Ausgabe 3/2008 – www.foo-magazin.de - 6,00 €

$foo

PERL MAGAZIN

Scaffolding
Gerüste für Webanwendungen

Debugger Einführung
Wichtige Kommandos und Tricks

Nr 07

Sichern Sie Ihren nächsten Schritt in die Zukunft

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

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

Perl Backend Developer (m/w)

Ihre Aufgaben sind:

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

Unsere Anforderungen an Sie sind:

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

Software Developer (m/w)

Ihre Aufgaben sind:

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

Unsere Anforderungen an Sie sind:

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

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

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

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

www.astaro.de

astaro
internet security

e-mail | web | net | security

VORWORT

Zusammenfassungen können die Aktivität erhöhen

Vienna.pm hat bei der YAPC::Europe 2007 einen größeren Gewinn erzielt, der gemäß der eigenen Satzung wieder Perl zu Gute kommen soll. Die Wiener Perl-Mongers haben beschlossen, mit dem Geld einen „Winter of Code“ zu veranstalten. Eine der unterstützten Aktivitäten ist die wöchentliche Zusammenfassung der Perl 5 Porters Mailingliste. David Landgren hat diese Aufgabe übernommen.

Doch was bringt so eine Zusammenfassung?

Anscheinend sehr viel. Mit solch einer Zusammenfassung erreicht man viele Leute außerhalb der Mailingliste, da die Zusammenfassung an verschiedenen Stellen veröffentlicht wird. So bekommen viel mehr Personen einen Eindruck davon, was bei den Perl 5 Porters los ist. Das bringt Transparenz und Interesse. Man muss sich nicht erst durch hunderte von Mails wühlen um ein interessantes Thema zu entdecken.

Landgren fügt jeder Zusammenfassung auch ein „ToDo der Woche“ hinzu. Eigentlich kann man alle ToDos auch so nachlesen. Einfach mal `perldoc perltodo` in der Kommandozeile eintippen. Aber wer schaut da schon rein? In der Zusammenfassung liest das doch der eine oder andere! So wurden schon ein paar ToDos auf Grund der Zusammenfassung erledigt oder zumindest angefangen.

Seit dem es die Zusammenfassungen gibt, ist allgemein die Aktivität auf der Mailingliste angestiegen - so mein subjektives Empfinden. Es werden mehr Patches geschickt, es werden alte Bugreports angefasst und es wird mehr diskutiert. Das Interesse am Perl-Kern scheint allgemein gesteigert zu sein!

Und das tut der Perl-Entwicklung auch ganz gut.

Offenbar gibt es jetzt auch Überlegungen, solche Zusammenfassungen auch für die Perl 6 Mailingliste einzuführen. Vielleicht hat das dort den gleichen positiven Effekt wie bei den Perl 5 Porters.

Wünschenswert ist es auf jeden Fall.

Die Codebeispiele können mit dem Code

ksjf325

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

Renée Bäcker

Ab dieser Ausgabe werden alle weiterführenden Links auf del.icio.us gesammelt. Für diese Ausgabe: http://del.icio.us/foo_magazin/issue7

IMPRESSUM

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

INHALTSVERZEICHNIS

ALLGEMEINES

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

Renée Bäcker

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

Ferry Bolhár-Nordenkampf

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

Herbert Breunung

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

Thomas Fahle

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

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

Moritz Lenz

Moritz Lenz wurde 1984 in Nürnberg geboren. Schon in seiner Schulzeit entwickelte er Vorlieben für Chemie, Physik und Informatik. Inzwischen studiert er Physik mit Nebenfach Informatik in Würzburg. Seit etwa vier Jahren ist Perl seine bevorzugte Programmiersprache. Zu seinen Lieblingsthemen gehören Kryptografie und Sicherheitsaspekte, reguläre Ausdrücke, Unicode und die Perl 6-Entwicklung.

Ronnie Neumann

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

Selenium - gib‘ Bugs keine Chance

Ohne Webapplikationen geht heute fast gar nichts mehr. Online-Händler erfreuen sich immer weiter steigender Umsätze, Mails können unterwegs mit Webmailern abgerufen werden und manch einer schreibt seine Lebensgeschichte in ein privates Blog.

Durch Userinteraktionen werden Webapplikationen komplex, da es keinen festen „Weg“ durch die Applikation gibt. Damit dennoch ein reibungsloser Ablauf möglich ist, muss die Anwendung schon während der Entwicklung getestet werden. Das ist besonders gut mit `Test::WWW::Selenium` möglich.

Selenium besteht aus mehreren Komponenten, wobei man nicht unbedingt alles installieren muss. Der wichtigste Teil ist der `Selenium Core`, das ein Paket mit etlichen JavaScript-Programmen und HTML-Seiten ist. Eine weitere Komponente ist `Selenium IDE`, ein Firefox-Plugin zur Aufnahme von Tests und schließlich die `Selenium Remote Control`. Die letztgenannte Komponente besteht aus einem Server und verschiedenen Bindings. Es gibt für sehr viele Programmiersprachen die Bindings für den Server, darunter auch Perl.

Für den Einstieg ist die `Selenium IDE` sehr gut geeignet, um Tests zu schreiben. Die IDE lässt sich ganz einfach wie jedes andere Firefox-Plugin auch installieren.

Tests mit der IDE aufnehmen

Nach dem Start von Firefox wird die IDE über `Extras -> Selenium IDE` gestartet (Abbildung 1). Durch Klicken auf den roten „Knopf“ wird die Aufnahme gestartet. Ab diesem Zeitpunkt wird jeder Klick auf einen Button und jeder Wert eines Formularfelds aufgenommen - allerdings nur das was auch vom User ausgeht. Voreingestellte Sachen werden nicht aufgenommen. Man kann also sagen, dass Selenium nur das Delta zwischen Start der Aufnahme und dem Ende aufnimmt.

Wenn die Seite abgearbeitet wurde für den Test, kann die Aufnahme beendet werden. Sofort danach kann überprüft werden, ob der Test auch das macht was er soll. Über `Files -> Export to...` kann der Test in mehrere Programmiersprachen übersetzt werden. Da wir uns hier im Perl-Bereich bewegen, ist es wohl klar in welche Sprache hier der Test exportiert werden soll.

Die `Selenium IDE` ist einfach und intuitiv, so dass man sich nicht erst einarbeiten muss. Installieren und einfach loslegen...

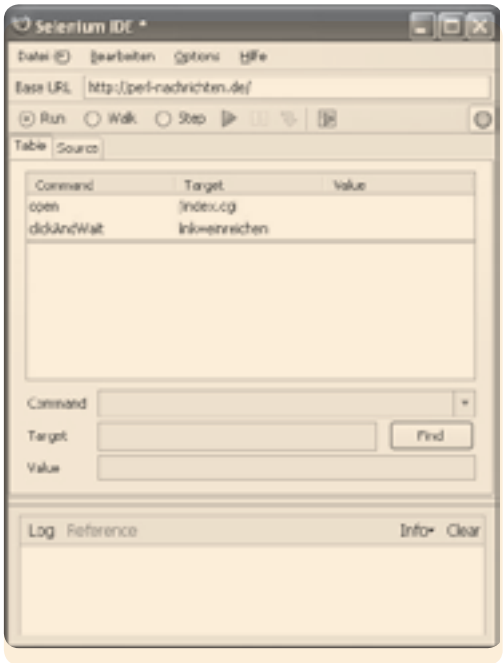

Abbildung 1: Selenium IDE

Tests als Perl-Skript mit Selenium RC

Die IDE eignet sich nicht so gut, um dauerhaft die Tests für die Webanwendung durchzuführen; sonst müsste man jeden aufgenommenen Test einzeln öffnen und ablaufen lassen. Das wird ganz schön stressig. Um sich diese Arbeit zu sparen, kann der `Selenium Server` mit den Perl-Modulen verwendet werden. Dazu muss das Modul `Test::WWW::Selenium` installiert werden.

Wurden die Tests mit der IDE aufgenommen und als Perl-Skript exportiert, müssen noch ein paar Kleinigkeiten geändert werden, damit sie funktionieren. Als erstes sollte die Shebang eingefügt werden. Weiterhin muss die Zeile 11 aus Listing 1 angepasst werden.

In der Browser-URL muss die Domain stehen, auf der das zu testende Skript läuft. In diesem Fall `http://perl-nachrichten.de`. Wenn das `localhost:444` drin bleibt, funktionieren die Tests nicht mehr. Dann taucht im Browser ein `Proxy Error` auf - dann keine Angst, die Anwendung ist nicht unbedingt fehlerhaft.

Nachdem die Skripte angepasst wurden, kann der Selenium-Server über `java -jar selenium-server.jar` gestartet werden. Der Server dient bei den Tests als Proxy und nach dem Start werden als erstes ein paar allgemeine Informationen ausgegeben.

In einer weiteren DOS-Box oder einem weiteren Terminal können die Testskripte aufgerufen werden. Beim ersten Kontakt mit Selenium wären jetzt zwei Monitore am Rechner sehr interessant. Auf einem Monitor den Browser, der durch den Selenium Server gestartet wird und auf dem anderen die zwei Fenster - das Fenster mit den Ausgaben des Servers und das Fenster in dem das Testskript aufgerufen wurde.

Der Browser ist in drei Teile aufgeteilt (Abbildung 2). In der linken oberen Ecke ist das Logo von Selenium zu sehen. Weiterhin gibt es dort zwei Tools, die man benutzen kann, um sich zum Beispiel die Logs anzusehen. In diesem Teil ist auch die Session-ID zu sehen, die bei jedem Test vergeben wird, um das Formular zu identifizieren.

Rechts oben sind die letzten Befehle des Test-Skripts zu sehen. Dort kann man verfolgen, welche Tests gerade ausgeführt werden und durch die farbige Hinterlegung ist schon auf die Schnelle zu erkennen, ob diese Tests erfolgreich waren oder nicht. Und der größte Teil ist für die Webseite vorgesehen. So kann live mitverfolgt werden, was im Browser passiert.

Abbildung 2: Selenium Browser

```
use strict;
use warnings;
use Time::HiRes qw(sleep);
use Test::WWW::Selenium;
use Test::More "no_plan";
use Test::Exception;

my $sel = Test::WWW::Selenium->new( host => "localhost",
                                    port => 4444,
                                    browser => "*firefox",
                                    browser_url => "http://localhost:4444" );

$sel->open_ok("/index.cgi/feedback");
$sel->type_ok("submittermail", "perl\@renee-baecker.de");
$sel->type_ok("message", "Testfeedback");
$sel->type_ok("submitter", "Renee Baecker");
$sel->click_ok("//input[\@value='Nachricht abschicken']");
$sel->wait_for_page_to_load_ok("30000");
```

Listing 1

In der Abbildung 3 ist die Ausgabe des Servers zu sehen. Dort werden alle Requests und Return-Codes ausgegeben. Zusätzlich werden dort noch - je nach eingestellten Debug-Level - weitere Informationen ausgegeben.

Im Fenster des Testskripts erscheint die Ausgabe im TAP-Format. TAP ist das `Test Anything Protocol`, das mittlerweile in vielen Programmiersprachen umgesetzt wird. Durch dieses Protokoll ist es möglich, einheitliche Ausgaben für Testskripte zu erzeugen. Durch das TAP ist auch schnell ersichtlich, welche Tests fehlschlagen.

Prüfungen

Mit Selenium können unterschiedliche Sachen getestet werden. Zum Einen kann schon das Ausfüllen des Formulars überprüft werden. Wie in Listing 1 erkennbar, wird schon ein Klick mit `click_ok` und ein Select-Feld mit `select_ok` überprüft. Aber auch andere Tests, die ähnlich zu `Test::WWW::Mechanize` sind, werden zur Verfügung gestellt. Mit `content_like` oder `title_is` können auch die Daten, die im Browser angezeigt werden, überprüft werden.

Warum Selenium?

Der große Vorteil - bei manchen Umgebungen sicherlich auch ein Nachteil - ist, dass Selenium einen Browser für die Tests verwendet. So können auch gleich - mit minimalen Anpassungen - Cross-Browser-Checks durchgeführt werden.

Da Java ebenso wie Perl auf sehr vielen Plattformen läuft, kann Selenium auf eben diesen Plattformen eingesetzt werden. Selenium unterstützt auch alle modernen Browser wie `Internet Explorer`, `Mozilla Firefox`, `Opera`, `Konqueror`, `Safari` und noch einige mehr.

Da ein Browser verwendet wird, können auch JavaScript-Komponenten getestet werden, wie es bei `Test::WWW::Mechanize` nicht möglich ist.

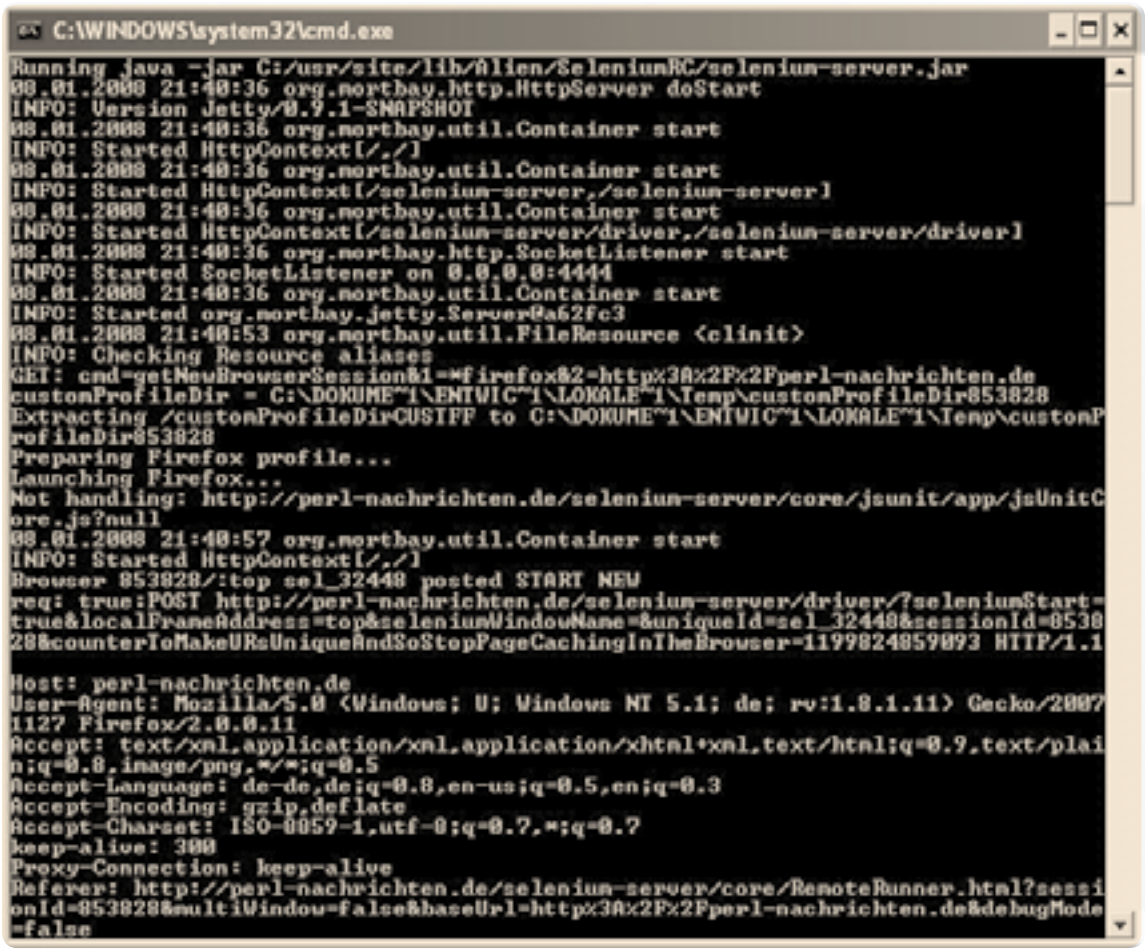

Abbildung 2: Selenium Server

Achtung Falle!

Ein Problem kann sein, dass man zwar Firefox oder einen anderen Browser installiert hat, ein Testskript findet den Browser aber nicht - und dass obwohl man selbst den Browser über die Shell oder mit einem Doppelklick auf das Icon starten kann.

Dann liegt es meistens daran, dass der Pfad zu dem Browser-Verzeichnis nicht in der `PATH`-Umgebungsvariablen liegt. Unter Windows kann dies temporär für die DOS-Box über `set PATH=%PATH%;C:\Pfad\` gelöst werden oder dauerhaft über `Start` -> `Systemverwaltung` -> `System` -> `Erweitert` -> `Umgebungsvariablen`.

Eine andere Möglichkeit ist es, im Testskript einfach `$ENV{PATH} .= ';C:\Pfad'` zu schreiben.

```
#!/usr/bin/perl

use Time::HiRes qw(sleep);
use Test::WWW::Selenium;
use Test::More "no_plan";
use Test::Exception;
use DBI;

#... Datenbank-Abfragen, in %tests werden zu jedem Feld
# alle möglichen Werte gespeichert

my $sel = Test::WWW::Selenium->new(
    browser     => "*firefox",
    browser_url => "http://perl-nachrichten.de",
);

do_tests( \%tests, $sel );

sub do_tests{
   my ($testref, $sel) = @_;
   my $fields = qw/submitter submittermail ... /;
   my @combs  = _get_combinations( $fields, $testref );
   _fill_fields( $_, $sel ) for @combs;
}

sub _fill_fields{
    my ($comb,$sel) = @_;
    $sel->open_ok("/index.cgi/news_form");
    for( qw/submitter submittermail headline newstext/ ){
        $sel->type_ok($_, $comb->{$_});
    }
    $sel->select_ok("catid", "value=".$comb->{catid});
    $sel->click_ok("//input[\@value='Neuigkeit einreichen']");
    $sel->wait_for_page_to_load_ok("30000");
}

sub _get_combinations{
   my @keys  = @{ +shift };
   my %tests = %{ +shift };

   my $first   = shift @keys;
   my @combs   = map{ { $first => $_->{value} } }@{$tests{$first}};

   for my $key ( @keys ){
      my @array = map{ $_->{value} }@{$tests{$key}};
      my @temp;

      for my $elem ( @combs ){
         my @xyz = map{ { %$elem, $key => $_ } }@array;
         push @temp, @xyz;
      }

      @combs = @temp;
   }
   return @combs;
}
```

Listing 2

Testinformationen speichern

In einem Projekt wurden viele nahezu identische Anwendungen geschrieben, so dass ein Großteil der Tests identisch sein sollte. Es war anzunehmen, dass ein Bug in der einen Anwendung genauso in einer anderen Anwendung aufgetreten ist. Und bei über 70 Anwendungen wurde schon mal das eine oder andere Skript vergessen. So kam dann die Idee auf, Testinformationen - also welches Feld mit welchen Daten gefüttert werden sollte - in einer Datenbank zu speichern. Ziel war es, dass die Testskripte ziemlich generisch sein sollten. Wenn der eine Entwickler in einem Skript einen Fehler gefunden hat, sollte ein Testfall geschrieben und die Werte in die Datenbank eingetragen werden. Alle Entwickler greifen auf die gleiche Datenbank zu, so dass immer alle aktuellen Testfälle in einen Test mit einfließen, ohne dass ein „unbeteiligter“ Entwickler sein Testskript anpassen muss.

Was dabei allerdings extrem wichtig ist, ist die Tatsache, dass bei der Namensgebung der Formularfelder keine Fehler passieren, damit nicht aus Versehen falsche Daten zum Testen genommen werden. Aber das fällt ziemlich schnell auf!

Wenn ein Testskript gestartet wird, holt es sich die notwendigen Informationen aus der Datenbank, kombiniert die Werte um ein möglichst großes Spektrum an Tests abzudecken und führt die ganzen Tests aus.

In Listing 2 ist ein Beispiel-Skript zu sehen.

Für eine einzige Anwendung ist das Ganze zu aufwändig. Dort ist es besser mit einem Versionskontrollsystem wie SVN oder CVS zu arbeiten und damit alle Entwickler auf einem aktuellen Stand zu halten. Für mehrere Anwendungen, die nahezu identisch sind, aber keine gemeinsamen Module nutzen, ist das eine ganz gute Lösung.

Einen kleinen Nachteil hat diese Umsetzung: Wenn ein Test fehlschlägt ist auf Grund der dynamischen Zusammensetzung der Testwerte nicht sofort ersichtlich, bei welchen Eingabedaten der Test fehlgeschlagen ist. Deshalb wird jedes Skript einzeln ausgeführt - aber auch das wird mit einem Perl-Skript erledigt. `make test` wird nicht verwendet, da dort nur eine „Zusammenfassung“ erscheint.

Ein weiteres Problem damit besteht darin, dass nicht 100%ig alle Testbedingungen dargestellt werden können. Dafür müssen weitere Tabellen oder extra Skripte geschrieben werden. Diese Methode hat sich jedoch für die Fälle bewährt, in dem der Ablauf relativ einfach ist.

