenzen als Hex-String zusammengebaut werden. Für die Übertragung zum Scanner muss dieser String ins Binärformat umgewandelt werden. Hierzu dient die Funktion `hex2bin`.

Umgekehrt werden mit `bin2hex` die Daten vom Scanner in ein lesbares Format umgewandelt.

Daten überträgt man mit den Methoden `bulk_write` und `bulk_read`. Bei `bulk_write` muss man keine Länge angeben.*Perl* kennt die Länge des Strings. Aber bei `bulk_read` muss man angeben, wieviele Bytes man lesen möchte. **ACHTUNG**: Man muss einen String als Puffer bereitstellen, der schon die nötige Größe besitzt. Andernfalls kann `bulk_read` die Daten nicht in den Puffer schreiben. Daher wurde `$data` vor dem `bulk_read` mit einem Leerzeichen initialisiert.

Erhält man als User keinen Zugriff auf das USB-Gerät, müssen die Skripte unter dem User *root* laufen. Dies liegt an den eingeschränkten Berechtigungen für normale User. Natürlich kann man kurzfristig die Berechtigungen in `/proc/bus/usb/00?/00?` ändern. Das ist aber nicht besonders sinnvoll, da jedes neue Einstöpseln den Dateinamen ändert und alle Berechtigungen sind dahin.

Wesentlich geschickter ist es, diese Arbeit von *hotplug* erledigen zu lassen. Voraussetzung ist dabei, dass der User Mitglied der Gruppe „Scanner“ ist. Bei den neusten Linuxsystemen sollte dies schon per Default so eingestellt sein.

Von Perl nach C

Nachdem die grundlegenden Funktionen in *Perl* vorlagen, war es sehr einfach die Steuerung nach *C* zu portieren. Hätte ich das Reverse Engineering mit *C* realisieren müssen, wäre sicher alles wesentlich umständlicher geworden.

Das Backend ist nun soweit, dass ich erste Scanversuche mit dem *Xsane* Frontend machen konnte. Es bedarf aber noch weiterer Arbeit, um damit korrekte Scans mit unterschiedlichen Auflösungen zu erzeugen.

Noch ein weiterer Punkt ist mir aufgefallen. Mit *Device::USB* muss man die Nummer (hier 02 und 83) des Endpoints angeben über den man Daten transferieren möchte. Ähnliche Funktionen der *SANE API* in *C*, wie sie mit `sanei_usb_write_bulk` aufgerufen werden, erfordern jedoch die Angabe des Endpoint-Index. Die *sanei*-Funktionen suchen über den Index die jeweiligen Endpoint-Nummern heraus. Für dieses Backend war der Index immer Null. Ein Lesen oder Schreiben über diesen Index wurde dann automatisch den beiden Endpoints zugeordnet. Dies ist anfangs etwas verwirrend und kostet Zeit bei der Fehlersuche.

Jürgen Ernst

Referenzen

[1] http://www.juergen-ernst.de/info_sane.html
[2] http://www.sane-project.org
[3] http://benoit.papillault.free.fr/usbsnoop/
[4] http://search.cpan.org/~gwadej/Device-USB-0.21/lib/Device/USB.pm
[5] http://www.cpan.org/

Wartbare Perl/Tk-Applikationen

Motivation

Meine Perl/Tk-Anwendungen sahen immer wie Kraut und Rüben aus: alles in einer Datei. Und wenn die Applikation immer größer wurde, habe ich bald nichts mehr gefunden, weil es völlig unstrukturiert war.

Im Laufe der Jahre habe ich einen Weg gefunden, wie man auch Perl/Tk besser beherrschen kann. Und diesen Weg will ich in diesem Artikel vorstellen. Dieser Weg beinhaltet einige recht gute Ideen, ist jedoch kein Allheilmittel gegen alle Probleme.

Der allgemeine Weg (das Übliche)

- warnings und strict verwenden
- so wenige globale Variablen wie möglich
- relative kurze Subroutinen mit aussagekräftigen Namen und klaren Interfaces
- Funktionalität logisch aufteilen (~MVC)

Mein Weg

Ich teile die Funktionalität gerne in mehrere Packages auf:

MyApp::Config - Konfiguration

Wenn eine externe Konfiguration verwendet wird (z.B. YAML, INI, ...), dient dieses Package als Schnittstelle dazu (z.B. in Kombination mit TieMyHashCI, das im Artikel zu Ties vorgestellt wird). Bei kleineren Programmen schreibe ich auch mal die Konfiguration selbst in diesen Namensraum. Zur Darstellung der Konfiguration verwende ich gerne ein Objekt oder auch einen globalen zweidimensionalen Hash, der nur lesend verwendet wird. Dafür bietet sich das CPAN-Modul *Readonly* an. Wenn ein globaler Hash verwendet wird, kann dieser eventuell mit dem *Exporter* in den aktuellen Namensraum importiert werden.

MyApp::GUI - Aussehen

Dieser Namensraum implementiert die Benutzerschnittstelle, also das Bauen und Anordnen der einzelnen Widgets. Damit hier kein absolutes Durcheinander entsteht, versuche ich, gemeinsame Funktionalität zusammenzufassen und

```
sub wButtonStandard {
   my( $parent, $text, $command, %options ) = @_;
   my %defaults = ( -font       => $Config{ButtonDefaultFont},
                    -background => $Config{ButtonDefaultColor},
                    -padx       => 3,
                    -pady       => 3,
                    %options,
   );
   if( ref $text ) {
       $defaults{-textvariable} = $text;
   }
   else {
       $defaults{-text} = $text;
   }

    return $parent->Button( -command => $command, %options );
}
```

Listing 1

Schnittstellen herauszuziehen. Dafür sehe ich die beiden folgenden Module vor:

MyApp::Widgets - Schnittstelle zu Tk

Dieser Namensraum dient als Schnittstelle zu den verwendeten Tk-Widgets, der versucht, (über die Konfiguration) sinnvolle Defaultwerte zu setzen und die Verwendung der Widgets so einfach wie möglich zu machen. Eventuell könnte man die Schnittstellenfunktionen zu den Widgets nach MyApp::GUI exportieren. Eine solche Schnittstellenfunktion könnte in etwa folgendermaßen aussehen (siehe Listing 1).

Dann kann er in MyApp::GUI z.B. folgendermaßen verwendet werden:

```
wButtonStandard( $parentFrame, "KlickMe",
   \&MyApp::Callbacks::cKlickSub )
       ->pack( -side => 'left',
       -background => '#ffdd00'
);
```

Wenn man mehrere Buttons mit diesen Werten braucht, bietet es sich an, dafür eine weitere Schnittstelle zu schreiben, z.B.

```
sub wButtonStandardYellow {
   return wButtonStandard( @_,
              -background => '#ffdd00' );
}
```

So sammelt man im Laufe der Zeit immer mehr Bausteine, die (hoffentlich!) helfen, einen besser lesbaren Code zu erzeugen. Und dieses MyApp::Widgets kann man meist auch gut für weitere Tk-Applikationen verwenden.

MyApp::Callbacks - Reaktionen auf Events

In diesem Namensraum implementiere ich gerne alle Callbacks, also z.B. die Funktionalität, wenn man auf einen Button klickt.

Beim Aufruf von Callbacks scheint man häufig globale Variablen benötigen, um z.B. weitere Widgets oder Infos an den Callback mitzugeben. Gottseidank kann man es meist vermeiden, indem man anstelle der Codereferenz eine Arrayreferenz angibt, z.B.

```
wButtonStandard( $parent, $text,
 [ \&MyApp::Callbacks::Irgendwas, "Param" ]
);
```

Hier wird der Funktion *Irgendwas* ein weiterer Parameter mitgegeben, der über `@_` ausgelesen werden kann.

Wenn Schnittstellen zu externen Systemen wie Datenbanken, Directories, ... verwendet werden sollen, kapsle ich die Zugriffe darauf meist in weiteren Namensräumen wie MyApp::DB oder MyApp::LDAP.

