Ausgabe 4/2010 – www.foo-magazin.de - 6,00 €

$foo

PERL MAGAZIN

Zentyal
... früher bekannt als eBox Plattform

Simple DirectMedia Layer
Perl wird bunt

How-To: App::perlbrew
Mehrere Perl-Installationen im Heimatverzeichnis

Nr 16

Die Österreichische Bibliothekenverbund und Service GmbH betreibt ein österreichweites Bibliothekenverbundsystem und ist Anbieter von IT-Dienstleistungen in diesem Bereich. Wir arbeiten hauptsächlich mit der Applikations-Software Aleph, Primo, MetaLib und SFX, den Datenbanksystemen Oracle und MySQL, einer Anzahl von Open Source-Tools sowie den Betriebssystemen Unix bzw. Linux. Für die Abteilung „Laufende Planung / Implementierung“ suchen wir eine/n

Junior Softwareentwickler/in

Sie haben Interesse an der Informationstechnologie und sind mit den Grundlagen der Softwareentwicklung vertraut. Idealerweise haben Sie bereits in Softwareentwicklungsprojekten gearbeitet. Sie sind bereit sowohl in den klassischen Technologien (Perl, Shell-Scripting) als auch den modernen Technologien (z.B. Java, XML, AJAX) zu arbeiten.

Aufgaben:

- (Weiter-) Entwicklung, Test, Dokumentation und Qualitätssicherung von Anwendungen oder Anwendungsmodulen und Tools (Perl/DBI, Shell Scripts, Oracle, SQL, XML)
- Erstellung von Lösungsbeschreibungen und detaillierten Designspezifikationen
- Fehleranalyse und -korrektur; Anwenderunterstützung
- Unterstützung und Mitarbeit bei komplexen Datenanalysen und -konvertierungen

Anforderungen:

- Fundierte technische Ausbildung (HTL, FH oder vergleichbare Ausbildung)
- Praktische Erfahrungen
  - in der prozeduralen und objektorientierten Programmierung,
  - im UNIX/Linux-Bereich,
  - im Bereich der relationalen Datenbanken (ORACLE, MySQL, Postgres)
- Grundkenntnisse im Bereich moderner Webtechnologien
- Engagiert, teamfähig und lernbereit
- Technisches Englisch

Vorteilhaft:

- Erfahrungen in der Programmiersprache **Perl**. Andernfalls sollten Sie bereit sein, im Rahmen ihrer Tätigkeit diese Programmiersprache zu erlernen.

Sie ergänzen ein kleines, engagiertes Team mit abwechslungsreichen Aufgaben im Umfeld der wissenschaftlichen Bibliotheken. Wir bieten die Möglichkeit zur flexiblen Arbeitszeit-einteilung, regelmäßige Weiterbildungsmöglichkeiten und ein angenehmes Arbeitsklima.

Ihre aussagekräftige Bewerbung richten Sie bitte per E-Mail mit dem Betreff *Junior Softwareentwickler/in* bis spätestens 30. November 2010 an: job@obvsg.at

Die Österreichische Bibliothekenverbund und Service GmbH
Brünnlbadgasse 17/2A
A-1090 Wien
Tel.: +43 1 4035158-0
www.obvsg.at
www.obvsg.at/wir-ueber-uns/jobs

VORWORT

Was war...

Das Jahr 2010 neigt sich langsam dem Ende zu und das nehme ich als Gelegenheit, mal einen ganz kurzen Abriss über das "Perl-Jahr 2010" zu schreiben.

Es ist wieder einiges passiert:

Anfang des Jahres wurde die 20.000er Marke an Distributionen auf CPAN überschritten. Das ist doch mal eine gewaltige Anzahl an Bibliotheken, die in den eigenen Programmen verwendet werden können. Leider ist nicht klar, welche Distribution die 20.000. war.

Am 29. Juli ist Rakudo * erschienen - und damit gibt es keine Entschuldigung mehr, sich Perl 6 nicht wenigstens anzuschauen. Rakudo ist noch weit davon entfernt, für Performancekritische Anwendungen eingesetzt werden zu können, aber es ist ein Anfang.

Aber auch Perl 5 ist nicht stehen geblieben: Im April wurde Perl 5.12.0 veröffentlicht und mittlerweile gibt es auch Perl 5.12.2. Die Releasezyklen haben sich geändert. Jeden Monat gibt es ein Developer-Release - im Moment in Vorbereitung auf Perl 5.14.0.

Ich habe es auch schon in mehreren Ausgaben erwähnt: Es gibt immer mehr Veranstaltungen, auf denen es Perl-Stände gibt. Von der FOSDEM über FrOSCon und LinuxTage bis hin zur CeBIT. Die große Anzahl an Besuchern an den Ständen zeigt, dass diese Stände erfolgreich und wichtig sind. Hier sind natürlich weitere Helfer gern gesehen. Im Wiki der Perl Foundation gibt es eine eigene Events-Seite, auf der man sich informieren kann, wo Hilfe benötigt wird.

Und auf einer weiteren Veranstaltung war ich gewesen. Nicht direkt für Perl, aber für $foo - die Buchmesse in Frankfurt. Bei einer Fachzeitschriften-Ausstellung gab es auch das Perl-Magazin zu bewundern. Mal schauen, ob die Messe etwas gebracht hat.

Eigentlich wollten wir - Frankfurt.pm - die YAPC::EU 2011 nach Frankfurt holen. Das hat leider nicht geklappt und die YAPC::EU im nächsten Jahr wird in Riga stattfinden. Herzlichen Glückwunsch an Andrew Shitov - und wir werden es wahrscheinlich für 2012 nochmal probieren.

Das war ein ganz kurzer Einblick in ein tolles Jahr. Wer sich auf der Seite http://ironman.enlightenedperl.org umschaut, wird sehen, dass es noch ganz viele andere Sachen gab.

Ich hoffe, dass 2011 wieder genauso spannend wird und dass Sie uns auch ins fünfte Jahr von `$foo` begleiten. Jetzt wünsche ich viel Spaß mit der Ausgabe 16.

Viele Grüße,
Renée Bäcker

Die Codebeispiele können mit dem Code

5fpou36

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/issue16

IMPRESSUM

Herausgeber:	Smart Websolutions Windolph und Bäcker GbR Untere Rützelstr. 1a D-65933 Frankfurt
Redaktion:	Renée Bäcker, Katrin Bäcker, André Windolph
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

MODULE

PERL

ANWENDUNG

WIN32

TIPPS & TRICKS

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.

Javier Amor Garcia

Javier Amor Garcia war ein ahnungsloser Anfänger als seine Entdeckung von Freier Software sein Interesse am Computerhandwerk geweckt hat. Er bekam seinen ersten Programmierjob mit Perl und verbrachte sechs Jahre mit seiner Firma. Javier denkt, dass es eine Schwäche von Perl ist, dass es oft mit einem Kamel anstatt eines Dromedars in Verbindung gebracht wird, das ja eindeutig das klügere der beiden Tiere ist.

Thomas Fahle

Perl-Programmierer und Sysadmin seit 1996.
Websites:

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

Tobias Leich

Tobias Leich arbeitet als Softwareentwickler bei einem Serviceunternehmen in der Nähe von Berlin. Dabei bestimmen Webservices zur internen Prozessoptimierung sein Tagesgeschäft. Nebenbei programmiert er an einer alternativen Benutzeroberfläche für eine Art Heim-Steuerungssystem und ist dabei auf SDL gestoßen. Sein Ziel ist die Entwicklung robuster Mensch - Maschine Schnittstellen die maßgeblich auf Sprachein- und ausgabe beruhen.

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.

Renée Bäcker

Zeitbasierte Konfiguration

In einem meiner Projekte sollen bestimmte Konfigurationseinstellungen je nach Uhrzeit unterschiedlich aussehen. In diesem Artikel möchte ich darstellen, wie ich diesen Teil gelöst habe.

Als ganz einfaches Beispiel soll hier in diesem Artikel die Annahme, dass eine Mandantenfähige Webseite je nach Kunde und Uhrzeit ein anderes Logo anzeigt, dienen. Die Konfiguration wird als XML-Datei gespeichert. In Listing 1 ist eine Beispielkonfiguration zu sehen.

Jeder Kunde hat also einen eigenen Bereich in der Konfiguration. Dort wiederum sind die Angaben zu den Logos gespeichert.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<website>
  <kunden>
    <kunde name="kunde1">
      <logo>
        <vevent>
          <dtstart>20100601T000000Z</dtstart>
          <dtend>20100601T150000Z</dtend>
          <rrule>RRULE:FREQ=DAILY;UNTIL=20110601T235959</rrule>
        </vevent>
        <src>http://example.invalid/logo.png</src>
      </logo>
      <logo>
        <vevent>
          <dtstart>20100601T150001Z</dtstart>
          <dtend>20100601T235959Z</dtend>
          <rrule>RRULE:FREQ=DAILY;UNTIL=20110601T235959</rrule>
        </vevent>
        <src>http://example.invalid/logo2.png</src>
      </logo>
    </kunde>
    <kunde name="kunde2">
      <logo>
        <vevent>
          <dtstart>20100601T000000Z</dtstart>
          <dtend>20100601T150000Z</dtend>
          <rrule>RRULE:FREQ=DAILY;UNTIL=20110601T235959</rrule>
        </vevent>
        <src>http://example.invalid/logo5.png</src>
      </logo>
      <logo>
        <vevent>
          <dtstart>20100601T150001Z</dtstart>
          <dtend>20100601T235959Z</dtend>
          <rrule>RRULE:FREQ=DAILY;UNTIL=20110601T235959</rrule>
        </vevent>
        <src>http://example.invalid/logo6.png</src>
      </logo>
    </kunde>
  </kunden>
</website>
```

Listing 1

Für die Darstellung der unterschiedlichen Zeiträume wird ein Teil des iCal-Formats verwendet. Der `<vevent>`-Block ist aus der Spezifikation von iCal abgeschaut. Damit muss man sich keine Gedanken über mögliche Formate machen und man kann auf CPAN-Module zurückgreifen.

Für den Kunden 1 wird zwischen 15:00 Uhr und 19:00 Uhr also Logo1 angezeigt, zwischen 00:00 und 15:00 Uhr Logo2. Diese zeitliche Eingrenzung gilt täglich. Man könnte dank dem iCal-Format sehr einfach Einstellungen erreichen, dass die Logos auch noch täglich unterschiedlich sind.

Parsen der Konfigurationsdatei

Für das Parsen der Konfigurationsdatei wird `XML::LibXML` verwendet, um die XML-Struktur in eine Perl-Datenstruktur umzuwandeln. Für den Kunden werden die Logo-Angaben erst in ein Array überführt. Dass erst alles "umständlich" in eine Perl-Datenstruktur umgewandelt wird hat auch damit zu tun, dass die Konfiguration ursprünglich im YAML-Format gespeichert wurde und das eigentliche Programm durch die Umstellung des Formats in XML nicht geändert werden sollte.

Das Parsen des XML wird in Listing 2 gezeigt.

Der Knotenname "kunde" taucht dann nicht mehr auf. Das hätte nur eine zusätzliche Ebene bedeutet und im YAML gab es diese Ebene auch nicht. Deshalb gibt es hier die Sonderbehandlung für den einen Knotennamen. Alle weiteren Ebenen werden in Hashes gespeichert.

Welcher Teil ist gerade gültig?

Jetzt zu dem Hauptgrund für diesen Artikel: Wie stelle ich fest, welcher Teil der Konfiguration gerade gültig ist? Wie

```
sub walk_tree {
    my ($self,$hashref,$node) = @_;

    my @children = $node->childNodes;

    CHILD:
    for my $child ( @children ) {
        my $name          = $child->nodeName;
        my @has_children = $child->childNodes;

        next CHILD unless @has_children;

        if ( $name eq 'kunde' ) {

            my $client = $self->client;
            $name = $child->getAttribute( 'name' );

            next CHILD if $name ne $client;

            my $client_config = $self->walk_tree( {}, $child );
            push @{ $hashref->{$name} }, $client_config;

            next CHILD;
        }

        $hashref->{ $name } = {} unless $hashref->{$name};
        if ( my ($text) = grep{ $_->nodeName eq '#text' }@has_children ) {
            $hashref->{ $name } = $text->textContent;
            next CHILD;
        }
        else {
            $self->walk_tree( $hashref->{ $name }, $child );
        }
    }

    return $hashref;
}
```

Listing 2

weiter oben schon angedeutet, benutze ich hierfür CPAN-Module. Für mein bevorzugtes Datum/Zeit-Modul `DateTime` gibt es eine "Erweiterung": `DateTime::Format::ICal`.

Das bietet die Möglichkeit, sowohl die Zeitangaben als auch die Wiederholungsregeln zu parsen und man bekommt `DateTime`-Objekte. Da es ja Wiederholungen gibt, muss man überprüfen, ob das aktuelle Datum zu einer Wiederholungsregel passt (siehe Subroutine `contains` in Listing 3).

Um Zeit zu sparen und keine unnötige Arbeit zu machen, wird wirklich nur die Teilkonfiguration für den Kunden geparst, für den die Anfrage vorliegt (siehe Listing 4).

Damit man bei Kunden, die die ganze Zeit über immer das gleiche Logo anzeigen wollen, nicht auch so eine zeitbasierte Konfiguration zu benötigen, wird so eine Konfiguration als "Default" angesehen (siehe Listing 5).

```
sub select_by_time {
    my ($self,$config) = @_;

    my @to_delete;
    my $requested_client = $self->client;

    if ( exists $config->{kunden} ) {

        CLIENT:
        for my $clientname ( keys %{ $config->{kunden} } ) {

            next CLIENT if $requested_client and $clientname ne $requested_client;
            next CLIENT if !$clientname;
```

Listing 3

```
if ( ref( $config->{kunden}->{$clientname} ) eq 'ARRAY' ) {
       my $default;
       my $local_config;

       PART:
       for my $part ( @{ $config->{kunden}->{$clientname} } ) {

           next PART if !$part or ref( $part ) ne 'HASH';

           # check if all needed info is given
           if ( !exists $part->{vevent} ) {
               $default = $part;
               next PART;
           }
```

Listing 4

```
elsif ( exists $part->{vevent} && ref( $part->{vevent} ) eq 'HASH' &&
        !( $part->{vevent}->{dtstart} && $part->{vevent}->{dtend}
                && $part->{vevent}->{rrule} ) ) {
    next PART;
}

my $start      = $part->{vevent}->{dtstart};
my $end        = $part->{vevent}->{dtend};
my $recurrence = $part->{vevent}->{rrule};

my ($date,$end_obj,$rec_obj);

eval {
    $date       = DateTime::Format::ICal->parse_datetime( $start );
    $end_obj    = DateTime::Format::ICal->parse_datetime( $end );
    $rec_obj    = DateTime::Format::ICal->parse_recurrence(
        recurrence => $recurrence,
        dtstart    => $date,
    );
    1;
};

next PART if !( $date && $end_obj && $rec_obj );
```

Listing 5

Bei Nicht-iCal-konformen Zeitangaben, stirbt das Skript. Deshalb das `eval`. Hier gebe ich den fehlerhaften String nicht aus, den könnte man aber noch in die Log-Datei schreiben.

```
my $today   = DateTime->now;
my $dur_obj = $end_obj - $date;
```

`DateTime` bietet mit der Funktion `now` eine sehr bequeme Möglichkeit, ein Objekt zum aktuellen Datum zu bekommen. In `$dur_obj` landet die Dauer die diese Einstellungen gültig ist. Das ist später in der Subroutine `contains` notwendig, weil über die Wiederholungsregel immer nur der Startpunkt der Wiederholung herauszufinden ist.

```
if (
  $self->contains($rec_obj, $dur_obj, $today)
) {
  $local_config = $part;
  last PART;
}
```

Wenn das aktuelle Datum innerhalb einer Wiederholungsregel liegt, dann brauchen die restlichen Logo-Konfigurationen nicht überprüft werden.

```
$config->{kunden}->{$clientname} =
             $local_config || $default;

if ( !$config->{kunden}->{$clientname} ) {
   push @to_delete, $clientname;
}


delete @{$config->{kunden}}{@to_delete};
```

Nicht benötigte Konfigurationsangaben müssen gelöscht werden.

Wie oben schon angedeutet, ist die Subroutine `contains` dafür da, zu überprüfen, ob das aktuelle Datum durch eine Wiederholungsregel abgedeckt wird (siehe Listing 6).

`DateTime::Format::ICal` erstellt für Wiederholungsregeln einen Iterator, so dass man selbst keine Datumsberechnungen anstellen muss. Bei jedem Schritt liefert der Iterator den Beginn einer Wiederholung. Sobald der Startbeginn irgendwann in der Zukunft liegt, muss nicht weiter überprüft werden. Da das aktuelle Datum ja nicht exakt der Startpunkt der Wiederholung sein muss, wird die Dauer einer Wiederholung benötigt. Dank `DateTime` muss man nicht selbst groß rumrechnen, sondern man kann mit einer "Addition" zu dem Endzeitpunkt kommen.

`DateTime` ermöglicht es, mit "normalen" Vergleichsoperatoren die Daten zu vergleichen. Dadurch wird auch für einen `DateTime`-Einsteiger klar, was der Code macht und der Code bleibt schlank.

Probleme beim Debuggen

An einer Sache bin ich schier verzweifelt - auf jeden Fall war für mich die Ursache nicht sofort ersichtlich. Aber wobei hatte ich Probleme? Zum Debuggen schreibe ich jede Menge Sachen in eine Log-Datei. Unter anderem habe ich dort zeitweise die Überprüfung auf die Gültigkeit der Konfiguration sehr genau mitgeloggt.

Für "dtstart" war "TZID=Europe/Berlin:20100601T000000" und für "dtend" war "TZID=Europe/Berlin:20110601T235959" eingetragen.

Dabei sind dann diese Zeilen aufgetaucht:

```
Aktuell 2010-06-04T12:42:18
    -> Start: 2010-06-04T00:00:00
       Ende:  2010-06-04T14:00:00
Aktuell > Start? 1
Aktuell < Ende?
```

Aber warum ist "2010-06-04T12:42:18" nicht kleiner als "2010-06-04T14:00:00"?

Das Problem liegt darin, das man bei der Stringifizierten Version nicht erkennt, für welche Zeitzone das gilt. Die Zeitangabe in der Konfiguration wurde mit der Zeitzonen-Angabe "Europe/Berlin" gemacht, der Server läuft aber in UTC-Zeit. Damit ist das "2010-06-04T12:42:18", das mit `my $day = DateTime->now` erzeugt wurde wirklich in UTC-Zeit, aber

```
sub contains {
    my ($self,$rec,$dur,$day) = @_;

    my $return = 0;

    my $iter = $rec->iterator;
    while ( my $dt = $iter->next ) {
        last if $dt > $day;

        my $end = $dt + $dur;

        next if $end < $day;
        if ( $end >= $day and $dt <= $day ) {
            $return = 1;
            $self->valid_to( $end->epoch );
        }
    }

    return $return;
}
```

Listing 6

das "2010-06-04T14:00:00" ist in Berliner Zeit, also "2010-06-04T12:00:00" in UTC-Zeit. Und `DateTime` nimmt für die Vergleiche immer die Epochen-Sekunden (UTC-Zeit). Damit ist dann auch das Ergebnis erklärbar. Aber darauf muss man erstmal kommen.

Caching

Die Verwendung des iCal-Formats und der entsprechenden Module sorgt dafür, dass das Auslesen der Konfiguration relativ lange dauert - es müssen ja immer wieder die Angaben geparst und dann ausgewertet werden.

