Ausgabe 2/2011 – www.foo-magazin.de - 6,00 €

$foo

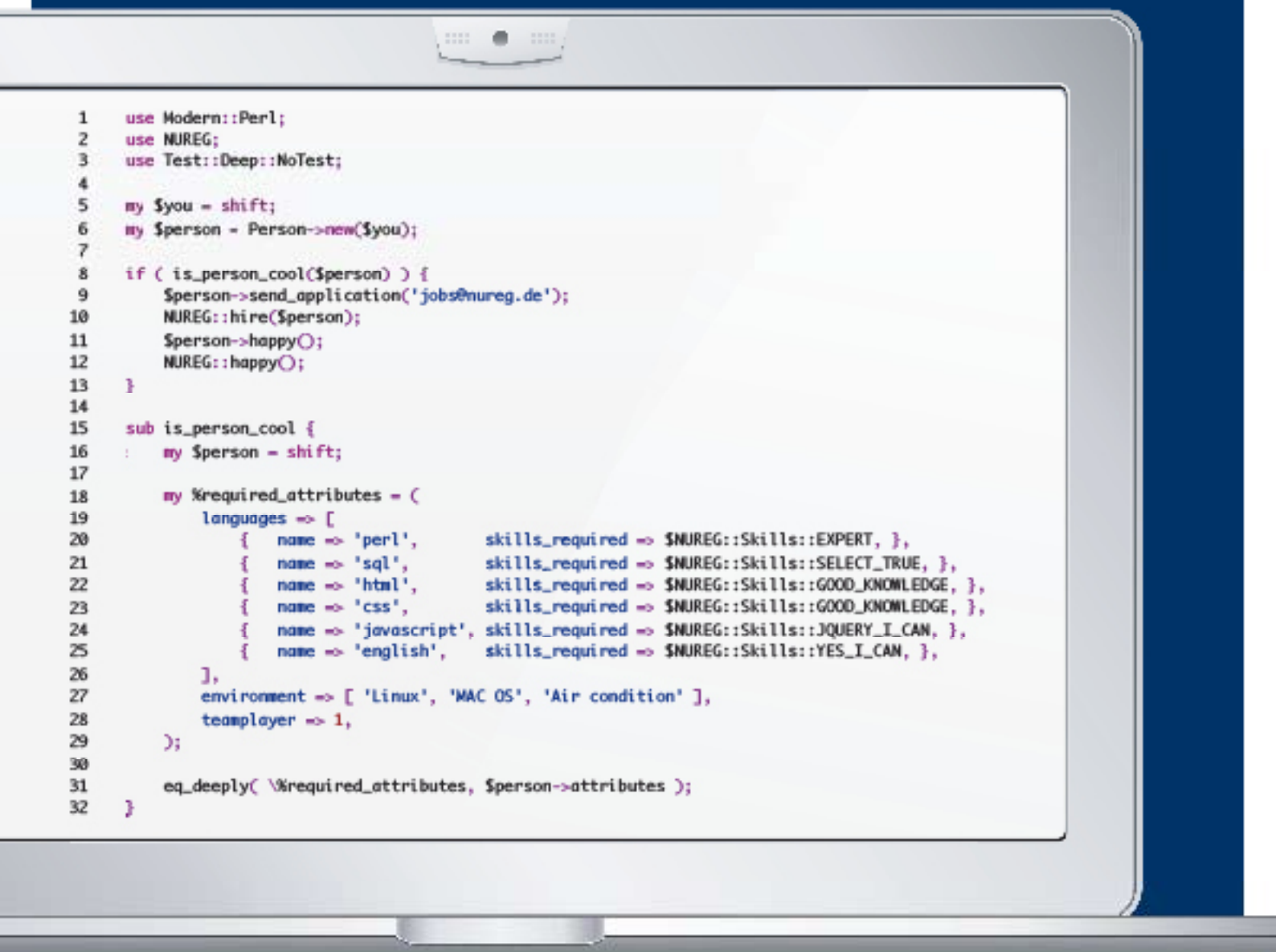
PERL MAGAZIN

CPAN-Distributionen
... im openSUSE Build Service paketieren

Tapper
- eine modulare Testinfrastruktur

Hack your applications!
Universelle Plugins scripten via Mailto

Nr 18



```perl
use Modern::Perl;
use NUREG;
use Test::Deep::NoTest;

my $you = shift;
my $person = Person->new($you);

if ( is_person_cool($person) ) {
    $person->send_application('jobs@nureg.de');
    NUREG::hire($person);
    $person->happy();
    NUREG::happy();
}

sub is_person_cool {
    my $person = shift;

    my %required_attributes = (
        languages => [
            {   name => 'perl',       skills_required => $NUREG::Skills::EXPERT, },
            {   name => 'sql',        skills_required => $NUREG::Skills::SELECT_TRUE, },
            {   name => 'html',       skills_required => $NUREG::Skills::GOOD_KNOWLEDGE, },
            {   name => 'css',        skills_required => $NUREG::Skills::GOOD_KNOWLEDGE, },
            {   name => 'javascript', skills_required => $NUREG::Skills::JQUERY_I_CAN, },
            {   name => 'english',    skills_required => $NUREG::Skills::YES_I_CAN, },
        ],
        environment => [ 'Linux', 'MAC OS', 'Air condition' ],
        teamplayer => 1,
    );

    eq_deeply( \%required_attributes, $person->attributes );
}
```

Verstehen wir uns richtig?

Dann sollten wir uns unterhalten. Wir suchen Verstärkung für die Weiterentwicklung unserer E-Commerce-Plattformen für renommierte internationale Kunden. Als Mitglied des Entwicklungs-Teams sind Sie für die Implementierung neuer Internet-Applikationen bzw. Erweiterung bestehender Systeme zuständig. Wenn Sie sich jetzt angesprochen fühlen, dann freuen wir uns auf Ihre Bewerbung als:

°°° | Software-Entwickler (m/w)

Es erwarten Sie ein angenehmes Arbeitsumfeld, ein Arbeitsplatz mit Zukunftsperspektive, abwechslungsreiche Tätigkeiten sowie attraktive Konditionen.

Ihre Bewerbungsunterlagen schicken Sie bitte vorzugsweise als Onlinebewerbung an jobs@nureg.de oder an folgende Adresse: NUREG GmbH | Personalabteilung/IT | Dorfäckerstraße 31 | 90427 Nürnberg

Mehr Infos unter www.nureg.de/jobs

NUREG °°°

VORWORT

Die Zukunft des Deutschen Perl-Workshops...

Vielen ist vielleicht aufgefallen, dass es in diesem Jahr noch keinen Deutschen Perl-Workshop gegeben hat. Aus verschiedenen Gründen hat sich die Organisation hingezogen, aber jetzt haben sich die Frankfurter Perlmongers bereiterklärt, in die Bresche zu springen und einen Workshop auszurichten.

Im Oktober findet der Workshop dann wieder - wie 2009 - in Frankfurt statt. Der genaue Termin und der "Call for Papers" werden noch veröffentlicht.

Aber darüber hinaus haben Diskussionen begonnen, unter welcher Organisation der Workshop in Zukunft ausgerichtet wird. Jürgen Christoffel hat auf der Mailingliste der Workshop-Organisatoren bekannt gegeben, dass sich die GbR ab 2012 aus dem Workshop herauszieht. Jetzt sind Überlegungen da, dass ein Verein gegründet werden kann, der sich in Zukunft um den Workshop kümmert.

Natürlich läuft so ein Verein nicht ohne die Mithilfe vieler Leute. Mitte des Jahres soll es ein Arbeitswochenende geben, auf dem sich Interessierte zu diesem Thema austauschen können und vielleicht schon ein Verein auf den Weg gebracht wird. Wann und wo das stattfindet, steht noch nicht fest, aber das werde ich auf den üblichen Kanälen bekannt geben. Wer darf daran teilnehmen? Jeder! Wer Interesse hat, an so einem Verein mitzuwirken, kann eine Mail an verein@perl-services.de schicken, damit Informationen direkt weitergegeben werden können.

Es wäre toll, wenn sich viele daran beteiligen würden...

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

Renée Bäcker

Die Codebeispiele können mit dem Code

8947bvdm

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/issue18

IMPRESSUM

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

INHALTSVERZEICHNIS

ALLGEMEINES

PERL

MODULE

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.

Rolf Langsdorf

Rolf "LanX" Langsdorf hat seit seiner Geburt in den tiefen Asiens viele Leben durchlaufen. Einmal hat er sein Wirken der Erforschung unbewiesener Vermutungen gewidmet und verendete mit einem Diplom der Mathematik. Dann hat er mehreren Großbanken geholfen, die Überwachung ihrer Server auf Stasi-Niveau zu heben und belastete sein Karma mit schicken Krawatten, blamierten IT-Giganten und Konten jenseits des Bafögsatzes. Später widmete er ein Leben dem E-Learning und meditierte in einem erimitischen interdisziplinären Promotionsstudium. In all diesen Inkarnationen kontemplierte er in immer stärkeren Maße über Perl und Emacs. Letzteres so sehr, dass er nun mit 20 Fingern reinkarniert wurde. Und obwohl er fortan alle "Modifier Keys" gleichzeitig erreichen kann, verfolgt unseren Guru die Paranoia, ob Metatasten metastasieren können.

Steffen Schwigon

Steffen Schwigon wurde 1972 geboren und hat kurz danach, ok, 1987, angefangen zu programmieren. Atari Basic, Pascal, C, C++, Java, Perl. In der Reihenfolge. Debian GNU/Linux ist sein Betriebssystem, XEmacs sein Editor. In Perl programmiert er seit 2002. Er ist aktives Mitglied der Dresden Perl Mongers.

Matthias Weckbecker

Matthias Weckbecker arbeitet als Software Engineer beim Nürnberger Linux-Vendor SUSE. Er ist dabei hauptsächlich verantwortlich für den Support der Enterprise Distribution sowie Sicherheit und Qualität dieser. In der restlichen Zeit unterstützt er nebenbei gerne das openSUSE Projekt, programmiert (natürlich) in Perl und schreibt hin und wieder Artikel.

Rolf Langsdorf

Wie erweitere ich Perls Syntax? - Teil 2
geLISPelte 6ismen: Macros und Gather/Take

Rückblick

Im ersten Teil unserer Reihe (Ausgabe 4/2010) haben wir gesehen, wie wir durch die Kombination *Funktionaler Programmierung* einerseits und Perls Möglichkeiten des *Prototyping* von Funktionen andererseits soviel *Syntactic Sugar* gewinnen konnten, dass wir Kontrollstrukturen einer anderen Sprache elegant nachbilden konnten.

Wir hatten dazu als Beispiel die "List Comprehensions" betrachtet, die es wie `map` und `grep` erlauben, die Verarbeitung von Listen mehrstufig zu notieren, dabei aber mehr Nutzungsmöglichkeiten bieten, z.B zur eleganten Notation von Iteratoren.

So wie wir mit einem Einzeiler in Haskell die ersten 10 Lösungen der Gleichung $a^2+b^2=c^2$ gewinnen können,

```
take 10 [ (a, b ,c) |
  c <- [1..n], b <- [1..c], a <- [1..b],
  a^2 + b^2 == c^2 ]
```

konnten wir nun in Perl folgendes notieren:

```
take { [$a,$b,$c] } 10
 => filter { $a**2 + $b**2 == $c**2 }
 => from {1..999} $c  => from {1..$c} $b
 => from {1..$b} $a
```

Syntax != Semantik

Und obwohl es uns hier gelang schleifenähnliche Konstrukte nachzubilden, haben wir streng genommen Perls Syntax nicht erweitert, sondern vielmehr die Ausdrucksfähigkeit unserer Lieblingssprache ausgeschöpft. Tatsächlich haben wir hier - was oft verwechselt wird - die Sprache **Semantisch** um ein Idiom bereichert. **Syntaktisch** sind `take()`, `filter()` und `from()` aber nur Funktionen.

Diese vermeintliche Spitzfindigkeit hat Auswirkungen im Bereich der Kompatibilität zu vielen Entwicklungswerkzeugen. Wir bleiben nämlich zu allen bisherigen Syntaxparsern kompatibel.

Also, weder würden Idiome, die nach unserem Ansatz designt sind, Probleme mit dem Highlighting in einem Editors bekommen, noch würde `Perl::Critic` hier anfangen zu meckern.

Bei vielen *echten* Syntaxerweiterungen, z.B. mittels XS oder Sourcefiltern oder `Devel::Declare` stolpert aber so manches Tool. Wer kümmert sich beispielsweise darum `Perl::Tidy` zu erweitern, so dass die Konstrukte von `MooseX::Declare` auch formatiert werden können?

Zu den letztmalig diskutierten acht Überlegungen, wie Sprachерweiterungen eine größtmögliche Akzeptanz erreichen könnten, kommt nun eine neue hinzu:

These 9: *Perl ist jetzt schon bereits zu schwer zu parsen, als das wir unnötig tiefe Eingriffe in Syntax und Lexer versuchen sollten.*

Was uns noch fehlte

Bei unserem Anwendungsbeispiel waren wir aber noch nicht gänzlich zufrieden, weil Haskell in manchen Bereichen noch die Nase vorne hatte:

1. DRY - Don't Repeat Yourself:
Vernünftiger Perlcode muss heutzutage `strict` sein. Um unsere Laufvariablen `$a`, `$b` und `$c` mit `my` zu deklarieren,

bräuchten wir aber eine extra Zeile vorweg und einen Block drumherum um den Scope einzuschränken.

2. Die Geschwindigkeit:
Bei zu vielen Rekursionsebenen für Iteratoren wirkt sich der Overhead der Funktionsaufrufe erheblich aus. In unserem obigen Beispiel sind es leicht Faktor 100 im Vergleich zu 3 geschachtelten Schleifen.

3. Unendlichkeit:
Haskell erlaubte mir mit `c <- [1..]` pro forma eine unendliche Liste zu durchlaufen, und ein Abbruchkriterium davon entkoppelt zu notieren. Perl5 kennt zwar tatsächlich den Bezeichner `Inf`, lässt diesen aber nicht mit dem Rangeoperator zu. (`Inf` gehört zur Fließkommabibliothek und wird nicht als Integer akzeptiert.)

Koroutinen in Perl6

Um diese Mängel angehen zu können, bietet es sich an einen thematischen Schlenker zu Perl6 zu machen. Dort ist für vergleichbare Aufgabenstellungen das `gather/take`-Konstrukt vorgesehen:

```
my @arr = gather {
   for 0..20 -> $i {
       take $i if  $i % 2;
   }
}
```

Das folgende Perl5 Pendant spiegelt dabei nur ungefähr die Funktionsweise wider:

```
@arr = do {
  my @tmp;
  for my $i (0..20){
    push @tmp,$_ if ( $i % 2);
  }
  @tmp
};
```

Zum Verständnis: Arrays in Perl6 sind defaultmäßig so genannte *Lazy-Lists*, will heißen das Array ist so "faul" ein Element erst bei Bedarf zu berechnen. Erst wenn ich `arr[5]` abrufe, werden die Elemente bis 5 berechnet und gespeichert. `take` gibt diese ähnlich einem `return` einzeln zurück, aber mit dem feinen Unterschied, dass der Zustand - das heißt alle Variablenbelegungen innerhalb des Codeblocks - nach dem letzten `take` eingefroren werden. Rufe ich später `@arr[10]` ab, wird beim letzten Ausstiegspunkt mit Element 6 usw. weitergemacht, als wäre nichts gewesen. (Nebenbei: Man kann deswegen in Perl6 sogar mit `@arr=1..Inf` gefahrlos "unendliche" Arrays anlegen.)

Hört sich zu akademisch an? Mal ein etwas anschaulicheres Beispiel:

Angenommen man will aus einem HTML File unbekannter Größe flexibel den x-ten Eintrag der y-ten Zeile der z-ten Tabelle extrahieren.

Dann könnte man mit `gather/take` Code schreiben, der `@table` befüllt. `@table[$z]` wird mit einem neuen `gather/take` durchlaufen, um `@tr[$z][$y]` zu erhalten und entsprechend wird `@td[$z][$y][$x]` "gefüllt". Die Anführungszeichen deswegen, weil solange man nicht auf diese Arrays zugreift, weder ein Parser angelaufen ist, noch Speicher belegt wurde. Ob das komplette HTML-File zerlegt in den Speicher passt oder wie lange das Programm zum Starten braucht, bereitet einem also keine Kopfschmerzen. Und ich kann jederzeit flexibel auf ganze Tabellen oder Zeilen zugreifen.

Das entspricht ziemlich genau dem Verhalten, das wir uns von unseren "Lazy Iteratoren" wünschen, mit dem Bonus, ein sprechendes `@` Sigil zu haben.

Man kann übrigens auch mehrere `take`s in einem Codeblock platzieren, beim zweiten Aufruf geht es einfach da weiter, wo das erste erreichte `take` ausgestiegen war. Solche Konstrukte werden deshalb in der Informatik auch als **Koroutinen** bezeichnet. Aktuelle Versionen einer anderen Scriptsprache deren Name auch mit "P" beginnt, kennen auch ein Statement namens `yield`, welches diesem `take` entspricht.

Ach ja ... und in Perl5 ?

Um diese "Einfriererei" in Perl5 ungefähr nachstellen zu können, greifen wir nun tief in die Tabu-Kiste und zaubern das gute alte `goto LABEL` heraus - siehe Listing 1.

Der Body-Teil beschreibt dabei die vom User gewünschte Logik, also `$i` bis 20 hochzuzählen und nur die ungeraden Ergebnisse zu "nehmen". Der restliche Rahmen-Code ist generisch und dient nur der Steuerung unserer Koroutine.

```
{
  my $RESUME;
  my $ONERUN=1;

  my $DB=0;   # Debugflag

  #-- Closure Vars
  my $i;

  $c_iter = sub {
    #-- Dispatcher
    goto $RESUME
      if defined $RESUME;

    #-- Body
    for ($i=0; $i<20; $i++){
      print "-DB:$i\n" if $DB;

      #-- take Emulation
      do {
       $RESUME="ENTER_1";
       return($i);
      ENTER_1:
      } if ( $i % 2 );
    }

    #-- Exit
    $RESUME = $ONERUN ? "FINISHED": ();
    FINISHED:
      return;
  }
}
```

Listing 1

Der `do`-block emuliert dabei den take-Mechanismus, indem eine Sprungmarke für den Wiedereinstieg gesetzt und gemerkt wird. Und da `$i` eine Closurevariable im äußeren Scope ist, geht sein Zustand beim Einfrieren nicht verloren. Als kleines Goodie können wir mit `$ONERUN` steuern, ob unser Iterator wieder von vorne beginnen soll, wenn er durchgelaufen ist.

Die ganze Methode ist flexibel genug um beliebig viele `take`s im Body zu realisieren. Weitere `do{}`-Blöcke können überall stehen, wo auch ein `take()` stehen sollte. Wir müssen nur drauf achten, die `ENTER` Label korrekt hochzuzählen.

Erstaunlich, oder? Eine Reihe von Leuten würden schwören, dass Koroutinen in Standard Perl nicht realisierbar sind. Wir wissen es nun besser ... allerdings in der hier gezeigten Form ist es hässlich und schwerlich wartbar!

Metaprogrammierung mit Templates

Was wir hingegen wirklich wollen, ist möglichst nahe an die Eleganz des Perl6 Codes zu kommen, und nur das Wesentliche zu notieren, wie in Listing 2.

```
my $iter= gather {
  for ($i=0; $i<=20; $i++) {
    take $i if ($i % 2);
  }
};

print $iter->()," " for 1..5;
#-> 1,3,5,7,9
```

Listing 2

```
my $DEBUG=1;

sub gather ($) {
  my $body =shift;
  $body =~ s/$TAKE\((.*)\)/do {...}/;
  my $code = qq{
$pre_tmpl
$body
$post_tmpl
};
  eval($code);
  return $c_iter;
};

gather '
   for my $i (0..20) {
       $TAKE($i)
         if ($i % 2);
   };
';
```

Listing 3

Das Standardmittel der **Metaprogrammierung** wäre einen Templatemechanismus zur Codegenerierung zu nutzen. Das hieße also der Funktion `gather` keinen Block, sondern einen String mit `$TAKE` Bezeichnern zu übergeben, der in den obigen Koroutinen-Code umgebastelt und `eval`uiert wird. Also ungefähr so etwas, wie in Listing 3 dargestellt.

Wir schlagen uns so etwas aber lieber aus dem Kopf, weil wir damit nicht nur jegliches Highlighting in unserem Editor verlieren würden, sondern auch alle Syntaxchecks zur Compilezeit. Zusätzlich müssten wir uns immer Gedanken machen, wie wir die Stringdelimiter (hier `'`) im Template verhindern oder escapen.

These 10: *Metaprogrammierung mit Code-Templates und* `eval` *ist zwar für spezialisierte Module geeignet, nicht aber als Spracherweiterung für Enduser.*

Introspektion mit B::Deparse

Was wir eher brauchen ist also eine Möglichkeit, an den Quellcode bereits erfolgreich geparsten Codes zu kommen, bevor er ausgeführt wird. Tatsächlich bietet uns das Modul

`B::Deparse` nicht nur diese Möglichkeit, sondern ist netterweise schon seit 1998 Bestandteil des Cores.

Zur Wirkungsweise: Perl kennt mehrere Compilerphasen, in dem der interpretierbare Code geparst und zu **Opcodes** (Perls "Maschinensprache") kompiliert wird, welche am Ende ausgeführt werden. Diese Opcodes kann man sich nicht nur mit Modulen wie `B::Concise` oder `B::Terse` anschauen...

```
>  perl -MO=Terse -e 'take(1+2)'
LISTOP (0x9ae1e08) leave [1]
    OP (0x9aec090) enter
    COP (0x9ac92f8) nextstate
    UNOP (0x9ad4bc8) entersub [3]
        UNOP (0x9ad18d0) null [142]
            OP (0x9ac9828) pushmark
            SVOP (0x9ad1a68) const [4] ...
            UNOP (0x9ad1890) null [17]
                PADOP (0x9ad1950) gv  ...
```

Listing 4

... man kann sogar aus diese Opcodes wieder einen gleichwertigen (aber nicht unbedingt originalen) Codetext zurückgewinnen, ohne dass er bereits ausgeführt wird.

```
> perl -MO=Deparse -e 'take(1+2)'
take(3);
```

Das kann nicht nur sehr nützlich zum Debuggen kompletter Files sein, `B::Deparse` bietet auch die Methode `coderef2text()` um Codeblöcke im laufenden Programm zu deparsen. Folgende `gather` Funktion kann nun unseren Wunschsyntax aus Listing 2 entschlüsseln, nach belieben verarbeiten, und mittels `eval` wieder kompilieren:

```
sub gather (&) {
  my $c_block=shift;
  use B::Deparse;
  my $deparse = B::Deparse->new();
  $body=$deparse->coderef2text($c_block);
  DBout( $body );
  # ... verarbeiten ..
  return eval "sub $body";
}
```

Die Debugausgabe zeigt uns welcher Code ermittelt wurde:

```
{
    for ($i = 0; $i <= 20; ++$i) {
        take($i) if $i % 2;
    }
}
```

Fein, jetzt Zeilen vorne und hinten dranzuhängen, ist wohl kein Problem mehr ... aber wie ersetze ich das `take()` nun auf eine *saubere* Weise, eine RegEx wie aus Listing 3 ist ja im Prinzip nur eine Art Codefilter. Eine RegEx kann nur mit unverhältnismäßigem Aufwand erkennen, ob unsere vermeintliche Funktion wirklich in validem Code oder innerhalb eines Strings oder eines Kommentars steht. Denn "*Only perl can parse Perl*"!

