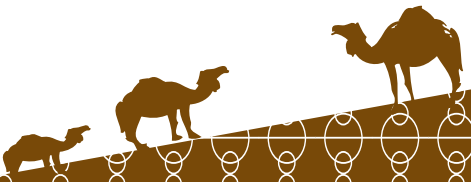
Ausgabe 3/2011 – www.foo-magazin.de - 6,00 €

$foo

PERL MAGAZIN



OpenGL
2D Spieleprogrammierung

Rollen für Nicht-Moose-Anwendungen
Role::Basic

OTRS "LinkedObjects"
Verknüpfte Objekte in OTRS

Nr 19

GeNUA ist ein deutscher Spezialist für IT-Sicherheit. Wir entwickeln Firewalls, Fernwartungs-Systeme und Lösungen für den verschlüsselten Datenaustausch via Internet. Zusätzlich bieten wir ein umfassendes Dienstleistungsspektrum, gestalten IT-Netze nach Sicherheitskriterien und übernehmen das komplette Management komplexer Netzwerke. Unsere Kunden sind große und mittelständische Unternehmen sowie sicherheitsbewusste Behörden. Am Standort Kirchheim bei München sind derzeit rund 150 Mitarbeiter beschäftigt.

ZUR UNTERSTÜTZUNG UNSERES ENTWICKLUNGSTEAM SUCHEN WIR

Perl-Hacker (m/w)

DIE AUFGABENBEREICHE:

Für Unix-basierte Firewall- und Sicherheits-Systeme:

- Entwicklung von applikationsspezifischen Proxies und Diensten auf unserer zweistufigen Firewall in Perl und C
- Konzeption und Umsetzung komplexer Lösungen im sicherheitskritischen Umfeld
- Fehlersuche und -behebung im Zusammenspiel von Applikationen und Systemkomponenten

Für unser internes Unternehmens-Informationssystem:

- Weiterentwicklung und Pflege des internen Unternehmens-Informationssystems
- Ermittlung und Analyse der Anforderungen in unseren Abteilungen im engen Kontakt mit unseren Mitarbeitern
- Planung und Umsetzung der eigenständig erarbeiteten Lösung

IHR PROFIL:

Für Unix-basierte Firewall- und Sicherheits-Systeme:

- Sehr gute Kenntnisse im Bereich Netzwerke TCP/IP und deren Dienste
- Sehr gute Kenntnisse in den Programmiersprachen Perl und C, sowie den gängigen Unix Tools (Shell, make, etc.)
- Erfahrungen mit den Komponenten eines Unix-Systems und deren Zusammenspiel

Für unser internes Unternehmens-Informationssystem:

- Sehr gute Kenntnisse in der Programmiersprache Perl, sowie den gängigen Unix Tools (Shell, make, etc.)
- Gute kommunikative Fähigkeiten und analytisches Denkvermögen
- Verständnis von betrieblichen Abläufen und Prozessen

WIR BIETEN:

- Unbefristeter Arbeitsvertrag mit leistungsgerechter Vergütung
- Flexible Arbeitszeit in einer angenehmen und teamorientierten Umgebung
- Hervorragende Arbeitsbedingungen bei einem der 100 besten Arbeitgeber Deutschlands (Great Place to Work 2011)
- Hohe Arbeitsplatzsicherheit – wir gehören zu den 500 wachstumsstärksten Mittelstandsunternehmen Europas
- Eine äußerst abwechslungsreiche Tätigkeit in einem wachsenden Unternehmen mit modernen, flachen Strukturen

Weitere Informationen zu den o. g. Stellen
finden Sie auf unserer Webseite unter:
http://www.genua.de/genua/career/jobboerse/index.html

Ihre vollständigen und aussagekräftigen Bewerbungsunterlagen senden Sie bitte unter Angabe Ihrer Gehaltsvorstellungen sowie Ihres frühesten Eintrittsdatums per Email an: GeNUA mbH, Personalwesen, z. Hd. Frau Julia Seifüssl, E-Mail: jobs@genua.de

GeNUA

Bewerbungen von Frauen und Männern werden gleichermaßen begrüßt. Mit der Übersendung Ihrer Bewerbungsunterlagen erklären Sie sich damit einverstanden, dass wir Ihre Unterlagen elektronisch erfassen und verarbeiten. Weiterhin werden wir zur Wahrung von Fristen und gesetzlichen Anforderungen Ihre Unterlagen aufbewahren und gegebenenfalls Kopien Ihrer Unterlagen erstellen.

Spenden für die Perl 5 Entwicklung, Perl-Workshop und mehr...

Wer die "TPF-News" verfolgt, wird wissen, dass Dave Mitchell derzeit von der Perl Foundation für das Bugfixing im Perl-Kern bezahlt wird. Dadurch wurden einige wichtige Bugs gefixt. Ein Teil der Spende von booking.com (50.000 US$ in 2008) wurde für diesen Grant aufgewendet, aber das zur Verfügung stehende Geld wird immer weniger.

Damit diese Arbeit auch in Zukunft fortgeführt werden kann, hat die Perl Foundation jetzt einen Spendentopf aufgestellt. Als Ziel wurden ursprünglich 25.000 US$ ausgegeben. Aber schon in der ersten Woche wurde das Ziel dank Wiener Perlmongers, cPanel und booking.com übertroffen - und die privaten Spenden sind da noch nicht einmal mit eingerechnet. Dennoch darf natürlich weitergespendet werden.

Je mehr zusammen kommt, umso länger können Dave Mitchell und Nick Clark an der Verbesserung des Perl-Kerns arbeiten und sich den Themen widmen, die sonst keiner anpacken möchte.

Am 19.-21. Oktober 2011 findet der 13. Deutsche Perl-Workshop in Frankfurt statt. Die Frankfurter Perlmongers freuen sich über jeden Teilnehmer und über viele Vortragsvorschläge, die noch bis zum 14. August eingereicht werden können. Der Termin für das nächste Jahr steht auch schon fest - siehe auch die Werbung der Erlanger Perlmongers in dieser Ausgabe.

In Zukunft möchten wir auch mehr Artikel über Perl-Projekte bringen. In dieser Ausgabe gibt es einen Artikel über LinkedObjects in OTRS. Aber wir würden auch gerne mehr aus anderen Projekten bringen. Wer Vorschläge hat, welche Projekte berücksichtigt werden sollen, schreibt bitte an info@perl-magazin.de. Dann können wir mit Programmierern des Projekts Kontakt aufnehmen und fragen ob diese Artikel besteuern können.

Jetzt wünsche ich Ihnen viel Spaß beim Lesen der 19. Ausgabe von $foo.

Renée Bäcker

Die Codebeispiele können mit dem Code

3nbxyl2

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

The use of the camel image in association with the Perl language is a trademark of O'Reilly & Associates, Inc. Used with permission.

Alle weiterführenden Links werden auf del.icio.us gesammelt. Für diese Ausgabe: http://del.icio.us/foo_magazin/issue19

IMPRESSUM

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

INHALTSVERZEICHNIS

ALLGEMEINES

6 Über die Autoren

48 Rezension - Perl - Klassiker

PERL

34 Serialisieren von Datenstrukturen in binäre Sequenzen

39 Perl 6 - Update 4

MODULE

8 Moose Tutorial Teil V

13 OpenGL - Teil 1

24 Role::Basic

29 WxPerl Tutorial - Teil 8

PROJEKTE

42 Verknüpfte Objekte in OTRS

NEWS

53 TPF News

55 CPAN News

57 Termine

58 LINKS

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.

Tom Kirchner

Tom Kirchner, Berliner Jahrgang 1984, studierter Medieninformatiker, fand vor vielen Jahren zu Perl und erledigt seitdem fast alles in Perl. Er arbeitet zur Zeit als Softwareentwickler in Hamburg sowie als Entwickler bei Perl-Services.de. Daneben ist er bei den Perl Mongers Hamburg aktiv und hŠlt VortrŠge. Sein besonderes Interesse liegt im Bereich Visual Analytics sowie allgemein Grafikprogrammierung und dort v.a. die Visualisierung von Daten mit OpenGL. Zu seinen Lieblingsmodulen auf CPAN zŠhlen u.a. OpenGL, Imager, DBI, Parse:: RecDescent und Dancer. Zu erreichen ist er unter http://www.tomkirchner.com.

Rolf Rost

Seit 1998 entwickle ich Perl-Scripts und Webanwendungen in Perl, schwerpunktmäßig zum Thema Netzwerkmanagement und Abwicklung von Kundenaufträgen im Intranet meiner Firma T-Systems, der ich bis 31.12.2010 angehörte.

Renée Bäcker

Moose Tutorial Teil V: Introspektion

In diesem Teil des Moose-Tutorials geht es um Introspektion. Damit ist das Sammeln von Informationen über Objekte und Klassen zur Laufzeit gemeint. Auch können die Objekte und Klassen zur Laufzeit geändert werden.

Zu den gesammelten Informationen gehören Daten wie die Attribute eines Objektes, welche Rollen in der Klasse eingebunden werden und welche Methoden ein Objekt kennt. Natürlich gibt es viele weitere Informationen, die gesammelt werden können. Dazu werden ein paar Beispiele von Anwendungsfällen gezeigt.

Bei Moose sind die Möglichkeiten der Introspektion sehr vielfältig. Man kann nahezu alle Informationen über ein Objekt oder eine Klasse während der Laufzeit bekommen. Das hängt damit zusammen, dass Moose sogenannte Meta-Klassen intensiv nutzt. In der letzten Ausgabe von $foo wurde schon mehrfach auf die Meta-Klassen von Moose hingewiesen. In dieser Ausgabe wird intensiv mit diesen Meta-Klassen gearbeitet um an die gewünschten Informationen über Klassen und Objekte zu kommen.

Meta-Klassen

Moose baut auf dem Meta-Object Protocol (MOP) auf und verwendet dazu das Modul `Class::MOP`. Moose selbst bringt nur einen kleineren Teil der Features ein, das meiste stammt aus `Class::MOP` (Attribute, Methoden-Modifier,...). Auch die ganzen Möglichkeiten der Introspektion kommen aus dem MOP.

Auf der Suche nach den Methoden, die die gewünschten Informationen liefern sollen, sind die entsprechenden Dokumentationen von `Class::MOP` die erste Anlaufstelle.

Wird eine Moose-Klasse geschrieben, erzeugt Moose ein Objekt einer Meta-Klasse. Steht in der Moose-Klasse ein `extends 'Parent'` - also Vererbung - wird im Hintergrund der Meta-Klasse über `superclasses` gesagt, von welchen Modulen die Klasse erbt.

An das Meta-Objekt einer Klasse kommt man mit der `meta`-Methode:

```
my $meta = Person->meta;
```

Was alles mit Meta-Klassen abgedeckt ist, hängt auch davon ab, welche Moose-Erweiterungen benutzt werden. Auf jeden Fall gibt es Meta-Klassen zu Klassen, Attributen, Rollen und Methoden. Über diese Dinge lässt sich viel herausfinden.

Anwendungsfälle für Introspektion

Doch wofür benutzt man Introspektion eigentlich? Die Anwendungsfälle sind sehr vielfältig. Häufig wird es beim Debugging verwendet. Gerade wenn sich Klassen oder Objekte während der Laufzeit verändern können, ist es wichtig, den Zustand zum Zeitpunkt des Fehlers zu kennen. Da sind nicht nur die Werte von Attributen interessant, sondern auch die Antwort auf die Frage, welche Attribute das Objekt überhaupt kennt.

Aber auch im normalen Ablauf kann Introspektion sehr nützlich sein. Wenn verschiedene Attribute unterschiedlich behandelt werden sollen und diese sich über bestimmte Eigenschaften unterscheiden, muss man mit Introspektion arbeiten.

Oder auch für Dokumentationszwecke ist es geeignet. Man kann die Informationen für den Text oder auch für Graphiken

direkt aus dem Programm holen und muss nicht alles selbst schreiben.

Beispiele für die letzten beiden Anwendungsfälle werden bei den konkreten Beispielen gezeigt.

... im Einsatz

Die folgenden Fragestellungen zeigen nur einen kleinen Ausschnitt von dem was möglich ist, sollen aber damit einen Eindruck davon vermitteln, wie man bei Moose an Informationen über Objekte und Klassen kommt.

Die Methoden einer Klasse

Die Frage, welche Methoden eine Klasse überhaupt kennt, taucht immer wieder auf. Mit Moose ist diese Frage leicht zu beantworten. Die Meta-Klasse `Moose::Meta::Class` bzw. `Class::MOP` kennt die Methode `get_all_methods`.

```
my $obj  = MethodListTest->new;
my @subs = $obj->meta->get_all_methods;

print $_->name,"\n" for @subs;

entwicklung@ubuntu$ perl script.pl
test
dump
DEMOLISHALL
meta
does
new
hallo
DESTROY
BUILDALL
BUILDARGS
DOES
```

Man kann sich jetzt etwas wundern, wenn man sieht, dass in der Klasse nur zwei Methoden definiert sind:

```
use Moose;

sub test {}
sub hallo {}

1;
```

In der Liste der Methoden tauchen hier noch einige Methoden auf, die durch das `use Moose` in die Klasse kommen. Auch Methoden von eingebundenen Rollen und Superklassen werden hier ausgegeben. Sollen die geerbten Methoden nicht mit ausgegeben werden, muss man die Liste filtern.

```
my @subs = $obj->meta->get_all_methods;

my @filtered = grep{
    $obj->meta->has_method( $_->name )
}@subs;

print $_->name,"\n" for @filtered;
```

oder man arbeitet mit `get_method_list`

```
my @subs = $obj->meta->get_method_list;
print $_,"\n" for @subs;
```

Gerade beim Debuggen eines Fehlers spielt es nicht nur eine Rolle, ob das Objekt eine Methode überhaupt kennt, sondern es sind noch mehr Informationen notwendig. Es kann ja sein, dass eine Rolle oder eine andere Klasse diese Methode mit `before` oder `after` modifiziert. Diese Information wird auch benötigt.

```
use ModifierTest;

my $obj = ModifierTest->new;
my $sub = $obj->meta->get_method( 'test' );

my @before = $sub->before_modifiers;
print "Sub was modified" if @before;
```

So erfährt man nur, ob Methoden-Modifier im Einsatz sind (in diesem Fall `before`), aber nicht wo diese Modifier definiert sind.

Nicht nur zu Dokumentationszwecken kann es interessant sein, zu wissen, von welchem Typ die Parameter einer Methode sein müssen. In normalen Moose-Klassen (ohne `MooseX::Declare`) gibt es keine Methodensignaturen, womit es auch nicht möglich ist, diese Informationen zu bekommen. Wird allerdings `MooseX::Declare` oder `MooseX::Method::Signatures` verwendet, kann man diese Informationen bekommen - auch wenn es aufwändig ist und etwas tiefergehende Introspektion zeigt.

Wenn eine Klasse eine Methode `test` implementiert, die als Parameter einen String (`Str`) erwartet:

```
use MooseX::Declare;

class MethodTest {
    method test (Str $msg) {
        print $msg;
    }
}

1;
```

Dann kann der Datentyp des Parameters so herausgefunden werden:

```
use MethodTest;

my $obj = MethodTest->new;
my $sub = $obj->meta->get_method( 'test' );

print $sub->name,"\n\n";

my $signature = $sub->parsed_signature;
my @pos = $signature->_positional_params;
for my $param ( @pos ) {
  for my $elem ( $param->params ) {
    if ( $elem->has_type_constraints ) {
      my $constraints =
          $elem->type_constraints;
      print $constraints->to_string;
    }
  }
}
```

Der Weg bis zu diesem Ergebnis ist recht beschwerlich, weil man sich durch einige Dokumentationen durchschlagen muss. Aber man kann das Schritt für Schritt angehen. Als erstes holt man sich das Objekt der Methode (`get_method`) und von dort aus geht man langsam vorwärts. Wenn man nicht weiß, wie man weitermachen soll, kann man wieder mit einer Liste von Methoden und Attributen anfangen. So landet man bei `parsed_signature`. Schaut man sich dann die Klasse der Signatur an, kommt man auf das Attribut `_positional_params` und so weiter.

Weiter oben wurde schon der Anwendungsfall genannt, dass Attribute je nach Eigenschaft unterschiedlich behandelt werden sollen. Es sollen Veränderungen an den Attributwerten getrackt werden. Dazu werden die Attribute als "is_trackable" gekennzeichnet:

```
my @attributes = qw(user_name website);
for my $attribute ( @attributes ) {
    has $attribute => (
        metaclass    => 'AttrInfo',
        is_trackable => 1,
        is           => 'rw',
        trigger      => sub{
            shift->_dirty_flag($attribute)
        },
    );
}
```

`AttrInfo` stellt hier das label `is_trackable` zur Verfügung. Bei all diesen Attributen soll nach der Initialisierung das "dirty_flag" zurückgesetzt werden. Wenn dann das Attribut nachträglich geändert wird, greift der Trigger und das "dirty_flag" wird wieder gesetzt. So kann im Destruktor festgestellt werden, welche Attribute zur Objektlebenszeit verändert wurden.

Um das unterschiedliche Verhalten von "is_trackable" und anderen Attributen zu ermöglichen, müssen die Attribute überprüft werden und wenn das Flag gesetzt ist, muss das "dirty_flag" zurückgesetzt werden. Dazu ist es erstmal notwendig, die Liste aller Attribute einer Klasse zu bekommen. Hierfür gibt es die Methode `get_attribute_list`:

```
my $meta = $self->meta;
my @attr = $meta->get_attribute_list;
```

Jetzt hat man die Namen aller Attribute, aber es fehlen noch die Metainformationen. Zu diesen Informationen gehören Name, Datentyp, Trigger und eben auch Flags wie "is_trackable". Die Informationen zu einem Attribut bekommt man über die `get_attribute`-Methode der Meta-Klasse:

```
my @trackable  = map{
    $meta->get_attribute( $_ )
}@attr;
```

In einer `for`-Schleife kann dann das "dirty_flag" zurückgesetzt werden wenn das Attribut ein Objekt von "AttrInfo" und das Flag "is_trackable" gesetzt ist:

```
for my $attribute ( @trackable ) {
    if (
        $attribute->isa( 'AttrInfo' ) &&
        $attribute->is_trackable
    ) {
        my $name = $attribute->name;
        $self->delete_flag( $name );
    }
}
```

Als nächstes soll die Vererbungshierarchie und die verbundenen Rollen als Bild dargestellt werden. Für die Grafik wird dabei das Modul `GraphViz` verwendet. Die Informationen über Superklassen und Rollen werden per Introspektion gesammelt (Listing 1). Damit das Bild nicht zu groß und unübersichtlich wird, werden nur Module betrachtet, die mit "Method" oder "Moose" anfangen.

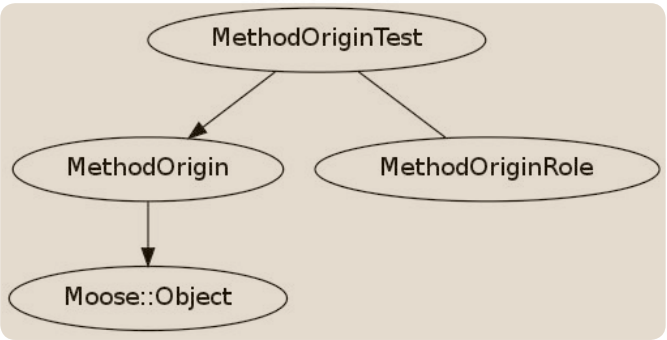


Abbildung 1: Graph mit Vererbung und Rollen

```
use MethodOriginTest;

my $start = 'MethodOriginTest';
find_parents_and_roles( 'MethodOriginTest', $gv );

sub find_parents_and_roles {
    my ($name,$gv) = @_;

    return if $name =~ m{ \A (?:Moose|Method) :: }xms;

    my $meta    = $name->meta;

    if ( $meta->can( 'superclasses' ) ) {
        my @parents = $meta->superclasses;
        if ( @parents ) {
            for my $parent ( @parents ) {
                # fuege Knoten und Verbindungen ein
                find_parents_and_roles( $parent, $gv );
            }
        }
    }
    if ( $meta->can( 'roles' ) ) {
        my @roles   = @{ $meta->roles || [] };

        if ( @roles ) {
            for my $role ( @roles ) {
                # fuege Knoten und Verbindungen ein
                find_parents_and_roles( $role->name, $gv );
            }
        }
    }
}
```

Listing 1

Für andere Dokumentationsaufgaben kann auch das Modul `MooseX::Documenter` herangezogen werden. Das Modul bietet ein paar Methoden, mit denen Informationen zu Elternklassen, Attributen etc. nicht selbst gesammelt werden müssen.

```
use MooseX::Documenter;
my $doc = MooseX::Documenter
          ->new('lib','MethodOriginTest');
my $attr    = $doc->local_attributes;
my $methods = $doc->local_methods;
```

Durch Vererbung oder aber auch das Einbinden von Rollen kommen zusätzliche Methoden in eine Klasse. Aber wo kommen diese Methoden ursprünglich her? Diese Frage lässt sich ebenfalls durch Introspektion sehr einfach beantworten:

```
use MethodOriginTest;

my $obj = MethodOriginTest->new;

my $sub =
    $obj->meta
        ->find_method_by_name( 'test' );

print $sub->original_package_name;
```

In anderen Beispielen wurde `get_method` verwendet. Das sucht aber nur direkt in der Klasse (hier: `MethodOriginTest`) sowie in den eingebundenen Rollen. Da die Methode aber aus einer Superklasse kommt, wird sie mit `get_method` nicht gefunden. `find_method_by_name` sucht auch in den übergeordneten Klassen.

Bis jetzt wurden nur Informationen abgefragt. Introspektion bedeutet aber auch, dass die Objekte bzw. die Klassen verändert werden können. Deshalb werden jetzt ein paar Beispiele gezeigt, wie Klassen zur Laufzeit verändert werden können.

```
my $obj = MyTest->new;
my $meta = $obj->meta;

eval{
    $obj->test();
    1;
} or print $@;

$meta->add_method(
    'test',
    sub{
        print 'Methode hinzugefügt';
    }
);

eval{
    $obj->test();
      1;
  } or print $@;
```

Gibt folgendes aus:

```
Can't locate object method "test" via \
  package "MyTest" at ... line 12.
Methode hinzugefügt
```

Mit `remove_method` kann man genauso gut Methoden löschen. Das Hinzufügen oder Löschen von Methoden während der Laufzeit macht sicherlich beim Debugging Sinn. In normalen Programmen wird das zu einem Albtraum in Sachen Wartbarkeit werden.

