Ausgabe 3/2013 – www.foo-magazin.de - 6,00 €

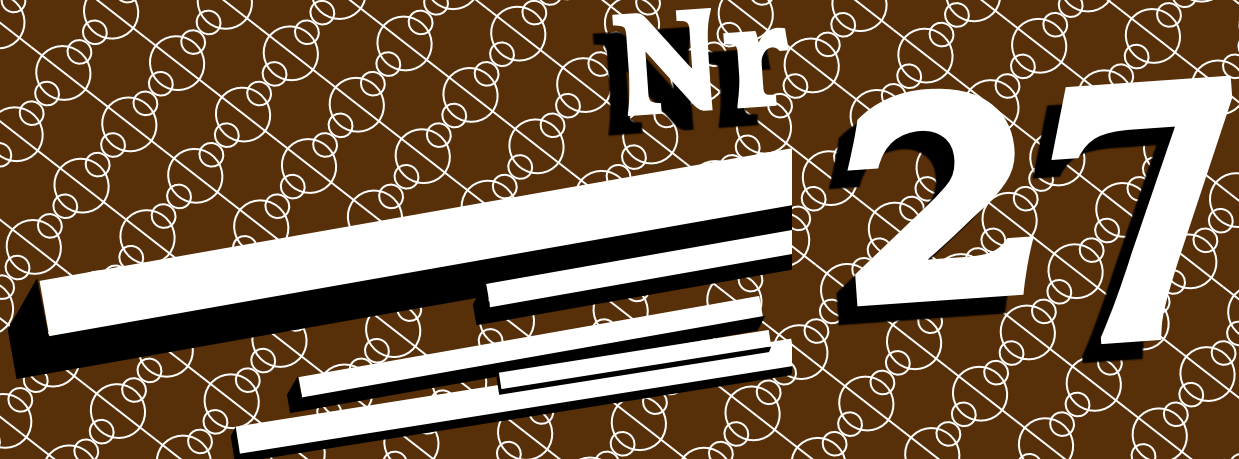
$foo

PERL MAGAZIN

Hannover.pm
Perl Usergroup in Hannover

Debugging
Das gefürchtete „Attempt to free unreference scalar"

VM ansteuern
mit VM::JiffyBox



FrOSCon

Free and Open Source Software Conference

24. + 25. August 2013

LPI-Prüfungen
Social Event
Hüpfburg

Über 100 Fachvorträge
Über 60 Aussteller und Projekte
Kinder- und Jugendtrack

Hochschule Bonn-Rhein-Sieg
Grantham-Allee 20, 53757 Sankt Augustin
www.froscon.de

Hochschule Bonn-Rhein-Sieg
University of Applied Sciences
Fachbereich Informatik

Gold-Sponsoren

tarent AG

doo

Kooperationspartner

DLR Deutsches Zentrum für Luft- und Raumfahrt
German Aerospace Center

LUUGA
Linux/Unix Usergroup Sankt Augustin

gestaltet von Nicole Schmitt, Grafikerin

VORWORT

FrOSCon, Modern Perl und Startups

Was haben Sie Ende August vor? Am 24./25. August findet die achte FrOSCon an der Hochschule Bonn-Rhein-Sieg in St. Augustin statt. Auch Perl ist in diesem Jahr wieder mit dabei und das gleich schlagkräftig: Es gibt wieder einen Perl-Stand und es ist endlich mal wieder etwas über Perl im Haupttrack der Vorträge zu hören. Außerdem haben wir bei dieser FrOSCon erstmals einen Developer-Room für beide Tage. Es gibt also jede Menge auf die Ohren.

Jetzt aber weg von den Veranstaltungshinweisen und hin zu anderen Themen: Im Juni gab es mehrere Blogposts zum Thema *Pre-Modern Perl* vs. *Post-Modern Perl*. Aus beiden Lagern haben sich Leute gemeldet, die ihren „Stil" verteidigen wollen. Häufig dreht es sich um die Verwendung von Moose oder eben nicht. In meinen Augen sollte jede den Stil verwenden, in dem sie sich wohlfühlt. Allerdings sollte man auch die Entwicklungen in der Community bzw. auf CPAN nicht aus den Augen verlieren - übrigens ein toller Nebeneffekt, wenn man Artikel für $foo schreibt: Man beschäftigt sich tiefer gehend mit einer Materie und kümmert sich um neuere Entwicklungen.

Auch mache ich *Modern Perl* nicht an Moose fest. Es ist mehr, für mich ist es das Bestreben les- und wartbaren Code zu schreiben. Natürlich passiert das bei mir häufig mit Moose bzw. dem leichtgewichtigen Geschwisterchen *Moo*.

In den letzten Wochen kam auch das Thema Perl und Startups regelmäßig auf. JT Smith, der WebGUI gestartet hat und mit PlainBlack ein erfolgreiches Unternehmen hat, das hauptsächlich Perl verwendet, hat in einem Blogpost dazu aufgerufen, einfach mal etwas mit Perl zu bauen. Das kann man dann verwenden, um ein Unternehmen zu gründen und damit Geld zu verdienen.

Questhub.io ist im Prinzip nach diesem Muster entstanden. Anfänglich war es die Anwendung hinter „PlayPerl" und dann hat Vyacheslav Matyukhin sich damit selbstständig gemacht. Ich habe zwar keine Ahnung ob er damit (schon) erfolgrich ist, aber das Vorgehen finde ich sehr gut und die Idee interessant.

Auch Curtis „Ovid" Poe widmet sich dem Thema. In seinem Blog auf http://blogs.perl.org interviewt er Gründer von Perl-Startups. In den ersten beiden Folgen hat er mit JT Smith über Lacuna Expanse und mit Alex Balhatchet von Lokku/Nestoria gesprochen.

Die ersten Reaktionen auf JTs initialen Blogpost gibt es schon. So hat Dave Cross sein „Political Web" vorgestellt. Auf der Webseite sind alle auffindbaren Informationen über Parlamentsmitglieder in Groß-Britannien zu finden. Und Jesse Shy hat Classmith.com vorgestellt.

Wer sich über den Start eines Startups informieren möchte, kann das z.B. über http://www.gruenderszene.de/ oder über JT Smiths neuen Blog unter http://plainblackguy.tumblr.com/ tun. Es wäre schön, wenn wir in Zukunft auch über deutsche Startups in der Perl-Szene hören und berichten könnten.

Viel Spaß beim Lesen der 27. Ausgabe des Perl-Magazins,

Renée Bäcker

Die Codebeispiele können mit dem Code

dbs32gs

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

Alle weiterführenden Links werden auf del.icio.us gesammelt. Für diese Ausgabe: http://del.icio.us/foo_magazin/issue27.

IMPRESSUM

Herausgeber:	Perl-Services.de Renée Bäcker Bergfeldstr. 23 D - 64560 Riedstadt
Redaktion:	Renée Bäcker
Anzeigen:	Renée Bäcker
Layout:	//SEIBERT/MEDIA
Auflage:	500 Exemplare
Druck:	print24 (Marke der unitedprint.com Deutschland GmbH) Friedrich-List-Straße 3 D - 01445 Radebeul
ISSN Print:	1864-7537
ISSN Online:	1864-7545
Feedback:	feedback@perl-magazin.de

INHALTSVERZEICHNIS

ALLGEMEINES

PERL

MODULE

ANWENDUNGEN

NEWS

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

Renée Bäcker

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

Herbert Breunung

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

Boris Däppen

Boris Däppen lernte Perl im Umfeld der Finanzdienstleister in Zürich kennen und vertiefte seine Kenntnisse der Sprache später bei perl-services.de. Er hat kürzlich als erster Absolvent den neuen Master „Technik und Philosophie" an der TU Darmstadt abgeschlossen und arbeitet nun als Freelancer in der Schweiz.

Wolfgang Kinkeldei

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

Sören Kornetzki

Sören ist Programmierer aus Hannover (ursprünglich Cuxhaven). 2008 berufsbedingt nach Hannover umgezogen setzt er neben ANSI-C hauptsächlich auf Perl und arbeitet für die Delticom AG. Unter dem Namen/burnersk/ ist er im IRC und auf CPAN unterwegs. Ehrenamtlich engangiert er sich für Perl, indem er den 16. deutschen Perl-Workshop in Hannover, sowie die dortige Monger-Gruppe mitorganisiert.

Ralf Peine

Jahrgang 1965
Dipl. Mathematik 1991
Software-/Tool-/CM-Architekt
Renesas Electronics Europe GmbH

Ich programmiere seit ca. 20 Jahren mit wachsendem Vergügen mit Perl, beherrsche aber auch viele andere Sprachen wie C/C++/C#, VB, PHP, Lisp, Java-Script, HTML, XML, Von Großrechner-Software bis zur Mikrocontroller-Digitaltechnik (wo ich mich jetzt bewege), habe ich schon so manches entworfen und entwickelt.

Auf meiner privaten Web-Domain könnt ihr euch PORF herunterladen:

- http://www.jupiter-programs.de/
- http://www.jupiter-programs.de/prj_public/porf/index.htm

Anmerkungen zu PORF könnt ihr in meinen Blog schreiben unter:
http://blogs.perl.org/users/jpr65/2013/05/perl-open-report-framework-0901-released.html

Meine aktuellen Favoriten in der Softwareentwicklungsmethodik sind

- Testdriven Development (nie mehr anders!)
- Operation und Integration
  (http://blog.ralfw.de/2013/04/software-fraktal-funktionale.html)

„So einfach wie möglich, aber nicht einfacher." (Albert Einstein)

Wolfgang Schemmel

Perl->1996. Webmaster, Web-Entwickler bei heise.de, oft in virtuose Quadrophonie aus Perl, Vim, Apache und Opera vertieft. Bayer. MotoGP-Fan. Perl-Patriot. Moose-ketier. Dev-Operateur. Apache-Versteher. Tastatur-Jockey. Logfile-Zauberer. HTTP-Ninja. Daten-Banker. Bug-Beseitiger. Sprach-Sezierer. Kommandozeilen-Kapitän. Pipe-Partisan. JSON-Jonglierer. Shell-Schamane. Vim-Veteran. Git-Guerilla. SQL-Sadist. Debian-Desperado. Perl-Patriot (Wiederholung, weil wichtig . Alliterations-Adept.

Steffen Müller

Das gefürchtete „Attempt to free unreference scalar"

Jedes mal wenn ich einen Fehler wie „Attempt to free unreferenced scalar: SV 0xDEADBEEF" sehe, rutscht mir das Herz in die Hose. Ich weiß, dass ich mit einer ausschweifenden Debugging-Sitzung dran bin. Das Debuggen von Speichermanagementproblem ist immer mühevoll. Das ist in Perl so und in jeder anderen Sprache. *valgrind* und ähnliche Tools können ein Geschenk des Himmels sein, aber auch sie kommen mit einigen Arten von Problemen nicht klar.

Das Problem verstehen

In den Perl-Diagnosemeldungen ist der folgende hilfreiche Absatz über die Warnung zu finden:

Perl hat versucht den Referenzzähler eines Skalars zu dekrementieren um zu prüfen ob er auf 0 gehen würde und hat dabei herausgefunden dass der Referenzzähler bereits 0 ist. Der Skalar hätte schon freigegeben werden sollen und wahrscheinlich wurde er schon freigegeben. Das könnte darauf hindeuten dass SvREFCNT_dec() zu häufig aufgerufen wurde oder das SvREFCNT_inc() zu selten aufgerufen wurde. Oder, dass das SV zu einem Zeitpunkt „mortalized" wurde als es noch nicht hätte passieren dürfen. Oder, dass der Speicher korrupt ist.

Lasst es uns etwas auseinander nehmen, da es einige Implementierungsdetails von Perl anspricht: Die aktuelle Implementierung von Perl5 verwendet ein referenzzählerbasiertes Speicherverwaltungsschema. Jeder Basiswert (ein Skalar, technisch gesehen ein Pointer auf ein SV-struct) hat einen Slot, der mitzählt, wie oft dieser Wert von etwas anderem referenziert wird. Wenn du eine neue Referenz auf einen SV erzeugst, erhöhst du diesen sogenannten *refcount*. Wenn du die Referenz auf den SV freigibst, musst du diesen Zähler verringern. Sobald dieser *refcount* 0 erreicht, gibt Perl den Speicher, der mit dem SV verbunden ist, frei und verringert dabei den refcount aller anderen SVs auf die der freigegebene SV eine Referenz hält. `SvREFCNT_inc()` und `SvREFCNT_dec()` sind die Perl-(C)-Makros, die genau das machen.

Wenn Du `SvREFCNT_inc()` einmal zu oft aufrufst oder `SvREFCNT_dec()` einmal zu wenig, dann „leaken" der SV und alles worauf dieser SV eine Referenz hält, weil sie nie zerstört werden - bis zur globalen Zerstörungsphase der `perl`-VM. Wenn Du das Gegenteil machst (zu viele `SvREFCNT_dec()`- oder zu wenige `SvREFCNT_inc()`-Aufrufe), wird der refcount des SV zu früh auf 0 gesetzt und es wird freigegeben, obwohl es immer noch von anderen Datenstrukturen referenziert wird. Leider sind das genau die, die in glückseliger Unwissenheit des bevorstehenden Fehlers durch einen ungültigen Speicherzugriff übrigbleiben.

Die oben erwähnte Warnung wird von Perl ausgegeben, wenn der Referenzzähler eines SV heruntergezählt wird und der Zähler bereits 0 ist. Beginnend mit dem Referenzzähler bei 0 bedeutet das, dass es nicht länger ein gültiger SV ist, der von Perl benutzt wird. Da das Speichersegment, in dem der SV gespeichert war, nicht länger zur Speicherung des originalen SV benutzt wird (siehe weiter unten für mehr Infos), könnte der Speicher mittlerweile für einen anderen SV benutzt worden sein. Sollte das so sein, wirst du die Warnung über den Skalar mit der schlechten Speicherverwaltung nicht sehen. Perl kennt Deine Absichten nicht und zählt fröhlich den Referenzzähler des neuen Speicherbewohners herunter. Das bedeutet schließlich, dass der Referenzzähler des neuen SV zu früh auf 0 gesetzt wird. Das wiederholt sich bis du es schaffst, den Speicher zu korrumpieren und davor zu warnen, bevor Perl eine Chance hat den Speicher wiederzuverwenden. Viel Spaß!

Aufmerksame C-Programmierer werden nun das Speicher-Debugging-Tool des Tages auf den Tisch bringen (mein Favorit ist und bleibt *valgrind/memcheck*), das mit dieser Art von Problemen durch Identifikation von ungültigen Zugriffen auf freigegebenen Speicher sehr effektiv umgeht. Ich wünschte es wäre so einfach! Das Schema ist doppelt falsch: Zum einen hat bei der oben gezeigten *Action at a distance* perfekt valider Code den ungültigen Speicherzugriff. Aber noch wichtiger ist: Das macht es Perls interner Speicherwaltung sehr wahrscheinlich, dass das Problem auftritt. Perl benutzt *slab allocation* um zu verhindern, wegen jedem einzelnen SV, das es erzeugt, über das Betriebssystem zu gehen, da viele *malloc*-Implementierungen der Betriebssysteme mangelhaft sind. Die Teile der SV-Struktur, die den Referenzzähler halten sind in so einem *slab* (in den Perl-Quellen als *arena* bezeichnet) zugewiesen, das typischerweise der Größe einer Speicherseite entspricht und so viele Elemente hält, wie in diese Größe passen. Perl verwendet eine Liste von unbenutzten Elementen in dem *slab* um effektiv SVs „zuweisen" und „freigeben" zu können. Das ist gut für die Performanz und um eine Fragmentierung des Speichers zu verhindern. Aber für das Debuggen von solchen Speicherproblemen verstärkt das die *Action at a Distance*-Problematik durch das häufige Wiederverwenden der SV *slab*s.

Wenn du Opfer der Warnung wirst, bringt eine Suche im Internet eine ganze Anzahl von Fällen zu Tage, in denen verzweifelte Leidensgenossen nach Hilfe von Experten fragen. Leider gibt es nicht das eine wahre Rezept zum Debuggen, das zu einer Lösung in allen möglichen Fällen führt und die wenigen spezifischen Hinweise, die es gibt, erfordern in der Regel, dass man eine spezielle Kopie von Perl für das Debugging baut.

Dein Perl aufrüsten

Es gibt eine Reihe von Optionen für `Configure`, die unterschiedlich gut in der Perl-Dokumentation abgehandelt werden, die dabei helfen eine Kopie von Perl zu bauen, das einige der oben genannten *Action at a distance*-Probleme vermeidet. Das Basis-Rezept (für *nix) zum Bauen deines eigenen Perls ist wie folgt (angenommen, du bist in einem Checkout des Perl-git-Repositories oder einem entpackten Release-Archiv):

```
$ sh Configure -des -Dusedevel
$ make
$ make test
```

Die `-d -e -s`-Optionen bedeuten im Prinzip: „Frage mich nicht irgendwelche Fragen und verwende vernünftige Standardeinstellungen für alles!" Die `-Dusedevel`-Option bedeutet einfach nur, dass `Configure` sich nicht darüber beschweren soll, dass du eine Entwicklerversion von Perl baust, wenn du aus einem *git clone* heraus arbeitest. Es ist im Grunde die „Ja, ich will es wirklich"-Option, die verhindern soll, dass Leute eine nicht-veröffentlichte Version von Perl in Produktivsystemen ausrollen. Um Dein Perl auf einer Multi-Core-Maschine schneller zu bauen und zu testen, kannst du einige `-j`-Magie verwenden:

```
$ sh Configure -des -Dusedevel
$ TEST_JOBS=5 make -j5 test
```

Auf diese Weise wird mit fünf parallelen Jobs kompiliert und getestet. Wenn du das neue Perl an einen bestimmten Ort installieren willst, dann füge noch `-Dprefix=/home/you/mydebugperl` hinzu und rufe `make install` auf. Auf dem Weg zu einem Perl mit besseren Debugging-Möglichkeiten, sollen für den Anfang Debugging-Symbole in der Ausgabe eingebunden und möglicherweise die C-Compiler-Optimierungen ausgeschaltet werden. Dann füge `-Doptimize="-g -O0"` zum `Configure`-Aufruf hinzu. Das ist praktisch zum Aufspüren von Problemen im Perl-Code, wenn du auf die *valgrind*-Ausgabe schaust. Als nächstes wird ein Perl gebaut, bei dem seine eigenen Debugging-Funktionalitäten aktiviert sind: Füge `-DDEBUGGING` hinzu. Fassen wir alles bisherige zusammen, bekommen wir:

```
$ sh Configure -des
 -Dprefix=/home/you/mydebugperl -Dusedevel \
  -Doptimize="-g -O0" -DDEBUGGING
```

Alles, was du bisher erreicht hast, ist natürlich eine Kopie von Perl, die massiv langsamer ist als dein produktives Perl (wahrscheinlich eine ganze Größenordnung langsamer) und die dir noch nicht wirklich dabei helfen wird, dein Referenzzähler-Problem zu debuggen. So ein Perl zu haben, ist das typische Sprungbrett für das Debuggen des Perl-Kerns und bis jetzt sind die Schritte an anderer Stelle sehr gut dokumentiert.

