Ausgabe 2/2010 – www.foo-magazin.de - 6,00 €

$foo

PERL MAGAZIN

XMLSocket-Klasse
in Verbindung mit Perl-Sockets

Devel::REPL
eine Shell für Perl

Komplexe Datenstrukturen mit Data::Dumper
Schluss mit HASH(0x814cc20)

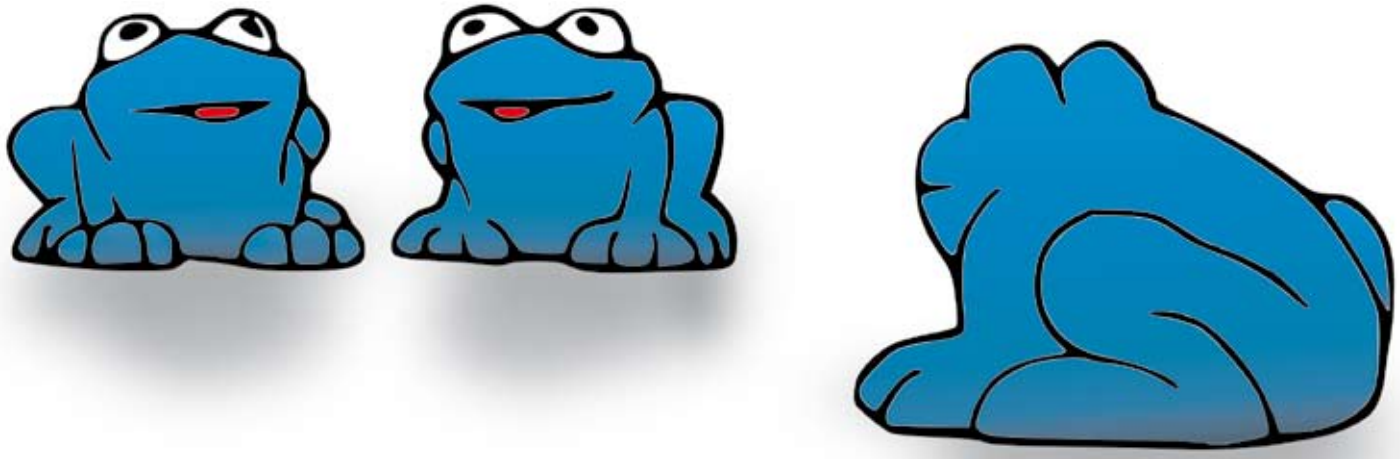
Nr 14

FrOSCon?

Free and Open Source Software Conference

Klar, am 21. & 22.08.2010 in Sankt Augustin!

Call for Papers 12.4. bis 23.5.



Deutschlands drittgrößte Free and Open Source Software Conference feiert ihr 5jähriges Jubiläum!

Highlights dieses Jahr sind:

- Hochkarätige Talks, Projekte und Workshops
- Große Geburtstagsparty am Samstagabend
- Hüpfburg
- Creative Contest und vieles mehr

Weitere Infos auf www.froscon.de und auf twitter

VORWORT

Messen, Konferenzen und Marketing von Perl

Schon im Vorwort der letzten Ausgabe bin ich kurz auf das Thema "Marketing von Perl" eingegangen. Diesmal kann ich kurz von zwei Veranstaltungen berichten.

Im Februar fand die FOSDEM 2010 in Brüssel statt. Gábor Szabó und einige weitere Perl-Programmierer haben hier einen Perl-Stand betrieben und den Besuchern Perl vorgestellt und gezeigt, dass Perl nicht unbedingt im Perl4-Stil geschrieben werden muss.

Anfang März stand dann eine sehr große Messe auf dem Plan: die CeBIT. Perl hatte hier neben 14 weiteren Open Source Projekten einen kostenlosen Stand in der Open Source Project Lounge.

Auf den zwei Veranstaltungen gab es unterschiedliche Besuchergruppen. Auf der FOSDEM waren es überwiegend Programmierer, mit denen man auch eher über Details reden konnte. Auf der CeBIT waren es hauptsächlich Besucher aus dem Management-Umfeld. Aber auch dort hatten wir viel zu tun und wir haben erfahren, dass Perl in sehr vielen Unternehmen eingesetzt wird - und das nicht nur für ganz kleine Perl-Skripte.

Insgesamt waren die zwei Veranstaltungen ein sehr großer Erfolg. Die Helfer haben auch viel darüber gebloggt.

Wir sind noch in der Lernphase und es passieren sicherlich noch einige Fehler, aber das wird sich im Laufe der Zeit ändern. Es wird auch langsam ein Fundus an Marketing-Materialien aufgebaut.

Im Wiki der Perl Foundation gibt es eine lange Liste von Veranstaltungen, bei denen es einen Perl-Stand und/oder Perl-Vorträge geben könnte. Da das aber nicht alles von einigen wenigen Personen gemacht werden kann, sind hier natürlich Freiwillige geplant. Sie kennen weitere Veranstaltungen, die in der Liste fehlen? Dann bitte einfach ergänzen.

Sollten Sie Interesse haben, bei so einer Veranstaltung zu helfen, können Sie sich auf der Events-Mailingliste events@lists.perlfoundation.org melden.

Besonders an unsere Schweizer Leser: Ich würde gerne auf dem FrOSCamp einen Perl-Stand machen. Würde jemand helfen? Dann bitte eine kurze Mail an die Events-Mailingliste oder an info@foo-magazin.de.

Jetzt wünsche ich Ihnen noch viel Spaß mit der neuen Ausgabe von $foo - Perl-Magazin.

Renée Bäcker

Die Codebeispiele können mit dem Code

3udnwo

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

Alle weiterführenden Links werden auf del.icio.us gesammelt. Für diese Ausgabe: http://del.icio.us/foo_magazin/issue14.

IMPRESSUM

Herausgeber:	Smart Websolutions Windolph und Bäcker GbR Untere Rützelstr. 1a D-65933 Frankfurt
Redaktion:	Renée Bäcker, Katrin Bäcker, André Windolph
Anzeigen:	Katrin Bäcker
Layout:	//SEIBERT/MEDIA
Auflage:	500 Exemplare
Druck:	powerdruck Druck- & VerlagsgesmbH Wienerstraße 116 A-2483 Ebreichsdorf
ISSN Print:	1864-7537
ISSN Online:	1864-7545

INHALTSVERZEICHNIS

ALLGEMEINES

ANWENDUNG

PERL

MODULE

TIPPS & TRICKS

NEWS

Hier werden kurz die Autoren vorgestellt, die zu dieser Ausgabe beigetragen haben.

Renée Bäcker

Seit 2002 begeisterter Perl-Programmierer und seit 2003 selbständig. Auf der Suche nach einem Perl-Magazin ist er nicht fündig geworden und hat so diese Zeitschrift herausgebracht. In der Perl-Community ist Renée recht aktiv - als Moderator bei Perl-Community.de, Organisator des kleinen Frankfurt Perl-Community Workshops, Grant Manager bei der TPF und bei der Perl-Marketing-Gruppe.

Ferry Bolhár-Nordenkampf

Ferry kennt Perl seit 1994, als sich sein Dienstgeber, der Wiener Magistrat, näher mit Internet-Technologien auseinanderzusetzen begann und er in die Tiefen der CGI-Programmierung eintauchte. Seither verwendet er – neben clientseitigem Javascript – Perl für so ziemlich alles, was es zu programmieren gibt; auf C weicht er nur mehr aus, wenn es gar nicht anders geht – und dann häufig auch nur, um XS-Module für Perl schreiben. Wenn er nicht gerade in Perl-Sourcen herumstöbert, schwingt er das gerne das Tanzbein oder den Tennisschläger.

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.

Colin Hotzky

Colin Hotzky studierte Informatik an den Universiäten in Leipzig und Liverpool. Schon während der Studienzeit kam er in Kontakt mit Perl. Für die Diplomarbeit nutzte er Perl zur Implementierung eines Suchverfahrens in irregulär strukturierten XML-Dateien. Heute arbeitet er als Software-Entwickler für die Siemens AG an einem Perl-basierten Kundenportal im Bereich ITSM und IT-Outsourcing.

Thomas Fahle

Perl-Programmierer und Sysadmin seit 1996.
Websites:

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

Dr. Johannes Mainusch

Dr. Johannes Mainusch, Vice President Operations, XING AG. Als Vice President Operations betreut Dr. Johannes Mainusch den Betrieb und die technische Entwicklung der XING-Plattform, einer High-Traffic Webseite mit über 8.3 Millionen Mitgliedern (Stand 11/2009) weltweit. Mit dem Fokus auf iterativen Projektmethoden war er zuvor sowohl für Lufthansa-Systems als auch beratend für zahlreiche Industrie- und Logistikunternehmen tätig.

Markus Schaffer

Markus Schaffer, 34,Jahre alt, lebt in München und ist Freelancer im Bereich Webdesign und -Programmierung. Als Webdesigner setzt er die gängigen Sprachen und Werkzeuge ein wie XHTML/CSS, Javascript, Flash/Flex/Actionscript und auf der Serverseite PHP, SQL und Perl, wobei ihn bei seinem Job vor allem die Kombination aus kreativem Design und kühlem, logischem Programmieren anspricht.
In seiner Freizeit entwickelt er gerne eigene Projekte, darunter auch kleinere Flash-basierte Internet-Multiplayer-Spiele.

Wer langsam ist, wird verlassen
Performance großer Webseiten
- Teil 2 -

Operations Monitoring - Messen, was los ist

Performanceoptimierung im Unternehmen beginnt immer mit der Messung des Status Quo, häufig aus gegebenem Anlass. Etwa: Das Unternehmen verliert Kunden wegen schlechter Performance, oder viele Kunden beschweren sich über schlechte Performance und alle bestehenden Messungen zeigen grüne Ampeln. Nur wer misst und den Status Quo kennt, kann sich objektiv verbessern. Hier einmal folgendes Beispiel, so geschehen bei XING im Sommer 2008. Damals meldeten sich häufiger Benutzer beim Support und beschweren sich über schlechte Performance. Die bestehenden Messungen haben aber keine Abnormalitäten gezeigt. Erst nach längerer Recherche haben sich die Ursachen dafür offenbart, darunter folgende:

- Auslieferung viel zu vieler Daten pro Pageview
- javascript Memory Leaks in der damals verwendeten Version der prototype-Library

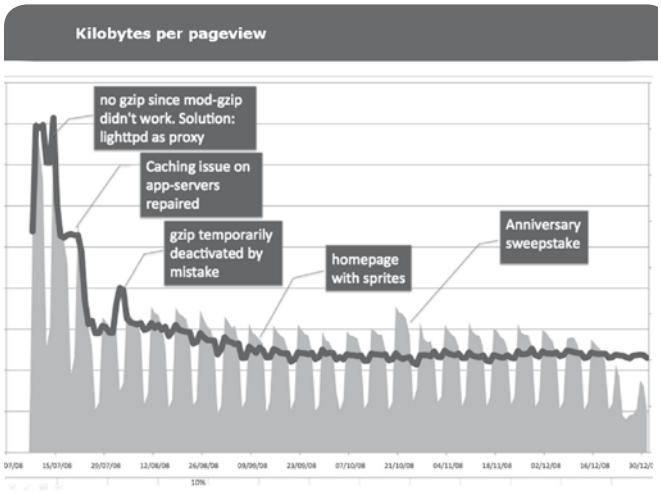


Abbildung 1: Kilobytes pro Pageview bei XING zwischen Sommer 2008 und Anfang 2009. Die Linie zeigt den Verlauf der täglich gemessenen KB/PV im Verlauf verschiedener Performancemaßnahmen.

Beide Fehler haben wir in den recht umfangreichen bestehenden Messungen nicht nachvollziehen können, da die Messungen nie die Situation am Browser des Nutzers einbezog. Erst eine Analyse des Verhältnisses zwischen der Menge der pro Tag ausgelieferten Daten und den Pageviews verwies auf die richtige Fährte (siehe Abbildung 1).