Die hinzugefügten Methoden können dann genauso untersucht werden wie andere Methoden auch. Z.B. mit

```
my $sub = $meta->get_method( 'test' );
print "\n", $sub->original_package_name;
```

Bereits in der letzten Ausgabe wurde gezeigt, wie mit dem Meta-Objekt weitere Rollen einem Objekt zugewiesen werden können. Das ist im Gegensatz zu dem Hinzufügen/Löschen von Methoden ein Anwendungsfall, der häufiger Sinn macht.

```
use Person;
use Speaker;

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

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

Der Artikel hat gezeigt, dass mit Introspektion eine Menge möglich ist. Man kann die Objekte und Klassen genau analysieren und kann mit den Ergebnissen Dokumentation erstellen oder auch das Verhalten einer Klasse ändern.

ThinkTank
Management Consulting

ThinkTank Management Consulting ist ein Personalberatungsunternehmen, das sich auf Führungs- und Fachkräfte in der IT-Industrie spezialisiert. Wir haben v.A. Positionen im Entwicklungs- und Beratungsbereich zu besetzen. Für verschiedene Kunden suchen wir dauerhaft engagierte

Perl-Entwickler

sowohl für Festanstellung wie auch für freiberufliche Projekte. Unsere Kunden sind mittelständische Unternehmen mit Schwerpunkt in der Webentwicklung, ansässig in Großraum Frankfurt, Nürnberg und Hamburg.
Mehr über die verschiedene Positionen finden Sie auf unsere Webseite: www.thinktank-group.com.
Bei Interesse können Sie uns unter +49 (0)6196 2021114 oder office@thinktank-group.com kontaktieren. Wir sind täglich, natürlich auch am Wochenende bis 21:00 Uhr gerne für Sie da.

IT-Fach- und Führungskräften, die sich generell eine Änderung wünschen, bieten wir eine kostenlose

Karriereberatung

an. Auf dem Hintergrund Ihres bisherigen Werdegangs zeigen wir Ihnen verschiedene Entwicklungsmöglichkeiten und geben Empfehlungen für weitere Schritte. Wir geben Ihnen auch wertvolle Tipps für den Aufbau ihrer Bewerbungsunterlagen. Kontaktieren Sie uns einfach unter den oben genannten Daten.

ThinkTank Management Consulting. Bad Soden a.T. 06196 2021114 - www.thinktank-group.com

Tom Kirchner

2D Spieleprogrammierung mit OpenGL - Teil 1

Diese Artikelreihe gibt eine kleine Einführung in die Spieleprogrammierung mit OpenGL und GLUT. Dabei wird das Spiel PONG programmiert und in mehreren Ausbaustufen verschönert. Der Einfachheit halber wird die gesamte Spielgrafik in 2D gehalten, aber die vermittelten Konzepte lassen sich leicht in 3D übertragen.

Grundlagen von OpenGL/GLUT

Was ist OpenGL?

OpenGL (Open Graphics Language) ist eine API, die von der Khronos Group entwickelt und gepflegt wird, mit der man grafische Primitive, z.B. geometrische Objekte und Bitmaps, rendern kann. Da OpenGL von den meisten Grafikkarten in der Hardware implementiert wird, geht das um ein Vielfaches schneller als über die CPU, was sich besonders bei grafikintensiven Spielen bemerkbar macht.

Was kann man mit OpenGL machen?

Neben der Spieleprogrammierung kann man OpenGL zur Programmierung aller möglichen grafischen Anwendungen verwenden: Visualisierungen, Bildbearbeitung, Animationen uvm. sind möglich. Zum Beispiel verwendet Mac OS X sowie viele andere GUI Toolkits zur Darstellung und Beschleunigung ihrer Oberflächen die Fähigkeiten von OpenGL. Adobe Photoshop sowie viele Bildbearbeitungsprogramme verwenden OpenGL für die eigentliche Bearbeitung der Bilder. Mit Einführung von OpenGL ES, einem OpenGL basierten Standard für mobile Endgeräte, ist es nunmehr auch möglich neben Windows, Unix und Mac OS X, OpenGL auch auf mobilen Endgeräten, wie Smartphones und Tablet PCs, zu verwenden.

GLUT und weitere Hilfsbibliotheken

Die Aufgabe von OpenGL ist klar umrissen, nämlich das Rendern von grafischen Primitiven in den Framebuffer der Grafikkarte. Mindestens eine weitere API wird benötigt, um das gerenderte Bild vom Framebuffer abzurufen und z.B. auf dem Bildschirm darzustellen. Hinzu kommen noch Aufgaben wie die Verarbeitung von Benutzereingaben. Eine oft verwendete API für diese Aufgaben ist GLUT (Graphics Language Utility Language).

GLUT übernimmt das Erzeugen von Fenstern, in denen dann der Framebufferinhalt per OpenGL gerendert wird. Tastendrücke, Mausklicks, Änderungen der Fenstergröße uvm. werden von GLUT abgefangen und können vom Programmier mit Callbacks verarbeitet werden.

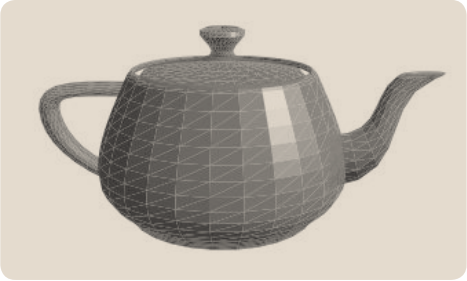


Abbildung 1: Der oft gezeigte Teapot, gerendet mit OpenGL. Man sieht, dass die Oberfläche nicht glatt ist, sondern aus vielen Dreiecken zusammengesetzt ist.

Abbildung 2: Der Teapot, diesmal ohne die Linien der Dreiecke.

Möchte man ein Spiel programmieren, so wünscht man sich oft auch Sounds. Im Zusammenhang mit OpenGL wird dabei oft OpenAL (Open Audio Language) genannt. Mit OpenAL lassen sich mehrere Audiodateien leicht abspielen und Positionen im Raum zuordnen, so dass ein 3D-Klang entsteht. Das Mixen und Verarbeiten der Sounds übernimmt OpenAL.

Exkurs: Was ist der Framebuffer?
Der Framebuffer ist ein Speicherbereich in der Grafikkarte und meist gibt es sogar mehrere. Er ist im einfachsten Fall ein 2D Array von Pixeln. Ein Pixel besteht aus ein oder mehr Werten, i.d.R. sind das 4 Werte, die Farbanteile des Pixels (rot, grün und blau) und als vierten Wert den Alpha-Wert, der die Transparenz des Pixels speichert.

Wie werden nun die grafischen Primitive, z.B. ein Dreieck, in den Framebuffer gerendert? Sie durchlaufen im Grunde die OpenGL Rendering Pipeline, ein immer gleicher Ablauf, der vereinfacht wie folgt aussieht:

• Geometrie: OpenGL sammelt alle geometrischen Informationen, die es braucht, um das Dreieck zu zeichnen. In diesem Fall sind das die drei Eckpunkte (Vektoren) des Dreiecks, gegeben als 2D oder 3D Koordinaten.

• Projektion:
Nun werden die drei Vektoren jeweils mit der Projektionsmatrix der Szene multipliziert. Das ist nötig, da die Vektoren ja im 3D Raum definiert sind und letztendlich auf einer 2D Fläche, dem Framebuffer, dargestellt werden sollen.

• Füllen (Shading)
Nun kennt OpenGL die Pixelfläche im Framebuffer, die von dem Dreieck verdeckt wird und kann nun diese Pixel entsprechend einfärben. Shading kann sehr einfach, z.B. per Flat Shading oder auch sehr komplex, z.B. globale Beleuchtung, implementiert werden. Das in OpenGL eingebaute "Smooth Shading" reicht für das hier entwickelte Programm aus.

Beim Betrachten des Ablaufs fallen zwei Dinge auf:

Erstens kennt OpenGL intern nur seine Primitive, also Dreiecke (Triangles), Vierecke (Quads) und einige mehr. Komplexe Oberflächen werden aus vielen dieser Primitiven zusammengesetzt.

Zweitens ist OpenGL sehr vergesslich, denn sobald die Vektoren (und anderen Dinge, wie z.B. Farben) die OpenGL Pipeline durchlaufen haben, werden sie vergessen. Nur ihr "Vermächtnis" im Framebuffer in Form von eingefärbten Pixelns bleibt bestehen.

Exkurs: Alternativen zu GLUT?
GLUT ist teilweise etwas angestaubt, aber nach Meinung des Autors das OpenGL Fenster-Toolkit, was am stabilsten, am einfachsten zu erlernen und dennoch übersichtlich ist.

Alternativen zu GLUT sind einerseits die bekannten voll ausgereiften GUI Toolkits wie z.B. Qt, GTK, wxWindows, die alle spezielle OpenGL-Widgets erzeugen können. All diesen Toolkits its gemein, dass sie umfangreich sind. Daneben gibt es noch leichtgewichtigere Frameworks, die sich meist auf die Themen konzentrieren, die auch GLUT abdeckt, z.B. GLFW und CPW.

GLFW hat gegenüber GLUT den Nachteil, dass es nur ein Fenster zur Zeit erzeugen kann, CPW dagegen läuft derzeit nur unter Windows.

Perl Bindings für OpenGL und GLUT

OpenGL ist von Haus aus eine C API, d.h. für die Verwendung mit Perl werden Bindings benötigt. Für Perl gibt es zwei OpenGL Bindings:

• **OpenGL (aka POGL)**: POGL lässt sich per `use OpenGL;` laden und mappt alle OpenGL und GLUT C Funktionen auf Perl-Funktionen. POGL ist sehr schnell und plattformkompatibel, allerdings läuft es nicht unter Mac OS X 10.6 (Snow Leopard) oder höher.

• **OpenGL::Simple & OpenGL::Simple::GLUT**: Diese Module bieten ebenfalls Bindings, wobei hier einige C Funktionen zusammengefasst wurden, was durch die flexibleren Perl-Datentypen im Gegensatz zu den nativen C Typen möglich wird.

Im Regelfall ist man mit POGL gut beraten. Wenn es aber darum geht, OpenGL unter Mac OS X 10.6 zu programmieren, scheiden die GLUT Bindings von POGL aus, d.h. man kann eine Kombination aus POGL und OpenGL::Simple::GLUT verwenden:

```
use OpenGL qw(
   :glfunctions :glconstants :glufunctions
   :gluconstants);
use OpenGL::Simple::GLUT qw(:all);
```

Wie ist die API aufgebaut?

OpenGL und GLUT bestehen aus C Funktionen und Konstanten, die in drei Gruppen eingeteilt werden:

- **Kernfunktionen:** Diese beginnen mit `gl`, z.B. `glVertex3f()`.
- **Utilityfunktionen**: Diese beginnen mit `glu` z.B. `gluOrtho2D()`.
- **GLUT-Funktionen**: Diese beginnen mit `glut`, z.B. `glutWindowCreate()`.

Während in Perl der Skalar Datentyp alle einfachen Datentypen wie Strings, Ganzzahlen, Fließkommazahlen etc. abdeckt, verarbeitet die Grafikkarte nur ganz bestimmte Datentypen und das verschieden effizient. Daher gibt es für einige Verwendungen jeweils mehrere C Funktionen, die verschiedene Parametertypen entgegennehmen. Diese Funktionen verfügen über einen Suffix, der Aufschluss über die Anzahl und Typ der Parameter gibt. Die wichtigsten Suffixe sind:

- *f: Die Funktion verarbeitet Fließkommazahlen, z.B. `glVertex3f()`.
- *i: Die Funktion verarbeitet Ganzzahlen, z.B. `glVertex3i()`.
- *fv: Die Funktion verarbeitet ein Array von Fließkommazahlen, z.B. `glVertex3fv()`.
- *iv: Die Funktion verarbeitet ein Array von Ganzzahlen, z.B. `glVertex3iv()`.

PONG Basisprogramm

Jetzt geht es endlich ans Programmieren. Als Einstieg dient ein Basisprogramm, was im Grunde schon alles enthält, was man für Pong braucht. Zuerst ein Screenshot des Basisprogramms - siehe Abbildung 3.

Alle im Folgenden verwendeten Funktionen sind einmal kurz im Quelltext erläutert sowie am Ende nochmals ausführlich in einer Übersicht beschrieben.

Programmkopf

Es werden alle benötigten Module geladen. Es gibt hier wie eben beschrieben zwei Varianten. Die erste gilt für alle Plattformen, außer Mac OS X 10.6:

```
use strict;
use OpenGL qw(:all);
```

Die zweite Variante gilt für Mac OS X 10.6 oder höher:

```
use strict;
use OpenGL qw(:glfunctions :glconstants
              :glufunctions :gluconstants);
use OpenGL::Simple::GLUT qw(:all);
```

main() Funktion

Dies ist der Einstiegspunkt in das Programm. Hier wird ein Fenster erzeugt, die Callbacks an Fensterereignisse gebunden sowie die Ereignisbehandlung gestartet - siehe Listing 1.

Funktion für die Anpassung der Fenstergröße

Diese Funktion wird einmal bei Programmstart aufgerufen sowie immer dann, wenn sich die Fenstergröße verändert. Die Funktion bietet sich an, um die Projektion der Szene zu definieren - siehe Listing 2.

In OpenGL wird jeder Punkt (Vektor) jedes Objektes, bevor es auf den Framebuffer gezeichnet wird, mit der Projektionsmatrix multipliziert. Details zu Projektionen sind im Exkurs

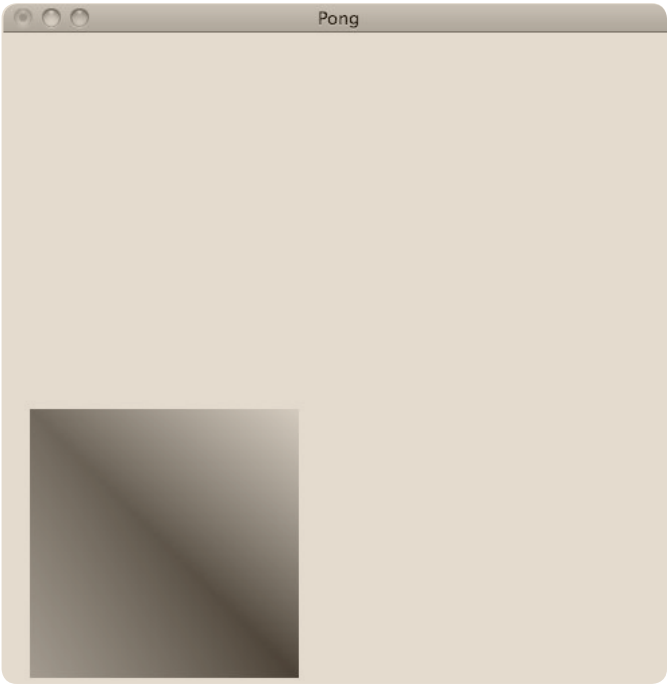


Abbildung 3: Das Basisprogramm zeichnet ein farbiges Quadrat in die linke untere Ecke des Fensters

beschrieben. Für Pong ist eine 2D Projektion sinnvoll, die so angelegt ist, dass eine Längeneinheit in der Szene (1.0) genau einem Pixel im Framebuffer entspricht. Wenn also z.B. ein Quadrat mit den Szenenkoordinaten (1/0), (1/1), (0/1) und (0/0) gezeichnet wird, so bedeckt es genau den Pixel im Fenster links unten.

```
main();

sub main {
  # "initialisiere GLUT"
  glutInit();

  # "Das folgende Fenster soll folgende Eigenschaften haben:"
  glutInitDisplayMode(
    GLUT_DOUBLE |       # 2 Framebuffer beinhalten
    GLUT_RGBA |         # pro Pixel 4 Werte (rot, gruen, blau, alpha) besitzen
    GLUT_DEPTH          # einen zusaetzlichen Buffer fuer Tiefeninformationen
  );

  # "Das folgende Fenster hat die Abmessungen 500x500"
  glutInitWindowSize(500, 500);

  # "Erzeuge ein Fenster mit Titel 'Pong'"
  glutCreateWindow("Pong");

  # "Binde die Perlfunktion 'display' an das Ereignis 'Neuzeichnen'"
  glutDisplayFunc(\&display);

  # "Binde die Perlfunktion 'reshape' an das Ereignis 'Fenstergroessenaenderung'"
  glutReshapeFunc(\&reshape);

  # "Aktiviere den Tiefeninformations-Buffer"
  glEnable(GL_DEPTH_TEST);

  # "Benutze weiss als Hintergrundfarbe des Fensters"
  glClearColor(1,1,1,1);

  # "Starte die Ereignisbehandlung (Endlosschleife)"
  glutMainLoop();
}
```

Listing 1

```
sub reshape {
  # die neue Breite und Hoehe kommen als Argumente an
  my ($w, $h) = @_;

  # "der Bereich des Fensters, fuer den die nachfolgenden Befehle gelten"
  glViewport(0,0, $w, $h);

  # "Die nachfolgenden Matrixoperationen beeinflussen die Projektion der Szene"
  glMatrixMode(GL_PROJECTION);

  # "Lade die Einheitsmatrix auf den aktuellen Stack"
  glLoadIdentity();

  # "Erzeuge eine Projektionsmatrix auf dem aktuellen Stack, die
  #  eine orthographische Projektion auf die Szene anwendet"
  glOrtho(
    0,      # linke Begrenzung der Projektion
    $w,     # Breite der Projektion
    0,      # untere Begrenzung der Projektion
    $h,     # Hoehe der Projektion
    -10.0,  # alle Objekte naeher als -10 werden nicht gezeichnet
     10.0,  # alle Objekte weiter entfernt als 10 werden nicht gezeichnet
  );
}
```

Listing 2

Das OpenGL Koordinatensystem ist so angelegt, wie in Abbildung 4 dargestellt.

Exkurs: OpenGL Matrix Stacks & Objekterzeugung
Um die OpenGL API möglichst einfach zu halten, auch um die Implementierung in der Hardware zu vereinfachen, hat man sich entschieden, OpenGL als eine sogenannte "Statemachine" anzulegen.

Im Gegensatz zur Objektorientierten Programmierung, wo der Zustand von Objekten (z.B. Fenster, geometrische Objekte, Transformationen etc.) zusammen mit den Objekten gespeichert wird, wird in einer Statemachine der Status einzelner Objekte über Funktionen modifizert bzw. abgerufen. Der Status selbst ist dabei in der Grafikkarte gespeichert.

Das Konzept soll anhand eines Vergleichs mit einer objektorientierten OpenGL API verdeutlicht werden:

Objekterzeugung mit fiktiver objektorientierter OpenGL API
Vorausgesetzt eine Klasse "Quad", die dem OpenGL Primitiv QUAD entspricht, und aus vier Vektoren besteht.

```
sub square_oo {
  my $square =
    new Quad(
      new Vertex(1,0),
      new Vertex(1,1),
      new Vertex(0,1),
      new Vertex(0,0));
  return $square;
}
```

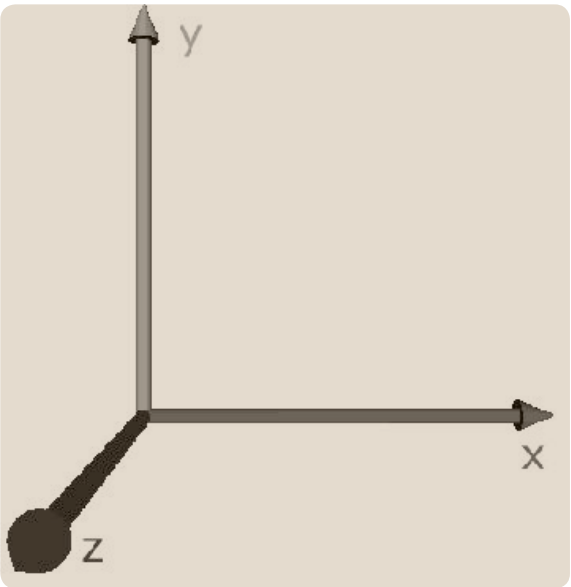


Abbildung 4: Das OpenGL Koordinatensystem.

Objekterzeugung in OpenGL

```
sub square {
  glBegin(GL_QUAD);
    glVertex2f(1,0);
    glVertex2f(1,1);
    glVertex2f(0,1);
    glVertex2f(0,0);
  glEnd();
  return 1;
}
```

Während in der fiktiven OO API der Zustand des Quadrats in der Instanz `$square` gespeichert wird, wird in OpenGL das Quadrat in "Echtzeit" erzeugt. Also die einzelnen Punkte, die das Quadrat beschreiben werden direkt an die Grafikkarte gegeben, die die vier Punkte zu einem Quadrat vereint und auf den Framebuffer projiziert (s.a. Framebuffer Exkurs).

Es ist klar, dass nach diesem Aufruf der Zustand des Quadrats verloren ist. Die Objekterzeugung muss also für jedes Frame neu angestoßen werden.

Um nun wirklich komplexe geometrische Objekte zu erzeugen, bedarf es einer Möglichkeit zur Bildung von Hierarchien. Dies hat man in OpenGL so gelöst, dass es mehrere Stacks gibt. Jedes Mal wenn man eine neue Matrix auf den Stack legt (Aufruf von `glPushMatrix()`), erzeugt man damit eine neue Hierarchieebene, mit eigenen Transformationen. Mit `glPopMatrix()` wird die aktuelle Hierarchieebene "abgeschlossen" und man gelangt wieder in die darüberliegende. Schematisch dargestellt:

```
# Es wird eine neue Hierarchieebene erzeugt
glPushMatrix();
  # Hier werden die Transformationen dieser
  # Hierarchieebene definiert
  # Verschieben um einen Vektor
  glTranslate3f(1,1,1);
  # Skalieren in 3 Dimensionen
  glScalef(2,3,4);
  # Rotation von 90 Grad um den Vektor (1/0/0)
  glRotatef(1,0,0, 90);

  # Hier werden nun die geometrischen
  # Primitive gezeichnet, auf die alle
  # Transformationen oben (und der
  # darueberliegenden Hierarchieebenen)
  # angewandt werden
  square();
glPopMatrix();
```

Ein folgt ein weiteres Beispiel, was mehrere Hierarchieebenen erzeugt und dadurch die square() Funktion wiederverwendet. Es werden drei Quadrate gezeichnet, die jeweils an

verschiedene Positionen verschoben werden und um verschiedene Winkel rotiert werden:

```
glPushMatrix();
  # Diese Transformationen werden auf alle
  # Unter-Hierarchieebenen angewandt:
  glScalef(2,2,1);

  # 1. Quadrat
  glPushMatrix();
    glTranslate3f(2,2,0);
    glRotatef(1,0,0, 90);
    square();
  glPopMatrix();

  # 2. Quadrat
  glPushMatrix();
    glTranslate3f(4,2,0);
    glRotatef(1,0,0, 90);
    square();
  glPopMatrix();

  # 3. Quadrat
  glPushMatrix();
    glTranslate3f(6,2,0);
    glRotatef(1,0,0, 90);
    square();
  glPopMatrix();
glPopMatrix();
```

Funktion für das Neuzeichnen und die Kamera

Die Funktion display(), die in der main() Funktion an das Ereignis "Neuzeichnen" gebunden wurde, wird immer dann aufgerufen, wenn das Fenster neu gezeichnet werden soll. - siehe Listing 3.Die Funktion definiert in der Regel die Kamera und übernimmt das Zeichnen der geometrischen Objekte.

Im Basisprogramm wird ein farbiges Quadrat dargestellt, was mit der folgenden Funktion erzeugt wird:

```
sub colorsquare {
  glBegin(GL_QUADS);
    glColor3f(1,0,0);
    glVertex3f(0,1,0);

    glColor3f(0,1,0);
    glVertex3f(0,0,0);

    glColor3f(0,0,1);
    glVertex3f(1,0,0);

    glColor3f(1,1,0);
    glVertex3f(1,1,0);
  glEnd();
}
```

Zeichnen der Pong Bestandteile

Es gilt nun das Basisprogramm so zu erweitern, dass die drei wichtigsten Pong Bestandteile gezeichnet werden: Der Ball und die beiden Spieler.

