Ausgabe 1/2014 – www.foo-magazin.de - 6,00 €

$foo

PERL MAGAZIN

Email::Stuffer
Ich schicke mal eben schnell eine E-mail

LWP::UserAgent, SSL und Proxy

Locale::TextDomain
Internationalisierungs-Framework auswählen

Nr

29

16. Deutscher Perl-Workshop

26. bis 28. März 2014

Hannover
Kulturzentrum FAUST

http://act.yapc.eu/gpw2014/

VORWORT

Der Anfang vom Ende... und ein Neuanfang?

Das achte Jahr "$foo - Perl-Magazin" bricht mit der aktuellen Ausgabe an und es ist gleichzeitig das letzte Jahr! Die Ausgabe 32 (November 2014) wird die letzte Ausgabe des Magazins sein.

Was mit einem schrecklichen Design und als "Proof of Concept" Anfang 2007 startete, hat sich immer weiterentwickelt - nun blicken wir auf bisher sieben Jahre freudiger aber auch stressiger Arbeit zurück. Der für mich sehr hohe Zeitaufwand ist auch der Grund warum ich die Arbeiten am Magazin einstellen werde. Jede Ausgabe fünf bis sechs Artikel zu schreiben kostet unheimlich Zeit und bei komplett neuen Themen ist der Recherche- und Testaufwand sehr hoch.

Im privaten und beruflichen Umfeld hat sich bei mir in diesen Jahren einiges verändert, so dass ich dem Magazin nicht mehr gerecht werden kann. Ich möchte ja nicht einfach etwas runterschreiben nur damit ich die rund 50 Seiten irgendwie gefüllt bekomme ohne den Lesern irgendeinen Nutzen bieten zu können.

Ich habe es leider nicht geschafft, immer genügend Autoren zu animieren einen (oder mehrere) Artikel zu schreiben. Denjenigen, die es gewagt und immer wieder getan haben, möchte ich ganz herzlich danken. So konnte ich auch in meinem eigenen Magazin immer etwas dazulernen!

Auch den treuen Lesern möchte ich danken. Ohne euch hätte es ja keinen Sinn gemacht, sich die Mühen zu machen und alle drei Monate Artikel zu schreiben, Korrekturzulesen, Texte zu setzen, mit der Druckerei zu kommunizieren, die Magazine einzutüten und zu verschicken.

Die letzten Aufgaben hat größtenteils meine Frau erledigt. Danke!

Aber wie in der Überschrift angedeutet, gibt es die Chance auf einen gewissen Neuanfang. Schon länger liegen Pläne und Layout für ein englischsprachiges Magazin in der Schublade. Wenn sich so viele Autoren finden, dass ich selbst keine Artikel mehr schreiben muss -- und Herbert Breunung (der für fast jede Ausgabe von $foo etwas geschrieben hat) arbeitet daran -- wird es ein neues Perl-Magazin geben. Man darf gespannt sein, ob es klappt!

Ich selbst werde in unregelmäßigen Abständen einzelne Artikel auf http://perl-academy.de veröffentlichen. Bei Interesse einfach hin und wieder auf der Webseite vorbeischauen...

Jetzt aber auf in ein neues Jahr voller Perl und viel Spaß mit der 29. Ausgabe von $foo

Renée Bäcker

Die Codebeispiele können mit dem Code

lcm7ehs

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/issue29

IMPRESSUM

Herausgeber:	Perl-Services.de Renée Bäcker Bergfeldstr. 23 D - 64560 Riedstadt
Redaktion:	Renée Bäcker, Katrin Bäcker
Anzeigen:	Katrin Bäcker
Layout:	//SEIBERT/MEDIA
Auflage:	500 Exemplare
Druck:	powerdruck Druck- & VerlagsgesmbH Wienerstraße 116 A-2483 Ebreichsdorf
ISSN Print:	1864-7537
ISSN Online:	1864-7545
Feedback:	feedback@perl-magazin.de

INHALTSVERZEICHNIS

ALLGEMEINES

MODULE

ENTWICKLUNG

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 Workshops, Grant Manager bei der TPF und bei der Perl-Marketing-Gruppe.

Markus Benning

Markus Benning (30) ist seit 2001 als Unix/Linux Administrator und Softwareentwickler in Erlangen tätig. Seine Hauptaufgaben sind Aufbau, Betrieb und Automatisierung von Internet und Serverdiensten. In seiner Freizeit geht er zahlreichen Bergsportarten nach. Oft ist er nach Feierabend an einem der zahlreichen Felsen in der Fränkischen Schweiz anzutreffen. Außerdem Debian-Lover, Vim-Lover.

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.

Thomas Fahle

Perl-Programmierer und Sysadmin seit 1996.
Websites:
http://www.thomas-fahle.de
http://Perl-Suchmaschine.de
http://thomas-fahle.blogspot.com
http://Perl-HowTo.de

Colin Hotzky

Colin Hotzky studierte Informatik an den Universiäten in Leipzig und Liverpool.Schon während der Studienzeit kam er in Kontakt mit Perl. Für die Diplomarbeit nutzte er Perl zur Implementierung eines Suchverfahrens in irregulär strukturierten XML-Dateien. Heute arbeitet er als Service Manager bei der evosft GmbH für einen Middleware-Service im Eisenbahnsektor.

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.

Steffen Winkler

Seit 1960 gibt es mich. Ich programmiere Perl seit Ende 2000, erst privat und dann auch beruflich. Zur Zeit bin ich bei der Firma IT-Technik A. Specht beschäftigt. Dort arbeite ich vorwiegend im Bereich der Webprogrammierung, u.a. für die Jochen Schweizer GmbH in München. Den Deutschen Perl-Workshop besuche ich seit 2003, seit 2007 dann auch die YAPC Europe und seit 2012 den London-Perl-Workshop.

Wolfgang Kindeldei

Imager - eine universelle Bildverarbeitungs-Bibliothek

Motivation

Immer wieder ergibt sich die Problemstellung, Thumbnails zu erstellen, Bilder mit Wasserzeichen, einem Logo oder Text zu versehen oder gar Graphiken von Grund auf selbst zu erstellen. Greift man zum Klassiker ImageMagick? Oder gibt es Alternativen? Mein persönlicher Favorit ist `Imager`, eine schnelle und universelle Bibliothek zur Bildbearbeitung, die auf diverse in C geschriebene Bibliotheken zurückgreift.

Um `Imager` schmackhaft zu machen, möchte ich mit einem kurzen Benchmark beginnen. Eine JPEG-Datei aus einer 16 Megapixel Kamera soll zu einem Thumbnail mit einer Breite von 150 Pixeln konvertiert werden. Alle erzeugten JPEG Dateien wurden mit jeweils der maximal möglichen Qualität erzeugt, damit auch Unterschiede in puncto Qualität vergleichbar zu machen.

Angetreten sind ImageMagick als Kommandozeile (Convert), `Image::Magick`, `GD`, `Imager` und `Image::Epeg` (siehe Listing 1).

Schön, `Imager` ist etwa doppelt so schnell wie ImageMagick, unabhängig davon, ob die Kommandozeilen- oder Perl-Version von ImageMagick benutzt wird. Absoluter Gewinner ist `Image::Epeg`, das nochmal um Faktor 4 schneller ist. Leider kann letzteres Modul lediglich JPEG Dateien lesen und erzeugen. Außerdem kennt diese Bibliothek keinerlei Bildmanipulations-Befehle, scheidet daher bei vielen Anforderungen aus. Die Qualität der erzeugten Bilder ist ebenfalls die schlechteste der vier Test-Kandidaten.

Dass ImageMagick langsamer ist als Imager, ist ebenfalls erklärbar. Imager beherrscht keinerlei Farbmanagement, kann also nicht mit Farbprofilen umgehen und kann Metadaten aus Bildern weder lesen noch schreiben. ImageMagick kann neben vielen weiteren Dingen damit umgehen, was der Ausführungszeit leider nicht gerade entgegen kommt. Auch entschädigt die längere Laufzeit mit geringfügig schärferen Ergebnissen.

Installation

Etwas Vorsicht ist beim Installieren von Imager geboten, denn Imager sucht nach zahlreichen C-Bibliotheken und bindet alles Verfügbare ein. Je nach Zustand des Systems, auf dem Imager compiliert wird, fällt der Funktionsumfang daher unter Umständen ganz anders aus. Fehlermeldungen gibt es nicht, denn dieses Verhalten ist zunächst so beabsichtigt, um möglichst flexibel zu sein.

Wer Transparenz möchte, kann sich zum Beispiel so behelfen, allerdings ist das natürlich kein Weg für automatisierbare Installationen.

```
          Rate Convert  Magick      GD  Imager    Epeg
Convert 1.14/s      --     -6%    -25%    -52%    -91%
Magick  1.22/s      7%      --    -19%    -48%    -90%
GD      1.52/s     32%     24%      --    -36%    -88%
Imager  2.37/s    107%     94%     57%      --    -81%
Epeg    12.3/s    974%    905%    711%    418%      --
```

Listing 1

```
$ cpanm --interactive Imager
  ...
Libraries found:
  FT2
  JPEG
  PNG
  T1
  TIFF
Libraries *not* found:
  GIF
  Win32
OK
Building and testing Imager-0.97 ... OK
Successfully reinstalled Imager-0.97
1 distribution installed
```

In diesem Fall kann Imager auf einem *nix System nicht mit GIF Dateien umgehen, da die dafür benötigte Bibliothek nicht gefunden wurde. Die zusätzliche Distribution `Imager::File::GIF` wird in diesem Fall einfach nicht mit installiert.

Wenn ein Projekt bestimmte Dateiformate benötigt, kann die zusätzliche Installation der diversen zusätzlichen Distributionen `Imager::File::*` oder `Imager::Font::*` vornehmen. Laufen die Tests der Installation erfolgreich durch, ist alles im grünen Bereich. Fehlschlagende Tests lassen dann relativ schnell Rückschlüsse auf fehlende oder inkompatible C-Bibliotheken zu.

```
$ cpanm Imager::File::GIF
--> Working on Imager::File::GIF
Fetching .../authors/id/T/TO/TONYC/
   Imager-File-GIF-0.88.tar.gz ... OK
Configuring Imager-File-GIF-0.88 ... N/A
! Configure failed for Imager-File-GIF-0.88.
  See .cpanm/build.log for details.
```

Funktionsumfang

Um einen Teil der Möglichkeiten von Imager kennen zu lernen, schauen wir uns ein Beispiel-Programm an. Aus einem vorgegebenen Bild wird eine neue Graphik erzeugt, die ein Abspiel-Symbol in der Mitte sowie ein Logo in einer Ecke besitzt. Eine Copyright Information in einer weiteren Ecke wird ebenfalls mit gezeichnet. Um erweiterbar zu sein und den Programmcode leichter nachvollziehbar zu halten, erzeugen wir eine einfache Klasse (Thumbnail), in der die verwendeten Operationen sowie ein Imager-Objekt gekapselt sind. Die Erzeugung unserer Graphik sieht dann so aus:

```
Thumbnail
  ->load($source_file)
  ->scale(THUMBNAIL_WIDTH, THUMBNAIL_HEIGHT)
  ->add_play_icon(PLAY_ICON_RADIUS)
  ->add_logo($logo_file)
  ->add_copyright
  ->save($thumbnail_file);
```

Die einzelnen Operationen sind relativ schnell abgehandelt und hoffentlich weitgehend selbstsprechend. Die von uns eingesetzte Klasse besitzt lediglich ein Attribut (image), in dem das Imager-Bild im jeweils aktuellen Zustand gehalten wird.

```
package Thumbnail;
use Moose;
use Imager;

has image => (
    is  => 'rw',
      isa => 'Imager',
);
```

Bild Datei lesen

```
sub load {
    my ($class, $file) = @_;

    my $self = $class->new;

    $self->image(
       Imager->new(file => $file));

    return $self;
}
```

Skalieren und in korrekte Größe bringen

Die Skalierung erfordert zwei Arbeitsschritte, denn die reine Skalierung behält das Seitenverhältnis des Bildes bei. Wir müssen also das skalierte Bild noch zuschneiden.

```
sub scale {
    my ($self, $width, $height) = @_;

    # skalieren -- erzeugtes Bild
    # ist zu groß
    my $image = $self->image->scale(
        xpixels => $width,
        ypixels => $height,
        type    => 'max',
        qtype   => 'mixing',
    );

    # zuschneiden -- Mitte ist Fixpunkt
    my $left =
     int(($image->getwidth  - $width)  / 2);
    my $top  =
     int(($image->getheight - $height) / 2);
    $image = $image->crop(
        left   => $left,
        right  => $left + $width,
        top    => $top,
        bottom => $top + $height,
    );

    $self->image($image);

    return $self;
}
```

Zeichnen eines einfachen Abspiel-Pfeils

```
sub add_play_icon {
    my ($self, $radius) = @_;

    my $center_x =
      int($self->image->getwidth  / 2);
    my $center_y =
      int($self->image->getheight / 2);

    # große schwarze Fläche
    $self->image->circle(
        color => '#000000',
        r     => $radius + 2,
        x     => $center_x,
        y     => $center_y,
        aa    => 1,
    );

    # etwas kleinere weiße Fläche
    $self->image->circle(
        color => '#ffffff',
        r     => $radius,
        x     => $center_x,
        y     => $center_y,
        aa    => 1,
    );

    # schwarzes Dreieck
    my $delta = ($radius - 2) / sqrt(2);
    $self->image->polygon(
        color  => '#000000',
        points => [
            [ $center_x - $delta,
                $center_y - $delta],
            [ $center_x + $radius - 2,
                $center_y ],
            [ $center_x - $delta,
                $center_y + $delta],
        ],
        aa     => 1,
    );

    return $self;
}
```

Hinzufügen eines Logos aus anderer Bild-Datei

```
sub add_logo {
    my ($self, $file) = @_;

    my $logo = Imager->new(file => $file);
    $self->image->paste(
        left => $self->image->getwidth
              - $logo->getwidth  - 1,
        top  => $self->image->getheight
              - $logo->getheight - 1,
        src  => $logo,
    );

    return $self;
}
```

Text zeichnen

```
sub add_copyright {
    my $self = shift;

    my $text = 'WKI';
    my $font = Imager::Font->new(
        file => "$FindBin::Bin/Twister.ttf",
    );

    $self->image->string(
        string => $text,
        x      => 5,
        y      => $self->image->
                    getheight - 10,
        color  => '#ffffff',
        font   => $font,
        size   => 20,
        aa     => 1,
    );

    return $self;
}
```

speichern des erzeugten Bildes

```
sub save {
    my ($self, $file) = @_;

    $self->image->write(
        file => $file,
    );

    return $self;
}
```

Zugabe

Imager kann noch viel mehr. Dank der brillanten Dokumentation sind die notwendigen Methoden und deren Argumente auch schnell herauszufinden. Nicht unerwähnt bleiben sollten die Eigenschaften dennoch:

• Füll-Funktionen
gezeichnete Flächen lassen sich nicht nur mit soliden Farben füllen, sondern auch mit Mustern, Schraffuren, Verläufen oder gekachelten Bildern.

• Transformationen
neben der gezeigten Skalierung stehen auch Streckungen, Rotationen und Spiegelungen zur Verfügung.

• Farb-Manipulationen
Diverse Operationen bieten Farb-Operationen, Helligkeits- und Kontrastveränderungen sowie diverse arithmetische Veränderungen beim Zusammenfügen verschiedener Einzelteile.

• Transparenzen
Imager kann wahlweise einen Alpha-Kanel mitführen, der beim Erzeugen von zum Beispiel PNG- oder GIF-Bildern transparente Bilder produzieren kann.

• Anti-Aliasing
fast jeder Zeichenbefehl beherrscht die Option `aa`, mit der die Kanten der gezeichneten Objekte geglättet werden, was optisch ansprechendere Ergebnisse zur Folge hat.

• Filter
Fast schon Photoshop-Niveau hat die Liste der zur Verfügung stehenden Filter Funktionen. Damit stehen eine ganze Reihe von Verwandlungs- und Verfremdungs-Befehle zur Auswahl.

Letzte Zugabe

Zahlreiche CPAN Autoren haben Imager ebenfalls in ihr Herz geschlossen und diverse Distributionen auf Imager aufgebaut. Hier eine kleine Auswahl:

• Imager::Montage
Erlaubt die Erzeugung einer Komposition aus einer Menge von Einzelbildern.

• Imager::Graph
Erzeugt Balken-, Torten- und Liniengraphiken mit sehr viel Variations-Möglichkeiten.

• Imager::Simple
Wie der Name vermuten lässt, erleichtert dieses Modul einige Arbeiten rund um Imager.

• Imager::Heatmap
Erzeugt Heatmaps aus statistischen Daten.

• Imager::QRCode
Erleichtert die Erzeugung von QR Codes.

Fazit

Wer auf Farbmanagement verzichten kann und auf bestimmte Dateiformate (z.B. EPS oder PDF) keinen Wert legt, hat mit Imager eine schnelle und vielseitige Bildverarbeitungs-Bibliothek, die ohne Zicken auch automatisiert installierbar ist und mit umfangreicher Dokumentation geliefert wird. Der Funktionsumfang ist ebenso überzeugend wie die zahlreiche zusätzlichen auf CPAN befindlichen auf Imager aufbauenden Distributionen.

Renée Bäcker

Interaktives Debuggen von Regulären Ausdrücken

Für viele Personen sind Reguläre Ausdrücke ein Buch mit sieben Siegeln. Das muss nicht sein und man sollte auch darauf achten, dass man es seinen Kollegen und Kolleginnen nicht unnötig schwer macht. So kann man mit einfachen Mitteln die Regulären Ausdrücke lesbarer machen -- z.B. in dem man Kommentare einfügt (dank dem x-Modifier) oder man auf Module wie `Regexp::Common` ausweicht:

```
# matchen einer IPv4-Adresse
$var =~ m/
   ([0-9]{1,3} \.) # erstes Oktett
   ([0-9]{1,3} \.) # zweites Oktett
   ([0-9]{1,3} \.) # drittes Oktett
   ([0-9]{1,3})    # letztes Oktett
/x;

use Regexp::Common;
# matcht integer zahlen
$var =~ m/$RE{num}{-int}/;
```

Aber manchmal kommt man nicht umhin, einen vielleicht auch komplexeren Regulären Ausdruck zu schreiben oder man hat einen solchen irgendwo im Code gefunden. Es soll auch vorkommen, dass Software nicht ganz fehlerfrei ist und bei komplexeren Regulären Ausdrücken soll das auch durchaus mal vorkommen. Was also tun? Am besten man besorgt sich ein neueres Perl. So ganz taufrisch muss es auch nicht mehr sein, denn es reicht ein Perl 5.10 (am besten jetzt die Installation starten wenn noch nicht vorhanden).