```
sub get_cache {
    my ($self, $config, $time) = @_;

    my $dir    = $self->cache_dir;
    my $client = $self->client;

    # get all files in the directory
    my %files;
    if ( opendir my $dirhandle, $dir ) {
        %files = map{ $_ => $dir . '/' . $_ }
                 grep{ /\.dat \z/x and -f $dir . '/' . $_ }
                 readdir $dirhandle;
        closedir $dirhandle;
    }

    # delete config cache files that are outdated
    my @old_files = grep{ /(\d+)\.dat/ && $1 < $time }keys %files;

    unlink @files{@old_files};
    delete @files{@old_files};

    my $cache_file;

    # if a file with timestamp for config file exists
    if ( my $config_check = first{ /\A \d+ _base\.dat \z/x }keys %files ) {

        # Timestamp of the config file, the cache files are
        # based on.
        my ($check_timestamp) = $config_check =~ /\A(\d+)_/;

        # get "last modified" time of config file
        my $stat = stat( $config )->mtime;

        # if cache files do not contain current info
        # delete them
        if ( $check_timestamp != $stat ) {
            unlink values %files;
            %files = ();
        }
        else {

            # does a cache file for the requested stream exist?
            my $possible_cache    = first{ /\A $client _ \d+ \.dat \z/x }keys %files;
            $possible_cache     ||= '';
            my ($check_timestamp) = $possible_cache =~ / (\d+) \.dat /x;

            # cache file is still valid
            if ( $possible_cache and $check_timestamp > $time ) {
                $cache_file   = $possible_cache;
            }
        }
    }

    return $files{$cache_file} if $cache_file;
    return;
}
```

Listing 7

Durch die vielen "Teilkonfigurationen" für Kunden und Logos gibt es eine Vielzahl an möglichen Konfigurationen. Für den Benutzer ist es aber inakzeptabel, bei jeder Anfrage 4 Sekunden zu warten, nur bis die Konfiguration ausgelesen und ausgewertet ist. Hier ist also ein Caching notwendig. Ich habe dafür ein ganz einfaches Caching eingebaut:

Ich speichere in einer Datei die Zeit der letzten Änderung der Konfigurationsdatei und für jede Teilkonfiguration speichere ich die serialisierten Daten in einer extra Datei. Kommt eine Anfrage, prüfe ich erst, ob die aktuelle Konfiguration schonmal geparst wurde. Wenn ja, wird als nächstes geprüft, ob die Teilkonfiguration schonmal benötigt wurde.

In Listing 7 ist zu sehen, wie diese Prüfung vorgenommen wird.

Im Cache-Verzeichnis wird erst nach der Datei gesucht, in der der Timestamp der Konfigurationsdatei gespeichert ist, auf der die Kunden-Caches basieren. Ist zwischenzeitlich die Konfigurationsdatei verändert worden, müssen die Kunden-Caches natürlich gelöscht werden.

Basieren die Kunden-Caches auf der aktuellen Konfigurationsdatei, wird dann geprüft, ob für den Kunden ein Cache vorliegt. Falls ja, wird anhand der Zeitangabe im Dateinamen überprüft, ob das für die aktuelle Uhrzeit noch gültig ist.

Im Projekt ist es so, dass es 4 Sekunden dauert, bis die Konfiguration ausgelesen ist und der initiale Kunden-Cache erzeugt ist. Der erste Nutzer muss also relativ lange warten. Bei den weiteren Benutzern wird dann der Cache ausgelesen (so lange der gültig ist) und das braucht noch nicht mal 0.01 Sekunden. Durch das Caching wurde so die Performanz deutlich erhöht.

Tobias Leich

Perl wird bunt - Simple DirectMedia Layer

Was ist SDL?

SDL ist zweierlei, zum einen die bekannte und weit verbreitete C Library libSDL und zum anderen ein Modul, das diese Library an Perl anbindet: SDL. Die libSDL läuft genauso wie Perl auf unterschiedlichsten Plattformen. Im Moment unterstützt sie offiziell: Linux, Windows, Windows CE, BeOS, MacOS, Mac OS X, FreeBSD, NetBSD, OpenBSD, BSD/OS, Solaris, IRIX, und QNX. Weiterhin enthält der Code u. a. Unterstützung für Amiga, Dreamcast und SymbianOS. Diese werden aber nicht offiziell unterstützt. Das SDL-Modul wurde bislang erfolgreich auf Linux, Windows, FreeBSD und Mac OS X getestet.

Fazit: Entwicklungen im Bereich Spiele und Multimedia sind ohne weiteren Aufwand portabel. Dabei steht einem ein Leistungsspektrum von Hardwarebeschleunigten 2D/3D-Grafiken, Mehrkanal-Soundmixerfunktionen und Eventqueues bis hin zur Lowlevel Schnittstelle zu Keyboard, Mouse und Joystick zur Verfügung.

Wie wird eine C-Library an Perl angebunden?

Für solche Fälle gibt es XS, ein Interface um C/C++ Subroutinen anzusprechen und Daten auszutauschen. Um nun genau zu wissen was die C-Funktion von einem erwartet benötigt man die Header-Dateien und statische Libraries der anzubindenden Bibliothek.

Genau diese Bindungen zur libSDL wurden im vergangenen Jahr komplett überarbeitet. Im Zuge dessen wurden für alle Subroutinen Tests und Dokumentationen geschrieben. Dank cpantesters sieht man eine breite Palette an unterstützten Betriebssystemen und Perlversionen.

Dieser Artikel soll aber keinen Einblick in XS geben, sondern die Funktionen und Möglichkeiten veranschaulichen die uns SDL bietet.

In diesem Moment gibt es mehrere Projekte die sich mit Frameworks und Gameengines beschäftigen, um die Spieleentwicklung mit SDL noch einfacher und schneller zu machen. Wir werden uns kurz mit den SDL Grundkonzept beschäftigen und danach eine dieser Erweiterung einbinden. Diese wird mit SDL mitgeliefert, sodass kein zusätzlicher Installationsaufwand betsteht.

Alle hier gezeigten Skripte und Beispiele basieren auf SDL in der Version 2.512. Dabei sind die Skripte hier aus Platzgründen nur Auszugsweise abgedruckt, die vollständigen Skripte und Grafiken sind abrufbar unter: http://perl-magazin.de

Weiterführende Informationen und Hilfe zu diesem Artikel gibt's es unter:

1. Web: http://sdl.perl.org
2. IRC: irc.perl.org im Channel #sdl
3. Mail: sdl-devel@perl.org
4. Git: http://github.com/PerlGameDev/SDL

Ich persönlich bin dort unter dem Nick „FROGGS“ zu finden.

Tutorial 1: Hallo bunte Welt!

Zuerst möchte ich mit dem bekannten „Hallo Welt“-Beispiel anfangen. Wir werden aber nicht wie gewohnt einen Text in eine Shell ausgeben lassen, sondern diesen Text zusammen mit einem Bild in ein Grafikfenster rendern.

Um eine Grafik oder einen Text darzustellen muss man erst einmal verstehen wie SDL arbeitet. Als Erstes legen wir fest, wie unser Fenster aussehen soll, dazu gehört u.a. die Höhe und Breite.

Abbildung 1: "Hallo Welt"

Das Objekt was wir daraus erhalten ist ein sogenanntes Surface, also ein Fläche, welche man sich wie eine Art Bild vorstellen kann. Auf dieser Fläche kann man nun weitere Surfaces darstellen, dies geschieht in zwei Schritten. Zum einen kopiert man ein Surface an eine bestimmte Position eines anderen Surfaces (genannt blit), zum anderen muss der Bereich des modifizierten Surfaces aktualisiert werden, um die Änderungen sichtbar zu machen (das nennet man flip bzw. update).

In Listing 1 das Beispielprogramm, welches obige Ausgabe (in Abbildung 1) bewirkt.

Als erstes werden natürlich die benötigten Module geladen. SDL::Rect und SDL::Surface sind so im darauffolgenden Code nicht sichtbar, diese werden aber benötigt da u.a. SDLx::App ein Surface, und `$earth->clip` ein SDL::Rect ist.

Anschließend erzeugen wir das Videofenster. 320 mal 240 Pixel sind für unser kleines Beispiel ausreichend. 32bit beträgt die Farbtiefe, also 16,7 millionen Farben plus 8-bit Alphakanal (Transparenz).

Wenn wir unser Programm nun ausführen würden, würden wir lediglich ein schwarzes Fenster sehen, welches sich sofort wieder schließt.

Jetzt wollen wir mal etwas Farbe ins Spiel bringen. Das geschieht am einfachsten mit einem Bild. Wir laden also unser Bild der Erdkugel mit Hilfe von SDL::Image::load(). Diese Funktion unterstützt recht viele Dateiformate nativ, weitere in Abhängigkeit zusätzlicher Bibliotheken (jpeg, png und tiff).

Die Funktion SDL::Image::load() gibt uns ein Surface zurück. Wir erinnern uns, ein Surface ist die reine Repräsentation von Bildinformationen. Das bedeutet, dass die Position, die das Bild auf dem Fenster einnehmen soll, darin noch nicht enthalten ist. Dafür werden wir den SDLx::Layer missbrauchen.

In einem SDLx::Layer können wir Bildinformationen (Surface), dessen Position, dessen Bildausschnitt und zusätzliche benutzerdefinierte Daten speichern. Wir begnügen uns in diesem Beispiel mit den ersten beiden Werten. Wir übergeben also das erhaltene Surface, und setzen die Position auf x = 110 und y = 50. Somit wird die linke Seite des Bildes 110 Pixel von links, und die Oberkante 50 Pixel von oben sein. Damit wird die Erdkugel annähernd auf unserem Fenster zentriert.

Die Funktion blit_surface macht nun die eigentliche Arbeit. Sie kopiert Bilddaten von einem Surface auf ein anderes, unter Berücksichtigung des Bildausschnittes (`$earth->clip`) und der Position (`$earth->pos`). Da wir keinen Bildausschnitt definiert haben entspricht `$earth->clip` den Abmessungen unseres Bildes der Erde.

```
use SDL;
use SDL::Image;
use SDL::Rect;
use SDL::Surface;
use SDL::Video;
use SDLx::App;
use SDLx::Layer;
use SDLx::SFont;

my $window = SDLx::App->new( title => 'Test', width => 320, height => 240, depth => 32 );

my $earth  = SDLx::Layer->new( SDL::Image::load('data/earth.jpg'), 110, 50 );
SDL::Video::blit_surface( $earth->surface, $earth->clip, $window, $earth->pos );

my $font   = SDLx::SFont->new('data/font.png');
SDLx::SFont::print_text( $window, 65, 150, "HALLO BUNTE WELT!" );

$window->flip();

sleep(5);                                                                    # hello_world.pl
```

Listing 1

Für den zweiten Schritt benutzen wir das Modul SDLx::SFont. Damit haben wir die Möglichkeit, mit einer einfachen Grafikdatei, Text darzustellen. Als erstes laden wir die Schrift, als zweites geben wir an auf welches Surface der Text ausgegeben werden soll, gefolgt von dessen Position (x, y) und natürlich dem Text selbst. Die Funktion print_text ruft im Verborgenen auch blit_surface auf, sodass nun die Bildinformationen der Erde, sowie des Textes auf unser Fenster kopiert werden. Um diese sichtbar zu machen rufen wir `$window->flip()` auf. Anschließend warten wir ein paar Sekunden und genießen den Ausblick.

Der Ablauf nochmal in Kürze:

- Fenstergröße setzen (new SDLx::App, set_video_mode)
- Surfaces erzeugen (SDL::Image, SDL::SFont, SDL::TTF, ...)
- Surfaces kopieren (blit_surface, print_text, ...)
- Bildschirm aktualisieren (flip, update_rect, ...)

Die wichtigsten drei Objekte von SDL

SDL::Surface

Dieses Objekt haben wir schon kennengelernt. Es ist ein Container für Bildinformationen und besitzt Methoden, die uns die Breite, die Höhe und das Pixelformat zurückgeben. Über das Pixelformat können wir u.a. erfahren, ob das Bild eine Palette hat und wie viele Farben es darstellen kann.

SDL::Rect

Das ist die wohl einfachste Struktur bei SDL. Ein SDL::Rect (Rechteck) speichert die Position (x und y), und die Dimension (Breite und Höhe).

SDL::Event

Das SDL::Event Objekt ist vermutlich auf den ersten Blick etwas angsteinflößend. Mit fast 40 Methoden und ebenso vielen Konstanten wirkt es aufgebläht. Dies verliert sich aber auf den zweiten Blick. Ein Event kann erst einmal grundlegend zwei Quellen haben: Ausgelöst durch das System selbst oder durch den Programmcode, durch benutzerdefinierte Events. Wir betrachten hier nur den ersten Teil.

Wir stellen uns nochmal unser „Hallo Welt“-Programm vor. Bewegen wir die Maus in dem Fenster oder drücken wir eine Taste wird ein Event ausgelöst. Dieser Event hat als wichtigstes Merkmal einen Typ, welcher aussagt, um was für einen Event es sich handelt. In unserem Beispiel die Mausbewegung (`SDL_MOUSEMOTION`) und der Tastendruck (`SDL_KEYDOWN`, `SDL_KEYUP`).

Wenn wir nun feststellen dass ein Event von einem bestimmten Typ ist, dann reduzieren sich die möglichen nutzbaren Methoden auf maximal fünf. So können wir etwa bei einer Mausbewegung den Status der Maustasten und die absolute sowie relative Position der Maus erfragen. Der Tastaturevent hingegen verrät uns in erster Linie nur welche Taste gedrückt bzw. losgelassen wurde.

Der Herzschlag eines Spiels

Die Architektur eines Spiels ist vergleichbar mit der eines Daemons. Es gibt Schleifen (Heartbeats), die die Events abarbeiten und so lange laufen, bis zu einer anderen Schleife gesprungen oder das Programm durch den Benutzer beendet wird.

Solche Heartbeats sind u.a. für das Menü und das Spiel an sich sinnvoll. Dabei ist der Ablauf in dieser Schleife immer der Gleiche. Als erstes prüfen wir, ob Events vorliegen, die wir abarbeiten können (Bsp: Tastendruck auf Pfeil rechts). Daraus folgt zum Beispiel das Bewegen des Players, also das Positionieren des Bildes (Surfaces) des Spielercharakters an die neue Stelle. Nachdem alle Events abgearbeitet wurden zeichnen wir das Fenster neu und beginnen die Schleife von vorn.

Diesem Prinzip folgt auch das folgende Kapitel. Ein nahezu vollwertiges Solitaire Kartenspiel.

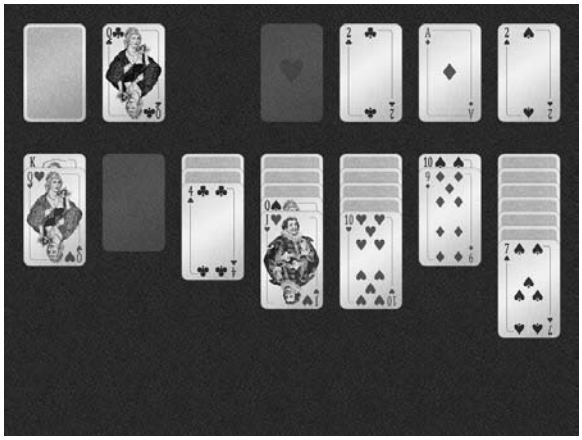


Abbildung 2: Solitaire Kartenspiel

Games::Solitaire

Meine Herangehensweise bei diesem Spiel war wie folgt:

1. Fenster erstellen, Hintergrundbild anzeigen
2. Eventloops erzeugen mit Debugausgabe für Klick, Doppelklick, Quit etc
3. Laden der Kartengrafiken
4. Erster Test: wir verschieben eine Karte mit der Maus
5. Spiellogik: - Regelwerk für Karten nehmen und ablegen
   - Durchklicken des Stapels links oben

Erster Schritt - Hintergrundbild mit Platzhaltern

Hier sehen wir ein paar Neuerungen, auf die ich nun eingehen möchte. Dem Konstruktor von SDLx::App werden Flags übergeben, es wird ein Objekt von der Klasse SDLx::LayerManager erzeugt und es werden Hashes an die SDLx::Layer übergeben.

SDL_HWSURFACE und SDL_HWACCEL

Diese Flags geben an, dass das Surface unserer Applikation im Speicher der Grafikkarte erzeugt wird. Dies sollte uns ein wenig mehr Performance bringen.

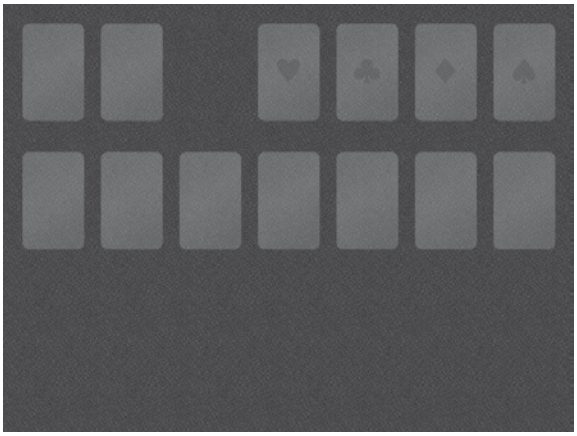


Abbildung 3: Hintergrundbild

```
use SDL::Image;
use SDL::Surface;
use SDLx::App;
use SDLx::LayerManager;
use SDLx::Layer;

my $display = SDLx::App->new(title => 'Games::Solitaire',
                             width => 800, height => 600, depth => 32,
                             flags => SDL_HWSURFACE | SDL_HWACCEL);
my $layers  = SDLx::LayerManager->new();

init_background();
$layers->blit($display);
$display->flip();

sleep(5);

sub init_background {
  # Hintergrundbild
  my $background  = SDL::Image::load('data/background.jpg');
  $layers->add(SDLx::Layer->new($background, {id => 'background'}));

  my $empty_stack = SDL::Image::load('data/empty_stack.png');

  # zwei Platzhalter für Kartenstapel oben links
  $layers->add(SDLx::Layer->new($empty_stack,  20, 20, {id => 'rewind_deck'}));
  $layers->add(SDLx::Layer->new($empty_stack, 130, 20, {id => 'empty_deck'}));

  # sieben Stapel unten
  for(0..6) {
    $layers->add(SDLx::Layer->new($empty_stack, 20+110*$_, 200, {id => 'empty_stack'}));
  }

  # vier Stapel oben rechts
  for(0..3) {
    $empty_target = SDL::Image::load('data/empty_target_' . $_ . '.png');
    $layers->add(SDLx::Layer->new($empty_target, 350+110*$_, 20, {id=>"empty_target_$_"}));
  }
}                                                          # 01-solitaire.pl
```

Listing 2

Der SDLx::LayerManager

Diese Klasse hat sehr hilfreiche Features für layerbasierte Anwendungen wie diese. Genau wie man in Wirklichkeit eine Karte nimmt, bewegt und wieder ablegt, funktioniert der Layermanager. Zudem bietet er Funktionen, die die Karten vor bzw. hinter einer Karte ermitteln. So lassen sich leicht Stapel bewegen oder Überprüfungen anstellen, ob eine Karte oben liegt oder nicht. Der Layermanager übernimmt ebenso die Aufgabe des Kopierens der Surfaces, wie das Listing 2 zeigt.

Hashes an SDLx::Layer

Der SDLx::Layer bietet die Möglichkeit einen benutzerdefinierten Hash zu übergeben. In diesem können wir Informationen speichern, um die Layer später zu identifizieren oder um Statuswerte darin abzulegen, die wir ansonsten in einer eigenen Datenstruktur verwalten müssten.

Zweiter Schritt – Eventloops

Anstelle des „sleep(5)“ fügen wir nun folgende zwei Routinen ein. Bitte beachten, ab hier ist der Code nur noch in Auszügen abgedruckt. Der vollständige Code zu jedem Schritt ist auf http://perl-magazin.de zu finden.

So wie in Listing 3, wenn auch nicht ganz vollständig, sieht unser Eventloop aus. Die Events on_drag, on_mousemove und on_drop habe ich aus Platzgründen hier ausgelassen, diese sind aber vergleichbar mit den Gezeigten.

Wir sehen, es wird ein Hash mit anonymen Subroutinen definiert. Dort, wo momentan nur der Eventname in die Shell ausgegeben wird, werden wir später die Spiellogik einbauen. Diesen Hash übergeben wir der event_loop Funktion. Die Events, insofern welche vorliegen, sind in einer Queue gespeichert. Das bedeutet, wir müssen Stück für Stück einen Event von der Queue holen und verarbeiten. Haben wir einen Event gefunden, der den gewünschten Typ hat, rufen wir die passende Subroutine unseres Hashes auf. Auf diese Art können wir die event_loop Funktion an anderen Stellen wiederverwenden. Im Falle eines Menüs würde ein angepasster Hash übergeben, der die Menüaktionen darstellt.

Die Funktion pump_events im nebenstehenden Code stellt sicher, dass alle Events der verschiedenen Quellen (Maus, Tastatur, Windowmanager) in die Queue übertragen werden. Anschließend können wir mit poll_events einen Event holen, dabei gibt diese Funktion 1 zurück, so lange noch Events in der Queue sind.

Wenn wir diesen Code ausführen und mit der Maus in das Fenster klicken oder eine Taste drücken, wird in die Shell der jeweilige Event geschrieben.

Fazit: Wir haben jetzt mit etwas über 100 Zeilen Code ein Grafikfenster inklusive funktionierendem Eventsystem. Im nächsten Schritt geht es dann darum, die Karten darzustellen.