Macros

LISP ist nicht nur mit die älteste aller höheren Programmiersprachen und zugleich Urmutter vieler Perl-Features, sie ist auch immer noch erstaunlich lebendig. Dies verdankt sie nicht zuletzt ihrer hohen Flexibilität mittels **Macros** den minimalistischen Sprachumfang anpassen zu können. So ein Macro wird im Quelltext genauso wie eine Funktion mit Parametern notiert, ist also von der Syntax nicht zu unterscheiden. Allerdings wird diese besondere Funktion **expandiert**. Sprich sie wird bereits beim Parsen ausgeführt, und der returnierte String wird an der aktuellen Codestelle eingefügt und kompiliert.

Verwirrend? In Perlnotation wäre der Effekt ungefähr so:

```
sub macro {             # Definition
    "print $_[0]"
};

macro($a);              # Anwendung

print $a;               # Expansion
```

(Praktisch macht Perl übrigens so etwas Ähnliches bereits automatisch bei konstanten Funktionen, d.h. sie dürfen keine Parameter erlauben. Dieses *Inlining* hilft die Geschwindigkeit zu steigern.)

Diese Art der Metaprogrammierung funktioniert viel genauer und fehlerfreier als C-Präprozessoren oder Codefilter es könnten, weil es hier keine unabhängigen Parsingphasen gibt, die einander sabotieren könnten.

Unser Problem - und viele andere - wären also trivial lösbar, wenn Perl irgendwie Macros beherrschen würde. Dann könnte `take($i)` einfach zu dem gewünschten `do {...}` Block expandiert werden.

Monkeypatching

Und hier kommt uns gelegen, dass `B::Deparse` selbst komplett in Perl geschrieben wurde. Wenn wir uns im Debugger anschauen, wie der Opcode von Funktionsaufrufen entschlüsselt wird, finden wir die interne Routine `pp_entersub()`. (Analog zum Opcode `entersub` in Listing 4.) Sie liefert beim Parsen der Opcodes jeweils den String eines Sub-Aufrufs inklusive Parameter zurück, also in unserem Fall genau das `"take($i)"`. Und hier können wir uns einklinken, um unseren gewünschten Code einzuschleusen.

Hierzu wäre es auf lange Sicht sicher besser `B::Deparse` umzuschreiben oder, sagen wir zumindest, die Routine `pp_entersub()` um alternativ den String der Expansionen zurückzuliefern. Aber das würde entweder bedeuten den Modul-Maintainer von unserem Bedarf zu überzeugen und darauf zu hoffen, dass der Patch möglichst bald in die Standard-Distribution aufgenommen wird. Oder wir müssten einen eigenen Fork anlegen, modifizieren und einbinden. Letzteres mit der Konsequenz sich von der zukünftigen Entwicklung von `B::Deparse` abzukoppeln. Vom Aufwand ein Modul mit 4300 Zeilen zu modifizieren ganz zu schweigen.

```
my $c_pp_entersub;

#-- Deparse inkl. Expansion
sub expand_block {
  my $c_block=shift;
  use B::Deparse;

  #- Merke Referenz auf Originalfunktion
  $c_pp_entersub=
             \&B::Deparse::pp_entersub;

  #- Lokal mit Wrapper patchen
  no warnings;
  local *B::Deparse::pp_entersub =
                       \&expand_macro;
  use warnings;

  #- Deparse
  return B::Deparse->new()
              ->coderef2text($c_block);
}


#-- Wrapper
sub expand_macro {

  #- Parameter durchreichen zu Original
  my $call_string=$c_pp_entersub->(@_);

  #- extrahiere Subroutinen Aufruf
  $call_string=~/^(\w+)\((.*)\)$/;
  my $sub_name  =$1;
  my $arguments =$2;

  #- Nichtmakros ungeändert durchwinken
  return $call_string  unless
           $IS_MACRO{$sub_name};

  DBout("$call_string =>
          '$sub_name','$arguments'\n");

  #- Makros expandieren
  return eval "$sub_name('$arguments')";
}
```

Listing 5

Tatsächlich ist Perl aber flexibel und dynamisch genug, um schon kurzfristig zu sehr eleganten Erfolgen zu kommen. Wir können nämlich nach dem Einbinden mit `use` auch bei einem "fremden" Modul interne Funktionen überschreiben. Es reicht zur Laufzeit die Einträge ihrer *Symboltabelle* zu manipulieren und so eine geeignetere Routine "einpflanzen". Diese Technik der "Operation am lebenden Körper" wird auch **Monkeypatching** genannt.

Um dabei ein möglichst geringes Konfliktrisiko einzugehen, reicht es eine Wrapperfunktion zu schreiben, die die Makros unter den deparsten Funktionen abfängt. Also nach dem Aufruf wird gleich das originale `pp_entersub()` aufgerufen und alle erhaltenen Parameter reingereicht. Und nur wenn der Rückgabestring eines unserer Makros - hier `take()` - beschreibt, ersetzen wir ihn mit unserem neuen Codesnippet.

Da wir hier die originale Funktion `pp_entersub` nicht anfassen, bleibt alles transparent für das manipulierte Modul und wir haben bei dieser Technik keine Konflikte zu befürchten - weder bei alten noch bei allen künftigen Versionen von `B::Deparse`, solange sich die Schnittstelle - einen String zurückzugeben - nicht ändert.

Jetzt können wir auch unser `take()`-Makro so anlegen, dass es den gewünschten `do{}`-Block als String zurückgibt:

```
my $counter=0;

sub take {
  $counter++;
  return <<"_EXPANSION_";
do {
  \$RESUME="ENTER_$counter";
  return($_[0]);
 ENTER_$counter:
}
_EXPANSION_
}

# Markiere als Makro in globalem Hash
# (Eleganteres führt gerade zu weit)
our %IS_MACRO= (take => 1);
```

Und unser `gather()` braucht nur noch den erhaltenen Codeblock zu expandieren:

```
my ($pre_tmpl, $post_tmpl) =
   (<<'__PRE_TMPL__', <<'__POST_TMPL__');
{
  #-- Steuerung
  my $RESUME;
  my $ONERUN=1;

  #-- Closure vars
  my $i;

  $c_iter = sub {
    #-- Dispatcher
    goto $RESUME
      if defined $RESUME;

__PRE_TMPL__

    #-- Exit
    $RESUME = $ONERUN ? "FINISHED": ();
    FINISHED:
      return;
  }
}
__POST_TMPL__


sub gather (&) {
  my $c_block=shift;
  my $body = expand_block($c_block);

  #- Schönheitskorrekturen
  $body =~ s/^{/    #-- Body/s;
  $body =~ s/}$/    # Body --/s;

  $body = qq{
$pre_tmpl
$body\n
$post_tmpl
};
  DBout($body);

  my $c_iter;
  eval "$body";
  return $c_iter;


}
```

Wenn wir nun den Code mit unserer Wunschsyntax aus Listing 2 ausführen, sehen wir, dass alles klappt! Listing 1 ist übrigens nichts anderes als die Debugausgabe des generierten Codes in `$body`.

Fromme Lügen

Nun, um diesen Artikel einigermaßen übersichtlich zu halten, habe ich bisher noch ein paar Sachen unterschlagen! (Jetzt aber bitte nicht ärgern! Dafür gibt es den Weihnachtsmann wirklich...)

1. Die Closurevariablen

In Listing 1 hatten wir einfach frech eine statische Closurevariable `my $i` angenommen. Woher soll aber der Code in `$pre_tmpl` denn dynamisch wissen, welche Variablen im Body eingefroren werden sollen?

Nun seit `5.9` kennt Perl das Feature `state` als Alternative zu `my`. Wir können damit statische Variablen bereits im Body implizit deklarieren, ohne diese im äußeren Scope nochmal spiegeln zu müssen:

```
my $iter= gather {
  for (state $i=0; $i<=20; $i++) {
    take $i if ($i % 2);
  }
};
```

Nun ist `5.10` relativ neu, müssen wir also den Anspruch möglichst Rückwärtskompatibel zu bleiben jetzt beerdigen? Mitnichten, denn wenn wir in einer alten Version trotzdem `state` verwenden, sieht der Lexer kein Builtin sondern einen Funktionsaufruf, d.h. der Opcode `entersub` wird generiert und wir können unseren Makromechanismus wieder nutzen. Dieses Makro `state()` braucht aber nur zum Leerstring zu expandieren. Wir nutzen es um die Parametervariablen zu erfassen, um sie hinter `my` im `$pre_tmpl` einzufügen. (Fleißaufgabe für den geneigten Leser!)

2. take() kann nicht in beliebigem Code stehen

Um die Sprunglabels anspringen zu können, müssen sie im gleichen lokalen Scope stehen wie das `goto`. Das scheitert aber manchmal, insbesondere wenn man in einen `sub`-Block reinzuspringen versucht. Geschenkt, aber leider kann man auch nicht in `foreach`-Schleifen reinspringen:

```
> perl -e '
    goto LABEL;
    for $i (1..20) {LABEL: print }
'
Can't "goto" into the middle of a foreach
loop at -e line 1.
```

Und jetzt wird vielleicht dem Einen oder Anderen klar, warum wir bisher in unseren Beispielen immer C-Style-Fors (also `for ($i=0; $i<=20; $i++)`) genutzt haben.

Um dieses Problem zu umgehen, könnten wir jetzt in `B::Deparse` auch die Funktion patchen, die `foreach` deparst und zu einem C-Style-For umschreiben. Das ist zwar machbar, aber bei weitem nicht so trivial, wie das bisher Gezeigte.

3. Fehlerhandling

Ein anderer grundsätzlicher Ansatz wäre es, nun den Usern `foreach`es in `gather`-Blöcken zu verbieten, was zwar lästig aber nicht tragisch wäre, weil es ja wirklich nicht an alternativen Konstrukten mangelt.

Grundsätzlich sollten wir aber dann Verstöße auch unmittelbar während der Compilierung bereits als Fehler melden! Es wäre nämlich nicht sehr schön, wenn der User nach längerer Laufzeit eine Fehlermeldung bekäme, dass irgendwo in seinem Code ein `goto` gescheitert ist, insbesondere wenn er in seinem ganzen Code kein `goto` sieht.

These 11: *Hat eine Erweiterung Einschränkungen, dann sollte eine Fehlnutzung möglichst fehlertolerant abgefangen werden und/oder dem User mitgeteilt werden.*

In unserem speziellen Fall, ist diese Überprüfung aber gar nicht so schwer, weil wir den Macromechanismus selbst dazu nutzen können, um die Integrität unseres Codes zu überprüfen:

Wir können `take()` nämlich in einem Testlauf auch so expandieren lassen, dass auf alle Sprungziele `ENTER_#` direkt ein `return` folgt.

```
sub take {
  $counter++;
  return <<"_EXPANSION_";
do {
 ENTER_$counter:
  return($_[0]);
}
_EXPANSION_
}
```

So können wir in diesem Testcode alle Labels gefahrlos anzuspringen versuchen und auf eventuelle Fehler reagieren. Gibt es keine Fehlermeldung, dann sind auch alle Labels in der eigentlichen Expansion erreichbar. Gibt es im Testlauf doch Fehler, dann fangen wir diese ab und wandeln sie in eine verständlichere Meldung um z.B.:

"ERROR: Unzulässiges take() in Zeile x. Vielleicht innerhalb eines neuen Subs oder einer Foreach Schleife?"

Ausblick

Die Anwendungsmöglichkeiten von Macros sind viel umfangreicher und selten erfordern sie `goto`s. In der nächsten Folge werden wir nun nicht nur versuchen die List-Comprehensions aus der letzten Folge zu verbessern. Wir werden einen Ausblick wagen, welche sonstigen neuen Perl6-Features sich nun eventuell bereits in Perl5 umsetzen ließen.

Matthias Weckbecker

CPAN-Distributionen im openSUSE Build Service paketieren

Erfahren Sie in folgendem Artikel wie unkompliziert und bequem es für Sie als Software-Entwickler ist, Ihre Programme und Bibliotheken als binäre Pakete für Endbenutzer bereitzustellen mit Hilfe des neuen openSUSE Build Services.

Der openSUSE Build Service

Der openSUSE Build Service (kurz: OBS) ist ein weitgehend betriebssystemunabhängiges Werkzeug für Entwickler zum Erstellen und Verteilen von Programmen und Bibliotheken als Pakete. Diese Pakete können als binäres RPM- oder DEB-Format erzeugt und für unterschiedliche Architekturen gebaut werden, wie i586 und x86_64. Neben openSUSE unterstützt der Build Service weitere gängige Linux-Distributionen wie Fedora, Red Hat, CentOS, Debian und Ubuntu.

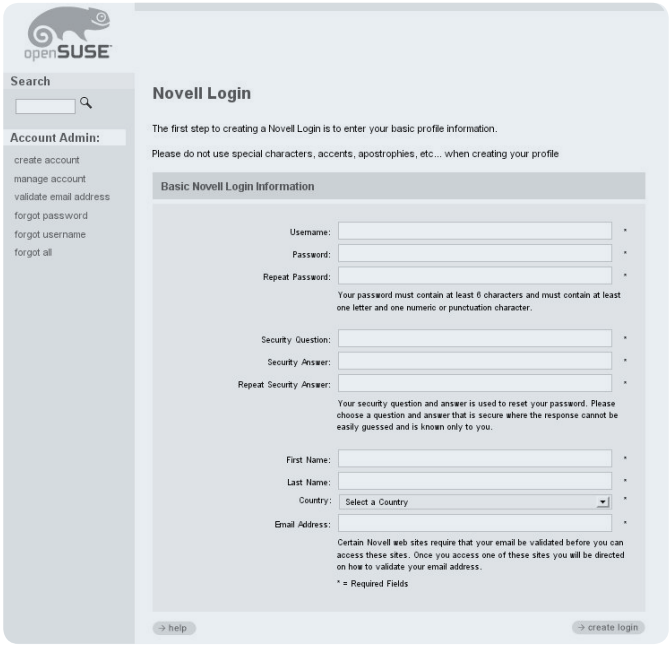


Abbildung 1: Registrierungsformular für den Build Service

Registrierung im Build Service

Um den Build Service verwenden zu können, ist eine Registrierung erforderlich. Diese kann auf *https://secure-www.novell.com/selfreg/jsp/createOpenSuseAccount.jsp* schnell und unkompliziert erledigt werden (siehe Abbildung 1). Nachdem die Registrierung abgeschlossen wurde, müssen wir uns auf *http://build.opensuse.org/* nur noch einloggen, auf *"Let me build my packages"* klicken und ein Heimatprojekt anlegen.

Installation und Konfiguration des Kommandozeilen-Clients

Zum bequemen Arbeiten auf dem Build Service empfiehlt es sich, den Kommandozeilen-Client `osc` zu installieren. Dieser ist mit

```
$ sudo zypper install osc # openSUSE und SLE
```

bzw.

```
$ sudo apt-get install osc # Ubuntu
```

zügig installiert. Ebenso zügig und schnell funktioniert auch dessen Konfiguration. Es genügt ein kurzes

```
$ osc
```

und der Benutzer wird nach den im vorherigen Schritt angelegten Zugangsdaten des OBS-Accounts gefragt. Nach Eingabe dieser Daten wird die Konfiguration in `$HOME/.oscrc` abgespeichert und kann ggf. noch erweitert und angepasst werden. In der Datei befindet sich Dokumentation in Form von Kommentaren zu den jeweiligen Konfigurationsdirektiven.

Das entfernte Heimatprojekt auschecken

Nachdem `osc` installiert wurde, muss das Heimatprojekt im Build Service - genau wie bei Subversion und GIT - noch ausgecheckt werden. Mit:

```
$ osc checkout home:$benutzername
```

landet das (noch) leere Projekt im aktuellen Verzeichnis.

Eine CPAN-Distribution in das Heimatprojekt des OBS ablegen

In das noch leere Verzeichnis auf dem openSUSE Build Service wollen wir eine Bibliothek aus dem CPAN hinzufügen. Als Beispiel wurde Acme::Drunk ausgewählt. Zunächst wechseln wir mit `cd` in unsere ausgecheckte, lokale Kopie und erzeugen innerhalb dieser ein neues Unterverzeichnis mit dem Namen der Bibliothek, die wir paketieren wollen. Die Namenskonvention lautet unter openSUSE und Fedora folgendermaßen: perl-$modulname, also z. B. perl-Acme-Drunk, oder perl-Acme.

```
$ cd home:$benutzername/
$ osc mkpac perl-Acme-Drunk
$ cd perl-Acme-Drunk/
$ wget http://search.cpan.org/CPAN/authors/
       id/C/CW/CWEST/Acme-Drunk-0.03.tar.gz
```

Bauanleitung und CHANGELOG für perl-Acme-Drunk anlegen

Wir befinden uns in perl-Acme-Drunk innerhalb unseres lokal ausgecheckten Heimatprojekts. Als nächstes muss die Bauanleitung angelegt werden. Anhand dieser wird Acme::Drunk gebaut. Zusätzlich benötigen wir eine CHANGELOG-Datei:

```
# Bauanleitung für RPM-Format
$ touch perl-Acme-Drunk.spec
$ osc vc   # CHANGELOG anlegen
```

Unser perl-Acme-Bleach Verzeichnis sieht nun folgendermaßen aus:

```
. .. Acme-Drunk-0.03.tar.gz
 perl-Acme-Drunk.changes perl-Acme-Drunk.spec
```

Wir können nun mit

```
$ (cd ..; osc add perl-Acme-Drunk/)
```

kurzerhand alle Dateien für den nächsten Commit in unser entferntes Heimatprojekts auf dem Build Service markieren. Mit `osc commit` könnten wir auch schon jetzt unsere Änderungen auf dem Server ablegen. Allerdings sollten wir vorher noch die Bauanleitung schreiben.

Die spec-Bauanleitung für perl-Acme-Drunk schreiben

Die Präambel

Unsere Bauanleitung perl-Acme-Drunk.spec startet mit einer Präambel - siehe Listing 1.

Sie enthält verschiedene Attribute wie z. B. die Gruppe, Version- und Releasenummern, Abhängigkeiten, Quelldateien, Beschreibungen, Autoren, usw. Eine vollständige Aufzählung aller RPM-Gruppen ist übrigens im openSUSE-Wiki unter der folgenden URL zu finden: *http://de.opensuse.org/Paketbau/SUSE-Paketkonventionen/RPM-Gruppen*

Der %prep Abschnitt

Der %prep Abschnitt dient zur Vorbereitung des Builds (preparation). Hier werden z. B. Korrekturen mit `%patch` angewandt und, was eigentlich noch viel wichtiger ist, das Quellpaket mit `%setup` ausgepackt und automatisch mit `cd` in das ausgepackte Verzeichnis innerhalb der Buildumgebung gewechselt. Der Abschnitt sieht üblicherweise folgendermaßen aus:

```
%prep
%setup -n Acme-Drunk-%{version}
```

Der %build Abschnitt

Innerhalb des %build Abschnitts wird üblicherweise das Makefile generiert und `make` aufgerufen. Nicht anders sieht es in der spec-Datei aus für Acme::Drunk:

```
%build
perl Makefile.PL
make
make test
```

Der %install Abschnitt

Normalerweise würde jetzt ein `make install` ausgeführt werden. Dies wird uns aber durch ein RPM-Makro schon au-

```
#norootforbuild
Name: perl-Acme-Drunk
Group: Development/Libraries/Perl
Version: 0.03
Release: 1
Autoreqprov: on
Requires: perl = %{perl_version}
Requires: perl-Acme-Drunk
Summary: Acme::Drunk - Get Drunk, Acme Style
License: Artistic License
URL:  http://cpan.org/modules/by-module/Acme/
Source: Acme-Drunk-%{version}.tar.gz
BuildRoot: %{_tmppath}/%{name}-%{version}-build>

%description
Calculating an accurate Blood Alcohol Concentration isn't as easy as it
sounds. Acme::Drunk helps elevate the burden placed on the Average Joe, or
Jane, to know if they've had too much to drink.

Authors:
--------
Casey West <casey@geeknest.org>
```

Listing 1

tomatisch abgenommen. Wir müssen also lediglich folgende zwei Aufrufe innerhalb des %install Abschnitts ausführen:

```
%install
%perl_make_install
%perl_process_packlist
```

Für den interessierten Leser, der wissen möchte zu was die obigen RPM-Makros exakt expandieren, sei die Datei "suse_macros" unter /usr/lib/rpm ans Herz gelegt.

Der %clean Abschnitt

Nachdem die Arbeit erledigt ist, wird ein obligatorisches `rm -rf $RPM_BUILD_ROOT` innerhalb des %clean Abschnitts ausgeführt:

```
%clean
rm -rf $RPM_BUILD_ROOT
```

Der %files Abschnitt

Die spec-Datei ist jetzt fast vollständig. Das einzige, was noch fehlt, ist der %files Abschnitt. Hier werden alle Dateien, die während %install in der Buildumgebung installiert wurden, aufgelistet und in das RPM-Paket gespeichert. Folgendermaßen sieht der Abschnitt aus:

```
%files
%defattr(-, root, root)
%doc Changes README MANIFEST
%doc /usr/share/man/man3/Acme::Drunk.3pm.gz
%{perl_vendorlib}/Acme/Drunk*
%{perl_vendorarch}/auto/Acme/Drunk
/var/adm/perl-modules/perl-Acme-Drunk
```

Unsere Arbeit in den OBS abgeben

Sobald die spec-Datei fertig ist, muss alles in den OBS abgegeben werden. Dies geschieht wie folgt:

```
$ osc vc    # evtl. CHANGELOG erneuern
$ osc commit -m "initial commit" # abgeben
```

Build-Targets festlegen

Bevor wir perl-Acme-Drunk jetzt auf dem Build Service bauen können, müssen wir natürlich noch festlegen, für welche Distribution und Architektur wir das Paket überhaupt bauen möchten. Mit

```
$ osc meta -e prj
```

können wir die Metadaten des Projekts und somit die Build-Targets editieren. Die Konfiguration des Projekts befindet sich im XML-Format. Die folgende Konfiguration baut das Paket perl-Acme-Drunk für openSUSE und Fedora auf i586 und x86_64 - siehe Listing 2.