Renée Bäcker

TPF-TICKER

TPF erhält große Spende

Die Perl Foundation hat 200.000 US$ für die Entwicklung von Perl 6 von Ian Hague, dem Mitgründer von „Firebird Management LLC“, bekommen.

Ungefähr die Hälfte des Geldes wird über Grants und andere Wege an Perl 6 Entwickler gegeben.

Ian Hague wird bei der Vergabe der anderen Hälfte mitentscheiden.

Großer Dank an Ian Hague!

Planet YAPC

Karen und Marty Pauly haben eine Seite aufgesetzt, auf der alle Infos zu YAPCs zu finden sind: http://planet.yapc.org/.

Web::Scraper

Parsen von Webseiten ist ein wichtiges Thema - auch in einigen Module ist das ein entscheidender Teil des Codes. Aber HTML selbst zu Parsen zählt wohl zu den komplexesten Dingen. Gerade Einsteiger versuchen sich dann oft mit Regulären Ausdrücken (Listing 1). Damit kommen sie aber nicht allzu weit, weil das HTML der Webseiten nicht immer sauberes HTML ist.

Und was, wenn sich das Design auch nur ein klein wenig ändert? Dann ist man mit den Regulären Ausdrücken immer dabei den Code anzupassen, weil hier die Reihenfolge der Tags nicht stimmt. Oder es wurde etwas anderes geändert, wie z.B. eine neue CSS-Klasse hinzugefügt. Für den Nutzer am Browser ist das häufig nicht zu sehen, aber das HTML-Gerüst hat sich geändert.

Wer auf Reguläre Ausdrücke verzichten will, kann sich ein Modul auf CPAN aussuchen - es gibt einige, die HTML parsen können. Eines der bekanntesten dürfte HTML::Parser von Gisle Aas sein (Listing 2).

```
#!/usr/bin/perl

use strict;
use warnings;

my $content = do{ local $/; <DATA> };
my ($description) = $content =~ /< meta \s+
    name="description" \s+
    content="(.*?)"/x;
print $description;

__DATA__
<html>
  <head>
      <name="description"
       content="Beschreibung">
  </head>
  <body>
      <h1>Test</h1>
  </body>
</html>
```

Listing 1

Die HTML::Parser-Lösung ist schon einiges sauberer, aber eher umständlich und lang. Hier ist die Wartbarkeit durch globale Variablen und die „Länge“ etwas eingeschränkt. Soll nicht nur 1 Tag gesucht werden, sondern viele verschiedene, werden die Handler auch dementsprechend komplex. Allerdings kann hier noch auf bestimmte Ereignisse „reagiert“ werden. Hier lassen sich weitergehende Bedingungen einbauen.

Ein Vorteil von HTML::Parser ist auch die ausgereifte Entwicklung. Das Modul gibt es schon seit einiger Zeit, so dass Bugs wohl eher selten auftreten dürften.

Mit Web::Scraper gibt es ein Modul, mit dem sich solche Aufgaben flexibel und einfach lösen lassen. Das Modul ist der Perl-Port des Ruby-Skripts „scrapi.rb“. Für das Heraussuchen der Informationen kann zwischen der XPath- und der CSS-Syntax gewählt werden. In diesem Artikel wird nur mit der XPath-Syntax gearbeitet.

```
#!/usr/bin/perl

use strict;
use warnings;
use HTML::Parser;

# Im DATA-Bereich soll der gleich HTML-
# Code stehen wie im RegEx-Beispiel
my $content     = do{ local $/; <DATA> };
my $description = "";
my $parser      = HTML::Parser->new();

$parser->handler(
       start => \&_start, "tagname,attr" );
$parser->parse( $content );

print $description;

sub _start{
    my ($tag,$attr) = @_;
    return unless $tag eq 'meta';
    return unless $attr->{name}
                    eq 'description';
    $description = $attr->{content};
}
```

Listing 2

```
#!/usr/bin/perl

use strict;
use warnings;
use Web::Scraper;

# Im DATA-Bereich soll der gleich HTML-
# Code stehen wie im RegEx-Beispiel
my $content     = do{ local $/; <DATA> };
my $parser      = scraper {
    process 'meta[name="description"]',
   description => '@content';
};
my $result = $parser->scrape( $content );
print $result->{description};
```

Listing 3

In Zeile 10 in Listing 3 wird ein neues Parser-Objekt erzeugt. Der Funktion scraper wird ein Code-Block übergeben, in dem eine Art „Domain Specific Language“ steht. Hier werden die ganzen `process`-Anweisungen bestimmt, wobei der erste Parameter für process die Tags bestimmt, für die Informationen herausgesucht werden sollen. Im Beispiel von Listing x ist es ein ‚meta‘-Tag, das das Attribut ‚name‘ hat mit dem Wert ‚description‘. Hier kann man schon einen Vorteil des Moduls gegenüber Regulären Ausdrücken erkennen: Es ist egal, ob das Tag noch mehr Attribute hat oder nicht. Auch die Reihenfolge der Attribute spielen keine Rolle.

Als weitere Parameter erwartet `process` einen Hash. Die Schlüssel in diesem Hash sind die Schlüssel, anhand derer die Informationen am Schluss ausgelesen werden können. Als Wert kann man eine „Beschreibung“ übergeben. Diese Beschreibung kann ein Attribut - erkennbar an dem ‚@‘ sein, eine Sub-Referenz, ein neues `Web::Scraper`-Objekt oder ‚TEXT‘ für den Text der zwischen den Tags steht.

Die Funktionen `scraper` und `process` kann man beliebig schachteln. Das ist für Blöcke wichtig. Wenn es zum Beispiel mehrere div-Blöcke gibt, in denen es Links gibt. Es werden aber nur die Links im div-Block mit der id ‚content‘ benötigt, dann muss es erst ein `process` für div‘s geben und dann ein `scraper` mit `process` für die Links (Listing 4).

Als Ergebnis des Filterns wird eine Hashreferenz zurückgegeben, in der alle Informationen gespeichert sind. In dem Beispiel hat der Hash nur einen Schlüssel - ‚description‘ - und den entsprechenden Wert.

```
my $parser = scraper {
    process 'div[id="content"]',
   'results[]' => scraper {
        process 'a', 'url' => '@href';
    };
};
```

Listing 4

```
my $parser = scraper {
    process 'div[id="content"]',
   'results[]' => scraper {
        process 'a', 'url' => '@href';
    };

    result 'results'
};
```

Listing 5

result

Den Rückgabewert von `scrape` ist standardmäßig eine Hashreferenz mit allen Informationen. Dies ist häufig aber nicht gewünscht - vor allem wenn man in der Hashreferenz nur einen Schlüssel hat. In diesen Fällen kann der Rückgabewert verändert werden.

Im Beispiel von Listing 5 wird eine Hashreferenz zurückgeliefert, die so ähnlich wie in Listing 6 aussieht.

```
$VAR1 = {
    results => [
                    'element',
                ],
}
```

Listing 6

Durch das Einfügen von `result 'results'` - die `[]` müssen weggelassen werden - wird die Arrayreferenz zu dem Schlüssel ‚results‘ zurückgeliefert. Dann sieht der Rückgabewert wie in Listing 7 aus.

```
$VAR1 = [
      'element',
]
```

Listing 7

Sonderfälle

Bei manchen Tags (script, style) funktioniert die Verwendung von ‚TEXT‘ nicht. In solchen Fällen muss man ‚RAW‘ nehmen. Ein Beispiel dafür ist in Listing 8 zu sehen.

```
#!/usr/bin/perl

use strict;
use warnings;
use Web::Scraper;

# Im DATA-Bereich soll der gleich HTML-
# Code stehen wie im RegEx-Beispiel
my $content     = do{ local $/; <DATA> };
my $parser      = scraper {
    process 'script', 'code[]' => 'RAW';
};
my $result = $parser->scrape( $content );
print $_,"\n" for @{ $result->{code} };

__DATA__
<html><head>
  <script>alert('Test')</script>
</head></html>
```

Listing 8

CLI

Wer Perl-Einzeiler liebt und „mal eben schnell“ eine Webseite parsen will, für den bietet Web::Scraper ein kleines Programm, mit dem man auf der Kommandozeile das HTML parsen kann. Dazu wird das Programm `scraper` in der Kommandozeile mit einer URL als Parameter gestartet. Danach können die einzelnen `process`-Befehle eingegeben werden. Diese sehen genauso aus wie in einem Perl-Skript. Zu beachten ist hierbei, dass bei einem Zugriff auf Attributwerte der Attributname wegen des `@` in einfachen Anführungszeichen stehen muss.

Mit den Befehlen `d` und `y` kann man sich die gesammelten Informationen im Data::Dumper- beziehungsweise im YAML-Format anzeigen lassen. Wie einfach alle Links aus einer Seite ausgegeben werden können, zeigt Listing 9.

Ingesamt kann man sagen, dass sich Web::Scraper sehr gut zum Parsen von Webseiten eignet. Wer noch mehr Einfluss darauf haben will, was bei bestimmten Tags passiert, der sollte besser ein Modul wie HTML::Parser verwenden. Sollen aber einfach nur Informationen aus dem HTML gezogen werden, kommt mit Web:Scraper meist schneller zum Ziel..

Die Dokumentation von Web::Scraper ist dürftig. Bei bestimmten Konstellationen ist „Spielen“ angesagt: Wie muss der `process`-Befehl aussehen, damit ich meine Informationen bekomme? Gibt es einen einfachen Befehl, mit dem ich an die Information komme? Viele nützliche Tipps finden sich im use.perl.org-Journal von Tatsuhiko Miyagawa.

Renée Bäcker

```
C:\>scraper http://www.foo-magazin.de

scraper> process 'a', 'urls[]' => '@href';

scraper> d
$VAR1 = {
  'urls' => [
    'http://www.foo-magazin.de/index.cgi?sid=581956d5de5c43773938f7fa6ef1701c',
    undef,
    '#d8',
    '?action=index;issue=8;position=8;sid=',
    '#d13',
    '?action=index;issue=8;position=13;sid=',
    '#d16',
    '?action=index;issue=8;position=16;sid=',
    #... noch mehr Links
    '?action=thema;sid=581956d5de5c43773938f7fa6ef1701c',
    '?action=imprint;sid=581956d5de5c43773938f7fa6ef1701c',
  ]
};

scraper>
```

Listing 9

Scaffolding?

Als vor einiger Zeit Ruby on Rails erstmals die Bühne betrat, war dies nicht nur technisch beeindruckend, sondern die Autoren haben auch eine perfekte Präsentation ihrer Ideen geliefert. Dazu gehörten insbesondere die kleinen Videos in denen man sehen konnte, wie mit Rails innerhalb weniger Minuten ein einfaches Gerüst für eine Webapplikation gebaut werden kann. Diese Technik wird als „scaffolding“ bezeichnet, auf deutsch: „Gerüste bauen“. Man kann sicher geteilter Meinung über Rails sein, aber in Folge erschienen jede Menge neuer Frameworks für fast alle üblichen Skriptsprachen, die viele der Ideen aus Rails aufgriffen und weiter entwickelten.

Ob man ein Framework für die Programmierung von Webapplikationen nutzt, oder eher „klassisch“ an entsprechende Problemstellungen herangehen möchte, das muss jeder für sich entscheiden. Die Frameworks bieten viel, aber zu dem Preis, dass man sich auf ihre Konzepte voll einlassen muss. Wem das zu unflexibel ist, oder wer den Einarbeitungsaufwand meiden möchte, kommt aber nicht umhin ein wenig neidisch auf die Werkzeuge zu schauen, die diese Frameworks mitbringen. Die Möglichkeit ein einfaches Grundgerüst zu generieren, dass einem einen Teil der Arbeit abnimmt, insbesondere aber stetig wiederholende Tipp-Arbeit, übt doch einen besonderen Reiz aus.

Vor einiger Zeit entschied ich mich selbst ein einfaches Skript zu entwickeln, das mir einen Teil der wiederkehrenden Arbeiten abnehmen sollte und es mir erlaubt schneller und effizienter zu entwickeln. Es handelt sich im wesentlichen um einen „proof-of-concept“, also kein vollständig ausgereiftes Werkzeug. Es beinhaltet noch etliche Lücken, auf die ich im folgenden zum Teil auch eingehen werde.

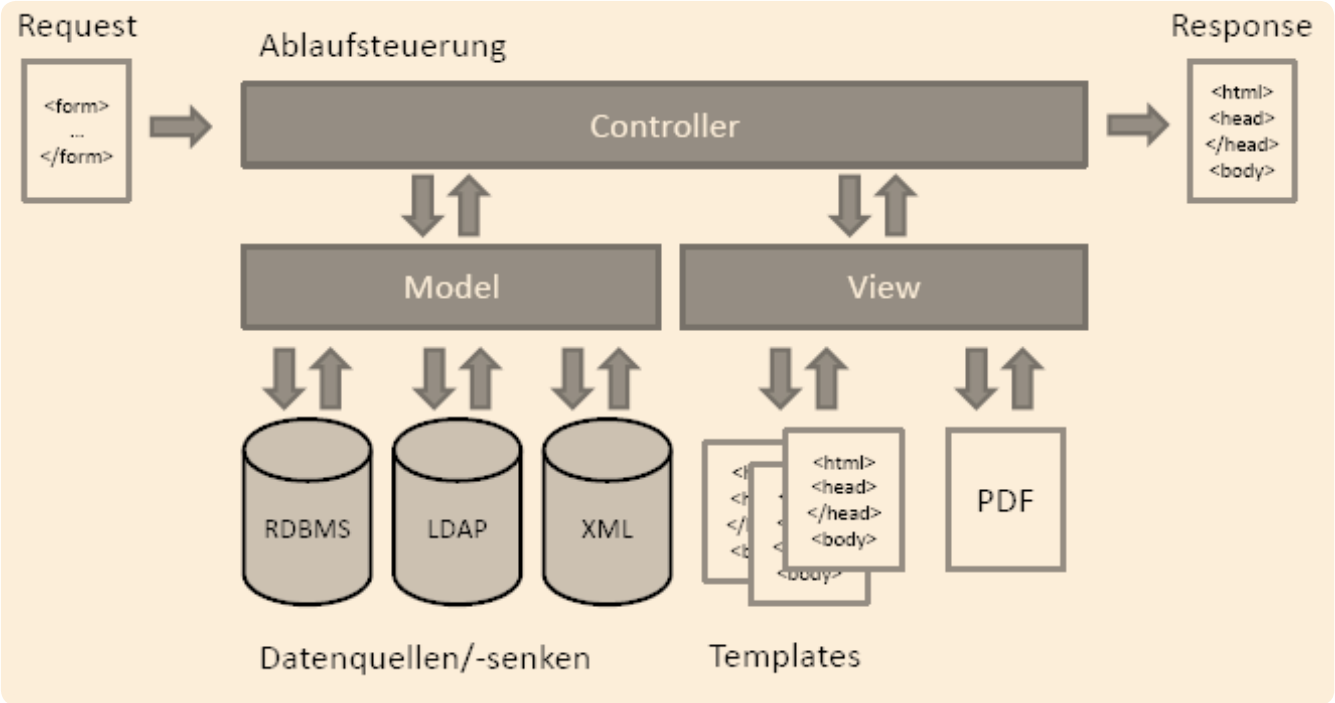

Abbildung 1: MVC

MVC

Ein Konzept, dass heute bei nahezu jeder Webanwendung Berücksichtigung findet, ist das Entwurfsmuster MVC. MVC steht für Modell-View-Controller, wobei die Grundidee dabei ist Ablauflogik, Repräsentation von Daten und Zugriff auf Daten voneinander logisch zu trennen. Wie stark man diese Abstraktion umsetzt und ob dieses Paradigma immer zu 100% befolgt werden kann, sei jedem selbst überlassen. Aber eine Orientierung an der Grundidee gehört heute zum „guten Ton“ in der Entwicklung von Webanwendungen. Deshalb folgt das, von *scaffold.pl* zu erzeugende, Grundgerüst einer Webapplikation, auch diesem Muster.

Am Anfang steht die Datenbank...

Eine der wesentlichen Feststellungen ist, dass eigentlich die wesentlichen Informationen die man benötigt bereits in der jeweiligen Datenbank existieren. Dort finden sich Metaangaben zu Tabellen und deren Feldern. MySQL erlaubt diese sogar via SQL abzufragen, mit dem DESCRIBE-Befehl.

```
DESCRIBE table;
```

Das Skript *scaffold.pl* ist das zentrale Element, entsprechend ist es auch genau der Punkt an dem eine Datenbank-Verbindung aufgebaut werden muss und die Information zur jeweiligen Tabelle abgefragt wird. Die notwendigen Konfigurationsinformationen für den Verbindungsaufbau, werden aus einer .yaml-Datei gelesen, die im config-Verzeichnis abgelegt sein muss:

```
---
host: localhost
schema: foo
user: root
passwd: secret
```

Mit diesen Informationen und dem von der Kommandozeile ausgelesenen Tabellen-Namen wird die Datenbank-Verbindung aufgebaut (siehe Listing 1).

Der DESCRIBE-Befehl liefert Informationen über die Spalten der Tabelle, die Typen, die Längenbeschränkungen usw. - zum jetzigen Zeitpunkt werden aber nur die Spaltennamen genutzt und deshalb per `map` in der letzten Zeile des abgebildeten Ausschnitts herausgezogen. Der Inhalt der Variable $cols ist ein anonymes Array mit den Namen der jeweilgen Spalten:

```
$cols = [
    'ID',
    'name',
    'yada yada yada',
];
```

Diese Informationen dienen als Basis für die folgenden Schritte. Es sei angemerkt, dass die Informationen aus der Datenbank, auf die hier erstmal verzichtet wurden, sinnvollerweise genutzt werden sollten um eine Validierung übergebener Daten zu bewerkstelligen. Aus Komplexitätsgründen habe ich hier erstmal darauf verzichtet.

Entitäten und Mengen

Die gewonnenen Informationen über die Spalten der jeweiligen Tabelle sollen auf Objekte abgebildet werden, die im nächsten Schritt genutzt werden um die jeweiligen Modelle zu erzeugen, welche später die Abstraktionsschicht zur Datenbank bilden. Ich habe mir angewöhnt, zwei Klassen zu bilden: Eine die jeweils eine Spalte einer Tabelle repräsentiert (eine Entität) und eine Klasse, die eine Menge dieser Entitäten darstellt und in einer Liste abbildet. Entsprechend benötige ich zum `scaffolding` auch zwei Klassen, die eben jene Klassen für das Modell erzeugen können: *ScaffoldRow* und *ScaffoldSet*.

```
my $config = LoadFile( "config/database.yaml" );
my $table  = shift @ARGV || 'test'; # better use GetOpt::Long instead

my $dsn = "DBI:mysql:database="
        . $config->{schema} . ";host="
        . $config->{host} . ";port=3306";

my $dbh = DBI->connect($dsn, $config->{user}, $config->{passwd})
    or die "can't connect to database!";

my $cols = [ map { $_->[0] } @{$dbh->selectall_arrayref("DESCRIBE table")} ];
```

Listing 1

```
package ScaffoldRow;
use Moose;
use HTML::Template::Compiled;

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

sub as_list {
    my $self = shift;
    my $o = join( ' ', grep { $_ ne 'ID' } @{$self->columns} );
    return $o;
}

sub as_self_list        { # aus Platzgründen entfernt ... }
sub as_param_list       { # aus Platzgründen entfernt ... }
sub as_moose_properties { # aus Platzgründen entfernt ... }

sub to_perl {
    my $self = shift;
    my $htc = HTML::Template::Compiled->new(filename => 'code_templates/row.pl.tmpl');
    $htc->param( table => $self );
    return $htc->output;
}

1;
```

Listing 2

```
package M::<%= table.name %>;
use Moose;

<%= table.as_moose_properties %>
has 'dbh' => ( is => 'rw' );

sub create {
    my $self = shift;
    # return unless $self->name =~ /^\w+$/; # without name, nothing to add
    die "M::<%= table.name %>::create() has no DBH!" unless ref $self->{dbh};

    $self->{dbh}->do("INSERT INTO <%= table.name %> SET <%= table.as_param_list %>",
        undef, <%= table.as_self_list %>)
        or die $self->{dbh}->errstr;
}

sub retrieve    { # aus Platzgründen entfernt ... }
sub update      { # aus Platzgründen entfernt ... }
sub del         { # aus Platzgründen entfernt ... }
sub sanitize    { # aus Platzgründen entfernt ... }

sub as_edit_form {
   my $self = shift;
   my $o    = '
   <h3>Eintrag bearbeiten</h3>
   <form>
   <input type="hidden" name="ID" value="' . $self->ID . '" />
       <table>';
   $o .= "<tr><td>$_: </td><td><input name='$_' value='".$self->$_()."'/></td></tr>"
       for qw/ <%= table.as_list %> /;
   $o .= '<tr><td colspan="2"><input type="submit" name="action" value="update" /></td></tr>
       </table>
   </form>
       ';
       return $o;
   }

   sub as_form     { # aus Platzgründen entfernt ... }

   1;
```

Listing 3

ScaffoldRow

ScaffoldRow (siehe Listing 2) hat fünf Methoden. Die letzte Methode `to_perl()` wird genutzt um ein Code-Template zu befüllen, wobei als einziger übergabeparameter eine Referenz auf *$self* übergeben wird, um die anderen vier Methoden innerhalb des Templates aufrufen zu können. Diese fügen die Informationen über die Spalten der Tabelle in Form von Listen, SQL-Parametern oder Attributen für Moose in das Template per Methodenaufruf ein. Das zugehörige Code-Template entspricht der Klasse, die jeweils eine Spalte/Entität der Tabelle repräsentiert (siehe Listing 3)

Es wird eine Klasse erzeugt, mit vier Methoden `create`, `retrieve`, `update` und `del`, die die Standard-Operationen zur Datenbank hin abbilden. Zwei weitere wichtige Methoden dienen der Erzeugung von HTML-Formularen zum Bearbeiten oder Erzeugen einer Entität der Tabelle. Das Template selbst ist ein normales HTML::Template::Compiled-Template. Man sieht wie innerhalb der ASP-Style Tags die jeweiligen Methodenaufrufe aus **ScaffoldRow** erfolgen.

