Ausgabe 2/2012 – www.foo-magazin.de - 6,00 €

$foo

PERL MAGAZIN

XML und Perl

XML:: Compile

Perl 5.16

Was ist neu?

App-Entwicklung

mit Moose, MooseX::Declare, MooseX::App::Cmd
und weiteren Freunden aus dem MooseX::* Namensraum

Nr 22

Die OTRS Gruppe ist der Hersteller der OTRS-Produkt-Suite und innovativer Servicelösungen. Wir bieten Beratung, Softwareentwicklung, Support und Managed Services für OTRS. Die OTRS Gruppe ist mit Büros in Nordamerika, Europa, Lateinamerika und Asien vertreten.
Zur Verstärkung unseres Teams suchen wir bundesweit schnellstmöglich mehrere

OTRS Developer (w/m)

zur Festanstellung.

Sie realisieren individuelle Auftrags- und Produktentwicklungen für unsere Kunden und arbeiten mit an der stetigen Weiterentwicklung unserer Produkte.
Sie bringen mit:

- gute Perl Kenntnisse
- Erfahrung mit LAMP Systemen
- Erfahrung mit Software OTRS sowie mit Modulentwicklung und Architektur
- Gute Kenntnisse in Bezug auf Directories und Datenbanken: LDAP, AD, MySQL, Oracle, SQL Server
- Erfahrung im Einsatz von modernen Web Technologien: AJAX, JSON, Java Script, HTML
- Erfahrung im Umgang und mit der Administration von Linux Systemen

Neben den beschriebenen Fähigkeiten bringen Sie Teamgeist und Integrationsfähigkeit mit und verfügen über gute Englisch-Kenntnisse. Von der Open Source Idee sind Sie genauso begeistert wie wir.
Bei der OTRS AG haben Sie die Möglichkeit, Ihre berufliche Zukunft zu gestalten und sich maßgeblich an der strategischen Entwicklung des Unternehmens zu beteiligen. Wir bieten Ihnen herausfordernde und anspruchsvolle Aufgaben in einem jungen und dynamischen Team. Ihre Standorte können Straubing/Süddeutschland und/oder Bad Homburg/Rhein-Main-Gebiet sein. Oder Sie werden aus Ihrem Home-Office für uns tätig.

Kontakt:

Sabine Riedel, Tel.: 06172 681988-53 oder schicken Sie uns Ihre aussagekräftige Bewerbung per Mail ausschließlich an jobs@otrs.com

VORWORT

Zuwachs für zwei Familien

Es ist doch immer wieder schön, wenn es Zuwachs in den Familien gibt. Und heute können wir Zuwachs in gleich zwei Familien feiern:

Erstens in der Perl-Familie. Bei Redaktionsschluss dieser Ausgabe war Perl 5.16 zwar noch nicht veröffentlicht, aber die Perl 5 Porters waren gerade heftigst dabei, die letzten Fehler zu beseitigen. Mit Perl 5.16.0 gibt es wieder ein paar Neuerungen und Bugfixes, von denen die wichtigsten Punkte in einem Artikel in dieser Ausgabe betrachtet werden. Was in dem Artikel aber nicht zu finden ist, das sind die vielen kleinen Verbesserungen und Bugfixes, die in den letzten Jahren in den Perl-Kern geflossen sind. Hier macht sich die intensive Arbeit von Dave Mitchell und Nicholas Clark bezahlt, die aus dem Spendentopf für Perl 5 Geld für ihre Arbeit bekommen.

Aber man darf die vielen fleißigen Perl 5 Porters nicht vergessen, wie z.B. Father Chrysostomos und James E Keenan, die den Request Tracker gesäubert haben indem sie alte Bugmeldungen mit neueren Perl-Versionen getestet haben. Im `perldelta` von Perl 5.16.0 sind die vielen Helfer aufgelistet. Danke für eure Arbeit!

Auch bei $foo gibt es Zuwachs: Im Juli wird es die erste Ausgabe unseres englischsprachigen Perl-Magazins "PerlMag" geben. Dort gibt es von vielen Autoren interessante Artikel, wobei es (fast) keine Überschneidungen mit $foo geben wird. Mit "PerlMag" füllen wir die Leere, die nach "The Perl Journal" und "The PerlReview" entstanden ist. Wir hoffen, dass das Magazin gut angenommen wird und wir der Perl-Community einiges zurückgeben können.

Ansonsten suchen die Frankfurt Perlmongers auch noch einen Ausrichter für den nächsten Deutschen Perl-Workshop. Als diese Ausgabe in den Druck gegangen ist, hatte sich noch keine Perlmonger-Gruppe bereiterklärt, die Veranstaltung 2013 durchzuführen. Es gibt viele schöne Städte in Deutschland, die ich mal (wieder) besuchen würde. Sollte sich hier jemand finden, der den Workshop ausrichten möchte, dann bitte eine kurze Mail an vorstand@frankfurt.pm

Ab der nächsten Ausgabe möchten wir die CPAN-News etwas umgestalten. In jeder Ausgabe sollen je zwei Module etwas ausführlicher - d.h. jeweils 1 Seite - und die restlichen vier Module wie bisher vorgestellt werden. Über die Module, die ausführlicher dargestellt werden sollen, darf jeder abstimmen. Dazu werden wir immer ca. 2 Monate vor dem Erscheinen der nächsten Ausgabe eine Abstimmung auf der Webseite starten...

Jetzt aber viel Spaß mit der neuen Ausgabe des Perl-Magazins.

Renée Bäcker

Die Codebeispiele können mit dem Code

5lgmcr4

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/issue22

IMPRESSUM

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

INHALTSVERZEICHNIS

ALLGEMEINES

MODULE

ANWENDUNGEN

PERL

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.

Herbert Breunung

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

Daniel Bruder

Freiberuflicher Programmierer aus München und Liebhaber der Sprache Perl seit Beginn seines Studiums der Computerlinguistik. Auf dem Weg ein "richtiger" Perl-Hacker zu werden.

Ulli Horlacher

Ulli "Framstag" Horlacher arbeitet als UNIX-Admin und -Programmierer am Rechenzentrum der Universität Stuttgart. Seine Lieblingssprache entdeckte er vor 20 Jahren mit Perl 3 unter VMS. Internet, Serverbetriebssysteme und Serverdienste sind seine Arbeitsschwerpunkte. Gelegentlich hält er auch (Perl-)Kurse an der Universität Stuttgart. Weitere Perl-UNIX-Software von ihm ist unter *http://fex.rus.uni-stuttgart.de/fstools/* zu finden. Seine Freizeit verbringt er meistens mit Fahrrad- bzw. Tandemfahren und dem Bau von innovativer Illuminationshardware: *http://tandem-fahren.de/Mitglieder/Framstag/LED/*

Mark Overmeer

Mark Overmeer ist ein fleißiger Programmierer - mit einigen großen Perl Projekten. Auf CPAN findet man Mail::Box --for automatic email processing-- und XML::Compile --XML processing--. Außerdem arbeitet er an CPAN6, dem Nachfolger des CPAN Archivs. Mark hat einen Abschluss als Master in Computing Science und arbeitet zur Zeit als selbständiger Programmierer, Systemadministrator und Trainer. Er ist Geschäftsführer von NLUUG (früher bekannt als die Niederländische UNIX User Group), SPPN (Niederländische Perl Promotion Foundation) und Oophaga (CAcert Equipment). Und das neben vielen anderen Aktivitäten. Mehr Informationen unter *http://solutions.overmeer.net*.

Herbert Breunung

WxPerl-Tutorial - Teil 10: Editorkomponenten

Untertitel dieser Folge ist *Mächtige Widgets 2* und es wird noch einen dritten Teil dieser Miniserie geben, der über `Wx::Grid`, `Wx::PropertyGrid`, `Wx::Ribbon`, `Wx::HTML` und `Wx::XRC` zu berichten weiß. Widgets Nummer zwei und drei sind gänzlich frisch und kamen erst diesen Monat mit Version 0.995 des Moduls `Wx` hinzu (*http://www.wxperl.co.uk/binaryoasis/2012/03/wxwidgets-293.html*).

Nach dem absehbaren Ende dieses Tutorials in der Ausgabe 4/2012 wird es wahrscheinlich zu einem Buch ausgeweitet. chromatic sicherte zu, es als *Modern Perl Book* zu publizieren und der öffentliche Speicherort steht ebenfalls bereits fest (*http://bitbucket.org/lichtkind/wxperlbook*). Jede Mitarbeit und Kritik ist willkommen.

TextCtrl

Für die einfachen Texteingaben gibt es die `TextCtrl`, doch ihre Anwendung hat mehrere kleine Tücken. Bei der ersten Benutzung könnte ein übergroßer *Wx::Caret* (Textcursor) aufblinken. Das liegt daran, dass dieses Widget einzeilige oder mehrzeilige Texte darstellt, je nachdem, ob im Stil `wxTE_MULTILINE` enthalten ist (Parameter: Elternelement, ID, Inhalt, Position, Größe, Stil). Wird kein Stil angegeben, gilt der einzeilige Modus, in dem die Zeile die Höhe des Widgets ausfüllt und ebenso der Caret. Deshalb sollte man diesem Widget in dem Zustand niemals eine vertikale *proportion* oder eine feste Höhe zuweisen, da es selbst am besten weiß, wie hoch die Zeile ausfällt.

Auch eine vertikale Scrollbar sollte man nicht erwarten. Einreihige Ausgaben bekommen generell keine, selbst wenn der Text länger als das Widget wird. Und wer `wxTE_MULTILINE` verwendet, muss noch `wxTE_DONTWRAP` dazugeben, damit nicht einfach am rechten Ende umgebrochen wird und die logischen Zeilen unkenntlich werden. Dann allerdings taucht die Scrollbar auf, falls sie wirklich benötigt wird. Auch vertikale Scrollbarren erscheinen nur bei Bedarf.

Eine weitere Merkwürdigkeit hat der Modus für Passworteingaben (alle Buchstaben werden als Punkt angedeutet). Er wird durch die Konstante `wxTE_PASSWORD` aktiviert, die nur im einzeiligen Modus einen Effekt hat. Mit `wxTE_READONLY` vermag der Nutzer den Inhalt erwartungsgemäß nicht mehr verändern. Der Programmierer kann sehr wohl dann noch die Methoden `ChangeValue`, `AppendText`, `WriteText` (am Caret einfügen) und `Remove` verwenden. Nur wird das Versprechen, dass der Zustand des Textes danach als unverändert gilt (abzufragen per `IsModified`), nach eigenen Versuchen unter Windows nicht erfüllt. `DiscardEdits` als Gegenteil von `MarkDirty` kann aushelfen. Auch einige andere, wie zum Beispiel die `wxTE_PROCESS_TAB` betreffenden Angaben sind weitestgehend nichtig.

Der letzte interessante Stil ist `wxTE_PROCESS_ENTER`. Er bewirkt, dass ein Drücken der Entertaste mit dem Ereignis `EVT_TEXT_ENTER` abfangbar wird. Ruft der Callback `$_[1]->Skip`, wird danach noch `EVT_TEXT` aktiv, wie beim Drücken jeder anderen Taste auch, die den Textinhalt verändert. Für komplexere Tastatursteuerungen sollte man auf `EVT_KEY_DOWN` reagieren, siehe Folge 6. Es sei auch an die in Folge 7 genauer erläuterten Validatoren erinnert. Durch eine lange Zeile lassen sich von vornherein bestimmte Inhalte verbieten. Das folgende Beispiel erlaubt nur ASCII-Buchstaben:

```
use Wx::Perl::TextValidator;
$text->SetValidator (
    Wx::Perl::TextValidator->new (
        qr/[a-zA-z]/
) );
```

Bei anspruchsvolleren Lösungen kann von `Wx::Perl::TextValidator` geerbt werden, die Methode `Validate` überschrieben und darin die Eingabe durch `return 0;` abgelehnt werden. Wollte man es nur als Ereignis nutzen und während der Validierung andere Aktionen starten, darf

```
return $self->SUPER::Validate( @_ );
```

nicht fehlen. (`$self` bitte mit `shift` aus `@_` ziehen, da `SUPER::Validate` nichts damit anfangen kann.) Gleiches gilt für `TransferFromWindow` (vor `Validate`) und `TransferToWindow` (nach `Validate`). Soll einfach nur die Länge der Eingabe begrenzt werden, reicht `SetMaxLength`. Uneinsichtige Nutzer könnte man auf die Überschreitung des Limits hinweisen, wenn `EVT_TEXT_MAXLEN` anschlägt.

Der praktische Funktionsumfang endet mit `LoadFile` und `SaveFile`. Man könnte Texte auch im begrenzenden Maße gestalten, indem man ausgewählte Stücke wie folgt formatiert.

```
$text->SetStyle(6,12, Wx::TextAttr->new (
    Wx::Colour->new(0,0,222),
    Wx::Colour->new(200,200,200),
    Wx::Font->new(
        10, wxFONTFAMILY_DEFAULT,
        wxFONTSTYLE_ITALIC,
        wxFONTWEIGHT_BOLD, 0
    )
) );
```

Grenzen der TextCtrl

Das sieht auch noch im Drag'n-Drop-Popup gut aus, aber unter Windows ist von der Farbenpracht nichts mehr übrig. Für mehr als einfachen schwarzweißen Text sollte es schon `RichTextCtrl` sein, denn `TextCtrl` mit `wxTE_RICH` ist nur eine Windows-Geschichte. Die Arbeitsteilung zwischen beiden Widgets ist recht klar und sinnig. Einzeilig und längenbegrenzt ist die `RichTextCtrl` von sich aus nicht und kann mit Konstanten auch nicht dazu gebracht werden. (Aber es ließe sich ein einfacher Validator schreiben der dafür sorgt.)

Ein echter Nachteil der `TextCtrl` ist, dass die vom Programmierer gesetzte Textauswahl unter Windows nicht sichtbar ist und unter Linux anders aussieht als das vom Nutzer Markierte. Ein `undo` und `redo` dürften Anwender jedoch am meisten vermissen (nicht so in der `RichTextCtrl`). Das wiegt umso schwerer, als die `TextCtrl` die Basis weiterer Widgets ist, denen die gewohnte Reaktion auf *Strg+Z* auch fehlt. Es betrifft die Zellen des `Grid`, die bereits vorgestellte `ComboBox` und jede weitere `ComboCtrl` wie etwa die `SpinCtrl`, ein Textfeld für Zahlen mit zwei Knöpfen zum Vergrößern und Verkleinern der Zahl (`SpinButton`).

Es berührt leider auch die schicke, neue `SearchCtrl`, die wie aus einem aktuellen Browser entnommen aussieht. Der Knopf zum Start der Suche ist von Anfang an da, der zum Löschen des Inhalts ist per `ShowCancelButton(1)` zuschaltbar (es gibt auch `ShowSearchButton`). Die zusätzlichen Ereignisse `EVT_SEARCHCTRL_SEARCH_BTN` und `EVT_SEARCHCTRL_CANCEL_BTN` reagieren auf die grauen Knöpfe und sollten auch genutzt werden, denn ohne

```
EVT_SEARCHCTRL_CANCEL_BTN(
    $search, -1, sub {$search->Clear});
```

tut das Kreuz gar nichts.

Ein weiteres, kombiniertes Widget ist der `Wx::Perl::BrowseButton`. Der Namensraum `Wx::Perl` deutet an, dass es in Perl geschrieben ist und wohl nicht zum CPAN-Modul `Wx` gehört, wie auch in diesem Fall. Zum gleichen Zweck ließe sich auch die schon vorgestellte `FilePickerCtrl` verwenden, doch der auf dem Knopf angezeigte Name der ausgewählten Datei lässt sich nun mal nicht so gut kopieren und bearbeiten wie die `TextCtrl` des `Wx::Perl::BrowseButton`. Beide verwenden auch unterschiedliche Systemdialoge.

RichTextCtrl

Die Fähigkeiten dieses Widgets reichen bereits in den nächsten Teil hinein, wenn es um Layoutmöglichkeiten ohne *Sizer* geht. Mit etwas weniger Aufwand als vorher gezeigt kann der Text gefärbt, eingedickt, unterstrichen und eingerückt werden. Anstatt jedem unterschiedlichen Stück ein neu zu definierendes `TextAttr` zu geben (was weiterhin möglich ist), macht die Methode `ApplyBoldToSelection` die zuvor mit `SetSelection($von, $bis)` gesetzte Auswahl fett. Der einfachste Weg hier einen formatierten Text zu hinterlassen, ist jedoch ein Weg, der dem Schreiben in eine `DC` sehr ähnelt (Folge 7). Durch Befehle, die mit "Begin" anfangen wie `BeginAlignment` und `BeginFontSize` setzt man etwa Bündigkeit und Größe der Schrift und fügt Text mit `WriteText` ein. `WriteImage` baut sogar Bilder (`Wx::Image`) ein. Zwischendurch setzt man Farbe oder

Einrückung neu oder beendet Modi mit `EndItalic`. Ein `EndSuppressUndo` am Ende (nach `BeginSuppressUndo`) sorgt dafür, dass dieser Text vom Anwender nicht änderbar ist. All diese Zustände lassen sich auch später im laufenden Betrieb durch Tasten oder Knöpfe ausgelöst starten um einen kleinen Textprozessor zu erhalten. Einrückungen setzt man mit `BeginLeftIndent` und um einen Absatz mit zwei nummerierten Stichpunkten um 10mm eingerückt anzufangen, schreibt man:

```
$rich->BeginNumberedBullet(2, 100);
```

`BeginSymbolBullet('-', 100, 60);` verwendet entsprechend Zeichen statt Zahlen.

Sehr praktisch kann es werden, dass der formatierte Inhalt in Pufferobjekten speicherbar ist und dadurch zwischengelagert werden kann oder in ein anderes Widget übertragbar oder anfügbar ist. Beim Laden und Speichern bitte darauf achten, `wxRICHTEXT_TYPE_ANY` nach dem Dateinamen anzugeben und nicht `wxTEXT_TYPE_ANY`, das bei der `TextCtrl` noch ausreichend war.

Dass die `RichTextCtrl` für härtere Einsätze ausgelegt ist sieht man unter anderem an den Methoden `Freeze` und `Thaw`, mit denen sich steuern lässt, wann das Widget vom Renderer neu gezeichnet wird (nach `Thaw`). So wird ein Flackern während umfangreicher Operationen verhindert.

Die Druckunterstützung ist hervorragend. Es ist nahezu trivial eine Druckvorschau zu erhalten oder den formatierten Text auszudrucken. Dies wird beim nächsten Mal genauer gezeigt, da `Wx::HTML` über beinahe identische Fähigkeiten verfügt.

Scintilla

Sobald es um Quelltext geht, reicht selbst die `RichTextCtrl` nicht. Kaum ein Programmierer verzichtet heute noch gerne auf Syntaxhervorhebung, das farbliche Kennzeichnen semantisch unterschiedlicher Teile des Quelltextes. Dafür hat WxWidgets das bekannte *Scintilla* als `Wx::STC` verpackt. Diese von Neil Hodgson entworfene und betreute Komponente ist nicht allein das Herz der in Perl geschriebenen Editoren *Kephra* und *Padre* sondern auch von *Komodo*, *Notepad++*, *Geany*, *Anjunta* und vieler, vieler mehr [1].