Gleichzeitig wurde bei XING begonnen, stichprobenartige Messungen direkt am Browser des Benutzers durchzuführen, um ein Bild der Performance 'vor Ort' zu erhalten. Dazu wurde bei der Auslieferung der 'index.html' Datei im Kopf ein Javascript-Zeitstempel gesetzt und bei Erreichen des Javascript-Events 'OnDomLoaded', also bei Fertigstellung des Browserinhalts, die Zeitdifferenz gemessen und per AJAX-Request an XING zurückgemeldet. Diese Verfahren wurden inzwischen verfeinert und bilden heute die Basis der Perfor-

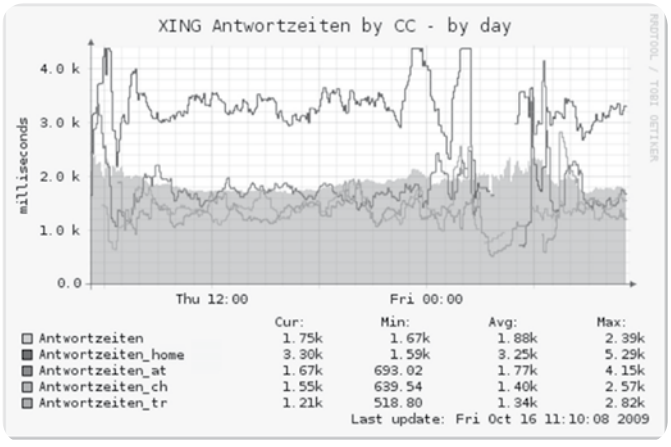


Abbildung 2: Dargestellt sind in grau die durchschnittlichen Antwortzeiten der XING-Seite für einen Tag. Zu sehen ist weiterhin die Verbesserung der Performance durch die Maßnahme Prefetching. Als Linien sind die durchschnittlichen Antwortzeiten verschiedener Regionen sowie der eingeloggten Startseite zu sehen.

mancemessungen bei XING. So lässt sich auch jede durchgeführte Performanceoptimierung direkt durch Messergebnisse belegen.

So zeigt beispielsweise Abbildung 2 die Verbesserung der durchschnittlichen Antwortzeiten durch das HTTP-Prefetching und die damit erreichte Optimierung des Cachings im Oktober 2009.

Fazit: Die Messung der Performance am Browser des Benutzers ermöglicht es, Performanceoptimierungen ebenso nachzuvollziehen, wie Veränderungen beispielsweise nach Releases.

Erfahrungsberichte und Pläne - praktische Tipps

Prefetching

Was macht der Browser, wenn die Seite fertig geladen und gerendert ist? Nichts, nada, idlen, rien! Er wartet darauf, dass der User eine Interaktion (neue Seite, AJAX-Request etc.) initiiert. Dabei lässt sich diese Zeit wesentlich sinnvoller nutzen.

Wenn der Benutzer zum Beispiel etwas in ein Suchfeld eintippt, wird die darauf folgende Seite mit ziemlicher Sicherheit die 'Suchergebnisseite' sein. Oder wenn der User auf xing.com das Login-Formular ausfüllt, wird er als nächstes höchstwahrscheinlich die eingeloggte Startseite zu sehen bekommen.

Diese Erkenntnis nutzen wir, um so bestimmte Assets (Bilder, CSS- und Javascript-Dateien) vor der eigentlichen Verwendung in den Browser-Cache zu laden. Werden diese Assets dann tatsächlich über den Quellcode der ausgeliefeten Seite referenziert, wird die entsprechende Datei aus dem Browsercache geladen und nicht neu beim Server angefragt. Die Technik lässt sich am Besten als 'anticipated preloading' beschreiben. Man vermutet, was der User als nächstes macht, am Besten basierend auf genauen und ausführlichen Trackingdaten zum Userverhalten, und lädt dann statischen Content im voraus.

Die Erfolge dazu sind durchaus messbar (siehe obere Linie in der Abbildung 4).

Seit Oktober verwenden wir das 'anticipated-preloading' bei allen Login-Formularen, um Assets vor deren Verwendung zu laden. Dabei laden wir (Stand: Dezember 2009) Grafiken der meistbenutzten Seiten von *https://www.xing.com* vor. Das Vorladen auch von Grafiken, die nicht unmittelbar benutzt werden führt dazu, dass der User sie XING-Seite immer mit einer 'primed cache experience', also mit vorgeladenem Cache, wahrnimmt. Damit umgeht man die Problematik des schlechten Browser-Cachings.

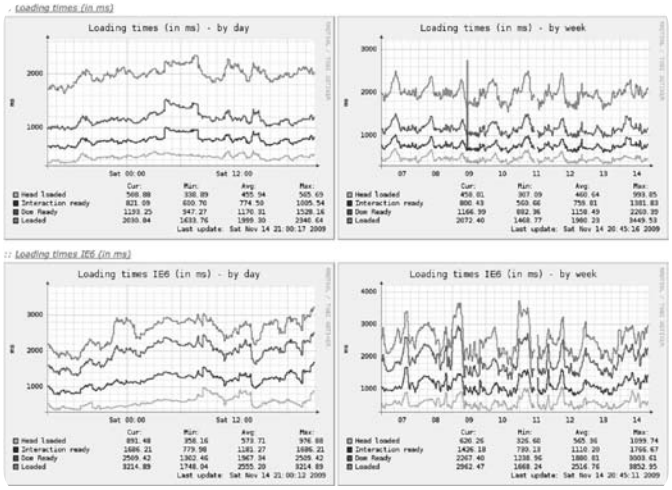


Abbildung 3: Abgebildet sind die Renderingzeiten für die XING Website über alle Browser gemittelt und speziell für den Internet Explorer 6.

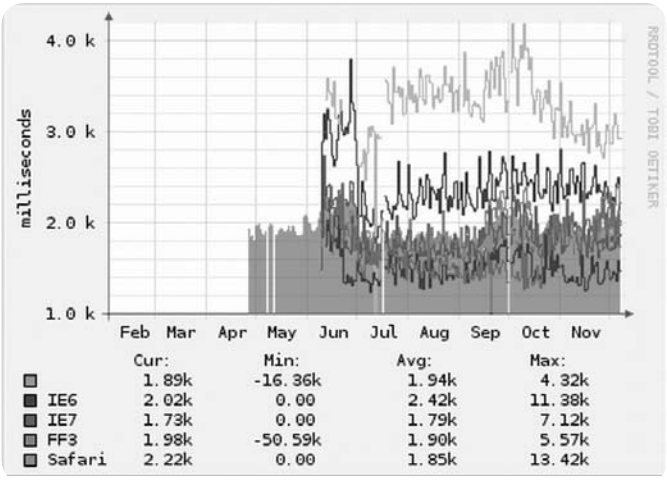


Abbildung 4: Durchschnittliche Antwortzeit auf XING.

Kombiniert mit anderen Maßnahmen können wir hier eine Verringerung der Antwortzeit von gut 20% feststellen. Weitere Unterstützung bei dieser Technologie ist hierbei von Mozilla zu erwarten. Dort gibt es seit der Version 3.0 für http und seit 3.5 auch für https ein 'Prefetch Tag':

```
<link rel="prefetch" href="/images/big.jpg">
```

Mit Hilfe dieses Tags wird der Browser angewiesen, nach vollständigem Laden der Seite das unter 'href' spezifizierte Bild vorzuladen. So fällt beim folgenden Seitenaufruf ein Request weg. Das funktioniert auch mit kompletten Seiten:

```
<link rel="next" href="2.html">
```

Näheres dazu beispielsweise unter *https://developer.mozilla.org/en/Link_prefetching_FAQ*.

Browser Caching und warum es alleine keine Lösung ist

Jeder Browser heutzutage kommt mit bis zu drei verschiedenen Caches: Am bekanntesten ist sicherlich der 'Disk Cache', der sich in der Regel im Profilordner des Benutzers befindet. Er fasst per default 50MB bei den Browsern Internet-Explorer 7/8 sowie dem Firefox 2 und neueren Versionen.

Dazu gibt es den '**Memory Cache**', der im flüchtigen Hauptspeicher des Systems lokalisiert ist. Er dient ausschließlich dem besseren Surferlebnis. Seiten in der Browser-History werden hier flüchtig gespeichert. Ebenso Assets aus SSL-Verbindungen, wenn diese aus Sicherheitsgründen nicht im Disk Cache hinterlegt werden sollen. In Mozilla-basierten Browsern ist dieser per default 30MB groß, Microsoft-Browser sind hier intransparent und überlassen den Memory Cache erwartungsgemäß dem Betriebsystem.

Bei Browsern, die HTML5 und Session Storage unterstützen, gibt es zudem den Offline Disk Cache'. Dieser ist vergleichbar mit einer SQLite-Datenbank und über JavaScript bedienbar. In der Regel ist dieser mit 500MB bemessen. GoogleMail nutzt dieses Feature, um seine Anwendung auf offline verfügbar zu machen.

Bei XING fokussieren wir uns beim Caching auf die optimale Nutzung des Disk Caches. Der Memory Cache lässt sich nicht gezielt nutzen und wäre zudem keine nachhaltige Optimierungsmaßnahme. Die gezielte Nutzung von Offline Disk Cache steht in unseren Augen noch nicht in einem gesunden Kosten-Nutzen-Verhältnis, da eine breite Unterstützung fehlt.

Der Idealzustand beim clientseitigen Caching wird auf Abbildung 5 gezeigt:

Da sämtliche Assets bereits beim Client gecached sind, werden keinerlei Request an den Server gesandt, sondern lediglich das HTML sowie etwaige Trackingpixel neu gezogen. Maximal soll die noch die Aktualität vereinzelner Assets mit einem 304-Request beantwortet werden. Das ist dann der minimale Seitenaufruf: Minimaler Traffic, nur die nötigsten Requests und eine, auf die Downloadzeit bezogen, minimale Verarbeitungszeit. Aber ist dieser Zustand wirklich realistisch? Auf unserer Plattform haben Internet Explorer 8, Opera und FF2/3.x einen Anteil von gut 50% mit steigender Tendenz (siehe Abbildung 6).

Bei diesen Browsern ist der Disk Cache im Lieferzustand 50MB (Opera 20MB) groß. Das scheint auf den ersten Blick im Web eine ausreichende Größe zu sein, ist doch ein kompletter Page Request selten größer als 500KB.

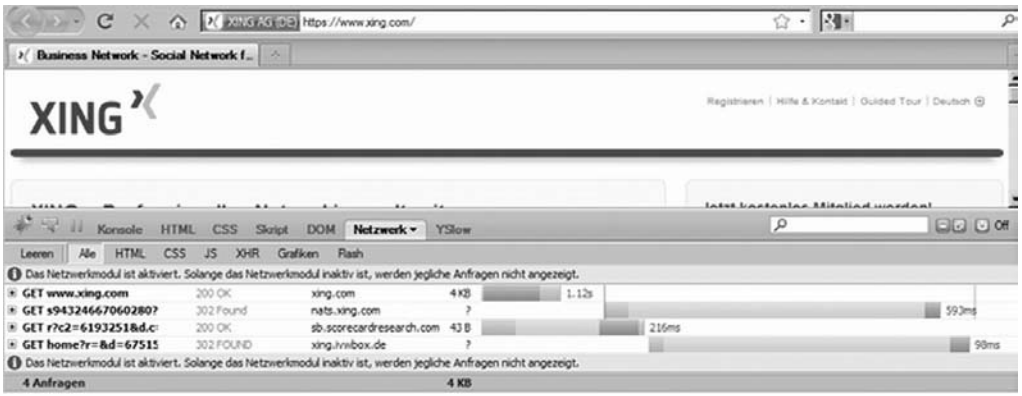


Abbildung 5: Der optimale Caching-Zustand beim Laden der XING-Seite.

