$foo

Perl-Magazin

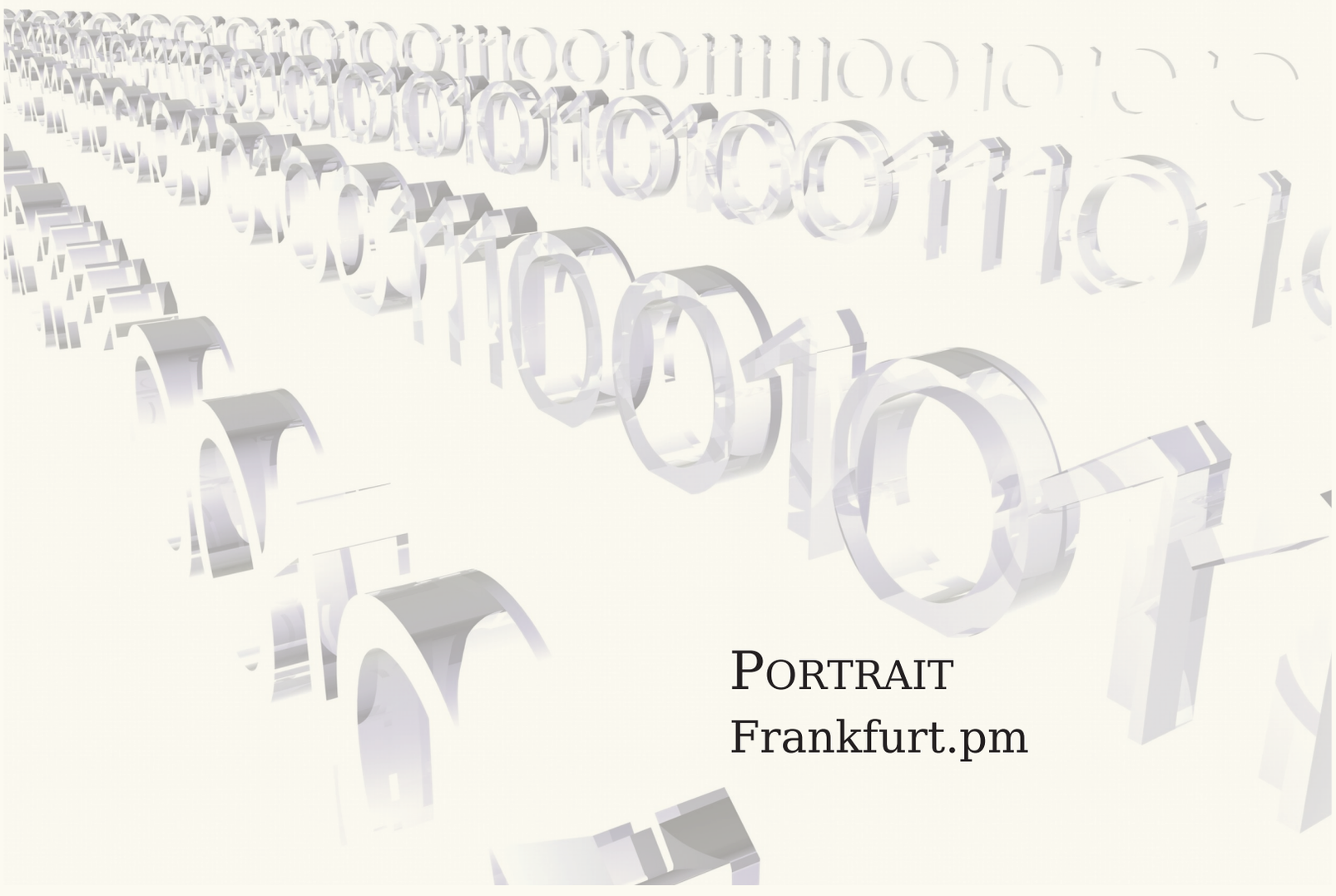
www.foo-magazin.de

Ausgabe: Sommer 2007
Preis: € 6,00

Perl 5.10 Advanced Regular Expressions
Mächtiges noch mächtiger

Perl::Critic
Gute Kritiken - ein Traum für Code

FAQ mit Catalyst
Webanwendungen mit Catalyst



Portrait
Frankfurt.pm

There's More Than One Way to Get There

Conference Theme
Social Perl



Getting to

YAPC

EU::2007

http://vienna.yapceurope.org

28th to 30th August 2007

Vienna, Austria

at the

University of Economics and Business Administration

VIENNA'PM

Larry Wall, Damian Conway,
Mark Jason Dominus
and many more



Der 9. Deutsche Perl-Workshop ist vorbei. An drei Tagen wurden 40 Vorträge und zwei Tutorials gehalten. Besonders erfreulich war, dass ein großer Teil der Anwesenden zum ersten Mal am Workshop teilgenommen haben. Viele davon sind Studenten der FH München gewesen. Es müsste mehr Hochschulen und Universitäten geben, die Perl-Kurse anbieten.

Auch für die Perlmonger-Szene war es ein guter Workshop: In München wurde Hanover.pm (Hannover) aus der Taufe gehoben. Die ganz junge Gruppe hat Jonathan Worthington direkt vom Workshop mit zu ihrem ersten Treffen genommen.

Seit Start dieses Magazin haven wir viel Feedback bekommen. Dabei wurde hauptsächlich das Layout der ersten Ausgabe kritisiert. Wir haben am Layout gearbeitet und hoffen, dass es Ihnen jetzt besser gefällt - Feedback ist immer willkommen!
Hier hat sich bewahrheitet, dass Programmierer häufiger die schlechten Designer sind.

In dieser Ausgabe gibt es erstmals Artikel, die nicht von mir stammen - und ich hoffe, dass der Anteil solcher Artikel weiter ansteigt. Herbert Breunung gibt in dieser Ausgabe einen Einblick in die Entwicklung von Perl 6. Wie mit Perl 6 programmiert wird, zeigt dann die nächste Ausgabe. Ein weiterer - prominenter - Autor ist zu uns gestoßen: Mike Schilli - der vielen aus seiner monatlichen Kolumne "Perl-Snapshot" im Linux-Magazin bekannt sein wird. Er wird in Zukunft von seltsamen Dingen berichten...

Zukünftig wird es für Sie leichter, Beispiele aus den Artikeln nachzuvollziehen und auszuprobieren: Wir werden die Quellcodes aus den Artikel auf unserer Webseite http://www.foo-magazin.de online zur Verfügung stellen. Damit Sie als Käufer der Zeitschrift auch einen Vorteil anderen gegenüber haben, ist der Quellcode nur mit einem Zugangscode zugänglich. Der Code für diese Ausgabe lautet

Q9XN7

Viel Spaß beim Lesen der zweiten Ausgabe von $foo wünscht

Renée Bäcker

Inhaltsverzeichnis

Überblick $foo Sommer 2007

Titelthemen

10 Perl 5.10 Advanced Regular Expressions
27 Catalyst - Wir bauen uns eine FAQ
37 Frankfurt.pm
51 Gute Kritiken - ein Traum für Code (Perl::Critic)

Themen

05 Die Autoren
06 9. Deutscher Perl-Workshop
09 YAPC::Europe CfP
17 Perl6
24 Neues von CPAN
33 CGI::Application
36 CSI Perl
38 YUR.tv - Interview mit Malte Ubl
39 Testen mit Perl
48 Rezension - Algorithmen mit Perl
55 Win32::OLE
59 Podcasts
60 Termine
61 Links

Impressum

Herausgeber: Smart-Websolutions Windolph und Bäcker GbR
Maria-Montessori-Str. 13
D-64584 Biebesheim

Redaktion: Renée Bäcker, André Windolph
Layout: Renée Bäcker, Katrin Blechschmidt
Anzeigen: Katrin Blechschmidt

Auflage: 500 Exemplare
Druck: powerdruck Druck- & VerlagsgesmbH
Wienerstrasse 114
A-2483 Ebreichsdorf

Die Autoren

- ohne die es hier nichts zu lesen gäbe



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, Organi-sator des kleinen Frankfurt Perl-Community Workshop und Mitglied im Orga-Team des deutschen Perl-Workshops.



Herbert Breunung

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



Mike Schilli

Michael Schilli kümmert sich bei Yahoo in Sunnyvale/Kalifornien um die Perl-Belange des Unternehmens. Er schreibt seit nunmehr fast 10 Jahren die monatliche Perl-Kolumne "Perl-Snapshot" im Linux-Magazin.

9. Deutscher Perl-Workshop

Der Deutsche Perl-Workshop ist das größte jährliche Treffen für deutschsprachige Perl-Entwickler. An drei Tagen werden rund 40 Vorträge und 2 Tutorials gehalten. 2007 fand der Deutsche Perl-Workshop vom 21. - 23. Februar an der Fachhochschule München statt.

1. Tag (21.02.2007)

Am 21. Februar 2007 begann der 9. Deutsche Perl-Workshop um kurz nach 9 Uhr. Traditionell beginnt die Veranstaltung mit Tutorials. In diesem Jahr gab es zwei parallele Tracks: CPAN6 von Mark Overmeer und "Advanced Perl 5.10 Reguläre Ausdrücke" von Yves Orton. Der Inhalt des RegEx-Tutorials ist ausführlich in dieser Ausgabe zu finden.

Um 14 Uhr ging es mit der offiziellen Begrüßung durch den Gastgeber an der Fachhoschule - Professor Plate - und Jürgen Christoffel, der "Chef" des Orga-Teams, weiter. Im Anschluss an die Begrüßung zeigte Max Maischein wie man relativ einfach Microsoft Office mit Win32::OLE automatisieren kann - indem man die Aktionen mit dem Makrorekorder aufnimmt und den VBA-Code nach Perl "übersetzt".

Nach dem Office-Vortrag gab es eine ganze Reihe von Vorträgen, die mit "Web" zu tun hatten:

Den Anfang machte Simon Dassow, der einige Module, die er im Zusammenhang mit Catalyst einsetzt vorstellte.

Nur am Rande mit "Web" hatte der Vortrag von Alvar Freude zu tun, der über Code-Optimierung gesprochen hat. Er hat ganz gut gezeigt, wie man die Performance von Programmen steigern kann. Den Vortrag fand ich sehr gut, weil er einige Punkte genannt hat, die wahrscheinlich in ziemlich vielen Programmen "verkehrt" gemacht werden. Wie die Optimierung aussieht ist von Programm zu Programm und von System zu System unterschiedlich. Zusätzlich hat Alvar Freude ein paar Fallen bei der Optimierung aufgezeigt - Codeänderungen, die auf den ersten Blick eine Performanceverbesserung bedeuten müssen, aber beim Test genau die gegenteilige Wirkung zeigten.

Nach der Kaffeepause referierte Michael Kröll etwas über CMS und WebDAV, zuerst ging er auf das WebDAV-Protokoll ein, mit dem über HTTP auf Daten zugegriffen werden kann. Anhand seines XIMS Web-CMS' demonstrierte Michael Kröll die WebDAV-Implementierung.

Interessant war auch den Vortrag von Stefan Fischerländer, der darüber berichtet hat, dass er Perl im Backend einer Web-Suchmaschine einsetzt. Perl wird dabei als Crawler, Indexer und zur Link-Analyse eingesetzt. Im Frontend-Bereich der Suchmaschine wird allerdings kein Perl eingesetzt, da es hier - im Gegensatz zum Backend - sehr stark um Performance geht. Dass sich Perl für ein Backend-System einer Suchmaschine eignet, belegte Fischerländer mit ein paar Zahlen. So läuft das System seit 2 Jahren fehlerfrei - bis auf einen kleinen Programmierfehler - und bearbeitet pro Monat zwischen 5 und 10 Terabyte an Daten.

Der Abschlussvortrag für den ersten Tag handelte von dem Rollenspiel "Crossfire+", für das Marc Lehmann einen Client in Perl geschrieben hat und dafür ein eigenes Widget-Toolkit geschrieben hat. Große Teile des Crossfire+-Servers sind ebenfalls in Perl geschrieben.

Der Tag endete mit dem Social Event statt. Diese Veranstaltung ist eine gute Gelegenheit für die Perl-Entwickler, mit anderen Teilnehmern Kontakt zu knüpfen und über Perl zu diskutieren.

2. Tag (22.02.2007)

Relativ pünktlich ging es mit dem ersten Vortrag von Steffen Ullrich über "Firewalls mit Perl" los. Dabei ging es darum, wie GeNUA ihre Firewall zum Teil mit Perl realisiert haben - und welche Herausforderungen es dabei gab. Bei der Firewall ist die GUI in Perl geschrieben und die Auswertung wird ebenfalls mit Perl gemacht. Die Herausforderungen, die während der Realisierung entstanden, betrafen hauptsächlich Probleme durch die chroot-Umgebung, wodurch ein dynamisches Nachladen nicht möglich ist. Dieses Problem wurde dadurch umgangen, dass alle nötigen des Programms gelöst wurden. Probleme sind auch durch fehlende Maintenance von (CORE)Modulen aufgetreten. So haben die Entwickler von GeNUA auch ein Modul komplett übernommen.

Danach gab Wolfgang Kinkeldei eine Einführung in Jifty. Der Vortrag war ganz gut, da man hier einen guten Überblick über Jifty bekommen hat. Und Wolfgang hat an einem kleinen CMS-Beispiel gezeigt, wie man mit Jifty ziemlich schnell eine kleine Anwendung schreiben kann. Allerdings zeigt sich auch ein Nachteil von Jiftygegenüber Catalyst: Bei Jifty ist man auf die Template-Engine und den Datenbank-Layer festgelegt, während man bei Catalyst seine eigenen Lieblingsmodule verwenden kann.

In der Kaffeepause gab es ziemliche Probleme mit dem Beamer. Etwas verspätet ging es mit Marc Lehmanns neuen Möglichkeiten in der Shell weiter. Danach war ich an der Reihe mit "Testen mit Perl", wobei ich Test-driven development vorgestellt habe. Schon während des Vortrags zeigte sich, dass es extrem wichtig ist, dass eine "außenstehende" Person mal über die Tests schaut. Ich hatte einige Grenzfälle und Sonderfälle in meinen Tests nicht berücksichtigt. Man hat relativ schnell gemerkt, welche Teilnehmer sehr häufig mit dem Thema "Softwaretests" zu tun haben und welche eher selten damit Kontakt haben. Insgesamt gab es einige nützliche Anmerkungen und eine gute Diskussion!

Nach mir war Andreas König mit "Neuerungen in CPAN.pm" dran. Da gibt es ein paar echt nützliche Änderungen! Zum Beispiel wird die Ausgabe farbig dargestellt, was zur Fehlerfindung beiträgt... Noch viel wichtiger sind jedoch die neuen "Distropref"-Angaben, mit denen man zu jeder Distribution eines Moduls genaue Vorgaben festlegen kann. So können damit die langwierigen "Dialoge" ("Soll noch Datei xy installiert werden?") umgangen werden oder bestimmte Versionen von Modulen von der Installation ausgeschlossen werden.

Nach der Mittagspause gab es zwei Vorträge von Thomas Klausner. Der erste war über "Forking Perl". Dabei hat er mögliche Probleme mit fork aufgezeigt. Der zweite Vortrag hat zeigte die bessere Möglichkeit auf: POE.

Danach war ich wieder an der Reihe. Ich zeigte, wie man UML und Perl zusammenbringen kann. Dabei stellte ich vor, wie man mit Perl-Modulen Klassen- und Sequenzdiagramme erzeugen kann. Die geeigneten Perl-Module findet man wie immer auf CPAN.

Mark Overmeer stellte sein Modul XML::Compile vor, bevor Marc Lehmann das Unicode-Modell von Perl erläutert hat.

Im Anschluss hat Jonathan Worthington etwas zu Perl6 erzählt. Zum Beispiel wie das mit der Vererbung und mit Rollen funktioniert.

Als letztes standen die Lightning-Talks auf dem Programm:

Perl-Magazin - Renée Bäcker
Win32::GuiTest - Renée Bäcker
IO::Socket::SSL - Steffen Ullrich
use forks; no Threads - Alvar Freude
cperl-mode - Steffen Schwigon
Faster - Marc Lehmann
vim - Tina Müller
Net::SIP - Steffen Ullrich
Radio Perl - Jochen Lillich

3. Tag (23.02.2007)

Der dritte Tag begann kurz nach 9 Uhr mit dem Vortrag "I18N und Dokumentation mit Perl" von Jens Klöcker. Dabei ist er auf ein paar Produkte von "e/t/s didactic media" eingegangen, die mit Perl umgesetzt wurden.

Als zweites war Marc Lehmann an der Reihe. Er hat sein Modul IO::AIO vorgestellt, mit dem man asynchrones IO realisieren kann.

Vor der Pause war noch Stefan Hornburg mit einem Vortrag über SpamAssassin dran. Er hat gezeigt, wie man mit verschiedenen Modulen - wie ClamAV::Client, Mail::IMAPClient und anderen - Spam aus den Mails herausfiltert. Außerdem ist er darauf eingegangen, welche Hindernisse es dabei geben kann.

Nach der Pause hat Jonathan Worthington die theoretische Informatik sehr praktisch erläutert. Dabei hat er sich Perl6 und Parrot bedient, um einige Sachen zu erklären.
Jonathan hat eine echt gute Art, Vorträge zu halten.

Ein weiterer englischsprachiger Vortrag folge. Richard Foley hat den Perl-Debugger erklärt. Da ich mit dem Debugger nicht so bewandert bin, war der Vortrag ganz gut für mich... Vor der Mittagspause hat Rolf Schaufelberger noch über den Plat_forms-Contest berichtet - aus der Sicht eines Teilnehmers (er hat mit seinem Unternehmen plusW teil-genommen).
Im Anschluss an den Vortrag hat sich eine allgemeine Diskussion über fehlerhaftes Marketing bei Perl

entwickelt. Vielleicht werden ein paar mehr Leute in diesem Bereich aktiv - was mich sehr freuen würde.

Nach der Pause hat wieder Rolf Schaufelberger vorne gestanden und hat eine Einführung in das CMS-System "Bricolage" gegeben. Es sieht so aus, als könnte es eine gute Alternative zu den PHP-Systemen sein.

Max Maischein hat dann über ein Windows-Thema referiert: Er hat sein Modul DBD::WMI vorgestellt. Da ich den Vortrag schon zweimal gehört habe, war es für mich persönlich nicht so interessant (auch wenn das Thema an sich sehr interessant ist).

Für Reports haben Armin Nolte und Dietmar Fiehn ein Programm geschrieben, das aus einer Datenbank die Daten holt und dann mit Excel Grafiken erzeugt, die dann in einer PowerPoint-Präsentation eingefügt werden. Da in dem Projekt ein eigener Quellcode-Parser geschrieben wurde, wurde bei der Diskussion danach auf Perl::Critic hingewiesen. Einige Teile haben sich ganz interessant angehört.

Wolfgang Schemmel sorgte dafür, dass Datenbanken "arbeitslos" werden. Er hat memcached und das Modul Cache::Memcached vorgestellt. Er ist dabei auch darauf eingegangen, welche Probleme es mit Alternativen (z.B. Filesystem) gibt.
Einen Mehrbenutzerfähigen Editor stellte Max Maischein vor. Der "Zaphod" genannte Editor, wurde mit Perl umgesetzt und das Frontend ist HTML mit CSS. Max Maischein demonstrierte die Funktionen des Editors.

Der letzte offizielle Vortrag des Workshops war von Alvar Freude, der über "Perl Best Practices" gesprochen hat. Dabei zeigte er einige "unschöne" Code-Fragmente und zeigt wie man es besser machen kann. Dabei gab es auch eine Diskussion darüber, wie weit man sich an Damian Conways Vorschläge halten soll.

Vor der Abschlussbesprechung mit Verlosung hat Nicholas Clark einen kurzen Vortrag gehalten. Zur Überraschung aller, hat er den Vortrag in Deutsch gehalten! Er hat es wirklich sehr gut gemacht. Es war eigentlich weniger ein Vortrag als vielmehr ein Aufruf an alle, den in Kürze erscheinenden "Release Candidate 1" von Perl 5.8.9 ausgiebig auf allen möglichen Plattformen zu testen und mögliche Bugs an die Perl-Core-Entwickler zu melden (z.B. über perlbug@perl.org).

Nach der Abschlussbesprechung, bei der noch Bücher, T-Shirts und einige andere Sachen verlost wurden, war der 9. Deutsche Perl-Workshop vorbei.
Der 10. Deutsche Perl-Workshop wird im Großraum Erlangen-Nürnberg stattfinden. Der genaue Ort und das genaue Datum (wird aber vermutlich wieder im Februar sein) wird noch bekanntgegeben.

Renée Bäcker

YAPC::Europe 2007 - Call for Papers

Die diesjährige YAPC::Europe findet vom 28. - 30. August 2007 in Wien statt. Das Organisations-Team lädt jeden dazu ein, Vorschläge für einen oder mehrere Vorträge einzureichen.

Die Konferenz in Wien steht unter dem Motto "Social Perl". Wenn Perl einem Unternehmen geholfen hat, Leute zusammenzubringen, oder wenn Perl "sozial" zu anderen Programmiersprachen ist, oder wie Perl von Ideen anderer Programmiersprachen profitieren kann, dann fällt das alles unter das Motto "Social Perl".

Das Motto soll als "lockerer" Leitfaden für Themen von Vorträgen dienen. Es ist aber kein Muss, sich an das Motto zu halten.

Vorträge werden nach der Länge eingeteilt. Es gibt vier Typen:

Standardvortrag

Ein Standardvortrag ist 20 Minuten lang und ist der bevorzugte Vortragstyp.

Langvortrag

Für einen Langvortrag sind 40 Minuten vorgesehen. Diese Art von Vorträgen ist jedoch erfahrenen Sprechern vorbehalten.

YAPC::Europe

Die YAPC::Europe hat sich als eine wichtige Veranstaltung für Perl-Programmiererinnen und -Programmierer in Europa etabliert und zieht auch immer wieder "große Persönlichkeiten" an. In 2007 werden zum Beispiel Larry Wall, Damian Conway, Audrey Tang und Mark Jason Dominus kommen.



Tutorial und Hackathon/BOF/Workshop

In einem 60-90 minütigen Tutorial soll eine Einführung in ein Thema gegeben werden. Die bis zu drei Stunden langen Hackathon/BOF-Sessions können für sehr tiefgehende Beschäftigung mit einem Thema oder auch für tatsächliches gemeinsames Arbeiten an einem Projekt verwendet werden.

Die Konferenzsprache ist Englisch.

Die Vorschläge für Vorträge sollen über die Webseite eingereicht werden, unter der die YAPC::Europe::2007 zu finden ist (http://vienna.yapceurope.org/).
Wer erst wissen möchte, was andere über das Thema denken, kann es im Wiki der YAPC diskutieren.

Einige wichtige Daten

27. Mai 2007

Ende der Frist für die Einreichung von Vortragsvorschlägen.

15. Juni 2007

Die Sprecher werden per Mail benachrichtigt.

22. Juli 2007

Die Unterlagen für den Tagungsband müssen eingereicht sein.

Das wichtigste Datum ist natürlich der 28. August 2007 - der Beginn der YAPC::Europe::2007.

Renée Bäcker

Terminänderung YAPC

Auf Grund eines parallelen Kongresses hat die YAPC-Europe einen neuen Termin: 28.-30. August 2007

YAPC::Europe 2008

Die YAPC::Europe Foundation ist auf der Suche nach einer Gastgeberstadt für die YAPC im nächsten Jahr.

Auf der YEF-Webseite kann man sich auch die Bewerbungen der letzten Jahre anschauen.
Wer Interesse hat, die YAPC zu organisieren, sollte sich bis zum 30. Juni an die YEF wenden.

Mehr Power, mehr Macht

Reguläre Ausdrücke in Perl 5.10

Reguläre Ausdrücke (RegEx) sind in Perl so mächtig wie in kaum einer anderen Sprache. Und mit Perl 5.10 werden RegEx's noch mächtiger. Yves Orton hat an der RegEx-Engine "geschraubt".

Viele neue Features von Perl 5.10 wurden schon in der ersten Ausgabe von `$foo` vorgestellt. Die Änderungen bei den Regulären Ausdrücken sind aber zu groß, so dass es einen eigenen Artikel gibt.