Mit Perl 5.10 hat sich sehr viel beim Thema "Reguläre Ausdrücke" getan (siehe auch $foo Ausgabe 5 - "Neue RegEx-Features in Perl 5.10"). Durch diese ganzen Änderungen haben sich auch viele Möglichkeiten ergeben. Eine davon wird hier in diesem Artikel gezeigt.

Es geht jetzt also darum, wie man sich in einer unbekannten RegEx zurecht finden oder einen Fehler in einer RegEx finden kann. "Debugging" gehört normalerweise nicht gerade zu den Lieblingsaufgaben eines Programmierers, doch manchmal ist es unerlässlich. Für Reguläre Ausdrücke gibt es einen eingebauten Debugger, der sich über `-Mre=debug` einschalten lässt:

```
perl -Mre=debug -e '"foobar" =~ /regex/'
```

Wenn statt *regex* ein *(.)b* steht, bekommt man eine ganze Seite mit Debug-Informationen (Listing 1).

Für den Einsteiger nicht so ganz übersichtliche und wichtige Informationen gehen in der Fülle der Daten einfach unter. Das Debugging mit dem eingebauten Regex-Debugger wurde auch schon in Ausgabe 15 "Regex Debugging" näher beschrieben. Hier soll ein neuerer Debugger gezeigt werden, mit dem man besser sehen kann was in einem Regex so alles passiert.

Ein Perl 5.10 sollte mittlerweile (vielleicht mit Hilfe von perlbrew oder plenv) installiert sein. Wie gesagt, gibt es mit der neu geschriebenen Regex-Engine wesentlich mehr Möglichkeiten. Eine davon hat Damian Conway genutzt und den Debugger `Regexp::Debugger` geschrieben.

Wer das Modul installiert hat, bekommt das Tool `rxrx` mitgeliefert. Damit gleich mal das Beispiel von oben probieren:

```
$ rxrx -e '"foobar" =~ /(.)b/'
```

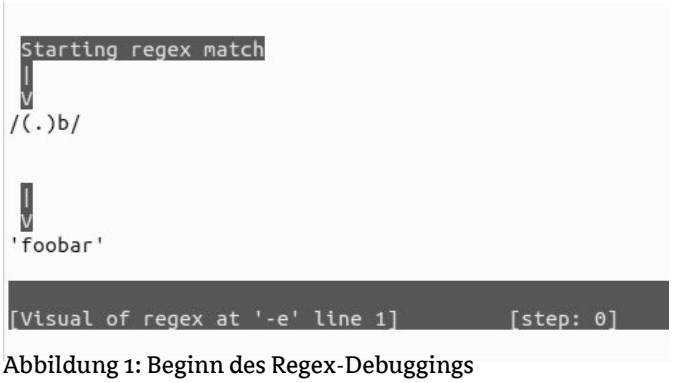


Abbildung 1: Beginn des Regex-Debuggings

```
'foobar' =~ /(.)b/; # ein beliebiges Zeichen vor 'b'

Freeing REx: `","'
Compiling REx `(.)b'
size 8 Got 68 bytes for offset annotations.
first at 3
   1: OPEN1(3)
   3:   REG_ANY(4)
   4: CLOSE1(6)
   6: EXACT <b>(8)
   8: END(0)
anchored "b" at 1 (checking anchored) minlen 2
Offsets: [8]
        1[1] 0[0] 2[1] 3[1] 0[0] 4[1] 0[0] 5[0]
Guessing start of match, REx "(.)b" against "foobar"...
Found anchored substr "b" at offset 3...
Starting position does not contradict /^/m...
Guessed: match at offset 2
Matching REx "(.)b" against "obar"
  Setting an EVAL scope, savestack=3
   2 <fo> <obar>          |  1:  OPEN1
   2 <fo> <obar>          |  3:  REG_ANY
   3 <foo> <bar>          |  4:  CLOSE1
   3 <foo> <bar>          |  6:  EXACT <b>
   4 <foob> <ar>          |  8:  END
Match successful!
Freeing REx: `"(.)b"
```

Listing 1

Damit landet man schon im Debugger. In Abbildung 1 ist der Ausgangsbildschirm zu sehen. Mit der *Space*-Taste kann man jetzt Schritt für Schritt durch die Abarbeitung des Regulären Ausdrucks gehen. Man bekommt auch mitgeteilt wenn die Regex-Engine Backtracken muss (Abbildung 2). Zusätzlich erhält man die Information wenn etwas in einer *Capturing Group* landet (Abbildung 3).

Man kann sich auch verschiedene Darstellungen des Ablaufs anzeigen lassen. Mit der *j*-Taste bekommt man die JSON-Darstellung der einzelnen Schritte (Listing 2)

Diese vier JSON-Objekte zeigen das Matching von "(.)". Mit der *v*-Taste kommt man wieder in den vorher gezeigt visuellen Modus zurück. Man kann aber nicht nur vorwärts gehen, mittels - kann man sogar rückwärtsgehen, so dass man den Ablauf nicht von vorne starten nur weil man etwas schnell mit der Tastatur war.

Bei komplexeren Regulären Ausdrücken kann es schon eine ganze Weile dauern bis man in einzelnen Schritten durch den Ablauf gegangen ist. In solchen Fällen kann man auch Schritte überspringen. Mittels *m* kann man bis zum nächsten erfolgreichen match springen, mit *f* bis zum nächsten erfolglosen match.

Abbildung 2: Auch Backtracking ist manchmal nötig

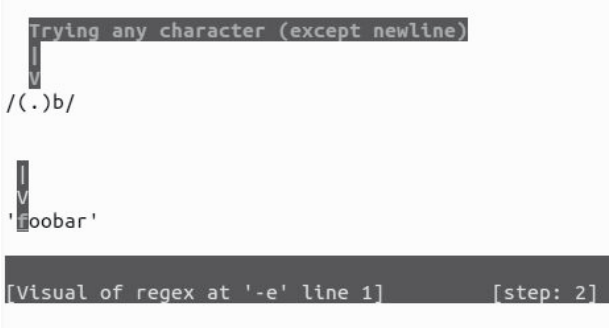


Abbildung 3: Informationen welche Zeichen gematcht werden

```
 {
 "str_pos" : 2,
 "event" : {
    "construct_type" : "_capture_group",
    "msg" : "Capture to $1",
    "event_type" : "pre",
    "is_capture" : 1,
    "indent" : "    ",
    "depth" : 2,
    "regex_pos" : 0,
    "capture_name" : "$1",
    "construct" : "(",
    "desc" : "The start of a capturing block ($1)",
    "quantifier" : ""
 },
 "regex_pos" : 0
}
,
  {
 "str_pos" : 2,
 "event" : {
    "construct_type" : "_metacharacter",
    "msg" : "Trying any character (except newline)",
    "event_type" : "pre",
    "indent" : "      ",
    "depth" : 3,
    "regex_pos" : 1,
    "matchable" : 1,
    "construct" : ".",
    "desc" : "Match any character (except newline)",
    "quantifier" : ""
 },
 "regex_pos" : 1
}
,
  {
 "str_pos" : 3,
 "starting_str_pos" : "2",
 "event" : {
    "construct_type" : "_metacharacter",
    "msg" : "Matched",
    "event_type" : "post",
    "indent" : "      ",
    "depth" : 3,
    "regex_pos" : 1,
    "matchable" : 1,
    "construct" : ".",
    "quantifier" : ""
 },
 "regex_pos" : 1
}
,
  {
 "str_pos" : 3,
 "starting_str_pos" : "2",
 "event" : {
    "construct_type" : "_capture_group",
    "msg" : "End of $1",
    "event_type" : "post",
    "is_capture" : 1,
    "indent" : "    ",
    "depth" : 2,
    "regex_pos" : 2,
    "capture_name" : "$1",
    "construct" : ")",
    "desc" : "The end of $1",
    "quantifier" : ""
 },
 "regex_pos" : 2
}
```


Listing 2

Sehr praktisch ist auch das Eventlog (Abbildung 4) mit einer übersichtlichen Darstellung der einzelnen Events. Diese Darstellung lässt sich über *e* erreichen. Eine Erklärung der einzelnen Regex-Bestandteile -- wie man es auch von *YAPE::Regexp::Explain* kennt -- lässt sich über *d* erreichen.

Leider gibt es auch ein paar Beschränkungen bei `Regexp::Debugger`. Wird der x-Modifier verwendet und sind Kommentare in dem Regulären Ausdruck, dann funktioniert der Debugger nicht richtig. Außerdem kann das Modul nicht mit der Interpolation von Variablen umgehen, wenn diese mittels `qr` erzeugte Regexp-Objekte sind:

```
my $ident = qr{ [^\W\d]\w* }x;
$str =~ m{ ($ident) : (.*) }xms;
```

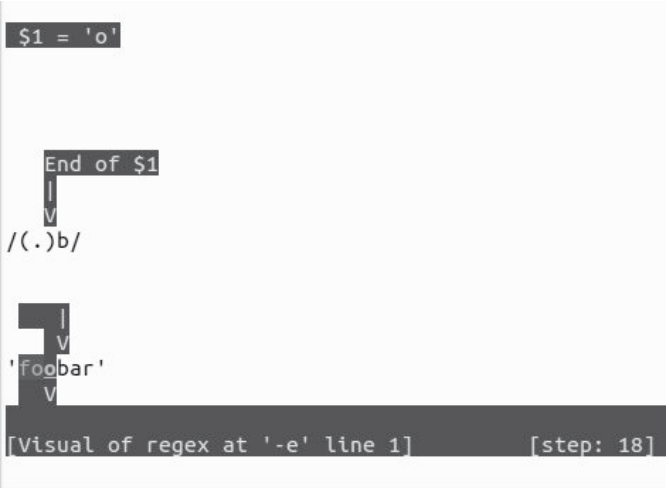


Abbildung 4: Das "o" wurde gecaptured

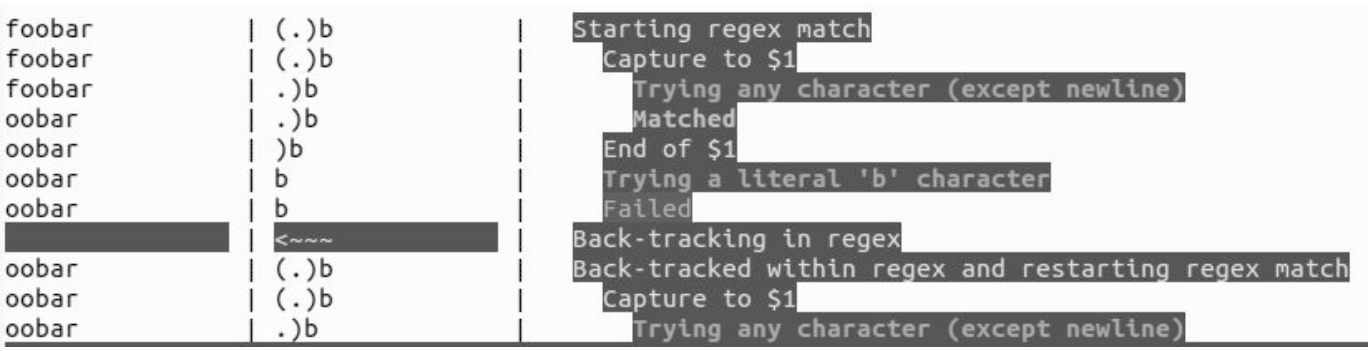


Abbildung 5: Das Eventlog

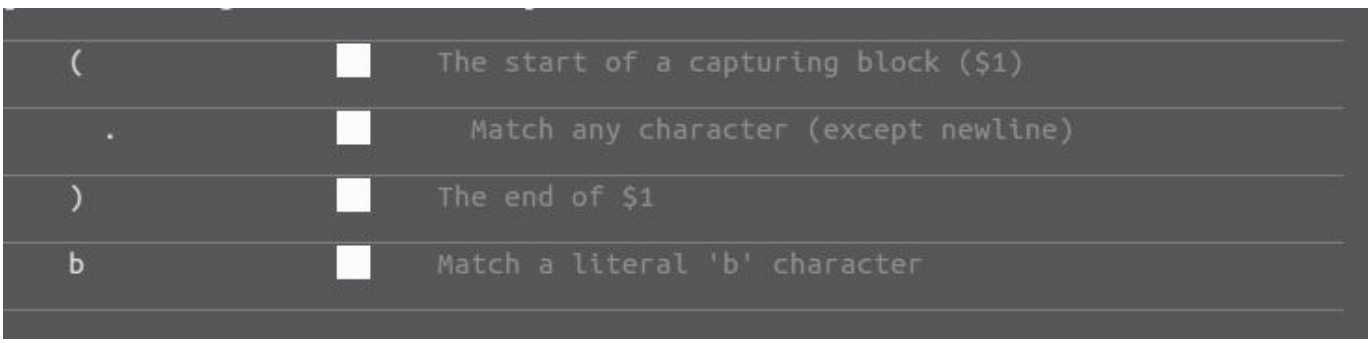


Abbildung 6: Die Bestandteile der Regex werden erklärt

Steffen Winkler

Internationalisierungs-Framework auswählen

In die Vergangenheit geschaut

Damit fing es wohl an: `Locale::Maketext::TPJ13`. Im Prinzip steht da drin, dass *GNU gettext* wegen Problemen bei der Übersetzung von Strings im Plural nicht die 1. Wahl ist. Und nun gibt es *Maketext*.

Natürlich hat sich GNU gettext über die Zeit verbessert. Ab und zu wird dann auch etwas verschlafen und man denkt, Maketext ist nach wie vor die 1. Wahl.

Im Anschluss an meinen Vortrag "DBD::PO - Mit SQL GNU gettext PO files bearbeiten" 2009 in Frankfurt/Main gab es eine rege Diskussion, sowohl in Frankfurt als auch bei Erlangen-PM. Irgendwann werde ich dieses `DBD::PO` auch noch auf die neue `DBI`-Version anpassen, aber wie immer - die liebe Zeit.

Warum also GNU gettext, obwohl noch einige Frameworks im CPAN Maketext benutzen?

GNU gettext ist weitaus mehr

Es geht um den gesamten Prozess, der auch nachfolgend beschrieben wird.

Der erste Schritt ist das Extrahieren der *pot*-Dateien. Die pot-Datei ist nichts anderes als eine *po*-Datei, nur eben ohne Übersetzungen.

po ist die Abkürzung für "portable object". GNU gettext po-Files kann man benutzen, um Programme mehrsprachig zu machen. In der Datei stehen neben dem Originaltext und der Übersetzung verschiedene Kommentare und Flags.

Dann werden diese pot-Dateien in po-Dateien von einem Übersetzer übersetzt, so dass es dann wenigstens für jede Sprache und Region eine Datei gibt.

Dann werden diese po-Dateien in binäre mo-Files umgewandelt und werden dabei von allem Ballast befreit. Letztendlich bleiben nur noch die für die Laufzeit-Übersetzung notwendigen Dinge enthalten. Im Prinzip kann man auch mit den po-Files auch direkt arbeiten, wenn die Ladezeit nicht stört.

Dafür gibt eine Bibliothek mit vielen nützlichen Tools.

Und dann ist beschrieben, wie man sich die tatsächliche Übersetzung zur Laufzeit vorstellt.

Perl-Module

`Locale::Maketext` benutzt genau genommen auch GNU gettext, weil typisch auch mit po/mo-Dateien gearbeitet wird.

Im CPAN findet man einige Module, die sich von Maketext verabschiedet haben und `Locale::TextDomain` benutzen, z.B. `Locale::Simple` und `Dancer::Plugin::Locale`.

Dann sagt man von `Locale::TextDomain`, dass es die System-Locale benutzt/verändert. Das ist vielleicht gut, vielleicht auch nicht. Aber auch `Locale::Maketext` versucht irgendwie die Sprache zu ermitteln.

Es gibt auch Module, die `Locale::TextDomain` nicht mehr benutzen und trotzdem die Funktionalität bereitstellen. Als Beispiel seien hier `Log::Report(::Message)` und `Locale::TextDomain::OO` genannt.

Laufzeit-Übersetzung mit gettext/Maketext benutzen

Was sind die Unterschiede? Wo sind die Grenzen?

Egal welches Internationalisierungs-Framework man vom CPAN benutzt, man muss mit Einschränkungen leben. Bei guter Wahl sind diese aber sehr gering.

Am Anfang ist der Quelltext der Anwendung

Die folgenden Befehle könnten so in einer Anwendung zu finden sein:

```
print 'You can log out here.';
printf 'He lives in %s, %s.',
    $town, $address;
printf '%d people live here.',
    $people;
printf 'These are %d books.',
    $books;
printf 'He has %s houses in %s, %s.',
    $houses, $town, $address;
printf '%s books are in %s shelves.',
    $books, shelves;
```

Hier bekommt der Benutzer nur die englische Ausgabe zu sehen. Gerade wenn das Programm nicht nur im englischsprachigen Raum laufen soll, ist eine Internationalisierung unerlässlich.

Locale::Maketext::Simple ausprobieren

Verwendet wird dabei das Basismodul `Locale::Maketext` und ein Modul, welches gettext po/mo-Files einliest. Das ist `Locale::Maketext::Lexicon::Gettext`. `Locale::Maketext::Simple` exportiert die Funktion "loc".

```
[_n] mit n = 1, 2, ...
```

ist die generelle Schreibweise für Platzhalter. In den `[]` kann ein Funktionsname vorangestellt werden, nachgestellt die Parameter. Das Trennzeichen ist das `,`.`quant`, kurz `*`, ist der Funktionsname für Plural-Verarbeitung.

```
print loc('You can log out here.');

print loc(
    'He lives in [_1], [_2].',
    $town,
    $address,
);

print loc(
  '[quant,_1,person lives,people live]'
  . ' here.',
  $people,
);
```

Ich habe keine Idee, wie man nachfolgende Phrase mit `quant` schreiben soll. Mit `quant` schreibt man so etwas wie Wert und nachfolgender Maßeinheit. Hier beginnt die Pluralform aber schon vor dem Wert. Das Problem ist, `quant` verlangt das Weglassen von "_1" in den Pluralformen und auch das Weglassen des darauf folgenden Leerzeichens. Also erfinde ich die Funktion `myplural` (Listing 1).