Da `STC` lange nicht mehr erneuert wurde, beschloss der unermüdliche Padre-Entwickler Ahmad M. Zawawi eine aktuelle Version von Scintilla als `Wx::Scintilla` ins CPAN zu stellen. Ein Wechsel verlangt seit der neusten Version keine Änderung im Quellcode mehr, außer:

```
use Wx::STC;
our @ISA = 'Wx::StyledTextCtrl';
```

zu

```
use Wx::Scintilla;
our @ISA = 'Wx::ScintillaTextCtrl';
```

Jedoch kann nur eines der beiden Module von einem Script geladen werden. Am besten leitet man eine eigene Klasse davon ab. Bei massiven Klassen wie diesen lohnt ein OOP-Interface. In der `new`-Methode steht dann:

```
my $self = $class->SUPER::new($parent, -1);
```

Im Gegensatz zu den ersten beiden Textfeldern, braucht es bei Scintilla etwas Aufwand bevor man es wirklich benutzen möchte. Zuallererst wäre die Definition der Farben vorzunehmen. Scintilla verwendet fest einkompilierte Lexer, was es einerseits sehr schnell macht, aber andererseits nur wenig Kontrolle übrig lässt. Lediglich die Farben und Schlüsselworte können geändert werden. Wie das geht, zeigt die Dokumentation von `Wx::Scintilla` [2] oder die Quellen von Kephra [3], welche den flexibleren `StyleSetSpec`-Befehl verwenden. Beide sind aber nicht ideal, da man um vollständig änderungsbeständig zu sein, statt der Zahlen im ersten Parameter die benannten Konstanten verwenden sollte [4].

Zum guten Aussehen gehört auch eine ansehnliche Schriftart, die ich darin gefunden habe:

```
my $wx_font = Wx::Font->new(
    10, wxDEFAULT, wxNORMAL, wxNORMAL,
    0, 'DejaVu Sans Mono'
);
$self->StyleSetFont(
    wxSTC_STYLE_DEFAULT, $wx_font
) if $wx_font->Ok > 0;
# 'Courier New' ist auch nicht schlecht
```

Ein Trick besteht darin zuerst die Schrift zu setzen, dann die Ränder und als drittes die Farben für das Syntax-Highlighting. Scintilla kennt neben dem zusätzlichen Rand, mit dem der Text links und rechts auf dem meist weißen Feld eingerückt ist

```
    $self->SetMargins( $links, $rechts );
```

drei Ränder, die frei belegt werden können. Einen für Zeilennummern, einen für Lesezeichen und Marker und einen für das Code-Falten. Gezählt wird von links nach rechts und die Breite wird auch gesetzt.

```
$self->SetMarginWidth( $nr,  $pixel );
$self->SetMarginSensitive( $nr, 1 );
$self->SetMarginType(
                $nr, wxSTC_MARGIN_NUMBER );
```

Die zweite Zeile verhindert, dass ein Klick auf den Rand den daneben stehenden Text markiert, die dritte macht aus dem Rand einen für Zeilennummern. `wxSTC_MASK_FOLDERS` hätte den Rand für Faltmarker reserviert und `0x01FFFFFF` für jegliche Marker (Lesezeichen, Debugger und mehr). Weil auch schöne Ränder das Auge erfreuen (färbt alle drei Randarten, nur der Faltrand wird aufgehellt):

```
$self->StyleSetForeground (
    wxSTC_STYLE_LINENUMBER,
    Wx::Colour->new(123, 123, 137 )
);
$self->StyleSetBackground (
    wxSTC_STYLE_LINENUMBER,
    Wx::Colour->new(1226, 226, 222 )
);
```

Da es zu einer elenden Probiererei ausufern kann, eine gute Farbe zu finden, baute ich mir einmal eine vielseitige Hilfsfunktion, die Namen, aber auch verschiedene Zahlenformate annehmen kann [5]. Um mich wohl zu fühlen braucht es noch einige Einstellungen mehr:

```
# der Caret blinkt einmal pro Sekunde
$self->SetCaretPeriod( 500 );
# er ist zwei Pixel dick
$self->SetCaretWidth( 2 );
$self->SetCaretForeground(
        Wx::Colour->new( 0, 0, 255 ) );
$self->SetCaretLineBack(
        Wx::Colour->new( 245, 245, 161 ) );
$self->SetCaretLineVisible(1);
# Farben der Textauswahl
$self->SetSelForeground(
       1, Wx::Colour->new( 243, 243, 243 ) );
$self->SetSelBackground(
       1, Wx::Colour->new( 0, 17, 119 ) );
# sichtbare Leerzeichen und Tabs
$self->SetWhitespaceForeground(
       1, Wx::Colour->new( 204, 204, 153) );
$self->SetViewWhiteSpace(1 );
# rechter Rand nach 80 Zeichen
$self->SetEdgeColour(
       Wx::Colour->new( 200, 200, 255) );
$self->SetEdgeColumn( 80 );
$self->SetEdgeMode( wxSTC_EDGE_LINE );
```

Ganz entscheidend ist die einzig wahre Größe der Tabs. Diese muss dreifach gesetzt werden. Einmal die Größe der Tabs, dann die Größe der Einrückung und zuletzt die Abstände der gepunkteten Linien, welche die Einrückungsebenen andeuten. Der letzte Befehl bestimmt ob überhaupt Tabs verwendet werden (1) oder eine Anzahl ($size) an Leerzeichen.

```
$self->SetTabWidth( $size );
$self->SetIndent( $size );
$self->SetHighlightGuide( $size );
$self->SetUseTabs( 0 ); # oder 1
```

Der Umfang von Scintilla ist riesig und wird nicht einmal von den meisten Editoren voll ausgeschöpft. Wegen der vielen Methoden und Konstanten ist die WxWidgets-Dokumentation hier recht unübersichtlich. Die offizielle Scintilla-Dokumentation [6] ist zwar schön thematisch sortiert, aber ganz auf C ausgerichtet und daher immer noch umständlich zu gebrauchen. Die Bequemste ist immer noch die alte, für WxPython geschriebene Yellowbrain-Dokumentation [7].

Aber eine schöne und einfach zu realisierende Sache ist die Klammerhervorhebung. Darunter versteht man das Einfärben von zugehörigen Klammern, wenn der Caret sie berührt. Eine wichtige Zutat dafür ist die Funktion:

```
$self->BraceMatch($pos);
```

Das Resultat ist die Position der anderen Klammer des Paares, oder -1 wenn es keine gibt. Scintilla kann nach folgenden Klammern suchen: ( ) [ ] { } < >. Nun braucht man nur beide Positionen zu färben:

```
$self->BraceHighlight( $match, $pos );
$self->BraceHighlight( -1, -1 );
```

Letzteres schaltet die Lichter wieder ab, was er tut wenn beide Werte gleich sind. Wurde eine einzelne Klammer entdeckt, sollte auch das gemeldet werden:

```
$self->BraceBadLight( $pos );
```

Damit das wirklich funktioniert müssen auch dazu Farben bestimmt werden.

```
$self->StyleSetForeground(
          wxSTC_STYLE_BRACELIGHT, $color );
$self->StyleSetBackground(
          wxSTC_STYLE_BRACELIGHT, $color );
$self->StyleSetForeground(
          wxSTC_STYLE_BRACEBAD, $color );
$self->StyleSetBackground(
          wxSTC_STYLE_BRACEBAD, $color );
```

Um die aktiven Klammern noch besonders hervorzuheben sollen sie fett werden.

```
$self->StyleSetBold(
         wxSTC_STYLE_BRACELIGHT, 1 );
$self->StyleSetBold(
         wxSTC_STYLE_BRACEBAD,   1 );
```

Wer gut aufgepasst hat, weiß, dass sich hier drei Befehle zu einem `StyleSetSpec` zusammenfassen lassen. Für die Luxusfassung dieses Features soll auch die Einrückungsmarkierung aufleuchten. In ordentlichem Code wird sie direkt die Klammern verbinden und die zugehörige Klammer rückt noch einfacher ins Blickfeld.

```
$self->StyleSetForeground(
  wxSTC_STYLE_INDENTGUIDE, $color );
```

Das Folgende wird auch ausgeführt, wenn es passende Klammern gibt:

```
my $indent = $self->GetLineIndentation
    ( $self->LineFromPosition( $pos ) );
$self->SetHighlightGuide( $indent )

# schaltets wieder aus
$self->SetHighlightGuide(0)
```

Eine große Anzahl unterstützter Sprachen und weitere Funktionen wie Autovervollständigung, mehrfach farbige Textauswahl, Bilder, Darstellung unsichtbarer Zeichen, Macrorecorder, Tastenbelegung sowie frei setzbare Undo-Punkte heben diese Software in die erste Liga des derzeit frei Erhältlichen. Die seit einer Dekade diszipliniert in kleinen Schritten voranschreitende Entwicklung ist sicher ein weiterer Pluspunkt. Der aus Kuala Lumpur stammende Kein Hong Man hat in Jahren der Feinarbeit die Syntaxeinfärbung von Perl, welche zu den schwersten überhaupt zählt, auf einen Stand gebracht, der sich mit *Vi* oder *Emacs* messen kann. Dennoch ist Scintilla eher auf statische Sprachen wie C ausgerichtet und keine ideale Basis für Entwicklungswerkzeuge für Perl. Mehrere wichtige interne Datenstrukturen sind sehr effizient, aber vor dem Programmierer verborgen. Für dynamische Sprachen, wo sich zur Laufzeit alles ändern kann, ist das eine Beschneidung. Das wird auch spürbar wenn man in Perl ein HTML-Template hat, in dem womöglich Javascript enthalten ist. Letzteres wurde mühsam dem PHP-Lexer aufgepfropft, aber ideal wäre ein Umschalten der Lexer innerhalb eines Dokuments.

Im nächsten Teil geht es wieder um mächtige Widgets, aber auch um die Frage wie man die *GUI* ohne *Sizer* anordnet. Manche Anfänger tun sich damit schwer. Große Widgets können mehr Inhalte anordnen. Es gibt aber auch drei Wege ganz auf *Sizer* zu verzichten: *XRC*, *FBP* und *AUI* (bereits vorgestellt). Wenn der Autor fleißig ist, gibt es vielleicht sogar bald einen vierten.

Links:

[1] http://www.scintilla.org/ScintillaRelated.html
[2] http://metacpan.org/module/Wx::Scintilla
[3] http://kephra.svn.sourceforge.net/viewvc/kephra/dev/base/share/config/syntaxhighlighter/perl.pm?revision=610&view=markup
[4] http://metacpan.org/module/Wx::Scintilla::Constant#Lexical-states-for-SCLEX_PERL
[5] http://bitbucket.org/lichtkind/kephra/src/64dd03dff99d/lib/Kephra/App/Util.pm
[6] http://www.scintilla.org/ScintillaDoc.html
[7] http://www.yellowbrain.com/stc/index.html

Yet Another

Perl Conference

EUROPE 2012

$WO =

GOETHE UNI-Frankfurt

$WEB =

http://yapc.eu/2012

$WANN =

20.-22. August 2012

EVOZON | $foo | XING | YAPC::Europe | united domains

YAPC::Europe 2012 in Frankfurt

Mark Overmeer

XML and Perl

Perl als Klebstoffsprache

Als das Internet noch jung war, wurde Perl zur ultimativen Sprache um das Betriebssystem, die Datenbanken und Webseiten zusammenzubringen. Es war nicht allzu schwierig, die Featureliste von awk, shell und sed zu übertreffen; Perl wurde bald zu einem gewichtigen Spieler. Aber man bleibt nicht automatisch an der Spitze: Eine Sprache muss sich mit der Welt um sie herum weiterentwickeln. Perl hat seine führende Position in vielen Gebieten verloren, zum Beispiel durch das totale Ignorieren von XML-basierten Standards.

Nicht ohne Grund hat XML unter Perl-Programmierern einen schlechten Namen. Perl-Leute mögen Programme, die mächtig sind, sich an DWIM ("Do what I mean") halten und effizient arbeiten. Die XML-Umgebung ist extrem geschwätzig, formell und wurde oft von Leuten mit wenig Programmiererfahrung designed.

Zum Beispiel wurden die XML Schemata ganz klar von Bibliothekaren konzipiert. Der Standardtyp eines Elements ist *anyType*, was in Programmierersprache bedeutet: "Es ist mir egal, es ist jetzt Dein Problem". Programmierer beginnen mit einem klar definierten einzelnen Bit, fügen diese zu Bytes zusammen, und so weiter.

Also hassen Perl-Leute XML von Natur aus. "Sie" haben sich in YAML verliebt, das sehr einfach scheint und daher sehr stark bevorzugt wird. Regel 1 in der Programmierung: Wenn Du denkst, dass etwas Neues viel besser und schneller ist als etwas Existierendes, hast Du wahrscheinlich nicht alle Komplikationen verstanden. So gibt es aktuell keine volle Unterstützung des neuen YAML Standards in Perl. Das neue Ideal ist JSON, YAML ist vorbei.

Einführung von XML::Compile

In der Zwischenzeit hat sich die "Professionelle Welt" auf XML standardisiert. Wenn Perl wieder Boden als generische Klebstoffsprache gut machen möchte, sollte es Implementierungen für die meisten XML-basierten Standard bieten. Es gibt nur ganz wenige dieser Implementierungen auf CPAN. Wahrscheinlich weil es keine Basis-Standard-Bibliothek gab, um diese umzusetzen.

Auf CPAN kannst Du einige Dutzend Module finden, die XML lesen und schreiben können. Aber diese können XML nur auf dem einfachsten Weg behandeln. Moderne XML-Protokolle sind Schema-basiert: Typen und Strukturen sind im Detail beschrieben. Die Schemata werden größer und größer. Es ist schrecklich, hunderte dieser Elemente in so einem Standard nach Perl zu übersetzen, Anweisung für Anweisung, Knoten für Knoten - so wie es die meisten CPAN-Module verlangen.

Seit 2006 habe ich an einer intelligenten Unterstützung für XML Schemata gearbeitet, später auch für SOAP und WSDL Standards. Schemata sind groß und Perl ist relativ langsam, also habe ich entschieden, die Last der Schema-Verarbeitung in die Initialisierungsphase des Programms zu legen um die Laufzeit schnell zu machen. In anderen Implementierungen (wie Java-Bibliotheken) werden Nachrichten zur Laufzeit gegen das Schema gematcht; sie laufen interpretiert. `XML::Compile` übersetzt die Schemata in wiederverwendbare Codereferenzen, die XML nach Perl, Perl nach XML übersetzen und sogar Beispiel-XML-Nachrichten und -Perl-Hashes generieren können.

XML lesen

Eigentlich ist das Schema-getriebene Lesen sehr einfach mit `XML::Compile`:

```
# compile once
use XML::Compile::Schema;
my $schema =
  XML::Compile::Schema->new($schemafn);
my $reader =
  $schema->compile(READER => $type);

# run often
my $hash   = $reader->($xml);

# Data::Dumper is your friend
print Dumper $hash;
```

Das XML (`$xml`), das verarbeitet werden soll, kann entweder ein String, ein Dateiname mit XML oder ein `XML::LibXML::Document` sein. `XML::Compile` basiert auf `XML::LibXML`, einem XS-Wrapper um Gnomes *libxml2*. Stelle sicher, dass Du eine aktuelle Version von beidem installiert hast, weil in den älteren Versionen viele Bugs enthalten sind.

Etwas schwieriger ist das `$type`. Du musst herausfinden welches Element im Schema das oberste Element in Deiner Nachricht ist. Vielleicht aus der Dokumentation. Und dann haben diese Element sowohl einen Namensraum als auch einen Namen in diesem Namensraum: ein Paar. Es ist nicht sehr praktisch, Paare herumzureichen. Deshalb erstellt Du `$type` so:

```
my $type = "{$namespace}$name";

# cleaner
use XML::Compile::Util 'pack_type';
my $type = pack_type $namespace, $name;
```

Nach ein paar Monaten habe ich festgestellt, dass es sehr unpraktisch ist, diese kompilierten Codereferenzen zwischen Funktionen und Modulen hin- und herzureichen. Es wäre viel schöner eine Art generischen Schemamanager zu haben. Also wurde ein neues Modul hinzugefügt, das diese kompilierten Strukturen einsammelt. Dieses Modul vereinfacht auch die Typspezifikation mit Präfixen. Das erste Beispiel umgeschrieben:

```
# compile once
use XML::Compile::Cache;
my $schema =
    XML::Compile::Cache->new(
       $schemafn, allow_undeclared => 1);
$schema->prefix(xyz => $namespace);

# run often
my $hash = $schema->reader(
                      "xyz:$name")->($xml);
```

Der *reader* wird bei der ersten Verwendung kompiliert. Das ist aber nicht das was Du in Daemons haben willst: In diesem Fall willst Du immer alle möglichen *reader* kompilieren bevor die Kindprozesse geforkt werden. In diesem Fall kannst Du das benutzen:

```
# compile once in the parent
use XML::Compile::Cache;
my $schema =
   XML::Compile::Cache->new($schemafn);
$schema->prefix(xyz => 'http://something');
$schema->declare(READER => "xyz:$name");
$schema->compileAll;

...fork...
# run often in any child
my $hash =
   $schema->reader("xyz:$name")->($xml);
```

Validierung

Alle Elemente in `$xml` werden während des Einlesens validiert. Zur selben Zeit werden sie zu benutzbaren Perl-Werten umgewandelt. Zum Beispiel müssen für einige Datentypen die optionalen Leerzeichen entfernt werden (whitespace 'collapse'). Die *dateTime*-Werte werden in passende Zeiteinheiten übersetzt, base64-kodierte Daten werden dekodiert, die boolschen Werte *false* und *true* werden zu '0' und '1', und so weiter.

Einige Parameter können zur Optimierung der Übersetzungen genutzt werden. Mit diesen Standardwerten:

```
validation      => true
check_values    => true
check_occurs    => true
ignore_facets   => false
sloppy_integers => false
sloppy_floats   => false
```

Die letzten beiden Parameter können oft verwendet werden. Die Schematypen *integer* und *float* erlauben Werte, die nicht in Perls Integer und Float passen. `XML::Compile` steckt diese

Werte in ineffiziente `Math::BigInt` und `Math::BigFloat` Objekte. In vielen Fällen hätte der Schema-Designer besser Typen wie 'unsignedShort' verwendet. Nachlässig ("sloppy") zu sein bedeutet: Zusichern, dass keine großen Werte vorkommen.

```
$schema->declare(
  READER => "xyz:$name", sloppy_float => 1);
```

Ein typisches Programm von mir enthält viele Überprüfungen der Eingaben, auf die Struktur der Daten und die Plausibilität der Werte. In meinem Code wird 3/4 der Logik für diese Aufgabe verwendet: Erstellen und Handling von Eingabeparametern von Skripten und Funktionen. Wie auch immer, wenn das Schema strikt ist, wird das XML (automatisch) validiert bevor es in Dein Programm kommt. Es werden weniger Tests benötigt. Wenn zum Beispiel das Schema das Folgende enthält:

```
<element name="gender">
  <simpleType>
    <restriction base="token">
      <enumeration value="male"    />
      <enumeration value="female"  />
      <enumeration value="unknown" />
    </restriction>
  </simpleType>
</element>
```

`XML::Compile` erzeugt automatisch eine Fehlermeldung beim Geschlecht `none`, also braucht Dein Programm das nicht mehr zu verifizieren. Die Wirklichkeit ist aber härter: Viele Schemata sind unspezifisch.