Hier werden keine RegEx-Grundlagen gezeigt, sondern nur die Veränderungen wie sie in Perl 5.10 kommen. Yves Orton hat ganze Arbeit geleistet!

In dem Artikel ist häufig von "Fehler" die Rede. Damit ist kein Fehler gemeint, der einen Programmabbruch hervorruft, sondern ein Ereignis, das die RegEx-Engine zum Backtracking zwingt.

Engine

An der RegEx-Engine hat sich eine Sache grundlegend geändert: Sie arbeitet jetzt iterativ und nicht mehr rekursiv. Damit werden Stack Overflows vermieden. Das hat zur Folge, das Ausdrücke, die die Engine bis Perl 5.8.x zum Crash gebracht haben, jetzt durchlaufen, auch wenn dies sehr, sehr langsam ist.

Auch für Programmierer ändert sich einiges an der Engine. Es gibt eine erweiterbare Schnittstelle, so dass eine beliebige RegEx-Engine verwendet werden kann. Auf CPAN sind schon die Module für `PCRE` und `POSIX` zu finden. Über das Modul `re::engine::Plugin` (befindet sich noch in der Entwicklung) ist es sogar möglich, Plugins in Perl zu schreiben.

Diese Plugins müssen nicht für ein ganzes Programm geladen werden, sondern können auch nur für einen eingeschränkten Gültigkeitsbereich geladen werden. Genauso ist der Debug-Modus für die Regulären Ausdrücke jetzt auch lexikalisch begrenzt.

```
1 use re::engine::POSIX;
2 if ("I like Spoo" =~ /like \(.*\)/) {
3     print "Yay $1\n";
4 }
```

Listing 1

Die Unterschiedlichen Engines für Reguläre Ausdrücke haben unterschiedliche Notationen. So dass man schon darauf achten muss, welche Engine gerade eingesetzt wird. Die Regulären Ausdrücke in Perl haben den größten Sprachumfang, das heißt hier gibt es mehr Sonderzeichen als bei anderen Engines (Listing 1).

Possessive Quantoren

Bisher war es so, dass das Backtracking nur über das `/(?>X)/` Konstrukt "kontrolliert" werden konnte, wobei `X` für einen beliebigen Ausdruck steht (siehe auch `perldoc perlretut` "Using independent subexpressions to prevent backtracking"). Mit den erweiterten Regulären Ausdrücken ist die Schreibweise einfacher geworden: Hinter den Quantor wird einfach noch ein "+" an den Hauptquantor gehängt, also wird zum Beispiel `/(?>X+)/` einfach `/X++/`.

Standardmäßig wird Perl ein Backtracking durchführen, wenn ein Subpattern mit Quantor dafür sorgt, dass der Rest des Ausdrucks nicht matcht. Da dies nicht immer gewollt ist, bietet die neue RegEx-Engine mit den "possesiv" Quantoren eine Möglichkeit, das zu unterbinden.

Ein Beispiel ist in Listing 2 zu sehen.

Der Ausdruck in Zeile 1 matcht, weil die RegEx-Engine erst alle "a" einfängt und dann merkt, dass "ab+" nicht match, ein "a" wieder freigibt, so dass "ab+" matchen kann.

```
1 # matcht
2 'aabb' =~ /a*ab+/;
3 # "alte Schreibweise" - matcht nicht
4 'aabb' =~ /(?>a*)ab+/;
5 # "neue Schreibweise" - matcht nicht
6 'aabb' =~ /a*+ab+/;
```

Listing 2

Der Ausdruck in Zeile 2 matcht nicht und ist gültige Perl5.8-Syntax. Dass der Ausdruck nicht matcht, liegt daran, dass `(?>a*)` ein unabhängiges Subpattern ist, das alle "a" matcht. Und da es unabhängig ist, gibt es kein "a" frei, obwohl der restliche Ausdruck nicht matcht.

In Zeile 3 ist es genauso wie in Zeile 2, nur dass die "alte Schreibweise" `(?>a*)` durch die "neue Schreibweise" `a*+` ersetzt wurde.

Quantoren

Der Begriff "Quantor" stammt aus der Prädikatenlogik und gehören neben den Junktoren zu den Grundzeichen der Logik. Quantoren erlauben eine Aussage darüber, wie häufig der Ausdruck auf den String zutrifft.

Bei den Regulären Ausdrücken gibt es

- * - 0,1 oder viele Male
- + - 1 oder viele Male
- ? - 0 oder 1 Mal
- {n} - genau n-Mal
- {n,} - mindestens n-Mal
- {n,m} - mindestens n- aber höchstens m-Mal

Rekursive Muster

Rekursive Muster werden verwendet, wenn es um Klammerung geht. Wenn festgestellt werden soll, ob ein Satz oder ein Ausdruck symmetrisch geklammert ist (gleich viele "(" wie ")"), wird ein rekursiver Regulärer Ausdruck verwendet (Listing 3).

Dieser Code ist unübersichtlich und außerdem noch langsam. Auch die Verwendung der globalen Variable is unschön.
Mit der neuen RegEx-Engine wird es wenigstens etwas übersichtlicher. Dazu kann `(?1)` verwendet werden (Listing 4).

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5
 6 our $pat;
 7 $pat = qr!\((?>(?>[^()]+)|(??{$pat}))*\)!;
 8 my $string = '(x(x)y(y))';
 9
10 if( $string =~ /^$pat$/ ){
11     print "matcht!\n";
12 }
```

Listing 3

Der Code ist einiges schneller und es ist keine globale Variable mehr nötig. Es gibt kein inneres und äußeres Muster mehr. Durch die Pattern-Rekursion ist es möglich, den Inhalt eines bestimmten Puffers als Unterausdruck zu verwenden.

`(?1)` nimmt den ersten Puffer des Pattern. `(?R)` ist ein besonderer Puffer. Durch diese Anweisung wird der komplette Ausdruck als Unterausdruck verwendet.

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5
 6 my $pat = qr!\((?>[^()]++|(?1))*\)!;
 7 my $string = '(x(x)y(y))';
 8
 9 if( $string =~ /^$pat$/ ){
10     print "matcht!\n";
11 }
```

Listing 4

Ein einfaches Beispiel für die Pattern-Rekursion ist dieses hier:

```
qr/(A)(?1)(?2)(B)/;
```

Dieser Ausdruck erzeugt einen Treffer bei `AABB` im String. Das `(?1)` nimmt das `(A)` als weiteres Muster und `(?2)` das `(B)`.

Mit der Pattern-Rekursion ist es möglich, Grammatiken zu beschreiben.

So kann eine Grammatik geschrieben werden, die überprüft, ob ein eingegebener Name aus einem Vornamen, einem optionalen Initial und einem Nachnamen besteht und alles darf nur aus den Buchstaben A-Z und a-z bestehen (siehe Listing 5).

Das `(?(DEFINE)...)` in Zeile 8 gehört zu der Gruppe der erweiterten Ausdrücke, genauer gesagt ist es solch ein Pattern: `(?(Bedingung) "wahr"-Zweig | "falsch"-Zweig`.

Das `DEFINE` als Bedingung bewirkt, dass der "wahr"-Zweig nicht direkt ausgeführt wird. Außerdem ist ein "falsch"-Zweig nicht erlaubt. Die Sub-Pattern werden nur durch den Rekursionsmechanismus (beginnend bei `(?&name)`) ausgeführt.

p-Modifier

Es gibt einen neuen Modifier für Reguläre Ausdrücke: `/p`. Das `p`

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5
 6 my @tests = ('Hugo Schneider', 'George W Bush', 'Test Per Son');
 7
 8 my $regex = qr! (?(DEFINE)
 9    (?<name>     (?&vorname) \s (?&initial)? \s (?&nachname) )
10    (?<vorname>  [A-Z][a-z]+ )
11    (?<initial>  [A-Z] )
12    (?<nachname> [A-Z][a-z]+ )
13  )
14 (?&name)!x;
15
16 for my $name ( @tests ){
17     print "$name ok\n" if $name =~ /^$regex$/;
18 }
```

Listing 5

steht für "preserve" und bedeutet, dass die Variablen ${^PREMATCH}, ${^MATCH} und ${^POSTMATCH} nach der Ausführung des RegEx zur Verfügung stehen. Ein kleines Beispiel, das das demonstriert ist in Listing 6 zu sehen.

Diese Variablen entsprechen den Variablen $`, $& und $', haben aber ein paar Vorteile:

Die neuen Variablen sind nur genau dann definiert wenn es

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5
 6 my $string = 'ade';
 7 $string =~ /d/p;
 8
 9 print qq~
10 PREMATCH : ${^PREMATCH}
11 MATCH    : ${^MATCH}
12 POSTMATCH: ${^POSTMATCH}
13 ~;
```

Listing 6

einen erfolgreichen Treffer gab und der p-Modifier verwendet wurde. Außerdem bremst der p-Modifier die RegEx-Engine nicht aus (im Gegensatz zu $` und Co.).

Speicherung der Treffer

Bis Perl 5.8.x wurde die Treffer eines Regulären Ausdrucks in durchnummerierten Variablen gespeichert ($1, $2 .. $#, siehe auch `perldoc perlvar`). Mit Perl 5.10, wird die benannte Speicherung der Treffer möglich sein.

Ein erheblicher Nachteil der durchnummerierten Variante ist, dass sich durch das Einfügen eines weiteren Klammerpaares alles weitere verschiebt (Listing 7).

Gibt `es ein` aus. Wird jetzt noch `ist` gespeichert, steht `ein` nicht mehr in $2, sondern in $3. Also muss der Programmierer den Code an mehreren Stellen anfassen. Das ist eine große Fehlerquelle. Wenn der Programmierer nicht daran denk, und der Reguläre Ausdruck sieht dann `di(es) (ist) (ein) Test` aus, ist die Ausgabe falsch.

Mit benannten Speichervariablen passiert das nicht mehr. Die Deklaration erfolgt über `(?<name_des_puffers>pattern)` oder `(?'name_des_puffers'pattern)`. Die Werte werden dann in dem Hash `%+` gespeichert, in dem die vergebenen Namen die Schlüssel sind.
Muss innerhalb der RegEx auf den

```
1 my $re     = qr/di(es) ist (ein) Test/;
2 my $string = 'dies ist ein Test';
3
4 if( $string =~ /$re/ ){
5     my $es  = $1;
6     my $ein = $2;
7     print "$es $ein";
8 }
```

Listing 7

```
1 my $re     = qr/di(?<puffer1>es) ist (?'name2'ein) Test/;
2 my $string = 'dies ist ein Test';
3
4 if( $string =~ /$re/ ){
5     my $es  = $+{puffer1};
6     my $ein = $+{name2};
7     print "$es $ein";
8 }
```

Listing 8

Puffer zurückgegriffen werden, kann dies mit `\k<name>` oder mit `\k'name'` erfolgen.

Ab Perl 5.10 könnte obiges Beispiel so aussehen wie in Listing 8.

Backreferences

Die Backreferences können Probleme bereiten, da sie mehrdeutig sein können. So kann bei `\11` nicht genau gesagt werden, ob das eine Referenz auf den ersten Treffer und dann eine 1 (literal) ist oder ob das eine oktale Zahl ist. Das traditionelle escapen funktioniert in diesem Fall nicht. Um es eindeutig zu machen, muss das Konstrukt `(?:\1)1` genommen oder der x-modifier verwendet werden.

Die neue Schreibweise für Backreferences verhindert diese Probleme und fügt gleichzeitig ein paar neue Features hinzu. Anstelle von `\1` wird jetzt `\g1` oder `\g{1}` verwendet. Mit der neuen RegEx-Engine sind auch relative Referenzen möglich. Mit `\g-1` beziehungsweise `\g{-1}` wird der letzte Treffer genommen. Bei

```
qr/Die(s) (e)in T\g-1st/
```

muss nach dem "T" ein "e" auftauchen, weil dass das letzte Klammerpaar vor der Referenz ist.

Die Backreferences können auch zusammen mit der Neuerung "benannte Speicherung" eingesetzt werden. So ist ein Konstrukt wie in Listing 9 möglich.

Quantoren

Ein TRIE, oder auch "Prefix-Tree", speichert mehrere Zeichenketten gleichzeitig. TRIES werden verwendet, um gleichzeitig und effizient nach Zeichenketten zu suchen.

In einem TRIE repräsentiert jeder Knoten eine Zeichenkette, die durch das Aneinanderreihen der Kanten-buchstaben entsteht. Die Abbildung 1 zeigt einen TRIE für die Worte "Perl", "Pen", "Park".

Aho-Corasick ist ein Algorithmus, der auf der Suche von Zeichenfolgen beruht. Er vergleicht einen Eingabetext mit einer endlichen Anzahl von bekannten Mustern. Es ist eine Art "Wörterbuch-Abgleich".

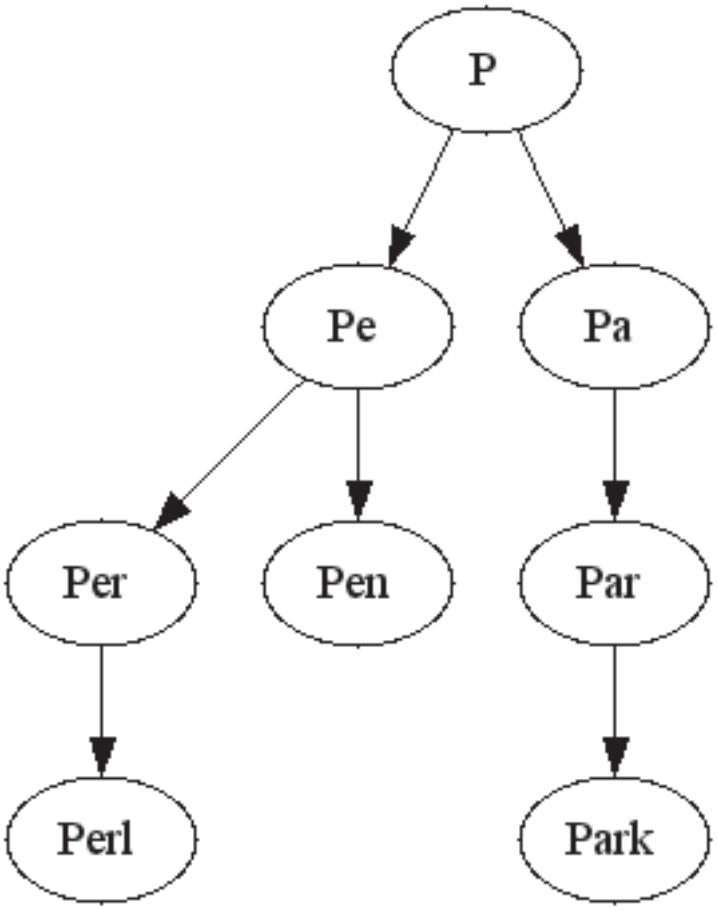


Abbildung 1

```
qr/Dies (?'name'ist) ein Test, \g{name} das ein Test\?/
```

Listing 9

Neue Optimierungen

Haben Ausdrücke Alternativen mit einem fixen Bestandteil, zum Beispiel `/un(fug|sinn|fall)/`, so bildet die RegEx-Engine einen einzigen TRIE. Wenn ein TRIE der erste Operator des Regulären Ausdrucks ist, der einen Treffer erzielt, erzeugt die Engine einen Aho-Corasick-Automaten und verwendet diesen für die Bestimmung des Startpunktes des Regulären Ausdrucks.

Diese Optimierung ist wesentlich effizienter als herkömmliche Alternativen, da die Anzahl der Subpatterns im TRIE die Effizienz nicht beeinflusst. Das heißt, dass es völlig egal ist, wie viele Zeichenketten in dem TRIE gespeichert sind.

werden. In der RegEx-Engine bis Perl 5.8 wurde die Zeichenklasse immer als Zeichenklasse behandelt, was Zeit kostet wenn nur ein Zeichen enthalten ist. Die neue RegEx-Engine macht daraus das Literal - also wird intern aus `/[.]/` ein `/\./`.

Backtracking-Kontrolle

Ein großer und wichtiger Teil bei den neuen Features sind die Muster, mit denen eine Kontrolle des Backtrackings ermöglicht wird. Dazu wurden Muster der Art `(*VERB:ARG)` eingeführt. Bei den meisten dieser Kontroll-Verben ist das Argument optional, bei zwei dieser Muster gibt es keine Argumente.

```
1 #!/usr/bin/perl
2
3 use strict;
4 use warnings;
5
6 my $string = 'foofoo';
7 $string =~ /(fo+)*(?{$& and print "ohne: ",$&,"\n"})(?!)/;
8 # vs.
9 $string =~ /(fo+)*(*PRUNE)(?{$& and print "mit: ",$&,"\n"})(?!)/;
```

Listing 10

Eine weitere Optimierung besteht darin, dass aus einer Reihe von Alternativen der gemeinsame Prefix gefunden wird. So wird aus `/foam|foal|foad/` einfach `/foa[mld]/`.

Zeichenklassen, die nur ein Zeichen beinhalten werden gerne für das Escapen von Sonderzeichen verwendet. Soll zum Beispiel der "." gematcht werden, so kann dies entweder als `/\./` oder als `/[.]/` geschrieben

PRUNE

Mit `(*PRUNE)` wird der Backtracking-Baum "beschnitten" wenn der Reguläre Ausdruck nicht matcht. Ein Beispiel ist in Listing 10 dargestellt.

Beim Ausdruck ohne `(*PRUNE)` wird das Backtracking angestossen und jeder mögliche Treffer zu jedem möglichen Startpunkt wird ausgegeben. Mit `(*PRUNE)` wird zu jedem Startpunkt immer nur der längste Treffer ausgegeben. Der längste Treffer ist bei `fo+` in dem String das `foo`.

Für `(*PRUNE)` existiert auch noch das Äquivalent `\v`.

SKIP

`(*SKIP)` ist ähnlich wie `(*PRUNE)`, nur dass es den Cursor bei einem Treffer an das Ende des Treffers setzt und nicht nur eine Position weitergeht. Bemerkbar macht sich das in einem Fall, wie er in Listing 11 zu sehen ist.

Normalerweise wird der Cursor um eine Position weiter gesetzt wenn der Reguläre Ausdruck in einen Fehler läuft. So wird mit `(*PRUNE)` zuerst bei der Position 0 des String angefangen, das `oof` wird gefunden, durch das `(?!)` wird ein Fehler erzwungen. Der Cursor wird auf die Position 1 gesetzt (das zweite `o`) und es wird erneut versucht einen Treffer zu landen.

Mit `(*SKIP)` beginnt es auch bei der Position 0, das `oof` wird gefunden und auch in dem Beispiel wird ein Fehler durch das `(?!)` erzwungen. Jetzt wird der

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5
 6 my $string = 'oofoof';
 7 $string =~ /(o+f)(?{$& and print "ohne: ",$&,"\n"})(?!)/;
 8 # vs.
 9 $string =~ /(o+f)(*SKIP)(?{$& and print "mit: ",$&,"\n"})(?!)/;
10 # vs.
11 $string =~ /(o+f)\V(?{$& and print 'mit \V:',$&,"\n"})(?!)/;
12
13 # mit PRUNE
14 $string =~ /(o+f)(*PRUNE)(?{$& and print "mit SKIP: ",$&,"\n"})(?!)/;
```

Listing 11

Cursor allerdings nicht auf die Position 1 gesetzt, sondern hinter den Treffer, hier also auf die Position 4.

Das kurze Äquivalent zu `(*SKIP)` ist das `\V`.

MARK

Mit `(*MARK)` wird eine Marke gesetzt, wenn ein Ausdruck einen Treffer erzielt. Wenn es zusammen mit `(*SKIP)` eingesetzt wird, wird der Cursor bei einem Fehler an die Marke gesetzt.

Dieses Verb lässt sich auch gut dazu verwenden bei Ausdrücken, die mit "oder" verknüpft sind, festzustellen welcher Ausdruck für den erfolgreichen Treffer verantwortlich ist (Listing 12).

Für `(*MARK:NAME)` gibt es die Kurzschreibweise `(*:NAME)`.

THEN

Es ist ähnlich zum `::` Operator in Perl6. Wenn Der Reguläre Ausdruck einen Fehler erzeugt, wird mit der Alternative in der innersten Klammerung weitergemacht. Es ist eine Art `if/else` für Reguläre Ausdrücke.

(COND (*THEN) FOO | COND2 (*THEN) BAR | COND3 (*THEN) BAZ)

COMMIT

Wenn der Reguläre Ausdruck auf das `(*COMMIT)` trifft, schlägt der ganze Ausdruck fehl und die RegEx-Engine versucht nicht durch Backtracking einen Treffer zu finden (Listing 13)

Das waren die Verben, denen man einen Namen mitgeben kann. Die folgenden zwei Abschnitte behandeln die zwei Verben, die keinen Namen akzeptieren.

FAIL

Dieses Muster erzeugt keine Treffer und schlägt B<immer> fehl. So kann die RegEx-Engine zum Backtracking gezwungen werden. Es entspricht dem `(?!)` und am Sinnvollsten ist der Einsatz in Verbindung mit dem `(?{})`- oder `(??{})`-Konstrukt.

Beispielcode wird in Listing 14 gezeigt.

Beim ersten Beispiel wird nur der maximale Treffer ausgegeben, während die RegEx-Engine mit dem `(*FAIL)` zum Backtracking gezwungen wird.

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5
 6 use vars qw($REGMARK $REGERROR);
 7
 8 my $string = 'ade';
 9 $string =~ /a (?:b(*MARK:b)|
10                 c(*MARK:c)|
11                 d(*:d)) e/x
12     and print "Zweig, der gematcht hat: ",$REGMARK,"\n";
```

Listing 12

```
1 #!/usr/bin/perl
2
3 use strict;
4 use warnings;
5
6 my $string = 'foofoofoobar';
7 $string =~ /(fo+)*[BC]ar/;
8 # vs.
9 $string =~ /(fo+)*(*COMMIT)[BC]ar/;
```

Listing 13

ACCEPT

(*ACCEPT) ist noch nicht stabil und Yves Orton rät in der perldoc noch davon ab, es im produktiven Code zu verwenden. Mit diesem Verb wird ein Treffer bis zu der Stelle an der das (*ACCEPT) steht angenommen, egal wie der Reguläre Ausdruck dann noch weiter geht (siehe Listing 15).

Beim ersten Regulären Ausdruck wird nur eine Warnung ausgegeben, weil $1 nicht initialisiert wurde. Beim zweiten Ausdruck wird nach dem b durch das (*ACCEPT) $1 gesetzt und es wird aaab ausgegeben.

Bei den Mustern **mit** Argumenten werden zusätzlich die package-Variablen $REGERROR und $REGMARK gesetzt. Bei einem Fehler wird $REGERROR auf den Wert des Arguments gesetzt - wenn das Muster bei dem Fehler involviert war.

Debugging

Immer ganz hilfreich für das Debuggen von Regulären Ausdrücken ist die Verwendung von use re 'Debug';. Damit werden jede Menge Informationen über Status der RegEx-Engine ausgegeben und so kann man auch gut erkennen, warum ein Regulärer Ausdruck einen Treffer hat oder nicht.

Renée Bäcker

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5
 6 my $string = 'aaab';
 7 print "Ohne FAIL: \n";
 8 $string =~ /a+b?(?{ print "\t",$&,"\n" })/;
 9 print "Mit FAIL: \n";
10 $string =~ /a+b?(?{ print "\t",$&,"\n" })(*FAIL)/;
```

Listing 14

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5
 6 my $string = 'aaab';
 7 $string =~ /(a+b?c)/;
 8 print "Ohne ACCEPT: $1\n";
 9 $string =~ /(a+b?(*ACCEPT)c)/;
10 print "Mit ACCEPT: $1\n";
```

Listing 15

Perl6

- Der Himmel für Programmierer I

Vor fast 7 Jahren wurde eine neue Generation Perl ausgerufen. Dieser Artikel informiert darüber, was sich bis heute daraus ergeben hat und was wir realistisch erwarten können. Im zweiten Teil, der in der nächsten Ausgabe erscheinen wird, möchte ich mehr auf die Ziele und den genauen Syntex der Sprache eingehen, soweit es im Moment fest steht.