Fazit

Dieser Weg kann schon einiges an Ordnung in eine Perl/Tk-Anwendung bringen. Es ist jedoch nicht der einzige Weg; vermutlich gibt es noch bessere. Gerade bei sehr komplexen Anwendungen ist wohl ein OOP-Weg mit den entsprechenden Designpatterns vorzuziehen.

Martin Fabiani

Was... Perl? Hab ich hier nicht installiert!

Da hat man ein schönes Programm geschrieben, das im Prinzip auch etliche Leute haben wollen und dann scheitert es daran, dass auf dem Zielrechner kein Perl installiert ist. Die Lösung sind ausführbare Dateien. In diesem Artikel sollen mehrere Wege gezeigt werden, wie man .exe-Dateien aus Perl-Programmen für Windows erstellt.

TIMTOWTDI

Ich selbst stand vor kurzem vor dem Problem, dass ich aus einem Programm eine .exe-Datei erstellen musste. Mein Programm „SWS Website Backups“ sollte ja auch auf Kundenrechnern laufen. Angefangen habe ich mit `PAR`, mittlerweile wohl die verbreitetste Methode, und habe dann noch weitere Programme getestet.

Einige denken vielleicht gleich an `perlcc`, ein Programm, das mit Perl mitgeliefert wurde. Wurde? Ja, denn mit Perl 5.10 wurde das Programm aus dem Perl Core rausgenommen. Es war fehlerhaft und hat nur mit einfachen Programmen funktioniert.

Die hier aufgeführten Tools wurden alle mit einem ganz simplen Perl/Tk-Skript getestet (siehe Listing 1).

```
#!/usr/bin/perl

use strict;
use warnings;
use Tk;
use Tk::BrowseEntry;

my $mw  = tkinit();

my $b = $mw->BrowseEntry(
    -choices => [1,2,3] )->pack;

 MainLoop;
```

Listing 1

Damit kann man alles mögliche testen. Für einige Tools wurden dann noch extra Skripte geschrieben, mit denen sich spezifische Features austesten ließen.

PAR

PAR ist ein Perl-Modul, das man auf CPAN findet. Ursprünglich von Audrey Tang gestartet, entwickelte es sich rasch zu dem Standardweg zu einer ausführbaren Datei. Das Modul wuchs immer weiter und inzwischen sind zwei Pakete daraus geworden: `PAR` und `PAR::Packer`. Mit dem Modul kommt auch ein Programm mit dem Namen `pp` mit.

Der Name „PAR“ ist an das „JAR“-Format (Java ARchives) angelehnt und das `PAR`-Modul erzeugt nicht nur .exe-Dateien. Man kann das Modul zum Schnüren von Archiven verwenden, in denen alles enthalten ist, was das Perl-Programm benötigt. Damit kann man auch Programme mit allen notwendigen Dateien auf Rechnern verteilen, auf denen schon Perl installiert ist.

Ich verwende dazu immer das `pp`-Programm. Man kann sich zwar selbst einen Packer schreiben, der das PAR::Packer-Modul verwendet, aber mit dem kleinen Programm kann man alles erreichen...

Der einfache Befehl lautet

```
pp -o Programm.exe Programm.pl
```

PAR überprüft mit dem Modul `Module::ScanDeps`, welche Module alles in die .exe gepackt werden müssen. So muss man nicht seine eigenen Module mit der .exe mit ausliefern. Allerdings werden dabei nur Module gefunden, die direkt eingebunden sind. Werden weitere Module dynamisch - z.B.

über ein Plugin-Mechanismus - nachgeladen, werden diese Module nicht gefunden. In dem Fall muss man diese Module extra angeben.

```
pp -MExtra::Modul -o Programm.exe Programm.pl
```

Ähnlich sieht es mit externen Bibliotheken aus. Soll der Endanwender keine zusätzlichen Programme installieren müssen und sollen deshalb externe Bibliotheken (z.B. libgd,...) mit ausgeliefert werden, muss auch das extra angegeben werden.

```
pp -l "voller/Pfad/zur/Library" \
  -o Programm.exe Programm.pl
```

„SWS Website Backups“ besteht aus mehreren Perl-Skripten, die auch einzeln ausgeführt werden können. Auch das kann man mit PAR realisieren:

```
pp -o Programm.exe Programm.pl Programm2.pl
```

Jetzt muss man beim Starten der .exe allerdings darauf achten, dass man den Namen des auszuführenden Perl-Programms mit angeben muss.

```
C:>.\Programm.exe Programm2.pl
```

In meiner Anwendung ist es so, dass es im Prinzip ein „Hauptskript“ gibt, dass bei einem Doppelklick auf die .exe ausgeführt werden soll. Ich habe dann einen kleinen Patch für `PAR.pm` geschrieben, mit dem der Name der .exe maßgebend ist wenn kein Parameter angegeben wurde. Noch ist dieser Patch nicht in die offizielle PAR-Version eingeflossen (siehe Listing 2).

Bisher wurden einfache Konsolenprogramme gepackt und verteilt. Wenn man die .exe ausführt, erscheint immer eine DOS-Box und das ist meistens nicht gewünscht. Der „typische“ Windows-Nutzer verlangt nach einer GUI und dabei ist die DOS-Box natürlich störend. Soll die DOS-Box abgeschaltet werden, muss einfach die Option `-gui` an das Programm `pp` übergeben werden.

Damit ich mir die ganzen Optionen, die ich bei der .exe-Generierung setzen, muss nicht merken muss, schreibe ich mir immer ein kleines Skript, dass die Generierung übernimmt. Dann brauche ich immer nur dieses eine kleine Skript aufrufen und muss dann hinterher nicht feststellen, dass ich eine Option vergessen habe, die ich sonst immer verwendet habe (siehe Listing 3).

Wie bei vielen Dingen, gibt es auch bei PAR einige Sachen auf die man achten muss.

Da ist bzw. war ein Problem mit der Darstellung von Unicode-Zeichen in Perl/Tk-Anwendungen. Packt man eine Perl/Tk-Anwendung und man sieht dann nur noch „kryptische“ Zeichen kann es daran liegen, dass PAR (noch) nicht richtig mit den Zeichen umgeht. In einem Test hat es gereicht, einfach in dem Perl-Programm das Modul `Encode::Unicode` einzubinden.

Auch bei der Verwendung von Daten im `__DATA__`-Bereich muss man aufpassen. Fehlt eine Leerzeile vor dem `__DATA__`-Bereich, kommt PAR nicht damit klar.

```
399      if( !$member ){
400          my $name = (split /[\\\/]/, $0)[-1];
401          $name =~ s/\.[^\.]+$//;
402          $member = _first_member( $zip,
403              "script/$name",
404              "script/$name.pl",
405              $name,
406              "$name.pl",
407          ) or die qq(PAR.pm: Can't open perl script „$name": No such file or directory);
408      }
```

Listing 2

```
#!/usr/bin/perl

use strict;
use warnings;

my $retval = qx{ pp -gui -o Programm.exe Programm.pl };
print "<<$retval>>";
```

Listing 3

Auf Perl-Community.de hat ein User geschrieben, dass seine Anwendung abstürzt und dass ein Verzeichnis `/tmp/par-hfk/cache-*` daran schuld ist. Das liegt dann daran, dass in dem Cache Zombies auftreten können. Taucht das bei der eigenen .exe auf, sollte die Datei mit der Option `--clean` erzeugt werden.

Für Verwirrung bei den PAR-Einsteigern sorgt, dass es verschiedene Versionen von `PAR::Packer` in den PPM-Repositories gibt. Man muss darauf achten, für welche Perl-Version man das Modul installiert. So gibt es für die verschiedenen Perl-Versionen auch die entsprechende PPM-Version. Verwendet man die falsche Version, taucht bei einem Start der .exe meist ein Fehler auf, der besagt, dass eine bestimmte Methode nicht gefunden werden konnte.

Hinter PAR steht nicht mehr nur Audrey Tang, sondern eine ganze Reihe von Personen, die sich an der Entwicklung beteiligen. Allen voran Steffen Müller. Zur Koordination steht eine Mailingliste zur Verfügung und Anfragen werden sehr zeitnah beantwortet und bearbeitet. Bugs werden in der Regel sehr zeitnah gefixt und neue Features werden meistens erst auf der Mailingliste diskutiert und dann umgesetzt.