Nachdem die Build-Targets mit einem optionalen Titel und eine Beschreibung für das Heimatprojekt festgelegt wurden, kann die Datei gespeichert werden. `osc` übernimmt anschließend alle Änderungen auf den Build Service Servern.

```
<project name="home:NAMEDESHEIMATPROJEKTS">
<title>Home Project</title>
<description/>
 <repository name="Fedora_14">
 <path project="Fedora:14" repository="standard"/>
 <arch>i586</arch>
 <arch>x86_64</arch>
 </repository>
 <repository name="openSUSE_11.3">
 <path project="openSUSE:11.3" repository="standard"/>
 <arch>i586</arch>
 <arch>x86_64</arch>
 </repository>
 <repository name="openSUSE_Factory">
 <path project="openSUSE:Factory" repository="snapshot"/>
 <arch>i586</arch>
 <arch>x86_64</arch>
 </repository>
</project>
```

Listing 2

Den Build-Prozess starten und verfolgen

Vermutlich hat der Build Service mittlerweile bereits begonnen, die RPM-Pakete für uns zu erstellen. Wenn nicht, sollten wir sicherstellen, dass auch wirklich alle Dateien im entfernten Heimatprojekt vorhanden sind (`osc commit`). Nachdem wir uns vergewissert haben, können wir den Build-Prozess auch manuell noch starten mit:

```
$ osc rebuild
```

Es ist möglich, auch während oder nach dem Build-Prozess jederzeit die Log-Datei zu verfolgen mit

```
$ osc buildlog openSUSE_Factory i586
$ osc buildlog openSUSE_11.3 i586
$ osc buildlog Fedora_14 i586
```

und ggf. Fehler und Warnungen anschließend beheben.

Software verteilen

Das Repository

Wenn perl-Acme-Drunk fertig gebaut ist (siehe `osc r`), wird es automatisch verteilt. Hierzu exportiert der Build Service das gebaute RPM-Paket automatisch in ein Repository, welches sich bequem in das Paket-Management-System einbinden lässt. Zum Beispiel mit zypper auf openSUSE:

```
$ zypper ar http://download.opensuse.org/
 repositories/home:/NAME/openSUSE_11.3/ repo1
```

Nachdem das Repository eingebunden ist, kann perl-Acme-Drunk ganz einfach mit

```
$ zypper install perl-Acme-Drunk
```

installiert werden. Der Link zu obigem Repository eignet sich hervorragend, um auf der Projektseite der jeweiligen Software veröffentlicht zu werden.

Pakete in andere Projekte abgeben

Es ist ebenfalls möglich, seine paketierten Programme und Bibliotheken (z. B. unser perl-Acme-Drunk) in andere Projekte auf dem Build Service abzugeben. Perl-Pakete können in das dafür vorgesehene devel:languages:perl Projekt - einem Projekt für Perl-Entwickler - abgegeben werden. Mit folgendem Befehl erzeugen wir eine Anfrage an einen der zuständigen Betreuer des Projekts:

```
$ osc submitreq home:$benutzername
     perl-Acme-Drunk devel:languages:perl
```

Nachdem das Paket durch einen Betreuer überprüft und akzeptiert wurde, ist es ebenfalls in devel:languages:perl verfügbar. Somit können Benutzer dieses Projekts bzw. Repositories von dem Paket auch profitieren. Im devel:languages:perl Projekt sind eine Vielzahl interessanter CPAN Pakete bereits vorhanden.

Bauanleitungen automatisiert erzeugen lassen

Wer die Bauanleitung für das Paket nicht selber entwickeln möchte, kann auch auf eine automatische Generierung dieser durch das cpan2dist-Tool zurückgreifen. cpan2dist wird mit Perl mitgeliefert. Durch die Installation von weiteren Plug-ins ist es möglich, beispielsweise spec-Dateien oder rules-Dateien (Bauanleitung unter Debian) automatisch generieren zu lassen - siehe Listing 3.

Die generierte Bauanleitung kann dann im Anschluss einfach in beispielsweise das eigene Heimatprojekt abgegeben werden.

Ausblick

Der Build Service ist ein quelloffenes Projekt, welches stetig verbessert und weiterentwickelt wird. So wurde beispielsweise kürzlich die Möglichkeit implementiert, so genannte _service-Dateien verwenden zu können. Diese Dateien erlauben es, Source-Tarballs automatisch von einem bestimmten Server zu beziehen und anhand einer spec-Datei zu bauen. Dies bringt einige Vorteile mit sich: Zum einen ist es nicht mehr notwendig, dass die Quelldateien von vornherein im Projekt rumliegen und zum anderen ist es möglich, immer die aktuelle Version zu beziehen. Die _service-Datei für Acme::Drunk würde so aussehen, wie in Listing 4 dargestellt.

Der Quellcode vom Build Service kann übrigens mit GIT folgendermaßen ausgecheckt werden:

```
$ git clone git://gitorious.org/opensuse/
    build-service
```

Weitere Informationen zum openSUSE Build Service

Weitere Informationen zum openSUSE Build Service sind im openSUSE-Wiki unter anderem zu finden unter folgenden Adressen:

1. *http://de.opensuse.org/Build_Service*
2. *http://de.opensuse.org/Build_Service/Tipps_und_Tricks*
3. *http://en.opensuse.org/openSUSE:Build_Service_Tutorial*

```
$ cpan2dist --format CPANPLUS::Dist::SUSE Acme::Drunk
$ cpan2dist --format CPANPLUS::Dist::Deb Acme::Drunk
$ cpan2dist --format CPANPLUS::Dist::Fedora Acme::Drunk
```

Listing 3

```
<services>
<service name="download_url">
 <param name="protocol">http</param>
 <param name="host">search.cpan.org</param>
 <param name="path">/CPAN/authors/id/C/CW/CWEST/Acme-Drunk-0.03.tar.gz</param>
 </service>
</services>
```

Listing 4

Steffen Schwigon

Tapper Testinfrastruktur

Abstract

Tapper ist eine modulare Testinfrastruktur. Sie ermöglicht das vollautomatische Betreiben eines Maschinenpools, das Schedulen verschiedenster Workloads mit definierbaren Prioritäten, das Entgegennehmen und Sammeln von Testergebnissen in einer Datenbank und das Evaluieren dieser Daten.

Das wichtigste Feature von Tapper ist die komplette Behandlung des gesamten QA-Lebenszyklus, von der Zeitplanung als Teil eines Projektplanes über das Scheduling von Maschinen und die Ausführung von Testworkloads bis zur Auswertung der Ergebnisse - und gleichzeitig das Optionale all dieser Schritte.

In dieser Gesamtheit ist Tapper recht umfangreich. Man kann sich jedoch eigene, kleinere Infrastrukturen aus Einzelteilen zusammensetzen; z.B. nur eine Testdatenbank mit Webfrontent oder nur die Automatisierung für andere Zwecke.

Ich werde in diesem Artikel die wichtigsten Eigenschaften aufzählen, einige Einsatzszenarien skizzieren und ein paar Highlights zeigen.

Das Ziel ist es, mit diesem aus verschiedenen Detailstufen gemischten Überblick zu helfen, Tapper so weit zu verstehen, dass man weiß wann man es braucht. Danach kommt man mit der StepByStep-Anleitung und dem Handbuch weiter.

Ziele

• Das Hauptziel von Tapper war das Testen von Betriebssystemen, mit und ohne Virtualisierung. Um das wichtigste Missverständnis gleich zu klären: Tapper **verwendet nicht** einfach Virtualisierung, sondern es **testet** die Virtualisierung, konkret Xen und KVM. Die untersuchten Probleme passieren dabei sehr viel früher, noch bevor "normale" Software überhaupt startet. Andere, meist leichtere Anwendungsfälle, sind ebenso möglich, siehe Kapitel `Einsatz-Szenarien` am Ende.

• Die Ausführung von Test-Workloads erfolgt vollautomatisch. Sie kann aber auch manuell erfolgen, da "bleeding-edge" Hardware oft nicht gleich automatisierbar ist. Alle Testergebnisse landen in einer gemeinsamen, zentralen Ergebnisreport-Datenbank.

• Testergebnis-Reports werden in einem äußerst einfachen Testprotokoll formatiert. So wird für das Erzeugen solcher Reports auch unter extremen Low-level-Bedingungen, wie eben beim Testen von Betriebsystemen keine besondere Software vorausgesetzt. Es geht mit jeder Sprache in jeder primitiven Umgebung, in der sinngemäß ein `echo` funktioniert. Von Anfang an war das standardisierte "Test Anything Protocol" ("TAP") die übergreifende Idee, die die verschiedenen Abstraktionsschichten miteinander verbindet und die Integration verschiedenster Fremdkomponenten ermöglicht.

• Testreports können Daten enthalten. Damit können im gleichen Testprotokoll z.B. Benchmark-Ergebnisse transportiert werden. TAP bietet hierfür eingebettetes YAML.

• Das Erstellen, Senden und Evaluieren von Testergebnissen soll keine lokalen Toolchains benötigen. Alle APIs können mit einfachen `echo`- und `netcat`-Sequenzen bedient werden, selbst Upload und Download von Attachments oder komplexe Ergebnisabfragen.

Für die Philosophie der Einfachheit der Formate, Protokolle und APIs, die Möglichkeit, das System ohne client-seitige

Toolchains zu bedienen oder die Automatisierung zu ignorieren, haben wir den Begriff "non-aristocratic participation model" ("nichtaristokratisches Teilnahmemodell") verwendet.

Das bedeutet, dass die Einstiegshürde nur aus einem minimalen Grundverständnis des Testprotokolls "TAP", wenigen Meta-Infos und einem Tool wie "netcat" besteht. Jeder kann die Werkzeuge seiner Wahl verwenden oder selber herstellen, ohne auf eine elitäre Gruppe angewiesen zu sein.

Trotzdem skaliert das System auch für User bis zur vollen Komplexität, die man nach und nach erlernen kann.

Wichtigste Eigenschaften

Tapper löst komplexe Anforderungen. Die lassen sich nicht alle im Detail beschreiben, daher hier eine Vorschau zu
- Vokabular
- Automatisierung
- Scheduling
- Web-Frontend
- Ergebnis-Evaluierung
- Testplan-Support
- Support zum Schreiben von Tests
- Use-Cases und
- Technologien

Vokabular

Tapper läuft üblicherweise auf einem zentralen Steuerrechner, der die Automatisierung übernimmt, die Datenbanken hält, das Web-Frontend und alle APIs bedient.

Das zentrale Automatisierungsprogramm ist das sogenannte **Master Control Programm** (**MCP**).

Tapper installiert auf Testmaschinen zur Testprogrammausführung und -überwachung ein sogenanntes **Program Run Control** (**PRC**).

Die gesamte Automatisierung verwendet zur Spezifikation der zu installierenden Maschine kleine YAML-Snippets, genannt **preconditions**, die verschachtelt zu größeren abstrakten Setups (für Virtualisierung) kombiniert werden können. Sie können auch in einer Form als sogenannte **macro preconditions** verwendet werden, dann werden sie zusätzlich durch Template-Toolkit geschickt.

Ein Testergebnis, formatiert in TAP mit Metainfos, ist ein **Report** und ein automatisierter Testlauf ist ein **Testrun**.

Reports werden im **Test Anything Protocol** (**TAP**) formatiert. Es können auch **TAP-Archive** verwendet werden (wie sie `prove -a` erzeugt), was mehrere TAP-Reports in einer .tar.gz-Datei bündelt.

Es gibt zwei Datenbanken: eine **TestrunDB** für Verwaltung und Scheduling von automatischen Testläufen ("Testruns") und eine **ReportsDB**, die Testergebnisse ("Reports") speichert, egal ob aus automatisierten oder manuellen Testläufen.

Ein **Web-Frontend** präsentiert im wesentlichen Reports, kann aber (mit etwas Vorbereitung) auch automatisch auszuführende Testruns eintragen.

Automatisierung

Hardware-Reset und Netzwerk-Booten

Tapper kann komplette Testmaschinen von Grund auf installieren.

Das Bootstrappen einer Testmaschine zu ermöglichen, mittels Hardware-Reset, Netzwerk-Boot (über PXE, TFTP und NFS) und Maschinen-Selbst-Installation anhand abstrakter Spezifikationsfiles, ist der einzige komplizierte Teil beim Aufsetzen einer vollautomatisch laufendenden Infrastruktur. Alles andere danach ist leicht.

Diese Automatisierung ist aber optional, nicht überall müssen Maschinen von Grund auf installiert werden.

Tapper startet den gesamten Prozess mit einem Reboot. Entweder läuft die Maschine noch und es geht mit `SSH` oder Tapper macht ein Hardware-Reset, im einfachsten Fall z.B. mit einer über HTTP schaltbaren Steckdose (siehe Plugin `Tapper::MCP::Net::Reset::PM211MIP`).

Maschinen müssen entsprechend einer grub-Config im Netzwerk booten; diese Config wird vom zentralen "Master Control Program" ("MCP") geschrieben, je nach notwendiger Phase: Booten des Installers oder Booten des installierten Systems.

Während der Installation einer Testmaschine werden kurze Textsnippets mit Statusrückmeldungen an den zentralen Tapper-Host gesendet, damit Tapper mit Timeouts den Fortschritt oder das Fehlschlagen kontrollieren kann. Eigene Maschinen-Installer können integriert werden, wenn sie nur diese Statusmeldungen absetzen können, wozu ein `echo` und `netcat` reicht.

Die Tapper Dokumentation (`Tapper::Doc`) beschreibt ein Beispiel-Setup, um das Booten von Testmaschinen mit DHCP so zu konfigurieren, dass sie über Netzwerk einen Kernel von einem TFTP-Server mit PXE booten und ein kleines Installer-Linux von NFS starten. Ebenso wird beschrieben, wie man ein solches Installer-NFS-Root-Image aufsetzt.

Mit diesem über Netzwerk gestarteten Linux kann sich die Testmaschine praktisch selbst installieren. Die Installation wird vom `Tapper::Installer` gesteuert.

Installation Image- oder Kickstart/Autoyast-basiert
Üblicherweise wird ein Basis-OS-Image auf die Festplatte kopiert und in dieses werden die restlichen Pakete installiert, bis hin zu Images für virtualisierte Gäste und wiederum die Installation von Paketen in diese hinein.

Wird die Installation nicht mit Images gemacht, sondern mit externen Installern wie Kickstart oder AutoYast, müssen in deren Installscripte nur besagte kurze Rückmeldungen eingebaut werden ("start-install", "boot-start", etc.). Ist darin kein `netcat` verfügbar, geht auch ein Python- oder Perl-Snippet mit einem Socket-open/write/close-Einzeiler.

Ausführung von Testscripts im Virtualisierungs-Host und Gästen
Nochmal betont: Tapper **verwendet nicht** einfach nur Virtualisierung, es **testet** Virtualisierung. Entsprechend aufwendig ist das Installieren einer solchen Testmaschine, mit Testprogrammen im Hostsystem und in das Gästen.

Bei der Installation wird der sogenannte `Tapper::PRC` mit installiert und in die Startup-Config eingetragen. Da der zentrale Host den Status kennt, schreibt er die Netzwerk-Boot-Config um, damit die Maschine nun von der installierten Festplatte bootet.

Der `PRC` hat eine Config installiert bekommen, in der steht, welche der installierten Skripte er ausführen soll. Damit werden die Workloads gestartet. Die virtualisierten Gäste machen praktisch das gleiche mit eigenen `PRC`s in sich.

Der `PRC` übernimmt lokale Timeoutprüfungen der Testprogramme und sendet Statusmeldungen an den zentralen Server, der seinerseits auch nochmal ein globales Timeout von außen überwacht.

Report-Gruppierung (und Meta-Infos im allgemeinen)
Alle Workloads senden ihre Ergebnisse selbst und unabhängig voneinander. Das ist notwendig z.B. in virtualisierten Gästen, die streng voneinander getrennt sind. Es ist aber auch nützlich, um generell unabhängig Einzelteile zum Gesamtbild beizutragen, z.B. verschiedene Testprogramme der selben Umgebung.

Der inhaltliche Zusammenhang verschiedenster Einzeltestreports erfolgt über Meta-Informationen, die die Tests mitsenden: eine "report group ID", für die man z.B. die ID des Testlaufes ("Testrun") verwenden kann, welche durch den `PRC` als Environment-Variable `$TAPPER_TESTRUN` an die Testskripte verfügbar gemacht wird.

Im Testreport wird die Gruppierungsinfo als ein Meta-Info-Header in einer TAP-Kommentarzeile dieser Art eingebettet:

```
# Tapper-reportgroup-testrun:  12345
```

In der Reports-Datenbank werden die verschiedenen Reports anhand der gemeinsamen Group-ID wieder zusammengeführt und entsprechend z.B. im Webfrontend dargestellt.

Ist die Gruppierung nicht ein gemeinsamer Testrun, sondern "irgendwas", wird ein anderer Header verwendet:

```
# Tapper-reportgroup-arbitrary: 7a123a64dids2
```

Den Gruppenidentifier kann man sich hierfür irgendwie konstruieren, z.B. eine md5-Hash oder ein konstanter String für alle Tests am gleichen Tag, oder etwas sprechendes wie "mein-zomtec-experiment".

Abstrakte Spezifikationsfiles
Der Setup-Prozess wird spezifiziert durch kleine Config-Snippets, aus sogenannten "preconditions": ein Snippet z.B. pro zu installierendem Image, pro zu installierendem Paket und pro zu startendem Testprogramm.

Das sieht im einzelnen z.B. so aus:

```
precondition_type: image
arch: linux64
image: suse/
    sles11_sp1_x86-64_baseimage.tar.gz
mount: /
partition:
  - testing
  - sda2
  - hda2
```

Hier soll ein Image soll kopiert werden, ein SLES11.tgz, welches das Root wird. Die Zielpartition findet der Installer entweder anhand eines Labels "testing", oder einer Bezeichnung "sda2" oder "hda2". Die Architektur "linux64" ist für Installerdetails wichtig, wenn z.B. ein `chroot` in die Images erfolgt.

Anderes Beispiel:

```
precondition_type: copyfile
protocol: local
name: /data/tapper/autoreport/zomtec-test.sh
dest: /
```

Das besagt, es soll ein File `zomtec-test.sh` nach `/` kopiert werden. Die Kopiermethode ist ein lokales Kopieren (das Quell-Verzeichnis ist über NFS im Installer schon gemountet).

Und da das ein Testprogramm war, haben wir auch eine passende "precondition" für das Ausführen dieser:

```
precondition_type: testprogram
program: /zomtec-test.sh
timeout: 300 # 5 minutes
```

All diese kleinen Snippets tauchen üblicherweise in einem Komplex-Configfile auf, wo sie Teil eines großen YAML-Files sind. Etwa so wie in Listing 1 dargestellt.

Im virtualisierten Fall wird es noch etwas komplexer, da gibt es spezielle Virtualisierungs-Preconditions mit hierarchischen Untereinträgen für die Gäste, aber mit den gleichen Strukturen.

Das Schreiben solcher "preconditions" kann mühselig sein. Alternativ könnte man auch sein Basisimage (oben das SLES11.tgz) mit allen Dependencies vorbereiten. Dann ist es aber nicht so flexibel, wenn man beispielsweise die "preconditions" mit einem Programm erzeugt, welches z.B. durch eine Liste aller verfügbaren Test-Workloads rotiert und die entsprechenden `copyfile` und `testprogram`-Snippets erzeugt.

Und genau dafür werden solche Spec-Files auch noch durch Template-Toolkit ausgewertet, man kann darin z.B. Variablen einbetten. Obige Zeile für das zu verwendende Basisimage könnte z.B. so lauten:

```
image: [% baseimage %]
```

Daher ist der Begriff für das alles eine sogenannte "macro precondition".

Eine solche kann man nun an ein Kommandozeilentool übergeben und dabei Template-Variablen setzen:

```
tapper-testrun new \
        --macroprecond /tmp/zomtec.mpc \
        -Dbaseimage=rhel/rhel5.6_beta2.tar.gz
```

Den Rest übernimmt Tapper: einen "Testrun" eintragen, auf eine passende Maschine schedulen, diese Maschine aufsetzen, die Tests starten und die Ergebnisse aufheben.

Precondition-Producer

Das Konzept der "preconditions" ist sehr zentral und daher in vielen Bereichen weiter verfeinert. Will man z.B. komplette "preconditions" dynamisch erst zum Zeitpunkt des Schedulings konstruieren, kann man einen sogenannten "precondition producer" definieren. Die "precondition" wird dann einfach durch ein Perl-Plugin erstellt.

Dieses Plugin könnte z.B. aus einer abzuarbeitenden Matrix von komplett verschieden strukturierten Tests den nächsten auswählen. Wenn das Plugin "Testmatrix" heißt, dann ist die "precondition" einfach:

```
precondition_type: produce
producer: Testmatrix
```

```
---
precondition_type: image
arch: linux64
image: suse/
   sles11_sp1_x86-64_baseimage.tar.gz
mount: /
partition:
  - testing
  - sda2
  - hda2
---
precondition_type: copyfile
protocol: local
name: /data/tapper/autoreport/zomtec-test.sh
dest: /
---
precondition_type: testprogram
program: /zomtec-test.sh
timeout: 300 # 5 minutes
```

Listing 1

Im Originaleinsatz von Tapper werden mit diesem Mechanismus sämtliche aktuellen Distributionen von Novell und Red Hat mit verschiedenen KVM-und Xen-Gastkombinationen (wiederum aus Novell, RedHat und sogar Windows) und verschiedener Konfiguration (32bit/64bit, PAE, etc.) konstruiert. Die dazu notwendigen, ebenfalls dynamisch erzeugten Xen/KVM-Configs bekommen temporäre Dateinamen, die dann in den "preconditions" z.B. in "copyfile"-Aktionen verwendet werden.

Klingt komplex, ist im Kern trotzdem wieder das gleiche: erneut werden einfach nur "precondition"-Snippets erzeugt, nur eben diesmal mit "Dritt-Programmen".

Gegenseitig abhängige Maschinen
Für Tests, an denen mehrere Maschinen beteiligt sind, z.B. Netzwerkbenchmarks, gibt es Szenario-Specfiles. Der Scheduler und Installer kümmern sich dann um das synchronisierte Aufsetzen der Maschinen und Starten der Workloads.