Vorwort

Gleich vorweg: ich bin begeistert. Perl6 ist für mich die mächtigste und durchdachteste Sprache unter dem Erdenhimmel. Und das sage ich nicht obwohl, sondern weil ich verschiedene Sprachen mehr oder weniger kenne und auch Ruby, C#, Python, Java, Fortran, Forth, TECO, ja sogar PHP etwas vorteilhaftes abgewinnen kann. Natürlich sollte man einen Rohbau nicht in den Himmel heben, aber Perl6 ist längst mehr als Wunschdenken und findet seine Anhänger bereits jenseits der Perlgemeinde. Doch beginnen wir am Anfang:

Wie alles begann ...

Auf der OSCON 2000 kündigte Larry Wall spontan ein neues Langzeitprojekt an, als Antwort auf eine spürbare Motivationslosigkeit. Perl5 funktionierte einfach nur und an der Sprache änderte sich wenig, auch in Bereichen in denen es nötig war.
Selbstverständlich brach die Arbeit am Interpreter nie ab, aber die Probleme stapelten sich, sodaß auf anderen Wiesen das Gras manchmal grüner funkelte. Die Perlianer brauchten ein neues, aufregendes Ziel, an dem sie zusammenarbeiten und auf das sie sich freuen konnten. Also sagte damals Larry sinngemäß: "In Ordnung Leute, viel besser als Perl5 bekomme ich es alleine nicht hin. Lasst uns alle zusammen tun und bei der Gelegenheit alles



Verbessern was wir haben: die Sprache, den Interpreter, das CPAN, unsere Kultur. Ich warte auf eure Vorschläge."

Das Sortieren der ersten Ideen

Es regnete 361 Stück. Und so vergingen die ersten 2 Jahre mit warten, lesen und beantworten dieser Vorschläge. Das Ergebnis sortierte Larry thematisch und veröffentlichte es als "Apokalypsen", die Damian Conway nochmal zu Exegesen umformulierte und mit Beispielen anreicherte um Larrys Vorstellungen einer breiteren Leserschaft zugänglich zu machen. Apokalypse mag etwas nach Weltuntergangsfilm klingen, aber Larry meint es im ursprünglichen Sinne: eine Offenbarung von etwas bisher Verborgenem. Da dies meistens etwas angenehmes für "Gute" oder unangenehmes für "Böse" bedeutet, war Larry's einleuchtende Empfehlung hierzu "Sei einfach nicht böse.". Auch das Wort Exegese stammt aus der lateinisch christlichen Theologie und bedeutet Erläuterung, sowie die später aufkommenden Synopsen für Vergleiche oder auch Zusammenfassung stehen.

Perl6 entwickelt sich

Nachdem Herr Wall die Ideen aus seinem und anderen Köpfen zu einem konsistenten Ganzem verband, begann der Spass aber erst. In einer bis heute anhaltenden Debatte in Mailinglisten, IRC-Channel, Telefonkonferenzen und auf Hackathons kommen immer wieder Ideen auf, über die debattiert wird. Als Folge dessen, werden neue Teile zugefügt, alte geändert und öfters auch in einer Kettenreaktion grössere Teile der Syntax überarbeitet um alle inneren Wiedersprüche aufzulösen und ein möglichst einfaches, mächtiges und vor allem konsistentes Regelwerk zu schaffen. Dabei hat Larry Wall ($larry) selbstverständlich immer das letzte Wort, aber es bildete im Designteam auch ein harter Kern mit leicht wechselnder Besetzung heraus, zu dem ebenfalls Damian Conway, Audrey Tang, chromatic, Allison Randal, Nicholas Clark, Patrick Michaud und einige mehr gehören und den man auch @larry nennt. Die aktuell diskutierten Themen werden derzeit wöchentlich von Ann Barcomb zusammengefasst und können u.a. auf perl.com nachgelesen werden oder im heimischen Mailpro-

gramm, daß die Liste perl6-announce@perl.org empfängt. Der jeweils aktuelle Stand der Sprache läßt sich in den Synopsen u.a. auf perl.org nachlesen. Die Synopsen (S01-S29) sind knappe Definitionen der Änderungen gegenüber Perl5. Sie sind nach dem gleichen inhaltlichen Schema eingeteilt wie die Apokalypsen und Exegesen, die nicht mehr gepflegt werden und daher als "historische Dokumente" um Jahre veraltet sind.

Perl heute

Natürlich gibt es Menschen, denen dieses jahrelange hin und her nicht gefällt, die in Perl6 eine akademische Spielwiese sehen, die zwar interessante ausbrütet, aber nicht Perl genannt werden sollte, weil sonst der Eindruck entsteht, daß Larrys Tafelrunde die Zukunft der Sprache verspielt. Die anstehenden Probleme "sollten besser mit dem alten Interpreter und so schnell wie möglich" gelöst werden. Dies ist auch (teilweise) passiert. Perl 5.10 (derzeit noch



5.9.4) hat eine erneuerte und erweiterte Regex-Engine. Fast alle anderen Neuheiten sind Ideen aus dem Perl6-Designprozess, was 5.10 nicht nur zu beeindruckenden Neuerungen verhilft, wie dem "smartmatch"-Operator oder den "given" und "when" Befehlen (siehe letzte $foo-Ausgabe) sondern auch den Übergang zu Perl6 erleichtern wird.

Das gründlich überarbeitete, sehr praktische Perl6-Modell der Objektorientierung kann man ebenfalls schon mit dem Modul namens "Moose" ausprobieren. Die gründlich überarbeiteten Regexen kann man mit re::engine::PCRE testen. Darüber hinaus enthält das CPAN eine ganze Gruppe von Modulen im Namensraum "Perl6::*", die rückwärts portierte feature enthalten. Perl5 ist also auch dank Perl6 lebendig und interessant. Es stimmt aber schon, daß es bis vor etwa 15 Monaten ein Tal in der Perl5-Entwicklung gab, bis sich in diesen Kreisen die Erkenntnis durchgesetzt hat, daß Perl6 eben nicht die nächste Perlversion ist (dies ist 5.10) auf die man nur warten braucht, sondern es ist eine neue Generation Perl, ein loses Netz aus Forschungsprojeken von Freiwilligen die eine neue Platform für mehrere Jahrzehnte bauen wollen, was verständlicherweise seine Zeit benötigt.

Die ewige Debatte

Leider ist dieses Mißverständis immer noch weit verbreitet und führte bei vielen zu der falschen Wahrnehmung, Perl insgesamt sei ins Stocken geraten. Viele meinen deswegen, daß Perl6 kein passender Name sei, weil es reichlich Erwartung auf sich zieht, während das tägliche Handwerk noch längere Zeit im dadurch weniger beachteten und veraltet wirkenden Perl5 getan wird. Auch die Tatsache, daß das gesamte Vorhaben ein großer Satz nach vorne und auch ins Ungewisse ist, schürt eine eifrige Debatte über Vor- und Nachteile verschiedener Abwägungen. Ich glaube aber, daß diese Diskussion wichtig und notwendig ist, auch wenn vieles Wiederholung ist, für die manch einer kein Ohr hat, denn dies ist ein Freiwilligenprojekt. Es lebt von den Menschen, die sich all diese Fragen gestellt haben und zu der Entscheidung kamen, daß es ihnen wichtig ist, hier dabei zu sein, aus welchem Grund auch immer. Und damit ihr für euch auf diese Fragen brauchbare Antworten findet, möchte ich in den folg-



enden Absätzen zeigen, was die wichtigsten Projekte, die mit Perl6 zu tun haben, bis jetzt geschaffen haben.

Die Interpreterfrage

Kurz nach der Bekanntgabe von Perl6, als noch niemand so recht wußte wohin es gehen sollte, war man sich zumindest einig, daß ein neuer Interpreter gebraucht wurde. Der alte (perl mit kleinem p) ist zwar ein hochoptimiertes Kunstwerk, aber stellenweise derart schwer zu pflegen, daß selbst Pumpking Nicholas der Erste berichtete das z.B. in der Regex-Engine Drachen lauern, an die selbst er sich nicht rantraut. Zwar gibt es noch genügend Ritter die tapfer und kundig genug für perls Weiterentwicklung sind, aber es ist anstrengend und zeitraubend der alten Maschine neue Kunststücke beizubringen. Und Perl6 wird davon viele bringen.

Besonders 2 davon sind mit perl kaum realisierbar, da sie den Umbau von fast dem ganzen Quellcode verlangen würden: ein einfacherer XS-Ersatz und Lisp-ähnliche Macros statt der magischen aber gefürchteten Sourcefilter, die langsam und nicht frei von unangenehmen Seiteneffekten sind. Diese beiden Punkte sind aber wichtig, weil sie Perls Erweiterbarkeit und damit den man mit heutiger Erfahrung anders machen würde, beschloß man einen neuen zu schreiben, bei dessen Entwurf bereits alle neuen

Möglichkeiten von Perl6 berücksichtigt werden können.

Ein großartiger Plan

Das Problem dabei ist, daß Perl6 noch lange nicht bis in alle Details ist und vor 5 Jahren kaum mehr als eine rohe Skizze war. Um nicht auf @larry warten zu müssen und den Interpreter unabhängig vom sich entwickelnden Perl6 ent-werfen zu können, sollte er eine eigene Sprache bekommen, in der am Ende ein Parser geschrieben würde, der Perl6 versteht und es in diese, eigene Sprache übersetzen kann. Diese kann der geplante Interpreter einlesen und zu Bytecode kompilieren, der von der zugehörigen Virtuellen Maschine (VM) unter jedem OS (Betriebsystem) ausgeführt werden kann.
Das Prinzip kennt man ja von Java oder .Net. Deswegen kam auch an dieser Stelle die Frage auf: Warum schreiben wir eine eigene VM, wenn es bereits mehrere gibt. Ganz einfach: Weil sie erstens jeweils einer Firma gehören (Sun und Microsoft) und Perl damit kein freies Softwareprojekt mehr wäre, was für die allermeisten Perlbenutzer nicht hinnehmbar ist und rechtliche Problemen nicht ausschließt. (Mono gab es damals noch nicht, da es zur gleichen Zeit entstand.) Zweitens sind diese VM für weit weniger dynamische Sprachen entworfen als Perl, Ruby oder Python. Ein Perlinterpreter weiß einiges mehr über den gegenwärtigen Zustand des Programmes und hat daher auch mehr Möglichkeiten den Speicherbedarf oder die Ablaufgeschwindigkeit zu optimieren. Diesen Vorteil man nicht vergeuden, aber es war auch vorgesehen, auch Bytecode für andere VM herzustellen oder den Bytecode anderer VM ausführen zu können, denn eines der Prinzipien der praktischen "glue-language" Perl, alles verbinden zu können möglichst mit allem zusammenzuarbeiten. Deswegen wollte man auch von Beginn an verschiedene Parser für diesen Interpreter haben, damit er auch viele Sprachen verstehen kann, die sich dann gegenseitig aufrufen und gemeinsam ihre Bibliotheken (Module) nutzen können. Somit



würde dieser Interpreter nicht nur neue, wunderbare Möglichkeiten für Perl und andere Sprachen eröffnen sondern erfüllte auch einen lange gehegten Traum vieler Perl-Programmierer: Kompilierbare Perl-Programme, denn im Gegensatz zu perl sollte der neue Interpreter den von ihm erzeugten Bytecode auch in Dateien abspeichern können (so wie python das bereits seit längerem tut).
In dieser Form kann das Programm wesentlich schneller starten und auch verteilt werden (kopieren reicht), ist aber nicht mehr für Menschen lesbar und veränderbar. Das kommt jedoch Firmen entgegen, die unfreie (proprietäre) Software erstellen wollen und deswegen Perl als Sprache für Applikationen gemieden haben.

Parrot, der neue Papagei der Perl Piraten

Was für einem groß(artig)e Vision, doch wer verwirklicht sie? In der Anfangszeit gab es mit Chip Salzenbergs "Topaz" und Simon Cozens "Sapphire" zwei Versuche einen neuen Interpreter zu schreiben, die man eher als Machbarkeitsstudien beschreiben könnte. Erst Dan Sugalski, der sich bei seiner Arbeit am VMS-Port mit Perl's Innenleben vertraut machte, hatte einen wirklichen Plan, Fleiß und die nötige Er-fahrung um eine solch anspruchs-volle Software zu entwerfen. Auch wenn er leider nicht mehr dabei ist, so haben seine heutigen Nachfolger, seinen klugen und vorrausschauenden Entscheidungen viel zu verdanken.

Er beschloß zum Beispiel das die VM ein registerbasierndes Design bekommt und kein stackbasierendes wie Java oder oder .Net. Dies wird zwar schwerer zu programmieren sein, aber die akademische Forschung der letzten Jahrzehnte legt nah, daß eine solche VM schneller sein wird, was besonders bei komplexeren Programmen mit mehreren Prozessen oder anspruchsvolleren Fähigkeiten wie den "continuation" deutlich werden wird. Auch die bereits angedeutete Sprache des Interpreter und PMC-Register, die komplexe, definierbare Datentypen beinhalten können, sind grundlegende Ideen, die auf Dan zurückgehen. Eines stammt aber definitiv nicht von ihm und das ist der Name.

Als Dan anfang 2001 mit seiner Arbeit begann, wurde der Name ungefähr zeitgleich aber ganz woanders gefunden. Simon Cozens hatte sich auf perl.com wirklich gut gemachten Aprilscherz geleistet. Dreist inszenierte er ein Interview mit Guido van Rossum und Larry Wall, die feier-lich die Vereinigung von Python und Perl bekannt gaben. Statt des angekündigten Python 3000 (das bis heute nur eine Idee ist) und Perl6 sollte es einen gemeinsamen Nachfolger namens Parrot geben, der das Beste beider Sprachen vereinen sollte. Und genau das wird Parrot auf eine wesentlich nervenschonendere Art auch tun.

Was versteht der Papagei? - PBC, PASM, PIR

Trotz mehrfach wechselnder Besetzung und relativ vielen Überarbeitungen entwickelt sich Parrot stetig vorran, was vor allem ein Ver-dienst von Dan Sugalski, der es ihm zu viel wurde wesentliche Teile implementierte, Chip Salzenberg (Perl 5.4 Pumpking), Leo(pold) Tötsch und Will Coleda, Parrots jetziger Projektmanager, ist.

Für Außenseiter ist dies allerdings schwer einschätzbar, da man schon den Quellcode einsehen und ihn verstehen muß, um wirklich den Entwicklungsstand erkennen. Die meisten Subsys-teme, wie z.B. das IO-System, (verantwortlich für das lesen und schreiben von Dateien) sind in unterschiedlicher Qualität implementiert, einzig das System für die Objektorientierung ist so gut wie nicht vorhanden. Nach aussen zeigt sich allerding ein anderes Bild. Es gibt einige in Parrot Assembler (PASM) geschriebene Testprogramme, die Funktion und Geschwindigkeit Bereiche überprüfen und Parser für Sprachen wie z.B. Ruby, TCL, Python, Scheme und Basic, die sich in diversen Rohzuständen befinden, also nicht benutzbar sind.

Folglich zieht diese Spielwiese bisher nur Kinder an, die Freude am Programmieren in C und PASM haben, aber kaum jemand, der Parrot wirklich als Interpreter benutzt. Das wird sich wohl erst mit den Entwicklungen ändern, die ich im nächsten Absatz beschreibe. Warum ist eigentlich Parrot in C geschrieben? Tja, weil C sehr schnell ist, viele Programmierer C können und vor allem weil Perl6 überall da laufen soll, wo es bereits Perl5 tut. Das umschließt viele, mitlerweile seltene Systeme, deren gemeinsamer Nenner fast nur C-Compiler wie der GCC sind. Die andere Sprache ich nannte war PASM. Wie oben erwähnt, ist Parrot eine registerbasierende VM, also eine Software, die einen Prozessor, ähnlich einer CPU, simuliert.

Entsprechend dazu wird Parrot in einer assemblerartigen Sprache programmiert, die all jene grundlegenden Rechenoperationen hat, die man aus der Hardwareprogrammierung kennt, nur das hier die Register, je nach Typ, noch einiges mehr können, wie z.B. Stringoperationen, die an Perl erinnern. Auch gibt es Register für Objekte aber keine für Speicheradressen, weil vollautomatische Speicherverwaltung Aufgabe der VM ist und es ja keinen direkten Kontakt zum Arbeitsspeicher geben darf, sonst wären die Programme nicht mehr hardwareunabhängig. Dafür ist die Registeranzahl nicht begrenzt, man muß sie nur zu Beginn eines Parrotprogrammes wie ein Objekt erzeugen (anmelden). Trotzdem sind einige Registertypen wie Int(eger) schon dafür ausgelegt, von echten CPU-Registern abgebildet zu werden, was Parrot hierbei sehr beschleunigt. Dazu ein kleines Beispiel aus einem offiziellen Benchmarkprogramm zur Verdeutlichung:

```
      new P0, .PerlArray
      set I0, 100000
      time N0
    lp1:
      push P0, I0
      dec I0
      if I0, lp1
      time N1
      sub N1, N0
      print "constr.time "
      print N1
      print "\n"
```

Auf diese Art kommt man natürlich kaum vom Fleck. Deswegen hat der kluge Dan auch eine zweite Version von PASM entworfen die Elemente höherer Sprachen enthält und damit kürzere Programme und somit ein schnelleres Programmieren ermöglicht. Diese wird PIR (Parrot Intermediate Representation) genannt. PASM bleibt aber vor allem dann eine wertvolle Alternative wenn grundlegende Dinge definiert werden oder Programmteile besonders optimiert werden sollen. Dateien die PASM enthalten enden auf .pasm, .pir weist auf PIR und .pbc auf Parrot Bytecode. Sowohl PASM als auch PIR, werden zu PBC kompiliert, können aber nicht ineinander umgewandelt werden.

PGE, TGE, Partridge und ein totes Pony

Einmal im Jahr erzählt Larry Wall auf der OSCON vor großem Publikum, wie Perl sich gerade entwickelt und wie er sich die Zukunft vorstellt. Natürlich kommt ein guter Teil der Zuhörer wegen Larrys witzigem Erzählstil und den interessanten Einsichten, die oft nicht sofort erkennbar etwas mit



Perl zu tun haben, aber auch für jeden, der Perl6 weniger ernsthaft verfolgt, bieten diese "State of the Onion" - Reden eine unterhaltsame Zusammenfassung und oft auch wichtige Ankündigungen. Perl6 selbst war hier im Jahre 2000 ausgerufen worden, 2001 hörte man hier erstes von Parrot und 2003 daß es einen Perl5-Parser für Parrot geben wird. Damit wurde Ponie (Perl5 on new interpreter engine) zum vielleicht bekanntesten PIR-Programm und trug auch zwei wichtige Botschaften in die Welt. Nummer eins: Perl5 bleibt solange lauffähig, solange es Parrot geben wird. Und ebenso wichtig: all die wundervollen Perlmodule können auch von Perl6 aus benutzt werden. Doch leider sucht man jetzt andere Wege das zu ermöglichen, weil Ponie Zeit

seines kurzen Lebens (2003-2006) eine Kopfgeburt blieb und leider nie die Unterstützung der Perl-gemeinde erhielt, die es gebraucht hätte. Arthur Bergman and später Nicholas Clark (Perl 5.8 Pumpking), die beide bei der Firma Fotango arbeiten und für dieses Projekt von ihr bezahlt wurden, haben hier dennoch unglaubliches geleistet, was auch dem jetzigen Interpreter zugute kam, da durch diese Anstrengungen einige Fehler in perl entdeckt und behoben und viele Ecken gesäubert wurden. Ponie hat allerdings noch einen kleinen Bruder namens Punie (zu deutsch Winzling), ein von Allison Randal gestarteter Perl1-Parser für Parrot, der vor allem als Testlauf für die Entstehung des späteren Perl6-Parsers dient.

Während die Ponie-Entwickler die perl-Interna studierten, um eine Bug-für-Bug-Kompatibilität mit dem manchmal verwirrenden Verhalten der aktuellen Perl5-Version zu erreichen, ist Punie relativ einfach und nach den Regeln des modernen Compilerbaus gestrickt. Ausserdem versucht Allison, die ihre Position als Leiterin der Perl Foundation aufgab, um mehr Zeit für Punie zu haben, seine Teile möglichst wiederverwendbar zu halten. So entstand die TGE ("Tree Grammar Engine"), die zusammen mit der PGE ("Parrot Grammar Engine") zu "Partridge" gehört, Parrot's neuer Werkzeugkiste für den Bau von Parsern. Aus einer Faszination für die "Perl6 rules" begann Patrick Michaud im Herbst 2004 in PIR einen Compiler für sie zu schreiben, den er PGE nannte.

Rules werden in Perl6 die regulären Ausdrücke ersetzen. Einmal weil sie mächtiger sind und auch weil ihre Schreibweise klarer geordnet ist, als bei den Perl5 Regex, deren Syntax in den letzten Jahren überall dort erweitert wurde, wo noch Sonderzeichen frei waren, denen man eine Bedeutung geben konnte. Der dritte Vorteil der rules gegenüber den Regex ist, daß man sie zu größeren Sinneinheiten zusammenfassen kann. Was in Perl5 oft mit einer sehr komplexen Regex gelöst wurde (z.B. parsen von HTML), kann man in Perl6 mit mehreren verschachtelten, aber wessentlich einfachereren "Regeln" formulieren, nach denen ein Text schrittweise in seine inhaltliche Bestandteile zerlegt wird. Die Arbeit von Parsern für Programmiersprachen kann so, sehr lesbar und knapp, beschrieben werden. Und mit Partridge reicht diese knappe Beschreibung einer Sprache, um daraus automatisch einen Parser zu erhalten, der die Sprache ins PIR übersetzt. Das begeistert nicht nur die Parrot-Mannschaft, weil Parser somit viel einfacher und schneller zu schreiben sind, als in PIR, sondern auch Interpreter- und Compilerbauer außerhalb dieser Kreise, weil dies effektiver ist als die heute üblichen Werkzeuge wie lex und yacc. Doch wie funktioniert Partridge genau? Ein Satz von Perl6-rules, die eine Sprache vollständig beschreiben, werden im ersten Schritt von der PGE erkannt und zu einem "parse tree" verwoben, einer Baumstruktur, die die logische Verschachtelung der rules wiedergibt. So etwas ähnliches machen Browser, wenn sie die verschachtelten HTML-Tags in einen DOM-Baum umwandeln an dessen Ästen und Zweigen die Daten der einzelnen Tabellen und Bilder "hängen", bevor daraus eine Webseite gerendert wird. Das ganze Javascript und die CSS-Anweisungen lassen sich so wesentlich einfacher auf die HTML-Struktur anwenden, wenn dem Programm anhand des DOM-Baumes die inaltlichen Zusammenhänge der Seite offen liegen. Genauso lassen sich die Rules wesentlich einfacher in PIR umwandeln, wenn man es auf relativ einfache Veränderungen an den Verzweigungen und Endknoten eines "parse tree" reduzieren kann. Und genau dafür wurde die TGE entwicklet, die in 3 Durchläufen diesen Baum durchwandert und jedesmal nach einem anderen Satz von Regeln, den Inhalt aller Knoten umformt, bis daraus PIR entsteht, der von Parrot kompiliert und danach ausgeführt werden kann. Warum nicht auf einmal, der unbedarfte Beobachter fragen? Nun, so ist es weit weniger fehleranfällig TGE bleibt einfach und wiederverwendbar. Außerdem sind die von der TGE erzeugten Zwischenformen, PAST und POST genannt, sehr nützlich für die Zusammenarbeit mit anderen Werkzeugen, man könnte auch nachträglich hier Optimierer für PAST oder POST zwischenschalten.
Schließlich ist es auch nicht wichtig wie lange oder in wieviel Zwischenschritten ein Parser kompiliert. Wichtig ist nur, wie schnell er laufen wird oder ob er überhaupt fertig wird. Programmierer, die freiwillig an Parrot schreiben, brauchen auch ihre sichtbaren Erfolge und hiermit kann man Parrot-Parser in Rekordzeit schreiben. Deswegen stellt man gerade alle aktiven Parser auf PGE um. Sogar Larry hat anfang 2007 damit begonnen Perl6 Spielregeln in "rules" zu formulieren. Dadurch könnten wir bald einen funktionsfähigen und schnellen Perl6-Interpreter bekommen, auch wenn die Sprache sich noch ändern wird.