Verschachtelte Hashes

Das Ergebnis des Einlesens ist eine einzige (manchmal sehr tiefe) Struktur von verschachtelten HASHes und ARRAYs. Lass uns einen Blick auf die Übersetzung eines üblichen Konstrukts werfen. XML Schemata haben die folgenden Bausteine:

```
<record>
  <price>3.14</price>

  <name lang="nl">ijsje</name>

  <order number="PO1234">
    <paid>false</paid>
  </order>
</record>
```

Das erste Element innerhalb des Eintrags ist ein *simple* Type. Danach folgt ein *simple* mit Attributen (`complexType` mit `simpleContent`). Nur der mittlere Fall ist etwas schwieriger auf einen einzelnen HASH zu mappen: Der Wert hat keinen Namen. Der `reader` gibt das obige Beispiel als folgende Struktur zurück:

```
record =>
  { price => Math::BigFloat->new('3.14')
  , name  => {lang => 'nl', _ => 'ijsje'}
  , order => {number => 'PO1234', paid => 0}
  }
```

Wenn ein Element mehrfach vorkommen kann, wird es immer als ARRAY zurückgegeben. Zum Beispiel:

```
<element name="x" type="int"
         maxOccurs="unbounded" />
x => [ 1, 2, 3 ]
x => [ 4 ]
```

Die vielen Nicht-Schema-getriebenen XML verarbeitenden Module - wie `XML::Simple` - würden im letzten Fall kein Array zurückliefern. Sie können mit Elementwiederholungen nur umgehen, wenn sie sie in der Nachricht sehen. Das macht die Verwendung der Daten ein wenig komplex:

```
# when you use XML::Simple
my $r = $data->{...}{x} || [];
my @x = ref $r eq 'ARRAY' ? @$r : $r;

# when you use XML::Compile
my @x = @{$data->{...}{x} || []};
```

Warum nicht einfach XML::(LibXML::)Simple verwenden?

Ein Seiteneffekt der Schema-basierten Verarbeitung ist die "Reinigung" der Werte. Oft sind die Typen im Schema sehr unterschiedlich zu den Typen in Perl, aber nur wenn sie im Detail betrachtet werden. Der Schematyp "integer" zum Beispiel, kann mit Leerzeichen um den Wert benutzt werden, aber auch mit Leerzeichen zwischen den Ziffern! Und er muss mindestens Dezimalzahlen mit 19 Ziffern (64 bit) unterstützen. Ist Dein Perl bzw. das Perl des Kunden mit 64-bit Integern kompiliert?

Wenn Du `XML::Simple` zum Einlesen Deiner Dateien verwendest, musst Du entweder sehr viel Zeit auf die Validierung und Bereinigung verwenden oder darauf vertrauen,

dass die Gegenseite die selbe Teilmenge aus dem Wertebereich verwendet wie Du es aus den Beispielen heraus erwartest. Du meinst vielleicht zu wissen, dass alle XML-Integer in die Perl-Integer passen. Du sagst vielleicht voraus, dass ein boolscher Wert mit '0' oder '1' kodiert wird und nicht mit 'true' und 'false'. Es ist auf jeden Fall auf lange Sicht eine unsichere Situation.

Schlüssel umschreiben

Standardmäßig sind die Schlüssel im Perl HASH die einfachen Namen der Elemente im XML; der Namensraum wird ignoriert. Das ist oft kein Problem, weil die Schemata sehr selten so sehr miteinander verstrickt sind dass Namenskollisionen entstehen. Andererseits möchtest Du vielleicht den Namensraum aus Gründen der Klarheit, aus Spaß oder nur zur Dokumentation sehen:

```
$schema->addKeyRewrite('PREFIXES');  # all!
$schema->addKeyRewrite('PREFIXES(abc,xyz)');
```

Auch ist der bevorzugte Stil der XML-Designer, kleingeschriebene Namen mit "-" zwischen den Worten zu benutzen. Hashschlüssel mit "-" sind in Perl 5 unhandlich (Perl 6 liebt sie). Mit der nächsten Zeile werden die Bindestriche in Unterstriche umgewandelt:

```
$schema->addKeyRewrite('UNDERSCORES');
```

Du kannst sogar HASHes mit "Name-zu-Schlüssel"-Mappings übergeben. Du kannst das dazu verwenden, Elemente direkt zu Datenbankfeldern zu mappen. Diese Key-Mappings beeinflussen aber nur die Schlüssel in Perl, nicht das XML das gelesen oder geschrieben wird.

Wenn zum Beispeil die `PREFIXES`- und `UNDERSCORE`- Regeln aktiviert sind, wird das Element mit dem Namen `{http://something}my-name` nicht in einem Schlüssel "my-name" sondern in dem Wort "xyz_my_name" enden.

Erzeugen von Beispielen

Diese Datenstrukturen können groß werden und die Schemata können sehr komplex sein, über mehrere Dateien mit komplexen Vererbungsverbindungen verteilt. Um Dir zu helfen, die Datenstrukturen zu verstehen, kann `XML::Compile` kommentierte Beispiele erzeugen:

```
print $schema->template(PERL => $type);
print $schema->template(XML  => $type);
```

Die Ausgabe kann nicht direkt verwendet werden (weil es alle Optionen einer Auswahl ausgibt), aber sie wird sich beim Verstehen der Datenstruktur als hilfreich erweisen. Wenn Du Regeln zum Umschreiben der Schlüssel definiert hast, werden Sie auch auf die Perl-Beispiele angewendet.

Schreiben

Schließlich kannst Du auch perfekte *writer* für XML Elemente schreiben. Du muss nicht auf die Reihenfolge der Elemente, Namensräume, Rundungsfehler, Typekonvertierung und so weiter achten: Es macht einfach was man möchte! Das Ergebnis wird validiert. Ich werde hier nur die Version mit `::Cache` zeigen:

```
use XML::Compile::Cache;
my $schema = XML::Compile::Cache->new($xsdfn
  , allow_undeclared => 1);
$schema->prefix(p => $namespace);

my $doc  = XML::LibXML::Document->new('1.0'
  , 'UTF-8');
my $node = $schema->writer("p:$name")
                       ->($doc, $hash);
$doc->setDocumentElement($node);
print $doc->toString(1);
```

Wenn Dein Schema hässliche Konstrukte wie "any" verwendet, musst Du mehr als einen *writer* aufrufen um den gesamten DOM-Baum aufzubauen. All diese Subbäume des Ergebnisdokuments müssen mit dem gleichen `$doc` gebaut werden. Das ist der Grund für die drei zusätzlichen Zeilen rund um den eigentlichen *writer*.

Sei gewarnt, dass es in `XML::LibXML` ein paar Unterschiede zwischen dem Aufruf von `toString()` auf einem Knoten und dem Dokument gibt. Nur der letzte Fall stellt sicher, dass das utf-8 encoding korrekt ist.

Ein Protokoll implementieren

Ein Protokoll, das auf XML basiert, ist sehr einfach zu unterstützen, so lange es ein Schema dazu gibt. Wir haben *reader* und *writer* in den vorherigen Beispiel gesehen: Nur ein Dutzend Zeilen Code. Wir müssen das in ein *package* verpacken und etwas Abstraktion bieten. Du kannst einige Beispiele auf CPAN finden, für verschiedene Verwendungen von XML wie die Nutzung von Dateien und SOAP. Schau Dir an, welches am besten passt. Das Basismodul liefert auch ein paar Beispiele mit.

Als erstes musst Du überlegen, wohin Du *xsd* (XML Schema) und *wsdl* (Beschreibung von Webservices) Dateien installierst. Üblicherweise werden auf UNIX-Systemen diese Dateien unter */usr/share/* oder */usr/local/share/* gespeichert. Das ist im Allgemeinen nicht sehr nützlich: Wenn Dein Programm startet, wo sind dann diese Dateien? Es ist auch nicht Plattformunabhängig. Wusstest Du, dass **alles** im lib-Verzeichnis einer Distribution installiert wird? In `Geo::KML` benutze ich zum Beispiel folgendes:

```
lib/Geo/KML.pm
lib/Geo/KML/xsd/kml-2.1/kml21.xsd
lib/Geo/KML/xsd/kml-2.2.0/kml22gx.xsd
lib/Geo/KML/xsd/kml-2.2.0/ogckml22.xsd
```

Schemata für verschiedene Versionen des Schemas werden angeboten - relativ zum Hauptmodul. Dieses Modul sammelt die Dateien mit den folgenden Zeilen ein:

```
package Geo::KML;
sub new(%)
{   my ($class, %args) = @_;
    my $version = $args{version} || '2.2.0';
    (my $dir    = __FILE__)
        =~ s!\.pm$!/xsd/kml-$version!;
    my @xsd     = glob "$dir/*.xsd";
    my $schema  =
        XML::Compile::Cache->new(\@xsd);
    bless {schema => $schema}, $class;
}

sub writeKML($$)
{   my ($self, $data, $filename) = @_;
    my $doc = XML::LibXML::Document->
                new('1.0', 'UTF-8');
    my $xml = $self->{schema}->
                writer('kml')->($doc, $data);
    ...
}
```

Es ist sehr einfach den Speicherort der konstanten Dateien auf Basis des Speicherorts des Pakets mit `__FILE__` zu ermitteln. Die Konstante gibt den absoluten Speicherort der Datei an. Achtung: *glob* behandelt Pfade mit Leerzeichen nicht korrekt.

Natürlich musst Du *declare* und *prefix* Deklarationen zum `$schema`-Objekt hinzufügen. Das ist oftmals auch versionsabhängig und die Anzahl der Versionen kann wachsen. Daher erstelle ich für gewöhnlich einen HASH mit diesen Einstellungen.

Das obige Setup bietet eine schöne Abstraktion, bei der das Schema außerhalb des Zugriffsbereichs des Endbenutzers liegt. Das ist auf der einen Seite sauber, auf der anderen Seite ist es unpraktisch zum Debuggen und zur Template-Generierung. Die andere Lösung wäre, Vererbung zu verwenden:

```
package Geo::KML;
use base 'XML::Compile::Cache';
sub init($)
{   my ($self, $args) = @_;
    my $version = $args{version} || '2.2.0';
    (my $dir    = __FILE__)
         =~ s!\.pm$!/xsd/kml-$version!;
    $self->importDefinitions(
              [glob "$dir/*.xsd"] );
    $self;
}

sub writeKML($$$)
{   my ($self, $data, $filename) = @_;
    my $doc    = XML::LibXML::Document->
                    new('1.0', 'UTF-8');
    my $xml    = $self->writer('kml')->
                    ($doc, $data);
    ...
}
```

Du kannst weitere Schemata mit `importDefinitions()` laden - zu jeder Zeit. So viele Du willst, oder unterschiedliche Versionen des Schemas wenn Du willst. Es kann auch dazu benutzt werden, fehlerhafte Schema-Komponenten zu überschreiben: Die zuletzt geladene Definition überschreibt die vorherigen Definitionen mit dem gleichen Namen.

Der *reader* ist ein wenig schwieriger: Wenn eine Datei zum Einlesen bereitgestellt wird, wissen wir nicht, welches Top-Element wir erwarten sollen. Wir können auch die Initialisierung mit einer bestimmten Version beginnen bis wir das Wurzelelement kennen.

```
my $data = Geo::KML->readKML($source);

package Geo::KML;
sub readKML($)
{   my ($class, $source) = @_;
    my ($root, %details) =
          XML::Compile->dataToXML($source);
    my $rootns  = $root->namespaceURI;
    my $version = $self->
           namespace2version($rootns);
    my $self = $class->
           new(version => $version);
    $self->reader($kml)->($root);
}
```

Normalerweise wird der *reader* Dateinamen und String automatisch nach XML parsen, mit der Methode `dataToXML()`. In diesem Fall müssen wir die Quelle selbst parsen um herauszufinden welcher Namensraum im Wurzelelement des Dokuments benutzt wird. Einige Magie ist nötig, um den Namensraum in die verwendete Version des Protokolls zu übersetzen.

In diesem Moment wurde das niedrigste Level an Support erreicht. Es wird erwartet, dass Benutzer die Datenstruktur verstehen und erzeugen können, die deren Information in XML repräsentieren. Werden wir nur so ein niedriges Level unterstützen? Es wäre viel schöner, eine Abstraktionsschicht über diesem Modul hinzuzufügen um die Struktur der Nachricht und Unterschiede der Versionen aus Benutzersicht zu verbergen. Manchmal ist das einfach, oft ist es eine große Aufgabe. Für KML ist es zu viel Arbeit.

Fazit

`XML::Compile` ist die Basis für eine wachsende Anzahl von Modulen auf CPAN: `XML::Compile::SOAP`, `::SOAP::Daemon`, `Geo::KML` und viele mehr. Wenn Dir die Geschichte gefällt, werde ich einige davon in der nächsten Ausgabe von $foo erläutern.

Daniel Bruder

Good Practices: App-Entwicklung mit Moose, MooseX::Declare, MooseX::App::Cmd und weiteren Freunden aus dem MooseX::* Namensraum

Dieser Artikel zeigt Wege auf, wie sich nahezu mühelos Kommandozeilen-Applikationen schreiben lassen und führt dazu einige Module und Ideen vor, die nicht nur viel Freude bei der Entwicklung garantieren, sondern auch Kollaborationen extrem vereinfachen.

Was soll eine Kommandozeilen-App sein?

Unter Kommandozeilen-Applikationen sollen Programme wie z.B. `git`, mit folgender Struktur gemeint sein:

```
git log --author "Foo Ba*"
```

Um die Begriffe, die im Folgenden verwendet werden, klar zu definieren, haben wir folgendes:

- Das Programm, die "Kommandozeilen-App" `git`
- Das Kommando `log`
- Den Parameter `--author`
- Den Wert für den Paramater `--author`: `"Foo Ba*"`
- Den akzeptierten Wertebereich für das Argument `--author`: in diesem Fall ist das vom Typ `regex` bzw. `pattern`, wobei das erst bei Aufruf von `git log --help` oder `git help log` sichtbar wird.
- Die Stuktur `<App> help <command>`

Um selbst eine Kommandozeilen-Applikation mit einer ähnlichen Struktur zu realisieren stelle ich in diesem Artikel die Kombination aus `Moose`, `MooseX::App::Cmd` und einigen Freunden aus dem `MooseX::*` Namensraum vor.

Nehmen wir an, wir schreiben das Kommandozeilen-Front-End für ein fiktives Modul `Package::Installer`, genannt `pin`.

Bei Aufruf soll es folgendermaßen reagieren:

```
$ pin
Available commands:

  commands: list the application's commands
      help: display a command's help screen

      info: demo of an info command
   install: an install command
    remove: a remove command
    update: the update command
```

Die Kommandozeilen-Applikation `Package::Installer`, bzw. die ausführbare Datei, `pin` versteht also 4 Kommandos: `info`, `install`, `remove` und `update`, deren Namen selbsterklärend sind.

Die Ausführung von `pin install` ohne Optionen und Argumente liefert uns folgendes - siehe Listing 1.

Es meldet also, dass zumindest die Option `--module` belegt sein muss, und dass diese Option auch auf die Kurzoption `-m` hört. Desweiteren gibt es

- Optionen wie `-v` bzw. `--verbose` (vom Typ `Bool`)
- eine Option `--filemode`, welche nur 2 Werte akzeptieren darf, `0777` und `0755`
- eine Option für das Verzeichnis, in welches installiert werden soll. (Diese Option akzeptiert nur gültige Verzeichnisnamen und legt notfalls dieses Verzeichnis an, wenn es nicht existiert)
- (als Demonstration) eine Option `--global` für Optionen, die zwischen allen Kommandos geteilt werden
- und, zu guter Letzt, die Option `--help` bzw. `--usage`, bzw. `-?` für die Hilfe, die hier abgebildet ist.

Appetit geweckt? Los geht's!

```
$ pin install
Required option missing: module
usage: pin [-?dmv] [long options...]
     -m --module         [ModuleName] The module's name which to install
                         (demo, nothing will happen!) [required]
     --global            [Bool] Some common option which is shared by all
                         commands.
     -v --verbose        [Bool] Verbose output? [Default: 0]
     --filemode          [enum: 0777 / 0755] Filemode with which to create
                         directories unless they exist. [Default: 0777]
     -d --to --dir       [Dir] Specify the directory to which to install to
     -? --usage --help   Prints this usage information.
```

Listing 1

```
Package-Installer
|-- bin
|   `-- pin                          # Die ausführbare Datei
`-- lib
    `-- Package
        |-- Installer
        |   |-- Command
        |   |   |-- info.pm          # Kommando 'info'
        |   |   |-- install.pm       # Kommando 'install
        |   |   |-- remove.pm        # ....
        |   |   `-- update.pm
        |   `-- Command.pm           # Package::Installer::Command wird die Basis-Klasse
        |                            #    für Kommandos
           `-- Installer.pm          # Package::Installer
```

Listing 2

Die Struktur

Als erstes legen wir unsere grundsätzliche Struktur an, welche aussieht, wie in Listing 2 dargestellt.

Das Listing für lib/Package/Installer.pm verwendet MooseX::Declare und ist denkbar einfach:

```
use MooseX::Declare;
class Package::Installer
              extends MooseX::App::Cmd {
    # hier brauchen wir erstmal nix weiter
}
```

Wie man sieht, brauchen wir an dieser Stelle erstmal nichts weiter zu definieren. `Package::Installer` erbt von `MooseX::App::Cmd`, was uns das Wesentlichste zur Verarbeitung dessen, was auf `@ARGV` hereinkommen wird, bereits zur Verfügung stellt. Dazu gleich mehr.

Das ausführbare Programm

An dieser Stelle will ich auch gleich das Listing für die ausführbare Datei `bin/pin` vorstellen, welche auch denkbar einfach ist:

```
#!/usr/bin/env perl

use strict;
use warnings;

use FindBin;
use lib "$FindBin::Bin/../lib";

use Package::Installer;

Package::Installer->run();
```

Auch dieses ist schon völlig ausreichend und wird bleiben, wie es ist. Das Listing für die Struktur unserer Kommandozeilen-App zeigt bereits die schöne Auftrennung in die verschiedenen Kommandos welche sich im Namespace `Package::Installer::Command::*` wieder finden.

Die Implementation

Bevor wir an die Implementation der einzelnen Kommandos gehen, kümmern wir uns noch um Optionen, welche zwischen allen Kommandos geteilt werden. Das Listing für den Aufruf von `pin install` (welches am fehlenden `--module <Modulname>` gescheitert ist) hat unter anderem die Option `--global` gezeigt. Optionen, die allen Kommandos unter-

einander gemeinsam sind, werden wir nun in einer kleinen Zwischenebene, `Package::Installer::Command`, unterbringen, welche gleich noch eine größere Rolle spielen wird (siehe Listing 3).

Zwischenebenen und Vererbungen

`Package::Installer::Command` wird unsere "Basis-Klasse" für alle tatsächlichen Kommandos, also `info`, `install` und alle anderen, die wir gleich implementieren werden – allerdings vorher noch einige Anmerkungen zu Listing 3.

`Package::Installer::Command` erbt von zwei Klassen, `MooseX::App::Cmd::Command` und `Package::Installer`. In aller Kürze sei erwähnt, dass in diesem Fall die Reihenfolge der Vererbung als auch die runden Klammern um die Klassen, von welchen geerbt wird, wichtig sind, da `MooseX::Declare`, dass uns diesen wunderschönen Weg, Klassen zu definieren, anbietet (wir müssen uns nicht um Dinge wie `__PACKAGE__->meta->make_immutable`, `namespace::autoclean` oder `1;` am Ende der Datei, Datei-weites Scoping und anderes mehr kümmern *sondern* bekommen gleichzeitig noch `MooseX::Method::Signatures` frei Haus...) da `MooseX::Declare` an dieser Stelle den `@ISA` Pfad komplett neu schreibt.