Das Grundprinzip ist immer wieder ähnlich: wieder werden die gleichen Einzelbausteine zu einer großen Config kombiniert.

Komplexes Timeout Handling
Das Timeout-Handling umfasst den kompletten Testzyklus, in mehreren Einzelphasen: die Installation, das Booten, die Einzeltestscripte, die Gesamtzeit; inklusive Virtualisierungs-Szenarios mit entsprechenden Gast-Timeouts.

Reboot handling
Neben dem initialen Booten zu Installationszwecken kann auch ein mehrfaches Booten als Teil des Testszenarios durchgeführt werden.

Console logging
Das ist vor allem im Zusammenhang mit Betriebssystemtests nützlich. Tapper verwendet momentan die `conserver`-Infrastruktur, um einen Host ab Reset-Zeitpunkt zu beobachten und lädt den gesamten Mitschnitt als Attachment zum Test hoch.

Scheduling

Der Scheduler ist zwar Teil der Automatisierungsschicht, aber darin ein eigener komplexer Teil, der auch als eigenständiger Scheduler verfügbar sein könnte. Hier nur die wesentlichsten Eigenschaften:

Optimale Ausnutzung von "zu wenig" Maschinen für "zu viele"
Use-Cases: "Optimal" ist im Zusammenhang mit Scheduling ein sportlicher Begriff, wenn man die konkreten Laufzeiten von Tests nicht kennt.

Was Tapper von anderen Infrastrukturen jedoch unterscheidet, ist, dass die Automatisierung zusammen mit dem Scheduler auch mit beliebig wenig Testmaschinen auskommt. Im Gegensatz zu anderen Vorbildern mit riesigen Maschinenpools, in denen üblicherweise einfach Gruppen von Maschinen pro Use-Case bereitgestellt werden.

In einer Welt von "bleeding edge"-Maschinen gibt's diesen Luxus aber nicht, und die wenigen Maschinen müssen rund um die Uhr alle verschiedenartigen Setups ausführen.

Multiplexing von Use-Case-Queues
Der Scheduler multiplext Testruns, die in "Queues" einsortiert sind. Das ist nichts weiter als eine logische Aufteilung. Man kann frei Queues anlegen und Bandbreiten zuteilen.

Üblicherweise steuert man damit, welche Use-Cases man öfter als andere ausführen will.

Beispiel: man testet mehrere verschieden Entwicklungszweige von Xen. Die älteren Versionen wie Xen 3.2 und 3.4 will man seltener testen, den Entwicklungsbranch Xen-unstable aber ganz oft, weil sich dort mehr ändert.

Man definiert also drei Queues mit relativen Prioritäten (die in Wirklichkeit Bandbreiten sind) - siehe Listing 2.

Grob bedeutet das, dass von 700 Testläufen durchschnittlich 100 aus der Xen32-Queue, 100 aus der Xen34-Queue und 500 aus der XenUnstable-Queue ausgeführt werden. Allerdings existieren evtl. noch andere Queues, die in die Anteilsberechnung mit eingehen.

```
tapper-testrun newqueue --name=Xen32 --priority=100
tapper-testrun newqueue --name=Xen34 --priority=100
tapper-testrun newqueue --name=XenUnstable --priority=500
```

Listing 2

Werden Testruns Queues zugeordnet, sieht das z.B. so aus:

```
tapper-testrun
 new --queue=Xen34 --macroprecond zomtec.mpc
```

Nähert sich z.B. ein neues Maintenance-Release im Xen34-Branch, dreht man einfach für paar Tage dessen Bandbreite hoch:

```
tapper-testrun
 updatequeue --name=Xen34 -p2000
```

Es gibt neben den gezeigten **Bandbreiten** noch eine echte High-**Priority**-Queue, die vor allen anderen Queues verwendet wird. Diese wird z.B. für das Scheduling von multi-host-Setups verwendet, oder wenn hohe Bandbreiten nicht ausreichen, um etwas wichtiges zeitnah zu schedulen.

Core-Scheduling-Algorithmus austauschbar
Der Kern-Scheduling-Algorithmus ist austauschbar. Als default wird "Weighted Fair Queuing" verwendet. In den Unittests wird beispielsweise ein Dummy-Algorithmus verwendet, der sich leichter vorhersagen lässt.

Host/Queue-Bindung für gezieltes Scheduling
Weiterhin kann man eine Testmaschine so markieren, dass auf ihr nur Testruns von einer bestimmten Queue geschedult werden. So kann ein Entwickler seine Tests in diese Queue eintragen und muss die Maschine nicht mit anderen teilen.

Damit kann man das "poor man's" Maschinenpooling realisieren, was in anderen Testinfrastrukturen das übliche Vorgehen ist. Die Maschine steht jedoch eventuell zeitweise unbenutzt rum.

Feature-getriebenes Matching von Maschinen
Der Scheduler verwaltet zu jeder Maschine Features und bietet eine Expression-Syntax an, um Anforderungen zu formulieren.

Beispielsweise sind das Memory, Anzahl Cores, Vendor, etc. oder komplexen Kombinationen davon.

Beispiel:

```
tapper-testrun
 new --requested_feature='mem > 4096 &&
              cores >= 24 && vendor eq AMD'
```

Die Expressions sind echte Perl-Expressions, ohne strict, aber mit `Safe.pm` beschränkt auf Basisexpressions.

Automatisches Neu-Einstellen für kontinuierliches Testen
Wenn Testruns in Queues immer weiter "runterrutschen", bis sie endlich dran sind, können sie automatisch wieder oben neu eingestellt werden. Das Feature nennt sich "auto_rerun" und kann beim erstmaligen Einstellen eines Testruns aktiviert werden.

```
tapper-testrun
 new --macroprecond zomtec.mpc
     --queue=Xen34
     --auto_rerun
```

Sinnigerweise nutzen die "preconditions" in diesem Fall erweiterte Features, um jedesmal etwas anderes zu testen (siehe "Precondition-Producer"), z.B. immer das jüngste Kernelimage aus einem Buildverzeichnis.

Multi-host scenarios
Für Tests an denen mehrere Maschinen teilnehmen, gibt es eine spezielle Form von "Scenario-Preconditions", die die mehreren Maschinen beschreiben und groben Synchronisierungssupport bieten.

Hierbei werden zwei Testmaschinen zeitlich möglichst dicht beieinander installiert und der Start der Testruns passiert etwa zeitgleich. Supergenaue Synchronisation müssen die Workloads ggf. noch selber machen, falls notwendig, oft aber synchronisieren die sich schon gut genug.

Web-Frontend

Frontend zu Datenbanken
Das **Web-Frontend** ist eine Catalyst-basierte Applikation. Sie präsentiert im wesentlichen Testergebnisse aus der "ReportsDB".

- Highevel-Präsentation der Testergebnisse
- Übersichtslisten, Detailsansichten
- Filtermöglichkeiten nach Zeitraum, Testsuiten, Maschinen, Erfolgsstatus
- RSS-Feeds aus solchen Filtern

Man kann (mit etwas Vorbereitung) auch automatisch auszuführende Testruns in die "TestrunDB" eintragen.

Start von Testruns

Für den Start von Testruns aus dem Web-Frontend schreibt man wieder einmal "macro preconditions", in die man Kommentarblöcke einbauen kann und Spezifikationen, welche Template-Variablen darin optional oder Pflicht sind.

Beispiel siehe Listing 3.

Für manche der Werte, die in den Web-Formularen nicht einfach als Textfeld, sondern als spezielle `HTML::FormFu`-Elemente gerendert werden sollen, gibt's das oben in der Precondition eingetragene File `kernel_reboot.yml`. Hier wird das Formularfeld "Tests" als Checkbox konfiguriert:

```
---
elements:
  - type: Checkboxgroup
    name: Tests
    label: 'Test suites to be used'
    options:
      - [ 'CTCS', 'CTCS testuite' ]
      - [ 'LMBench', 'LMBench' ]
      - [ 'LTP', 'LTP' ]
      - [ 'Kernbench', 'Kernbench' ]
      - [ 'Aim', 'Aim' ]
      - [ 'ReAim', 'ReAim' ]
      - [ 'HTS', 'HTS' ]
      - [ 'LLC', 'LLC' ]
      - [ 'Parselog', 'Parse logfiles']
      - [ 'RHv7', 'RHv7' ]
      - [ 'Phoronix', 'Phoronix' ]
```

Die Applikation rendert aus all dem generisch ein Formular, inklusive Plausibilitätsprüfungen und trägt die "macro precondition" dann in die TestrunDB ein, analog zum Kommandozeilentool:

```
# tapper-description: Build a kernel from git and test it
# tapper-mandatory-fields: giturl
# tapper-optional-fields: gitchangeset
# tapper-config-file: kernel_reboot.yml
### Usecase: "Build a kernel from git and test it by rebooting"
###
### Mandatory fields:
###
###  - git-url:        a full URL that contains the test
###                    kernel sources
###
### Optional fields:
###
###  - git-changeset: a branch name or commit ID or
###                   anything that git checkout understands
###                   (default is HEAD)
###
###  - Tests:          Choose one or more workloads to run on this
###                    kernel; possible values are:
###                      - CTCS       - CTCS test suite
###                      - LMBench    - LMbench benchmark suite
###                      - LTP        - Linux Test Project suite
---
arch: linux64
image: suse/sles10.tar.gz
mount: /
partition: sda2
precondition_type: image
---
precondition_type: kernelbuild
git_url: [% giturl %]
[% IF gitchangeset %]
changeset: [% gitchangeset %]
[% ELSE %]
changeset: HEAD
[% END %]
---
[% IF (Tests.CTCS) or (Tests == 'CTCS') %]
precondition_type: testprogram
program: /opt/tapper/bin/ctcs
timeout: 10800
---
[% END %]
# ... usw., schnipp ...
```

Listing 3

```
tapper-testrun new --macroprecond=...
  -Dgiturl=... -Dgitchangeset=...
```

Das Web-Frontend bietet alle "macro preconditions" zur Auswahl, die es in einem bestimmten Verzeichnis findet.

Ergebnis-Evaluierung

Programmierbare Abfrage-API

Wenn das Web-Frontend mit dem Sichten der Ergebnisse in Übersichtsform nicht mehr reicht, man z.B. einen bestimmten Benchmark-Wert, der tief in einem Report versteckt ist, über die Zeit aus mehreren Reports einer bestimmten Benchmarksuite extrahieren will, kann man die "Reports-API" verwenden.

Das Thema ist weiter unten in "Ausgewählte Themen / Mit der Reports-API Werte auslesen" ausführlicher beschrieben.

Komplexe Abfragen auf der Ergebnis-Datenbank

Die Abfrage-API bietet eine Mischung aus SQL und XPath, konkret `SQL::Abstract` und `Data::DPath` an, um auf einem Document-Object-Model, konkret einem `TAP-DOM` zu arbeiten.

Keine client-seitigen Tools benötigt

Ein Texteditor und `netcat` reichen. Man schickt Templates an einen Netzwerkport und bekommt als Antwort das ausgefüllte Template zurück.

Basiert auf Templates (TT oder Mason)

Mason ist etwas mächtiger, führt aber praktisch serverseitig fremden Perl-Code aus. Falls man den Templates nicht vertraut, kann man auch nur Template-Toolkit zulassen (via Config).

Testplan-Support

Testpläne sind die Kombination mehrere Tapper-Features, um einen kompletten QA-Workflow in ein Projekt-Management einzubetten.

Testplan-Hierarchie

Wieder einmal ist die zugrundeliegende Technologie eine Variante von "macro preconditions". Man verwaltet Testplane in einer Verzeichnishierarchie, wie üblich mit Template-Support.

Zusammenarbeit mit TaskJuggler

TaskJuggler (*http://taskjuggler.org*) ist salopp gesagt eine programmierbare Projektverwaltung.

Für die Zusammenarbeit mit Tapper legt man dedizierte QA-Tasks zu Feature-Implementierungs-Tasks an. Mit Tapper kann man die Task-Hierarchie von TaskJuggler auf die Testplan-Verzeichnis-Hierarchie mappen. Damit kann man automatisch Testruns schedulen, wenn die QA-Tasks im Projektplan aktuell werden.

Aus den Testergebnissen kann andererseits auch ein Ergebnis-Report generiert und an TaskJuggler gesendet werden, ein sogenanntes "timesheet" im TaskJuggler-Jargon. TaskJuggler rendert aus diesem Timesheet wiederum eine Statusübersicht zum Projektplan.

Kompletter QA-Lebenszyklus

Es mag öde klingen, aber wenn man QA im "Enterprise-Modus" betreibt, will man den gesamten Lebenszyklus von Testplanung, Testausführung und Testberichten bedienen können.

Mit TaskJuggler und Tapper hat man all das in programmierfreudiger Form unter Kontrolle, ohne in einer Projektplan-GUI rumklicken zu müssen.

Beziehung zum autotest.kernel.org-Projekt

Beim Thema Betriebssystemtesten kommt üblicherweise `autotest` zur Sprache.

Der Hauptfokus des `autotest`-Projektes liegt auf dem Testen des Linuxkernels. Es bietet eine breite Abdeckung von Linuxkernel-Funktionstests und Wrapper um viele existierende Testsuiten.

`Tapper` bietet viele komplexe Ausführungsszenarien wie Virtualisierung (mit KVM/Xen), Distributions-Tests (RHEL, SLES, Debian), SimNow-Testen und Benchmarken. `Tapper` kann alle mit frei wählbaren Bandbreiten Tests auf große oder beliebig kleine Maschinenpools multiplexen.

Der `autotest.kernel.org`-Client kann in einer `Tapper`-Infrastruktur über einen minimalistischen Wrapper benutzt werden, der das TAP-Export-Feature von `autotest` nutzt.

`Tapper` komplementiert es dann mit Testplan-Support, einer Ergebnis-Datenbank und einer homogenen Ergebnis-Abfrage-API.

Ausgewählte Themen

Perl-Tests Tapper-konform machen

Will man seine bereits TAP-konformen Tests (wie sie in der Perl-Welt der Standard sind) um die für Tapper notwendigen Meta-Informationen ergänzen, fügt man seiner Testsuite z.B. ein Testscript `t/00-tapper-meta.t` dieser Art hinzu:

```
#! /usr/bin/env perl
use Test::More;
eval "use Tapper::Test";
plan skip_all =>
  "Tapper::Test not available" if $@;
tapper_suite_meta();
```

Man führt seine Testsuite mit `prove` aus, lässt dabei die Ergebnisse in ein Archiv packen und schickt das an den Tapper-Reports-Receiver:

```
$ prove -vl -a results.tgz t/
$ cat results.tgz | netcat
    $TAPPER_REPORT_SERVER $TAPPER_REPORT_PORT
```

Die Metainformationen sehen etwa so aus, wie in Listing 4 dargestellt.

Es sind ganz normale TAP-Kommentarzeilen, die aber durch Tapper speziell als Metainfo-Headers interpretiert werden.

Mit der Reports-API Werte auslesen

Die Reports-API beruht darauf, dass

- Testreports als `TAP::DOM` bereitgestellt werden
- in diesen TAP-DOMs mittels `Data::DPath` gesucht werden kann
- die Auswahl der zu untersuchenden Reports mit `SQL::Abstract` definiert wird
- und alles gemeinsam in ein Template eingebettet ist, welches man mit `netcat` an den Server schickt, der es ausgefüllt zurückschickt

Über diesen Mechanismus gab es einen ausführlichen Vortrag auf der YAPC::EU 2009 (*http://xrl.us/tapperyapceu2009*), damals noch unter anderem Namen.

Die Einzeltechnologien `Data::DPath` und `TAP::DOM` wurden im $foo-Perl-Magazin vorgestellt.

Beispiel 1

Das ganze kann wie folgt aussehen.Die Kommandozeile:

```
$ cat report.mas  |
    netcat tapperserver 7358  >  result.txt
```

Das Template `report.mas` enthält:

```
#! mason <<EOF
Planned oprofile tests:
% foreach $plan (reportdata
    '{ suite_name => "oprofile" } ::
    //tap/tests_planned') {
   <% $plan %>
% }
EOF
```

Das Ergebnis in `result.txt` ist z.B.:

```
Planned oprofile tests:
   3
   4
   17
```

Beispiel 2

Verwendet man nun das ganze direkt in einem `gnuplot`-Template mit self-contained values, kann man das wie folgt kombinieren. Template - siehe Listing 5.

```
# Tapper-suite-name:              Tapper-Test
# Tapper-suite-version:           3.000002
# Tapper-suite-type:              software
# Tapper-machine-name:            birne
# Tapper-language-description:    Perl 5.012002,
#                                 /home/ss5/perl5/perlbrew/perls/perl-5.12.2/bin/perl
# Tapper-uname:                   Linux ss5-work 2.6.31-22-generic #73-Ubuntu SMP
# Tapper-osname:                  Ubuntu 9.10
# Tapper-cpuinfo:                 2 cores [Intel(R) Atom(TM) CPU N270   @ 1.60GHz]
# Tapper-ram:                     993MB
# Tapper-starttime-test-program:  Wed, 16 Mar 2011 01:11:08 +0100
```

Listing 4

```
#! mason debug=1 <<EOTEMPLATE
TITLE = "success ratio: CTCS"
set title TITLE offset char 0, char -1
set style data linespoints
set term png size 1200, 800
set output "CTCS_ratio.png"
set yrange [80:110]
plot '-' using 0:2 with linespoints lt 3 lw 1 title "ratio"

% my @time  = reportdata '{ suite_name => "CTCS" } :: /report/created_at_ymd';
% my @ratio = reportdata '{ suite_name => "CTCS" } :: //success_ratio';
% foreach my $i (0..@ratio) {
    <% $time[$i] %> <% $ratio[$i] %>
% }
EOTEMPLATE
```

Listing 5

Die Kommandozeile:

```
$ cat CTCS_ratio.gnuplot |
   netcat tapperserver 7358 | gnuplot
```

erzeugt direkt eine Datei CTCS_ratio.png, siehe Abbildung 1.

Alternativ kann man die Werte auch nur in YAML oder JSON dumpen und mit der Toolchain seiner Wahl weiterbearbeiten.

Einsatz-Szenarien

Betriebssystemtests

Der Klassiker und komplexeste Anwendungsfall. Man benötigt praktisch alle Komponenten.

Benchmarking

Ähnlich wie Betriebssystemtests - der Schwerpunkt liegt jedoch auf reproduzierbarer Ausführungsumgebung und dem Tracking von Werten. Durch das Aufsetzen einer Maschine immer von Grund auf, basierend auf stabilen OS-Images, ist die Umgebung immer wieder gleich. Das ist eine Grundvoraussetzung für wertvolle Benchmarkwerte.

TAP-Datenbank

Man möchte einfach nur manuell Tests in verschiedenen Umgebungen ausführen, die Ergebnisse aber speichern und auswerten.

Nur Automatisierung

Einfach nur Automatisierung, ohne Tests. Das könnte z.B. ein Build-System sein, das mehr wechselnde Umgebungen (Compiler, Libs, Bitness) braucht als man Maschinen hat. Die Ergebnisse könnten gebaute Packages sein, die z.B. in einem NFS-gemounteten Verzeichnis landen.

Hier profitiert man von der Wiederholbarkeit, der Überwachung mit Timeouts, dem Scheduling nach Prioritäten/ Bandbreiten.

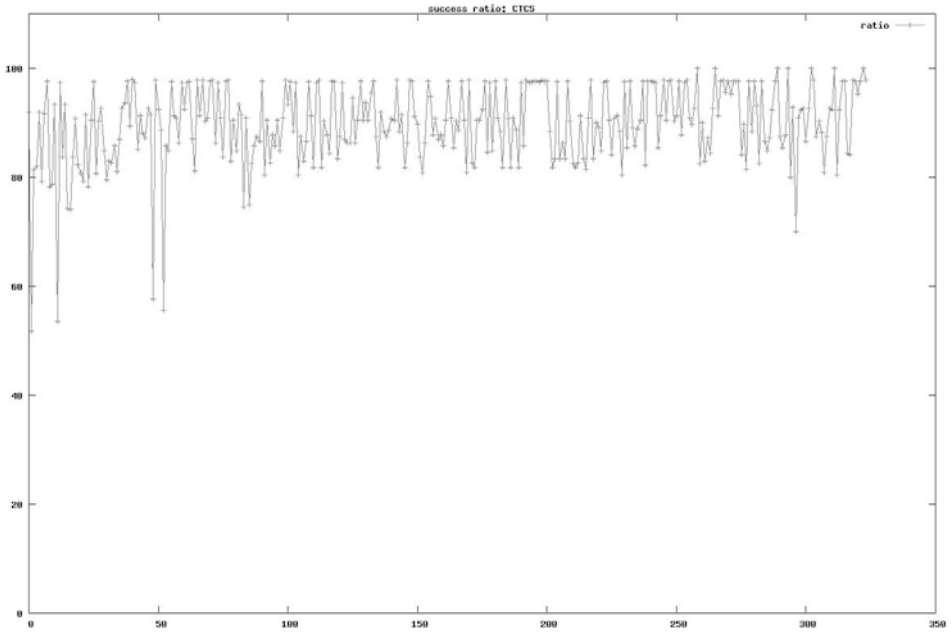


Abbildung 1

Rolf Langsdorf

Hack your applications! Universelle Plugins scripten via Mailto

Die Aufgabenstellung

Letztens stand ich vor dem Problem, dass ich aus einer Präsentation heraus spontan oder auf Hörerwunsch per Klick kleine Demoanwendungen zum Thema starten wollte, ohne aufwändig die Oberfläche wechseln zu müssen.

Mein Quellformat - Emacs Org-Mode - erlaubte mir HTTP Links zu notieren, sie mit "Demo1" oder ähnlichem zu benennen und diese in diversen Zielformate (HTML, Latex, PDF,...) durchzureichen. Wäre nun schön, wenn der PDF-Reader daraus Aktionen initiieren könnte.

Der erste Lösungsgedanke war also, mir mit `HTTP::Proxy` eine kleine Webapplikation zu schreiben, der ich dann auf einem bestimmten Port Requests schicke, um die Aktionen zu starten. Z.B. ein Fenster mit einer Demo nach vorne poppen zu lassen und zum Schluss verschwinden zu lassen. Ich müsste aber immer erst daran denken, den Server im Hintergrund zu starten, damit so etwas ginge.

Die Idee

Aber ein anderer Weg eröffnet weit mehr Freiheiten. Denn praktisch alle Applikationen, die HTTP Links unterstützen, erlauben auch das **mailto:** Protokoll. Und das geht nicht nur mit Markups in HTML-Browsern, PDF-Readern und Powerpoint. Bei vielen Tools wie Editoren oder sogar vielen Terminalkonsolen läuft auch eine Patternsuche, die automatisch Mailadressen "aktiviert".