Auf Locale::TextDomain umschreiben

`Locale::TextDomain` gehört zur Distribution libintl-perl. Es gibt mehrere exportierte Funktionen. Der Funktionsname ist einfach gebaut.

- x steht für Platzhalter,
- n für Pluralform und
- p für Kontext.

```
print loc(
  '[myplural,_1,It is _1 book,These are _1 books].',
  # ???????? ^^^^^ ??? ^^^^^^^^^ ???
  $books,
);

print loc(
  'He has [quant,_1,house,houses] in [_2], [_3].',
  $houses,
  $town,
  $address,
);

print loc(
  '[quant,_1,book is,books are] in [*,_2,shelf,shelves].',
  $books,
  $shelves,
);
```

Listing 1

Genau gesagt gibt es auch noch `d` und `c` aber die sind z.B. als `dcgettext` usw. implementiert:

- d für TextDomain und
- c für Kategorie

Hinweis zwischendurch: Die Dateien liegen typischerweise in der nachfolgenden Struktur vor, aber man kann es auch anders machen.

```
.../de-de/LC_MESSAGES/project.po
^^^ ^^ ^^ ^^^^^^^^^^ ^^^^^^^-- TextDomain
|   |  |  +-------------------- Kategorie
|   |  +----------------------- Region
|   +-------------------------- Sprache
+------------------------------ Suchpfad
```

Nicht alle Varianten aus `n`, `p` und `x` sind implementiert. Wenn man `x` auch ohne Platzhalter benutzt und sich an die alphabetische Reihenfolge hält, bleiben `__x`, `__nx`, `__px` und `__npx` übrig.

Die Parameterreihenfolge ist, wenn vorhanden:

1. Kontext
2. Singular
3. Plural
4. Anzahl für Plural-Auswahl und
5. dann der Hash mit den Platzhalterdaten.

Typisch schreibt man Kontext, Singular und Plural in "Entwickler"-Englisch, welches dann für die jeweiligen Englischsprachigen Länder zu Ende übersetzt wird.

Listing 2 zeigt die Funktionen mit den jeweiligen Parametern:

Und die Beispiele sind in Listing 3 zu sehen.

Bei den `n`-Funktionen wird die Anzahl meist zwei Mal übergeben. Zum einen damit festgestellt werden kann, ob Singular oder Plural zur Anwendung kommt und zum anderen für den Platzhalter in der Ausgabe selbst.

Es ist wenig clever, das zu optimieren, denn dann kann man das nicht mehr abbilden:

```
Lesen Sie die Anlage zu diesem Schreiben.
Lesen Sie die Anlagen zu diesem Schreiben.
```

Die Anzahl der Anlagen kommt im Text nicht vor aber trotzdem unterscheidet sich der Text durch die Anzahl.

Was sieht man auf den ersten Blick?

Was sofort auffällt ist, dass `Locale::Maketext` durchnummerierte Parameter hat. Das birgt die Gefahr, dass man durcheinander kommt wenn es viele Parameter werden. Man kann sie verwechseln. Der Übersetzer, weiß nur, dass etwas eingefügt wird aber nicht was:

```
[_1] is a [_2] in [_3].
```

`Locale::Maketext` kann mit mehreren Pluralformen in einer Textphrase umgehen.

```
[quant,_1,book is,books are] in \
  [*,_2,shelf,shelves].
```

Der Text bei Pluralformen (`quant`) ist nicht mehr automatisch übersetzbar, weil eine Art "oder"-Block enthalten ist.

In diesem "oder"-Block ist der Platzhalter wie z.B. `_1` nicht mehr enthalten. Damit sind Pluralformen nicht darstellbar, welche bereits vor der Zahl beginnen.

```
[myplural,_1,It is _1 book,\
  These are _1 books].
```

Die Funktion "myplural" gibt es natürlich nicht.

`Locale::TextDomain` hat dagegen benannte Parameter, welche sich besser übersetzen lassen, weil der Übersetzer den Sinn des Satzes trotz Platzhalter immer noch verstehen kann.

```
{name} is a {locality} in {country}.
```

Manchmal übersetzen die Übersetzer auch die Platzhalter. Das muss man dann mit denen besprechen, dann passiert das nicht mehr.

Bei mehreren Pluralformen in einer Textphrase muss diese zerlegt werden oder nutzt, wenn man dazu gezwungen wird, den Hack aus der in Locale-TextDomain-OO-[version]/example/16_multiplural_mo_utf-8.pl gezeigt wird.

Was man nicht gleich erkennt.

Etwas suptiler sind die Unterschiede wenn es um die Behandlung des Plurals geht.

Anzahl der Pluralformen

Die Anzahl der Pluralformen kann von Sprache zu Sprache unterschiedlich sein, wie später noch Beispiele zeigen. Das muss bei der Übersetzung berücksichtigt werden. Deshalb sollten das die Tools auch unterstützen. Der folgende Vergleich zeigt, dass diese sich aber erheblich unterscheiden:

Locale::Maketext:

Folgende Pluralformen werden von `Locale::Maketext` unterstützt:

- Singular
- Singular + Plural
- Singular + Plural + Zero

```
__('msgid')

__x(
  'msgid',
      name1 => $value1,
      name2 => $value2,
      # ...
)

__n('msgid', 'msgid_plural', $count)

__nx(
  'msgid', 'msgid_plural', $count,
      name1 => $value1,
      name2 => $value2,
      # ...
)

__xn(
  'msgid', 'msgid_plural', $count,
      name1 => $value1,
      name2 => $value2,
      # ...
)

__p('context', 'msgid')

__px(
  'context', 'msgid',
      name1 => $value1,
      name2 => $value2,
      # ...
)

__np(
  'context', 'msgid', 'msgid_plural',
      $count
)

__npx(
  'context', 'msgid', 'msgid_plural',
      $count,
      name1 => $value1,
      name2 => $value2,
      # ...
)
```

Listing 2

Locale::TextDomain:

Mit `Locale::TextDomain` ist man da wesentlich flexibler:

- 2 in der Quellsprache
- beliebig viele in der Zielsprache

Im Header jeder po/mo-Date steht die Information *Plural-Forms*. Das ist die Berechnungsvorschrift als C-Code - mit einer Ausnahme, `OR` anstatt von `||` ist erlaubt. Diese ist sprachabhängig unterschiedlich in den einzelnen po/mo-Dateien gespeichert. `Locale::Maketext` ignoriert diesen Eintrag. Im Folgenden zwei Beispiele für so eine Berechnungsvorschrift:

```
print __('You can log out here.');

print __x(
  'He lives in {town}, {address}.',
     town => $town,
     address => $address,
);

print __nx(
  '{num} person lives here.',
  '{num} people live here.',
      $people,
      num => $people,
);

print __nx(
  'It is {num} book.',
  'These are {num} books.',
    $books,
    num => $books,
);

print __nx(
  '{num} house in {town}, {address}.',
  '{num} houses in {town}, {address}.',
    $houses,
    num     => $houses,
    town    => $town,
    address => $address,
);

print __nx(
  '{num} book is',
  '{num} books are',
    $books,
    num => $books,
),
__nx(
  ' in {num} shelf.',
  ' in {num} shelves.',
    $shelves,
    num => $shelves,
);
```

Listing 3

Deutsch/Englisch:

```
"Plural-Forms: nplurals=2; \
  plural=n != 1\n";
```

Russisch:

```
"Plural-Forms: nplurals=3; \
  plural=n%10==1 && n%100!=11"
" ? 0 : n%10>=2 && n%10<=4 && \
   (n%100<10 || n%100>=20) ? 1 : 2;\n"
```

Wenn man eine spezielle Variante für die 0 haben will, um so etwas wie "keine Bücher" anstatt "0 Bücher" abzubilden, kann man die Formel entsprechend abwandeln. Dann ist nplurals auch eins mehr. Wenn man das nicht immer braucht, kann man die TextDomain oder wenn möglich auch die Kategorie ändern, dann entsteht eine separate po/mo-Datei für jede Sprache und Region.

Ein Beispiel aus dem Russischen:

```
0 books -> книг (Plural 2)
1 book -> книга (Singular)
2 .. 4 books -> книги (Plural 1)
5 .. 20 books -> книг (Plural 2)
21 books -> книга (Singular)
22 .. 30 books -> книг (Plural 2)
...
100 books -> книг (Plural 2)
101 books -> книга (Singular)
102 .. 104 books -> книги (Plural 1)
105 .. 120 books -> книг (Plural 2)
121 book -> книга (Singular)
122 .. 124 books -> книги (Plural 1)
125 .. 130 books -> книг (Plural 2)
...
```

Drei Pluralformen haben z.B. auch Tschechisch, Litauisch, Polnisch, Rumänisch und Slowakisch. Vier Pluralformen haben z.B. Slowenisch und Keltisch. In der EU kommen wir also mit vier Pluralformen aus. Sechs Pluralformen hat Arabisch.

Weil `Locale::Maketext` *Plural-Forms* in po/mo-Dateien ignoriert, sind damit nur Sprachen mit 2 Pluralformen möglich, also Singular und Plural, so wie wir das aus Deutsch und Englisch kennen.

Es gibt eine Funktion "quant", welche im Prinzip "quant2" (Singular + 1. Plural) entspricht, wenn man von der Nullform absieht. Man könnte für `Locale::Maketext` eine Funktion "quant3" bis "quant6" definieren. Damit müsste aber der Programmierer schon wissen, welche Textphrasen 2, 3, 4, 5 oder 6 Pluralformen benötigen. Weil er das nicht weiß, muss er dann immer "quant6" benutzen. Damit schreibt er sich die Finger wund.

Position der Worte im Satz in unterschiedlichen Sprachen

Die Position der einzelnen Worte kann in unterschiedlichen Sprachen unterschiedlich sein, d.h. in einer Sprache heißt es

```
I have 2 books.
```

und in einer anderen

```
2 books I have.
```

Wenn das so ist, muss man bei `Locale::Maketext` komplette Sätze in den Pluralformen schreiben. Das kann der Englisch programmierende nicht wissen. Der Konflikt wird also erst während der Übersetzung bekannt.

Wenn man den Konflikt umgehen möchte, schreibt man immer die kompletten Sätze.

Das funktioniert aber auch nicht immer, weil `Locale::Maketext` nach `quant` immer `_1` erwartet und dann kommt das implizit hinzugefügte Leerzeichen und danach der Text.

Gebraucht würde aber:

```
[myplural,_1,It is _1 book.,\
  These are _1 books.]
```

Das ist dann aber nichts anderes als `Locale::TextDomain`.

Komma in den Pluralformen oder die "join and never can split"-Falle

Durch simple Stringverkettung mit Komma darf kein Komma in verketteten Texten sein.

Gibt es einen Quotingmechanismus wie bei Text::CSV? Mir ist keiner bekannt. Es gibt nur das ~ als Escape-Zeichen. Ich habe es bisher nur bei `~[[_1]~]` gesehen, damit man auch in der Lage ist `[> oder <]` im Text zu haben, die nichts mit Platzhaltern zu tun haben.

```
I need 1 book, computer or \
  notebook to do this.
```

Hier ein dreckiger Workaround mit ";".

```
I need [*_1,book; computer or notebook,\
  books; computers or notebooks] \
  to do this.
```

Möglicherweise funktioniert auch hier das Escape-Zeichen ~ (nicht getestet):

```
I need [*_1,book~, computer or notebook,\
  books~, computers or notebooks] to do \
  this.
```

Wert und Maßeinheit werden ggf. umgebrochen

Durch Stringverkettung mit Leerzeichen entstehen Zeilenumbrüche zwischen Wert und Maßeinheit.

Das ergibt je nach Zeilenlänge

```
I have
1 book.
```

oder

```
I have 1
book.
```

Für Locale::TextDomain kann man schreiben:

```
I have {num}\N{NO-BREAK SPACE}book.
I have {num}\N{NO-BREAK SPACE}books.
```

In `Locale::Maketext` ist das Leerzeichen unveränderbar im Modulcode enthalten.

Auszug aus dem po-File für Locale::Maketext

Hier wird die gettext-Notation benutzt, also `%1` anstatt `[_1]` und `%funktion($1,singluar,plural)` anstatt `[funktion,_1,singular,plural]`. (Listing 4).

Übersetzungsdateien

Im Folgenden werden Beispielhaft Auszüge aus Übersetzungsdateien gezeigt. `msgid` ist die Phrase, wie sie in den Programmen verwendet werden, `msgstr` die Übersetzung. Gibt es die Phrase auch in Pluralform(en), dann wird die Phrase in Plural bei `msgid_plural` angegeben und die `msgstr` bekommen einen Index.

Auszug aus der po-Datei für Locale::TextDomain

Hier ist eine Übersetzungsdatei für `Locale::TextDomain` zu sehen. Im Header sieht man die Angabe der Berechnungsformel für den Plural unter *Plural-Forms* und man sieht Beispiele für Phrasen im Singular und Phrasen die auch im Plural vorkommen können. (Listing 5)

```
# header
msgid ""
msgstr ""
"...\n"
"Plural-Forms: nplurals=2; plural=n != 1;\n"
"..."

msgid "You can log out here."
msgstr "Sie können sich hier abmelden."

msgid "He lives in %1, %2."
msgstr "Er wohnt in %1, %2."

msgid "%quant(%1,person lives,people live) here."
msgstr "%quant(%1,Mensch wohnt,Menschen wohnen) hier."

# schlechter Workaround (kein Singular vor dem Placeholder)
msgid "This are %quant(%1,book,books)."
msgstr "Das sind %quant(%1,Buch,Bücher)."

msgid "%quant(%1,book is,books are) in %quant(%2,shelf,shelves)."
msgstr "%quant(%1,Buch ist,Bücher sind) in %quant(%2,Regal,Regalen)."
```

Listing 4

```
# header
msgid ""
msgstr ""
"...\n"
"Plural-Forms: nplurals=2; plural=n != 1;\n"
"..."

msgid "You can log out here."
msgstr "Sie können sich hier abmelden."

msgid "He lives in {town}, {address}."
msgstr "Er wohnt in {town}, {address}."

msgid "{num} person lives here."
msgid_plural "{num} people live here."
msgstr[0] "{num} Mensch wohnt hier."
msgstr[1] "{num} Menschen wohnen hier."

msgid "It is {num} book."
msgid_plural "These are {num} books."
msgstr[0] "Es ist {num} Buch."
msgstr[1] "Es sind {num} Bücher."

msgid "He has {num} house in {town}, {address}."
msgid_plural "He has {num} houses in {town}, {address}."
msgstr[0] "Er hat {num} Haus in {town}, {address}."
msgstr[1] "Er hat {num} Häuser in {town}, {address}."

msgid "{num} book is"
msgid_plural "{num} books are"
msgstr[0] "{num} Buch ist"
msgstr[1] "{num} Bücher sind"

msgid " in {num} shelf."
msgid_plural " in {num} shelves.
msgstr[0] " in {num} Regal."
msgstr[1] " in {num} Regalen."
```

Listing 5

po-File Englisch/Russisch übersetzt

Wie weiter oben gezeigt, ergibt sich bei der Übersetzung ins Russische das Problem, dass es im Russischen mehrere Pluralformen gibt.

für Locale::Maketext

Dieses Beispiel für `Locale::Maketext` verdeutlicht diese Probleme, die in den Kommentaren innerhalb der Datei verdeutlicht werden (Listing 6).

für Locale::TextDomain

In dieser Übersetzungsdatei ist ein Vorteil von `Locale::TextDomain` zu sehen. Es können die Übersetzungen von mehreren Plural-Formen angegeben werden (Listing 7).

Hier ist ein Kommentar für den Übersetzer hinzugefügt, damit er weiß, dass da etwas getrennt wurde. Ansonsten ist die Regel, so viel wie möglich in eine Phrase packen, und auch die Satzzeichen wie . ! ? oder : am Ende nicht vergessen, damit der Übersetzer es inhaltlich korrekt übersetzen kann.

```
# Translate this phrase together with
# the next one.
msgid "{num} book is"
msgid_plural "{num} books are"
msgstr[0] "{num} книга"
msgstr[1] "{num} книги"
msgstr[2] "{num} книг"

# Translate this phrase together with
# the previous one.
msgid " in {num} shelf."
msgid_plural " in {num} shelves."
msgstr[0] " на {num} полке."
msgstr[1] " на {num} полках."
msgstr[2] " на {num} полках."
```

Beugen von "in {town}"

Je nach verwendung des Städtenamens, muss dieser evtl. gebeugt werden:

```
Berlin -> Берлин
in Berlin -> в Берлине
```

Wenn man das will, muss man Platzhalterwerte auch wieder übersetzen und dann erst einfügen.

```
# header
msgid ""
msgstr ""
"...\n"
"Plural-Forms: nplurals=3; plural=n%10==1 && n%100!=11"
" ? 0 : n%10>=2 && n%10<=4 && (n%100<10 || n%100>=20) ? 1 : 2;\n"
"..."

msgid "You can log out here."
msgstr "Выход из системы."

# Hier wäre Beugung des Stadtnamens notwendig:
# Москва -> в Москве
# Киев -> в Киеве
# Мытищи -> в Мытищах (nicht regulär)
msgid "He lives in %1, %2."
msgstr "Он живет в %1, %2"

# Das ist nicht korrekt übersetzbar.
# Die Pluralform für 2 bis 4 (человека живут) ist nicht speicherbar.
msgid "%quant(%1,person lives,people live) here."
msgstr "%quant(%1,человек живет,человек живут) здесь."

# Das ist nicht korrekt übersetzbar.
# Die Pluralform für 2 bis 4 (дома) ist nicht speicherbar.
msgid "He has %quant(%1,house,houses) in %2, %3."
msgstr "У него %quant(%1,дом,домов) в %2, %3."

# Das ist nicht korrekt übersetzbar.
# Die Pluralform für 2 bis 4 (книги) ist nicht speicherbar.
msgid "%quant(%1,book is,books are) in %quant(%2,shelf,shelves)."
msgstr "%quant(%1,книга,книг) на %quant(%1,полке,полках)."
```

Listing 6

```
# header
msgid ""
msgstr ""
"...\n"
"Plural-Forms: nplurals=3; plural=n%10==1 && n%100!=11"
" ? 0 : n%10>=2 && n%10<=4 && (n%100<10 || n%100>=20) ? 1 : 2;\n"
"..."

msgid "You can log out here."
msgstr "Выход из системы."

# Hier wäre Beugung des Stadtnamens notwendig:
# Москва -> в Москве
# Киев -> в Киеве
# Мытищи -> в Мытищах (nicht regulär)
msgid "He lives in {town}, {address}."
msgstr "Он живет в {town}, {address}."

msgid "{num} person lives here."
msgid_plural "{num} people live here."
msgstr[0] "{num} человек живет здесь."
msgstr[1] "{num} человека живут здесь."
msgstr[2] "{num} человек живут здесь."

msgid "It is {num} book."
msgid_plural "These are {num} books."
msgstr[0] "Это {num} книга."
msgstr[1] "Это {num} книги."
msgstr[2] "Это {num} книг."

msgid "He has {num} house in {town}, {address}."
msgid_plural "He has {num} houses in {town}, {address}."
msgstr[0] "У него {num} дом в {town}, {address}."
msgstr[1] "У него {num} дома в {town}, {address}."
msgstr[2] "У него {num} домов в {town}, {address}."
```

Listing 7

Das geht, macht aber das automatische Übersetzen der Phrase unmöglich, in dies eingefügt werden soll. Außerdem kann man diese dann auch wieder nur schwer manuell übersetzen, weil der Zusammenhang wieder etwas verloren geht.