```
game();

sub event_loop {
  SDL::Events::pump_events();
  while(SDL::Events::poll_event($event)) {
    my $type = $event->type;

    if ($type == SDL_MOUSEBUTTONDOWN) {
      my $time = Time::HiRes::time;
      if ($time - $last_click >= 0.3) {
        $handler->{on_click}->();
      }
      else {
        $handler->{on_dblclick}->();
      }
      $last_click = $time;
    }
    elsif ($type == SDL_KEYDOWN) {
        $handler->{on_keydown}->();
    }
    elsif ($type == SDL_QUIT) {
      $handler->{on_quit}->();
    }
  }
}

sub game {
  $handler = {
    on_quit      => sub {
      $loop = 0;
    },
    on_click     => sub {
      printf("click\n");
    },
    on_dblclick  => sub {
      printf("dblclick\n");
    },
    on_keydown   => sub {
      printf("keydown\n");
    },
  };

  while($loop) {
    event_loop();
    if(scalar @{$layers->blit($display)}){
      $display->flip();
    }
  }
}                          # 02-solitaire.pl
```

Listing 3

Dritter Schritt – Laden der Kartengrafiken

Nun kommt etwas mehr Farbe ins Spiel. Für dieses Spiel habe ich alle Karten als separate Bilder erstellt. Man könnte die Kartengrafiken auch dynamisch erstellen, indem man die Symbole (Herz, Pik, etc) und deren Werte auf die Kartensurfaces rendert. Dass würde den Code dieses Tutorials aber nur aufblähen. Deshalb heißt es nun, alle 52 Karten zu laden, zu positionieren und deren Werte in dem Hash des Layers zu speichern.

Die neu entstandene Funktion nennen wir passenderweise init_cards und rufen sie direkt nach init_background auf (siehe Listing 4).

Kurz zur Erläuterung, die ersten 28 Karten werden auf die sieben Stapel unten verteilt, erst eine, dann zwei, dann drei Karten u.s.w. Anschließend werden die restlichen Karten auf den Stapel oben links gelegt. Die Karten die mit `$visible=1` gekennzeichnet sind liegen auf den Stapeln oben auf und

```
sub init_cards {
  my $stack_index    = 0;
  my $stack_position = 0;
  my @card_value     = shuffle(0..51);

  for(0..51) {
    my $image   = 'data/card_back.png';
    my $visible = 0;
    my $x       = 20;
    my $y       = 20;

    if($_ < 28) {
      if($stack_position > $stack_index) {
        $stack_index++;
        $stack_position = 0;
      }

      if($stack_position == $stack_index){
        $image
        = sprintf('data/card_%d.png',
                  $card_value[$_]);
        $visible = 1;
      }

      $x =  20 + 110 * $stack_index;
      $y = 200 +  20 * $stack_position;
      $stack_position++;
    }

    $layers->add(
      SDLx::Layer->new(
        SDL::Image::load($image),
        $x, $y,
        {id      => $card_value[$_],
         visible => $visible}
    ));
  }
}                          # 03-solitaire.pl
```

Listing 4

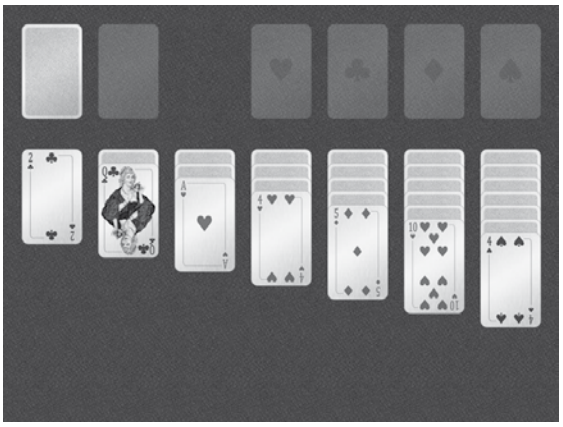


Abbildung 4: Laden der Kartengrafiken

sind sichtbar, diese erhalten das Bild des Kartenwertes, alle anderen werden verdeckt, also mit dem Bild des Kartenrückens dargestellt.

Wenn wir nun unser Skript ausführen, sieht es schon wie ein richtiges Solitaire aus. Die Karten werden Dank List::Util::shuffle natürlich zufällig verteilt.

Wie in einem vorherigen Kapitel erwähnt nutzen wir den Hash des Layers um diesen zu identifizieren. Dafür geben wir ihm als id den Kartenwert mit, wobei die Null das Herz-Ass, die Eins die Herz-Zwei und die 13 beispielsweise das Kreuz-Ass ist.

Mit dieser id wollen wir nun arbeiten. Um die kommenden Aufgaben zu bewältigen, müssen wir unter anderem herausfinden welche Karte sich bei einem Klick in dem Fenster unter dem Mauszeiger befindet.

Daraus ergeben sich zwei Problemstellungen, wir brauchen als erstes die Mauskoordinaten des Klicks und anschließend den Layer der an dieser Position ist. Da der Klick ein Event ist, nämlich einer vom Typ `SDL_MOUSEBUTTONDOWN`, sind diese Informationen in dem Eventobjekt gespeichert, welches im Eventloop beschrieben wird. Wir erinnern uns, die Funktion poll_event speichert dies in einer SDL::Event Struktur, in unserem Fall `$event`.

Die Koordinaten der Maus sind bei einem Klick in den Methoden `$event->button_x` und `$event->button_y` hinterlegt. Diese müssen wir also der nachfolgenden Routine übergeben.

Diese hält der Layermanager für uns bereit. Die Funktion by_position gibt uns zu einem Koordinatenpaar den passenden Layer zurück. Auf diesen können wir dann mittels der Methode data auf unseren Hash zugreifen, welcher beim Erzeugen des Layers angegeben wurde. Wir fügen also testweise folgendes mit in die `on_click` Subroutine ein:

```
printf("%s\n",
  $layers->by_position($event->button_x,
                       $event->button_y)
        ->data->{id}
)                      # 03-A-solitaire.pl
```

Auf diese einfache Art und Weise bekommen wir den Wert einer Spielkarte, bzw. „background“ wenn wir auf das Hintergrundbild klicken.

Vierter Schritt – wir verschieben eine Karte mit der Maus

Jetzt geht's ans Eingemachte, oder doch nicht? Normalerweise würde man wie folgt vorgehen: Wir klicken in unser Fenster, merken uns die Koordinaten und finden heraus, welcher Layer an dieser Stelle ist. Daraufhin müssten wir bei jedem Frame die neuen Mauskoordinaten herausfinden und den Layer an die neue Position zeichnen. Darüber hinaus müssten wir das für die anderen, zu bewegenden Karten, ebenfalls tun, um einen Stapel von Karten zu verschieben.

Dies können wir mit Hilfe vom Layermanager etwas abkürzen. Wir lassen uns den angeklickten Layer geben (`by_position`), anschließend übergeben wir diesen und die Layer die darauf liegen (ahead) an die Funktion attach. Dies bewirkt, dass der bzw. die Layer sich mit der Maus bewegen. Das einzige was wir tun müssen ist regelmäßig die Funktion blit des Layermanagers aufrufen, was in unserem Beispiel bereits in unserer Heartbeat-Schleife gemacht wird.

```
on_click     => sub {
  unless(scalar @selected_cards) {
    my $layer = $layers->by_position(
      $event->button_x, $event->button_y);

    if(defined $layer) {
      @selected_cards
      = ($layer, @{$layer->ahead});

      $layers->attach(
        @selected_cards,
        $event->button_x,
        $event->button_y);
    }
  }
},

on_drop      => sub {
  if(scalar @selected_cards) {
    for(@selected_cards) {
      $_->foreground;
      $_->detach_xy($_->pos->x,
                    $_->pos->y);
    }
    @selected_cards = ();
  }
},                       # 04-solitaire.pl
```

Listing 5

Wenn dann die Maustaste wieder losgelassen wird, rufen wir die Funktion `detach_xy` auf. Zusätzlich bringen wir die abzulegenden Karten in den Vordergrund (foreground) - siehe Listing 5.

Das war es im Grunde schon. Wir sehen die Kartengrafiken und können diese mit der Maus verschieben. Was brauchen wir mehr? Die Spielregeln, richtig.

Letzter Schritt – die Spiellogik

Die Spiellogik besteht in diesem Fall nicht in einer AI oder dergleichen. Wir müssen lediglich dafür sorgen, dass Karten nicht beliebig aufgenommen oder abgelegt werden können. Dazu kommt noch das Durchklicken des Kartenvorrats im Stapel oben links und das Ablegen der Karten per Doppelklick oben rechts. Diese Kontrollen werden wir jetzt im Ansatz durchführen.

Karten aufnehmen

In der Subroutine, die in dem Handler bei on_click aufgerufen wird, haben wir bereits eine Condition, die wir im Folgenden modifizieren und erweitern werden. Es wird bereits geprüft dass der angeklickte Layer definiert ist. Das ist gut, aber natürlich nicht ausreichend. Als nächstes müssen wir sicherstellen, dass eine Spielkarte angeklickt wurde. Dies geschieht mit der Überprüfung ihrer id. Wir erinnern uns, in der Funktion `init_cards` haben wir nur Spielkarten numerische ids vergeben, das können wir nun ausnutzen. Des Weiteren sollen aber Karten nur aufgenommen werden können, die aufgedeckt sind. Dies besagt die Eigenschaft visible. Wir fügen also entsprechende Conditions hinzu.

```
if(defined $layer
&& $layer->data->{id} =~ m/^\d+$/
&& $layer->data->{visible}) {
  ...
}                        # 05-solitaire.pl
```

Das war einfach, der nächste Schritt wird da schon anspruchsvoller.

Karten ablegen

Folgende Regeln gelten für das Ablegen der Karten:

1. Könige dürfen unten auf leere Felder gelegt werden.
2. Karten dürfen im unteren Bereich auf Karten gelegt werden, die den nächst-höheren Wert und die entgegengesetzte Farbe haben.

Bevor wir den ersten Fall angehen, benötigen wir aber noch den Layer, auf dem wir die Karte ablegen wollen. Das geht am einfachsten so:

```
my $drop_target
= $selected_cards[0]->behind->[0];
```

$selected_cards[0] ist die unterste Karte die gerade verschoben wird und behind->[0] ist die Karte die als erstes unterhalb dieser Karte auf dem Spielfeld liegt.

Nun können wir das drop_target auf dessen id und dessen Position, sowie die in der Hand gehaltene Karte auf deren Wert überprüfen. Anschließend legen wir die Karte genau auf das drop_target.

```
if($drop_target->data->{id}=~/empty_stack/
&& $drop_target->pos->y >= 200
&& $selected_cards[0]->data->{id}%13==12) {
  $layers->detach_xy(
    $drop_target->pos->x,
    $drop_target->pos->y);
}                          # 06-solitaire.pl
```

Der zweite Fall sieht schon etwas komplexer aus - siehe Listing 6.

Als erstes überprüfen wir, ob das drop_target eine numerische id hat, also eine Spielkarte ist. Diese muss natürlich aufgedeckt sein (visible). Der Wert der abzulegenden Karte muss um eins geringer sein, als die Karte auf die wir sie ablegen wollen. Zum Schluss prüfen wir noch, ob die Farben unterschiedlich sind.

Dabei sehen wir wieder, der farbenunabhängige Wert der Karte berechnet sich: id Modulo 13. So ergibt das für jedes Ass Null, für jede Zwei die Eins, für jeden König die Zwölf etc.

Die Kartenfarbe (Null bis Vier) erhalten wir, indem wir die id der Karte durch 13 teilen. Nach Modulo 2 erhalten wir für die roten Kartenfarben Null und für die schwarzen Eins. Dies prüfen wir auf Ungleichheit, sodass nur Karten unterschiedlicher Kartenfarben aufeinander gelegt werden können.

Für den Fall das keine der beiden Konditionen zutreffen muss die Karte wieder an den Ursprungsort zurück, dies geschieht wie folgt:

```
...
else {
  $layers->detach_back;
}                          # 06-solitaire.pl
```

Ablegen per Doppelklick

Den Doppelklick haben wir für die Funktion des Ablegens der Karten auf den Stapeln oben rechts reserviert. Wir müssen im Grunde nur prüfen, ob die Karte, die wir ablegen möchten, ein Ass und das Feld leer, oder genau der nächst-höhere Wert des gleichfarbigen Stapels ist. Wenn das der Fall ist, setzen wir die Position der angeklickten Karte (siehe Listing 7).

Nun sind nur noch kleine Punkte offen, die für das Gameplay nötig sind. Das bereits angesprochene Durchklicken des Kartenvorrats oben links und das automatische Aufdecken der Karten nachdem eine verschoben wurde.

Diese Routinen ähneln den bereits Gezeigten aber so sehr, dass ich hier nicht näher darauf eingehen möchte. Es sei nur so viel gesagt, dass die Codes des vollständigen Spiels auch diese Funktionen abdecken.

Das Spiel selbst wird auch in naher Zukunft auf CPAN veröffentlicht werden und einige Verbesserung erfahren. So sind zum einen die hier noch völlig außer Acht gelassenen Sounds, sowie eine Online-Highscore angedacht.

```
...
elsif($drop_target->data->{id} =~ m/^\d+$/
&& $drop_target->data->{visible}
&& $drop_target->pos->y >= 200
&& $selected_cards[0]->data->{id} % 13 + 1  == $drop_target->data->{id} % 13
&& int($selected_cards[0]->data->{id}/13) % 2 != int($drop_target->data->{id}/13) % 2) {
  $layers->detach_xy($drop_target->pos->x, $drop_target->pos->y + 20);
}                                                    # 06-solitaire.pl
```

Listing 6

```
on_dblclick => sub {
  $layers->detach_back;
  my $layer  = $layers->by_position($event->button_x, $event->button_y);

  if(defined $layer
  && !scalar @{$layer->ahead}
  && $layer->data->{id} =~ m/\d+/
  && $layer->data->{visible}) {
    my $target = $layers->by_position(370 + 110 * int($layer->data->{id} / 13), 40);

    if(($layer->data->{id} % 13 == 0 && $target->data->{id} =~ m/empty_target_/)
    || $layer->data->{id} - 1  == $target->data->{id}) {
      $layer->attach($event->button_x, $event->button_y);
      $layer->foreground;
      $layer->detach_xy($target->pos->x, $target->pos->y);
    }
  }
},                                                    # 07-solitaire.pl
```

Listing 7

Fazit

Spiele mit Perl zu programmieren ist nicht nur möglich, es sind sehr einfach. Und ich bin mir sehr sicher, dass von SDL in der nächsten Zeit noch viel mehr zu hören sein wird. Es sind mehrere vielversprechende Projekte im Gange, die für Schlagzeilen sorgen werden.

Join us today!

Renée Bäcker

Moose Tutorial - Teil 2: Methoden mit Moose

Im ersten Teil des Moose Tutorials ging es um Attribute von Klassen. Es wurde gezeigt, wie mit verschiedenen Optionen sehr generische Attribute erzeugt werden können. In dieser Ausgabe geht es um Methoden und welche Möglichkeiten man hat, Methoden mit Moose-Mitteln zu erweitern. Ab dieser Ausgabe werden alle Beispiele darauf ausgerichtet sein, eine Anwendung für Zeitschriften zu schreiben.

Inhalt der Anwendung soll es sein, eine Zeitschrift zusammenzustellen und dann zu publizieren. Dabei sollen mit jedem neuen Teil des Tutorials ein paar Sachen nach und nach verbessert und erläutert werden.

So eine "Zeitschrift" hat die verschiedensten Attribute wie in Listing 1 zu sehen ist. Auf die Attribute werde ich an dieser Stelle nicht näher eingehen.

Methoden in Moose-Klassen

Natürlich hat so eine Zeitschrift nicht nur Attribute, sondern auch noch einige Methoden. Die sind in Listing 1 noch ausgenommen und sind eigentlich an Stelle des `# mehr Code...` zu finden. Aber das Thema Methoden wird erst im Laufe dieser Ausgabe näher beleuchtet.

Grundsätzlich sehen die Methoden in Moose-Klassen genauso aus wie in Standard-Perl-Klassen (Listing 2).

Ein Beispiel-Skript ist in Listing 3 zu sehen. Darin wird ein Artikel-Objekt erzeugt und dann ein Zeitschriften-Objekt.

```
package FooApp::Zeitschrift;

use Moose;
use Moose::Util::TypeConstraints;

subtype 'Article' => as 'Object' => where { $_->isa( 'FooApp::Zeitschrift::Article' ) };
subtype 'ISSN'    => as 'Str'    => where { $_ =~ m{ \A [0-9]{4} - [0-9]{4} \z }xms };

has Articles => (
    is         => 'rw',
    isa        => 'ArrayRef[Article]',
    auto_deref => 1,
);

has ISSN    => ( is => 'ro', isa => 'ISSN' );
has Imprint => ( is => 'ro', isa => 'Str'  );
has Title   => ( is => 'ro', isa => 'Str', required => 1  );
has IssueNr => ( is => 'ro', isa => 'Int', required => 1  );

has PreferredOutput => ( is => 'ro', isa => 'Str', default => 'text' );

# mehr Code...

1;
```

Listing 1

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

    my $articles = '';
    $articles .=
     $_->render . "\n\n" for $self->Articles;

    my $text = sprintf "%s - %s\n%s\n\n\n%s",
        $self->Title,
        $self->IssueNr,
        $self->ISSN || '',
        $articles;

    return $text;
}
```

Listing 2

Das dann die folgende Ausgabe erzeugt:

```
$foo - Perl-Magazin - 16
1234-5678


Test1
----------
Text1
```

Soweit ist hier nichts Besonderes festzustellen. Es ist auch nichts von "modernem" Perl-Code zu sehen. Aber es gibt auch einen angenehmeren Weg mit Moose.

Lästige Schreibarbeit sparen

Wer die Module `MooseX::Declare` oder `MooseX::Method::Signatures` verwendet, kann sich viel der lästigen Schreibarbeit sparen: Damit muss man sich nicht mehr selbst darum kümmern, dass `$self` deklariert wird und auch default-Werte kann man in einer Signatur angeben.

```
method methodenname ( Int $farbwert = 255 ) {
    # $self ist hier bekannt...

}
```

Mit den Signaturen kann man auch bestimmen, von welchem Typ ein Parameter sein muss (hier: Int), ob der Parameter optional oder verpflichtend ist. Und durch das `method` weiß `MooseX::Declare`, dass hier ein Objekt als erster Parameter übergeben wird. Da die Variable für das Objekt von den meisten Perl-Programmierern `$self` genannt wird, sorgt `MooseX::Declare` dafür, dass `$self` in der Methode bekannt ist.

Dadurch spart man sich das

```
my ($self,$param1,...) = @_;
```

in den ganzen Methoden. Einzelheiten zu Methodensignaturen habe ich schon in Ausgabe 12 von $foo in einem extra Artikel beschrieben.

Multi-Methoden

Mit dem Modul `MooseX::MultiMethods` kann man das sogar noch weiterführen und etwas machen, das für Java-Programmierer schon seit jeher möglich ist: Mehrere Methoden mit dem gleichen Namen verwenden, die eine unterschiedliche Signatur haben.

Ein Beispiel wäre, wenn man eine Klasse hat, die eine Methode `show` hat. Ist der übergebene Parameter ein Objekt vom Typ `URI`, soll die Adresse im Browser angezeigt werden. Ist der Parameter aber vom Typ `FooApp::Zeitschrift::Article`, soll einfach der Titel und der Text ausgegeben werden (Listing 4).

Da in den Signaturen Klassennamen als Variablentypen genommen werden, die Moose nicht bekannt sind, muss man diese erst mittels `class_type` bekannt machen.

Danach können dann die einzelnen Methoden definiert werden. Wichtig ist das zusätzliche Wort `multi`. Ansonsten ist alles möglich, was `MooseX::Method::Signatures` auch anderen Methoden zur Verfügung stellt.

Methoden-Modifikatoren

Etwas Geniales bei Moose ist die Möglichkeit, Methoden dynamisch zu "modifizieren". Damit ist nicht das Ersetzen der Methode mit einer eigenen Methode gemeint, sondern dass eigener Code vor und/oder nach einer Methode ausgeführt

```
use FooApp::Zeitschrift::FooMagazin;
use FooApp::Zeitschrift::Article;

my @articles = (
    FooApp::Zeitschrift::Article->new(
        Titel  => 'Test1',
        Text   => 'Text1',
        Author => 'reneeb',
    ),
);

my $foo =
    FooApp::Zeitschrift::FooMagazin->new(
      Articles => \@articles,
      IssueNr  => 16,
      ISSN     => '1234-5678',
      Title    => '$foo - Perl-Magazin',
);

print $foo->publish;
```

Listing 3

wird. Ich habe das auch schon kurz in der Ausgabe 14 gezeigt, in der es um Plugins für `Devel::REPL` ging.

Diese Modifikatoren werden hauptsächlich bei Rollen verwendet (auf dieses Thema werde ich in der nächsten Ausgabe eingehen). Aber auch in den "normalen" Moose-Klassen kann man diese verwenden. Z.B. um noch etwas vor oder nach einer automatisch generierten Methode (z.B. Accessor für Attribute) zu machen.

Ich habe oben die Methode `publish` der Zeitschriften-Klasse gezeigt. Jetzt soll es aber so sein, dass beim FooMagazin nach dem `publish` automatisch eine Mail an die Abonnenten geht. Dabei soll das Testskript nicht angepasst werden.

```
package FooApp::Zeitschrift::FooMagazin;

use Moose;
extends 'FooApp::Zeitschrift';

after 'publish' => sub {
    print "benachrichtige alle " .
          "Abonnenten...\n";
};

1;
```

Wird jetzt das Testskript ausgeführt, wird nach der `publish`-Methode der Basisklasse die anonyme Subroutine ausgeführt, die bei hier in der Klasse angegeben ist.

```
package FooApp::MultiMethods;

use Moose;
use MooseX::MultiMethods;
use Moose::Util::TypeConstraints;
use Browser::Open qw(open_browser);

use URI;
use FooApp::Zeitschrift::Article;

BEGIN {
    class_type( 'URI' );
    class_type(
           'FooApp::Zeitschrift::Article' );
}

multi method show ( URI $uri ) {
    open_browser( $uri );
}

multi method show (
   FooApp::Zeitschrift::Article $article ) {
    print $article->Titel .
        "\n\n" .
        $article->Text;
}
```

Listing 4

Nach dem Schlüsselwort (`after`, `before`, `around`) wird der Subroutinenname angegeben, dessen Subroutine modifiziert werden soll. Gibt man hier einen Namen an, für den es keine entsprechende Subroutine gibt, wird schon während zur Compilezeit ein Fehler geworfen: `The method 'falscher_name' was not found in the inheritance hierarchy for xxx`.

Möchte man mehrere Methoden so verändern, kann man auch eine Liste von Subroutinennamen übergeben:

```
after qw(publish test hallo) => sub {
    print "Nach der Methode\n";
};
```

Allerdings kann man bei dieser Lösung in der anonymen Subroutine nicht feststellen, welche Originalmethode aufgerufen wurde.

Sollen alle Methoden modifiziert werden, die mit "issue" anfangen, kann man auch einen Regulären Ausdruck übergeben:

```
after qw/^issue/ => sub {
    # ...
};
```

Am Schluss übergibt man eine Subroutine, die dann vor bzw. nach der Originalmethode ausgeführt wird.

Bei `before` und `after` bekommt die anonyme Subroutine das Objekt übergeben, sowie die Parameter für die Originalmethode. Leider gibt es keine Möglichkeiten, bei `before` die Parameter anzupassen oder bei `after` den Rückgabewert zu manipulieren.

Auch die Ausführung der Originalmethode kann man bei `before` nicht verhindern. Man könnte ja eigentlich auf die Idee kommen, mit `before` eine Validierung von Parametern einzuführen und bei einem Fehlschlag die eigentliche Methode gar nicht auszuführen. Das ist so leider nicht möglich.

Für solche Dinge muss man sich mit `around` beschäftigen - oder `die` in dem `before` benutzen.