Perl 6 User's Golfing System

Keinen Interpreter für Perl6 zu haben, war bis jetzt eine große Hürde für die an Perl6 Interessierten. Diese hatten sich seitenweise durch Synopsen zu lesen um irgendwann einmal, vielleicht, danach programmieren zu können. Deswegen hat die wunderbare Audrey Tang im Februar 2005, als die allgemeine Stimmung wirklich an einem

Tiefpunkt angelangt war, damit begonnen, mal schnell einen Interpreter in Haskell zu schreiben, den sie "Perl6 User's Golfing System" nannte, kurz Pugs, was zu deutsch Mops heißt, aber eher eine Anspielung an den Haskell-Interpreter HUGS ist. Dieses "golfen" ist eine Sportart, in der das kürzeste Programm gewinnt, womit Audrey wohl andeuten wollte, daß Pugs nicht dafür gedacht ist, große Anwendungen damit laufen zu lassen. Dazu ist Haskell auch viel zu langsam und speicherhungrig.
Auch wenn Pugs mit dem Glasgow Haskell Compiler (GHC) kompiliert wird, ist das Ergebnis immernoch wesentlich langsamer und aufgeblähter, als man es von perl gewohnt ist. Dazu kommt, daß es den GHC nur für die größten 3 Betriebssysteme gibt. Der Vorteil an Haskell ist jedoch, daß es sehr abstrakt ist, was ein komprimiertes also schnelles Programmieren ermöglicht. Haskell kann auch bereits viel vom Perl6 Funktionsumfang, so daß viele Perl6-Funktionen 1:1 mit Haskell-Befehlen simuliert werden können. Selbst die bereits erläuterten "rules" gibt es schon im Haskell-Land in form der "Parsec"-Bibliothek. Deswegen konnte bereits nach wenigen Wochen ein Interpreter entstehen, der den grundlegenden Befehlssatz verstand. Dies lockte viele Interessierte an, die damit experimentieren wollten. Außerdem kannte Audrey durch ihre vielen Projekte die halbe (Perl-)Welt und fing an, viele Köpfe für Haskell und die Mitarbeit an Pugs zu begeistern. Selbst $larry lernte so Haskell besser kennen, was sicherstellt, daß Perl6 wirklich alle coolen Ideen aus Haskell absorbiert und sie in einer Form anbietet die mehr Spielraum für Individualität bietet (TIMTOWTDI). Pugs jedenfalls gewann sehr schnell Zuspruch und Helfer und wuchs unkontrolliert in alle Richtungen, da Audrey jeden mitmachen ließ der es wollte und jeden auch ermutigte eigene Ideen auszuprobieren. Misses Tang nennt das "-OFun", übersetzt: oberstes Ziel ist die Optimierung der Freude an der Sache, was nicht nur die Stimmung insgesamt aufhellte, sondern auch wunderbar Entwickler dazu motivierte, Quellcode zu schreiben, aber auch Dokumentation, Beispiele und Tests.
Mit seinen mehreren tausend, mitlerweile nach Perl portierten, Testfällen beherbergt Pugs die offizielle Per6-Testsuite. Dies wird auch der Entwicklung des Perl6-Parsers unter Parrot sehr helfen. Aber im Moment hilft Pugs vor allem den Leuten im IRC-Kanal auf irc://irc.freenode.net/perl6. Dort steht seit ungefähr einem Jahr der chatbot namens evalbot bereit, der mit der neueste Version von Pugs verbunden ist. An ihn gesandte Nachrichten werden als Perl6-Quellcode angenommen, ausgeführt und das Ergebnis wieder im IRC ausgegeben. Dies kürzt nicht nur darüber ab, was Perl6 in gegebenen Situationen tun würde, sondern hilft auch beim Auffinden von Grenzfällen, an die niemand vorher dachte. Somit unterstützt die Entwicklung der Synopsen wesentlich aber auch Pugs-Fehler werden so immer wieder schnell gefunden. Wer auch einmal derart Pugs spielen möchte, ohne es sich gleich installieren zu müssen, besucht die Internetadresse http://run.pugscode.org/.

Mit der dort im Browser laufenden Pugs-Shell kann man Perl6 querfeldein nach eigenem Belieben entdecken. Sollte dies gefallen, führt der nächste Weg nach http://rakudo.org/perl6/, wo die Installation genau beschrieben wird. Windowsbenutzer haben sogar den Vorteil fertige Builds von http://jnthn.net/perl6/ laden zu können, aber auch Debian-Pakete werden regelmäßig geschnürt. Bereiche wie die threads, Metaprogrammierung mit Macros und einige Feature der rules und werden von Pugs noch nicht abgedeckt, aber ansonst ist es wirklich schon einsetzbar, es wurden ja z.B. bereits Sudokulöser, Chatbots und eine einfache Wiki in Perl6 für Pugs geschrieben. Aber Pugs kann noch sehr viel mehr. Vorhin schrieb ich, daß sich Pugs unkontrolliert in alle Richtungen entwicklet und das meinte ich wörtlich.

Das Repository beinhaltet viele Unterprojekte wie z.B. Perl6 Übersetzer nach Javascript oder anderen Sprachen und einen Interpreter namens Mini-Perl der etwas Perl6 mit hilfe eines Perl5-Interpreters ausführen kann. Dieses "Mini-Perl" von Flavio S. Glock erlebt gerade seine dritte Auflage, wobei jede Version natürlich mächtiger als der Vorgänger ist. Es ist auch bereits möglich Pugs zusammen mit perl zu kompilieren was die Ausführung beliebiger Perlmodule innerhalb von Pugs-Programmen erlaubt. Auch Larry hat seine, auf einer früheren OSCON angekündigte, Drohung wahr gemacht und ca. Herbst 2006 damit begonnen einen Überstzer von Perl5 nach Perl6 in Haskell zu schreiben, der sich MAD nennt und im Pugs-Repository unter misc/pX/Common/P5_to_P6_Translation zu finden ist aber direkt aus dem alten Perl-Interpreter heraus aufgerufen werden soll. Es gibt auch Bestrebungen von Pugs aus PIR zu erzeugen und Pugs direkt als Parrot-Parser zu verwenden.

Wie man sieht, ist Pugs ein sehr buntes und interessantes Projekt, daß viel für Perl6 geleistet hat und in keinster Weise Parrot schadet wie manche behaupten, da es in den meisten Fällen andere Menschen anzieht als Parrot oder perl. Pugs-Entwickler neigen eher zu Experimenten und sind zu einer bedeutenden Keimzelle vieler

Entwicklungen geworden. Nicht zuletzt ist die wechselseitige Unterstützung von Pugs und Parrot die Ursache für den weitverbreiteten Optimismus bei den Entwicklern.

und was ist mit dem CPAN?

Wie von Larry angekündigt, wurde die Sprache überarbeitet (konkretes dazu in der nächsten Ausgabe, im Teil 2), neue Interpreter entstehen (Pugs, Parrot) und ich erwähnte kulturelle Neuerungen innerhalb des Pugs-Projektes. Aber was ist mit dem CPAN? Über die Jahre hat es seine schlichte Grundstruktur bewahrt



und seine Reichhaltigkeit an Modulen ist eine tragende Säule von Perl's Beliebtheit. Um das nicht zu stören, waren alle Erweiterungen der letzten Jahre eher angheftet.

Die automatisierten Tests auf verschiednen Plattformen (CPANTS), die Überprüfung der Kwalitee (Einhaltung formaler Standarts), das Phalanx-Projekt zur Verbesserung der wichtigsten Module, die Archivierung aller alten Versionen und verschiedene Möglichkeiten der Kommentierung der Dokumentation ließen sich auch auf diese Weise noch ganz gut hinzufügen. Aber mit Perl6 kommen neue Aufgaben auf das CPAN hinzu, die einen gründlichen Neuentwurf aller Regeln fordern. Perlprogramme werden auch binär, also kompiliert, sein können. Das bisherige CPAN ist textbasiert. Das Archiv sollte für viele Sprachen offen sein, da Parrot viele Sprachen ausführen wird. Und eine dritter Punkt ist die Entwicklung von Perl selbst. Heute werden größere Softwareprojekte denn je in Perl geschrieben, was auch wesentlich höhere Anforderungen an sein Archiv stellt. Leider haben diese Aufgaben bisher wenig Aufmerksamkeit bekommen. Erst im November 2005 haben Audrey und Jos Boumans einige Gedanken aufgeschrieben die als Synopse 22 (S22) archiviert sind.
Florian Ragwitz brachte hierzu seine Erfahrung als Paketbinder bei Debian ein, doch seit dem tat sich dort nichts und die S22 ist immernoch als Entwurf (Draft) markiert. Ganz woanders, 2006 auf der YAPC::EU in Birmingham, gab Mark Overmeer sein Vorhaben namens CPAN6 bekannt und verfolgt mit seinen Mitstreitern seit dem stetig seine Ziele. Da sich das CPAN6 am Ende seiner Planungsphase befindet und die Software noch in Entwicklung ist, möchte ich kurz nur auf die Ziele dieses Ansatzes vorstellen, die über das bereits Erwähnte hinausgehen.

Markov stellt sich ein heterogenes Netz, sich selbst verwaltender Distributionen vor, daß sich zum Verteilen aller möglichen Daten eignet und den Adminis weniger Arbeit aufbürded, als das jetzige CPAN. Das soll z.B. dadurch erreicht werden, daß Projektleiter selbständig ihre Hoheit an einen Nachfolger abgeben können, was bisher manuell geschehen mußte. Auch einzelne Rollen innerhalb der Projekte sollen klarer verteilt sein und können und können vom den Mitgliedern dieses Projektes selber definiert und verändert werden. Die Implementation ist komplexer gestaltet und erlaubt eine Kooperation mit dem Verteilungssystem der Platform was vor allem bei BSD und Linux-Systemen wie Debian, Suse oder Gentoo zu Konflikten geführt hat.
Auch ist geplant die Installation überschaubarer zu halten: Lizenzänderungen, Speicherverbrauch, (mit allen Abhängigkeiten) und andere Anforderungen werden bei der Installation erkennbar.
Verschiedene Versionen eines Moduls können hiermit installiert werden und auch verschiedene Versionen und Arten von Parsern oder Interpretern sollen von Programmen bei ihrer Installation eingefordert werden können. Netzwerk- oder firmenweite Installationen sind mit Subrepositories als automatisierbar eingeplant und verschlüsselte Verbindungen und Prüfsummen sollen das CPAN6 weniger Anfällig für Angriffe und Mißbrauch als das CPAN machen. Eine direkte Synchronisation mit dem CVS oder SVN der Entwickler mag auch mehr Klarheit über die erhaltene oder gewünschte Versionsnummer geben, deren Schema hier mehr Freiheit gegeben wird.
Unabhängig davon können einzelne Versionen als stabil oder aktuell markiert werden aber alle anderen Versionen bleiben zugreifbar und werden automatisch archiviert. Dies und vieles mehr umfasst das Konzept. Aber noch ist das CPAN6 nicht der offizielle Nachfolger und wer Ideen und Antrieb hat es besser zu machen, wird sicher mit offenen Armen empfangen. Hiermit schließe ich meinen Rundgang durch die aufregende neue Welt des Projektes Perl6 und empfehle euch ernsthaft, keine Scheu davor zu haben Fragen zu stellen oder auszuprobieren, was euer Interesse geweckt hat. Alle Beteiligten sind ganz normale Menschen die das auch zeigen und mir fast immer mit meinen Fragen geholfen haben. Auch dieses angenehme Klima bewog mich mein eignes kleines Perl6-Vorhaben auszurufen: das offizielle Perl6-Tutorial, das ihr im Pugs-SVN finden werdet oder auch in der Wiki auf pugscode.org.

Herbert Breunung

Neues von CPAN

- 6 Module im Schnelldurchlauf

Wie in jeder Ausgabe werden an dieser Stelle sechs Module vorgestellt, die neu auf CPAN sind oder eine neue Version online haben.

mocked

Mocked ist dafür gedacht, in Tests ein Pseudo-Modul zu laden. In Tests ist es notwendig, Grenzfälle zu testen. Oft müssen diese Grenzfälle mit Modulen, die das Verhalten des Originals etwas abändern, erzeugt werden.

```
use mocked 'LWP::Simple';
my $text = get($url);
```

Hier wird ein eigenes Modul "LWP::Simple" geladen, das im Verzeichnis `t/lib/` liegt.



Image::Info

Welche Größe hat ein Bild? Welcher Bildtyp ist es? Diese und noch ein paar andere und einfache Fragen kann das Modul `Image::Info` beantworten.

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use Config::Any;
 6
 7 my $file = 'test.ini';
 8
 9 my $cfg = Config::Any->load_files({
10                 files => [$file],
11                 use_ext => 1,});
12
13 my ($c_file,$config) = each %{$cfg->[0]};
14 for my $type( keys %{$config->{Perl_Medien}} ){
15     my $value = $config->{Perl_Medien}->{$type};
16     print "$type: $value\n";
17 }
```

Listing 1

Für das Coverbild, das auf http://perl-magazin.de für die 1. Ausgabe zu finden ist, gibt das Programm

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use Image::Info qw(image_info);
 6
 7 my $file = 'Issue1.png';
 8 my $info = image_info($file);
 9 print $info->{height} ,
10       ' x ' ,
11       $info->{width};
```

folgendes aus:

```
283 x 200
```

Config::Any

Viele Programme brauchen eine Konfiguration und wer diese aus dem Programm ausgliedern will, benötigt eine Config-Datei. Doch welches Format ist das richtige? Was, wenn das Format umgestellt werden soll? Eine Möglichkeit, um größere Code-Anpassungen zu verhindern ist `Config::Any`. Auch wenn mehrere Konfigurationsdateien benötigt werden, die vielleicht in unterschiedlichen Formaten sind, hilft das Modul (Listing 1).
Und die passende ini-Datei:

```
[Perl_Medien]
Magazin=$foo
Netcast=Radio Perl
```

liefert als Ausgabe

```
C:\>config_any.pl
Netcast: Radio Perl
Magazin: $foo
```

Test::YAML::Meta

Dieses Modul ist etwas für Modulautoren. Damit kann die META.yml-Datei der Distribution überprüft werden. Wer das benutzt kann die Kwalitee der Distributionen hoch halten.

In der META.yml-Datei sind einige Informationen über das Modul gespeichert, zum Beispiel der Autor, die Version und der Name.

`Test::YAML::Meta` überprüft, ob die .yml-Datei im richtigen Format ist.

Ein einfaches Test-Programm sieht so aus:

```
1 use Test::More;
2 eval "use Test::YAML::Meta";
3 plan skip_all =>
  "Test::YAML::Meta required for
  testing META.yml" if $@;
4 meta_yaml_ok()
```

Es gibt noch eine weitere Funktion, mit der die Datei auf eine bestimmte Spezifikation hin überprüft werden kann.

```
1 meta_spec_ok(undef,'1.3',$msg);
```

Path::Resource

Dieses Modul existiert bisher nur als "Developer Release", aber das Modul könnte ganz nützlich sein für alle, die mit Webprogrammierung zu tun haben.
Mit diesem Modul kann man Pfade parallel für das Dateisystem und als URI pflegen.

Das kann ganz hilfreich sein, wenn eine Datei einerseits innerhalb des Skripts verarbeitet werden soll, andererseits damit als URI umgegangen werden soll (zum Beispiel für einen `redirect`).

Wenn die HTML-Dateien im Dateisystem unter /var/web188/htdocs/ und über http://www.domain.tld/ zu erreichen sind.

```
 1 use Path::Resource;
 2
 3 my $rsc = Path::Resource->new(
 4    dir => "/var/web188/htdocs/",
 5    uri => "http://www.domain.tld";
 6 );
 7
 8 my $filersc =
 9    $rsc->subfile("info.html");

10 # /var/web188/htdocs/info.html
11 if( -M $filersc->file < 10 ){
12 # http://www.domain.tld/info.html
13   redirect( $filersc );
14 }
```

XML::RSS::FromHTML::Simple

Nicht alle Webseiten bieten einen RSS-Feed für die neuesten Einträge an. Mit diesem Modul kann man sich recht einfach selbst ein solches RSS-Feed bauen.

Das Modul lädt eine HTML-Seite und man kann einen Bereich bestimmen, dessen Links betrachtet werden sollen. Und aus diesen Links wird dann ein RSS-Feed erstellt.

So kann einfach kann ein RSS-Feed für die News von "$foo" generiert werden:

```
 1 use XML::RSS::FromHTML::Simple;
 2
 3 use constant RSS =>
 4    'XML::RSS::FromHTML::Simple';
 5
 6 my $url = 'http://perl-
       magazin.de/?action=news';
 7 my $proc = RSS->new({
 8     url      => $url,
 9     rss_file => "news.xml",
10 });
11
12 $proc->link_filter( sub {
13     my($link, $text) = @_;
14     my $return = 0;
15     if( $link=~ m!news_detail! ){
16         $return = 1;
17     }
18     return $return;
19 });
20
21 $proc->make_rss()
22        or die $proc->error();
```

Renée Bäcker



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

Catalyst - Wir bauen uns eine FAQ

Webanwendungen leicht gemacht. Catalyst ist ein Framework zum Erstellen von dynamischen Webapplikationen.

Wer häufiger mit der Entwicklung von Webanwendungen zu tun hat, holt sich gerne Unterstützung von der Programmiersprache, in der die Anwendung geschrieben wird. Das Thema "Web-Frameworks" dürfte mit "Ruby on Rails" einen seiner Höhepunkte erlebt haben.

Aber auch in Perl gibt es Hilfe. Neben dem etwas älteren Maypole gibt es das ganz neue Jifty. Das bekannteste Web-Framework im Perl-Umfeld dürfte aber Catalyst sein.

Das Framework geht dabei nach dem Model-View-Controller (MVC)-Schema vor. In diesem Artikel soll anhand einer einfachen Anwendung gezeigt werden, wie man mit Catalyst eine Web-Anwendung erstellen kann. Hier wird eine FAQ-Liste erstellt, in die jeder Nutzer etwas eintragen kann. Um es einfach zu halten, wird auf Authentisierung etc. verzichtet - das wird in einer der nächsten Ausgabe ausführlicher behandelt.

Installation

Die Installation von Catalyst kann ein wenig frickelig sein. Für diesen Artikel habe wurde Catalyst auf drei Maschinen installiert. Zweimal auf Windows und einmal auf RedHat. Bei Windows ging es mit PPM und `Bundle::Catalyst` sehr gut, die manuelle Installation war dagegen ziemlich umständlich. Bei RedHat hat es mit Perl 5.9.4 nicht so richtig hingehauen.

Bei Shadowcat Systems gibt es ein Installations-Skript [1], mit dem es ganz gut funktioniert.

Erzeugen des Rumpfes

Nach der Installation von Catalyst startet man mit dem ersten Befehl:

```
~1>catalyst.pl PerlFAQ
```

Dadurch werden schon einige Basisdateien angelegt wie in Listing 1 zu sehen ist.

Catalyst und MVC

Catalyst trennt die drei Schichten Model, View und Controller.

Für die Datenspeicherung (Model) werden einige Module wie DBIx::Class und Net::LDAP unterstützt.

Mit View sind die HTML-Ausgaben oder RSS-Feeds gemeint. Auch hier werden mehrere Module unterstützt. So kann zum Beispiel HTML::Template::Compiled oder Template-Toolkit verwendet werden.

Die Controller werden von den Programmierern mit Leben gefüllt - sie enthalten die Programmierlogik. Diese können mit mod_perl, POE, FastCGI, CGI oder auch mit dem mitgelieferten Server ausgeführt werden.

Wenn diese Dateien angelegt sind, besteht schon ein Grundgerüst für die Anwendung.
Für Tests ist auch gleich ein eigener Server dabei, der unter PerlFAQ/script/perlfaq_server.pl gespeichert ist. Wenn dieser gestartet ist, kann im Browser unter http://localhost:3000/ eine Standardseite angeschaut werden. Während der Entwicklung ist es ganz praktisch, den Server mit der Option `--restart` zu starten. Damit wird der Server angewiesen, neu zu starten falls sich bei den Modulen oder an der Konfiguration etwas geändert hat.

Zu den wichtigsten Dateien des Grundgerüsts zählen die Konfigurationsdatei `perlfaq.yml` und `PerlFAQ::Controller::Root`.
`Root.pm` lädt benötigte Plug-Ins und erweitert gegebenenfalls die Catalyst-Engine.
Das Modul `PerlFAQ` konfiguriert die Anwendung und startet sie. Hier wird in diesem Beispiel festgelegt, wo die Datenbank liegt. Normalerweise sollte so etwas in der YAML-Datei konfiguriert werden, aber hier wird es der Einfachheit halber direkt in dem Modul festgelegt.

Eingangsseite anpassen

Wenn der Benutzer auf die Webseite kommt, soll er mehr sehen als nur die Catalyst-Willkommensseite.

HTML::Template::Compiled

Da für die Beispielanwendung `HTML::Template::Compiled` verwendet werden soll, muss noch

```
created "PerlFAQ"
created "PerlFAQ\script"
..
created "PerlFAQ\lib\PerlFAQ\Model"
created "PerlFAQ\lib\PerlFAQ\View"
created "PerlFAQ\lib\PerlFAQ\Controller"
created "PerlFAQ\perlfaq.yml"
created "PerlFAQ\lib\PerlFAQ.pm"
created "PerlFAQ\lib\PerlFAQ\Controller\Root.pm"
created "PerlFAQ/README"
created "PerlFAQ/Changes"
...
created "PerlFAQ\script/perlfaq_create.pl"
```

Listing 1

das View-Modul Catalyst::View::HTML::Template::Compiled installiert werden.

Danach kann ein neuer View erzeugt werden. Auch dafür bietet Catalyst Unterstützung:
Unter `PerlFAQ/script` liegt `perlfaq_create.pl`, das bei der Erstellung der verschiedenen Komponenten hilft.

Mit

```
perlfaq_create.pl view
HTC
HTML::Template::Compiled
```

wird ein View-Modul namens `HTC.pm` erzeugt. Und es wird `HTML::Template::Compiled` als Template-Engine verwendet.

Das Template - für die Startseite eine einfach HTML-Datei - wird unter PerlFAQ/root/ abgelegt.

Controller anpassen

Als nächstes muss der Controller angepasst werden. Die Datei, die geändert werden muss, liegt unter PerlFAQ/lib/PerlFAQ/Controller/ und heißt `Root.pm`.

In der Datei ist zu sehen, dass es eine abgeleitete Klasse von `Catalyst::Controller` ist. Und die Subroutine, die für die Anzeige der Startseite ist, heißt `default`.

In der Subroutine wird einfach der Name des Templates in den Stash eingetragen und dann ein forward auf das HTC-Modul gemacht. Nach der Änderung sieht die default-Subroutine wie folgt aus:

```
sub default : Private{
    my ( $self, $c ) = @_;
    $c->stash->{template}
            = 'index.tmpl';
    $c->forward(
   'PerlFAQ::View::HTC' );
}
```

Alle notwendigen Information für die Templates werden über den sogenannten stash ausgetauscht.

Datenbank anbinden

Jetzt soll die Anwendung aber nicht so bleiben. Es soll eine Wissenssammlung werden. Das Wissen soll in einer Datenbank gespeichert werden, die aus zwei Tabellen besteht (Listing 2)

Diese zwei Tabellen müssen in ein Schema gepasst werden. Dazu wird bei `Catalyst` das Modul `DBIx::Class` verwendet. In der ersten Ausgabe von $foo wurde `DBIx::Class` vorgestellt.

Auch für diesen Teil der Anwendung - der Model-Schicht - gibt es Hilfe von Catalyst. Die Dateien werden unter PerlFAQ/lib/PerlFAQ/Model gespeichert.