Eine andere Alternative wäre, dass man einen Filter für die Platzhalter-Ersetzung implementiert, vielleicht so: `{town: rule_name}` Dieser sollte dann in der Lage sein, die jeweilige grammatikalische Regel umzusetzen.

Es ist auf jeden Fall etwas Bastelei. Es hängt immer von den Anforderungen ab, wie weit man wirklich gehen muss. Nur wenn ein System von vorn herein schon mit vielen Einschränkungen gestrickt ist, dann kommt man zu schnell an die Grenzen.

neutral/masculin/feminin singular/plural

Beugen von Substantiven:

```
maskulin singular -> Arzt
feminin singular -> Ärztin
maskulin plural -> Ärzte
feminin plural -> Ärztinnen
```

Beugen von Verben:

```
feminin: Mascha ist zur Schule gegangen.
  -> Маша пошла в школу.
maskulin: Petja ist zur Schule gegangen.
  -> Петя пошёл в школу.
```

Kontext

Als Perl-Programmierer kennen wir uns mit Kontexten aus. Auch in der natürlichen Sprache gibt es den Kontext, so hat das Wort "Ball" eine andere Bedeutung je nachdem ob ich sage "Ich gehe auf einen Ball" oder "Pass mal den Ball rüber!". Solche unterschiedliche Bedeutungen müssen auch in der Übersetzung berücksichtigt werden. Deshalb gibt es bei `Locale::TextDomain` die Möglichkeit unterschiedliche Kontexte anzugeben. Das wird entsprechend in den po-Dateien eingetragen:

```
msgid "design"
msgstr "Design"

msgctxt "automobile"
msgid "design"
msgstr "Konstruktion"

msgctxt "verb"
msgid "design"
msgstr "zeichnen"
```

Im Perl-Code können dann die `p` Funktionen verwendet werden:

```
print __p( "automobile", "desgin" );
# Konstruktion

print __p( "verb", "design" );
# zeichnen

print __( "design" );
# Design
```

TextDomain N Funktionen

Die TextDomain-Module haben Funktionen mit einem *N* vorn dran, wie z.B. `N__`. Die machen gar nichts, sie geben einfach zurück, was man ihnen gibt. Aber der Extractor sieht sie. Man kann damit Mappingtabellen aufbauen, ohne alle mappings durch die Laufzeit-Übersetzung zu jagen.

Der Extractor kennt damit die Texte, welche zu übersetzen sind.

```
my %name_of = (
  '#FF0000' => N__('red'),
  '#00FF00' => N__('green'),
  '#0000FF' => N__('blue'),
);
```

Deswegen ist es hier erlaubt, __ mit Variablen aufzurufen.

```
print __( $name_of{'#FF0000'} );
```

Ansonsten ist `__($var)` nicht möglich, weil der Inhalt von $var nicht extrahierbar ist.

Der Extraktor ist eine Software, welche im Quelltext z.B. nach dem Muster `__('...'` sucht. Er speichet den gefundenen Text `'...'`, samt der Stelle, wo er gefunden wurde in einer po/pot-Datei. Abhängig vom Typ der Datei (*.pl, *.pm, *.js, ...) wird er bestimmte Teile nicht untersuchen, wie z.B. Kommentare, POD oder alles nach `__END__`.

Software für Übersetzungsbüros

In einem mir bekannten Fall, benutzt das Übersetzungsbüro die Software "SDL Trados". Es beruht wie andere vergleichbare Software auf einem "translation memory". Das funktioniert sehr gut für statische Dokumente.

Für die Dynamik, welche durch Plural und Kontext in der Softwarelokalisation real existiert, scheint solche Software weniger geeignet. Sie geht von eine 1:1-Übersetzung aus. Man muss also damit rechnen, dass die anteilmäßig eher geringe Teil mit Kontext oder Pluralformen nicht gut softwareunterstützt erbracht werden kann.

In dem Fall musste das pot-File in XML umgewandelt werden und dann die Zielsprache mit der Quellsprache vorbelegt werden. Diese Leistung hätte man eher vom Übersetzungsbüro erwartet.

Empfehlung: Testübersetzung einer kleineren Datei durchführen lassen. Diese sollte alle typischen Konstrukte enthalten. Und das je Sprache, weil teilweise Subunternehmen eingebunden werden.

Vielleicht ist es auch sinnvoll, die speziellen Dinge in einem separaten File auszulagern und auf 100% manuelle Übersetzung zu bestehen. Bei den eher statischen Dingen kann man maschinell übersetzen und danach manuell korrigieren lassen.

Bibliographie

GNU gettext
wikipedia http://en.wikipedia.org/wiki/Gettext
gettext homepage http://www.gnu.org/software/gettext/gettext.html

Singular, Plural, Dual, Trial, Quadral
wikipedia - Dual http://de.wikipedia.org/wiki/Dual_(Grammatik)
wikipedia - alle Formen http://de.wikipedia.org/wiki/Sursurunga
sourceforge - welche Sprache - welche Pluralform http://translate.sourceforge.net/wiki/l10n/pluralforms

CPAN-Modul Locale::Maketext
CPAN https://metacpan.org/pod/Locale::Maketext

CPAN-Modul Locale::Maketext::Simple
CPAN https://metacpan.org/pod/Locale::Maketext::Simple

veralteter Artikel von Sean M. Burke über Software-Lokalisierung
CPAN https://metacpan.org/pod/Locale::Maketext::TPJ13

CPAN-Modul Locale::TextDomain
CPAN https://metacpan.org/pod/Locale::TextDomain

CPAN-Modul Locale::Simple
CPAN https://metacpan.org/pod/Locale::Simple

CPAN-Modul Dancer::Plugin::Locale
CPAN https://metacpan.org/pod/Dancer::Plugin::Locale

CPAN-Modul Log::Report
CPAN https://metacpan.org/pod/Log::Report

CPAN-Modul Locale::TextDomain::OO
CPAN https://metacpan.org/pod/Locale::TextDomain::OO

Danke für die Unterstützung, die vielen Ideen, Beispiele und Korrekturen.
Nikolai Prokoschenko http://rassie.org/
Nikolai Prokoschenko - On the state of i18n in Perl http://rassie.org/archives/247

Markus Benning

LWP::UserAgent, SSL und Proxy

Der Einsatz von SSL mit einem Proxy birgt mit `LWP::UserAgent` einige Tücken. Je nach eingesetzter Version bereitet ersteres oder letzteres Probleme.

LWP::UserAgent 5.8xxx und Net::SSL

Würde man sich allein auf die Dokumentation von `LWP::UserAgent` verlassen, so würde der erste Versuch, einen Proxy für *https://* URLs zu setzen, wahrscheinlich so aussehen:

```
$ua->proxy(
  'https' => 'http://proxy.domain.tld:3128'
);
```

Auf den ersten Blick funktioniert das sogar. Wenn man allerdings genauer hin sieht, stellt man fest dass lediglich ein regulärer HTTP Request an den Proxy gesendet wird:

```
$ua->get('https://www.google.de/');
```

Resultiert in einem GET Request an den Proxy:

```
GET https://www.google.de/ HTTP/1.1
```

Dieser holt dann die Seite vom Webserver und liefert sie per HTTP zurück an den Client.

Alle SSL Parameter werden in diesem Fall ausschließlich vom Proxy Server bestimmt, da dieser die SSL Verbindung herstellt und hält. Der Proxy handelt die SSL Verbindung aus. Von ihm hängt ab, welche CAs, Chiphers, Protokoll-Versionen akzeptiert werden. Außerdem ist es nicht möglich, Client Authentifizierung zu verwenden. Das Verhalten, dass man in den meisten Fällen erwarten würde und man vom Webbrowser gewohnt ist, wäre das Verwenden der HTTP 'CONNECT' Methode. Mit dieser kann der Client den Proxy Server anweisen eine TCP-Verbindung zu einem Ziel herzustellen und direkt durch zuleiten. Der Client sendet hierzu einen CONNECT Request:

```
CONNECT www.google.de:443 HTTP/1.1
Host: www.google.de:443
¿
```

Der Server antwortet hierauf dann entweder mit einem Fehler-Code oder bei Erfolg mit einem 200 OK Status. Wird ein 200 OK Status geliefert, endet die HTTP Konversation und der Client kann die Verbindung wie eine direkte TCP Verbindung zum gewünschten Host verwenden.

In unserem Fall würde dieser eine SSL Verbindung aushandeln und den HTTP Request ausführen.

`LWP::UserAgent` unterstützt dies jedoch nicht. Stattdessen bietet das unterliegende `Net::SSL` Modul, welches für die SSL Implementierung verwendet wird, diese Funktionalität. Die Parameter hierzu müssen per Umgebungsvariablen übergeben werden. Die möglichen Parameter kann man der `Net::SSL` Dokumentation entnehmen. Ersetzt man die Konfiguration mittels `$ua->proxy('https' => 'http://proxy.domain.tld:3128')` durch das Setzen der entsprechenden Umgebungsvariable:

```
$ENV{'HTTPS_PROXY'} =
  'proxy.domain.tld:3128'; # ohne http://
```

wird die Verbindung per HTTP CONNECT über den Proxy hergestellt.

Weiter fällt dann jedoch auf, dass das Zertifikat der Gegenstelle in keiner Weise überprüft wird. Mit weiteren Variablen kann man `Net::SSL` anweisen, nur SSLv3 einzusetzen und das Zertifikat gegen eine CA Liste zu überprüfen:

```
$ENV{'HTTPS_CA_FILE'} =
  '/etc/pki/tls/certs/ca-bundle.crt';
$ENV{'HTTPS_VERSION'} = 3;
```

Jedoch bleibt es trotz dessen immer noch an einem selbst, zu überprüfen, ob das Zertifikat der Gegenstelle auch mit dem Hostnamen übereinstimmt. Hierzu gäbe es die Möglichkeit, das Subject des Serverzertifikats über einen Header auszulesen, denn LWP selbst setzt:

```
my $x509_subject =
  $response->header(
    'client-ssl-cert-subject'
  );
```

Doch wer will dies schon bei jedem Request selbst implementieren?

LWP::UserAgent 6.xx mit IO::Socket::SSL

Besser ist die Situation mit SSL bei `LWP::UserAgent` ab der Version 6.00. Hier wird von Haus aus ein CA Bundle über eine Abhängigkeit auf das Modul `Mozilla::CA` angezogen und verwendet. Der Hostname wird auch bereits per Voreinstellung überprüft.

Will man die Parameter anpassen, muss man nicht mehr den Weg über die Umgebungsvariablen einschlagen, sondern kann die Parameter direkt an `LWP::UserAgent` übergeben:

```
$ua->ssl_opts(
    verify_hostname => 1,
    SSL_ca_file =>
      '/etc/pki/tls/certs/ca-bundle.crt',
    # SSL_version => 'TLSv12',
    # SSL_cipher_list =>
    #   'DHE-RSA-AES256-GCM-SHA384',
);
```

Verwendet werden können alle Parameter, die `IO::Socket::SSL` unterstützt. Dieses wird ab LWP 6.xx per Voreinstellung anstatt `Net::SSL` verwendet. Ein Fallback auf `Net::SSL` findet nur noch statt, wenn `IO::Socket::SSL` nicht verfügbar ist oder ein Fallback über `$ENV{'PERL_NET_HTTPS_SSL_SOCKET_CLASS'}` erzwungen wird.

Der Nachteil liegt allerdings darin, dass `Net::SSL` jetzt auch nicht mehr die Proxyverbindung per HTTP CONNECT für uns herstellt und LWP dies aktuell noch nicht selbst beherrscht. An diesem Punkt setzt das Modul `LWP::Protocol::connect` an, indem es diese Lücke füllt.

LWP::Protocol::connect

Das Module implementiert das Protokoll *connect*, das dann in der Proxykonfiguration von LWP::UserAgent verwendet werden kann:

```
$ua->proxy(
  'https' =>
    'connect://proxy.domain.tld:3128'
);
```

Statt einem GET Request verwendet das Modul die erwünschte CONNECT Methode für den Verbindungsaufbau.

Da das Modul selbst wiederum LWP Funktionalität verwendet, um die CONNECT Anfrage zu erstellen, kann auch der reguläre Mechanismus für die Authentifizierung verwendet werden.

Gibt man z.B. entsprechende Daten in der Proxy-URL an, wird ein "Proxy-Authentication:" Header mit einer Basic Authentication verwendet:

```
$ua->proxy(
  'https' =>
  'connect://'
      . 'user:pw@proxy.domain.tld:3128'
);
```

Ein anderer Weg ist, die Zugangsdaten des Proxys an LWP::UserAgent über `< ->credentials` zu übergeben:

```
$ua->credentials(
  "localhost:3128",
  "Squid proxy-caching web server",
  "user",
  "pw",
);
```

Die Daten stehen dann allen `LWP::Authen::`-Modulen zur Verfügung. LWP wird versuchen, das passende `LWP::Authen::`-Modul zu verwenden - je nachdem welche Art von Authentication der Proxy Server erfordert.

In der aktuellen Version 6.09 ist noch Unterstützung für http hinzugekommen. Damit können unverschlüsselte HTTP Requests ebenfalls über eine CONNECT Verbindung gesendet werden, wenn der Proxy dies zulässt. Der Nutzen ist zwar auf dem ersten Blick beschränkt, aber eventuell kann mit dieser Funktion dem ein oder anderen Proxy auf die Sprünge geholfen werden. Zum Beispiel wäre denkbar, dass dieser bestimmte HTTP Methoden nicht erlaubt.

Ausblick LWP::UserAgent

Es wurden bereits Patches in `libwww-perl` und `LWP::Protocol::https` integriert, welche das Verhalten von *https_proxy* entsprechend anpassen, sodass Requests per CONNECT erfolgen. Wahrscheinlich werden diese Änderungen in die nächste libwww-perl Version 6.06 einfließen.

Boris Däppen

Email::Stuffer - Ich schicke mal eben schnell eine E-mail

„Ich schicke mal eben schnell eine E-mail“, dieser Satz gehört fest zur IT und das seit etwa 30 Jahren. Die E-Mail ist damit eindeutig ein Urgestein und trotzdem weiterhin beliebt. Für viele Anwender ist der Begriff inzwischen zu einem Synonym von „Webmail“ geworden. Der aufgeklärte Informatiker aber weiß: dahinter steckt ganz schön viel Technik.

Eine E-Mail, die man erwartet, ist immer noch ein freudiges Ereignis. Das konnte uns die Spam-Flut noch nicht austreiben. Auch der Perl-Programmierer freut sich ab und zu über die Post seiner Skriptchen.

Beim Skripten geht „Ich schicke mal eben schnell eine E-Mail“ aber nicht so gut über Webmail (ok, auch das wäre möglich). Man benötigt Code welcher das übernimmt. Und, weil es tausend Fallstricke gibt, nimmt man am liebsten ein fertiges Modul von CPAN. Gerade der Gelegenheitsprogrammierer, der ab und an mal wieder was zum Automatisieren hat, steht immer wieder vor der Frage, welches Modul er für seine Arbeit verwenden soll. Module, die E-Mails verschicken, gibt es zuhauf. *Ricardo Signes* - seines Zeichens CPAN-Autor mit impact - hat die Liste nun um eins erweitert.

`Email::Stuffer` versucht (aufs Neue), dem Anspruch von „Ich schicke mal eben schnell eine E-Mail“ gerecht zu werden. Es geht quasi darum dem Programmierer das „Webmail-Erlebnis“ beim Programmieren zu ermöglichen: Keine Fallstricke, keine Details, schick das Ding ab wie ich es dir gebe! Da es für die meisten Server-Konfigurationen dann eben doch nicht reicht einfach die Synopsis in den Code zu kopieren, soll hier ein robustes Grundgerüst gezeigt werden, womit man einsteigen kann. Ganz an das „Webmail-Erlebnis“ reicht es doch nicht heran. Aber immerhin bis zum „E-Mail-Client-Konfigurationserlebnis“.

Damit die Kommunikation mit dem Server klappt, sollten dringend die Module `Authen::SASL` und `MIME::Base64` installiert werden. Zwar funktioniert `Email::Stuffer` auch ohne. Es greift aber automatisch auf die Module zurück wenn der Server dies zur Authentifikation erfordert. Dies dürfte meistens der Fall sein. Um den Mailserver spezifizieren zu können, laden wir noch das Modul `Email::Sender::Transport::SMTP`. Dies wird nicht gebraucht wenn `sendmail` auf dem Server konfiguriert ist. Bei mir hat der Standard-Fall aber noch nie geklappt. Meistens muss man doch selber sagen wohin, also zu welchem Mailserver, die Post gehen soll.

Die obersten Zeilen unsere Skriptes sehen nun also so aus:

```
use Email::Stuffer;
use Email::Sender::Transport::SMTP;
use Authen::SASL;
use MIME::Base64;
```

Nun kommen wir zur Konfiguration des Mailservers. Hier muss man nicht viel mehr angeben als in jedem normalen Mailclient. Falls `sendmail` bereits eingerichtet ist, sollte man sich das wie gesagt sparen können, da dies per Grundeinstellung genutzt würde. Beim `port` muss man sich erkundigen, je nach Mailserver hat man mit `25` oder `465` die besten Chancen.

```
my $transport =
 Email::Sender::Transport::SMTP->new({
    host          => 'mail.beispiel.de',
    port          =>  25,
    sasl_username => 'ich@beispiel.de',
    sasl_password => 'geheimwort',
});
```

Das wars auch schon. Ab jetzt sind wir im „Webmail-Modus“. Eine einfache E-Mail sieht dann z.B. so aus:

```
Email::Stuffer->to ('der@beispiel.de'    )
    ->from ('ich@beispiel.de'  )
    ->subject ('Juchu'         )
    ->text_body ("Hi,\ndie neue Foo ist da!")
    ->transport ($transport)
    ->send_or_die();
```

Wer es gerne bunt mag kann anstatt `text_body()` die Methode `html_body()` verwenden. Anhänge können mit `attach_file()` hinzugefügt werden.