Bei `around` wird zusätzlich noch die Code-Referenz auf die Originalmethode übergeben:

```
around 'publish' => sub {
    my ($code,$self,@params) = @_;
    print "before\n";
    my $text = $self->$code( @params );
    print $text;
    print "after\n";
};
```

So man dann auch die Möglichkeiten, die Parameter bzw. den Rückgabewert der Originalmethode zu verändern.

Für die nächsten Tests verwenden wir eine neue Klasse:

```
package FooApp::Test;

use Moose;

sub test {
    print "test";
}

1;
```

Bei `before` und `after` sollte man darauf verzichten, den Originalcode in der anonymen Subroutine aufzurufen Das führt zu einer unendlichen Rekursion.

```
before 'test' => sub {
    my ($self) = @_;

    print "before\n";
    $self->test;
};
```

liefert

```
[...ganz häufig 'before'...]
before
before
before
Deep recursion on anonymous subroutine
    at C:/lib/FooApp/Test.pm line 13.
Deep recursion on anonymous subroutine
    at C:/lib/Class/MOP/Method/Wrapped.pm
    line 89.
Deep recursion on anonymous subroutine at
    C:/lib/Class/MOP/Method/Wrapped.pm line
    47.
```

Wer viele Klassen implementiert, möchte vielleicht mit jeder Klasse die selbe Methode etwas "verändern". So könnte eine Klasse `FooApp::Zeitschrift::FooMagazin` von `FooApp::Zeitschrift` erben und eine Klasse `FooApp::Zeitschrift::FooMagazin::SpecialEdition` von `FooApp::Zeitschrift::FooMagazin`. Die Methode `publish` der Klasse Zeitschrift kennen wir bereits. FooMagazin möchte nach dem `publish` noch die Abonnenten informieren, dass es eine neue Ausgabe gibt. Es gibt also ein `after 'publish' => sub {...}`.

SpecialEdition möchte neben der Abonnenten-Information noch eine Pressemitteilung an Buchhandlungen und News-Sites mailen. Auch hier wird das nach dem `publish` gemacht, also ebenfalls ein `after 'publish' => sub {...}`.

Die anonyme Subroutine aus der FooMagazin-Klasse wird vor der anonymen Subroutine aus SpecialEdition ausgeführt. Bei den `before`-Einstellungen ist es genau anders herum. Bei `before` und `around` wird die zuletzt registrierte Subroutine zuerst ausgeführt. Bei `after` wird die zuerst registrierte Subroutine auch zuerst ausgeführt.

Wer sich etwas mehr verschachtelte Methodenmodifizierer anschauen möchte, kann das Skript `test2.pl` in den Source-Codes zu dieser Ausgabe anschauen. Dort gibt es drei Klassen:

- **Test**: Hier gibt es nur die Methode `test` (T)
- **MiddleClass**: Hier gibt es die Modifizierer `before` (B-M-1), `after` (A-M-1) und `around` (R-M-1) in dieser Reihenfolge.
- **EndClass**: Hier gibt es `before` (B-E-1), `after` (A-E-1), `around` (R-E-1), `after` (A-E-2) und wieder `before` (B-E-2).

Die Werte in den Klammern sollen einfach nur Kurznamen sein, um die Ausführungsreihenfolge übersichtlicher zu gestalten. `B-M-1` steht dabei für `before` in der Klasse `MiddleClass` und es ist das erste `before`, dass es in der Klasse gibt. Alle dies Modifizierer sind für die Methode `test` aus der Klasse `test` eingerichtet.

Mit dieser Einstellung gibt es folgende Ausführungsreihenfolge:

```
B-E-2
B-E-1
R-E-1
  B-M-1
  R-M-1
    T
  A-M-1
A-E-1
A-E-2
```

Wie man sieht werden erst die `before`s und die `around`s der "untersten" Klasse ausgeführt, bevor es zur nächst "höheren" Klasse geht. Und bei den `after`s geht das rückwärts.

Methodenaufrufe weiterreichen

Für eine Schnittstelle zu einem externen Programm müssen die Daten für die Zeitschrift im XML-Format vorliegen. Diese sollen "von außen nach innen" zusammengeführt werden. In der Subklasse sollen in dem Fall keine Funktionen von Superklassen aufgerufen werden.

Für diesen Fall gibt es `augment` und `inner`. In der Superklasse `FooApp::Zeitschrift` fügen wir die Subroutine `as_xml` ein:

```
sub as_xml {
    my ($self) = @_;

    my $inner = inner();
    return sprintf
     '<zeitschrift>%s</zeitschrift>', $inner;
}
```

In der FooMagazin-Klasse müssen wir `Moose` sagen, dass es hier die `as_xml`-Funktion erweitert werden soll. Hierzu wird `augment` verwendet.

```
augment 'as_xml' => sub {
    my ($self) = @_;

    my $title  = $self->Title;
    my $issn   = $self->ISSN;

    my $format = <<'    FORMAT';

    <titel>%s</title>
    <issn>%s</issn>
    FORMAT

    my $xml = sprintf $format, $title, $issn;
    return $xml;
};
```

Wenn jetzt `as_xml` für ein Objekt der FooMagazin-Klasse aufgerufen wird, wird die Methode in der Superklasse aufgerufen. Der `inner`-Aufruf ruft die anonyme Subroutine, die bei `augment` angegeben wurde. Damit ergibt sich die Ausgabe:

```
<zeitschrift>
    <titel>$foo - Perl-Magazin</title>
    <issn>1234-5678</issn>
</zeitschrift>
```

Diese `inner`-Aufrufe können auch in der Subklasse eingebaut werden:

```
augment 'as_xml' => sub {
    my ($self) = @_;

    my $title  = $self->Title;
    my $issn   = $self->ISSN;
    my $inner  = inner();

    my $format = <<'    FORMAT';

    <titel>%s</title>
    <issn>%s</issn>
    %s
    FORMAT

    my $xml = sprintf $format, $title,
              $issn, $inner;
    return $xml;
};
```

In der SpecialEdition-Klasse wird dann folgender `augment`-Aufruf eingebaut:

```
augment 'as_xml' => sub {
    my ($self) = @_;

    my $xml =
       sprintf '<info>%s</info>',
                $self->Info;
    return $xml;
};
```

Dann wird für ein SpecialEdition-Objekt auch die Zusatzinfo ausgegeben. `Moose` ist dabei so schlau, dass es immer nur bis zu der Ebene heruntergeht, zu der auch das Objekt gehört. Also wenn ich die `as_xml`-Methode eines Zeitschriften-Objekts aufrufe, wird nur der `as_xml`-Code der Klasse Zeitschrift ausgeführt. Das `inner()` erzeugt also keinen Fehler. Rufe ich die `as_xml`-Methode eines FooMagazin-Objekts auf, wird der Code aus der Klasse Zeitschrift und durch das `inner()` auch der Code aus der Klasse FooMagazin ausgeführt.

Methoden überschreiben

Bei der Vererbung werden Methoden mitvererbt. Dadurch kennt das FooMagazin-Objekt die Methode `publish`, obwohl diese nur in der Superklasse definiert wird. Manchmal muss man eine Methode in der Subklasse aber überschreiben.

Natürlich kann man einfach

```
sub publish {
    my $self = shift;
    print "in " . __PACKAGE__ . "\n";
}
```

schreiben. Dann müsste man aber

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

schreibe, um die `publish`-Methode der Superklasse aufzurufen. `Moose` bietet hierfür etwas, das viel "netter" aussieht: `override` und `super`.

Mit `override` kann man `Moose` mitteilen, dass eine Methode überschrieben werden soll. Benutzt man diese Möglichkeit, kann man `super()` nutzen, um die Methode der Superklasse aufzurufen. Das würde dann wie folgt aussehen:

```
override 'publish' => sub {
    my $self = shift;
    print "in " . __PACKAGE__ . "\n";
    super();
}
```

Ausblick

In diesem Artikel wurden einige Möglichkeiten in Bezug auf Methoden deutlich gemacht. In der nächsten Folge des Moose Tutorials werde ich auf das Thema Vererbung eingehen.

Herbert Breunung

WxPerl Tutorial - Teil 5: Fenster

Vorschau und Ergänzung

Bereits vor einem Jahr öffnete das erste Beispiel dieses Tutorials ein Fenster, doch diese Folge wird sich genauer diesem Thema widmen. Natürlich mit Ausnahmen, da bereits im zweiten Teil die genaue Arbeitsweise von Wx::Dialog erklärt wurde und die Möglichkeiten von Wx::Frame erst nächstes mal Thema werden, wenn eine erste Applikation gebaut wird. Ein Schwerpunkt wird dieses mal `Wx::AUI` sein, was an die letzte Folge anknüpft und besonders Programmierern zu empfehlen ist, welche sich mit den im dritten Teil vorgestellten Sizern nicht anfreunden mögen.

Alle Wx-Nutzer freuen sich vielleicht, dass die Perl Foundation die Verbesserung der WxPerl-Dokumentation unterstützt. (siehe h*ttp://news.perlfoundation.org/2010/06/grants-update-wxperl-documenta.html*) Stolz macht mich, dass Eric Roode als Index seiner Arbeiten meine WxPerlTafel (*http://wxperl.pvoice.org/w/index.php/WxPerlTablet*) benutzt, von der es auch unter *http://wiki.perl-community.de/Wissensbasis/WxPerlTafel* eine deutsche Version gibt. Es ist eine Schnellübersicht zu den wichtigsten Wx-Themen.

Mutter aller Fenster

Wer mehr über Fenster wissen will, sollte auch ihr Oberhaupt näher kennen. Die Applikation ist quasi der Wurzelknoten aller Ereignissteuerung. Das erste Beispiel bestand im wesentlichen aus einer Klasse, die von `Wx::App` abgeleitet wurde, um deren `OnInit` Methode überschreiben zu können, bevor wir eine Instanz dessen bildeten und diese starteten (`MeineKlasse->new()->MainLoop;`). Manchmal muss man auch die App mit `$app->ExitMainLoop` selber runterfahren. Die Referenz dazu bekommt man von `wxTheApp` oder besser `$Wx::wxTheApp` oder als ersten Parameter der Klassenmethoden wie `OnInit`. Nach dem `ExitMainLoop` wird natürlich `OnExit` ausgeführt, welches manchmal nützlich ist. Ganz im Gegensatz zu `OnRun`, welches unter Perl keinen Effekt hat. Stattdessen empfehle ich den `EVT_IDLE`, weil dann gleichzeitig auch klar ist welches Widget grad Rechenzeit braucht. Nach Beendigung wird ein Idle-Signal ordnungsgemäß die Objekthierarchie hoch gereicht. Empfehlenswert ist auch ein `$app->Yield` welches die ausstehenden Events sofort ausführt. Ich hatte z.B ein Icon in der Werkzeugleiste das eine sehr aufwendige Routine auslöste. Ohne `Yield` sähe das Icon noch während der Berechnung heruntergedrückt aus, selbst wenn der Benutzer es schon lange zuvor losgelassen hat. Da Wx nun mal für C++-Programmierer entworfen wurde, ist hier sehr vieles für uns nicht wirklich nützlich. Die Kommandozeilenparameter an die App hat Mattia zum Glück wie gewohnt in `@ARGV` abgelegt, kein `argc` und `argv` nötig. Auch `GetClassName` ist überflüssig, soweit man mit `SetClassName` nichts selber speicherte und `GetAppName` entspricht `__PACKAGE__`.

'Fenster' ne 'Window'

Hört jemand "Fenster", mag er denken, es geht um `Wx::Window`. Das wäre allerdings ein Mißverständnis, da diese Klasse den Reichtum an Methoden beherbergt, die alle sichtbaren Objekte (Widgets) gemein haben. Früher, als WxWidgets noch WxWindows hieß, war das einleuchtender.

Topfenster

Die bunten, mit der Maus verschiebbaren Rahmen, innerhalb derer sich die App, oder wenigstens eine Funktionseinheit abspielt, die wir als Fenster bezeichnen, heißen im Wx-Jargon `TopLevelWindow`. Über diese Klasse können wir die Größe, das Icon, die Titelleiste und Darstellungsmodus, des Fensters festlegen. Diese Fähigkeiten erben `Wx::Frame`, `Wx::MiniFrame`, `Wx::Dialog`, `Wx::Wizard` und die MDI-Fenster, die Thema eines der nächsten Absätze werden.

Das `Wx::Frame` ist meist das Hauptfenster einer Applikation. Es hat Methoden die den Bau oder Anbindung von Werkzeugleisten, Menüleisten und Statuszeilen vereinfachen. Diese werden Hauptthema der nächsten Folge.

Das `MiniFrame` ist wirklich die minimalste Lösung und an der besonders schmalen Titelleiste erkennbar. Es wird z.B. sichtbar wenn ein Teil einer Werkzeugleiste sich abkoppelt, um an anderer Stelle wieder angedockt zu werden. Es lässt sich aber auch gut als sparsames Hauptfenster verwenden.

Das gilt nur bedingt für den `Wx::Dialog`. Er bringt zwar schon ein `Panel` mit (siehe letzte Folge) und würde damit 2, 3 Zeilen einsparen, aber ohne ein

```
EVT_CLOSE ($dialog,
  sub { $dialog->Destroy() } );
```

würde die App nach Beendigung hängen, da Dialoge nur unsichtbar werden und auf Wiederverwendung ausgerichtet sind. Mehr dazu gab es in Folge 2.

Auch der `Wx::Wizard` ist eine Art Dialog, der bei komplexen Aufgaben helfen soll. Da er heute kaum Verwendung findet, sei er hier nur der Übersicht wegen erwähnt.

Der schnelle Bildschirm

Im weiten Sinne ist auch der `SplashScreen` ein Fenster, wenn auch ohne Ränder. In Wx kann er leider keine Animationen abspielen, sondern nur einfache Bilder (`Wx::Bitmap`) zeigen, verschwindet wenn angeklickt und wird meist etwa so gebildet - siehe Listing 1.

Statt es mit `wxSPLASH_CENTRE_ON_SCREEN` mitten auf den Bildschirm zu setzen, geht auch `wxSPLASH_CENTRE_ON_PARENT` sofern anstelle `undef` ein Elternfenster vorher erzeugt und angegeben wurde, oder man sagt `wxSPLASH_NO_CENTRE` und gibt die Position anstelle `wxDefaultPosition` direkt an. Möglich ist auch ein `wxSPLASH_TIMEOUT`, dass nach $n Millisekunden (anstelle der 0) das Bild ohne `Destroy` ausblenden lässt. Das kann bequem sein, aber auf einem sehr schnellen Rechner kann das unnötig stehende Bild lästig werden, auch wenn es mit einem Klick darauf verschwindet.

Leider erscheint dieser `SplashScreen` nicht sofort, sondern nach Initialisierung der `Wx::App`, was in manchen Fällen als zu spät empfunden wird. Für den Fall gibt es `Wx::Perl::SplashFast`, dass auch durch seine kompakte Syntax gefällt. Es startet eine eine kleine Dummy-App die lediglich einen `SplashScreen` zeigt.

```
use Wx::Perl::SplashFast
      ('/pfad/zum/logo.jpg',3000);
```

Diese Zeile kann noch vor einem `use Wx;` stehen. Nur wenn man es kontrolliert statt zeitgesteuert den Splashscreen beenden möchte, muss es schon etwas mehr Code sein:

```
my $sc;
BEGIN {
    require Wx::Perl::SplashFast;
    $sc = Wx::Perl::SplashFast->new
              ('/pfad/zum/logo.jpg');
}
...
$sc->Destroy();
```

```
my $sc = Wx::SplashScreen->new(
       Wx::Bitmap->new( $bild_datei_pfad, wxBITMAP_TYPE_ANY ),
       wxSPLASH_CENTRE_ON_SCREEN | wxSPLASH_NO_TIMEOUT,
       0, undef, -1, wxDefaultPosition, wxDefaultSize,
       wxSIMPLE_BORDER | wxFRAME_NO_TASKBAR | wxSTAY_ON_TOP
);
...
$sc->Destroy();
```

Listing 1

Fenstereigenschaften

Folgendes sollte man unbedingt über `TopLevelWindow` wissen. Einige Eigenschaften der Fenster wie etwa Titelleiste und die Art der darin enthaltenen Knöpfe, sowie Veränderbarkeit der Größe müssen gleich zu Beginn mit dem *Style*-Parameter angegeben werden. Steht dort eine -1 so entspricht das `wxDEFAULT_FRAME_STYLE` was wiederum einem `wxMINIMIZE_BOX | wxMAXIMIZE_BOX | wxRESIZE_BORDER | wxSYSTEM_MENU | wxCAPTION | wxCLOSE_BOX | wxCLIP_CHILDREN` entspricht. Ein nachträgliches Ändern wie etwa mit:

```
$frame->SetWindowStyle
        ($frame->GetWindowStyleFlag()
         & ~&Wx::wxRESIZE_BORDER);
```

wurde bereits im zweiten Teil vorgestellt. Was jetzt an dem Fenster stören mag ist das unansehnliche Platzhalter-Icon. Bis ihr etwas passendes findet geht auf jeden Fall immer:

```
$fenster->SetIcon( Wx::GetWxPerlIcon );
```

später dann sicher ein:

```
Wx::InitAllImageHandlers();
$frame->SetIcon(
      Wx::Icon->new
        ( $icon_pfad, &Wx::wxBITMAP_TYPE_ANY)
);
```

Für das Laden von Bildern gibt es je Typ einen Handler. Die für ".ico" oder "*.xpm" sind immer mit dabei, aber damit es keine Probleme mit "*.gif", "*.jpg" oder "*.png" gibt, hab ich `Wx::InitAllImageHandlers();` hinzugefügt. Mehr zu Bildern im nächsten Teil.

Das dem Fenster eine Mindest- (`SetMinSize`) oder Maximalgröße (`SetMaxSize`) zugewiesen werden kann wurde auch schon erwähnt. Das Verändern der Titelleiste (`SetTitle`), die Durchsichtigkeit (`SetTransparent(0-255)` 0=Unsichtbar), falls vom OS unterstützt (mit `CanSetTransparent` abfragen), oder ein Vollbildmodus (`ShowFullScreen`) sind ebenfalls trivial.

Falls der Benutzer grad woanders beschäftigt ist, also `IsActive` unwahr gibt, kann ein Fenster auch mit `RequestUserAttention` auf sich aufmerksam machen, was meist einem Blinken des Reiters in der Taskleiste entspricht.

Mehrere Fenster und Ereignisse

Natürlich kann eine App mehrere Hauptfenster haben, mit wiederum beliebig vielen Kindfenstern. Ein Kind entsteht, wenn bei der Erzeugung als erster Parameter ein anderes Fenster (Elter) angegeben wird. Bleibt dieser Parameter `undef`, wird es ein Fenster erster Ordnung. Die App wartet das Schließen (`Destroy`, nicht `Show(0)` oder `Hide`) aller dieser Fenster ab bevor sie sich beendet und die Methode `$app->OnExit` auslöst. Außer man hat mit `$app->SetExitOnFrameDelete(0)` dieses Verhalten augesetzt. `$app->SetTopWindow($frame1);` hat damit nichts zu tun und ist lediglich ein optischer Befehl der ansagt, welches Fenster die anderen überlagern darf.

Das Schließen eines Fensters beendet auch gleichzeitig alle Kinderfenster. Will man unbedingt vorher noch etwas ausgeführt haben, so sollte der Abbau des Fensters mit `EVT_DESTROY($fenster, $ID, $callback)` abgefangen werden, da `EVT_CLOSE($fenster, $callback)` nur ausgelöst wird wenn der Nutzer das Fenster direkt schließt (oder ein `$fenster->Close();`-Befehl erteilt wird). Dann könnte ein fehlendes `$event->Skip` oder besser noch ein `$event->Veto` in der Callback-Routine die Schließung verhindern. Wie normale Widgets, kennen auch Fenster `EVT_SIZE`, das während Größenänderungen ausgelöst wird, sowie `EVT_SET_FOCUS` und `EVT_KILL_FOCUS`, falls der Fokus die Fenster wechselt. Fenster hören aber noch zusätzlich auf das Ereignis `EVT_MOVE` das während Bewegungen 100 mal je Sekunde ausgelöst wird, weswegen oft `EVT_MOVE_START` und `EVT_MOVE_END` ausreichend sind.

Viele Fenster in Einem

Benutzt ein Programm mehrere Fenster, gibt es das Problem der Zuordnung. Dialoge und `MessageBox`'s frieren die anderen Fenster ein (siehe Folge 2), aber wenn Fenster zusammenarbeiten sollen, ist das keine Option. In den 90'ern gab es vor allem unter Windows, wo es bis heute keine virtuellen Arbeitsflächen gibt, dazu den Lösungsansatz, dass alle Fenster eines Programmes sich nur innerhalb eines Oberfensters aufhalten dürfen. Das nannte man "Multiple Document Interface", kurz MDI. Wer das mal probieren will, dem reicht ein schlichtes `use Wx::MDI;`. Im nächsten Schritt erzeugt man anstatt eines `Wx::Frame` ein `Wx::MDIParentFrame`

und die Kinder als `Wx::MDIChildFrame`. Die werden innerhalb eines Bereiches im Fenster angeordnet der ähnlich einem `ScrolledWindow` Scrollbalken hat und eine virtuelle Fläche, die größer als der Bildschirm selber werden kann. Die Tasten "Strg+Tab" lassen zwischen den Fenstern wechseln. Auch der Rest der MDI-API ist denkbar einfach und die Technik wird oft als veraltet angesehen, weswegen ich es dabei belasse.