Das *perlhacktips*-Dokument erklärt eine Reihe von komplizierteren Optionen für das Speicherdebugging. Der Ab-

schnitt *PERL_DESTRUCT_LEVEL* ist von besonderem Interese. Es stellt sicht heraus, dass Perl sich standardmäßig nicht um das Säubern der Speicher-*slabs* kümmert, wenn es fertig ist. Es lässt das im Allgemeinen das Betriebssystem erledigen (ich glaube, weil es weniger Overhead hat). Das Setzen der Umgebungsvariablen `PERL_DESTRUCT_LEVEL` während der Programmausführung lässt Perl pedantischer werden. Es ist wichtig, solche Sachen für Tools wie *valgrind* von Anfang an ersichtlich zu machen:

```
$ PERL_DESTRUCT_LEVEL=
 2 perl your_buggy_program.pl
```

Wenn du das Gegenteil des „Attempt to free..."-Problem hattest, also SVs mit zu hohem Referenzzähler, dann bekommst du mit diesem Setup Benachrichtigungen über leckende Skalare. Der nächste Schritt zum Grund des Problems schließt die `Configure`-Optionen `-DDEBUG_LEAKING_SCALARS`, `-DDEBUG_LEAKING_SCALARS_FORK_DUMP` und `-DDEBUG_LEAKING_SCALARS_ABORT` ein. Diese sind größtenteils in *perlhacktips* dokumentiert.

Um das vermutete Referenzzähler-Problem einfacher aufzufinden, können wir eine Option angeben, die für das `Purify`-Tool gedacht ist: `-Accflags=-DPURIFY` (das heißt: Füge *-DPURIFY* zu den C-Compiler-Optionen hinzu. Mit dieser C-Definition wird Perl keine *slabs* für das Zuweisen von SVs verwenden, was deine Chancen erhöhen sollte, seltsames Verhalten aufzuspüren. Zusätzlich können wir Perl anweisen, freigegebene Speicherbereiche mit einem bekannten Muster (`0xEF`) zu überschreiben. Das verhindert, dass Fehler durch das Wiederverwenden von Speicher, der vorher für SVs benutzt wurde, verschleiert werden. Ähnlich wie die Purify-Option ist das auch eine C-Compiler-Definition. So bekommen wir: `-Accflags="-DPURIFY -DPERL_POISON"`.

Nur um alle Tools zusammenzubringen, zeige ich hier, wie ich schließlich mein Perl gebaut habe. Die Einstellungen `-Dcc` und `-Dld` erlauben es mir, *ccache* mit *gcc* für schnelleres wiederholtes Kompilieren zu verwenden:

```
$ sh Configure -Doptimize="-g -Wall
     -Wextra -O2" -DDEBUGGING -Dusedevel  \
 -Dprefix=/home/you/mydebugperl
     -Dcc=ccache\\ gcc\\ -g -Dld=gcc \
 -Uusethreads -de -DPERL_TRACK_MEMPOOL
     -DDEBUG_LEAKING_SCALARS_FORK_DUMP \
 -DDEBUG_LEAKING_SCALARS -Accflags=
    "-DPURIFY -DPERL_POISON" \
 -DDEBUG_LEAKING_SCALARS_ABORT
```

Mein unreferenzierter Skalar

Ein Perl, das mit allen oben genannten Debugging-Tools ausgestattet ist, hat meine Debugging-Bemühungen in vielen Fällen viel einfacher gemacht. In der Regel ist ein Teil der oben gezeigten Möglichkeiten ausreichend. Leider hat es mir in diesem Fall nur das sorgfältige Durchgehen meines Codes und Dumpen der Adressen von vielen SVs, um den einen zu finden, der zu früh freigegeben wurde, erlaubt, den einen einzelnen Befehl zu finden, der mir den Tag verdorben hat:

```
SvREFCNT_dec(*fetched_sv);
```

Der Code hat einen Hash-Zugriff verwendet, der das Element erzeugt, wenn es beim Zugriff nicht existiert (`HV_FETCH_LVALUE|HV_FETCH_JUST_SV`-Modus von *hv_common*), aber fälschlicherweise davon ausgegangen ist, dass ein Inkrementieren des Referenzzählers Teil der Operation ist. Unnötig zu sagen, dass ich den Fehler beseitigt habe und dann einen schwierigen Siegestanz aufgeführt habe.

Der eigentliche Fehler wurde nur durch viele, viele Millionen Tests zu Tage gebracht, die gegen unsere Perl/XS-Implementierung der Bibliothek für Serialisierung und Deserialisierug zum Schutz gegen Attacken laufen. Aber das ist ein anderes Thema.

Boris Düppen

VM ansteuern mit VM::JiffyBox

Die Problemstellung

Mit dem Produkt *JiffyBox* der Firma *Domainfactory* lassen sich virtuelle Maschinen sehr flexibel einsetzen. Sie können quasi *on the fly* erstellt und verworfen werden. Innert Minutenfrist steht eine neue Maschine bereit. Wahlweise auch mit dem Backup einer anderen Maschine aufgespielt. Dies ist weiter nichts Spezielles und gehört zum Standardangebot von aktuellen Anbietern. Demnach soll das hier auch keine versteckte Werbung darstellen.

Die große Flexibilität der virtuellen Maschinen eröffnet völlig neue Szenarien und Möglichkeiten. Für alle erdenklichen Zwecke lässt sich kurz eine Maschine mit einem spezifischen Abbild erstellen, etwas damit machen, um die Maschine dann am Schluss gegebenenfalls wieder zu verwerfen und nur die Ergebnisse bzw. Erkenntnisse zu behalten. Dies kann z.B. im Umfeld von *Continuous Integration* zur Anwendung kommen. Um die Möglichkeiten der Virtualität aber wirklich auszuschöpfen, genügt das Webinterface längst nicht mehr. Es ist einfach zu mühsam, sich von Hand durch die Weboberfläche zu klicken um neue Maschinen zu erstellen. Was es braucht um wirklich von der flexiblen Handhabbarkeit der Maschinen Gebrauch zu machen, ist die Möglichkeit zur Automation, beispielsweise über eine API.

JiffyBox bietet eine REST-Schnittstelle an, welche genau diese Anforderungen erfüllen soll. Soweit es der Autor abschätzen kann, bedient sich auch das Webfrontend des Anbieters selbst dieser Schnittstelle. Das heißt, dass sich dasjenige, was man klicken kann, auch automatisieren lässt. Die REST-Schnittstelle geht klassisch über HTTP und ist in der Dokumentation des Anbieters vorbildlich beschrieben. Die Dokumentation kann von angemeldeten Nutzern vom Webfrontend als PDF heruntergeladen werden.

Die REST-Schnittstelle ist eine feine Sache. Sie lässt sich vergleichsweise einfach ansprechen, da viele Clients zur Auswahl stehen, Beispielsweise `curl` für die Kommandozeile oder einfach einer der üblichen Webbrowser. Es bedarf aber doch einiges Aufwandes, um darüber komplexere Abläufe umzusetzen. Für jeden Aufruf muss aus verschiedenen Informationen eine URL zusammengesetzt werden. Auch eine Fehlerbehandlung sollte nicht fehlen. Zudem muss jeweils das Resultat des Aufrufs - welches als JSON vorliegt - in für die Programmiersprache der Wahl spezifische Variablen-Strukturen übertragen werden. Dies alles kann schnell in unlesbarem, repetitivem, schlecht wart- und erweiterbarem Code enden. Hier bietet sich die Implementation einer Schnittstelle an, welche die repetitiven Teile übernimmt und die API dem Entwickler so präsentiert wie er es gewohnt ist. Ein Perl-Modul, welches dies für *JiffyBox* anbietet, fehlt bisher, ganz im Gegensatz z.B. zu *Amazon EC2*. *Rex::Commands::Cloud* bietet zwar eine Schnittstelle zu *JiffyBox*, diese ist aber sehr *Rex*-spezifisch. So ist die Idee zum Modul *VM::JiffyBox* geboren.

Perl-Modul als Projekt für Auszubildende

Ein generisches Modul zu schreiben, welches nicht nur die eigenen Bedürfnisse, sondern auch diejenigen der CPAN-Gemeinschaft berücksichtigen soll, bedeutet zumindest kurzfristig einen Mehraufwand und damit ein Mehr an Kosten. Längerfristig ist dies nicht zwingend der Fall, da es einiges zur Wartbarkeit des eigenen Codes beiträgt, wenn man ihn nach CPAN-Standards programmiert. Dennoch kommt es häufig genug vor, dass unter dem Druck des geschäftlichen Alltags die Zeit für ein gutes Modul fehlt. Die reine Lehre fin-

det nicht immer den Weg in die Praxis. Es stellt sich also die Frage, wie ein solches Modul, zum Mehrwert aller, gestemmt werden kann.

Eine erstaunlich einfache Antwort ist, den Auszubildenden im IT-Betrieb für das Projekt zu motivieren. Nur allzu oft werden gerade die Auszubildenden im täglichen Betrieb vergessen oder werden mit eher langweiligen Aufgaben abgespeist. Wenn es der Erscheinungstermin des Moduls hergibt, kann die Beteiligung eines Auszubildenden zur Entwicklung eines CPAN-Moduls sehr viel beitragen. Seine Lohnkosten sind geringer, was die Finanzierung eines solchen Moduls erst mal begünstigt. Natürlich wird die Entwicklung länger dauern und es ist auch Unterstützung erforderlich. Gerade dieser längere Prozess, welcher duch Gespräche begleitet wird, kommt aber auch dem Modul zugute. Schließlich lohnt sich für die Firma dieser Einsatz auch, da der Auszubildende alles, was er an diesem CPAN-Modul gelernt hat, wieder komplett in seine Tätigkeit im Lehrbetrieb mit einbringt. Vom Mehrwert für die Perl- bzw. CPAN-Gemeinschaft insgesamt ganz zu schweigen. Was gibt es Besseres, als jungen Programmierern die aktive Beteiligung am CPAN zu lehren? Das Projekt konnte - in diesem Rahmen - dank der Firma *plusW* von $foo-Autor Rolf Schaufelberger, zusammen mit dem Auszubildenden des Unternehmens und dem Verfasser dieses Artikels umgesetzt werden.

Das Design

Es gibt im Kontext von *JiffyBox* grundsätzlich zwei unterschiedliche Ziele für die API. Alle Anfragen sind entweder direkt an den Hypervisor gerichtet oder sie richten sich an eine der virtuellen Maschinen, welche durch eben diesen Hypervisor behütet werden. Es bietet sich an, dieses Markmal als Konzept für eine objekt-orientierte Code-Gestaltung zu nehmen. Das sieht dann ungefähr so aus, dass man das Objekt Hypervisor fragt, was er für virtuelle Maschinen beherbergt und wie man diese erreichen kann. Der Hypervisor fungiert dann als *Factory* (nach GoF-Pattern) und stellt auf Anfrage ein Objekt zur gewünschten Maschine bereit.

Kennt man diese Aufteilung, ist das Lesen der Dokumentation des hier entwickelten Moduls um einiges einfacher. Wenn man eine Frage an den Hypervisor hat, dann schaut man unter `VM::JiffyBox` nach. Will man hingegen etwas über eine spezifische Maschine wissen, so schaut man sich die Methoden unter *VM::JiffyBox::Box* an.

Praktisch sieht die Umwandlung der REST-URL zum OO-Modell so aus, dass die URL gesplittet wird, und die Teile jeweils an die Objekte übergeben werden, welche die Verantwortung und die Hoheit der Information haben. Das Abfragen einer Maschine betreffend seiner Informationen geschieht beispielsweise so, wenn man (z.B. mit `curl`) direkt über die REST-Schnittstelle geht:

```
curl https://api.jiffybox.de/<auth_token>/
 v1.0/jiffyBoxes/<box_id>
```

Diese URI wird nun wie folgt aufgeteilt:

```
Hypervisor ->
  https://api.jiffybox.de/<auth_token>/v1.0/
VM          -> jiffyBoxes/<box_id>
```

Die URL der API, der Token zur Autorisierung und die Version der Schnittstelle gehören zum Hypervisor. Spezifische Informationen, wie z.B. die ID der Maschine gehören den jeweiligen Maschinen.

Wenn im Objekt-orientierten Modell direkt auf dem VM-Objekt operiert wird, muss jede VM wissen, woher, aus welcher Fabrik, von welchem Hypervisor sie kommt, da diese Information für das Zusammensetzen der URL vonnöten ist. Dies wird dadurch bewerkstelligt, dass der Hypervisor sich beim Erstellen des VM-Objektes jeweils dort einträgt (Subscriber-Pattern der GoF). Im fertigen Modul sieht das dann so aus:

```
# Abfrage von Informationen auf dem
# Hypervisor (API)
my $vm_id = $hypervisor->
  get_id_from_name('Produktion5');

# Kontruktion des VM-Objekts (Factory)
my $vm = $hypervisor->get_vm($vm_id);

# Abfrage der VM nach Informationen (API)
my $info = $vm->get_details();
```

In allen Fällen kann die URL jeweils komplett zusammengesetzt werden, da die beteiligten Objekte ihre Informationen mit einbringen.

Ein paar Beispiele

Nach den Ausführungen zum Design ist klar: Als erstes wird immer ein Objekt des Hypervisors benötigt. Diesem geben wir bei der Erzeugung die Informationen mit, über welche er die Hoheit hat: den Autorisierungs-Token, welcher im Admin-Interface des Anbieters generiert werden kann.

```
my $hypervisor = VM::JiffyBox->
                  new(token => $auth_token);
```

Wir können den Hypervisor nun über Details ausfragen:

```
my $info_hashref =
   $hypervisor->get_details();
```

Die Informationen des Hypervisors (hier in `$info_hashref`) können z.B. so aussehen:

```
$VAR1 = {
  'messages' => [],
  'result'   => {
      # Information über alle VMs
  },
};
```

Wie zu sehen ist, gibt die Abfrage nicht eigentlich Informationen über den Hypervisor zurück, sondern vielmehr einfach alle Informationen über alle beherbergten Maschinen.

Ein Objekt einer spezifischen VM lässt sich auf zwei verschiedene Arten erzeugen. Entweder man erzeugt eine Repräsentation einer bereits existierenden VM, indem man z.B. die ID der Maschine angibt. Dies setzt keinen eigentlichen API-Aufruf ab, sondern erzeugt lediglich die nötigen Objekte für spätere Aufrufe (bzw. das spätere Zusammensetzen der URL).

```
my $vm = $hypervisor->get_vm($vm_id);
```

Oder man erstellt eine neue Maschine, was dann einen asynchronen API-Aufruf nach sich zieht (der Server muss informiert werden, dass eine neue Maschine erstellt werden soll). Die neue Maschine wird wahlweise auf Basis einer Distribution oder eines Backups erzeugt. Die Angaben zur Preiskategorie (`planid`) und dem zu verwendenden Systemabbild (`backupid` oder `distribution`) müssen hierfür der API-Dokumentation des Anbieters entnommen werden oder vom Hypervisor erfragt werden.

```
my $vm  = $hypervisor->create_vm(
            name     => 'Produktion3',
            planid   => $preis_kat,
            backupid => $install_img,
          );
```

Hat man erst ein Objekt einer virtuellen Maschine, so kann man darauf operieren. Sie lässt sich z.B. starten:

```
$vm->start();
```

Oder gibt ihre Informationen preis:

```
my $info = $vm->get_details();
```

Die Informationen einer Maschine (hier in `$info`) können z.B. so aussehen, wie in Listing 1 dargestellt.

Aus dieser Hash-Struktur lassen sich spezifische Informationen direkt abfragen. Die IP-Adresse wird hier beispielsweise so adressiert:

```
$info->{result}->{ips}->{public}->[0];
```

Ein spezifischer Wert, wie z.B. eben die IP-Adresse, lässt sich so auch direkt vom Aufruf der Methode her abfragen. Dies erlaubt kurzen prägnanten Code.

```
my $ip_addr = $vm->get_details->{result}
                              ->{ips}
                              ->{public}->[0];
```

Caching und Fehlerbehandlung

Um mit wenig Aufwand sparsame Requests schreiben zu können wurde ein einfacher Cache geschaffen. Alle Aufrufe welche einen API-Request durchführen speichern das Ergebnis auch in einem internen Cache, statt es nur zurückzugeben. Auf diese Weise lässt sich weiterhin auf die Daten zugreifen, auch wenn beim Aufruf nicht gleich alles ausgewertet oder gespeichert wurde. Dies erspart einem das Anlegen von Wegwerf-Variablen, nur um den Rückgabewert zwischenzuspeichern.

Der wichtigste Cache heisst `last` und beinhaltet jeweils die Daten vom letzten API-Zugriff. Zusätzlich gibt es noch spezifische Caches. Der `backup_cache` speichert Beispielsweise das Ergebnis der letzten Backup-Abfrage und der `details_cache` das Ergebnis des letzten Aufrufes von `get_details()`. Folgende drei Aufrufe setzen so zusammen nur eine einzige Abfrage an den Server ab:

```
my $state =
   $box->get_details->{result}->{status};
my $plan  =
   $box->last ->{result}->{plan}->{id};
my $ip    =
   $box->last ->{result}->{ips}->{public}
      ->[0];
```

Achtung: Hier droht eine Programmierer-Falle. Späteres einfügen von API-Calls, z.B. bei einem Code-Update, verändert u.U. den Inhalt von `last`. Dieses Feature sollte also besser im gleichen Block bzw. Absatz Anwendung finden und nicht quer über den ganzen Code verteilt werden.