Dabei muss man jedoch einige gravierende Aspekte beachten:

1.) Der Großteil der Nutzer leert seinen Cache niemals.

2.) Durch immer größereWeb (2.0) Sites und gut dokumentierte Anleitungen (z.B. Steve Souders) werden auch die Bemühungen der Betreiber immer größer, den Cache des Besuchers maximal zu nutzen.

3.) Der Cache funktioniert grob betrachtet wie eine FIFO-Liste. D.h. Dateien, die zuerst reinkommen, wandern in der Regel auch zuerst wieder raus, wenn der Speicher voll ist und sie nicht kürzlich aufgerufen wurden.

4.) Einige Browser cachen SSL Content wie den von xing.com erst in den neuen Versionen (Firefox ab 3.0 und auch nur, wenn der Cache-Header 'Public' ist).

Bedingt durch die Entwicklung des Webs wird der Quotient aus kilobytes/pageview tendenziell immer größer. Unsere eigenen Messungen auf xing.com bestätigen diesen Trend. Immer mehr Content und mittlerweile optimierte Caching Header (siehe 2.) führen dazu, dass der Disk Cache beim Besucher schnell gefüllt wird. Ist er dann voll (siehe 1.) werden Daten vom Browser gelöscht, die eigentlich dort bleiben sollten.

Welche Datei zuerst gelöscht wird, bestimmt eine Kombination aus 'der Gesamtzeit der Datei im Cache' und der 'letzten Verwendung'.

Was ist die Konsequenz daraus? Ein Großteil der Nutzer kommt nicht in Genuss, die Assets komplett aus dem Browser-Cache zu beziehen, sondern muss die Assets neu beim Server anfragen. Das macht gut 40% mehr Ladezeit aus. Wenn man es genau misst, so ist der Anteil der Besucher, die die Seite mit einem ganz oder teilweise leeren Cache besuchen erstaunlich hoch.

Mit einer eigenen, JavaScript-basierten Messung dazu ergeben sich bei uns Werte von 45% bis 62% an Usern, die xing.com mit einer 'empty cache experience' besuchen. Dies ist ein unerwartet schlechter Wert: siehe die mittlere Linie in Abbildung 7.

Unsere Messungen decken sich mit einer Studie von Yahoo. Diese messen mit einer Backend basierten Methode auch etwa 40% - 60% 'empty cache experience'. (*http://yuiblog.com/blog/2007/01/04/performance-research-part-2/*)

Fazit: Ein starkes Caching wirkt sich in jedem Fall positiv auf die User Experience aus und es sollte nicht darauf verzichtet werden. Allerdings dürfen diese Bemühungen nicht die einzigen sein. Es gibt zuviele nicht beeinflußbare Faktoren, die schnell dafür sorgen, dass das komplette clientseitige Caching ausgehebelt wird und der User doch alle Assets neu herunterladen muss. Hierfür sollten dann Strategien wie Rationalisierung und Modularisierung in Verbindung mit Lazy-Loading verwendet werden. Nur diese Kombination von effizientem Caching und minimalen Requests garantiert eine nachhaltig positive User Experience.

Within Viewport

Es wird nun keinen mehr erstaunen, dass wir bei XING viel Aufwand treiben, um die Performance zu verbessern und die Last auf unseren Servern zu verringern. Die Einführung von Open-Social-Applikationen im Sommer 2009 war eine neue

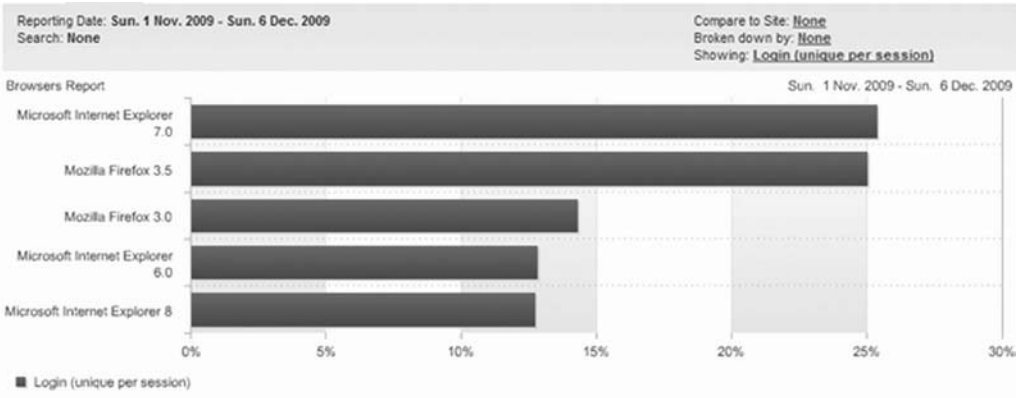


Abbildung 6: Die prozentuale Verteilung der Top-5-Browser bei XING.

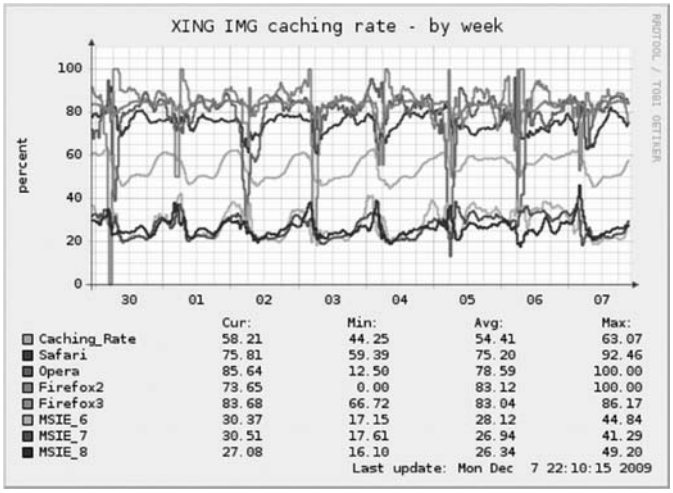


Abbildung 7: Die Image-caching-Rate verschiedener Browser gemessen bei XING.

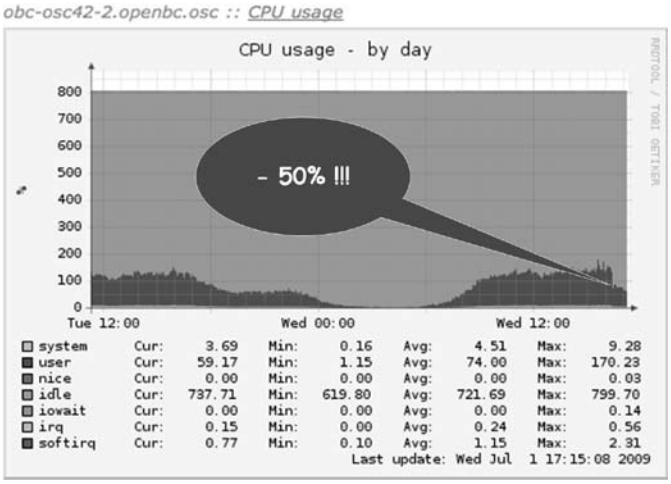


Abbildung 8: 'In-Viewport'-Nutzung für Open-Social-Gadgets und die 50%ige Verbesserung der Serverlast.

Herausforderung. Die Applikationen werden teils auf externen Servern betrieben, wir integrieren sie dann per iFrame auf der Startseite (Home View) oder auf der Profilseite und auf einer eignen Seite, dem so genannten Canvas View.

Wenn ein Nutzer die XING-Startseite im Browser aufruft, werden die verschieden Applikationen in iFrames geladen. Elegant mit einer Einschränkung: Bei jedem Laden der Startseite haben die Browser der Kunden und auch unsere internen Server viel zu tun. Im Extremfall des Internet Exporers werdenüber 8 Megabyte Ram für jeden dargestellten iFrame im Browser benötigt. Auch wenn viele dieser iFrames nur angezeigt werden, wenn der Benutzer im Browser nach unten scrollt, so muss der Browser doch alle iFrames laden und verarbeiten.

Aber, muss das so sein? Die Antwort ist 'Nein!'. Das bei XING implementierte Algorithmus enthält dazu folgende Schritte:

- Entdecke, ob eine Applikation im Viewport ist.
- Lade und stelle den iFrame nur dar, wenn ein leeres Patzhalterelement sichtbar ist.
- Prüfe regelmäßig, ob die erste Bedingung erfüllt ist.

In der ersten implementierten Lösung wurde der iFrame dargestellt und das 'src' attribut nur dann gesetzt, wenn das Element sichtbar ist. Das zog allerdings einige Probleme in einem speziellen Browser nach sich. Falls sie es noch nicht erraten haben, hier einige Tipps: Es handelt sich um den gleichen Browser, für den Youtube und Amazon demnächst den Support einstellen werden. Derselbe Browser, der eine Warnung über gemischtem Https- und Http- Content auswirft wenn das 'src'-Element im iFrame leer ist. Ein weiterer Grund dafür, es etwas anders zu implementieren liegt darin, dass das iFrame-Tag in allen Browsern sehr lange für die Darstellung braucht.

Die tatsächlich implementierte Lösung enthält ein leeres DOM-Element, z.B. , auf der Seite, dass dann durch den iFrame mit der Applikation ausgetauscht wird, sobald es im Viewport, also in der Anzeige der Benutzers erscheint.

Der letzte Schritt war dann noch, alle 200 Millisekunden zu prüfen, ob der iFrame nun sichtbar ist, ebenso wenn das Browserfenster in der Größe verändert wurde oder im Browser gescrollt wurde.

Das Schöne an dieser Verbesserng: Nicht nur die Nutzer profitieren davon, auch unsere Server. Wir haben die Last auf unseren Open-Social-Servern durch diese Performance-Maßnahme um 50% gesenkt (siehe Abbildung 8).

Fazit: Die Messung der Performance am Browser des Benutzers ermöglicht es, Performanceoptimierungen ebenso nachzuvollziehen, wie Veränderungen beispielsweise nach Releases.

Diese Funktion wurde als Plugin fur Prototype auf GitHub von XING veröffentlicht. Laden sie es ich und bauen sie sich

das in ihrer Umgebung ein. Das Plugin fügt ein Javascript-Event namens 'within:viewport' hinzu, das folgendermaßen verwendet wird - siehe Listing 1.

Dieses Plugin kann auch für andere 'teure' Operationen, wie beispielsweise das bedarfsgerechte Laden großer Bilder verwendet werden.

Konklusion

Es ist viel Arbeit, eine Webseite mit guter Performance zu betreiben und weiter zu entwickeln. Die Verantwortung dafür zieht sich durch das gesamte Unternehmen. Nötige Voraussetzung sind ein ein gemeinsames Ziel, das ständige Messen des Status Quo und Know-How zur Identifizierung und Umsetzung der Maßnahmen. Lohnt sich der Aufwand? Ja sicher, denn es geht bei Performance um Gewinnung neuer Kunden und deren Bindung an die Webseite! Inhalt und Performance sind die zwei Schlüssel zum Erfolg profitabler Webseiten.

Dr. Johannes Mainusch mit Beiträgen
von Christopher Blum und Björn Kaiser

```
loadIframe = function() {
var placeholder, iframe;

placeholder = this;
iframe = new Element("iframe", {
width: 315,
height: 315,
src: "http://iframe-url"
 });
 placeholder.replace(iframe);
 };


 // This is where the magic happens...
 $("placeholder").observe("within:viewport", loadIframe);
```

Listing 1

Perl und die Feuerwehr - Teil II

In der letzten Ausgabe habe ich gezeigt, wie die graphische Oberfläche für das Programm erstellt wurde, das ich für die Feuerwehr geschrieben habe. In dieser Ausgabe sind weitere Teile der Anwendung dran:

- Plugins für Eingabe- und Ausgabe-Geräte
- Ein Feuermelder für die Anwendung
- Abspielen der Sirene

Im Moment plane ich nur zwei Plugins für Ein- und Ausgabe-Geräte. Eins für einen Feuermelder als Eingabe-Gerät und das zweite Plugin soll für die Sirene zuständig sein. Ich plane das dennoch als Plugins, um in Zukunft schneller neue Geräte einbinden zu können. Ich könnte mir da z.B. weitere Melderarten vorstellen oder eine Mini-Löschanlage.