Die Neue Fensterordnung

Ein anderes 3-Buchstaben-Akronym ist AUI. Es steht für "Advanced User Interface" und ist eine Bibliothek die von einem Unternehmen geschrieben wurde um Wx um mehrere Fähigkeiten zu erweitern, die vorher nur umständlich zu bekommen waren, um die Optik an einigen Stellen aufzufrischen und um die Gestaltung des Gesamtlayout zu vereinfachen. Freundlicherweise wurden die Quellen freigegeben und sind seither als Wx::AUI auch Teil von WxPerl. Den schnellsten Ausflug in dieses Gebiet lässt mich einem Notebook machen. Zuerst das erwartbare `use Wx::AUI;` zufügen und in der Zeile der Erzeugung `Wx::Notebook` mit `Wx::AuiNotebook` austauschen. Gut, auch die Ereignisse haben erweiterte Namen a la `EVT_AUINOTEBOOK_PAGE_CHANGING` und die Style-Konstanten beginnen mit `wxAUI_NB` statt `wxNB`. Es gibt hier derer auch einige mehr, aber es genügt `wxAUI_NB_TAB_MOVE` zu setzen, um in den Genuß von beweglichen Reitern zu kommen. Wo die Knöpfe zum Schließen der Tabs sind läßt sich durch die Vergabe von `wxAUI_NB_CLOSE_ON_ALL_TABS`, `wxAUI_NB_CLOSE_ON_ACTIVE_TAB` oder `wxAUI_NB_CLOSE_BUTTON` (am rechten Rand) steuern. Mit `wxAUI_NB_WINDOWLIST_BUTTON` kann man auch ein Menü mit allen Tab-Titeln bekommen. Ein weiterer Vorteil dieser Leisten ist: den Reitern kann ein Icon ohne den Umstand einer `Wx::ImageList` zugewiesen werden. Es reicht ein:

```
$book->SetPageBitmap($nr, $bitmap);
```

Aber so richtig interessant wird es mit `wxAUI_NB_TAB_SPLIT`, welches das Spalten (vertikal oder horizontal) der Reiterleiste erlaubt. Der Benutzer kann den Reiter samt zugehörigen Panel aus der Leiste ziehen und daneben eine neue Reiterleiste damit eröffnen oder in eine solche einfügen. Beide Reiterleisten zählen aber als ein AUIPanel. Erst mit `wxAUI_NB_TAB_EXTERNAL_MOVE` kann der Reiter die Panel zu einer getrennt erzeugten Leiste wechseln.

Wie angedeutet arbeitet AUI mit Panel (siehe letzt Folge) welche die beschriebenen Eigenschaften haben können. Um diese zu benutzen bedarf es aber etwas mehr des Aufwandes. Als erstes brauchen wir dazu einen AUIManager der leicht geschaffen ist.

```
our $manager = Wx::AuiManager->new();
```

Das `our` ist sehr wichtig, da im Hintergrund bei jeder Umdockaktion alles geprüft und gespeichert werden muss. Würde ich den Manager mit `my` als lexikalisch lokal für die `sub` deklarieren (lokal für das `package` würde gehen), hätte AUI nicht mehr den gewünschten Zugriff und es käme spätestens während des Beendens der App zu unerklärlichen Abstürzen. Als nächstes weisen wir dem Manager ein Fenster zu:

```
$manager->SetManagedWindow( $frame );
```

Nun können wir gewöhnlich Panel (Teil4) mit Widgets bestücken und dies gut mit Sizern darauf festzurren. Einzelne Widgets nimmt der AUI-Manager aber auch. Entscheidend ist das `Wx::AuiPaneInfo`:

```
$manager->AddPane (
      $window,
      Wx::AuiPaneInfo->new->.....
);
```

Hinter dem `new` können wir beliebig viele Methoden hängen, eine jede wird eine Eigenschaft dieses Widgets bestimmen. Die Methodennamen wie: `BottomDockable`, `Floatable`, `Movable`, `Resizable`, `Caption("...")`, `CaptionVisible`, `Position( $n )` sowie viele weitere sind dabei selbsterklärend. Nur eins sollte man am Ende, nach allen `AddPane` nicht vergessen:

```
$manager->Update;
```

Es gibt auch die AuiPaneInfo-Methode `ToolbarPane` die ich vor allem benutze, wenn ich dem `$manager` `Wx::Toolbar` gebe, die dann zu frei verschiebbaren Werkzeugleisten-Stücken werden. Aber das ist bereits das Thema der nächsten Folge und soll ein anderes Mal erzählt werden. Diese Geschichte endet hier aber noch nicht, da das *AUI* noch sehr schöne Möglichkeiten der Introspektion und des Marshalling hat. Introspektion bedeutet hier, ich kann jederzeit prüfen, wo ein Widget angedockt ist und welche sonstigen Eigenschaften es jetzt hat.

```
$auiInfo = $manager->GetPane($widget);
```

Mit all den Methoden die mit "Is..." anfangen können Fragen gestellt werden oder mit den Restlichen das Verhalten zur Laufzeit geändert werden. Nur nicht danach `$manager->Update;` vergessen.

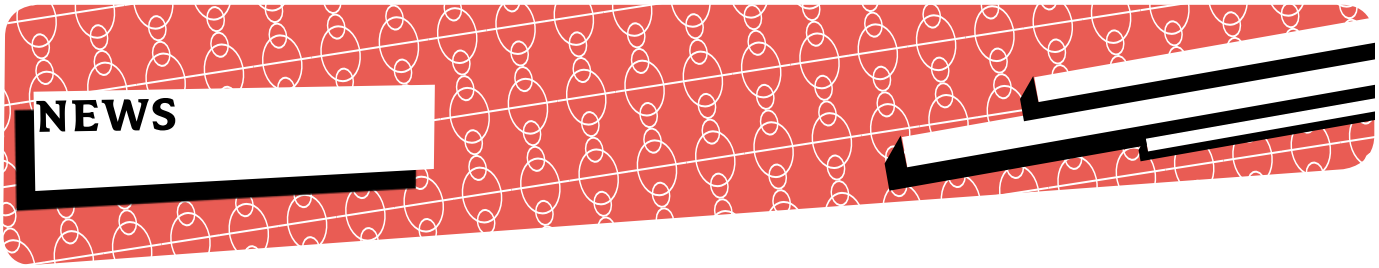
Marshalling bezeichnet das Speichern und wieder Herstellen eines Zustandes, auch an einem anderen Ort, hier wohl in einem anderen Fenster.

```
my $string = $manager->SavePaneInfo
      ($auiInfo);
# ... anderswo, andere Zeit
$manager->LoadPaneInfo
      ($string,  Wx::AuiPaneInfo->new());
```

Das war Marshalling für ein Widget, die Wiederherstellung des gesamten Fensters:

```
my $string = $manager->SavePerspective();
...
$manager->LoadPerspective($string, 1);
```

Die 1 spart das `$manager->Update;` für sofortige Sichtbarkeit der Änderung. Vergesst bitte nur nicht, dass so nur die Anordnung wiederhergestellt wird. Die Erzeugung der Widgets, deren Zuweisung an den Manager und Wiederherstellung der Widgetinhalte muss vorher geschehen.



Renée Bäcker

Plat_Forms Contest 2011

Nach 2007 wird es im Januar 2011 die zweite Auflage des Plat_Forms Contest geben. Es wäre sehr schön, wenn sich wieder drei Teams finden würden, die daran teilnehmen.

Plat_Forms ist ein Wettbewerb, in welchem Gruppen der Spitzenklasse aus drei Programmierern gegeneinander antreten, um die gleichen Maßgaben an ein webbasiertes System innerhalb von zwei Tagen umzusetzen. Dabei kommen unterschiedliche Technologieplattformen zum Einsatz (z.B. Java, .NET, Perl, PHP, Python, Ruby, Scala, Smalltalk usw.).

Der Sinn ist es, nicht die beste Webentwicklungsumgebung zu bestimmen, sondern neue Erkenntnisse über die echten (statt behaupteten) Vor- und Nachzüge und fulgurativen Eigenschaften jeder Plattform zu erlangen. Die Bewertung untersucht viele Gesichtspunkte jeder Lösung, sowohl äußerliche (Nutzbarkeit, Funktionsweise, Zuverlässigkeit, Sicherheit, Leistung usw.) als auch innerliche (Aufbau, Modularität, Verständlichkeit, Anpassungsfähigkeit usw.).

Vielleicht finden sich unter unseren Lesern ja komplette Teams (die aus 3 Personen bestehen) oder Einzelpersonen, die gerne an dem Wettbewerb teilnehmen würden. Die Teams melden sich am Besten direkt auf der Mailingliste des Contests (siehe http://www.plat-forms.org). Einzelpersonen können sich an plat_forms2011@perl-magazin.de wenden und wir werden versuchen, Kontakte zwischen den Einzelpersonen herzustellen.

Rolf Langsdorf

Wie erweitere ich Perls Syntax? - Teil 1 "List Comprehensions"

Oft bekommt man erzählt wie toll dieses Sprachfeature oder jenes Sprachkonstrukt sei, zu dem Perl5 "nicht wirklich" in der Lage sei. Man bekäme es zwar *semantisch* umgesetzt, aber die notwendige *Syntax* sei "zu umständlich", "wenig intuitiv", "zu Fehleranfällig" oder einfach "nur hässlich".

Andererseits bietet Perl dank der diversen unterstützten *Paradigmen* und Möglichkeiten des *Syntactic Sugar* eine Fülle an Kombinationsmöglichkeiten, um den Sprachumfang zu erweitern. Das CPAN ist dementsprechend auch nicht arm an Modulen, die mit einer Vielzahl an Featuren glänzen ... aber dennoch nur wenig genutzt werden!

Woran liegt das? Um uns dem Komplex systematisch zu nähern, versuchen wir im folgenden ein konkretes Idiom beispielhaft umzusetzen, Schwachstellen auszumachen und schrittweise zu verbessern.

Dabei wollen wir nicht nur verschiedene Techniken und Kniffe ausprobieren, sondern dabei Thesen formulieren, was den nun eine *zufriedenstellende* Erweiterung ausmacht. Die vorgestellten Kriterien sollen dabei nicht nur als Diskussionsgrundlage sein, sondern idealerweise auch künftigen Modulentwicklern eine Orientierung geben.

Gleich vorangestellt unser erstes Kriterium:

These: *Akzeptable Extensions setzen nur "etablierte" CORE Features voraus!*

Die Scheu vieler Programmierer neue Features im Code zu nutzen, liegt hauptsächlich in fehlender Rückwärtskompatibilität begründet. Der Installationsstand vieler Produktivsysteme kann nämlich um Jahre zurückliegen und die Hürden zur Nachinstallation von Modulen können erheblich sein.

Wir stützen uns deshalb im folgenden nur auf Standardpakete die schon seit Jahren bereits zum Installationsumfang gehören. Daraus ergibt sich dann auch:

These: *XS-Erweiterungen - sprich die Kompilierung und Einbindung neuen C-Codes - ist kein primärer Lösungsansatz! Diese können nur Option für spätere Geschwindigkeitsoptimierungen und nicht Vorraussetzung sein.*

Werden wir konkret...

List Comprehensions

"List Comprehensions" beschreiben die Möglichkeit, die aus der Mathematik bekannten Klauseln der Mengenschreibweise mit in einer Programmiersprache idiomatisch nachzubilden.

Beispielsweise die Aufgabe "Ermittle die Liste aller ganzzahlige Lösungen des Satzes von Pythagoras bis zu einer Größe n" kann algebraisch kompakt notiert werden:

$$\{ \forall a,b,c \in \mathbb{N} \mid c<n \ , \ a^2 + b^2 = c^2 \ \}$$

(Diese sogenannten "Pythagoreischen Tripel" finden sich in der 9. Programmieraufgabe bei "projecteuler.net")

Haskell (und laut Englischem Wikipedia noch 15 weitere Sprachen) erlaubt nun die obige Notation kompakt nachzubilden:

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

(Tatsächlich ist es ein Einzeiler, der Umbruch ist dem Format geschuldet!)

Es ist sogar ohne Weiteres möglich nur die ersten 5 Lösungen für eine "unendlichen" Lösungsmenge anzufordern, wenn wir `n` weglassen und ein `take` voransetzen.

```
take 5 [(a, b ,c) | c<-[1..], b<-[1..c] etc.]
```

Letzteres in Perl nachzubilden erfordert schon etwas mehr als eine triviale 3 fach Schleife:

```
use constant
  MAXRANGE => ((~0-3)/2);

my @tripel;
my $takes=4;
TRIPEL:

for my $c (1..MAXRANGE) {
  for my $b (1..$c) {
    for my $a (1..$b) {
      if ( $a**2 + $b**2 == $c**2 ) {
        push @tripel, [$a,$b,$c];
        last TRIPEL  unless --$takes;
      }
    }
  }
}
```

Listing 1

Ein Dump von `@tripel` ergibt dann auch:

```
[[3,4,5],[6,8,10],[5,12,13],[9,12,15]]
```

Weder `map`, `grep` noch postfix-`for`s helfen eine schönere Notation zu finden. Man vergleiche hierzu auch die Diskussion bei *http://www.perl-community.de/bat/poard/thread/12655*.

Einige Besonderheiten gilt es im Code hervorzuheben:

Unendliche Iteratoren

Haskell lässt c hier von 1 bis "unendlich" laufen. Zur Simulation müssen wir den größten Integer MAXRANGE ermitteln, den Perls Range-Operator noch akzeptiert. Auch ist es nicht selbstverständlich, dass Perl hier keine Memoryprobleme bekommt, in älteren Versionen hat `foreach` die Liste erst vorkalkuliert, statt darüber zu iterieren. Diese "Magie" klug auf den Range zu reagieren versagt aber bereits, wenn man versucht mit `for (reverse 1..MAXRANGE)` herunterzuzählen.

Lazyness

Wir müssen in Perl immer noch ein temporäres Array `@tripel` anlegen und bestücken, wo Haskell "faul" einen Stream weitergeben kann. Statt direkt die ersten 5 Ergebnisse abzugreifen, kann man diesen Stream auch weiterreichen und andere Auswahlkriterien anwenden lassen. Die ausgewählten Tripel würden dann erst bei Bedarf berechnet.

Syntactic Sugar

Gut wir konnten unser Ziel zwar in Perl umsetzen, aber wie bekommen wir es knapp und intuitiv realisiert, ohne dafür mindestens 7 Zeilen hinschreiben zu müssen?

Als ersten Schritt können wir uns *Prototypen* zu nutze machen. Dieser Mechanismus erlaubt uns einen großen Teil des Syntax von Builtin-Funktionen nachzubauen. Schreiben wir beispielsweise:

```
sub X (&;@) { ... }
```

erhalten wir eine Funktion X mit einem Interface

```
X BLOCK [LIST]
```

D.h (fast) analog zu `grep` oder `map` kommt eine Liste hinten rein und vorne raus, die wir durch einen Codeblock verändern. Mit dem Unterschied, dass die Liste wegen des "`;`" optional ist und die Elemente Array-Referenzen sind.

Mit unserer X-Funktion kann das *Kreuzprodukt* aus den Ergebnissen der Blöcken gebildet werden. Also

```
@kreuzprodukt= X{1..2} X{"a","b"};
print Dumper \@kreuzprodukt;
```

ergibt

```
$VAR1 = [[1,'a'],[2,'a'],[1,'b'],[2,'b']];
```

Perl6 kennt übrigens etwas sehr ähnliches, den *cross operator* `X` der eine Liste aller Permutationen aus den Elementen zweier Listen erzeugt. Zum Beispiel gilt dort:

```
my @a = (1, 2);
my @b = (3, 4);
my @c = @a X @b;
# @c is ((1,3),(1,4),(2,3),(2,4))
```

(Anmerkung: Perl6 kann Listen schachteln, in Perl5 brauchen wir Arrays)

Da wir aber in unserem Konstrukt beliebige Code-Blöcke und nicht nur Listen erlauben, können wir die Analogie zu `map` weitertreiben und zusätzlich `$_` belegen. Bei einer Reihung von `X{}`-Aufrufen soll die Defaultvariable - wie bei geschachtelten Schleifen - den letzten Faktor enthalten.

So können wir nun die einfachere Version unserer Aufgabe lösen und die Tripel bis `c<=10` kompakt ermitteln:

```
grep {$_->[0]**2 + $_->[1]**2 == $_->[2]**2}
    X{1..$_} X{1..$_} X{1..10};
# ergibt [[3,4,5],[6,8,10]]
```

Die Umsetzung ist nicht besonders schwer:

```
sub X (&;@) {
   my $c_block=shift;
   my @result;

   #- keine Liste?
   unless (@_) {
     @result=map {[$_]} $c_block->()
   }

   #- laufe durch Liste
   else {
     for my $a_old (@_) {
       $_=$a_old->[0];
       for my $new ($c_block->()) {
         push @result, [$new, @$a_old];
       }
     }
   }
   return @result;
}
```

Listing 2

Wurde die optionale Liste weggelassen, liefert der Codeblock Elemente, die in je ein eigenes Unterarray gepackt werden.

Andernfalls wird diese Liste von Arrays durchlaufen und die Elemente aus dem Block vorne angehängt. Der erste Wert des Arrays stammt also immer aus dem unmittelbar vorhergehenden Schritt und kann `$_` zugewiesen werden.

Psychologisch hat diese Lösung den großen Vorteil, sich in den gewohnten Mechanismen von `map` und `grep` intuitiv einzufügen:

1. Hinten kommt eine Eingabe-Liste rein, wird gemäß eines Codeblocks verarbeitet und kommt vorne als Liste raus.

2. `$_` dient als Laufparameter durch die Eingabeliste

3. `X{}` kann in beliebiger Länge angereiht und mit `map/grep` kombiniert werden.

Ergo überall wo Perl Listen erlaubt, kann `X{}` genutzt werden:

```
print "@$_" for X{0..9} X{0..9};
```

These: *Syntaxerweiterungen müssen etablierte Mechanismen und Interfaces möglichst nachahmen, um an "intuitiver" Akzeptanz zu gewinnen und "kulturell" kompatibel zu sein.*

4. Weiterhin haben wir den zusätzlichen Benefit einen Ersatz für den Cross-Operator aus Perl6 nachbauen zu können.

Unsere Syntaxerweiterung erfüllt also vielfältige Bedürfnisse, was die "Investition" in neue Syntax rechtfertigt.

These: *Man sollte möglichst mehrere Features in einem neuen Idiom zusammenfassen, statt eine Reihe zu spezialisierter Einzelidiome produzieren.*

Allerdings ist nun Vorsicht angesagt, dieses nicht zu übertreiben. Wer den neuen *Smartmatch*-Operator `~~` kennt, mag mir zustimmen, dass zu viele Sonderfälle den Zugang erschweren.

These: *Eine Erweiterung darf nicht so überladen sein, dass man seinen Zweck nicht mit wenigen Worten erfassen könnte.*

Wir können hingegen zusammenfassen:

`X` *ist eine* `map`*-artige Funktion*, die alle Permutationen von Listen aus Eingabeblöcken erzeugt.

Eine ähnlich knappe Zusammenfassung des `~~` Operators zu finden, fällt mir schwer! (Bin ich nicht smart genug für den Smartmatch? ;-)

Funktionaler Ansatz

Leider haben wir aber unsere Ziel mit Haskell mitzuhalten noch nicht erreicht. Die `X{}` Lösung ist nicht "Lazy", erlaubt keine unendlichen Listen und wir hantieren umständlich mit Konstrukten wie `$_->[0]` herum, statt unsere Variablen benennen zu können.

Konkretes Beispiel, folgender Ansatz um die ersten 5 Lösungen zu erhalten:

```
@temp =
  grep {$_->[0]**2+$_->[1]**2 == $_->[2]**2}
    X{1..$_} X{1..$_} X{1..1000} ;
print $temp[0..4];
```

würde schon aus Speichergründen scheitern und den Spass `X{1..MAXRANGE}` einzutragen, überlasse ich gerne den Hardwarespezialisten. (Wohlgemerkt 1000 ist in diesem Fall natürlich überdimensioniert. Aber erst einen "vernünftigen"

Wert durch manuelles Probieren ermitteln zu wollen, kann nicht das Ziel sein.)

Um *lazy* agieren zu können müssten wir in Perl *Iteratoren* statt fertiger Listen nutzen, d.h. Funktionen, die die gewünschten Listenelemente einzeln zurückgeben.

```
while ( ($elem) = $c_iteriere_liste->() ) {
     tu_was_mit($elem);
}
```

Ist die Liste durchlaufen, wird eine leere Liste zurückgegeben und die `while`-Schleife bricht ab. Und da wir hier nicht `iteriere_liste()` schreiben, sondern eine Code-Referenz auswerten, können wir diese - wie bei jedem anderen Scalar auch - weiterreichen.

Da aber `map` nicht mit Listeniteratoren arbeiten kann, müssen wir uns einen Ersatz bauen.

Mit folgenden Interfaces, die ineinandergreifen:

```
take MAPBLOCK [COUNT,] ITERATOR
            -> list: LIST / scalar: ITERATOR

from LISTBLOCK VAR [,IN-ITERATOR]
            -> scalar: IN-ITERATOR
```

könnten wir auch folgenden "sprechenden" Code notieren:

```
take { [$a,$b] } 5 => from {1..10} $a =>
                           from {1..$a} $b;
```

MAPBLOCK soll die Funktionsweise eines `map {}` Blockes haben, d.h. eine Abbildung der durchlaufenen Werte beschreiben.

Um das Umzusetzen nutzen wir folgende Prototypen

```
sub take (&$;$) {...};
sub from (&$;$) {...}
```

Und können uns mit zusätzliche Klammern den Aufrufmechanismus verdeutlichen:

```
take ( {...} [WERT] ,
     ( from { ... } $var1 ,
            ( from {...} $var2 ) ) )
```

Wir sehen, das hintere `from` wird zuerst ausgeführt und liefert einen IN-Iterator, der zum 3. Argument des vorderen `from` wird. Dieser verarbeitet diesen und liefert einen neuen IN-ITERATOR als letztes Argument an `take` zurück. Die *fetten Kommas* `=>` sind hierbei nur *Syntactic Sugar*, d.h. wir könnten auch "richtige" Kommas hinschreiben, gewinnen aber an Lesbarkeit, weil wir durch diese Konvention die Aufrufebenen besser visuell trennen können.