ScaffoldSet

ScaffoldSet (siehe Listing 4) hat nur zwei Methoden: `as_list` und `to_perl`. Es erstellt die Klasse des Modells, die mit der ganzen Tabelle arbeitet, bzw. die Interaktion mit der Tabelle steuert.

Das Code-Template stellt wieder mehrere Methoden bereit, insbesondere Methoden zur Abfrage einer Menge von Datensätzen auf Basis bestimmter Kriterien, und Methoden zur Ausgabe der Datensätze als HTML-Tabelle, unter Verwendung des Moduls **HTML::Table** (siehe Listing 5).

Erzeugung der Modell-Dateien in scaffold.pl

Innerhalb des Skriptes scaffold.pl werden Verzeichnisse unterhalb von **/draft** erzeugt, für Modelle, Views und sonstige Dateien. Im folgenden Abschnitt sieht man auch die Verwendung der Klassen **ScaffoldRow** und **ScaffoldSet** bei der Erstellung der beiden Klassen des Modells (siehe Listing 6).

View - ein generisches Template

Aus praktischen Erwägungen sollte das Template, das durch das scaffolding erzeugt wird, generisch einsetzbar sein, ohne einen konkreten Bezug zum jeweiligen Modell zu haben. Auf dieser Basis können später beliebige Anpassungen vorgenommen werden (siehe Listing 7).

Hieraus ergibt sich auch der Vorteil eines sehr kompakten Templates, was sehr zur Lesbarkeit beiträgt. Dies ist aber ein Vorteil, der prinzipiell immer bei der Verwendung von Templating-Modulen entsteht, die wie HTML::Template::Compiled die Nutzung von Methoden-Aufrufen innerhalb des Templates erlauben. Das `$set` stellt eine Methode zur Verfügung die ein Suchformular erzeugt. Selbst wenn also noch keiner Ergebnissmenge als Folge einer Abfrage erzeugt wurde, ist es im eigentlichen CGI notwendig ein leeres **$set**-Objekt zu erzeugen, um den Methodenaufruf zu ermöglichen. Die Methode **as_html_table** gibt eine HTML-Tabelle aus, die durch Verwendung des Moduls HTML::Table erzeugt wird.

```
package ScaffoldSet;
use Moose;
use HTML::Template::Compiled;

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

sub as_list {
    my $self = shift;
    my $o = join( ' ', grep { $_ ne 'ID' } @{$self->columns} );
    return $o;
}

sub to_perl {
    my $self = shift;
    my $htc = HTML::Template::Compiled->new(filename => 'code_templates/set.pl.tmpl');
    $htc->param( table => $self );
    return $htc->output;
}

1;
```

Listing 4

```
package M::set_<%= table.name %>;
use Moose;
use HTML::Table;
use M::<%= table.name %>;

has 'set'   =>      ( isa => 'ArrayRef', is => 'rw' );
has 'dbh'   =>      ( is => 'rw' );

sub retrieve_all    {  # aus Platzgründen entfernt ... }

sub retrieve {
    my $self = shift;
    my $what    = shift;
    $what =~ s/(.*)/%$1%/;
    my $search  = shift || 'title';
    die "unknown field!" unless grep { $search eq $_ } qw/ <%= table.as_list %> /;

    my $sth = $self->{dbh}->prepare(
        "SELECT * FROM <%= table.name %> WHERE $search LIKE ? ORDER BY ?"
    );
    $sth->execute($what, $search);
    while (my $ref = $sth->fetchrow_hashref()) {
        my $entity = M::<%= table.name %>->new(
            ID => $ref->{'ID'},
            dbh => $self->{dbh},
        );
        $entity->{$_} = $ref->{$_} for qw/ <%= table.as_list %> /;
        push @{$self->{set}}, $entity;
    }
    $sth->finish();
}

sub as_html_table {
    my $self = shift;
    my $t = HTML::Table->new;
    foreach my $row (@{$self->{set}}) {
        $t->addRow(
            '<a href="<%= table.name %>.pl?action=delete&ID=' . $row->ID . '">
            <img src="images/Delete.png" width="24" height="24" alt="Delete" /></a>',
            '<a href="<%= table.name %>.pl?action=edit&ID=' . $row->ID . '" >
            <img src="images/Modify.png" width="24" height="24" alt="Modify" /></a>',
            $row->ID,
            map { $row->$_ } qw/ <%= table.as_list %> /
        );
    }
    for (0 .. $t->getTableRows) {
        my $class = $_ % 2 == 0 ? 'even' : 'odd';
        $t->setRowClass($_, $class);
    };
    $t->setClass('results');
    return $t->getTable;
}

sub as_search_form  { # aus Platzgründen entfernt ... }

sub as_tuple        { # aus Platzgründen entfernt ... }

1;
```

Listing 5

```
mkdir './draft' or warn $!   unless -e './draft';
mkdir './draft/M' or warn $! unless -e './draft/M';
mkdir './draft/V' or warn $! unless -e './draft/V';
mkdir './draft/images' or warn $! unless -e './draft/images';

print "Creating MODEL ./draft/M/$table.pm ...\n";
open( my $r_fh, '>', "./draft/M/$table.pm" ) or warn $!;
    my $sc_row = ScaffoldRow->new( name => $table );
    $sc_row->columns($cols);
    print $r_fh $sc_row->to_perl;
close $r_fh;

print "Creating MODEL ./draft/M/set_$table.pm ...\n";
open( my $s_fh, '>', "./draft/M/set_$table.pm" ) or warn $!;
    my $sc_set = ScaffoldSet->new( name => $table );
    $sc_set->columns($cols);
    print $s_fh $sc_set->to_perl;
close $s_fh;
```

Listing 6

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta name="generator" content=
  "HTML Tidy for Mac OS X (vers 12 April 2005), see www.w3.org" />

  <title>localhost - local page</title>
  <link rel="stylesheet" type="text/css" href="default.css" />
</head>

<body>
  <h2>Datenbank</h2>
  <%INCLUDE NAME="nav.tmpl"%>
  <div id="main">
    <h3>Suche<img src="images/Search.png" width="24" height="24" alt="Search" /></h3>
    <%= set.as_search_form %>
    <hr />
    <%IF set %>
    <h3>Tabelle</h3>
    <%= set.as_html_table %>
    <hr />
    <%/IF%>

    <%IF edit %>
        <%= entity.as_edit_form %>
    <%ELSE%>
        <%= entity.as_form %>
    <%/IF%>
  </div>
  <%IF message %>
    <div id="message">
      <pre><%= message %></pre>
    </div>
  <%/IF%>
</body>
</html>
```

Listing 7

Das **$entity**-Objekt stellt zwei Methoden zur Verfügung: Die Methode **as_edit_form** zur Bearbeitung eines konkreten Eintrags und die Methode **as_form** um neue Einträge in der Datenbank zu erstellen. Die erste Variante wird nur dargestellt, wenn ein Bearbeitungskontext gewünscht ist, was durch die Verwendung der Variablen **$edit** in einem boolean-Kontext innerhalb des Templates abgefragt wird.

Wird zu debugging-Zwecken eine **$message** übergeben, wird diese am Ende der erzeugten HTML-Seite mit ausgegeben.

Ablaufsteuerung

Das eigentliche CGI, dass die Ablaufsteuerung als Controller übernimmt, wird durch die Klasse **ScaffoldCtrl** erzeugt, wie die Klassen der Modelle auch, aus einem Code-Template (siehe Listing 8).

Das Code-Template ergibt das spätere CGI und erledigt die üblichen Aufgaben, wie das aufbauen der Datenbankverbindung, ein Methoden-Dispatch auf Grundlage der übermittelten Aktion und Ausgabe von HTTP-Header sowie des View-Templates (siehe Listing 9).

Das Ergebnis ...

Scaffolding erlaubt schnell ein funktionierendes Grundgerüst zu bauen. Wie bereits Eingangs erwähnt, ist dies durchaus beeindruckend. Aber wie beim Hausbau, ist das Gerüst nur ein Werkzeug, dass die folgenden Arbeiten vereinfachen soll. Darüberhinaus stellt sich die Frage, wieviel Funktionali-

Abbildung 2: Example

```
package ScaffoldCtrl;
use Moose;
use HTML::Template::Compiled;

has 'name'      => ( is => 'rw');
has 'dsn'       => ( is => 'rw');
has 'dbuser'    => ( is => 'rw');
has 'dbpasswd'  => ( is => 'rw');
has 'columns'   => ( isa => 'ArrayRef', is => 'rw', default => sub {[]} );

sub as_list {
    my $self = shift;
    my $o = join( ' ', grep { $_ ne 'ID' } @{$self->columns} );
    return $o;
}

sub to_perl {
    my $self = shift;
    my $htc = HTML::Template::Compiled->new(filename => 'code_templates/controller.pl.tmpl');
    $htc->param( table => $self );
    return $htc->output;
}

1;
```

Listing 8

tät man bereits duch das „Gerüst“ abgedeckt haben möchte. In meinem Beispiel fehlen wichtige Aspekte die zu gutem Prodktionscode gehören müssen, wie z.B. die Validierung von Benutzereingaben, die Sicherheit vor HTML- bzw. JS-Injection und andere Dinge. Die Scaffolding-Werkzeuge größere Frameworks berücksichtigen diese Problemstellungen. Mir ging es aber primär darum auszuprobieren wie ein solches Werkzeug entwickelt werden kann. Es ist spannend zu sehen wie Dinge „out-of-the-box“ wie magisch funktionieren. Sie verstecken aber auch vieles vor dem Programmierer. Ab und an ist es sinnvoll einen Blick hinter die Kulissen zu werfen, egal ob man eher „klassisch“ entwickelt, oder ein Framework nutzt. Anbei noch ein Bild, dass eine mit dem im Artikel beschriebenen Werkzeug erstellte primitive Webapplikation zeigt (siehe Abbildung 2).

Ronnie Neumann

```
#!/usr/bin/perl

use strict;
use warnings;

use CGI;
use CGI::Carp qw/warningsToBrowser fatalsToBrowser/;
use HTML::Template::Compiled;
use DBI;
use Data::Dumper;

use M::set_<%= table.name %>;
use M::<%= table.name %>;

my $q = CGI->new;
my $action = $q->param('action');

my $dsn = '<%= table.dsn %>';
my $dbh = DBI->connect($dsn, '<%= table.dbuser %>', '<%= table.dbpasswd %>')
       or die "can't connect to database!";

my $set = M::set_<%= table.name %>->new(dbh => $dbh);
my $message = Dumper $q;

my $tmpl = HTML::Template::Compiled->new(filename => 'V/<%= table.name %>.tmpl');

my $controller = {
    'create' => sub {
        my $entity = M::<%= table.name %>->new(
            dbh => $dbh,
            map { $_ => $q->param($_) } qw/<%= table.as_list %>/,
        );
        $entity->create;
        $set->retrieve_all();
        $tmpl->param( set => $set );
    },
    'retrieve'  => sub { # aus Platzgründen entfernt ... },
    'edit'      => sub { # aus Platzgründen entfernt ... },
    'update'    => sub { # aus Platzgründen entfernt ... },
    'delete'    => sub { # aus Platzgründen entfernt ... },
};

if (exists $controller->{$action}) {
    $controller->{$action}()
} else {
    $tmpl->param( set => $set, entity => M::<%= table.name %>->new );
    # yada yada yada
}

$tmpl->param( message => $message );
print $q->header, $tmpl->output;

$dbh->disconnect;
```

Listing 9

autobox - Ruby-Feeling in Perl

Viele Ruby-Fans schwören darauf, dass alles ein Objekt ist. Nicht nur Objekte von Klassen (sog. „first class objects“), sondern auch Zahlen und Strings. Auf CPAN ist das Modul `autobox` zu finden, mit dem sich ein ähnliches Verhalten in Perl realisieren lässt. In Listing 1 ist ein einfaches Beispiel zu sehen.

`autobox` übersetzt die Aufrufe automatisch in `TYP->funktion( Aufrufender, Parameter )`. Bei Skalaren ist der Typ `SCALAR`, bei Hashreferenzen `HASH` usw. Damit autobox weiß, welche Funktion ausgeführt werden soll, muss diese erst einmal für den entsprechenden Typ definiert werden. Da ich hier als Aufrufenden Wert einen Skalar habe, muss ich die Funktion `to` in dem Package `SCALAR` definieren. Dementsprechend kann man auch Funktionen für Hashreferenzen erstellen (siehe Listing 2).

Diese Zuordnung von Skalar auf den Typ kann man auch ändern. Hat man z.B. in einem Modul `MyScalars` Funktionen für einfache Skalare (also keine Referenzen) hinterlegt, kann man `autobox` anweisen bei Skalaren eben dieses eigene Modul zu verwenden. Dies wird beim `import` von `autobox` gemacht:

```
use autobox SCALAR => 'MyScalars';
```

Diese Verhaltensänderung kann auch bei den anderen Datentypen (`HASH`, `ARRAY`,...) durchgeführt werden. Seit der Version 2.30 vom 9. Mai 2008 funktioniert autoboxing auch wenn der Methodenname in einer Variablen gespeichert ist, also

```
my $sub_name = 'some_sub';
1->$sub_name();
```

`autobox` kann man auch - wie `strict` oder `warnings` - „an- und ausschalten“. Das deaktivieren von `autobox` geschieht über `no autobox`.

Und mit `autobox::Core` werden etliche Built-in-Funktionen von Perl so umgeschrieben, dass dass autoboxing für Zahlen und Strings auch mit eben diesen Funktionen klappt (siehe Listing 3).

```
#!/usr/bin/perl

use strict;
use warnings;
use autobox;

sub SCALAR::to{ return $_[0] .. $_[1] };

my @list = 1->to( 5 );
print $_,"\n" for @list;

__END__
rbaecker@foo ~/test $ perl autobox.pl
1
2
3
4
5
```

Listing 1

```
#!/usr/bin/perl

use strict;
use warnings;
use autobox;

sub HASH::print {
    while( my ($key,$val) = each %{$_[0]} ){
        print $key, " -> ", $val, "\n";
    }
}

{test => 1, hallo => 2}->print;
__END__
rbaecker@foo ~/test $ perl hashref.pl
test -> 1
hallo -> 2
```

Listing 2

Man kann im Prinzip für beliebige Module eine „autobox-Erweiterung“ schreiben, wie es z.B. für Date::Time gemacht wurde. Die Verwendung des Moduls wirkt sich allerdings nachteilig auf die Performance aus (siehe Listing 4).

Es gibt noch weitere Sachen zu beachten, so z.B. dass man in manchen Fällen Klammern verwenden muss (z.B. `(\&foo)->subname()`). Aber durch die Verwendung von `autobox` wird manches lesbarer, bzw. die Reihenfolge der Abarbeitung von Befehlen wird deutlicher. Das Modul ist auf jeden Fall - trotz ein paar kleiner Nachteile - sehr interessant, auch wenn ich es (noch) nicht in einem größeren Projekt einsetzen würde.

Renée Bäcker

```
#!/usr/bin/perl

use strict;
use warnings;
use autobox;
use autobox::Core;

'$foo'->uc->print("\n");
__END__
rbaecker@foo ~/test $ perl core.pl
$FOO
```

Listing 3

```
#!/usr/bin/perl

use strict;
use warnings;
use Benchmark qw(cmpthese);

use autobox;
use autobox::Core;

cmpthese(
  100000,
  {
    classic => sub{ print uc('$foo'),"\n" },
    autobox => sub{ '$foo'->uc->print("\n")
},
  }
);
__END__
            Rate autobox  classic
autobox  73529/s      --     -33%
classic 109890/s     49%       --
```

Listing 4

TPF-TICKER

MediaWiki-Parser-Grant -> Abgebrochen

In seltenen Fällen wird ein Grant abgebrochen. Das kann unterschiedlichste Gründe haben. Die Perl-Foundation und Shlomi Fish haben sich darauf geeinigt den im Juli 2007 genehmigten Grant abzubrechen. Damit kann Shlomi Fish seine Arbeitskraft in andere Projekte stecken und die Perl-Foundation kann das Geld für andere Grants verwenden.

Rezension - Higher Order Perl

Mark Jason Dominus
ISBN 1-55860-701-3
erschienen 2005 bei Morgan Kaufmann.

Um funktional programmieren zu lernen, muss man nicht Lisp lernen und sich mit zig Klammern herumschlagen. Perl hat fast alles unter der Haube, was man für funktionale Programmierung braucht. Doch wenige Programmierer benutzen diese mächtigen Werkzeuge, größtenteils weil sie in den bisherigen Lehrbüchern stiefmütterlich behandelt wird.

„Higher Order Perl“ ändert das. Der Leser wird behutsam und Schritt für Schritt an Rekursion, Referenzen auf Funktion, Callbacks und schliesslich Closures herangeführt. Mit vielen zum Teil sehr praxisnahen Beispielen wird gezeigt, wie durch Verwendung der genannten Techniken ähnliche Stücke Programmcode zu einem Stück zusammengefasst werden können, und wie man mit Callbacks die Flexibilität von Funktionen deutlich erhöht.

Ein Abschnitt zu Caching-Techniken dürfte alle performancebegeisterten Perlhacker erfreuen. Das Core-Modul `Memoize` (das übrigens auch von Mark Jason Dominus geschrieben wurde) wird vorgestellt, inklusive diverser darin verwendeten Techniken und einer ausführlichen Diskussion, wann man es sinnvoll verwenden kann und wann nicht.

Dann werden fortgeschrittenere Konzepte wie „lazy streams“ vorgestellt, mit deren Hilfe beeindruckende Programme konstruiert werden, zum Beispiel diverse Parser, eine Regex-Engine und ein System zum algebraischen Lösen von linearen Gleichungssystemen.
Einige der Beispiele im Buch sind sehr praxisorientiert, wie etwa das Entfernen von HTML-Tags und ein Crawler für Webseiten.

„Higher Order Perl“ ist sehr gut zu lesen, allerdings nicht am Stück von vorne bis hinten - dafür wird zu viel ungewöhnliches Wissen vermittelt. Auch braucht man Zeit dafür, das Gelesene in das Repertoire der eigenen Programmiertechniken einzubauen. Der Text ist ab und an durch kleinere Anekdoten aufgelockert, die recht gut zum Thema passen und den Leser zum schmunzeln bringen. Grafiken gibt es wenige. Die vorhandenen Grafiken sind klar und einfach gehalten und illustrieren anschaulich das aktuelle Problem.

Das Buch ist für jeden empfehlenswert, der mit Perl sicher umgehen kann und das Gefühl hat, seine Programmierprobleme zwar mit Perl lösen zu können, aber nicht immer elegant. Auch zukünftigen Perl 6-Hackern kommt das Wissen um funktionale Programmierung zu Gute, weil Perl 6 viele funktional angehauchte Features hat. Wer Ambitionen als Perlprogrammierer hat, kommt nicht um dieses Buch herum.

Moritz Lenz

Typeglobs

Dieser Artikel beschreibt Perls globale (Paket-) Variable und deren Implementierung.

Globale Variable gibt es, seitdem es Perl gibt - wird eine Variable nicht deklariert, so wird sie automatisch als global eingerichtet. Globale Variable sind im *gesamten* Programm sichtbar, sogar aus Code, der aus anderen Dateien hinzugeladen wurde (manche Module verwenden solche Variablen zu Konfigurations- oder Debuggingzwecken). Es gibt keine „eingeschränkte Sicht“ - eine globale Variable ist immer sichtbar; allerdings kann es notwendig werden, zum Variablennamen auch den Namensraum (`package`) anzugeben, in dem die Variable angelegt wurde, um sie ansprechen zu können.

Der `local` Befehl ermöglicht das temporäre Wegsichern und Ersetzen eines *Variablenwertes* (der automatisch zu einem späteren Zeitpunkt wieder hergestellt wird), die Variable selbst jedoch bleibt dieselbe und kann jederzeit von überall angesprochen werden.

Stashes und Packages

Ganz allgemein repräsentiert eine Variable ein Stück benannten Speichers. Daher geht es beim Zugriff darauf zunächst immer um die Umsetzung eines Namens in einen Adresswert.

Perl besitzt bereits von Haus aus eine Einrichtung zum Umsetzen von Namen in andere Werte - den `Hash` (assoziatives Array). Es wird daher wenig verwundern, dass auch globale Variable in Form von Hashes implementiert sind. Jeder solche Hash repräsentiert einen Namensraum (package); die *Schlüssel* der Hashes sind die Namen aller Symbole, die in diesem Package existieren (Variable-, Funktions-, Format- und Handlenamen). Solche Hashes kann man sich als Symboltabellen vorstellen und man bezeichnet sie daher auch als `Symbol Table Hashes` oder kurz `Stashes`.

Eine globale Variable gehört immer genau einem Package an, kann aber auch aus allen anderen Packages durch Vorsetzen ihres Packagenamens angesprochen werden. Stash- und Packagenamen sind identisch, allerdings muss man Stashnamen, so man sie direkt verwendet, zwei Doppelpunkte nachsetzen, damit sie der Parser als solche erkennt. Der Name des Stashes, der das package `Hugo` bereitstellt, lautet somit `%Hugo::` (beachte das führende „%“, es handelt sich ja um einen Hash). Der Name des Stashes jenes Packages, das von Perl verwendet wird, wenn nichts anderes ausgewählt wurde (Default Package), lautet `%main::`.

Die `package` Direktive definiert zum Kompilierungs-Zeitpunkt ein neues Package. Sie sorgt dafür, dass der zugehörige Stash angelegt wird, falls er noch nicht existiert, und dass ab dann alle neuen Symbole, die nicht voll qualifiziert (d.h., mit expliziter Angabe eines Packagenamens versehen) oder ausdrücklich als lexikalisch gekennzeichnet sind, in den Stash als neue Elemente aufgenommen werden [1].

Zur Laufzeit weiß dann der Interpreter, in welchem Package nach den jeweiligen Variablen Ausschau zu halten ist [2].

Genauso, wie ein Hash andere Hashes (in Form von Referenzen) enthalten kann, kann ein Stash weitere Stashes enthalten, wenn - vereinfacht ausgedrückt - in seinen Elementen keine Symbole, sondern wiederum Stashes abgelegt sind. Dadurch lassen sich Packages in der bekannten Weise