```
sub display {
  # "Loesche den Framebuffer, genauer die Farben und Tiefeninformationen"
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  # "Die folgenden Anweisungen beziehen sich auf die Szenengeometrie"
  glMatrixMode(GL_MODELVIEW);

  # "Lade die Einheitsmatrix auf dem Geometrie-Matrix-Stack"
  glLoadIdentity();

  # "Definiere die Kamera mit folgenden Eigenschaften"
  gluLookAt(
    0.0, 0.0, 2.0, # Position der Kamera
    0.0, 0.0, 0.0, # Dorthin schaut die Kamera
    0.0, 1.0, 0.0  # Das ist der Vektor, der nach oben zeigt
  );

  # Zeichnen der Geometrie
  draw();

  # "Zeige den soeben befuellten Framebuffer an"
  glutSwapBuffers();
}

sub draw {
  glPushMatrix();
    glTranslatef(20, 20, 0);
    glScalef(200, 200, 1);
    colorsquare();
  glPopMatrix();
}
```

Listing 3

Der Pong Ball

Der Ball wird durch einen kreisrunde Fläche repräsentiert, die man am Einfachsten mit dem OpenGL Primitiv "TRIANGLE_FAN" zeichnet:

```
sub ball {
  my $segments = 16;
  my @verts = ();
  foreach my $a (0..$segments) {
    my $angle = $a * (360 / $segments);
    push @verts, [sin($angle * 3.1415 / 180),
                  cos($angle * 3.1415 / 180)];
  }

  glBegin(GL_TRIANGLE_FAN);
    glColor3f(0.5,0.8,1);
    glVertex3f(-0.2, 0.2, 0);

    glColor3f(0.3,0.6,0.8);
    map { glVertex3f(@{$_}, 0) } @verts;
  glEnd();
}
```

In der draw() Funktion wird nun statt `colorsquare()` die `ball()` Funktion aufgerufen. Da der Ball innerhalb eines 1x1 Bereiches gezeichnet wird und er damit nur einen Pixel ausfüllt, wird er auf 10 x 10 Pixel skaliert und dann an die Position (50/50) verschoben.

```
sub draw {
  glPushMatrix();
    glTranslatef(50, 50, 0);
    glScalef(10, 10, 1);
    ball();
  glPopMatrix();
}
```

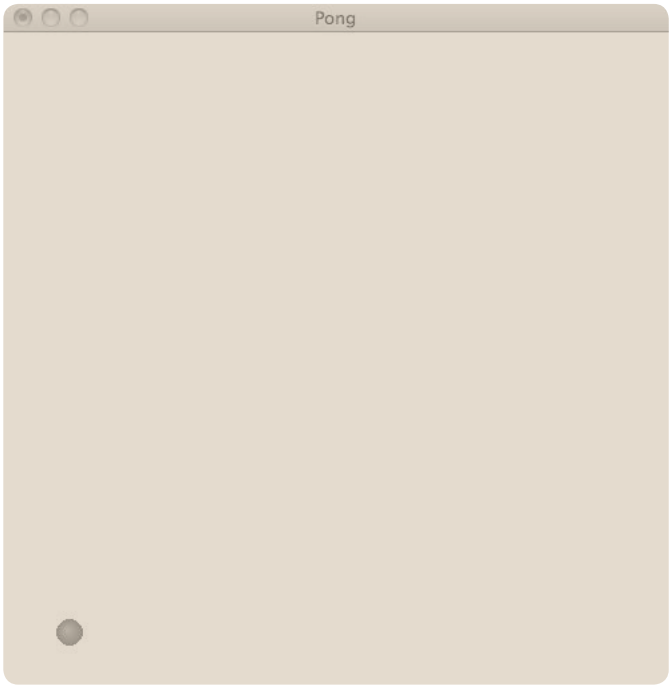


Abbildung 5: Das Programm - mit Pong Ball - sieht jetzt so aus.

Die Pong Spieler

Ein Spieler wird als längliches Rechteck gezeichnet. Ein solches Rechteck können wir aus dem Quadrat des Basisprogramms herleiten, indem wir es in der einen Richtung stauchen.

```
sub player {
  my @red = (1,0,0);
  my @lightRed = (1,0.8,0.8);

  glBegin(GL_QUADS);
    glColor3f(@red);
    glVertex3f(0,1,0);

    glColor3f(@red);
    glVertex3f(0,0,0);

    glColor3f(@lightRed);
    glVertex3f(1,0,0);

    glColor3f(@lightRed);
    glVertex3f(1,1,0);
  glEnd();
}
```

In der `draw()` Funktion rufen wir nun die `player()` Funktion zweimal auf, für jeden Spieler einmal, wobei wir die Spieler auch an verschiedene Positionen verschieben.

```
sub draw {
  # Ball
  glPushMatrix();
    glTranslatef(250, 250, 0);
    glScalef(10, 10, 1);
    ball();
  glPopMatrix();

  # Spieler 1
  glPushMatrix();
    glTranslatef(20, 100, 0);
    glScalef(20, 50, 1);
    player();
  glPopMatrix();

  # Spieler 2
  glPushMatrix();
    glTranslatef(460, 300, 0);
    glScalef(20, 50, 1);
    player();
  glPopMatrix();
}
```

Letztendlich beinhaltet das Programm nun alle statischen Bestandteile, die es braucht um PONG zu implementieren. Im zweiten Teil der Artikelreihe werden der Ball und die Spieler physikalisch korrekt animiert sowie mit Texturen versehen.

Beschreibung der verwendeten OpenGL Funktionen

Im Nachfolgenden Abschnitt werden folgende Kurzbezeichnungen für die Parameter-Datentypen verwendet: i für Ganzzahl, r für reelle Zahl zwischen 0 und 1, sub für Funktionsreferenz, c für Konstante und s für String (Zeichenkette).

glutInit()

`glutInit()` initialisiert die GLUT Bibliothek. Erst danach sollten andere GLUT-Funktionen aufgerufen werden.

glutInitDisplayMode(c Variablenname z.B. GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH)

`glutInitDisplayMode()` setzt den Display-Modus des nächsten zu erzeugenden GLUT-Fensters. Der Modus wird über eine oder mehr mit Bit-OR verknüpften GLUT Konstanten definiert, die jeweils bestimmten Eigenschaften des Fensters entsprechen. Die wichtigsten Konstanten sind `GLUT_DOUBLE`, was einem Fenster mit zwei Framebuffern entspricht, `GLUT_RGB` bzw. `GLUT_RGBA`, was einem Fenster mit 3 bzw. 4 Farbwerten pro Pixel entspricht. `GLUT_DEPTH` gibt an, dass das Fenster einen Tiefenpuffer enthält, der dazu dient überlappungen von Primitiven beim Zeichnen zu berücksichtigen.

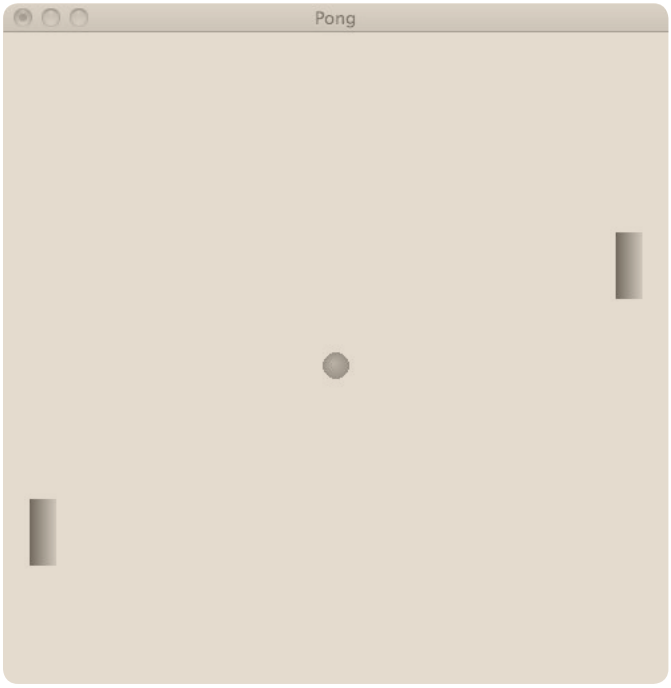


Abbildung 6: Das Programm - mit Pong Spielern - sieht jetzt so aus.

glutInitWindowSize(i Breite, i Höhe)

`glutInitWindowSize()` setzt die Maße des nächsten zu erzeugenden Fensters.

glutCreateWindow(s Titel)

`glutCreateWindow()` erzeugt ein Fenster mit dem angegebenen String als Titel.

glutDisplayFunc(sub Funktion)

`glutDisplayFunc()` setzt die Funktion, die ausgeführt wird, wenn der Fensterinhalt neu gezeichnet werden soll. Das Ereignis wird ausgelöst, wenn `glutPostRedisplay()` ausgeführt wird.

glutReshapeFunc(sub Funktion)

`glutReshapeFunc()` setzt die Funktion, die ausgeführt wird, wenn sich die Fenstergröße ändert, sowie das erste Mal, wenn das Fenster angezeigt wird.

glEnable(c Variablenname z.B. GL_DEPTH_TEST)

`glEnable()` aktiviert bestimmte Teile der OpenGL Rendering Pipeline, die Auswirkungen auf die nachfolgenden OpenGL Befehle haben. Durch die angegebene Konstante wird der aktivierte Teil angegeben, z.B. aktiviert `GL_DEPTH_TEST` die Funktionalität, für jedes Primitiv vor dem Zeichnen einen Abgleich mit dem Tiefenpuffer durchzuführen und so zu entscheiden, ob es verdeckt wird oder nicht.

glClearColor(r Rotanteil, r Grünanteil, r Blauanteil, r Transparenz)

`glClearColor()` setzt die Farbe die verwendet wird, um den Framebuffer mit einem Aufruf der Funktion `glClear()` zu initialisieren.

glutMainLoop()

`glutMainLoop()` startet die Ereignisbehandlung von GLUT. Diese Funktion endet niemals, d.h. ein Aufruf von `glutMainLoop()` ist oft die letzte Anweisung im Programm.

glutGet(c Variablenname z.B. GLUT_ELAPSED_TIME)

Mit `glutGet()` kann man Stati und andere Informationen zum aktuellen Zustand von GLUT abrufen. Die angegebene Konstante definiert, was man abrufen möchte, z.B. gibt `GLUT_ELAPSED_TIME` die seit Programmstart vergangene Zeit an.

glutPostRedisplay()

`glutPostRedisplay()` weist GLUT an, sobald wie möglich den Fensterinhalt neu zu zeichnen, d.h. die per `glutDisplayFunc()` definierte Funktion aufzurufen.

glViewport(i X, i Y, i Breite, i Höhe)

`glViewport()` setzt den Bereich des aktuellen Fensters, in das gezeichnet werden soll. Wird durch eine nachfolgende Aktion, z.B. Zeichnen eines Dreiecks, ein Pixel außerhalb des aktuellen Viewports betroffen, so bleibt dieser Pixel unverändert. Der Ursprung der Pixelkoordinaten ist die untere linke Ecke des Fensters.

glMatrixMode(c Variablenname z.B. GL_PROJECTION)

`glMatrixMode()` setzt den aktuellen Matrix-Stack, worauf sich z.B. nachfolgende Aufrufe von `glLoadIdentity()` etc. beziehen.

glLoadIdentity()

`glLoadIdentity()` legt die Einheitsmatrix auf den aktuellen Matrix-Stack. Diese Matrix dient als Ausgangspunkt für weitere damit multiplizierte Matrizen und hat für sich alleine keinen Effekt.

glOrtho(r Linke, r Rechte, r Untere, r Obere Grenze, r naheste Grenze, r weiteste Grenze)

`glOrtho()` multipliziert die oberste Matrix des aktuell ausgewählten Matrix-Stacks mit einer orthogonalen Projektionsmatrix mit den angegeben Eigenschaften. Die orthogonale Projektion wird definiert über die Grenzen, links, rechts, unten und oben, ausgedrückt in Szenen-Längenangaben. Zusätzlich kann über die "nahe" und "ferne" Grenze noch angegeben werden, wann Objekte zu nah oder zu weit entfernt sind, um noch Teil der Darstellung zu sein.

glBegin(c Variablenname z.B. GL_LINE_LOOP)

`glBegin()` leitet die Definition eines OpenGL Primitivs ein. Z.B. bedeutet `GL_LINE_LOOP`, dass die nachfolgenden Aufrufe von `glVertex()` Punkte definieren, die Teil des OpenGL Primitivs "Line Loop" sind.

glColor4f(r Rotanteil, r Grünanteil, r Blauanteil, r Transparenz)

`glColor4f()` setzt die aktuell für nachfolgende Punkte verwendete Farbe, mit oder ohne Transparenz.

glVertex3f(r X, r Y, r Z)

`glVertex3f()` fügt einen 3D Punkt in das aktuell zu definierende OpenGL Primitiv an. Je nach Typ des Primitivs und wie viele `glVertex3f()` Aufrufe im selben Primitiv davor stattfanden, hat der definierte Punkt eine andere Bedeutung. Der erste Aufruf von `glVertex3f()` in einem Primitiv vom Typ "GL_TRIANGLES" gibt die erste Ecke des ersten Dreiecks an, der zweite Aufruf die zweite, der dritte die dritte Ecke. Der vierte Aufruf dann den ersten Punkt des zweiten Dreiecks usw.

glEnd()

`glEnd()` ist das Gegenstück zu `glBegin()` und beendet ein OpenGL Primitiv.

glClear(c Variablenname z.B. GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

`glClear()` initialisiert bestimmte Teile des Framebuffers. über die angegebene Konstante wird definiert, welcher Teil initialisiert wird, z.B. setzt `GL_COLOR_BUFFER_BIT` die Farbwerte des Puffers auf die mit `glClearColor()` gesetzten Werte.

gluLookAt(Augenposition: r X, r Y, r Z, Blickzentrum: r X, r Y, r Z, Hoch-Vektor: r X, r Y, r Z)

`gluLookAt()` erzeugt eine Transformationsmatrix, die eine Kamera mit perspektivischer Sicht auf die Szene simuliert. Die angegebenen Eigenschaften definieren die Kameraposition, wohin sie sieht und einen Vektor, der nach oben zeigt, also meist in Richtung der Y-Achse: (0,1,0);

glutSwapBuffers()

Wurde das aktuelle Fenster mit zwei Framebuffern ausgestattet, d.h. es wurde `glutInitDisplayMode()` mit `GLUT_DOUBLE` aufgerufen, so rendert man grundsätzlich in den gerade nicht angezeigten Framebuffer. Ist das Bild fertig gerendert, so wechselt `glutSwapBuffers()` zwischen den beiden Framebuffern, das neue Bild wird angezeigt. Dieses Vorgehen verhindert die Anzeige von unvollständigen Bildern während des Renderns, was als Flackern wahrgenommen wird.

glPushMatrix()

`glPushMatrix()` setzt eine leere Matrix auf den aktuellen Matrix-Stack.

glTranslatef(r X, r Y, r Z)

`glTranslatef()` multipliziert eine Translations-Matrix mit der obersten Matrix des aktuell ausgeähnlten Matrix-Stacks.

glScalef(r X, r Y, r Z)

`glScalef()` multipliziert eine Skalierungs-Matrix mit der obersten Matrix des aktuell ausgewählten Matrix-Stacks.

glPopMatrix()

`glPopMatrix()` entfernt die oberste Matrix vom aktuellen Matrix-Stack.

Hier könnte Ihre Werbung stehen!

Interesse?

Email: werbung@foo-magazin.de
Internet: http://www.foo-magazin.de (hier finden Sie die aktuellen Mediadaten)

delphi HR-SYSTEMS GMBH

Wir sind ein international tätiger IT-Dienstleister. Zu unseren Kunden zählen namhafte global agierende Konzerne und renommierte mittelständische Unternehmen. Wir suchen zum nächstmöglichen Termin an unserem Standort in Hamburg kompetente Unterstützung einen

Developer (m/w)

In einem dynamischen Umfeld bieten wir Ihnen interessante Aufgaben und die Perspektive, auch langfristig mit uns zu wachsen. Unterstützen Sie uns bei der regelmäßigen Weiterentwicklung und kundenspezifischen Anpassung unserer eRecruiting-Plattform, um aktuelle Standards und Anforderungen zu implementieren. Dass wir dabei Ihre fachliche wie auch persönliche Entwicklung fördern, ist selbstverständlich.

Ihre Aufgaben:

- Eigenständige Neu- und Weiterentwicklungen im Front- und Backend
- Sicherstellung der Funktionsfähigkeit unserer Applikation auf allen gängigen Browsern (Usability und Performance)
- Entwicklung und Umsetzung neuer Features

Folgende Qualifikationen sollten Sie mitbringen:

- Kenntnisse in der Entwicklung von Web-Applikationen (Perl, PHP, Python, jQuery, AJAX, JSON, ExtJS)
- Gute SQL-Kenntnisse

Das setzen wir voraus:

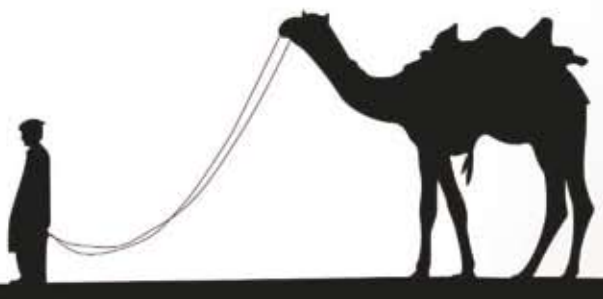
- Selbstständiger, strukturierter und ergebnisorientierter Arbeitsstil
- Team- und Kommunkationsfähigkeit
- Gute Englischkenntnisse

Seit mehr als 20 Jahren vertrauen Konzerne, erfolgreiche mittelständische Unternehmen und renommierte Dienstleister auf unseren professionellen Service. Mit d.vinci entwickeln und vertreiben wir eine der führenden eRecruiting-Lösungen. Überdurchschnittliches Wachstum kennzeichnet unsere Geschäftsentwicklung der letzten Jahre. Dadurch sind wir in der Lage, unseren Mitarbeitern überaus interessante, langfristige Karriereperspektiven zu eröffnen.

Bewerben Sie sich online oder senden Sie uns Ihre kompletten Bewerbungsunterlagen.

delphi HR-Systems GmbH
Nagelsweg 37-39
20097 Hamburg
Tel. (0 40) 37 47 99 -70
E-Mail: dvinci@delphi-mb.de
www.dvinci.de

d.vinci



```
perl -e '$a = qq/Perl is for cowards\n/;
$a=~s/cowards/heroes/;print $a'
```

Renée Bäcker

Rollen für Nicht-Moose-Anwendungen - Role::Basic

In den letzten Jahren wurde `Moose` immer beliebter und damit auch das Tool "Rollen". Wer sich intensiver mit Rollen auseinandergesetzt hat, wird sie vermutlich nicht mehr missen wollen. Bisher war es so, dass das Stichwort "Rollen" im Zusammenhang mit "Moose" zu sehen war. Für die Standard-Objektorientierung in Perl 5 gab es keine Rollen.

Curtis 'Ovid' Poe hat das geändert und das Modul `Role::Basic` geschrieben. Aber erst werden die Ideen hinter den Rollen gezeigt, bevor die Möglichkeiten des Moduls näher gezeigt werden.

Was sind Rollen eigentlich?

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

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

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

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

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

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

Was kann Role::Basic?

Nach diesem Exkurs in die Rollen-Thematik geht es jetzt um die Möglichkeiten von `Role::Basic`. Als Beispiel soll hier eine Klasse genommen werden, die eine Person repräsentiert (Listing 1).

Und für Personen gibt es zwei verschiedene Rollen: Sprecher und Teilnehmer einer Konferenz (Listing 2).