Die Rückgabe von `take()` ist dabei kontextabhängig, im Listenkontext soll analog zu `map` eine Liste zurückgegeben werden, doch im Skalarkontext reichen wir einen normalen Iterator zurück, um ebenfalls in der Lage zu sein Streams weiterzureichen. Eine mögliche Umsetzung dieses Konzeptes zeigt Listing 3

Wir sehen die `IN-ITERATOR`en beschreiben dabei "innere" Iteratoren, die nur intern im Zusammenspiel unserer Befehlskombinationen genutzt werden und sich von den oben beschriebenen "normalen" Iteratoren unterscheiden müssen:

1. Statt die durchlaufenen Elemente zu returnieren, werden die zugeordneten Lauf-`VAR`-riablen belegt.

2. Der Rückgabewert hingegen hat die Eigenschaft, beim letzen Listenelement - und nur dann - einen falschen Wert zurückgeben und anschließend wieder die Liste von vorne zu durchlaufen.

3. Jeder `IN-ITERATOR` ruft rekursiv auch die eingeschachtelten `IN-ITERATOR`en auf, die wiederum ihre Lauf-`VAR`-iablen belegen.

Bei der Umsetzung von `from` nutzen wir noch aus, dass `$_[1]` ein *Alias* und keine Kopie des übergebenen Parameters ist, d.h. mit `\$_[1]` erhalten wir tatsächlich die Referenz auf die Variable die beim Aufruf angegeben wurde - siehe Listing 4.

Erwähnenswert ist noch dass wir `$idx` immer bereits für den nächsten Durchlauf inkrementieren und returnieren. D.h. nur beim letzten Listenelement wird der falsche Wert 0 zurückgegeben.

U.a. sind wir nun in der Lage folgendes zu schreiben:

```
my @arr =
  take {($c**2+$b**2==$a**2)
         ? [$a,$b,$c] : ()}
   5 => from {1..20} $a => from {1..$a} $b =>
      from {1..$b} $c;
```

```
sub take (&$;$){
  my ($c_block,$count,$c_iter_in) = @_;

  #-  Nur 2 Args ? => der 2. war Iterator
  unless (defined $c_iter_in) {
    $c_iter_in=$count;
    $count = -1;
  }

  my @collect;
  my $result;
  my $last = 0;

  #- Listenkontext?
  if (wantarray) {
    while (1) {
      my $idx_in = $c_iter_in->();
      my @result = $c_block->();

      if ( $idx_in and       # nicht fertig
           @result != 0 )    # nicht leer
        {
          push @collect,@result;
          $count--;
        }

      return @collect unless    #- Fertig ?
        $count and
          $idx_in;
    }
  }

  #- Scalar oder Void-Kontext?
  else {
    my $finnish;

    #- mache Iterator
    my $c_iter = sub {
      while (1) {
        if ($finnish or $count==0) {
          return;          # Leere Liste
        }

        unless ($c_iter_in->()) # iterieren
          { $finnish = 1; }     # flag

        my @result=$c_block->();
        if ( @result !=0 ) {
          $count--;
          return @result;
        }
      }
    };
    return $c_iter;
  }
}
```

Listing 3

Im Grunde simulieren wir hier in `take` ein funktionales `grep`! Mit einer entsprechenden Umsetzung, z.B. `filter` genannt, wäre die Notation natürlich kompakter.

```
filter { $c**2 + $b**2 == $a**2 } ...
```

(Womit wir eine Fleißaufgabe für den interessierten Leser hätten... =)

```
sub from (&;$$) {
  my $c_block   = shift; # Block;
  my $s_var   = \$_[0];  # Laufvariable
  my $c_iter_in = $_[1]; # IN-Iterator

  undef $$s_var;         # reset

  #-- weitere closure Variablen
  my @blocklist;
  my $idx;
  my $idx_in;

  my $c_iter = sub {

    #-- Initialisierung
    unless ($idx) {      # erster Lauf?
      @blocklist=$c_block->(); # init Liste
      unless (@blocklist) {    # leer?
        return;                # Abbruch!
      }
      $idx = 0;                # init idx
    }

    #-- Zuweisung Listenelemente
    $$s_var=$blocklist[$idx];

    #-- Innere Schleife fortschalten
    if ($c_iter_in) {    # vorhanden?
      # fortschalten
      $idx_in = $c_iter_in->();
      if ($idx_in) {     #  nicht fertig?
        #   concat Index
        return $idx.$idx_in;
      }
    }

    #-- Inkrementiere Index modulo
    # Listengroesse
    $idx = ($idx+1) % @blocklist;

    return $idx;
  };

  return $c_iter;
}
```

Listing 4

Vorläufiges Fazit

Das hier beschriebene Methode Iteratoren durchzureichen, ist äußerst mächtig:

1. Obwohl unsere Bausteine `from` und `take` relativ aufwändig erscheinen, tragen sie tatsächlich nur wenig zur Zeitkomplexität bei, weil sie nur einmalig aufgerufen werden. Die Arbeit wird in den konstruierten Iteratoren verrichtet, die erst bei Bedarf - *lazy* - ausgeführt werden.

2. Wir sind flexibel in der Erweiterung unseres Instrumentariums, wir könnten nun ohne weiteres noch ein Funktion `filter GREPBLOCK ITERATOR` realisieren, welches die Stelle von `grep` einnäme. Spezialisten wie Haskell kennen

noch einige andere Konstrukte, die sich bei List Comprehensions anwenden lassen.

3. Wir könnten auch mehrere Laufvariablen zulassen. Mit `from {0..8} $a,$b,$c, ...` könnte uns Mehrfachzuweisungen für Schleifen erlauben, die in Perl5 sonst nur schwer umzusetzen sind. D.h. `$a` liefe dann durch `0,3,6`. (Perl6 führt deswegen einen speziellen Syntax mit `->` ein.)

4. Da wir auf *Magie*, *Codefilter* und andere gewagte Eingriffe in den Parser verzichtet haben, bleibt das Tor für eine spätere XS-Umsetzung oder gar Übernahme in den Sprachstandard weit offen. Eine Funktion nach C zu portieren ist weit einfacher, als z.B. dem Compiler einen neuen Operator beizubringen.

Summa Summarum:

These: *Lieber funktionale Ansätze mit Hilfe von Standardmitteln wie Prototypes umsetzen, als zu versuchen eine isolierte Lösung ohne Entwicklungspotential in den Sprachkern zu hacken.*

Bei all den positiven Seiten müssen noch Schwachstellen unseres Ansatzes Erwähnung finden:

1. Wir müssen unsere Laufvariablen unter `strict` vorweg deklarieren, sowas wie

```
... from {0..9} my $c => from {1..$c} my $d;
```

würde leider nicht wie gewünscht funktionieren, da eine Deklaration erst ab dem nächsten Statement wirkt. M.a.W die beiden `$c` würden unterschiedliche Variablen bezeichnen. Um das zu umgehen müssten wir eine Zeile mit

```
my ($c,$d);
```

voranstellen, was natürlich redundant wäre.

These: *Akzeptable Lösungen sollten möglichst DRY ("Don't repeat yourself") sein.*

2. Jede weitere Rekursionsebene im Iterator erhöht den Overhead beträchtlich, weil Funktionsaufrufe in Perl relativ teuer sind. Die linearisiere 3fach Schleife in Listing 1 kann in trivialen Fällen durchaus um den Faktor 100 schneller sein.

These: *Akzeptable Lösungen dürfen sich in der Ausführungsgeschwindigkeit nicht um Größenordnungen von klassischen Ansätzen unterscheiden.*

3. Wir haben immer noch keine Lösung gezeigt, um "unendliche Iterationen" zu beschreiben.

Wie wir diese Probleme in den Griff kriegen können, soll Thema des nächsten Teils dieser Artikelreihe sein. Ich freue mich schon auf euer Feedback, die Mailingliste der Darmstadt.pm kann gerne dafür genutzt werden (siehe http://darmstadt.perlmongers.de/)

Zu guter Letzt

Es sei angemerkt, dass es in diesem konkreten Fall natürlich weit schellere Algorithmen gibt, um die gesuchten Tripel zu berechnen. Schon die 3. Schleifenebene ist unnötig, wenn man die Wurzelfunktion nutzt. Darum ging es uns aber auch nicht! Hei wir brauchten nur ein Beispiel. :)

Javier Amor Garcia

Zentyal

Zentyal (früher bekannt als eBox Plattform) ist ein Linux Server für kleine und mittlere Unternehmen. Zentyal integriert über 30 Open Source Anwendungen in einer einfach bedienbaren Technologie, die es erlaubt, das Netzwerk über ein benutzerfreundliches Webinterface einzurichten.

Zentyal kann als Gateway dienen, um Internetzugriff zu gewähren, alle Basis-Netzwerkdienste zu administrieren, das Netzwerk vor Angriffen zu schützen und Zugriff auf die notwendigen Kommunikations- und Groupware-Dienste zu erlauben.

Als Gateway erlaubt es Zentyal, den Zugriff auf das Internet zu managen und den Traffic zwischen verschiedenen DSL-Verbindungen zu verteilen. Es ermöglicht auch die Steuerung des Datenverkehrs (Quality of Services - QoS) und bietet einen HTTP Cache Proxy.

Zentyal bietet als Sicherheitsserver alle Sicherheitsdienste, die für ein Unternehmensnetzwerk benötigt werden - von der Firewall über ein Intrusion Detection System, sicheren Zugriff auf Netzwerk-Ressourcen über VPN, Filter für Webinhalte bis hin zu Antivirus-Funktionen usw.

Die Netzwerkinfrastrukturdienste beinhalten DHCP, DNS, FTP und NTP Server, eine Zertifizierungsstelle (CA) zum managen von Zertifikaten von unterschiedlichen Services und einen HTTP-Server, um virtuelle Domains und Benutzerseiten bereitzustellen.

Als Office-Server bietet Zentyal neben einem grundsätzlichen Verzeichnismanagement (User und Gruppen) für die Authentifizierung von Clients eine große Anzahl von Diensten für das Resource-Sharing wie Dateien, Drucker, Kalender, Kontakte oder Aufgaben.

Die Kommunikationsdienste beinhalten E-Mail mit Antispam, Antivirus und Webmail, Instant Messaging und VoIP.

Das Modulare Design von Zentyal bietet eine große Flexibilität und Integration zwischen verschiedenen Netzwerkdiensten. Bei diesem Design hat jeder Service oder Gruppe von Services ein eigenes Modul, die einzeln installiert und aktiviert werden können. Das erlaubt es uns, Dienste nahtlos hinzuzufügen oder zu entfernen.

Dieser Artikel wird die Grundlagen legen, um ein Modul für Zentyal zu schreiben, so dass Du Deine beliebtesten Dienste zum Server hinzufügen kannst.

Basisarchitektur von Zentyal

Zentyal Module erzwingen die Konfiguration der Netzwerkdienste, die es verwaltet. Wenn Zentyal gestartet wird, wird jedes Modul in einer bestimmten Reihenfolge neugestartet und jedes Modul macht folgendes:

1. Herunterfahren des Dienstes
Um einen sauberen Stand zu gewährleisten, stoppt das Modul alle Daemons, die davon abhängen und räumt auf.

2. Durchsetzen der Konfiguration
Jedes Modul macht alles Nötige, um seine Konfiguration im System bekannt zu machen. Solche Aktionen kann das Erzeugen einer Konfigurationsdatei, Verändern des LDAP-Verzeichnisses oder das Ausführen eines Kommandos bedeuten.

3. Starten des Dienstes
Zum Schluss wenn alle Konfigurationen gemacht wurden, führt das Modul alle Arbeiten aus, damit der Dienst erreichbar ist, wie zum Beispiel das Starten eines Daemons.

Die Module sollen so unabhängig wie möglich sein, aber einige Verbindungen zwischen den Modulen sind notwendig, um ein vollständig integriertes System zu haben. Das kann passieren, weil einige Module Dienste bereitstellen, die von anderen Modulen benötigt werden (z.B. Konfiguration des Netzwerkinterfaces, LDAP oder Webserver) oder um einfach sicherzustellen, dass keine inkompatiblen Konfigurationen existieren (z.B. weil zwei Daemons auf dem gleichen Port lauschen).

Ein Modul besteht aus:

• Eine Haupt-Perlklasse
Das ist die Hauptschnittstelle zum Modul. Einige Basismethoden, die das Verhalten des Moduls steuern, können dort überschrieben werden. Wenn Zentyal Zugriff auf die API eines Moduls verlangt, bekommt es eine Instanz dieser Klasse.

• Perlklassen
Natürlich wäre es schlechte Praxis, den Code aller Module in einer Klasse zu haben (mit Ausnahme von ganz kleinen Modulen) benötigen wir mehrere Perl-Dateien um unser Design umzusetzen.

• Perlklassen für das Webinterface
Diese Klassen definieren das Webinterface. Eines der Ziele des Zentyal Entwicklungsframeworks ist es, den Entwickler von HTML und CGI-Code zu befreien. Wir werden über das Interface im nächsten Abschnitt detaillierter zu sprechen kommen.

• Hilfsskripte
In einigen Modulen macht es Sinn, ein paar Hilfsskripte anzubieten, die Aufrufe vom Modul oder von der Konsole ermöglichen. Zusätzlich benutzen wir einige Skripte, um eine nahtlose Migration zwischen verschiedenen Versionen des gleichen Moduls zu ermöglichen.

• Verschiedene Dateien
Andere Dateien werden für die Funktionen des Moduls benötigt, wie zum Beispiel Templates für Konfigurationsdateien oder LDIF-Dateien. Wir könnten genauso gut Dateien für das Interface benötigen, wie z.B. JavaScript oder Bilder.

Benutzerkonfiguration

Zentyal wäre nicht so nützlich, wenn der User nicht die Einstellungen des Moduls ändern könnte. Die Konfiguration jedes Moduls ist über das Zentyal Webinterface zugänglich.

Wenn der User die Bearbeitung der Konfiguration abgeschlossen hat, muss er diese Änderungen speichern. Sobald diese gespeichert sind, startet Zentyal alle geänderten Module sowie die Module, die von den geänderten Modulen abhängen, neu. Dadurch wird die neue Konfiguration aktiv.

Webinterface

Die einzelnen Seiten des Webinterfaces von Zentyal werden mit den Definitionen aus den Klassen des Datenmodells generiert. Diese Modell-Klassen können einfach eine Sammlung von Attributen sein, die als Webformular oder eine Liste von Attributen, die als Tabelle gerendert werden.

Vielmehr ist jedes Attribut als Instanz einer `Type`-Klasse definiert. Diese Klasse enthält die Informationen wie dessen Daten gespeichert, geholt, angezeigt und validiert werden müssen.

Die Modell-Instanzen können geholt werden und nach den Werten der Attribute gefragt werden. Ein typisches Beispiel wäre, die Modelle zu benutzen, um die Konfigurationseinstellungen zu speichern, sie in der Hauptklasse auszulesen und eine neue Konfigurationsdatei mit den Werten der Einstellungen zu erzeugen.

Weiterhin gibt es eine Art von Klassen, die `Composites` genannt wird. Diese Art gibt es, um verschiedene Modelle in derselben Seite zu aggregieren und das Layout der Seite zu kontrollieren.

Voraussetzungen für diesen Artikel

Perl-Kenntnisse werden vorausgesetzt (uhh!) und Du brauchst eine Zentyal 2.0-Installation. Kenntnisse über das Debian Paketmanagement und autotools sind von Vorteil, aber keine Voraussetzung: Du kannst einfach blind meinen Anweisungen zu den Paketgenerierungen folgen.

Du brauchst auf Deiner Entwicklungsmaschine auch eine `Privoxy`-Installation oder Zugang zu den Inhalten des `/etc/privoxy`-Verzeichnisses (z.B. von einem installierten Privoxy oder durch Extrahieren der Daten aus dem Privoxy-Paket).

Du musst das `emoddev`-Paket installiert haben; dieses Tool ist für das initiale Modul-Scaffolding zuständig. Das Ubunut-Paket davon ist im Zentyal 2.0 Repository verfügbar. Um es zu Deinen apt-Quellen hinzuzufügen, füge diese Zeile in der Datei `/etc/apt/sources.list` hinzu: `deb http://ppa.launchpad.net/zentyal/2.0/ubuntu lucid main`.

Und schließlich steht der Code für diesen Artikel unter http://people.zentyal.org/~jag/foo-zentyal-module.tar.gz zum Download bereit.

Planungen für ein neues Modul

Wenn ein neues Modul geschrieben werden soll, ist es die erste Aufgabe zu entscheiden, welchen Dienst dieses Modul bereitstellen soll. Danach kannst Du die Software aussuchen und entscheiden, welche Konfigurationsoptionen der User einstellen können soll.

Zentyal hat das Ziel, einfach zu bedienen und flexibel zu sein. Aber wenn Du wählen musst, wähle die Einfachheit aus der Sicht des Benutzers. In der Praxis bedeutet das, dass es nicht klug ist, alle Konfigurationsoptionen über das Webinterface einstellbar zu machen. Idealerweise wählt man ein Subset, die in 80% der Fälle verwendet werden.

Danach solltest Du Dir überlegen, welche Aktionen durchgeführt werden sollen, um die verfügbaren Konfigurationen zu erstellen. Du solltest sie in einem System testen, bevor Du mit dem Modul beginnst. Wir müssen auch mögliche Konflikte mit bereits existierenden Zentyal-Modulen beachten: Diese Konflikte sollten entweder automatisch aufgelöst werden oder die konfliktreichen Konfigurationen sollten durch das Modul nicht erlaubt werden.

Beispielmodul: HTTP-Proxy, der Werbung blockiert

Für unser Beispielmodul wählen wir einen Proxy für HTTP-Traffic, der Werbung blockiert.

Die Software, die wir für diesen Dienst verwenden ist `Privoxy`. Also ist unsere erste Aufgabe, die Software zu installieren und dessen Konfiguration zu testen.

Nachdem wir etwas damit gespielt haben, entscheiden wir, dass wir nur zwei Einstellungen verändern können:

- Auf welchem Port gelauscht wird
- Die Adresse des Systemadministrators (optional)

Zusätzlich müssen wir die Änderungen an der Firewall implementieren, die für diesen Dienst notwendig sind, um den vom Server ausgehenden HTTP-Traffic zu erlauben (sonst kann der Proxy seinen Job nicht erledigen) und Proxy-Verbindungen von Hosts außerhalb des internen Netzwerks zu verbieten. Wir werden später sehen, wie das umzusetzen ist.

In ebox-moddev einsteigen

Jedes Modul für Zentyal benötigt etwas Scaffolding, und dafür haben wir ein Tool, das die Hauptklassen und die Paketinfrastruktur für uns erzeugt. Dieses Tool heißt `ebox-moddev`.

Wir rufen es auf, damit wir einen guten Startpunkt für unser Modul haben:

```
ebox-moddev-create \
--menu-separator 'Test modules' \
--main-class Privoxy --module-name privoxy \
--firewall-helper --version 0.1
```

Lass uns einen Blick auf die Parameter werfen:

--module-name
Der Name wird von Zentyal zur Identifikation des Moduls verwendet. Üblicherweise wird ein Name mit Kleinbuchstaben verwendet.

--main-class
Bestimmt den Namen der Hauptklasse des Moduls. Die Hauptklasse ist die Klasse, die die Methoden bereitstellt, die vom Framework für den ordnungsgemäßen Betrieb des Dienstes benötigt werden. Es ist auch ein Interface für andere Module, die Methoden aus unserem Modul aufrufen müssen. Es ist der Name für ein Perl-Paket und wir benutzen hier den CamelCase-Stil. Dem Namen wird automatisch in den `EBox::`-Namensraum geschoben.

--menu-separator
Das ist der Bereich im linken Menü, in dem das neue Modul auftauchen wird.

--firewall-helper
Dieser Parameter zeigt, dass das Modul etwas mit der Firewall machen will. Es ist eine Klasse, die Firewall-Regeln zu der Zentyal-Firewall hinzufügt.

--version
Schließlich noch die Versionsangabe, die die Version unseres Moduls bestimmt. Da es die erste Version sein wird, haben wir `0.1` gewählt.

Wir werden alle Moduldateien im Verzeichnis `privoxy` haben. Die Hauptklasse wird in `privoxy/src/EBox/Privoxy.pm` zu finden sein. Öffne diese Datei mit dem Editor Deiner Wahl.

Eine Bemerkung über Namen und Namensräume

Wie ich schon kurz angedeutet habe, wurde Zentyal früher 'eBox Platform' genannt. Da es hauptsächlich auf normalen Servern läuft und nicht in speziellen Hardwareboxen, haben wir uns entschieden, den Namen zu ändern. Da die meisten Teile der API unter dem alten Namen geschrieben wurden, werden wir sehr viele Namen sehen, die noch auf den alten Namen verweisen (eBox).

Modulkontrolle

Unsere erste Aufgabe wird es sein, die Sachen zur Kontrolle des Daemons einzufügen, so dass er über das Modul gestoppt und gestartet werden kann.

Die Kontrolle an sich wird automatisch durch das Framework ausgeübt; die Hauptklasse des Moduls muss nur die Methode `_daemons` implementieren, die festlegt, welche Daemons von dem Modul verwendet werden. Es werden zwei Arten von Daemons unterstützt: Daemons, die ein `upstart` Skript verwenden und solche, die ein klassisches `init.d` Skript verwenden.