Fehler können an verschiedensten Stellen auftreten. So kann z.B. der HTTP-Request schief gehen, oder der API-Server meldet ein Problem in JSON verpackt. Um es für den Programmierer einheitlich zu machen und ihm den ständigen Gebrauch von `eval` zu ersparen gilt die Regel, dass jede Methode bei einem Fehler den Wert `0` zurück geben soll. Auf diese Weise kann man solche Sachen machen:

```
my $id =
   $hypervisor->get_id_from_name($name);
die „No id found for $name" unless($id);

$vm->start() || die „Problem beim Start";
```

```
$info = {
  'messages' => [],
  'result' => {
    'recoverymodeActive' => bless( do{\(my $o = 0)}, 'JSON::PP::Boolean' ),
    'runningSince' => 1372774763,
    'status' => 'READY',
    'name' => 'Produktion1',
    'activeProfile' => {
      'rootdiskMode' => 'ro',
      'runlevel' => 'default',
      'status' => 'READY',
      'name' => 'Standard',
      'created' => 1372773775,
      'disks' => {
        'xvda' => {
          'created' => 1372773758,
          'status' => 'READY',
          'filesystem' => 'ext4',
          'name' => 'Debian Wheezy (7.0)',
          'sizeInMB' => 76288,
          'distribution' => 'debian_wheezy_32bit'
        },
        'xvdb' => {
          'created' => 1372773757,
          'status' => 'READY',
          'filesystem' => 'swap',
          'name' => 'Swap',
          'sizeInMB' => 512
        }
      },
      'kernel' => 'xen-pvops',
      'rootdisk' => '/dev/xvda'
    },
    'running' => bless( do{\(my $o = 1)}, 'JSON::PP::Boolean' ),
    'host' => 'vmhost-2-1-10-11',
    'created' => 1372773755,
    'ips' => {
      'private' => [ '198.51.100.0' ],
      'public' => [ '203.0.113.0' ]
    },
    'metadata' => {},
    'plan' => {
      'pricePerHour' => '0.02',
      'name' => 'CloudLevel 1',
      'diskSizeInMB' => 76800,
      'id' => 20,
      'cpus' => 3,
      'pricePerHourFrozen' => '0.005',
      'ramInMB' => 2048
    },
    'id' => 12345,
    'isBeingCopied' => $VAR1->{'result'}{'recoverymodeActive'},
    'manualBackupRunning' => $VAR1->{'result'}{'recoverymodeActive'}
  }
};
```

Listing 1

```
# Die folgenden Argumente müssen
# bekannt sein
my $auth_token = $ARGV[0];
my $box_name   = $ARGV[1];
my $clone_name = $ARGV[2];

# Hypervisor vorbereiten
my $jiffy = VM::JiffyBox->new(token => $auth_token);

# Den Server-Namen des Servers von welchem eine Kopie
# erstellt werden soll, in seine ID konvertieren
my $master_box_id = $jiffy->get_id_from_name($box_name);

# Objekt für den zu kopierenden Server erzeugen
my $master_box = $jiffy->get_vm($master_box_id);

# Informationen für das Erstellen des Klons holen
my $backup_id  = $master_box->get_backups->{result}->{daily}->{id};
my $plan_id    = $master_box->get_details->{result}->{plan}->{id};

# Klon erstellen (asynchron)
my $clone_box  = $jiffy->create_vm( name     => $clone_name,
                                    planid   => $plan_id,
                                    backupid => $backup_id,
                                  );

# Abbrechen, wenn beim Klonen etwas schief gegangen ist
unless ($clone_box) {
    # FAIL
    die $jiffy->last->{messages}->[0]->{message};
}

# Warten bis der Klon tatsächlich erzeugt wurde
do {
    say „waiting for clone to get READY";
    sleep 15;
} while (not $clone_box->get_details->{result}->{status} eq 'READY');

# Starten des Klons
$clone_box->start();

# Warten bis der Klon gestartet ist
do {
    say „waiting for clone to be 'running'";
    sleep 15;
}
while (not $clone->get_details->{result}->{running} eq 'true');

# IP-Adresse des Klons holen
# ohne erneuten API-Aufruf
my $ip = $clone->details_cache->{result}->{ips}->{public}->[0];

##########################################
# IRGENDWELCHE DINGE MIT DEM KLON MACHEN  #
# ZUM BEISPIEL MIT SSH AUF DIE IP         #
##########################################

# Klon wieder stoppen
$clone->stop();

# Warten bis der Klon gestoppt ist
do {
    print „waiting for clone to be 'stopped'";
    sleep 15;
}
while (not $clone->get_details->{result}->{running} eq 'false');

# Klon wieder verwerfen
$clone->delete();
```

Listing 2

Da die Fehlermeldungen je nach Herkunft verschiedener Art sein können (einfacher String oder komplexe Datenstruktur), wird diese momentan einfach im Cache `last` abgelegt und kann von dort aus z.B. mit `Data::Dumper` ausgegeben werden:

```
my $id =
  $hypervisor->get_id_from_name($name);
die Dumper($hypervisor->last) unless($id);
```

Bis zum Erscheinen des Artikels ist geplant, einen spezifischen Cache names `error` zur Verfügung zu stellen, der die Fehlermeldungen jeweils als String serialisiert enthält, so dass auch das Ausweichen auf `Data::Dumper` entfällt.

Komplexes Szenario: On-the-fly Dummy-Clone

Im Folgenden nun ein komplexeres Anwendungsszenario. Mit der Hilfe von `VM::JiffyBox` soll automatisch eine Kopie eines Servers erstellt werden, indem sein letztes Backup auf einer neuen Maschine gestartet wird. Dies hat den Vorteil, dass die so geklonte Maschine im Betrieb nicht gestört wird, denn das Erstellen eines wirklichen live-Klons ist ressourcenintensiv.

Das Beispiel in Listing 2 zeigt, dass sich bereits mit dieser frühen Version von `VM::JiffyBox` wirkmächtige Kommandos kurz und prägnant formulieren lassen. Es besteht aber auch noch viel Potential für Verbesserungen. Viele Kommandos wie z.B. `REST-PUT` sind asynchron, was der Grund für das Verwenden der `do-while`-Schleife ist. Hier wäre es z.B. viel besser, so etwas schreiben zu können:

```
$clone->stop(wait => 1);
```

Oder wenigstens:

```
do { } while (not $clone->is_running());
```

Idee für die Zukunft: Plugins

Die Grundfunktionalität der API ist relativ bescheiden: erstellen, starten, stoppen, Informationen holen (und ein paar weitere). Richtig interessant wird es aber, wenn man die Grundfunktionen mit den geholten Informationen kombinieren kann. Hier lassen sich dann erst die richtig interessanten Dinge anstellen. Die bereits erwähnte Methode `get_id_from_name()` ist so ein Beispiel. Sie kommt nicht in der API vor, sondern bedient sich dieser um ein einfacheres Angebot bereit zu stellen: Einen Servernamen in eine ID zu übersetzen.

Ohne diese Methode müsste ich dafür jedes mal diesen Code aufrufen:

```
# Gesuchter Wert
my $vm_id = '';

# Informationen holen
my $info  = $hypervisor->get_details;

# Suche Name in Informationen
foreach my $vm (values %{$info->{result}}) {
    if ($vm->{name} eq 'Produktion8') {
        $vm_id = $vm->{id};
        last; # EXIT LOOP
    }
}
```

Da jeder Nutzer andere Vorstellungen darüber hat, was nützliche und oft gebrauchte Aufrufe sind, würde sich hier ein Plugin-System anbieten. Die Methode `get_id_from_name()` ist im Moment in das Modul fest eincodiert. Lässt man dies so, hat dies den Nachteil, das jeder seine Lieblingsmethode als Patch für das Hauptmodul einreichen müsste, wenn er sie anderen zur Verfügung stellen möchte. Viel besser aber wäre, wenn er sie völlig unabhängig als Plugin auf CPAN veröffentlichen könnte. Auf dieser Weise steht die Möglichkeit offen, dass sich im Idealfall irgendwann eine große Sammlung von Plugins bilden kann. Diese Funktionalität ist bis zur Abgabe des Artikels und vermutlich auch zur Veröffentlichung noch nicht implementiert. Falls Interesse daran besteht, kann man gerne mit den Modul-Autoren Kontakt aufnehmen.

Testing und Debugging

Manchmal kommt es vor, dass etwas nicht wie gewünscht funktioniert. Oder man möchte erst einmal wissen, was da eigentlich unter der Haube passiert, bevor der Request auf das System losgelassen wird. `JiffyBox` bietet hier die Möglichkeit, etwas Einsicht zu bekommen. Wenn man beim Hypervisor das Attribut `test_mode` auf einen wahren Wert setzt, geben die API-Methoden die Informationen über den

Request zurück, den sie abgesetzt hättet, wenn es kein Test wäre.

```
my $hypervisor = VM::JiffyBox->new(
              token => 'super_secret_token',
              test_mode => 1
                 );
my $vm = $hypervisor->get_vm(12345);
my $req_info = $vm->get_details

print Dumper($req_info),"\n";
```

Dieser Aufruf giebt nun lediglich die URL zurück, auf welche die Abfrage stattgefunden hätte::

```
$VAR1 = {
  'url' => 'https://api.jiffybox.de/
    super_secret_token/v1.0/jiffyBoxes/12345'
};
```

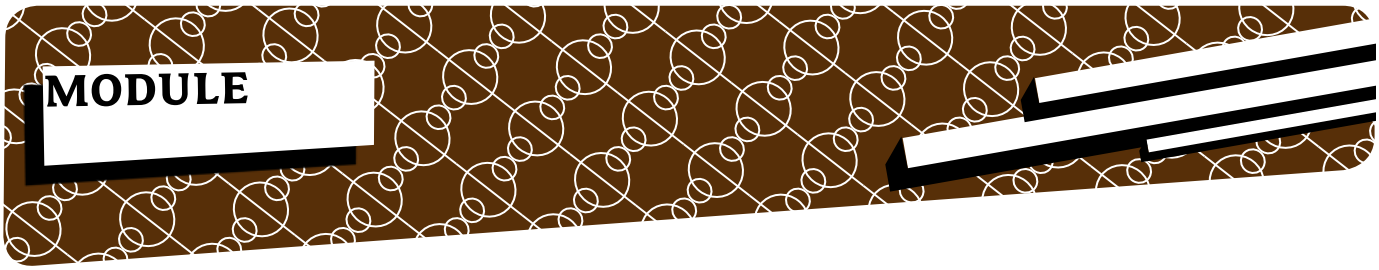
Manche Aufrufe benutzen intern auch JSON als POST-Parameter. Der etwas komplexere Aufruf von `create_vm()` ist z.B. so ein Fall.

```
$vm->create_vm(
   name     => 'Produktion2',
   planid   => 20,
   backupid => 'abcdefg'
;)
```

Hier wird dann einfach das JSON mit ausgeliefert:

```
$VAR1 = {
  'url' => 'https://api.jiffybox.de/
         super_secret_token/v1.0/jiffyBoxes',
  'json' => '{„backupid":"abcdefg",
           "planid":20,"name":"Produktion2"}'
};
```

Diese Funktionalität findet u.a. auch in der Testsuite des Moduls Anwendung, um zu gewährleisten, dass die Methoden das Erwartete auch nach Updates weiterhin liefern.



Renée Bäcker

Modern Art des Profilens

Wieso braucht das Programm so lange? Wo ist der Flaschenhals? Nur einen Blick auf den Perl-Code zu werfen reicht in den seltensten Fällen um problematischen Code zu entdecken. Häufig verstecken sich die Performanzfresser sehr gut. Die Performanz ist in vielen Fällen aber ein entscheidender Faktor. Mit `Devel::NYTProf` gibt es einen wunderbaren Profiler für Perl.

Arten des Profilens

Das Profiling in Perl kann auf drei Arten gemacht werden: Befehlsorientiert, Unterprogrammorientiert und bestimmte Anwendungsbereiche. Die bisherigen Profiler konnten immer nur eins.

Befehlsorientiert

Beim Befehlsorientierten profilen wird die Zeit gemessen, die zwischen der Ausführung des einen Befehls und dem nächsten Befehl vergeht. Sobald ein neuer Befehl erreicht wird, wird die Zeit berechnet, die seit dem Beginn des vorherigen Befehls vergangen ist. Diese wird dann der Code-Zeile zugeordnet, in der der vorherige Befehl begann.

Standardmäßig ermittelt der befehlsorientierte Profiler die erste Zeile des aktuellen Blocks und die erste Zeile des aktuellen Befehls und summiert deren Zeiten. `Devel::NYTProf` ist der einzige Profiler für Perl, der auch auf Block-Basis profilen kann.

Ein weiteres Problem von vielen Befehlsorientierten Profilern wird in `NYTProf` gelöst: Wenn ein Statement eine Subroutine aufruft und dann etwas anderes ausführt, das keinen neuen Befehl ausführt, z.B.

```
funktion() + mkdir('./verzeichnis');
```

wird die Zeit, die für das Erstellen des Verzeichnisses benötigt wird, auf den letzten Befehl von `funktion` addiert. Das kann die Zeiten des letzten Befehls von `funktion` enorm verändern. Dauert es in diesem Beispiel lange, bis das Verzeichnis erstellt ist, erkennt man das nicht bei den meisten Befehlsorientierten Profilern. Stattdessen wundert man sich, warum der letzte Befehl in `funktion` so viel Zeit benötigt und eventuell fängt man dann an der falschen Stelle mit dem Optimieren an.

`Devel::NYTProf` umgeht dieses Problem, in dem es die opcodes abfängt - die anzeigen, dass die Kontrolle an ein anderes Statement ging - und das Profiling dementsprechend anpasst.

Subroutinen-orientiert

Profiler, die Subroutinen-orientiert arbeiten, messen die Zeit zwischen dem "Betreten" und dem "Verlassen" der Subroutine. Dabei wird der Zähler für Aufrufe inkrementiert und die Zeit addiert. Für jede aufgerufene Subroutine wird der Zähler und die Dauer extra gespeichert - und das für jede Stelle, an der die Subroutine aufgerufen wird.

Das "Betreten" der Subroutine wird durch das Abfangen des `entersub` opcodes erkannt; das Verlassen über perl's eigenen Stack. Das Ergebnis ist schnell und sehr robust.

Allerdings gibt es bei den meisten Subroutinen-orientierten Profilern ein Problem: Man kann sie durch Sprungbefehle wie `goto` oder `next` durcheinander bringen.

Anwendungsbereich

Im großen Gegensatz zu den zwei zuvor genannten Arten steht das Profiling eines bestimmten Anwendungsbereichs. Damit ist zum Beispiel die Untersuchung nur der Datenbank-Abfragen gemeint. In Ausgabe 10 von $foo wurde im Artikel "111% DBIx::Class" gezeigt, wie man Datenbankabfragen mit `DBIx::Class` profilen kann. Hier wird dann nicht der Perl-Code betrachtet, sondern nur die Datenbankabfragen an sich.

Das kann dann ganz sinnvoll sein, wenn man bei der Untersuchung des Perl-Codes festgestellt hat, dass eine bestimmte Subroutine sehr viel Zeit benötigt. Wenn diese Subroutine Datenbank-Abfragen beinhaltet, kann man diese dann untersuchen um festzustellen, ob die Datenbank der Flaschenhals ist.

`Devel::NYTProf` kann sowohl zeilenorientiert als auch unterprogrammorientiert profilen, was neben anderen Features ein großer Pluspunkt ist. So muss man seine Programme nicht doppelt profilen.

CPU-Zeit vs. Real Time

Beim Schreiben eines Profilers stellt sich auch die Frage, ob der Zeitaufwand in CPU-Zeit oder in Realer Zeit gemessen werden soll. Das kann unter Umständen sehr unterschiedliche Ergebnisse hervorbringen. Der Vorteil der Messung in CPU-Zeit ist, dass die Auslastung des Rechners quasi keine Rolle spielt. Aber die Zeiten werden maximal in 0,01-Sekunden-Schritten gemessen. Die Messung in realer Zeit hat da eine viel bessere Auflösung (0,000001 Sekunden). Zusätzlich wird beim Profilen mit realer Zeit auch die non-CPU-Wartezeit (z.B. warten auf die Datenbank oder Zugriffe auf die Festplatte) mit ausgegeben.

Ein Programm profilen

Aber wie untersucht man jetzt das Programm? Die Einbindung von `Devel::NYTProf` ist einfach. Genau wie bei anderen `Devel::`-Modulen kann man beim Aufruf von Perl mit dem `-d`-Parameter das Modul einbinden:

```
perl -d:NYTProf anwendung.pl
```

Alternativ kann auch die Umgebungsvariable `PERL5DB` gesetzt werden:

```
PERL5DB='use Devel::NYTProf'
```

Das Beispielprogramm für diesen Artikel ist in Listing 1 dargestellt:

```
#!/usr/bin/perl

use strict;
use warnings;

use B ();

test();

B::walkoptree( B::main_root(), 'debug' );

sub test {
    my $a = 3;
}

sub UNIVERSAL::debug {
    print $_[0]->name,"\n";
}
```

Listing 1

Ausgabeformate

Die Profiling-Ergebnisse können von Haus aus bei der Auswertung in zwei Formate umgewandelt werden: CSV oder HTML. Die CSV-Daten können dann z.B. leicht in eine Datenbank importiert werden, was sinnvoll sein kann, wenn man die Performanz einer Anwendung über einen längeren Zeitraum beobachten will.

Für die Auswertung der Daten gibt es entsprechend den Formaten die Programme `nytprofcsv` und `nytprofhtml`, die mit dem Modul mitgeliefert werden.

Die Profiling-Daten werden mit folgendem Befehl in CSV-Dateien umgewandelt:

```
nytprofcsv -f nytprof.out
```

Damit wird ein Verzeichnis `nytprof` erzeugt und darin die CSV-Dateien abgelegt - für jedes Modul/Skript drei Dateien: `<Modul>-line.csv`, `<Modul>-block.csv` und `<Modul>-sub.csv`.

Für den normalen Gebrauch ist das HTML-Format sehr gut geeignet. Ähnlich wie bei der Generierung von CSV-Dateien werden mit `nytprofhtml` HTML-Dateien erstellt. Zusätzlich noch ein paar CSS- und JavaScript-Dateien. In Abbildung 1 ist die Startseite der generierten HTML-Dateien zu sehen.