Wenn man zum Schluss mit `send()` verschickt, liefert `Email::Stuffer` einen Wert zurück der mit `if` auf Erfolg geprüft werden kann. Bequemer ist hier `send_or_die()`, so bricht das Programm gleich ab, wenn etwas schief gegangen ist.

Fazit: Das Modul bietet eine einfache Schnittstelle und kommt dem „Ich schicke mal eben schnell eine E-mail“ schon sehr nahe. Es ist aktuell und gepflegt. Die Dokumentation ist zwar nicht sehr einsteigerfreundlich, aber dafür gibt es ja das $foo-Magazin. Wer demnächst mal wieder Post zu verschicken hat, der soll es doch hiermit ausprobieren. Danke Ricardo!

Kompletter Code

```
 1 use Email::Stuffer;
 2 use Email::Sender::Transport::SMTP;
 3 use Authen::SASL;
 4 use MIME::Base64;
 5
 6 my $transport = Email::Sender::Transport::SMTP->new({
 7     host          => 'mail.beispiel.de',
 8     port          =>  25,
 9     sasl_username => 'ich@beispiel.de',
10     sasl_password => 'geheimwort',
11 });
12
13 Email::Stuffer->to        ('der@beispiel.de'          )
14               ->from      ('ich@beispiel.de'          )
15               ->subject   ('Juchu'                    )
16               ->text_body ("Hi,\ndie neue Foo ist da!")
17               ->transport ($transport)
18               ->send_or_die();
```

Wolfgang Kinkeldei

DDD Workshop bei Erlangen.PM

Hintergrund

Bereits im Jahr 2003 hat Eric Evans in seinem Buch "Domain Driven Design" [1] eine Vorgehensweise beschrieben, mit der größere IT-Projekte umgesetzt werden können. Durch den Einsatz von Entwurfs-Mustern mit klar definierten Verwendungs-Szenarien lässt sich ein großes Projekt dabei schnell in kleine Einzelteile zerlegen die dann ebenso problemlos implementierbar sind. In praktisch nachvollziehbarer Form wird dieses Vorgehen von Vaughn Vernon in seinem 2013 erschienenen Buch "Implementing Domain Driven Design" [2] vertieft und erweitert.

Die vordefinierten Muster regeln im großen Maßstab die Einbindung fremder Systeme sowie Schnittstellen zur Außenwelt und leisten Unterstützung bei der Einteilung der umzusetzenden Anforderungen in kleinere Einheiten. Im Detail-Maßstab wird die Aufteilung der Funktionalitäten in Klassen geregelt sowie die Strukturierung der erzeugten Klassen abgedeckt.

Sämtliche Klassen sind voneinander unabhängig und besitzen typischerweise keine gegenseitigen Wechselwirkungen. Das ermöglicht den einfachen Austausch einer Klasse durch eine alternative Implementierung mit gleichem Interface.

Zentraler Dreh- und Angelpunkt bei allen Aktivitäten rund um DDD ist eine allgemeingültige von allen Beteiligten gesprochene Sprache. Die darin verwendeten Begriffe müssen dem Jargon der Fachleute entsprechen, für die die das IT Projekt entwickelt wird. Die Bezeichnungen sämtlicher Aktoren, Aktivitäten, Eigenschaften und Anwendungsfälle müssen sich in identischer Schreibweise im Code und den Tests wiederfinden. Theoretisch muss ein Anwender ihm vorgelegte Code-Fragmente anhand der verwendeten Begriffe verstehen können.

Anhand eines kleinen Projektes haben wir uns bei Erlangen.PM an einem Samstag mit den Grundkonzepten von Domain Driven Design (kurz DDD) vertraut gemacht. Die Umsetzung eines Anwendungsfalles aus dem Beispielprojekt möchte ich nachfolgend zeigen.

Unser Projekt

Der Begriff "Projekt" ist eigentlich zu hoch gegriffen, denn der Umfang der Aufgabenstellung wäre bequem an einem Sonntag Nachmittag zu erledigen. Aber für Lernzwecke ist eine nicht zu umfangreiche Aufgabenstellung gerade richtig.

Wir wollen ein System erstellen, das Messwerte (z.B. Temperatur) von entfernten Sensoren entgegennimmt und speichert. Die gespeicherten Messwerte sollen dann in einer weiteren Ausbaustufe zu größeren Zeitintervallen (z.B. Stunde oder Tag) zusammengefasst werden, denn für die gegebene Anforderung sind für vergangene Zeiträume bestenfalls Minimal- oder Maximal-Werte relevant, die einzelnen Ablesungen hingegen nicht mehr. Immer wenn ein neuer Messwert eintrifft, soll darüber entschieden werden, ob ein Alarm ausgelöst werden muss. Über ein geeignetes Interface soll eine Anzeige der Messwerte der einzelnen Sensoren möglich sein.

Da Sprache das zentrale Element von DDD ist, fangen wir damit an, die Sprache zu entwickeln. Hätten wir einen Auftraggeber, so könnten wir den Großteil der Sprache vermutlich aus den Anforderungs-Dokumenten herauslesen, müssten jedoch für Durchgängigkeit und Widerspruchsfreiheit sorgen. In unserem Fall fällt die Sprache einfach aus. Da wir in Perl englische Schlüsselworte einsetzen, greifen wir ebenfalls zu Englisch als die Sprache der Wahl.

- A sensor provides a measurement result as an integer value.
- A sensor has a unique 3-part name (eg. 'rio/bad/temperatur') [sensor_id]
- A measurement consists of the result and the timestamp it was measured on.
- Every sensor knows its latest measurement.

Umsetzung

Bei so einfachen Anforderungen juckt es förmlich in den Fingerspitzen, eine Klasse "Sensor" zu erzeugen, die aus den notwendigen Attributen zur Aufnahme des Namens, des letzten Messwertes und einem Zeitstempel besteht. Jedesmal wenn ein neuer Messwert angeliefert wird, werden die notwendigen Attribute gesetzt. Fertig.

Aber das hat mit der fachlichen Logik unseres Projektes nichts zu tun. Beim Einsatz von DDD werden niemals von außen irgendwelche Attribute gesetzt. Der Grund dafür ist, dass das Setzen von Attributen nichts mit den umzusetzenden Anwendungsfällen zu tun hat. Blicken wir zurück auf die Definition unserer Sprache, so lautet die Formulierung bei unserem Anwendungsfall "provide measurement result". Damit unser Code diesen Fall abdeckt und verständlich macht, müssen wir eine Methode `provide_measurement_result` erzeugen, die zum Anliefern eines neuen Messwertes verwendet wird. Damit niemand aus Versehen doch Attribute verändert, könnten wir sie noch vor Überschreiben schützen.

In der Welt von DDD werden Dinge, die von außen angesprochen werden und identifizierbar sind (in unserem Fall über den eindeutigen Namen), in Form eines so genannten **Aggregate** implementiert. Der Code dafür könnte vereinfacht so aussehen. Um die Konzentration auf das Wesentliche zu lenken, sind sämtliche `package` Namen stark gekürzt, notwendige Abhängigkeiten oder Rollen fehlen teilweise und unwichtige Details sind ausgelassen worden.

```
package Sensor;
use Moose;
extends 'DDD::Aggregate';

has id => (
    is     => 'ro',
    isa    => 'SensorId', # ein Moose Typ
);

has latest_measurement => (
    is     => 'rw',
    isa    => 'Measurement',# ein Moose Typ
    writer => '_set_latest_measurement',
);

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

    $self->_set_latest_measurement(
      $result_or_value);
    $self->publish(
        MeasurementProvided->new(
            sensor_id   => $self->id,
            measurement =>
                  $self->latest_measurement
        )
    );
}
```

So einfach dieser Code auch aussieht, scheinen eine Menge an Funktionen fehlen. Unklar ist zunächst, wie diese Objekte entstehen. Ebenso festzulegen gilt es, wie und wohin die Daten dieser Objekte persistiert oder aus der gewählten Persistenz wieder zurückgewonnen werden. Und ohne ersichtlichen Grund wird eine `publish` Methode aufgerufen. Gehen wir die Punkte einfach der Reihe nach durch.

Damit unsere Software wartbar wird und Klassen leichter gegen anders implementierte Alternativen austauschbar sind, müssen wir uns an zahlreiche Prinzipien des Objektorientierten Designs [3] halten. Das Single Responsibility Principle fordert, dass eine Klasse nur für eine Aufgabe zuständig ist. Damit darf die Sensor Klasse nicht für ihre Erzeugung oder Belange der Persistierung zuständig sein. Insofern ist obiger Code absolut ausreichend für diese Klasse.

Was die Entstehung von Objekten angeht, so trennt DDD die Entstehung in eine initiale Konstruktion eines Objektes und das Wiedergewinnen einer Objekt-Instanz aus der gewählten Persistierung. Ersteres durch eine **Factory** Klasse übernommen, für letzteres ist ein **Repository** zuständig.

Eine Factory könnte (in unserem Fall extrem einfach) so aussehen:

```
package SensorCreator;
use Moose;
extends 'DDD::Factory';

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

    return Sensor->new(
        id => $sensor_id,
    );
}
```

Zugegeben, das war nicht gerade eine Herausforderung. Aber um der Trennung nach Verantwortungen gerecht zu werden, ist diese Klasse notwendig.

Nicht wesentlich schlimmer ist die Implementierung unseres Aggregates. Allerdings müssen wir hierbei sowohl das Wiedergewinnen (`by_name`) als auch das Speichern (`save`) umsetzen. In diesem Beispiel wählen wir die Persistierung als Dateien im JSON-Format. Die Logik im Hintergrund wird durch `MooseX::Storage` übernommen und ist in den Basisklassen definiert.

```
package AllSensors::File;
use Moose;
extends 'DDD::Repository';

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

    my $file =
       $self->_file($sensor_id)->stringify;
    return if !-f $file;

    return Sensor->load($file);
}

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

    $sensor->store($self->_file(
                  $sensor->id)->stringify);
}
```

Damit bleibt nur noch zu klären, was es mit der `publish` Methode in unserem Aggregat auf sich hat. Hier erfüllen wir die Forderung nach möglichst geringen Abhängigkeiten zwischen den einzelnen Bausteinen. Unsere Bausteine sind lose miteinander gekoppelt. Ein Aggregat löst bei jeder Veränderung seines Zustandes einen `DomainEvent` aus. Auf das Auftreten bestimmter Ereignisse warten beliebig viele (oder kein) `Service`, um bei übermittelten Zustandsänderungen weitere Aktionen auszuführen. Auf diese Weise können die hier nicht behandelten Erweiterungen der Messwert-Komprimierung sowie die Alarmierung bei gefährlichen Messwerten eingebunden werden. Die einzelnen Baugruppen wissen jeweils nichts voneinander.

Ein solcher Service könnte in etwa so aussehen:

```
package AlarmCheck;
use DDD::Service;

on MeasurementProvided => sub {
    my ($self, $event) = @_;

    $self->check_alarm(
        $event->sensor_id->name,
          $event->measurement
    );
};

sub check_alarm {
    my ($self, $sensor_id, $measurement)
       = @_;

    # TODO: prüfen und gegebenenfalls
    # Alarmierung
}
```

Zusammenspiel der Komponenten

So ansprechend das Ziel auch zunächst klingt, möglichst kleine jeweils nur für eine Aufgabe zuständige Klassen zu erstellen, irgendwann muss entschieden werden, welche Klasse in unserem Geflecht wofür eingesetzt wird. Auch muss ein Weg gefunden werden, die Objekte der von außen ansprechbaren Dienste zu instantiieren. Bei größeren Projekten kann das schon eine Herausforderung sein.

An dieser Stelle greifen wir auf `Bread::Board` zurück, indem wir eine DSL darauf aufbauen, die die DDD-Begrifflichkeiten als Schlüsselworte definiert.

In diesem vereinfachten Fall könnte unsere Domänen-Definition so aussehen:

```
package Domain;
use DDD::Domain;

has storage_dir => (
    is  => 'ro',
    isa => 'Path::Class::Dir',
);

aggregate 'sensor';

repository all_sensors => (
    isa          => 'AllSensors::File',
    dependencies => {
        dir => dep('/storage_dir'),
    },
);

factory 'sensor_creator';

service 'measurement';
```

Dabei entsteht eine Klasse, entsprechend der DDD-Schlüsselworte Attribute oder Methoden bereit stellt, mittels derer die diversen Bestandteile ansprechbar sind.

Immer dann, wenn wir Zugriff zu unserer Domänen-Logik benötigen, erzeugen wir einfach eine Instanz der `Domain` Klasse. Bei dauerhaft laufenden Prozessen wie z.B. einer Catalyst- oder Dancer-Anwendung ist es natürlich ratsam, diesen Schritt beim Start der Anwendung zu tun, um nicht wertvolle Zeit bei jedem Zugriff zu verlieren.

Wie wird eigentlich unsere Domänen-Logik benutzt? Das Lehrbuch schlägt vor, dass die API in Form einer `Application` genannten Schicht durch `Application-Service` Objekte abgedeckt wird. Lediglich diese werden von außen angesprochen und decken sämtliche Anwendungsfälle durch entsprechende Methoden ab. Außerdem kümmert sich diese Schicht um Transaktionen. Damit soll sichergestellt werden, dass sämtliche Daten immer konsistent gehalten werden. In unserem einfachen Fall genügt ein einziger Service "measurement". Auf Transaktions-Management verzichten wir zunächst.

```
package Measurement;
use DDD::Service;

sub provide_result {
    my ($self, $sensor_id, $result) = @_;

    my $domain = $self->domain;

    my $sensor = $domain->all_sensors
                    ->by_name($sensor_id)
        // $domain->sensor_creator
                    ->new_sensor($sensor_id);

    $sensor->
        provide_measurement_result($result);

    $domain->all_sensors->save($sensor);
}

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

    my $domain = $self->domain;

    return $domain->
       all_sensors->by_name($sensor_id);
}
```

Benutzung unserer Domänen-Logik

Die beiden Anwendungsfälle sind relativ einfach abzubilden. Zunächst die Anlieferung von Messwerten:

```
my $domain = Domain->new(
     storage_dir => '/path/to/storage');
$domain->measurement->provide_result(
           'rio/aussen/temperatur', 42);
```

Und ebenso einfach ist die Abfrage des letzten Messwertes:

```
my $domain = Domain->new(
    storage_dir => '/path/to/storage');
my $sensor = $domain->measurement->
    sensor_by_name('rio/aussen/temperatur');
say "Außentemperatur in Rio ist: ",
  $sensor->latest_measurement;
```

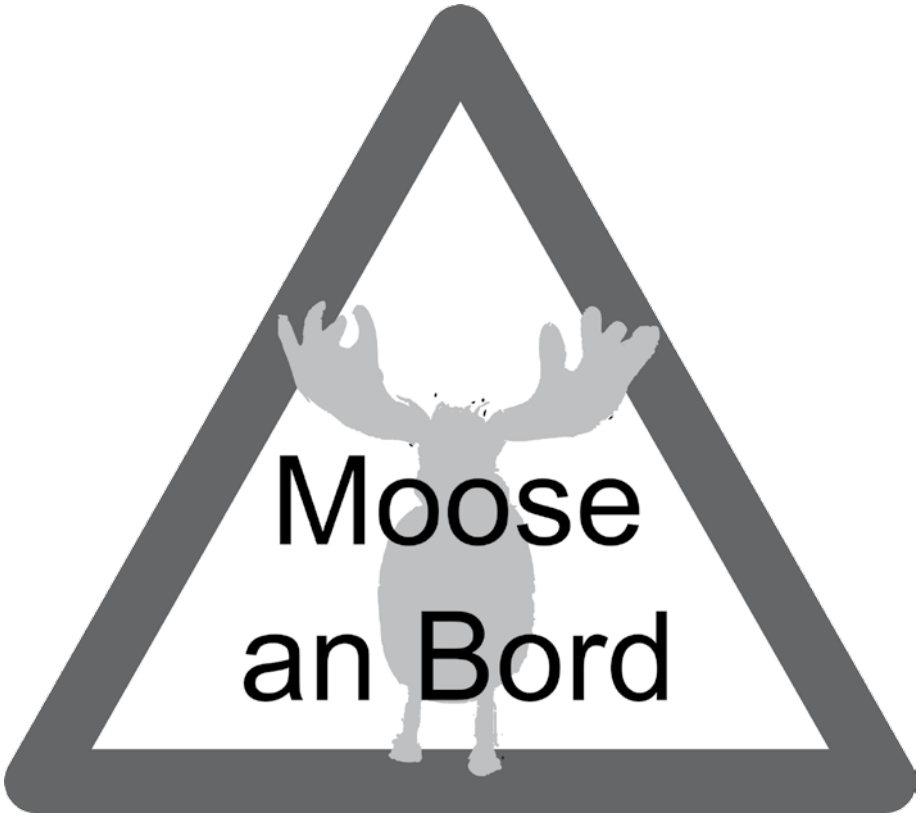
Fazit

Was im Kleinen etwas schwerfällig aussieht und umständlich wirkt, beinhaltet eine saubere und strukturierte Geschäftslogik. Durch die strikte Verwendung vorgegebener Muster und Strukturen erhält man automatisch aufgeräumten übersichtlichen Programm-Code, der durch die Größe der Klassen und die lose Koppelung sehr leicht auf annähernd beliebig viele Schultern (bzw. Köpfe) verteilbar und mit relativ einfachen Mitteln testbar ist.

Wer neugierig ist, kann gerne einen Blick auf die noch frühe Version meiner DDD-Basisklassen unter [4] werfen oder sich das Beispiel-Projekt [5] in vollem Umfang studieren. Anregungen, Kritik oder Verbesserungsvorschläge sind jederzeit willkommen, gerne kann der Workshop nochmals abgehalten werden.