Die Anbindung der Datenbank geschieht über die folgenden vier Klassen:

- PerlFAQ::Model::DBIC

Diese Klasse (Listing 3) sorgt für den Verbindungsaufbau zur Datenbank und legt fest, welche Klasse für das Schema verantwortlich ist. Das Modul `PerlFAQ` wird hier geladen, um an den Pfad zur Datenbank zu kommen.

Namensräume und Attribute

Die Controller bestimmen den Namensraum, in dem die Methoden ansprechbar sind. Das bedeutet, dass der URL-Pfad zum Controller "User" mit "/user" anfängt.

Funktionsattribute bestimmen das Verhalten beim Aufruf einer Methode. ": Private" ist - wie der Name schon sagt - privat und für interne und/oder spezielle Aufrufe zuständig. Als "normal" lässt sich ": Local" bezeichnen, welches einfach nur den Methodennamen an den Controller-Pfad anhängt.

"default" spielt dabei eine besondere Rolle. Diese Subroutine wird aufgerufen, wenn keine Methode des Controllers angegeben ist. Wenn der Pfad genau der Klasse entspricht wird die "index"-Subroutine aufgerufen - wenn sie existiert, ansonsten wieder "default".

```
CREATE TABLE Faq(
    Faq_ID       INT          NOT NULL PRIMARY KEY,
    Frage        VARCHAR(255) NOT NULL,
    Antwort      TEXT         NOT NULL,
    Kategorie_ID INT          NOT NULL,
);

CREATE TABLE Kategorie(
    Kategorie_ID INT          NOT NULL PRIMARY KEY,
    Kategorie_Name    VARCHAR(100) NOT NULL,
);
```

Listing 2

• PerlFAQ::Model::Database

Diese Klasse (Listing 4) gibt an, welche Klassen für die Datenbank zuständig sind und geladen werden müssen.

• PerlFAQ::Model::Database::Faq

Diese Klasse (Listing 5) repräsentiert die "Faq"-Tabelle der Datenbank.

• P::M::Database::Kategorie

Diese Klasse (Listing 6) stellt die Tabelle "Kategorie" dar.

Die Datenbank wurde vorher mit einem Helferscript erzeugt und eine erste Kategorie wurde auch schon angelegt. Bevor weiter an dem Controller geschrieben wird, wird noch ein Template erzeugt, das die Kategorien auflistet. Auch dieses Template wird unter PerlFAQ/root/ gespeichert.

Nachdem das Template existiert, wird der Controller entsprechend angepasst.
Wenn http://localhost:3000/faq/ aufgerufen wird, soll die Auflistung der Kategorien gezeigt werden. Hinter jeder Kategorie ist ein Link, über den man später zu den Fragen kommt. Doch das wird später gemacht. Um die Liste anzuzeigen, wird in dem Controller "FAQ" die Subroutine `index` geschrieben. Dort wird die Datenbank-Abfrage gemacht und der Stash für das Template gefüllt. Zum Schluss wird die View-Klasse "HTC" aufgerufen (Listing 7).

Wenn das gemacht ist, kann man sich im Browser schon mal die kleine Liste der Kategorien anschauen (Abbildung 1).



Abbildung 1

Übersicht der Fragen

Als nächstes soll die Liste der Fragen der einzelnen Kategorien implementiert werden. An den Model-Klassen muss nichts mehr geändert werden. Bei den Funktionen, die jetzt noch kommen, muss immer nur die Controller-Klasse angepasst werden und ein entsprechendes Template erzeugt werden. Deswegen wird in diesem Artikel auch nur noch gezeigt, wie man bis zu den Fragen und Antworten kommt. Wie man dann noch Kategorien einfügen kann, ist eine gute Übung.

Als erstes wird für die Übersicht der Fragen wieder ein entsprechendes Template erzeugt.

Die Fragen einer Kategorie sollen unter
http://localhost:3000/faq/fragen/#
erreicht werden, wobei das # durch die ID der Kategorie zu ersetzen ist. Wie man schon aus der URL erkennen kann, muss jetzt im Controller die Subroutine `fragen` geschrieben werden (Abbildung 2).

Hier muss das Attribut "Local" gewählt werden, da der Methodenname (fragen) an den Controller-Pfad (/faq/) angehängt werden soll. Wieder werden die Template-Parameter über den Stash übergeben (siehe Listing 8).

```
 1 package PerlFAQ::Model::DBIC;
 2
 3 use base qw/Catalyst::Model::DBIC::Schema/;
 4 use PerlFAQ;
 5
 6 my $dir = PerlFAQ->config->{dbdir};
 7
 8 __PACKAGE__->config(
 9     schema_class => 'PerlFAQ::Model::Database',
10     connect_info => [
11         "DBI:SQLite:$dir/perlfaq.db",
12     ],
13 );
14
15 1;
```

Listing 3

```
 1 package PerlFAQ::Model::Database;
 2
 3 use base qw(DBIx::Class::Schema);
 4
 5 __PACKAGE__->load_classes(qw/Faq Kategorie/);
 6
 7 1;
```

Listing 4

```
 1 package PerlFAQ::Model::Database::Faq;
 2
 3 use strict;
 4 use warnings;
 5 use base qw(DBIx::Class);
 6
 7 __PACKAGE__->load_components(qw/PK::Auto Core/);
 8 __PACKAGE__->table('Faq');
 9 __PACKAGE__->add_columns(qw/Faq_ID Frage Antwort Kategorie_ID/);
10 __PACKAGE__->set_primary_key('Faq_ID');
11 __PACKAGE__->belongs_to(Kategorie_ID => 'PerlFAQ::Database::Kategorie');
12
13 1;
```

Listing 5

```
 1 package PerlFAQ::Model::Database::Kategorie;
 2
 3 use strict;
 4 use warnings;
 5 use base qw(DBIx::Class);
 6
 7 __PACKAGE__->load_components(qw/PK::Auto Core/);
 8 __PACKAGE__->table('Kategorie');
 9 __PACKAGE__->add_columns(Kategorie_ID   => {accessor => 'Cat_ID'},
10                          Kategorie_Name => {accessor => 'Cat_Name'});
11 __PACKAGE__->set_primary_key('Kategorie_ID');
12 __PACKAGE__->has_many(Faq => 'PerlFAQ::Model::Database::Faq',
13                   { 'foreign.Kategorie_ID' => 'self.Kategorie_ID' });
14
15 1;
```

Listing 6



Abbildung 1

Fazit

Catalyst ist ein sehr flexibles Framework, das eine gute Hilfe bei der Erstellung von Webapplikationen ist. Für Applikationen, die auf Shared Servern bei einem Hoster laufen sollen, dürfte es aber auf Grund der benötigten Module eher ungeeignet sein.
Wenn die Applikation aber auf einem Server laufen sollen, der selbst verwaltet wird, bietet Catalyst ein mächtiges Werkzeug. Die Anwendungen können dann auch problemlos unter mod_perl 1&2 laufen.

Renée Bäcker

Frage und Antwort

Als letztes soll nun die Seite gezeigt werden, auf der Frage und Antwort zu sehen sind. Diese wird unter
http://localhost:3000/faq/antwort/#
zu sehen sein (Abbildung 3). # steht hier für die ID der Frage. Auch hier wird die Methode im Controller "FAQ" implementiert und das Attribut ist wieder "Local".

Bevor die Seite aufrufbar ist, muss ein entsprechendes Template erzeugt werden.

Damit wäre die kleine Demonstrationsapplikation fertiggestellt.



Abbildung 1

```
 1 # zeige alle Kategorien an
 2 sub index : Private{
 3     my ( $self, $c ) = @_;
 4
 5     $c->stash->{template} = 'kategorien.tmpl';
 6     $c->stash->{entries} = [
 7         map{ {
 8                 kategorie => $_->Cat_Name,
 9                 id        => $_->Cat_ID,
10             } }$c->model('DBIC::Kategorie')->all
11     ];
12
13    $c->forward( 'PerlFAQ::View::HTC' );
14 }
```

Listing 7

```
 1 sub fragen : Local{
 2     my ( $self, $c, $id ) = @_;
 3
 4     $c->stash->{template}  = 'fragen.tmpl';
 5     $c->stash->{kategorie} =
 6         map{ $_->Cat_Name }$c->model('DBIC::Kategorie')->search(
 7             {
 8                 Kategorie_ID => $id
 9             })->all;
10     $c->stash->{entries}   = [
11         map{ {
12                 id    => $id,
13                 fid   => $_->Faq_ID,
14                 frage => $_->Frage,
15              }
16         }$c->model('DBIC::Faq')->search({
17             Kategorie_ID => $id,
18         })->all
19     ];
20
21     $c->forward( 'PerlFAQ::View::HTC' );
22 }
```

Listing 8

```
 1 sub antwort : Local{
 2     my ( $self, $c, $id ) = @_;
 3
 4     my ($entry) = $c->model('DBIC::Faq')->search({
 5         Faq_ID => $id,
 6     })->all;
 7
 8     $c->stash->{template} = 'antwort.tmpl';
 9     $c->stash->{frage}    = $entry->Frage;
10     $c->stash->{antwort}  = $entry->Antwort;
11
12     $c->forward( 'PerlFAQ::View::HTC' );
13 }
```

Listing 9

CGI::Application

Webanwendungen schnell erstellt

Was wäre das WWW ohne dynamische Webseiten? Es wäre wohl nicht eines der wichtigsten Medien unserer Zeit.

Es gibt viele Möglichkeiten, eine Webapplikation zu programmieren - also ganz nach dem Perl-Motto TIMTOWTDI ("There is more than one way to do it").
Die Applikation kann von Grund auf selbst geschrieben werden oder es wird eines der Frameworks zu Hilfe genommen. `CGI::Application` ist ein sehr leichtgewichtiges Framework, das vielleicht nicht ganz so stark unterstützend wirkt wie zum Beispiel `Catalyst`, aber es ist häufig völlig ausreichend.

Es bietet ein Grundgerüst, an dem man sich gut entlanghangeln kann und es sorgt dafür, dass zumindest eine Grundstruktur in der Anwendung ist. Eine Webanwendung, die mit `CGI::Application` entwickelt wurde, durchläuft einige Phasen. Als erstes wird die init-Phase durchlaufen, dann die prerun bevor die setup-Phase an der Reihe ist. Nachdem die runmodes durchlaufen sind, wird die postrun-Methode aufgerufen.

Es müssen nicht alle Phasen implementiert werden, aber es kann viel Arbeit sparen.

```
1 #!/usr/bin/perl
2
3 use strict;
4 use warnings;
5 use Kalender;
6
7 my $fk = Kalender->new();
8 $fk->run();
```

Listing 1

```
1 package FooKalender;
2
3 use strict;
4 use warnings;
5 use base qw(CGI::Application);
```

Listing 2

Die ganze Logik der Anwendung wird in Module gepackt, so dass das Skript, das der Besucher der Seiter aufruft, nur wenige Zeilen lang ist (Listing 1).

In diesem Artikel wird als Beispiel ein kleiner Terminkalender programmiert.
Da es nur eine Demonstration ist, wurde auf einige Sicherheitsaspekte verzichtet. So ist es jedem Nutzer erlaubt, Termine einzutragen und zu löschen - ganz ohne Authentifikation.

Weiterhin soll es möglich sein, Termine zu bearbeiten und weitere Monate anzeigen zu lassen.

Hauptmodul

Als erstes wird ein Modul erzeugt, das mehr oder weniger die gesamte Logik enthält.
Wie vorhin beschrieben, wird als erste Methode `setup` aufgerufen.

Wenn Code noch **vor** der `setup`-Methode ausgeführt werden soll, muss eine `cgiapp_init`-Methode implementiert werden.

Das Modul muss eine Subklasse von `CGI::Application` sein (Listing 2).

zusätzliche Module

Für eine bessere Struktur werden zwei zusätzliche Module geschrieben, die hier nicht gezeigt werden, aber in den Quellcodes auf der Webseite
http://www.foo-magazin.de
zu finden sind. Zum einen ist dies ein Modul für die Datenbankabfragen und ein zweites Modul, das den Kalender erzeugt.

Diese zwei Module werden mit

```
6 use Kalender::Month;
7 use Kalender::DB;
```

geladen.

setup

In der `setup`-Methode (Listing 3) werden alle wichtigen Sachen der Anwendung konfiguriert:
Welche Aktionen (`runmodes`) gibt es? Die Verbindung zur Datenbank wird hergestellt. Welche Aktion wird standardmäßig ausgeführt?

`CGI::Application` reicht das Objekt an die `setup`-Methode weiter.

Die Methoden `template`, `action`, `db` und `kalender` sind einfache getter/setter-Methoden, um Attribute zu setzen oder auszulesen.

Von `CGI::Application` stammen dagegen die Methoden `start_mode`, `mode_param` und `run_modes`. Mit `start_mode` wird festgelegt, welche Subroutine ausgeführt werden soll, wenn gar kein runmode angegeben ist. Wird im Browser hier also einfach nur http://localhost/index.cgi aufgerufen, wird die Subroutine `show_today` ausgeführt.

```
 1 sub setup{
 2   my ($self)  = @_;
 3
 4   $self->template( './kalender.tmpl' );
 5   $self->action( '/index.cgi' );
 6   $self->db( Kalender::DB->new() );
 7   $self->kalender( Kalender::Month->new() );
 8
 9   $self->start_mode( 'show_today' );
10   $self->mode_param( 'action' );
11
12   $self->run_modes(AUTOLOAD   => \&show_today,
13                    day        => \&show_day,h,);
14                    insert     => \&insert_date,
15                    show       => \&show_date,
16                    edit       => \&edit_date,
17                    update     => \&update_date,
18                    add        => \&add_date,
19                    delete     => \&delete_date,
20                    month      => \&show_dmont
21  }
```

Listing 3

Mit `mode_param` legt man fest, welcher Parameter den runmode bestimmt. Wenn hier also http://localhost/index.cgi?action=add aufgerufen wird, dann wird der runmode `add` ausgeführt.

`run_modes` legt fest, welche runmodes es geben soll. Ein besonderer runmode ist `AUTOLOAD`. Damit wird festgelegt, welche Subroutine ausgeführt wird, wenn ein falscher Wert für den runmode-Parameter übergeben wird. Wenn http://localhost/index.cgi?action=foo aufgerufen wird, wird die Subroutine `show_today` ausgeführt.

cgiapp_prerun

In der nächsten Phase wird `cgiapp_prerun` ausgeführt. Diese Methode ist dafür gedacht, Code auszuführen, der für viele oder sogar alle Aktionen gleich ist. Also zum Beispiel die Überprüfung von Session-Daten oder in diesem Beispiel die Überprüfung, ob das Datum im korrekten Format ist.

In der `prerun`-Phase kann auch die auszuführende Aktion geändert werden. In diesem Beispiel ist es so, dass bei einem Fehler einfach eine Warung ausgegeben werden soll. `get_current_runmode` liefert den runmode zurück, der gestartet werden soll.
`query` liefert das `CGI`-Objekt, so dass man die Methoden von `CGI.pm` nutzen kann.

```
 1 sub cgiapp_prerun{
 2   my ($self)   = @_;
 3
 4   my $rm       = $self->get_current_runmode;
 5   if( $rm eq 'day' or
 6      $rm eq 'insert' or
 7      $rm eq 'update' ){
 8
 9      my %params = $self->query->Vars;
10      if($params{day} >= 1 && $params{day} <= 31){
11         warn "Tagesangabe ist in Ordnung\n";
12      }
13   }
14 }
```

Listing 4

Aktionen

Die Subroutinen die im Dispatcher (die `run_modes`-Methode, die in der `setup`-Methode aufgerufen wird) festgelegt sind, sind dann die Aktionen der Webanwendung. In dieser Kalender-Applikation haben wir acht Aktionen.

Als Beispiel wird hier die Subroutine `show_date` gezeigt (Listing 5).

Jeder runmode bekommt das Objekt der Applikation übergeben. Über dieses Objekt kommt man auch an das `CGI`-Objekt und kann alle Methoden der Klasse nutzen. In Zeile 5 werden zum Beispiel in der Datenbanken die Informationen zu einem Termin abgefragt.

In den Zeilen 7 und 8 wird der Hash aufgebaut, der alle Parameter für das Template enthält.
Die `do_template`-Methode lädt das Template und macht die Ersetzungen.

Wie man erkennt, wird in dem Modul nie etwas direkt ausgegeben. Die Ausgabe - sowohl

Header als auch Inhalt - wird von CGI::Application übernommen.
In den Subroutinen muss nur der Inhalt mit einem return an CGI::Application geliefert werden.

Am Schluss wird noch cgiapp_postrun ausgeführt. Diese Methode eignet sich für das "Aufräumen": Löschen von temporären Dateien, schließen von Datenbankverbindungen und ähnliches.

```
1 sub cgiapp_postrun{
2   my ($self)   = @_;
3   $self->db->end;
4 }
```

```
1 sub show_date{
2    my ($self) = @_;
3
4    my %params = $self->query->Vars;
5    my @info = $self->db->get_date( $params{id} );
6
7    my %hash;
8    @hash{qw/ID DAY MONTH YEAR TITLE DESC/} =
9         @info;
10
11   return $self->do_template(
12                    \%hash,
13                    './date.tmpl',);
14 }
```

Listing 5

Es gibt noch einige weitere Methoden von CGI::Application, die jedoch nicht ganz so häufig eingesetzt werden. Diese sind in der Dokumentation ganz gut beschrieben.

Plugins

Für CGI::Application wurden schon viele Plugins entwickelt, die das Entwickeln von größeren und sichereren Webanwendungen vereinfachen. Hier ein kleiner Überblick über ein paar wichtige Plugins in CGI::Application::Plugin::*:

HTCompiled

Standardmäßig ist die Template-Engine von CGI::Application das HTML::Template-Modul. Mit diesem Plugin wird stattdessen das Modul HTML::Template::Compiled von Tina Müller verwendet. Es gibt auch Plugins für Template-Toolkit und andere Template-Engines.

Authentication

Dieses Plugin ist ein ganzes Framework für die Authentizierung von Nutzer. Es bietet etliche Treiber - so kann die Authentizierung per htpasswd genutzt werden oder über eine Datenbank.

ActionDispatch

CGI::Application::Plugin::ActionDispatch erlaubt das Parsen von PATH_INFO mit Hilfe von Attributen:

```
1 # /city/philadelphia
2 sub city :
  Path('city/') {
3   my $self = shift;
4   my($city) =
  $self->snippets();
5   # ....
6 }
```

Session

Mit dem Session-Plugin wird die Einbindung von CGI::Session sehr einfach. Auf das Objekt von CGI::Session kann dann einfach mit

```
$self->session;
```

zugegriffen werden.

DBH

Wer DBI in seiner Anwendung direkt verwenden will und die DB-Zugriffe nicht auslagern will, kann DBI mit dem DBH-Plugin sehr einfach einbinden.

JSON

Damit kann man einfach JSON-Header erzeugen.

Pros und Cons

Ein großer Vorteil bei der Verwendung von CGI::Application gegenüber dem manuellen Erstellen der Webapplikation ist, dass man viel Arbeit von verschiedenen "Strecken" (Aktionen) vereinheitlichen kann. Das macht die Anwendung wartbarer und übersichtlicher.
Datenbankverbindungen müssen meist nicht so häufig aufgebaut werden als wenn man die Anwendung komplett selbst schreibt. Das wirkt sich meist positiv auf die Performance aus. Und wenn sich mal die Zugangsdaten ändern, so muss der Code nur an einer Stelle angefasst werden.

Der Nachteil von CGI::Application gegenüber Frameworks wie Catalyst oder Jifty ist, dass immer noch sehr viel vom Programmierer selbst gemacht werden muss. Allerdings wird für die Arbeit mit CGI::Application nicht "das halbe CPAN" benötigt.
Welches Framework man verwendet ist keine Frage, die generell beantwortet werden kann. Es hängt von der Anwendung und von anderen Rahmenbedingungen ab.

Renée Bäcker

CSI Perl

- Mysteriöse Perl-Fälle aufgeklärt

Kontext auf Abwegen

Hin und wieder gibt einem Perl echt Rätsel auf. Neulich fragte ein Kollege, warum

```
($ret) ||= foo();
```

die Funktion `foo()` in skalarem Kontext aufruft. Gute Frage!
Eigentlich ist das Konstrukt oben ja nichts anderes als

```
($ret) = $ret || foo();
```

wobei der "oder"-Operator stärker bindet als das Gleichheitszeichen. Wer die Präzedenzregeln nicht so genau im Kopf hat, darf auch

```
($ret) =
  ( $ret || foo() );
```

schreiben, um klar zu machen, was abgeht.

Doch auch die Kontextregeln für den Oder-Operator sind alles andere als offensichtlich.
Hier sind sie: Links von einem Oder-Operator herrscht **immer** skalarer Kontext. Die rechte Seite hingegen 'erbt' den Kontext der Umgebung.

Im obigen Fall ist der ererbte Kontext für `foo()` ein Listkontext, da links vom Gleichheitszeichen eine Liste steht. Stünde dort ein Skalar, wie in

```
$ret =
  ( $ret || foo() );
```

befindet sich `foo()` in skalarem Kontext. Ganz schön kompliziert, oder?

Machen wir die Probe aufs Exempel: Schreiben wir zunächst eine Funktion `foo()`, die ihren Kontext ausgibt. Sie liefert außerdem 0 zurück, damit Konstrukte wie `foo() || foo()` auch beide Aufrufe von `foo()` auslösen und Perl nicht vorschnell die Abkürzung wählt.

```
sub foo {
    print
       (wantarray ?
       "list" :
       "scalar",
       "\n");
    return 0;
}
```

Die Perl-Funktion `wantarray()` liefert einen wahren Wert zurück, wenn `foo()` im Listkontext steht.
Zurück zu unserem Test:
Steht auf der linken Seite unserer Testzuweisung eine Liste

```
($ret) =
  ( foo() || foo() );
```

dann lautet die Ausgabe

```
scalar
list
```

denn die rechte Seite des Oder-Operators 'erbt' den Listkontext von der linken Seite der Zuweisung. Steht links hingegen ein Skalar, wie in

```
$ret  =
  ( foo() || foo() );
```

dann steht `foo()` beides Mal in skalarem Kontext:

```
scalar
scalar
```

denn auch hier 'erbt' die rechte Seite des Oder-Operators den ihn umgebenden Kontext, der von der linken Seite der Zuweisung herrührt.

Warum erbt aber die rechte Seite in

```
($ret) ||= foo();
```

nicht den Listkontext, obwohl das Konstrukt, wie eingehend erläutert, beinahe gleichbedeutend mit

```
($ret) = $ret || foo();
```

ist? Eine Diskussion mit den Stammgästen auf perlmonks.com brachte die Wahrheit ans Licht:
Die beiden Konstrukte sind nicht äquivalent, da die längere Version den Ausdruck `$ret` **zweimal** evaluiert -- was zwar bei einer normalen skalaren Variablen nichts ausmacht, bei einer mit `tie()` gebundenen Variablen oder einer Funktion mit :lvalue-Attribut allerdings böse ungeahnte Nebeneffekte auslösen könnte.

Es gilt also, im Kopf zu behalten: Der ||=-Operator evaluiert seine rechte Seite **immer** in skalarem Kontext. Vorsicht damit!

Mike Schilli

Frankfurt.pm

Die Perl-Programmierer vom Main

Perlmonger-Gruppen sind eine gute Gelegenheit, andere Perl-Programmierer kennenzulernen. Manche Gruppen halten mehr technische Vorträge, andere machen mehr "Social meetings". Frankfurt.pm in Frankfurt am Main macht beides.

Wann es genau mit Frankfurt.pm angefangen hat, weiß anscheinend niemand so ganz genau. Auch auf der Webseite von Frankfurt.pm wird man im Dunkeln darüber gelassen, wann die Gruppe offiziell angemeldet wurde.

Erst ab 2001 tauchen erste Hinweise auf Treffen auf. In dem Jahr hat Sören M. Laird Sörries die Gruppe übernommen – oder, wie er selbst es beschreibt, "geputscht".
Seit dieser Zeit gibt es regelmäßige Treffen: Immer der erste Dienstag nach dem Monatsersten um 19 Uhr im Café Activ. Zu den Treffen kommen in der Regel 6 bis 8 Personen, aber an manchen Treffen geht die Teilnehmerzahl hoch bis zu 14 Personen.