```
Alpha::Beta::Gamma
```

verschachteln. Dennoch bleibt jeder Stash eigenständig und Variablen in verschachtelten Stashes stehen in keinerlei Beziehung zueinander. Die Stashes selbst sind miteinander derart verbunden, dass der übergeordnete Stash einen Verweis auf den untergeordneten enthält (in obigem Beispiel enthält

also `%Alpha::` ein Element, das auf `%Beta::` zeigt, und dieser Stash wiederum ein Element auf `%Gamma::`). Damit wird auch das rekursive Durchsuchen von Symboltabellen möglich.

Stashelemente

Die Elemente eines Stashes benennen *Symbole*, d.h. die Namen von Variablen, Funktionen, Formaten oder Handles, die in dem durch den Stash repräsentierten Package angelegt wurden. Diese Elemente kann man genauso wie die Elemente eines jeden anderen Hashes ansprechen, so z.B. das Element `test` im Package `Hugo`:

```
$Hugo::{test}
```

Beachte, dass die Doppelpunkte hier nicht als Trenner zwischen Package- und Variablennamen verwendet werden; sie sind vielmehr Teil des Namens des Stashes `%Hugo::`. Im Unterschied dazu steht

```
$Hugo{test}
```

für das Element `test` des „normalen“ Hashes `%Hugo`.

Symbolelemente im Default Package `%main::` werden genauso angesprochen, so wird das Element `test` mit

```
$main::{test}
```

bezeichnet. Weil es sich hier um das Default Package handelt, gestattet der Parser auch das Weglassen des Packagenamens, sodass man auch kürzer

```
$::{test}
```

schreiben kann.

Beachte, dass die Doppelpunkte am Namensende nur für Hashes eine besondere Bedeutung haben, sie werden hierdurch als Stash gekennzeichnet. Für alle anderen Wertetypen von Perl sind sie bedeutungslos; der Skalar `$Hallo::` und das Array `@Hallo::` unterscheiden sich durch nichts von anderen Variablen (außer vielleicht durch den auffälligen Namen). Die Doppelpunkte sind aber dennoch Teil des Namens und daher sind `$Hallo::` und `$Hallo` zwei unterschiedliche Variable.

Perl versteckt den Zugriff auf Stashes und deren Elemente nicht, sondern behandelt sie wie jeden anderen Hash, d.h., Hash-Operatoren wie `each`, `exists`, `keys`, `values` und `delete` können wie gewohnt verwendet werden, wodurch das Durchsuchen und Bearbeiten von Symboltabellen aus Perl-Code heraus möglich wird. Hiervon machen auch etliche Module Gebrauch.

Dieser Ansatz birgt natürlich auch gewisse Gefahren - so sollte man sich darüber im Klaren sein, dass in einem Skript nach Ausführen von

```
undef %main::;   # oder kuerzer: undef %::;
```

oder

```
%main:: = (ALLES => 'WEG');
```

dessen Laufruhe möglicherweise etwas leiden wird. Auch das Entfernen von Stashelementen mit `delete` sollte mit Bedacht geschehen. Generell ist beim modifizierenden Zugriff auf Stashes durchaus Vorsicht angebracht; Perl ist wie immer mit seinen Möglichkeiten freizügig, erwartet aber auch hier, dass der Programmierer weiß, was er tut.

Typeglobs

Im Unterschied zu den meisten Sprachen erlaubt Perl das Benennen einer Variable und einer Funktion mit demselben Namen. So können eine Variable `test`, eine Funktion `test` und sogar ein Handle `test` gleichzeitig nebeneinander existieren, ohne sich im Geringsten zu beeinflussen.

Bei globalen Variablen wird dies durch eine Komponente namens Typeglob ermöglicht. Darunter kann man sich eine Datenstruktur mit sechs Feldern (*Slots*) vorstellen, die Platz für alle Wertetypen vorsieht, die Perl kennt:

- Skalare (und Referenzen)
- Arrays
- Hashes (und Stashes)
- Funktionen (benannte Subroutinen und Methoden)
- IO Handles (Dateien, Verzeichnisse, Pipes, Sockets)
- Format Handles

Meist ist pro Typeglob nur ein Slot belegt, die anderen sind leer [3]. Werden aber, wie oben beschrieben, in einem Skript z.B. ein Skalar und ein Array gleich benannt, so werden die

Werte beider Variable im zugehörigen Typeglob in den jeweiligen Slots abgelegt. Davon abgesehen, dass sie über denselben Namen angesprochen werden, haben diese beiden Variablen aber nichts gemeinsam und könnten genauso gut unter verschiedenen Namen existieren.

Was haben nun Stashelemente und Typeglobs miteinander zu tun? Nun, die Antwort ist naheliegend: *Der Wert* eines Stashelements ist ein Typeglob [4].

Daher geschieht z.B. beim Zugriff auf die Variable `$Hugo::test` vereinfacht gesagt folgendes:

• Im Stash `%Hugo::` wird nach dem Element `test` gesucht. Der Wert des Elements ist in ein Typeglob.
• Da dem Namen ein $-Zeichen vorangestellt ist, wird auf den Skalar-Slot des Typeglobs zugegriffen, um den Variablenwert zu erhalten.

Tatsächlich kann man sich das Sigil (Vorzeichen) in Variablenamen als Selektor für den auszuwählenden Typeglob-Slot vorstellen. Bei jenen Datentypen, für die Perl keine Sigils vorsieht, ergibt sich der anzusprechende Slot aus dem Umfeld, in dem der jeweilige Name vorkommt. Für das Aufrufen von Funktionen gibt es außerdem noch die Notation `name()` als häufigere (und verständlichere) Alternative zu `&name` (außerdem sind diese beiden Aufrufe in ihrer Wirkung nicht gleich, siehe perlsub).

Perl erlaubt das Ansprechen von Typeglobs auch direkt mit dem *-Sigil, und somit sprechen die folgenden Ausdrücke denselben Typeglob (im Package `main`) an:

```
$main::{hugo}        *main::hugo      *hugo
```

Ersterer wird allerdings erst zur Laufzeit ausgewertet (da ja auf ein Hashelement, dessen Inhalt beim Kompilieren noch nicht bekannt ist, zugegriffen wird), die beiden anderen schon beim Kompilieren, da hier der Typeglob direkt adressiert werden kann. Daher ist die Stern-Notation schneller, aber weniger flexibel.

Typeglobs und Referenzen

Im vorigen Abschnitt wurde festgestellt, dass Typeglobs Felder (*Slots*) für die sechs möglichen Wertetypen von Perl enthalten. Das ist aber nur die halbe Wahrheit; tatsächlich enthalten die Slots, so sie belegt sind, **Referenzen**, die auf die eigentlichen Werte zeigen. Und da Perl seit der Version 5 den direkten Umgang mit Referenzen gestattet, kann man so auch die Slots von Typeglobs auslesen oder ihnen neue Werte zuweisen. Das wird in den folgenden Abschnitten näher betrachtet.

Zuweisungen an Typeglobs

Einem Typeglob kann man, ähnlich wie Skalaren, Referenzen auf Werte zuweisen. Eine solche Zuweisung ist gleichbedeutend mit der direkten Zuweisung des Wertes an den entsprechenden Variablentyp (siehe Listing 1).

Der Code in den drei Spalten ist von der Wirkung her identisch, wenngleich die linke Schreibweise die Geläufigste ist (links unten findet sich eine bereits beim Kompilieren verarbeitete *Deklaration*; die Zuweisungen finden hingegen erst zur Laufzeit statt). In einigen, später gezeigten Fällen führt allerdings an der Zuweisung über den Typeglob kein Weg vorbei.

Die rechte Spalte zeigt die Zuweisung ebenfalls über den Typeglob, wobei hier durch *temporäre Dereferenzierung* wiederum *Werte* und keine Referenzen zugewiesen werden. Da es in Perl keine Funktionsliterale gibt (sondern nur Referenzen darauf), ist die Zuweisung einer Funktion auf die in dieser Spalte gezeigte Art nicht möglich.

Beachte die Verwendung von `\do {...}` für die Zuweisung einer anonymen Skalarreferenz, für die Perl keine eigene Syntax vorsieht. Achtung: die Verwendung von `\3` ergäbe eine Referenz auf einen schreibgeschützten und daher im Vergleich zur linken Spalte nicht identischen Wert.

Besonderes Augenmerk verdient das Konstrukt in der letzten Zeile - es zeigt den Einsatz von `sub` in einer Deklaration (die vom Parser ausgewertet wird) in der linken Spalte, und den Einsatz als Operator mit einer zur Laufzeit durchgeführten

```
$x = 3;             *x = \do {my $x = 3};    ${*x} = 3;
@y = (1, 2);        *y = [1, 2];             @{*y} = (1, 2);
%z = (s => 3);      *z = {s => 3};           %{*z} = (s => 3);
sub a {...};        *a = sub {...};
```

Listing 1

Zuweisung in der mittleren Spalte. In letzterem Fall liefert `sub` eine Referenz auf eine anonyme Subroutine, die durch Zuweisung an einen Typeglob einen Eintrag in der Symboltabelle erhält und damit ihre Anonymität verliert. Tatsächlich kann man eine vom Parser verarbeitete Deklaration

```
sub mysub {
  ...
}
```

genauso gut auch als

```
BEGIN {
  *mysub = sub { ... };
}
```

schreiben (allerdings ist erstere klarerweise schneller, weil sie direkt vom Parser abgehandelt wird). Diese Technik - das Generieren von Funktionen zur Laufzeit durch Zuweisung von Codereferenzen an Typeglobs - ist recht beliebt und wird z.B. von Wrapper-Funktionen verwendet.

Beachte, dass bei den obigen Globzuweisungen auf der linken Seite stets dieselbe Angabe - eben der Typeglob - zu finden ist. Perl erkennt selbst anhand des zuzuweisenden Referenztyps, welcher Slot des Typeglobs durch die Zuweisung zu belegen ist.
Ein weiterer Punkt soll nicht unerwähnt bleiben: die Zuweisung eines Typeglobs an einen anderen Typeglob in der Form

```
*alpha = *beta;
```

Sie bewirkt, dass beide Typeglobs nun dieselben Slots gemeinsam verwenden. Wird daher an `$alpha` ein Wert zugewiesen, so kann dieser nun auch mit `$beta` angesprochen werden. Das gilt sinngemäß auch für `@beta`, `%beta`, `&beta` und die Handles `beta`. Diese Technik bezeichnet man als *Aliasing* und sie bildet die Basis für Module wie Exporter, die dadurch Symbole (meistens Funktionsnamen) eines Packages in einem anderen Package abbilden und es so ermöglichen, fremde (*importierte*) Funktionen wie lokale Funktionen ohne Package-Prefix anzusprechen.

Wird versucht, an einen Typeglob etwas anderes als eine Referenz zuzuweisen:

```
*hugo = 'test';
```

so nimmt Perl an, dass eine Alias-Zuweisung stattfinden soll und macht daraus

```
*hugo = *test;
```

Da dies erst zur Laufzeit passiert (der Compiler generiert ungeniert den Code für die Zuweisung des Stringliterals an den Typeglob), gibt es in diesem Fall auch mit `use strict` keinerlei Hinweise auf dieses nicht ganz saubere Konstrukt.

Da man Typeglobs nicht mit `our` deklarieren kann, darf man sie - wie Funktions- und Handlenamen - immer unqualifiziert, d.h. ohne Packagenamen ansprechen, auch mit `use strict 'vars'`.

Abschließend sei noch kurz der Befehl

```
undef *hugo;
```

erwähnt, der alle Slots des Typeglobs löscht, also einem

```
undef $hugo;
undef @hugo;
undef %hugo;
...
```

entspricht. Hier kann man sich das „*“ tatsächlich als Wildcard („Lösche alles mit dem Namen `hugo`“) vorstellen.

Es ist zu beachten, dass das Löschen eines Stashelements nicht dieselbe Wirkung hat wie das `undef`-Setzen eines Typeglobs. Wird z.B. mit

```
delete $main::{hugo};
```

das Stashelement gelöscht, so wird ein darauf folgender Zugriff

```
print $hugo;
```

dennoch funktionieren, da die Umsetzung des Elementkeys in eine Typeglob-Adresse in der `print`-Anweisung bereits beim Kompilieren geschieht. Daher wird der Stash zur Laufzeit hierfür nicht mehr benötigt und das Löschen bleibt wirkungslos. Das `undef`-Setzen eines Typeglobs führt hingegen erst zur Laufzeit zur Neuinitialisierung aller Slots und hat somit die gewünschte bereinigende Wirkung.

Da an Typeglobs nur Referenzen zugewiesen werden können, ist die Zuweisung von `undef`

```
*hugo = undef;
```

unzulässig und wird mit einem

```
Undefined value assigned to typeglob
```

kommentiert, ansonsten aber ignoriert. Erlaubt ist hingegen

```
*hugo = \undef;
```

und es bleibt dem Leser als Aufgabe überlassen, sich zu überlegen, was hier geschieht [5].

Auslesen von Typeglobs

Die in Typeglobs enthaltenen Referenzen können ausgelesen und wie jede andere Referenz an Skalare zugewiesen oder sonst weiterverarbeitet werden. Allerdings wäre die Notation

```
$ref = *glob;
```

nicht mehr eindeutig, da hieraus nicht hervorgeht, *welche* Referenz (d.h. welcher Slot) angesprochen werden soll. Stattdessen existiert eine hashelement-ähnliche Syntax zum Ansprechen der sechs Slots eines Typeglobs:

```
$scalarref = *glob{SCALAR};
$arrayref  = *glob{ARRAY};
$hashref   = *glob{HASH};
$coderef   = *glob{CODE};
$handleref = *glob{IO};
$formatref = *glob{FORMAT};
```

Alle diese Zuweisungen liefern Referenzen des jeweiligen Typs zurück, die dann ganz normal dereferenziert werden können. So kann man nach obiger Zuweisung mit

```
@{$arrayref}
```

auf das Array, dessen Referenz dem Array-Slot des Typeglobs entnommen wurden, zugreifen. Anders gesagt, sind die beiden folgenden Ausdrücke von der Wirkung her identisch:

```
\@hugo      *hugo{ARRAY}
```

und daher auch diese beiden:

```
@hugo     @{*hugo{ARRAY}}
```

wobei die linke Schreibweise sicher kürzer und geläufiger ist, die rechte aber die internen Abläufe beim Zugriff veranschaulicht.

Beachte, dass diese Konstrukte nur auf der *rechten* Seite einer Zuweisung (d.h. als Quelle) vorkommen dürfen, auf der linken Seite akzeptiert sie der Parser nicht. Das ist auch nicht notwendig, da Perl dann anhand des Referenztyps selbst erkennt, welcher Slot des Typeglobs zu belegen ist.

Ist der angegebene Slot des Typeglobs nicht in Verwendung, wird `undef`, im Falle von `SCALAR` eine Referenz auf `undef` zurückgeliefert (dieser Sonderfall muss daher beim Testen ggf. berücksichtigt werden).

Die nahe Verwandtschaft von Typeglobs und Referenzen führt dazu, dass an Stellen, an denen eine Referenz erwartet wird, auch ein Typeglob verwendet werden kann (aber nicht immer auch umgekehrt!).

Erwähnenswert sind auch noch die folgenden Konstrukte:

```
$globref  = *glob{GLOB};
$name     = *glob{NAME};
$package  = *glob{PACKAGE};
```

Sie liefern eine Referenz auf den Typeglob selbst, sowie den Symbolnamen (d.h., den Keynamen des Stashelements, das den Typeglob enthält), und den Packagenamen zurück. Auf den ersten Blick nicht sonderlich interessant, werden diese Konstrukte wichtig, wenn man Typeglobs in Variablen als Argumente an Funktionen übergibt. Die Funktion kann dann bei Bedarf Package- und Symbolnamen des übergebenen Typeglobs ermitteln und über die Referenz wiederum direkt darauf zugreifen, was im Umgang mit Datei- und Formathandles wichtig werden kann.

Beachte auch, dass es sich trotz der ähnlichen Syntax nicht um Hashes und Hashelemente handelt und dass daher abgesehen von den gezeigten Beispielen keine anderen Operationen zulässig sind. Die Keynamen sind fix vorgegeben, andere Namen (z.B. `*glob{FUNCTION}`) liefern den Wert `undef` zurück.

Ein Typeglob selbst wird von Perl beim Zuweisen wie ein Skalar betrachtet und daher ist die Anweisung

```
$glob = *hugo;
# Gibt "*main::hugo" aus (im Package main)
print $glob;
```

durchaus zulässig. Hier enthält der Skalar `$glob` nun eine Kopie von `*hugo`, wobei die Slots der Kopie dieselben Referenzen wie das Original enthalten, d.h., auf dieselben Werte zeigen. Daher kann man mit

```
*{$glob}
```

auf die Kopie oder z.B. mit

```
 $slot = 'HASH';
 my %hash = %{*{$glob}{$slot}};
```

über die Kopie auf den ursprünglichen Hash (hier also `%hugo`) zugreifen. Da man Typeglobs in Skalaren abspeichern kann, kann man sie natürlich auch in Hash- oder Arrayelementen ablegen und so bei Bedarf an eine Funktion z.B. ein Typeglob-Array übergeben.

Es ist allerdings wichtig zu beachten, dass solcherart abgespeicherte Typeglobs nur sog. **Fake-Kopien** sind, was vereinfacht gesagt bedeutet, dass man auf sie bzw. ihre Slots nur mehr *lesend* zugreifen darf. Der Versuch, einen Slot durch Zuweisung einer Referenz zu beschreiben, bewirkt, dass die Typeglob-Kopie verworfen wird und der Skalar statt der Kopie nur mehr die zugewiesene Referenz enthält:

```
# Erzeugt in $glob ein Fake-Kopie von *hugo
$glob = *hugo;
# Gibt "*main::hugo" aus
print $glob;
# Array-Slot soll ueberschrieben werden
*{$glob} = [1,2,3];
# $glob enthaelt jetzt Arrayreferenz
# gibt "ARRAY(0x.......)" aus
print $glob;
```

Der ursprüngliche Typeglob (`*hugo`) bleibt hiervon unberührt.

Möchte man auch das Modifizieren von Typeglobs, die in Skalaren enthalten sind, ermöglichen, wird man *Globreferenzen* (siehe nächster Absatz) verwenden. Man kann aber auch mit

```
*newglob = $glob;
```

den Typeglob duplizieren und dann nicht mehr über den Skalar, sondern direkt auf den neuen Typeglob zugreifen. Das ist identisch mit der weiter oben beschriebenen Methode des *Aliasing*; d.h., man kann dann über beide Typeglobs wieder auf *dieselben* Slots zugreifen, die natürlich auch wieder ohne Probleme modifiziert werden können.

Man kann auch Typeglobs über Referenzen ansprechen, wenn man den Referenz-Operator \ oder die weiter oben erwähnte Hash-Notation verwendet:

```
$globref = \*glob;
$globref =  *glob{GLOB};
```

liefert beides eine Referenz auf den Typeglob zurück, wodurch das Anlegen von Fake-Kopien vermieden wird. Das kann außerdem für Funktionen, die unterschiedliche Referenztypen entgegennehmen, hilfreich sein, da man dann mit der `ref` Funktion prüfen kann, ob eine Globreferenz übergeben wurde:

```
sub globderef {
  my $arg = shift;
  die 'Not a glob reference'
      unless ref $arg eq 'GLOB';
  print 'Got glob reference  to ',
        *{$arg}, "\n";
}
```

Hier ist es wichtig, vor der Dereferenzierung des Typeglobs sicherzustellen, dass eine entsprechende Referenz als Argument mitgegeben wurde, was mit `ref` einfach geschehen kann. Würde der Typeglob direkt übergeben, wäre eine solche Prüfung nicht möglich.

Man kann eine solche Funktion auch mit einem Prototyp `*` deklarieren, z.B.