Bevor ich die zwei Plugins zeige, möchte ich kurz zeigen, wie der Plugin-Mechanismus umgesetzt ist. In dieser Anwendung müssen zwei Schritte unternommen werden, um ein Plugin einzubinden:

1. Eintragen des Plugins in der Konfiguration
2. Speichern des Plugins unter `<APP_HOME>/Plugins`

Das nicht automatisch alles aus dem Plugins-Verzeichnis geladen wird, hängt damit zusammen, dass unter Umständen einzelne Plugins gar nicht angesteuert werden sollen. Außerdem wird so vermieden, dass man zwei Plugins-Ordner braucht (einen für Ein- der andere für Ausgabe-Geräte).

Die Konfiguration ist mit YAML umgesetzt und ist sehr einfach - siehe Listing 1.

Ich denke, die `type`-Angabe ist eindeutig. Unter `module` ist der Name des Moduls einzutragen. In diesem Fall existieren also `Feuermelder.pm` und `Sirene.pm`.

Für die Buttons im Bedienfeld sind "Events" festgelegt und auch Input-Plugins können "Events" auslösen. Diese müssen dann fest einprogrammiert sein. Die Output-Plugins reagieren entsprechend auf "Events". Diese sind unter `event` in der YAML-Datei eingetragen.

Wird ein Button gedrückt oder ein angeschlossenes Gerät löst einen Event aus, so ruft das Programm alle Plugins auf, die für dieses Event registriert sind. Das ganze passiert in einem Event-Handler (siehe Listing 2).

Der Feuermelder

Der Rauchmelder ist mit einem Arduino-Board gebaut. Damit ist es auch für Technik-Einsteiger sehr einfach, (fast) beliebige Geräte umzusetzen. Außerdem bietet das Arduino-Board die Möglichkeit, das Gerät direkt per USB an den PC anzuschließen. Und es gibt Erweiterungen, so dass man das Gerät auch per Netzwerk ansteuern kann.

Der Einfachheit halber, setze ich hier die Verbindung per USB ein. Der Feuermelder ist sehr simpel gemacht: Ein Druckknopf, bei dessen Betätigung eine rote LED angeht.

```
---
plugins:
  - type: input
    module: Feuermelder
    active: 1
    label: Feuermelder
  - type: output
    module: Sirene
    event: AlarmStart|SoundOff
    sound_file: $BASE/files/sirene.wav
    active: 0
    label: Sirene
```

Listing 1

```
sub run {
    my ($self, %args) = @_;

    my $event = $args{event} || 'none';

    my $logger = Log::Log4perl->get_logger;
    $logger->debug( "run plugins for event $event" );

    for my $plugin ( $self->plugins( $event ) ) {
        my $file = $plugin->{file};

        eval {

            $logger->debug( "require $file" );

            require $file;

            my $object = $plugin->{package}->new( %{$plugin} );
            $object->run( %args );
            1;
        } or $logger->debug( "Could not load $file: $@" );
    }
}

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

    my @plugins = $self->{__BMA}->plugins->get_type( 'output' );
    @plugins = grep{ '|' . $_->{event} . '|' =~ /\|\Q$event\E\|/ }@plugins;

    return @plugins;
}
```

Listing 2

Arduino

Arduino ist eine OpenSource Physical-Computing-Plattform, mit der es auch für Einsteiger möglich ist, mit Mikrocontrollern zu experimentieren. Man muss nicht Löten, da alles mit Steckteilen zu machen ist. Es gibt auch verschiedene Erweiterungen, so dass man das Board nicht nur per USB mit dem Rechner verbinden kann, sondern auch per Ethernet.

Processing

Das Arduino-Board kann man mit Processing, einer Java-basierenden Sprache programmieren. Entwicklungsumgebungen gibt es für alle gängigen Betriebssysteme.

Sketches

Sketches sind die Programme für das Arduino-Board. Diese kann man in der IDE schreiben und auf Knopfdruck auf das Arduino hochladen. Das kann einige Sekunden in Anspruch nehmen.

```
void loop(){
  // read the state of the pushbutton value:
  buttonState = digitalRead(buttonPin);

  // check if the pushbutton is pressed.
  // if it is, the buttonState is HIGH:
  if (buttonState == HIGH) {
    // turn LED on:
    digitalWrite(ledPin, HIGH);
    // send "1" to PC
    Serial.println(1);
  }
  else {
    // turn LED off:
    digitalWrite(ledPin, LOW);
    // send "0" to PC
    Serial.println(0);
  }
}
```

```
my $conn = Win32::SerialPort->new( $port ) or die $^E;

$conn->databits( $data );
$conn->baudrate( $baud );
$conn->parity( $parity );
$conn->stopbits( $stop );

my $is_alarm     = 0;

while ( 1 ) {
    my $char = $conn->lookfor;

    no warnings 'numeric';

    if ( defined $char and $char and int( $char ) == 1 ) {
        if ( !$is_alarm and $can_alarm ) {
            $is_alarm = 1;

            $$text = 'start_alarm';

            my $thread_event = Wx::PlThreadEvent->new(-1, $$done_event, $$text);
            Wx::PostEvent($handler, $thread_event);
        }
    }

    $conn->lookclear;
}
```

Listing 3

Neben der Steuerung der LEDs schickt das Arduino-Board auch ein Signal an den PC. Dieses wird mit Perl und dem Modul `Win32::SerialPort` realisiert. Das Plugin reagiert auf das Signal des Melders und startet das Szenario. Durch die Verwendung von `Win32::SerialPort` ist das Plugin vorerst nur unter Windows lauffähig, aber die Erweiterung, dass je nach Betriebssystem `Win32::SerialPort` oder `Device::SerialPort` genommen wird, ist nicht wirklich aufwändig (Listing 3).

Eine Schwierigkeit mit solchen Eingabe-Plugins besteht darin, dass parallel zum "normalen" Ablauf auch auf einkommende Ereignisse geachtet werden muss. Hier kommen dann Threads ins Spiel. Threads und graphische Oberflächen ist keine ganz simple Kombination. Aber ähnlich wie bei `Perl/Tk` kann man auch bei `WxPerl` mit `threads` arbeiten.

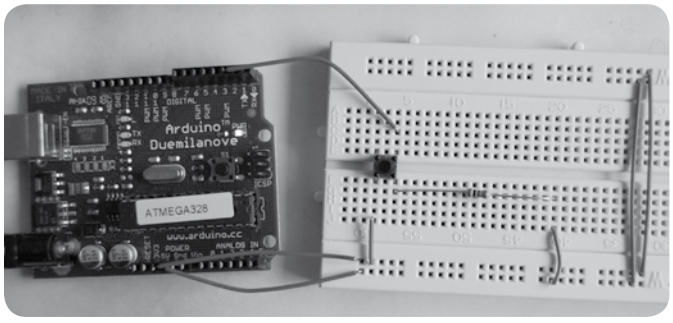


Abbildung 1: Arduino "Feuermelder"

Man muss `threads` und `threads::shared` vor `Wx` einbinden. Die "Kommunikation" zwischen den Threads und der Wx-Oberfläche geschieht über Variablen, die ge"shared" sind und Events, die in den Threads ausgelöst werden.

Für das Erzeugen der Threads und Abarbeiten der Events ist ein extra TaskManager zuständig. Bevor ich die Oberfläche zusammenbaue, wird dieser TaskManager instanziert (Listing 4).

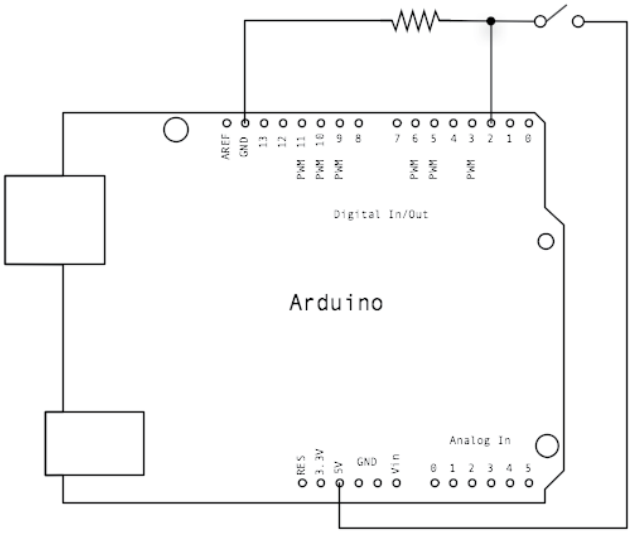


Abbildung 2: schematisches Schaltbild

```
my @input_plugins = $self->plugins->get_type( 'input', 1 );
my $task_manager  = BMA::TaskManager->new(
    BMA => $self,
);
$task_manager->run_tasks(
    @input_plugins,
);
```

Listing 4

```
sub run_tasks {
    my ($self, @plugins) = @_;

    for my $plugin ( @plugins ) {
        my $file = $plugin->{file};
        my $object;

        eval {
            # lade Modul und erzeuge Object
        } or next;

        my $sub = $plugin->{package}->can( 'run' );

        my $task = threads->create(
            $sub, $object, $self->{BMA}, \$SERVICE_POLL_EVENT, \$PULL_TEXT
        );

        push @{ $self->{workers} }, $task;
    }

    return 1;
}
```

Listing 5

```
sub run {
    my ($self,$handler,$done_event,$text) = @_;
    sleep 5; # hier steht im Plugin der Code, um den Arduino abzufragen

    $$text = 'start_alarm';

    my $thread_event = Wx::PlThreadEvent->new(-1, $$done_event, $$text);
    Wx::PostEvent($handler, $thread_event);
}
```

Listing 6

In der Methode `run_tasks` werden dann die Threads erzeugt (Listing 5).

Jedes Plugin wird geladen und instanziert. Im Thread wird dann die `run`-Methode des Objekts ausgeführt. Im TaskManager gibt es zwei Variablen, die mittels `threads::shared` in allen Threads zur Verfügung stehen: `$SERVICE_POLL_EVENT` und `$PULL_TEXT`. Über diese Variablen und extra Events kann das Plugin dann mit der Anwendung kommunizieren (Listing 6)

In `$text`, das eine Referenz auf `$PULL_TEXT` ist, wird der Methodenname für die WxPerl-Oberfläche gespeichert. Danach wird ein Event erzeugt (`Wx::PlThreadEvent->new`) und danach gefeuert (`Wx::PostEvent`).

Dadurch bekommt die Oberfläche mit, dass ein Input-Plugin etwas tun möchte. Bei den Events wird in meinem Programm die Methode `on_service_poll_event` ausgeführt. Dort wird dann geprüft, ob die Oberfläche die Methode kennt, die das Plugin gerne ausgeführt hätte. Ist sie vorhanden, wird sie ausgeführt (Listing 7).

```
sub on_service_poll_event {
    my ($self) = @_;
    my $sub = $self->can( $PULL_TEXT );

    $self->$sub() if $sub;
}
```

Listing 7

Herbert Breunung wird in einer der nächsten Folgen des WxPerl-Tutorials genauer auf Threads eingehen.

Die Sirene

Sobald ein Melder ausgelöst wird, wird eine Sirene ausgelöst.

Der Ton macht die Musik

Da die Anwendung erstmal für Windows gedacht ist, ist das Abspielen von .wav-Dateien denkbar einfach. Für diese Aufgabe hat das CPAN mal wieder einiges zu bieten. Ich benutze hier `Win32::Sound`.