Die mit PAR gepackten Dateien sind recht träge beim Start; vor allem beim ersten Start. Dann wird nämlich das Archiv entpackt und in einen temporären Ordner gespeichert. Erst danach wird das Programm gestartet. Die .exe ist sehr groß, da der Perl-Interpreter mit in die .exe gepackt wird.

Insgesamt gesehen ist PAR aber eine gute Lösung. Und durch die aktive Community gibt es auch zu (fast) jedem Problem eine Lösung.

Ein großer Pluspunkt von PAR gegenüber anderen Packern ist, dass man mit PAR auch ausführbare Dateien für alle Betriebssysteme erstellen kann. Damit ist gemeint, dass ich auch mit pp auf einem Linux-System eine ausführbare Datei für Linux erstellen kann und auf einem Windows-System eine .exe für Windows.

Wenn es nur um das Packen geht - also keine ausführbare Datei - kann man sogar „multiarch“-Archive erstellen. Das ist dann vorteilhaft, wenn eine Anwendung für mehrere Rechner verteilt werden soll, auf denen unterschiedliche Betriebssysteme laufen und wenn in der Anwendung Module mit kompilierten Anteilen (XS, C,...) verwendet werden. Als „mulitarch“-Archiv sind dann für jedes Betriebssystem die entsprechenden Kompilate mit gepackt.

Cava

Die Bedienung von Cava ist sehr einfach und intuitiv, so dass keine großartige Erklärung notwendig ist. Auch Cava bietet die Möglichkeit, sowohl .exe-Dateien zu erstellen bei denen sich eine DOS-Box öffnet, als auch Anwendungen, bei denen keine DOS-Box erscheint.

Die von Cava generierten Skripte sind wesentlich kleiner, da der Perl-Interpreter und die Module nicht in eine einzelne .exe gepackt werden, sondern in extra Verzeichnissen gespeichert werden. Das bringt auch Geschwindigkeitsvorteile beim Starten. Mit Cava generierte .exe-Dateien starten wesentlich schneller als mit PAR gepackte Dateien.

Ein Plus von Cava ist die GUI (Abbildung 1). Dort kann man alle notwendigen Einstellungen vornehmen. Dort kann man auch für die einzelnen Skripte eigene Namen für die .exe vergeben. Muss bei PAR der Name der .exe gleich dem Perl-Skript sein, ist mit Cava eine freie Namensgebung möglich. Unter „Skripte“ fügt man einfach alle Skripte hinzu, die in die .exe gepackt werden sollen, und unter „Executables“ legt man dann fest, welches Skript welcher .exe-Datei zugeordnet ist. Für jedes Skript wird dann eine einzelne .exe generiert. Die ganzen Einstellungen werden dann in einer Datei gespeichert, so dass man das Perl-Skript jederzeit mit den alten Einstellungen neu packen kann. So ist kein eigenes Skript notwendig, wie es bei PAR der Fall ist.

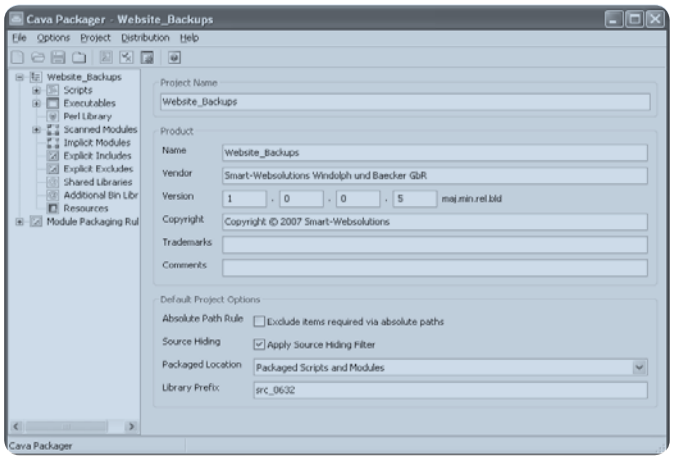


Abbildung 1: Cava GUI

Ein weiterer Vorteil von Cava ist, dass man von dem Programm auch gleich einen Installer mit InnoScript erzeugen kann. So kann man alle wichtigen Schritte von einem Programm aus machen.

Wie bei PAR ist es bei Cava möglich, die Programme durch einen Filter laufen zu lassen und somit den Perl-Code zu verschleiern. Das ist von einigen gewollt, um ihren Perl-Code wenigstens etwas vor Kopierern zu schützen. Wenn der User aber etwas erfahrener ist, wird er solche Programme mit Modulen wie `B::Deparse` wieder einigermaßen herausbekommen.

Wenn man bei Cava zusätzliche Dateien mit ausliefern will - was bei PAR der `-a`-Option entspricht - muss man diese in einen extra Ordner speichern und dann unter „Resources“ diesen Ordner angeben. Einzelne Dateien kann man nicht direkt einbinden.

Hinter Cava steht anscheinend nur ein einzelner Entwickler - vielleicht auch weil das Tool noch nicht so übermäßig weit verbreitet ist - und daher dauert ein Bugfix ein klein wenig länger als bei PAR. Aber eine Anfrage meinerseits wurde innerhalb von einer Woche abgearbeitet und der Bug war gefixt.

Perl2Exe

Perl2Exe ist - genauso wie `pp` von PAR - ein Kommandozeilentool, mit dem Perl-Skripte in ausführbare Dateien umgewandelt werden können. Auf der Webseite von IndigoStar kann man sich zum Testen eine Trial-Version herunterladen. Wenn man mit dem Tool zufrieden ist und es später verwenden will, muss man eine Lizenz kaufen. Es gibt für unterschiedliche Betriebssysteme unterschiedliche Versionen. Allerdings ist die „Pro“-Version mit 149 US$ pro Lizenz nicht ganz billig.

Nur mit der „Pro“-Version lassen sich die Perl-Programme so „kompilieren“, dass die DOS-Box nicht erscheint. Andere wesentliche Features lassen sich aber auch mit der „Lite“-Version verwenden - diese ist mit 49 US$ auch einiges billiger.

Mit dem oben gezeigten Testskript gab es gleich Probleme, weil Perl2Exe nach `use`- und `require`-Befehlen sucht und die genannten Module mit in die .exe einkompiliert. Doch ob das `require` in einem `eval` steht wird dabei nicht berücksichtigt.

Etwas umständlich finde ich die Art und Weise, wie man angeben muss, welche zusätzlichen Module und Dateien mit in die .exe gepackt werden sollen. Dafür müssen im Perl-Quelltext spezielle Kommentare eingebaut werden. Es können auch nicht alle Dateien eingebunden werden, da nur bestimmte Dateiendungen erlaubt sind.

Über einen solchen Kommentar muss man auch sagen, wenn ein Modul nicht mit eingepackt werden soll. Bei dem oben beschriebenen Fehler, dass ein `require` trotz Block-`eval` erkannt wird, muss man mit `#perl2exe_exclude "Win32Util.pm"` sagen, dass das Modul nicht vorhanden ist. Das ist ziemlich umständlich und man muss den Code manuell ändern, wenn man `Win32Util` installiert hat.

Es gibt sogar eine „Enterprise“-Version für 449 US$, mit der sich laut Feature-Übersicht auch von Windows aus .exe-Dateien für andere Plattformen wie Linux oder Solaris erzeugen lassen. Dazu muss man sowohl die Windows- als auch die Linux-Version von Perl2Exe installieren. Laut IndigoStar wird Perl2Exe mit den gängigsten Modulen ausgeliefert und wenn zusätzliche Module mit XS- oder C-Anteilen benötigt werden, müssen diese auf der Zielplattform kompiliert werden und dann auf das Windows-System kopiert werden.

Das hat den Nachteil, dass Perl2Exe selten mit den neuesten Modulen arbeitet.