```
sub deref(*) {
 ...
}
```

dann erhält sie stets eine Globreferenz übergeben, egal ob sie nun mit einem Typeglob oder mit einer Referenz darauf aufgerufen wurde.

Wie hier zu erkennen ist, ist die Syntax zum Dereferenzieren einer Globreferenz und zum Auslesen einer Typeglob-Kopie aus einer Skalarvariablen die gleiche:

```
# Skalar enthaelt Fake-Kopie
$globvar =  *glob;
# Skalar enthaelt Globreferenz
$globref = \*glob;
*newglob = *{$globvar};    # oder
*newglob = *{$globref};
```

Sauberer ist aber der Ansatz mit der Globreferenz, weil dadurch das Anlegen einer Fake-Kopie vermieden wird.

Verwendung von Stashes und Typeglobs

Soviel zur Theorie über Stashes und Typeglobs. Wofür man sie beim Programmieren gebrauchen kann - wie z.B. die Bearbeitung von Symboltabellen -, zeigt der Artikel in der nächsten Ausgabe.

Eine weitere Verwendung vor Perl 5 war die Übergabe von Referenzen an und deren Verarbeitung in Funktionen, da es den Referenzoperator \ und den Dereferenzierungsoperator -> dort noch nicht gab. Dabei machte man sich die enge Verwandtschaft von Typeglobs und Referenzen zunutze. Um z.B. einer Funktion Zugriff auf ein Array zu ermöglichen, konnte man Folgendes schreiben:

```
sub my_func {
  *array = shift;
  print join ',',@array;
  @array = (2,4,6);
}

@val = (1,2,3);
my_func(*val);        # Gibt "1,2,3" aus
print join ',',@val;  # Gibt "2,4,6" aus
```

Hier wird über den Typeglob tatsächlich ein Zugriff auf das Array (und nicht etwa auf eine Kopie) gestattet, wodurch Änderungen am Array auch außerhalb der Funktion bestehen bleiben.

Diese Art der Kodierung wurde in Perl 5 durch die Einführung von Referenzen überflüssig und sollte nicht mehr verwendet werden. Es ist aber möglich, dass man in älterem Perl-Code noch darauf stößt, sodass es kein Fehler ist, darüber Bescheid zu wissen.

Ferry Bolhár-Nordenkampf

[1] Von dieser Regel gibt es Ausnahmen: die Symbole ARGV, ARGVOUT, ENV, INC, SIG, STDERR, STDIN und STDOUT werden *immer* im Stash %main:: angelegt (und später dort gesucht), auch wenn mit package gerade ein anderer Namensraum ausgewählt ist. Sollen diese Symbole in anderen Stashes angelegt werden, so ist der Symbolname immer vollqualifiziert anzugeben. Jedoch bleibt die spezielle Bedeutung der einzelnen Variablen auf jene im Stash %main:: beschränkt.

Beachte, dass hier von **Symbolen** die Rede ist: Obiges trifft daher z.B. auf $ENV, @ENV, %ENV und &ENV (sowie das Dateihandle ENV und das Formathandle ENV) zu, d.h., auf alle Wertetypen mit den angeführten Namen (*ENV).

Es trifft weiters auch auf alle Interpunktions-Variable (solche, deren Name aus keinem alphabetischen Zeichen besteht oder damit beginnt) zu. Bei solchen Variablen erlaubt der Parser außerdem keine Angabe eines Packagenamens, so dass diese Variable *immer nur* im Stash %main:: vorkommen können.

[2] „Wissen“ ist hier im übertragenen Sinn zu verstehen - die Stashelemente werden bereits beim Kompilieren angelegt und der vom Compiler generierte Code greift nur über deren Adressen auf den Inhalt zu. Der Zugriff über den Stash wird nur notwendig, wenn ein Variablenname erst zur Laufzeit gebildet wird, wie das z.B. in Code wie

```
my $package = 'Hugo';
my $name = 'Test';
${$package.'::'.$name} = 'Hallo';
```

geschieht (d.h., über symbolische Referenzen) . Hier kann beim Kompilieren keine Elementadresse verwendet werden, da der Name des Stashelements selbst erst zur Laufzeit gebildet werden muss.

[3] Bis Perl 5.9.2 ist aus Implementierungsgründen der Skalar-Slot eines Typeglobs immer belegt; wird er nicht verwendet, so enthält er den Wert \undef. Ab Version 5.9.3 kann mittels einer Option beim Kompilieren von Perl entschieden werden, ob dieses Verhalten beibehalten oder der Skalar-Slot wie alle anderen Slots nur bei tatsächlichem Bedarf belegt werden soll. Es ist aber auch dann sichergestellt, dass beim Zugriff auf

```
*Symbol{SCALAR}
```

immer `\undef` zurück geliefert wird, d.h. bestehender Code muss nicht geändert werden.

[4] Ab Perl 5.9.3 gibt es eine sehr effektive Form, Funktionen, die nur Konstanten zurückliefern, zu definieren, indem man dem jeweiligen Stashelement eine Referenz auf den zurückzuliefernden Wert (der ein Skalarwert sein muss) zuweist:

```
BEGIN{
   $main::{ConstFunc} = \3;
}
```

Hier wird die Funktion `ConstFunc` definiert, die die Konstante `3` zurückliefert. Die Zuweisung muss, da sie das Verhalten des Parsers beeinflusst, in einen `BEGIN`-Block gesetzt und lexikalisch vor dem ersten Funktionsaufruf notiert werden. Obiger Code entspricht der bekannten Deklaration

```
sub ConstFunc() {3}
```

Die gezeigte Zuweisung ist allerdings effizienter und speicherschonender, da sie ohne Typeglob auskommt. Bei Modulen, die viele Konstante definieren (z.B. `POSIX`), kann das eine sehr positive Auswirkung auf die Performance haben.

In solchen Fällen kann ein Stashelement somit statt einem Typeglob auch eine Skalarreferenz enthalten. Code, der Stashelemente ausliest, sollte das entsprechend berücksichtigen.

[5] Es wird an den Typeglob eine Referenz auf `undef` zugewiesen. `undef` ist ein skalarer Wert, daher handelt es sich um eine Skalarreferenz, die den Skalarslot des Typeglobs löscht. Anders gesagt bewirken die folgenden Befehle dasselbe:

```
*hugo = \undef;
$hugo =  undef;
```

Beachte, dass der vom Compiler generierte Code jedoch unterschiedlich ist, d.h., die beiden Befehle bewirken dasselbe, sind aber *nicht identisch*.

TPF-TICKER

Grants im 2. Quartal 2008

Nach einer längeren Zeit der Abstimmung und Beratung sind jetzt die Ergebnisse der 2008Q2-Runde des Grant Committees da:

Die folgenden 5 Projekte werden gefördert:

* Perl on a Stick (Adam Kennedy)
* Test::Builder 2 (Michael Schwern)
* Make localtime() and gmtime() Work Past 2038 (Michael Schwern)
* Fixing Bugs in the Archive::Zip Perl Module (Alan Haggai Alavi)
* SMOP - Simple Meta Object Programming (Daniel Ruoso)

Sollten noch mehr Projekte gefördert werden, werden es diese Vorschläge sein:

* Perl Survey (Kieren Diment)
* Module Installation Configuration Wizard (Michael Schwern)
* Improve POE::Component::IRC (Hinrik Örn Sigurðsson)
* CPAN Stability Project (Michael Schwern)
* Extending BSDPAN (Colin Smith)
* Automatic INSTALL generation (Michael Schwern)

Debugger

Einführung in den Perl Debugger

Real Programmers don‘t need debuggers,
they can read core dumps.

Larry Wall

Viele Perl-Programmierer mögen und nutzen den Perl-Debugger aus unterschiedlichsten Gründen nicht. Auch ich habe mich jahrelang gesträubt, den Perl-Debugger zu benutzen. Wenn man jedoch einige Male mit dem Debugger gearbeitet hat, wird man ihn gerne in seinen Werkzeugkasten aufnehmen.

Der nachfolgende Artikel soll einen einfachen Einstieg in den Perl-Debugger ermöglichen, indem die wichtigsten grundlegenden Kommandos und Tricks Schritt für Schritt erläutert werden.

Features oder Was kann der Debugger?

Der Perl-Debugger beherrscht die typischen Debuggerfeatures wie Haltepunkte (Breakpoints), Überprüfung von Variablen, Methoden, Code und Klassenhierachien, Verfolgung (Trace) der Code-Ausführung und Subroutinen-Argumente.

Zusätzlich erlaubt der Debugger die Ausführung eigener Befehle (Perl-Code) vor und nach jeder Programmcode-Zeile und die Änderung von Variablen und Code während der Ausführung.

Die Verfolgung geforkter Programme, Threads und remote debugging wird ebenfalls unterstützt.

Der Perl-Debugger kann in den Apache Web Server (mod_perl) integriert werden und es gibt natürlich auch graphische Benutzeroberflächen.

Bevor man den Debugger anwirft

If debugging is the process of removing bugs,
then programming must be the process of putting them in.

Edsger W. Dijkstra

Viele Bugs lassen sich durch einen defensiven Programmierstil vermeiden. Ein gutes Perl-Programm verwendet daher `strict` und `warnings`.

Debugger Starten

Der Perl-Debugger wird stets mit dem Kommandozeilenschalter `-d` aufgerufen.

Parameter, die an das zu debuggende Programm übergeben werden, können auf der Kommandozeile angegeben werden.

```
perl -d programm [arg0 arg1 ...]
```

Dies gilt auch für CGI-Programme, hier entsprechen die Parameter den Formularfeldern.

```
perl -d programm.cgi [param1= param2=]
```

Der Debugger kann auch als Perl-Shell gestartet werden.

```
perl -d -e 0
```

Erste Schritte - Der Debugger als Perl-Shell

Für die ersten Schritte mit dem Perl-Debugger bietet sich die Nutzung als Perl-Shell an.

```
perl -d -e 0
... Ausgabe unterdrückt
main::(-e:1):    0
  DB<1>
```

Der Debugger meldet sich mit einem eigenem Prompt (`DB<n>`) und wartet auf die Eingabe eines Kommandos. Zum Beenden des Debuggers dient das Kommando `q`, Hilfe erhält man über das Kommando `h`.

Code eingeben

Nun kann beliebiger Perl-Code eingeben werden.

```
perl -d -e 0

  DB<1> print "Hallo Foo-Magazin"
Hallo Foo-Magazin

  DB<2> print 6 * 7; print "\n"; print 6 x 7
42
6666666
```

Das Zeilenende wird als Anweisungsende aufgefasst, das übliche Semikolon am Zeilenende kann daher entfallen. Mehrere Anweisungen in einer Zeile müssen allerdings durch ein Semikolon getrennt werden.

Continuation Lines (Fortführungszeilen)

Falls mehr als eine Zeile zur Eingabe des Codes benötigt wird, muss das Zeilenende mit einem \ maskiert werden.

```
  DB<1> foreach my $number ( 1 .. 2 ) { \
  cont:    print "$number\n" \
  cont: }
1
2
```

Variableninhalte ausgeben mit `p`

Das Kommando `p` erlaubt die einfache Ausgabe der Werte von Variablen.

```
  DB<1> $scalar = "Text"
  DB<2> @array = qw(eins zwei)
  DB<3> %hash = ( 'a' => '1', 'b' => '2')
  DB<4> p $scalar
Text
  DB<5> p @array
einszwei
  DB<6> p %hash
a1b2
```

Datenstrukturen und Variableninhalte mit `x` anzeigen

Die Ausgabe von `p` ist für skalare Variablen durchaus brauchbar. Eine übersichtlichere Darstellung lässt sich mit dem Kommando `x` erzeugen.

Einfache Variablen und Datenstrukturen

Betrachten wir zunächst einmal die grundlegenden einfachen Perl-Datenstrukturen.

Das Kommando `x` evaluiert seine Parameter im Listenkontext.

```
  DB<7> x $scalar
0  'Text'
```

Bei skalaren Variablen wird eine Liste mit einem Element zurückgegeben.

```
  DB<8> x @array
0  'eins'
1  'zwei'
```

Bei Array-Variablen wird eine Liste der Elemente mit dem jeweiligen Index zurückgegeben.

```
  DB<9> x %hash
0  'a'
1  1
2  'b'
3  2
```

Auch bei Hashes wird eine Liste mit den entsprechenden Indizes ausgegeben.

Wenn man jedoch dem Kommando `x` eine Referenz auf eine Variable übergibt, wird die Ausgabe wesentlich informativer.

```
  DB<10> x \$scalar
0  SCALAR(0x8427f00)
   -> 'Text'

  DB<11> x \@array
0  ARRAY(0x8427e1c)
   0  'eins'
   1  'zwei'

  DB<12> x \%hash
0  HASH(0x8427d38)
   'a' => 1
   'b' => 2
```

Komplexe Datenstrukturen betrachten

Das Kommando `x` eignet sich weiterhin sehr gut zum Betrachten komplexer Datenstrukturen.

Beispiel: Hash of Arrays (HoA)

```
  DB<1>  %HoA = ( \
  cont: flintstones => [ "fred", "barney" ],\
  cont: jetsons => [ "george", "jane", \
                       "elroy" ],)

  DB<2> x \%HoA
0  HASH(0x8427de0)
   'flintstones' => ARRAY(0x81547bc)
      0 'fred'
      1  'barney'
   'jetsons' => ARRAY(0x8427d50)
      0  'george'
      1  'jane'
      2  'elroy'
```

Ich benutze `x` sehr gerne beim Einlesen von Daten aus Dateien, indem ich durch das Programm steppe (s.u.) und mir nach jedem eingelesenem Datensatz anzeigen lasse, ob die Daten auch korrekt und vollständig in die Datenstruktur übernommen werden.

Module, Packages und Klassen inspizieren

Geladene Module M

Das Kommando `M` zeigt alle geladenen Module, auch die Pakete, die vom Debugger oder anderen geladenen Modulen geladen werden, an.

```
 DB<1> M
'Carp.pm' => '1.04
     from /usr/share/perl/5.8/Carp.pm'
... Ausgabe gekürzt
```

Angezeigt werden neben dem geladenen Modul die Versionsnummer und auch der Ort von dem das Paket geladen wurde. Das erleichtert die Suche nach Fehlern, die auf unterschiedlichen Modulversionen auf dem Entwicklungs- und Produktionsserver beruhen erheblich.

Inheritance i

Das Kommando `i` zeigt den Vererbungsbaum (@ISA) für geladene Pakete an.

```
  DB<1> use FileHandle

  DB<2> i FileHandle
 FileHandle 2.01, IO::File 1.13, IO::Handle
1.25, IO::Seekable 1.1, Exporter 5.58
```

Dabei geht der Debugger rekursiv durch die Pakete, die vom untersuchten Paket geladen werden.

Ein Perl-Programm im Debugger

Beispielprogramm

Für die nachfolgenden Übungen wird ein kleines Beispielprogramm (`example.pl`) verwendet.

```
1 #!/usr/bin/perl
2 use warnings;
3 use strict;
4 foreach my $number (1..2) {
5  &display($number);
6 }
7 sub display {
8  my $number = shift;
9  $number = sprintf("*** %02d ***\n",
                     $number);
10  print $number;
11 }
```

Dieses simple Programm erzeugt folgende Ausgabe:

```
perl example.pl
*** 01 ***
*** 02 ***
```

Beispielprogramm im Debugger aufrufen

```
perl -d example.pl
...
main::(example.pl:4): \
            foreach my $number (1..2) {
```

Der Debugger stoppt das Programm in der ersten ausführbaren Zeile (hier 4).

Blättern im Code

Mit dem Kommando `l` kann man durch den Code vorwärts blättern, mit `-` rückwärts. Die Angabe von Zeilennummern bzw. Zeilenbereichen, Subroutinen oder Variablen ist ebenfalls möglich.

Beispiel Zeile 1 bis 15 anzeigen

```
 DB<1> l 1-15
1      #!/usr/bin/perl
2:     use warnings;
3:     use strict;
4==>     foreach my $number (1..2) {
5:      &display($number);
6      }
7      sub display {
8:      my $number = shift;
9:      $number = sprintf("*** %02d ***\n", \
                          $number);
10:     print $number;
11     }
```

Doppelpunkte (:) nach der Zeilennummer zeigen an, wo Breakpoints oder Aktionen (s.u.) gesetzt werden können. Der nach rechts zeigende Pfeil (==>) gibt die aktuelle Position im Programm an.

Subroutinen auflisten S

Das Kommando `S [[!] ~pattern]` listet alle Unterprogramme auf, die dem regulären Ausdruck `pattern` (nicht !) entsprechen.

Ohne Muster werden alle Subroutinen, auch die Unterprogramme, die von Paketen oder dem Debugger selbst geladen werden, angezeigt. Diese Liste kann sehr, sehr lang werden.

Daher empfiehlt sich die Angabe eines Pakets

```
 DB<1> S main
main::BEGIN
main::display
```

oder die Verwendung eines Patterns.

```
 DB<2> S display
 Term::ReadLine::Gnu::XS::display_readline_
version
 Term::ReadLine::Gnu::XS::shadow_redisplay
 main::display
```

Hier sieht man sehr schön, das wirklich alle geladenen Subroutinen durchsucht werden.

Variablen anzeigen

Das Kommando `V` zeigt alle Variablen im aktuellen Paket an. Wie bei den Subroutinen werden alle geladenen Variablen angezeigt. Diese Liste kann ebenfalls sehr, sehr lang sein.

Daher empfiehlt sich auch hier die Verwendung des Paketnamens in Verbindung mit einem Suchbegriff (`eq`)

```
 DB<3> V Carp Verbose
$Verbose = 0
```

oder die Verwendung des Paketnamens in Verbindung mit einem regulärem Ausdruck

```
 DB<4> V Carp ~V
$Verbose = 0
$VERSION = 1.04
```

Mit `my` deklarierte Variablen werden jedoch mit `V` nicht angezeigt. Hierzu kann das Kommando `y` verwendet werden, welches das CPAN-Modul PadWalker benötigt.

Das Programm ausführen (continue)

Das Kommando `c [line|sub]` (continue) führt den Code bis zur angegebenen Zeile oder bis zum angegebenen Unterprogramm aus.

Ohne Parameter wird der Code von der aktuellen Zeile bis zum nächsten Breakpoint oder Watch (s.u.) oder bis zum Ende des Programms ausgeführt.

```
main::(example.pl:4): \
      foreach my $number (1..2) {
  DB<1> c
*** 01 ***
*** 02 ***
```

Da hier keine Breaks oder Watches (s.u.) gesetzt wurden, läuft das Programm einfach durch.

Restart R

Sobald das Programm einmal vollständig durchgelaufen ist, ist ein Neustart des Programms im Debugger notwendig. Intern verwendet der Perl Debugger (perl5db.pl) dazu die Funktion `exec()`.