Wesentlich wichtiger zu wissen ist aber, dass `MooseX::App::Cmd` zwei wesentliche Klassen zur Vererbung – besser gesagt: zur Er-erbung – anbietet:

- `MooseX::App::Cmd` für die Applikation selbst (wird also nur von einer Klasse geerbt) und
- `MooseX::App::Cmd::Command`, von welchem jedes Kommando erbt.

MooseX::App::Cmd

`MooseX::App::Cmd` kümmert sich um die gesamte Verarbeitung der Argumente in `@ARGV`. Dabei kpmmert es sich nicht nur um das Herauslesen und Unterscheiden zwischen "Kommando", "Option", "Argument" und "zusätzliche Parameter" aus `@ARGV`, sondern auch um die Validierung der Argumente hinsichtlich ihres Datentyps. Das haben wir im letzten Listing bei der gezeigten Option `--global` gesehen, welches vom Typ `Bool` ist und nur entsprechende Argumente/Werte akzeptiert.

Verheiratung und Synergie-Effekte: App::Cmd + MooseX::Getopt == MooseX::App::Cmd

Zum Hintergrund ist wieder in aller Kürze zu sagen, dass `MooseX::App::Cmd` die Verheiratung des schon länger existierenden `App::Cmd` mit dem ebenso hervorragenden `MooseX::Getopt` darstellt. Wie wir im Folgenden noch sehen werden, ergibt diese Mischung ein sehr mächtiges Werkzeug. Und eben dieses wollen wir uns hier zu Nutze machen.

MooseX::Getopt

Beim Blick auf das Listing fällt gleich ein Teil dieser Kombination auf: `with MooseX::Getopt::Dashes`. Das so genannte "Konsumieren" der "Role" `MooseX::Getopt::Dashes` übernimmt für uns die Übersetzung zwischen Optionen, die als `has 'some_option'` im Code vorkommen, auf der Kommandozeile aber im wesentlich gebräuchlicheren Format `--some-option` auftreten sollen! (Nebenbei bemerkt: `--some_option` wird dort immer noch möglich sein, auch wenn nicht explizit in der Hilfe erwähnt).

`MooseX::Getopt` als der zweite Teil des Duos von `MooseX::App::Cmd` übernimmt die Übersetzung von den Attributen der Moose-Klassen in das Anbieten von Optionen auf der Kommandozeile. Dabei geht es sehr intelligent nach folgenden Regeln vor:

- Das Attribut `has 'foo' => ...` wird zur Kommandozeilen-Option `--foo`
- Das "private" Attribut `has '_bar' =>` wird nicht angeboten, außer es wird explizit gewünscht.

```
use MooseX::Declare;

class Package::Installer::Command extends (MooseX::App::Cmd::Command, Package::Installer)
  with MooseX::Getopt::Dashes {
    use MooseX::Types::Moose -all;
    has global => (
        is            => 'rw',
        isa           => Bool,
        default       => 1,
        documentation => q{[Bool] Some common option which is shared by all commands.},
    );
}
```

Listing 3

Desweiteren wird `MooseX::Getopt` die Datentypen der Werte, die von der Kommandozeile hereinkommen, validieren und gegebenenfalls ablehnen. (Nebenbemerkung `MooseX::Getopt` baut die Moose-Attribute nach `Getopt::Long::Descriptive` um.)

Diese Information sollte uns fürs Erste an dieser Stelle genügen, und uns wissen lassen, dass wir unsere Moose-Datentypen von `MooseX::Getopt` für die Aufnahme von Werten von der Kommandozeile zur Verfügung gestellt bekommen. Das ist im vorangegangenen Listing ersichtlich: die Option `--global` hat einen Default-Wert von 1 *und* wird diesen für alle Kommandos zu unserer App, welche gleich folgen werden, in gleicher Art und Weise anbieten. Um deutlich zu machen, wie schön und wie mächtig die Kombination von `MooseX::Getopt` mit `App::Cmd` in `MooseX::App::Cmd` ist: Da `--global` vom Typ Bool und per default auf `1` steht, haben wir bereits auch die gültige Option `--no-global`, um den voreingestellten default-Wert wiederum negieren zu können, frei Haus bekommen!

MooseX::Types::Moose

Eine kurze Nebenbemerkung noch zur Zeile `use MooseX::Types::Moose -all;`: Diese Zeile importiert alle Datentypen von Moose derart, dass wir diese als (vermeintliche) "Barewords" verwenden dürfen. Das hat den entscheidenden Vorteil, dass diese (also: die Verwendungen selbiger) zur Compile-Zeit geprüft werden (und daher als "Barewords" geschrieben werden können).

Die Kommandos

Nach diesen Bemerkungen kommen wir aber endlich zu den Kommandos! Steigen wir mal einfach beim simpelsten ein, dem Kommando `info`, also den Aufruf

```
$ bin/pin info
This is Package::Installer (Demo) version
  v0.1 and this is the value of the global
  var: 1
```

und schauen uns noch kurz die Hilfe zu `info` an (Listing 4).

```
$ bin/pin help info
usage: pin [-?] [long options...]
     --global           [Bool] Some common option which is shared by all
                        commands.
     --long-info        [Bool] Long info? (use --no-long-info to shut
                        --long-info off [Default: on])
     -? --usage --help  Prints this usage information.
```

Listing 4

Das Kommando selbst wird in Listing 5 dargestellt.

Besprechung des Listings

Hier also das erste Kommando, `Package::Installer::Command::info`. Es erbt, da wir (und andere) bereits alles so schön vorbereitet haben, von unserer eigenen "Basis-Klasse für Kommandos", `Package::Installer::Command`. Die ganzen Vorbereitungen zeigen auch schon erste Früchte: im folgenden bitte nicht erschrecken, im Gegenteil: dies sollte man einmal C++-Programmierern vorführen – hier die Vererbungslinie von `Package::Installer::Command::info` um deutlich zu machen, auf wessen Schultern wir bereits stehen. Sie ist nämlich mittlerweile schon etwas angewachsen (und von rechts nach links zu lesen):

```
Linear @ISA   Package::Installer::Command::
 info, Package::Installer::Command, MooseX::
 App::Cmd::Command, Moose::Object, App::
 Cmd::Command, App::Cmd::ArgProcessor,
 Package::Installer, MooseX::App::Cmd, App::
 Cmd
```

Nicht schlecht, oder? Oder anders ausgedrückt: Mehrfachvererbung kann so leicht sein!

Die nächste interessante Zeile findet sich bei `use Package::Installer::Types ':all'`. Hier importieren wir unsere eigenen Datentypen, in diesem Fall hätten wir auch nur den einzigen Datentyp, den wir hier verwenden `VersionString`, alleine über `use Package::Installer::Types 'VersionString'` importieren können. Die selbstdefinierten Datentypen werden wir im folgenden gleich betrachten.

Als nächstes fällt `method execute` auf. `execute()` ist einfach der Einstiegspunkt und das, was `MooseX::App::Cmd` aufrufen wird wenn dieses Kommando auf der Kommandozeile angefordert wird. An dieser Stelle machen wir einfach nur eine Informationsausgabe und belassen es dabei.

(Den so genannten "Turtle-Operator" `@{[]}` verwende ich hier, um `$self->name` und ähnliches innerhalb von doppelten Anführungszeichen verwenden zu können und dem Perl-Parser schmackhaft zu machen.)

```
use MooseX::Declare;

class Package::Installer::Command::info extends Package::Installer::Command {
    use MooseX::Types::Moose -all;

    use Package::Installer::Types ':all';

    use feature 'say';

    method execute {
        say "This is @{[$self->_name]}"
          , $self->long_info ? " version @{[$self->my_version]}" : ""
          , " and this is the value of the global var: @{[$self->global]}"
    }

    has my_version => (
        is      => 'ro',
        isa     => VersionString,
        traits  => [ 'NoGetopt' ],
        default => '0.1',
        coerce  => 1,
        documentation => q{This will not be offered as an option},
    );


    has long_info => (
        is      => 'rw',
        isa     => Bool,
        traits  => [ 'Bool', 'Getopt' ],
        default => 1,
        documentation => q{[Bool] Long info? (
                                  use --no-long-info to shut --long-info off [Default: on])},
    );

}
```

Listing 5

Als nächstes ist der "trait" `NoGetopt` bei `my_version` interessant: da `my_version` kein "privates Attribut" `_my_version` ist, teilen wir per trait mit, dass dieses Attribut nicht als Option angeboten werden soll, fertig. Wir hatten ja gesagt, dass `MooseX::Getopt` nicht-private Attribute anbietet, und private nicht anbietet, außer man teilt ihm etwas anderes mit – was hiermit schon geschehen ist.

Weiterhin auffällig ist der Default Wert von `0.1`, das nachfolgende `coerce => 1` und das "v0.1" in der Ausgabe unseres Kommandos: `"This is Package::Installer (Demo) version v**0.1"`. Da wir die Bool-Eigenschaften von `long_info` und die Übersetzung in `--long-info`, bzw. `--no-long-info` schon besprochen haben, schauen wir uns doch mal den selbst definierten Datentyp an und schauen wie dieser von `MooseX::Getopt` in ein `Getopt::Long::Descriptive`-Argument umgewandelt wird, an – und zwar in diesem `package`:

```
package Package::Installer::Types v1.0.0 {
     use Moose::Util::TypeConstraints;
     use MooseX::Types::Moose -all;
     use MooseX::Types -declare => [qw(
         VersionString
         ModuleName
     )];

     subtype VersionString,
          as Str,
       where { m/ ^ v\d{1,2}\.\d{1,2} $ /x },
     message {
         qq<Unable to parse $_ as a
            VersionString. Needs to be
            something like `v1.0'>};

      coerce VersionString,
        from Num,
         via { 'v' . $_ },
        from Str,
         via { /^v/ ? $_ : "v$_" };
}
```

Datentypen auf der Commandline

- Zeile 1 ist nur dazu da, um die neuen Möglichkeiten in Perl 5.14 aufzuzeigen, aber nicht wichtig...
- `Moose::Util::TypeConstraints` in Zeile 2 brauchen wir, um `subtype` und `coerce` in unseren Namensraum zu bekommen

• `use MooseX::Types::Moose -all` kennen wir schon und erlaubt uns hier `Str` ohne Anführungszeichen als `subtype VersionString, as Str, ...` zu schreiben
• mit `use MooseX::Types -declare =>` *deklarieren* wir unsere eigene Datentypen hier an zentraler Stelle, *efinieren* sie auch in diesem Paket, und können diese dann in allen anderen Modulen (wie wir bereits gesehen haben) mit `use Package::Installer::Types ':all'` einfach als "Barewords" verwenden.

Moose und die Definition von eigenen Datentypen sind nicht Teil dieses Artikels. Sie sind in $foo schon ausreichend besprochen worden und werden in weiteren Artikeln zu Moose noch besprochen. Daher sei nur so viel zu den Definitionen hier gesagt:

• `VersionString` ist ein `subtype` zu `Str`. Interessant wird das dadurch, weil `Moose::Getopt` damit ausreichend Informationen hat, um diesen Datentyp für uns nach `Getopt::Long::Descriptive` "umrechnen" zu können! Desweiteren bieten wir eine `coercion` in der Form an, dass wir `Num` akzeptieren und einfach ein 'v' davorhängen oder einen String akzeptieren und ein 'v' davorhängen, wenn keines da ist – und fertig ist der Datentyp!

Wir haben den Aufruf von `pin install` schon gesehen und dieser hatte nach einem Modul `--module` vom Typ `ModuleName` verlangt. Daher sehen wir gleich noch die Definition von diesem (eigenen) Datentyp an dieser Stelle bevor wir sogleich zur Implementation des Kommandos `install` kommen:

```
use Module::Util;

subtype ModuleName,
     as Str,
  where {
   Module::Util::is_valid_module_name($_) },
message {
       qq<$_ is not a valid module name.> };

 coerce ModuleName,
   from Str,
    via { m|/|x and s|(\.pm)?$|.pm|x
        and Module::Util::path_to_module($_)
        or $_
    };
```

Nachdem wir dank `Moose` und `MooseX::Declare` die einzelnen Klassen, Hierarchien, Vererbungen und Attribute schon so schön **deklarativ** beschreiben konnten, darf an dieser Stelle auch mal wieder etwas kryptischeres, gutes altes Perl zur Sprache kommen... Was der Datentyp `ModuleName` macht, bzw. als Werte von der Kommandozeile akzeptiert, ist folgendes: Der User kann die Modulnamen auf folgende Arten und Weisen gleichermaßen auf der Kommandozeile angeben:

```
$ pin install Foo::Bar      # Klassisch
$ pin install Foo/Bar.pm    # Praktisch
$ pin install Foo/Bar       # Wieso nicht
                            # auch so?
```

Um möglichst bald zur Besprechung des Kommandos `Package::Installer::Command::install` zu kommen, an dieser Stelle nur so viel: Wir verwenden schlicht das existierende `Module::Util` zur Validierung und bieten diesem in der `coercion` einen Modulnamen an, der
• entweder '/' enthält, und in diesem Fall – mit einem garantierten '.pm' am Ende und mit `Module::Util::path_to_module` transformiert – zur Validierung an die `subtype`-Definition und den dortigen Aufruf von `Module::Util::is_valid_module_name($_)` abgegeben wird
• oder eben schlicht, was wir auf der Kommandozeile erhalten haben, an `Module::Util::is_valid_module_name` zur Validierung durchreicht.

Mehr können (bzw. wollen oder müssen) wir an dieser Stelle nicht tun, um "passend zu machen was nicht passend ist".

Package::Installer::Command::install

Das Kommando `Package::Installer::Command::install` geht nun wieder den selben Weg, wie schon das Kommando `info`:
• Es erbt von `Package::Installer::Command`.
• Es definiert/deklariert seine Attribute und sagt welche als Optionen auf der Kommandozeile angeboten werden sollen.
• Es definiert eine `method execute` (oder wahlweise auch eine `method run`), z. B. auf folgende Weise.

```
method run {
  printf "Going to install module `%s' to
  directory `%s' with permissions `%s'
  (verbose is %d)...\n",
              ( $self->module,
                $self->dir->absolute->
                   resolve->stringify,
                $self->filemode,
                $self->verbose)
}
```

(Selbstverständlich müsste an dieser Stelle in einem echten Einsatz ein bisschen mehr geschehen.)

Im ursprünglichen Aufruf des Programms mit `pin install` hatten wir schon gesehen, dass die Option `--module` nicht nur zwingend erforderlich ist, sondern auch das Alias `-m` getragen hat. Wie kommt das zustande, bzw. wie kann ich das definieren? Ganz einfach, über das `Getopt`-trait und `cmd_aliases`:

```
has module => (
    is            => 'rw',
    isa           => ModuleName,
    traits        => [ 'Getopt' ],
    cmd_aliases   => [ 'm' ],
    required      => 1,
    documentation => q{
        [ModuleName] The module's name which
         to install (demo, nothing will
         happen!) [required]
    },
);
```

Um gleichzeitig noch die Frage zu klären, woher die Beschreibung der Option beim Aufruf von `pin install` kam: `MooseX::Getopt` verwendet schlicht den bis dahin von `Moose` ungenutzten `documentation`-tag als Hilfsausgabe innerhalb von `App::Cmd`. (Und woher die Meldung "Required option missing: module" kam, bleibt dem Leser zur Aufgabe überlassen.)

Andere Optionen

Ab jetzt sollte es also einfach sein, weitere Optionen zu dem Kommando `install` hinzuzufügen.

Schauen wir uns noch eine andere interessante Option an, `--filemode`. Das hat ausgesagt, nur 2 Werte zu akzeptieren, `0777` und `0755`. `Moose::Getopt` arbeitet auch hier hervorragend mit den `Moose`-Datentypen zusammen, es erlaubt uns nämlich auch, ein anonymes `enum` zu verwenden:

```
has filemode => (
    is          => 'rw',
    isa         => enum([ qw< 0777 0755> ]),
    defaul      => '0777',
    documentation => q{[enum: 0777 / 0755]
           Filemode with which to create
           directories unless they exist.
           [Default: 0777]},
);
```

Hervorragend, oder? Diese Option wird in der Folge also nur diese beiden Werte mit `--filemode 0777` oder `--filemode 0755` akzeptieren – oder sich lauthals beschweren (Man muss dazu sagen, dass man an dieser Stelle noch einen geeigneten Hook setzen müsste, um dem User den langen `Moose`-Stacktrace zu ersparen).

Eine weitere Option zur Auswahl war `--dir` mit den Aliasen `-d` und `--to`. Auch hier wieder verlassen wir uns auf bereits existierende Infrastruktur, in diesem Fall das hervorragende `Path::Class` und dessen Portierung zu einem `Moose`-Datentyp:

```
use MooseX::Types::Path::Class 'Dir';
has dir => (
    is            => 'rw',
    isa           => Dir,
    coerce        => 1,
    traits        => [ 'Getopt' ],
    cmd_aliases   => [ 'to', 'd' ],
    # default       => '.',
    trigger       => \&_trigger_dir,
    lazy_build    => 1,
    documentation => q{[Dir] Specify the
                       directory to
                       which to install to
                     },
);
```

Wir können nun nicht nur durch Import des Datentyps `Dir` diesen wieder als (vermeintliches) "Bareword" schreiben und damit die Prüfung, ob dies ein erlaubter Datentyp für unsere `isa =>` -Anweisung in die Compile-Zeit verschieben. Wir machen uns an dieser Stelle auch das `lazy_build`-Verhalten zu Nutze und schreiben noch einen Trigger, der das Verzeichnis anlegt in welches das genannte Modul installiert werden soll, falls das Verzeichnis noch nicht angelegt ist. Allerdings auch nur für den Fall, dass auch tatsächlich ein Verzeichnis angegeben wurde! Das schöne dabei ist, dass wir für `$self->dir` alle Funktionalität von `Path::Class::Dir` zur Verfügung haben, also wie im Listing zu `method run` gesehen, folgendes schreiben können:

```
$self->dir->absolute->resolve->stringify
```

Die Implementation des Verzeichnis-Anlegens durch einen geeigneten `trigger` oder "around-method-modifier" und das Versorgen mit einem `default` bleibt dem Leser zur Übung überlassen. Eine Erwähnung des hervorragenden `MooseX::Declare` möchte ich an der Stelle allerdings doch noch machen, denn es erlaubt folgende wunderschöne Konstrukte, die leider noch nicht so Bekanntheit erlangt haben, wie sie es verdienen:

```
method _trigger_dir(Dir $dir_set,
  Dir $dir_set_before?) {
    if ($dir_set_before) { ... }
}
```

und

```
method _build_dir(Dir $set?) {
    ...
    return return $set // '.'
}
```

bzw. einfach direkt:

```
method _build_dir(Dir $dir = '.') {
    return $dir
}
```

und auch

```
around dir(Dir $set?) { ... }
```

Ausblick

Es bleibt noch zu erwähnen, dass mit dem Genannten auch folgendes zu implementieren ein leichtes wäre aber dem Leser zur Übung überlassen bleiben muss:

```
has module => (
    ...
    isa => ArrayRef[ModuleName],
    ...
);
```

Man sieht, die Kombination aus `Moose`, `MooseX::Declare` und `MooseX::App::Cmd` bringt einen beträchtlich weit und erlaubt es, Kommandozeilen-Applikationen deklarativ, modular und, vor allem, mit Freude und Leichtigkeit zu implementieren. Sie verschafft dem User eine vertraute Basis auf der Kommandozeile mit einem leicht zugänglichen, da bekannten Interface auf Grundlage von bewährten Konventionen.