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

    Win32::Sound::Volume( '100%' );
    Win32::Sound::Play( $self->sound_file,
                                SND_ASYNC );
}
```

Listing 8

Ebenso einfach ist es, jegliche Sound-Ausgabe zu stoppen:

```
sub stop_sound {
    Win32::Sound::Stop();
}
```

Listing 9

Wenn die Anwendung soweit stabil ist, werde ich mir auch anschauen, wie so ein Plugin unter Linux aussehen könnte.

Alarm! Alarm!

Damit die Sirene ausgelöst werden kann, muss noch das entsprechende Event getriggert werden. An der Stelle, an der ein Alarm beginnt wurde ein

```
$self->event(
    Event => 'StartAlarm',
);
```

Listing 10

eingefügt. Die `event`-Methode ruft nur die `run`-Methode auf und übergibt noch zusätzliche Standard-Parameter wie z.B. das Objekt der Anwendung und der Name des Events. Im Fall des Sirenen-Plugins reagiert das Plugin auf mehrere Events. Und jedesmal unterschiedlich. In der `run`-Methode muss dann also unterschieden werden, welches Event ausgelöst wurde und dann entsprechend reagieren (Listing 11).

Und fertig ist der BMA-Trainer. Das ganze ist OpenSource und auf Github (http://github.com/reneeb/BMATrainer) zu finden. Ich hoffe, dass noch mehr Feuerwehren damit etwas anfangen können.

Renée Bäcker

```
sub run {
    my ($self,%args) = @_;

    return unless $args{event};

    if ( $args{event} eq 'AlarmStart' ) {
        $self->logger->debug( 'starte alarm' );
        $self->start_sound;
    }
    elsif ( $args{event} eq 'SoundOff' ) {
        $self->logger->debug( 'schalte alarm aus!' );
        $self->stop_sound;
    }
}
```

Listing 11

XMLSocket-Klasse in Verbindung mit Perl-Sockets

Mit Perl ist viel mehr möglich als "nur" dynamische Webseiten zu erstellen, auch auf die Socket-API von C lässt sich damit zugreifen und in Verbindung mit Flash als Client lassen sich damit Chats und sogar Multiplayer-Spiele realisieren. Was ist ein Socket? Um es kurz zu machen: ein Socket ist eine Schnittstelle in einem Netzwerk, die es ermöglicht Daten beinahe in Echtzeit zu übertragen. Dabei fungiert ein Perl-Script als zentraler Server, lauscht auf Verbindungen und eingehende Daten, parst die Daten und sendet sie bei Bedarf weiter an andere Computer, die auch mit dem Perl-Server verbunden sind.

Zum Vergleich: ein Apache-Webserver macht eigentlich dasselbe. Er lauscht auf so genannte HTTP-Requests, die wir senden, wenn wir einen Link anklicken. Der Webserver parst die Daten und liefert uns die gewollte Seite aus. Der vorteilhafte Unterschied zwischen einem Apache-Webserver und einem Perl-Socket-Server liegt aber darin, dass es nicht ganze HTML-Seiten sind, die über die Leitungen fließen, sondern schlanke Strings.

Kleines Beispiel: Wir klicken einen Button in Flash an, auf dem steht: Zeig mir die Server-Zeit. Flash sendet durch einen (TCP-)Socket den String "[aktion:serverzeit]\0" an den Perl-Socket-Server. Das "\0" fügt der Flash-Player übrigens automatisch hinzu, um damit das Ende des Strings zu signalisieren. Der Perl-Server parst den String und findet heraus: aha, ich soll die Serverzeit ausliefern. Als Antwort schickt er nun den String "[serverzeit:12:32:54]\0".

Es sei an dieser Stelle auch gleich erwähnt, dass die XMLSocket-Klasse nicht zwingend Strings im XML-Format verlangt. Und das ist auch gut so, denn ein

```
"<xml version="1.0">
   <aktion>serverzeit</aktion>
 </xml>"
```

im Vergleich zu "[aktion:serverzeit]" ist ganz klar bandbreitenschonender, weil man einfach viel weniger Daten senden muss. Natürlich hätte ich auch "{aktion}{serverzeit}" schreiben können oder auch "[a][s]" oder gar "[s]". Welches Protokoll man auch verwendet, solange man es selbst versteht und solange es logisch ist, passt alles.

Perl, der Meister der Regular-Expressions. Damit wird es sehr einfach, die Daten zu parsen.

```
$flashDaten =~ /\[aktion:(\w+)\]/;
my $aktion = $1;

if ($aktion eq "serverzeit") {
    my ($sec,$min,$hour) = localtime(time);
    my $antwort =
          "[serverzeit:$hour:$min:$sec]\0";
    sende_zurück($antwort);
}
```

Ok, ich glaube man hat mich verstanden: Sockets dienen hauptsächlich dazu möglichst kleine Daten möglichst schnell zu senden/empfangen, und bieten damit die Möglichkeit viele Clients gleichzeitig zu bedienen. Man stelle sich einen Chat vor auf dem sich gerade hunderte User unterhalten. Die Nachrichten müssen so schnell wie möglich weitergeleitet werden, um überhaupt einen Chat möglich zu machen. Es reicht nicht, die Nachrichten in einer Datenbank zu speichern und alle paar Sekunden mittels eines HTTP-Requests die Seite zu aktualisieren. Es gibt zwar solche Chats, z.B. mit PHP, MySQL und Javascript (für automatische HTTP-Requests) realisiert – aber die taugen doch nicht wirklich was; und zwingen mit steigender Useranzahl selbst die stärksten Server in die Knie.

Ein ganz nützliches Perl-Modul, mit dem die Arbeit mit Sockets zum Kinderspiel wird, ist das POE-Modul. POE heisst Perl Object Environment, es ist objektorientiert und dadurch ist die Syntax auch leicht verständlich.

```
my $xml = '<?xml version="1.0"?>
    <!DOCTYPE cross-domain-policy SYSTEM "/xml/dtds/cross-domain-policy.dtd">
        <cross-domain-policy><site-control permitted-cross-domain-policies="master-only"/>
        <allow-access-from domain="localhost" to-ports="5006" /></cross-domain-policy>';
```

Listing 1

Was ist POE? POE ist ein Framework, welches es wie ein Betriebssystem erlaubt, mehrere Prozesse in einem Perl-Skript gleichzeitig laufen zu lassen, über sogenannte Sessions. Es lassen sich damit wunderbare Programme schreiben, in denen verschiedene Sessions sich um verschiedene Aufgaben kümmern. Da es für viele Bereiche – vor allem im Netzwerk-Bereich – vorgefertigte POE-Module gibt, schreibt sich ein Socket-Server damit fast schon von alleine. Die Hauptarbeit überlassen wir einfach POE und können uns direkt um die lustigen Sachen kümmern wie die Logik unseres Servers, Datenparsen usw.

Ladet euch das POE-Framework von CPAN herunter und installiert es. Falls ihr keinen eigenen Server habt, sondern einen Webspace ohne Root-Zugang, kein Problem. Erstellt einfach einen Ordner auf dem Webspace, ladet dort das POE-Modul hoch und inkludiert es anstatt einfach nur:

```
use POE;
```

mit...

```
use lib "absoluter/pfad/zum/POE-Ordner/lib";
use POE;
```

Nun aber endlich mal zu einem Beispiel. Ich zeige euch einen kompletten Perl-Socket-Server, der eigentlich zwei Server beinhaltet: der eigentliche Hauptserver kümmert sich um den Flash-Film, der andere Server – nennen wir ihn Policy-File-Server – kümmert sich um die Auslieferung einer virtuellen XML-Datei, dem sogenannten Policy-File. Policy-File, was ist das?

Nun ja, machen wir es kurz: Adobe verlangt aus Sicherheitsgründen die Auslieferung eines virtuellen Policy-Files, quasi eine Datei in der geschrieben steht, dass der Flash-Film dazu berechtigt ist auf bestimmte Ports des Servers zugreifen zu dürfen.

Diese Datei liegt in unserem Perl-Skript in einem String und wartet nur darauf, in die große weite Welt verschickt zu werden. Sie sieht, bei mir zumindest, so aus, wie in Listing 1 dargestellt.

Wie man sieht, steht darin geschrieben, dass localhost auf den Flash-Film zugreifen darf, und der Flash-Film auf den Server auf Port 5006.

Auf Port 5005 lauscht ein zweiter Server, also eine zweite POE-Session, die sich um nichts anderes kümmert als auf eine Verbindung und einen String "<policy-file-request/>\0". Schickt ein Flash-Film – oder wer auch immer – solch einen String, bekommt er als Antwort die $xml-Datei.

Hier mal eine kleine Grafik - siehe Abbildung 1.

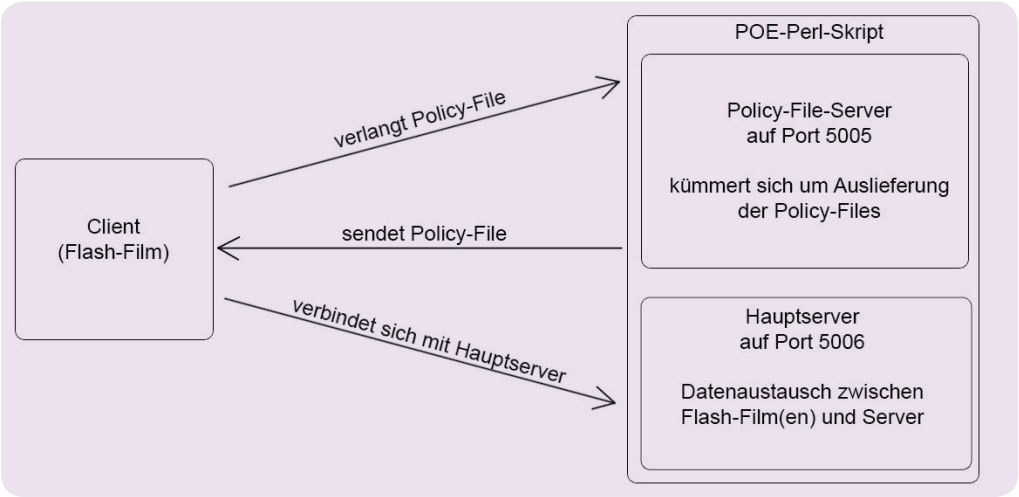


Abbildung 1: Ablauf Kommunikation zwischen Client und Server

Und nun der komplette Code für den Socket-Server - siehe Listing 2.