Generell ist das erzeugen von .exe-Dateien mit Perl2Exe einfach

```
perl2exe.exe Name.pl
```

Damit wird das Programm Name.exe erzeugt. Soll die .exe einen anderen Namen bekommen, kann man diesen mit der `-o`-Option festlegen

```
perl2exe.exe -o=MyName.exe Name.pl
```

Von der Startgeschwindigkeit her sind .exe-Dateien, die mit Perl2Exe erzeugt wurden, schneller als solche, die mit PAR gepackt wurden.

PerlApp

PerlApp ist ein Tool, das - genau wie ActivePerl - von ActiveState stammt und mit dem „Perl Development Kit“ (PDK) geliefert wird. Genau wie Perl2Exe ist das Tool nicht kostenlos und kostet mit dem kompletten PDK stolze 245 US$. Allerdings sind da einige Tools mit dabei. Ein kleineres Paket mit PerlApp ist „PDK Deployment Tools“, das 145 US$ kostet.

Zum Testen kann man sich eine Trial-Version herunterladen. Bei der Installation mit dem .msi-Paket kann es sein, dass mit den Standardeinstellungen eine Fehlermeldung auftaucht, bei der etwas von PerlNet steht. Wenn dieser Fehler erscheint, muss man die Installation nochmal von vorne beginnen und bei den zu installierenden Tools unter „Deployment Tools“ das „PerlNET“ deaktivieren oder vorher das .NET-Framework installieren.

Dieses Tool kann sowohl von der Konsole / DOS-Box als auch mit einer GUI (Abbildung 2) bedient werden. In der GUI bekommt man im Reiter „Output“ auch den passenden Konsolenbefehl geliefert. Ähnlich wie die anderen Tools scannt PerlApp die Perl-Anwendung nach eingebundenen Modulen. Hierbei ist jedoch zu beachten, dass Module, die mit `require` eingebunden werden, manuell in eine Liste von „zusätzlichen Modulen“ hinzugefügt werden müssen.

Was mir persönlich sehr gut gefällt ist, dass PerlApp nach dem Scannen anzeigt, an welcher Stelle im Code das Modul eingebunden wurde. Und als zusätzliche Information bekommt man noch eine Aussage darüber, welche weiteren Module das eingebundene Modul verwenden.

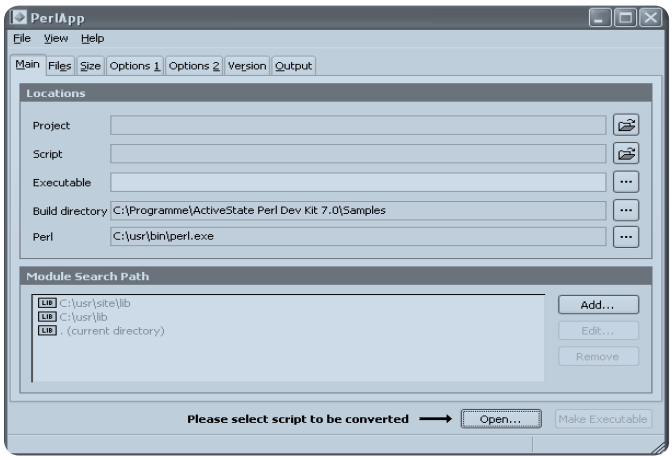


Abbildung 2: PerlApp GUI

Im Gegensatz zu anderen Tools, können hier die Perl-Anwendungen so kompiliert werden, dass die auch ge“debug“t werden können. Zusätzliche Informationen zu der .exe-Datei - wie Version oder Hersteller-Unternehmen - können genauso wie bei Cava angegeben werden. Und es gibt noch zusätzliche Optionen, die die Größe der ausführbaren Datei beeinflussen.

Wie bei den anderen Tools kann auch bei PerlApp der Name der entstehenden .exe frei gewählt werden. Was ich bei diesem Tool allerdings vermisse ist die Möglichkeit, mehrere Skripte zu der .exe hinzuzufügen und somit ausführbar zu machen. Man kann zwar ein zusätzliches Skript scannen lassen und dessen eingebunden Module werden in die .exe einkompiliert, aber die Skripte selbst werden nicht hinzugefügt.

Welches Tool für eigene Projekte verwendet werden soll, kann ich nicht sagen, denn es hängt immer von den eigenen Bedürfnissen ab. Wer vor der Aufgabe steht, eine ausführbare Datei zu erzeugen, der sollte sich die Tools selbst anschauen und ein wenig Testen. Mit allen Tools kommt man in der Regel zum Ziel. Für bestimmte Aufgaben sind aber einige Tools besser geeignet als Andere.

Renée Bäcker

Bericht 10. Deutscher Perl-Workshop

Der 10. Deutsche Perl-Workshop fand vom 13. - 15. Februar 2008 im Regionalen Rechenzentrum an der Universität in Erlangen statt. Traditionell begann der erste Tag mit den Tutorials. Max Maischein hilt ein Tutorial über „Advanced Perl“, während ich in einem anderen Raum beim XS-Tutorial von Marcus Holland-Moritz war. Das XS-Tutorial war sehr gut und hat mich in meinem Vorhaben bestärkt, mich etwas mehr mit XS zu beschäftigen. Ich konnte Marcus auch davon überzeugen, mal eine XS-Einführung für $foo zu schreiben.

Nach den Tutorials stand die Begrüßung durch den Hausherren und Jürgen Christoffel an. Die Begrüßung durch den Hausherr war kurz, prägnant und sehr unterhaltsam. Jürgen Christoffel erklärte dann ein paar organisatorische Dinge. Wie schon in den Jahren zuvor gab es nur einen einzelnen Track - bis auf die Tutorials - und die „normalen“ Vorträge begannen - wie beim 1. Deutschen Perl-Workshop - mit einem Vortrag von Andreas König. Auf meinen persönlichen Wunsch, wie er auch im Vortrag betonte, hat er über CPAN::Reporter gesprochen. Mit CPAN::Reporter können Mails über den Erfolg einer Modulinstallation versendet werden. Diese Mails sind sehr nützlich für Modulautoren, weil man selbst nur ganz selten so viele Plattformen zum Testen zur Verfügung hat wie die CPAN-Tester. Doch auf dieses Thema gehe ich in einem weiteren Artikel genauer ein.

Max Maischein hielt dann einen Vortrag über eines meiner Lieblingsmodule: Web::Scraper. Mit diesem Modul lassen sich sehr einfach Informationen aus HTML-Quelltexten ziehen. Man benötigt keine Regulären Ausdrücke oder andere Module, die das Programm gleich stark ausweiten.

Vor der Kaffeepause kam noch ein sehr interessanter Vortrag über „Debuggen mit Coro“ von Marc Lehmann. Es ist schon ganz interessant, was man mit Coroutinen machen kann.

Der letzte Vortrag des ersten Tages war ein Rückblick über 10 Jahre Deutscher Perl-Workshop von Jürgen Christoffel. Immerhin gab es zwei Personen, die an allen 10 Perl-Workshops teilgenommen haben - Respekt!

Der zweite Tag begann mit einem Vortrag von Steffen Winkler über seine Erfahrungen mit „Perl Best Practices“. Er hat sich kritisch mit den „Regeln“ von Damian Conway auseinandergesetzt. Der Morgen war von „Steffen“s und „Stefan“s beherrscht. So war als nächstes Stefan Hornburg an der Reihe, der über die Integration von Ticket-Systeme über die REST-Schnittstelle, gesprochen hat.

Mit Kaffee gestärkt ging es in den zweiten Teil des Tages. Wieder ein „Steffen“! Steffen Ullrich hat einige Fallstricke vorgestellt, die er in seiner täglichen Arbeit entdeckt hat. Da ich selbst solche Fehler immer wieder sehe, finde ich so einen Vortrag sehr gut.

Danach war Tina Müller an der Reihe. Sie ging auf einige typische Sicherheitslücken in Webanwendungen ein. Sie hat dabei ein noch aktives „Sicherheitsloch“ bei Spiegel online gezeigt. Das war inhaltlich ein sehr guter Vortrag und hat einige Lösungsansätze gezeigt wie man diese Löcher stopfen kann.

Um „Probleme mit base.pm“ ging es in Max Maischeins Vortrag vor der Mittagspause. Dabei hat er sein Modul parent.pm vorgestellt, das einige dieser Problem beheben soll.