Kritik und Anmerkungen

Es ist selbstredend ironisch `Package::Installer` als ein Beispiel zu nehmen: Die Liste der Abhängigkeiten des hier vorgestellten Moduls braucht ja bereits einen guten Installer für Pakete!

Anders gesprochen bedeutet das folgendes: Auf die hier vorgestellte Art und Weise Module schreiben zu können macht unglaublichen Spaß und lässt einen an jeder Ecke neue faszinierende Dinge entdecken. Auf der anderen Seite jedoch muss das Hereinnehmen von Paket-Abhängigkeiten, wie überall, wohl überdacht sein.

Welche User sind angesprochen? Wo wird mein Modul hauptsächlich zum Einsatz kommen? Können an diesen Orten schnell, effizient und einfach Module installiert werden? Haben die User, die mein Modul benutzen wollen eventuell sowieso schon den allergrößten Teil, der hier verwendeten Module installiert? Bzw. ist davon auszugehen, dass sie ihre Umgebung bereits gut vorkonfiguriert haben mit `cpanm`, `local::lib`, `perlbrew` etc.?

Ein weiterer Punkt, der nicht außer Acht gelassen werden darf sind die Laufzeiten. Auch hier muss wieder hinsichtlich des Einsatzes des fertigen Moduls abgestimmt werden: Geht es um rasante Laufzeiten oder um lang laufende, dafür aber schwerer zu konfigurierende, komplizierter zu startende Anwendungen? Das Laden von Moose ist bekanntlich nicht ebenso so schnell wie klassische blessed-references, dafür aber wesentlich eleganter, kürzer und sicherer und vor allem in Verbindung mit dem hervorragenden `MooseX::Declare` eine wahre Freude.

Bei der Kritik ist zu guter letzt ein Punkt nicht zu vergessen: die Kombination aus `Moose`, `MooseX::Declare`, und, womöglich noch `MooseX::MultiMethods` und evtl. auch noch `Moose::Autobox` bringt wieder die Kritiker auf den Plan mit der Frage "Ja... aber ist das noch Perl?". Anzunehmen ist, dass die Frage eher lautet "Ja... aber ist das noch PERL?" ;-) – trotz allem bleibt aber (neben der Laufzeit) ein eher praktischer Punkt zu bedenken: das syntax-highlighting, perltidy und Perl Critic funktionieren (noch) nicht wie gewohnt und ausnahmslos auf dieser Art von Perl-Code. Diese sind schlicht noch nicht auf die alle neuen Syntax-Zuckerl abgestimmt, werden es aber sicher noch werden.

Auf der Positiv-Seite bleibt aber doch einiges festzuhalten:

• Die Kollaboration mit anderem ist extrem einfach: Nicht nur, dass die Struktur extrem übersichtlich, modular, und auf einfachste Weise erweiterbar ist. Es reicht sogar, wenn ich *egal wo* in meinem `@ISA`-Pfad ein `package Package::Installer::Command::foobarbaz` herumliegen habe.

Dieses wird von Perl gefunden und damit auch vom Modul `Package::Installer` angeboten!

• Wie an manchen Stellen schon durchgeblitzt ist, lassen sich durch die geschickte Kombination von den hervorragend ausgelegten Modulen aus dem `MooseX::*`-Namespace extrem praktische Synergieeffekte erzielen. Dieses Paradigma des "Postmodernen objektorientierten Programmierens" mit `Moose`, und `MooseX::Declare`, und weitergehend mit `MooseX::MultiMethods`, `Moose::Autobox` und noch vielen weiteren mehr kann einen wahren Sturm der Freude auslösen und passiert viele bisherige Programmier-Paradigmen auf der Überholspur. Wie gesehen kann die zusammengenommene Power von vielen guten Modulen kann zum Teil beträchtlich sein!

• Zu guter Letzt kann man sich auf diese Art schon einen guten Vorgeschmack auf Perl6 holen und sich bereits in den kommenden Objektorientierungen orientieren.

Viel Spaß damit!

Ulli Horlacher

SpreadApp

In $foo 21 (1/2012) gab es den Artikel "Konfigurationsmanagement und Software-Deployment mit Rex". In bester Perl TIMTOWTDI Tradition ist dies nun eine Antwort darauf.

Das Ausgangsproblem: Klassen von UNIX Hosts, die gleich administriert werden sollen. Das bedeutet, dass auf ihnen die selbe Dateien verteilt und dieselben Kommandos ausgeführt werden müssen.

Das Zusatzproblem: Nicht jeder Host ist zu jedem Zeitpunkt erreichbar, z.B. weil er in Wartung ist oder wegen einer Netzstörung nicht erreichbar ist. Trotzdem muss automatisch sichergestellt sein, dass schlussendlich alle Hosts auf demselben Stand sind.

Für die zentrale Administration und Softwareverteilung gibt es zwar eine ganze Reihe von Open-Source-Lösungen, die aber auch ihre spezifischen Nachteile haben:

rsync:
• kann nur Dateien verteilen, aber keine Befehle
• funktioniert nur, wenn Zielsystem erreichbar

Rex:
• komplizierte Syntax
• funktioniert nur, wenn Zielsystem erreichbar

puppet, chef, ...:
• komplex und aufwändig
• benötigt auf jedem Zielsysteme Client-Software

Es scheint so, dass die vorhandene Software entweder funktional nicht ausreichend oder zu kompliziert ist. Was macht also der Admin? Schimpft über die mangelhafte Softwaresituation und schreibt sich sein passendes Tool in Perl selber. Selbst ist der Admin :-)

Das neue Tool sollte:

• die Problemstellung lösen und dabei einfach zu verstehen und zu bedienen sein
• leicht zu erweitern sein
• keine spezielle Software auf Client-Seite benötigen
• auf einem zentralen Admin-Server laufen, der auf alle Zielsystem root-Zugriff via `ssh` hat
• selbstständig alle noch ausstehenden Aufgaben abarbeiten

Herausgekommen ist dabei das Programm "spread" in den Geschmacksrichtungen "cspread" (command spread) und "fspread" (file spread). Die letzten beiden sind symbolic links auf das original spread. Also je nachdem über welchen Namen spread aufgerufen wird, verhält es sich unterschiedlich.

Funktionsweise

Befehle oder Dateien werden zuerst zu einem "Job" zusammengefasst und im spread Spool abgelegt, ähnlich wie bei SMTP die E-Mails. Danach wird sofort versucht, diesen Job auf die Hosts der ausgewählten Klasse zu verteilen. `STDIN` und `STDERR` werden dabei direkt angezeigt. War die Auslieferung für den betreffenden Host erfolgreich, wird der Job aus dem Spool gelöscht. Ansonsten verbleibt er dort und es wird via `crontab` automatisch alle 5 Minuten (konfigurierbar) eine erneute Auslieferung versucht. Bei Erfolg bekommt der Admin eine E-Mail zugestellt. So wird sichergestellt, dass alle Hosts auf demselben Administrationsstand sind.

spread

```
usage: spread [-l] [-L [class]]
          [-v] [-r] [-E]
options: -v            verbose mode
         -l            list jobs in spool
         -L            list host classes
         -r            retry jobs in spool
         -E            edit config file
```

Mit spread kann die zentrale Konfigurationsdatei editiert werden, die Jobs oder die Host-Klassen aufgelistet werden, sowie ein Abarbeiten der Spool-Jobs erzwungen werden.

Die spread Konfigurationsdatei kann z.B. so aussehen:

```
$spool = '/var/spool/spread';
$default_class = 'linux';
@{$class{'mailrelay'}} =
  qw(smtp1 smtp2 smtp3 smtp4 smtp5);
@{$class{'LXC'}} =
  qw(fex flupp gopher);
@{$class{'lucid'}} =
  qw(zentux vms1 vms2 zoo radius1 radius2);
@{$class{'linux'}} =
  sort(@{$class{'lucid'}},@{$class{'LXC'}});
```

Alle interne Variablen können vorbelegt werden. Außerdem können beliebig viele Host-Klassen definiert werden.

Im obigen Beispiel werden zuerst das Spool-Verzeichnis und die Default-Klasse als einfache Skalare festgelegt. Die Host-Klassen `mailrelay`, `LXC`, `lucid` und `linux` bilden ein "*hash of arrays*", wobei die `linux`-Klasse eine sortierte Zusammenfassung der Klassen `lucid` und `LXC` ist. Die Array-Elemente sind die Hostnamen, wobei aus Übersichtlichkeitsgründen der Domainname weggelassen worden ist.

In Worten wäre das also:

Die Klasse `mailrelay` besteht aus den Hosts

`smtp1 smtp2 smtp3 smtp4`

Die Klasse `LXC` besteht aus den Hosts

`fex flupp gopher`

Die Klasse `lucid` besteht aus den Hosts

`zentux vms1 vms2 zoo radius1 radius2`

Die Klasse `linux` besteht aus den Hosts

der Klassen `LXC` und `lucid`.

Da die Konfigurationsdatei eine normale Perl-Datei ist, die mit require geladen wird, kann hier beliebiger Perl-Code drin stehen. Damit wird eine maximale Flexibilität erreicht.

Ist keine Konfigurationsdatei vorhanden, dann legt spread ein Musterbeispiel an, das leicht angepasst werden kann, selbst ohne Perl-Kenntnisse.

cspread

```
usage: cspread [-v] [-c class]
     [-h host1,host2,...]
     [-xhosta,hostb,...] jobname 'command...'
usage: cspread [-v] [-c class]
     [-h host1,host2,...]
     [-x hosta,hostb,...] jobfile
options: -v          verbose mode
         -c class      name of server class
                       (default: linux)
          -h host1,... use these hosts
                       instead of class
          -x hosta,... exclude these hosts
                       example: cspread test
                       "uname -a; uptime"
```

```
root@obertux:~# cspread -c LXC test 'uptime; uname -a'

fex:
 18:43:06 up 8 days,  2:46, 14 users,  load average: 0.06, 0.14, 0.14
Linux fex 2.6.38-13-server
   #54~lucid1-Ubuntu SMP Wed Jan 4 14:38:03 UTC 2012 x86_64 GNU/Linux
(fex)

flupp:
 18:43:07 up 8 days,  6:50,  0 users,  load average: 0.09, 0.16, 0.12
Linux flupp 2.6.38-13-server
   #54~lucid1-Ubuntu SMP Wed Jan 4 14:38:03 UTC 2012 x86_64 GNU/Linux
(flupp)

gopher:
 18:43:07 up 8 days,  6:50,  0 users,  load average: 0.09, 0.16, 0.12
Linux gopher 2.6.38-13-server
   #54~lucid1-Ubuntu SMP Wed Jan 4 14:38:03 UTC 2012 x86_64 GNU/Linux
(gopher)
```

Listing 1

Mit `cspread` werden Kommandos auf den Zielsystemen ausgeführt. Die Kommandos können direkt als Argument übergeben werden oder stehen in einer Batch-Datei, wobei hierbei der Dateiname gleich dem Jobname ist. Die Zielsysteme werden entweder über ihren Klassennamen spezifiziert oder explizit aufgelistet. Dabei ist es auch möglich spezielle Hosts auszulassen. Werden keine Zielsysteme angegeben, so wird automatisch die Klasse `$default_class` verwendet.

Der Jobname wird benötigt um den betreffenden Job im Spool zu identifizieren.

Es gibt eine Einschränkung bei den Kommandos: sie dürfen keine Eingaben vom Anwender erwarten, da spread `STDIN` auf `/dev/null` setzt. Es funktionieren deshalb nur nicht-interaktive Kommandos. Beispiel in Listing 1.

fspread

```
usage: fspread [-v] [-c class]
    [-h host1,host2,...]
    [-x hosta,hostb,...]
    [-d dir]
    [-p script]
    [-e script] jobname file...
options: -v             verbose mode
         -c class       name of server class
                        (default: linux)
         -h host1,...   use these hosts
                        instead of class
         -x hosta,...   exclude these hosts
         -d dir         destination directory
                        (must be absolute
                        path)
         -p script      prolog script (BEFORE
                        file installation)
         -e script      epilog script (AFTER
                        file installation)
argument: file          may be "-" which
                        means: read filenames
                        from STDIN
example: fspread profile /etc/profile*
```

Mit `fspread` werden Dateien auf die Zielsysteme kopiert. Dabei wird intern `tar` verwendet, so dass die Dateiattribute erhalten bleiben. Optional kann auch ein alternatives Zielverzeichnis angegeben werden. Die Auswahl der Zielsysteme erfolgt wie bei cspread. Zusätzlich kann noch je ein Script vor oder nach dem Kopieren ausgeführt werden. Dies ist z.B. sinnvoll um einen Dienst erst zu stoppen, neue Software dafür zu installieren und ihn dann wieder zu starten. Dies ist mit einem Aufruf möglich. Beispiel in Listing 2.

Anmerkung: Host `smtp2` wurde explizit ausgelassen und Host `smtp5` war nicht erreichbar. Deshalb verbleiben die Dateien vom Job "local" im Spool und es wird ab sofort alle 5 Minuten versucht sie dennoch auszuliefern.

`[cf]spread` besteht Perl-sei-Dank aus nur 380 Zeilen Code (inklusive Kommentare) und benötigt keine Zusatz-Perl-Module. Als einzige externe Softwarekomponente ist ssh notwendig, was sich aber sowieso auf jedem vernetzten UNIX befinden sollte.

Somit ist die Installation von spread extrem einfach: Herunterladen von *http://fex.rus.uni-stuttgart.de/download/spread* und passend abspeichern, z.B. in */root/bin/*

Noch mit "chmod +x spread" ausführbar machen und das war's dann auch schon. Die symbolischen Links zu `cspread` und `fspread` werden automatisch beim ersten Aufruf angelegt.

Wie oben beschrieben, muss der spread-User (normalerweise root) ssh-Zugang auf alle Zielsysteme haben. Sonsts wird es schwierig mit der Verteilung.

```
root@obertux:/# fspread -c mailrelay -x smtp2
   local /usr/local/bin/ /root/.profile
tar: Removing leading `/' from member names
/usr/local/bin/
/usr/local/bin/fexsend
/usr/local/bin/fexget
/usr/local/bin/xx
/root/.profile

smtp1:
spreading files...
(smtp1)

smtp3:
spreading files...
(smtp3)

smtp4:
spreading files...
(smtp4)

smtp5:
host unreachable
(smtp5)
```

Listing 2

Renée Bäcker

Ticket oder nicht Ticket?

Nicht nur diese Frage kann mit Postmaster-Filtern in OTRS beantwortet werden. Viele Tickets werden bei OTRS wie bei den meisten anderen Ticket-/HelpDesk-Systemen über E-Mails erzeugt. Die E-Mails werden hier von dem so genannten Postmaster abgearbeitet. Bis dann der User die E-Mail als Ticket zu sehen bekommt, werden verschiedene Stufen abgearbeitet. Die E-Mail wird eingelesen (entweder von einem Postfach abgeholt oder über ein Skript herein geschoben), vorgefiltert, ein Ticket wird erzeugt und nachträglich noch einmal gefiltert (siehe Abbildung 1).

In diesem Artikel werde ich verschiedene Wege zeigen, wie solche Filter umgesetzt werden können.

Abbildung 1: Ablauf PostMaster

Die erste Möglichkeit ist das Einrichten von Filtern über die Weboberfläche. Im Adminbereich unter *PostMaster Filter* können die Einstellungen vorgenommen werden. Für die einzelnen Felder, die in einer Mail vorkommen, kann man Bedingungen angeben. Es handelt sich hierbei um Reguläre Ausdrücke. Aus diesem Grund sollte man genau bedenken, wie die Bedingungen für einen Filter aussehen.

Die Bedingungen der einzelnen Felder werden mit "UND" verknüpft. Also nur dann, wenn alle Bedingungen im Filter auf die Mail zutreffen, werden die Aktionen ausgeführt, die im unteren Teil angegeben werden können.

In Abbildung 2 ist ein Beispiel für einen Filter zu sehen, der alle Mails von einem bestimmten Absender in die Junk-Queue verschiebt und automatisch schließt.

Abbildung 2: PostMaster Filter, der SPAM filtert

Stärken und Schwächen dieser Filter

Alle gerade beschriebenen Filter werden ausgeführt, wenn die Tickets schon erzeugt wurden. Man kann hier also keine Mails ablehnen. Der Vorteil liegt aber darin, dass so ein Filter schnell eingerichtet ist. Durch diese Einfachheit in der Bedienung ist aber auch eine gewisse Unflexibilität gegeben. So gibt es keine "ODER"-Verknüpfung der einzelnen Bedingungen und auch eine Umkehrung ist nicht immer zu erreichen.

Wer sich mit Regulären Ausdrücken auskennt, kann mit Negativem Look-Behind etc. helfen, aber auch damit kann man nicht alle Probleme lösen. In Abbildung 3 ist ein Filter zu sehen, der auf Mails von einem bestimmten Absender wartet und der Betreff **kein** "Wichtig" am Anfang des Betreffs enthält.

Ein Problem eher allgemeiner Natur ist, dass nicht alle Mailserver die gleichen Header setzen. Ein Beispiel: Eine Mail wird per *BCC* verschickt. In welchem Mail-Header kann man dann die Empfängeradresse finden? Richtig: Es kommt darauf an! Manchmal in `X-Envelope-To`, bei anderen wiederum in `X-Delivered-To` und es gibt noch weitere Header. Manchmal muss man eine Mail erst analysieren um zu wissen, auf welche Felder man achten muss.

Sollte das Feld noch nicht in den DropDowns zur Auswahl stehen, muss man das Feld in der SysConfig unter *Ticket -> Core::PostMaster* bei dem Konfigurationsparameter *PostmasterX-Header* hinzufügen.

Ein weiteres Problem ist bei einem Kunden aufgetaucht. Dort kam eine Mail von einem Dienstleister an. Wenn man die Ticketansicht aufgemacht hat, hat man den Text "5.00/5 Sterne" gesehen. Diese Tickets sollten nun automatisch verschoben und geschlossen werden. Klingt, als wäre es ein einfacher Filter. Aber es war eine HTML-Mail und im Plain-Text-Teil der Mail stand nur, dass es keinen Plain-Text gibt. Die Filter, die über die Oberfläche erstellt werden, sehen aber nur den Plain-Text einer Mail. Letztendlich musste ein Filter als Perl-Modul geschrieben werden.

Dieses Problem und die anderen Schwächen zeigen, dass man unter Umständen einen anderen Mechanismus für Postmaster-Filter wählen muss: Perl-Module als Filter. Für einfache Aufgaben sind die über die Weboberfläche erstellten Filter aber gut geeignet.