```
 Debugged program terminated.  Use q to quit
or R to restart,
   use o inhibit_exit to avoid stopping after
program termination,
   h q, h R or h o to get additional info.
   DB<1>R
 Warning: some settings and command-line op-
tions may be lost!
 ...
 main::(example.pl:4):    foreach my $number
(1..2) {
```

Mit ActiveState-Perl 5.8.x unter Windows funktioniert das leider nicht. Hier ist ein Neustart des Debuggers auf der Kommandozeile zwingend erforderlich. Dank der History-Funktion von `command.com` ist dies kein wirkliches Hindernis für den Einsatz des Debuggers.

Step Into

`s` führt die nächste Programmzeile aus und springt dabei in die Unterprogramme (Step Into)

Beispiel Step Into

```
     DB<1> s
   main::(example.pl:5):
 &display($number);
     DB<1> s
   main::display(example.pl:8): \
      my $number = shift;
     DB<1> s
```

```
  main::display(example.pl:9): \
$number = sprintf("*** %02d ***\n",$number);
      DB<1> s
    main::display(example.pl:10):     print
$number;
      DB<1> s
    *** 01 ***
    main::(example.pl:5):
&display($number);
```

Step Over

`n` führt die nächste Programmzeile aus und springt dabei über die Unterprogramme (Step Over)

Beispiel Step Over

```
  DB<1> n
main::(example.pl:5):     &display($number);
  DB<1> n
*** 01 ***
main::(example.pl:5):     &display($number);
  DB<1> n
*** 02 ***
```

Aus Subroutinen aussteigen mit return

Das Kommando `r` (return) kehrt aus Unterprogrammen zurück und zeigt den Rückgabewert an. Das ist beispielsweise in Verbindung mit `s` (Step Into) recht nützlich.

```
  DB<1> s
main::(example.pl:5):     &display($number);
  DB<1> s
main::display(example.pl:8): \
      my $number = shift;
```

Der Debugger steht jetzt in der Subroutine display().

```
  DB<1> r
*** 01 ***
void context return from main::display
main::(example.pl:5):     &display($number);
```

Als Bonus wird der Rückgabewert der Subroutine, hier void, angezeigt.

Actions, Breakpoints und Watches

Actions, Breakpoints und Watches auflisten

Das Kommando `L [a|b|w]` listet die gesetzten Actions (a), Breakpoints (b) und Watches (w). Ohne Parameter werden alle Aktionen, Breakpoints und Watches angezeigt.

Breakpoints

Das Kommando `b [line|sub [condition]]` setzt einen Haltepunkt in der angegebenen Zeile bzw. vor Ausführung des Unterprogramms. Zusätzlich kann eine Bedingung (beliebiger Perl-Code) festgelegt werden.

`B (line|*)` löscht den Haltepunkt in Zeile line oder alle Haltepunkte.

Breakpoint beim Einstieg in die Subroutine display() setzen

```
  DB<1> b display
```

und prüfen, ob und wo er gesetzt wurde

```
  DB<2> L
example.pl:
 8:     my $number = shift;
   break if (1)
```

Der Haltepunkt befindet sich in Zeile 8. `c` führt das Programm bis zum nächsten Breakpoint aus.

```
  DB<2> c
main::display(example.pl:8): my $number =
shift;
```

Das Programm bis zum nächsten Haltepunkt weiter ausführen: Hier wird das Unterprogramm einmal durchlaufen.

```
  DB<2> c
*** 01 ***
main::display(example.pl:8): \
      my $number = shift;
```

Das Programm bis zum nächsten Haltepunkt weiter ausführen: Hier wird das Unterprogramm noch einmal durchlaufen.

```
  DB<2> c
*** 02 ***
```

Sobald ein Haltepunkt erreicht wird, stoppt der Debugger das Programm. Jetzt können Variablen mit den bereits vorgestellten Methoden inspiziert und geändert werden.

Beispiel:

Breakpoint in Zeile 5 setzen

```
  DB<1> b 5
```

und das Programm bis zum Haltepunkt ausführen.

```
  DB<2> c
main::(example.pl:5):     &display($number);
```

Die skalare Variable $number inspizieren,

```
    DB<2> x \$number
  0  SCALAR(0x825a11c)
     -> 1
```

ändern

```
  DB<3> $number = 42;
```

und das Programm bis zum nächsten Breakpoint laufen lassen.

```
  DB<4> c
 *** 42 ***
main::(example.pl:5):     &display($number);
```

Die Änderung der Variable erfolgt nur im Debugger, der Originalcode muss nicht (!) geändert werden.

Actions

Das Kommando `a [line] command [condition]` setzt in Zeile Nummer line eine Aktion (beliebiger Perl-Code). Zusätzlich kann eine Bedingung (beliebiger Perl-Code) festgelegt werden. Eine Aktion wird vor der Ausführung der Zeile ausgeführt. `A [line|*]` löscht die Action in Zeile line bzw. alle (*).

Aktion in Zeile 5 setzen.

```
  DB<1> a 5 print "A5a $number "; $number++;
print "A5b $number\n"
```

Und das Programm einmal durchlaufen lassen.

```
  DB<2> c
A5a 1 A5b 2
*** 02 ***
A5a 2 A5b 3
*** 03 ***
```

Übergabeparameter an Subroutinen lassen sich mit Actions aus `@_` ermitteln.

```
  DB<1> a 8 print "Parameter $_[0]\n"

   DB<2> c
Parameter 1
*** 01 ***
Parameter 2
*** 02 ***
```

List Code revisited

Sobald Haltepunkte oder Actions gesetzt sind, werden diese im Programmlisting `l` durch ein b bzw. a nach der Zeilennummer gekennzeichnet.

```
  DB<1> b display
  DB<2> a 5 print "A5 $number";
  DB<3> a 8 print "A8 $number";
  DB<4> l 4-8
4==>    foreach my $number (1..2) {
5:a     &display($number);
6     }
7     sub display {
8:ba    my $number = shift;
```

Ein Haltepunkt hat Vorrang vor einer Aktion.

Variablen beobachten - Watches

Das Kommando `w [expr]` setzt einen Beobachter für expr (beliebiger Perl-Code), während `W (expr|*a)` einen oder alle Watches löscht.

```
  DB<1> w $number
  DB<2> c
Watchpoint 0:    $number changed:
    old value:    ' '
    new value:    '1'
main::(example.pl:5):     &display($number);
```

Graphische Benutzeroberflächen

Als Alternative zur Kommandozeile gibt es auch einige graphische Benutzeroberflächen. Nachfolgend eine kleine Auswahl.

ptkdb - Der Klassiker

http://ptkdb.sourceforge.net/

ddd (DataDisplayDebugger)

http://www.gnu.org/software/ddd/

Eclipse Perl Integration

http://e-p-i-c.sourceforge.net/

Komodo IDE

Kommerzielles Produkt von Active-State
http://activestate.com/products/komodo_ide/

Affrus - für Macs
Kommerzielles Produkt von Late Night Software Ltd
http://www.latenightsw.com/affrus/index.html

Referenzkarten zum Ausdrucken

Andrew Ford: Perl Debugger Reference Card
http://refcards.com/refcard/perl-debugger-forda

The Perl Debugger Quick Reference Card, a PDF courtesy of O'Reilly
http://www.rfi.net/debugger-slides/reference-card.pdf

Perl Debugger Quick Reference
http://www.perl.com/2004/11/24/debugger_ref.pdf

Einführung in Perl Debugger
http://www.ims.uni-stuttgart.de/lehre/teaching/2006-WS/Perl/debugger4up.pdf

Weiterführende Literatur

In diesem Artikel kann ich nicht auf alle Features des Perl-Debuggers eingehen. Wer mehr wissen möchte, sollte sich die nachfolgenden Bücher einmal genauer ansehen.

Perl Debugger Pocket Reference.
Foley, Richard:
Perl Debugger Pocket Reference.
(O'Reilly) ISBN: 0596005032
Klein, fein, handlich, gut.

Pro Perl Debugging: From Professional to Expert
Foley, Richard, Lester, Andy:
Pro Perl Debugging: From Professional to Expert
(Apress) ISBN: 1590594541
FMTEYEWTK: Ein wirklich ausführliches und gutes Buch mit zahlreichen kommentierten Beispielen.

Perl Debugged
Peter Scott, Ed Wright:
Perl Debugged
ISBN: 02011700549
(Out of print, aber noch im Handel erhältlich)
Ein sehr gutes Buch über Perl-Programmierung und Debugging. Meines Erachtens eines der besten Perl-Bücher überhaupt.

Thomas Fahle

Perl 6 Tutorial - Teil 4 : Hashes und Kontrollstrukturen

Wieder mal ein Vorwort

Herzlich Willkommen zu Folge 4. Ein Editor der Wahl ist weiterhin hilfreich, und um die heutigen Beispiele auszuführen, könnte man sich schon Parrot installieren, da Rakudo gerade entscheidende Fortschritte erzielt. Informationen dazu hält die Seite „www.parrotcode.org“ bereit. Wer das scheut, kann aber auch weiterhin Pugs verwenden.

Noch Operatoren übrig?

Nach „Operatoren für Skalare“ und „Operatoren für Arrays“ ist sicher „Operatoren für Hashes“ der nächste logische Titel. Nur leider gibt es keine speziellen Operatoren für Hashes in Perl 6. Außer vielleicht den „dicken Pfeil“ („fat arrow“). Ähnlich dem Komma, das einen Array erzeugt, weist der „=>“ den Interpreter an, einen Hash zu bilden. Deswegen wird er auch manchmal „hash composer“ genannt.

```
$ziffern = (7); # Zuweisung eines Wertes
$ziffern = 7,; # Referenz auf einen Array
%ziffern = 1 => 'adin', 2 => 'dwa';
# geht auch
%ziffern = {1 => 'uno', 2 => 'due'};
```

Da runde Klammern nun noch gruppieren und geschweifte Klammern per default einen anonymen Codeblock darstellen, braucht der Interpreter auch diesen Hinweis um zu erkennen, dass der Programmierer gerne einen Hash hätte. Weil geschweifte Klammern darüber hinaus immer noch für Hashreferenzen stehen können, wird in der letzten Beispielzeile eine solche erzeugt. Nur wird diese - anders als in Perl 5 - automatisch dereferenziert wenn sie einem Hash zugewiesen wird. Dieses Prinzip wurde in der letzten Folge schon an Hand von Arrays demonstriert. In Perl 5 war der „fat arrow“ nur eine lustige Schreibweise für ein Komma. Es half dem Programmierer zu erkennen dass hier ein Hash ensteht, aber er hätte genauso gut schreiben können:

```
%ziffern = (0, 'null', 1, 'eins');
%ziffern = {1, 'jeden', 2, 'dva'};
```

Dies funktioniert auch weiterhin wunderbar. Aber nicht weil nun Hashes immer noch Arrays mit gerader Länge sind, sondern weil es eine Regel gibt, wie das Array in einen Hash(kontext) umgewandelt wird. Und ja, in der zweiten Zeile liefert ein anonymer Block ein Array zurück. In diesem Beispiel stört das nicht. Aber es kann zu sehr unerwarteten Ergebnissen führen, da syntaktisch gleiche Blöcke in Sonderfällen andere Rückgabewerte haben könnten. Um es eindeutig zu halten, können die Operatoren `%` und `hash` den Hashkontext erzwingen und anzeigen.

```
$hashref = hash{ 0, 'null', 1, 'eins' };
$hashref = %( 0, 'null', 1, 'eins' );
# oder klassisch:
$hashref = {'1' => 'a', '2' => 'b',
     '3' => 'c'}
# oder auch \('1' => 'a', '2' => 'b',
#     '3' => ,c')
$hashref = %( 1..3 Z 'a'..'c' );
```

Ein Hash kann aber auch in einen anderen Kontext überführt werden. Im boolschen Kontext meldet er, ob er überhaupt Schlüssel besitzt, im numerischen Kontext die Anzahl der Schlüssel und im Stringkontext gibt er seinen gesamten Inhalt als String aus.

```
my %buch = ( 1..3 Z 'a'..'c');
? %buch # Bool::True
+ %buch # 3, entspricht %buch.elems
~ %buch # "1\ta\n2\tb\n3\tc\n"
```

Weit praktischer ist aber meistens die Ausgabe der folgenden, selbsterklärenden Hashmethoden, welche ein Array zurückgeben.

```
# (1, 2, 3), entspricht keys %buch
%buch.keys;
# ('a', 'b', 'c'), auch values %buch
%buch.values;
%buch.kv; # (1, 'a', 2, 'b', 3, 'c')
```

`keys` und `values` sind altbekannte Perl 5-Befehle, die es selbstverständlich auch weiterhin gibt. Aus ihren Initialien leitet sich der Name der Methode `.kv` ab, die abwechselnd zugehörige Schlüssel und Werte in einem Array aufzählt. Etwas anders funktioniert dagegen die Methode `.pairs`. Sie liefert key und value in einem Skalar, also ein Array von Wertepaaren. Das Gleiche erhält man auch, wenn ein Hash einem Array zugewiesen wird.

```
# (1 => 'a'), (2 => 'b'), (3 => 'c')
%buch.pairs;
@buch = %buch; # dito
```

Werdet ein Paar

Key und value in einem Skalar? Das mag zuerst nach einem Array oder vielleicht einer Junction klingen, ist aber einfach ein Wertepaar. Das Paar gehört zu den völlig neuen Datentypen, die Perl 6 einführt. Eigentlich sind es auch Paare, die mit dem „fat arrow“ erzeugt werden. In den Synopsen wird „=>“ desweg oft auch „pair constructor“ genannt.

```
my $pair = 'Jakub' => 'Helena';
$pair.isa(Pair) # ergibt Bool::True
$pair.key # 'Jakub'
$pair.value # 'Helena'
```

Paare können aber auch noch anders generiert werden.

```
my $pair = :Jakub('Helena'); # oder kürzer:
# <> ist das neue qw()
my $pair = :Jakub<Helena>;
```

Selbstverständlich funktioniert auch diese Schreibweise für Hashes.

```
%regent = :Atreides<Paul>, :
Harkonnen<Wladimir>, :Ordo<Executrix>;
$hashref = {pair <Atreides Paul Harkonnen
            Wladimir Ordo Executrix> };
```

Allerdings dient das nicht der optischen Vielfalt, sondern zeigt, dass Paare mehr Aufgaben haben als Hashbausteine zu sein (obwohl sie dazu sehr geeignet sind). Ungeordnete Listen von Wertepaaren mit eindeutigen Schlüsseln entsprechen sicher eher der Idee eines Hashes als Listen mit gerader Länge. Das nächste Beispiel ist ein bewusst herbeigeführter Unfall, der die Schwäche von Perl 5 demonstriert. Einzelne Werte die dem Hash entnommen wurden können jede Zugehörigkeit zwischenden Schlüsseln und Werten zerstören.

Perl 5:

```
my %h = (1=>2, 3=>4);
%h = grep {$_ % 2} %h;
say %h; # {1 => 3 }
```

Zum Glück wurde für Perl 6 auch diese Schmuddelecke aufgeräumt.

```
%h = grep {$_.value % 2} %h.pairs;
```

Diese Fassung macht nicht nur mehr Sinn, sondern ist auch noch lesbarer, da hier explizit steht, was geschieht. Sie kann auch keine unsinnigen Ergebnisse liefern, da korrekterweise immer Paare durchgereicht werden. Doch um zu beweisen dass die neue Ordnung nicht einschränkt, hier das Perl 5-Beispiel in Perl 6-Syntax.

```
%h = grep {$_ % 2} %h.kv;
```

Parameterpaare und $@%

Mit Paaren lassen sich wundersame Datenstrukturen wie Hybride aus Hash und Array erzeugen oder Paarlisten, in denen Schlüssel mehrfach vorkommen können. Aber die wichtigste Verwendung von Paaren außerhalb der Hashes sind benannte Parameter. Diese wurden in Perl 5 oft mit Hilfe von Hashes simuliert. Jedoch bietet Perl 6 dafür einen expliziten Mechanismus der viele interessante Techniken ermöglicht. Subroutinen und Parameterübergaben sind aber das Thema der nächsten Folge. Doch selbst „Kernfunktionen“ der Sprache haben benannte Parameter, welche die zweite Syntax der Paare benutzen. Die in Folge 2 vorgestellten Dateitestoperatoren `:r` `:w` `:e` sind nichts weiter als Attribute des Objektes vom Typ Filehandle. Dabei drückt die Formulierung `:r` aus: „Ich hätte gerne den Wert, der unter dem Schlüssel „r“ gespeichert ist“. Ähnliches gilt auch beim Öffnen einer Datei.

```
my $fh = open "datei.txt", :r;
```

Die Datei wird im Lesemodus geöffnet, indem der benannte Parameter „r“ gesetzt wird. In dem Beispiel fällt auch auf, dass Filehandle kein eigener Variablentyp sind und auch keinen eigenen Namensraum mehr besitzen. Auch Verzeichnishandler und Formate werden nun in Skalaren gespeichert. Die Formate weichen dazu aus dem Sprachkern in ein Modul. Auch Token wie `__LINE__` oder `__PACKAGE__`, die man als Systemvariablen mit eigenem, sonderbaren Namensraum ansehen könnte, sind in ihrer neuen Schreibweise `$?LINE` und `$?PACKAGE` eher als Variablen erkennbar. Ihr seltsamer Namensraum nennt sich nun „compiler hint variables“. Er umfasst Informationen, die ab dem Zeitpunkt der Kompilierung feststehen und auch nicht geändert werden dürfen. Eine Zuweisung wie `$?FILE = 'was anderes';` würde einen Compile-Error ausgeben. Dieser Sonderstatus wird mit der sekundären Sigil (Twigil) „?“ markiert. Achtung: In Perl 6 bezeichnet Token eine sehr einfache Form einer Regex.

Die Streichung der 4 Sondernamensräume für Variablen (Handler, Formate und Token) sind weitere Beispiele des Grundsatzes: „keine Ausnahmen“ und der Forderung nach einem deklarativen Syntax, der Inhalt eines Ausdrucks deutlicher sichtbar macht. Die verbliebenen Namensräume können eindeutig an der Sigil unterschieden werden. Oder einfacher: $wunder, @wunder und %wunder sind drei verschiedene Variablen.

Datentyp Code

Der vierte Namensraum ohne Sigil, ist der der Subroutinen. Und wie bei Hashes und Arrays, können Referenzen auf Routinen ebenfalls in Skalaren gespeichert werden.

```
my $coderef = sub { say "Moin #perl" }
```

Die besitzen dann den Datentyp „Code“. (Das fehlende Semikolon ist kein Fehler. Perl 6 verlangt nur dann ein solches, wenn mehrere derartiger Deklarationen ohne den Befehl `sub` in eine Zeile schreibt. Als Eselsbrücke: ein „}“ am Zeilenende impliziert ein Semikolon.) Das `sub` könnte man also auch weglassen, was nichts am Datentyp und an der Benutzung der Variable ändert.

```
my $coderef = { say "Moin #perl" }
# dito, ebenfalls noch möglich
my $cr = \{ say "Moin #perl" }
$cr.isa(Code) # ergibt Bool::True
$coderef(); # führt den Block aus
```

Trotzdem sind die Unterschiede zwischen einer „sub“ und einem Block so zahlreich, dass man damit viele Seiten füllen kann, sie alle aufzuzählen. Deshalb wird diese Folge auch eher Blöcke behandeln und erst die nächste die Sonderform „Subroutine“. Einfache Blöcke besitzen nämlich nur Grundfähigkeiten und Befehle wie `while`, `if`, `package` und `sub` fügen dem weitere Funktionalität hinzu, weswegen sie im Perl 6-Jargon Blockmodifikatoren genannt werden. Da diese Modifikatoren „builtins“ und keine Routinen sind, kann man sich jetzt nicht nur die runden Klammern sparen. Es wird empfohlen auf sie zu verzichten. Und wenn nicht anders möglich, sollte man zumindest darauf achten, zwischen Modifikator und den runden Klammern Leerzeichen zu lassen, es sei eine `sub` namens `if` möchte angesprochen sein.