Bei Bedarf wird auch mal ein zweites Treffen organisiert, zum Beispiel, wenn Besuch von anderswo in Frankfurt halt macht oder auf dem Deutschen Perl-Workshop.

Die Mitglieder von Frankfurt.pm sind eine bunte Mischung aus Selbständigen und Angestellten aus den verschiedensten Branchen. Die Telekommunikation



und Banken sind genauso vertreten wie die Nachrichtenagenturen und Printmedien.
Dementsprechend breit ist auch das Spektrum der Verwendung von Perl – von Active Directory und Anwendungsautomation über die Printaufbereitung bis zum puren Spaß mit Perl ist alles vertreten.

Die meisten Treffen sind von Gesprächen geprägt, die häufig bei Perl anfangen und bei ganz allgemeinen Themen wie "Leben und Radfahren in Münster" enden.
Vorträge gibt es unter dem Jahr eher selten, hier hält Max Maischein anscheinend ein Monopol. Zum Ausgleich gibt es traditionell im September den Frankfurter Perl-Community Workshop, an dem ein ganzes Wochenende lang Vorträge gehalten werden.

Auch sonst sind die Mitglieder von Frankfurt.pm sehr aktiv. Einige sind im Organisationsteam des Deutschen Perl-Workshops, Martin Fabiani ist verantwortlich für das große deutsche Perl-Forum unter http://perl-community.de und der kleine Frankfurter Perl-Community Workshop wird natürlich auch von den Mitgliedern organisiert.

Beim Deutschen Perl-Workshop sind die Mitglieder von Frankfurt.pm auch regelmäßig als Vortragende oder Teilnehmer dabei.

Weitere Informationen und Mailingliste unter http://frankfurt.pm.org

Renée Bäcker

PM-Ticker

++ Neue Perlmonger-Gruppe ++

Auf dem Deutschen Perl-Workshop in München wurde eine neue Perlmonger-Gruppe gegründet.

Hanover.pm (Hannover) wurde aus der Taufe gehoben. Am 1. März gab es auch das erste Treffen und dafür haben sich die Perlmonger auch gleich einen interessanten Gast gesichert: Jonathon Worthington, ist vom Workshop aus direkt mit nach Hannover gefahren und hat beim ersten Treffen einen Vortrag gehalten.

Die Webseite sollte relativ bald unter http://hanover.pm.org erreichbar sein.

YUR.tv

- Informationen aus erster Hand von Malte Ubl

Über Malte

FM: Hallo Malte, bevor wir uns über Dein aktuelles Projekt unterhalten, stell' Dich doch einfach mal vor.

MU: Auch, wenn ich hier als Perl-Entwickler, interviewt werde, bin ich im Berufsleben erst einmal Mitglied der Geschäftsleitung bei Schaffhausen ITC Solutions, einer Internetagentur aus Elmshorn, nördlich von Hamburg. Ich mache viel Kundenberatung, aber versuche immer noch so viel Code wie möglich zu schreiben. Ca. 1994 habe ich als Schüler bei einer Firma Namens Integrated Case & Concepts angefangen, die mit Smalltalk die Materialwirtschaft für Mercedes Benz gebaut haben. Ich denke der frühe Smalltalk-Einfluss hat mich sehr geprägt. Konsequenterweise sollte ich wohl ins Ruby-Camp wechseln :). Mit Perl habe ich so 1997 angefangen und dann relativ schnell, das MVC-Paradigma von Smalltalk gestohlen und damit Web-Anwendungen gebaut.

YUR.tv - das Projekt

FM: Anfang des Jahres hast Du mit YUR.tv angefangen. Was ist das?

MU: Kurz gesagt: Ein Internetfernsehsender, der auf nutzer-generiertem Content basiert.

FM: Wo liegt der große Unterschied zu Videoportalen wie YouTube.com usw.?

MU: Ich möchte an dieser Stelle noch nicht so sehr ins Detail gehen. Jeder kann sich gern unter www.yur.tv für das Beta-Programm anmelden. Das wichtigste an YUR.tv ist vielleicht, dass wir damit irgendwann auch 1:1 ins normale Fernsehen gehen können. Wir versuchen den Unterschied von Internetfernsehen und der klassischen Wohnzimmerunterhaltung zu verwischen.

FM: Welche Rolle spielt "Perl" bei dem Projekt? Und warum wurde "Perl" für dieses Projekt gewählt?

MU: Zunächst einmal ist YUR.tv mit Perl geschrieben. Wir nutzen dabei unser eigenes MVC-Framework und ORM-Tool. Internetfernsehen ist zurzeit noch Rocket-Science. Man muss viele Sachen zusammen bauen, die noch nicht so richtig zusammen funktionieren, wenn sie überhaupt funktionieren. Perl als Swiss-Army-Knife und nicht zuletzt CPAN sind für diese Aufgabe einfach perfekt geeignet.

FM: Mit welchen Problemen hast Du am meisten zu kämpfen – aus programmier-technischer Sicht?

MU: Bei YUR.tv übernimmt ein Perl-Programm viele Aufgaben, die in einem klassischen TV-Studio der Cutter übernehmen würde. Wir, das heisst meine Entwickler und ich, haben das HTTP-Request-Response-Paradigma so in unseren Kopf gebrannt, dass die Entwicklung von Applikationen die Zeit- und Eventgesteuert sind, nicht so ganz einfach von der Hand geht. Andererseits war das aber auch eine willkommene Gehirnmassage.

Die Zukunft von YUR.tv

FM: In welche Richtung soll sich "YUR.tv" entwickeln?

MU: In erster Linie soll YUR.tv DER Internetfernsehkanal werden. Da es im Augenblick noch kaum Konkurrenz gibt, sind wir guter Hoffnung, dass es klappen könnte.
Wir haben YUR.tv allerdings auch immer als R&D-Projekt gesehen. Dabei raus gekommen ist für uns ein neues AJAX-Framework, ein neues Request-Validierungsframework, Erfahrung mit Video-software, ein Content-Delivery-Network und noch mehr Sachen, die ich schon wieder vergessen habe. Selbst, wenn YUR.tv niemand gucken will, sind wir als Agentur jetzt ganz vorne im Bewegt-Bild-Internet dabei. Diese Erfahrung zahlt sich jetzt bereits mit Kundenprojekten aus.

FM: Vielen Dank, Malte, dass Du uns hier etwas über "YUR.tv" erzählt hast. Wir wünschen Dir viel Erfolg damit.

PS: Alle Perl-Entwickler können sich gern bei uns bewerben. Als Willkommensgruß bekommt jeder ein eigens an die persönlichen Fähigkeiten angepasstes Super-Helden-T-Shirt mit verstecktem Kamel aus unserer Grafik-Design-Abteilung :)

Testen mit Perl

In der ersten Ausgabe von $foo ging es um allgemeine Test-Module. In dieser Ausgabe werden Module vorgestellt, mit denen Konsolenprogramme und Web-Applikationen getestet werden können.

Testen der Dokumentation

Ein zweiter Bereich, der gerne vernachlässigt wird, ist die Dokumentation. Bei einem Großteil der Programmierer ist dieser Teil vermutlich noch unbeliebter als das Testen von Software. Doch Dokumentation ist wichtig, um Kunden einen Einblick in die Software zu gewähren. Die Dokumentation ist aber auch entscheidend für die Wartbarkeit eines Programms:
Ist das Programm nicht dokumentiert, so fällt es häufig schwer, nach drei oder vier Monaten das Programm zu erweitern. "Wie war das nochmal? Was macht der Codeteil?" - so oder so ähnliche Sätze hört man dann vor den Bildschirmen.

Um sich selbst und andere Programmierer zu etwas Disziplin zu "zwingen", erlaubt es Perl, die Dokumentation zu überprüfen. Es dreht sich hier um die Dokumentation in `POD`. In den folgenden Sektionen werden zwei Module beschrieben, die für das Testen der Dokumentation verwendet werden können.

Test::Pod

Das erste Modul zum Testen der Dokumentation ist `Test::Pod`. Es überprüft die `POD`-Dokumentation auf Korrektheit und vor allem, ob überhaupt etwas da ist! Die Dokumentation wird darauf überprüft, ob sie der Syntax entspricht, die in `perldoc perlpod` beschrieben ist.

Es wird allerdings nicht überprüft, ob das Programm vollständig dokumentiert ist.

Das Modul exportiert drei Methoden in das Testskript:

pod_file_ok

Diese Funktion wird verwendet, wenn die `POD`-Dokumentation einer einzelnen Datei überprüft werden soll.
Dementsprechend kann man der Funktion einen Dateinamen übergeben. Wie von den meisten Testfunktionen kann auch `pod_file_ok` ein Name für den Test übergeben werden.

Der Aufruf dieser Funktion sieht dann beispielsweise so aus:

```
pod_file_ok('My/Module.pm',
       'test of My::Module')
```

Hier ein Beispielskript:

```
 1 #!/usr/bin/perl
 2
 3 use Test::Pod
         tests => 1;
 4 pod_file_ok($0);
 5
 6 =pod
 7
 8 =head1 Test
 9
10 this is a test for
11 Test::Pod
12
13 =cut
```

Die Ausgabe dieses Programms sieht dann wie folgt aus:

```
C:\community>perl
  test_pod_1.pl
1..1
ok 1 - POD test for
  test_pod_1.pl
```

all_pod_files_ok

Soll die Dokumentation von mehreren Dateien überprüft werden, so kann die Funktion `all_pod_files_ok` verwendet werden. Der Funktion wird als erstes Argument eine Referenz auf ein Array übergeben, in dem die zu überprüfenden Dateien stehen.

all_pod_files

Diese Funktion liefert alle Dateien, die `POD` enthalten. Das ist ganz praktisch, wenn man nicht für jede zusätzliche Datei, die `POD` enthält, das Testskript ändern möchte.

Test::Pod::Coverage

Ein weiterer wichtiger Punkt bei den Tests ist die Überprüfung auf Vollständigkeit der Dokumentation. Denn was bringt ein Seitenlanger `POD`-Text, wenn nicht das Wichtige - die Schnittstellen und die Funktionen - dokumentiert ist. `Test::Pod::Coverage` überprüft, ob auch wirklich alle "öffentlichen" Subroutinen in der Dokumentation auftauchen.

Als "privat" wird angesehen, was mit einem _ anfängt oder eine General-Sub wie `DESTROY` ist. Man kann natürlich auch selbst definieren was "privat" ist. Die Funktionen sind nahezu identisch mit denen von `Test::Pod`, nur dass statt `files` ein `coverage` in den Funktionsnamen auftaucht.

Testen von Konsole-Applikationen

Im vorangegangenen Abschnitt wurden die Module beschrieben, die hauptsächlich bei Tests von Modulen eingesetzt werden. Die

Test-Strategie sollte jedoch weitergeführt werden und auch auf Applikationen angewendet werden. Dieser Abschnitt und die beiden folgenden Abschnitte zeigen, wie man auch Skripte und andere Applikationen testen kann.

Test::Cmd

`Test::Cmd` ist ein kleines Modul, mit dem man Konsolen-Applikationen testen kann. Die Verwendung des Moduls ist etwas gewöhnungsbedürftig, aber es ist ganz gut geeignet, um Programme zu testen.

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use Test::Cmd;
 6 use Test::More tests => 3;
 7
 8 my $test = Test::Cmd->new(prog => 'cmd_bsp.pl',
 9             workdir => './testing') or die $!;
10 $test->interpreter('perl');
11
12 for(1..3){
13     my $nr = rand 2998;
14     $test->run(stdin => $nr);
15     my $result = $test->stdout();
16     like($result,qr/>>$nr<<$/);
17 }
```

Listing 1

Ein einfaches Skript, das getestet werden soll:

```
1 #!/usr/bin/perl
2
3 use strict;
4 use warnings;
5
6 print q~Geben Sie
    eine Zahl ein: ";
7 chomp(my $zahl =
    <STDIN>);
8
9 print
   '>>',$zahl,"<<\n";
```

Es gibt einfach die Zahl, die eingegeben wurde, umrahmt von ">>" und "<<" aus.

Getestet wird es mit dem Skript in Listing 1.

Hier wird überprüft, ob das Programm tatsächlich alle möglichen Eingaben wieder ausgibt.
`Test::Cmd` bietet verschiedene Einstellmöglichkeiten zum Testen. Man kann dem zu testenden Programm auch Kommandozeilenargumente übergeben und so verschiedene Einstellungen testen.

Der Test zeigt, dass das Programm alles richtig macht:

```
~/entwicklung 28> perl
  test_cmd.pl
1..3
ok 1
ok 2
ok 3
```

Das Modul legt ein temporäres Verzeichnis an, in dem verschiedene Dateien zwischengespeichert werden.
Dort werden für die verschiedenen Durchläufe zum Beispiel die Ausgaben des getesteten Programms und die Ausgaben auf `STDERR` in Dateien gespeichert, damit diese Informationen später wieder abgerufen werden können.

```
~/entwicklung 4> ll testing
total 2
-rw-rw-r--  1 reneeb   pr   0 Oct 11 13:32 stderr.1
-rw-rw-r--  1 reneeb   pr   0 Oct 11 13:32 stderr.2
-rw-rw-r--  1 reneeb   pr 136 Oct 11 13:32 stdout.1
-rw-rw-r--  1 reneeb   pr   0 Oct 11 13:32 stdout.2
```

Listing 2

Ein solches Verzeichnis ist beispielhaft in Listing 2 dargestellt.

Bei Beendigung des Testskripts räumt das Modul wieder auf, das heißt Verzeichnis und Dateien werden gelöscht.

Das Modul geht relativ strikt vorwärts. Zum Überprüfen, ob das Skript durchgelaufen ist, oder ob es abgebrochen wurde, stehen die Funktionen `pass` beziehungsweise `fail` zur Verfügung. Das Modul liefert leider keine Funktionen, um die Ausgaben des getesteten Programms zu überprüfen.
Das muss man dann mit `Test::More` und dessen Funktionen machen (wie im Beispielskript).

Test::Expect

Ein weiteres Modul zum Testen von Konsole-Applikationen ist `Test::Expect`. Dafür ist es jedoch notwendig, dass das zu testende Programm einen Prompt hat. Dadurch erkennt das Modul, wann es Eingaben senden kann. Zusätzlich muss das Programm durch einen Befehl zu Beenden sein, da sonst das Testskript hängt, wenn noch Eingaben fehlen.
Aus diesem Grund wurde obiges Beispielskript erweitert (Listing 3).

Der Test mit `Test::Expect` ist in Listing 4 zu sehen.

Ohne die eingebaute Abfrage nach `quit` würde das Testskript stehen bleiben und darauf warten, dass es etwas an das zu testende Skript senden kann. Die Dokumentation zu `Test::Expect` ist leider sehr

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5
 6 for(0..2){
 7     print "Geben Sie eine Zahl ein: ";
 8     chomp(my $zahl = <STDIN>);
 9     if($zahl eq 'quit'){
10         exit;
11     }
12     print '>>',$zahl,"<<\n";
13     sleep 2;
14 }
```

Listing 3

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use Test::Expect;
 6 use Test::More tests => 3;
 7
 8 expect_run(
 9     command => 'perl cmd_bsp.pl',
10     prompt  => 'Geben Sie eine Zahl ein: ',
11     quit    => 'quit',
12 );
13
14 expect(3,'>>3<<');
```

Listing 4

```
~/entwicklung 31> perl test_expect.pl
1..1
ok 1 - expect_run
ok 2
ok 3
# Looks like you planned 1 test but ran 2 extra.
```

Listing 5

rudimentär. So ist an keiner Stelle vermerkt, dass man bei `Test::More` zwei Tests pro `expect`-Aufruf rechnen muss und dass `expect_run` auch einen Test beinhaltet. Sieht man das Testskript, so erwartet man erstmal nur **einen** Test: bei dem `expect`-Aufruf.

Die Ausgabe ist in Listing 5 aufgeführt.

An der Ausgabe kann man dann erkennen, dass `expect_run` einen Test beinhaltet. Ein Blick in den Quellcode des Moduls offenbart dann auch, wo die zwei zusätzlichen Tests herkommen: `expect` ruft intern zwei Funktionen auf, die jeweils einen Test machen.

Zwischenfazit "Konsole-Applikationen"

Mit beiden Modulen lassen sich Konsolen-Anwendungen testen. Allerdings ist die Verwendung der Module nicht ganz so intuitiv wie bei anderen Test-Modulen. Das sollte aber keinen abhalten, solche Anwendungen zu testen.

Die Dokumentation von beiden Modulen ist nicht besonders gut.
`Test::Cmd` hat weniger Abhängigkeiten und lässt sich leichter einsetzen.

Testen von GUI-Applikationen

Viele Programme sind keine Konsolen-Programme sondern Programme mit Graphischer Oberfläche (GUI).
Diese sind vielleicht mit `Perl/Tk`, `WxPerl` oder einer ganz anderen Sprache entwickelt. Auch GUIs sollten getestet werden. Ist beim Menü überall die richtige Funktion hinterlegt? Machen die Funktionen das was sie tun sollen?

X11::GUITest

Mit `X11::GUITest` können Applikationen mit einer GUI getestet werden, die ein X11-Framework verwenden. Diese Frameworks sind zum Beispiel GTK+, Qt, Xt und Motif. Es gibt noch weitere Frameworks.

Für das Modul `X11::GUITest` stehen nicht so viele Funktionen zur Verfügung wie für das später vorgestellte `Win32::GuiTest`.

Da Tastatureingaben immer an das aktive Fenster geschickt werden, muss das Programm im aktiven Workspace laufen und es darf zwischendurch kein anderes Fenster aktiviert werden.
Es wäre wünschenswert, wenn man Tastatureingaben an ein ausgewähltes Fenster schicken könnte.

Unter Solaris ist folgender Test erfolgreich gelaufen:

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use Tk;
 6
 7 my $mw = tkinit();
 8 $mw->bind('<Alt-c>',\&test);
 9
10 MainLoop;
11
12 sub test{
13     open(my $fh,'>','test.txtt') or die $!;
14     print $fh 'FooBar';
15     close $fh;
16
17     exit;
18 }
```

Listing 6

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use X11::GUITest qw(StartApp
         SendKeys FindWindowLike);
 6
 7 # starte die GUI
 8 StartApp('/pfad/zu/test.pl');
 9
10 # Zeit lassen zu Starten
11 sleep(2);
12
13 # Der Titel wird von Perl/TK gesetzt
14 # wenn nicht explizit angegeben
15 my ($window) = FindWindowLike('Test');
16
17 if(!$window){
18     die "Couldn't find a window\n";
19 }
20
21 # Sende Alt-C
22 SendKeys('%(c)');
23 sleep(2);
24 print 'ok' if(-e 'test.txtt');
```

Listing 7

Abo?

Einzelheft?

Alles auf unserer Webseite



http://foo-magazin.de

TPF-Ticker

++ Grant-Updates++

Der erste Perl6-Microgrant wurde an Steve Peters vergeben. Peters wird sich darum kümmern, dass Parrot unter verschiedenen Betriebssystemen mit unterschiedlichen Compilern compiliert werden kann.

Adam Kennedy hat der Perl-Foundation mitgeteilt, dass er sein "Proof of Concept" eines Perl-Refactoring Editors nicht beenden kann.

Zwei weitere Grants wurden bewilligt:

* Chris Dolan schreibt neue Policies für Perl::Critic
* Michael Peters wird Smolder, ein Webbasiertes Programm für Smoke-Tests verbessern.

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use Win32::GuiTest qw(:ALL);
 6
 7 my $dir = 'C:\PCE-0.3\pce';
 8 chdir $dir;
 9
10 system("start pce.exe");
11 sleep(2);12 my @windows = FindWindowLike(undef,"PCE");
13
14 print "Couldn't find a window\n" unless(@windows);
15 print "more than one window\n" if scalar(@windows) > 1;
16
17 # simulate Ctrl+Shift+S
18 SendKeys("+^s");
19 # send PID + '.txt'
20 SendKeys($$.".txt");
21 # press <ENTER>
22 SendKeys("{ENTER}");
```

Listing 8

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use Win32::GuiTest qw(:ALL);
 6
 7 # starte die GUI
 8 my $dir = 'C:\PCE-0.3\pce';
 9 chdir $dir;
10 system "start pce.exe";
11
12 # Zeit lassen zu Starten
13 sleep 6;
14
15 my ($window) = FindWindowLike('PCE');
16 if(!$window){
17     die "Couldn't find a window\n";
18 }
19
20 SendKeys("^n");
21 sleep 1;
22 SendKeys("Win32::GuiTest");
23
24 MouseMoveAbsPix(80,60);
25 select undef,undef,undef,0.3;
26 SendMouse("{LeftClick}");
27
28 SendKeys($$.".txt");
29 SendKeys("{ENTER}");
```

Listing 4

Das Skript `test.pl` (Listing 6) erzeugt ein Fenster mit `Perl/Tk` und wenn der Anwender `Alt-C` drückt, wird eine Datei erzeugt.

Das Skript leistet nichts großartiges, soll aber ein Beispiel sein, wie man mit `X11::GUITest` die Funktionalität einer GUI überprüfen kann. Doch wie testet man jetzt diese GUI? Auch hier reicht ein kleines Skript (Listing 7).

Wie man sieht, ist es relativ einfach, eine GUI zu testen.

Win32::GuiTest

`Win32::GuiTest` ist das Windows-Pendant zu `X11::GuiTest`. Als Beispiel-Applikation soll der Editor "kephra" von Herbert Breunung getestet werden.

Wichtig ist, dass man eine sehr neue Version des Moduls verwendet, da viele wichtige Funktionen erst in den neuesten Versionen implementiert sind. In dem Test soll überprüft werden, ob die "Speichern unter..."-Funktion richtig funktioniert.

Dazu wird nur die Funktion `SendKeys` benötigt. Das Modul bietet für sehr viele wichtige Funktionen die entsprechenden Methoden.

Das Beispielprogramm ist in Listing 8 dargestellt.

In einem Beispiel, in dem die Maus navigiert werden soll, um das Programm über die Toolbar neu zu speichern, wird die Funktion `MoveMouseAbsPix` benö-tigt, um die Maus zu einem bestimmten Punkt zu navigieren. Mit `SendMouse` wird dann der Mausklick simuliert (Listing 9).

Für alle Funktionen ein Beispiel zu zeigen, wäre zu aufwendig, weshalb es bei den zwei gezeigten Beispielen bleibt.

Zwischenfazit "GUI-Testing"

Das GUI-Testing sollte aus mehreren Teilen bestehen: Dem Test des Codes, Test der Oberfläche und dem Usability-Test. Wie man den Code testen kann, ist in den vorangegangenen Abschnitten beschrieben worden. Den Usability-Test kann man nicht wirklich automatisieren. Hier müssen echte Personen ran.

Der Test der Oberfläche dient dazu, dass die Buttons auch das Richtige tun und dass die Verknüpfung von Oberfläche und Code das gewünschte Ergebnis erzielt. Diese Tests sollten gemacht werden, denn was bringt schon der beste Code, wenn sich unter dem "Datei öffnen"-Button die "Suche"-Funktionalität verbirgt. Und wenn man etwas speichern will, wird auf einmal die Festplatte gelöscht.

Das sind Fehler, die in keinem Programm passieren sollten. Mit `Win32::GuiTest` und dem Modul `X11::GUITest` sind zwei Möglichkeiten vorhanden, die diese Tests der Oberfläche erleichtern.

Testen von Web-Applikationen

Ein weiteres Gebiet, auf dem häufig die Tests vernachlässigt werden sind die Web-Applikationen. Noch nicht einmal große Firmen - die es sich eigentlich leisten könnten und von einer funktionierenden Seite profitieren - testen ausreichend ihre Web-Applikationen.

Web-Applikationen sollten neben der Usability auf zwei Punkte hin getestet werden: Die Funktionalität und wie sich die Anwendung unter Last verhält. Bevor Lasttests gemacht werden, sollte erst die Funktionalität überprüfen.

Denn was bringt eine Webseite, die gut skaliert aber nicht die gewünschten Funktionen bietet.
Für Lasttests gibt es Tools wie `JMeter`, das in Java geschrieben ist, und `openSTA`, das es nur für Windows gibt. Auch die Möglichkeit, mit `HTTP::Recorder` Lasttests durchzuführen bietet Perl.

Um die Funktionalität zu überprüfen, gibt es vor allem zwei Module, die dies erleichtern: `Test::WWW::Mechanize` und das Modul `Test::WWW::Selenium`. Man kann sich die Testsuite auch mit `LWP` zusammenbauen, aber die zwei genannten Module bieten schon alle wichtigen Funktionen.

In den folgenden Sektionen werden die Module für die Funktionstests und das Modul für Lasttests vorgestellt.

Test::WWW::Mechanize

`Test::WWW::Mechanize` setzt - wie der Name schon sagt - auf `WWW::Mechanize` auf. Mit dem Modul kann man sehr leicht die Funktionalität einer Webseite überprüfen und prüfen ob immer das richtige Ergebnis geliefert wird.

Man kann natürlich die Module `WWW::Mechanize` und `Test::More` selbst kombinieren, aber das macht unnötig Arbeit und wird schnell unübersichtlich.

Mit dem Code in Listing 10 wird die Webseite `http://foo-magazin.de` überprüft. Die ersten drei Tests betreffen die Startseite. Hier wird erstmal überprüft, ob eine Antwort kommt. Irgendwo im Seiteninhalt muss dann das Wort "Perl" auftauchen. Der dritte Test checkt dann, ob die Seite den richtigen Titel enthält.

Die darauffolgenden drei Tests überprüfen eine weitere Seite und ob man von der Startseite zu der

"Preise"-Seite kommt. Bei den Preisen muss dann ein Preis für ein Einzelheft genannt werden. Und es muss ein Preis von 6 Euro auftauchen...

Die Ausgabe des Testskripts sieht dann so aus:

```
C:\>test_mechanize.pl
1..6
ok 1
ok 2 - found Perl
ok 3 - Perl-Magazin
  title
ok 4 - go to prices
ok 5 - contains price
  for one magazine
ok 6 - and there's a
  price of 6 Euro
```

Test::WWW::Selenium

Selenium ist in Java geschrieben - was hat das dann hier zu suchen? Es ist nur der Server in Java geschrieben, der Rest in Perl. Um mit `Test::WWW::Selenium` zu arbeiten, ist eine relativ neue Version der Java Runtime Environment (JRE) nötig - für ein neueres `Test::WWW::Selenium` ist die JRE 1.5 notwendig.

Im Gegensatz zum vorher gezeigten `Test::WWW::Mechanize` wird nicht Perl als Browser verwendet, sondern ein lokal installierter Browser. Dadurch ist es auch möglich, `JavaScript` zu testen, was mit `WWW::Mechanize` nicht möglich ist.

`Selenium` kann mit den gängigsten Browsern arbeiten:

- Internet Explorer
- Firefox
- Safari
- Opera
- und andere



Abbildung 2: Website-Test mit Selenium

Installation

Bei der Installation muss folgendes installiert werden: Die Module `Test::WWW::Selenium` und `Alien::SeleniumRC`.
`Alien::SeleniumRC` enthält die Server-Komponente, und das Test-Modul hält die Funktionen für die Tests vor.

Wenn alles installiert ist, kann es mit dem Testen losgehen. Es soll wieder eine Webseite des Perl-Magazins getestet werden (siehe Listing 11).

Bevor der Test durchgeführt werden kann, muss noch der Selenium-Server gestartet werden, der dann als Proxy dient. Der Server nimmt die Anfragen aus dem Skript entgegen und leitet diese dann weiter.

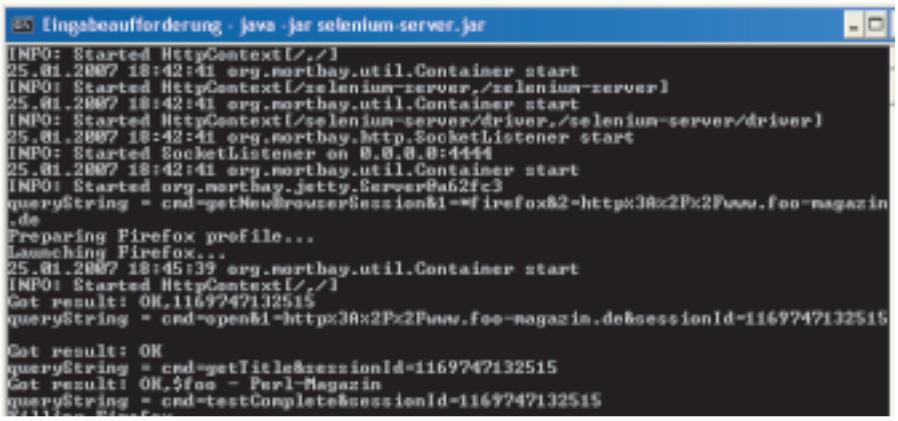


Abbildung 2: Meldungen des Servers auf der Konsole

Der Server startet den Browser. Dort ist die Seite dann in drei Teile geteilt:

Teil 1 ist eine Teil von Selenium, in dem man auch verschiedene Tools auswählen kann. Unter den Tools ist die Session-ID für den Test zu sehen.
Rechts nebendran sind immer die letzten Befehle des Test-Skripts zu sehen. Und der größte Teil ist für die Webseite vorgesehen. So kann live mitverfolgt werden, was im Browser passiert.

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use Test::More tests => 2;
 5 use Test::WWW::Mechanize;
 6 my $mechanize = Test::WWW::Mechanize->new();
 7
 8 # check "home"
 9 $mechanize->get_ok( 'http://www.foo-magazin.de' );
10 $mechanize->content_like(qr/Perl/i, "found Perl" );
11 $mechanize->title_is( '$foo - Perl-Magazin', 'Perl-Magazin title' );
12
13 # check "preis"
14 $mechanize->follow_link_ok( {text => 'Preise'}, 'go to prices' );
15 $mechanize->content_contains( 'Einzelheft',
16                               'contains price for one magazine' );
17 $mechanize->content_contains( '6.00 &euro;',
18                               'and there\'s a price of 6 Euro' );
```

Listing 10

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use Test::More tests => 2;
 6 use Test::WWW::Selenium;
 7
 8 my $url = 'http://www.foo-magazin.de';
 9
10 my $sel = Test::WWW::Selenium->new(
11                host => 'localhost',
12                port          => 4444,
13                browser       => '*firefox',
14                browser_url   => $url,
15                default_names => 1,);
16 $sel->open_ok($url);
17 $sel->title_like(qr/Perl.Magazin/);
```

Listing 11

HTTP::Recorder

`HTTP::Recorder` eignet sich hervorragend für Lasttests von Webapplikationen. Im Zusammenspiel mit `HTTP::Proxy` kann man über einen Browser die gewünschte Webseite besuchen, einen kleinen manuellen Test machen (zum Beispiel in Forum einloggen und Thema starten) und dabei alles "aufnehmen". Für die Aufnahme ist `HTTP::Recorder` zuständig.
In einem Skript kann man dann die aufgenommenen Aktionen "abspielen" und somit die Webseite einem Lasttest unterziehen.

Zwischenfazit "WebApp-Testing"

Die drei vorgestellten Module haben alle ihre Daseinsberechtigung.
`Test::WWW::Selenium` bietet zwar mehr Möglichkeiten, aber häufig reicht `Test::WWW::Mechanize` vollkommen aus. Dann sollte man auch `Test::WWW::Mechanize` verwenden. Zudem kennen sich einige schon mit dem Modul `WWW::Mechanize` aus - was auch für `Test::WWW::Mechanize` spricht.

`Test::WWW::Selenium` ist zudem etwas langsamer als das Gegenüber `Test::WWW::Mechanize`.
Wenn kein Browser installiert ist, kann man ist das Test-Modul `Test::WWW::Selenium` ungeeignet.

Es hängt also vom Einzelfall ab, welches Modul verwendet werden sollte.

Geht es weniger um Funktionstests als um Lasttests, ist `HTTP::Recorder` ein sehr gutes Modul. Im Gegensatz zu `WWW::Mechanize` muss man die URIs und Parameter nicht alle kennen, die gesendet werden. Man kann eine Aktion aufnehmen und dann wiedergeben.

Fazit

Wenn man schon von Anfang an, eine Testsuite aufbaut, ist es sehr einfach die Tests immer auf einem aktuellen Stand zu halten. Tests können viel Zeit sparen, da man durch Tests fehler schneller aufspürt und schneller die verantwortlichen Stellen findet.
Perl ermöglicht es, (fast) alles zu testen - nicht nur Perl-Programme!

Die Empfehlung lautet: Sobald eine Zeile Quellcode geschrieben wird, sollte auch ein Test dazu geschrieben werden.

Renée Bäcker

Werden Sie selbst zum Autor...

... wir freuen uns über Ihren Beitrag!



Info@foo-magazin.de

TPF-Ticker

++ Perl6-Microgrants++

Das Unternehmen "Best Practical" hat der Perl-Foundation 5000 US$ für die Weiterentwicklung von Perl6 gespendet.
Dieses Geld wird in 10 kleine Grants aufgeteilt. Programmierer, die an Perl6 weiterent-wickeln wollen, können sich unter der E-Mail-Adresse perl6-microgrants@perl.org für einen solchen Grant bewerben.

Mehr Informationen darüber, wie eine solche Bewerbung aussehen sollte ist im Blog der Perl-Foundation zu finden.

++ Grant-Updates++

Christopher H. Laco hat extrem viel für seinen "Mango"-Grant getan. Wishlists, Schema für Provider, Authentifizierungs-Plugin und vieles mehr. Mehr zu Mango:
http://handelframework.com/wiki/Mango

Parrot-Grant: Der nächste Design-Meilenstein wurde erreicht: IO PDD. Parrot 0.4.8 wurde am 16. Januar released.
Mehr zu Parrot: http://www.parrotcode.org/

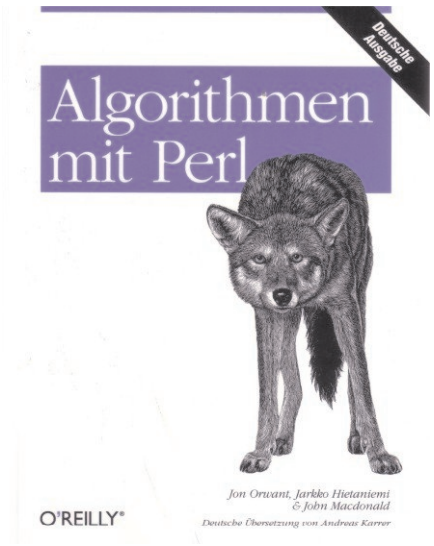
THE Perl FOUNDATION

Algorithmen mit Perl

Jon Orwant, Jarrko Hietaniemi & John Macdonald
O'Reilly, ISBN 3897211416
deutsch

Algorithmen sind das Handwerkszeug eines jeden Programmierers. So wie ein Tischler Handbewegungen verinnerlicht um Flächen glatt zu schleifen oder einer einen eleganten Schwung zu geben, lernen Programmierer bekannte Lösungsansätze zu verstehen, frei abzuwandeln und zu kombinieren. Diese häufig auftretenden Grundideen zum Sortieren, Strukturieren und Umwandeln von Daten könnten in jeder turing-kompletten Sprache vermittelt werden, aber hier wollen wir uns ja mit Perl befassen, was uns zu diesem Buch führt. Wer allerdings noch überhaupt kein Perl kann, sollte zuerst zum Kamelbuch ("Programmieren mit Perl") greifen. Programmieranfänger nehmen natürlich besser das Lamabuch ("Einführung in Perl") und wer eigentlich fertige Lösungen für praktische Aufgaben suchte, hat mehr Freude am Widderbuch ("Perl Kochbuch").

Während meines Studiums wurden Algorithmen anhand des Standardwerkes "Algorithmen in C" von Robert Sedgewick gelehrt. Dies ist sicher ein gutes und fundiertes Buch, aber "Algorithmen mit Perl" gefällt mir um einiges besser, da es lockerer geschrieben ist, und auf seinen 650 Seiten weit mehr Wissen bietet als die 720 Seiten des Klassikers. Während der sich vor allem mit dem Sortieren von ganzen Zahlen, Buchstaben, geometrischen Punkten, Baumstrukturen und Graphen beschäftigt, bekommen wir hier zusätzlich eine reichhaltige Praxis in den Bereichen Mengen, Matrizenrechnung, geometrische Algorithmen, Zahlensysteme, Kryptographie, Wahrscheinlichkeit, Statistik und numerische Analysis. Wie kann das sein, ist doch die Schrift hier nicht kleiner und beinhalten doch beide Bücher in etwa ähnlichem Verhältnis Text, Listings mit Quellcode und veranschaulichende Graphiken?
Ich denke das hat zwei Gründe. Zum einem sind unsere drei Autoren um bildliche, leicht greifbare Vergleiche nie verlegen. Sie kommen damit wesentlich schneller zum Wesentlichen und halten sich nur so lang, wie zum Verständnis notwendig, mit Mathematik und theoretischer



Informatik auf. Was nicht heißt, daß daran gespart wurde. Auch auf diesem Gebiet enthält das Buch mehr Inhalt, nur fehlt hier zum Glück die akademische Langatmigkeit. Dafür gibt es wertvolle Tipps zu Perl, wichtigen Modulen, der Programmierpraxis, Datenstrukturen (für manche war allein dieser Punkt schon ein Kaufgrund) und den Algorithmen, die Herr Sedgewick wohl nur unter 4 Augen raus-rücken würde.

Der andere Grund, warum das "Wolf-Buch" dichter und entspannter ist, ist die Programmiersprache. In Perl kann man Algorithmen meist kürzer und auf die wesentliche Logik reduziert ausdrücken. Auch optimiert der Interpreter Dinge wie den benötigten Speicherplatz für Arrays selber, was Algorithmen gegenüber ihrer C-Version ebenfalls vereinfachen kann. Deswegen ist Perl in Kombination mit diesem Buch auch sehr gut für akademische Zwecke geeignet.

Für einen genaueren Eindruck möchte ich zuletzt noch in das Kapitel 4 einsteigen, weil Sortierverfahren ganzer Zahlen fast das "Hello World" der Algorithmik sind und es das erste Kapitel ist, in dem es wirklich um Algorithmen geht.

Nach einem kernigen Terry Pratchet-Zitat wird der Leser erstmal mit einer allgemeinen Definition auf das Thema fokussiert. Dann werden die Grundbegriffe erklärt und es wird angerissen warum man über das Thema bescheid wissen sollte was unversehnes zu Perls interner "sort"-Funktion führt. Bei der Erläuterung der Möglichkeiten dieses Befehl, wird erstmals auf Details eingegangen und Perlcode gezeigt. Daraufhin graben sich die Autoren tiefer in Perl's Werkzeugkiste und stellen verschiedene Module und auch die berüchtigten Schwarzschen Transformation aber auch andere caching-Methoden vor. Erst danach werden die klassischen Sortieralgorithmen in Perl nachprogrammiert (natürlich erst die

einfachereren mit quadratischer dann die mit linear-logarithmischer Laufzeit). Hier kann man auch mit den detaillierten, schwarzweißen Graphiken Schritt für Schritt nachvollziehen, wie diese Algorithmen funktionieren und in der abschließenden Analyse werden Stärken und Schwächen der Verfahren ausgewertet und verglichen. Womit sich ein Kreis schließt, weil man jetzt erst Perls interne sort-Funktion vollständig begreift, die ebenfalls mit einer der vorgestellten Algorithmen benutzt.

Ich glaube das wirklich schöne an dem Buch ist, daß die Schreiber einerseits den Stoff und die dazugehörige Praxis beherrschen und daher beides auch sehr souverän und klar vermitteln können. Sie überfordern keine Neulinge aber langweilen auch keine Fortgeschrittenen. Jeder kann sich so viel aus den Kapiteln nehmen wie er braucht, ohne Grundlegendes zu verpassen. Andererseits haben sie auch noch Freude beim Erzählen und gehen auf die Bedürfnisse der Lernenden ein. Ich hatte beim Lesen wirklich das Gefühl, daß mir ein Freund Fragen beantwortet und nicht das ein Professor mich von oben herab belehren will.

Daher stellt sich mir fast die Frage:
Könnte man an dem Buch noch irgendetwas verbessern? Ja, es könnte mehr Leser haben.

Herbert Breunung

Rätsel ???

Für alle, die Spaß an Rätseln haben, ist dies hier gedacht. Sie können Ihre Lösung an uns mailen:

raetsel@foo-magazin.de

Unter allen Einsendungen, die ein paar Tests bestehen, verlosen wir 3 Ausgaben von "Mastering Perl" (englisch), das im Mai erscheinen wird. Einsendeschluss ist der 30. Juni 2007. Der Rechtsweg ist ausgeschlossen.

Schreibe eine Funktion "func", die escape-Sequenzen in Texten erweitert. Die escape-Sequenzen haben folgende Form:

```
X<.....>
```

Wobei das "X" für irgendeinen Buchstaben oder eine Zahl steht. "....." ist ein beliebiger String, der ausgeglichene "<....>" Sequenzen enthält. Durch einen Backslash vor einem "<" oder einem ">" verliert es die besondere Bedeutung. So ist

```
A<a b c D<edfg\>hij> k l m>
```

eine gültige Sequenz. Als Argumente der "func"-Funktion werden der String, bei dem Ersetzungen durchgeführt werden sollen, und eine Hashreferenz übergeben, die die escape-Sequenzen auf die entsprechenden Funktionen für die Ersetzung mappt. Wenn "func" eine escape-Sequenz wie folgt erkennt:

```
W<some text>
```

sollte es erst den Text in "some text" ersetzen und Backslashes entfernen - dort wo es erlaubt ist (vor "<" oder ">"). Danach muss im Hash nach der Funktion gesucht werden, die für "W" vorgesehen ist, und ab-schließend soll die ganze Sequenz durch den Rück-gabewert der Funktion ersetzt werden. Ein Beispiel:

```
expand_escapes(
   'Test: R<1\>4>, S<Test>, R<lrep>'
  {
    R => sub{ scalar reverse $_[0] },
    S => sub{ join " ", split //,
              $_[0] },
  }
);
```

Ergebnis:

Test: 4>1, T e s t, perl

Gute Kritiken - ein Traum für Code

In vielen Unternehmen gibt es Programmierrichtlinien, viele Programmierer halten sich an die Vorschläge aus Damian Conway's Buch "Perl Best Practices". Doch wie kann sichergestellt werden, dass die einzelnen Richtlinien eingehalten werden?
Das Modul `Perl::Critic` ist genau dafür gedacht!

Jeffrey Ryan Thalhammer - der Autor von `Perl::Critic` - bezeichnet sein Modul als ein erweiterbares Framework um Programmierstandards auf Perl-Code anzuwenden.
Es analysiert den Quellcode mit Hilfe des Moduls `PPI` von Adam Kennedy.

Die Programmierstandards, die standardmäßig von `Perl::Critic` angewendet werden, sind in "Perl Best Practices" von Damian Conway beschrieben, allerdings lassen sich die Regeln auch ein- und ausschalten. Wer nicht immer mit den Regeln von Conway einverstanden ist, kann sich `Perl::Critic` so konfigurieren, dass es auf eigene Regeln passt.

Auch komplett eigene Standards lassen sich implementieren, wofür aber Kenntnisse von `PPI` notwendig sind. Wie eigene Standards implementiert werden, wird in der nächsten Ausgabe von `$foo` gezeigt.

In diesem Artikel geht es darum, wie `Perl::Critic` verwendet werden kann und wie man die Bordmittel weitgehend für eigene Bedürfnisse anpassen kann. Das ganze passiert an einem Beispielskript (Listing 1).

Zur Überprüfung wird das Kommandozeilentool `perlcritic`, das mit dem Modul mitgeliefert wird, verwendet. Es wird einfach mit `perlcritic skript.pl` gestartet.

```
 1 #!/usr/bin/perl
 2
 3 print "Dies ist ein tolles Programm\n";
 4
 5 use strict;
 6 use warnings;
 7
 8 my $number = calc();
 9 unless($number == 42){
10     print "falsche Antwort!\n";
11 }
12 else{
13     print "42 ist die Antwort auf alle Fragen\n";
14 }
15
16 sub calc{
17     print "Term eingeben (Form: 3 + 4 * 12): ";
18     my $expression = <STDIN>;
19     chomp $expression;
20
21     my $res;
22     my $string = '$res = $expression';
23     eval $string;
24
25     return undef unless $string > 0;
26     return $res;
27 }
```

Listing 1

In der Ausgabe (Listing 2) ist ersichtlich, dass nicht alles nach den "Best Practices" programmiert wurde.
Hier wird angemerkt, dass es Code vor der Einbindung von `strict` gibt. Das sollte relativ klar sein, warum das unerwünscht ist. Der nächste Kritikpunkt des Tools ist der String-`eval`, denn es wird immer wieder kompiliert.
Aber warum soll kein `undef` explizit zurückgegeben werden?

Wer keine Ausgabe von "Perl Best Practices" hat, braucht nicht lange zu grübeln.
Mit dem Parameter `--verbose` kann `perlcritic` dazu gebracht werden, etwas geschwätziger zu werden (siehe Listing 3).

Mit diesem `--verbose`-Wert von 10 wird genau erklärt, warum eine Regel existiert und wie man das verbessern kann.

Policies

Die Regeln sind in den sogenannten **Policies** festgehalten. `Perl::Critic` hat die Regeln nach dem Inhaltsverzeichnis von "Perl Best Practices" organisiert. So finden sich die

```
C:\>perlcritic step0.pl
Code before strictures are enabled at line 3, column 1.
  See page 429 of PBP.  (Severity: 5)
Expression form of "eval" at line 23, column 5.  See page 161 of PBP.
  (Severity: 5)
"return" statement with explicit "undef" at line 25, column 5.  See
  page 199 of PBP.  (Severity: 5)
```

Listing 2

Regeln zur Dokumentation im Namensraum Perl::Critic::Policy::Documentation

Die Module selbst sind sprechend benannt. Das Modul, dass leere Strings (durch Anführungszeichen begrenzt) anmerkt, heißt dementsprechend ProhibitEmptyQuotes und gehört zum Namensraum P::C::Policy::ValuesAndExpressions Wie man schon am Namen der Regel erkennen kann, sind leere Strings unerwünscht.

Einteilung der Fehler

Die Abweichungen vom Programmierstandard werden in 5 verschiedene Grade eingeteilt. Die schwerwiegenden Abweichungen sind dabei vom Level "5", während kleine optische Abweich-ungen vom Level "1" sind. Standardmäßig werden nur die Meldungen vom Level "5" angezeigt.
Auch die Ausgabe aus dem Testlauf zeigen nur die "Fehler", die sehr schwerwiegend sind.

Die Einteilung in 5 Level ist sehr sinnvoll, da man sich nach und nach an die kleinen Abweichungen vom Programmierstandard annähern kann. Wenn standardmäßig **alle** Meldungen angezeigt bekommen würde, wäre man von der Masse an Meldungen regelrecht erschlagen. Man sollte sich also von den oberen Level nach unten durcharbeiten.
In der folgenden Tabelle kann man gut sehen, dass auch bei einem relativ kurzen Programm die Meldungen der unteren Level in der Überzahl sind. Welche Meldungen angezeigt werden sollen, kann man über den `--severity`-Parameter des Konsolenprogramms einstellen.
Wenn man sich die Meldungen der unteren Level anschaut, dann

Level	Anzahl Meldungen
5	3
4	1
3	0
2	6
1	2

merkt man schnell, dass hier wirklich nur optische Dinge angemeckert werden.

Regeln anpassen

Die Regeln, die Damian Conway in seinem Buch aufstellt sind seine persönliche Meinung. Bei den meisten Regeln muss man ihm Recht geben -wie zum Beispiel die Verwendung von `strict`, andere Regeln sind diskussionswürdig. Bevor man seine eigenen Standards ändert und man sich nicht wohl fühlt, sollte man lieber Conways Regeln anpassen.

critic-Pseudopragma

Im Prinzip hat Conway Recht, wenn er sagt, dass man keine Namen von built-in-Funktionen für eigene Subroutinen verwenden sollte. Wenn aber eine built-in-Funktion tatsächlich überschrieben werden soll - was aus den unterschiedlichsten Gründen

```
C:\>perlcritic --verbose=10 step0.pl
Code before strictures are enabled at line 3, column 1.
  TestingAndDebugging::RequireUseStrict (Severity: 5)
      Using strictures is probably the single most effective way to improve
      the quality of your code. This policy requires that the `'use strict''
      statement must come before any other statements except `package',
      `require', and other `use' statements. Thus, all the code in the entire
      package will be affected.
Expression form of "eval" at line 23, column 5.
  BuiltinFunctions::ProhibitStringyEval (Severity: 5)
      The string form of `eval' is recompiled every time it is executed,
# und so weiter
```

Listing 3

vorkommen kann, kann man diese Methode vor Perl::Critic schützen.

Das geht mit einem "Pseudopragma". Das Pseudopragma ist kein richtiges Pragma, da es nur ein erweiterter Kommentar ist. Mit ##no critic kann Perl::Critic temporär ausgeschalten werden. Code innerhalb der Subroutine nicht.

Der "no critic"-Anweisung kann man auch Argumente übergeben. So kann man auch einzelne Regeln ausschließen. Die Argumente werden als Reguläre Ausdrücke betrachtet, so dass man auch sehr schnell eine ganze Gruppe von Perl::Critic nach einer Datei mit dem Namen .perlcriticrc.

Die Konfigurationsdatei ist wie eine .ini-Datei formatiert, wobei der Sektionsname dem Namen des Moduls für die Regel entspricht (siehe Listing 5).