Postmaster-Filter selbst programmieren

Mit den selbstgeschriebenen Modulen kann man nicht nur fertige Tickets noch verändern, man kann auch auf die Mails einwirken, wenn noch gar kein Ticket erstellt wurde. Für diesen Artikel soll ein Postmaster-Filter geschrieben werden,

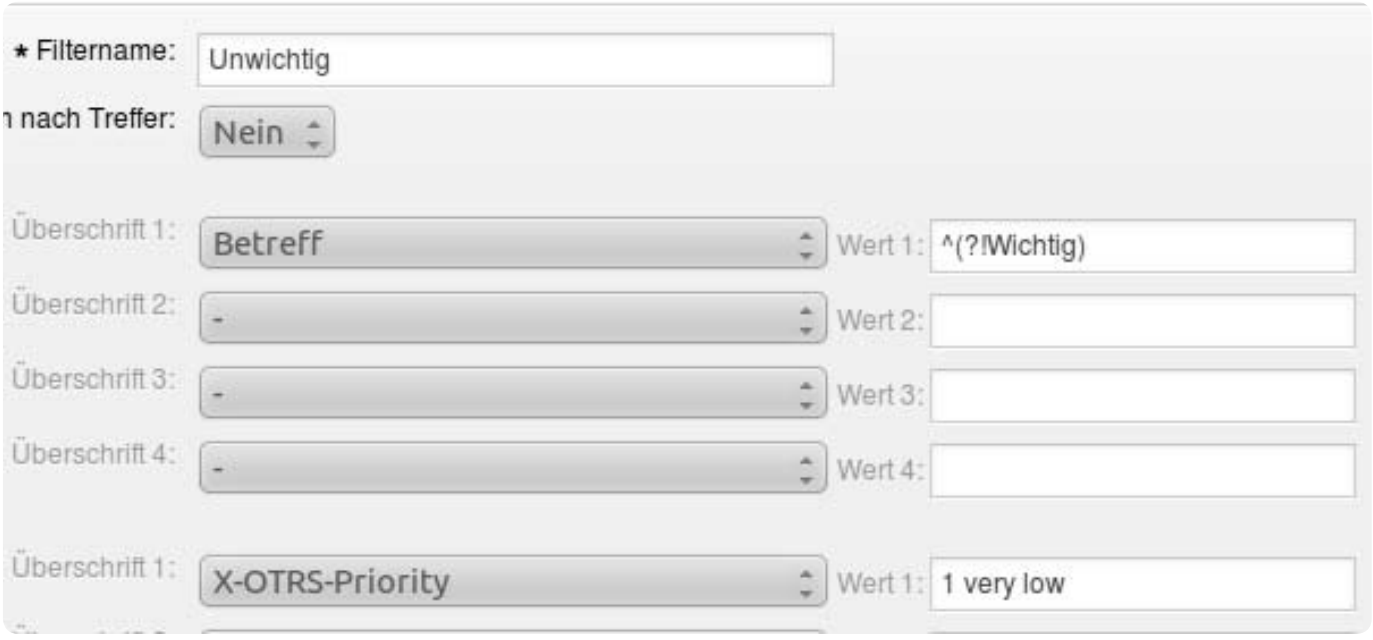


Abbildung 3: Wissen über Reguläre Ausdrücke ist notwendig

der mit Mails umgehen kann, die von einer (fiktiven) Bezahlseite kommen und in denen der Kunde bei der Bezahlung noch ein Kommentar hinterlassen kann. Damit nicht alle Tickets durchgeschaut werden müssen, ob ein Kommentar eingegeben wurde, erstellt der Filter automatisch ein Ticket mit dem Kommentar. Außerdem sollen alle Mails mit den Bezahlinfos in eine Queue "Bezahldienst" geschoben und geschlossen werden.

Zum Parsen der Mails benutzt OTRS verschiedene Module aus dem `MIME::*`-Namensraum und `Mail::Internet` aus der *MailTools*-Distribution.

Gehen wir von folgender Mail aus:

```
Return-Path: <perl@renee-baecker.de>
X-Original-To:
  mailinglisten@renee-baecker.de
Delivered-To: perl@renee
Received: [...]
Message-ID:
  <4F7A126D.7070606@renee-baecker.de>
Date: Mon, 02 Apr 2012 22:56:13 +0200
From: =?ISO-8859-1?Q?Renee_B=E4cker?=
         <perl@renee-baecker.de>
MIME-Version: 1.0
To: mailinglisten@renee-baecker.de
Subject: Bezahldienst: Bezahlung eingegangen
Content-Type: text/plain; charset=ISO-8859-1
Content-Transfer-Encoding: 7bit

Der Betrag von 11,53 EUR ist Ihrem Konto
  gutgeschrieben worden.

Der Kunde hat folgende Notiz hinterlassen:

Die Lieferung erfolgte prompt, vielen Dank!
```

Aus dieser Mail erstellt OTRS einen Hash, der an den Postmaster-Filter übergeben wird, dazu gleich mehr.

Ein Postmaster-Filter muss zwei Methoden zur Verfügung stellen: den Konstruktor `new` und die Methode `Run`. In der Methode `Run` müssen die Aktionen implementiert werden, die der Filter leisten soll. Es werden die folgenden Parameter übergeben:

- TicketID
- JobConfig
- GetParam

Sollte die Mail eine Nachricht sein, die zu einem anderen Ticket gehört, steht die ID des Tickets in *TicketID*. Über die SysConfig können weitere Einstellungen zu dem Filter gemacht werden. Existieren solche Einstellungen, werden diese über *JobConfig* an den Postmaster-Filter übergeben.

Die einzelnen Teile aus der Mail stehen in einer Hashreferenz, die über den Parameter *GetParam* in den Filter kommt.

Die Mail von oben wird dann übergeben, wie ins Listing 1 dargestellt.

Dazu kommen noch die ganzen `X-Header`, die über die SysConfig eingestellt werden, für den Beispiel-Filter aber ohne Bedeutung sind.

```
{
  'Content-Type' => 'text/plain; charset=utf-8',
  'X-Sender' => 'perl@renee-baecker.de',
  'X-OTRS-FollowUp-ArticleType' => 'email-external',
  'Resent-From' => '',
  'User-Agent' => 'Mozilla/5.0 (X11; Linux x86_64; rv:11.0) Gecko/20120310 Thunderbird/11.0',
  'Message-Id' => '<4F7A126D.7070606@renee-baecker.de>',
  'From' => "Renee B\x{e4}cker <perl\@renee-baecker.de>",
  'Body' => 'Der Betrag von 11,53 EUR ist Ihrem Konto gutgeschrieben worden.

Der Kunde hat folgende Notiz hinterlassen:

Die Lieferung erfolgte prompt, vielen Dank!
',
   'X-Mailer' => '',
   'ReplyTo' => '',
   'Reply-To' => '',
   'Message-ID' => '<4F7A126D.7070606@renee-baecker.de>',
   'Subject' => 'Bezahldienst: Bezahlung eingegangen',
   'To' => 'mailinglisten@renee-baecker.de',
}
```

Listing 1

Man kann natürlich die geparste Mail mit eigenen Headern anreichern. Diese tauchen nirgends auf, aber sie eignen sich sehr gut dazu, Informationen von einem "Pre"-Filter an einen "Post"-Filter zu übergeben. Das kann man z.B. verwenden wenn in einem Prefilter bestimmt wird, dass ein neues Ticket erstellt werden soll und im Postfilter dann mit anderen Tickets verlinkt werden soll.

Gibt es Anhänge an der Mail, gibt es zusätzlich den Schlüssel *Attachment* in der hashreferenz. Zu diesem Schlüssel wird eine Arrayreferenz übergeben, in der alle Informationen zu den Anhängen zu finden sind:

```
[
  {
      Filename    => $Filename,
      Charset     => $Charset,
      MimeType    => $MimeType,
      ContentType => $ContentType,
      Content     => $Content,
  },
]
```

Doch zurück zum Beispiel. Bevor der Filter geschrieben wird, müssen die Bedingungen festgelegt werden, auf die der Filter reagieren soll. Das sollte möglichst eng gefasst werden, damit der Filter wirklich nur bei den festgelegten Mails greift. Da viel mit Regulären Ausdrücken usw. gearbeitet wird, sollte man den Lauf des Filters möglichst früh abbrechen.

Bei dem Beispiel-Filter muss die Mail von "perl@renee-baecker.de" kommen und der Betreff ist ebenfalls feststehend. Nur wenn Absender und Betreff passen, wird der Text der Mail untersucht. Und dann soll ein neues Ticket aus der Notiz erstellt werden. Und das Ticket aus der Originalmail soll verschoben und geschlossen werden.

```
my $Mail = $Param{GetParam};

return 1 if $Mail->
    {From} !~ /perl\@renee-baecker.de/;
return 1 if $Mail-> {Subject}
    ne 'Bezahldienst: Bezahlung eingegangen';
```

```
my ($Note) = $Mail->{Body} =~ m!Der Kunde hat folgende Notiz hinterlassen:\n(.*)!s;
my $ArticleID = $Self->{TicketObject}->ArticleCreate(
    TicketID         => $TicketID,
    ArticleType      => $Mail->{'X-OTRS-ArticleType'},
    SenderType       => $Mail->{'X-OTRS-SenderType'},
    From             => $Mail->{From},
    ReplyTo          => $Mail->{ReplyTo},
    To               => $Mail->{To},
    Cc               => $Mail->{Cc},
    Subject          => $Mail->{Subject},
    MessageID        => $Mail->{'Message-ID'},
    InReplyTo        => $Mail->{'In-Reply-To'},
    References       => $Mail->{'References'},
    ContentType      => $Mail->{'Content-Type'},
    Body             => $Note,
    UserID           => 1,
    HistoryType      => 'EmailCustomer',
    HistoryComment   => "\%\%$Comment",
    OrigHeader       => $Mail,
    Queue            => $Queue,
);
```

Listing 2

```
<?xml version="1.0" encoding="iso-8859-1"?>
<otrs_config version="1.0" init="Application">
    <ConfigItem Name="PostMaster::PostFilterModule###010-Comments" Required="0" Valid="1">
        <Description Translatable="1">Postmaster filter to handle comments.</Description>
        <Group>PerlServices</Group>
        <SubGroup>Foo::PostMaster</SubGroup>
        <Setting>
            <Hash>
                <Item Key="Module">Kernel::System::PostMaster::Filter::Comments</Item>
            </Hash>
        </Setting>
    </ConfigItem>
</otrs_config>
```

Listing 3

Der Returnwert muss 1 sein, weil sonst im Log die Meldung auftaucht, dass der Filter nicht erfolgreich durchgelaufen sei.

Wenn diese Bedingungen auf die Mail zutreffen, können über *X-OTRS-*-Angaben der Status und die Queue gesetzt werden:

```
$Mail->{'X-OTRS-Queue'} = 'Raw';
$Mail->{'X-OTRS-State'} =
  'closed successful';
```

Und das Ticket muss erzeugt werden:

```
# create new ticket
   my $NewTn    =  $Self->{TicketObject}
                     ->TicketCreateNumber();
   my $TicketID = $Self->{TicketObject}
                     ->TicketCreate(
    TN           => $NewTn,
    Title        => $Mail->{Subject},
    Queue        => $Queue,
    Lock         => 'unlock',
    Priority     => '3 normal',
    State        => 'new',
    CustomerID   => $Mail->{From},
    CustomerUser => $Mail->{From},
    OwnerID      => 1,
    UserID       => 1,
);
if ( !$TicketID ) {
    return;
}
```

Damit sind die Metadaten für das Ticket angelegt. Fehlt noch der Text bzw. der Artikel - siehe Listing 2.

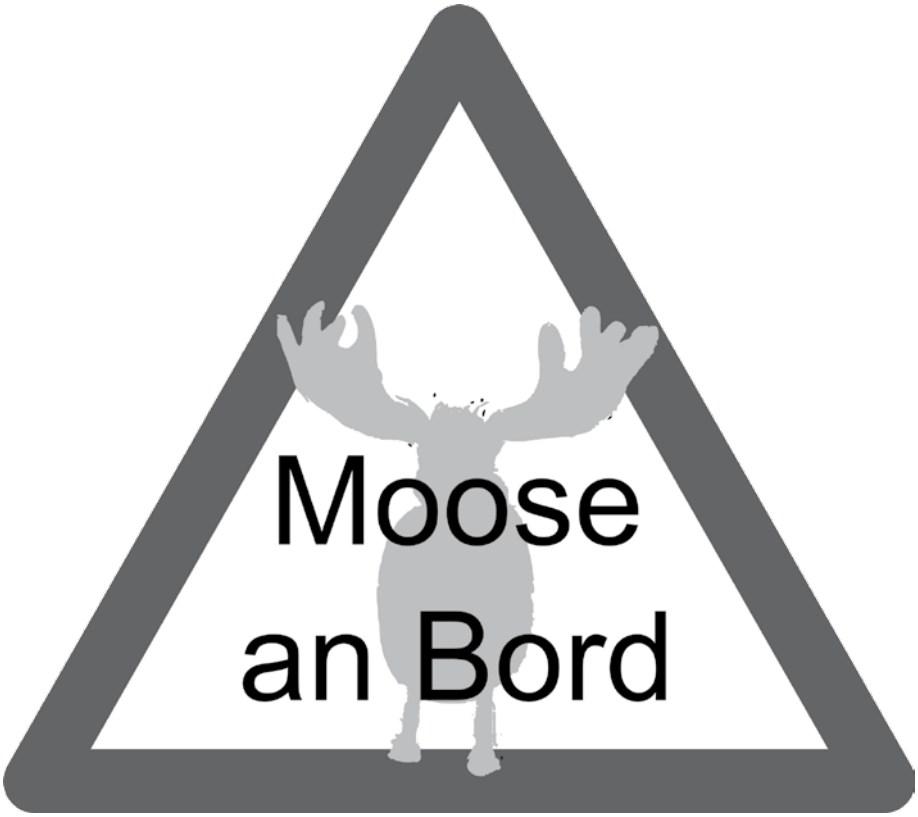
Die meisten Angaben können direkt aus der Originalmail übernommen werden. Einzig der Text enthält nur noch einen Teil der Originalmail.

Bevor der Filter vom Postmaster verwendet wird, muss man diesen über die Konfiguration aktivieren. Hierzu gibt es zwei Wege. Der bequemste Weg ist, unter *Kernel/Config/Files/* eine XML-Datei abzulegen, die folgenden Inhalt hat - siehe Listing 3.

Alternativ kann man in der Datei *Kernel/Config.pm* in der Subroutine `Load` folgenden Eintrag machen:

```
$Self->{'PostMaster::PostFilterModule'}
          ->{'010-Comments'} = {
     Module => 'Kernel::System::PostMaster::
                  Filter::Comments',
 };
```

Hierbei sollte man größeren Wert auf den Namen legen. OTRS sortiert die Schlüssel (Namen) der Filter und führt diese in aufsteigender Reihenfolge aus. Man sollte darauf achten, welchen Namen man vergibt, damit nicht nachfolgende Filter die eigenen Einstellungen überschreiben. Wenn nur Werte gesetzt werden, die von anderen Filtern nicht angefasst werden, ist der Name egal.



Perl-Services.de

Programmierung - Schulung - Perl-Magazin

info@perl-services.de

Renée Bäcker

Was ist neu in Perl 5.16?

Noch ist Perl 5.16 nicht erschienen, aber durch die monatlichen (Entwickler-)Releases kann man ständig am Geschehen dran bleiben. Die Entwicklerversionen enden einmal im Jahr in einer stabilen Version, so dass wir bald Perl 5.16 "in den Händen halten" dürfen. Durch die verkürzten Releasezyklen landen nicht so viele neue Features in den stabilen Versionen. Das ist aus Sicht der Wartbarkeit auch gar nicht so übel.

Hier werden die ganzen "kleinen" Bugfixes nicht näher ausgeführt, sondern nur die größeren Änderungen.

Noch ein Hinweis zum Text: Im Text steht hin und wieder `use v5.16` während im Code dann `use v5.15` steht. Da Perl 5.16 noch nicht erschienen ist, musste alles mit der Entwicklerversion (hier: 5.15) getestet werden.

Jetzt aber zu den Neuerungen!

__SUB__

Die meisten werden die Token `__FILE__` etc. kennen. Mit `__FILE__` bekommt man den Pfad zu der Datei, in der man sich gerade befindet. Etwas ähnliches wurde mit `__SUB__` für Subroutinen eingeführt. Damit bekommt man eine Referenz auf die aktuelle Subroutine. Das erleichtert das Schreiben von rekursiven Closures. Um dieses neue Feature nutzen zu können, muss es mit `use feature 'current_sub'` oder `use v5.16` aktiviert werden.

```
use v5.15.9;

use strict;
use warnings;

my $fac = sub {
    my ($i) = @_;

    return 1 if $i == 1;

    return $i * __SUB__->($i-1);
};

print $fac->(6);
```

Noch bessere Unterstützung von Unicode

Schon in den vergangenen Versionen wurde stetig an einer besseren Unterstützung von Unicode gearbeitet, das hat sich auch bei Perl 5.16 nicht geändert. Das erstreckt sich über verschiedene Bereiche: Reguläre Ausdrücke, Variablennamen, Strings etc.

Mit Perl 5.16 wird Unicode 6.1 ausgeliefert und nahezu vollständig unterstützt. Seit Unicode 6.0 gab es ein paar Umbenennungen von Zeichen und einige Kategorien haben sich geändert. Perl 5.16 kann sowohl mit den alten als auch mit den neuen Namen umgehen. So hieß das Zeichen mit dem Codepoint *U+000A* früher *LINE FEED (LF)*, jetzt heißt es *LINE FEED*.

```
use v5.15;

use strict;
use warnings;
use utf8;

my $¼ = "test\n";

say "yes" if $¼ =~ /\N{LINE FEED (LF)}/;
say "yes (new)" if $¼ =~ /\N{LINE FEED}/;

say "test\N{LINE FEED}<<";
```

An dem Beispiel kann man noch weitere Veränderungen sehen: Man kann jetzt auch Variablen mit Unicode-Zeichen im Namen verwenden. In diesen Teilen können Unicode-Zeichen verwendet werden:

- Methodennamen (inklusive jener, die an `use overload` übergeben werden)
- Typeglobnamen (inklusive Variablennamen, Subroutinen und Dateihandles)
- Paketnamen
- Konstanten
- goto
- Symbolische Dereferenzierung
- Zweites Argument von `bless()` und `tie()`
- Rückgabewert von `ref()`
- Paketnamen, die von `caller()` geliefert werden
- Subroutinen-Prototypen
- Attribute
- verschiedene Warnungen und Fehlermeldungen, die Variablennamen, Werte oder Methodennamen beinhalten.

Auch bei der Verwendung von Unicode-Zeichen in String mit der `\N{...}`-Syntax hat sich etwas geändert. Das `use charnames (...)` kann weggelassen werden, das Pragma wird jetzt automatisch geladen.

Da die Namensgebung bei Unicode nicht immer ganz konsistent ist (Mal mit "-", mal ohne; mal mit " ", mal ohne, ...), kann man bei `use charnames :loose` angeben. Damit muss man sich nicht unbedingt merken ob da jetzt ein Bindestrich war oder nicht. Allerdings hat diese Flexibilität auch einen Performancenachteil. Der Lookup wird ungefähr 2-3 mal langsamer.