Links

[1] http://www.domainlanguage.com/ddd|http://www.domainlanguage.com/ddd
[2] https://vaughnvernon.co/?page_id=168|https://vaughnvernon.co/?page_id=168
[3] http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod|http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
[4] https://github.com/wki/DDD|https://github.com/wki/DDD
[5] https://github.com/wki/DDD-StatisticsCollector|https://github.com/wki/DDD-StatisticsCollector



Perl-Services.de

Programmierung - Schulung - Perl-Magazin

info@perl-services.de

Colin Hotzky

Interview "Web, Systemadministrator und 'glue'"

Steffen Winkler ist Perl-Programmierer, Redner und Mitbegründer der Perl Mongers Erlangen. Seine Aussage: "Alles was ich bisher machen musste, ging mit Perl vorzüglich."

Vor circa vierzehn Jahren kam Steffen Winkler zum ersten Mal in Kontakt mit Perl. Programmierte er zuerst nur für sich privat, wurde es ihm schnell zu einer Passion. Sein Wissen reicht von regulären Ausdrücken, über Webanwendungen bis zur untersten Verwaltung der Daten. Als Perl Monk teilt er gerne seine Erkenntnisse mit anderen oder gibt Tipps und Ratschläge bei seinen Vorträgen beim Deutschen Perl Workshop.

Herr Winkler, können Sie sich noch an ihr erstes Perlprogramm erinnern?

Ja, ganz genau. Damals sollte ich spezielle Binärdaten einer SPS textuell visualisieren.

Sie haben bisher unzählige Programme geschrieben, sowie bei großen und kleinen Projekten mitgearbeitet. Hat sich dadurch ihr Programmierstil über die Jahre verändert?

Da ist nichts mehr, wie es einmal war. Man lernt nie aus. Bücher, Foren, Projekte und der Austausch mit anderen beeinflussen den Stil und verändern den Einsatz der Sprache.

Auf dem Markt gibt es unzählige Programmiersprachen. Warum bevorzugen Sie Perl vor allen anderen?

Alles was ich bisher machen musste, ging mit Perl vorzüglich. Fotos Bearbeiten. Musik schneiden. Logs lesen. Datenbanken abfragen. Natürlich auch schreiben. In einer Türenfabrik entwickelte ich eine Systemlandschaft, bei der es möglich war, zu jedem Zeitpunkt den Ort eines Produktes im Fertigungsprozess darzustellen. Dazu gehört die Steuerung einer Fräsmaschine, Auswertung der Logs eines Industriescanners und Lesen und Schreiben in unterschiedliche Datenbanken. Die Anwendung läuft seit Jahren fehlerfrei.

Für welche Aufgaben verwenden Sie Perl bei Ihrer derzeitigen Tätigkeit? Wie unterstützt sie Perl dabei?

Jetzt geht es im wesentlichen um Web. Aber auch im Bereich Middleware sind wir aktiv. Dort werden verschiedenste Services verbunden - eine wahre Freude mit Perl.

Für welche Aufgaben eignet sich Ihrer Meinung nach Perl am besten?

Web, Systemadministration und "glue" - Perl als Bindeglied zwischen heterogenen Systemen.

Sie sind sehr stark in der Perl Community unterwegs. 2005 gründeten Sie zusammen mit Richard Lippmann die Perl Mongers Erlangen. Wie kam es dazu?

Richard und ich kannten uns vom Perl-Workshop. In der Region Erlangen/Nürnberg sind durch die hohe Anzahl an IT-Firmen auch viele Perl-Entwickler unterwegs. So bot es sich an, eine Plattform zum Erfahrungsaustausch anzubieten. Bei den monatlichen Treffen geht es um Perl-Alltags-Themen, es gibt Themen-Vorträge und gemeinsame Workshop-Fahrten. Wenn ich Zeit habe, bin ich immer dort.

Ich sage immer: Selbst, wenn das Thema nicht so interessant war, habe ich wenigstens gut gegessen. Aber das stimmt natürlich nicht.

Auch Themen, die nicht direkt für mich entscheidend sind, stärken auf jeden Fall meine Allgemeinbildung. Und bei unseren Treffen kann man alles fragen, auch wenn es mit Perl nichts zu tun hat. Das finde ich wichtig.

Was raten Sie Neulingen, die zum ersten Perlerfahrungen sammeln?

Bücher lesen, nicht nur nachschlagen und natürlich ausprobieren.

Menschen treffen, sich vernetzen, sagen was man wie macht und so Zustimmung, Ablehnung oder Ratschläge bekommen.

Einzelkämpfer schreiben nach 'zig Jahren immer noch furchtbaren Code.

Infos zum Artikel

Steffen Winkler studierte Industrielle Elektronik mit der Spezialisierungsrichtung Mikrorechentechnik an der Ingenieurschule für Elektrotechnik und Maschinenbau in Lutherstadt-Eisleben.

Im Jahr 2000 fing er privat mit der Perlprogrammierung an und ist seitdem damit beruflich sehr erfolgreich.

Er gibt auch viel von seinem Wissen zurück an die Community. So ist er Mitbegründer der Perl Mongers Erlangen und betreibt ein eigenes kleines PPM-Repository. Regelmäßig sieht man ihm als Redner beim Deutschen Perl Workshop.

Zurzeit arbeitet er als Software-Entwickler und System-Administrator IT-Technik Andreas Specht.

Thomas Fahle

How To - Freien Speicherplatz auf einem Dateisystem ermitteln

Der freie Speicherplatz auf Festplatten bzw. Dateisystemen kann sowohl mit Filesys::Df als auch mit Filesys::DfPortable von Ian_Guthrie einfach ermittelt werden.

Filesys::DfPortable funktioniert unter Linux und Windows, während Filesys::Df nur mit unixoiden Betriebssystemen funktioniert, dafür aber ein bereits geöffnetes Datei Handle als Argument verarbeiten kann.

Filesys::Df

Die automatisch importierte Funktion `df` liefert eine Referenz auf einen Hash mit Informationen über die freien und belegten Blöcke des Dateisystems zurück.

Da nicht alle Dateisysteme Inodes unterstützen, ist es erforderlich, explizit zu fragen, ob Informationen zu Indodes vorliegen - siehe Listing 1.

Das Programm erzeugt z.B. folgende Ausgabe:

```
Total 1k blocks:             423301588
Total 1k blocks free:        230537836
Total 1k blocks avail to me: 209035272
Total 1k blocks used:        192763752
Percent full:                48
Total inodes:                26886144
Total inodes free:           26093840
Inode percent full:          3
```

Die Blockgröße lässt sich über den zweiten Parameter der Funktion `df()` angeben. Wer lieber in Bytes rechnet, verwendet einfach die Blockgröße 1 (siehe Listing 2).

Das Programm erzeugt z.B. folgende Ausgabe:

```
Total bytes:              433460826112
Total bytes free:         236072210432
Total bytes avail to me:  214053584896
Total bytes used:         197388615680
Percent full:             48%
Total inodes:             26886144
Total inodes free:        26093880
Inode percent full:       3
```

Filesys::Df kann auch Informationen über ein Dateisystem an Hand eines bereits geöffneten Dateihandles ermitteln. Das ist recht praktisch, wenn man wissen möchte, wie viel Platz für das gerade laufende Perl Programm ($0) noch zur Verfügung steht - siehe Listing 3.

Ausgabe wie oben.

Filesys::DfPortable

Filesys::DfPortable funktioniert, wie bereits gesagt, unter Linux und Windows und ist Filesys::Df ziemlich ähnlich.

Die automatisch importierte Funktion `dfportable()` liefert eine Referenz auf einen Hash mit Informationen über die freien und belegten Blöcke des Dateisystems zurück.

Die Blockgröße lässt sich über den zweiten Parameter der Funktion `dfportable()` angeben. Default Blockgröße ist dieses Mal 1, also Ausgabe in Bytes - siehe Listing 4.

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

use Filesys::Df;

# Default output is 1K blocks
my $df = df('/tmp');

if ( defined($df) ) {
    print 'Total 1k blocks:
           $df->{blocks}\n';
    print 'Total 1k blocks free:
           $df->{bfree}\n';
    print 'Total 1k blocks avail to me:
           $df->{bavail}\n';
    print 'Total 1k blocks used:
           $df->{used}\n';
    print 'Percent full:
           $df->{per}\n';

    # Only filesystems that support inodes
    if ( exists( $df->{files} ) ) {
        print 'Total inodes:
               $df->{files}\n';
        print 'Total inodes free:
               $df->{ffree}\n';
        print 'Inode percent full:
               $df->{fper}\n';
    }
}
else {
    warn 'Woops - something went wrong.\n';
}
```

Listing 1

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

use Filesys::Df;

my $df = df('/tmp', 1);    # output is bytes

if ( defined($df) ) {
    print 'Total bytes:
           $df->{blocks}\n';
    print 'Total bytes free:
           $df->{bfree}\n';
    print 'Total bytes avail to me:
           $df->{bavail}\n';
    print 'Total bytes used:
           $df->{used}\n';
    print 'Percent full:
           $df->{per}%\n';

    # Only for filesystems that
    # support inodes
    if ( exists( $df->{files} ) ) {
        print 'Total inodes:
               $df->{files}\n';
        print 'Total inodes free:
               $df->{ffree}\n';
        print 'Inode percent full:
               $df->{fper}\n';
    }
}
else {
    warn 'Woops - something went wrong.\n';
}
```

Listing 2

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

use Filesys::Df;

open( ME, '<', $0 ) or
      die 'Can't open myself $!';

# Get information for filesystem
# at file handle
my $df = df( \*ME, 1 );

if ( defined($df) ) {

    print 'Total bytes:
           $df->{blocks}\n';
    print 'Total bytes free:
           $df->{bfree}\n';
    print 'Total bytes avail to me:
           $df->{bavail}\n';
    print 'Total bytes used:
           $df->{used}\n';
    print 'Percent full:
           $df->{per}%\n';

    # Only for filesystems that
    # support inodes
    if ( exists( $df->{files} ) ) {
        print 'Total inodes:
               $df->{files}\n';
        print 'Total inodes free:
               $df->{ffree}\n';
        print 'Inode percent full:
               $df->{fper}\n';
    }
}
else {
    warn 'Woops - something went wrong.\n';
}
```

Listing 3

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

use Filesys::DfPortable;

# Display output in 1K blocks
my $dfp = dfportable('C:', 1024);

if ( defined($dfp) ) {
   print'Total 1k blocks:
         $dfp->{blocks}\n';
   print'Total 1k blocks free:
         $dfp->{bfree}\n';
   print'Total 1k blocks avail to me:
         $dfp->{bavail}\n';
   print'Total 1k blocks used:
         $dfp->{bused}\n';
   print'Percent full:
         $dfp->{per}\n';

   # Only filesystems that support inodes
   if ( exists( $dfp->{files} ) ) {
        print 'Total inodes:
               $dfp->{files}\n';
        print 'Total inodes free:
               $dfp->{ffree}\n';
        print 'Inode percent full:
               $dfp->{fper}\n';
   }

} else {
    warn 'Woops - something went wrong.\n';
}
```

Listing 4

Das Programm erzeugt z.B. folgende Ausgabe:

```
Total 1k blocks:              26109948
Total 1k blocks free:         4299136
Total 1k blocks avail to me: 4299136
Total 1k blocks used:         21810812
Percent full:                 84
```

Statt des Laufwerksbuchstabens können auch UNC_Pfade wie '\\\\Server\\Freigabe' verwendet werden.

Filesys::DfPortable unterstützt, wie Filesys::Df, Inodes - das ist meiner Ansicht nach unter Portabilitätsaspekten etwas sinnfrei, da Windows Inodes erst gar unterstützt.

Wer unter Windows einfach nur wissen möchte, wieviel Speicherplatz noch verfügbar ist, und daher nicht alle `bells and whistles` von Win32::DriveInfo benötigt, gewinnt mit Filesys::DfPortable ein wenig Portabilität.

Renée Bäcker

OTRS-Pakete testen und "verifizieren"

Das Thema "Testen" ist ein ganz wichtiges und immer und überall taucht es auf - wenn auch nicht immer ganz so offen. Auch wenn es darum geht, OTRS-Erweiterungen zu programmieren, kommt man ohne Tests nicht aus. Allerdings ist die Kultur des Testens dabei nicht ganz so weit verbreitet wie unter CPAN-Autoren.

Das hat zum einen damit zu tun, dass ein Großteil der Funktionalitäten über die Weboberfläche erreicht wird und OTRS (noch) keine wirklichen Oberflächentests bereitstellt und zum anderen hat es häufig damit zu tun, dass auf Kundenauftrag gehandelt wird und die Erweiterungen nicht "veröffentlicht" wird. Dadurch entfällt ein wenig der Druck Plattformneutral zu programmieren. Man hat ja in der Regel dann vorgeschrieben, für welches Betriebssystem, welche OTRS-Version und welches Datenbanksystem entwickelt wird.

Das ist aber kein idealer Zustand und daran soll sich etwas ändern. Für die Erweiterungen, die auf OPAR - einem Archiv für OTRS-AddOns im Stile des CPAN - geladen werden, soll es zukünftig automatisierte Testläufe geben. Dabei sollen verschiedene Aspekte geprüft werden. Nicht nur Funktionstests sollen laufen, sondern auch Metainformationen sollen gesammelt werden.

Ein kleiner Überblick:

- Einhalten von (OTRS-)Programmierrichtlinien
- Unittests sollen fehlerfrei durchlaufen
- Stabilität und Performanz der OTRS-Installation soll nicht beeinträchtigt werden
- Keine ungewünschte Kommunikation nach außen
- Sind Templates und Funktionalität (auch JavaScript) sauber getrennt
- Bleiben Artefakte nach der Deinstallation übrig

.. und noch viele weitere Sachen

Durch diese Tests und Metainformationen bekommen sowohl die Benutzer als auch die Programmierer jede Menge Hilfe. Die Benutzer sehen, welche Auswirkungen die Installation der Erweiterung auf das eigene System hat und die Entwickler bekommen Hinweise wenn die Erweiterung nicht ganz sauber umgesetzt ist.

Der Ablauf sieht dann so aus:

1. Der Programmierer lädt das Paket auf OPAR hoch
2. Das Paket wird in eine Queue gestellt, in der alle Pakete stehen die überprüft werden sollen
3. Das Paket wird analysiert um zu erfahren welche OTRS-Version(en) benötigt werden
4. Ist das Paket an der Reihe, wird eine Virtuelle Maschine erstellt
5. Auf der Virtuellen Maschine wird die benötigte Software installiert: OTRS, Perl-Pakete, Datenbank, etc.
6. Metainformationen werden gesammelt
7. Unittests von OTRS werden ausgeführt und Zeiten werden gemessen
8. Paket und dessen Abhängigkeiten werden installiert und nach jedem Schritt werden die Unittests von OTRS erneut ausgeführt
9. Pakete werden deinstalliert
10. Alle Ergebnisse werden gepackt und and OPAR geschickt
11. Information an Autor und Darstellung auf der Webseite

In diesem Artikel werden die einzelnen Schritte betrachtet wie sie umgesetzt wurden und wo es evtl. Probleme gab. Das System befindet sich derzeit noch in der Entwicklung. Der meiste Code davon ist auf Github im Account http://github.com/reneeb/ zu finden. Verbesserungsvorschläge bitte dort eintragen.

Die Schritte 1 und 2 sind bereits umgesetzt. OPAR ist eine Anwendung, die mit Hilfe von Mojolicious umgesetzt wurde. Die Queue ist als einfache Tabelle in der Datenbank umgesetzt, in der ein Cronjob alle x Minuten nachschaut ob es neue Pakete gibt die überprüft werden sollen. Bisher werden einfach Metainformationen aus dem Paket geholt wie Dokumentation, benötigte OTRS-Versionen, Abhängigkeiten, werden Programmierrichtlinien eingehalten. Das Sammeln der Metainformationen passiert im Moment noch auf dem Server, auf dem auch OPAR liegt. Um diesen aber zu entlasten, wird dieser Schritt auf die Virtuelle Maschine ausgelagert. Nur die Information der benötigten OTRS-Versionen wird hier noch gesammelt. Dazu später mehr.

Damit kommen wir schon zu Schritt 4: Für die Prüfung des Pakets wird eine Virtuelle Maschine erstellt. Die Entscheidung fiel zu Gunsten vom JiffyBox, den Cloud Servern von Domainfactory. Eigene Server sollten nicht vorgehalten werden, da auf OPAR (noch) nicht der ganz große Traffic zu finden ist und eigene Server damit zu teuer wären. JiffyBoxen werden hier im Unternehmen immer wieder eingesetzt um Abnahmesysteme für Kundenprojekte aufzusetzen. Damit ist der notwendige Account und das nötige Wissen vorhanden. Außerdem bietet JiffyBox eine nette API, für die es auch Perl-Module gibt (siehe auch $foo Ausgabe 27 - "VM ansteuern mit VM::JiffyBox").

Zum Aufsetzen und konfigurieren der Virtuellen Maschine wird Rex (http://rexify.org) eingesetzt. Damit bleibt alles Perl und Perl-Wissen ist in diesem Unternehmen genug vorhanden. Hinzu kommt noch, dass es für JiffyBox auch schon fertigen Code gibt, der eine solche JiffyBox erstellt.

Mit Rex können solche Maschinen sehr einfach erstellt und konfiguriert werden. Dazu kommen sogenannte *rexfiles* zum Einsatz. Das Ganze erinnert an `make` und die *Makefiles*. Für eine bessere Integration in das Gesamtsystem "OPAR" werden hier aber keine *rexfiles* erzeugt, sondern die Module aus Rex kommen direkt zum Einsatz.