Diese Startseite ist eine Zusammenfassung der Ergebnisse. In der oberen Tabelle sind die Subroutinen aufgeführt, die am meisten Zeit verbrauchen. Darunter sind die "langsamsten" Dateien aufgelistet. Die Werte sind durch Farbe hinterlegt, so dass man schon auf den ersten Blick die "kritischen" Daten erfassen kann. Die "Grenzen" für die einzelnen Farben werden für jeden Programmlauf basierend auf den jeweiligen statistischen Daten neu ermittelt. Man kann also nicht sagen, dass ein "200ms-Befehl" immer rot ist. In einem allgemein langsamen Skript kann das durchaus zu den schnellsten Befehlen gehören. Dann wird die Zeit grün hinterlegt. Von dieser Startseite aus, kann man sich durch die Profiling-Ergebnisse navigieren. Insgesamt kann man sagen, dass die Dateien untereinander sehr gut verlinkt sind.

Wie bei den CSV-Dateien gibt es auch bei der HTML-Auswertung verschiedene Dateien pro Modul/Skript. Bei jeder Ansicht (line/block/sub) gibt es ganz oben eine kurze Zusammenfassung, um welche Datei es sich handelt, wie viele Statements in dieser Datei ausgeführt wurden und wie viel Zeit insgesamt darauf verwendet wurde (siehe Abbildung 2).

Unter der Übersicht, gibt es eine Liste der Subroutinen mit den dazugehörigen Profiling-Daten. Diese Spalten gibt es dabei:

• Calls : Wie oft wurde diese Subroutine aufgerufen

• P: Von wievielen Stellen im Code wurde diese Subroutine aufgerufen

• F : Aus wievielen Dateien heraus wurde diese Subroutine aufgerufen

• Exclusive Time: Zeitverbrauch der Subroutine, ohne die Subroutinen, die darin aufgerufen wurden

• Inclusive Time: Zeitverbrauch der Subroutine, inklusive der Subroutinen, die darin aufgerufen wurden

In der Tabelle mit dem Quellcode werden dann noch Angaben darüber gemacht, wieviele Statements in dieser Zeile abgearbeitet werden und wieviel Zeit dafür verbraucht wurde. Im Quellcode sind auch die Kommentare sehr nützlich, da dort noch erweiterte Informationen zu finden sind. Z.B. Angaben darüber, wieviel Zeit für eine gewisse Anzahl von Aufrufen einer anderen Subroutine verbraucht wurde. Der Kommentar sieht dann beispielsweise so aus:

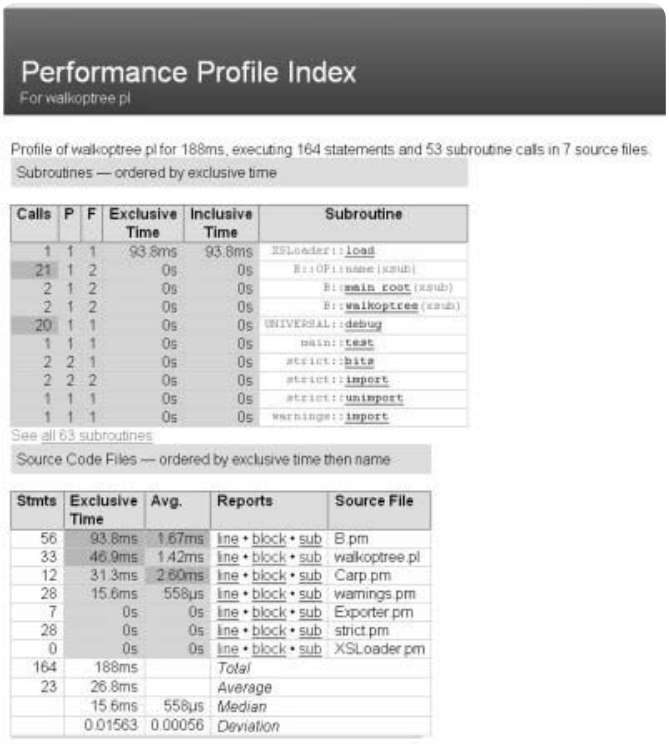


Abbildung 1

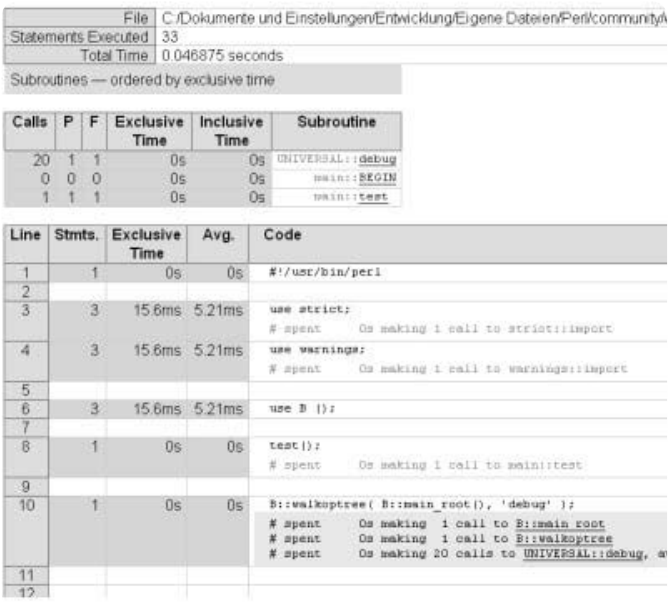


Abbildung 2

```
# spent 0s making 20 calls to
   UNIVERSAL::debug, avg 0s/call
```

Die Subroutinen-Namen sind auch verlinkt, so dass man gleich zu der anderen Stelle springen kann. Umgekehrt gibt es auch Links, zu den Stellen, von denen aus die Subroutine aufgerufen wurde.

Ein Vorteil von `Devel::NYTProf` ist auch, dass man zwischen den Verschiedenen Darstellungen (line/block/sub) switchen kann. So kann man sich immer die gewünschten Daten anschauen. Nicht wundern: Die Zeiten für eine Subroutine werden immer bei der ersten Zeile der Subroutine zusammengefasst und bei der Block-Auswertung bei der ersten Zeile des Blocks (siehe Abbildung 3).

Devel::NYTProf und Webanwendungen

Das bisher gezeigte Beispiel behandelt ein einfaches Skript, eine Webanwendung sollte bei Perfomance-Problemen ebenfalls untersucht werden. Aus diesem Grund kann man `Devel::NYTProf` auch in den Apachen einbinden. Damit kann man die Zeitfresser bei allen möglichen Nutzer-Interaktionen herausfinden. Wer jetzt noch eine große Testsuite aus z.B. Selenium-Tests hat, kann jetzt unkompliziert die Anwendung profilen.

Man muss nur noch das Modul einbinden. Das geht über die Shebang:

```
#/usr/bin/perl -d:NYTProf
```

Beim Aufruf der Anwendung wird jetzt das Modul geladen und das Profiling wird wie oben beschrieben durchgeführt.

Aber auch mit mod_perl-Anwendungen kommt das Modul zurecht. Zuvor muss das Modul aber noch in den Apachen eingebunden werden. Für die Interaktion mit Apache gibt es das Modul `Devel::NYTProf::Apache`, das mit der Apache-Direktive `PerlModule` geladen wird:

```
PerlModule Devel::NYTProf::Apache
```

Damit wird bei jedem Request die aufgerufene Anwendung untersucht. Die Daten landen dann in `/tmp/nytprof.$$.out`. `$$` steht dabei für die ID des Prozesses.

`Devel::NYTProf` erkennt auch, wann die Kontrolle von perl an mod_perl übergeht (wenn der Request abgearbeitet wurde). Dadurch wird verhindert, dass der zuletzt ausgeführte Befehl die Zeit von mod_perl (z.B. Wartezeit auf den nächsten Request) angerechnet bekommt. Das würde sonst die Ergebnisse verfälschen.

Catalyst

Möchte man Catalyst-Anwendungen untersuchen, kann man so vorgehen:

				sub find {			
365	39004	0.14204	4e-06	my $self = shift;	365	2744927	9.28967
366				my $wanted = $self->_	366		
				# spent 0.37123s maki			
367					367		
368				# Use a queue based s	368		
369				my @found = ();	369		
370				my @queue = $self->ch	370		
				# spent 0.04848s maki			
371	5572	0.01181	2e-06	eval {	371		
372	2539326	7.23647	3e-06	while (my $Element =	372		
373	1	0.84905	0.84905	my $rv = &$wanted($s	373		

Abbildung 3

```
$ perl -d:NYTProf script/yourapp.pl
$ ab -n 10000 http://localhost:3000/path/to/
                    slow/page
$ GET http://localhost:3000/quit
$ nytprofhtml
```

Der dritte Schritt ist notwendig, damit `Devel::NYTProf` sauber beendet wird. `quit` sollte dabei so aussehen:

```
sub quit :Global { exit(0) }
```

Devel::NYTProf und ge'fork'te Anwendungen

`Devel::NYTProf` kann auch Anwendungen profilen, die mit `fork` arbeiten. Dabei leidet die auch nicht Performanz darunter. Das Modul erkennt den fork und beginnt eine neue Datei mit den Profiling-Daten. Die Prozess-ID wird dabei an den Dateinamen angehängt.

Ein weiterer Weg für das Profilen von Webanwendungen ist die Verwendung von `Plack`. `Plack` unterstützt sogenannte Middleware - und auf CPAN gibt es eine Unmenge davon. Eines der Middleware-Module fügt ein Panel für Debugging-Zwecke in die HTML-Ausgabe ein. Und dafür gibt es ein Plug-in, das die Ausführung mit `Devel::NYTProf` aufzeichnet.

Dieses Module sind auch über die Kommandozeile zu aktivieren, so dass man das Profiling einschalten kann, ohne an der Anwendung an sich etwas zu ändern:

```
plackup -e 'enable "Debug",
    panels =>["Profiler::NYTProf"];'
script/app.psgi
```

Dancer und Mojolicious liefern automatisch schon Plack-fähigen Output, so dass das ein bequemer Weg für den Entwickler ist. Im Browser bekommt man dann ein Panel angezeigt, über das man sich die Aufzeichnungen von `Devel::NYTProf` anzeigen lassen kann. Das ist in Abbildung 4 zu sehen.

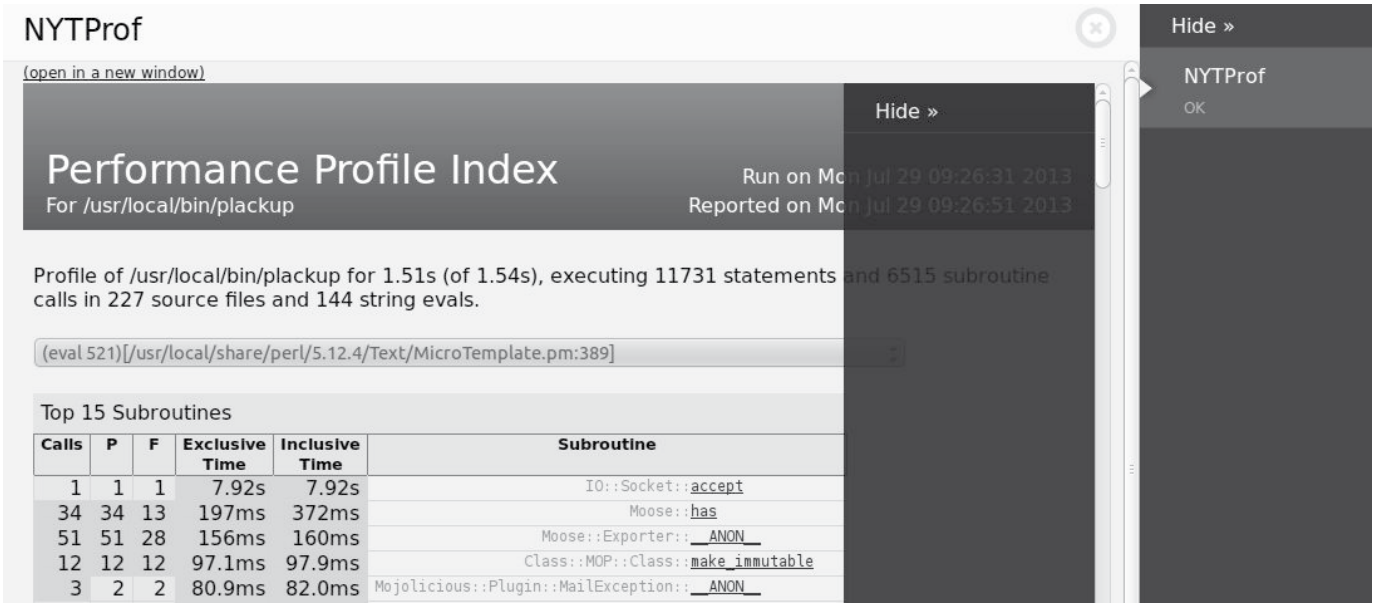


Abbildung 4

MODULE

Wolfgang Kinkeldei

DBIx::Class für Fortgeschrittene

Rückblick

Die letzten Artikel zu `DBIx::Class` (kurz DBIC) im $foo-Magazin sind schon eine Weile her. In der Zwischenzeit hat sich DBIC auch stark weiterentwickelt, zahlreiche zusätzliche Plugins oder Zusatzmodule sind auf CPAN verfügbar.

So leicht auch die ersten Anfänge sind, so gibt es doch Dinge, die man in SQL kennt und kann, die aber in DBIC schwer formulierbar scheinen oder immer wieder auf Schwierigkeiten stoßen: Joins und Subqueries.

Beziehungen (Relationships)

Da DBIC den direkten Kontakt des Entwicklers mit SQL eher vermeiden will, gibt es kein direktes Gegenstück zum SQL 'JOIN' Befehl. Es existiert wohl eine `join` Option, die der `search` Methode mitgegeben werden kann, das ist aber auch schon die einzige Gemeinsamkeit. Der Schlüssel zum erfolgreichen Zugriff auf zusätzliche Tabellen ist die Definition und Benutzung der Beziehungen zwischen den Tabellen.

Zur Verdeutlichung soll ein einfaches Datenbank-Schema dienen (siehe Abbildung 1). Nichts umwerfendes, aber ausreichend um die relevanten Begriffe vollständig unterzubringen. Das vorliegende Schema benutzt ausschließlich englische Begriffe, die Tabellen-Namen sind im Singular. Primär- und Fremdschlüssel sind identisch benannt und bestehen aus dem Tabellen-Namen gefolgt von `_id`. Die Tabellen selbst enthalten lediglich wenige exemplarische Attribute. Diese sind ohne jegliche Namenszusätze sprechend bezeichnet.

Zunächst benötigen wir eine Schema-Klasse, damit wir Verbindungen mit der Datenbank aufbauen und auf die einzelnen Resultset-Klassen zugreifen können. Da wir keine Moose-Eigenschaften benötigen, erzeugen wir die Klasse ganz konventionell. Die `$VERSION`-Variable wird an dieser Stelle nur dazu benötigt, damit das von mir an dieser Stelle eingesetzte Modul `DBIx::Class::DeploymentHandler` in der Lage ist, versionsabhängig die notwendigen SQL-Anweisungen zur Aktualisierung gegenüber der Vorgängerversion zu erstellen. Wir ignorieren dies für den Moment.

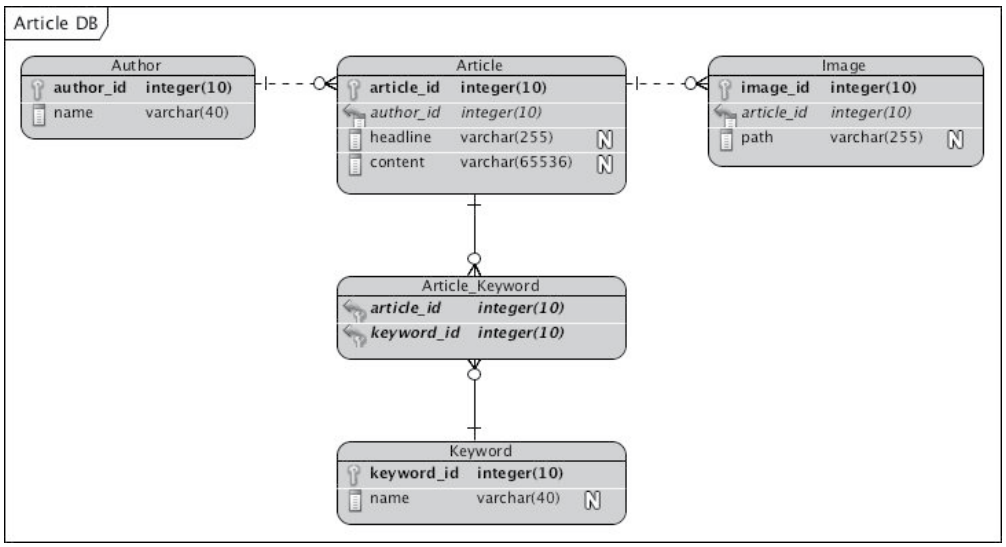


Abbildung 1

```
package DbDemo::Schema;
use base 'DBIx::Class::Schema';

# für DBIx::Class::DeploymentHandler
our $VERSION = 1;

__PACKAGE__->load_namespaces;

1;
```

Nun gehen wir daran, die einzelnen Tabellen zu modellieren. Da die Syntax der Tabellen-Definition mit purem `DBIx::Class` relativ unschön aussieht, sei an dieser Stelle der Einsatz von `DBIx::Class::Candy` gezeigt. Es bietet eine Syntax, die Moose ähnelt. Im Gegensatz zu Moose allerdings werden die Merkmale für eine Spalte als Hash-Referenz vorgenommen.

```
package DbDemo::Schema::Result::Author;
use DBIx::Class::Candy;

table 'author';

primary_column author_id => {
    data_type         => 'int',
    is_auto_increment => 1,
};

column name => {
    data_type => 'varchar',
};

has_many articles =>
       'DbDemo::Schema::Result::Article',
       'author_id';

1;
```

Entscheidend an dieser Stelle ist die letzte Deklaration `has_many`. Damit wird die Beziehung `articles` definiert, die über den Fremdschlüssel `author_id` der entfernten Tabelle `Article` eine Beziehung zum Autor (diese Tabelle) herstellt. Der Plural an dieser Stelle ist durchaus beabsichtigt und macht semantisch Sinn, denn ein Autor hat vermutlich mehrere Artikel verfasst.