```
#!/usr/bin/perl

use warnings;
use strict;
use POSIX;
use IO::Socket;
#use lib "/var/www/cgi-bin/POE-1.007/lib";
use lib "c:/strawberry/perl/bin/POE-1.007/lib";
use POE;

my %inbuffer  = ();
my %outbuffer = ();
my @connections = ();

my $xml = '<?xml version="1.0"?>
    <!DOCTYPE cross-domain-policy SYSTEM "/xml/dtds/cross-domain-policy.dtd">
        <cross-domain-policy><site-control permitted-cross-domain-policies="master-only"/>
        <allow-access-from domain="localhost" to-ports="5006" /></cross-domain-policy>';


POE::Session->create(
    inline_states => {
        _start           => \&main_server_start,
        event_accept     => \&main_server_accept,
        event_read             => \&client_read,
        event_write            => \&client_write,
        event_disconnect => \&client_disconnect
    }
);

POE::Session->create(
    inline_states       => {
    _start              => \&policy_server_start,
    policy_accept       => \&policy_server_accept,
    policy_read         => \&policy_server_read
    }
);

POE::Kernel->run();

sub main_server_start {
    print "Starte Hauptserver...\n";
    my $server = IO::Socket::INET->new(
        LocalPort => 5006,
        Listen    => 10,
         Reuse     => "yes",
    ) or die "can't make server socket: $!\n";
    $_[KERNEL]->select_read($server, "event_accept");
}

sub main_server_accept {
    my ($kernel, $server) = @_[KERNEL, ARG0];
    my $new_client = $server->accept();
    print "Neuer Client auf Hauptserver wurde akzeptiert!\n";
    push(@connections, $new_client);
    $kernel->select_read($new_client, "event_read");
}

sub client_read {
    my ($kernel, $client) = @_[KERNEL, ARG0];
    my $rv = $client->recv(my $data, POSIX::BUFSIZ, 0);

    unless (defined($rv) and length($data)) {
        $kernel->yield(event_disconnect => $client);
        return;
    }
    $inbuffer{$client} .= $data;

    while ($inbuffer{$client} =~ s/(.*\0)//) {
        my $buff = $1;
        print "Dateneingang von Client: $buff\n";

        foreach my $user (@connections) {
        if ($user ne $client) {
        $outbuffer{$user} .= $buff;
        $kernel->select_write($user, "event_write");
        }
        }
    }
}
```

Listing 2

```
sub client_write {
    my ($kernel, $client) = @_[KERNEL, ARG0];

    unless (exists $outbuffer{$client}) {
        $kernel->select_write($client);
        return;
    }

    my $rv = $client->send($outbuffer{$client}, 0);
    print "Daten an Client gesendet: $outbuffer{$client}\n";

    unless (defined $rv) {
        return;
    }

    if ( $rv == length($outbuffer{$client}) or $! == POSIX::EWOULDBLOCK) {
        substr($outbuffer{$client}, 0, $rv) = "";
        delete $outbuffer{$client} unless length $outbuffer{$client};
        return;
    }
    $kernel->yield(event_disconnect => $client);
}


sub client_disconnect {
    my ($kernel, $client) = @_[KERNEL, ARG0];
    my @connectionsTemp = ();

    foreach my $user (@connections) {
        if ($user ne $client) {
            push(@connectionsTemp, $user);
        }
    }

    @connections = @connectionsTemp;

    delete $inbuffer{$client};
    delete $outbuffer{$client};
    $kernel->select($client);
    close $client;
    print "Client auf Hauptserver hat die Verbindung beendet.\n";
}

sub policy_server_start {
    print "Start Policy-File Server...\n";
    my $server = IO::Socket::INET->new(
        LocalPort => 5005,
        Listen    => 1,
        Reuse     => "yes",
    ) or die "can't make server socket: $!\n";
    $_[KERNEL]->select_read($server, "policy_accept");
}

sub policy_server_accept {
    my ($kernel, $server) = @_[KERNEL, ARG0];
    my $new_client = $server->accept();
    print "Neuer Client auf Policy-File-Server akzeptiert!\n";
    $kernel->select_read($new_client, "policy_read");
}

sub policy_server_read {
    my ($kernel, $client) = @_[KERNEL, ARG0];
    $client->recv(my $data, POSIX::BUFSIZ, 0);
    if ($data eq "<policy-file-request/>\0") {
        print "Policy-File-Request von Client... Sende virtuelles Policy-File!\n";
        $client->send($xml,0);
    }
    $kernel->select($client);
    close $client;
    print "Client auf Policy-File-Server hat die Verbindung beendet.\n";
}
```

Fortsetzung - Listing 3

Ganz kurz, was der Hauptserver macht: Er lauscht auf Verbindungen, akzeptiert diese, schreibt die Verbindungen oder besser gesagt die Referenzen dazu in das Array @connections, sendet eingehende Daten an die Verbindungen (im Array @connections, ausser an den Sender), löscht beendete Verbindungen wieder aus @connections (das ist wichtig, damit der Server nicht versucht Daten an bereits beendete Verbindungen zu senden – was zu einem sehr unschönen Programmabbruch führen würde). Daten parsen tut er nicht, aber falls er das tun soll, könnte es etwa so aussehen wie schon oben erwähnt, mit Regular Expressions:

```
$buff =~ /\[aktion:(\w+)\]/;
my $aktion = $1;

if ($aktion eq "serverzeit") {
    my ($sec,$min,$hour) = localtime(time);
    my $anwort =
          "[serverzeit:$hour:$min:$sec]\0";
    $outbuffer{$client} .= $antwort;
    $kernel->select_write(
              $client, "event_write");
}
```

Nun zum Flash-Teil. Wir erstellen nun einen Flash-Film, der einfach ein blaues Viereck zeigt. Dieses Viereck lässt sich verschieben, und wenn man es verschoben hat, sendet es die neuen Koordinaten des Vierecks an die anderen Flash-Filme, die noch mit dem Server verbunden sind. Die anderen Flash-Filme platzieren das Viereck dann an der neuen Position.

Ich zeige erstmal den Actionscript-Code. Es ist übrigens möglich den Code via der freien Flex-SDK zu kompilieren, falls man kein Flash besitzt. Und das schöne an der Flex-SDK ist – außer dass sie nichts kostet – dass man sie auch unter Linux einsetzen kann. Da POE unter Windows nur zu Versuchszwecken taugt (POE verursacht zumindest beim strawberry-Perl unter Windows nach kurzer Zeit eine CPU-Auslastung von 100%) ist das natürlich sehr praktisch. Falls ihr an der Flex-SDK interessiert seit, benutzt Google, ist leicht zu finden.

Erstellt in Flash einen neuen Film, speichert ihn und gebt in den Eigenschaften als Dokumentklasse TestSocket an. Mit TestSocket ist die Datei TestSocket.as (im Code-Download zu dieser Ausgabe enthalten) gemeint, die ihr zusammen mit dem neuen Flash-Film in denselben Ordner kopiert. Oder kopiert die AS-Datei in den bin-Ordner der Flex-SDK und kompiliert den Code mittles Eingabeaufforderung bzw. Terminal dort mit mxmlc TestSocket.as. Danach bindet ihr die SWF-Datei in eine HTML-Seite ein und ladet diese zusammen mit der SWF-Datei auf euren Webspace bzw. xampp bzw. /var/www/ oder wo auch immer ihr eure Webseiten liegen habt hoch.

```
public class TestSocket extends Sprite {

    private var socket:XMLSocket =
       new XMLSocket();
    private var host:String = "localhost";
    private var secPort:uint = 5005;
    private var mainPort:uint = 5006;

    private var infoBox:TextField =
       new TextField;
    private var dragBox:Sprite =
       new Sprite;
    private var stageRect:Rectangle;

    private var regExp:RegExp =
       new RegExp(
         /\[x:(\d+)\]\[y:(\d+)\]/
       );

    public function TestSocket() {
        addChild(infoBox);
        infoBox.x = infoBox.y = 0;
        infoBox.width = stage.stageWidth;
        infoBox.height = stage.stageHeight;
        infoBox.text = "Die Verbindung zum"
           + " Server wird hergestellt...";

        Security.loadPolicyFile(
    "xmlsocket://" + host + ":" + secPort);