Zusätzlich zu all dem finden sich auch noch überall Menüeinträge, die einem anbieten, ganze Dokumente oder Inhalte zu versenden. Beispielsweise im Hauptmenü "Datei" des Firefox gibt es "URL senden", per Rechtsklick geht das auch für einzelne Grafiken oder Links.

Und nicht genug, dass man den Mailversand oft per Tastenkombinationen anstoßen kann, dort wo es eingebettete Scriptsprachen wie Javascript gibt, kann man ihn sogar programmieren. Die Kontrolle über das lokale Mailprogramm liefert einem also eine Applikationen, Betriebssysteme und Dokumentenformate überspannende universelle Pluginschnittstelle, wo wir unsere Perlscripte (zumindest einseitig) andocken können, ohne uns mit sowas wie z.B. XUL-Spezifikationen abplagen zu müssen.

Den Mailer hacken

Heutige Betriebssysteme erlauben recht einfach einzustellen, unter welchem Pfad der Mailer gefunden wird. In meinem Ubuntu reicht es sich zu `System / Einstellungen / Bevorzugte Anwendungen / Emailbetrachter` durchklicken und z.B. `/home/lanx/bin/chkMail.pl %s` eintragen.

Historisch ist es so, dass man in der `.mailcap` Datei einstellen konnte, von wo und mit welchen Parametern eine Applikation zu einem bestimmten MIME-Type gestartet wurde. Details dazu siehe `man mailcap`.

Manchmal kann man es auch auf Applikationsebene - z.B. im `about:config` des Firefox - spezifizieren.

Und wenn man einfach faul ist, benennt man das eigene Programm wie den Default-Mailer und sorgt dafür, dass es sich im Suchpfad früher findet.

Wrapper

Nun will man ja eventuell immer noch regulär E-Mails schreiben können, deswegen bietet es sich an, das eigene Programm als Wrapper anzulegen der die Mailadresse untersucht und per Default den richtigen Mailer aufruft. Entspricht die Adresse einem bestimmten Pattern, etwa einer nicht existierenden Domäne wie "@plugin" wird die "Mail" abgefangen und das Kommando ausgeführt.

Das Besondere dabei ist, dass dem Mailer eine Mailto-URL übergeben wird. Das heißt man kann hier noch viele Zusatzparameter URL-Encoded mitgeben. Neben `to`, `subject`, `cc` und `bcc` wird in der Regel auch `body` unterstützt. Oft ist man sogar frei in der Wahl zusätzlicher Felder im Suchstring.

Folgendes Script kann zum Testen genommen werden, es poppt unter X ein Fenster mit allen übermittelten Feldern auf, das sich nach 20 sek wieder schließt.

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

my $input=shift;

my $uri = URI->new($input,"mailto");

#- Titel
my $txt="CHKMAIL\n\n";


#- Adresse extrahieren
$txt.="to: ". $uri->to ." \n";

my %fields=$uri->query_form;

#- sonstige Felder extrahieren
while ( my ($key,$value) = each %fields ) {

  $txt .= "$key:  $value\n";
}


#- poppe Meldefenster mit Feldern auf
open my $xmsg, "|-",
   'xmessage -nearmouse -timeout 20 -file -';
print $xmsg $txt;
print $xmsg "Original-URL: $input";
close $xmsg;
```

Jetzt würde es reichen die Aktionen thematisch in Unterdirectories zu organisieren und auf Anfrage zu `require`n.

Beispielhafte Anwendungen

Hier mal Anwendungsgebiete, die sich anböten:

1. Aktionsfelder durch verschiedene Dokumentenformate tunneln

Wie im Eingangsszenario beschrieben, kann ich nun in Org-Mode folgendes notieren:

```
[[mailto://praesentation@plugin?subject=
   startdemo][Starte Demo]]
```

Und in jedem Ausgabeformat - sei es HTML (z.B. mit S5), PDF (z.B. mittels Beamer-Style & LaTeX) oder Powerpoint oder direkt im Emacs - kann ich auf "Starte Demo" klicken.

Wird die fertige Präsentation irgendwann an jemand Fremden weitergegeben, wundert er sich höchstens, dass sein Mailer nach einem Klick hochpoppt.

2. Browserplugins in Perl

Schreibt man `mailto:name@domain` in die Adressleiste eines Browsers, wird der Mailer aufgerufen. Entsprechend kann man mit folgendem Javascript-Code Mails versenden:

```
javascript:window.location.href =
   "mailto:name@domain.plugin?body=" +
      escape(daten);
```

Dieser Code kann auch in einem Bookmark stehen. Und in diesem Bookmarklet kann noch mehr JS-Code stehen, der gewünschte Daten der aktuellen Seite extrahiert, und unserem Programm mitteilt.

Z.B. könnten wir auf einer TV Seite Fernsehsendungen auswählen und unser Perlscript programmiert den Videorecorder.

3. Start eines lokalen Servers aus dem Browser heraus

Und wenn wir - wie oben beschrieben - einen kleinen Proxy brauchen um in beide Richtungen zwischen Browser und Perl-Programm zu kommunizieren, können wir ihn so on-demand starten. Ein `window.setIntervall()` in unserem Bookmarklet reicht, um einen Port bis zum Start zu pollen. `Pod::POM::Web` ist ein Beispiel für ein nützliches Tool, bei dem man meist zu faul ist, es manuell zu starten. (Es erlaubt die lokal installierten Module zu browsen.)

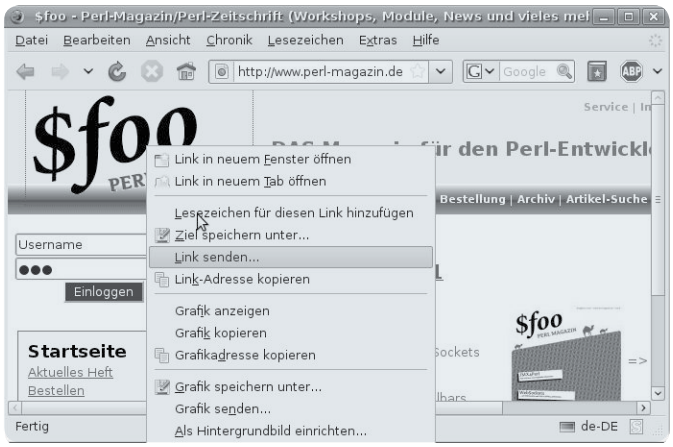


Abbildung 1: Firefox bietet bei Bildlinks sowohl "Link senden" als auch "Grafik senden"

Abbildung 2: Popup mit Mail-Parameter für "Grafik senden"

Fazit

Für alle die dargestellten Anwendungsgebiete gibt es schon reichlich andere Lösungen, aber selten sind sie so vielseitig und leicht zu konfigurieren wie diese. Und wie oft hat man die Gelegenheit Plugins (hauptsächlich) in Perl umzusetzen... =)

Wir sind Vorreiter im Bereich innovative Prämiendienstleistungen und übernehmen sowohl Einkauf, Lagerung und Versand als auch den After-Sales-Service und zugehörige IT-Services. Unser Dienstleistungspool umfasst Werbeprämien, Prospekte, Punkte- sowie Prämien-Onlineshops zur Kundenbindung und Kundenneugewinnung. Zu unseren Kunden zählen u.a. Verlage, Banken, Versicherungen sowie große Handels- und Industrieunternehmen.

Zur Verstärkung unserer IT-Abteilung suchen wir am Standort unseres Unternehmens in Weingarten bei Karlsruhe kurzfristig einen

Web-Entwickler (m/w)

Ihre Aufgaben
Als Mitarbeiter in einem Entwicklungsteam liegt Ihr Arbeitsschwerpunkt in der Erstellung von Internet-Shopsystemen und Online-Applikationen. Sie entwickeln unser zentrales Shopsystem weiter, erarbeiten zusammen mit Kunden oder nach Kundenvorgaben individuelle Lösungen und bringen eigene Ideen und Lösungsansätze in die Projekte ein.

Die Erstellung von kleineren Stand-Alone-Lösungen und der kontinuierliche Ausbau unserer Online-Infrastruktur im Team mit anderen Mitarbeitern der IT-Abteilung, Freelancern und IT-Dienstleistern liegt ebenfalls in Ihrem Aufgabenbereich.

Ihr Profil / Qualifikationen
Optimalerweise verfügen Sie über eine abgeschlossene informatische Ausbildung und/oder einige Jahre Beruferfahrung als Entwickler von Online-Applikationen.

Sie haben sehr gute Kenntnisse in der Programmiersprache **Perl / modPerl**.

Von Vorteil sind weiterhin Kenntnisse und Praxis in den Bereichen:
- HTML + CSS + JavaScript
- Programmiersprache PHP
- Linux-Administration (Suse)
- Webserver (Apache)
- Datenbankadministration (MySQL u.a.)
- Erstellen von Dokumentationen
- Webservice / XML / Schnittstellen
- Gefühl für nutzerfreundliches Interface-Design / grundlegende grafische Kompetenz

Sie sind ein Teamplayer, arbeiten lösungsorientiert, denken sich schnell in neue Projekte ein, verfügen über Organisationstalent und gutes analytisches Denken.

Sofern Sie zusätzlich über ein hohes Maß an Leistungsbereitschaft und Motivation verfügen, sollten wir uns kennen lernen.

Wir bieten
- einen umfassenden Verantwortungsbereich mit viel Gestaltungsspielraum
- eine solide finanzierte Firma mit rund 30 Jahren Branchenerfahrung
- ein motiviertes Team, das Sie bei technischen und kaufmännischen Fragen unterstützt
- abwechslungsreiche Projekte und neue Herausforderungen
- Freiraum zur Verwirklichung eigener Ideen
- eine flache Firmenhierarchie und enge Zusammenarbeit mit der Geschäftsführung
- eine kreative Arbeitsumgebung mit hoher Flexibilität

Wenn wir Ihr Interesse geweckt haben, dann senden Sie uns Ihre Bewerbungsunterlagen unter Angabe des frühestmöglichen Eintrittstermins und Ihrer Gehaltsvorstellung an:

IPO PrämienServices GmbH
Rudolf-Diesel-Str. 10
76356 Weingarten
E-Mail: job63@ipo-ps.de
www.ipo-ps.de

IPO
PRÄMIENSERVICES

Erste Fragen beantwortet Ihnen unser Leiter IT, Herr Haberey, telefonisch unter 07244 / 600-61.

Renée Bäcker

Moose Tutorial - Teil 4: Rollen und Traits

In der letzten Ausgabe wurde gezeigt, wie mit Moose eine Vererbungshierarchie aufgebaut wird. Das Problem dabei ist, dass dadurch extrem tiefe Vererbungsbäume entstehen können. Das wiederum führt zu schlechterer Wartbarkeit und auch Erweiterungen sind nicht immer ganz einfach umzusetzen.

Aus diesem Grund wird in diesem Teil des Moose Tutorials das Rollenkonzept vorgestellt. Zusätzlich gibt es ein verwandtes Thema, das ebenfalls in dieser Ausgabe besprochen wird - Traits.

Rollen und Traits - zwei unbekannte Wesen?!

So etwas wie die Moose-Rollen gibt es in anderen Programmiersprachen nicht. Es gibt Sprachen, die ähnliche Konstrukte kennen, aber die unterscheiden sich immer in einigen Punkten von den Rollen.

Rollen sind Codeteile, die ein Verhalten implementieren, die eine Klasse haben soll. Diese Rollen können in verschiedenen Klassen eingesetzt werden, sie sind also wiederverwendbar. Sie selbst sind aber keine Klassen, können also nicht instantiiert werden. Auch Subklassen von Rollen können nicht erzeugt werden.

Rollen werden direkt in eine Klasse eingebunden, d.h., dass alle Methoden und Attribute, die in der Rolle definiert werden, dann in der Klasse zur Verfügung stehen. Von außen sieht es so aus, als wären diese Methoden und Attribute direkt in der Klasse definiert worden.

Warum sollte ich das dann nicht direkt in der Klasse definieren? Rollen werden typischerweise für Verhalten erstellt, die viele unterschiedliche Klassen haben sollen - die auch nicht unbedingt etwas miteinander zu tun haben. So sollen sowohl Menschen als auch Fische schwimmen, aber es wird mir wohl jeder zustimmen, wenn ich sage, dass Menschen und Fische in der Vererbungshierarchie nicht unbedingt als "Geschwister" anzusehen sind. Und damit dieses Verhalten nicht in jeder dieser Klassen definiert werden muss, wird eine Rolle erstellt.

Jetzt wird vermutlich der Einwand kommen, dass das auch durch Vererbung gelöst werden kann. Richtig, das kann es. Aber wie schon weiter oben erwähnt, kann die Vererbung zu sehr großen Vererbungsbäumen führen. Es gibt auch Fälle, in denen die Vererbung besser ist als der Einsatz von Rollen. Es muss daher immer von Fall zu Fall unterschieden werden, was das passende Konzept ist. Ein weiterer Vergleich zwischen Rollen und Vererbung folgt noch in diesem Artikel.

Mit Rollen kann auch definiert werden, welches Verhalten eine Klasse umsetzen muss. Das bedeutet, dass die Rolle bestimmt, dass die Klasse "loggen" können muss. Aber sie implementiert dieses Verhalten nicht. Java-Programmierer kennen so etwas unter "Interfaces".

Als Traits werden in der Moose-Dokumentation die Rollen bezeichnet, die auf Meta-Klassen angewendet werden. Wie diese Traits umgesetzt und verwendet werden, wird am Ende des Artikels gezeigt.

Rollen im Einsatz

Nach dem eher theoretischen Teil kommen jetzt endlich Rollen zum Einsatz. Dazu bleiben wir bei dem Beispiel der letzten Ausgaben von $foo: Es gibt verschiedene Zeitschriften, die als PDF ausgegeben werden und welche, die als HTML ausgegeben werden. In der letzten Ausgabe wurde das mit Vererbung gelöst. Einen Teil des Vererbungsbaums ist in Abbildung 1 zu sehen.

In dieser Ausgabe soll das ohne Vererbung gelöst werden.

Moose-Rollen verwenden `Moose::Role`. Dieses Modul stellt - genau wie `Moose` - einige Schlüsselwörter wie `has` bereit. Danach folgt Code, wie er auch in normalen Moose-Klassen vorkommt.

Eine einfache Rolle, die eine Methode und ein zusätzliches Attribut zur Verfügung stellt:

```
package PDF;

use Moose::Role;

has PDFVersion => ( is => 'ro' );

sub pod_to_pdf {
    # call pod2pdf
}

no Moose::Role;

1;
```

Soweit nichts Ungewöhnliches. Warum aber `Moose::Role` statt `Moose`? Rollen können ein paar Sachen nicht, die normale Moose-Klassen können. Zum Beispiel gibt es für Rollen keinen Konstruktor.

Jetzt existiert eine Rolle und sie soll in einer Klasse "Zeitschrift" verwendet werden. Damit wird die Zeitschrift PDF-fähig. Zum Einbinden einer Klasse gibt es das Schlüsselwort

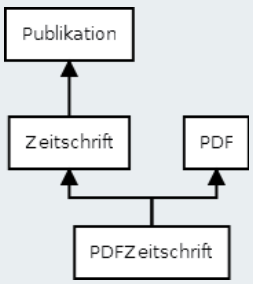


Abbildung 1: Vererbungshierarchie für PDF-Zeitschriften

`with`. Nach dem Laden von `Moose` steht es in der Klasse zur Verfügung.

```
package Zeitschrift;

use Moose;
with 'PDF';

has title => ( is => 'ro' );

no Moose;

1;
```

Da die "Zeitschrift" die Rolle "PDF" einbindet, haben Objekte der Klasse das zusätzliche Attribut "PDFVersion" und die zusätzliche Methode `pod_to_pdf`. Die Attribute, die über die Rollen integriert werden, können auch direkt bei der Objekterzeugung angegeben werden.

```
use Zeitschrift;

my $foo = Zeitschrift->new(
    title => '$foo',
    PDFVersion => 2.5,
);

print $foo->title, ': ',
      $foo->PDFVersion,"\n";
$foo->pod_to_pdf;
```

In der Klasse selbst (und auch in deren Subklassen) kann man die Attribute, die über die Rollen hinzukommen, erweitern. Das funktioniert auf dieselbe Art und Weise wie bei der Vererbung: Bei dem Attributnamen, der an `has` übergeben wird, wird ein '+' vorangestellt.

```
package Zeitschrift;
use Moose;
with 'PDF';

has title => ( is => 'ro' );

has '+PDFVersion' => (
    is => 'ro',
    default => 1.4,
);
```

Hier muss aber darauf geachtet werden, dass die Einbindung der Rolle (`with 'PDF';`) vor der Attributerweiterung kommt. Sonst bricht das Programm ab, weil Moose sonst das Attribut nicht finden kann (Listing 1).

Rollen "statisch" und "dynamisch" anwenden

In der Beispiel-Klasse in Listing 2 wird schon in der Klassendefinition festgelegt, welche Rollen verwendet werden. Das funktioniert in vielen Fällen auch einwandfrei. Z.B. weiß man schon von Anfang an, ob eine Klasse Logging betreiben

```
Could not find an attribute by the name of 'PDFVersion' to inherit from in \
    Zeitschrift at C:/strawberry/perl/site/lib/Moose/Meta/Class.pm line 628
    Moose::Meta::Class::_process_attribute('Moose::Meta::Class=HASH(0x20aa14c)',\
     '+PDFVersion', 'default', 1.4, 'definition_context', 'HASH(0x20aa050)',\
      'is', 'ro') called at C:/strawberry/perl/site/lib/Moose/Meta/Class.pm line 300
```

Listing 1

```
package OTRS::OPM::Analyzer::Role::BasicXMLCheck;

use Moose::Role;
use XML::LibXML;

with 'OTRS::OPM::Analyzer::Role::Base';

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

    return if $document->{filename} !~ m{ \.xml \z }xms;

    my $content = $document->{content};
    my $check_result = '';

    eval {
        my $parser = XML::LibXML->new;
        $parser->parse_string( $content );
    } or $check_result = $@;

    return $check_result;
}

no Moose::Role;

1;
```

Listing 2

soll oder nicht - und dementsprechend wird eine Logging-Rolle eingebunden oder nicht.

Es gibt aber auch Fälle, in denen eine Rolle dynamisch einer Klasse zugewiesen werden soll. Z.B. kann bei der Analyse von OTRS-Erweiterungen mit dem Paket OTRS::OPM::Analyze im Konstruktor angegeben werden, welche Prüfungen gemacht werden sollen:

```
my $analyzer = OTRS::OPM::Analyzer->new(
    configfile => $config,
    roles => {
        opm  => [qw/Dependencies/],
        file => [qw/BasicXMLCheck/],
    },
);
```

Die einzelnen Prüfungen sind als Rollen umgesetzt (Listing 2).

Wenn die Datei analysiert werden soll, werden dynamisch die Rollen geladen, die im Konstruktor angegeben wurden. In diesem Beispiel sind das die Prüfungen `BasicXMLCheck` und `Dependencies`.

Das Laden der Rollen findet in einer Schleife statt (Listing 3). Was es mit dem `alias` auf sich hat, wird in einem späteren Abschnitt erläutert. Hier geht es erstmal darum, dass man nicht nur "statisch" die Rollen verwenden kann, sondern diese zur Laufzeit lädt.

```
for my $role ( @{ $roles{$area} } ) {
    with __PACKAGE__ . '::Role::' . $role
      => {
            -alias => {
                check => 'check_' . lc $role
            }
        };
}
```

Listing 3

Ein dritter Ansatz ist es, nur einzelnen Objekten eine Rolle dynamisch zuzuordnen, z.B. wenn sich ein User auf der Webseite einer Konferenz befindet. Am Anfang ist dieser User eine ganz normale Person, ohne besonderes "Verhalten". Dann entschließt sich dieser Nutzer, sich für diese Konferenz anzumelden. Ab sofort ist der User ein Teilnehmer, der ein paar zusätzliche Informationen angibt - z.B. wie oft er in vorherigen Jahren an der Konferenz teilgenommen hat. Ein paar Minuten später entscheidet sich der User dafür, nicht einfach nur teilzunehmen, sondern auch einen Vortrag zu halten. Also wird er zu einem Vortragenden.

Diese "Typänderungen" kann man natürlich in Klassen abbilden - muss man aber nicht. Hier bieten sich Rollen an. Der User ist und bleibt eine Person. Nur weil er zum Teilnehmer wird, wird er nicht eine andere Art von Mensch, er hat nur ein anderes "Verhalten".

Für solche Fälle kann man dem Nutzer dynamisch neue Rollen zuweisen:

```
use Person;
use Speaker;

my $person = Person->new( name => 'renee' );

Speaker->meta->apply( $person );
$person->speak;
```

Als erstes wird ein neues Objekt von `Person` erzeugt. Damit haben wir ein einfaches Objekt, das keinerlei spezielles Verhalten hat. `Speaker` ist eine Rolle, die die Methode `speak` bereitstellt. Danach wird die Rolle auf das Objekt angewendet. Ab sofort kann die Person einen Vortrag halten.

Hier tauchen zum ersten Mal die Meta-Klassen im Code auf. Auf diese wird an dieser Stelle nicht näher eingegangen, da Meta-Klassen Thema der nächsten Ausgabe sind.

Durch solch ein dynamisches Hinzufügen von Rollen verhindert man, dass man viele verschiedene Klassen definieren muss, die alle möglichen Kombinationen von Rollen verwenden.

Einsatzszenarien für Rollen

Wie Rollen allgemein eingesetzt werden, wurde in den vorigen Abschnitten gezeigt. Am Anfang wurde auch schon gesagt, dass die Rollen in Moose eine Mischung aus Smalltalk Traits, Ruby Mixins und Java Interfaces sind. In den folgenden Abschnitten werden einfach ein paar Einsatzszenarien für Rollen gezeigt. Natürlich können die einzelnen Sachen auch gemischt werden. Wie bei Perl allgemein gibt es eigentlich keine Einschränkungen.

Rollen für zusätzliche Attribute

Dieses Szenario wurde weiter oben schon gezeigt. Hier werden einfach zusätzliche Attribute zur Verfügung gestellt, ohne dass zusätzliches Verhalten implementiert wird. Ein Beispiel wäre die allgemeine Klasse 'Person'. Mitarbeiter sind Personen, die nicht mehr dürfen als andere, aber sie haben ein Gehalt und eine Personalnummer.