```
if $a > 6 {
     say "Zu alt für den Kindergarten"
}
if($a > 6) { ... } # Error
```

Doch beginnen wir mit einem nackten Block.

Closure

Blöcke sind in Perl 6 alles, was von geschweiften Klammern umschlossen ist, soweit nicht darin ein => oder `pair` steht, oder die voranstehenden Operatoren `%` oder `hash` eine Hashreferenz erzwingen. Sie teilen Code in logische Einheiten. Dadurch werden die Quellen übersichtlicher und verständlicher. Es erlaubt aber auch Variablen einem Block zuzuordnen. Das hat den großen Vorteil, dass man damit heimtückische Überschneidungen mit gleichnamigen Variablen anderer Programmteile ausschließen kann. Außerdem spart es Arbeitsspeicher, wenn die Lebensdauer einer Variable auf den Block begrenzt wird, in dem ihr Inhalt gebraucht wird.

Der erste Versuch in die Richtung war der Befehl `local`, der bereits vor Perl 5 eingeführt wurde. Doch der überlagert eine eventuell exisierende, gleichnamige Variable temporär, bis die Ausführung des aktuellen Blocks beendet ist. Deswegen wurde die Anweisung in `temp` umbenannt. Eine echte Bindung zwischen Variable und Block erzeugt `my`, welches Perl 5 brachte. Seitdem wird eindringlich empfohlen `my` anstatt `local` zu verwenden. Wenn nämlich aus einem Block (in dem eine `local`-Variable definiert wurde) eine Routine aufgerufen wird, überschreibt die „lokale“ Variable dort

gleichnamige Brüder. Da das selten den Erwartungen entspricht, sollte man es mit `my` oder `state` verhindern. Eine so erzeugte, lexikalisch lokale Variable wäre innerhalb der Subroutine unbekannt und wird nur eingeblendet, während ein Befehl des betreffenden Blocks ausgeführt wird. Einziger Unterschied zwischen beiden Befehlen: `my` initialisiert die Variable nach jedem Neustart des Blocks von neuem mit dem Anfangswert, wohingegen `state` das nur beim ersten Mal tut und sich danach den Inhalt bis zum nächsten Blockaufruf merkt. Dieses Verhalten von `state`-Variablen nennt man in der Informatik „Closure“ und ist seit Jahren Bestandteil funktionaler Programmierung. Es ließ sich auch mit einem zusätzlichen, umschließenden Block und `my` simulieren, was aber umständlich und fehleranfällig war. Für einfache Closures wurde deshalb der `state`-Befehl von Perl 6 nach 5.10 zurückportiert.

Einfache Ausführung mit Rückhand

In Blöcken stehen meist mehrere Befehle, von denen der zuletzt ausgeführte den Rückgabewert des Blocks angibt. Um ihn ihn einer Variable zuzuweisen, muss der Befehl `do` verwendet werden, auch wenn nackte Blöcke immer ausgeführt werden, solang sie keine Parameter an eine Routine sind. (wie bei map, sort etc.)

```
my $zahl = do {
     rette_regenwald(); glückszahl() }
# oder länger :
my $coderef = {
     rette_regenwald(); glückszahl() }
my $zahl = $coderef();
```

Auch auf einzelne Befehle, die nicht von geschweiften Klammern umgeben sind, kann `do` angewendet werden, um sie als Ausdruck zu evaluieren. Daher ist auch diese umständliche Umschreibung des ternären Operators möglich.

```
$x = do if $a { $b } else { $c };
$x = $a ?? $b :: $c; # kürzer
```

Da `do` einen Block immer genau einmal ausführen lässt, werden solche Abschnitte jetzt auch „do once loop“ genannt. Um mit dieser einfachen Regel nicht zu brechen, wurden Schleifenkonstrukte wie `do { ... } while ()` in `repeat { ... } while ()` umbenannt. Auch alle weiteren Verwendungsmöglichkeiten von `do` die Perl 5 kennt, wurden gestrichen. Dafür führte Larry mit `gather` eine Variante von `do` ein, die einem Block erlaubt eine Liste zurückzugeben. Die einzelnen Werte werden mit dem Befehl `take` einzeln oder in Gruppen ausgewählt.

```
# enthält: 'Gauguin', 'Cezanne'
my @maler = gather {
   my $r = take 'Gauguin';
   # $r enthält 'Gauguin'
   take 'Cezanne' if $r;
}
# enthält: 1,10,2,20
my @x = gather for 1..2 {
   take $_, $_ * 10;
}
```

In Kombination mit Schleifen und bedingten Anweisungen entfaltet `gather/take` erst seine volle Mächtigkeit. Die Beispiele zeigen auch wie Blockmodifikatoren ohne weitere Klammern kombiniert werden können, was eine Konsequenz der bereits vorgestellten Regeln ist. Auch bei nachgestellten Konstrukten kann damit manche Zeilen vereinfacht werden.

Perl 5:

```
{ print if $_ % 2 } for 0..10;
```

Perl 6:

```
.print if $_ % 2 for ^11;
```

Soll in Perl 6 ein Befehl (als Methode) auf `$_` angewendet werden, sollte er mit einem Punkt beginnen. Die genauen Hintergründe dieser Designentscheidung wurden in der ersten Folge erläutert. Doch zurück zu den Rückgabewerten. So wie `take` sie für den umgebenden `gather`-Block bestimmt, gibt es auch einen Befehl für den skalaren Rückgabewert, der gleichzeitig die Ausführung des Blocks abbricht. Dieser Befehl heißt zur allgemeinen Überraschung nicht `return`. `return` ist Subroutinen und artverwandten Funktionseinheiten vorbehalten, aus Gründen, die in der nächsten Folge genau beschrieben werden. Das schlichte Verlassen des umschließenden Blocks wird von einem `leave` ausgeführt. Soll ein bestimmter Block beendet werden, so muss dessen Anfang durch ein „Label“ (LABELNAME:) markiert sein, um mit `LABELNAME.leave` aus ihm hinauszutreten. Daraus folgt ebenfalls, dass Perl 6 die mittelalterlichen Freuden extensiver `goto`-Benutzung keineswegs einschränkt. Selbst ein fortraneskes „computed goto“ kann unverändert (wie in Perl 5) geschrieben werden.

Der mit `leave` definierte Rückgabewert kann aber noch mehr, als nur in eine Variable zugwiesen zu werden. Er kann auch die Ausführung von Blöcken steuern, die mit Auswahloperatoren verknüpft sind.

```
tu_was() && das_auch();
# ist andere Schreibweise für:
{ tuwas() } && { das_auch() };
# ist andere Schreibweise für:
{ tuwas() } and { das_auch() };
```

Doch `&&`, `||` und `^^` wurden schon in Folge 2 behandelt und für die Ops macht es keinen Unterschied ob sie die Ergebnisse von einzelnen Befehlen oder ganzen Blöcken verknüpfen. Wirklich neu und speziell nur für Blöcke sind hingegegen `andthen` und `orelse`. Grundlegend entsprechen sie den Befehlen `and` und `or`. Beendet der linke Block erfolgreich, lässt `andthen` ebenfalls den rechten ausführen. Allerdings, um die Kontextvariable `$_` weiterhin benutzen zu können. Sie wird in diesem Sonderfall sozusagen „Closure“ zweier Blöcke. `orelse` eignet sich vor allem zur Fehlerbehandlung. Ist der linke Rückgabewert nicht positiv, startet `orelse` den rechten Block, in dem die Fehlermeldung (Sondervariable $!) weiterhin bekannt ist.

Bedingte Ausführung ist einfach

In der Regel werden jedoch bedingte Anweisungen mit den vertrauten Schlüsselwörtern `if`, `elsif`, `else` und `unless` formuliert. Letzeres ist zwar eine Eigenheit von Perl, die nicht mehr mit `elsif` und `else` kombiniert werden darf, aber die Funktionsweise dieser 4 Befehle sollte keiner weiteren Erklärung bedürfen. Auch `given`, `when` und `default` wurden mit ihrer Einfürung zur Version 5.10 genügend vorgestellt. Vergessen wurde dabei manchmal lediglich, dass `given` und `when` sich auch in anderen Zusammenhängen benutzen lassen. `given` eignet sich hervorragend umständliche Ausdrücke zu vereinfachen wie folgende:

```
$tiefer.objekt.namensraum.var1 =
$tiefer.objekt.namensraum.var2 +
$tiefer.objekt.namensraum.var3;
# oder :
given $tiefer.objekt.namensraum
{ $_.var1 = $_.var2 + $_.var3 }
```

Da `given` ebenso wie `for` (es gibt kein `foreach` mehr) die Kontextvariable setzt, können `when`-Klauseln auch in `for`-Schleifen verwendet werden.

```
for @beine {
  print "Ist $_ ";
  when 1 { say "überhaupt ein Tier?" }
  when 6|8|10 { say "ein Insekt?" }
  when > 20 { say "eine Raupe?" }
  default { say "ein Zweibein"}
}
```

`default` ist nur ein Alias auf `when *` (oder `$_ ~~ Whatever`) und im Gegensatz zu einem „C-switch“ entfällt auch das notorische `break`, da ein implizites `break` jede `when`-Klausel beendet. Wer mag kann aber auch jederzeit mit `break` die Klausel und den umgebenden „contextualizer“-Block manuell verlassen. Wenn nur zur nächsten Klausel gesprungen werden soll, dann ist `continue` das Mittel der Wahl. Noch weitere besondere Sprungbefehle gibt es für Schleifen.

Schöne Schleifen

Auch hier wurden die alten Standarts kaum angerührt. `while` und `until` blieben unverändert, lediglich ein `repeat { ... } while $a < 3;` ist gewöhnungsbedürftig. Ansonsten wird der Ausdruck nach `while` und `until` (ebenso wie bei `if` und `unless`) in den boolschen Skalarkontext evaluiert. Selbstverständlich wird dann bei einem Ergebnis von Bool::True (beziehungsweise Boole::False) der Block ausgeführt. `for` ist jetzt ein reiner Array-Iterator und evaluiert entsprechend in den „lazy list“-Kontext. Genau deshalb wird die zeilenweise Abarbeitung einer Datei jetzt so geschrieben:

```
my $fh = open "aufsatz.txt";
for =$fh {
...
```

Würde jemand `while` anstatt `for` nehmen, so lieferte `=$fh` im Skalarkontext den gesamten Dateiinhalt auf einmal ab. Bis dahin ist alles logisch, doch was geschah mit der klassischen „C-Stil“-for-Schleife. Ihr Ausdruck wird eindeutig in den void-Kontext evaluiert und deshalb bekam sie ein eigenes Schlüsslwort. Inhaltlich unterschiedliches soll auch optisch unterscheidbar sein. Deswegen heißt die Allzweckschleife nun „Schleife“ (`loop`). Da der leere Kontext immer das selbe Ergebnis liefert, ist auch `loop { ... }` (ein Synonym für `loop (;;) { ... }`) eine nettere Schreibweise für `while 1 { ... }`, das weiterhin möglich ist. Eine weitere gute Nachricht: die Sprungbefehle `last` (verlasse die Schleife), `redo` (wiederhole den Durchgang) und `next` (zur nächsten Iteration) funktionieren jetzt einwandfrei. Bei `do { ... } while ();` hat hier Perl 5 immer noch Probleme, denn das sind nachträglich eingeführte Konstrukte und keine echten Schleifen. Wie `leave` können auch diese Sprungbefehle mit Labelmarken wie z.B. `LABEL.next;` oder `next LABEL;` kombiniert werden.

Blockparameter

Im letzten Beispiel fehlte noch ein entscheidendes Detail. Für `for`-Schleifen wird oft eine lokale Laufvariable definiert:

Perl 5:

```
for my $i (1..10) { ...
```

Weil hier das `my` nicht innerhalb des Blockes steht, für das es seine Gültigkeit besitzt, würde das gegen die klaren Regeln in Perl 6 verstoßen. Deshalb gibt es dafür nun einen besonderen Syntax der es gleichzeitig zulässt mehrere Parameter an einen Block zu übergeben. Da der Syntax symbolisieren soll, dass hier schnell ein paar Werte in den Block „hineingeworfen“ werden, nennt sich diese Formulierung „pointy block“. Sie erlaubt elegante Lösungen, für die bisher wesentlich kompliziertere Formulierungen notwendig waren.

```
for 1..10 -> $i { ...
for %h.kv -> $k, $v { ...
for @a Z @b Z @c -> $a, $b, $c { ...
while berechne() -> $wert { ...
```

Auch wenn `while` in den boolschen Kontext evaluiert, bekommt `$wert` den Betrag den `berechne()` liefert. Wie beschrieben sind diese Variablen Parameter und nicht lexikalisch lokal und können daher auch nicht innerhalb des Blocks verändert werden. Braucht man jedoch unbedingt veränderbare Laufvariablen, so muss dies mit einem leicht geänderten Syntax gekennzeichnet sein.

```
for 1..10 <-> $i { ...
```

Wenn kein Variablennamen angegeben wäre, landen die Inhalte natürlich in $_ oder @_. Nun ist `@_[2]` nicht immer der schönste Variablennamen und verwirrt manch einen Neuling. Für solche Fälle erfand Larry die automatisch benannten Parameter die auch einen weiteren Sonderfall in Perl 5 mit einer allgemeinen Regel ersetzen. Damit ist der `sort`-Befehl gemeint, in dessen Block die einmaligen Sondervariablen $a und $b bekannt sind, welche jetzt `$^a` und `$^b` geschrieben werden. Die Twigil (sekundäre Sigil) „^“ kennzeichnet die Platzhalter-Variablen. Diese dürfen benutzt werden, ohne ihre Namen zu deklarieren. Man muss nur darauf achten, dass alphabetisch sortiert, ihre Namen der Reihenfolge der Parameter entsprechen, in der sie dem Block übergeben wurden.

```
for 1..10 { say $^i ...
for 1..10 { say $^cc ...
$coderef1 = { say $^a, $^b };
$coderef2 = { say $^z, $^zz };
$coderef1('a','d'); # sagt ade
$coderef2('a','d'); # dito
```

Mehr dazu in Teil 5, wenn in Subroutinen die hohe Schule der Parameterübergabe zelebriert wird.

Herbert Breunung

FrOSCon

Free and Open Source Software Conference - 2008

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

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

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

kontakt@froscon.de - www.froscon.de

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

PDF Rechnungen - Beispiel Automatische Rechnungsgenerierung

Dieses Beispiel soll eine tatsächliche Anwendung von „PDF mit Perl“ zeigen. Bei $foo kommen Bestellungen als E-Mails an und aus diesen Mails werden dann automatisch Rechnungen generiert. Dies wurde notwendig, um Bestellungen schneller abarbeiten zu können. Der Workflow ist in Abbildung 1 dargestellt.

Beim Auslesen der Mails muss dabei noch darauf geachtet werden, dass kein Spam oder sonstige E-Mails geparst werden. Die Identifizierung erfolgt über den Betreff und den Sender der Mail. Unerwünschte Mails werden sofort gelöscht. Nach dem die Mail ausgelesen und geparst wurde, wird für den Kunden eine neue Rechnung erzeugt. Dazu liegt eine Rechnungsvorlage im PDF-Format vor. Diese Vorlage wird mit den Kundendaten ergänzt und die daraus resultierende Rechnung wird als neue PDF-Datei gespeichert und ausgegeben.

Die E-Mails sind ganz normaler „Menschenlesbarer“ Text und nicht im YAML-, XML- oder sonst einem Format. Das hat damit zu tun, dass die Anwendung „historisch“ gewachsen ist und diese Funktionalität auf die Schnelle implementiert werden musste. Der E-Mail-Parser ist dennoch relativ einfach und funktioniert einwandfrei. Dieser besteht aus einfachen Regulären Ausdrücken.

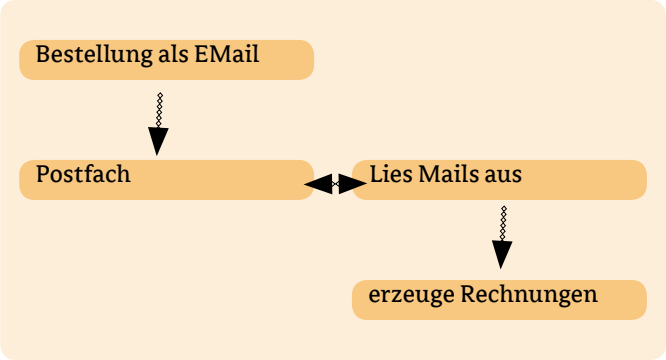

Abbildung 1

Für das Auslesen des Mail-Accounts erfolgt mit dem Modul Mail::POP3Client. Hier wird ein normaler Mail-Account-Zugang genommen, da das Tool sowohl auf dem Webserver als auch in einer eigenen GUI läuft, die mit Perl/Tk umgesetzt wurde (siehe Listing 1).

Mit dem Objekt kann man auf jede einzelne Mail zugreifen und Informationen über die Mail bekommen. Etwas umständlich ist, dass hier nicht mit einem Iterator gearbeitet wird, sondern dass man immer über die Mail-Nummer auf die Informationen zugreifen muss. Für jede Mail wird erst der Header überprüft, ob es eine Bestell-Email ist oder nicht. Ist dies nicht der Fall, wird die Mail gelöscht und es geht mit der nächsten Mail weiter. Für Bestellmails werden in der Funktion `_parse_order` die Details wie Adresse und bestellte Ware geparst. Alle Bestellungen werden als Array zurückgeliefert.

In einem weiteren Modul wird dann die Rechnung generiert. Am Anfang wurde mit OpenOffice eine Dummy-Rechnung als PDF erzeugt. Diese Dummy-Rechnung beinhaltet schon unsere Unternehmensadresse, eine Grafik und ein paar feststehende Informationen. Damit wird die weitere Generierung vereinfacht, weil man sich nur noch um die Platzierung der dynamischen Inhalte kümmern muss (siehe Listing 2).

Für die Erzeugung der Rechnungen verwenden wir `PDF::Reuse`, da es nicht so viele Abhängigkeiten wie `PDF::API2` hat und uns die Funktionalitäten ausreichen. Es muss nur etwas Text positioniert werden. Für jede Bestellung wird jetzt eine Rechnung erzeugt. Dass pro Bestellung eine Datei erzeugt wird, hängt damit zusammen, dass `PDF::Reuse` anscheinend nicht mehrmals die gleiche Datei als Vorlage verwenden kann. Jedenfalls gab es immer einen Fehler wenn wir versucht haben, eine einzige Datei mit allen Rechnungen zu generieren.

```
sub get_mails{
    my $pop = Mail::POP3Client->new(
        USER     => "ACCOUNT_USER",
        PASSWORD => "ACCOUNT_PASSWD",
        HOST     => "HOSTNAME",
    );

    MAIL: for my $i ( 1..$pop->Count ) {
        # check subject
        for my $headline( $pop->Head( $i ) ) {
            next unless $headline =~ /^Subject:\s/i;
            unless( $headline =~ /^Subject:\s+\[\$foo\]\s+(?:Abo-)?Bestellung/ ){
                # delete mail
                $pop->Delete( $i );
                next MAIL;
            }
        }

        # get order details
        my $body = join "\n", $pop->Body( $i );
        my %order_info = _parse_order( $body );
        push @orders, \%order_info;

        # delete mail
        $pop->Delete( $i );
    }
    $pop->Close();

    return @orders;
}
```

Listing 1

Für ausländische Kunden können wir die Rechnungen (noch) nicht automatisch generieren, da hier noch Versandmehrkosten hinzugerechnet werden müssen, die von Land zu Land unterschiedlich sind und auch von der Anzahl der bestellten Hefte - bzw. vom Gewicht - abhängt. Ein Modul zur Abfrage bei der Post ist aber in Arbeit.

Mit `prFile` wird festgelegt, in welche Datei die Rechnung geschrieben wird. Danach wird mit `prForm` unsere Vorlage geladen und mit allen weiteren Befehlen von `PDF::Reuse` wird diese importierte Vorlage (nicht das Original) verändert. Ein Nachteil von `PDF::Reuse`, der in diesem Tool aber keine Rolle spielt, ist, dass immer auf der aktuellen Seite gearbeitet wird. Man kann eine Seite nicht in einem Objekt oder einem normalen Skalar speichern und dann am Ende des Programms noch einmal darauf zugreifen.

In den Funktionen wie `_print_info` werden die Bestelldetails auf der Vorlage platziert. Die Platzierung ist etwas mühsam, da immer genau angegeben werden muss, wo etwas stehen soll und es gibt kein richtiges Hilfstool zur Bestimmung der Koordinaten. Ein Problem dabei ist, dass unterschiedliche Module den Koordinatenursprung unterschiedlich platzieren.

Die Platzierung der Details erfolgt dann mit `prText`. Mit `PDF::Reuse` kann aber nicht nur einfacher Text eingefügt werden, sondern es besteht die Möglichkeit Bilder, JavaScript und andere Elemente hinzuzufügen.

Damit die Inhalte nicht verlorengehen und die Rechnung auch wirklich auf der Platte liegt, muss noch `prEnd` aufgerufen werden, das den Puffer leert.

Zum Schluss wird in der Funktion `create_bill` noch eine große PDF-Datei erzeugt, in der alle Einzelrechnungen zusammengeführt werden. Dies wird gemacht, damit nur eine einzige Datei geöffnet, gedruckt und ggf. bearbeitet werden muss. Mit `prFile` wird wieder eine neue Datei erzeugt und mit `prDoc` werden die Einzelrechnungen importiert. Der Unterschied zu `prForm` besteht darin, dass diese Seite nicht als Formular bzw. Hintergrund verwendet wird.

`PDF::Reuse` ist für solche kleinen Aufgaben sehr gut geeignet. Wenn man darauf achtet, welche Funktionen automatisch importiert werden, sollten auch keine Konflikte mit eigenen Funktionen auftreten.

Renée Bäcker

```
use PDF::Reuse;

my $temp_path = '/temp_verzeichnis/';
my $file      = $temp_path . 'Vorlage.pdf';

sub create_bill{
    my ($class,$orders) = @_;

    my @files;
    for my $order ( @$orders ){

        next unless $order->{customer}->{country} eq 'D';

        my $r_nr    = $order->{r_nr};
        my $bill    = $temp_path . 'Rechnungen' . $r_nr . '.pdf';
        push @files, $bill;

        prFile( $bill );
        prForm( $file );
        _print_info( $r_nr, $order->{customer}->{cust_nr} );
        _print_customer( $order->{customer} );
        my $x = _print_articles( $order->{article} );
        _print_footer( $x );

        prEnd();
    }

    my $summary = $temp_path . 'Rechnungen' . time . '.pdf';
    prFile( $summary );
    prDoc( $_ ) for @files;
    prEnd();

    unlink @files;

    return $summary;
}

sub _print_info{
    my ($rnr) = @_;

    my @info = (localtime time)[3..5];
    my $date = sprintf "%02d.%02d.%04d", $info[0], $info[1]+1, $info[2]+1900;
    prText( 468, 615, $date );

    my ($kdnr) = $rnr =~ /^(.*?)-/;
    prText( 170, 532, $kdnr );
    prText( 190, 518, $rnr );
}
```

Listing 2

Der Flip-Flop-Operator

Hinter dem Flip-Flop-Operator versteckt sich der Range-Operator. Dessen gebräuchlichste Verwendung ist es wohl, Listen zu erstellen. In vielen Programmen sieht man so etwas wie

```
for( 1..10 ){
   # do anything
}
```

Hier wird eine Liste von 1 bis 10 aufgebaut. Mit dieser Eigenschaft dürften wohl die meisten den Range-Operator kennen. Weit seltener kommt der Range-Operator mit seiner Eigenschaft als Flip-Flop vor.

Im Skalaren Kontext liefert der Range-Operator einen Boolschen Wert zurück und ist somit „bistabil“. Diese Flip-Flop-Eigenschaft eignet sich zum Beispiel sehr gut, wenn man einen bestimmten Bereich aus einem Text ausgeben will und man nicht selbständig mit if's einen Boolschen Zustand pflegen will.

In Listing 1 ist ein Textausschnitt zu sehen, von dem nur der Text zwischen `START` und `STOP` interessant ist.