Wie man PDFs als Druckvorlagen erstellen kann, zeigte Wolfgang Kinkeldei in seinem Vortrag über „Druckvorlagen mit Perl und AppleScript“. Dabei hat er aus Informationen aus einer YAML-Datei mit Hilfe von PDF::API2 und mit einem ferngesteuertem Adobe InDesign eine PDF-Datei erzeugt. Den InDesign-Teil werde ich mir mal genauer anschauen, da $foo auch mit InDesign gesetzt wird.

Selbst veränderte Filme zeigte Max Maischein. Er hat mit OpenGL und Perl einen Videomixer geschrieben. Einige der

Ergebnisse waren sehr gut. Max hat wohl manchmal zu viel Zeit - vielleicht auch weil er 10 Minuten seiner Vortragszeit für zwei Lightning Talks zur Verfügung gestellt hat: Als erstes hat Jonathan Worthington über „Rakudo“ berichtet. Rakudo ist ein „Perl 6 on Parrot“, das auch gleich mit Perl 6-Rules geschrieben wurde. Damit kann man heute schon einiges von Perl 6 einsetzen. Jonathan hat auch während des Perl-Workshops einige Änderungen an Rakudo vorgenommen. Das war ein richtig guter Lightning Talk. Der zweite Lightning Talk war von Wolfram Schneider, der das „Zack Gateway“ vorgestellt hat. „Zack Gateway“ ist eine Büchersuchmaschine, die mit Perl umgesetzt wurde.

Nach den zwei Lightning Talks gab es dann einen Kurz-Vortrag von Marc Lehmann über sein Event-Modul „EV“. Es ist vielseitiger als anders Event-Module. Als letzter längerer Votrag des Tages sprach Jonathan Worthington über Parallelität. Dabei ging er auf Threads und Fallen wie Deadlocks ein. Er hat auch kurz erklärt wie die Nebenläufige Programmierung in Perl 6 implementiert sein soll.

Den Abschluss des Tages bildeten die Lightning Talks. Ich halte diese Art von Vorträgen für sehr geeignet, um ein Gefühl für Vorträge zu bekommen. Es sollten sich noch viel mehr trauen, solche Vorträge zu halten. Als erstes hat Steffen Ullrich über sein Modul Devel::TraceObjects gesprochen. Danach hat Jost Krieger seinen Weg von Kommandozeilenskripten zu einem Perl-Skript für Auswertungen von Dateien gezeigt. Lars Dieckow zeigte einige Stellen im Web, an den sich „der typische Perlprogrammierer“ aufhält. Danach hat Martin Fabiani zwei Lightning Talks gehalten. Der erste zeigte, wie Martin seine Tk-Anwendungen übersichtlich hält und im zweiten Talk hat er sein Programm excelPerl vorgestellt, mit dem er Excel-Dateien auf der Kommandozeile bearbeiten kann. Eine Geschichte über „Schneewittchen und die sieben Module“ erzählte Lars Dieckow, wobei er das siebte Modul auf Grund des Zeitmangels nicht mehr vorstellen konnte. Slaven Rezic hat einen Weg vorgestellt, mit dem man Datenstrukturen mit Kwalify verifizieren kann. Eine Art, Objekte „lazy“ zu evaluieren zeigte Steffen Winkler mit seinem Modul Object::Lazy.

Nach den Vorträgen hat uns Gert Büttner vom Regionalen Rechenzentrum noch das „Computer Museum“ gezeigt. Zuerst haben wir uns die ganz alten PCs und „Schlepptops“ angeschaut und sind dann in den Serverraum mit den modernen Hochleistungsrechnern gegangen. Das war sehr interessant! Direkt im Anschluss hat uns ein Bus zum Social Event nach Nürnberg zur DATEV gebracht. Die Firma hat freundlicherweise den gesamten Social Event gesponsort. Zur Begrüßung gab es Sekt und nach der Begrüßung durch die DATEV gab ein Buffet mit sehr gutem Essen. Zur Unterhaltung spielte die „Wilde Hilde“ etwas Musik und nach dem Essen brachte sie die Leute zum Lachen. Ich war zwar erst bei fünf Perl-Workshops dabei, aber es ist trotzdem toll, wie sehr sich die einzelnen Social Events unterscheiden.

Der Freitag begann mit Problemen, die Jürgen Christoffel mit „Never touch a running system“ kommentierte. Nathanael hatte Probleme, die Anwendung für seinen Vortrag zum Laufen zu bringen. Auch während des Vortrags stürzte der Rechner einmal ab, aber diese Probleme wurden durch den guten Vortrag wettgemacht. Nathanael zeigte, was man mit Perl und Asterisk alles machen kann. Die Lacher hatte er bei der „Sexhotline für Perl-Programmierer“ auf seiner Seite.

Um Jabber ging es dann im Vortrag von Robin Redeker, der sein Modul Net::XMPP2 vorstellte, bevor ich in meinem „Perl und Ajax“-Vortrag drei Module vorgestellt habe, mit denen man Perl-Anwendungen mit Ajax-Funktionalitäten anreichern kann. Auch nach der Kaffeepause war ich wieder an der Reihe. Ich habe das Tool Selenium vorgestellt, mit dem Webanwendungen sehr einfach getestet werden können.

„Die Schöne und das Biest“ zeigte anschließend Martin Fabiani. Vor dem Mittagessen zeigte er „die Schöne“ - LDAP und die Anbindung an Perl-Programme mittels Net::LDAP - und danach das „Biest“, das Active Directory.

Danach habe ich vorgestellt, wie eigene Policies für Perl::Critic erstellt werden können. Meiner Meinung nach ein wichtiges Thema, denn als Freelancer sieht man jede Menge fremden Code, der häufig auch schon von zig Leuten bearbeitet wurde und da Unternehmen kaum Programmierrichtlinien für Perl haben, kommt da jede Menge Wildwuchs zusammen. Den letzten Vortrag - Wx::XRC von Herbert Breunung - habe ich fast nicht mitbekommen, da ich nach Hause musste.

Für mich persönlich hat sich der Workshop wieder gelohnt. Auch wenn ein Großteil der Vorträge nicht so besonders interssant für mich waren - weil ich mit der Thematik nichts zu tun habe, aber die Gespräche mit anderen Programmierern sind sehr wichtig. Die lokalen Organisatoren haben ganze Arbeit geleistet und den Workshop sehr gut organisiert. Jetzt dürfen wir auf den Workshop 2009 gespannt sein, der von Frankfurt.pm organisiert wird.

Renée Bäcker

FrOSCon

Free and Open Source Software Conference - 2008

23.-24. August 2008
Bonn - St. Augustin

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

Call for Papers bis zum 2.6.08
Call for Projects bis zum 18.5.08

kontakt@froscon.de - www.froscon.de

Fachhochschule Bonn-Rhein-Sieg, Grantham-Allee 20, 53757 St. Augustin

Einer nach dem Anderen bitte...

Es gibt etliche Operatoren in allen Programmiersprachen und zu einem Großteil sind sie auch überall gleich. Jeder setzt die Operatoren ein und macht sich wenig Gedanken darüber, wie und in welcher Reihenfolge diese abgearbeitet werden. In fast allen Fällen macht der Compiler oder Interpreter auch das was man möchte.Perl nennt das DWIM „Do what I mean“.

In einigen Fällen scheint auf den ersten Blick auch alles zu funktionieren, aber bei einem gründlicheren Test würde auffallen, dass vielleicht doch nicht alles so funktioniert wie es soll.

Für die Operatoren gibt es festgelegte Rangordnungen, wie man sie zum Beispiel auch aus der Mathmatik kennt: „Punkt-vor Strichrechnung“. So erkennen wir auf Anhieb in welcher Reihenfolge der Ausdruck `2 + 6 * 3` berechnet wird.

Damit ist schon die Rangordnung innerhalb der Operatoren `*`, `/`, `+`, `-` geklärt. Aber allein in Perl gibt es über 50 Operatoren - und da immer zu wissen welcher Operator eine höhere Wertigkeit hat, ist nicht so einfach. Wichtig ist noch zu wissen, dass alle C-Operatoren die gleiche Wertigkeit in Perl behalten haben.