```
package Person;

use strict;
use warnings;

sub new {
    my ($class) = shift;
    my $self = bless {}, $class;
    return $self;
}

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

    $self->{_name_} = $name if @_ == 2;
    return $self->{_name_};
}

1;
```

Listing 1

Durch das Einbinden von `Role::Basic` wird dieses Modul zu einer Rolle, die jetzt in beliebigen Klassen verwendet werden kann. Um diese Rolle jetzt in der Klasse `Person` einzubinden, müssen nur noch folgende Anweisungen eingefügt werden:

```
use Role::Basic 'with';
with 'Speaker';
```

Sobald das in der Klasse steht, können alle Objekte der Klasse auch `speak` ausführen:

```
my $obj = Person->new;
$obj->speak;
```

Das war der einfachste Fall. Wenn mehrere Rollen eingebunden werden sollen, ist darauf zu achten, dass `with` nur einmal aufgerufen werden darf. Das ist ein großer Unterschied zu Moose! Sollen die Rollen `Speaker` und `Attendee` angewendet werden, muss es

```
with 'Speaker', 'Attendee';
```

heißen. Die Reihenfolge in der die Rollen eingebunden werden, spielt keine Rolle.

Im Laufe der Entwicklung verändern sich auch Rollen. Soll eine eingebundene Rolle mindestens in der Version "X.Y" vorliegen, kann das direkt mit angegeben werden:

```
with 'Speaker' => { -version => 'X.Y' };
```

Wenn die Klasse selbst eine Methode `speak` umsetzt, kommt beim Einbinden der Rolle kein Fehler und es "gewinnt" die Methode der Klasse. Dabei wird dem Programmierer auch nicht mitgeteilt, dass die Rolle eine Methode implementiert, die es in der Klasse schon gibt. Dies kann man nur ändern, in-

```
package Speaker;

use strict;
use warnings;
use Role::Basic;

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

    print "blabla";
}

1;
```

Listing 2

dem Umgebungsvariablen gesetzt werden. Es werden diese zwei Variablen ausgewertet:

```
$ENV{PERL_ROLE_OVERRIDE_WARN}
$ENV{PERL_ROLE_OVERRIDE_DIE}
```

Bei der ersten Umgebungsvariablen wird einfach eine Warnung ausgegeben. Dazu einfach die Umgebungsvariable auf einen "wahren" Wert setzen (z.B. in einem `BEGIN`-Block)

```
use Role::Basic 'with';

BEGIN {
    $ENV{PERL_ROLE_OVERRIDE_WARN} = 1;
}

with 'Speaker';
```

Und dann kommt beim Ausführen die Warnung

```
Role 'Speaker' not overriding method \
 'speak' in 'Person'
```

Der Rest vom Programm wird aber weiter ausgeführt. Bei einem `die` kommt die gleiche Meldung, aber das Programm bricht ab.

Wird man so darauf aufmerksam gemacht, dass die Methode der Rolle nicht zum Tragen kommt, kann diese unter einem anderen Namen importiert werden. `Role::Basic` bietet die Möglichkeit, über einen `alias` die Methoden umzubenennen:

```
with 'Speaker' => {
    -alias => { speak => 'give_talk' },
};
```

Damit steht die Methode `speak` der Rolle den Objekten als `give_talk` zur Verfügung:

```
my $obj = Person->new;
$obj->speak;     # Methode der Klasse
$obj->give_talk; # Methode der Rolle
```

Genauso muss auch verfahren werden, wenn zwei verschiedene Rollen gleichnamige Methoden implementieren. Im Gegensatz zu der eben beschriebenen Situation wird man hier als Programmierer aber gleich darauf hingewiesen dass es einen Konflikt gibt:

```
Due to a method name conflict in roles \
  'Attendee' and 'Speaker', the method \
  'speak' must be implemented or excluded \
  by 'Person' (...)
```

So ein Konflikt kann auf mehreren verschiedenen Wegen gelöst werden. Der erste wurde gerade vorgestellt - über `alias`. Für den Fall, dass zweimal der gleiche Zielname beim `alias` angegeben wird

```
with 'Speaker'  => {
        -alias => { speak => 'hallo' }
    },
    'Attendee' => {
        -alias => { pay   => 'hallo' }
    };
```

wirft `Role::Basic` einen Fehler:

```
Cannot alias 'pay' to 'hallo' as a \
  method of that name already exists in \
  Person
```

Zu beachten ist, dass Methoden, die durch `-alias` einen neuen Namen erhalten auch weiterhin unter dem Originalnamen erreichbar ist - wenn diese nicht durch die Klasse oder eine andere Rolle überschrieben wird.

Das heißt, dass bei

```
with 'ObjectLogger' => {
    -alias => { log_object => 'test' }
};
```

sowohl

```
$obj->log_object;
```

als auch

```
$obj->test;
```

funktioniert.

Wer die Methode wirklich umbenennen will, muss `-rename` verwenden, das in einem späteren Abschnitt gezeigt wird.

Wenn in der Rolle eine Methode implementiert wird, die in der Klasse überhaupt nicht benötigt wird, kann diese Methode auch ausgeschlossen werden. Das funktioniert mit `exclude`:

```
with 'ObjectLogger' => {
    -excludes => 'log_object'
};
```

Sollen mehrere Methoden ausgeschlossen werden, kann eine Arrayreferenz angegeben werden.

```
with 'ObjectLogger' => {
    -excludes => [ qw/meth1 meth2/ ],
};
```

Wie oben schon erwähnt, kann zum Umbenennen einer Methode `-rename` benutzt werden. Das ist eine Kombination aus `-exclude` und `-alias`. Ein

```
with 'ObjectLogger' => {
    -rename => { 'log_object' => 'test' }
};
```

entspricht

```
with 'ObjectLogger' => {
    -alias    => {
        'log_object' => 'test'
    },
    -excludes => 'log_object',
};
```

In einigen Fällen muss die Rolle auf Methoden von Klassen zugreifen, um überhaupt funktionieren zu können. Das ist dann der Fall, wenn die Methoden der einzelnen Klasse sehr unterschiedliche aufgebaut sein muss. Zum Beispiel eine Logging-Rolle, die die Objekte als JSON ausgeben. Dann sieht die Rolle so aus:

```
package ObjectLogger;

use Role::Basic;
use JSON;

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

    warn JSON->new->encode(
        $self->as_perl
    );
}

1;
```

Das Objekt muss als Perl-Datenstruktur vorliegen, deshalb wird `$self->as_perl` aufgerufen. Die Rolle kann aber nicht wissen, wie das Objekt aussieht und welche Informationen in der Datenstruktur gespeichert werden sollen. Deshalb kann die `as_perl`-Methode nicht in der Rolle implementiert werden.

Die Rolle kann aber nicht auf Gut Glück darauf hoffen, dass die Klasse wirklich so eine Methode zur Verfügung stellt. Damit der Programmierer darauf aufmerksam gemacht wird, dass er eine `as_perl`-Methode schreiben muss, sollte in der Rolle mit `requires` gearbeitet werden:

```
requires 'as_perl';
```

Wird die Methode in der Klasse vergessen, bricht das Programm gleich ab:

```
'ObjectLogger' requires the method \
  'as_perl' to be implemented by 'MyClass'
```

Natürlich können mehrere Pflichtmethoden angegeben werden:

```
requires 'as_perl', 'test';
```

Ob ein Objekt das Verhalten einer bestimmten Rolle hat, lässt sich mit der Methode `DOES` herausfinden. `Role::Basic` benutzt dabei aber nicht die Methode, die ab Perl 5.10 im Kern enthalten ist, sondern führt einen eigenen Hash mit der entsprechenden Information.

Wenn die Klasse eine Rolle mit `with` einbindet

```
package MyClass;

use Role::Basic 'with';
with 'Rolle';
```

dann liefert ein Objekt der Klasse `MyClass` bei

```
my $has_role = $object->DOES( 'Rolle' );
```

einen wahren Wert.

... im Vergleich zu Moose

Weiter oben in diesem Artikel wurde schon erwähnt, dass sich dieses Modul an den Rollen bei `Moose` orientiert - es soll ein Subset eben dieser Rollen sein. Damit möchte Curtis 'Ovid' Poe sicherstellen, dass ein Umstieg auf Moose sehr leicht fällt. An dieser Stelle werden `Role::Basic` und die `Moose`-Rollen verglichen.

Rollen und Objekte

Die Rollen mit `Role::Basic` können nicht auf einzelne Objekte angewendet werden. Das ist bei `Moose` sehr praktisch, wenn ein Objekt während der Laufzeit neues Verhalten "erlernt" - z.B. vom einfachen Teilnehmer zum Vortragenden wird (siehe `Moose`-Artikel in der letzten Ausgabe). Aber zum Großteil werden Rollen fest in der Klasse eingebunden - und das beherrscht `Role::Basic`.

Methoden-Modifier

`Role::Basic` bietet auch keine Methoden-Modifier. Hier stehen nur normale Perl Subroutinen zur Verfügung. Wer das Verhalten von Methoden-Modifiern nachbilden möchte, muss sich noch das Modul `Hook::LexWrap` anschauen. Allerdings wird es nicht ganz einfach sein, wenn die gleiche Reihenfolge wie bei Moose eingehalten werden soll.

In den Quellen zu dieser Ausgabe ist auch das Paket `Role::Basic::MethodModifier` zu finden, mit dem einfache Methoden-Modifier (`before` und `after`) möglich sind.

Damit ist es dann auch möglich in der Rolle die Modifier wie bei `Moose` zu verwenden:

```
use Role::Basic;
use Role::Basic::MethodModifier;

before 'test' => sub { print 'hallo' };
after 'test' => sub { print "end\n\n" };
```

Zusätzlich zu `Role::Basic` muss in der Rolle noch `Role::Basic::MethodModifier` geladen werden. Sonst gibt es nichts zu beachten. Wird die Rolle in einer Klasse verwendet, schaut das Modul mit `can` nach, ob die Klasse eine Methode kennt die modifiziert werden soll. Und wenn das der Fall ist, wird diese Methode mit `wrap` modifiziert.

Das Ganze funktioniert aber nur für eine Ebene. Wird die Rolle in einer anderen Rolle eingebunden, dann haben die Methoden-Modifier nur auf diese Rolle einen Einfluss und nicht auf die Klassen, die diese zweite Rolle einbinden.

Ein `before` lässt sich relativ leicht auch mit dem Modul `Sub::Prepend` umsetzen.

(Dynamisches) Einbinden von Rollen

Bei `Role::Basic` ist der Aufruf von `with` nur ein einziges Mal pro Klasse erlaubt. Wenn es mehrfach aufgerufen wird, bricht das Programm ab. Durch dieses Verhalten ist es leider nicht möglich, Rollen dynamisch zu laden.

Mehr Funktionalität in Role::Basic

Es gibt auch den Fall, dass `Role::Basic` mehr kann als die `Moose`-Rollen. Dazu zählen die Möglichkeiten von `allow` und `rename`. Der Einsatz dieser Funktionalitäten bedeutet aber gleichzeitig eine Reduktion der Kompatibilität zu den Rollen bei `Moose`.

Fazit

Wie dieser Artikel gezeigt hat, können mit `Role::Basic` Rollen auch in Standard-Perl5-OO eingesetzt werden und so zu flacheren Vererbungshierarchien führen ohne auf Funktionalität zu verzichten. Moose-Rollen sind an einigen Stellen zwar mächtiger, aber wenn Moose aus irgendwelchen Gründen nicht eingesetzt werden kann, ist `Role::Basic` eine gute Alternative.

Deutscher Perl® Workshop 2012
in Erlangen
5. bis 7. März 2012

Schwerpunkte:
modernes Perl,
Einsteiger,
praktische Workshops

Call for Papers und Anmeldung

http://conferences.yapceurope.org/gpw2012

Veranstalter:
Kommunikationsnetz Franken
http://www.franken.de/gpw2012

KNF

Herbert Breunung

WxPerl Tutorial - Teil 8: Reiche Applikationen

Während es im vorigen Teil um das Auslassen von Fehlerquellen ging, dreht sich dieses mal alles um das Hinzufügen von Bequemlichkeit und Auswahl. Denn TIMTOWTDI bedeutet auch: Es gibt mehr als einen Weg der Ein- und Ausgabe. Deshalb werden Zwischenablage, Drag'n Drop, Drucken und weitere Themen beschrieben. Wie versprochen bietet der neunte Teil komplexe Widgets, der zehnte alternative Layoutmöglichkeiten. Der letzte Teil wird Wege aufzeigen eigene, nie zuvor gesehene Widgets zu bauen.

Meister der Tasten

Manche Menschen schätzen es sehr, alles von der Tastatur aus steuern zu können. Deshalb mögen sie oftmals gar keine graphischen Oberflächen, doch es lässt sich jede Oberfläche auch tastaturfreundlich gestalten.

```
$widget2->MoveBeforeInTabOrder($widget1);
$widget2->MoveAfterInTabOrder($widget3);
```

So wenig Aufwand genügt, um sinnvoll mit der *Tab*-Taste oder *Shift+Tab* zwischen den Widgets wechseln zu können. Bei Texteingaben sollte darauf geachtet werden, dass der Stil kein `wxTE_PROCESS_TAB` enthält. Denn dann muss das Drücken von *Tab* über `EVT_KEY_DOWN` abgefangen werden. Wenn die Navigation ausgefeilter wird, sollte das ohnehin geschehen (wie im sechsten Teil gezeigt) und mit

```
Wx::Window::SetFocus( $widget )
```

wird die Aufmerksamkeit der Applikation versetzt. Um den sichtbaren Bereich eines Widgets zu verschieben, helfen die Methoden `PageUp`, `PageDown` oder

```
$widget->SetScrollPos (
     wxHORIZONTAL,
     # oder wxVERTICAL,
     $pixel,
     # jetzt neu zeichnen?
     $neu_zeichnen
);
```

um den sichtbaren Bereich pixelgenau zu verschieben. Bei Listen, Reiterleisten und Ähnlichem ist es meist `SetSelection`, welches den Fokus innerhalb eines Widget setzt.

Und & Davor

Auch Menüs lassen sich mit wenig Aufwand für Tastatursteuerung aufrüsten. Wenn bei der Definition der Einträge ein '&' vor einen Buchstaben gesetzt wird ("Datei &Speichern"), simuliert das Drücken dieses Buchstaben einen Klick auf diesen Eintrag. Zuvor muss man jedoch den Fokus ins Menü rücken, was unter Linux mit *F10* geht, unter Win mit *Alt* und mit *Ctrl+F2* am Mac. Falls ein Buchstabe in einem Menü zweimal hervorgehoben ist, wird lediglich der erste Eintrag angewählt, beim zweiten Drücken der zweite, beim dritten mal wieder der erste und so weiter. Dann hat man *Enter* zu drücken, wenn der Gewünschte ausgewählt ist.

Was viele nicht wissen: dieser Mechanismus geht bei allen Widgets mit Label. Auch Knöpfe werden so per *Alt*+Buchstabe betätigt. Unter Windows werden die Buchstaben im Label allerdings erst unterstrichen, wenn *Alt* gedrückt ist.

Accelerator Table

Die Ansteuerung des Menüs per Taste mag einigen sinnlos erscheinen, da Funktionen auch über die dort angezeigten Kombinationen auslösbar sind. Aber ein Menü dient auch als Orientierungshilfe, nicht jede Funktion ist wichtig genug für eine Tastenkombination und es gibt damit noch weitere Schwierigkeiten. Im sechsten Teil wurde auch gezeigt, dass Menüeinträge, die wie folgt erstellt werden:

```
$menu_item = Wx::MenuItem->new(
    $menu, $open_ID,
    "File Open\tCtrl+O"
);
```

automatisch einen Eintrag in der *Accelerator Table* hinterlassen. Damit löst die Tastaturkombination *Strg+O* innerhalb des gesamten Programms immer aus, was mit

```
EVT_MENU( $frame, $open_ID, \&tu_was );
```

definiert wurde. Und genau das ist das Problem. Zwar lassen sich die Kombinationen nachträglich ändern:

```
$menu_item->SetText("&New\tCtrl+N");
```

Das ist sehr wichtig weil ein

```
$frame->SetAcceleratorTable
    ( Wx::AcceleratorTable->new() );
```

keine Einträge löscht (lokale Tabellen am Widget können so gelöscht werden). Aber was ist, wenn eine Taste in verschiedenen Teilen des Programms unterschiedliche Dinge auslösen soll? Meist geht man dem Problem aus dem Weg und erwähnt solche Kombinationen nur in der Hilfe. Aber im Menü schlägt es sich oft einfacher und schneller nach. Also erwähnen wir die Tasten, ohne das Wx es merkt. Leider werden bei den meisten möglichen Kniffen die Tasten nicht unter Linux angezeigt. Wenn man aber das Tabzeichen ("\t") gegen Leerzeichen austauscht und einen fehlenden Buchstaben gegen 2 Leerzeichen ersetzt, (was manchmal zu einigen Millimeter Abweichungen führt da Menüs selten Proportionalschriftarten benutzen), wird der *Keycode* angezeigt, ohne das ihn Wx als solchen wahrnimmt. Das empfiehlt sich auch bei alternativen *Emacs*-artigen Tastaturkombinationen, die man auch im Menü anzeigen möchte.

Um Tastaturkombinationen trotzdem in Menü-ID's umzusetzen, kann man die *Accelerator*-Tabelle von Hand setzen. Bitte dabei beachten: das `ord 'o'` dient nur der Lesbarkeit, eine *111* hätte es ebenfalls getan. Obwohl die *Keycodes* bei der Abfrage des Events immer dem *ASCII*-Wert des Großbuchstaben entsprechen, funktioniert der folgende Code:

```
$widget->SetAcceleratorTable (
    Wx::AcceleratorTable->new (
        [wxACCEL_CTRL, ord 'o',
            $ID_FILE_OPEN],
));
```

So lässt sich noch an anderer Stelle mit

```
...
[wxACCEL_CTRL, ord 'o',
  $ID_SOCKET_OPEN],
```

die gleiche Tastaturkombination auf eine andere ID mappen (legen), die dann beide zentral abgefangen werden können. Man kann sie aber auch dezentral verarbeiten, weil bei dieser Programmierweise das Menü-Event am Widget ausgelöst wird (und nicht wie beim Hauptmenü nur am Hauptfenster) und danach schrittweise zum jeweiligen Elter hochgereicht wird. So können etwa an einem `Wx::Notebook` die Menü-Events aller Reiter-Flächen per `EVT_MENU` oder `EVT_TOOL` (beide Befehle sind austauschbar) abgegriffen werden. Durch einen `$event->Skip`-Befehl beim Abfangen eines Ereignisses wird das Ereignis weiterhin nach oben weitergereicht.

Zwischenablage

Solange man sich nur in einfachen Textfeldern wie in:

```
my $textfeld = Wx::TextCtrl->new (
    $frame, -1, $inhalt,
     [-1, -1], [-1, -1],
     # sind einzeilig per default!
    wxTE_MULTILINE
);
```

bewegt, ist das Kopieren und Einfügen per Knopfdruck eingebaut (*Strg+C* und *Strg+V*). Wenn man die `Wx::StyledTextCtr` mit Syntaxhervorhebung und vielem mehr haben möchte (wird in Folge 9 besprochen), darf man beim Beenden des Programms

```
wxTheClipboard->Flush;
```

nicht vergessen, oder der kopierte Text, den man woanders einfügen wollte, ist in den ewigen RAM des großen Computers übergetreten. Unüblicherweise ist `wxTheClipboard`

eine Konstante (es ist ein Hilfskonstrukt, dass nur WxPerl kennt), sie muss also wie die Anderen per

```
use Wx qw/wxTheClipboard/;
```

oder `/:everything/` angemeldet werden. Und selbst dann bekommt man noch eine schöne Fehlermeldung, weil alles was Drag 'n Drop und die Zwischenablage betrifft, im Modul `Wx::DND` liegt. Linuxbenutzern nützt aber auch das nicht, da durch einen wxGTK-Bug bis mindestens Wx-Version 2.8.n (abzufragen über die Konstante `wxVERSION_STRING`) `Flush` außer Betrieb ist. Die FreeDesktop-Initiative änderte die Spezifikation für die Zwischenablage um das Problem zu lösen.

Wie sähe es aus, wenn wir mit eigenem Code das Kopieren des Markierten Textes in die Zwischenablage simulieren?

```
use Wx 'wxTheClipboard';
use Wx::DND;
...
EVT_KEY_DOWN($textfeld, sub {
    my ($textfeld, $event) = @_;
    my $key = $event->GetKeyCode;
    # Strg + C gedrückt
    if ( $key == ord('C') and
      $event->ControlDown) {
        wxTheClipboard->Open;
        wxTheClipboard->SetData (
            Wx::TextDataObject->new (
               $textfeld->GetStringSelection
        ) );
        wxTheClipboard->Close;
        wxTheClipboard->Flush;
    }
    elseif(...) {...}
    else { $event->Skip }
});
```

das entsprechende Gegenstück zum Auslesen der Zwischenablage wäre:

```
wxTheClipboard->Open;
if (wxTheClipboard->IsSupported(
 wxDF_TEXT )) {
    my $data = Wx::TextDataObject->new;
    my $ok = wxTheClipboard->GetData($data);
    if ($ok) {
        $text = $data->GetText;
    }
    else {
        warn "Zwischenablage war leer!";
    }
}
wxTheClipboard->Close;
```

Um Graphiken abzugreifen, braucht es entsprechend die Konstante `wxDF_BITMAP` und das Objekt `BitmapDataObject` mit der Methode `GetBitmap`.

Neben dem `Wx::TextDataObject` gibt es auch Spezialisten für URLs und Dateinamen. Die ließen sich auch als Text ablegen, aber es kann sehr praktisch werden, mehrere Daten parallel in der Zwischenablage zu haben. Gerade wenn in Gimp eine *Bitmap* kopiert wird, möchte der Nutzer sicher nicht den vorher zwischengelagerten Text verlieren. Hier ist froh, wer das `Wx::DataObjectComposite` kennt.

```
my $data = Wx::DataObjectComposite->new;
$data->Add (
    Wx::TextDataObject->new( 'lirum ...' ), 1
);
my $bitmap = Wx::Bitmap->new
    ('file-open.xpm', wxBITMAP_TYPE_ANY);
$data->Add( Wx::BitmapDataObject->new(
              $bitmap ), 0 );
wxTheClipboard->SetData( $data );
```

Die 1 am Ende des einen `Add`-Befehls markiert lediglich den bevorzugten Datensatz. Und dennoch fehlt noch etwas. Wie wär es, wenn sich beliebige Datenstrukturen durch die Zwischenablage schleusen ließen? Mit dieser kleine Klasse:

```
package PlDataObject;
use base qw(Wx::PlDataObjectSimple);
use Storable qw(freeze thaw);

sub new {
    my( $class, $data ) = @_;
    my $self = $class->SUPER::new(
     Wx::DataFormat->newUser(__PACKAGE__) );
    $self->{Data} = $data;
    return $self;
}
sub SetData {
    my( $self, $data ) = @_;
    $self->{Data} = thaw $data ;
    return 1;
}
sub GetDataHere {
    my ($self) = @_;
    return freeze $self->{Data}
      if ref $self->{Data};
}
sub GetPerlData { $_[0]->{Data} }
```

geht es fast in dem gewohnten Gang.