Queueing

Nach dem Auslesen der Queue-Tabelle wird für jeden Eintrag ein fork gemacht:

```
# init fork manager
my $max_processes =
  $self->config->get( 'fork.max' );

my $fork_manager  =
  Parallel::ForkManager->new(
    $max_processes
  );

$logger->trace(
  'init fork manager with max ' .
   $max_processes . ' processes'
);

for my $tmpjob ( @job_info ) {

  # do the fork
  $fork_manager->start and next;

  # create job and run it
  my $job = OTRS::OPR::Daemon::Job->new(
      %{$tmpjob},
  );
  $job->run if $job;

  # exit the forked process
  $fork_manager->finish;
}

$fork_manager->wait_all_children;
```

Im Job selbst wird dann das Paket analysiert um zu erfahren, welche OTRS-Version auf der Virtuellen Maschine überhaupt benötigt wird. Dazu kann man das Paket `OTRS::OPM::Analyzer` verwenden:

```
use OTRS::OPM::Analyzer::Utils::OPMFile;

my $opm_file = '/path/to/package.opm';
my $opm      =
OTRS::OPM::Analyzer::Utils::OPMFile->new(
  opm_file => $opm_file,
);

my @otrs_versions = $opm->framework;
```

VMs einrichten

Für jede dieser OTRS-Versionen wird dann eine Virtuelle Maschine erzeugt:

```
use Rex;
use Rex::Commands::Cloud;

cloud_service 'Jiffybox';
cloud_auth $api_key;

for my $otrs ( @otrs_versions ) {
    my $box = cloud_instance create => {
        image_id => "ubuntu_12_4",
        name     => "test-$otrs",
        plan_id  => $plan_id,
        password => $root_pwd
    };

    cloud_instance start => $box->{id};
}
```

Auch das könnte man noch parallelisieren.

Im nächsten Schritt muss dann die ganze Software installiert werden. Wenn immer wieder dasselbe gemacht werden muss, kann man sich auch ein Paket für Rex schreiben. Die Schritte zur OTRS-Installation haben sich in den letzten Versionen nicht geändert. Grundsätzlich sieht es so aus, dass zuerst ein User "otrs" angelegt wird mit dem Heimverzeichnis */opt/otrs*.

```
create_user "otrs" =>
  home => '/opt/otrs/',
  groups => [ "www-data" ],
  no_create_home => 1;
```

Danach wird die passende OTRS-Version heruntergeladen, entpackt und nach */opt/otrs* geschoben.

```
my $tmp = "/tmp/otrs-$version.tar.gz";
if ( ! is_file $tmp ) {
    run "wget -O $tmp $otrs_url";
}

if ( ! is_dir '/opt/otrs/Kernel' ) {
    run "tar zxf $tmp -C /opt";
    unlink $tmp;
}
```

Die Basiskonfiguration muss angelegt werden, die Rechte müssen gesetzt werden und die Datenbank muss erstellt werden.

Für Standardaktionen wie Webserver, Datenbank und Perl-Pakete installieren gibt es schon fertige Befehle für Rex:

```
include qw/Rex::Database::MySQL;
  Rex::Database::MySQL::Admin;
  Rex::Database::MySQL::Admin::User;
  Rex::Database::MySQL::Admin::Schema/;

install "mysql-server";

Rex::Database::MySQL::Admin::
                          Schema::create({
  name => $param->{schema}->{name},
});

Rex::Database::MySQL::Admin::User::create({
  name => $param->{user}->{name},
  host => $param->{user}->{host},
  password => $param->{user}->{password},
  rights => $param->{user}->{rights},
  schema => $param->{schema}->{name}.".*",
});

for my $file ("otrs-schema.mysql.sql",
   "otrs-initial_insert.mysql.sql",
   "otrs-schema-post.mysql.sql") {
 file "/tmp/$file",
 source => "files/db/$otrs_version/$file";

 Rex::Database::MySQL::Admin::execute({
   sql => "/tmp/$file",
   schema => $param->{schema}->{name},
 });

 unlink "/tmp/$file";
}

install "apache2";
```

Damit ist die Virtuelle Maschine in dem Zustand, in dem das große Testen beginnen kann.

Metainformationen zum Paket sammeln

Der nächste Schritt ist das Sammeln der Metainformationen. Auch hier kommt das Paket `OTRS::OPM::Analyzer` zum Einsatz. Es parst zum einen die .opm-Datei -- die Informationen im XML-Format vor -- und zum anderen führt es je nach Konfiguration folgende Aufgaben aus:

- Ist die .opm-Datei wohlgeformt und valide
- Sind Unittests in dem Paket vorhanden
- Ist Dokumentation vorhanden
- Ist es eine Open Source Lizenz, unter dem das Paket steht
- Welche Abhängigkeiten hat das Modul
- Werden System-Aufrufe gemacht
- Hält sich der Programmierer an die Programmierrichtlinien
- Basistests für Templates

Diese einzelnen Aufgaben sind als Rollen für die Hauptklasse umgesetzt und wird entweder auf die einzelnen Dateien des Pakets losgelassen (z.B. die Überprüfung der Programmierrichtlinien) oder auf die .opm-Datei.

Eine Schwierigkeit besteht darin, dass keine Aussage über die Qualität von Dokumentation und Unittests abgegeben werden kann. Jedenfalls nicht automatisiert. Wann ist eine Dokumentation gut, wann nicht? Wenn die Dokumentation in einem Textformat vorliegt kann man vielleicht noch prüfen bzw. raten in welcher Sprache die Dokumentation vorliegt -- und freie Module für den internationalen "Markt" sollten in Englisch vorliegen. Aber ob die Dokumentation auch tatsächlich die Funktionalität des Moduls beschreibt lässt sich mit (einfachen) Mitteln nicht feststellen. Dazu sind auch zu wenig Informationen im Paket an sich enthalten was es denn macht. Da diese Schwierigkeiten im Moment nicht behoben werden können, wird nur überprüft ob überhaupt etwas vorhanden ist. In der abschließenden Bewertung der Qualität des Pakets haben diese Punkte auch nicht den ganz großen Stellenwert.

Bei OPAR wird -- genau wie beim CPAN -- nicht von Qualität gesprochen, sondern von Kwalitee. Die phonetische Ähnlichkeit soll auch daraufhin deuten, dass es sich nicht um eine absolute Qualität handelt sondern um einen maschinelle Auswertung.

Jede der Rollen liefert einfach das Ergebnis der Prüfung zurück. Die Ergebnisse aller Prüfungen werden gesammelt und später verarbeitet.

Für die Prüfung ob die .opm-Datei wohlgeformt und valide ist, wurde eine XSD-Datei auf Basis der bekannten Pakete und des Codes im OTRS-Code entwickelt. Gegen diese Schema wird die Datei dann mit Hilfe von `XML::LibXML` und `XML::LibXML::Schema` geprüft:

```
my $parser = XML::LibXML->new;
my $tree   = $parser->parse_file(
  $self->opm_file,
);

$self->tree( $tree );

# check if the opm file is valid.
try {
    my $xsd = do{ local $/; <DATA> };
    XML::LibXML::Schema->new(
      string => $xsd
    );
  }
catch {
    $self->error_string(
      'Could not validate against XML '
      . 'schema: ' . $_
    );
};
```

Programmierrichtlinien einhalten

Die Programmierrichtlinien werden mit `Perl::Critic` geprüft. Die Regeln dazu stehen teilweise in der Entwicklerdokumentation von OTRS, teilweise sind sie aus dem offiziellen OTRS-Code hergeleitet. Leider gibt es keine kompletten offiziellen Programmierrichtlinien. Für OTRS gibt es auf CPAN auch eine Sammlung dieser Richtlinien in dem Paket `Perl::Critic::OTRS`.

`Perl::Critic` arbeitet mit `PPI` für die statische Analyse des Quellcodes. Damit wird ein Baum aus den einzelnen Token aufgebaut und für jeden dieser Knoten können Regeln implementiert werden. Dazu muss man sich sowohl Positiv- als auch Negativbeispiele für Quellcode aufschreiben. Z.B. soll ein `push` auf `@ISA` nicht erlaubt sein. Was weiterhin erlaubt sein soll ist jedoch das `push` auf `@ISA` eines anderen Pakets und die direkte Zuweisung.

Damit hat man folgende Positivbeispiele:

```
@ISA = qw(ParentClass);
@ISA = ('ParentClass');

my $caller = caller(0);
push @{"$caller\::ISA"}, 'ParentClass';
push(@{"$caller\::ISA"}, 'ParentClass');
```

Und folgende Negativbeispiele:

```
push @ISA, 'Data::Dumper';
push( @ISA, 'Data::Dumper' );
CORE::push @ISA, 'Data::Dumper';
CORE::push( @ISA, 'Data::Dumper' );
```

Die schaut man sich genau an. Was macht `PPI` aus diesen Codestücken und lässt sich daraus eine Regel erstellen? Um sich anzuschauen was `PPI` mit den Codestücken macht, kann man `PPI::Dumper` verwenden:

```
use PPI;
use PPI::Dumper;

my $code = join '', <STDIN>;
my $doc  = PPI::Document->new( \$code );
my $dump = PPI::Dumper->new( $doc );
$dump->print;
```

Ein Ausschnitt aus der Baumstruktur ist in Listing 1. zu sehen:

```
PPI::Document
  PPI::Token::Whitespace        '    '
  PPI::Statement
    PPI::Token::Symbol    '@ISA'
    PPI::Token::Whitespace      ' '
    PPI::Token::Operator        '='
    PPI::Token::Whitespace      ' '
    PPI::Token::QuoteLike::Words
                            'qw(ParentClass)'
    PPI::Token::Structure       ';'
  PPI::Token::Whitespace        '\n'
  PPI::Token::Whitespace        '    '
  PPI::Statement
    PPI::Token::Symbol    '@ISA'
    PPI::Token::Whitespace      ' '
    PPI::Token::Operator        '='
    PPI::Token::Whitespace      ' '
    PPI::Structure::List        ( ... )
      PPI::Statement::Expression
        PPI::Token::Quote::Single
                              ''ParentClass''
    PPI::Token::Structure       ';'
  PPI::Token::Whitespace        '\n'
  PPI::Token::Whitespace        ' \n    '
```

Listing 1

Nach der Analyse kann man sich überlegen wie die Regel schließlich aussieht. Steht die Regel fest, kann man eine Regel für `Perl::Critic` schreiben. Die Regel für den oben gezeigten Code ist in Listing 1 zu finden. Die typische Regel für `Perl::Critic` setzt 5 Methoden um:

```
use Perl::Critic::Utils qw{ :severities :classification :ppi };
use base 'Perl::Critic::Policy';

use Readonly;

our $VERSION = '0.01';

Readonly::Scalar my $DESC => q{Use of "push @ISA, ..." is not allowed};
Readonly::Scalar my $EXPL => q{Use RequireBaseClass method of MainObject instead.};

sub supported_parameters { return; }
sub default_severity     { return $SEVERITY_HIGHEST; }
sub default_themes       { return qw( otrs ) }
sub applies_to           { return 'PPI::Token::Word'  }

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

  return if $elem ne 'push' and $elem ne 'CORE::push';

  my $sibling = $elem->snext_sibling;
  return if !$sibling->isa( 'PPI::Token::Symbol' ) and !$sibling->isa(
                                                  'PPI::Structure::List' );

  if ( $sibling->isa( 'PPI::Token::Symbol' ) ) {
      return if $sibling ne '@ISA';
  }
  elsif ( $sibling->isa( 'PPI::Structure::List' ) ) {
      my $symbol = $sibling->find( 'PPI::Token::Symbol' );

      return if !$symbol;
      return if $symbol->[0] ne '@ISA';
  }

  return $self->violation( $DESC, $EXPL, $elem );
}
```

Listing 2

- supported_parameters
- default_severity
- default_themes
- applies_to
- violates

Über `supported_parameters` kann festgelegt werden, welche Parameter in der `Perl::Critic`-Konfiguration für diese Regel eingestellt werden können. `Perl::Critic` arbeitet mit verschiedenen Leveln -- also verschiedene Schweregrade -- von Regeln. Welches Level die Regel im Standard hat wird über `default_severity` festgelegt. Um für verschiedene Projekte `Perl::Critic` einsetzen zu können, gibt es die sogenannten *Themes*. Es ist wahrscheinlich, dass für OTRS-Projekte andere Regeln gelten als für Projekte bei anderen Kunden. Teilweise werden die auch genau in die gegengesetzte Richtung gehen. Damit sich solche Regeln nicht in die Quere kommen, können diese mit den Themes aktiviert bzw. ausgeschlossen werden.

Da in diesem Projekt aber nur OTRS-Regeln umgesetzt werden, bekommen die alle den gleichen Level und das gleiche Theme zugewiesen. Die interessanten Methoden sind als `applies_to` und `violates`. `applies_to` muss eine Liste an Knotennamen liefern, auf diese Regel angewandt werden soll. In diesem Fall prüft `Perl::Critic` den Code immer dann auf diese Regel wenn es auf einen Knoten vom Typ *PPI::Token::Word* trifft.

In der `violates`-Methode wird dann die Regel ausprogrammiert. Und hier ist dann auch die Analyse der Positiv- und Negativbeispiele wichtig. Wenn der Code gegen die Regel verstößt, muss eine Beschreibung und eine Erklärung zu der Regel zurückgegeben werden, genauso wie der Knoten der den Regelverstoß ausgelöst hat.

Noch sind nicht alle Regeln für OTRS umgesetzt, aber das folgt im Laufe der Zeit.

Damit sind die ganzen Metainformationen gesammelt und das Paket ist auf die Einhaltung der Programmierrichtlinien überprüft. Damit sind wir beim nächsten Schritt: Die Unittests von OTRS müssen ausgeführt und die Zeiten gemessen werden.

Unittests ausführen und Zeiten messen

OTRS liefert ein Unittestscript mit. Dieses liefert leider kein TAP, das Ergebnis kann also nicht mit den Standard-Perl-Tools ausgewertet werden. Um nicht gleich in das OTRS eingreifen zu müssen, wurde ein Parser für die Ausgabe geschrieben, der die Daten auswerten kann. Das Format mit den meisten Informationen, die OTRS bietet ist das XML-Format.

Das Skript von OTRS erlaubt es auch, nur einzelne Testskripte laufen zu lassen, aber hier soll ja grundsätzlich alles getestet werden.

Zur Zeitmessung ist noch eines zu sagen: Es kann nicht einfach die Zeit der gesamten Testsuite genommen werden, da Pakete auch neue Testdateien hinzufügen. Und es ist nicht überraschend wenn zusätzliche Tests die Gesamtdauer der Testausführung vergrößert. Man würde also eine Warnmeldung bekommen dass das Paket die OTRS-Instanz ausbremst obwohl das gar nicht der Fall ist.

Insgesamt gilt, dass man versuchen muss, möglichst viele Störfaktoren auszuschließen. Die Virtuelle Maschine sollte bei einem neuerlichen Lauf der Testsuite nicht auf einmal unter Volllast stehen, da das die Tests natürlich ausbremst.

```
use Capture::Tiny qw(:all);
use OTRS::Unittest::XMLParser;

my ($out, $err) = capture {
    system '/opt/otrs/bin/otrs
       UnitTest.pl -o XML';
};

my $results =
        OTRS::Unittest::XMLParser->new(
    xml => $out,
);

for my $test ( @{ $result->tests } ) {
    say sprintf "%s -> %s",
           $test->name, $test->duration;
}
```

In OPAR selbst werden nur Informationen darüber benötigt, welche Unittests fehlschlagen und wie groß der Geschwindigkeitsverlust bzw. -gewinn für den jeweiligen Test ist. Diese Daten werden auf der Platte abgelegt und für die finale Auswertung bereitgehalten.

Das Paket und dessen Abhängigkeiten installieren

Ein weitaus größeres Problem ist das Paket und dessen Abhängigkeiten zu installieren. Der Paketmanager von OTRS gibt das nicht her, also wird auch hier neuer Code benötigt. Eines der Probleme ist wie die Abhängigkeiten aufgelöst werden können. Man muss dabei beachten, dass es verschiedene Quellen der Pakete geben kann: OPAR, OTRS oder ein Unterordner von OTRS.

Ein CPAN-Client hat den Vorteil, dass es wirklich ein einziges anerkanntes Repository für Pakete gibt. Das fehlt in der OTRS-Community. Außerdem muss der OTRS-Paketmanager auch berücksichtigen welche OTRS-Version im Einsatz ist und muss bei der Installation von Paketen ggf. Änderungen an der Datenbank vornehmen oder Perl-Code ausführen.

Zusätzliche Schwierigkeit besteht darin, dass es zwei Arten von Abhängigkeiten geben kann: Weitere OTRS-Erweiterungen oder Perl-Module.

Der Ablauf der Installation sieht dann folgendermaßen aus:

1. Hole die Liste der Perl-Abhängigkeiten
2. Installiere die Perl-Abhängigkeiten
3. Hole die Liste der OTRS-Abhängigkeiten
4. Prüfe für jede OTRS-Abhängigkeit aus welcher Quelle sie kommt
5. Prüfe für jede OTRS-Abhängigkeit welches OPM-Paket für die vorliegende OTRS-Version benötigt wird
6. Beginne bei 1. für das neue OTRS-Paket
7. Installiere das OTRS-Paket

Die Liste der Perl-Abhängigkeiten ist in der .opm-Datei zu finden. Um diese zu parsen wird wieder `OTRS::OPM::Analyzer` verwendet.

```
use OTRS::OPM::Analyzer::Utils::OPMFile;

my $opm_file = '/path/to/package.opm';
my $opm      =
OTRS::OPM::Analyzer::Utils::OPMFile->new(
  opm_file => $opm_file,
);
my @perl_deps = grep{
  $_->{type} eq 'CPAN';
} $opm->dependencies;
```

Zuerst wird noch geprüft, ob eine Installation überhaupt notwendig ist:

```
for my $cpan_dep ( @perl_deps ) {
    my $module  = $cpan_dep->{name};
    my $version = $cpan_dep->{version};

    eval "use $module $version" and next;
}
```

Für die Installation der Perl-Abhängigkeiten wird `cpanm` verwendet, das in Schritt 4 installiert wird. Um die Ausgabe von `cpanm` überprüfen zu können, wird die Ausgabe mit Hilfe von `Capture::Tiny` eingefangen:

```
use Capture::Tiny ':all';

my ($out, $err, $exit) = capture {
  system 'cpanm', $module;
};
```

Wenn die Installation des Perl-Moduls fehlschlägt, wird auch die Installation des OTRS-Pakets abgebrochen:

```
if ( $out !~
   m{Successfully installed $dist} ) {
  die "Installation of dependency failed!";
}
```

Damit sind die Perl-Abhängigkeiten abgehandelt. Die OTRS-Abhängigkeiten sind der kompliziertere Part an diesem Schritt. Die unterschiedlichen Quellen bieten jeweils eine XML-Datei an, in der die verfügbaren Pakete aufgelistet sind. Darin sind auch die OTRS-Versionen vermerkt, für die das jeweilige Paket zur Verfügung steht.