`Privoxy` gehört zu letzterer Gruppe, also fügen wir diese Methode zu der Hauptklasse hinzu:

```
sub _daemons
{
    return  [
        {
            name => 'privoxy',
            type => 'init.d',
            pidfiles => [
                '/var/run/privoxy.pid'
            ]
        }
    ];
}
```

Der Name des Daemons ist der Name des `init.d` Skripts. Wir sollten auch die PID Datei(en) angeben, die von dem Daemon verwendet werden. So weiß Zentyal, ob der Daemon läuft oder nicht. Wenn das Zentyal-Framework definiert ist, werden wir diese Angaben nutzen, um das Modul zu starten und zu stoppen.

Webinterface

Das Webinterface wird dazu benutzt, auf die Privoxy-Einstellungen zuzugreifen, die wir sichtbar machen wollen. In unserem Beispiel darf der User einstellen, auf welchem Port Privoxy lauscht und eine optionale Mailadresse des Administrators. Da es nur diese beiden Einstellungen mit einer einzigen Modell-Klasse, wird keine Composite-Klasse benötigt, die die Webseiten besser arrangiert.

Um unser Modul zu erstellen, werden wir den Modulrumpf editieren, der durch `ebox-moddev` erstellt wurde. Öffne die Datei `privoxy/src/EBox/Privoxy/Model/Settings.pm`. Wir werden als erstes `use`-Statements für die Typen `EBox::Types::Port` und `EBox::Types::MailAddress` einfügen, weil wir diese zum Speichern der Einstellungen benötigen.

```
use EBox::Types::Port;
use EBox::Types::MailAddress;
```

Dann werden wir uns die `_table`-Methode anschauen. Diese Methode liefert einen Hash zurück, der die Struktur und die Basiseigenschaften des Formulars festlegt. In Wirklichkeit kommt der Name `_table` aus der `DataForm`-Klasse, die eine Subklasse von `DataTable` ist.

In dem Hash interessiert uns die Eigenschaft `tableDescription`, die die Felder des Formulars definiert. In dem Beispiel wird es der Liste `@fields` zugewiesen. Also definieren wir die Felder in dieser Liste.

```
my @fields = (
    new EBox::Types::Port(
        'fieldName'     => 'port',
        'printableName' =>
            __('Listening port'),
        'defaultValue'  => '8118',
        'editable'      => 1
    ),
    new EBox::Types::MailAddress(
        'fieldName'     => 'adminAddress',
        'printableName' =>
            __('Administrator email'),
        'editable'      => 1,
        'optional'      => 1,
    ),
);
```

Mit diesem kleinen Stück Code haben wir bereits das Speichern und die Validierung der Einstellungen. Jetzt müssen wir noch diese Einstellungen mit der Konfiguration des Dienstes verbinden.

Erstellen der Konfiguration

Wir können die Konfiguration durch das Bearbeiten der Konfigurationsdatei von Privoxy erzeugen. Zentyal verwendet üblicherweise Texttemplates um die Datei zu generieren. Die Bibliothek, die dabei verwendet wird, ist Mason.

Da die Standardkonfiguration von Privoxy sehr vernünftig ist, werden wir diese als Basis für unser Template benutzen. Also gehen wir zum Verzeichnis `privoxy/stubs` und kopieren die Konfigurationsdatei von Privoxy über das Template, das von `ebox-moddev` erstellt wird.

```
cp /etc/privoxy/config service.conf.mas
```

Am Anfang der Datei werden wir einen Abschnitt einfügen, der angibt welche Argumente von dem Template benötigt werden:

```
<%args>
$port
$adminAddress
</%args>
```

Dann werden wir Code einfügen, der die Einstellungen durch die übergebenen Werte ersetzt. Als erstes werden wir den Port einstellen. Suche nach dem Abschnitt 4.1 der Konfigurationsdatei und ersetze `listen-address 127.0.0.1:8118` durch `listen-address:<% $port %>`.

Da die Administratoradresse optional ist, werden wir eine Bedingung benutzen. Suche nach dem Abschnitt 1.3 und füge folgendes ein:

```
% if ($adminAddress) {
admin-address <% $adminAddress %>
% }
```

Achte besonders darauf, dass keine Leerzeichen vor dem %-Symbol ist. Sonst wird es nicht korrekt interpretiert.

Wir haben jetzt die Benutzereinstellungen und das Template für die Konfigurationsdatei. Also sollten wir jetzt zu der Hauptklasse `EBox::Privoxy` zurückkehren und das Modul anweisen, die Konfiguration bei jedem Neustart neu zu schreiben.

Immer wenn wir eine Datei eines anderen Ubuntu-Pakets ändern, sollten wir den User warnen. Sonst brechen wir die Distributions-Regeln. In Zentyal wird dies durch Überschreiben der `usedFiles`-Methode in der Hauptklasse gemacht:

```
sub usedFiles
{
    my ($self) = @_;
    return [
        {
         file => '/erc/privoxy/config',
         module => 'privoxy',
         reason => __(
               'Privoxy configuration file'),
        },
     ];
}
```

Wenn der Benutzer das erste Mal das Modul aktiviert, wird ein Dialog ihn wegen der überschriebenen Datei und anderen Änderungen, die durch Zentyal an Paketen von dritten gemacht wurden, warnen.

Danach sollten wir die `_setConf`-Methode überschreiben. Diese Methode wird in jedem Modul aufgerufen wenn die Konfiguration gesetzt werden soll.

```
sub _setConf
{
    my ($self) = @_;
    my $model = $self->model('Settings');
    my $port = $model->portValue();
    my $adminAddress =
        $model->adminAddressValue();

    $self->writeConfFile(
        '/etc/privoxy/config',
        'privoxy/service.conf.mas',
        [
           port => $port,
           adminAddress => $adminAddress,
        ]
    );
}
```

Wir können im Code sehen, dass die `model`-Methode benutzt wird, um das Modellobjekt über seinen Namen zu holen. Und dann können wir die Werte der Felder mit der `nameValue`-Methode bekommen. Wir haben diese Methoden nicht geschrieben, da sie automagisch durch Perls `AUTOLOAD`-Feature bereitgestellt werden. Als nächstes rufen wir die `writeConfFile`-Methode auf und benutzen die Einstellungen als Parameter für das Template.

Jetzt haben wir ein Modul für Zentyal, das uns die Kontrolle des Privoxy-Daemons erlaubt und sowohl den Port als auch die Administratoradresse setzen lässt.

An diesem Punkt können wir unsere Arbeit durch das Generieren eines Ubuntu-Pakets prüfen. Aber bevor wir das tun, sollten wir die Kontrolldatei für das Paketieren anpassen, damit es Privoxy als Abhängigkeit hat. Um das zu tun, öffnen wir `privoxy/debian/control` und fügen `privoxy` zu dem `Depends`:-Feld hinzu (mehrere Werte sollten durch Komma getrennt werden).

Rufe dieses Kommando im `privoxy`-Verzeichnis auf, um das Paket zu generieren:

```
dpkg-buildpackage -us -uc
```

Jetzt können wir das brandneue Paket in unserem Zentyal Server installieren. Dann kannst Du versuchen, das Modul zu aktivieren. Gehe zu der Konfigurationsseite, ändere ein paar Einstellungen und klicke den `Save changes`-Button oben rechts.

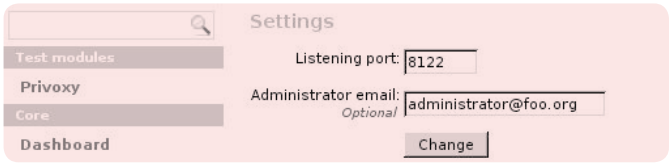


Abbildung 1: Die Seite für unseren Proxy

Wie Du in Abbildung 1 sehen kannst, wurde die Konfiguration mit den von Dir gewählten Parametern geändert und der Daemon wurde neu gestartet (das kannst Du über dessen PID prüfen). Dann kannst Du es als Proxy für einen Deiner Computer im Netzwerk einstellen. Funktioniert es?

Die Antwort auf diese Frage hängt davon ab, ob die Zentyal-Firewall aktiviert ist und welche Regeln diese hat. Offensichtlich müssen wir unser Modul in der Firewall integrieren, um sicherzustellen, dass das Verhalten auf einem richtig konfigurierten System wie erwartet ist. Mache mit dem nächsten Abschnitt weiter.

Firewallintegration

Man könnte meinen, dass hier unsere Arbeit beendet ist, aber damit das Modul funktioniert, muss es voll in das System integriert werden. In Zentyal gibt es ein Modul, das die Firewall kontrolliert und wir müssen dafür sorgen, dass unser Privoxy-Modul mit der Firewall zusammenspielt oder wir nicht-legitime Verbindungen erlauben und legitime Verbindungen verbieten.

Es gibt zwei Hauptprobleme wenn Module interagieren: Probleme mit Abhängigkeiten und Änderungen im Verhalten eines Moduls.

Abhängigkeiten

Die Probleme mit den Abhängigkeiten können in zwei weitere Probleme aufgeteilt werden: Aktivierte Abhängigkeiten und die initiale Reihenfolge der Abhängigkeiten.

Eine aktivierte Abhängigkeit ist notwendig wenn ein Modul ein anderes Module benötigt, das schon aktiviert ist um ordentlich zu funktionieren. Das kann durch die Implementie-

rung der Methode `enableModDependencies` erreicht werden. Diese Methode sollte eine Liste der benötigten Module zurückliefern.

In unserem Privoxy-Modul brauchen wir das Netzwerk-Modul um sicherzustellen, dass die Netzwerk-Konfiguration durch den Benutzer definiert wurde und das Firewall-Modul um sicherzustellen, dass Verbindungen zum externen Netzwerkinterface nicht erlaubt sind. Wir werden trotzdem nur das Firewall-Modul in die Liste eintragen, weil dieses Modul selbst eine aktive Abhängigkeit auf das Netzwerk-Modul hat.

```
sub enableModDepends
{
    my ($self) = @_;
    return ['firewall']
}
```

Eine Initialisierungs-Abhängigkeit ist notwendig, wenn ein Modul ein anderes Modul braucht, das schon gestartet wurde, bevor es selbst gestartet werden kann.

Unser Privoxy-Modul sollte nach dem Firewall-Modul gestartet werden. Sonst wird das Firewall-Modul seine Basisregeln später setzen und wir werden nicht das erwartete Ergebnis bekommen. Diese Art von Abhängigkeiten kann entweder durch Bearbeiten der Datei mit den Moduleinstellungen - die gleiche, die angibt welches die Hauptklasse ist - oder durch Überschreiben der Methode `depends`. Ich bevorzuge letzteres, weil Du die Ausgabe ändern kannst, so dass die Abhängigkeiten für Deine Systemkonfiguration widergespiegelt werden.

```
sub depends
{
    my ($self) = @_;
    return ['firewall']
}
```

In diesem Fall sind die Aktiv-Abhängigkeiten und die Initialisierungsabhängigkeiten die selben. Aber das ist nicht immer so, so dass wir diese Arten unterscheiden müssen.

Hinzufügen von Firewallregeln

Die Interaktion zwischen Modulen ist mehr als nur die Abhängigkeit. Tatsächlich ist es so, dass einige Module ihre Konfiguration in Abhängigkeit der Anwesenheit und die Konfiguration anderer Module ändern sollten.

So ist die Satz von iptables-Regeln, die durch das Firewall-Modul generiert werden, von der Konfiguration der Module abhängig, die die Firewall benutzen.

Der einfachste Weg ist, zu prüfen, ob ein bestimmtes Modul da ist und dann die notwendigen Aktionen auszuführen, die auf dessen Anwesenheit und Konfiguration beruhen. Der Nachteil dieser Methode ist, dass die Module sehr stark gekoppelt sind und es ist schwierig die Verbindungen zu mehr als einem anderen Modul zu managen.

Wenn in komplexeren Fällen mehrere Module involviert sind, ist es besser eine Art von Virtuellem Interface zwischen den Modulen zu haben. Dann kann das Modul, das eine bestimmte Methode aufrufen muss, einfach diese Module mit einer bestimmten Subklasse aufrufen oder eine bestimmte Methode implementieren.

Die Module, die Firewallregeln hinzuzufügen wollen, müssen Kinder der `EBox::FirewallObserver`-Klasse sein. Sie sollten auch die Methoden `usesPort` und `firewallHelper` überschreiben. Letztere Methode liefert ein Objekt mit den Informationen, welche Regeln das Firewall-Modul für das Privoxy-Modul hinzufügen soll.

Der Scaffolding-Generator hat eine leere Methode `usesPort` für uns erzeugt. Wir ersetzen den Rumpf hiermit:

```
sub usesPort
{
    my ($self, $protocol, $port, $iface) =
                                        @_;

    if ($protocol ne 'tcp') {
        # privoxy uses only TCP connections
        return undef;
    }

    if ($port ne $self->port()) {
        return undef;
    }

    # it listens on all interfaces, so we
    # dont check $iface (we will use the
    # firewall to restrict access to the
    # internal interfaces)
    return 1;
}
```

Genauso wurde eine Basis für die Methode `firewallHelper` erzeugt. Wir sollten den Konstruktoraufruf so anpassen, dass der Port als Argument übergeben wird.

```
sub firewallHelper
{
    my ($self) = @_;

    my $model = $self->model('Settings');
    my $port = $model->portValue();
    my $fw =
        new EBox::Privoxy::FirewallHelper(
          port => $port
        );
    return $fw;
}
```

Danach können wir zur Klasse `FirewallHelper` gehen. Wir finden diese unter `privoxy/src/EBox/Privoxy/FirewallHelper.pm`. Wir müssen den Konstruktor so anpassen, dass der Port-Parameter berücksichtigt wird.

```
sub new
{
    my ($class, @params) = @_;

    my $self = { @params };
    $self->{net} = EBox::Global->modInstance(
        'network'
    );
    bless($self, $class);
    return $self;
}
```

Dann sollten wir noch unsere Regeln hinzufügen. Die Klasse `FirewallHelper` besitzt eine Methode um die Regeln zu liefern, die die Firewall in jede Regelkette hinzufügen muss.

Wir ändern die Methode für die `output` Kette so, dass sichergestellt ist, dass der Proxy die Requests an HTTP-Server weiterleiten kann.

```
sub output {
    return [ "--dport 80 -j ACCEPT" ];
}
```

Auf ähnliche Art und Weise müssen wir die Methode `input` anpassen. Diese Methode liefert Regeln, die wir für interne Netzwerkinterfaces durchsetzen.

```
sub input
{
   my ($self) = @_;

   my $port = $self->{port};
   return
   ["--protocol tcp --dport $port -j ACCEPT"]
}
```

Wir fügen keine Regeln für den Input von externen Interfaces hinzu, weil wir keine Verbindungen von externen Netzwerken erlauben wollen.

Jetzt haben wir die Firewallintegration abgeschlossen. Wir können erneut ein Paket mit `dpkg-buildpackage -us -uc` erzeugen und dieses installieren. Dann sollten wir noch prüfen, ob wir uns immer von Hosts im internen Netzwerk verbinden können aber nicht von Hosts von außerhalb.

Weitere Verbesserungen

Es gibt ein paar mögliche Verbesserungen für dieses Modul. Eine wäre, mehr Privoxy Einstellungen verfügbar zu machen; Wie dem auch sei, da Zentyal auf die einfache Bedienbarkeit fokussiert ist, muss man sorgfältig auswählen, welche und wie viele Einstellungen verfügbar gemacht werden.

Renée Bäcker

Regelmäßige Backups

Für einen Kunden habe ich eine webbasierte Anwendung mit Datenbankanbindung und Uploadmöglichkeiten geschrieben. Und diese Daten sollen regelmäßig gesichert werden - auch dafür existiert ein kleines Skript, das die Daten zusammen mit Meta-Informationen in ein Archiv schreibt. Damit das nicht immer wieder vom Administrator per Hand gemacht werden muss, sollte das ganze automatisch und regelmäßig passieren.

Derjenige, der die Anwendung betreut, kennt sich mit dem Task-System auf Windows nicht aus. Unter Unix findet sich `cron` und Unix-Anwender kennen sich eher mit `cron` aus, als Windows-Anwender mit dem Task-System. Damit ich den Task nicht einrichten muss und nicht bei einem Server-Umzug das ganze wieder eingerichtet werden muss, habe ich ein entsprechendes Perl-Skript geschrieben.

Wie so häufig, gibt es auf CPAN ein Modul, das für diese Aufgabe bestens geeignet ist: `Win32::TaskScheduler`. Allerdings ist die Installation aus den Sourcen auch mit StrawberryPerl zumindest in der Version 2.0.3 des Moduls nicht ohne weiteres möglich. Aber da auch mit StrawberryPerl ein PPM-Client (PPM = Perl Package Manager) mitgeliefert wird und das Modul in einem Repository existiert, ist die Installtion dennoch möglich.

```
my %commands = (
    run       => \&do_backup,
    install   => \&install_task,
    uninstall => \&uninstall_task,
);

$command ||= 'help';
my $sub = $commands{$command} ||
                          $commands{help};
$sub->();
```

Listing 1

Mit diesem Modul kann man Tasks mit beliebiger Konfiguration einrichten. Hier soll der oben beschriebene Task gezeigt werden. Anstelle des Backups wird hier einfach eine Nachricht in einem kleinen Fenster angezeigt.

Das Skript soll für alle Aufgaben bezüglich des Tasks zuständig sein, also Einrichten des Tasks, Deinstallieren des Tasks und Erstellen des Backups. Damit die jeweils richtige Funktion aufgerufen wird, benutze ich einen Hash als Dispatchtabelle (Listing 1).

In diesem Artikel soll auf das Erstellen des Backups nicht näher eingegangen werden. Dort wird einfach mit `Archive::Tar` ein Archiv gepackt und dieses an eine bestimmte Stelle verschoben. Hier soll es nur um die Einrichtung und das Deinstallieren der Aufgabe gehen.

Als erstes muss die Aufgabe eingerichtet werden (Listing 2). Ab der Version 2.0.0 von `Win32::TaskScheduler` muss vor dem Zugriff auf den Aufgabenplaner von Windows ein neues Objekt erzeugt werden. Das war vorher nicht der Fall. Als erstes muss eine neue Aufgabe hinzugefügt werden (`New-WorkItem`). Der Methode muss ein eindeutiger Namen übergeben werden. Ist dieser Name schon in der Verwendung, schlägt das Hinzufügen fehl. Weiterhin muss die Trigger-Konfiguration übergeben werden. Die Konfiguration kann etwas tricky sein. Bei meinen ersten Tests mit dem Modul habe ich ständig Fehler "Null pointer exception" bekommen - ohne genaue Fehlermeldung. Nach ein wenig Ausprobieren bin ich darauf gestoßen, dass es an einem Fehler in der Konfiguration lag (fehlende Angabe von `MinutesDuration`).

In der Konfiguration wird festgelegt, ab wann die Aufgabe ausgeführt werden soll und wann die Aufgabe enden soll. Mit `MinutesInterval` wird bestimmt, in welchen Interval-

```
sub install_task {
    my $scheduler = Win32::TaskScheduler->New;

    my %task_config = (
        BeginYear   => 2010,
        BeginMonth  => 8,
        BeginDay    => 1,
        EndYear     => 2201,
        EndMonth    => 8,
        EndDay      => 1,
        StartHour   => 12,
        StartMinute => 1,
        MinutesDuration => 1440,
        MinutesInterval => 5,
        TriggerType => 1,
        Type => {
            DaysInterval => 1,
        },
    );

    my ($has_set_name, $saved) = ("","");
    my $has_new_item = $scheduler->NewWorkItem( $task_name, \%task_config );
    if ( $has_new_item == 1 ) {
        $has_set_name = $scheduler->SetApplicationName( $task_app );
        $scheduler->SetParameters( $task_param );

        $scheduler->SetAccountInformation( 'Administrator', 'passwort' );

        if ( $has_set_name ) {
            $saved = $scheduler->Save;
            $scheduler->Activate( $task_name ) if $saved;
        }
    }

    print "Aufgabe eingerichtet" if $saved;
}
```

Listing 2

len die Aufgabe ausgeführt werden soll. Vom `TriggerType` hängt dann ab, was in dem Subhash `Type` angegeben werden muss.

Können in der Konfiguration im `Type`-Hash mehrere Werte möglich sein (z.B. bei `Months`), werden diese mit einem "|" verbunden. Beispiele dafür kommen in den nächsten Beispielen.

Um es lesbarer zu halten kann man die Konstanten aus dem Modul benutzen. Es gibt diese Trigger-Typen:

• TASK_TIME_TRIGGER_ONCE
Eine Aufgabe muss nur ein einziges Mal ausgeführt werden. Hier werden keine weiteren Angaben im `Type`-Hash benötigt.

• TASK_TIME_TRIGGER_DAILY
Im `Type`-Hash muss `DaysInterval` angegeben werden, das bestimmt, wie viele Tage zwischen den Ausführungen liegen. Die Konfiguration

```
MinutesInterval => 5,
TriggerType     =>
   $scheduler->TASK_TIME_TRIGGER_DAILY,
Type             => {
    DaysInterval => 10,
},
```

bedeutet, dass die Ausgabe an jedem 10. Tag alle 5 Minuten ausgeführt wird.

• TASK_TIME_TRIGGER_WEEKLY
Für Tasks, die im Wochenrythmus ausgeführt werden sollen, gibt es den Trigger `TASK_TIME_TRIGGER_WEEKLY`. Benutzt man diesen Trigger, müssen im `Type`-Hash zwei Angaben gemacht werden - `WeeksInterval` und `DaysOfTheWeek`.