Neues Tutorial zum Thema Objektorientierung

Es gab sehr viele fleißige Helfer, die die Dokumentation von Perl an vielen Stellen angepasst haben. Der größte Brocken war mit Sicherheit, dass die alten Dokumentationen zur Objektorientierung (*perltoot*, *perltooc* und *perlboot*) aus dem Kern geworfen wurden. Stattdessen gibt es eine komplett neugeschriebene Dokumentation unter *perlootut*. Das neue Tutorial zeigt sowohl die Basics der Standard-Perl5-Objektorientierung, weist aber auch auf moderne Frameworks wie `Moose` hin.

study für mehr als einen String

Mit `study` kann man Zeit dafür aufwenden, Perl einen String untersuchen zu lassen. Das **kann** Zeit sparen wenn auf diesen String viele Reguläre Ausdrücke angewendet werden, da Perl dann schon weiß, wie der String aufgebaut ist.

Bisher konnte immer nur für einen String `study` angewendet werden. Diese Beschränkung wurde in Perl 5.16 aufgehoben.

Mittlerweile ist `study` auch ein no-op. Dadurch kommt es auch mit Strings klar, die länger sind als 2**31 Zeichen.

Subroutinen im CORE::-Namensraum

Die meisten der Kernfunktionen sind jetzt über den `CORE::`-Namensraum als Subroutinen verfügbar. Damit kann man Referenzen von ihnen bekommen und auch über `&foo()` aufrufen. Folgendes ist jetzt möglich:

```
use v5.15;
BEGIN { *corepush = \&CORE::push; }

my @array;
corepush @array, 5, 1;
say "@array";
```

Sicherheitsfix: File::Glob::bsd_glob() Speicherfehler

Die Funktion `bsd_glob()` in `File::Glob` hat Flags ausgewertet, die eigentlich nicht unterstützt wurden. Ein Angreifer könnte mit dem Flag *GLOB_ALTDIRFUNC* einen Zugriffsfehler oder Segmentation Fault provozieren, was wiederum zu einem *Denial of Service* oder Codeeinschleusung führen könnte. In der neuen Version wurden alle nicht-unterstützten Flags abgeschaltet.

$[wurde entfernt

... jedenfalls teilweise. Im Kern ist es nicht mehr direkt enthalten, weil es in der Regel eher hinderlich ist, wenn man den Start-Index eines Arrays verändert. Mit Perl 5.16 wird aber das Pragma `arybase` mitgeliefert, über das man `$[` doch wieder benutzen kann.

```
use feature 'say';

$[ = 5;

my @array = (4);

say $#array;
say $array[5];
__END__
Use of assignment to $[ is deprecated at ...
5
4
```

Ein `use v5.16;` sorgt aber dafür, dass man der Variablen höchstens noch den Wert "0" zuweisen kann.

```
use v5.15;

$[ = 5;

my @array = (4);

say $#array;
say $array[5];
__END_
Assigning non-zero to $[ is no
  longer possible at ...
```

Debugger-Änderungen

Der Debugger ist ein Riesenskript, das nicht leicht zu warten ist. Dennoch haben sich einige Änderungen ergeben. Das `feature`-Paket steht jetzt für die Kommandos im interaktiven Debugger zur Verfügung.

Dem Kommando `t` (*trace*) kann ein numerischer Parameter übergeben werden, der angibt, wieviele Subroutinenlevel ge*trace*t werden sollen.

Über die neuen Kommandos `disable` und `enable` können Breakpoints deaktiviert bzw. aktiviert werden. Beim Setzen von Breakpoints mittels `b` kann die Datei angegeben werden: `b [file]:[line] [condition]`.

Sonstiges

Noch ein kleines "Neuerungen-Medley" zum Schluss:

Es gibt viele kleine Performanz-Verbesserungen. Z.B. werden Unicode-Properties in regulären Ausdrücken nicht mehr linear sondern binär gesucht; `use v5.15` lädt nicht mehr *feature.pm*; `DESTROY`-Methoden werden unter bestimmten Bedingungen nicht mehr aufgerufen; `substr` im *void*-Kontext ist rund doppelt so schnell wie früher; und vieles mehr ...

Viele Module wurden auf neuere Versionen gebracht und `Version::Requirements` wurde als *DEPRECATED* markiert.

`defined(@array)` warnt jetzt auch bei Paketvariablen, wer `\E` ohne `\Q`, `\L` oder `\U` verwendet bekommt jetzt auch eine Warnung. Statt `length(@array)` meinen die meisten wohl eher `scalar(@array)`; `length` auf Arrays oder Hashes erzeugt jetzt eine Warnung. Es gibt auch neue Fehler und noch ein paar weitere neue Warnungen.

Dieser Artikel hat nur einen Ausschnitt aus den Neuerungen in Perl 5.16 zeigen können. Da ist noch viel mehr passiert. Das *perldelta* von Perl 5.16 ist lang, also Zeit mitbringen zum Lesen!

Herbert Breunung

Rezension - Spaß bei der Arbeit

chromatic
mit Damian Conway & Curtis »Ovid« Poe
Perl Hacks
324 Seiten, PDF 5MB
1. Auflage O'Reilly 2006
ISBN 978-3-89721-691-4
E-Book € 10,00

Peter Seibel
Coders at Work
Bedeutende Programmierer und ihre Erfolgsgeschichten
560 Seiten, Format 14,8 x 21,0
1. Auflage, mitp 2011
ISBN 978-3-8266-9103-4
Softcover €29,95

Da es seit Jahresanfang nur wenige Neuerscheinungen gab, die sich nicht mit Web, mobilen Plattformen und den dortigen sozialen Umgangsformen befassen, folgen jetzt zwei Kommentare in eigener Sache. Der Marktrundgang wird mit der nächsten Ausgabe nachgeholt. In einem Perlmagazin müssen Bücher zu Perl klar den Vorzug haben. Es darf jedoch auch um die Kunst des Programmierens im weiteren Sinn gehen. Um hierbei eine ausgewogene Mischung zu erhalten, wird es hier meist ein Perl-Buch und einen Nicht-Perl-Titel geben. Nur leider gibt es nicht so viele Bücher über Perl, um das jedes Quartal durchzuhalten. Und manchmal haben interessante Schriftstücke schlicht ein anderes Format. Deshalb soll ab dieser Folge ebenfalls die Regel gelten: Es soll jedes Mal ein gedrucktes und ein elektronisches Werk rezensiert werden, wobei auch Aufsätze, Artikelsammlungen, Minibücher und Selbstverlegtes beachtet werden. Der Inhalt ist entscheidend. Das soll jedoch nicht bedeuten, daß jedes Buch hier eine Empfehlung ist. Besonders die letzte Folge sollte das deutlich gemacht haben. Diese Kolumne versteht sich als Dienst für alle an Perl Interessierte, alles einzusammeln was "Perl" auf dem Schild trägt und es unter die Lupe zu nehmen. Denn wer hat schon Zeit einer bezahlten Tätigkeit nachzugehen und das Netz zu überwachen? Um möglichst alles abzudecken, werden manche Werke nur mit wenigen Sätzen im Vorwort abgehandelt. Bei anderen soll detaillierter beschrieben werden, warum sie nach des Autors Meinung (nicht) empfehlenswert sind. Auf jeden Fall wird immer so geplant, dass mindestens ein Titel Lesefreude ausstrahlt.

Elektronische Bücher

Das Verlagswesen ist im Umbruch. Die großen Häuser meiden oft noch stärker Risiken als zuvor und Einzelpersonen fangen an ihre Arbeiten elektronisch oder per Print-on-Demand zu verbreiten. Peteris Krumins englische Lektüre über Perl-Einzeiler für $9,95 (*http://www.catonmat.net/blog/perl-book/*), Gilbert Steglers "deutsches Zukunft.pl" (*http://www.amazon.de/gp/product/B007B15Q3C/ref=kinw_myk_ro_title* für €4,11 (Rezension folgt) und Michael Mangelsdorfs "Web-Entwicklung mit Perl und Mojolicious" *http://www.amazon.de/Web-Entwicklung-mit-Perl-Mojolicious-Starthilfe/dp/3848200953/ref=sr_1_1?s=books&ie=UTF8&qid=1332878231&sr=1-1* (€8,20) sind nur drei jüngste Beispiele. Tom Christiansen verlangt für seine Schrift zum aktuellen Stand von Unicode in Perl 5 (*http://98.245.80.27/tcpc/scripts/perlunicook.html* kein Geld. Die *PDL*-Entwickler schreiben ebenfalls an einer freien, umfassenden Abhandlung *http://pdl.perl.org/content/pdl-book-toc.html*.

Eine Einführung in Perl der anderen Art ist Greg Londons kostenfreies "Impatient Perl" *http://www.greglondon.com/iperl/index.htm*, das für Schnelldenker ausgelegt ist. Beachtenswert ist auch Robert Naglers freies "Extreme Perl" *http://www.extremeperl.org/bk/home* welches mehr das "Extreme Programmieren" als Perl beleuchtet, aber dennoch einige nützliche Anregungen enthält.

Es werden sich aber hoffentlich noch neue Wege auftun, wie gute Dokumentation bezahlende Leser findet um seine Urheber zu entlohnen und trotzdem der Allgemeinheit zu Gute kommt. Die Perl Foundation bezahlt immer wieder Personen um Dokumentation zu schreiben. Meist sind es die wichtigen Projekte wie Moose oder Mojolicious, die so unterstützt werden. Einiges an Perl-Dokumentation war als bezahlter Artikel für *perl.com* entstanden und ist dank der Großzügigkeit von O'Reilly in den Perlkern gewandert.

Der Trainer und Buchautor Brian d Foy bewegte dort in den letzten Monaten viele Details und begann mit *perlexperiment* eine Übersicht der als "experimentell" markierten Funktionen. Mark Jason Dominus schaffte es seinen Verlag zu überreden, sein exzellentes "Higher Order Perl" (Rezension in $foo 2/2008) nach einer bestimmten Auflage als PDF frei auf *http://hop.perl.plover.com/book/* zur Verfügung zu stellen und verlässt sich auf Spenden. Dass so etwas gut funktionieren kann, beweist der Galileo Verlag, der regelmäßig sogenannte *openbooks* frei zugänglich macht, die er gleichzeitig auf Papier über den Handel vertreibt. Das umfassende Handbuch zu Linux *http://openbook.galileocomputing.de/linux/* ist erst unlängst in seiner fünften Auflage erschienen.

Cromatic, seines Zeichens Lektor bei O'Reilly, gründete mit der ehemaligen Chefin der Perl Foundation Allison Randal selbst einen Verlag namens Onyx Xeon um den Titeln auf die Sprünge zu helfen, welche die Gemeinschaft braucht, die aber kein Verlag wagen würde. Er tat den Anfang mit dem überaus populären "Modern Perl" (Rezension $foo 1/2011). Das wurde erst diesen Monat aktualisiert (nur Fehlerbereinigungen) und kann gleichwohl frei geladen oder gedruckt gekauft werden. Oder man lädt und spendet. Damit drückt man nicht nur seinen Dank aus und stellt Cromatics Aufmerksamkeit für die nächsten Auffrischungen sicher - es ist auch eine Vorauszahlung für weitere Schmöker, denn "Modern Perl Books" soll eine Reihe werden. Mindestens 5 Teile sind bereits in Arbeit.

Was trotz mancher Kontroverse das Buch "Modern Perl" ebenfalls auszeichnet, ist das zugehörige Blog *http://www.onyxneon.com/books/modern_perl/*, welches ständig neue Gedanken und Wissen veröffentlicht und das Gespräch mit den Lesern sucht. Auch "Effektiv Perl Programming" (Rezension 4/2010) *http://www.effectiveperlprogramming.com/* und "Learning Perl" (Rezension 2/2011) *http://www.learning-perl.com/* gehen diesen Weg. Von letzterem erschien Januar 2012 auf englisch ebenfalls ein 164-seitiges Arbeitsbuch welches für $3.99 ausschließlich elektronisch zu beziehen ist. Für die Perl-Lehrer unter den Lesern ist das sicher eine Option.

Das Buch der Bücher bleibt aber noch eine Weile das Kamel. Endlich erschien vor wenigen Wochen die schwergewichtige, vierte Auflage (*http://yfrog.com/z/oc5tncdpj*). Diese wird nun definitiv in der nächsten Ausgabe der $foo besprochen. Für dieses Mal gibt es einen anderen bekannten Titel, der leider nur noch digital vom Verleger angeboten wird.

Perl Hacks

Ein Hack ist eine schnelle Lösung. Sie kann praktisch sein, lustig oder die Grenze zum Missbrauch überschreiten, aber im Gegensatz zu hohen Versprechungen tut sie was sie soll, und zwar sofort.

Außerdem verlangt sie ein tieferes Verständnis der Materie, das man sich meist erst mit Ausdauer erarbeiten muss. Da manche Autoren die Aura eines Meisters gerne auch ohne vorherigen Aufwand versprühen, ist ein wenig Vorsicht bei Titeln, welche Worte wie "Hack" oder "cool" enthalten, angebracht ("Wicked Cool Perl Scripts"). Doch bei dieser Publikation ist sie unbegründet. Allein die Liste der Autoren enthält drei umtriebige Individuen, die mit CPAN-Modulen, YAPC-Vorträgen und umfangreichen Schriftwerken jahrelang ihren Sachverstand bewiesen haben, der noch von 18 weiteren Veteranen angereichert wurde.

Ist es nicht eher ironisch, dass ausgerechnet Schreiber, die andernorts für "Modernes Perl" und "Beste Praktiken" einstehen, hier erklären wie man alle Regeln beugen kann? Ganz und gar nicht. Zum einen bezieht sich eine ganze Reihe der 101 Hacks auf Informationsbeschaffungen (`corelist`, `perldoc -l`) und weitere, eher harmlose, aber nützliche Aktionen. Zum anderen bedarf es echter Könner, um zu überbli-

cken was wann angebracht ist. Denn der Wert dieses Schmökers besteht nicht nur im meist direkt anwendbaren Wissen. Jeder "Hack" wird wie eine Kurzgeschichte erzählt, die einem die Informationen und das Gefühl vermitteln wann sich der Einsatz lohnt.

"Perl Hacks" mag als kurzweiliges Spaßbuch daherkommen, welches in kleinen weitestgehend unabhängigen Kapiteln (meist nicht länger als 3 Seiten), anhand von Beispielen schrittweise einen Weg vom Problem bis zur Lösung weist. Aber trotz der unterhaltsamen, knappen und zielführenden Sprache erfährt man so viel über das Ökosystem Perl als hätte man einige Jahre nur damit verbracht. Einzig schade ist, dass einiges nicht mehr ganz auf dem Stand der Zeit ist. Die letzten 6 Jahre veränderten glücklicherweise vieles in der Perlwelt.

Der Inhalt ist breit gefächert. Neben zu erwartenden Themen wie Editoren (vi und emacs), Shell und Datenbanken geht es im Weiteren um Module, Objekte, Debugging, Testen bis hin zu Kapiteln, in denen die Freude an der abgefahrenen Idee überwiegt. Natürlich geht es ganz tief in Perls Rachen hinein. *Source-Filter*, die *Symboltabelle*, *Optree*-Manipulationen oder *Inlining* von *C*-Code per *P5NCI::Library* sind einige Themen, die man sonst verlegen lächelnd den Großmeistern überlässt. Doch hier liegen sie klein portioniert in Feinkost-Schalen - eine gute Gelegenheit es selbst einmal zu probieren. Das zeigt: Perl selbst zu kompilieren ist nicht so schwer wie gedacht. Aber einen systematischeren Rundgang durch solche Themen, wenn auch nicht in solch einer Tiefe, bietet Mastering Perl (Rezension in 1/2008).

Hier haben sich die Autoren bewusst nur stille Ecken der Sprache herausgesucht, an denen die meisten ängstlich oder achtlos vorbeigehen. Denn wer weiß schon, dass sich mit *Scalar::Util* zwei Werte in einer Variable speichern lassen. Es wurden auch Sachverhalte gewählt, die keinen großen Aufbau benötigen. Zum Beispiel begreift sich die Ablaufreihenfolge diverser `BEGIN`-, `INIT`- und `CHECK`-Blöcke wesentlich einfacher, wenn ein kleines und deutliches Beispiel den Nebel lichtet.

Obwohl nicht jeder Hack die große Offenbarung war, ist es bedauerlich, dass O'Reilly die Reihe eingestellt hat. Außerdem gibt es keine Aktualisierungen oder Fortsetzungen oder wenigstens ein Blog, in dem ab und zu weitere Hacks erscheinen.

Da das Buch nur noch als eBook erhältlich ist, wäre das *epub*-Format ideal oder wenigstens ein angepasstes *PDF*, das ohne weiteres Zutun den Bildschirm spezialisierter Lesegeräte besser ausnutzt. Das mag wegen der vielen Codebeispiele nicht trivial sein, würde aber die in Hängematten lesenden Programmierer erfreuen.

Coders at Work

Nun zu noch leichterer Literatur: 15 verdiente Szenegrößen, wirklich praktisch arbeitende "Coder", die jeweils mit einer Seite vorgestellt werden, plaudern frei von der Leber weg aus ihrem Leben und Denken. Das klingt in etwa nach "Visionäre der Programmierung" (Rezension 4/2011) - ein direkter Vergleich verdeutlicht auch Inhalt und Richtung dieses Werks.

Das "frei" war wörtlich gemeint, denn es gehört zu den süffisanten Freuden der Lektüre, Ken Thomson, der einen Vorgänger von *C* schrieb, dabei zu erleben wie er "Compiler sind Mist" ausruft. Auch *Unix*, *Tex*, *Plan 9*, *UTF-8*, *Smalltalk*, *Lisp*, *Erlang*, *Haskell*, *Java*, *Ghostscript*, *Javascript*, *JSON*, *OpenID*, *Firefox* und *Emacs* wurden von den Gesprächsteilnehmern erfunden oder mitentwickelt. Sie arbeiteten für *IBM*, *Sun*, *SGI*, *Netscape*, *Live Journal*, *Yahoo!*, *Google* und diverse Universitäten, was genügend Stoff für Anekdoten ist.

Tatsächlich konzentriert sich die Abhandlung auf den Werdegang. Jedes Interview beginnt mit der Frage "Wie sind Sie zur Programmierung gekommen?" und fließt beinahe nahtlos die Karriere entlang. Die thematischen Schwerpunkte bestimmten meinem Empfinden nach die Gefragten. Selbstverständlich hat sich Autor und Programmierer Peter Seibel auf jedes Gespräch vorbereitet, jedoch im Gegensatz zu den "Visionären" wahrscheinlich weniger Fragen vorformuliert. Dort werden auch mehr spezielle technische Details besprochen als bei den "Coders".

Die generellen Themen neben den beruflichen Herausforderungen sind: "Was macht gute Software aus?", "Wie erkennt man gute Programmierer?" und "Wie sollte man letztere ausbilden?". Knuths *literalische Programmierung* wird wiederholt angesprochen. Dabei sammelte Seibel Literaturvorschläge, weswegen die Bibliographie des in Perl kundigen Autors ein feine Liste beachtenswerter Titel beinhaltet, die auch "Higher Order Perl" umfasst. Dieser Hinweis kam von

Brad Fritzpatrick, der sich als einziger positiv über Perl auslässt und mit seinen 32 Lenzen mit Abstand der jüngste Porträtierte ist.

Mindestens ein Drittel begann mit dem Programmieren in der Lochkarten-Ära, aber zum Glück breitet sich nirgends die Stimmung aus: "Opa erzählt vom Krieg". Im Gegenteil - der Text liest sich sehr flüssig und man begreift, welche Geschichte die heutige Informatik hat und mit welcher Haltung sie geschrieben wurde. An manchen Stellen wäre eher eine Vertiefung wünschenswert.