```
my $buchautoren = {
    Damian => 'PBP',
    MJD => 'HOP',
};
wxTheClipboard->Open;
wxTheClipboard->SetData(
    PlDataObject->new (
        $buchautoren
    )
);
wxTheClipboard->Close;
```

Und auf der anderen Seite holt man die Daten mit:

```
my $data = PlDataObject->new();
if( wxTheClipboard->IsSupported(
  $data->GetFormat ) ) {
    my $ok = wxTheClipboard->GetData($data);
    if( $ok ) {
        my $PerlData = $data->GetPerlData();
        ...
```

DND

Was die Zwischenablage für die Tastatur ist, ist das Drag' n Drop für die Maus. Auch hier können die verschiedensten Inhalte übertragen werden, optisch und haptisch nachvollziehbar, allerdings nur nacheinander. Mit der Zwischenablage ließen sich 10 Datensätze durch kopieren akkumulieren (zusammenführen) und auf einmal einfügen.

Solange man auch hier nur die `Wx::TextCtrl` benutzt, läuft alles wie gewünscht. Soll es einmal die `Wx::RichTextCtrl` oder `Wx::StyledTextCtrl` sein, oder muss mehr als einfacher Text verschoden werden, wird wieder einmal sichtbar, dass Wx in C++ geschrieben ist. Es muss eine eigene Klasse geschrieben werden, welche das Loslassen und eine die das Auffangen der Daten handhabt. Unter Windows lässt sich ja noch das Ziehen von Dateien mit

```
$widget->DragAcceptFiles(1);
```

aktivieren, aber für Lösungen die überall gleichermaßen funktionieren braucht es ein paar Zeilen mehr. Als erstes muss bekannt gegeben werden, dass unsere Klasse `DateiZiel` für das Auffangen sämtlicher Daten an diesem Widget zuständig ist.

```
$widget->SetDropTarget
    ( DateiZiel->new($widget) );
```

Es klingt umständlich `DateiZiel`, nochmal das Widget zu übergeben, aber die Klasse braucht eine Referenz des Widgets, das später reagieren soll, was nicht immer das Widget sein muss, über dem die Datei losgelassen wird, siehe Listing 1.

Wird Text verschoben, wird wie zu erwarten nicht von `Wx::FileDropTarget`, sondern `TextDropTarget` geerbt, dessen Methode `OnDropText` neu zu definieren ist. Dabei ist der vierte Parameter dann der Text, inklusive der Zeilenumbruchszeichen. Da Linux zwischen beidem nicht so gut unterscheidet, empfiehlt es sich immer noch einmal zu prüfen, ob $data bei `OnDropText` nicht mit `'file:'` beginnt.

```
package DateiZiel;
use base 'Wx::FileDropTarget';
use File::Slurp;

sub new {
    my $class = shift;
    my $self = $class->SUPER::new();
    $self->{WIDGET} = shift;
    return $self;
}

sub OnDropFiles {
    my( $self, $x, $y, $files ) = @_;
    for my $file ( @$files ) {
        $self->{WIDGET}->Append(
            read_file( $file );
        );
    }
    return 1;
}
```

Listing 1

Bei Daten beliebiger Art erbt man von `Wx::DropTarget`. In dem Falle empfiehlt es sich auch die Methode `OnDragOver` zu implementieren. Sie sollte den vierten Parameter zurückgeben, um das Interesse des mit `SetDropTarget` gesetzten Widgets an diesen Daten zu bekunden. Soll der Drop geblockt werden, genügt `return wxDragNone;`

Normalerweise ist im `$data`-Parameter eines der vorgestellten `DataObject`. Die werden geprüft und ausgepackt, wie bei der Zwischenablage beschrieben. Spezialisierte Klassen `Wx::FileDropTarget` nehmen einem diese zwei kleinen Schritte ab.

Um selber eine Datenquelle fürs DND zu erzeugen, muss auch ein extra Objekt generiert werden. Dabei leitet man vom Widget ab, welches die Quelle darstellt. Die zusätzliche Methode `OnDrag` wird aufgerufen, wenn der Nutzer zu ziehen beginnt.

```
sub OnDrag {
    my( $widget, $event ) = @_;
    my $source =
      Wx::DropSource->new ($widget );
    $source->SetData( $data );
    $source->DoDragDrop( 1 );
}
```

Den Cursor setzen

Während des DND sorgt Wx dafür, dass sich die Cursor anzeigen, ob jetzt etwas mitgeschliffen wird und wo es wieder abgelassen werden darf. Aber woher weiß der Nutzer, wo man etwas aufgreifen darf? Darauf kann ein *Label* hinweisen, geht aber auch mit einem Cursor.

```
my $img = Wx::Image->new
    ( 'icon.xpm', wxBITMAP_TYPE_ANY );
$widget->SetCursor (
    Wx::Cursor->new( $img )
);
```

Das verändert den Cursor nur dann, wenn er sich über dem Widget befindet.

Schleppt man allerdings selbstgebaute Daten und setzt den Cursor durch die `OnDrag`-Methode, muss er beim Loslassen auch selbst normalisiert werden.

```
EVT_LEFT_UP( $self, sub {
   $self->SetCursor( wxNullCursor );
);
```

Es gibt auch Standardcursor des Betriebssystems, welche nur per Konstante gesetzt werden.

```
   Wx::Cursor->new( wxHOURGLASS_CURSOR );
```

Trayicon

Das Icon einer Applikation wird denkbar einfach mit `SetIcon` gesetzt, wie im fünften Teil gezeigt. Doch manchmal braucht es auch ein Icon im Tray (Leiste der aktiven, im Hintergrund laufenden Programme), nicht nur am Fenster. Der Bedarf danach sinkt zwar seit *Unity* und *Windows 7*, aber wer mag, ist mit folgendem Code dabei:

```
my $icon = Wx::Icon->new
    ( $icon_pfad, wxBITMAP_TYPE_ANY );
my $tbico = Wx::TaskBarIcon->new();
$tbico->SetIcon( $icon, "Labeltext" );
```

Wie die dazu passenden Kontextmenüs gebaut werden, zeigte der sechste Teil, nur muss hier `EVT_TASKBAR_CLICK` aufgefangen werden und beim Herunterfahren mit `RemoveIcon`, `Destroy` und `undef $tbico` aufgeräumt werden.

Farbdatenbank

Sämtliche Farben in *Wx* werden durch Werte zwischen 0 und 255 mit

```
Wx::Colour->new( $rot, $gruen, $blau );
```

gesetzt. Doch es geht oft einfacher. Mit diesem Befehl kann man viele der üblichen Farbnamen in ein `Wx::Colour`-Objekt übersetzen.

```
my $c = Wx::ColourDatabase::Find( $name );
```

Einige der Farben sind aber nicht immer wie erwartet. So sieht das Orange unter Windows bedenklich nach rot aus, weswegen hier ein finaler Check angebracht ist.

Dennoch ist dieses Modul sehr nützlich, weil die Datenbank erweiterbar ist (`AddColour($name, $farbe)`) und auch eine Rücksuche (`FindName`) erlaubt.

Drucken

Fast alles was man dazu wissen muss, gab es bereits in der letzten Ausgabe. Denn so wie man eine `PaintDC` "bemalt", geht es auch mit einer `PrinterDC`. Lediglich `StartPage`, `EndPage` und `EndDoc` kommen hinzu.

```
use Wx::Print;
my $printout = Wx::Printout->new
    ( $PrinterDC, $title );
my $vorschau = Wx::PrintPreview->new
    ( $preview, $printout );

my $printer = Wx::Printer->new;
$printer->Print( $frame, $printout, 1 );
$printout->Destroy;
```

`$preview` ist eine für die Vorschau präpariertes *Printout*. Und `$vorschau` ist ein *Panel*, welches die Vorschau auf die zu druckenden Seiten anzeigt, mitsamt einer Werkzeugleiste für das Zoomen und Blättern. Es kann auch auf einen `Wx::Dialog` gelegt werden. Wer aber zum Beispiel mehrere Seiten drucken will oder andere Einstellungen des vollen Dialogs braucht, nimmt eine Lösung mit `Wx::PrintDialog`, die im nächsten Teil nachgereicht wird. Auch das hauseigene Logging, das nur eine alternative Ausgabemöglichkeit ist, wird dann vorgestellt.

Rolf Rost

Serialisieren von Datenstrukturen in binäre Sequenzen

In diesem Artikel geht es um die Erzeugung binärer Sequenzen, wobei dafür ein spezieller Algorithmus auf die vorliegende Datenstruktur angewandt wird. Beispiele für Algorithmen zum Serialisieren von Arrays und Hashes werden im Verlauf des Artikels vorgestellt, ohne dabei einen Anspruch auf Vollständigkeit zu erheben.

Der umgekehrte Weg, die Erzeugung einer Datenstruktur aus einer Binärsequenz, wird als Deserialize oder Unserialize bezeichnet und wendet den Serialize-Algorithmus sinngemäß umgekehrt an.

Ziel des Artikels ist es, dem Leser verständlich zu machen, wie sequentielle Binärdateien aufgebaut sein können und wie solche Dateien mit Perl erzeugt und gelesen werden können.

Der Sequenz-Begriff im Sinne des vorliegenden Artikels

Allgemeinhin wird unter einer Sequenz eine Folge von Daten verstanden (Niklaus Wirth). Diese Definition wird im vorliegenden Artikel wie folgt qualifiziert: Eine Sequenz versteht sich als Folge von Bytes. Diese Bytes stellen Rohdaten dar und somit kann die Sequenz auch als Binary bezeichnet werden.

Eine Binary wird sequentiell geschrieben, sequentiell gelesen und ist plattformunabhängig. Sequenzen können in Dateien gespeichert oder auch über Sockets übertragen werden, sofern dafür gesorgt ist, dass der entsprechende Handler (Dateihandle, Sockethandle) die Sequenz nicht verändert.

Rohdaten, Bytes können, aufeinander folgend in einer Sequenz verschiedene Inhalte darstellen, zum Beispiel MP3s, Images oder Text. Darüber hinaus ist es auch möglich, in einer Sequenz verschiedene Content-Types unterzubringen, beispielsweise text/html, image/gif usw. als Multipart-Message, wobei mit einem entsprechenden Serialize/Deserialize-Algorithmus die Sequenz erzeugt und umgekehrt aus der Sequenz die einzelnen Komponenten wiederhergestellt werden.

Der wesentliche Unterschied zwischen einer sequentiellen Binärdatei und einer Textdatei schließlich, besteht darin, dass es in der Binärdatei keine Zeilenumbrüche oder andere Zeichen gibt, die für eine Strukturierung bedeutsam sind. Neben Binärdateien, die als Sequenz geschrieben oder gelesen werden, gibt es noch Binärdateien mit wahlfreiem Zugriff, darauf wird in diesem Artikel nicht weiter eingegangen. Es ist jedoch möglich, über die aus einer Sequenz erzeugte Datenstruktur einen Quasi-Random-Access auf die Datei zu bekommen, was mit tie() erreicht werden kann, wie später noch gezeigt wird.

Basisfunktionen in Perl

Während die Perl-Funktion read() leicht verständlich ist, wird im Folgenden auf die Funktionen pack() und unpack() etwas näher eingegangen. Zu read() sei nur soviel gesagt, dass es nicht die Umkehrfunktion zu write() ist, read() bekommt als ersten Parameter ein Handle und liest aus diesem eine bestimmte Anzahl an Bytes (dritter Parameter) auf einen Puffer, der als zweiter Parameter übergeben wird:

```
read($handle, $buffer, $length);
```

Die Perl-Funktionen pack() und unpack()

Die Funktion pack() erzeugt aus den Argumenten der Liste eine Bytefolge unter Anwendung der gegebenen Schablone,

so ist es in der Dokumentation zu lesen. Beispiele mit einem Argument in der Liste:

```
my $bytes = pack("N", 99); # erzeugt 4 Bytes
my $bytes = pack("N", 0xFFFF);
# erzeugt ebenfalls 4 Bytes
```

Das Interessante hierbei ist, dass mit Schablone "N" und einem Integer-Wert stets genau vier Bytes erzeugt werden. Dieser Umstand ist in Algorithmen zum Serialisieren und Deserialisieren von größter Bedeutung, weil damit das Wiederherstellen einer Datenstruktur aus einer Sequenz erst möglich wird.

Entsprechend der Anzahl der Argumente in der Liste lässt sich die Schablone erweitern, untenstehendes Beispiel erzeugt aus einer Liste mit drei numerischen Argumenten und der Schablone "NNN" folgerichtig dreimal vier, also 12 aufeinander folgende und zusammenhängende Bytes:

```
# erzeuge 12 Bytes
my $bytes = pack "NNN", 1, 2, 3;
```

Die Schablone "N" steht für "An unsigned long in "network" (big-endian) order", das Gegenstück für "big-endian" heißt "little-endian", wofür es die Schablone "V" gibt. Das "V" wiederum steht für "VAX" (Großrechnerarchitektur) und das folgende Beispiel macht den Unterschied zwischen little und big endian deutlich:

```
printf "%s %s", pack("N", 1464816196),
  pack("V", 1464816196); # WORD DROW
```

Die somit erzeugten Bytes sind einmal als "WORD" und zum anderen als "DROW" lesbar, ersteres ist der big endian. Mit der im Beispiel gewählten Zahl 1464816196 sind die mit pack() erzeugten Bytes als Buchstaben lesbar, das ist jedoch nicht mit jeder Zahl der Fall. Neben den Schablonen "N" und "V" gibt es weitere Schablonen, siehe dazu `perldoc -f pack` und welche Schablone für numerische Werte zum Einsatz kommt, hängt vom Verwendungszweck ab. Wichtig ist es, bei einer Serialisierung und Deserialisierung dieselbe Schablone zu verwenden.

Prinzipiell wurde weiter oben schon ein erster Algorithmus beschrieben, mit welchem eine Liste numerischer Werte unter Anwendung der Perl-Funktion `pack()` in eine Binary serialisiert werden kann. Das nächste Beispiel macht es anschaulicher, es zeigt auch die Wiederherstellung:

```
# Eine Liste mit 100 Ganzzahlen
my @list = (0..99);
# Serialize
my $binary = pack "N*", @list;
# Deserialize
my @res = unpack "N*", $binary;
# Ausgabe zur Kontrolle
print join "\n", @res;
```

Steht ein Dateihandle zur Verfügung, kann die `$binary` in eine Datei geschrieben werden, das Dateihandle ist dazu in den `binmode(HANDLE)` zu schalten. Zum Lesen der Datei ist diese im binmode zu öffnen, dazu jedoch später mehr.

In den nächsten Abschnitten soll die Anwendung der Funktionen `pack()` und `unpack()` auf textliche Inhalte näher erläutert werden. Betrachte untenstehendes Beispiel:

```
print pack "A50" , "Ein bisschen Text ...";
```

Von Vorteil ist es hierzu, einen Editor zu verwenden, der in der Ausgabe auch die Leerzeichen sichtbar macht, kurzum, mit Schablone "A50" wird der Textstring auf eine Länge von 50 Bytes gebracht. Das "A" steht hierbei für "ASCII", es ist jedoch ohne weiteres möglich, auch einen nicht-ASCII-Text (z.B. UTF-8-kodiert) mit Schablone "A" auf eine definierte Länge zu bringen, zu beachten ist dabei lediglich die Tatsache, dass UTF-8-kodierte Zeichen unter Umständen mehr als ein Byte beanspruchen, ein UTF-8-kodiertes 'ä' beansprucht beispielsweise zwei Bytes.

Sind die Zeichen in einem Perl-Script UTF-8-kodiert, würde ein

```
print pack "A", 'ä';
```

nur das erste Byte ausgeben, dieses hat den numerischen Wert C3 (hexadezimal), ein vollständiges UTF-8-kodiertes 'ä' hat jedoch zwei Bytes in der Folge C3A4, in dezimal 228. Ähnlich der Schablone "A" ist die Schablone "Z" zu verwenden, der Unterschied ist der, dass mit Schablone "Z50" zwar auch 50 Bytes erzeugt werden, die fehlenden Bytes jedoch nicht mit Leerzeichen, sondern mit binären Nullen aufgefüllt werden, wobei zu beachten ist, dass das letzte Byte immer für eine binäre Null reserviert ist, der Zeichenkette also verloren ist. Mit der Perl-Funktionen `unpack "A50", $binary` oder `unpack "Z50", $binary` werden die Padding-Bytes wieder entfernt, mit Schablone "A50" auch ein etwaiger Zeilenumbruch am Ende.

Records mit konstanter Länge

Unter Anwendung der Schablonen "N" und "Z" lassen sich Records mit einer fest vorgegebenen Länge erzeugen, siehe Beispiel für einen Record, bestehend aus einer Zahl und einer Zeichenfolge:

```
# Serialize
my $record = pack "NZ50", 99,
              'Ein bischen   Text';
# 54 (Bytes sind es)
print length($record), "\n";
# Deserialize
my ($zahl, $text) = unpack "NZ50", $record;
# 99 Ein bischen Text
printf "%u %s\n", $zahl, $text;
```

Ein Record ist als Komponente einer Sequenz zu betrachten. Zum Erzeugen von Sequenzen mit Records konstanter Länge ist der Algorithmus recht einfach, zu beachten ist jedoch wie bereits weiter oben angemerkt, dass mit der gewählten Schablone kein Datenverlust entsteht. Hier ein einfacher Algorithmus zum Serialisieren einer Liste mit Namen, die sich in der Länge nur geringfügig unterscheiden:

```
use IO::File;
use strict;
use warnings;

my $fh = new IO::File '/tmp/binfile',
 O_RDWR|O_BINARY|O_CREAT|O_TRUNC or die $!;
my @names = qw(
 Hans Otto Anna Horst Manfred);

# Serialize
foreach my $name(@names){
    print $fh pack("Z20", $name);
}

seek $fh, 0, 0;

# Deserialize
while(read $fh, my $record, 20){
    my $name = unpack "Z20", $record;
    print "$name\n";
}

undef $fh;
```

Zum Serialisieren von Hashes kann dieser Algorithmus sinngemäß angewandt werden, so dass Schlüssel und Wert jeweils einen Record bilden. Sequenzen mit Records konstanter Länge genügen jedoch nicht immer allen Anforderungen. Es besteht die Gefahr, dass Zeichenketten abgeschnitten werden wenn die Schablone nicht passt oder die Schablone ist überdimensioniert. Records mit variabler Länge lösen das Problem, dazu werden die einzelnen Längen ermittelt und die Längenangaben sequentiell mit gespeichert. Konstant bleiben müssen dann nur noch die Längenangaben, was jedoch ohne Weiteres machbar ist, da es sich hierbei um ganze Zahlen handelt.

Pragma utf8 und die Perl-Funktion length()

Betrachte untenstehenden Code:

```
print length('ä'),"\n"; # 2
use utf8;
print length('ä'),"\n"; # 1
no utf8;
print length('ä'),"\n"; # 2
```

Hintergrund: In dem hier vorliegenden Dokument liegen alle Zeichen in einer UTF-8-Kodierung vor. Im ersten Aufruf der `length()`-Funktion wird die Länge des Zeichen 'ä' korrekt ausgegeben als zwei Bytes. Nach Aktivieren des Pragmas `utf8` arbeitet `length()` jedoch zeichenorientiert und gibt folgerichtig die Länge des Zeichen 'ä' mit "eins" an. Nach dem Ausschalten des Pragmas mit `no utf8;` arbeitet `length()` wieder byte-orientiert.

Dieser Umstand ist insofern beachtenswert, als dass vor dem Ermitteln der Längenangaben zum Serialisieren in Rohdaten sichergestellt werden muss, dass die `length()`-Funktion erwartungsgemäß funktioniert.

Records mit variabler Länge

Mit dem Verständnis des vorausgegangenen Abschnitts kann obenstehender Algorithmus umgeschrieben werden, so dass die Längenangabe der einzelnen Listenelemente mit gespeichert wird - siehe Listing 1.

Die so erzeugte Datei ist deutlich kleiner, vorher 100 Bytes, nunmehr 44 Bytes. Wesentlich jedoch ist es an diesem Algorithmus, dass dieser unabhängig von der Länge der einzelnen Listenelemente funktioniert und selbstverständlich darf die Liste auch mehr als nur fünf Elemente enthalten. Auch dieser Algorithmus ist zum Serialisieren eines Hash brauchbar, die Längenermittelung erfolgt dann sowohl für den Schlüssel, als auch für den dazugehörigen Wert und es wird paarweise serialisiert.

```
use IO::File;
use strict;
use warnings;

my $fh = new IO::File '/tmp/binfile',
 O_RDWR|O_BINARY|O_CREAT|O_TRUNC or die $!;
my @names = qw(
 Hans Otto Anna Horst Manfred);

# Serialize into binary
foreach my $name(@names){
    print $fh pack("N", length($name)).$name;
}

seek $fh, 0, 0;

# Deserialize
while(read $fh, my $bytes, 4){
    my $len = unpack("N", $bytes);
    read($fh, my $name, $len);
    print "$name\n";
}

undef $fh;
```

Listing 1

Etwas mehr Disziplin bitte

Das Modul `IO::File` exportiert einige Konstanten, mit denen der Modus der zu öffnenden Datei festgelegt werden kann, von besonderem Interesse ist die Konstante `O_BINARY`. Damit wird plattformunabhängig sichergestellt, dass die Datei im Binärmodus geöffnet wird. Es ist möglich, dass die in der Datei enthaltenen Bytes aus der Sicht des Betriebssystems Zeilenumbrüche oder andere Kontrollelemente darstellen. Befindet sich beispielsweise am Ende einer Datei das Byte "0x1A", wird das bereits beim Öffnen der Datei auf einer Win32-Plattform entfernt, weil im Default die Datei als Textdatei betrachtet wird. Die Binary würde unbrauchber werden, mit dem Modus `O_BINARY` wird das verhindert.