Ähnlich sieht es auf der gegenüberliegenden Seite, der Tabelle `Article`, aus. Auch hier wird die Beziehung zum Autor aufgebaut - neben vielen anderen Dingen, die mit dem Artikel in Berührung kommen. Singular oder Plural wird je nach Multiplizität bewusst eingesetzt.

```
package DbDemo::Schema::Result::Article;
use DBIx::Class::Candy;

table 'article';

primary_column article_id => {
    data_type         => 'int',
    is_auto_increment => 1,
};

column author_id => {
    data_type   => 'int',
    is_nullable => 0,
};

column headline => {
    data_type => 'varchar',
};

column content => {
    data_type => 'varchar',
};

belongs_to author =>
    'DbDemo::Schema::Result::Author',
    'author_id';

has_many images   =>
    'DbDemo::Schema::Result::Image',
    'article_id';

has_many article_keywords =>
 'DbDemo::Schema::Result::ArticleKeyword',
 'article_id';

many_to_many keywords =>
    'article_keywords',
    'keyword';

1;
```

In epischer Breite sind die diversen Beziehungs-Schlüsselworte in `DBIx::Class::Relationship` beschrieben. Die drei wichtigsten sind nachfolgend gelistet. Zusätzlich gibt es noch `has_one` und `might_have`, auf die hier allerdings verzichtet werden soll. Auch gibt es noch wesentlich mehr Möglichkeiten, die SQL-Auswirkung einer Beziehung zu definieren, indem eine Beziehungs-Definition noch einen letzten Parameter in Form einer Hash-Referenz erhält oder anstelle der Spalte bei Argument 3 die Join-Bedingung in alternativer Form definiert wird.

has_many

definiert eine 1:n Beziehung. Hier taucht der Primärschlüssel der eigenen Tabelle als Fremdschlüssel in der entfernten Tabelle auf. Normalerweise wird in der entfernten Tabelle eine `belongs_to` Beziehung definiert.

belongs_to

definiert eine n:1 Beziehung und wird für das Gegenstück einer `has_many` Beziehung verwendet

many_to_many
n:m Beziehungen werden über eine Zwischentabelle modelliert. Jede der beiden Seiten unterhält eine `has_many` Beziehung zur Zwischentabelle und von der Zwischentabelle zu jedem Ende gibt es je eine `belongs_to` Beziehung. Mittels einer zusätzlichen `many_to_many` Beziehung kann die Zwischentabelle quasi übersprungen werden. Die nachfolgenden Abfragen werden das verdeutlichen.

Befüllen unserer Datenbank

Bevor wir uns Abfragen in Verbindung mit Beziehungen ansehen können, brauchen wir ein paar Daten. Wer es gewohnt ist, sich händisch mit SQL durch das Geflecht der Tabellen zu hangeln und Teil für Teil seine Daten zu erstellen, wird staunen, wie einfach das mit Beziehungen sein kann:

```
$schema->resultset('Article')->create(
    {
        headline => 'Foo #1',
        content  => 'Erster $foo Artikel',
        author   => {
            name => 'Viel Schreiber',
        },
        images => [
            { path => '/path/to/img1.png' },
            { path => '/path/to/img2.png' },
        ],
        article_keywords => [
          { keyword => { name => 'DBIC' } }
          { keyword => { name => 'Perl' } },
        ],
    }
);
```

Wie? Kein lästiges Sich-um-Fremdschlüssel-kümmern-müssen? Nein! Einfach die Daten in der richtigen Struktur bereitstellen und es werden genau so viele Zeilen in den richtigen Tabellen angelegt wie notwendig. Lediglich die `many_to_many` Beziehung kann an dieser Stelle nicht verwendet werden. Stattdessen muss man den Weg über die Zwischentabelle gehen. Wer mehrere Datensätze auf diese Weise auf einmal erzeugen möchte, kann dies auch mit der Methode `populate` tun.

Alternativ kann man die einzelnen Tabellen auch gerne einzeln befüllen, macht nur weniger Freude und ist deutlich umfangreicher.

```
my $article =
 $schema->resultset('Article')->create(
    {
        headline => 'Foo #1',
        content  => 'Erster $foo Artikel',
    }
);

$article->create_related(
    'authors',
    {
        name => 'Viel Schreiber',
    }
);

# ... usw.
```

Abfragen mit Beziehungen

Hat man Beziehungen definiert, so kann man diese auch in Abfragen verwenden. Das kann entweder implizit geschehen (was in der Regel weitere SQL-Abfragen zur Folge hat) oder explizit ausgeführt werden. Implizit heißt, dass jede Beziehung einfach in Form einer Methode zur Verfügung steht. Mit Singular bezeichnete Methoden liefern das jeweilige Objekt auf der „anderen" Seite. Im Plural formulierte Methoden liefern im Skalar-Kontext ein `ResultSet`-Objekt, das für weitere Abfragen genutzt werden kann, im Listen-Kontext die Objekte, die man durch Verfolgen der Beziehung erwarten würde.

```
my $article =
  $schema->resultset('Article')->find(1);

# Zugriff auf eine Beziehung,
# aber Auslösen einer SQL Abfrage
my $author = $article->author;

# entspricht dieser Abfrage:
my $author =
      $schema->resultset('Author')
      ->find($article->author_id);
```

Will man die separat ausgelösten Abfragen vermeiden, so kann man der eigentlichen Abfrage die `prefetch` Option mitgeben, in der spezifiziert wird, welche Beziehung(en) zusätzlich mit abgefragt werden sollen.

```
my $article
  = $schema->resultset('Article')->find(
    1,
    {
        prefetch => 'author',
    }
);

# gleicher Zugriff wie oben, aber löst
# keine weitere Abfrage aus
my $author = $article->author;
```

Natürlich kann man in einer Abfrage gleich sämtliche Beziehungen mit einem mal bedienen. Abhängig von der bei `prefetch` angegebenen Datenstruktur erfolgt dann der „Weg" durch die Datenbank.

```
my $article =
  $schema->resultset('Article')->find(
    1,
    {
        prefetch => [
           'author', 'images',
           { article_keywords => 'keyword' }
        ],
    }
);

# keine weiteren Abfragen durch diese
# Aufrufe:
say 'Artikel: ',     $article->headline;
say 'Autor: ',       $article->author->name;
say 'Bilder: ',      join(', ',
     map { $_->path } $article->images);
say 'Schlagworte: ', join(', ',
     map { $_->name } $article->keywords);
```

Schaut man sich das erzeugte SQL der Abfragen an, so wird man feststellen, dass der Einsatz von `prefetch` pro angegebener Beziehung einen `JOIN`-Befehl in das erzeugte SQL setzt. Warum haben wir dann nicht `join` als Option der obigen Abfrage benutzt, sondern `prefetch`? Oder: Was ist der Unterschied zwischen den beiden?

Auf SQL-Ebene gibt es zunächst keinen Unterschied. Der Unterschied besteht in der Verarbeitung der Ergebnis-Menge und der anschließenden Bereitstellung der Ergebnisse als `DBIx::Class::Row`-Objekt(e). Eine Abfrage mit `join` liefert genau die Menge an Zeilen wie die SQL-Abfrage Zeilen ausspuckt. Nutzt man hingegen `prefetch`, entsteht eine Struktur, die dem entspricht, was wir zum Befüllen unserer Datenbank genutzt haben und die Semantik hinter dem Datenmodell widerspiegelt. Allerdings hat das seinen Preis. In unserem Fall haben wir 2 Bilder und 2 Schlagworte, das SQL liefert also 4 Datensätze für den Artikel mit jeweils allen Kombinationen von Bild und Schlagwort. Nicht tragisch, doch hätten wir 100 Schlagworte und 100 Bilder, würde die Abfrage 10_000 Zeilen liefern, aus der dann 1 Artikel-Objekt mit 1 Autor, 100 Bildern und 100 Schlagworten erzeugt wird. Damit können scheinbar harmlose Abfragen schnell zum Performance-Killer werden.

Der Einsatz von `join` hingegen ist dann sinnvoll, wenn man mittels `+select` / `+as` neue Ergebnis-Spalten definieren will und die gelieferte Anzahl an Ergebnis-Zeilen erwünscht ist.

Verschachtelte Abfragen

Manchmal macht es Sinn, Abfragen komplexer zu gestalten. Sei es, um die Anzahl der Abfragen klein zu halten oder weil die Reduktion der Ergebnismenge in der Datenbank einfach effizienter ist, als wenn man im abfragenden Programm nochmals Hand anlegen muss. Würde man natives SQL erzeugen, wären Unterabfragen (Subqueries) ein beliebtes Mittel. Geht so etwas mit DBIC und wenn ja, wie?

Zunächst müssen wir uns überlegen, an welchen Stellen innerhalb einer SQL-Abfrage Unterabfragen möglich sind. Nachfolgend sind die Stellen mit `???` gekennzeichnet, an denen mittels DBIC Unterabfragen möglich sind. SQL erlaubt noch, im `FROM` Teil einer Abfrage bzw. in Verbindung mit `JOIN`s Unterabfragen einzusetzen, hier ist DBIC jedoch limitiert. In solchen Situationen ist die Benutzung von Datenbank-Views oder der Einsatz von `DBIx::Class::ResultSource::View` empfehlenswert.

```
my $xxx_rs =
  $schema->resultset('Article')->search(
    {
        # Vergleich gegen ID-Werte einer
        # Unterabfrage
        author_id => { -in => ??? },
        author_id => { -not_in => ??? },

        # Test auf Ergebnismenge einer
        # Unterabfrage
        exists => ???,
        -not => { exists => ??? },
        # oder !=, <, >, ...
        spalte => { '=' => ??? },
    },
    {
        # Unterabfrage liefert neue
        # Spalte(n)
        '+select' => [ ??? ],
        '+as' => [ 'result_of_subquery' ],
    }
);
```

Auch wenn diese Beispiele bei unserem einfachen Datenmodell eher suboptimal sind, denke ich, dass die Prinzipien dennoch deutlich werden.

Beispiel: Einschränkung auf bestimmte Autoren

```
my $favourite_author_ids =
   $schema->resultset('Author')->search(
    {
        # irgendeine Bedingung
    }
)->get_column('author_id')->as_query;

my $popular_article_rs =
    $schema->resultset('Article')->search(
   {
        author_id =>
         { -in => $favourite_author_ids },
    }
);
```

Beispiel: Einschränken auf Artikel, die mindestens ein Bild haben

```
my $image_for_article =
  $schema->resultset('Image')->search(
    {
        'i.article_id' =>
           { -ident => 'a.article_id' },
    },
    {
        alias => 'i',
    }
)->as_query;

my $image_article_rs =
  $schema->resultset('Article')->search(
    {
        exists => $image_for_article,
    },
    {
        alias => 'a',
    }
);
```

Genau genommen hätte man im zweiten Beispiel auf das Setzen des Alias 'a' verzichten können, wenn man stattdessen den Standard-Alias 'me' verwendet hätte. Aber sicher ist sicher ...

Und ein letztes Beispiel: eine Unterabfrage zum Liefern neuer Spalten, hier die Anzahl der Bilder pro abgefragtem Artikel.

```
my $nr_images =
 $schema->resultset('Image')->search(
    {
        'i.article_id' =>
           { -ident => 'a.article_id' },
    },
    {
        alias => 'i',
    }
)->count_rs->as_query;

my $article_rs =
  $schema->resultset('Article')->search(
    {
        # irgendeine Bedingung
    },
    {
        '+select' => [ $nr_images ],
        '+as' => [ 'nr_images' ],
        alias => 'a'
    }
);

while (my $article = $article_rs->next) {
    say 'Artikel „' .
           $article->headline . '" hat ' .
        $article->get_column('nr_images') .
           ' Bild(er)';
}
```

Werden neue Ergebnis-Spalten via `as` definiert, so ergibt sich leider keine Accessor-Methode für den Zugriff auf die Werte. Daher ist die Benutzung der Methode `get_column()` notwendig. Wer Abfragen des letzten Typs benötigt, kann sich `DBIx::Class::Helper::ResultSet::CorrelateRelationship` einmal genauer ansehen, es bietet einen etwas einfacheren Weg, diese Art von korrelierten Beziehungen zu verwenden.

Examplarischer Beispielcode ist zu finden unter *https://github.com/wki/foo_article_db*.

Boris Däppen

TAP-Archive von prove auslesen mit Convert::TAP::Archive

Perl hat eine große Tradition, was das Testen anbelangt. Entsprechend ausgebaut ist die Infrastruktur hierfür. Üblicherweise wird das Kommandozeilen-Programm `prove` verwendet, um Tests laufen zu lassen.

`prove` zeigt die Ergebnisse standardmäßig als kurze Zusammenfassung auf `STDOUT` an, bietet aber auch die Möglichkeit, spezifische Formatter anzugeben um die Ausgabe anders darzustellen. Es existieren beispielsweise Formatter für HTML oder JUnit.

Klont man sich z.B. das Repository des Moduls `VM::Jiffy-Box` und führt dann folgendes Kommando darin aus, sieht das wie folgt aus:

```
$ prove -Ilib
t/00_load.t ..... ok
t/01_create.t ..... ok
t/03_need_to_die.t ... ok
t/05_box-requests.t .. ok
All tests successful.
Files=4, Tests=33,  5 wallclock secs (
    0.12 usr  0.03 sys +  1.34 cusr
    0.16 csys =  1.65 CPU)
Result: PASS
```

Das `-Ilib` sorgt dafür, dass der lokale Code im Ordner geladen wird und nicht das Modul welches evtl. im System installiert ist. Im Falle eines Fehlers (in diesem Falle ein verlorenes Komma am Ende der URL) sieht die Standard-Ausgabe aus, wie in Listing 1 dargestellt.

```
$ prove -Ilib
t/00_load.t ..... ok
t/01_create.t ..... ok
t/03_need_to_die.t ... ok
t/05_box-requests.t .. 1/?
#   Failed test 'URL for stop (non-historic)'
#   at t/05_box-requests.t line 69.
#          got: 'https://api.jiffybox.de/AUTH_TOKEN/v1.0/jiffyBoxes/BOX_ID,'
#     expected: 'https://api.jiffybox.de/AUTH_TOKEN/v1.0/jiffyBoxes/BOX_ID'

#   Failed test 'URL for stop (historic)'
#   at t/05_box-requests.t line 101.
#          got: 'https://api.jiffybox.de/AUTH_TOKEN/v1.0/jiffyBoxes/BOX_ID,'
#     expected: 'https://api.jiffybox.de/AUTH_TOKEN/v1.0/jiffyBoxes/BOX_ID'
# Looks like you failed 2 tests of 15.
t/05_box-requests.t .. Dubious, test returned 2 (wstat 512, 0x200)
Failed 2/15 subtests

Test Summary Report
-------------------
t/05_box-requests.t (Wstat: 512 Tests: 15 Failed: 2)
  Failed tests:  3, 10
  Non-zero exit status: 2
Files=4, Tests=33,  2 wallclock secs ( 0.10 usr  0.01 sys +  1.12 cusr  0.11 csys =  1.34 CPU)
Result: FAIL
```

Listing 1

Um die Ausgabe etwas übersichtlicher zu gestalten, lässt sich ein Formatter für HTML angeben. Hierbei sollte `-Q` als Argument mitgegeben werden, damit die normalen Testergbnisse nicht mit ausgegeben werden.

```
prove -Ilib -Q --formatter=
   TAP::Formatter::HTML
```

Dies erzeugt die grafische Darstellung auf Abbildung 1. Die Grafik lässt sich mit der Maus aufklappen und gibt dann mehr über die Tests, bzw. die Fehler preis.

`prove` bietet auch die Möglichkeit, solche Tests zu archivieren.

```
prove -Ilib -a tests.tar.gz
```

Hier wird nun ein Archiv namens `tests.tar.gz` angelegt welches die Testergebnisse beinhaltet:

```
$ tar tzfv tests.tar.gz
t/01_create.t
t/05_box-requests.t
t/03_need_to_die.t
t/00_load.t
meta.yml
```

Ein Problem besteht nun, wenn man diese archivierten Tests anzeigen möchte. Hierfür hat sich nach Recherche des Autors keine einfache Möglichkeit gefunden. Auch größere Projekte wie `Tapper` oder `Smolder`, welche mit TAP-Archiven umgehen können, entpacken die Archive jeweils mühsam „händisch". Das geht zwar, weil es ja nur ein paar Dateien in einem Archiv sind, ist aber dennoch unschön. Denn es gibt ja bereits fertige Formatter, um Testergebnisse zu konvertieren. Besser wäre es doch, wenn sich die vorhandenen Formatter auch auf diese Archive anwenden ließen. `prove` selbst scheint hier aber eine Einbahnstraße gebaut zu haben. Archive lassen sich mit `prove` zwar erstellen aber nicht lesen.

Nach einigen Recherchen [1] hat Renée die entscheidenden Punkte gefunden, um ein Archiv mit den vorhanden Formattern zu rendern:

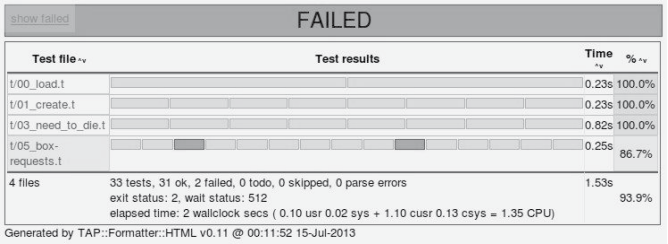


Abbildung 1

```
use TAP::Harness::Archive;
use TAP::Harness;
use TAP::Formatter::HTML;

my $formatter = TAP::Formatter::HTML->new;

my $harness   = TAP::Harness->new(
   { formatter => $formatter });

$formatter->really_quiet(1);
$formatter->prepare;

my $session;
my $aggregator = TAP::Harness::Archive
          ->aggregator_from_archive({
    archive      => '/path/to/test.tar.gz',
    parser_callbacks => {
        ALL => sub {
            $session->result( $_[0] );
        },
    },
    made_parser_callback => sub {
        $session = $formatter->open_test(
                                $_[1], $_[0] );
    }
});

$aggregator->start;
$aggregator->stop;

$formatter->summary($aggregator);
```

Dies ist nun ein ziemlich kompliziertes Stück Code, auf welches man nur anhand der Dokumentation von Test::Harness und Konsorten auch nicht so ohne Weiteres kommt. Um den Anwendungsfall des Lesens und formatierten Ausgebens von TAP-Archiven einfacher zu machen, wurde daher das Modul`Convert::TAP::Archive` erstellt. Von nun an genügt der Aufruf dieser wenigen Zeilen:

```
use Convert::TAP::Archive qw(
    convert_from_taparchive);

my $html = convert_from_taparchive(
               '/path/to/test.tar.gz',
               'TAP::Formatter::HTML',
           );
```

Vielen Dank an Renée Bäcker für die Unterstützung bei der Lösungsfindung und auch an die Firma *plusW* [2] bzw. Rolf Schaufelberger für die Ermöglichung des Moduls.