```
Dies ist ein längerer Text
mit vielen unnötigen Zeilen
vor dem eigentlich Wichtigen.

START
Wichtiger Text
über drei
Zeilen
STOP

Und noch unwichtigeren Zeilen
nachher
```

Listing 1

Listing 2 zeigt einen Code, wie man es mit einem eigenen „Flip-Flop“ machen kann (Der Text aus Listing 1 ist im `__DATA__`-Bereich):

```
#!/usr/bin/perl

use strict;
use warnings;

my $bool = 0;

while( my $line = <DATA> ){
   if( $line =~ /^START/ ){
      $bool = 1;
   }

   print $line if $bool;

   if( $line =~ /^STOP/ ){
      $bool = 0;
   }
}
```

Listing 2

Der Flip-Flop ist solange `false` wie der Ausdruck auf der linken Seite falsch ist. Ist der linke Ausdruck wahr, wird der Flip-Flop auch `true` und bleibt die solange bis der Ausdruck auf der rechten Seite wahr ist. Danach wird der Flip-Flop wieder `false`. Damit wird es recht einfach, den wichtigen Teil aus dem Text zu ziehen (siehe Listing 3).

```
#!/usr/bin/perl

use strict;
use warnings;

my $bool = 0;

while( my $line = <DATA> ){
   print $line if $line =
     ~ /^START/ .. $line =~ /^STOP/;
}
```

Listing 3

Wichtig ist noch zu wissen, dass jeder Flip-Flop sein eigenen Boolschen Status hat, so dass Flop-Flops auch verschachtelt werden können (siehe auch Listing 4).

```
if( $line =~ /^START/ .. $line =~ /^STOP/ ){
     if( $line =~ /ber/ .. $line =~ /drei/ ){
        chomp $line;
        print '*' . $line . "*\n";
     }
     else{
        print $line;
     }
  }
```

Listing 4

In diesem Listing ist aber auch ein anderes Phänomen dargestellt. Der Flip-Flop kann im selben Durchlauf erst `true` und gleich wieder `false` werden, der `if`-Block wird dennoch mindestens das eine Mal ausgeführt. Das hat damit zu tun, dass der rechte Ausdruck evaluiert wird, sobald der Range-Operator evaluiert wird. Dadurch ergibt sich die Ausgabe:

```
START
Wichtiger Text
*über drei*
Zeilen
STOP
```

Möchte man dieses Verhalten unterbinden, soll der rechte Ausdruck also erst im nächsten Durchlauf evaluiert werden, muss man statt `..` einfach `...` verwenden. Dadurch ergibt sich folgende Ausgabe:

```
START
Wichtiger Text
*über drei*
*Zeilen*
*STOP*
```

In den meisten Fällen liefern aber sowohl `...` als auch `..` die gleichen Ergebnisse.

Renée Bäcker

TPF-TICKER

Google Summer of Code 2008 -> Projekte der TPF

Beim diesjährigen „Google Summer of Code“ ist nach einer Pause auch die Perl-Foundation wieder vertreten.

Der Perl-Foundation wurden 6 Slots zugeteilt:

* Flesh out the Perl 6 Test Suite
* wxCPANPLUS
* Native Call Interface Signatures and Stubs Generation for Parrot
* Incremental Tricolor Garbage Collector
* Math::GSL
* Full Text Search Implementation for the content management system, Bricolage

Stuttgart-pm

Stuttgart gilt als bedeutender IT-Standort. Das perfekte Klima für Perl Mongers – dennoch dauerte es bis zum Frühjahr 2003, bis Christian Renz den Anfang machte und zur Gründung von Stuttgart.pm aufrief. Und im August 2003 ging es dann richtig los. Neben dem gemeinsamen Fokus auf die Lieblingssprache gab es noch einen anderen handfesten Grund, warum es gerade zu diesem Zeitpunkt zur Gründung einer Stuttgarter Perlmongers-Gruppe kam. Im Jahr 2004 stand ein neuer Perl-Workshop an, für den zum damaligen Zeitpunkt noch kein Standort feststand und den die Stuttgarter gerne in ihrer Region veranstalten wollten. So kam es auch: der Perl-Workshop 2004 fand in Schorndorf statt, keine 30 Kilometer von Stuttgart entfernt. Die lokale Organisation wurde hauptsächlich von Rolf Schaufelberger durchgeführt und von der WRS (Wirtschaftsförderung Region Stuttgart) mit einem kleinen Betrag unterstützt. Zu den Vortragenden gehörten unter anderem Autrijus Tang und Leopold Toetsch.

Das erste Stuttgart.pm-Treffen war am 2. September 2003, und seit dieser Zeit treffen sich die Stuttgarter Perl Mongers wenn möglich an jedem ersten Dienstag im Monat. Die Organisation erfolgt über die Mailingliste, so dass die üblichen kurzfristigen Absprachen oder Änderungen jeden Teilnehmer schnell erreichen. Eine Anmeldung bei der Mailingliste ist via http://stuttgart.pm.org/ möglich.

Stuttgart.pm ist eine recht heterogene Gruppe. Hier finden sich angestellte Programmierer ebenso wie selbständige Perl-Entwickler, und Autoren von CPAN-Modulen genauso wie Perl-Einsteiger. Auch wer nicht primär mit Perl arbeitet, darf vorbeikommen. Gesprächsthemen gibt es schließlich genug, und Perl ist gar nicht immer die Hauptsache Stuttgart.pm hat daher einen großen „Social Meeting“-Charakter. Fachthemen kommen dennoch nicht zu kurz. Zu den Höhepunkten bei Stuttgart.pm zählen sicherlich die sporadisch stattfindenden Vorträge. So gab beispielsweise Marcus Holland-Moritz zum Einstieg in die Gruppe ein XS-Tutorial und Stas Bekman als Gast ein Tutorial über mod_perl 2. Im Frühjahr 2008 war brian d foy während seiner Deutschlandtour auch Gast bei Stuttgart.pm.

Die Interessen der Stuttgarter Perl Mongers sind weit gestreut. Abgesehen von der auffälligen gemeinsamen Abneigung gegen Windows haben die Perl Mongers aus dem „Ländle“ ein Faible für Testing, mod_perl, PostgreSQL, Web-Entwicklung, sauberen Code, Performance-Optimierung, XS und vieles mehr. Von rund einem Dutzend Stamm-Teilnehmern sind bei einem Treffen meistens gerade mal die Hälfte da – Arbeit, Urlaub, einer der vielen anderen IT-Stammtische in der Region oder anderweitige Verhinderung haben eben auch mal Vorrang.

Weitere Perl-Fans sind natürlich immer willkommen!

Daher: Wer Lust hat teilzunehmen, meldet sich auf der Mailingliste (über die Website erreichbar) an oder findet den aktuellen Termin unter http://stuttgart.pm.org/ Wir treffen uns meist in einem indischen Restaurant in Innenstadt-Nähe, im Sommer auch häufig im Biergarten. Mehr Informationen auf der Webseite und über die Mailingliste.

Alvar Freude

Leserbrief

Sehr geehrter Herr Bäcker,

Unabhängig von unserer sonstigen Korrespondenz habe ich mit Interesse den im Betreff genannten Bericht gelesen.

Dabei sind mir einige Kleinigkeiten aufgefallen, die ich Ihnen in der Folge gerne zur Kenntnis bringen würden. Gegen eine Veröffentlichung in $foo (z.B. als Leserbrief) habe ich nichts einzuwenden.

Sie schreiben u.a. über attributes:
„attributes stellt einige Builtin-Attribute zur Verfügung, wie z.B method oder lvalue."

Das Modul attributes.pm selber stellt keine Builtin-Attribute zur Verfügung, diese werden - wie ja schon der Name verrät - durch den Core selber bereitgestellt. Das läßt sich anhand folgenden Beispiels erkennen:

```
BEGIN {
  $INC{attributes.pm} = 1;
}

my $x;

sub Test:lvalue {
  $x;
}
Test() = 12;
print Test();       # Gibt 12 aus
```

Hier gaukeln wir dem Core vor, dass attributes.pm schon geladen wurde (weil dies sonst geschieht, sobald der Parser eine Attributdeklaration entdeckt; daher ist auch das explizite Laden mit „use attributes" überflüssig). attributes.pm wird also nicht geladen, trotzdem wird das lvalue Attribut richtig verarbeitet.

Das Laden von attributes.pm ist nur notwendig, wenn - wie im Text in Listing 5 gezeigt - eine oder mehrere seiner Funktionen aufgerufen werden sollen; die Builtin-Attribute (je nach Perl-Version und Compileroptionen „lvalue", „locked", „method" und „assertion" für Funktionen/Methoden und „unique" bzw. „shared" für Variable) sind fix in den Core eingebaut. Und natürlich zeigt sich der eigentlich Zweck von attributes.pm beim Einsatz eigener Attribute und schreiben entsprechender Handler, da es hierfür ein Rahmenwerk zur Verfügung stellt, auf das andere Module wie das von Ihnen erwähnte Attribute::Handlers aufsetzen.

Drei kurze Fragen noch zu dem Codebeispiel in Listings 6-8: Abgesehen davon, dass darin in Listing 6 und 7 die abschließende „1;" fehlt, damit der Code geladen werden kann, ist mir unklar:

- Wozu wird in Listing 6 der Typeglobname der Variable „$name" zugewiesen?
- Wo ist die in Listing 6 verwendete „type" Methode bzw. die „level_error" Funktion beschrieben?
- Wo ist die in Listing 8 verwendete „new" Methode von TestShell.pm?

Ein Wort auch noch zu dem Artikel über base.pm: Letzteres ist deswegen so umfangreich, weil es auch die Vererbung von Feldern (die durch Klassendefinitionen z.B. in der Form

```
my Class $scalar;
```

eingerichtet werden) unterstützt. Das erwähnte Modul „parent.pm" tut das nicht. Ich gebe allerdings gerne zu, das man diese Funktionalität (Vererbung von Feldern) wohl in 95% der üblichen Fälle nicht brauchen wird.

Freundliche Grüße aus Wien nach Biebesheim,

Ferry Bolhár-Nordenkampf

Neue Perl-Podcasts & Merkwürdigkeiten

Perlcast.com

Jeff Horwitz

Nach langer Pause gibt es wieder einen Perlcast: Jeff Horwitz wurde über das Apache-Modul mod_parrot interviewt

Merkwürdigkeiten in Perl

In dieser Miniserie sollen einige Merkwürdigkeiten in Perl besprochen werden, auf die man eigentlich nie stößt. Aber wenn man doch mal auf sie trifft, rufen sie meistens nur Verwirrung hervor.

```
use warnings;
my $string = '1 and true';
if( $string == 0 ){
    # mach was
}
```

gibt eine Warnung aus (`Argument "1 and true" isn't numeric in numeric eq (==) at - line 3.`),

```
use warnings;
my $string = '0 but true';
if( $string == 0 ){
    # mach was
}
```

jedoch nicht. Warum?

Perl verwendet in vielen Funktionen die 0 als Rückgabewert wenn etwas nicht erfolgreich war. In manchen Fällen soll die „0“ aber gleichzeitig als wahrer Wert (so dass man `if( funktion() ) {...}` anstatt `if( funktion() == 0 ) {...}` schreiben kann) und als Zahl 0 gebraucht werden. Für diesen Fall gibt es mehrere Lösungen wie ‚0e0‘ oder ‚0.‘. Perl verwendet jedoch den String `0 but true`.

Während ‚0e0‘ kürzer sind und auch „mathematisch“ korrekt, ist der String `0 but true` wirklich aussagekräftig.

Und damit nicht bei Perl-eigenen Funktionen eine Warnung ausgegeben wird, ist dieser Spezialstring von den Warnungen ausgenommen.

Smart-Websolutions

Windolph und Bäcker GbR

Perl-Programmierung

info@smart-websolutions.de

CPAN News VII

Perl::Version

Auch wenn das Modul `Perl::Version` heißt, ist es auch für andere Versions-Nummern zu gebrauchen. Häufig steht man vor dem Problem, dass in einem Modul oder einem Programm Versionsangaben gemacht werden, die mit einem einfachen numerischen Vergleich nicht zu vergleichen sind und Stringvergleiche bringen falsche Ergebnisse.

Mit `Perl::Version` kann man solche Versionsnummern, die wie die Perl-Versionen aufgebaut sind, vergleichen.

```
#!/usr/bin/perl

use strict;
use warnings;
use Perl::Version;

my $v1 = Perl::Version->new(1.5.1);
my $v2 = Perl::Version->new(1.5);
my $v3 = Perl::Version->new(1.5.10);

print $v2 < $v1 ? "$v2 < $v1\n" :
                  "$v2 > $v1\n",
 $v1 < $v3 ? "$v1 < $v3\n" : "$v1 > $v3\n",
 $v2 < $v3 ? "$v2 < $v3\n" : "$v2 > $v3\n";
```

perltugues

Perl ist nicht gleich Perl ;-) Sprachenfreaks können in mehreren Sprachen Perl programmieren.`perltugues` ist eher ein Modul, das in die Acme::-Ecke gehört, aber es ist ganz witzig, wenn man mal in Portugiesisch programmieren möchte.

```
#!/usr/bin/perl

use strict;
use warnings;
use perltugues;

inteiro: i;

para i ( de 1 a 5 a cada 2 ){
    escreva '$foo - Perl-Magazin',
            quebra de linha;
}
```

Time::Simple

Zeitrechnungen sind nicht ganz einfach, mit Time::Simple werden sie aber einfach. Man muss sich um vieles nicht mehr selbst kümmern und die Programme werden lesbarer.

```
#!/usr/bin/perl

use strict;
use warnings;
use Time::Simple;

my $time = Time::Simple->new;
print "Es ist ", $time->hours, " Uhr, ",
      $time->minutes," Minuten und ",
      $time->seconds,"\n";

$time += 60 * 60;

print "In 60 Minuten ist es $time\n";
```

Business::KontoCheck

Wer Business-Anwendungen schreibt und Kontodaten einsammelt, möchte sie ja auch validieren. Mit Business::KontoCheck kann man überprüfen, ob die eingegebene Kontonummer zu der Bankleitzahl passt. Die Validierung mit diesem Modul ist allerdings nur für Konten aus Deutschland und Österreich möglich.

```
#!/usr/bin/perl

use strict;
use warnings;
use Business::KontoCheck qw(kto_check);

if( kto_check( '70070000', '012345500',
  '../.cpan/build
     /Business-KontoCheck-2.6/blz.lut' ) ){
  print "Kontodaten i.O.\n";
}
```

LWP::Online

Manche Module und Programme benötigen eine Internetverbindung, um korrekt zu funktionieren. Mit LWP::Online von Adam Kennedy kann man das jetzt sehr einfach überprüfen. Allerdings wird zurzeit nur auf http-Verbindungen geprüft und Proxies kann man nicht setzen. Aber es ist ein Anfang...

```
#!/usr/bin/perl

use strict;
use warnings;
use LWP::Online qw(online);

print 'Internetverbindung besteht'
    if online();
```

Benchmark::ProgressBar

Manche Benchmarks können ganz schön lange dauern. Dann weiß man manchmal nicht, ob das Programm hängt oder ob der Benchmark wirklich so lange dauert. Mit Benchmark::ProgressBar bekommt man einen Fortschrittsbalken während des Benchmarks angezeigt...

```
#!/usr/bin/perl

use Benchmark::ProgressBar qw(:all);

my @dates;

for( 0 .. 1000 ){
    my $str = (int rand 31) . '-' . ( ( int rand 100 ) + 2000 );
    push @dates, $str;
}

cmpthese(
   100000,
   {
      test1 => sub{ my @array = map{$_->[1]}
                      sort{$b->[0]<=>$a->[0]}
                      map{[join('', reverse map {sprintf '04d', $_}split /-/),$_]}@dates
                     },
      test2 => sub{ my @array = map { /^(....)(..)$/; "$2-$1" }
                                sort { $a <=> $b }
                                map { sprintf('%04d%02d', reverse /^(\d+)-(\d+)$/) } @dates;
},
    }
);
```

Termine

August 2008

04. Treffen Vienna.pm
05. Treffen Frankfurt.pm
07. Treffen Dresden.pm
13.-15. YAPC::Europe in Kopenhagen
18. Treffen Erlangen.pm
21. Treffen Darmstadt.pm
21.-25. OSCON in Portland
23./24. FrOSCon in St. Augustin
26. Treffen Bielefeld.pm

September 2008

01. Treffen Vienna.pm
02. Treffen Frankfurt.pm
04. Treffen Dresden.pm
TWiki - Meetup in Berlin
11./12. Nagios Konferenz
13. 2. Russischer Perl-Workshop
15. Treffen Erlangen.pm
18. Treffen Darmstadt.pm
18./19. Italienischer Perl-Workshop
20. Treffen Bielefeld.pm

Oktober 2008

02. Treffen Dresden.pm
06. Treffen Vienna.pm
07. Treffen Frankfurt.pm
11./12. Pittsburgh Perl-Workshop
16. Treffen Darmstadt.pm
20. Treffen Erlangen.pm
28. Treffen Bielefeld.pm

Hier finden Sie alle Termine rund um Perl.

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

Uhrzeiten und weiterführende Links zu den einzelnen Veranstaltungen finden Sie unter

http://www.perlmongers.de

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

termine@foo-magazin.de

LINKS

http://www.perl-nachrichten.de

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

http://www.perl-community.de

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

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

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

http://www.perl-workshop.de

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

http://www.perl-foundation.org

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

http://www.Perl.org

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

// SEIBERT / MEDIA

BESSERE ATMOSPHÄRE? MEHR FREIRAUM?

Wir suchen erfahrene Perl-Programmierer/innen (Vollzeit)

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

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

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

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

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

Warum Fachleute auf Schulungen gehen sollten

Externe Schulungen sind wie eine Studienfahrt mit Fremden, die am gleichen Thema arbeiten. 5 Tage lang 'mal nicht mit den eigenen Kollegen im immer gleichen Brei schwimmen. Es ist nämlich nicht richtig, alles im Selbststudium lernen zu wollen: Jede(r) von uns hat die Fundamente seines Könnens in Schulen und Universitäten gelernt, und gerade wer im Betrieb stark belastet ist, hat keine Chance, schwierige Themen „nebenbei“ am Arbeitsplatz zu erlernen oder neue Kollegen anzulernen.

Schauen Sie 'mal: www.Linuxhotel.de

Wer sich wirklich intensiv auf eine Arbeit einläßt, will auch Spaß dabei haben! Im Linuxhotel kombinieren wir deshalb ganz offen eine äußerst schöne und luxuriöse Umgebung mit höchst intensiven Schulungen, die oft bis in die späten Abendstunden zwanglos weitergehen. Natürlich freiwillig! Wer will, zieht sich zurück! Und weil unser Luxushotel ganz weitgehend per Selbstbedienung läuft, sind wir gleichzeitig auch noch sehr preiswert.