```
use IO::File;
use strict;
use warnings;

my $fh = new IO::File '/tmp/binfile',
  O_RDWR|O_BINARY|O_CREAT|O_TRUNC or die $!;
binmode($fh, ":raw");
```

Die Plattformunabhängigkeit wird außerdem mit der Perl-Funktion `binmode()` sichergestellt, wobei die Disziplin ":raw" angewandt wird (Default). Mehr dazu in `perldoc -f binmode`.

Weitere Algorithmen zum Serialisieren

Oftmals liegen Strukturen vor, die als Werte wiederum Hashes enthalten oder auch Arrays. Betrachte untenstehende Datenstruktur:

```
my $ref = {
    foo => {
        color => 'blue',
        align => 'left',
    },
    bar => {
        color => 'green',
        align => 'right',
        css   => '/style.css',
    },
};
```

Es wird ein Hash referenziert, objektorientiert betrachtet, ist die Referenz eine Sammlung der Objekte "foo" und "bar", die selbst bestimmte Attribute haben, jedoch in verschiedener Anzahl und möglicherweise gibt es weitere, bisher unbekannte Attributnamen. Beiden Objekten gemeinsam ist die Tiefe der Struktur, die somit nicht rekursiv durchlaufen werden muss, es reicht eine Iteration. Ein Algorithmus zum Serialisieren könnte wie folgt realisiert werden, das Sequenz-Handle sei SH und alle numerischen Angaben bekommen mit pack()-Schablone "N" genau vier Bytes:

- Durchlaufe jeden Key in einem Schleifenkörper
- Ermittele die Länge des Keys
- Schreibe die Länge des Keys in SH gefolgt von der Bytefolge des Keys
- Ermittle die Anzahl der Attribute zum Key
- Schreibe die Anzahl der Attribute in SH
- Schreibe nun paarweise die Schlüssel und Werte als Längenabgabe plus Schlüssel bzw. Wert

Aus der so erzeugten Binary ist die Datenstruktur durch Anwendung des Algorithmus in umgekehrtem Sinn wiederherstellbar, zuerst werden vier Bytes gelesen um die Länge des ersten Keys zu ermitteln, mit dieser Längenangabe wird danach der Key selbst gelesen. Nach dem Lesen weiterer vier Bytes steht die Anzahl der Attribute fest, zum Deserialisieren derer bietet sich eine `for()`-Schleife an.

Random Access mit der Perl-Funktion tie() erreichen

Aufgrund der Tatsache, dass die meisten Perl-Objekte Hash-Referenzen sind, sei abschließend auf `Tie::Hash` etwas näher eingegangen. Zum Erstellen einer eigenen Klasse "myHashFile" wird das Erbe von `Tie::StdHash` angetreten:

```
package myHashFile;
use strict;
use Tie::Hash;
our @ISA = qw(Tie::StdHash); # Inherit
```

Geerbt werden über das `@ISA`-Array alle Methoden der Basisklasse `Tie::Hash`. Eine mögliche Verwendung der eigenen Klasse:

```
use myHashFile;
# TIEHASH()-Method der Basisklasse
tie %myhash, 'myHashFile', 'hash.bin';
# STORE()-Method der Basisklasse
%myhash = %ENV;
# ruft STORE()
$myhash{foo} = 'bar';
# UNTIE()-Method der Basisklasse
untie %myhash;
# impliziter Aufruf DESTROY(),
# wenn main beendet wird
```

Zum Verständnis sind die Aufrufe der Superklasse-Methoden als Kommentare obenstehend notiert. Bestimmte Methoden müssen nun in der eigenen von `Tie::Hash` abgeleiteten Klasse überschrieben werden, damit die Bindung an eine Datei erzielt wird. Die Methoden, die sich dazu eignen sind `TIEHASH()` und `UNTIE()` oder `DESTROY()`, in Ersterer erfolgt die Deserialisierung, also die Erstellung der Datenstruktur "Hash" aus einer Datei.

Zum Serialisieren des Hash zurück in die Datei gibt es die Möglichkeit, dies entweder in `UNTIE()` oder `DESTROY()` zu machen, das hängt von der konkreten Aufgabenstellung ab. So wird die Methode `UNTIE()` erst aufgerufen, wenn in der main ein `untie(%hash)` erfolgt, die `DESTROY()`-Methode hingegen wird automatisch aufgerufen. Gemäß den Regeln des Perl-Objekt-Modells werden Methoden der Superklasse überschrieben (überlagert), wenn es in der abgeleiteten Klasse Methoden mit gleichem Namen gibt, unter Beachtung der Groß- und Kleinschreibung:

```
package myHashFile;

# Überschreiben der Super-Methode
sub TIEHASH{
    # Deserialize from File
}

# Überschreiben der Methode der Basisklasse
sub UNTIE{
    # Serialize to File
}
```

Weitere Möglichkeiten ergeben sich mit `$obj =tied(%myhash);`, es ist dann möglich in der eigenen Klasse Methoden zu definieren, die über das Objekt `$obj` aufgerufen werden und spezielle Aufgaben erfüllen, beispielsweise eine Datenbankanbindung.

Literaturangaben

Eric Foster Johnson, Perl Module, mitp-Verlag, ISBN 3-8266-0570-5

Niklaus Wirth, Algorithmen und Datenstrukturen mit Modula 2, ISBN 3-519-02260-5

Herbert Breunung

Perl 6 - Der Himmel für Programmierer - Update 4

Wieder ging ein Jahr, und was tat sich um Perl 6? Nicht sehr viel und das ist gut so. Denn wie aufmerksame Leser dieser Reihe wissen, bezeichnet Perl 6 nur die Sprache. Das umfasst lediglich die Spezifikation in ihren 3 bereits vorgestellten Teilen: *Synopsen* (menschlich lesbare Form), *Testsuite* (aka *roast*) und der `STD.pm` (Defintion der Syntaxregeln mit Perl 6-Grammatiken). Die Änderungen der Synopsen werden immer feiner. Im letzen Jahr kamen an neuen Befehlen lediglich hinzu: `roll` (zufällige Auswahl aus einer Liste ohne zurücklegen wie `pick`, beim *roll*en eines Würfels kann ja auch immer jede Zahl erziehlt werden), `categorize` (eine Art `map`, das die Werte einer Liste unter mehreren oder keinem Schlüsseln eines Hashes ablegen kann), `gcd` (größter gemeinsamer Teiler) und `lcm` (kleinstes gemeinsames Vielfaches).

Dass die Syntax sich stabilisiert, kann als deutliches Zeichen gewertet werden, dass man sich dem Ziel nähert. Auch die Diskussionen darüber haben in der *p6l* Mailingliste stark nachgelassen. So haben die Implementatoren eine Chance, ihren notorischen Rückstand aufzuholen.

Die Syntopsen sind umgezogen. Sie können weiterhin auf *http://perlcabal.org/syn/* gelesen werden, aber die Quellen liegen nun in *http://github.com/perl6/specs*. Der fleißige Moritz Lenz, Hüter der Testsuite, löste im Herbst das *Pugs-Repository* auf, in dem sie bisher lagen, und spaltete es in mehrere themenbezogene Projekte auf, die unter dem Dach *http://github.com/perl6* zusammengefasst sind. Alles was sich nicht zuordnen ließ, liegt dort als *Mu* (*most undefined* im Perl 6 - Jargon). Das erhöhte die Sichtbarkeit der Änderungen und erleichtert die Mitarbeit.

Panda

Unter diesem Oberprojekt finden sich auch die Quellen der Seite *Perl6.org*, inklusive der Liste aller installierbarer Perl 6-Module (*modules.perl6.org*). Das dort ca. 45fach sichtbare Abbild eines Pandakopfes zeigt, dass sich fast alle gepflegten Module mit *Panda* installieren lassen. Nach *proto*, *neutro*, *pls* und *ufo* (Makefile-Generator) schrieb tadzik aka Tadeusz Sosnierz jenen Installationshelfer, der mit wenig Code und der gesammelten Erfahrung die drängenden Probleme löste.

Rosetta

Manche der Module sind recht umfangreich, wie etwas das Webrahmenwerk `Web.pm`, dass von *Sinatra* und *Mojolicious* beeinflusst ist und auf der Erfahrung mit der nun eingestellten *November*-Wikisoftware fußt. Kleinere Module, wie etwa `Config::INI` oder `JSON::Tiny`, eignen sich gut um Perl 6 anhand von echtem Code zu lernen. Wer diesen Ansatz bevorzugt, sollte auch *http://rosettacode.org/wiki/Category:Perl_6* kennen. Auf dieser Seite lassen sich Sprachen anhand von Codestücken vergleichen, welche gängige Probleme lösen. Larry Wall, sowie einige Entwickler, nutzen dieses Wiki regelmäßig um zu sehen, wie diese Lösungen in Perl 6 ausfallen und ob sich Perl 6 verbessern lässt.

Niecza

Auch die Reihen der Implementationen haben sich etwas gelichtet. Von den zwei neuen Compilern `YAPSI` (in Perl 6 geschrieben) und *Niecza* möchte ich diesmal den zweiten hervorheben, weil er neben Rakudo der derzeit wichtigste ist. Er verdeutlicht den von *Perlito* und *Sprixel* angefangenen Trend zu mehr Geschwindigkeit. Das beinahe pünktlich erschienene *Rakudo* * (einen Monat nach dem letzten Artikel, Sommer 2010), konnte viel, war recht stabil, ließ aber während der Ausführung gerne auf sich warten. Das gefiel unter anderem Stefan O'Rear überhaupt nicht, worauf er im September begann, einen besonders schnellen Compiler zu schreiben. Der Name ist ein eigenwilliges Kondensat der Wendung "habe keine Zeit" auf Slowakisch. (Derzeit wohnte ein Haupttentwickler von Rakudo in Bratislava.) Um schnell zu sein, wählte O'Rear *.Net* beziehungsweise *Mono* als Plattform, was sich bewährte. Ein Programm zur Berechnung eines Apfelmännchens benötigte, verglichen mit Rakudo, nur ein Fünfzig- bis Neunzigstel der Zeit. Da sorear danach sucht, die interne Struktur von Rakudo zu imitieren, um leichter Funktionalität zu übernehmen, wird sein Projekt damit auch für Rakudo wertvoller, als Testumgebung für Optimierungen und eine Portierung.

Rakudo

Geschwindigkeit aber auch mehr Präzision sind derzeit Rakudos oberste Ziele. Während der letztens vorgestellte *ng*-Zweig vor allem die Regex-Engine für komplexere Schverhalte verbesserte (*Protoregexes*), aber *Rakudo* etwas verlangsamte, ist mit dem neuen *nom*-Zweig eine deutliche Beschleunigung bereits jetzt absehbar. Trotz genauerer Einhaltung von Details, ist diese Entwicklung 3- bis 14 mal schneller als ihr Vorgänger (*ng*).

Das hat seine Ursache in einem vollständig überarbeiteten Meta-Objekt-Modell, welches erlaubt, einmal gewonnene Informationen an den "richtigen" Stellen zu speichern, um sie später für Optimierungen anzuwenden. Da Perl 6 intern sowieso streng in Objekten organisiert ist, war die Verwendung eines solchen Modells zwingend.

Es wird derzeit *6model* genannt und baut auf den Erfahrungen von *SMOP* und `Class::MOP` auf, dem Herzen von Moose. Entwickelt hat das *6model* Jonathan Worthington (*http://6guts.wordpress.com*) mit Unterstützung einer Spende der TPF. Derzeit schreibt er darauf aufbauend zusammen mit Patrick Michaud und anderen *nom*, was für "new object model" steht oder einfach für "essen" in Babysprache.
Natürlich hätte man das *6model* eher entwerfen können, aber bisher verließen sich die *Rakudo*-Autoren auf *Parrot*, dessen Objektmodell immer noch nicht die nötigen Funktionen und Informationen bereitstellen kann, um damit einen Perl 6-Compiler zu schreiben. Mit *nom* wird *Rakudo* wieder ein Stück unabhängiger von *Parrot* (wie schon zuvor mit *NQP-rx*) und lässt eine Portierung auf andere VM mittelfristig realistisch erscheinen.

Die Mini-Sprache *NQP*, in der bald 100 Prozent von Rakudo geschrieben sind, weil die letzen Reste *C* und *PIR* nach *NQP* ausgelagert werden, macht es leichter Rakudo auf andere Platformen zu setzen. Das wäre ein großer Vorteil zu Python oder Ruby, dass bis jetzt nicht voll kompatibel zu JRuby ist.

Parrot

Ebenfalls im Parrot-Lager ist man sich bewusst, dass die Geschwindigkeit die derzeit wichtigste Hürde ist. Nach 2 *Garbage-Kollektoren*, die nach dem Prinzip *mark and sweap* arbeiten, schrieb man in 2 Jahren den Dritten (*gms* genannt), welcher die Variablen in Generationen aufteilt und sie nach diesem Schema abarbeitet.Das stellte sich nicht nur als zuverlässiger heraus, sondern auch um rund 30% schneller und drastisch genügsamer an RAM (auf ca. 10%). Genaue Daten finden sich unter *http://github.com/pmichaud/rpbench-results*.

Bei *Parrot* führte das Objektmodell nicht die Tagesordnung, was wohl auch Ursache für *6model* war. Das derzeit größte Problem, nach Ansicht der *Parrot*-Entwickler, ist der bei der Ausführung ständige Wechsel zwischen Ebenen, die in unterschiedlichen Sprachen geschrieben wurden. Ein minimaler Kern, auf dem der Rest aufbaut, soll erhebliches Optimierungspotential haben. Ein "finaler" Prototyp wurde bereits in Perl 5 implementiert. Da der nur wenige (<40)

simple Befehle umfasst und keinerlei "Magie", wurde er *M0* getauft. Ebenen "darüber", können damit auch M1, M2, .. und so weiter getauft werden. *Lorito* bezeichnet die Umfassende Neugestaltung, *Parrot* auf diese neue Basis zu stellen, sowie einige weitere, längst fällige Renovierungen durchzuführen. Dass man dabei das Objektsystem mit einem Henkel versehen und gegen *6model* austauschen sollte, ging den Mannen um Christoph Otto erst kürzlich auf, wird aber gerade Teil der Lorito-Spezifikation. Die Konsequenz wäre eine weitere Beschleunigung für Rakudo und es würde auch die *NQP*-Implementation auf *Parrot* vereinfachen.
Lorito wäre ein Grund für die Autoren der auf *Parrot* aufbauenden Compiler, Rosella (*http://whiteknight.github.com/Rosella*) zu nutzen. Diese Werkzeugkiste, die das Testen und andere tägliche Probleme erleichtert, bietet eine API die auch nach der Umstellung voll gültig sein sollte.

Wie eben angedeutet, hat die Parrot-Foundation 2010 ihre Führungsriege fast vollständig ausgewechselt. Der Presse ließ sich entnehmen, dass im August Allison Randal technische Architektin von Ubuntu wurde. Als Programmiererin bleibt sie der VM glücklicherweise erhalten. Will Coleda, Jeff Horwitz, cromatic und Patrick Michaud hatten weniger wohlklingende Gründe, aber insgesamt ist es Zeichen für ein gesundes Projekt, das eine "neue" Generation in die Leitung nachrücken kann.

Zum Abschluss möchte ich daran erinnern, dass es 1994 Stimmen gab, die meinten: "Perl 5 wird niemals jemand benutzen", weil es "viel zu langsam" sei. Perl 6 ist allerdings nochmals um eine Größenordnung komplexer, weswegen jede helfende Hand gebraucht wird.

„Eine Investition in Wissen bringt noch immer die besten Zinsen.“

(Benjamin Franklin, 1706-1790)

Aktuelle Schulungsthemen für Software-Entwickler sind: AJAX - interaktives Web * Apache * C * Grails * Groovy * Java agile Entwicklung * Java Programmierung * Java Web App Security * JavaScript * LAMP * OSGi * Perl * PHP – Sicherheit * PHP5 * Python * R - statistische Analysen * Ruby Programmierung * Shell Programmierung * SQL * Struts * Tomcat * UML/Objektorientierung * XML. Daneben gibt es die vielen Schulungen für Administratoren, für die wir seit 10 Jahren bekannt sind.

Siehe linuxhotel.de

Verknüpfte Objekte in OTRS

Eintausendundeine Anfrage zu demselben Thema oder zumindest ähnlich gelagerte Fälle. Je mehr man mit einem Massengeschäft zu tun hat, umso eher wird einem genau das begegnen. Es ist dann natürlich sinnvoll, solche Anfragen zu verknüpfen. So kann man sich leicht durch die Anfragen "navigieren" und Lösungen von Kollegen finden.

OTRS bietet aber nicht nur die Möglichkeit, Anfragen zu verknüpfen, sondern beliebige Objekte miteinander zu verbinden. Das ist praktisch wenn die ITSM-Erweiterung im Einsatz ist und man Anfragen mit Configuration Items verknüpfen kann. So kann man schnell die notwendigen Zusatzinformationen für das Ticket finden.

In bestimmten Fällen reichen die eingebauten Verknüpfungsmöglichkeiten nicht aus. In diesem Artikel wird erst gezeigt, wie neue Verknüpfungsmöglichkeiten für existierende Objekte erschaffen werden können und danach wird darauf eingegangen, wie neue Objekte und damit neue Verknüpfungen eingeführt werden.

Zwischen Tickets können standardmäßig "Normale" Verbindungen geschaffen werden, damit werden gleichwertige Tickets verbunden. Als zweite Möglichkeit gibt es die Eltern-Kind-Beziehung. Damit schafft man gewisse Hierarchien und es gibt die Möglichkeit, dass im OTRS das Elternticket erst dann geschlossen werden kann, wenn alle Kindtickets geschlossen sind.

Jetzt soll die Verknüpfung "Massenstörung Eltern-Kind" eingeführt werden. Wenn z.B. der Mailserver ausfällt, werden sich in größeren Firmen garantiert einige hundert Benutzer melden und jeweils ein Ticket aufmachen. Es ist aber mühselig, sich um jedes einzelne Ticket zu kümmern. Darum soll es zu dem Vorfall ein Masterticket geben, in dem der Agent alle Informationen zusammenträgt. Die Tickets der Endanwender werden diesem Ticket untergeordnet und wenn das Masterticket geschlossen wird, werden auch gleichzeitig alle untergeordneten Tickets geschlossen und die Anwender benachrichtigt.

Dieses automatische Schließen der Kindtickets wird im Endeffekt über ein Ticket-Event-Modul gelöst. Das wird in einer weiteren Ausgabe von $foo besprochen. Aber damit dieses Schließen getriggert werden kann, muss es diesen neuen Verknüpfungstyp "Massenstörung Eltern-Kind" geben.

Da hier nur bestehende Objekte - nämlich Tickets - miteinander verknüpft werden, ist hier keinerlei Programmierarbeit notwendig. Aber der Vollständigkeit halber wird es hier dennoch gezeigt.

Der neue Verknüpfungstyp wird einfach über die SysConfig hinzugefügt (Listing 1). Möglichkeit 1 ist, es als Datei unter <OTRS_HOME>/Kernel/Config/Files/MassIncident.xml zu speichern und dann als OTRS-Administrator die SysConfig aufzurufen, damit die Änderungen aktiv werden.

Die unüblichere Variante ist es, die Daten direkt in das Modul Kernel::Config innerhalb der Subroutine `Load` zu schreiben:

```
$Self->{'LinkObject::Type'}
   ->{'MassIncident'} =  {
        'SourceName' => 'MassIncident Parent',
        'TargetName' => 'MassIncident Child'
};
$Self->{'LinkObject::PossibleLink'}
   ->{'0600'} =  {
        'Object1' => 'Ticket',
        'Object2' => 'Ticket',
        'Type' => 'MassIncident'
};
```