       configureListeners(socket);
       socket.connect(host, mainPort);
    }

    // weitere Methodendeklarationen
}
```

Startet nun den Server mittels Terminal oder Eingabeaufforderung und ruft über den Browser die HTML-Seite mit dem Flash Film auf. Wenn alles geklappt hat, macht weitere Fenster oder Tabs auf und verschiebt das Viereck. Es müsste nun auf allen anderen Seiten ebenfalls verschoben worden sein.

Ach ja, ein besonderes Schmankerl für die eigene Webseite ist es natürlich einen unsichtbaren Flash-Film laufen zu lassen, der eine Socket-Verbindung herstellt. So ist ein Datenaustausch auch ohne Seitenreload möglich. Dabei kann man Javascript als Brücke verwenden, um so empfangene Daten direkt im Browser auszugeben bzw. anzuzeigen. Man stelle sich einfach vor, dass ein User namens "Hans" dem User mit der Uid 12 eine Nachricht schreibt, ganz normal über die Webseite. Nun wird eine Funktion in Javascript aufgerufen, die dem Flash-Film sagt, er soll eine Nachricht an den Server schicken, z.B. "[post][12][Hans]". Der Server parst diesen String und findet heraus: aha, ich soll dem User 12 Bescheid sagen, dass er soeben Post von Hans bekommen hat. Der Server

schickt an den Socket des Users 12 den String "[post][Hans]". Nun erscheint am rechten unteren Bildschirmrand des Postempfängers – der ja noch nichts von seinem Glück weiß, weil er die Seite noch nicht aktualisiert hat – eine kleine schicke Box, in der steht: "Sie haben Post von Hans!". Posteingang-Benachrichtigung ohne Seitenreload beinahe in Echtzeit, realisiert mit einem TCP-Socket.

Sehr vorteilhaft ist es natürlich diesen unsichtbaren Flash-Film in einer Hauptseite laufen zu lassen, in der die eigentlichen Webseiten in einem iframe dargestellt werden. Warum? Weil sonst bei jedem neuen Seitenaufruf die Verbindung unterbrochen und dann wieder hergestellt wird.

Es gibt übrigens eine Javascript-Klasse, die diese Brücke herstellt, gebt mal in einer Suchmaschine "JSXML Bridge Javascript Flash" ein. Diese Klasse arbeitet mit der ExternalInterface-Klasse von Flash.

Zuguterletzt noch zwei Punkte:

• Warum muss ich jeden String, den ich vom Perl-Socket-Server an einen Flash-Client schicke mit einem "\0" beenden? Weil Flash sonst ewig auf das Ende der Nachricht warten würde.

• Kann jedes Programm, dass Socketverbindungen möglich macht, von jedem anderen Menschen dazu benutzt werden meine auf Port 5005 und 5006 lauschenden Server mit Nachrichten geradezu zu überfluten? Und ob. Stichwort Flooding. Leider können die Server nicht erkennen, ob Daten von einem Flash-Film, einem C++-Programm, einem anderen Perl-Skript oder von sonstwo geschickt werden. Hier ist es wirklich sehr wichtig, sich sicherheitstechnisch abzusichern, will man nicht von seinem Webhoster gesperrt werden. Ich z.B. habe es so realisiert, dass nachdem sich ein User in meine Webseite eingeloggt hat seine IP-Adresse über einen UNIX-Socket an das POE-Skript gesendet wird; diese IP-Adresse wird für ein paar Sekunden erlaubt, d.h. Nur IP-Adressen werden gestattet, die praktisch "von oben" ein okay bekommen haben, und das nur für ein paar Sekunden. Jede andere IP-Adresse wird gar nicht erst reingelassen bzw. sofort wieder rausgeworfen. "Tut mir Leid, sie stehen nicht auf der Gästeliste!". Außerdem muss der Client auch seine Session-ID mitschicken, und es wird geprüft, ob diese in der Datenbank unter den eingeloggten Usern steht. Wenn nicht, auf Wiedersehen.

Okay, und das ist auch das Stichwort für mich, ich muss weg. Ich hoffe ich konnte dem einen oder anderen einen groben Einblick in die Welt der Sockets bieten und wünsche euch viel Spaß beim Erstellen von coolen Flash-Perl-Applikationen.

Markus Schaffer

Perl 6 - Der Himmel für Programmierer - Update 3

Mit dem Erscheinen von Rakudo * gibt es wieder einen Anlass für einen Rundblick über die Perl 6-Landschaft. Vieles geschah im letzten dreiviertel Jahr, seit dem letzten Update. Wer sich in kürzeren Abständen informieren mag: Auf lithology.blogspot.com gab es wieder wöchentliche Zusammenfassungen der Entwicklung von Sprache, Parrot und Rakudo. Der nächste *summariser* wird wieder gesucht.

Die neue Übersicht

perl6-projects.org ist nach *perl6.org* umgezogen und ist nun sogar von *dev.perl.org* aus verlinkt. Somit wurde es ein offizieller Anlaufpunkt zum Thema Perl 6 und bietet eine schnelle Übersicht zu allem derzeit Nennenswertem. Daniel Wright gab freundlicherweise die Adresse her und es kamen neue Inhalte hinzu, vor allem über die Implementationen, die kommenden Perl 6-Bücher und Mailinglisten. Die Autoren sind auf http://perl6.org/about/ zu finden, von denen Moritz Lenz (Programmierung) hervorzuheben ist und Susanne Schmidt (Su-Shee), die der Seite ein neues, schickes und buntes Aussehen gab. In der rechten, oberen Ecke ist dort der Schmetterling Camelia zu sehen, in dessen Flügeln man ein P und eine 6 entdecken kann. Er ist das Maskottchen der Perl 6-Sprache, nennt sich auch "spokesbug" und wird in ASCII »ö« geschrieben.

Perl 6 Wikis

Meine eigenen Pläne, november-wiki.org zu einer ähnlichen Anlaufstelle wie perl6.org zu machen, sind nicht weit gediehen. Die Software hatte immer wieder Fehler. Deshalb war ich vor allem als Hausmeister der offiziellen Perl 6-Wiki unterwegs. Dort hat Conrad Schneiker, der die 1000$ für November auslobte, beträchtliches an Inhalten geschrieben. Allerdings ist sie mit der Zeit zugewuchert, sodass es nötig wurde, in einer mehrtägigen Aktion vieles umzuschreiben und neu zu verknüpften. Dabei half mir Martin Berends der nebenbei mit anderen Parrot 2.1.1 und Rakudo #26 veröffentlichte. Dennoch breche ich den Ausbau der November-Inhalte nicht ab, da es auch Rakudo hilft.

Auch die Perl 6-Tafeln auf http://wiki.perl-community.de/Wissensbasis/Perl6Tafel werden immernoch, wenn auch langsam, in deutsch und englisch weitergeschrieben. Deren Anhang A wird bereits vom Dokumentationswerkzeug u4x (userdocs for x-mas) verwendet. Es ist mit *grok* vergleichbar, richtet sich aber an Rakudonutzer.

Die Übersetzung des achtteiligen Perl 6-Tutorials, welches hier in der $foo erschien nach http://www.perlfoundation.org/perl6/index.cgi?perl_6_tutorial ist auch so ein Ziel von mir, entscheidend ist jedoch, was die Programmierer tun.

Stand der Implementationen

Pugs und Elf sind derzeit ruhig, auch Mildew regt sich leider kaum. Da fällt auf: Ich versäumte bisher *gimme5* zu erwähnen, was sicher nicht Larry Wall's Wunsch ist, ihm 5 Finger entgegen zu strecken, sondern ein Übersetzer von Perl 6 nach Perl 5. Natürlich übersetzt es nicht den ganzen, riesigen Sprachumfang, sondern nur etwas von den *Regex*, *grammars* und etwas mehr um aus der `STD.pm` *viv* zu erzeugen. Dies ist ein Perlprogramm, welches den vollständigen Sprachumfang zumindest einlesen und in seine logischen Einheiten

zerlegen kann. Damit löste Larry sein Huhn-Ei-Problem, als er Perl 6 in Perl 6 definieren wollte. Da die `STD.pm` die maßgebliche Perl 6-Spezifikation ist und die Synopsen nur eine für Menschen lesbare Version, ist es für Interpreterautoren verlockend, die `STD.pm` oder *viv* gleich als Parser zu verwenden. Doch die Komplexität und Ansprüche der `STD.pm` verhinderten bisher, dass Rakudo sie einsetzt. Mildew allerdings verwendet die `STD.pm`.

Ebenso versäumte ich Sprixel zu erwähnen, den Matthew Wilson (diakopter) vor allem aus Gründen mangelnder Geschwindigkeit entwarf. Der Name ist ein Anagramm aus "Perl six". Er verwendet Larry's STD.pm als Parser, der den Quellcode in eine JSON ähnliche Datenstruktur umwandelt, die letztlich von Googols V8 im Browser ausgeführt wird. Derzeit schreibt er an einem neuen "Backend" in C# für Microsoft's CLR. Sprixel kennt nur einen sehr kleinen Befehlssatz und wenige Mitwirkende, erreicht aber eine beachtliche Geschwindigkeit. Einer der Mitentwickler (Martin Berends) griff die Idee auf und schuf unter dem Arbeitsnamen *vill* eine Brücke von STD zur *LLVM*. Dies ist eine freie VM die auch Compiler-Werkzeuge beinhaltet. Damit können Perl 6-Scripte tatsächlich zu eigenständig lauffähigen Programmen kompiliert werden. Die Programme sind aber weit weniger mächtig als diejenigen für Rakudo. Denn mithilfe der `STD.pm` erkennt *vill* den ganzen Sprachumfang, kann aber nur einen winzigen Bruchteil in LLVM-Assembler übersetzen.

Eine weiterer Interpreter jenseits von Parrot feierte am 21. Januar die Version 3.0: Perlito von Flavio S. Glock, dass ich damals als MiniPerl6 vorstellte, versteht auch nur einen Teil von Perl 6. Es hat aber nicht den Anspruch alles zu können, sondern möchte eine Infrastruktur bieten, die späteren Implementationen helfen soll. So baut bereits kp6 (kinda perl 6) auf Perlito auf.

Auf http://www.perlito.org/js/ kann man gut nachvollziehen, wie Quellcode in die einzelnen Zwischenstufen umgewandelt und endlich ausgeführt wird. Um Perlito zu benutzen benötigt man mindestens Perl 5 und eventuell SBCL Lisp, wenn man sein Perl 6 zu Lisp und nicht Perl 5 kompiliert hat. Andere *backends* können auch Javascript, JavaVM und Googles "neue" Erfindung Go sein. Ab Version 4.0 (20.März) ist Perlito unter Perl wesentlich schneller (30%) geworden und kennt jetzt auch `eval`.

Parrot

Einen Tag vor MiniPerl 3.0 kam Parrot 2.0 heraus. Anders als im letzten Update angekündigt war das kein Zeichen für große Änderungen in der API. Große Versionen von Parrot sollen jetzt jedes Jahr kommen, damit die Entwickler von Hochsprachen und Distributionen sicherer planen können. Parrots Subsysteme sind soweit vollendet, wenn auch in unterschiedlicher Qualität. Das letztens vermisste System für parallele Aufführung steht jetzt, das für Sicherheit ist derzeit im Aufbau.

Auch ein Paketarchiv namens *Plumage*, das Geoffrey Broadwell begann, entsteht. Es ist nicht ganz, was man von der `CPAN.pm` her kennt, denn hier gibt es keine FTP-Proxy und die Quellen werden direkt aus den Git- und SVN-Archiven der Projekte gezogen. Aber so lassen sich alle angegliederten Vorhaben dezentral organisieren, wie zum Beispiel die Hochsprachcompiler, die ja nicht eigentlich zu Parrot gehören. Jeder soll ja die Möglichkeit haben, eigene Parrotsoftware in selbstgewählten Archivinfrastrukturen zu schreiben. Prominentestes Plumage-Modul ist vielleicht Blizkost, ein verpackter Perl 5.10-Interpreter, der gut möglich einen soliden Weg abgibt, Perl 5 von Rakudo aus aufzurufen.

Im Ganzen vollzieht sich Parrots Fortschritt in vielen Details. Beispiele: Jonathan Leto schraubt unter anderem an der Interoperabilität der Hochsprachen und chromatic vor allem an der Geschwindigkeit. Hier liegt Parrot noch weit hinter den Erwartungen. Einige windige Benchmarks befanden Rakudo bis zu mehrere hunderte male langsamer als Perl 5. Auch wenn Vergleiche auf Bytecodeebene zeigten, dass Parrot wesentlich langsamer als die CLR ist, sind das keine zuverlässigen Zahlen. Sie helfen nur, die kürzlich erreichte Beschleunigung von Parrot um das Siebenfache, zu relativieren. Weitere Beschleunigungen haben chromatic und Patrick bereits angekündigt.

Erwähnenswert wäre noch, dass die PMC (parrot magic cookie) einen sachlich klingenden Beinamen für offizielle Präsentationen bekamen, der ihren Charakter dennoch genau beschreibt: *polymorphic container*. PMC sind Registertypen, die sich weniger an einer Hardware-CPU orientieren wie `int` oder `float`, sondern an mehr Dingen wie Perl-Strings oder Listen.

PCT

Wesentlich mehr Sichtbares tat sich auf der Ebene darüber. Die Parrot Compiler Tools, eine in PIR geschriebene Werkzeugkiste, die helfen soll Hochsprachcompiler für Parrot mit wenig Mühe aufzubauen, hat sich wieder mal gewandelt. Im vorvorigen Jahr wurde die *TGE* (Tree Grammar Engine), ein Umwandler abstrakter Syntaxbäume (*AST*), zugunsten der objektorientierten *HLLCompiler* eingemottet. Diese sind mächtiger, robuster und schneller. Ebenso wurde jetzt auch die *PGE* (Parrot Grammar Engine), eine Perl 6-Regex engine durch eine neues NQP mit regulären Ausdrücken ersetzt (nqp-rx). Dadurch kam unter anderem die wichtige Funktion der *proto-regex* hinzu, die erlaubt komplexe und sich dynamisch verändernde Regex einfacher zu schreiben.

NQP ist eine sehr einfach Form (Teilmenge) von Perl 6. Sie enthält weit mehr Hochsprachenelemente als das immer noch stark assemblerartige PIR und vereinfacht dadurch in den meisten Fällen das Schreiben der Hochsprachcompiler.

Rakudo

Da Rakudo mit dem neuen NQP die Spezifikation genauer erfüllen kann, teilte sich im Oktober 2009, kurz nach dem *release #22* die Entwicklung, um frühestmöglich dieses Ziel zu erreichen. Der neue, auf *nqp-rx* basierende Zweig nannte sich *rakudo-ng* und war erst Mitte Februar, als Version #26, für Nutzer freigegeben. Eine wichtige Neuerung war auch, dass *Arrays* endlich so lazy wie in Haskell sind. Ausdrücke die den Inhalt eines Arrays definieren, werden je Arrayelement so spät wie möglich evaluiert. Das war schon lange von den Synopsen gefordert aber nicht früher umsetzbar. Rakudo hatte nach #26 immer noch Probleme (rückfällige Tests), die größtenteils behoben werden sollen, bevor es als Rakudo * angeboten werden kann.

Rakudo Star (Stern) war Patrick Michauds Idee. Er stellte während der YAPC::EU im August 2009 fest, dass spätestens im nächsten Frühjahr Rakudo reif genug für ein breiteres Publikum sein müsste. Weite Teile der Syntax wie *OOP*, *junctions*, Signaturen u.s.w. sind bereits umgesetzt und werden sich kaum noch ändern. Viele Details wie die pipe-Operatoren ("==>") und vor allem die optionale Typisierung fehlen noch, fallen aber weniger ins Gewicht. Daher werden die heute für Rakudo geschrieben Programm größtenteils auch unter Perl 6.0.0 laufen.

Ein anderer Grund für Rakudo * ist seine Stabilität. Masak und die anderen November-Autoren fanden viele Fehler. Deshalb werden mit Rakudo * keine Alphatester gesucht, sondern Entwickler von Programmen und Bibliotheken.

Eine Breite vorher unbedachter Fälle soll der Sprache und Interpreter den Feinschliff geben und dazu führen, dass Perl 6.0 mit allen erforderlichen Bibliotheken erscheinen kann. Denn mit *proto* gibt es auch auf der Rakudo-Seite einen kleinen Modulinstallierer, aber mit einem kleinem Webframework, JSON, URI und etwas Graphik erschöpft sich auch das Angebot. Carl Mäsaks *proto* versteht sich auch nur als Prototyp. Alle darüber angebotenen Module werden bald über *Plumage* und damit für alle Sprachen erhältlich sein.

Eine weitere treibende Kraft für Rakudo war neben November auch Ian Hague. Er hat der Perl Foundation am 14. Mai 2008 eine riesige Spende von 200.000$ gegeben, die zur Hälfte in die Entwicklung von Perl 6 fließen wird. Davon wurden bisher Beträge an die Hauptentwickler von Rakudo ausgezahlt: Patrick Michaud (Protoregex und weitere Erweiterungen des Kerns), Jonathan Worthington (Signaturen) und Jerry Gay (Kommandozeilenparameter).

Aber was bedeutet eigentlich Rakudo *? Diese Veröffentlichung passt nicht in die bekannten Schemata und deshalb gab es in einem etwas bunkerähnlichen Konferenzraum, im Keller eines edlen Kopenhagener Hotels, auch einen kollektiven Gehirnsturm, zu dem ich den kläglichen Vorschlag "0.5" einbrachte. Aber Rakudo ... hat keinen fest umrissenen Funktionsumfang und der Abgabetermin für 6.0.0 steht auch nicht fest. Dem davon ausgehenden Druck will man sich auch entziehen. Deshalb kam Patrick am nächsten Tag im Kongresssaal mit einem Namen, der genau seine Absicht ausdrückt, dass Rakudo "gut genug" ist. Er enthält auch eine kleine Spitze für Wissende, denn "*" wird in Perl 6 in mancherlei Kontext als "Whatever" gelesen, im Sinne: tu Irgendetwas, gib mir Alles, den Rest oder das Ende. In dem Sinne ist Rakudo * auch ein Überraschungsei. Ohne Schokolade, aber dafür mit voller Elchkraft können Programmierer alles was sie aus Moose und Perl6::* kennen hier ausprobieren.

Nur dieses mal ohne die syntaktischen Grenzen von Perl 5 und mit viel Neuem. Alte Bekannte wie *slurp*, dass jetzt im Sprachkern ist, kann man auch finden. Wer kommerzielle Kunden beliefern will, sollte nicht auf Perl 6 warten. Wer aber jetzt vollständige Perl 6-Programme schreiben will, etwa für unkritische Werkzeuge, die gerne ein paar Sekunden länger laufen dürfen, sollte Rakudo * unbedingt testen. Auch ein freies Perl 6-Buch wird als PDF im Lieferumfang enthalten sein. Da der Projektleiter derzeit von seiner Familie gebraucht wird, aber auch der Wechsel zu *nqp-rx* anspruchsvoller als erwartet ist, verschiebt sich die Veröffentlichung wohl auf Mai, spätestens Juni.

Herbert Breunung

Was ist neu in Perl 5.12?

Perl 5.12 ist da! Rund 8 Monate nach Perl 5.10.1 gibt es ein neues Major-Release von Perl. Pumpking für Perl 5.12 ist Jesse Vincent. Mit den Vorbereitungen zu Perl 5.12 hat sich auch etwas mit den Release-Zyklen getan. Ab Perl 5.11 (ungerade Nummern bei den Hauptversionen sind Entwicklerversionen) wurde jeden Monat ein Release gemacht. Damit sollte mehr Stabilität und bessere Wartbarkeit erreicht werden.

Aber nicht nur im Hintergrund hat sich einiges bei Perl 5.12 getan. Auch im Code. In den nachfolgenden Abschnitten werden einige (wichtige) Änderungen kurz erläutert.

Implizites "use strict"

Von vielen wurde das schon länger gefordert, um eine sauberere Programmierung zu fördern - ein implizites "strict". Um Rückwärtskompatibel zu bleiben, ist das implizite "strict" allerdings nicht standardmäßig angeschaltet, sondern muss über `use 5.12.0` aktiviert werden.

Ein `use 5.12.0` ist äquivalent zur