```
package Employee;
use Moose::Role;

has employee_nr => (
    is => 'ro',
    isa => 'Int',
    required => 1,
);

has salary => (
    is => 'rw',
    isa => 'Int',
);

1;
```

Rollen für zusätzliche Methoden

Ähnlich wie Rollen für zusätzliche Methoden gibt es den Anwendungsfall, in dem einfach zusätzliche Methoden zur Verfügung gestellt werden. Das dürfte so ziemlich das häufigste Einsatzszenario sein, in dem Rollen zur Anwendung kommen.

```
package PDF;
use Moose::Role;

sub as_pdf {
    print 'publish as PDF';
}
```

Rollen als Label

Eigentlich ist das ein Spezialfall vom vorherigen Einsatz. Es wird eine zusätzliche Methode bereitgestellt, die aber nichts macht als einfach nur "1" zurückzugeben.

Das macht dann Sinn, wenn die Rolle die Frage "Ist das Objekt XY?" mit ja oder nein beantworten soll - z.B. haben wir viele Personen in einem Unternehmen? CIO und CEO gehören zum Vorstand. Wenn jetzt eine Anwendung ein Objekt eines Mitarbeiters erzeugt, soll die Rolle (Listing 4) einfach die Frage "Ist der Mitarbeiter Mitglied im Vorstand?" beantworten.

```
package Vorstand;
use Moose::Role;

sub is_vorstand { 1 }

no Moose::Role;
1;
```

Listing 4

Dann lässt sich diese Rolle als Label verwenden:

```
use Vorstand;
my $employee = Employee->new;
Vorstand->meta->apply( $employee );

if ( $employee->is_vorstand ) {
    print "Sie bekommt ein neues Auto";
}
```

Rollen als Interface

In einigen Fällen soll die Rolle einfach nur festlegen, welche Methoden von einer Klasse implementiert werden sollen. Das entspricht in etwa den Java-Interfaces. In den Rollen wird dabei die Methode nicht ausprogrammiert, sondern einfach nur gesagt, dass alle Klassen, die diese Rolle einbinden, die Methode implementieren müssen.

Das eignet sich dann besonders, wenn man sicherstellen möchte, dass Rollen oder Klassen bestimmte Methodennamen verwenden.

In Moose gibt es das Schlüsselwort `requires`, um diese Pflichtmethoden zu benennen. In diesem Beispiel fordert die Rolle `Address`, dass Klassen die Methode `address` implementieren müssen.

```
package Address;

use Moose::Role;

requires 'address';

no Moose::Role;

1;
```

Die Klasse "Person" benutzt die Rolle und definiert die Methode `address` (Listing 5).

Würde "Person" diese Methode nicht implementieren, würde Moose schon beim Kompiliervorgang eine Fehlermeldung ausgeben (Listing 6).

```
'Address' requires the method 'address'\
  to be implemented by 'Person'
```

Rollen in einer Rolle kombinieren

Bei den Rollen besteht außerdem die Möglichkeit, verschiedene Rollen in einer Rolle zu vereinigen. Das ist dann von

```
package Person;

use Moose;
with 'Address';

sub address {
    print "Adresse...";
}

no Moose;

1;
```

Listing 5

Vorteil, wenn häufiger die gleichen Kombinationen an Rollen in Klassen eingesetzt werden. Dann wird einfach eine neue Rolle erstellt, die alle anderen Rollen einbindet:

```
package PersonenVerhalten;

use Moose::Role;
with 'Logger', 'PersonenInterface',
     'Programmer';

no Moose::Role;
1;
```

In den Klassen, die diese Rollen benötigen, müssen jetzt nicht immer alle Rollen geladen werden, sondern nur noch die neu erstellte.

```
package Speaker;

use Moose;
with 'PersonenVerhalten';

1;
```

Parametrisierbare Rollen

Bisher wurden nur Rollen gezeigt, die einfach mit `with` eingebunden wurden. In den Rollen selbst war dann das Verhalten starr beschrieben. Das reicht aber manchmal nicht aus. Man möchte den Rollen noch Parameter übergeben, um das gewünschte Verhalten näher zu spezifizieren.

Um das zu erreichen, wird das Modul `MooseX::Role::Parameterized` verwendet.

Als Beispiel soll eine Rolle dienen, mit der Logdateien geschrieben werden können. Verschiedene Komponenten einer Anwendung sollen aber in unterschiedliche Logdateien schreiben. Deswegen muss man einen Parameter angeben, in dem der Dateinamen steht.

Die Rolle sieht wie folgt aus:

```
package Logger;

use MooseX::Role::Parameterized;

parameter file => (
    is       => 'rw',
    isa      => 'Str',
    required => 1,
);

role {
    my $p = shift;

    method log => sub {
        if ( open my $fh, '>>', $p->file ) {
            print $fh $_[1],"\n";
        }
    };
};

1;
```

Mit dem Schlüsselwort `parameter` definiert man den Parameter. Hier stehen alle Möglichkeiten von Attributen in Moose zur Verfügung (siehe auch Moose Tutorial Teil 1 in $foo Nr. 15).

`role` wird ein Block übergeben, der dann die Rolle erzeugt. Als erster Parameter wird ein Objekt übergeben, in dem alle Parameter gespeichert sind. Auf die übergebenen Parameter greift man über die getter-Methoden zu.

Mit `method` werden die Methoden der Rolle definiert, als erster Parameter wird der Methodenname und als zweiter Parameter eine anonyme Subroutine übergeben.

Jetzt kann die Rolle in verschiedenen Klassen verwendet werden.

```
package Tier;

use Moose;
with 'Logger' => { file => 'tier.log' };

[...]
```

```
package Person;

use Moose;
with 'Logger' => { file => 'person.log' };
```

Damit loggen die beiden Klassen in unterschiedliche Dateien.

Kannst Du ...?

Wenn es um Vererbung geht und man möchte sicherstellen, dass ein Objekt von einem bestimmten Typ ist, wird die Methode `isa` verwendet.

```
my $cgi = CGI->new;
print 'ja' if $cgi->isa( 'CGI' );       # ja
print 'ja' if $cgi->isa( 'Anything' ); #
```

Um abzufragen, ob ein Objekt ein bestimmtes Verhalten hat, also eine bestimmte Rolle verwendet, gibt es die Methode `does`:

```
package Zeitschrift;

use Moose;
with 'PDF';

no Moose;
1;
```

```
package PDF;

use Moose::Role;
no Moose::Role;

1;
```

```
use Zeitschrift;
my $foo = Zeitschrift->new;
print 'ja' if $foo->does( 'PDF' );      # ja
print 'ja' if $cgi->isa( 'Anything' ); #
```

Mittels der Methode `does` kann man also überprüfen, ob eine Rolle auf das Objekt angewendet wurde.

"Probleme" bei Rollen

Am Anfang wurde schon etwas zu dem Thema Rollen vs. Vererbung gesagt. Hier sollen Punkte gezeigt werden, über die man stolpern kann, wenn man Rollen einsetzt.

Rollen können nicht so viel wie normale Klassen. Rollen können nicht von anderen Rollen erben. Das Schlüsselwort `extends` existiert nicht und auch der Versuch, Attribute der Ausgangsrolle mit `has '+Attribut' => ( ... )` zu erweitern, scheitert.

Soll jetzt die PDF-Rolle von oben generell Anwendung finden, aber die PDF-Version soll als Defaultwert '1.4' haben, so kann keine Rolle 'PDF14' erstellt werden, die einfach das Attribut 'PDFVersion' erweitert. Es muss vielmehr eine Rolle erstellt werden, die die Rolle 'PDF' mit `with` einbindet und das Attribut komplett neu definiert.

```
package PDF14;

use Moose::Role;
with 'PDF';

has 'PDFVersion' => (
    is      => 'ro',
    default => sub { 1.4 },
);

no Moose::Role;

1;
```

Bei einem einzelnen Attribut ist das noch machbar. Wenn es aber eine Rolle mit Adressdaten gibt und eine neue Rolle soll die erlaubten Werte der Addressattribute einschränken, wird das viel Arbeit.

Ein weiteres Feature von Moose-Klassen, das von Rollen nicht unterstützt wird, ist `augment` (siehe $foo Ausgabe 16). Mit `augment` werden die gleichnamigen Methoden von der obersten Klasse in der Hierarchie bis zur untersten Klasse aufgerufen. Da dieses Feature auf dem Prinzip der Super-/Subklassen basiert, kann das mit Rollen nicht funktionieren.

Rollen können auch nicht instantiiert werden - es sind keine Klassen. Manch einer könnte in Versuchung geraten und ein Objekt der Rolle zu erzeugen um an das Verhalten, das die Rolle zur Verfügung stellt, zu kommen. Das funktioniert aber nicht!

Konflikte mit Methoden

Gerade wenn man mit vielen gleichartigen Rollen zu tun hat, kann es vorkommen, dass mehrere Rollen gleichnamige Methoden zur Verfügung stellen. Z.B. die Prüfungs-Rollen bei OTRS::OPM::Analyzer implementieren alle eine Methode `check`. Was passiert jetzt? Bei Mehrfachvererbung ist es so, dass die erste Klasse gewinnt (Listing 6) - da unterscheidet sich Moose nicht von der Standard-Objektorientierung in Perl 5.

```
package Kind;

sub new {...}
sub spielen { print __PACKAGE__ }

package CDPlayer;

sub new  {...}
sub spielen { print __PACKAGE__ }

package MischMasch;

our @ISA = qw(Kind CDPlayer);

sub new { ... }

package main;

my $mm = MischMasch->new;
$mm->spielen; # Kind
```

Listing 6

Dabei gibt es auch keine Warnungen oder gar Fehlermeldungen. Man muss also genau wissen was man macht.

Rollen helfen hier, so einen Konflikt gleich zu entdecken. Wenn `Kind` und `CDPlayer` als Rollen umgesetzt sind und `MischMasch` beide Rollen benutzt (Listing 7), dann gibt es eine Fehlermeldung (Listing 8).

Zwei Wege, wie mit so einem Konflikt umgegangen werden kann, sind schon in der Fehlermeldung genannt. Man kann einfach eine Methode mit dem gleichen Namen wie in den Rollen auch in der Klasse implementieren. Methoden der Klasse haben Vorrang vor den Methoden der Rollen. Wenn Moose also eine Methode in der Klasse erkennt, spielen die Methoden in den Rollen keine Rolle mehr.

Man muss dabei natürlich beachten was die Methoden machen, wenn man in der Klasse eine Methode verwendet, die in der Rolle Werte setzen würde. Wenn diese im weiteren Programmablauf noch benötigt werden, werden Fehler passieren, die man so nicht erwartet.

```
package MischMasch;

use Moose;
with 'Kind', 'CDPlayer';
```

Listing 7

```
Due to a method name conflict in roles 'CDPlayer' and 'Kind', the method 'spielen'\
 must be implemented or excluded by 'MischMasch' at \
 C:/strawberry/perl/site/lib/Moose/Meta/Role/Application.pm line 77
```

Listing 8

Eine weitere Möglichkeit ist, bei einer Rolle explizit die Methode auszuschließen. Das funktioniert mit `excludes`.

```
package MischMasch;

use Moose;
with 'Kind', 'CDPlayer' =>
                 { -excludes => 'spielen' };
```

Wenn mehrere Methoden ausgeschlossen werden sollen, kann eine Arrayreferenz übergeben werden.

Positionskämpfe

Im Großen und Ganzen spielt es keine Rolle, wann eine Rolle geladen wird - und in welcher Reihenfolge. Es gibt aber eine Ausnahme: Wenn es um Methoden-Modifier geht. Hier spielt es eine wichtige Rolle.

```
use Moose;

has title    => ( is => 'ro' );
has format   => ( is => 'rw' );
has articles => (
    traits     => ['Array'],
    is         => 'rw',
    isa        => 'ArrayRef[Str]',
    auto_deref => 1,
    handles    => {
        add_article => 'push',
    },
    default    => sub{ [] },
);

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

    for my $article ( $self->articles ) {
        print "füge $article in ausgabe",
              "datei hinzu...\n";
    }
}

no Moose;
```

Listing 9

```
# [ ... Vererbung etc ... ]
around publish => sub {
    my ($code,$self,@arg) = @_;

    warn "around in Zeitschrift";
    $self->add_article( 'cover.jpg' );
    $self->$code( @arg );
};
# [ ... ]
```

Listing 10

In Ausgabe 16 von $foo wurde schon ausführlich beschrieben, welchen Einfluss die Reihenfolge der Methoden-Modifier auf die Reihenfolge der Ausführung hat. Hier soll nur exemplarisch an `around` gezeigt werden, dass es eine Rolle spielt, wann welche Rolle geladen wird.

Wenn die Rolle an sich keine Methoden-Modifier benutzen, ist es natürlich egal, da das somit keine Methodenaufrufe verändert.

Die Klasse "Publikation" hat eine Methode `publish` (Listing 9). In der Klasse "Zeitschrift" wird mit `around` gearbeitet, um eine Seitengröße für das "publish" festzulegen (Listing 10).

Die Rolle "PDF" arbeitet aber auch mit `around`, um noch ein Cover zu der Zeitschrift hinzuzufügen:

```
use Moose::Role;

around 'publish' => sub {
    my ($code,$self,@arg) = @_;

    warn "around in PDF";
    $self->format( 'PDF' );

    $self->$code( @arg );
};

no Moose::Role;
```

Als erster Fall soll hier gezeigt werden, was passiert, wenn die Rolle am Anfang der Klasse eingebunden wird.

```
use Moose;
extends 'Publikation';
with 'PDF';

around publish => sub { ... }
```

Die Ausgabe für diesen Fall ist in Listing 11 zu sehen.

Da `around` in umgekehrter Definitions-Reihenfolge ausgeführt wird, taucht hier zuerst das `around` aus der Klasse "Zeitschrift" auf und danach erst das `around` aus der Rolle.

Der zweite Fall ist, dass die Rolle erst nach dem `around` eingebunden wird (Listing 12). Die Ausgabe dazu ist in Listing 13 zu finden.

```
around in Zeitschrift at Zeitschrift.pm line 10.
around in PDF at PDF.pm line 10.
füge article1 in ausgabedatei hinzu...
füge article2 in ausgabedatei hinzu...
füge cover.jpg in ausgabedatei hinzu...
```

Listing 11

```
use Moose;
extends 'Publikation';

around publish => sub { ... }

with 'PDF';
```

Listing 12

```
around in PDF at PDF.pm line 10.
around in Zeitschrift at Zeitschrift2.pm line 9.
füge article1 in ausgabedatei hinzu...
füge article2 in ausgabedatei hinzu...
füge cover.jpg in ausgabedatei hinzu...
```

Listing 13

Hier ist die Reihenfolge genau umgekehrt. Dass das eine Rolle spielt, sollte man immer im Hinterkopf haben, damit man von diesem Verhalten nicht überrascht wird. In der Regel wird man die Rollen am Anfang der Klasse einbinden. Wenn aber Rollen dynamisch angewendet werden, sind die Methoden-Modifier ja erst nachträglich hinzugekommen. Im Fall von `around` werden diese also zuerst ausgeführt.

To BUILD or not to BUILD?

Die Methode `BUILD` wird bei Moose direkt nach dem Erzeugen eines Objekts ausgeführt. Im Gegensatz zur normalen Reihenfolge, werden die `BUILD`-Methoden von der Elternklasse bis zur Kindklasse ausgeführt. Für jede Klasse gibt es nur eine `BUILD`-Methode. Das kann zu Irritationen führen, wenn Rollen etwas direkt nach der Objekterzeugung ausführen möchten und dazu eine `BUILD`-Methode implementieren.

Wenn Rollen eine solche BUILD-Methode implementieren, darf die Klasse an sich keine BUILD-Methode haben. Sonst wird die Methode der Rolle einfach ignoriert.

Da Moose keine Möglichkeit bietet, Klassen bestimmte Methodennamen zu verbieten, ist die Verwendung von `BUILD` in Rollen nicht zu empfehlen.

An verschiedenen Stellen wird darauf hingewiesen, dass man in der Rolle mit Methoden-Modifiern arbeiten kann und einfach

```
after BUILD => sub {
    # ... Code ...
};
```

benutzen kann. Das bringt aber Probleme mit sich, wenn die Klasse keine BUILD-Methode hat. Als Lösung könnte man einfach eine leere BUILD-Methode in der Rolle haben:

```
sub BUILD {}
after BUILD => sub {
    # ... Code ...
};
```

Das funktioniert so lange, bis eine weitere Rolle den gleichen Trick anwendet. Wie weiter oben schon gezeigt wurde, werden solche Methodenkonflikte nicht automatisch aufgelöst.

Deshalb ist in so einem Fall das geschickteste, eine `BUILD`-Methode in der Klasse zu fordern - mittels `requires`.

Nützliche Rollen auf CPAN

Es gibt wirklich viele nützliche Rollen und viele davon sind nicht auf ein spezifisches Problem beschränkt, sondern sind für viele Anwendungen interessant. Einige solcher Rollen sind schon auf dem CPAN vorhanden und wirklich nützliche Rollen werden in den folgenden Abschnitten kurz vorgestellt:

MooseX::LogDispatch und MooseX::Log::Log4perl

Es empfiehlt sich eigentlich immer, Logging in die Anwendungen einzubauen. Das erleichtert die Fehlersuche enorm. So ein Logging-Verhalten lässt sich gut mit Rollen umsetzen. Auf CPAN gibt es mit `MooseX::Log::Log4perl` und `MooseX::LogDispatch` zwei solcher Rollen, die mit den bekanntesten Logging-Modulen umgehen können.

Aus der Dokumentation von `MooseX::Log::Log4perl` stammt Listing 14.

```
package MyApp;
use Moose;
use Log::Log4perl qw(:easy);

with 'MooseX::Log::Log4perl';

BEGIN {
  Log::Log4perl->easy_init();
}

sub foo {
  my ($self) = @_;
  $self->log->debug("started bar");    ### logs with default class catergory "MyApp"
  ...
  $self->log('special')->info('bar');  ### logs with category special
}
```

Listing 14

MooseX::Getopt

Kommandozeilenprogramme erlauben häufig die Angabe von Parametern. In Nicht-Moose-Anwendungen kommt dafür sehr oft `Getopt::Long` zum Einsatz. Für Moose-Anwendungen gibt es auf CPAN `MooseX::Getopt`. Die Parameter werden dabei zu Attributen der Moose-Klasse (Listing 15) und im Skript wird nicht mit `new` ein neues Objekt der Klasse erzeugt, sondern mit `new_with_options` (Listing 16).

```
package OPMAnalyzer;

use Moose;
with 'MooseX::Getopt';

has opm => ( is => 'ro', isa => 'Str' );

no Moose;

1;
```

Listing 15

```
use OPMAnalyzer;

my $app = OPMAnalyzer->new_with_options;
print $app->opm;
```

Listing 16

```
C:\>perl script.pl
Use of uninitialized value in print \
   at script.pl line 8.

C:\>perl script.pl --opm test.opm
test.opm
```

MooseX::SimpleConfig

Sehr viele Anwendungen erlauben Einstellungen in Konfigurationsdateien. Um Moose-Anwendungen mit Konfigurationsdateien auszustatten, kann `MooseX::SimpleConfig` genommen werden. Das verwendet `Config::Any` und erlaubt so verschiedene Konfigurationsformate. Auch hier wird das Objekt dann nicht mit `new` erzeugt. Hier steht dann `new_with_config` zur Verfügung.

Die Konfigurationsparameter müssen sich in der Klasse als Attribute wiederfinden. Hier ist darauf zu achten, dass der richtige Typ genommen wird.

Als Beispiel dient die folgende Konfigurationsdatei:

```
---
opm: test.opm
mailopt:
  smtp: host
  user: name
```

opm ist ein einfacher String und *mailopt* ist eine Hashreferenz. Dementsprechend sieht die Klasse folgendermaßen aus:

```
use Moose;
with 'MooseX::SimpleConfig';

has opm => ( is => 'ro', isa => 'Str' );
has mailopt => (
  is => 'ro',
  isa => 'HashRef',
);

no Moose;
```

Das Skript erzeugt ein Objekt - wie schon erwähnt - mit `new_with_config`. Der Parametername `configfile` ist dabei durch die Rolle vorgegeben.

```
my $app = OPMAnalyzer2->new_with_config(
    configfile => 'test.yml',
);
print $app->opm, " -> ",
      $app->mailopt->{smtp};
```

Diese Rolle spielt auch mit der Rolle `MooseX::Getopt` zusammen. In der Klasse wird als zusätzliche Rolle noch `MooseX::Getopt` eingebunden und dann kann das Objekt mit `new_with_options` erzeugt werden. Wenn dann das Skript mit

```
C:\>perl script.pl --configfile test.yml
```

aufgerufen wird, wird das `configfile` an die Rolle weitergegeben und die Konfigurationsdatei wird geparst.

Data::Serializable

Wenn Daten serialisiert werden müssen - z.B. mit JSON oder Storable - ist `Data::Serializable` die richtige Rolle. Über das Attribut `serializer_module` kann das Modul eingestellt werden, mit dem die Daten serialisiert werden sollen. In dem Beispiel wird `JSON` genommen.

```
use Moose;
with 'Data::Serializable';

has attr1 => (
    is => 'ro',
);

sub obj2str {
    my $self = shift;
    my %attr = (
        attr1 => $self->attr1,
    );
    $self->serializer_module( 'JSON' );
    $self->serialize( \%attr );
}

no Moose;
```

Hier wird in der Methode das Modul festgelegt. Man könnte stattdessen aber auch bei der Objekterzeugung den Parameter `serializer_module` übergeben und das Modul so festlegen.

Traits

Zur Erinnerung: Traits sind das gleiche wie Rollen. Nur werden sie auf einzelne Instanzen angewendet und nicht auf komplette Klassen. Die folgenden Abschnitte zeigen, wo in Moose Traits vorkommen können und wie eigene Traits entwickelt werden können.

Traits kommen bei Moose an verschiedenen Stellen vor. Die erste Gelegenheit, bei der man mit Traits in Berührung kommt, sind wohl Traits, mit denen man das Verhalten von Attributen beeinflussen kann. Hier gibt es auch schon einige vorgefertigte Traits, die zum Einsatz kommen, wenn Attribute z.B. Hash- oder Arrayreferenzen sind.

Wenn für Attribute Traits angewendet werden sollen, werden diese bei der Attributdefinition angegeben (Listing 17).

```
has 'where' => (
    traits  => ['Array'],
    is      => 'bare',
    isa     => ArrayRef,
    default => sub { [] },
    handles => {
        add_where  => 'push',
        has_where  => 'count',
        last_where => [ 'get', -1 ],
        where      => 'elements',
    },
);
```

Listing 17

In dem Beispiel ist das Attribut *where* eine Arrayreferenz. Für dieses Attribut wird das Trait `Moose::...` verwendet. Über dieses Trait werden die "typischen" Arrayfunktionen bereitgestellt, die über `handles` deligiert werden können. Die wichtige Aussage über Traits ist hier, dass das Array-Trait ein bestimmtes Verhalten für das Attribut bereitstellt.

Im Gegensatz zu einer Rolle sind die Methoden aber - ohne die Einstellungen über `handles` - nicht direkt über das Attribut erreichbar.