Links:

[1] http://stackoverflow.com/questions/17469728/extract-and-format-information-from-tap-archive

[2] http://plusw.de

Renée Bäcker

RegEx Module - Zahme Regenechsen...

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

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

Heinzelmännchen für Programmierer

Regexp::Common

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

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

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

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

Regexp::Assemble

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

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

```
#!/usr/bin/perl

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

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

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

Listing 1

```
#!/usr/bin/perl

use Regexp::Assemble;

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

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

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

Listing 2

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

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

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

Regexp::MatchContext

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

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

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

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

Listing 3

```
$ time perl regex.pl

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

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

Listing 4

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

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

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

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

Was passiert da eigentlich?

YAPE::Regexp::Explain

Reguläre Ausdrück sind teilweise verdammt schwer zu verstehen. Gerade Perl-Einsteiger schauen häufiger mal mit drei Fragezeichen im Gesicht auf ein Perl-Programm. "Was macht denn das da? Das ist ja total kryptisch." bekommt man dann

```
#!/usr/bin/perl

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

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

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

Listing 5

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

In der Ausgabe des Programms (siehe Listing 7) erkennt man, wie das Modul die einzelnen Teile des Regulären Ausdrucks erklärt. Dadurch ist es auch sehr gut dazu geeignet, die verschiedenen Konstrukte wie `(?:)` kennenzulernen.

Durch die Einrückung der Erklärungen ist gut ersichtlich, zu welchem Klammerpaar ein Teil des Ausdrucks gehört.

GraphViz::Regex

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

```
#!/usr/bin/perl

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

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

Listing 6

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

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

```
#!/usr/bin/perl

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

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

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

Listing 8

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

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

matches as follows:

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

Listing 7

Ralf Peine

Report::Porf::Framework

Einleitung

Anfang des Jahres stand ich vor der Aufgabe, eine Liste mit Hashes als Tabelle in verschiedenen Formaten auszugeben. Dazu musste es doch Perl-Module auf dem CPAN geben. Gibt es auch: Mehr als 5000 Treffer erzeugt die Suche nach „Report". Aber diese Reports sind spezialisiert für bestimmte Aufgaben und stellen meistens nur ein Ausgabeformat zur Verfügung.

Für „Report & Framework" gibt es genau einen Treffer: `Data::Report`, letzte Änderung am 17.08.2008, Version 0.10. Eine kurzer Blick in die Doku des Frameworks zeigt, dass es Klassen für die Plugins verwendet, die die Formate definieren. Der dort gewählte Ansatz kann zu sehr langsamer SW führen, Informationen über die Performance gibt es nicht.

Außerdem war mir die Anwendung der Frameworks/Reports, die ich mir angesehen habe, zu kompliziert und nicht Perl-gemäß.

Deshalb startete ich mit der Entwicklung eines eigenen, allgemeinen, offenen Report-Frameworks: Perl Open Report Framework, kurz PORF, ist der aktuelle Arbeitsname. (ORF ist schon durch irgendeinen östereichischen Fernsehsender belegt.)

Aktuell liegt die Version 0.920 für euch zum Download auf meiner Homepage bereit.

Falls jemand ein Report-Framework mit ähnlichen Eigenschaften kennt, informiert mich bitte. Dann kann ich mir die Weiterentwicklung von PORF eventuell sparen.

Einen Report in 4+n Statements konfigurieren und ausgeben

Das Ziel von PORT ist es, einen Report mit 4+n Statements zu konfigurieren und die Ergebnisse in eine Datei zu schreiben. Das ist in der aktuellen Version bereits performant umgesetzt - siehe Listing 1.

Erste Tests mit Kollegen ergaben, dass sie die API von PORF genauso leicht verständlich finden wie ich. Im einzelnen:

0. Abhängigkeiten

Es existieren bislang keine weiteren Abhängigkeiten zu anderen Perl-Modulen, bis auf einige absolute Basis-Module wie z.B. `FileHandle`. Auch das Report-Modul selbst benötigt kein `use`, weil kein `Report->new()` durch den Anwender aufgerufen werden muss bzw. darf.

1. Framework erzeugen

Mit Anweisung 1 holt man sich ein vorkonfiguriertes Report-Framework ab. Es ist sehr wichtig, sich nicht selbst eines mit `new` zu erzeugen, denn die Konfiguration des Frameworks und der Reports erfordert einige Arbeit in der „Framework-Factory". Damit werde ich mich in einem weiteren Artikel befassen.

2. Report erzeugen

Das Framework kann jetzt unverändert (Out Of The Box) verwendet werden, um einen Report im Format `$format` zu erstellen. Als Formate sind aktuell „Text" und „HTML" unterstützt, „CSV" und weitere im Aufbau. Zur Konfiguration des Reports werden hier die Klassen `HtmlReportConfigurator` und verwendet, die den Report gesteuert durch das Framework unsichtbar im Hintergrund vorkonfigurieren. Man kann beliebige eigene Reportkonfiguratoren erstellen und in einer eigenen Frameworkinstanz bereitstellen.

```
use Report::Porf::Framework;  # „use" zaehle ich nicht als Statement

# --- Report erzeugen lassen ---

my $report_framework = Report::Porf::Framework::Get();             # 1.
my $report           = $report_framework->CreateReport($format);   # 2.

# --- Spalten konfigurieren, die Daten liegen als Hash vor ---

# --- n = 3 für 3 Spalten ---
$report->ConfigureColumn(-header => 'Vorname',  -value_named => 'Prename' ); # lang
$report->ConfCol        (-h      => 'Nachname', -val_nam     => 'Surname' ); # kurz
$report->CC             (-h      => 'Alter',    -vn          => 'Age'     ); # minimal

# --- Konfiguration abschließen, das ist notwendig ---

$report->ConfigureComplete();                                      # 3.

# --- Daten ausgeben ---

$report->WriteAll($person_rows, $out_file_name);                   # 4.

# --- Fertig ! ---
```

Listing 1

```
*============+============+============*  # Fette Trennlinie
|   Vorname  |  Nachname  |    Alter   |  # Überschriften
*------------+------------+------------*  # Normale Trennlinie
| Vorname 1  | <Zelle>    | 7.69230769 |  # Eine Datenzeile mit Zellen
| Vorname 2  | Name 2     | 15.3846153 |
| Vorname 3  | Name 3     | 23.0769230 |
| Vorname 4  | Name 4     | 30.7692307 |
*============+============+============*
```

Listing 2

`CreateReport($format)` liefert eine Instanz der Klasse `Report::Porf::Report` zurück, die einen Report im geforderten Ausgabeformat erzeugen kann. Es handelt sich hier immer um dieselbe Klasse, unabhängig vom gewünschten Format. Das vereinfacht die Anwendung sehr, denn in einfachen Fällen fängt man sich keine (unerwünschten) Abhängigkeiten zu weiteren Klassen/Modulen ein.

1..n Spalten des Reports konfigurieren

Je nach Geschmack und Erfahrung kann man zwischen kurzen und langen Bezeichnern wählen. Hier ist die minimale Konfiguration angegeben: Überschrift und Zugriff auf den Wert einer Zelle (Cell). Ein Datensatz für eine Datenzeile (Row) liegt hier als Hash vor. Arrays, Klassen und freie Zugriffe werden auch unterstützt.

3. Konfiguration abschließen

Der Abschluss der Konfiguration mit `$report->ConfigureComplete();` ist notwendig, da es viele Möglichkeiten gibt, die anschließende Ausgabe durchzuführen. Außerdem kann man die Konfiguration ab jetzt nicht mehr verändern. Der Versuch, noch einmal `$report->ConfigureColumn(...);` aufzurufen, endet in einer Warnung.

Daten in Datei schreiben

Wählt man das Text-Format und schreibt man die Daten in eine Datei, erhält man folgendes Ergebnis - siehe Listing 2.

Man kann die Datenausgabe auch zeilenweise oder sogar zellweise durchführen lassen oder sich nur die Ergebnisse als String abholen, andernfalls hätte das Framework den Titel „offen" nicht verdient.

Weitere Konfigurationsmöglichkeiten

Wechsel des Ausgabeformats

Um die Tabelle als HTML auszugeben, belegt man `$format` einfach mit „HTML" statt „Text", und führt den Code noch einmal durch.

```
sub CreateAgeReport {
    my $format = shift;

    my $report_framework = Report::Porf::Framework::Get();              # 1.
    my $report           = $report_framework->CreateReport($format);    # 2.

    $report->ConfigureColumn(-header => 'Vorname',  -value_named => 'Prename' ); # lang
    $report->ConfCol        (-h      => 'Nachname', -val_nam     => 'Surname' ); # kurz
    $report->CC             (-h      => 'Alter',    -vn          => 'Age'     ); # minimal

    $report->ConfigureComplete();                                       # 3.

    return $report;
}
```

Listing 3

Es bietet sich an, alle Zeilen bis auf Anweisung 4 (Daten schreiben) in eine eigene Sub zu packen, siehe Beispiel in Listing 3.

Dann kann man die Daten als HTML oder Text ausgeben mit

```
my $report = CreateAgeReport($format);
$report->WriteAll(
  $person_rows, $out_file_name);
```

Konfiguration der Spalten

Bis hierher hätte man das alles noch irgendwie mit `join(...)` erledigen können. Aber einige der folgenden Konfigurationsmöglichkeiten für die Spalten sind damit nicht mehr leicht implementierbar.

Layout-Optionen

```
-header   -h constant: Text
-align    -a constant: (left|center|right)
                       (l   |   c   |    r)
# Der Text wird gekürzt/verlängert
-width    -w constant: integer
-format   -f constant: string für sprintf
-color    -c constant / sub {...}
```

Die `sub {...}` ermöglicht eine bedingte Einfärbung durch die Daten auf eine einfache Art. Hier zeigt sich einmal mehr die *Macht des EVAL {}*. Wie das genau funktioniert, erläutert das zweite Beispiel.

Nicht alle Optionen sind in jedem Format verfügbar. Unbekannte Optionen werden einfach ignoriert.

Datenzugriff

Daten liegen in Perl typischerweise als Array, Hash oder Instanz einer Klasse vor. Oder auch irgendwie anders. Für jeden Datentyp gibt es eigene, komfortable Zugriffsmethoden durch den Report. Es ist sogar möglich, sie miteinander zu kombinieren (falls die Daten das hergeben...)

GetValue Alternative 1: ARRAY

```
my $prename = 1;
my $surname = 2;
my $age     = 3;

# long
$report->ConfigureColumn(
     -header => 'Vorname',
     -value_indexed => $prename );
# short
$report->ConfCol        (
      -h      => 'Nachname',
      -val_idx       => $surname );
; # minimal
$report->CC             (
      -h      => 'Alter',
      -vi            => $age     )
```

GetValue Alternative 2: HASH

```
# long
$report->ConfigureColumn(
     -header => 'Vorname',
     -value_named => 'Prename' );
# short
$report->ConfCol        (
     -h      => 'Nachname',
     -val_nam     => 'Surname' );
# minimal
$report->CC             (
      -h      => 'Alter',
      -vn          => 'Age'     );
```

GetValue Alternative 3: OBJECT

```
# long
$report->ConfigureColumn(
     -header => 'Vorname',
     -value_object => 'GetPrename()');
# short
$report->ConfCol        (
     -h      => 'Nachname',
     -val_obj      => 'GetSurname()');
# minimal
$report->CC             (
      -h      => 'Alter',
      -vo           => 'GetAge()'    );
```

GetValue Alternative 4: Free

```
$report->ConfigureColumn(
    -h => 'Vorname',
    -value =>    '"Dr. „ . $_[0]->{Prename}'
);
 $report->ConfCol        (
    -h => 'Nachname',
    -val => sub { return $_[0]->{Surname}; };
 );
 $report->CC             (
    -h => 'Alter (Monate)',
    -v =>     '(12.0 * $_[0]->GetAge())'
 );
```

Der Report erzeugt sich in jedem Fall eine anonyme Sub, ähnlich wie bei 'Nachname' und 'Alter' in Alternative 4. In der aktuellen Implementierung ist das ein mehrstufiger Prozess. Man sollte sehr darauf achten, hier keinen Syntaxfehler in den `value`-Optionen einzubauen. Die Fehlersuche kann sich sehr schwierig gestalten. Das ist ein Nachteil des Frameworks, der sich nur vermeiden lässt, wenn man die Performance und den Komfort drastisch reduziert.

Weitere Beispiele

Bedingte Einfärbung

Für alle Personen aus der Liste, die noch nicht mindestens 18 Jahre alt sind, soll die Alter-Zelle rot eingefärbt werden. Dazu benötigt man eine sub {} für die Konfiguration der Zellfarbe:

```
$report->ConfigureColumn(
   -header  => 'Age',
   -width   => 15,
   -align   => 'Right',
   -format  =>  „%.3f years",
   -color   =>
       sub { return $_[0]->{Age} <=
                    18 ? „": '#EECCCC'; },
   -vn      => 'Age', );
```

Außerdem wird mit `-format` noch die Anzahl der Nachkommastellen auf 3 beschränkt. Man beachte, dass man in der Sub Zugriff auf die/den gesamten Datensatz/-zeile hat, sodass auch Bedingungen mit mehreren Parametern als Input möglich sind.

Spezialanzeige für Werte

In diesem Beispiel wird die Augenanzahl eines Würfels im HTML-Report auch in einer zusätzlichen Spalte als Grafik angezeigt:

Count	Throws	Dices
1	1	
2	6	
3	10	
4	11	
5	12	
6	13	
7	0123456789ab	

Abbildung 1: Tabelle mit Würfeln

Dazu benötigt man zunächst 10 Bilder, die 0 bis 9 Augen auf einer Würfelseite anzeigen. Je nach Wert des Wurfs wird dann das entsprechende Bild in der HTML-Tabelle angezeigt.

Dazu wird eine spezielle -value Option verwendet:

```
sub { return $dices_to_image->($_[0]
   ->{'Dices'}); }>.
```

Insgesamt erhält man:

```
 $report->ConfigureColumn
     (-header => 'Dices', -a => 'C',
      -value => sub {
             return $dices_to_image->
                ($_[0]->{'Dices'});
             }
     )
   if $report->IsFormat('HTML');
```

Für alle, die noch nicht so häufig mit dem `sub {}` Befehl gearbeitet haben: `$dices_to_image` ist eine Referenz auf die aufzurufende Funktion, und die Variable wird automatisch vom erzeugenden Code in die anonyme `sub{}` importiert.

`$_[0]` ist die erste Variable aus der Argumentenliste, die für die `-value`-Option immer mit dem auszugebenden Datensatz belegt ist, hier also mit der Wurfnummer und den erzielt(en) Wert(en) des Würfel-Wurfs: `$_[0]->{'Dices'}` enthält eine Abfolge gewürfelter Werte als String.

Das nachgestellte `if` sorgt dafür, dass diese zusätzliche Spalte nur erzeugt wird, falls es sich um einen HTML-Report handelt.

`$dices_to_image` wird folgendermaßen mit einer Sub belegt:

```
my $dices_to_image = sub {
my $throws = shift; # Dice throw values

my $result = '';
my $number_html_code;

foreach my $number (split (//, $throws)) {
  if ($number =~ /^\d$/) {
    $number_html_code =
      „<img src='dice_$number.jpg'/>"
  }
  else {
    $number_html_code = „?";
  }
  $result .= $number_html_code;
}

return $result;
};
```

Bis auf die erste Zeile ist das eine Übungsaufgabe für den Perl-Fortgeschrittenen-Kurs. Also eine kleine Fingerübung für erfahrene Perl-Programmierer. Zurück liefert diese Sub eine Abfolge von `<img ...>`-HTML-Befehlen, die im Browser die Bilder mit den Würfelaugen in der gewünschten Reihenfolge ausgibt.

Ein direkter Aufruf einer „normalen" Sub wäre auch möglich, aber durch die Verwendung einer anonymen Sub geht man allen Problemen bei der Namensauflösung aus dem Weg, die andernfalls entstehen könnten.

Eigenschaften

Im Folgenden möchte ich noch die wesentlichen Eigenschaften erläutern, die ich bei anderen Report-Modulen so nicht gefunden habe:

Offenheit

Es ist leicht möglich, eigene Varianten von Report-Konfiguratoren zu erstellen oder neue Formate zu unterstützen. Auch die gleichzeitige Verwendung von verschiedenen Konfiguratoren für dasselbe Format ist durch die Möglichkeit, mehrere Framework-Instanzen zu verwenden, sehr einfach zu realisieren.

Unabhängigkeit von anderen Modulen

Bewusst verwendet (Standard)-PORF z.B. keine HTML-Funktionen aus CPAN-Modulen, damit das System wirklich offen bleibt. Der Anwender kann selbst entscheiden, welche Module er in eigenen/adaptierten Report-Konfiguratoren verwendet.

Performance

Auf meinem alten Win-XP Laptop mit 1 GByte Hauptspeicher, 800 MHz AMD 64 Bit Single-Core und Perl 5.14.2 kann man zwischen 10.000 und 100.000 Tabellen-Zeilen pro Sekunde in eine Datei auf die Festplatte schreiben.

Zweck und Eingrenzung der Funktionalität

PORF wurde entwickelt, um Massendaten leicht, komfortabel, flexibel und schnell ausgeben zu können, bevorzugt in Listenform. Die Konfiguration soll generisch und Perl-gemäß funktionieren.

Datenbeschaffung und -Verarbeitung sind nicht Bestandteil des Frameworks und sollen es auch nicht sein. PORF kümmert sich nur um Formatierung und Ausgabe, ist also ein komfortables Ausgabemodul.