Dass man leicht mal in die Falle der Wertigkeiten tappen kann, sieht man bei diesem Stück Code:

```
my $link = 'htt://foo-magazin.de';
if (! $link =~ m!^https?://!i ){
   print "Fehler!";
}
```

Die Intention des Programmierers ist, einen Fehler auszugeben wenn ein Link *nicht* mit `http://` oder `https://` anfängt. Sieht alles logisch aus. `$link =~ m!^https?://!i` überprüft, ob der Link mit `http://` anfängt und dann einfach mit `!` negiert. Alles klar, oder?
Hier durchkreuzt aber die Wertigkeit von `!` die Pläne. Dieser Operator hat eine sehr hohe Wertigkeit und so wird erst das `!` angewendet und danach erst `=~`. Also wird erst `$link` negiert und dann wird überprüft, ob die Negation von `$link` mit `http://` anfängt - und das ist nie der Fall, so dass nie „Fehler!“ ausgegeben wird.

Ich will...

... dass der Code das macht was *ich* will!

Um solche Fehler zu vermeiden, kann man mehrere Wege gehen: Entweder man verwendet andere Operatoren und verringert so die Anzahl der Operatoren, man verwendet synonyme Operatoren oder man gruppiert Code-Stücke.

Durch Gruppierung wird alles eindeutig, aber so kann es passieren, dass der Code schnell unübersichtlich wird, weil zu viele öffnende und schließende Klammern enthalten sind:

```
my $link = 'htt://foo-magazin.de';
if (!($link =~ m!^https?://!i)){
   print "Fehler!";
}
```

Duch die Gruppierung wird allerdings deutlich gemacht, dass zuerst das Matching gemacht werden soll und danach das Ergebnis negiert wird - der Code macht das was sich der Programmierer ursprünglich überlegt hat.

Die nächste Möglichkeit ist die Verwendung anderer Operatoren. Wer Perl‘s Operatoren kennt, weiß, dass es für `!(...`

=~) einen alternativen Operator gibt: !~ Damit sieht der Code folgendermaßen aus:

```
my $link = 'htt://foo-magazin.de';
if ( $link !~ m!^https?://!i ){
   print "Fehler!";
}
```

Der Vorteil dieser Möglichkeit liegt darin, dass die if-Bedingung nicht durch Klammernpaare unübersichtlich wird und durch das !~ ist auf den ersten Blick erkenntlich, dass das Matching negiert werden soll. Ein Nachteil ist allerdings, dass viele Einsteiger diesen Operator gar nicht kennen.

Für viele Operatoren gibt es auch einen synonymen Operator, der eine andere Wertigkeit hat. So hat and eine niedrigere Wertigkeit als &&, so wie or eine niedrigere Wertigkeit als || hat. Genauso gibt es für ! einen entsprechenden Operator: not

```
my $link = 'htt://foo-magazin.de';
if ( not $link =~ m!^https?://!i ){
   print "Fehler!";
}
```

Jetzt hat =~ eine höhere Wertigkeit als not und das Matching wird zuerst ausgeführt und danach die Negierung durch das not. Der Vorteil dieser Möglichkeit liegt darin, dass es sehr gut zu lesen ist - fast wie ein vollständiger englischer Satz. Außerdem wird mit =~ ein Operator verwendet, der den meisten (Perl-)Programmierern bekannt ist.

Dieses Beispiel zeigt, wie leicht man in solche Fallen tappen kann. Hier ist es immer wichtig zu testen, ob der Code mit allen möglichen Eingaben auch das macht was er soll. Hier wäre mit wenigen Tests schon klar geworden, dass mit dem Ausdruck etwas nicht stimmt.

Auch das noch...

Ein weiteres - klassisches - Beispiel für so einen Fehler ist der folgende Code:

```
open FH, '<', $file || die $!;
```

open() scheint zwar eine Funktion zu sein, ist aber in Wirklichkeit ein Listenoperator (siehe auch perldoc perlop).

Während der Programmierer in 99,9% der Fälle will, dass die Datei geöffnet wird und in einem Fehlerfall das Programm mit einer Fehlermeldung abgebrochen wird, ist es hier so, dass das open eine niedrigere Wertigkeit als das || hat und somit wird erst $file || die $! gemacht und danach erst das open. Das open ist sogar eines der niedrigsten Operatoren überhaupt (wie alle Listenoperatoren, die rechts stehen), so dass nur noch and, not und or niedriger stehen.

Der obige Code müsste also so aussehen:

```
open FH, '<', $file or die $!
```

Periodensystem der Operatoren

Es gibt ein „Periodensystem der Operatoren“ von Mark Lentczner, das allerdings von 2004 ist und auf den (voraussichtlichen) Operatoren von Perl6 basiert. Allerdings stimmt die Rangordnung auch für die bestehenden Operatoren in Perl 5.

http://www.ozonehouse.com/mark/blog/code/PeriodicTable.pdf

Was macht der Compiler?

Mit dem Modul B::Concise kann man sich ganz gut anschauen, was der Compiler bei ! beziehungsweise not macht. Zur Veranschaulichung wird folgender Code genommen:

```
my $link = 'htt://foo-magazin.de';
if (! $link =~ m!^https?://!i ){}
```

Aufgerufen wird es mit perl -MO=Concise,-exec link.pl. -M ist der Kommandozeilenswitch anstelle eines use im Programm. So kann man statt perl -e „use CGI;“ einfach perl -MCGI schreiben. Das Modul O ist ein Modul, das sogenannte „Compilerbackend“-Module einbindet. Davon gibt es mehrere und alle bieten einen gewissen Blick in Perl-Internas.

Das (Teil-)Ergebnis des Aufrufs ist in Listing 1 zu sehen.

```
C:\>perl -MO=Concise,-exec link.pl
...
7  <0> padsv[$link:1,4] s
8  <1> not sK/1
9  </> match(/"^https?://"/) sKS/RTIME
...
link.pl syntax OK
```

Listing 1

Im Gegensatz dazu der Code, bei dem das `!` durch `not` ersetzt wurde (Listing 2).

```
C:\>perl -MO=Concise,-exec link.pl
...
7  <0> padsv[$link:1,4] s
8  </> match(/"^https?://"/) sKS/RTIME
9  <1> not sK/1
...
link.pl syntax OK
```

Listing 2

Im ersten Beispiel erkennt man, dass der Skalar genommen wird `padsv` als nächstes negiert wird `not` und dann erst der Match gemacht wird (`match`). Mit dem `not` statt dem `!` ist die Reihenfolge von `match` und `not` getauscht.

Renée Bäcker

TPF-TICKER

Act!-Updates

Act! ist ein Konferenz-Verwaltungs-Tool, das in Perl geschrieben ist. Viele der Perl-Workshops und YAPCs werden darüber administriert. Mittlerweile gibt es Act! in vielen Sprachen - unter anderem in Russisch, Portugiesisch, Italienisch, Englisch und Hebräisch.

CRM gesucht, das in Perl geschrieben ist

Die Perl-Foundation ist/war auf der Suche nach einem CRM-System, das in Perl geschrieben ist. Scheinbar gibt es nicht allzu viele davon. Ob Jim Brandt schon etwas gefunden hat, ist nicht klar. Wer also ein solches CRM-System kennt, kann sich an die Perl-Foundation wenden.

Einige Hinweise zu Perl-CRMs sind im Blog gegeben worden: http://news.perlfoundation.org/2008/01/i_need_a_crm_package.html

Vienna.pm
Was bisher geschah...

Vienna.pm wurde 1999 von Roland Bauer gegründet. Da der werte Autor damals noch nicht dabei war (Schande über ihn!), kann über diese frühe Phase nur berichtet werden, was im Mailinglisten-Archiv auffindbar ist. Dieses erzählt von einigen Treffen sowohl technischer als auch sozialer Natur.