```
has 'where' => (
    traits  => ['Array'],
    is      => 'bare',
    isa     => ArrayRef,
    default => sub { [] },
);
```

Es gibt hier keine Möglichkeit, die Methode `count` aufzurufen, man muss den Weg über `handles` gehen.

Eigene Traits

Neben den nativen Traits kann man natürlich auch eigene Traits schreiben. Damit lässt sich die Mächtigkeit von Attributen noch weiter steigern. Im folgenden soll ein Trait geschrieben werden, mit dem Attribute mit einem Label versehen werden können.

```
package AttritbuteLabel;
use Moose::Role;
has 'label' => (
    isa => 'Str',
    is  => 'rw',
);
```

Wie man sieht, ist es eine ganz normale Rolle.

Ein einfaches Beispiel für eine Moose-Klasse, die dieses Trait verwendet, sieht wie folgt aus.

```
package MyClass;

use Moose;
 has 'foo' => (
    isa    => 'Str',
    is     => 'ro',
    traits => [qw/AttritbuteLabel/],
);
```

Jetzt gibt es zwei Wege, wie man das Label für das Attribut setzen kann: Erstens bei der Definition des Attributs - wenn man das Label schon kennt - und zweitens über die Metaklasse.

Soll das Label schon bei der Attributdefinition gesetzt werden, sieht das wie folgt aus:

```
has 'foo' => (
    isa    => 'Str',
    is     => 'ro',
    traits => [qw/AttritbuteLabel/],
    label  => 'Testattribut',
);
```

Der zweite Weg geht über die Metaklasse der eigenen Moose-Klasse. Danach holt man sich das Objekt des Attributs und kann für dieses Objekt des Attributs das Label gesetzt werden:

```
use MyClass;

my $c = MyClass->new;
$c->meta->get_attribute( "foo" )
    ->label( 'NeuesLabel' );
```

Wofür man das produktiv einsetzen kann, wird im nächsten Teil des Moose-Tutorials gezeigt.

Traits - nicht nur für Attribute...

Bis jetzt wurden Traits nur für Attribute angeschaut. Man kann diese aber auch auf andere Moose-Klassen anwenden. Im folgenden Beispiel soll ein Trait erstellt werden, mit dem der Status eines Objekts gespeichert werden soll.

```
package MyClass;

use Moose '-traits' => 'ObjectStatus';

has foo => (
    is => 'rw',
    isa => 'Str',
    trigger => sub{
        my ($self) = @_;
        $self->meta->status( 'changed' );
    },
);

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

    $self->meta->status( 'new' );
}

no Moose;

1;
```

Jetzt sollte man sich aber nicht wundern, dass das Objekt die Methode `Status` nicht kennt: Genau wie bei Traits für Attribute, werden die Traits auf die Metaklasse angewendet. Soll die Methode aufgerufen werden, muss das über den Umweg der Meta-Klasse passieren:

```
use MyClass;

my $obj = MyClass->new;
print $obj->meta->status,"\n";

$obj->foo( 'test' );
print $obj->meta->status,"\n";
```

Der Vorteil von (Klassen-)Traits gegenüber Rollen ist, dass man sich mit Methoden und Attributen, die die Klasse der Allgemeinheit zur Verfügung stellt, nicht in die Quere kommt. Man kann so sehr gut Meta-Informationen zu der Klasse bzw. dem Objekt speichern.

Ausblick

In ein paar Absätzen wurden die Meta-Klassen schon beiläufig erwähnt. Moose und die Objekte sind eine Sammlung vieler Klassen und diese Klassen beschreiben wie Moose-Klassen aussehen. In der nächsten Ausgabe sind diese Meta-Klassen Thema des Moose-Tutorials. Es wird gezeigt, wie man sehr viele Informationen über Objekte und Klassen bekommen kann.

Herbert Breunung

WxPerl Tutorial - Teil 7: Gestaltung komplexerer Programme

Nach dem sechsten Teil, der den Bereich der Grundlagen verließ und das Wissen bot, um erste, vollständige Anwendungen zu schreiben, die ein Menü, Werkzeugleiste und Statuszeile haben, geht es in diesem und nächsten Teil um weniger essentielle Zutaten einer Applikation, die aber dennoch bei größeren und reichhaltigeren Apps nicht fehlen dürfen: Zwischenablage, Drag'n Drop, Mauscursor, Popups, Drucken, Interfacesprache, Validatoren und Logging. Dabei wird auch die Frage berührt, wie verständliche Bedienoberflächen gestaltet sein sollten. Der darauf folgende Teil wird spezielle, meist sehr mächtige Widgets behandeln wie Schieberegler, moderne Sucheingaben, excelartige Tabellen und Textfelder mit Syntaxhervorhebung.

Gute Gestaltung

Auch wenn Design ein sehr breites Feld ist und eigener Geschmack immer einen Anteil dabei hat, so ist es doch sehr nützlich einige Fragen zu beantworten, bevor eine Bedienoberfläche entworfen wird. Das umfasst zum einen die angestrebte "Zielgruppe" woraus sich der "Führungsbedarf" ergibt. Aber noch wichtiger ist der Arbeitsablauf mit dem Nutzer ihre Aufgaben lösen. Wie in der physischen Welt sollte es Räume geben, die auf eine Aufgabe spezialisiert sind, und einen klaren und einfachen Weg, um sie zu wechseln.

Solche Räume können Fenster, Dialoge oder auch Flächen einer Reiterleiste oder anderer **BookCtrl* (siehe Folge 4) sein. Sie lassen sich ebenso durch Splitter (Folge 4) oder zusätzliche Werkzeugleisten (Folge 6) einblenden. Aber auch kreative Lösungen, wie beim KDE-Spiel Palapeli können sinnvoll sein, wo Reiterleiste und Hauptmenü zu einer dominierenden Hauptnavigationseinheit vereint wurden. (Wie man eigene Widgets erzeugt wird Thema der Folge 11.)

Die Auswahl, was in einen Interaktionsraum gehört, ähnelt dem Entwurf von Klassen, wo man auch erst nach einiger Erfahrung erkennt, welche Attribute und Methoden am besten wohin gehören. Trotzdem sollte es bereits bei der ersten, groben Planung entschieden werden; Einmal um das Zusammenspiel mit den Daten und der Logik zu klären (saubere MVC-Trennung), aber auch weil das Umgestalten von GUI aufwendig und mühsam ist. Um dieser Zwickmühle zu entkommen, kann man entweder die Oberfläche mit einem Programm wie *wxFormbuilder* oder *Dialog::Blocks* bauen, wo man gleichzeitig am schnellsten fast alle Optionen überblicken und testen kann (genaueres dazu in Teil 10), oder man schreibt einen Prototyp, der nur einen Teil der späteren Oberfläche umfasst. Die mit *WxFormBuilder* erstellten Fenster können wie eine gestartete Applikation angezeigt werden. Dadurch lässt sich die Optik unabhängig vom Zustand des Programms zuverlässig prüfen. Aber das ersetzt oft nicht einen arbeitenden Prototypen, dessen Funktionalität von den Modulen bereitgestellt wird, die ebenfalls Teil der Anwendung werden.

Um möglichst viel Code des Prototypen wiederverwenden zu können, kapsel ich Panel und Widgets in Klassen (siehe nächstem Abschnitt) und trenne das Erstellen der Widgets und deren optische Zusammenstellung in zwei Abschnitte. So bleibt die Fütterung der Widgets mit Defaultwerten aus den Datenmodulen an einem Fleck. Auch die Optimierung der Größen und Abstände ist einfacher, wenn sie auf einer Bildschirmseite überblickbar sind. Selbst ohne bunte Graphiken lässt sich mit ausgefeilten Proportionen von Größen und Abständen ein schöner und entspannender Anblick erzeugen. Das dazu erforderliche Herumprobieren wird minimiert, wenn man bedeutungsgleiche Abstände in Variablen mit verständlichen Namen packt.

Gewünschte Verhältnisse sollten ebenso mit Code formuliert werden und nicht immer wieder händisch nachgestellt werden. Braucht ein Widget 5 Pixel zusätzlichen Abstand, kann er mit `$sizer->AddSpacer(5);` oder

```
$sizer->Add(
    $widget,
    0,
    $ausrichtung,
    $rand+5,
);
```

eingefügt werden, ohne die Grundordnung aufzuheben.

Die Zeilen werden so auch kürzer, lesbarer und die Stellschrauben heben sich so ab. Die Wahl der Alternative hängt natürlich vom Inhalt der Variable `$ausrichtung` ab. In einer üblichen Anwendung wäre sie gefüllt mit:

```
$ausrichtung =
   wxLEFT | wxALIGN_CENTER_VERTICAL;
```

Zeilen mit unterschiedlichen Widgets sehen meist am besten aus, wenn sie vertikal mittig zentriert werden (`wxALIGN_CENTER_VERTICAL`). `wxLEFT` bevorzuge ich vor `wxALL`, weil ich meist, entsprechend der menschlichen Lesegewohnheit, die Anordnung zeilenweise von links nach rechts aufbaue. (Tabellendesigns sehen meist eintönig aus.)

Dennoch ist es ansprechender wenn die Abstände zwischen den Zeilen, den Abständen zwischen den Widgets gleichen. Bei gleichzeitiger Verwendung eine Variable für den Mindestabstand und `wxALL` wäre der Abstand zwischen den Widgets doppelt so groß wie zum Rand des Panels oder Fensters. Selbst bei mehreren durch eine `StaticLine` oder `StaticBox` geteilte Bereiche sieht das in der Regel nicht vorteilhaft aus.

Lebe im Moment

Bei der konkret-inhaltlichen Gestaltung ist diese esoterisch klingende Weisheit entscheidend. Schlanke Oberflächen wie bei Google oder Facebook zeigen den Weg und waren erfolgreich weil sie sich auf das Wesentliche konzentrieren (alles Ablenkende weglassen) und mit allen verfügbaren Informationen den Nutzer im Moment der Eingabe unterstützen. Die folgenden Absätze werden die Techniken und Wx-Objekte vorstellen um dies umzusetzen, aber mir ist wichtig, alle diese Elemente als Ausdruck einer Idee zu verstehen: Dem Nutzer jeden Aufwand abzunehmen - den Aufwand zu suchen, zu lesen, etwas doppelt einzugeben oder zu klicken, ja sogar etwas zu verstehen, wenn es für ihn nicht unbedingt notwendig ist.

Gerade weil Programmierer eher in Datenstrukturen leben, ist es gut, so früh wie möglich Nutzer oder Tester zu haben, was auch hinter dem Open-Source-Mantra steht: "release often and release early".

Erstkontakt

Ist jetzt alles Ablenkende im Hintergrund oder mit `$widget->Show(0)` und anschließendem `$sizer->Layout` ausgeblendet oder wenigstens durch `$widget->Enable(0)` eingegraut, kann der Anwender sich dennoch nicht im Klaren sein, was von ihm verlangt wird.

Verlagert sich der Fokus auf eine Texteingabe, kann auf `EVT_SET_FOCUS` reagiert werden und auf die gleiche Weise ein *Label* (`Wx::StaticText`) eingeblendet werden, das ein Beispiel zeigt. Die simpelste Art ihm zu helfen, die auch bei jedem Widget funktioniert, ist ein `$widget->SetToolTip($text);`. Der Text erscheint neben dem Mauscursor, sobald dieser für einige Zeit über dem Widget stehen bleibt.

Doch Bilder sind aussagekräftiger als Text, weshalb selbstgebaute Popups sehr wirkungsvoll sein können. Popups sind in Wx randlose und knopflose fensterartige Flächen, die intern für Kontextmenüs, Splashscreens und Tooltips verwendet werden. Sie lassen sich, wie jedes andere Widget, mithilfe einer `PaintDC` frei "bemalen".

DC

Diese Abkürzung steht für "draw context" und ist der Name des Wx-Malkastens. Er ist nicht ganz so mächtig wie *GD* oder *Image::Magick*, reicht aber für sehr viele Anwendungsfälle völlig aus. Man bekommt Stifte und Pinsel in allen definierbaren Farben (`Wx::Colour`) und Formen (`Wx::Bitmap`). Es gibt dort Zeichenbefehle (`Draw*`) von Linie bis Polygon, Ellipsenstück oder rotiertem Text.

Rechteckige Regionen lassen sich ausschneiden, mit logischen Operationen überlagern oder als Füllung benutzen. Wichtig ist nur zu beachten, welcher Pinsel, welche Maßeinheit oder welches Koordinatensystem gerade verwendet wird. Auch werden Textfarben unabhängig von Pinselfarben gesetzt.

Eine `PaintDC` wird meist dazu verwendet, einen Teil eines Fensters zu bemalen, denn es reagiert auf das Ereignis "das Fenster muss neu gezeichnet werden", welches `Wx::PaintEvent` heißt und mit `EVT_PAINT` an die Zeichenroutine angebunden wird.

Der erste Schritt zum Popup ist jedoch ein `package`, das von `Wx::PopupWindow` erbt (Listing 1).

Die Umlaute im String machen keine Probleme, solange die Datei auch als UTF abgespeichert ist. Text in obskuren Windows-Kodierungen wird von Wx ignoriert. Von dem Ellipsenbogen ist nur der Außenrand zu sehen, da die Füllfarbe vom `Brush` bestimmt wird, der auf Hintergrundgrün gesetzt war. Die vorletzte Zeile schließt das Fenster bei Linksklick. Das Popup wird bequem von außen gesteuert:

```
my $popup = Wx::DemoPopup->new( $frame );
$popup->Move( $x, $y );
$popup->Show;
```

wobei eine eigene Zeitsteuerung her muss. Timerevents wurden aber bereits im letzten Teil behandelt. Nur die Popup-Koordinaten beim `Move` verdienen besondere Aufmerksamkeit. Es sind absolute Bildschirmwerte, welche sich aus der Summe der Fensterkoordinaten (`$frame->GetPosition`), Koordinaten des Widgets im Fenster (`$widget->GetPosition`) und der Koordinaten des Cursor über dem Widget ergeben.

Letzteres gibt `$MouseEvent->GetPosition` ab und muss während `EVT_MOTION` abgefragt werden, da es kein Ereignis bei Ende der Mausbewegung gibt.

```
my $timer = Wx::Timer->new($frame, $TimerID);
my $cpos;
EVT_ENTER_WINDOW($widget, sub {
    $timer->Start(2000,1)
  });
EVT_LEAVE_WINDOW($widget, sub {
    $timer->Stop()
  });
EVT_MOTION( $widget, sub {
    $cpos = $_[1]->GetPosition
  });
EVT_TIMER ($frame, $TimerID, sub {
    my ($widget, $event) = @_;
    my $fpos = $frame->GetPosition;
    my $wpos = $$widget->GetPosition;
    $popup->Move(
        $fpos->x + $wpos->x + $cpos->x,
        $fpos->y + $wpos->y + $cpos->y
    );
    $popup->Show;
} );
```

```
package Wx::DemoPopup;
use base qw(Wx::PopupWindow);

sub new {
    my( $class, @args ) = @_;
    my $self = $class->SUPER::new( @args );
    # Höhe und Breite
    $self->SetSize( 300, 200 );
    EVT_PAINT( $self, sub {
        my( $self, $event ) = @_;
        my $dc = Wx::PaintDC->new( $self );
        my $orange = Wx::Colour->new( 182, 192, 0 );

        $dc->Clear();
        $dc->SetBrush( Wx::Brush->new( &Wx::wxGREEN, &Wx::wxSOLID ) );
        # grünes Rechteck als Hintergrund
        $dc->DrawRectangle( 0, 0, $self->GetSize->x, $self->GetSize->y );
        $dc->SetTextForeground( &Wx::wxBLUE );
        # 3 Grad nach links geneigter Text
        $dc->DrawRotatedText( "schräg", 30, 40, 3 );
        $dc->SetPen( Wx::Pen->new( $orange, 1, &Wx::wxSOLID ) );
        # Kreisbogen von 30 Grad bis 130
        # auf einer 120x120 grossen "Ellipse"
        $dc->DrawEllipticArc( 90, 90, 120, 120, 30, 130 );
    } );
    EVT_LEFT_DOWN( $self, sub {$self->Hide });
    return $self;
}
```

Listing 1

Das Fenster wird sich dennoch wahrscheinlich nicht am Cursor öffnen, weil in dieser Rechnung Menüleiste, Werkzeugleiste und Titelleiste nicht enthalten sind. Aber letztlich dient es der Bequemlichkeit, da es so leichter weggeklickt werden kann (`EVT_LEFT_DOWN`)

Hilfe

Aber manchmal ist selbst ein Bild nicht genug. Dann muss es echte Dokumentation sein, in der ungern gesucht wird. Dazu kennt `Wx` `Wx::Help`, welches nicht nur die bekannten Doku-Dialoge (`Wx::HtmlHelpController`) hat, einen Weg, diese an der richtigen Stelle aufgeblättert zu öffnen. Dieses Konzept funktioniert aber ganz anders als ein Tip. Windowsnutzer mögen noch aus vergangenen Tagen den letzten Knopf einer Werkzeugleiste kennen, auf dem ein gelbes Fragezeichen prangte.

Wenn man diesen drückte, verwandelte sich der Cursor in ein Fragezeichen und man konnte auf das Widget klicken, zu dem man etwas wissen wollte. So einen Knopf bekommt man dieser Tage mit

```
use Wx::Help;
Wx::ContextHelpButton->new
    ( $panel, &Wx::wxID_CONTEXT_HELP );
```

und er öffnet erst einmal den Tooltip, der mit

```
$widget->SetHelpText( $text );
```

festgelegt wurde. Er lässt sich auch per `$bar->AddControl` in eine Werkzeugleiste heben. Hat man gänzlich andere Pläne, so reicht es den Klick per

```
EVT_HELP($widget, -1, sub {...});
```

abzufangen. Wenn dabei auf `$event->Skip` verzichtet wurde, erscheint auch kein eventuell definierter Tooltip. Manch einer mag sich über den sonderbaren Parameter "Name" gewundert haben, den man - meist zuletzt - bei der Erzeugung jedes Widgets angeben kann. Hier machte er sich zum ersten Mal für mich nützlich, denn ich konnte mit `$widget->SetName( $index );` an jedes Element eine weitere Information heften, nämlich einen Index der Doku.

Um den Rahmen dieses Tutorials nicht zu überdehnen, wird das Format des `Wx::HtmlHelpController` nicht in jeder Einzelheit erläutert. Grundlegend besteht jede Seite aus abgespecktem HTML, denn für die Anzeige wird `Wx::HTML` verwendet.

Zwei weitere HTML-Seiten legen das Inhaltsverzeichnis und den Suchindex fest, eine dritte Datei im ".ini"-Format den einzelnen Dateien die ID und Funktion zu. Das fertige Verzeichnis wird per *ZIP* oder *CHM* gepackt, fertig ist die Dokumentationsdatei. Hat man `Wx::Demo` installiert so hilft ein Blick nach `../site_perl/../Wx/DemoModules/files/help`. Diese Datei ist schnell geladen:

```
$doku = Wx::HtmlHelpController->new(
    wxHF_FLATTOOLBAR | wxHF_DEFAULTSTYLE
);
$doku->AddBook( $dateipfad, 1 );
```

Und nun reicht ein

```
EVT_HELP( $self, -1, sub {
  my $win = $_[1]->GetEventObject;
  $doku->DisplaySectionId($widget->GetName);
});
```

und die Dokumentation ist leichter erreichbar.

Validatoren

Anstatt dem Nutzer zu erklären was er eingeben soll, ist es oft cleverer ihm keine fehlerträchtigen Eingaben zu erlauben oder im Falle eines Fehlers sofort zu meckern, anstatt sich später heimtückisch zu rächen. Eine wichtige Methode ist es auch, ihm in Echtzeit das geschätzte Ergebnis seiner Wahl vor Augen zu führen. Die Ribbonbar im neueren MS-Word macht zum Wohle ihrer Nutzer davon ausgiebig Gebrauch.

Wer seine Kunden kennt, setzt die Defaultwerte am Besten so, dass oft gar keine Eingabe nötig ist.

Statt in einem Textfeld zum Beispiel das Geschlecht abzufragen, wäre es besser zwei kleine *Radiobutton* mit "Mann" und "Frau" in einer Zeile anzuordnen.

Das wäre noch besser als eine `ComboBox`, deren Bedienung einen Klick mehr verlangt und auch etwas mehr Unruhe erzeugt. Nur ein `Wx::Choice` wäre hier eine gute Alternative. Es erlaubt keine Texteingaben, sondern kann auf ganzer Fläche mit der Maus aufgeklappt werden. Auch hier ist mit einem "langem Klick" die Wahl absolviert.

Wenn eine Zahl zwischen 5 und 105 gefragt ist und 50 ein guter Default ist, empfiehlt sich auch keine Eingabe per Tastatur, sondern

```
Wx::Slider->new(
    $panel, -1, 50, 5, 105,
    [-1,-1],[-1,-1],
    wxSL_LABELS | wxSL_HORIZONTAL
);
```

Denn ein langer Klick ist für viele weit weniger Aufwand. Auch ein `SpinCtrl` ist unpraktisch, wenn für jedes Inkrement einmal gedrückt werden muss werden muss. Ein Gedrückthalten lässt die Zahlen andererseits sehr schnell vorbeiziehen.

Dennoch schätzen manche die durch sie mögliche Wahl zwischen Tastatur- und Mauseingabe. Jedoch ist eine Tastaturkontrolle mit `EVT_KEY_DOWN( $slider, sub { ...` schnell geschrieben.

Durch `EVT_SCROLL` lässt sich nicht nur eine Echtzeit-Rückkopplung schneidern, sondern auch der Slider für nichtnumerische Aufgaben nutzen. Dazu muss zuerst die Konstante `wxSL_LABELS` weg und eine Arrayvariable mit der Liste aller möglichen Eingaben her. Dann kann bei `EVT_SCROLL` per Array der Index zum Beispiel in einen Tiernamen übersetzt werden, welchen die nebenstehende `Wx::TextCtrl` anzeigt.

Wx kennt keinen Unterschied zwischen einzeiligen und mehrzeiligen Texteingabefeldern. Diese unterscheiden sich nur in der Zugabe der Stilkonstante `wxTE_MULTILINE`.

`wxTE_PASSWORD` erzeugt Passworteingaben, die immer einzeilig sind.

Selbst wenn es eine Texteingabe sein muss, lassen sich Fehleingaben immer noch früh mit Hilfe von Validatoren abfangen. Mattia sei es gedankt, dass er für die Regex-träumenden Perlianer einen Textvalidator schuf, der nicht umständliche Konstanten wie `wxFILTER_ALPHA` benötigt, sondern wo ein

```
$textfeld->SetValidator(
    Wx::Perl::TextValidator->new('\d')
);
```

ausreicht, um die Eingabe von Nicht-Zahlen zu verhindern.

Validatoren können jedem Widget zugewiesen werden und selbstgeschriebene Validatoren (von `Wx::Validator` abgeleitete Klassen) können die Aufgabe übernehmen, beim Aktivieren einer `Checkbox` andere freizuschalten (`Enable`), weil bestimmte Kombinationen von Optionen ausgeschlossen sein müssen.

Localisation

Im vorangegangenem Teil deutete ich an, sämtliche Strings in einer geschachtelten Datenstruktur, kontextsensitiv zu speichern, was die Übersetzung und Handhabung vereinfacht. Diese Technik hat jedoch den entscheidenden Nachteil, dass Standarddialoge des Betriebssystems weiter die Zunge den Betriebsystems sprechen, selbst wenn das Programm auf kroatisch eingestellt ist. Wer das verhindern will und muss, hat keine Alternative zu den bekannten .po und .mo Dateien, die man mit dem Wx-Programm *PoEdit* erstellen kann. Aber auch Emacs und andere Editoren können mit *GNU gettext* umgehen.

Für viele gängige Sprachen bietet das C++-Wx jedoch selber solche Dateien an, die ständig freiwillige Übersetzer suchen (*http://www.wxwidgets.org/about/i18n.php*). Das ist eine einsteigerfreundliche Methode etwas zurückzugeben oder sich zu bedanken.

Um überhaupt dieses System nutzen zu können, muss eine App von Anfang an auf Mehrsprachigkeit ausgelegt werden. Dazu zählt das obligatorische:

```
use Wx::Locale( gettext => 't');
```

Außerdem muss jeder String, der übersetzt werden soll, in ein t('...') eingefasst werden.

Schritt zwei ist die Feststellung der aktuellen Sprache mit

```
Wx::Locale::GetSystemLanguage;
```

Das ergibt eine Zahl, die `wxLANGUAGE_GERMAN`, `Wx LANGUAGE_DEFAULT` oder einem von euch gewählten Wert entsprechen müsste. Auch die Auswahl der neuen Sprache geht über eine solche Konstante - siehe Listing 2.