Diagramme und längerer Text können von anderen Tools besser bereitgestellt werden und sind daher auch kein Bestandteil von PORF. Aber da man den vollen Durchgriff auf die Reports hat, kann man Teile (Tabellen) erstellen, die man in anderen Dokumenten verwenden/importieren kann.

Wie das genau funktioniert und wie man seine eigenen Konfiguratoren erstellen kann, werde ich in weiteren Artikeln beschreiben, falls ihr daran Interesse habt.

Ausblick

Zur Zeit werden nur einfache Listen unterstützt. Mehrzeilige Ausgaben, mehrzeiliger Aufbau mit Verbundzellen, Visitenkarten-Layouts bis hin zur freien Platzierung von Werten auf der Seite sind angedacht.

Aber in der Erstellung einer einfachen, konsistenten API steckt noch viel Arbeit. Wer in irgendeiner Weise mitarbeiten möchte, ist dazu herzlich eingeladen.

Weitere Konfiguratoren für CSV, Wikis, LaTeX sind in Vorbereitung.

Have Fun Using PORF

Ich würde mich freuen, von euren Erfahrungen mit PORF zu hören und wünsche euch viel Spaß mit Perl :)).

Herbert Breunung

Vorsicht Experiment!

Über die neuen Fehlermeldungen in Perl 5.18

Es ist immer wieder ein Fest zu sehen, wenn ein neues Perl aus *Repo* rollt. Neue Funktionen, entfernte Ärgernisse, bessere Dokumentation, mehr Sicherheit und Konsistenz oder weniger Speicherverbrauch - jedes Jahr ein Grund zur Freude. Außer, wenn ein Programm aufhört zu arbeiten, weil sich etwas geändert hat. Dass neue Funktionalitäten nicht stören, wird größtenteils ab 5.10 mit dem `feature`-Pragma erreicht, welches ab 5.16 `feature` auch wieder abschalten kann, um genau den Stand der Zeile für Zeile aktuell gewünschten Version zu simulieren. Was aber, wenn Funktionen in ihrem Verhalten angepasst werden müssen? Dann kann das Programm trotz `use feature ...;` aus den Gleisen springen.

Meist besaßen bereits vorher diese Funktionen den Status „experimentell". Mit anderen Worten: die Perl-Porter rechnen damit, daß die Funktion sich ändern könnte oder wieder abgeschafft wird. Lange Zeit war es aber nur Teilnehmern der *p5p*-Liste und aufmerksamen Lesern sämtlicher `perldelta`s bekannt, welche Funktionen experimentell sind. Brian d Foy sah das Problem und fasste ab 2010 auf der `perldoc`-Seite `perlexperiment` [1] jene `feature` zusammen, die nur auf Probe Bestandteil des Kernes sind. Aber wer liest schon neue Dokumentation, die tief im Index aufgelistet ist? Spätestens nach der dritten Änderung des Smartmatch Operators `~~`, auf die höchstwahrscheinlich noch eine vierte folgen wird, erstarkte die Forderung, diese Informationen den Nutzern ins Gesicht zu halten und sich die Benutzung der experimentellen Dinge quasi vom Programmierer quitieren zu lassen. Ab 5.18 braucht es ein `no warnings „experimental::feature_name"` um die ansonsten auftauchenden Warnhinweise zu unterdrücken. Wer nur `use v5.18;` schreibt, schwimmt im ruhigeren Fahrwasser der garantierten Funktionen und bekommt es sehr genau angezeigt wo er den Sicherheitsbereich verlässt:

```
Smartmatch is experimental at
   script.pl line 12.
```

Diese erhöhte Sicherheit vor erzwungenen Anpassungen an neue Versionen ist zweifellos positiv. Warum also die „Vorsicht" in der Überschrift?

Die laute Gefahr

Zum ersten weiß nicht jeder, dass `when` ebenfalls `~~` benutzt und so bestehende Programme plötzlich „unverständliche" Hinweise ausgeben.

```
given is experimental at script.pl line 11.
when is experimental at script.pl line 12.
```

Wer nun nach einem Blick in die perldoc schreibt:

```
no warnings „experimental::given";
no warnings „experimental::when";
```

bekommt zum Dank:

```
Unknown warnings category
   'experimental::given' at script.pl line 5.
BEGIN failed--compilation aborted at
   script.pl line 5.
```

Tatsächlich wäre die nicht ganz intuitive Lösung für beide Vorkommen:

```
no warnings „experimental::smartmatch";
```

Aber `given` birgt noch eine andere Gefahr. Es setzt den Inhalt der lokalen (`local $_`) und nicht mehr der lexikalischen Kontextvariable (`my $_`). Mit 5.18 wurden alle Befehle und

Kernmodule derart umgestellt, weil letzteres seit langem und immer noch experimentell ist. Das bedeutet, dass jetzt jegliches `my $_` Perl provoziert.

```
Use of my $_ is experimental at -e line 1.
```

Zur Beruhigung hilft:

```
no warnings „experimental::lexical_topic";
```

Für das neue `my sub { ... }` nehme man die Konstante `experimental::lexical_subs` und die frisch eingeführten Operatoren in einer Regex sind `experimental::regex_sets`. Alle vier Funktionen lassen sich zusammen kürzer ankündigen:

```
no warnings „experimental";
```

Die stille Gefahr

Es kann ja vorkommen, dass die Umstellung zu mehr Sicherheit zeitweilige Unsicherheiten beim Nutzer weckt. Aber dies wird leider ein Thema für die nächsten Releases bleiben. Nicht unbedingt, weil immer mehr experimentelle Funktionen hinzu kommen und hoffentlich auch einige ihre „ewigen Weihen" erhalten. Stattdessen gibt es derzeit noch einige offiziell experimentelle Funktionen, welche ohne Verwarnung benutzt werden können.

Das sind zum Beispiel `push` und andere Array- und Hashbefehle, welche auch Referenzen als ersten Parameter akzeptieren.

Das gilt auch für die *Special Backtracking Control Verbs* wie `(*ACCEPT)`, `(*COMMIT)` > und weitere [2].

Ebenfalls erlauben es `(?{ code })` und `(??{ code })` weiterhin, ungestört Code in einer Regex auszuführen. Ersteres gibt dann `$^R` zurück, das zweite wird evaluiert. Das zu ändern müsste doch möglich sein, da die neuen Mengenoperatoren für Zeichenklassen (`& + | - ^ !`) auch Bestandteile von Regulären Ausdrücken sind, jedoch bei Benutzung brav ihren experimentellen Status melden.

Conclusio

`no warnings „experimental::...";` ist wichtiger und nützlicher Befehl, ist aber noch dabei seine Regeln zu finden und etablieren.

Links

[1] http://perldoc.perl.org/perlexperiment.html
[2] http://metacpan.org/module/
perlre#Special-Backtracking-Control-Verbs

Herbert Breunung

Rezensionen - Einsteigerliteratur

Michael Fitzgerald
Einstieg in Reguläre Ausdrücke
O'Reilly, 156 Seiten
1.Auflage Dez. 2012
ISBN 978-3-86899-940-2
Gebunden: €19,90
PDF, EPUB: €16,00

Michael Mangelsdorf
Web-Entwicklung mit Perl und Mojolicious
Books on Demand, 97 Seiten
2. Auflage April 2012
ISBN 978-3-8482-0095-5
Broschürt: €8,20
Kindle: €6,49

Gabor Szabo
Perl Maven Pro
http://perlmaven.com/archive?tag=pro
11 Artikel, ständig aktualisiert
HTML: $9/Monat, $90/Jahr

Gerade wenn bekannte Leute ein Kinderbuch schreiben, gehe ich in Deckung. Denn eine einfache Sprache und ein paar nicht alltägliche Gestalten machen noch keine gute Lektüre für junge Leser. Da braucht es echtes Einfühlen, viel Phantasie und ausdauernde Arbeit am Text. Mit den Programmierbüchern für werdende Codeakrobaten ist es ähnlich. Nur, wer die Materie theoretisch und praktisch beherrscht, kann locker und einfach über Dinge schreiben, mit denen sich die Leser oft schwerer tun als vermutet. Und nur wer ein weites Feld überblickt, kann die wesentlichen Informationsbrocken auswählen, um aus ihnen einen leicht passierbaren Weg zu pflastern, der zu guter Software und einem tieferen Verständnis führt.

Wir haben das Glück, mit der *Einführung in Perl* (im Original „Learning Perl" - Rezension in Ausgabe 3/2011) von Randal L. Schwartz und Brian d Foy, *Beginning Perl* (4/2012) von Curtis „Ovid" Poe und *Modern Perl* von cromatic (2/2011) ein recht hohes Niveau gewohnt zu sein. Ob diese Liste erweitert werden kann, soll die entscheidende Frage der heutigen Folge werden.

Als Drittes wählte ich Gabors kostenpflichtigen Pro-Artikel, da der angekündigte „Beginner"-Titel noch wächst. Das bringt nicht nur Abwechslung, sondern kündigt auch an, dass nun ebenfalls rezensionswürdige Netzinhalte besprochen werden. (Die *Perl-Snapshots* der letzten Nummer könnte man schon dazu zählen.) Im Web lassen sich immer wieder schöne Dinge finden, wie etwa *Exploring Programming Language Architecture in Perl* von Bill Hails (*http://billhails.net/Book*), eine Anleitung zum Interpreterbau in Perl. Ebenfalls im Web unter *http://paperc.de* lassen sich alle Titel von *O'Reilly*, *d-Punkt*, *Hanser-Verlag*, *PACKT Publishing* und *No Starch Press* legal und kostenlos eine Stunde lang lesen, oder nur kapitelweise kaufen. Auch das nun folgende Buch. Denn nach der Trilogie über Software-Management, kommt als nächstes Perl-nahes Thema für die nächsten Ausgaben: *Reguläre Ausdrücke*.

Einstieg in Reguläre Ausdrücke

Es gibt ja wirklich Leute, die fürchten sich vor einer Regex, als wenn sie nicht verstandene Sonderzeichen in schaurige

Tiefen ziehen würden. Ihnen ein knöcheltiefes Planschbecken einzurichten, ist sehr begrüßenswert, da Regex einfach zu nützlich sind, um dauerhaft ohne sie auszukommen. Eigentlich ist es wie viele andere Becken schräg und kann zum anderen Ende hin bis zur Hüfte reichen, da die *Lookarounds* (*Lookahead* und *Lookbehind*) nicht zum Einfachsten gehören. Auch aufblasbare Armreifen in Form von graphischen Regex-Testprogrammen wie *RegexPal* werden dem Leser zu Beginn angelegt. Jedoch nicht ohne vorher kurz auf die Geschichte einzugehen, was im Standardwerk von Jeffrey Friedl leider übergangen wurde.

Michael Fitzgeralds pädagogische Methode scheint es zu sein, den Schüler nasszuspritzen, damit er merkt, dass Wasser nicht weh tut. Denn auch wenn das Niveau sich nur behutsam steigert und es für keinen Programmierer schwer sein sollte einzusehen, dass `[0-9]` für eine beliebige Ziffer steht, hätte mancher gerne vorher gewusst, wofür die eckigen Klammern und der Bindestrich stehen. Doch spätestens, wenn er das Beispiel eingibt und farbig hervorgehoben sieht, was dadurch gefunden wird und was nicht, sollten die Zweifel ausgeräumt sein.

Die Sprache ist unauffällig, sachlich, aber nicht steif, mit stellenweise originellen Bildern. Die Texte sind reduziert und beschränken sich wirklich auf das Wesentliche. So sind die Kapitel sieben und acht bereits nach sechs kurzen Seiten vorüber. Dennoch wird alles besprochen, was man so für den Anfang braucht. Zahlen, Strings, Grenzen, Alternationen, Gruppen, Rückwärtsverweise, Zeichenklassen, Unicode, Quantoren und Lookarounds gehören zum Programm. Einige größere Beispiele in `sed` und *Perl* schließen die 104 Seiten Unterricht bereits ab. Ausgiebige Tabellen, Index, Glossar und Hinweise zu weiterführenden Informationen hinterlassen das Gefühl eines sorgfältig ausgearbeiteten Buches. Das Einzige, was sich noch verbessern ließe, sind die recht bieder gewählten Beispiele. Fast nirgends findet sich ein: „Cool, dieses scheinbar aussichtslose, aber mich wirklich zwickende Problem kann ich jetzt mit einer Regex lösen." Aber selbst wenn der Verdacht, dass der Autor vor Beginn der Ausarbeitung kein Großmeister der Zeichenklassen war, stimmen sollte: Dieses Werk liefert, was es verspricht - gewandt und ohne ein sichtbares Zeichen der Schwäche. Damit gehört es qualitativ zum obersten Fünftel.

Web-Entwicklung mit Perl und Mojolicious

Michael Mangelsdorf schrieb leider für die gegenüber liegende Schublade. Er mag ein hervorragender Übersetzer sein mit einer guten Ausbildung und einer beeindruckenden Aufzählung von Kunden. Aber selbst für €6,50 darf es mehr sein als ein virtuelles „Händchen halten" mit den tröstenden Worten, dass dies alles nicht so schwer sei. Dabei hat er zwischendurch wunderbar klare Sätze, welche lässig den Kern des aktuellen Sachverhaltes ausdrücken. Aber ein Grundkurs für *HTML*, *CSS* (3 Seiten + Listings), *Javascript*, *ORLite*, *Perl* und Mojolicious auf unter hundert mittelgroßen Seiten? Er erklärt einfach zu wenig, um hilfreich zu sein, sondern beschreibt nur, wie er seine eigene Seite mit *Mojolicious* erstellt hat. Mit einer Hand voll Tags wird HTML noch lange nicht mein Freund. Und wer Perlbeispiele ohne `use strict;` und für Geld 2012 in Umlauf bringt, sollte dafür nicht unter drei Monaten seine Seite mit Original-CGI-Skripten von Matt Wright (1997) betreiben müssen.

Der Grundgedanke für dieses Buch ist sehr gut. Mojolicious pflegte den Slogan *The web in a box*, weil es wie PHP alles von sich aus mitbringt, um dynamische Netzseiten schnell und einfach aufzuziehen. Das lockt natürlich Menschen an, die sich nie zuvor für Perl interessiert haben. Ihnen locker die wesentlichen Bits zu zeigen, die man dabei anfasst (Routing, Datenbank, Templates), wäre ein Geschenk für die ganze Perlgemeinde. Doch wer mit zu wenig Inhalt und zu vielen inhaltlichen und Rechtschreibfehlern auf halbem Wege aufhört, ist wohl doch eher daran interessiert, eine schnelle Mark oder Werbung für die Adresse *http://ok-schalter.de/mojobuch* zu machen (oder beides). Unter besagter URL tummeln sich eine Reihe weiterer, scheinbar ähnlich schnell erstellter „Bücher". Am Ende der Einleitung stehen lediglich 5 Links: *Eclipse*, der 2010 eingestellte Eclipse-Editor *EPIC*, *Mojo*, *SQL-Studio* und *Active State*. Weder eine Art Dokumentation, *Strawberry-Perl*, noch sinnvolle Werkzeuge zum schnellen Überprüfen der Perlbeispiele wie *http://www.compileonline.com/execute_perl_online.php* werden empfohlen.

Perl Maven

Gabor Szabo zeigt, wie man es mit der Eigenwerbung richtig macht. Natürlich hilft es, wenn man ein in der Gemeinde bekannter Trainer, Blogger, Konferenzorganisator und Autor ist, aber das hat er sich auch erarbeitet. Dafür hat er sogar schon den *White-Camel-Award* für besondere nichttechnische Leistungen bekommen. Vor über zwei Jahren begann er über die Mailingliste *http://perlweekly.com* die wesentlichen Posts und Ereignisse der Woche zusammenzufassen. Er nutzte sie gleichzeitig um seine Trainingsklassen anzubieten und um sein in kleinen Schnipseln entstehendes Perl-Tutorial bekannter zu machen, welches aus unregelmäßigen Blog-Posts auf *http://szabgab.com/blog.html* entstand. Es ist beileibe nicht vollständig und wächst kreuz und quer al gusto. Aber es ist lehrreich, beinhaltet aktuelles Perl und Gabors langjährige Erfahrung als Tutor und Programmierer. Zudem vermochte er Menschen dazu bewegen es in etliche Sprachen zu übersetzen. Daraus entstand die Seite *Perl Maven*, die auch Screencasts und als „TV-Show" Interviews mit bekannten Perl-Entwicklern enthält. So etwas gab es seit 2010 nicht mehr, als der *http://perlcast.com* von Josh McAdams eingestellt wurde. Eine kleinere Schwesterseite (*http://perl6maven.com/archive*) befasst sich sogar mit Perl 6.

Doch zurück zum Perl 5-Maven. Seit Juni erscheinen dort ein- bis dreimal die Woche um ein vielfaches längere, sogfältig ausgearbeitete Artikel, welche nur für eine Gebühr sichtbar werden. Zum Beispiel das Stück über die *autovivification* gefiel mir sehr gut, weil dort schrittweise das teilweise heimtückische Verhalten der Arrays und Hashes mit Beispielen und zugehöriger Ausgabe seziert wurde. Zugegeben, die Sprache und der Humor wirkt manchmal spröde und schlicht, aber es ist didaktisch gut aufbereitet, die Beispiele passen und sind farbig. Praktisch sind auch die Tags unter der Überschrift. Sie sind gleichzeitig Knöpfe und rufen eine Übersicht mit ähnlichen Artikeln. Die Themen sind meist aktuell (derzeit behandeln einige Moo) und immer praktisch relevant. Hoffentlich werden die Artikel noch Einigen die 90 Eurocent im Schnitt wert sein, ist es ja auch ein bequemer Weg, die anderen guten Aktivitäten des Herrn Szabo zu unterstützen.

Ausblick

Nächstes mal wird einfach weitergemacht. Noch ein Buch über *Reguläre Ausdrücke* (von Jeffrey E. Friedl), noch eines über *Webprogrammierung mit Perl* von Simon A. Frank und noch ein Blick auf das, was Gabor so schreibt.