Irgendwann 2003 übernimmt der bescheidene Author selbst die „Führung“ der Gruppe von Marcel Grünauer. Aus verschiedenen und teilweise naheliegenden Gründen, die gerade dem deutschsprachigen Publikum nicht erklärt werden müssen, zieht der Autor aber die Bezeichnung „Chief Cat Herder“ dem - wie man so schön sagt - vorbelasteten „Führer“ vor.

Nach zwei Techmeets im Jahr 2003 fand dann 2004 der erste Österreichische Perl Workshop statt, bei dem sich ca 40 Besucher Innen aus Österreich, aber auch aus U.K. und Deutschland (/msg horshack: Dank für das Einrad!!) drei Tage lang ausgezeichnet unterhielten (nicht nur über Perl). Wegen des großen Erfolgs fanden auch 2004 und 2005 Österreichische Perl Workshops statt, allerdings nicht mehr im Museumsquartier, sondern bei Kapsch Carrier Com. Am dritten Perl Workshop fasste sich der harte Kern von Vienna.pm ein Herz und beschloss, sich um die YAPC::Europe 2007 (1) zu bewerben.

Nicht nur aber auch Dank der Unterstützung durch die WU Wien und nfotex setzte sich das Proposal von Vienna.pm durch und auf der YAPC::Europe 2006 in Birmingham konnten bereits einige Highlights präsentieren werden (wie z.B. die Tatsache, dass Larry Wall die Konferenz besuchen wird). Sehr kurz nach dieser Präsentation wurden allerdings die Nachteile einer YAPC-Organisation klar: während der Auktion wurde die Frisur der Organisatoren versteigert, weshalb diese dann im nächsten Jahr mit orangen Irokesen rumlaufen mussten.

Nach einem Jahr monatlicher Organisationstreffen mit 5 bis 10 Teilnehmer Innen war es dann Ende August 2007 soweit. Vienna.pm konnte ca 350 Gäste aus der ganzen Welt begrüssen, unter anderem auch Damian Conway, der gemeinsam mit Larry Wall zahlreiche Fragen zu Perl 6 beantwortete. Nach sehr interessanten (und für die Organisatoren anstrengenden) drei Tagen und einer kleinen Erholungspause stürzte sich Vienna.pm bereits in die nächsten Projekte.

Vienna.pm now!

Obwohl Vienna.pm vor der YAPC::Europe 2007 schon durchaus aktiv war, hat die Organisation dieses Events die Gruppenmitglieder dichter zusammenrücken lassen. Seit Oktober 2007 gibt es einmal im Monat (jeden erste Montag) ein sogenanntens Tech Social Meet. Nach einem ca ein-zweistündigem Vortragsblock in Räumen der Uni Wien besuchen die Teilnehmer Innen (ca 15 Personen) noch ein Gasthaus in der Nähe. Die Vorträge schwanken von kurzen Präsentationen von eigenem Code bis hin zu langen Catalyst-Tutorials. Hin und wieder werden die Treffen auch von Mitgliedern von Bratislava.pm besucht.

Die Mailingliste ist nach wie vor eher ruhig, dafür treiben sich immer mehr Leute im IRC-Channel #Austria.pm (auf dem Server: irc.perl.org) rum.

Mit einem kleinen Teil des Gewinns der YAPC::Europe 2007 mieteten wir einen Server, der von allen Vereinsmitglieder für Dinge wie ssh-access, Subversion-Repositories und Mailingliste, aber auch zum Testen und Betreiben semi-privater Projekte genutzt werden kann.

Der grösste Teil des Gewinns fliesst allerdings in ein Projekt namens Winter of Code (2) (das sich inzwischen langsam in einen Spring of Code umwandelt...). Als erstes Projekt finanziert Vienna.pm die Wiederaufnahme der p5p-Summaries durch David Landgren. Zur Zeit des Schreibens dieses Artikels immer noch in Entwicklung befindet sich ein System zur Verwaltung von TODO-Test-Grants (3). Die Idee ist hier, dass Maintainer von CPAN-Modulen (oder auch von Perl selber) TODO-Tests mit einem Preisschild versehen. Sobald jemand die notwendige Feature implementiert (oder Bugs fixt) und der TODO-Test erfolgreich absolviert wird, zahlt Vienna.pm der implementierenden Person die vorher festgelegte Summe aus. Mehr Informationen zu diesem und den anderen Projekten gibt es im Winter-of-Code Wiki (2).

The Future!

Auf der YAPC::Europe 2008 wird der „Krieg“ zwischen London.pm und Vienna.pm ausgetragen werden. Zu der Kriegserklärung durch London.pm leader Greg McCarrol kam es nachdem der werte Author in #Austria.pm vorgeschlagen hat, eine Perl Monger Gruppe namens not_London.pm zu gründen, damit endlich einmal nicht London.pm die grösste Gruppe der YAPC::Europe Teilnehmer Innen stellt. Der Krieg wird in den drei Disziplinen Armdrücken, Alkoholmissbrauch mit Koordiantionsübungen und Wettkochen ausgetragen. Mehr Infos dazu hier: (4). Ach ja, damit der „Krieg“ nicht ganz umsonst war: Wer sich für die YAPC::Europe 2008 registriert, kann als Monger-Group zusätzlich ‚not_LNDN.pm‘ angeben...

Auf der friedlichen Seite wird es auch dieses Jahr einerseits regelmässige Vienna.pm Treffen geben. Darüberhinaus hat soeben die Planung für den nächsten Österreichischen Perl Workshop begonnen, der als 2-tägiger Event im Oktober in Wien stattfinden wird. Wir freuen uns natürlich auch (vor allem) über Besuch aus außerhalb!

Links

0: http://vienna.pm.org
1: http://vienna.yapceurope.org
2: http://woc.useperl.at
3: http://todo.useperl.at
4: http://socialtext.useperl.at/vienna-pm/index.cgi?why_london_pm_declared_war_on_vienna_pm

Thomas Klausner

Parrot Grant Update November/Dezember

Patrick Michaud und viele Andere sind dabei, Parrot und Perl 6 weiterzuentwickeln. Dabei werden große Fortschritte erzielt. Im November wurde parrot 0.5.0 und im Dezember dann die Version 0.5.1 veröffentlicht.

Viel Neues gibt es dabei von dem Parrot Compiler Toolkit, mit dem relativ einfach neue Compiler gebaut werden können. Dabei ist auch NQP (Not-Quite Perl), ein Subset von Perl 6.

Weiterhin wurden bei einigen Design Documents die Meilensteine erreicht, so dass auch Allison Randal wieder etwas Geld bekommt.

Neue Perl-Podcasts

Perlcast.com

Ovid

Josh McAdams hat einen neuen Perlcast veröffentlicht: Er hat beim Nordischen Perl-Workshop 2007 Ovid über Logische Programmierung interviewed. Dabei steht Ovids Modul Al:: Prolog im Vordergrund. Auch SQL und Perl 6 werden angesprochen.
http://www.perlcast.com/audio/Perlcast_Interview_046.mp3

Stas Bekman und Philippe M. Chiasson

Bereits auf der OSCON 2006 hat Josh McAdams mit den beiden über mod_perl gesprochen. Auch wenn das Interview schon eine Weile her ist, sind die Informationen über mod_perl immer noch aktuell und interessant.
http://www.perlcast.com/audio/Perlcast_Interview_045.mp3

Radio Perl

Radio Perl - Folge 5

Nach fast einjähriger Pause ist Jochen Lillich wieder da! In der 5. Folge von „Radio Perl“ wird das Schwerpunktthema Perl::Critic behandelt. Daneben werden wieder Neuigkeiten und ein paar Module besprochen.
http://www.freistil-consulting.de/audio/download/51/radio_perl-20080112.mp3

CPAN News VI

Test::URI

Tests sind wichtig und dieses Modul ist nützlich, wenn in einem Programm URIs „zusammengebaut“ werden. Dann kann man die Ergebnisse des Programms mit diesem Modul überprüfen. Wird der Hostname richtig in der URI eingesetzt? Diese und weitere Fragen können mit diesem Modul beantwortet werden. Hier wird nicht überprüft, ob die URI tatsächlich existiert, sondern nur, ob die „Syntax“ korrekt ist.

```
#!/usr/bin/perl

use strict;
use warnings;
use Test::URI;
use Test::More tests => 2;

my $uri = 'http://perl-nachrichten.de';
uri_scheme_ok( $uri, 'http' );
uri_host_ok( $uri, 'perl-nachrichten.de' );
```