Möchte man einen Task also alle zwei Wochen dienstags und donnerstags ausgeführt haben, muss man das so konfigurieren:

```
TriggerType =>
        $scheduler->TASK_TIME_TRIGGER_WEEKLY,
Type => {
     WeeksInterval => 2,
     DaysOfTheWeek =>
           $scheduler->TASK_TUESDAY |
                 $scheduler->TASK_THURSDAY,
},
```

• TASK_TIME_TRIGGER_MONTHLYDATE

Dieser Trigger ist dafür da, in bestimmten Monaten zu einem bestimmten Datum eine Aufgabe auszuführen. Soll das Programm am 15. Juni und am 15. Dezember ausgeführt werden, muss die Konfiguration wie folgt aussehen:

```
TriggerType =>
   $scheduler->TASK_TIME_TRIGGER_MONTHLYDATE,
Type        => {
    Months => $scheduler->TASK_JUNE |
                  $scheduler->TASK_DECEMBER,
    Days   => 15,
},
```

Hier existiert auch ein Unterschied zur offiziellen Microsoft-API. Während dort mehrere Tage angegeben werden können, ist bei dem Modul immer nur 1 Tag möglich. Muss das Programm an mehreren Tagen in den Monaten ausgeführt werden, muss man mehrere Tasks anlegen.

• TASK_TIME_TRIGGER_MONTHLYDOW

Mit diesem Trigger-Typ ist es möglich, Aufgabe zu Zeiten wie "in den Monaten Februar, Mai, August und November am Montag der jeweils ersten Woche" auszuführen.

Für dieses Beispiel sieht die Konfiguration so aus:

```
TriggerType =>
   $scheduler->TASK_TIME_TRIGGER_MONTHLYDATE,
Type        => {
    Months =>
      $scheduler->TASK_FEBRUARY |
      $scheduler->TASK_MAY |
      $scheduler->TASK_AUGUST |
      $scheduler->TASK_NOVEMBER,
    DaysOfTheWeek => $scheduler->TASK_MONDAY,
    WhichWeek     => 1,
},
```

Wenn man nicht weiß, wie der Trigger aussehen soll, hilft es, auf einem Testrechner manuell so einen Task in der "Aufgabenplanung" (findet man bei "Zubehör" -> "Systemprogramme") anzulegen und dann den Trigger auszulesen:

```
use Data::Dumper;
use Win32::TaskScheduler;

my $task_name =
             'NameDesManuellAngelegtenTasks';
my $scheduler = Win32::TaskScheduler->New;
$scheduler->Activate( $task_name );

my %trigger;
my $result = $scheduler->GetTrigger(
                                0, \%trigger );
warn Dumper \%trigger;
```

Mit dem Modul kann man auch die anderen Tasks beobachten und die Ergebnisse (Status Code) der letzten Läufe bekommen:

```
$scheduler->GetStatus($status);
if ( $status == 267008 ) { #Ready
    print "\tStatus:ready\n";
}
elsif ( $status == 267009 ) { #Running
    print "\tStatus:RUNNING\n";
}
elsif ( $status == 267010 ) { #Not Scheduled
    print "\tStatus:Not Scheduled\n";
}
else {
    print "\tStatus:UNKNOWN\n";
}
$scheduler->GetExitCode($exitcode);
print "\tExitCode:".$exitcode."\n";
```

Gerade wenn man Software für Windows-Nutzer schreibt, ist dieses Modul sehr nützlich. Da die Dokumentation nicht die allerbeste ist, muss man ein paar Sachen ausprobieren.

Thomas Fahle

HowTo

App::perlbrew - Mehrere Perl-Installationen im Heimatverzeichnis

App::perlbrew - `Manage perl installations in your $HOME` - von Kang-min Liu ermöglicht die einfache Installation (ohne root-Rechte) und Verwendung mehrerer Perls in unterschiedlichen Versionen und Konfigurationen in einem eigenem Verzeichnis.

Das noch sehr junge Projekt bietet eine einfache Alternative zu dem leistungsfähigerem `buildperl.pl` aus dem Paket `Devel::PPPort`.

C-Compiler und Bibliotheken installieren

Um Perl kompilieren zu können, werden neben einem C-Compiler weitere Werkzeuge und Bibliotheken benötigt. Unter Ubuntu 10.04 installiert man dazu einfach folgende Pakete:

```
$ sudo apt-get install build-essential
$ sudo apt-get install libdb-dev libdb4.7
$ sudo apt-get install libgdbm-dev libgdbm3
```

Installation und grundlegende Konfiguration

App::perlbrew läßt sich entweder über die CPAN-Shell

```
cpan > install App::perlbrew
```

oder bevorzugt wie folgt

```
$ curl -LO http://xrl.us/perlbrew
$ chmod +x perlbrew
```

installieren. Anschließend steht das Kommandozeilen-Tool `perlbrew` zur Verfügung.

Die grundlegende Konfiguration erfolgt über die Option `init` - alle erforderlichen Dateien und Verzeichnisse werden per Vorgabe im Heimatverzeichnis im Ordner `~/perl5/perlbrew` angelegt.

```
$ perlbrew init

Perlbrew environment initiated,
required directories are created under
    /home/bob/perl5/perlbrew
Well-done! Congratulations!

Please add the following line to the end
of your ~/.bashrc

source /home/bob/perl5/perlbrew/etc/bashrc
...
```

Wer perlbrew in ein anderes Verzeichnis installieren möchte, setzt dazu die Umgebungsvariable `PERLBREW_ROOT`:

```
$ export PERLBREW_ROOT=/opt/perlbrew
$ perlbrew init
```

Nach dem die o.g. Änderungen in der Datei `~/.bashrc` ausgeführt wurden, aus der Shell abmelden und erneut einloggen, damit die Änderungen wirksam werden.

Mehrere Perls installieren

Mittels `perlbrew install` kann ein neues Perl installiert werden. Optionen, welche die Konfiguration des neuen Perls beeinflussen, werden über den Schalter `-D=` eingestellt. Die möglichen Optionen lassen sich der Datei `INSTALL` des jeweiligen Perls entnehmen.

Perl 5.12.1 mit Threads

```
$ perlbrew install perl-5.12.1 \
   -D=usethreads
...
Installed ... successfully.\
Run the following command to switch to it.
 perlbrew switch perl-5.12.1
```

Die überaus geschwätzige Ausgabe von `perlbrew` habe ich hier und in den weiteren Beispielen deutlich gekürzt.

Perl 5.12.1 mit Standard-Optionen

Installation eines Perl 5.12.1 mit Standard-Optionen, also ohne Threads - über den Schalter `-as` wird ein passender Name für dieses Perl gewählt.

```
$ perlbrew install perl-5.12.1 \
  -as perl-5.12.1-nothreads
...
Installed ... successfully. \
Run the following command to switch to it.
 perlbrew switch perl-5.12.1-nothreads
```

Perl 5.12.1 mit Threads und Debugging-Informationen

Installation eines Perl 5.12.1 mit Threads und zusätzlichen Debugginginformationen.

```
$ perlbrew install perl-5.12.1 \
   -D=DEBUGGING=both -D=usethreads \
   -as perl-5.12.1-debug
```

Zwischen den verschiedenen Perls hin- und herschalten

Ein Übersicht aller installierten Perls liefert die Option `installed`

```
$ perlbrew installed
  perl-5.12.1
  perl-5.12.1-nothreads
  perl-5.12.1-debug
  /usr/bin/perl
```

Über die Option `switch` kann auf ein anderes Perl umgeschaltet werden. Da `perlbrew` die Umgebungsvariable `PATH` verändert, ist es erforderlich, die Shell über diese Änderung per `hash -r` (manchmal auch rehash) zu informieren.

```
$ perl -v
This is perl, v5.10.1 (*) built \
for i486-linux-gnu-thread-multi

$ perlbrew switch perl-5.12.1
$ hash -r
$ perl -v

This is perl 5, version 12, \
subversion 1 (v5.12.1) built for \
i686-linux-thread-multi
```

Der Tipp-Aufwand läßt sich durch Shell-Aliase erheblich verkürzen:

```
$ alias p5121='perlbrew switch \
  perl-5.12.1; hash -r'
```

Zurück zum System-Perl gelangt man über die Option `off`:

```
$ perlbrew off
$ hash -r
$ perl -v

This is perl, v5.10.1 (*) built \
for i486-linux-gnu-thread-multi
```

CPAN-Module installieren

Wie oben erwähnt, verändert `perlbrew` beim Switchen die Umgebungsvariable PATH und setzt somit auch das `richtige` cpan-Programm in den Pfad.

```
$ perlbrew switch perl-5.12.1-nothreads
$ hash -r

$ cpan
cpan> install YAML
...
Appending installation info to \
/home/bob/perl5/perlbrew/perls/ ...
cpan> quit
```

Installierbare Perl-Versionen

Viele ältere Perl-Versionen lassen sich ohne Patches nicht mit aktuellen C-Compilern kompilieren.

Unter Ubuntu 10.04 lassen sich meiner Erfahrung nach nur Perl-Versionen, die jünger sind als 5.8.9, mit `perlbrew` installieren.

Für Perl-Versionen vor 5.8.9 empfiehlt sich nach wie vor das oben erwähnte `buildperl.pl`, welches die notwendigen Patches mitliefert.

Leserbriefe

Anmerkung zu CPAN-News

Bzgl. der CPAN News XIII: Ich verstehe die Erwähnung von Path::Executable nicht - IPC::Run::can_run tut das bereits aus dem Perl-Core heraus und für mehr Komfort gibt es seit Urzeiten File::Which.

Abgesehen davon ist es sehr unschön, das man Leserbriefe nicht über sein eigenes Mail-Tool verschicken kann.

Besten Gruss,
Jens Rehsack

Anmerkung der Redaktion:
Die Methode `can_run` existiert im Modul `IPC::Cmd`, das ab Perl-Version 5.9.5 im Perl-Core enthalten ist. Vielen Dank für den Hinweis. Die CPAN-News sind nicht zur Bewertung von Modulen gedacht, sondern einfach zur Vorstellung von Modulen.

Leserbriefe können mit jedem Mail-Tool auch an "feedback@foo-magazin.de" geschickt werden.

Anmerkung zum Artikel "HowTo: CPANTester" (Ausgabe 15)

Hi Renée,

der Artikel von mir über CPAN::Reporter wird am 31. August 2010 hinfällig sein, da die CPAN Tester die Mail-Schnittstelle einstellen und durch eine HTTP-Schnittstelle ersetzen.

Lässt sich der Artikel noch rausnehmen?

Grüße
Thomas

Anmerkung der Redaktion:
Als die Ausgabe 15 in Druck ging, war die Abschaltung der Mail-Schnittstelle noch nicht angekündigt. Wir möchten an dieser Stelle auf den Blog-Artikel von Thomas hinweisen, der auf die neue Schnittstelle eingeht: *http://perl-howto.de/2010/07/cpan-tester-20-cpan-tester-werden-ist-ganz-einfach.html*

TPF News

Grant Update: Manual für Spieleentwicklung mit SDL

Kartik Thakore hat von der Perl Foundation einen Grant bekommen, um ein Handbuch zur Spieleentwicklung mit SDL zu schreiben. Für die ersten Kapitel gibt es eine erste Version und für einige weitere Kapitel gibt es eine Themensammlung.

Hier ist der Fortschritt zu verfolgen:

- http://github.com/PerlGameDev/SDL_Manual
- http://sdlperl.ath.cx/releases/SDL_Manual.pdf

Abschlussreport vom "Google Summer of Code" 2010

Der diesjährige "Google Summer of Code" ist vorbei. Auch diesmal waren die Perl Foundation und die Parrot Foundation vertreten.

In diesem Jahr war Jonathan Leto der Organisator für die beiden Organisationen. In seinem Blog hat er jetzt einen Abschlussbericht veröffentlicht, der auch nochmal die beteiligten Projekte der Perl Foundation und der Parrot Foundation vorstellt.

Test::Builder2

Michael Schwern hat große Teile seines Grants umgesetzt. Es gibt noch ein paar Probleme, die er lösen muss. Ein detaillierter Bericht ist unter *http://use.perl.org/~schwern/journal/40421 zu finden*.

Grant akzeptiert: "Verbesserungen am Meta-Model"

Die Perl Foundation gewährt Jonathan Worthington einen weiteren Grant, um an Perl 6 zu arbeiten: Worthington wird das Meta-Model von Perl 6 verbessern. Nicht nur Rakudo sondern auch nqp-rx werden von den Arbeiten profitieren. Durch die Ergebnisse soll auch die Performanz verbessert werden.

Die komplette Beschreibung der Arbeiten ist unter http://news.perlfoundation.org/2010/07/hague-grant-application-meta-m.html zu finden.

Grants für das 3. Quartal 2010

Das Grant Komitee der Perl Foundation hat bekanntgegeben, dass drei Grants für das 3. Quartal 2010 akzeptiert wurden:

- "Manual for Game Development with SDL Perl" von Kartik Thakore
- "Perlbal documentation" von José Castro und Bruno Martins
- "Perl 6 Tablets" von Herbert Breunung

Hague Grant: Lists, Iterators, and Parcels

Der Grant betreffend Listen und Iteratoren wurde angenommen. Patrick Michaud wird daran arbeiten und Details bezüglich des Grants sind unter http://news.perlfoundation.org/2010/07/hague-grant-application-lists.html zu finden.

White Camel Awards 2010

Auf der diesjährigen YAPC::Europe hat brian d foy im Namen der Perl Mongers und O'Reilly Media die White Camel Awards 2010 verliehen.

In diesem Jahr bekamen diesen Preis José Castro für seinen Unermüdlichen Einsatz bei und für die Perl User Groups (Perl Mongers), Paul Fenwick für seine Leistungen rund um die "Werbung" für Perl und Barbie, der eine sehr wichtige Rolle bei den CPANTester einnimmt.

Allen drei Preisträgern "Herzlichen Glückwunsch". Mit diesem Preis sollen Personen ausgezeichnet werden, die sich im nicht-technischen Bereich um Perl verdient gemacht haben.

wxPerl-Dokumentation

Nach einer Auszeit hat Eric Roode wieder an der wxPerl-Dokumentation angefangen. Einige Inhalte hat er in das wxPerl-Wiki eingefügt:

http://wxperl.pvoice.org/w/index.php/Wx::App
http://wxperl.pvoice.org/w/index.php/Wx::Frame
http://wxperl.pvoice.org/w/index.php/Window_Styles
http://wxperl.pvoice.org/w/index.php/Wx::StaticText
http://wxperl.pvoice.org/w/index.php/Help:Editing

Fixing Perl5 Core Bugs

Seit 5 Monaten arbeitet Dave Mitchell an der Behebung von Bugs im Perl5 Core. Im Juli hat er über 75 Stunden an den Bugs gearbeitet. Eine Übersicht ist unter http://news.perlfoundation.org/2010/08/fixing-perl5-core-bugs-report-2.html zu finden.

Mittlerweile hat er rund 73% der im Grant festgeschriebenen 500 Stunden aufgebraucht und schon beachtliche Ergebnisse erzielt.

Im Monat August hat er insgesamt knapp 18 Stunden an dem Grant gearbeitet.

Embeding Perl into C++ Applications

Leon Timmermans kam diesmal nicht so gut voran wie bisher, was zum großen Teil an schulischen Verpflichtungen hängt. Durch Tests sind einige Bugs bei Regulären Ausdrücken zum Vorschein gekommen, die aber gefixt wurden. Mit einer älteren gcc-Version blieben die Versuche, Code unter Windows zu kompilieren, erfolglos. Als nächstes stehen Tests mit dem MSVC an.

Leon hat auf der YAPC::EU mit verschiedenen Leuten über seinen Grant gesprochen - auch über die Probleme unter Windows. Timmermans hat dazu Hilfestellungen bekommen. Aber durch schulische Verpflichtungen hat er sein selbstgestecktes Ziel noch nicht erreicht.

Hague-Grant abgeschlossen: Numeric und Real Support

Dieser Grant für Solomon Foster beinhaltete die Arbeit an den Rollen für "Numeric" und "Real". Diese beiden Rollen werden jetzt in Rakudo eingesetzt. "Numeric" ist die Rolle für alle numerischen Werte und "Real" ist alles was nicht "Complex" ist.

Die komplette Liste der Ergebnisse kann unter http://news.perlfoundation.org/2010/08/completed-hague-grant---numeri.html eingesehen werden.

CPAN News XVI

Const::Fast

Im Buch "Perl Best Practices" wird die Verwendung von `Readonly` empfohlen. Das Modul ist aber relativ langsam. Aus diesem Grund wurde `Const::Fast` entwickelt.

```
use Const::Fast;
const my $foo => 'a scalar value';
const my @bar => qw/a list value/;
const my %buz => (
    a => 'hash', of => 'something'
);
```

Image::Size

Manchmal möchte man wissen, wie groß ein Bild ist, weil die Anzeige dementsprechend angepasst werden soll. Für solche Fälle gibt es `Image::Size`. Die Verwendung ist selbstsprechend:

```
use Image::Size;
my $image = '/path/to/image.jpg';
my ($width,$height) = imgsize( $image );
```

JavaScript::Minifier::XS

"Moderne" Webanwendeung benutzen meist sehr viel JavaScript. Sehr viel JavaScript bedeutet häufig sehr viele Daten, die geladen werden müssen. Und für die Performanz gilt: "Je weniger Daten übertragen werden müssen, umso besser". Während der Programmierung und für die Wartbarkeit ist es besser, wenn man Code lesbar hält. Aber für den Browser ist es egal. Also kann JavaScript ruhig unleserlich sein. Mit `JavaScript::Minifier::XS` kann man alle unnötigen Whitespaces und Kommentare aus JavaScript entfernen.

```
use JavaScript::Minifier::XS qw(minify);

my $JS = <<"JS";
  var test = 'Hallo Welt';
  alert( test );
JS

print minify( $JS );
__END__
var test='Hallo Welt';alert(test);
```

Module::Want

Unsicher, ob ein Modul installiert ist, das man benutzen möchte? Dann kann man mit `eval{ require Module; 1; }` arbeiten. Aber das wird dann jedes Mal aufgerufen wenn perl an diese Codestelle kommt. Mit `Module::Want` passiert das nicht. Da werden die Verzeichnisse in `@INC` einmal abgefragt und dann immer wieder auf dieses Ergebnis zugegriffen.

```
use Module::Want;

if (have_mod('Data::Tabulate')) {
     # mach was
}
else {
     warn "
}
```

Acme::PM::Berlin::Meetings

Das letzte Berlin.pm-Treffen verpasst? Dann könnte in Zukunft `Acme::PM::Berlin::Meetings` helfen. Das Modul berechnet die nächsten Termine der Berliner Perlmongers. Jetzt noch mit Mails oder Notifications verbinden und schon geht der nächste Termin bestimmt nicht vergessen:

```
$ perl -MAcme::PM::Berlin::Meetings \
     -e 'print "$_\n" for next_meeting(6);'
2010-09-29T20:00:00
2010-10-27T20:00:00
2010-11-24T20:00:00
2011-01-05T20:00:00
2011-01-26T20:00:00
2011-02-23T20:00:00
```

Devel::bt

Es gibt leider Stellen, an denen es in Perl einen "Segmentation fault" gibt. Diese sollten an die Perl 5 Porters gemailt werden. Damit für die Porters die Fehlersuche einfacher wird, ist ein Backtrace sehr hilfreich. Mit dem Modul `Devel::bt` ist es mehr als simple einen solchen Backtrace zu erstellen. Der erste Code erzeug einen "Segmentation fault" und das zweite Listing zeigt dann das Ergebnis mit `Devel::bt`.

```
my @a = ( {}, {} );

sub f {
    my ($x) = @_;
    @a = ( {}, {} );
    0 for ();
}

map { f $_ } @a;

$ perl -d:bt bt_test.pl
*** glibc detected *** perl: munmap_chunk(): invalid pointer: 0x081fc738 ***
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6(cfree+0x1bb)[0xb769a6bb]
perl(Perl_mg_free+0x2a)[0x80b536a]
perl(Perl_sv_clear+0x280)[0x80ce650]
perl(Perl_sv_free+0x102)[0x80cecd2]
[...]
======= Memory map: ========
08048000-0814d000 r-xp 00000000 08:01 426130     /usr/bin/perl
0814d000-08151000 rw-p 00104000 08:01 426130     /usr/bin/perl
08151000-08219000 rw-p 08151000 00:00 0          [heap]
b7496000-b74a0000 r-xp 00000000 08:01 278549     /lib/libgcc_s.so.1
```

Termine

November 2010

02. Treffen Frankfurt.pm
Treffen Vienna.pm
04. Treffen Dresden.pm
05.-06. Österreichischer Perl-Workshop
08. Treffen Ruhr.pm
09. Treffen Stuttgart.pm
15. Treffen Erlangen.pm
17. Treffen Darmstadt.pm
Treffen München.pm
20. Hackday 2010 NWE.pm
24. Treffen Berlin.pm
30. Treffen Bielefeld.pm

Dezember 2010

02. Treffen Dresden.pm
04. London Perl-Workshop
07. Treffen Frankfurt.pm
Treffen Vienna.pm
13. Treffen Ruhr.pm
14. Treffen Stuttgart.pm
15. Treffen Darmstadt.pm
18. Russischer Perl-Workshop
20. Treffen Erlangen.pm
28. Treffen Bielefeld.pm
29. Treffen Berlin.pm

Januar 2011

04. Treffen Frankfurt.pm
Treffen Vienna.pm
06. Treffen Dresden.pm
10. Treffen Ruhr.pm
11. Treffen Stuttgart.pm
17. Treffen Erlangen.pm
19. Treffen Darmstadt.pm
Treffen München.pm
25. Treffen Bielefeld.pm
26. 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

Smart-Websolutions

Windolph und Bäcker GbR

Perl-Programmierung

info@smart-websolutions.de