Sören Kornetzki, Wolfgang Schemmel

Hannover.pm: Aller guten Dinge sind drei

Der Ursprung von Hannover.pm lässt sich ziemlich genau zurück verfolgen. Er liegt an einem Freitag Nachmittag in einem Hörsaal der Fachhochschule München. Es war der letzte Tag des Deutschen Perl-Workshops 2007, der 23. Februar. Sven Neuhaus und Wolfgang Schemmel (Perleone) gründeten voller Workshop-Motivation die Hannover Perl Mongers. Das erste und einzige Treffen fand bereits sechs Tage später am ersten März statt, mit Spezialgast Jonathan Worthington. Sogar die $foo Nr. 2 erwähnte es kurz in ihrem PM-Ticker [6]. Danach ließ der Enthusiasmus stark nach, Hannover.pm war einige Jahre scheintot.

Nach diesem Fehlstart rührte sich erst 2010 wieder etwas. Auf der CeBIT gab es einen Perl-Stand, an dem sich einige lokale Perl-User einfanden. Dort entstand auch die Idee, sich regelmäßig zu treffen. Sebastian Willing (sewi) übernahm daher Hannover.pm und hauchte der Gruppe neues Leben ein. Diesmal waren rund ein halbes Dutzend Leute mit den unterschiedlichsten Hintergründen bei den monatlichen Meetings dabei. Aber schon im nächsten Jahr gingen Interesse, Teilnahme und Häufigkeit der Treffen stark zurück. Im September 2011 fand das letzte Meeting der zweiten Ära statt.

Der erneute Winterschlaf war dieses Mal schon nach zwölf Monaten vorbei. Im Kielwasser der YAPC::EU 2012 in Frankfurt gab es wieder eine kritische Masse von Hannoveraner Perl-Usern, und Hannover.pm wurde Anfang September zum dritten Mal aus der Taufe gehoben. Gruppenleiter wurden Christian Walde (Mithaldu) und Sören Kornetzki (BURNERSK). Der harte Kern dieser Inkarnation hat einen deutlich höheren Perl- und Geek-Faktor als ihre beiden Vorläufer. Letztere ließen immerhin jeweils einen weiteren unverdrossenen Perl-Monger übrig, dem auch die langen Pausen nichts anhaben konnten: Hannover.pm-Urgestein Wolfgang Schemmel und den in der zweiten Instanz hinzugekommenen Olaf Schnath (Molaf). Neben den regelmäßigen Meetings zeugt auch die Organisation des Deutschen Perl-Workshops 2014 in Hannover von einer erfreulichen Konstanz und Langlebigkeit von Hannover.pm.

Weiterführende Links:

[1] Homepage:
http://hannover.pm/
[2] Doodle
http://www.doodle.com/
[3] 16. Deutscher Perl-Workshop
http://act.yapc.eu/gpw2014/
[4] Trello
https://trello.com/
[5] Kanban-Board
http://de.wikipedia.org/wiki/Kanban_%28Softwareentwicklung%29
[6] FrankfurtPM Vorstellung
http://perl-magazin.de/pm_gruppen/FrankfurtPM.pdf

CPAN NEWS

Type::Tiny

Eines der besten Features an Moose sind die Typen der Attribute. Bei der Definition von Attributen kann mit dem Parameter *isa* festgelegt werden, welche Art von Daten in dem Attribut gespeichert werden. Wird dann das Attribut gesetzt, wird eine Überprüfung ausgeführt, ob der neue Wert auch tatsächlich diesem Typen entspricht.

Außerhalb von Moose fehlt dieses Feature. Mit `Type::Tiny` gibt es ein Modul, das diese Lücke füllt (auf CPAN gibt es noch ein paar weitere Module für diesen Zweck). Somit ist es möglich, auch in anderen OO-Systemen wie `Moo` und/oder Standard-Perl5-OO Typen bei Attributen zu verwenden. Ein Vorteil von `Type::Tiny` gegenüber den anderen Modulen auf CPAN ist, dass es keine Abhängigkeiten außerhalb des Perl-Kerns hat. Dafür haben andere Module eine hübschere API.

```
use Type::Tiny;
use Types::Standard qw(Str);
use Type::Utils qw(enum);

my $PLZ = Type::Tiny->new(
    name       => 'PLZ',
    parent     => Str,
    constraint => sub {
        m{\A [0-9]{5} \z}xms
    },
    message  => 'PLZ ungültig',
);

my $Anrede = Type::Tiny->new(
    name       => 'Anrede',
    constraint => enum(
        'Anreden',
        [ qw/Herr Frau Firma/ ],
    ),
    message => 'Anrede kann nur Herr, '
        . 'Frau oder Firma sein',
);
```

Listing 1

Eigene Typen

Doch wie werden die Typen jetzt eingesetzt? Als Beispiel sollen ein paar Prüfungen bzgl. Adressen in Deutschland als Typen erstellt werden (siehe Listing 1).

- Postleitzahlen
- Anrede

Die Postleitzahl ist ein String bestehend aus fünf Ziffern, deshalb wird *Str* als Elterntyp angegeben. Dem Typ wird ein Name gegeben und beim *constraint* wird festgelegt wie die Prüfung aussieht. Über die Methode `check` können dann Werte geprüft werden, ob diese dem Typ entsprechen.

```
$PLZ->check( „01356" ) or
    die „1: „ . $PLZ->message;
$PLZ->check( „test" ) or
    die „2: „ . $PLZ->message;

$ perl plz.pl
2: PLZ ungültig at plz.pl line 17.
```

Das Deklarieren des Typs kann auch etwas einfacher gestaltet werden:

```
use Type::Utils qw(
    declare as where
);

my $PLZ = declare
    as Str,
    where { m{\A [0-9]{5} \z}xms };
```

... in eine Bibliothek

Bis jetzt wurden die Typen an der Stelle definiert, an der sie auch gebraucht wurden. Wenn man die Typen aber in mehreren Modulen benötigt, ist es praktisch, diese in ein eigenes Modul auszulagern:

```
package FooMagazin::Types;

use strict;
use warnings;

use Type::Library -base;
use Type::Tiny;
use Types::Standard -types;
use Type::Utils qw(
    declare as where
);

declare PLZ => (
    as Str,
    where { m{\A [0-9]{5} \z}xms },
);

1;
```

Über das *-base* wird eine Vererbungshierarchie geschaffen und macht `FooMagazin::Types` zu einer Subklasse von `Type::Library`. Das sorgt im Hintergrund dafür, dass die Typen dann in die einbindenden Klassen importiert werden können.

In der Klasse kann diese Bibliothek dann einfach eingebunden werden:

```
use FooMagazin::Types qw(PLZ);

has plz => (
    is  => 'rw',
    isa => PLZ,
);
```

Vorgefertigte Typen

Es gibt einige Typen, die im Standardgebrauch sehr häufig vorkommen wie Strings, numerische Werte, Objekte etc. Die Typen, die man am häufigsten braucht, werden von `Type::Tiny` schon in einer Bibliothek ausgeliefert: `Types::Standard`. Ein Beispiel wurde schon weiter oben gezeigt:

```
use Types::Standard qw(
    Bool Int ArrayRef Str
);

has 'answer_on_media' => (
    is  => 'ro',
    isa => Bool,
);

has timeout => (
    is  => 'ro',
    isa => Int,
);

has headers => (
    is  => 'ro',
    isa => ArrayRef[Str],
);
```

Typen im Einsatz

Die Typen nur zu definieren bringt ja nichts; sie müssen auch eingesetzt werden. An dieser Stelle soll kurz gezeigt werden, wie man die Typen in Moo- und in Standard-Objekten verwendet.

Moo

In Moo-Klassen können die Typen bei den Attributen über *isa* eingebunden werden:

```
use Moo;
use Type::Tiny;

my $PLZ = Type::Tiny->new( ... );

has plz => (
    is  => 'rw',
    isa => $PLZ,
);
```

Standard-Perl5-OO

In den Methoden muss man den Check selbst aufrufen und im Fehlerfall darauf reagieren.

```
sub new {
    return bless {}, shift;
}

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

    if ( @_ > 1 ) {
        if ( $PLZ->check( $plz ) ) {
            $self->{plz} = $plz;
        }
        else {
            die $PLZ->message;
        }
    }

   return $self->{plz};
}
```

Noch mehr ...

`Type::Tiny` setzt einen Großteil der Typen in `Moose` um. So bietet auch `Type::Tiny coercion`, also die Umwandlung von einem Typ in einen anderen. Auch `class_type` und `role_type` werden von `Type::Tiny` unterstützt.

Evented::Object

Für gleiche Dinge gelten hin und wieder unterschiedliche Regeln, die sehr individuell sein können. Bei dem einen Konto darf der Kunde bis zu 1000 EUR überziehen, bei einem anderen Konto darf der Kunde überhaupt nicht überziehen. Aber immer soll der Kunde benachrichtigt werden, wenn das Limit erreicht bzw. überschritten wurde oder es gibt mehrere Warnstufen.

Immer wenn eine solche Schwelle erreicht wurde, wird eine Nachricht ausgegeben. Das Erreichen einer solchen Schwelle ist ein Event und das Konto ist ein Objekt. Das Objekt muss also auf ein Event reagieren. Mit `Evented::Object` kann man Eventhandler an Objekte binden. Für das Konto könnte das so aussehen:

```
package Konto;

use parent 'Evented::Object';
use Moo;

has balance => (
    is => 'rw',
);

sub withdraw {
    my ($self, $amount) = @_;

    $self->balance(
        $self->balance - $amount
    );

    $self->fire(
        withdrawel => $self->balance
    );
}
```

Die Klasse erbt von `Evented::Object` und bekommt einige Methoden vererbt. Eine davon ist `fire`. Damit wird ein Event ausgelöst. In diesem Fall ist es ein Event mit dem Namen *withdrawel*. Als weitere Parameter bekommt `fire` die Werte übergeben, die dann der Methode übergeben werden, die auf das Event reagiert.

Jetzt gibt es also die Klasse *Konto*, die bei einer Geldauszahlung das Event *withdrawel* feuert. Das macht aber noch nicht allzuviel Sinn, da es jetzt noch Objekte braucht, die auf dieses Event reagieren.

```
use Konto;

my $account = Konto->new(
    balance => 500
);

$account->on(
    withdrawel => sub {
        my ($event, $balance) = @_;

        if ( $balance <= -1_000 ) {
            warn „Warnstufe 2 erreicht";
        }
        elsif ( $balance <= -500 ) {
            warn „Warnstufe 1 erreicht";
        }
    },
);
```

Mittels `on` wird ein Eventlistener installiert. Auch hier ist der erste Parameter der Name des Events. Als zweiter Parameter wird eine Subroutinenreferenz übergeben. Diese Sub wird dann aufgerufen wenn das dazu passende Event ausgelöst wird.

Man kann auch mehrere Eventlistener für ein Event definieren

```
my $account = Konto->new( balance => 500 )
  $account->on(
    withdrawel => sub {
        my ($event, $balance) = @_;

        if ( $balance <= -1_000 ) {
            warn „Warnstufe 2 erreicht";
        }
    }
);

$account->on(
    withdrawel => sub {
        my ($event, $balance) = @_;

        if ( $balance <= -500 ) {
            warn „Warnstufe 1 erreicht";
        }
    },
);
```

Jetzt werden bei jedem *withdrawel*-Event beide Subroutinen aufgerufen. Das ist aber kontraproduktiv, da bei einem Kontostand von -1.000 EUR beide Warnungen ausgegeben werden. Man kann dabei folgendermaßen vorgehen: Man vergibt Prioritäten und Namen an die Eventlistener.

```
$account->on(
    withdrawel => sub {
        # ...
    },
    name     => 'warn2',
    priority => 1,
);

$account->on(
    withdrawel => sub {
        # ...
    },
    name => 'warn1',
);
```

In diesem Fall hat der Eventlistener *warn2* eine höhere Priorität. Darin kann man dann andere Eventlistener ausschalten:

```
$account->on(
    withdrawel => sub {
        # ...
        if ( $balance <= -1_000 ) {
            $event->cancel('warn1');
        }
    },
    name     => 'warn2',
    priority => 1,
);
```

Wenn man nicht weiß, wie viele Eventlistener noch kommen und man keinen weiteren Eventlistener ausgeführt haben möchte, dann kann man das Eventhandling komplett stoppen:

```
$account->on(
    withdrawel => sub {
        # ...
        if ( $balance <= -1_000 ) {
            $event->cancel('warn1');
        }
    },
    name     => 'warn2',
    priority => 1,
);
```

Carp::Reply

Debugging ist eine der undankbarsten Aufgaben. Manchmal sucht man Stunden oder Tage nach einem Fehler und dann war es vielleicht nur ein kleiner Tippfehler. Aber dieser kleine Fehler bringt das Programm dazu abzubrechen. Ein erster Schritt in der Fehlersuche kann `Carp::Reply` sein. Das startet bei einem Abbruch (mittels `die`) eine REPL (read, evaluate, print loop), auf der man sich durch seinen Code navigieren kann. Ein kurzes Beispielskript:

```
use strict;
use warnings;

test($ARGV[0]);

sub test {
    die „need argument" if !$_[0];

    print $_[0];
}
```

Es ist hier ganz einfach zu überblicken, was geschieht. Übergibt man kein Argument, das zu *wahr* evaluiert, bricht das Programm ab. Jetzt ruft man das Programm mittels `perl -MCarp::Reply script.pl` auf und bei dem `die` wird die REPL gestartet:

```
$ perl -MCarp::Reply carp_reply.pl
need argument at carp_reply.pl line 9.
Now at carp_reply.pl:9 (frame 0)
Backtrace:
Trace begun at carp_reply.pl line 9
main::test(undef) called at
   carp_reply.pl line 6
> #bt
Backtrace:
Trace begun at carp_reply.pl line 9
main::test(undef) called at
   carp_reply.pl line 6
> #up
Now at carp_reply.pl:6 (frame 1)
> #env
HASH(0x238a570)
> #l
File carp_reply.pl:
  1: #!/usr/bin/perl
  2:
  3: use strict;
  4: use warnings;
  5:
* 6: test($ARGV[0]);
  7:
  8: sub test {
  9:     die „need argument" if !$_[0];
 10:
 11:     print $_[0];
>
```

Color::Scheme

Das Zusammenspiel von Farben ist wichtig. Es entscheidet darüber, ob eine Webseite oder ein Flyer gut aussehen oder „in den Augen weh tun". Doch zu wissen, welche Farben zusammenpassen, ist für ungeübte Personen nicht so einfach. Zum Glück gibt es `Color::Scheme`. Das Modul gibt die RGB-Werte der Farben aus. Wer erstmal etwas visuelles haben möchte, kann sich auf http://colorschemedesigner.com/ das Farbrad anschauen. Das Perl-Modul ist dann geeignet, wenn man Farbkombinationen automatisch ausgeben lassen möchte.

```
use Color::Scheme;

my $scheme = Color::Scheme->new
   ->from_hex('ff0000') # or ->from_hue(0)
   ->scheme('analog')
   ->distance(0.3)
   ->add_complement(1)
   ->variation('pastel')
   ->web_safe(1)

my @list = $scheme->colors();
# @list = (
     „999999","666699","ffffff","99cccc",
#    „999999","666699","ffffff","9999cc",
#    „669999","666699","ffffff","99cccc",
#    „cccccc","996666","ffffff","cccc99" )

my $set = $scheme->colorset();
# $set  = [
   [ „999999","666699","ffffff","99cccc", ],
#  [ „999999","666699","ffffff","9999cc", ],
#  [ „669999","666699","ffffff","99cccc", ],
#  [ „cccccc","996666","ffffff","cccc99"  ] ]
```

Term::Detect

Arbeitet man in einem Terminal, kann man sich Informationen über das Terminal holen. Dazu zählen Informationen wie die Unicode-Unterstützung und welches Terminal es ist:

```
perl -MTerm::Detect=
   detect_terminal
      -MData::Dumper -E '
      say Dumper \detect_terminal()'
$VAR1 = \{
          'unicode' => 0,
          'emulator_engine' => 'xterm',
          'default_bgcolor' => 'ffffff',
          'color_depth' => 256
        };
```

File::MMagic

Wenn man Dateien von anderen Personen bekommt und diese automatisiert verarbeitet, muss man prüfen, auf welche Art und Weise die Datei verarbeitet werden muss. Das kann je nach Dateityp anders sein. Zum Beispiel, wenn Informationen für ein System entweder per XML-, CSV- oder Excel-Datei kommen, muss die Datei jeweils anders behandelt werden. Ein Modul, mit dem man den Dateityp erraten kann ist `File::MMagic`:

```
use File::MMagic;
use FileHandle;

# use internal magic file
$mm = new File::MMagic;
# use external magic file
# $mm = File::MMagic->new('/etc/magic');
# if you use Debian
# $mm =
  File::MMagic->new('/usr/share/etc/magic');
$res = $mm->checktype_filename(
  „/somewhere/unknown/file");

$fh = new FileHandle
  „< /somewhere/unknown/file2";
$res = $mm->checktype_filehandle($fh);

$fh->read($data, 0x8564);
$res = $mm->checktype_contents($data);
```

NEWS

August 2013

01. Treffen Dresden.pm
06. Treffen Frankfurt.pm
Treffen Stuttgart.pm
10. Peking Perl-Workshop
12. Treffen Ruhr.pm
12.-14. YAPC:EU 2013
14. Treffen Niederrhein.pm
19. Treffen Erlangen.pm
21. Treffen Darmstadt.pm
24./25. FroSCon
28. Treffen Berlin.pm

September 2013

03. Treffen Frankfurt.pm
Treffen Stuttgart.pm
05. Treffen Dresden.pm
09. Treffen Ruhr.pm
11. Treffen Niederrhein.pm
16. Treffen Erlangen.pm
18. Treffen Darmstadt.pm
19.-21. YAPC::Asia
24. Treffen Bielefeld.pm
25. Treffen Berlin.pm

Oktober 2013

01. Treffen Frankfurt.pm
Treffen Stuttgart.pm
03. Treffen Dresden.pm
04.-05. OSDC.fr
09. Treffen Niederrhein.pm
14. Treffen Ruhr.pm
16. Treffen Darmstadt.pm
21. Treffen Erlangen.pm
29. Treffen Bielefeld.pm
30. Treffen Berlin.pm

Hier finden Sie alle Termine rund um Perl.

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

Uhrzeiten und weiterführende Links zu den einzelnen Veranstaltungen finden Sie unter

http://www.perlmongers.de

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

termine@foo-magazin.de

Perl-Services.de

Programmierung - Schulung - Perl-Magazin

info@perl-services.de