```
<ConfigItem Name="LinkObject::PossibleLink###0600-Massincident" Required="0" Valid="1">
    <Description Translatable="1">Links 2 tickets with a "MassIncident" link.</Description>
    <Group>Ticket</Group>
    <SubGroup>Core::LinkObject</SubGroup>
    <Setting>
        <Hash>
            <Item Key="Object1">Ticket</Item>
            <Item Key="Object2">Ticket</Item>
            <Item Key="Type">MassIncident</Item>
        </Hash>
    </Setting>
</ConfigItem>
<ConfigItem Name="LinkObject::Type###MassIncident" Required="1" Valid="1">
    <Description Translatable="1">
        Defines the link type 'MassIncident'. If the source name and the target
        name contain the same value, the resulting link is a non-directional one;
        otherwise, the result is a directional link.
    </Description>
    <Group>Ticket</Group>
    <SubGroup>Core::LinkObject</SubGroup>
    <Setting>
        <Hash>
            <Item Key="SourceName">MassIncident Parent</Item>
            <Item Key="TargetName">MassIncident Child</Item>
        </Hash>
    </Setting>
</ConfigItem>
```

Listing 1

Bei dem Konfigurationsparameter `LinkObject::PossibleLink` wird festgelegt, zwischen welchen Objekten diese Verbindung hergestellt werden kann. Der Typ nennt sich "MassIncident". Das ist der Wert, der in der Datenbank auch gespeichert wird. Über dieses "MassIncident" kann dann später identifiziert werden, auf welche Art und Weise die Objekte verbunden sind. Es besteht nämlich die Möglichkeit, verschiedene Verknüpfungen für ein und dasselbe Objekt zu etablieren.

Beide Objekte sind hier "Ticket", d.h. dass es zwischen "Ticket" und "ConfigItem" keine Verknüpfung vom Typ "MassIncident" geben kann.

Der zweite Konfigurationsparameter - `LinkObject::Type` - beschreibt den Verknüpfungstyp "MassIncident" genauer. Hier wird ausgesagt, dass der Ausgangspunkt der Verknüpfung das Elternticket der Massenstörung ist und der Endpunkt das Kindticket.

Wenn man jetzt auf den "Verknüpfung"-Link im Ticket (Abbildung 1) klickt und das Ticket mit einem anderen Ticket verknüpfen will, steht der neue Verlinkungstyp schon zur Verfügung (Abbildung 2).

Verknüpfungen mit neuen Objekten

Jetzt aber zu dem Teil, bei dem Verknüpfungen mit neuen Objekten ermöglicht werden soll. Im OTRS von Perl-Services.de laufen auch Anfragen zu OPAR (OTRS Package ARchive) auf. Häufig sind das Sachen, die auch auf GitHub als Issues auftauchen. In solchen Fällen sollen die Tickets mit einer

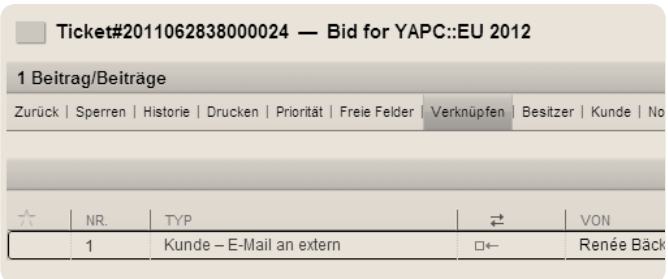


Abbildung 1: Verknüpfen-Link im Ticket-Menü

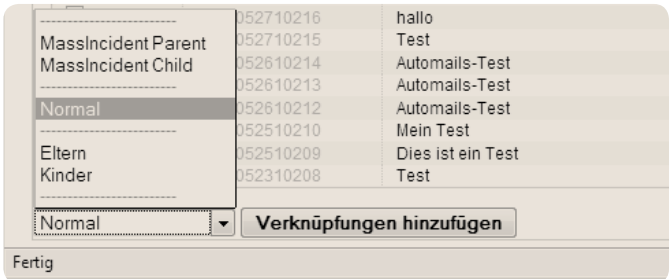


Abbildung 2: Massenstörung als Verknüpfungstyp

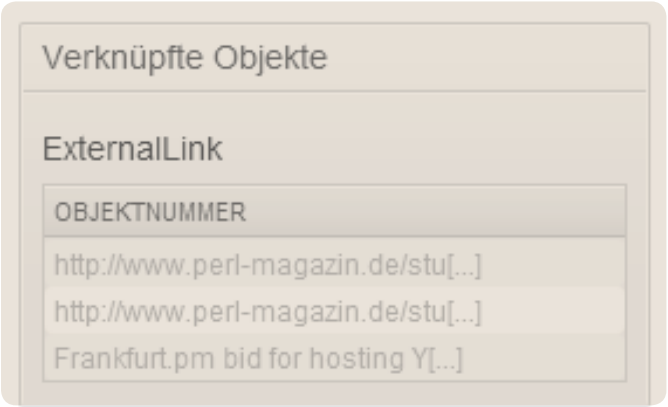


Abbildung 3: Verknüpfte URLs

URL verknüpft werden. Damit erscheint unter dem Ticket in der Tabelle der Link zu dem Issue auf GitHub (Abbildung 3).

Zunächst sollte man wissen, wie das System der verlinkten Objekte in OTRS funktioniert.

Zunächst wird betrachtet, wie es funktioniert, wenn eine Verlinkung erzeugt werden soll. Dazu muss OTRS erstmal wissen, welche Möglichkeiten es überhaupt gibt. Befinde man sich in der Ticketansicht, muss OTRS wissen, welche Verknüpfungstypen gibt es für Tickets. Ist gerade die Ansicht von ConfigItems geöffnet, werden die möglichen Verknüpfungstypen für ConfigItems benötigt.

Das zentrale Modul bei Verlinkungen ist Kernel::System::LinkObject. Das stellt alle notwendigen Funktionen zur Verfügung. Sollen die möglichen Verknüpfungstypen herausgefunden werden, liest das Objekt des LinkObject-Moduls die Konfiguration aus und wertet die Einträge von "LinkObject::PossibleLink" (siehe auch Beispielkonfiguration von oben) aus. Möglich sind die Werte, die bei `Object1` und `Object2` eingetragen sind.

Anschließend wird noch geprüft, ob ein entsprechendes Backendmodul in Kernel/System/LinkObject/ liegt, bei der Beispielkonfiguration also Kernel/System/LinkObject/Ticket.pm

Auch die Anzeige der Verlinkungen wird über das zentrale Modul Kernel::System::LinkObject gemacht. Über das Objekt des LinkObject-Moduls werden die Daten aus der Datenbank geholt und ein Hash mit allen möglichen Informationen erstellt. Daraus lassen sich die Informationen darüber welcher Objekttyp das Ziel ist, ob das Ticket Quelle oder Ziel ist und welcher Art die Verlinkung ist. So ein Hash ist in Listing 2 aufgeführt

In der Datenbank stehen nur IDs - siehe Listing 3.

Die Informationen an sich werden dann wieder über die Backendmodule geholt.

Mit diesem Wissen kann es jetzt an die Arbeit gehen. Da die Steuerung über das zentrale Modul Kernel::System::LinkObject geht, werden nur die Konfiguration und die Backendmodule benötigt. Wie diese aussehen, werden in den folgenden Abschnitten gezeigt.

Als erstes werden wieder Konfigurationsparameter wie oben benötigt. Diesmal ist das Quellobjekt ein Ticket und das Zielobjekt eine URL - siehe Listing 4.

```
$LinkList = {
    Ticket => {
        Normal => {
            Source => {
                12  => 'Ticket 1',
            },
        },
        ParentChild => {
            Source => {
                5 => 'Ticket2',
            },
            Target => {
                4  => 'Ticket 3',
            },
        },
    },
    URL => {
        ExternalLink => {
            Source => {
                1 => 'http://perl.org',
            },
        },
    },
};
```

Listing 2

```
mysql> select * from link_relation;
+------------------+------------+------------------+------------+---------+
| source_object_id | source_key | target_object_id | target_key | type_id |
+------------------+------------+------------------+------------+---------+
|                1 | 120        |                1 | 122        |       2 |
|                1 | 130        |                1 | 131        |       2 |
+------------------+------------+------------------+------------+---------+
```

Listing 3

```
<ConfigItem Name="LinkObject::PossibleLink###0600-ExternalLink" Required="0" Valid="1">
    <Description Translatable="1">Links a ticket with an URL.</Description>
    <Group>LinkExternalURL</Group>
    <SubGroup>Core::LinkObject</SubGroup>
    <Setting>
        <Hash>
            <Item Key="Object1">Ticket</Item>
            <Item Key="Object2">URL</Item>
            <Item Key="Type">ExternalLink</Item>
        </Hash>
    </Setting>
</ConfigItem>
```

Listing 4

```
<ConfigItem Name="LinkObject::Type###ExternalLink" Required="1" Valid="1">
    <Description Translatable="1">
        Defines the link type 'ExternalLink'. If the source name and the target
        name contain the same value, the resulting link is a non-directional one;
        otherwise, the result is a directional link.
    </Description>
    <Group>LinkExternalURL</Group>
    <SubGroup>Core::LinkObject</SubGroup>
    <Setting>
        <Hash>
            <Item Key="SourceName">ExternalLink</Item>
            <Item Key="TargetName">ExternalLink</Item>
        </Hash>
    </Setting>
</ConfigItem>
```

Listing 5

Da die Verknüpfung "ungerichtet" ist, wird für Quelle und Ziel nur ExternalLink als Typ angegeben, siehe Listing 5.

Sind es "gerichtete" Verknüpfungen - wie oben bei den Massenstörungen, muss man sich überlegen, was die Quelle und was das Ziel darstellt.

Neben der Konfiguration werden noch drei weitere Dateien benötigt. Das klingt erstmal nach viel Arbeit, aber bei zwei Modulen kann man viel von bestehenden Modulen übernehmen. Die drei Module, die benötigt werden sind

- Kernel::System::URL
- Kernel::Output::HTML::LinkObjectURL
- Kernel::System::LinkObject::URL

Die erste dieser Dateien ist für die Datenbankaktionen bezüglich der URLs da. Wenn URLs also hinzugefügt, gelöscht oder gesucht werden sollen. Der Code dieses Moduls ist an dieser Stelle uninteressant, für diesen Artikel wichtiger sind dagegen die beiden letztgenannten Module.

Kernel::Output::HTML::LinkObjectURL

Dieses Modul legt fest, wie die Daten angezeigt werden sollen wenn es um die Verknüpfung geht. Gibt es zu einem Ticket verknüpfte Daten, werden diese in einer Tabelle am Ende des Tickets angezeigt (Abbildung 3). Dabei werden für unterschiedliche Objekte auch unterschiedliche Informationen angezeigt. Bei Tickets sind dies z.B. die Ticketnummer, ... und bei externen URLs soll nur der Link angezeigt werden, wobei der User den Titel der Webseite sehen soll.

OTRS kennt für die Anzeige der Verknüpfungen zwei Sorten von Tabellen. In Abbildung 3 ist die einfache Tabelle zu sehen. Es gibt noch eine komplexe Tabelle (Abbildung 4), in der mehr Informationen angezeigt werden können und die auch im Hauptbereich zu sehen ist und nicht nur in der schmalen Spalte. Welche Art der Tabelle angezeigt wird, ist über die SysConfig einstellbar. Unter Framework->Core::LinkObject gibt es die Einstellung "LinkObject::ViewMode". Standardmäßig ist es auf "Simple" gestellt.

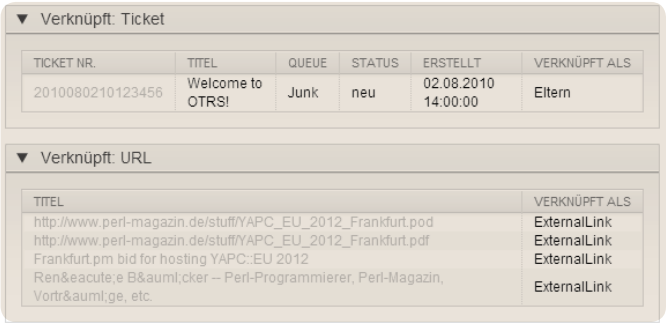


Abbildung 4: Verknüpfungen als komplexe Tabelle

Die Informationen für die Tabellen werden in den Subroutinen `TableCreate*` gesammelt und zur Verfügung gestellt. Für die komplexe Tabelle müssen viele Informationen bereitgestellt werden:

```
# define the block data
my %Block       = (
    Object    => 'URL',
    Blockname => 'Externe Links',
    Headline  => [
        {
            Content => 'Title',
        },
    ],
    ItemList => \@ItemList,
);
```

Die Angabe `Object` bestimmt das Backendmodul, der Blockname ist die Überschrift zu der Teil-Tabelle und bei `Headline` werden die Überschriften der Tabelle angegeben. Für jede Spalte muss ein anonymer Hash übergeben werden. Dabei ist "Content" immer der Schlüssel und der Wert ist die Überschrift.

In der `ItemList` sind dann die einzelnen Verknüpfungen zu finden. In der Liste muss pro Verknüpfung ein anonymer Hash genommen werden:

```
{
    Type    => 'Link',
    Key     => $URLID,
    Content => $URL->{Title},
    Link    => $URL->{URL},
},
```

Als `Type` kann gewählt werden zwischen "Text" und "Link". Ist das Element ein Link, wird die bei `Link` übergebene URL als Ziel und der bei `Content` übergebene Text als Linkbeschreibung herangezogen. Der `Key` dient der Sortierung.

Bei der einfachen Tabelle gibt es keine einzelnen Blöcke, sondern es landet alles in einem Block - wie auf Abbildung 3 zu sehen ist. Um dennoch eine Unterscheidung treffen zu können, zwischen welchen Objekttypen die Verknüpfung besteht, muss das im Hash - der zurückgegeben wird - enthalten sein. In dem Hash werden

```
%LinkOutputData = (
    ExternalLink::Source => {
        URL => [{
            Type    => 'Link',
            Content => 'Perl',
            Title   => 'Die Webseite ...',
            Link    => 'http://perl.org',
        }],
    },
);
```

Der Title ist der Alternativtext, der bei einem `MouseOver` über dem Link angezeigt wird.

In der Funktion `SearchOptionList` wird festgelegt, welche Felder für die Suche nach den Zielobjekten es im Formular (Abbildung 5) gibt. Nimmt man ein bestehendes Modul aus dem Standard als Vorlage, muss nur das Array `@SearchOptionList` angepasst werden.

```
my @SearchOptionList = (
    {
        Key  => 'URL',
        Name => 'URL',
        Type => 'Text',
    },
    {
        Key  => 'Title',
        Name => 'Title',
        Type => 'Text',
    },
);
```

Jedes Element in diesem Array ist ein Suchfeld. Der `Key` ist das Label und der `Name` eben der Name für das Eingabefeld. Als Typen stehen hier "Text" und "List" zur Verfügung. "Text" erzeugt ein einfaches Texteingabefeld, während "List" ein DropDown erzeugt. Für DropDown-Elemente muss in `SearchOptionList` vor dem Aufruf von `BuildSelection` der Wertebereich des DropDowns definiert werden. Im Modul für Tickets heißt es:

```
my %ListData;
if ( $Row->{Key} eq 'StateIDs' ) {

    # get state list
    %ListData =
        $Self->{StateObject}->StateList(
            UserID => $Self->{UserID},
        );
}
```

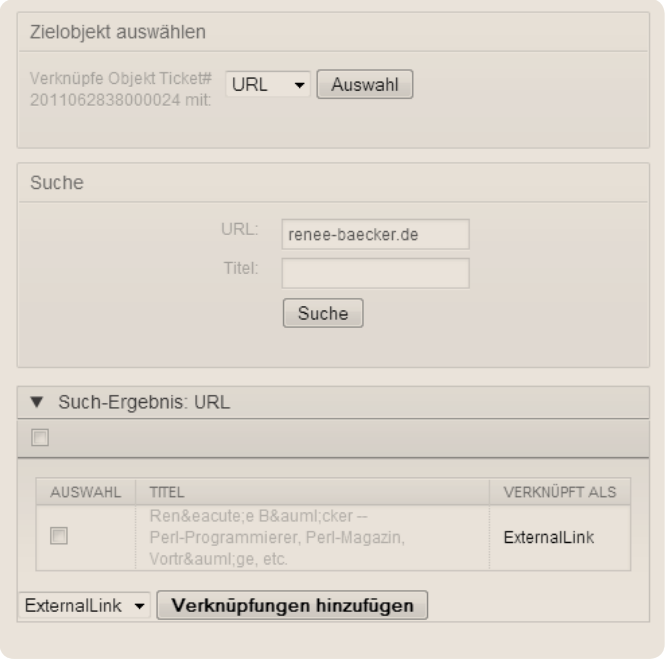
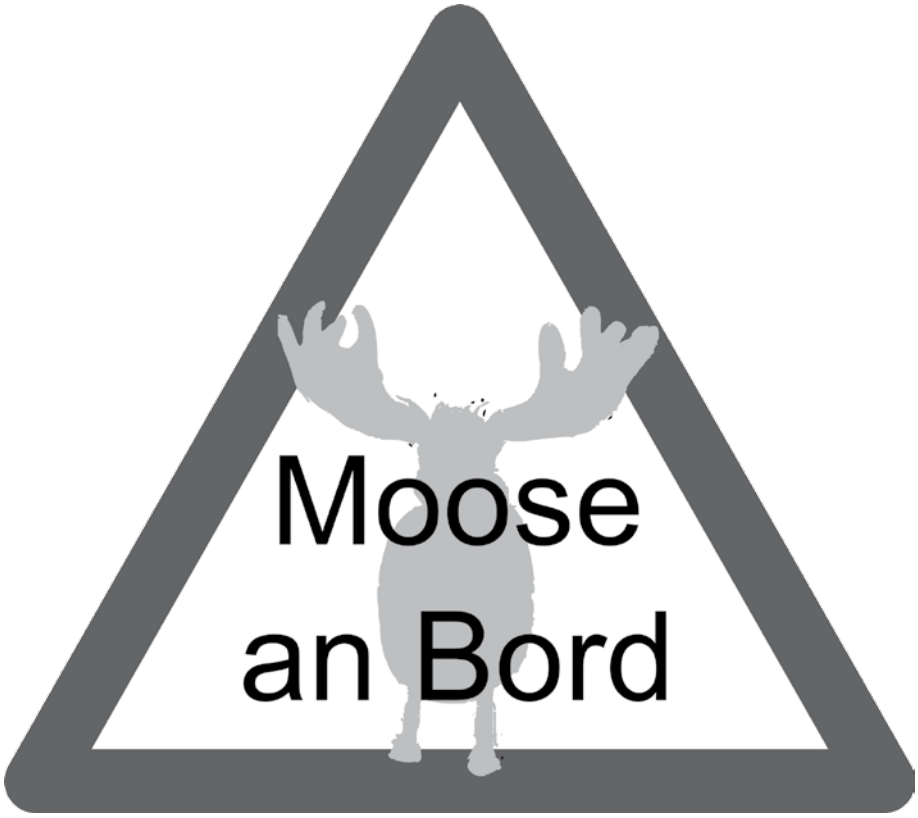


Abbildung 5: Formular um Verknüpfungen herzustellen

Kernel::System::LinkObject::URL

Im Backendmodul für die URLs wird dann die Suche nach den URLs implementiert. Diese Methode muss implementiert sein, da man im Formular nach Objekten suchen kann. Zusätzlich gibt es hier noch die Möglichkeit, auf LinkAdd*- bzw. LinkDelete*-Events zu reagieren. Das kann dann sehr sinnvoll sind, wenn noch zusätzliche Aktionen notwendig sind. Z.B. wenn der Abteilungsleiter informiert werden soll, wenn mit den Verknüpfungen neue Massenstörungen definiert werden.

Das Paket, das URLs als Zielobjekt von Verknüpfungen erlaubt, ist auch auf OPAR unter http://opar.perl-services.de zu finden. Einfach nach "LinkExternalURL" suchen.



Perl-Services.de

Programmierung - Schulung - Perl-Magazin

info@perl-services.de

Rezension - Perl - Klassiker

Einführung in Perl
Randal L. Schwartz,
Tom Phoenix & brian d foy
O'Reilly 5. Auflage 2009
ISBN 978-3-89721-887-1 € 34,90

6.Auflage, English
ISBN 978-1449303587 39,99 $
ISBN 978-1-4493-0458-4 (Ebook)

Perl - Best Practices
Damian Conway
O'Reilly, 1. Auflage 2006
584 Seiten
ISBN 978-3-89721-702-7 € 15
(derzeit nur als EPUB)

Einführung in Perl

In der letzten *$foo* habe ich *Modern Perl* als "Anfängerkurs" für ein "Perl-Trainingszentrum" vorgestellt. Damit sollte ausgedrückt werden, dass angehende Profis erst einmal den Inhalt diese Buches verinnerlichen sollten, bevor sie mit *Effective Perl Programming* ihr Detailwissen schärfen und vertiefen. Um mit Perl anzufangen ist aber chromatics Werk für die meisten Leser viel zu knapp. Dazu empfehle ich heute "Einführung in Perl", das legendäre Lamabuch.

Ungeachtet seines Alters, ist es noch auf dem aktuellen Stand, was der Großteil der Perlbücher nicht von sich behaupten kann. Erst Anfang Juni erschien die sechste Auflage, welche Perl 5.14 kennt. Der Termin für die deutsche Ausgabe, ist zur Zeit nicht bekannt, weswegen ich hier die fünfte Auflage bespreche, die aber schon von den großen "Neuerungen" berichtet, die 5.10 brachte.

Bereits kurz nach der Geburt von Perl 4 (1991!) und dem dazugehörigen Kamelbuch, dem allerersten Perltitel überhaupt, erschien das "Lama" 1993. Das "Kamel" war einfach zu tiefgründig und schwer für Anfänger und sein Koautor Randal L. Schwartz hatte gerade Material für einen Perl-Kurs beisammen.

Darin liegt nämlich der zweite große Vorteil dieses Buches. Es ist nicht nur von ausgewiesenen Kennern der Materie geschrieben, sondern basiert auf Lehrmaterial, das in vielen Jahren Praxis erprobt und optimiert wurde.

Die Kapitel bilden abgeschlossene Einheiten, haben verdaubare Größen und enthalten Übungsaufgaben mit Lösungen. Grundbegriffe der Programmierung wie Variablen und Schleifen werden zwar vorausgesetzt, aber ansonst nehmen sich die Autoren die Zeit, den Beginner Schritt für Schritt in die Gedankenwelt von Perl anhand kleiner Beispiele einzuführen. Die Sprache ist witzig, etwas großmäulig, und bemüht Vergleiche aus der Trickfilmserie die Flintstones, die Jüngere kaum mehr kennen dürften. Dieser Stil wurde von Randal geprägt, der Anfangs der alleinige Autor war. Die späteren Schreiber (Tom Phoenix in der dritten Auflage, danach Brian d Foy) mäßigten den Humor und überarbeiteten weite Teile des Textes mehrmals, weswegen es beim genauen Lesen etwas uneinheitlich wirkt.

Das war allerdings notwendig, da sich seit Perl 4 sehr viel veränderte. Auch wandelte sich das Konzept des Buches. Während am Anfang sehr schnell kleine Skripte für praktische Lösungen geschrieben wurden, ging es später weniger um CGI und Datenbanken, als um den systematischen Aufbau

von Grundkenntnissen. Der zugrundeliegende Kurs, den alle drei Autoren in Randal's Firma lehrten, wurde zudem zweiteilig, weswegen auch einige Inhalte in den Band 2, *Intermediate Perl* (Alpaka - Rezension Winter 2007), wanderten. (Der dritte Teil: *Mastering Perl* (Vikunja) wurde in der Ausgabe Frühjahr 2008 besprochen.)

Der erste Band will befähigen kleine Programme zu schreiben. Der Zweite weist den Leser in größere Programme und Datenstrukturen ein. Nach dem Dritten soll der Leser mit Hilfe von *Benchmarks*, *Logging* und *Debugging* mit den haarigen Problemen des Programmieralltags zurecht kommen aber auch in Teams arbeiten und Module veröffentlichen können.

Perl - Best Practices

Dieser Klassiker von 2006, über den in den letzten Jahren vielleicht mehr geschrieben wurde, als über jeden anderen Perlfolianten, wendet sich an Leser, die mindestens das *Lamabuch*, besser aber noch *Modern Perl* verstanden haben. Denn hier geht es nicht darum wie etwas in Perl umgesetzt wird, sondern wie man es besser, also robuster, effektiver und wartbarer tut. Und um das zu vermitteln ist Damian der Richtige. Er hat langjährige Erfahrung als Programmierer, als unterhaltsamer Dozent, kennt Perl bis in seine finsteren Winkel und vor allem seine Leser. Denn viele, die Perls Vielfalt schätzen, wehren sich mit Händen und Füßen gegen jeden, der den einen, wahren Weg predigt. Deswegen erfolgt auch immer wieder sein Rat: "Wenn ihr etwas anders machen wollt, dann wenigstens konsistent." Ich würde dem hinzufügen: ".. und seid euch der Konsequenzen, vor denen in diesem Buch gewarnt wird, bewusst".

Insgesamt war er aber mit seinen Ratschlägen erfolgreich. Bis auf wenige Ausnahmen (inklusive `Class::Std`), sind sie als Richtschnur angenommen und mit `Perl::Critic` sogar in Code gegossen worden, der über ihre Einhaltung wacht. (Wie eigene Regeln für `Perl::Critic` formuliert werden, wurde bereits in der dritten Ausgabe - Sommer 2007 - beschrieben.)

Auch auf diesen Seiten schreibt ein Meister des Faches anschaulich (mit "guten" und "schlechten" Beispielen), didaktisch souverän und zuweilen beißend ironisch. Mit Witz und deutlichen Metaphern führt er durch die Details der Syntax und wie sie seiner Meinung nach am besten zu bändigen sind. Dabei geht es von der Position eines Kommas bis zur optimalen Schnittstelle (*API*) eines Moduls. Auch Module (zum Teil von Damian selbst geschrieben) werden empfohlen und es fehlt auch nicht an Hinweisen übers Benchmarken, Testen, Loggen und Debuggen, sowie zur Fehlerbehandlung und Dokumentation.

Beim ersten Lesen entdeckt sicher jeder unbekannte Ecken an Perl und wundert sich vielleicht, wie ein einzelner Mensch das alles wissen kann. Tut er aber nicht. Das Dankeswort liest sich wie die Aufzählung eines guten Drittel der inneren Perlgemeinschaft. Daher beherbergt das Papier einen wertvollen Erfahrungsschatz, der hilft, Ideen klarer auszudrücken, Probleme zu umgehen und Programme zu beschleunigen. Denn zum Glück predigt hier kein Gelehrter aus seinem Turm wie schöner Code auszusehen hat. Allein schon der Anhang mit Editoreinstellungen zeigt, dass Damian weiß, was in der Praxis gebraucht wird.

Obwohl sich die meisten Erklärungen mit meiner Erfahrung decken, folge auch ich nicht allen Weisungen des Herrn Conway, der es, nach meinem Geschmack, mit seinem Mut neue Wege zu gehen manchmal übertreibt. Andererseits kennt er auch die akademische Literatur und bringt sie hiermit hübsch komprimiert. Das lässt sich ja immer noch dankend ablehnen, ist aber informativ und verständlich. Letztlich ist es ein sehr lesenswertes Buch, dass dringend auf den neuesten Stand gebracht werden sollte. Besonders mit den vielen Moosekomponenten und anderen neueren Modulen wie `autodie` und `Try::Tiny`, die teilweise in `Modern::Perl` erwähnt werden, dürften sich die "best practices" hier stark ändern oder müssen erweitert werden. Damian zeigte sich dieser Forderung einsichtig, allein die Zeit fehlte ihm bisher, die ihm allerdings sehr zu wünschen ist.

Herbert Breunung

Perl Tipps - Modulorte erkennen

Perl-Programmierer fragen nicht "Ey Mann - Wo is' mein Auto?", sie fragen "Ey Mann, Wo is' mein Modul?". Gerade unter Linux, wo schon das Betriebssystem sie an mehreren Orten lagert und eigenmächtige Installationen die Lage noch verkomplizieren, stellt sich die Frage schnell. Jedoch das kleine Modul `V` weiß zum Glück Auskunft zu erteilen:

```
$ perl -MV=Wx,Tk
```

Diese Zeile genügt für die Ausgabe:

```
Wx /usr/local/lib/perl5/site_perl/5.12.2/
  i686-linux/Wx.pm: 0.98 Tk Not found
```

So lassen sich die Speicherorte beliebig vieler Module bequem auf einmal abfragen, so wie es bisher nur für Module mit eingebetteter Dokumentation (*POD*) möglich war,

```
$ perldoc -l Wx Wax

No documentation found for "Wax".
/usr/local/lib/perl5/site_perl/5.12.2/
  i686-linux/Wx.pm
```

Sicher funktioniert auch folgende Zeile zuverlässig:

```
$ perl -E '$m=pop; eval "use $m"; $m =
  ~ s#::#/#g; say $INC{"$m.pm"}' Wx
```

Aber wer mag das schon jedesmal tippen. Außerdem hat `V` den Vorteil, dass es das gefragte Modul nicht lädt, was auch in größeren Skripten nützlich ist, wo der Speicher anderweitig benötigen wird. Hier kann `V` nicht helfen, da es seine Funde immer nur sofort auf `STDOUT` gibt. Da ein

```
use Wx;
say $INC{'Wx.pm'};
```

das Modul lädt und außerdem bei Fehlen des Moduls das Skript sterben lässt, nimmt man besser:

```
use Module::Info;
my $mod =
  Module::Info->new_from_module('Wx');
say $mod->inc_dir;
```

`Module::Info` liefert auch noch Version, Dateinamen und ob es im Kern enthalten ist. Sind Informationen darüber gefragt, welche anderen Module das Modul benutzt, von welchen Klassen es ableitet, welche Routinen (`sub`) es hat und welche es ruft, muss es aber schon geladen werden.

Will man den Ort des aktuellen Moduls wissen, hilft weder `$INC{'Modul/Name.pm'}` noch `__FILE__`, die beide hier locker das vom Startskript aus relative Verzeichnis ausgeben. Selbst `$mod->inc_dir` wirft hier nur das Verzeichnis aus `@INC`, von dem aus das Modul geladen wurde. Der Rest ließe sich aus dem Modulnamen rekonstruieren. Doch wozu der Aufwand, es geht einfacher:

```
use Dir::Self;
say "Dir: ", __DIR__;
```

Manchmal will aber das Modul wissen, woher es gerufen wurde. Dazu gibt es das Kernmodul FindBin.

```
use FindBin qw($File $Bin);
```

`$File` enthält den Dateipfad, `$Bin` nur das Verzeichnis. Beides gibt es auch noch mit der Vorsilbe *Real*, das alle Links auflösen lässt.

Das für das Skript aktuelle Verzeichnis gibt natürlich das altehrwürdige Kernmodul `Cwd` mit `Cwd::cwd()` heraus. Damit sind ebenfalls alle Links aufgelöst. Wer wenigstens relative Pfade in absolute umwandeln möchte (`Cwd::abs_path`), die auch Links sein dürfen, muss sich `Cwd::Ext` installieren und dies mit `Cwd::Ext::abs_path_nd()` tun.

Leserbriefe

Nachtrag zu "openSUSE Build Service"

Ein Nachtrag zu CPAN-Distributionen im openSUSE Build Service.

Im englischen openSUSE wiki pflege ich eine weitere Bauanleitung, speziell für perl Pakete:

http://en.opensuse.org/openSUSE:Packaging_Perl

Dort werden einige weitere Vereinfachungen und auch Alternativen geboten.

- Jürgen Weigert

Feedback zu"geLISPelte 6ismen: Macros und Gather/Take"

Hallo Katrin, Renée,

die Artikelserie von Rolf Langsdorf zu gelispeltem Perl finde ich recht interessant -- weiter so!

Ist es möglich, mit dem geschilderten Macro-Mechanismus ein Modul zu bauen, das einen bestimmten Funktionsaufruf in einer Compilation Unit umschreibt? Konkret stelle ich mir ein Logging-Macro vor, das seine Parameter nur dann auswertet, wenn wirklich geloggt werden soll. Dazu müsste

```
log $loglevel,
   complex_computation_of_what_to_log();
```

umgeschrieben werden in

```
do {
 do_log complex_computation_of_what_to_log()
}
if $loglevel < $GLOBAL_LOG;
```

oder so ähnlich.

Ich denke, ich wäre nicht der einzige, der solch ein Macro gut fände.

Im Moment mache ich sowas mit einem Source Filter und PPI. Das funktioniert erstaunlich gut. Trotzdem, Source Filter sind halt immer ein wenig fragil. Außerdem verlängert PPI die Ladezeit erheblich bei größeren Programmen.

Weiterhin möchte ich eine etwas andere Version des gather-take Iterators beisteuern, siehe Anhang. Das Programm benutzt Marc Lehmanns Coro::State Modul, das mehrere Ausführungs-Stacks im selben Perl Interpreter bereitstellt.

Meine Version funktioniert ohne die "Frommen Lügen" aus Rolf's Version. take() innerhalb von foreach-Schleifen funktioniert genauso wie die Verwendung von my-Variablen oder take() in Subroutinen, die in einem gather-Block aufgerufen werden.

Dafür kommen andere Einschränkungen. Man sollte sich die Liste der thread-spezifischen Variablen in Dokumentation zu Coro::State bzw. in Coro::Intro unter der Überschrift "What is mine, what is ours?" sehr gründlich zu Gemüte führen. Insbesondere im Zusammenhang mit lokalisierten globalen Variablen können wundervolle Effekte erzielt werden.

Der Test-Iterator ermittelt zuerst die ungeraden Zahlen zwischen 11 und 20. Sind diese alle, liefert er beim nächsten Aufruf einen Trenner-String. Danach wird ein weiterer Iterator innerhalb des gather-Blocks erzeugt, der die durch 3 teilbaren Zahlen beginnend ab 99 liefert. Bevor diese allerdings vom äußeren Iterator zurückgegeben werden, werden sie auf Ungeradzahligkeit gefiltert.

Daher ergibt sich für die ersten 20 Aufrufe folgender Output:

```
$ perl ~/x.pl
11
13
15
17
19
===============
99
105
111
117
123
129
135
141
147
153
159
165
171
177
```

Hiermit erlaube ich die Veröffentlichung dieses Textes als Leserbrief in einem $foo Magazin.

- Torsten Förtsch

(Anmerkung der Redaktion: Der Code von x.pl ist in den downloadbaren Codebeispielen zu finden)

TPF News

Grant-Update: Verbesserungen am Meta-Modell von Rakudo

Nachdem Jonathan Worthington viel Recherchearbeit geleistet hat und am Design - mit den dazugehörigen Prototypen - gearbeitet hat, wurde das sogenannte 6model in NQP (Not Quite Perl) eingebaut.

Auch die Integration in Rakudo ist teilweise fertiggestellt.

Grant-Update: Parrot Embed

Im "Parrot Embed"-Grant wurde ein neuer Meilenstein erreicht: Während die Tests mittlerweile über 95% des Codes abdecken, wurden Funktionen identifiziert, die aus Versehen überlebt haben. Diese wurden aus dem Code entfernt.

Um die Testabdeckung zu erhöhen, ist Jonathan Leto einen neuen Weg gegangen. Mit Hilfe von Testberichten und Papier hat er die Funktionen gesammelt, die noch nicht getestet wurden. Danach hat er die Funktionen danach geordnet, wie schwer sie zu testen sind und diese in aufsteigender Schwierigkeit abgearbeitet.

Durch die vielen neuen Tests wurden auch potentielle Bugs entdeckt.

Seit der Grant genehmigt wurde, wurde die alte Embed API aufgegeben und eine neue geschaffen. Leto arbeitet deshalb nicht mehr - wie im ursprünglichen Grant vorgesehen - an der alten Embed API, sondern an der neuen.

Neuer Chef des Marketing Kommitees

Mark Keating wurde zum neuen Vorsitzenden des Marketing Kommitees der Perl Foundation gewählt. Keating ist seit längerem sehr aktiv, Perl zu promoten. Sowohl in der Enlightened Perl Organisation als auch für die Perl Foundation beim "Google Summer of Code".

Mark Keating löst damit Dan Magnuszewski ab, der das Kommitee seit dessen Gründung 2009 geleitet hat.

Wer Ideen hat oder Keating unterstützen will, kann ihn unter mdk(at)perlfoundation.org anmailen.

Google Summer of Code 2011: akzeptierte Projekte

In diesem Jahr wurden 6 Studentenprojekte für die Perl Foundation akzeptiert:

1. André Walker – Rework Catalyst component setup code betreut von Tomas Doran und Eden Cardim
2. Brian Neil Fraser – Making the Perl Core UTF-8 clean betreut von Florian Ragwitz und Zefram
3. Carlos Ivan Sosa – Removing the upgrading necessity of the Dancer script with a module betreut von Sawyer X und Franck Cuny
4. Marc Green – Standardization of core documentation parsing tools betreut von Ricardo Signes und David E. Wheeler
5. Moritz Onken – CPAN search for the modern web betreut von Clinton Gormley und Olaf Alders
6. Tadeusz So□nierz – Pod parser for Rakudo betreut von Moritz Lenz und Carl Mäsak

Grant-Update: Strukturierte Fehlermeldungen in Rakudo

Die Perl Foundation hat den Grant-Antrag von Moritz Lenz angenommen.

Lenz wird Strukturierte Fehlermeldungen für Rakudo entwerfen und implementieren. Mit diesem System wird es ermöglicht, Fehler zu identifizieren und klassifizieren.

Der komplette Antrag kann unter http://news.perlfoundation.org/2011/02/hague-grant-application-struct.html eingesehen werden.

Mittlerweile hat Moritz Lenz den ersten Bericht zu seinem Hague-Grant veröffentlicht.

Als erstes hat er die Fehlermeldungen der verschiedenen Compiler vereinheitlicht.

Unter https://github.com/perl6/errors/ ist ein Repository zu finden, in dem die Spezifikation für die Fehlermeldungen gepflegt wird.

Lenz bittet um Feedback zu der Spezifikation.

Grant-Update: Fixing Perl 5 Core Bugs

Im März hat Dave Mitchell hauptsächlich an Bugs gearbeitet, die als Perl 5.14-Blocker eingestuft wurden.

In knapp über 22 Stunden hat Mitchell fünf Bugs gefixt und unter anderem an der Performance-Probleme bei der Funktion "keys" gearbeitet.

Außerdem hat Dave daran gearbeitet, die Smoke-Fehler zu beseitigen. Smoke-Tests laufen regelmäßig, um Fehler bei der Entwicklung des Perl-Kerns frühzeitig zu entdecken.

Spenden für Bugfixing im Perl 5 Kern

Die Perl Foundation hat einen Spendentopf für Perl 5 Arbeiten eingerichtet. Alle Spenden die dort auflaufen werden für Bugfixes und Maintenance Arbeiten am Perl 5 Kern verwendet.

In der Vergangenheit haben Nicholas Clark und Dave Mitchell solche Grants erhalten und solche Grants sollen auch in Zukunft vergeben werden.

Firmen und Privatpersonen können unter https://secure.donor.com/pf012/give spenden.

Aber schon jetzt kann sich jeder um so einen Grant bewerben. Ein Template für die Bewerbung und die genauen Kriterien sind unter http://www.perlfoundation.org/perl_5_core_maintenance_fund zu finden.

CPAN News XIX

App::Uni

Ein nützliches Tool für jeden, der viel mit Unicode zu tun hat, ist `App::Uni`. Das Modul liefert auch gleich das Programm `uni` mit, das durch die UnicodeDatabase.txt greppen kann. Das Modul ist dann besonders praktisch, wenn man ein Unicode zeichen hat und den dazugehörigen Codepoint benötigt oder wenn man wissen möchte, welches Zeichen zu einem Codepoint gehört.

```
$ uni smiling face
263A ☺ WHITE SMILING FACE
263B ☻ BLACK SMILING FACE

$ uni ☺
263A ☺ WHITE SMILING FACE
```

System::Timeout

Unter normalen Umständen gibt es bei `system`-Aufrufen keinen Timeout. Das Perl-Programm wartet so lange, bis das externe Kommando beendet ist. Soll aber nach einigen Sekunden automatisch wieder zu dem Perl-Programm zurückgekehrt werden, kann man `System::Timeout` verwenden. Als ersten Parameter muss man dann die Sekunden angegeben werden, bis der Timeout greifen soll.

```
use System::Timeout qw(system system_ex);
system("3", "sleep 9");
system_ex("3", "sleep 9");
```

YamlTime

Woran arbeite ich wie lange? Diese Frage stellt sich häufig - vor allem bei Freiberuflern wenn sie ihre Abrechnung machen wollen. Ingy döt Net hat mit YamlTime ein Programm auf CPAN veröffentlicht, mit dem das Time-Tracking sehr einfach ist. Zusätzlich werden die ganzen Informationen im YAML-Format gespeichert, so dass es problemlos von jedem lesbar ist und auch an anderer Stelle weiterverwendet werden kann.

Man muss sich zwar erstmal in den Konfigurationsdateien zurechtfinden und sich zusammensuchen wie was funktioniert, aber insgesamt ist das Programm ganz nützlich.

```
reb@ub:~/community/yt_dir$ yt new
Task Description:  Write "CPANNews XIX"
Customer Id:  foo
Project Id:  article
A Tag Word:  magazine
A Tag Word:
Started task 2011/06/13/0923.
reb@ub:~/community/yt_dir$ yt stop
Stopped task 2011/06/13/0923. Time: 0:01
reb@ub:~/community/yt_dir$ yt dump
time: 0:01
task: Write "CPANNews XIX"
cust: foo
proj: article
tags:
  - magazine
refs: []
note: ''
```

Net::LDAP::Server::Test

Funktioniert mein Code, der mit `Net::LDAP` arbeitet? Es existiert aber kein LDAP-Server? Für Tests kann dann `Net::LDAP::Server::Test` verwendet werden. Ist sehr praktisch für autarke Entwicklungsumgebungen.

```
use Test::More tests => 10;
use Net::LDAP::Server::Test;

ok( my $server =
 Net::LDAP::Server::Test->new(8080),
        "test LDAP server spawned");

# connect to port 8080 with your
# Net::LDAP code.
ok(my $ldap = Net::LDAP->new(
                'localhost', port => 8080 ),
          "new LDAP connection" );

# ... test stuff with $ldap ...

# server will exit when you call final
# LDAP unbind().
   ok($ldap->unbind(),
      "LDAP server unbound");
```

Smart::Match

Der Smartmatch-Operator wurde mit Perl 5.10 eingeführt. Selbst gute Perl-Programmierer haben hin und wieder Probleme zu wissen, wie der Smartmatch aussehen muss, um das gewünschte Ergebnis zu bekommen. Leon Timmermans hat mit `Smart::Match` ein Modul geschrieben, das die Verwendung des Operators vereinfacht. Um das Modul einsetzen zu können, benötigt man mindestens Perl 5.10.1.

```
use Smart::Match qw(positive any);
use feature 'switch', 'say';

my $foo = 19;
given ( $foo ) {
    when( positive ) { say 'test' }
}

$foo *= -1;

warn "Zahl muss positiv sein!\n"
  if not $foo ~~ positive;

say "Im Array enthalten"
  if $foo ~~ any( -19 .. 1 );
```

Pod::Cpandoc

Manchmal ist es notwendig, die Dokumentation eines Moduls anzuschauen, das nicht installiert ist. Also Browser öffnen und auf CPAN vorbeischauen. Diese Zeiten sind jetzt vorbei: Mit `Pod::Cpandoc` wird die Dokumentation solcher Module von CPAN geholt und dann wie bei `perldoc` angezeigt. Ist das Modul installiert, wird die Dokumentation des installierten Moduls angezeigt, so dass die Dokumentation der tatsächlichen Version auf dem Rechner entspricht.

```
entwicklung@ubuntu:~/community$ cpandoc Acme::Doublehelix
No documentation found for "Acme::Doublehelix".
entwicklung@ubuntu:~/community$ cpandoc Win32::GUI
entwicklung@ubuntu:~/community$ perldoc Win32::GUI
No documentation found for "Win32::GUI".
entwicklung@ubuntu:~/community$ cpandoc -v Win32::GUI
Ending switch processing.  Args are [Win32::GUI] with 0 errors.
/usr/local/bin/cpandoc => Pod::Cpandoc v0.07

Formatter class Pod::Perldoc::ToMan successfully loaded!
Will format with the class Pod::Perldoc::ToMan
Searching for Win32::GUI
Looking for Win32/GUI in /usr/bin ...
Going to query http://api.metacpan.org/pod/Win32::GUI
Found as /tmp/Win32-GUI-VxhC
About to try calling /usr/bin/sensible-pager /tmp/X4o0xITF0S
entwicklung@ubuntu:~/community$
```

Termine

August 2011

01.	Treffen Hamburg.pm
02.	Treffen Frankfurt.pm
	Treffen Vienna.pm
04.	Treffen Dresden.pm
08.	Treffen Ruhr.pm
	Treffen Hannover.pm
9.	Treffen Stuttgart.pm
15.	Treffen Erlangen.pm
15.-17.	YAPC:EU in Riga
17.	Treffen Darmstadt.pm
10-21.	FrOSCon
30.	Treffen Bielefeld.pm
31.	Treffen Berlin.pm

September 2011

01.	Treffen Dresden.pm
05.	Treffen Hamburg.pm
06.	Treffen Frankfurt.pm
	Treffen Vienna.pm
08.-09.	Italienischer Perl-Workshop
12.	Treffen Ruhr.pm
	Treffen Hannover.pm
13.	Treffen Stuttgart.pm
19.	Treffen Erlangen.pm
21.	Treffen Darmstadt.pm
	Trefen Munich.pm
27.	Treffen Bielefeld.pm
28.	Treffen Berlin.pm

Oktober 2011

03.	Treffen Hamburg.pm
04.	Treffen Frankfurt.pm
	Treffen Vienna.pm
06.	Treffen Dresden.pm
08.	Belgischer Perl-Workshop
08-09.	Pittsburgh Perl-Workshop
10.	Treffen Ruhr.pm
	Treffen Hannover.pm
11.	Treffen Stuttgart.pm
13.-15.	YAPC::Asia
17.	Treffen Erlangen.pm
19.	Treffen Darmstadt.pm
19.-21.	Deutscher Perl-Workshop
25.	Treffen Bielefeld.pm
26.	Treffen Berlin.pm
28-29.	Kieler OpenSource und Linuxtage

Hier finden Sie alle Termine rund um Perl.

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

Uhrzeiten und weiterführende Links zu den einzelnen Veranstaltungen finden Sie unter

http://www.perlmongers.de

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

termine@foo-magazin.de

LINKS

http://www.perl-nachrichten.de

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

http://www.perl-community.de

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

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

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

http://www.perl-workshop.de

Der Deutsche Perl-Workshop hat sich zum Ziel gesetzt, den Austausch zwischen Perl-Programmierern zu fördern.

http://www.perl-foundation.org

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

http://www.Perl.org

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

6th FrOSCon

Free and Open Source Software Conference

20. - 21. August 2011

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

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

Sponsored by:

Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Fachbereich Informatik

tarent

BOOKING.COM
online hotel reservations

www.froscon.de

YAPC Europe

Riga 2011

15.-17. August 2011