Aus den drei bekannten Quellen werden diese Informationen an zentraler Stelle gesammelt:

```
use OTRS::Repository;

my $repository = OTRS::Repository->new(
  sources => [
    'http://ftp.otrs.org/pub/otrs/itsm/
         packages33/otrs.xml',
    'http://opar.perl-services.de/otrs.xml',
    'http://ftp.otrs.org/pub/otrs/packages/
         otrs.xml',
  ],
);
```

`OTRS::Repository` parst diese XML-Dateien und macht die Informationen im Repository-Objekt verfügbar. Danach muss noch die passende OPM-Datei gefunden werden:

```
my $uri = $repository->find(
  name => $name,
  otrs => $otrs_version,
);
```

Mit `HTTP::Tiny` wird die Datei dann geholt:

```
my $response = HTTP::Tiny->new
                          ->get( $uri );
```

Und dann geht das Spiel von vorne los. Nach jeder Installation eines OTRS-Pakets müssen die Unittests wie oben beschrieben ausgeführt und ausgewertet werden. Damit wird sichergestellt, dass Fehler oder Beeinträchtigungen nicht schon durch Abhängigkeiten in das OTRS gebracht werden. Wenn es eine Erweiterung unter mehreren Quellen gibt, dann wird die Installation abgebrochen, da nicht eindeutig ist, welche Erweiterung denn installiert werden soll.

Bei der Installation an sich müssen die in der OPM-Datei enthaltenen Dateien in das OTRS eingespielt und ggf. noch Datenbankänderungen vorgenommen oder Perl-Code ausgeführt werden. Die Änderungen an der Datenbank sind ebenfalls direkt in der OPM-Datei enthalten (siehe Listing 3).

Auch der Perl-Code, der ausgeführt werden soll ist direkt in der OPM-Datei enthalten (siehe Listing 4)

Diese werden mit OTRS-Mitteln in SQL umgesetzt und ausgeführt. Dazu bedient sich das Modul den OTRS-Kernmodulen (Listing 5).

Meldung an OPAR

Die Ergebnisse der ausführlichen Tests müssen auch wieder zurück ins OPAR fließen, um den Autor benachrichtigen und die Ergebnisse auf der Webseite darstellen zu können. Im Laufe der Tests wurden die Ergebnisse immer wieder in ein temporäres Verzeichnis gespeichert. Die ganzen Daten, die

```
use Moo;
use IO::All;

has manager => (is => 'ro', builder =>  1);
has otrs    => (is => 'ro', required => 1);

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

  my $opm < io( $path );

  $self->manager->PackageInstall(
      String => $opm,
  );
}

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

  push @INC, $self->otrs;

  try {
      require Kernel::Config;
      require Kernel::System::Main;
      require Kernel::System::Encode;
      require Kernel::System::Log;
      require Kernel::System::DB;
      require Kernel::System::Time;
      require Kernel::System::Package;
  }
  catch {
      die "Can't load OTRS modules!";
  };

  my %objects =
    ( ConfigObject => Kernel::Config->new );
  $objects{EncodeObject} =
    Kernel::System::Encode->new( %objects );
  $objects{LogObject}    =
    Kernel::System::Log->new( %objects );
  $objects{MainObject}   =
    Kernel::System::Main->new( %objects );
  $objects{DBObject}     =
    Kernel::System::DB->new( %objects );
  $objects{TimeObject}   =
    Kernel::System::Time->new( %objects );

  my $package_object =
   Kernel::System::Package->new( %objects );
  return $package_object;
}
```

Listing 5

```
<DatabaseInstall Type="post">
    <TableCreate Name="ps_quick_close">
        <Column Name="id" Required="true" AutoIncrement="true" Type="INTEGER" PrimaryKey=
                                                                              "true"/>
        <Column Name="close_name" Required="true" Type="VARCHAR" Size="250"/>
        <Column Name="comments" Required="true" Type="VARCHAR" Size="250"/>
        <Column Name="subject" Required="false" Type="VARCHAR" Size="250"/>
        <Column Name="create_by" Required="true" Type="INTEGER"/>
        <Column Name="change_by" Required="true" Type="INTEGER"/>
        <ForeignKey ForeignTable="users">
            <Reference Local="create_by" Foreign="id" />
            <Reference Local="change_by" Foreign="id" />
        </ForeignKey>
</DatabaseInstall>
```

Listing 3

in diesem Verzeichnis gesammelt wurden, werden in einem JSON-Objekt gesammelt und an OPAR übertragen. Sollten Auffälligkeiten dabei sein, wird der Autor informiert damit er gegensteuern kann.

Auf der Webseite wird über eine Ampel dargestellt für wie risikoreich OPAR die Installation des Pakets bewertet. Durch einen Klick auf diese Ampel können Interessierte genau sehen was zu der dargestellten Einschätzung führt. Vielleicht ist das die einzelnen Personen uninteressant wenn z.B. die Dokumentation fehlt oder nur in einer anderen Sprache als Englisch vorliegt.

In Abbildung 1 ist das Gesamtsystem dargestellt. Vom Upload des Pakets über das Queueing und das Erstellen der Virtuellen Maschine bis zur Ausführung der Tests und Aggregation der Ergebnisse.

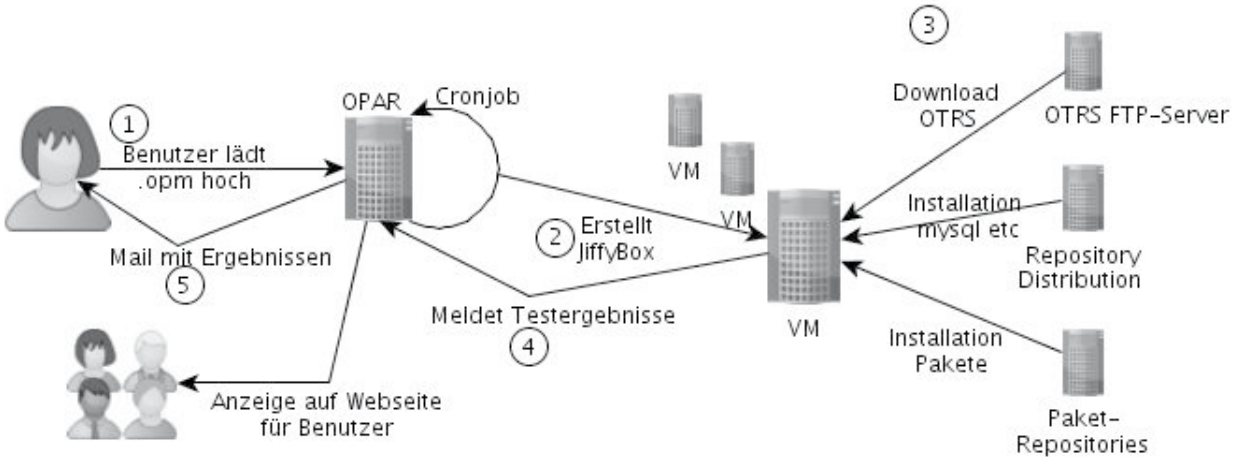


Abbildung 1

```
<CodeInstall Type="post"><![CDATA[
    my $FunctionName = 'CodeInstall';

    my $CodeModule = 'var::packagesetup::' . $Param{Structure}->{Name}->{Content};

    if ( $Self->{MainObject}->Require($CodeModule) ) {
        # Create new instance
        my $CodeObject = $CodeModule->new( %{$Self} );

        if ( $CodeObject ) {

            # start method
            if ( !$CodeObject->$FunctionName(%{$Self}) ) {
                $Self->{LogObject}->Log(
                    Priority => 'error',
                    Message => "Could not call method $FunctionName() on $CodeModule.pm",
                );
            }
        }

        #error handling
        else {
            $Self->{LogObject}->Log(
                Priority => 'error',
                Message => "Could not call method new() on $CodeModule.pm"
            );
        }
    }
]]>
</CodeInstall>
```

Listing 4

CPAN News XXIX

DateTime::Moonpig

Für ihr Abrechnungssystem brauchten Ricardo Signes und Mark Jason Dominus verschiedene Zeitfunktionen. Das Ganze sollte Objektorientiert sein. Der aktuelle Standard dafür ist `DateTime`. Mit diesem Modul haben die zwei aber verschiedene Probleme, die MJD ganz gut in seinem Blog unter http://blog.plover.com/prog/Moonpig.html auflistet.

Aus diesem Grund haben die beiden `DateTime::Moonpig` geschrieben. Das Modul erbt dabei von `DateTime` und kennt daher dessen Methoden. An einigen wichtigen Punkten wurden Änderungen vorgenommen: Die Addition und die Subtraktion wurde überladen um einfacher mit Daten rechnen zu können und einige *setter*-Methoden wurden "abgeschaltet", weil sie den Zeitwert verändern würden.

```
$birthday = DateTime::Moonpig->new(
  year   => 1969,
  month  =>    4,
  day    =>    2,
  hour   =>    2,
  minute =>   38,
);

$now = DateTime::Moonpig->new( time() );

# returns number of seconds difference
printf "%d\n", $now - $birthday;

# one minute later
$later   = $now + 60;
# two hours earlier
$earlier = $now - 2*3600;

if ($now->follows($birthday)) { ... }
if ($birthday->precedes($now)) { ... }
```

Im Gegensatz zu `DateTime` liefert `DateTime::Moonpig` die Zeit in UTC und nicht in der "floating" Zeitzone. In Anwendungen ist es immer besser, mit einer fixen Zeitzone zu rechnen und diese sollte UTC sein. Damit lassen sich die Zeitangaben sehr einfach für den jeweiligen Nutzer der Anwendung anpassen.

PerlIO::via::Timeout

Liest man Daten aus einer Quelle und möchte man vermeiden, dass das Lesen aus dieser Quelle den weiteren Programmablauf zu sehr blockiert, dann kann man mit `PerlIO::via::Timeout` arbeiten. Das kann ganz praktisch sein, wenn Daten von einem Socket gelesen werden sollen, die Gegenstelle aber nicht rechtzeitig Daten auf die Verbindung schreibt.

```
use Errno qw(ETIMEDOUT);
use PerlIO::via::Timeout qw(:all);
open my $fh, '<:via(Timeout)', 'foo.html';

# set the timeout layer to be 0.5
# second read timeout
read_timeout($fh, 0.5);

my $line = <$fh>;
if ($line == undef && 0+$! == ETIMEDOUT) {
  # timed out
  ...
}
```

Regexp::Common::time

Im Artikel über Regexp::Debugger wurde es schon kurz gesagt: Wenn man lesbare Reguläre Ausdrücke haben möchte, dann kann man evtl. auf eigene Reguläre Ausdrücke verzichten und stattdessen ein Modul hinzuziehen. Ein solches Modul ist `Regexp::Common::time`. Wie sich schon vom Namen her erahnen lässt, stellt das Modul einige Reguläre Ausdrücke zur Verfügung, mit denen man Zeiten matchen kann.

```
use Regexp::Common::time;

if ( $mailbody =~ m/($RE{time}{mail})/ ) {
    print "Found date in RFC2822-style:
              $1\n";
}

if ( $text =~ m/$RE{time}{iso}/ ) {
    print "The text contains a date
              in ISO 8601 format\n";
}
```

Action::Retry

Wenn man eine Aktion durchführt und bei einem Misserfolg die Aktion nochmal durchführen will, kann man entweder eine Endlosschleife nutzen und im Erfolgsfall die Schleife beenden oder man kann ein Modul wie `Action::Retry` verwenden. Das Modul führt die Aktion aus und wenn die Aktion erfolglos war wird einfach ein `sleep` gemacht und danach die Aktion erneut ausgeführt. Das geht so lange bis die Aktion erfolgreich war.

Man könnte damit z.B. etwas umsetzen, das sich versucht in einem Mailaccount anzumelden. Im Fehlerfall soll der Benutzer nach einem neuen Passwort gefragt und dann erneut versucht werden, sich anzumelden. Oder man versucht aus einer Datei zu lesen, die existiert aber vielleicht noch gar nicht und man muss warten bis ein anderes Programm diese Datei erstellt hat.

Ein einfaches Beispiel

```
use Action::Retry;

my $action = Action::Retry->new(
  attempt_code => sub {
      open my $fh, '<', '/pfad/datei.txt'
         or die $!;
      while ( my $line = <$fh> ) {
          print $line;
      }
      close $fh;
  },
);

$action->run();
```

Wenn die Datei nicht existiert und Perl beim `open` abbricht, wartet `Action::Retry` 1 Sekunde und führt dann den Code erneut aus. Das geht so lange bis die Datei gelesen werden kann.

Sollen die Zeiträume zwischen den Versuchen verlängert werden, kann man eine andere Strategie wählen. Statt *Constant* kann man *Fibonacci* oder *Linear* wählen:

```
use Action::Retry;

my $action = Action::Retry->new(
  strategy     => 'Fibonacci',
  attempt_code => sub {
      open_file_and_read();
  },
);

$action->run();
```

Bei der Strategie kann man auch eine Hashreferenz übergeben um diese zu konfigurieren. So kann man auch bestimmen, wie viele Versuche es maximal geben soll:

```
use Action::Retry;

my $action = Action::Retry->new(
  strategy     => {
      'Fibonacci' => {
          max_retries_number => 5,
      },
  },
  attempt_code => sub {
      open_file_and_read();
  },
);

$action->run();
```

Weiterhin gibt es die Möglichkeit, Code auszuführen wenn die Aktion schiefgelaufen ist:

```
use Action::Retry;

my $action = Action::Retry->new(
  on_failure_code => sub {
      print "Vielleicht eine andere ",
        "Datei ausprobieren...";
  },
  attempt_code => sub {
      open_file_and_read();
  },
);

$action->run();
```

Und wenn das Programm nicht abbricht bei der Aktion wird es im Standard als Erfolg anerkannt, egal ob die Aktion wirklich erfolgreich war oder nicht. Für diesen Fall kann man *retry_if_code* setzen:

```
use Action::Retry;

my $action = Action::Retry->new(
  retry_if_code => sub {
      my ($error, $hashref) = @_;
      my $m = $hashref->{attempt_result};

      if ( $m =~ /cannot open/ ) {
          return 1;
      }

      return 0;
  },
  attempt_code => sub {
      open_file_and_read();
  },
);

$action->run();
```

Hier ist es so, dass ein wahrer Wert das Modul veranlasst die Aktion erneut auszuführen und ein unwahrer Rückgabewert von *retry_if_code* heißt, dass die Aktion erfolgreich war.

Scalar::In

In Perl 5.10 wurde der Smartmatch eingeführt, mittlerweile ist dieser aber als "experimental" gekennzeichnet. Mit `Smart::Match` gibt es bereits ein Modul, das ein konsistentes Smartmatch bereitstellt. `Scalar::In` bietet zwei Funktionen, die einen Teil der Smartmatch-Funktionalität abbilden sollen.

```
use Scalar::In 'string_in';
string_in( undef, undef ); wahr
string_in( undef, 'A' ); # unwahr
string_in( 'A', undef ); # unwahr
string_in( 'A', 'A' ); # wahr
string_in( 'ABCDEFG', sub { 0 == index shift, 'ABC' } ); # wahr
string_in( 'A', $object ); # wahr wenn $object stringifiziert "A" ist
string_in( 'A', @{[ 'A' .. 'C' ]} ); # wahr
string_in( 'A', @{[ 'B' .. 'C' ]} ); # unwahr
string_in( 'A', @{[ qr{ \A [BC] \z }xms ]} ); # unwahr
string_in( undef, %{{ A => undef }} ); # unwahr
string_in( 'A', %{{ A => undef, B => undef, C => undef }} ); # wahr, Schlüssel "A" vorhanden
string_in( 'A', %{{ B => undef, C => undef }} ); # unwahr
```

Time::List

Wer immer eine Liste mit Daten haben möchte, die zwischen zwei Zeitpunkten liegen, der sollte sich `Time::List` anschauen. Bei der Instanziierung des Objekts kann man angeben, in welchen Schritten man die Daten haben möchte und in welchem Format Eingabe- und Ausgabedaten vorliegen.

```
use Time::List;
$timelist = Time::List->new(
    input_strftime_form => '%Y-%m-%d %H:%M:%S',
    output_strftime_form => '%Y-%m-%d %H:%M:%S',
    time_unit => DAY ,
    output_type => ARRAY ,
);

my ($start_time , $end_time , $array );
$start_time = "2013-01-01 00:00:00";
$end_time = "2013-01-05 00:00:00";
$array = $timelist->get_list($start_time , $end_time); # 5 elements rows
# "2013-01-01 00:00:00" ,"2013-01-02 00:00:00",
# "2013-01-03 00:00:00","2013-01-04 00:00:00","2013-01-05 00:00:00",

$start_time = "2013-01-01 00:00:00";
$end_time = "2013-01-01 04:00:00";
$timelist->time_unit(HOUR);
$array = $timelist->get_list($start_time , $end_time); # 5 elements rows
# "2013-01-01 00:00:00" ,"2013-01-01 01:00:00","2013-01-01 02:00:00",
# "2013-01-01 03:00:00","2013-01-01 04:00:00",
```

So ist es auch möglich, einfach zwischen zwei Formaten zu wechseln:

```
$timelist = Time::List->new(
    input_strftime_form => '%Y-%m-%d %H:%M:%S',
    output_strftime_form => '%d.%m.%Y %H:%M:%S',
    time_unit => DAY ,
    output_type => ARRAY ,
);
```

Termine

Februar 2014

01./02. FOSDEM
04. Treffen Hannover.pm
Treffen Frankfurt.pm
05. Treffen Niederrhein.pm
06. Treffen Dresden.pm
17. Treffen Erlangen.pm
18. Treffen Hannover.pm
26. Treffen Berlin.pm

März 2014

04. Treffen Hannover.pm
Treffen Frankfurt.pm
06. Treffen Dresden.pm
12. Treffen Niederrhein.pm
13.-16. QA-Hackathon
17. Treffen Erlangen.pm
18. Treffen Hannover.pm
18.-21. FOSSGIS
27./28. Deutscher Perl-Workshop

April 2014

01. Treffen Hannover.pm
Treffen Frankfurt.pm
03. Treffen Dresden.pm
09. Treffen Niederrhein.pm
15. Treffen Hannover.pm
21. Treffen Erlangen.pm
25. Niederländischer Perl-Workshop
29. Treffen Hannover.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

LINKS

http://www.perl-nachrichten.de

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

http://www.perl-community.de

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

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

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

http://www.perlfoundation.org

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

http://www.Perl.org

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

http://perl-academy.de

Moderne Objektorientierung
Reguläre Ausdrücke für Könner
Webentwicklung mit Mojolicious
Perl::Critic und Programmierrichtlinien