Test::Aggregate

Mit Test::Aggregate können Testskripte parallel gestartet werden. Allerdings muss man dabei einiges beachten, da die Parallelisierung einige „Schwierigkeiten“ mit sich bringt. Wird in einem Testskript ein „FindBin“ verwendet und ist das Skript, das Test::Aggregate verwendet nicht im gleichen Verzeichnis, so wird das Testskript fehlschlagen. Auch dürfen keine `_DATA`- oder `_END`- Bereiche verwendet werden. Der Autor „Ovid“ rät auch, Testskripte nach und nach in ein extra Verzeichnis zu verschieben. Damit kann man leicht überprüfen, ob ein Testskript mit Test::Aggregate ausgeführt werden kann oder nicht. Dieses Modul ist ganz nützlich, wenn man sehr viele Tests in vielen Testskripte hat und ein `make test` sehr lange dauert. Hier kann die Parallelisiserung einen großen Geschwindigkeitsvorteil bringen.

```
#!/usr/bin/perl

use strict;
use warnings;
use Test::Aggregate;

my $tests = Test::Aggregate->new({
    dirs => 't/',
    dump => 'test.dump',
});

$tests->run;
```

Pod::Extract::URI

In einigen Moduldokumentationen finden sich Links auf andere Webseiten, die weitergehende Informationen enthalten. Manchmal sind aber die Links längst tot, stehen aber immer noch in der Dokumentation. Den ersten Schritt um Links in der Dokumentation überprüfen zu können, bietet das Modul Pod::Extract::URI. Damit kann man sich alle URIs in der Dokumentation holen. Jetzt ist es ganz einfach mit LWP zu überprüfen, ob die Links noch aktiv sind.

```
#!/usr/bin/perl

use strict;
use warnings;
use Pod::Extract::URI;

my @uris = Pod::Extract::URI->uris_from_
file( __FILE__ );
print $_,"\n" for @uris;

= head1 Test

Dies ist ein Test mit http://foo-magazin.de
im freien POD-Text. Und jetzt
gleich mal etwas in einem Code-Snippet:

  my $t = 'http://perl-nachrichten.de';
  print $t;

= cut
```

Prompt::ReadKey

Mit Prompt::ReadKey kann man sehr schnell und übersichtlich Abfragen für das Terminal schreiben. Mit dem Beispiel bekommt man im Terminal `Perl-Magazin [ft]` angezeigt. Je nachdem was man wählt, gibt das Skript „$foo“ bzw. „The Perl Review“ aus. Ähnliche Abfragen kennt man aus Konsolenprogrammen wie `cpan` oder `cpanplus`.

```
#!/usr/bin/perl

use strict;
use warnings;
use Prompt::ReadKey;

my $p = Prompt::ReadKey->new;
my $name = $p->prompt(
    prompt => 'Perl-Magazin',
    options => [
        { name => '$foo', default => 1,
          keys => [qw(f)] },
        { name => 'The Perl Review' },
    ]
);

print $name,"\n";
```

GD:Icons

Minimalistische Icons können mit GD::Icons erzeugt werden. Mit diesem Modul können verschiedenste Formen und Farben kombiniert werden. Es ist wirklich ein ganz einfaches Modul, aber vielleicht entwickelt es sich noch.

```
#!/usr/bin/perl

use strict;
use warnings;
use GD::Icons;

my $icons = GD::Icons->new(
     shape_values => [‘square‘, ‘triangle‘,
        ‘diamond‘],
     color_values => [‘Blue‘, ‘Orange‘],
     icon_dir     => ‘.‘,
     icon_prefix  => ‘foo_magazin_test‘,
 );

$icons->generate_icons;
```

File::Assets

Das ist ein Modul, mit dem man CSS- und JS-Includes in Webanwendungen etwas organisieren kann. Man sammelt erst alle Dateien in einem Objekt und kann dann in einem Template die Include-Anweisungen für diese Dateien einsetzen.

```
#!/usr/bin/perl

use strict;
use warnings;
use File::Assets;

my $assets = File::Assets->new(
  base => [‘http://foo-magazin.de‘]
);
$assets->include( "/style.css" );
$assets->include( "/main.js" );
print $assets->export(‘css‘);
```

Termine

Mai 2008

05. Treffen Vienna.pm
06. Treffen Frankfurt.pm
08. Treffen Dresden.pm
15. Treffen Darmstadt.pm
    YAPC::Asia in Tokyo
16. Russischer Perl-Workshop
    YAPC::Asia in Tokyo
19. Treffen Erlangen.pm
24. Nordischer Perl-Workshop
25. Nordischer Perl-Workshop
28. Treffen Bielefeld.pm
30. Französischer Perl-Workshop
31. Französischer Perl-Workshop

Juni 2008

02. Treffen Vienna.pm
03. Treffen Frankfurt.pm
05. Treffen Dresden.pm
06. Portugiesischer Perl-Workshop in Braga
07. Portugiesischer Perl-Workshop in Braga
16. Treffen Erlangen.pm
    YAPC::NA in Chicago
17. YAPC::NA in Chicago
18. YAPC::NA in Chicago
19. Treffen Darmstadt.pm
25. Treffen Bielefeld.pm

Juli 2008

03. Treffen Dresden.pm
07. Treffen Vienna.pm
08. Treffen Frankfurt.pm
17. Treffen Darmstadt.pm
21. Treffen Erlangen.pm
30. 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

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

http://www.perl-nachrichten.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.perl-community.de

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

Der Deutsche Perl-Workshop hat sich zum Ziel gesetzt, den Austausch zwischen Perl-Programmierern zu fördern. Der 11. Deutsche Perl-Workshop findet 2009 in Frankfurt statt.

http://www.perl-workshop.de

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-foundation.org

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

http://www.Perl.org

Heise Zeitschriften Verlag

Der Heise Zeitschriften Verlag steht für qualitativ hochwertigen Journalismus. Wir verlegen mit c't und iX zwei auflagenstarke Computertitel, das Technologiemagazin Technology Review sowie das Online-Magazin Telepolis. Unsere Website heise online für Computer- und Internet-Interessierte zählt zu den meistbesuchten deutschen Special-Interest-Angeboten.

Zur Weiterentwicklung von heise online suchen wir zum nächstmöglichen Termin

Perl-Profis (m/w)

für die Entwicklung datenbankbasierter Web-Anwendungen.

Ihre Qualifikationen:

- mehrjährige Programmiererfahrung
- intensiver Kontakt mit DBI bzw. SQL sowie CPAN
- Erfahrungen mit einem Anwendungs-Framework (z. B. CGI::Application) und einem Template-System

Folgende Kenntnisse runden Ihr Profil ab:

- Ausgeprägte Erfahrung mit der Linux-Kommandozeile
- Sicherheitsaspekte von Web-Anwendungen (SQL-Injection, XSS)
- JavaScript-, CSS- und XHTML-Erfahrung
- Verarbeitung von XML und Unicode
- Objektorientiertes Programmieren
- Persistente Laufzeit-Umgebungen (FastCGI oder mod_perl)

Wir bieten Ihnen ein inspirierendes Arbeitsumfeld in einem erfolgreichen Team. Wenn es Sie reizt, am Erfolg von heise online mitzuwirken, freuen wir uns auf Ihre aussagekräftigen Bewerbungsunterlagen unter Angabe Ihrer Gehaltsvorstellung und des frühesten Eintrittstermins.

Für weitere Auskünfte steht Ihnen Herr Wolfgang Schemmel unter E-Mail ws@heise.de zur Verfügung.

Bewerbungen richten Sie bitte an:

Ein Unternehmen der Heise Medien Gruppe

Heise Zeitschriften Verlag GmbH & Co. KG
– Personalservice –
Aurelia Quignon

Helstorfer Straße 7
30625 Hannover
Telefon: 05 11/53 52-2 63
Aurelia.Quignon@heise-medien.de

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.