```
# erzeuge Speicher für neue Übersetzung
my $locale = Wx::Locale->new($langid);
# gebe Verzeichnis mit .po & .mo Dateien an
$locale->AddCatalogLookupPathPrefix( $pfad );
# such zur ID passenden Namen der Sprache
my $langname = $locale->GetCanonicalName();
my $kurzname = $langname ? substr($langname,0,2) : 'en';
# laden der neuen Strings wenn Datei existiert
$locale->AddCatalog( $kurzname )
     if -f File::Spec->catfile($pfad, $kurzname.'mo');
unless ($locale->IsOk) { ....
```

Listing 2

Da Dateizugriffe viele Fehlerquellen haben können, sollte die letzte Zeile nicht fehlen. Falls der Parser einen Fehler im Format der .mo-Datei findet, kann dann immer auf die vorherige *Locale*-Einstellung zurückgegriffen werden.

Herbert Breunung

Rezension - Programmieren mit Stil

chromatic
Modern Perl
1. Auflage Oktober 2010
167 Seiten, englisch
Onyx Neon
ISBN 0-9779201-5-1
$35 USD/22.95 GBP

Robert C. Martin
Clean Code
Refactoring, Patterns, Testen und Techniken für sauberen Code
1. Auflage 2009, mitp
deutsche Übersetzung
480 S., Softcover
ISBN 978-3-8266-5548-7
€ 39,95

Modern Perl

Thomas Kappler hat in der letzten $foo "Effective Perl Programming" vorgestellt - ein Buch wie ein Dojo (Trainingsraum für japanischen Kampfsport), in dem man sein Detailwissen über Perl schärfen kann. "Modern Perl" von chromatic ist der dazu passende Anfängerkurs. Er ist für Menschen, denen das Wissen und die Erfahrung eines Perlmeisters fehlt, die aber gerne mitdenken und eifrig bei der Sache sind.

Und es ist erfreulich mit wie wenig fett der Text auskommt. Im ersten Absatz wird die Benutzung der `perldoc` erklärt und alles weggelassen, das man dort nachlesen könnte. Stattdessen wird dem Neuling das Warum und Wozu anhand von alltäglichen Beispielen erläutert.

Doch es beginnt mit der Philosophie, nach der Perl aufgebaut ist und zeigt Details, an welchen man diese erkennt. In den meisten allgemeinen Perlbüchern, die kein Kamel auf dem Deckblatt tragen, fehlt das leider.

Im folgenden Kapitel wird die Organisation der Gemeinschaft beschrieben, was auf Papier auch oft zu kurz kommt, denn Perl ist weit mehr als nur Bits und API's.

Der Großteil des Buches behandelt jedoch die Syntax, inklusive `Moose` und allem was bis 5.12 hinzukam. Dennoch bleibt das Werk unter 170 Seiten, was für meinen Geschmack an manchen Stellen zu knapp ist. Aber als Einstieg ist es auf den Punkt gebracht. Es klart zum Beispiel den Blick, was Operatoren eigentlich sind und welche Eigenschaften ihr Verhalten bestimmen. Es ist auch wohltuend, dass sich das Werk nicht auf Anfänger-Perl beschränkt, sondern den gesamten Umfang der Sprache zumindest umreißt, also auch `AUTOLOAD`, tie oder negative *look-ahead assertions* erwähnt. Selbst verbleibende Schwächen von Perl wie Prototypen und *Barewords* werden ehrlich benannt.

Ab Seite 115 widmet sich der Text noch einem weiteren Ziel: der Erziehung zu "gutem Stil". Eigentlich überrascht das niemanden, der chromatics gleichnamiges Modul (`Modern::Perl`) kennt, welches alle neuen Funktionen aktiviert (`use feature ...`), sowie ein `use strict;` und `use warnings;` ausführt. Im Buch geht es aber auch um Tests (viele Beispiele nutzen Test::More), einfaches Logging, Nutzung von `perltidy` und `Perl::Critic` sowie Handhabung von Projekten. Mir gefällt vor allem wie er aufzeigt, dass idiomatisches und sauberes Perl Hand in Hand gehen können und sollten. Natürlich wird chroma-

tics Haltung auch in der p5p-Liste kritisiert, da er zuweilen wenig Verständnis für Stimmen hat, die Funktionierendes nicht ändern wollen.

Um Anfängern den Weg zu weisen, ist die Schrift aber sehr geeignet. Immerhin haben auch rund 80 bekannte Namen (inklusive Larry Wall) an der Verbesserung mitgewirkt. Und die Aktivität im zugehörigen git-Archiv lässt auf weitere Editionen hoffen. Der Titel sollte aber bitte nicht mit dem französischen "Perl Moderne" verwechselt werden, das auch dieser Tage erschien und von drei für Kompetenz bekannten Autoren geschrieben wurde. Jedoch konnte ich es bisher nicht lesen.

Allison Randal (ehemalige TPF-Chefin) und chromatic (Betreiber von *perl.com*), beide arbeiteten auch für O'Reilly, verstehen Onyx Neon als Dienst für die Gemeinschaft, um wichtige Bücher zu produzieren, die keinen Verlag finden. Daher kann der geneigte Perl-Freund sie mit dem Kauf von "Modern Perl" unterstützen, auch wenn es offiziell als freies *PDF* ladbar ist.

Noch mehr Inhalte zu verwandten Themen befinden sich im zugehörigen Blog auf *http://www.modernperlbooks.com*.

Clean Code

Wer in Sachen "guter Stil" in die Tiefe gehen will, der greife zu "Clean Code" von "Uncle Bob". Dies ist ein Klassiker, den mitp 2009 ins Deutsche übersetzte. Selbst wer bereits den Kurs bei Sensei Con Wei ("Perl Best Practices" - Rezension in der nächsten $foo) belegte, kann hier noch seine Fähigkeiten erweitern (und das ohne eine Zeile Perl). Ähnlich "Modern Perl" wurde "Perl Best Practices" von jemandem geschrieben, der Perl verstanden hat und konkrete Anweisungen gibt. Robert C. Martin geht es mehr um das Warum, auch wenn er dabei nicht so unterhaltend oder dreist ist wie Damian Conway.

Manches, was er schreibt, klingt sogar beim ersten Lesen selbstverständlich. Auch die empfohlenen Regeln dämmern vielen denkenden Programmierern von allein. Dennoch ist es keines dieser voluminösen und schnell heruntergeschriebenen Programmierbücher, die eigentlich nur den Platz im Regal versperren.

In diesem Text bündeln sich die Erfahrungen vieler Jahre und Programmierer.

Es geht dem Autor nicht darum, revolutionäre Konzepte zu vermitteln, sondern dem Leser ins ständige Bewusstsein zu rufen, woran Softwareprojekte leiden und scheitern und an welchen Details er den Verfall der Quellen früh erkennen und beheben kann.

Auch Herr Martin vergleicht das mit dem ständigen Üben und Verbessern der Bewegungsabläufe im Kampfsport. Ziel ist bewusstes Handeln, dessen dritte und vierte Folge gewünscht ist.

Nach der Beschreibung seiner Haltung beginnt er mit den scheinbar kleinen Dingen, wie der Wahl der Namen von Variablen, Funktionen und Klassen, um sich über die Gestaltung von Funktionen, Kommentaren, Objekten und der Formatierung bis zum Entwurf von Klassen und Schnittstellen aufzuschwingen. Natürlich geht es dabei auch um die Themen Ausnahmebehandlung, Nebenläufigkeit, Tests, Refakturierung und Projektmanagement.

Der Hauptgrund, warum das Buch weit dicker als sein Kerninhalt ist, sind lange Quellabdrucke, in wortreichem Java. Darin sehe ich allerdings den Hauptvorteil gegenüber Damians Schule. Seine Ratschläge weisen selten über eine Zeile oder den Block hinaus. "Bob" zeigt anhand größerer Beispiele, die in mehreren Wachstumsstadien festgehalten sind, wie selbst gute Ansätze zu unwartbarem Code führen können. Gerade die in Kapitel 14 eingenommene Perspektive eines Schachkommentators, der darlegt, wann der Spieler welche Alternativen mit welchem Ausgang hätte überdenken sollen, macht das Buch meiner Ansicht nach besonders wertvoll.

Der Text ist relativ nüchtern geschrieben, aber da der Schreiber kompetent ist und sich selbst auch nicht zu ernst nimmt, fällt das Lesen leicht.

Ein schöner Nebeneffekt dieses Werkes war für mich auch, in witzigen Zitaten und kurzen Beschreibungen einen guten Teil der "Best Practices"-Szene im Vorbeigehen etwas kennenzulernen. Auch die schlichten, aber individuellen Bleistiftzeichnungen, mit welchen jedes Kapitel beginnt, tragen zur Unterhaltung und zu manchem Schmunzeln bei.

TPF News

Grant Update: Fixing Perl5 Core Bugs

Ein kurzes Update zum Grant von Dave Mitchell: Im Januar hat Mitchell knapp über 42 Stunden an seinem Grant gearbeitet. Ein Großteil dieser Zeit wurde für die Untersuchung von Sicherheitslücken bei Benutzerdefinierten `\p{}`-Eigenschaften aufgewendet. 4 Bugs hat er im Januar gefixt, darunter 2 Bugs bezüglich "overload".

Für den Grant stehen noch 315 Stunden aus.

Bewerbung um Hague-Grant: Structured Error Messages

Moritz Lenz hat einen Antrag für einen Hague-Grant eingereicht: Er möchte strukturierte Fehlermeldungen planen und umsetzen. Damit soll ein System geschaffen werden, mit dem sehr einfach Fehlermeldungen von Compiler, Laufzeitfehler und eigene Fehler einfach identifiziert und klassifiziert werden sollen.

Dem Antrag wurde mittlerweile stattgegeben.

Keine Grants in Quartal 1 / 2011

Da der Vorstand der Perl Foundation dem "Grants Committee" nur ein begrenztes Budget für 2011 zugewiesen hat, wird der "Call for Proposals" für das erste Quartal 2011 ausgesetzt. Die Perl Foundation hat kein regelmäßiges Einkommen und ist auf Spenden angewiesen.

Grant Update: Perl 6 Tablets

Herbert Breunung hat an den Perl 6 Tablets gearbeitet. Die Tablets zu Einführung und Sprachdesign sind fertig und weitere Tablets stehen kurz vor der Fertigstellung.

Breunung bittet um Review der bestehenden Tablets, die unter http://www.perlfoundation.org/perl6/index.cgi?perl_6_tablets erreichbar sind.

Grant-Abschluss: Handbuch für die Spieleentwicklung mit SDL

Der Grant "Handbuch für die Spieleentwicklung mit SDL" wurde abgeschlossen. Während des Grants hat Kartik Thakore den Code von SDL aufgeräumt und Memory Leaks gefixt.

10 der 11 versprochenen Kapitel des Handbuchs wurden geschrieben. Das 11. Kapitel - "Profiling" - wurde mangels Inhalt gestrichen. Dafür gibt es drei zusätzliche Kapitel.

Das Handbuch ist unter https://github.com/PerlGameDev/SDL_Manual/blob/master/dist/SDL_Manual.pdf zu finden.

OPAR

OPAR- wenn OTRS nicht reicht

http://opar.perl-services.de

CPAN News XVIII

Algorithm::CheckDigits

Dieses Modul ist genau das Richtige für Programmierer, die mehreren Prüfsummen in ihren Programmen benötigen. Buchhändler könnten z.B. an Prüfsummen für ISBN und Kreditkarten interessiert sein. Die Dokumentation ist verbesserungswürdig, aber es ist trotzdem ein nützliches Modul.

```
use Algorithm::CheckDigits;

my $isbn = '978-3-423-24787-0';
my $visa = '4111 1111 1111 1111';

my $isbn_obj = CheckDigits( 'ISBN13' );
my $visa_obj = CheckDigits( 'visa' );

if (
    $isbn_obj->is_valid( $isbn ) &&
    $visa_obj->is_valid( $visa ) ) {
    print "Sie haben das Buch gekauft...\n";
}
```

namespace::autoclean

Vor ein paar Ausgaben gab es einen ausführlichen Artikel über `namespace::clean`. Welche Funktionen dort aus dem Namensraum gelöscht wurden, hängt von der Reihenfolge ab, in der Pakete eingebunden werden und wo das Pragma geladen wird. `namespace::autoclean` behebt diese Schwäche. Außerdem werden nur Funktionen gelöscht und keine Methoden.

```
package Foo;
use namespace::autoclean;
use Some::Package qw/imported_function/;

sub bar { imported_function('stuff') }

# later on:
Foo->bar;               # works

Foo->imported_function; # will fail.
# imported_function got cleaned after
# compilation
```

Map::Tube

Der nächste London Perl-Workshop kann kommen - mit `Map::Tube` existiert eine Schnittstelle zum U-Bahn-Plan von London. Damit lässt sich die kürzeste Route berechnen und die Linien einer Haltestelle herausfinden. Jetzt fehlt nur noch der Fahrplan...

```
use Map::Tube;

my $map = Map::Tube->new();
my @route = $map->get_shortest_route(
                    'Bond Street', 'Euston');
my @lines = $map->get_tube_lines(
                    'Wembley Park');
```

CSS::LESSp

Mit LESS kann man sich die Entwicklung von CSS vereinfachen: Man kann Variablen und Funktionen definieren, so dass erstmal weniger Schreibarbeit notwendig ist, um das CSS zu definieren. Damit kann der Browser aber nicht umgehen. Diese LESS-Dateien müssen in normales CSS umgewandelt werden. In Perl kann man das mit `CSS::LESSp` machen...

```
use CSS::LESSp;

my $less = q~
  @logo: #457301;
  h3 { color: @logo; }
  #sale { border: 1px solid @logo; }
  .alert { background: @logo; }
~;

my @css = CSS::LESSp->parse( $less );
print "@css";
__END__
h3 {  color: #457301;  }
#sale {  border: 1px solid #457301;  }
.alert {  background: #457301;  }
```

Devel::InPackage

Zu welchem `package` gehört eine Zeile einer Datei? Diese Frage lässt sich jetzt mit `Devel::InPackage` beantworten. Noch hat das Modul die eine oder andere Ecke, aber grundsätzlich ist das schon ein ganz praktisches Modul.

```
package MyTest;

our $var = 0;
my $test = 1;
sub test {};

package main;

use 5.010;
use Devel::InPackage qw(in_package);

# MyTest
say in_package(file => $0, line => 4);
# main
say in_package(file => $0, line => 9);
# main
say in_package(code => slurp(), line => 15);
# slurp liest Skript ein
```

Pod::Spell::CommonMistakes

Dokumentation zu schreiben macht meistens wenig Spaß. Leider schleichen sich auch immer wieder Rechtschreibfehler in das Pod. Ob man die typischen Fehler auch drin hat, kann man mit `Pod::Spell::CommonMistakes` überprüfen. Im Gegensatz zu `Pod::Spell` nimmt es eine eigene Wortliste und nicht keinen Spellchecker.

```
use Pod::Spell::CommonMistakes qw(check_pod);

my $result = check_pod( 'file.pod' );
for my $k ( keys %$result ) {
   print " Found: '$k' - Possible spelling:
     '$result->{$k}'?\n";
}
```

Termine

Mai 2011

02.	Treffen Hamburg.pm
03.	Treffen Frankfurt.pm
	Treffen Vienna.pm
05.	Treffen Dresden.pm
09.	Treffen Ruhr.pm
	Treffen Hannover.pm
10.	Treffen Stuttgart.pm
11.-14.	Linuxtage Berlin
16.	Treffen Erlangen.pm
18.	Treffen Darmstadt.pm
	Treffen München.pm
25.	Treffen Berlin.pm
31.	Treffen Bielefeld.pm

Juni 2011

02.	Treffen Dresden.pm
06.	Treffen Hamburg.pm
07.	Treffen Frankfurt.pm
	Treffen Vienna.pm
11.	Beijing Perl-Workshop
13.	Treffen Ruhr.pm
	Treffen Hannover.pm
14.	Treffen Stuttgart.pm
15.	Treffen Darmstadt.pm
20.	Treffen Erlangen.pm
28.	Treffen Bielefeld.pm
29.	Treffen Berlin.pm

Juli 2011

04.	Treffen Hamburg.pm
05.	Treffen Frankfurt.pm
	Treffen Vienna.pm
07.	Treffen Dresden.pm
11.	Treffen Ruhr.pm
	Treffen Hannover.pm
12.	Treffen Stuttgart.pm
18.	Treffen Erlangen.pm
20.	Treffen Darmstadt.pm
	Treffen München.pm
26.	Treffen Bielefeld.pm
27.	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

6th FrOSCon

Free and Open Source Software Conference

20. - 21. August 2011

ÜBER 60 AUSSTELLER UND PROJEKTE
ÜBER 100 VORTRÄGE
LPI-PRÜFUNGEN
KINDER- UND JUGENDTRACK
SOCIAL EVENT
HÜPFBURG

HOCHSCHULE BONN-RHEIN-SIEG, GRANTHAM-ALLEE 20, 53757 SANKT AUGUSTIN

Sponsored by:

Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

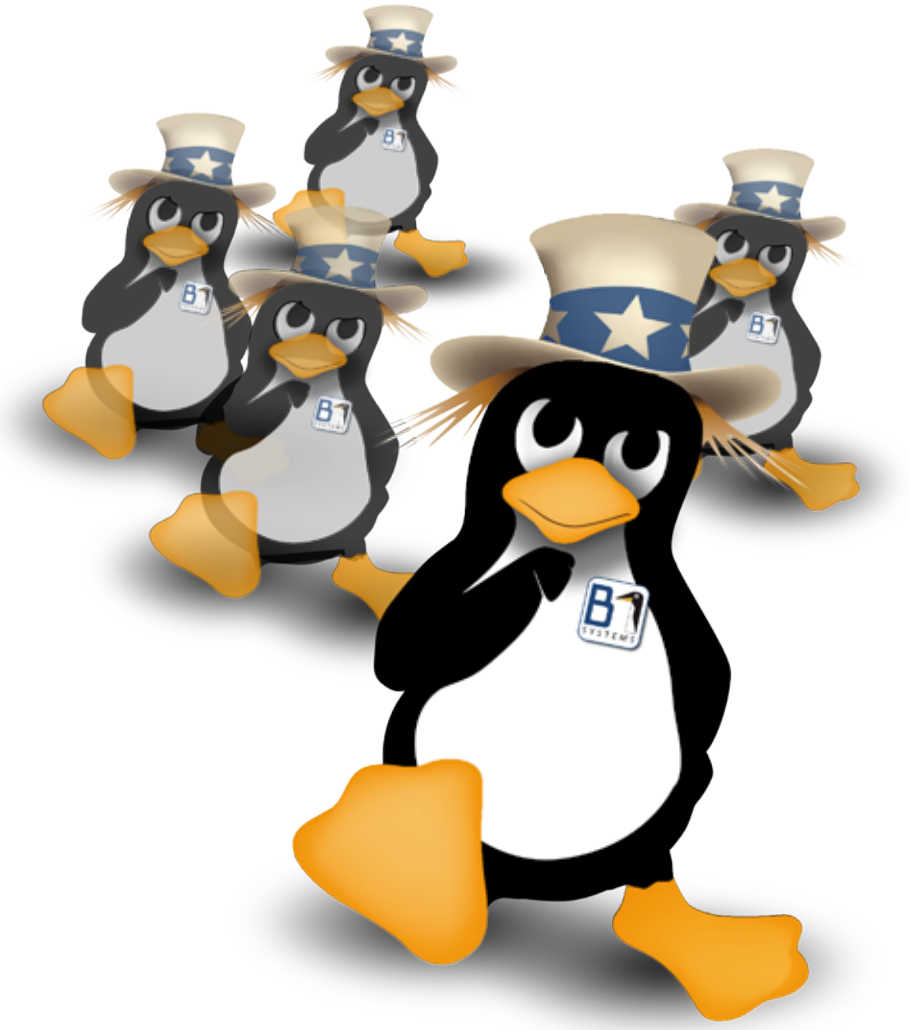
Fachbereich Informatik

tarent

BOOKING.COM
online hotel reservations

www.froscon.de

We want YOU



to B1 of us

Linux / Open Source Consulting, Training, Support & Development

jobs@b1-systems.de