Trotz der 560 Seiten (in einem handfreundlichen, etwas kleineren Format, was etwa 450 gewohnten Seiten entspricht) hat das Buch keine spürbare Schwere oder Überlänge, wozu auch der recht offene und menschliche Ton beiträgt. Allerdings verwenden diese gebildeten Informatiker viele Fachbegriffe, weshalb es kein Stoff für Anfänger ist. Selbst Fortgeschrittene müssen alle paar Seiten etwas nachschlagen. Das ist Literatur für entspanntes Dazulernen. Diese Seiten sind für Gehirne, die Freude daran haben nachzuvollziehen, wie technische Details, menschliche Interaktion und Firmenpolitik komplexe Projekte formten. Sie können aber auch einfach nur Unterhaltung für "Coder" sein.

Sicher ist das Gesagte oft nur Meinung und manche Weisheiten dämmerten sicher bereits den meisten Lesern, wie etwa: "Übermüdet oder unpässlich zu Programmieren ist so unproduktiv, dass man sich besser gleich schlafen legen sollte." Aber manchmal ist der einfache Rat der beste, weil er praxiserprobt ist. Beim gefühlten Schulterschluss mit den Legenden bemerkt man, dass es auch nur Menschen sind und sollte nicht immer überirdische Weisheit erwarten.

Trotzdem kann das Eintauchen in 15 verschiedene Sichtweisen die eigene Entwicklung zu einem umsichtigeren Gestalter sehr beschleunigen, wenn man die wichtigsten Erfahrungen anderer im Zeitraffer überfliegen kann und sie zu den eigenen Schätzen hinzufügt.

Leserbriefe

Ich habe erneut versucht, die VMware VIX API unter Debian nutzen zu können. Heureka - nachdem ich bei meinen letzten Versuchen auf Granit gebissen hatte, habe ich diesmal die Lösung gefunden! Ersten Tests nach zu urteilen funktioniert die API für Perl nun unter Debian 5, 6 und dem derzeitigen Stand der kommenden Version 7!

Für Interessierte die Lösung: Problematisch waren anscheinend nicht die Perl-Bindings, sondern die Bibliotheken, auf die über die Bindings zurückgegriffen wird. Mein Ansatz war diesmal, für alle infrage kommenden VMware-Bibliotheken zu ermitteln, ob diese ihrerseits möglicherweise nicht erfüllte Abhängigkeiten im Bereich dynamisch gelinkter Bibliotheken haben. Folgender Befehl brachte die Erleuchtung:

```
find /usr/lib -name "libvix*" -exec ldd {} \;
  2> /dev/null | grep 'not found' | sort |
   uniq
```

Auf einem funktionierenden openSUSE wurde hier lediglich eine fehlende Bibliothek moniert, auf dem nicht funktionierenden Debian waren es fünf. Die vier zusätzlichen Bibliotheken konnte ich nach kurzer Recherche dem Paket "libglib2.0-0" zuordnen. Die Beschreibung des Paketsystems hält hierzu folgende Informationen bereit: "GLib ist eine Bibliothek mit vielen nützlichen C-Routinen für Dinge wie Bäume, Hashes, Listen und Strings. Es ist eine nützliche Mehrzweck-C-Bibliothek, die von Projekten wie GTK+, GIMP und Gnome genutzt wird." Beide Linux-Distributionen hatte ich zwar in der Minimal-Variante installiert, aber der Begriff "minimal" wird offensichtlich unterschiedlich interpretiert. Die openSUSE-Paketierer sind etwas großzügiger gewesen und haben die "nützliche Mehrzweck-C-Bibliothek" jovial mitspendiert.

Rückblickend lässt sich also festhalten, dass die Meldung "HostConnect() failed, 6000 The operation is not supported for the specified parameters" durch das nicht installierte Paket "libglib2.0-0" zustande kam. Voll knorke wär 's gewesen, wenn ein wohlgesonnener VMware-Fraggle hierzu mal einen dezenten Hinweis in die Release-Notes eingestreut hätte. Aber nun gut - selbst ist der Monger ;-)

Nun kann ich doch noch auf dem von mir favorisierten OS das Projekt in Angriff nehmen, das mir ursprünglich mal vorschwebte.

Danke für 's Zuhören und freudige Grüße,
Stefan

TPF News

perl.com sucht Artikel

Seit dem letzten Jahr ist die Perl Foundation für die Domain perl.com zuständig. Dort sollen *Erfolgsgeschichten* von Perl und andere interessante Artikel veröffentlicht werden. Wer einen Artikel für perl.com hat, kann sich an editor@perl.com wenden.

Craigslist spendet 100.000 USD an Perl Foundation

Die Organisation Craigslist Charitable Fund spendet 100.000 USD an die Perl Foundation. Das Geld ist zum Teil für den Perl 5 Core Maintenance Fund gedacht. Der Rest soll für allgemeine Aufgaben der Perl Foundation genutzt werden, wie z.B. allgemeine Grants.

Craigslist ist ein Netzwerk von Foren, bei dem die Mitglieder alles Mögliche finden können. Mit rund 30 Milliarden Views pro Monat gehört die Organisation zu den Top-5-Unternehmen bei den englischsprachigen Seitenaufrufen.

Fixing Perl 5 Core Bugs

Mittlerweile hat Dave Mitchell rund 1.100 Stunden mit dem Fixen von Bugs verbracht. In der letzten Zeit hat er sich hauptsächlich damit beschäftigt die Code-Evaluierung innerhalb von regulären Ausdrücken zu festigen.

Improving Perl 5

Gerade erst wurde der Grant von Nicholas Clark verlängert. Diese Verlängerung umfasst 400 Arbeitsstunden von Clark, für die er 20.000 USD bekommen wird.

Zu den Arbeiten von Nick Clark gehören Änderungen an Porting-Skripten und aktuell an Cross-Compiling-Problemen.

Grants für das 1. Quartal 2012

Im ersten Quartal wird die Perl Foundation zwei Grants vergeben.

Zum einen "Alien::Base", bei dem Joel Berger eine Basisklasse für Alien::*-Module erstellt. Mit den Alien::*-Modulen sollen Installationsmodule für "kompilierte" Bibliotheken zur Verfügung gestellt werden, die von CPAN-Modulen benötigt werden.
Der zweite Grant ist "Cooking Perl with Chef". Chef ist ein Werkzeug für das Konfigurationsmanagement. Mit diesem Grant soll eine Anbindung an Perl-Projekte möglich sein.

Webseite: *http://news.perlfoundation.org/2012/02/2012q1-grant-results.html*

Google Summer of Code 2012 - Die Perl Foundation ist leider nicht dabei

In den vergangenen Jahren war die Perl Foundation immer beim Google Summer of Code als Mentorenorganisation dabei. In diesem Jahr wurde die Perl Foundation nicht angenommen. Über verschiedene Projekte wie BioPerl sind Perl-Projekte trotzdem möglich. Im nächsten Jahr gibt es wohl einen neuen Versuch, als Mentorenorganisation angenommen zu werden.

Reise-Grant für Ricardo Signes

Ende März fand in Paris der Perl QA-Hackathon statt. Die Teilnehmer dort haben an verschiedenen Projekten der Testinfrastruktur von Perl gearbeitet. Die Perl Foundation hat die Reisekosten von Ricardo Signes, dem aktuellen Perl Pumpking, übernommen. Dies ist aus Sicht der Foundation notwendig, damit die Bemühungen mit den Arbeiten am Perl-Kern koordiniert werden.

Renée Bäcker

Merkwürdigkeiten - Ein Buchstabendreher verändert die Welt

Manchmal haben kleine Fehler eine große Auswirkung. Ein Beispiel dafür sind die folgenden zwei Zeilen:

```
$ perl -we "q(di)"
$ perl -we "q(id)"
Useless use of a constant in void
  context at -e line 1.
```

Die erste Zeile läuft normal durch, aber bei der zweiten Zeile gibt es eine Warnung. Die besagt, dass der String "id" im void-Kontext sinnfrei ist. Warum ist aber der String "di" erlaubt? Wo liegt hier der Unterschied?

Ein Blick in die Vergangenheit

Ein Blick in alte Perl-4-Tage hilft zu verstehen, warum bei der ersten Variante keine Warnung kommt. Denn da gab es das POD-Format noch nicht, aber trotzdem sollte innerhalb des Programms dokumentiert werden. Und nicht nur mit Kommentaren, denn die kann man nicht als *manpage* anzeigen lassen. Zur Anzeige von Dokumentation wurde sehr häufig `nroff` benutzt. Deshalb wurden einige `nroff`-Befehle in Perl so berücksichtigt, dass sie keine Warnungen erzeugen. Und *di* ist einer dieser Befehle, und sie funktionieren auch noch in Perl 5.16 (und sie werden auch in zukünftigen Perl5-Versionen funktionieren).

`nroff` beachtet alles, was zwischen ".di X" und ".di" steht, wird in ein `nroff`-Makro "X" umgeleitet, und alles was zwischen ".ig Y" und ".Y" ("ig" ist ein weiterer `nroff`-Befehl) steht, wird ignoriert. Mit `__END__` wird der Perl-Parser angewiesen mit dem Parsen aufzuhören, weil kein Perl-Code mehr kommt. Danach kann man also die Dokumentation für `nroff` schreiben. Mischt man alles, kann man folgendes machen - siehe Listng 1.

Jetzt einfach mal ausprobieren:

```
$ perl nroff.pl
yes
$ nroff nroff.pl
prints simply "yes"
```

Weitere Ausnahmen von der Warnung

Es gibt weitere Ausnahmen von der *Useless use of a constant...*-Warnung: Alle Ausdrücke, die zu 1 oder 0 evaluieren:

```
$ perl -wle '5'
Useless use of a constant (5)
  in void context at -e line 1.
$ perl -wle '1'
$
```

Das hat jetzt nicht `nroff` als Hintergrund, sondern dass Konstrukte wie

```
my $zahl = 12345678;
1 while $zahl =~ s/(\d+)(\d{3})/$1.$2/;
```

möglich sind. Es gibt einige dieser Konstrukte, bei denen die "Arbeit" im Schleifenkopf erledigt wird, man aber einen Schleifenkörper benötigt. Der muss ja nichts machen.

Auch mit einer `until`-Schleife funktioniert das:

```
0 until do_somethin();
```

Solche Feinheiten von Perl findet man immer wieder und nicht immer ist das was wie ein Bug aussieht auch wirklich ein Bug.

```
'di ';
'ig 00 ';
print "yes\n";
.00 ;
'di
'; __END__
.TH $foo Nr 22 - Merkwuerdigkeiten
.AT 3
.SH DESCRIPTION
prints simply "yes"
```

Listing 1

CPAN News XXII

An dieser Stelle zeigen wir wieder sechs Module, die entweder komplett neu sind oder bei denen neue Versionen zur Verfügung stehen. Ab der nächsten Ausgabe wird das hier etwas anders aussehen (siehe auch Vorwort).

Google::Directions

Wie komme ich von A nach B? Diese Frage stellt man normalerweise seinem Navigationssystem. Wer diese Frage seinem Perl-Programm stellen möchte, sollte sich `Google::Directions` anschauen. Die Dokumentation ist (noch) nicht so toll, aber mit der API-Beschreibung direkt von Google und der rudimentären Dokumentation des Moduls, kommt man ganz gut zurecht. Für die Nutzung der API braucht man keinen Google-Account, sondern kann einfach loslegen.

So bekommt man den Weg von den Baseler Arkaden in Frankfurt zur Goethe-Uni in Frankfurt:

```
use Google::Directions::Client;

my $start = 'Baseler Arkaden, Frankfurt';
my $stop  = 'Goethe Universität, Frankfurt';

my $client = Google::Directions::Client-
>new();
my $directions = $client->directions(
    origin => $start,
    destination => $stop,
);

my $route = shift @{$directions->routes};
for my $leg ( @{ $route->legs } ) {
    for my $step ( @{ $leg->steps } ) {
        my $instruction =
           $step->html_instructions;
        $instruction =~ s!<.*?>!!g;
        print sprintf "%s (%s m)\n",
           $instruction, $step->distance;
    }
}
```

Time::UTC::Now

Alle denken, dass mit der Zeit ist ganz einfach. Wer auf der letzten YAPC::Europe in Riga war und Zefram gelauscht hat, weiß dass es extrem viele Fallstricke gibt. Kein Wunder, dass genau von ihm ein Modul stammt, mit dem man die UTC-Zeit korrekt herausfinden kann: `Time::UTC::Now`.

```
use Time::UTC::Now
 qw(now_utc_rat now_utc_sna
    now_utc_flt now_utc_dec);

($day, $secs, $bound) = now_utc_rat;
($day, $secs, $bound) = now_utc_rat(1);
($day, $secs, $bound) = now_utc_sna;
($day, $secs, $bound) = now_utc_sna(1);
($day, $secs, $bound) = now_utc_flt;
($day, $secs, $bound) = now_utc_flt(1);
($day, $secs, $bound) = now_utc_dec;
($day, $secs, $bound) = now_utc_dec(1);

use Time::UTC::Now
 qw(utc_day_to_mjdn utc_day_to_cjdn);

$mjdn = utc_day_to_mjdn($day);
$cjdn = utc_day_to_cjdn($day);
```

WWW::xkcd

Wahrscheinlich kennen viele schon xkcd. Dort findet man immer wieder tolle kleine Comicstrips, die teilweise auch Verweise auf Perl enthalten. Mit `WWW::xkcd` gibt es jetzt ein Modul, mit dem man einzelne Zeichnungen und viele Informationen darüber bekommen kann. Leider gibt es noch keine Suchfunktion.

Wer sich die Metadaten holt, bekommt auch ein Transskript des Comics, sowie URL des Bildes und Veröffentlichungsdatum.

In dem Beispiel wird ein Comic geholt, das zeigt, wie wertvoll Perl für die Karriere sein kann ;-)

```
use WWW::xkcd;

my $xkcd = WWW::xkcd->new;
my ($image,$comic) = $xkcd->fetch(519);

use Data::Dumper;
print Dumper $comic;
```

```
  $VAR1 = {
     'link' => '',
     'alt' => 'And the ten minutes striking
               up a conversation with that
               strange kid in homeroom
               sometimes matters more than
               every other part of high
               school combined.',
     'num' => 519,
     'month' => '12',
     'transcript' => '[[Bar graph title:
          Usefulness to career success]...',
     'safe_title' => '11th Grade',
     'img' =>
'http://imgs.xkcd.com/comics/11th_grade.png',
     'day' => '19',
     'title' => '11th Grade',
     'news' => '',
     'year' => '2008'
   };
```

File::HTTP

Da liegt eine Datei im Netz, die man gerne auslesen möchte. Dann heißt es erstmal "Datei herunterladen" bevor man mit `open` etc. arbeiten kann. Stopp! Mit `File::HTTP` ist es jetzt möglich, die gewohnten Befehle wie `open` oder `<>` zu benutzen. Schreiben kann man damit nicht, aber das Auslesen von Dateien ist schon mal sehr praktisch.

Wenn der Server DirectoryListings unterstützt kann man sogar `opendir` etc. verwenden. Sehr praktisch. Mit dem Beispiel kann man sich zum Beispiel die Quelle für diesen Artikel holen!

```
use File::HTTP qw(:open);

my $url =
    'http://downloads.perl-magazin.de/
       cpan_news.pod';
open my $fh, '<', $url or die $!;
while ( my $line = <$fh> ) {
    next if $line !~ /^=head2/;
    $line =~ s/=head2\s+//;
    print "vorgestelltes Modul: $line";
}
```

Contextual::Return

Welchen Kontext kennt Perl? Scalar, List und Void! Mit `Contextual::Return` kann man auf noch viel mehr Dinge reagieren, z.B. ob man in einem boolschen Kontext ist, ob ein Hash verlangt wird, oder, oder, oder. Damian Conway hat hier wieder zugeschlagen und das möglich gemacht. Dann muss man auch nicht mehr mit `wantarray` arbeiten.

```
use Contextual::Return;

sub foo {
    return
        SCALAR { 'Sommer 2012' }
        LIST   { 'Artikel1', 'Artikel2' }

        BOOL { 1 }
        NUM  { 22 }
        STR  { 'Ausgabe 22 (Sommer 2012)' }

        HASHREF  { {name => 'foo',
                    value => '22'} }
       ;
}

my $foo = foo();
print "This is $foo->{name} nr ", ($foo+0),
" with these articles: @{$foo}\n";
__END__
This is foo nr 22 with these articles:
Artikel1 Artikel2
```

List::MapMulti

Mit `List::MapMulti` können Konstrukte mit zwei `for`-Schleifen, in denen man beide Schleifenvariablen miteinander verwendet, vermieden werden.

```
use List::MapMulti qw/mapm/;

my @numbers = (2..10);
my @hardware   = qw/Server Laptop Desktop/;
my @names = mapm { "$_[0]$_[1]" } \@hardware,
                                  \@numbers;

print scalar(@names),"\n"; # says '27'
print $names[0],"\n";      # says 'Server2'
print $names[11],"\n";     # says 'Laptop4'
print $names[-1],"\n";     # says 'Desktop10'
```

Termine

Mai 2012

01.	Treffen Stuttgart.pm
08.	Treffen Frankfurt.pm
12.-13.	Perl Mova (Ukraine)
14.	Treffen Ruhr.pm
16.	Treffen Darmstadt.pm
21.	Treffen Erlangen.pm
29.	Treffen Bielefeld.pm
30.	Treffen Berlin.pm

Juni 2012

05.	Treffen Stuttgart.pm
	Treffen Frankfurt.pm
11.	Treffen Ruhr.pm
13.-15.	YAPC::NA
18.	Treffen Erlangen.pm
20.	Treffen Darmstadt.pm
26.	Treffen Bielefeld.pm
27.	Treffen Berlin.pm
29.-30.	Französischer Perl-Workshop

Juli 2012

03.	Treffen Frankfurt.pm
	Treffen Stuttgart.pm
09.	Treffen Ruhr.pm
16.	Treffen Erlangen.pm
16.-18.	OSCON
18.	Treffen Darmstadt.pm
24.	Treffen Bielefeld.pm
25.	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.perl-foundation.org

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

http://www.Perl.org

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

YAPC::EU 2012
August 2012 in Frankfurt

Die YAPC::EU ist mit rund 300 Teilnehmern die größte europäische Veranstaltung rund um Perl...

Interesse an Sponsoring?
sponsoring@yapc2012.de

BOOKING.COM

online hotel reservations

NOW HIRING!

SysAdmins

MySQL DBAs

Perl Devs

Software Devs

Web Designers

Front End Devs ...

Interested? Booking.com/jobs

Booking.com B.V., part of Priceline.com (Nasdaq:PCLN), owns and operates Booking.com (TM), one of the world's leading online hotel reservations agencies by room nights sold, attracting over 30 million unique visitors each month via the Internet from both leisure and business markets worldwide.

We use Perl, puppet, Apache, MySQL, Memcache, Git, Linux …and many more!

Established in 1996, Booking.com B.V. guarantees the best prices for any type of property, ranging from small independent hotels to a five star luxury through Booking.com. The Booking.com website is available in 41 languages and offers 120,000+ hotels in 99 countries.

- Great location in the center of Amsterdam
- Competitive Salary + Relocation Package
- International, result driven, fun & dynamic work environment