```
##no critic (EmptyQuotes)
#
# Das Argument von no critict matcht auf die Regel
# Perl::Critic::Policy::ValuesAndExpressions::ProhibitEmptyQuotes
my $foo = "";
```

Listing 4

Steht das ##no critic in der gleichen Zeile wie ein Statement, ignoriert Perl::Critic nur diese Zeile. Mit

```
sub format{ ##no critic
    # Code der Subroutine
  }
```

wird nur der Name der Subroutine ignoriert, der Code danach aber nicht.

Wenn ein größerer Bereich nicht analysiert werden soll, dann kann man diesen Bereich in einen ##no critic / ##use critic-Block stellen.

```
##no critic
sub format{
  # Code der Subroutine
}
##use critic
```

Jetzt wird die komplette Subroutine von Perl::Critic ignoriert.

Wenn die Analyse für einen Block ausgesetzt werden soll, kann man auch folgendes machen:

```
sub format{
  ##no critic
  # Code der Subroutine
}
```

Hier wird der Name der Subroutine noch analysiert, aber der Regeln ausschalten kann.

Listing 4 zeigt, wie Meldungen über "EmptyQuotes" verhindert werden.

Konfigurationsdatei

Um nicht in jedem Programm die gleichen Code-Fragmente mit den critic-Anweisungen von der Analyse ausnehmen zu müssen, kann man eine Konfigurationsdatei erstellen.
Ich werde - auch wenn Conway es anders vorschlägt - nicht auf Postfix-Bedingungen verzichten und unless-Blöcke werden auch in Zukunft im Code zu finden sein.

Da diese Meldungen global ausgeschaltet werden sollen, ist die Konfigurationsdatei die ideale Lösung. Die Datei muss im aktuellen Verzeichnis oder im HOME-Verzeichnis liegen - genau in der Reihenfolge sucht Perl::Critic nach einer Datei mit dem Namen .perlcriticrc.

Die Konfigurationsdatei ist wie eine .ini-Datei formatiert, wobei der Sektionsname dem Namen des Moduls für die Regel entspricht (siehe Listing 5).

Der Anfang des Modulnamens ("Perl::Critic::Policy") kann auch weggelassen werden. Perl::Critic geht davon aus, dass alle Policy-Module in diesem Namensraum liegen.

Um diese Regel komplett auszuschalten, reicht es ein "-" vor den Modulnamen zu schreiben, also

```
[-ValuesAndExpression
 ::ProhibitEmptyQuotes]
```

Nachdem alle Fehler beseitigt wurden und manche Regeln ausgeschaltet wurden, läuft das Programm durch und bestätigt, dass der Code den "Best Practices" der Perl-Programmierung entspricht.

Test::Perl::Critic

Bei der Entwicklung von Modulen ist es aber umständlich, immer wieder das Programm aufzurufen.

```
  [Perl::Critic::Policy::ValuesAndExpressions::Prohib
itEmptyQuotes]
  severity = 1
  arg1 = value1
  arg2 = value2
```

Listing 5

Da hilft es doch, wenn es ein Testskript dafür gibt. Das ganze kann auch sehr einfach über `Test::Perl::Critic` eingebunden werden.

Dieses einfache Skript in den t-Ordner zu den anderen Tests und danach wird bei jedem Testlauf auch auf die Einhaltung der Programmierstandards geprüft.

```
use Test::More;
use Test::Perl::Critic
    (-severity => 1);
all_critic_ok();
```

Natürlich kann der `severity`-Level höher gesetzt werden, je nach Wunsch, wie stark die Standards verwendet werden sollen.

Wenn die Standards über eine Konfigurationsdatei angepasst wurden, sollte diese mit in die Distribution gepackt werden und im Testskript benannt werden, damit die Installation auf einem anderen Rechner nicht fehlschlägt.

Das Pragma "criticism"

Die strengste Form der Überprüfung ist die Verwendung des `criticism`-Pragmas im Programmcode. Allerdings hat das auch einen erheblichen Geschwindigkeitsnachteil, weil bei jedem Compile-Vorgang der Code überprüft wird. Das ist eigentlich unnötig. Wenn der Testlauf mit dem Kommandozeilentool oder mit `Test::Perl::Critic` erfolgreich war und der Code nicht verändert wurde, dann entspricht er immer noch den "Best Practices".

Aber der Vollständigkeit halber wird hier gezeigt, wie die Verwendung des Pragmas aussieht.

```
1 use criticism;
2
3 $_ = "test";
4 print;
```

Wenn dieses Programm ausgeführt wird, wird gleich das fehlende strict angemeckert:

```
C:\>perl critic.pl
critic.pl:
 Code before strictures
    are enabled at line
    3, column 1. See
    page 429 of PBP.
test
```

Dieses Pragma kann man nicht deaktivieren wie zum Beispiel `strict` - es gibt also kein `no criticism`. Man kann nur mit dem oben genannten Pseudo-Pragma arbeiten. `criticism` ignoriert auch die Konfigurationsdateien.

Ausschau

In der nächsten Ausgabe wird gezeigt, wie man eigene Policies entwickeln kann.

Renée Bäcker

Werden Sie selbst zum Autor...

... wir freuen uns über Ihren Beitrag!



Info@foo-magazin.de

Win32::OLE

Anwendungen fernsteuern

"Herr Meier, die Excel-Tabellen anpassen - aber in allen 1000 Dateien!!" - Puuh, wenn man das doch nur automatisieren könnte...

Perl springt da zu Hilfe! Viele Anwendungen, die unter Windows laufen, können direkt angesteuert werden. Ganz sicher funktioniert es mit (fast) allen Microsoft Anwendungen, aber auch viele andere Programme wie zum Beispiel Adobe Photoshop oder Lotus Notes lassen sich automatisieren.

Das Modul der Wahl heißt `Win32::OLE`. Die `OLE`-Automation ist eine Technologie, die auf `COM` beruht (siehe Info-Kasten).

Die Anwendung (oder .dll), die die Schnittstelle zur Verfügung stellt, wird **Automation Server** genannt. Als **Automation Client** oder **Automation Controller** wird die Anwendung bezeichnet, die diese Schnittstelle benutzt.

`Win32::OLE` ermöglicht es, ein Programm zu schreiben, das als Automation Controller agiert. Dadurch kann auf Makros mit VBA oder auf andere Sprachen verzichtet werden.

Damit eine Anwendung automatisiert werden kann, muss sie in der Registry von Windows eingetragen sein. Dort kann man auch suchen, wenn die genaue Bezeichnung des Automation Servers nicht bekannt ist. So kann ein Perl-Skript mit Photoshop 5.0 gut funktionieren, aber mit Photoshop 7.0 klappt gar nichts - und alles nur weil sich der Name in der Registry geändert hat.

Die Automation solcher Anwendungen kann recht schwie-rig sein, weil nicht immer die Namen von Funktionen bekannt sind. Microsoft stellt auf der Homepage im "Developer"-Bereich eine Unmenge an Dokumentation zur Verfügung.

OLE und COM

OLE steht für "Object Linking and Embedding" und wurde von Microsoft entwickelt. Das Objektsystem und Protokoll ermöglicht es, sogenannte Verbunddokumente zu erzeugen. Das heißt, es ist möglich, Excel-Tabellen in einem Word-Dokument oder einer PowerPoint-Präsentation zu verwenden.
Bei einem Doppelklick auf die Tabelle wird dann Excel geöffnet und sie kann direkt bearbeitet werden.

Die OLE-Objekte können auf zwei Arten verbunden werden: Über eine Verlinkung (Object Linking) oder eingebettet (Embedding).

Im Kern von OLE steckt das Component Object Model (COM), eine Technologie, die es ermöglichen, einen "Server" zu instanzieren (eine .dll oder eine Anwendung) und dann über das Interface die Funktionen des "Servers" zu nutzen.

Daher sind die Microsoft-Anwendungen am "einfachsten" zu automatisieren. Generell sollte in der Dokumentation der Anwendung einiges zu der Schnittstelle zu finden sein.

Bestandteile von Win32::OLE

Für die Unterstützung des Programmierers besteht `Win32::OLE` aus vier Modulen. Das wichtigste Modul ist `Win32::OLE` selbst. Es stellt alle Funktionen zur Verfügung, um auf die Schnittstelle zuzugreifen.

Win32::OLE

`Win32::OLE` ist das wichtigste Modul, das hier benötigt wird. Es erzeugt den Client, der zur Automatisierung nötig ist. Es gibt drei Möglichkeiten, einen Client zu erzeugen. Zum einen gibt es die Methode `new`, die einen neuen Client erzeugt, die Methode `GetActiveObject` versucht, einen laufenden Client zu übernehmen. Die dritte Möglichkeit ist `GetObject`, mit dem eine Referenz auf ein spezifisches Objekt erzeugt wird. So wird mit

```
my $doc = Win32::OLE
->GetObject("test.doc");
```

eine Referenz auf die Datei `test.doc` erzeugt.

Win32::OLE::Const

Wer kennt sich schon mit den ganzen Konstanten aus, die für die Anwendung benötigt werden? Und Zahlen sind weniger sprechend als Worte. Aus diesem Grund kann

man mit Win32::OLE::Const die Konstanten für die Applikation importieren.

Mit

```
  use Win32::OLE::Const
'Microsoft Excel';
```

werden alle Konstanten, die in Excel benötigt werden, importiert.

Alle Konstanten, die importiert werden, können mit einem kleinen Skript angezeigt werden (Listing 1).

```
1 use Win32::OLE::Const;
2 my $const = Win32::OLE::Const
3                     ->Load('Excel.Application');
4
5 for my $key ( %$const ){
6     print $key, " -- ",$const->{$key},"\n";
7 }
```

Listing 1

Win32::OLE::Enum

Dieses Modul stellt einige Funktionen für "Collections" zur Verfügung. Collections sind ein spezieller Datentyp, die ein Array von Objekten oder Daten enthalten. Die Collections haben normalerweise ein paar Methoden um über die Objekte iterieren zu können, oder die Anzahl der Objekte zu bekommen und noch einige mehr.

Win32::OLE::Enum erweitert die Collections um einige Methoden. So kann man ein Array mit den Objekten bekommen oder um das "nächste" Objekt zu bekommen (Listing 2).

Win32::OLE::Variant

Der Application Server erwartet Parameter vom Datentyp Variant. Da diese in Perl nicht existieren, müssen die Parameter des Perl-Programms in diesen Datentyp umgewandelt werden. Genau dafür ist Win32::OLE::Variant gedacht.

Win32::OLE übernimmt die Umwandlung, aber in manchen Fällen soll der Variant-Typ selbst bestimmt werden. Wenn ein String explizit in ein Variant vom Typ VT_DATE gewandelt werden soll, geht das so:

```
  $dt = Variant(VT_DATE,
"August 24, 1970");
```

Zugriff auf die Server

In den folgenden Abschnitten werden zwei Beispiele gezeigt, wie mit Win32::OLE gearbeitet werden kann. Zum einen wird ein Word-Dokument erzeugt, in dem ein kleiner Absatz formatiert wird und für eine Adressliste werden alle E-Mail-Adressen aus Outlook gesammelt.

```
1 my $sheets = $xls->Workbook(1)->Sheets;
2 my $enum   = Win32::OLE::Enum->new($sheets);
3 while( defined my $sheet = $enum->Next ){
4     # ...
5 }
```

Listing 2

Standardmäßig sind die Parameter der Funktionen durch ihre Reihenfolge bestimmt und nicht durch Namen. Man findet also eher selten Aufrufe wie

```
$obj->method(
  -name => 'Name',
  -file => 'file');
```

sondern vielmehr Methodenaufrufe wie

```
$obj->method(
  'Name','file');
```

Da mit dieser Schreibweise häufig viele undefs übergeben werden müssen, kann auch eine Hashreferenz übergeben werden, die dann die Namen und die Parameter enthält.

Absätze in Word verändern

Das folgende Beispiel (Listing 3) soll kurz zeigen, wie man mit Hilfe von Win32::OLE ein Word-Dokument erzeugen kann. Es wird jedoch nur eine kleine Grundfunktion gezeigt.

Die Automatisierung kann von noch viel weiter gehen.

In Zeile 5 wird das Modul Win32::OLE eingebunden. Dadurch wird die Automatisierung überhaupt erst ermöglicht.

Ein neuer "Client" wird in Zeile 7 erzeugt. Der erste Parameter für die Methode new ist die Anwendung, die gestartet werden soll - übergeben wird die Class-ID oder die Programm-ID (wie in diesem Fall). Mit Quit als zweitem Parameter wird gesagt, wie die Anwendung beendet werden soll wenn der Perl-Client "stirbt". Quit ist in diesem Fall eine OLE-Methode. Man kann auch eine Code-Referenz übergeben.

Da bisher nur der Client erzeugt wurde, muss jetzt noch ein Dokument hinzugefügt werden. Das wird in Zeile 8 gemacht. In der Zeile 11 wird die Stelle "geholt", an der eingefügt werden soll.

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use Win32::OLE;
 6
 7 my $app = Win32::OLE
 8               ->new('Word.Application','Quit');
 9 my $doc = $app->Documents->Add;
10
11 my $sel = $app->Selection;
12
13 $sel->TypeText(q~$foo - Perl-Magazin~);
14 $sel->TypeParagraph;
15
16 $app->SaveAs('test.doc');
```

Listing 3

E-Mail-Adressen aus Outlook sammeln

Wer eine Liste der E-Mail-Adressen aus Outlook braucht, kann dies auf verschiedene Weisen lösen. Entweder ohne Perl, oder - was wesentlich bequemer ist - mit Perl und `Win32::OLE` (Listing 4).

In diesem Beispiel wird auch `Win32::OLE::Const` verwendet. In Zeile 3 werden die Konstanten, die für Outlook gebraucht werden, importiert. In Zeile 5 wird ein Client erzeugt.

In Zeile 7 wird dann der Namespace geholt (siehe Infokasten "Namespace und MAPI"). Warum in diesem Beispiel die Konstanten genutzt werden, wird in Zeile 8 deutlich. Die Konstante `olFolderContacts` wird hier verwendet um den Kontakte-Ordner von Outlook zu verwenden.

Die Kontakte werden dann als Collection mittels `Items` in Zeile 9 geholt. In der `for`-Schleife werden dann die einzelnen E-Mail-Adressen ausgegeben.

```
 1 use strict;
 2 use Win32::OLE;
 3 use Win32::OLE::Const 'Microsoft Outlook';
 4
 5 my $app    = Win32::OLE
 6              ->new('Outlook.Application', 'Quit');
 7
 8 my $space  = $app->GetNamespace("MAPI");
 9 my $folder = $space
10              ->GetDefaultFolder(olFolderContacts);
11 my $items  = $folder->Items;
12
13 for my $index ( 1..$items->Count ){
14     my $contact = $items->item($index);
15     print $contact->{EmailAddress},"\n";
16 }
```

Listing 4

Namespace und MAPI

Der Namespace ist ein abstraktes Stammobjekt für eine Datenquelle. Das Objekt wird für den Zugriff auf Speicherorte (zum Beispiel "Ordner") benötigt. Der einzige verfügbare Namespace bei Outlook ist "MAPI".

"MAPI" steht für "Messaging Application Programming Interface" und wurde von Microsoft entwickelt. Durch diese Schnittstelle ist es möglich, aus allen möglichen Anwendungen heraus Mail zu verschicken ohne dass der Mail-Client geöffnet werden muss.

Informationen sammeln

Wie eingangs schon erwähnt, kann die Automation schwierig werden, wenn ein paar Informationen fehlen. Es fängt damit an, dass man wissen muss, mit welcher Bezeichnung der Automation Server gestartet werden muss. Ein Blick in die Registry sollte helfen.

Die Registry kann gestartet werden, indem man unter Start ->Ausführen... einfach `regedit` eingibt. Danach öffnet sich der Registrierungs-Editor (außer das wurde von einem Admin gesperrt).

Unter dem Pfad `HKEY_LOCAL_MACHINE\SOFTWARE\Cl`asses\CLSID befinden sich alle angemeldeten OLE-Server. Hier kann dann auch nach der gewünschten Anwendung gesucht werden.

In den Code-Beispielen auf der Homepage von $foo befindet sich auch ein Skript, mit dem man die Registry durchsuchen kann.

Bei ActivePerl von ActiveState ist auch der sogenannte OLE-Browser dabei. Mit dieser Anwendung kann man überprüfen, welche Anwendungen von Perl aus ferngesteuert werden können.

Die Informationen über die Methoden des OLE-Objekts müssen in der Dokumentation der Anwendung gesucht werden.

Fazit

Manche Aufgaben können nur mit den Anwendungen direkt gemacht werden und `Win32::OLE` bietet eine sehr gute Schnittstelle zu den Windows-Programmen. Leider ist es recht aufwendig, an die möglichen Methodennamen und Parameter zu kommen. Microsoft stellt zwar für seine eigenen Produkte eine sehr umfangreiche Dokumentation zur Verfügung, aber das ist teilweise auch sehr unübersichtlich - auf Grund der Größe.

Wer aber viel unter Windows mit Perl zu automatisieren hat, der sollte sich das Modul auf jeden Fall mal anschauen.

Ausblick

In der nächsten Ausgabe von $foo wird gezeigt, wie man Excel relativ einfach Automatisieren kann.

Renée Bäcker

Perl-Community.de



Forum - Wiki - Linkliste - IRC - Perldoc - uvm.

Da kriegste was auf die Ohren

Mit "Radio Perl" und "Perlcast" gibt es zwei Netcasts zum Thema Perl. Hier werden die neuesten Aufnahmen kurz vorgestellt.

Radio Perl

14.03.2007 - Folge 4

Der 9. Deutsche Perl-Workshop ist das Hauptthema der vierten Folge von "Radio Perl". Jochen Lillich gibt einen schnellen Überblick über die Vorträge und seine Erfahrungen.

Wie in jeder Folge dürfen auch die Neuigkeiten aus der Perl-Welt nicht fehlen. Weiterhin werden 5 Module von CPAN kurz vorgestellt.



10.02.2007 - Folge 3

In der dritten Folge von "Radio Perl" geht es im Schwerpunktthema um neue Features in Perl 5.9.4. Jochen Lillich erklärt alle wichtigen Neuerungen und wie sie eingesetzt werden.

Er weist auf neue Artikel auf perl.com hin und stellt drei Module vor, die man im Web-Bereich einsetzen kann.

Perlcast.com

Josh McAdams

Josh McAdams steht diesmal auf der anderen Seite des Mikrofons. Er selbst wurde interviewed und berichtet über die Arbeit als Mit-Organisator der YAPC::NA::2006.

Bob Walsh

Bob Walsh erklärt in dem Interview, was man als MicroISV (ISV=IndependentSoftwareVendor) alles falsch machen kann. Er geht auch darauf ein, worauf man achten sollte und wie man bestimmte "Fallen" umgehen kann.



Uri Guttmann

Auf der OSCON 2006 hat sich Josh McAdams mit Uri Guttmann über dessen aktuelle Projekte unterhalten. Auch das Modul File::Slurp und das System "Stem" waren in dem Interview ein Thema.

Jeffrey Thalhammer

In dem Interview mit Jeffrey Thalhammer geht es um das Modul Perl::Critic, das auch in dieser Ausgabe von $foo vorgestellt wird. Jeffrey erklärt dabei, wofür das Modul gedacht ist und warum er das Modul geschrieben hat. Ein sehr interessantes Interview!

Termine

Veranstaltungen Mai, Juni und Juli 2007

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 den nächsten Ausgaben von $foo veröffentlicht werden sollen? Dann schicken Sie bitte eine E-Mail an **termine@foo-magazin.de**

Mai 2007

1. Mai	Stuttgart.pm
3. Mai	Dresden.pm
8. Mai	Frankfurt.pm
9. Mai	Hamburg.pm Cologne.pm
14. Mai	Ruhr.pm
16. Mai	Munich.pm
17. Mai	Darmstadt.pm Erlangen.pm
29. Mai	Bielefeld.pm
30. Mai	Berlin.pm

Juni 2007

5. Juni	Frankfurt.pm Stuttgart.pm
7. Juni	Dresden.pm
11. Juni	Ruhr.pm
13. Juni	Hamburg.pm Cologne.pm
21. Juni	Darmstadt.pm Erlangen.pm
25. Juni	YAPC::NA (Housten, TX)
26. Juni	Bielefeld.pm YAPC::NA (Housten, TX)
27. Juni	Berlin.pm YAPC::NA (Housten, TX)

Juli 2007

3. Juli	Frankfurt.pm Stuttgart.pm
5. Juli	Dresden.pm
9. Juli	Ruhr.pm
11. Juli	Hamburg.pm Cologne.pm
19. Juli	Darmstadt.pm Erlangen.pm
23. Juli	OSCON (USA, Portland)
24. Juli	Bielefeld.pm OSCON (USA, Portland)
25. Juli	Berlin.pm OSCON (USA, Portland)
26. Juli	OSCON (USA, Portland)
27. Juli	OSCON (USA, Portland)



http://www.Perl.org

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



http://www.perl-foundation.org

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



http://www.perl-workshop.de

Der Deutsche Perl-Workshop hat sich zum Ziel gesetzt, den Austausch zwischen Perl-Programmierern zu fördern. Der 10. Deutsche Perl-Workshop findet im Februar 2008 in Erlangen statt.



http://www.perl-community.de

Das 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. In der Linkliste von Perl-Community.de finden sich viele Verweise auf nützliche Seiten.



http://www.pm.org

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

// SEIBERT / MEDIA

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



Professionell Perl programmieren lernen?

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

//SEIBERT/MEDIA besteht aus den drei Kompetenzfeldern Technologie, Consulting und Design und gehört zu den erfahrenen und professionellen Internet-Agenturen in Deutschland. Wir entwickeln seit 1996 mit heute knapp 50 Mitarbeitern Intranets, Extranet-Systeme, Web-Portale aber auch klassische Internet-Seiten. Seit 2005 konzipiert unsere Designabteilung hochwertige Unternehmensauftritte und kommunikative Konzepte. Beratung im Bereich Online-Marketing und Usability runden das Leistungsportfolio ab.

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

Wir freuen uns auf Ihre Bewerbung unter http://www.seibert-media.net/jobs/.

//SEIBERT/MEDIA GMBH
RHEINGAU PALAIS / SÖHNLEINSTRASSE 8
65201 WIESBADEN
T. +49 611 20570-0 / F. +49 611 20570-70
BEWERBUNG@SEIBERT-MEDIA.NET
HTTP://WWW.SEIBERT-MEDIA.NET/JOBS/



Smart-Websolutions

Windolph und Bäcker GbR

Perl-Programmierung

info@smart-websolutions.de