```
use strict;
use feature ':5.12';
```

package NAME VERSION;

Mit Perl 5.12 wurden die Möglichkeiten der `package`-Anweisung erweitert. Man kann jetzt bei der Definition des `package`s die Version angeben.

```
package MeinModul 0.01;
```

Damit muss keine Paket-Variable `$VERSION` mehr definiert werden. Diese neue Syntax löst also

```
package MeinModul;
our $VERSION = 0.01;
```

ab.

Damit spart man sich nicht nur einige Zeichen beim Tippen, diese Syntax ist auch konsistent zu den `use`-Anweisungen, bei denen man mit

```
use MeinModul 0.01;
```

sagen kann, dass ein Modul eine bestimmte Version haben muss.

Yada-Yada-Operator

Wer kennt das nicht - man möchte eine grobe Idee skizzieren und in Blöcken noch keinen Code schreiben. Im Fließtext würde man "..." schreiben. Das geht jetzt auch im Perl-Code:

```
if ( $wahr ) {
    ...
}
```

Der Compiler meckert da nichts an, es ist also gültige Syntax. Damit man das aber nicht vergisst zu implementiert, wirft Perl zur Laufzeit einen Fehler mit der Meldung 'Unimplemented'.

Dieser Operator kann allerdings nicht als Ersatz für einen Ausdruck eines längeren Statements stehen, da hier Mehrdeutigkeiten mit der "..."-Version des Range-Operators vorkommen könnten. Das ist also nicht erlaubt:

```
print ...;
```

Perl 5.12 ist Jahr-2038-Konform

In Ausgabe 11 von $foo hat Thomas Fahle ja schon das Modul Time::y2038 von Michael Schwern vorgestellt. Die Funktionalität ist jetzt auch in den Perl-Kern eingeflossen, so dass die Zeit-Funktionen Jahr-2038-fähig sind. Das mag im Moment noch nicht für alle Programmierer relevant sein, aber man sollte besser schon früh gewappnet sein.

Überladen von qr//

Bisher konnte man das Verhalten von Objekten im String-, Boolschen- oder Numerischen-Kontext anpassen, indem man mit `overload` gearbeitet hat. Ab sofort kann man das Verhalten von Objekten bei Regulären Ausdrücken anpassen. Die Verhaltensänderung tritt dann in Kraft, wenn das Objekt auf der rechten Seite von `=~` verwendet wird.

```
package MeinModul 0.01;

use overload 'qr' => \&create_regexp;

sub new { return bless {}, shift }

sub color

sub create_regexp {
    my ($self) = @_;
    my $string = $self->{color};
    return qr/$string/;
}

package main;

my $obj = MeinModul->new;
$obj->color( 'blue' );

print "yes" if 'orange_blue' =~ $obj;
```

Aktualisierte Unicode-Quellen

Die Unicode-Datenbank wurde auf die Version 5.2 aktualisiert. In der perlunicode.pod Dokumentation ist beschrieben, wie man andere Versionen installiert und verwendet.

Flexibleres each

`each` kann jetzt auch auf Arrays angewendet werden.

```
my @array = qw(null eins zwei);
while ( my ($index,$value) = each @array ) {
    print "$index -> $value\n";
}
```

Mit `each` kommt man so auch an den Index des Elements eines Arrays, ohne dass man mit einer Zählvariablen arbeiten muss.

Filehandles

Filehandles sind ab sofort immer vom Typ `IO::File`. Vor Perl 5.12 war es so, dass Filehandles vom Typ `FileHandle` waren, wenn es im Speicher war, ansonsten vom Typ `IO::Handle`.

suidperl

Da der Wartungsaufwand zu groß geworden wäre und sich kaum einer wirklich damit auskennt wurde `suidperl` aus dem Perl-Kern entfernt.

Eigene Schlüsselworte

Mit dem Modul `Devel::Declare` kann man eigene Schlüsselworte definieren (siehe auch $foo Ausgabe 12). Die neue Perl-Version kann das von Haus aus. Dafür muss man allerdings ein wenig XS können. Als Beispiel kann man sich `XS::APItest::KeywordRPN` anschauen.

Neue Deprecation-Warnungen

Auch in dieser Version gibt es neue Warnungen bezüglich veralteten Codes. Diese Warnungen beinhalten:

- Einfaches << bei HEREDOCS, in der Bedeutung von <<""
- Liste von Werten ohne Komma-Trennung (in format)
- defined(%hash)
- defined(@array)

Es gibt noch weitere Features, die als "veraltet" angesehen werden. Die komplette Liste ist in der "errata"-Datei zu finden.

Und einige Module, die in zukünftigen Perl-Versionen aus dem Core rausfallen werden (und auf dem CPAN weiterleben), werden durch Warnungen gekennzeichnet. Neu ist, dass diese Warnungen standardmäßig eingeschaltet sind.

Wer diese Warnungen nicht ausgegeben haben möchte, muss diese Module per CPAN-Client installieren. Denn der Perl-Interpreter meckert die Module nur an, wenn diese aus dem lib-Verzeichnis geladen werden und nicht aus dem site-lib oder vendor-lib.

Änderungen am smart-match

Schon bei Perl 5.10.1 wurde am smart-matching einiges nachgebessert, um im Aufruf konsistent zu sein. Diese Änderungen wurden auch in Perl 5.12 übernommen. Außerdem wurde noch an `when` etwas geändert, so dass der Range-Operator als Flip-Flop in der `when`-Bedingung eingesetzt werden kann:

```
when( /START/ .. /STOP/ ) {
    print "innerhalb der START-STOP-Tags\n";
}
```

Und das "defined-or" kann innerhalb von `when` verwendet werden:

```
when ( $a // $b ) {
    # ...
}
```

Eigene Module zum Auffinden der passenden Methode

In Objektorientiertem Code mit Mehrfachvererbung ist ja immer die Frage da, aus welcher Klasse wird eine Methode aufgerufen, wenn mehrere Klassen die gleiche Methode implementieren. Bisher ist es so, dass eine Tiefensuche und dann in der Breite gesucht wird. Mit Perl 5.12 ist es auch möglich, über eine definierte API eigene Plugins zu schreiben. Wie diese Plugins aussehen müssen, ist in der perlmroapi.pod Dokumentation beschrieben.

Experimentelle Änderungen

Es gibt auch Änderungen, die sind (erstmal) als "experimentell" gekennzeichnet. Mit diesen Änderungen sollte man im Produktivcode vielleicht erstmal etwas vorsichtiger umgehen. Aber dennoch sollten sie zumindest mal ausprobiert werden. Die Perl-5-Porters sind auch für Feedback dankbar.

Ein paar Experimentelle Änderungen werden in den folgenden Abschnitten erläutert.

\N als Escape-Sequenz in Regex

Durch Groß-/Kleinschreibung werden in Regulären Ausdrücken häufig komplementäre "Gruppen" gekennzeichnet. So werden unter `\w` alle "Whitespaces" zusammengefasst und unter `\W` alle "Nicht-Whitespaces". In dieser Perl-Version wurde `\N` eingeführt als Gegenteil zu `\n`.

Die neue Sequenz mag für viele verwirrend sein, da mit `\N` auch Unicode-Zeichen eingeleitet werden:

```
/\N{LATIN SMALL LETTER A}/; # kleines a
```

Da es aber keine Unicode-Zeichen - jedenfalls noch nicht - gibt, deren Namen nur aus Zahlen besteht, gibt es hier keine Kollisionsgefahr.

```
/\N{4}/ # viermal 'kein Newline'
/\N{DISABLED CAR}/ # ein Unicode-Zeichen
/\N/ # einfaches 'kein Newline'
```

Mehr APIs für Interna

Nach und nach soll der Perl-Kern für andere Anwendungen "geöffnet" werden. Um die ganze Mächtigkeit von perl zu nutzen, muss man auf definiertem Weg auf die Interna zugreifen können. In Perl 5.12 wurden einige APIs eingeführt.

Sonstiges

Natürlich wurden auch viele andere Sachen in Perl ausgebaut und verbessert, wie die Einführung von Regulären Ausdrücken als Objekte erster Klasse oder die Definition von "stricten" und "laxen" Versionsnummer-Formaten. Viele Bugfixes seit Perl 5.10.0 sind auch schon in Perl 5.10.1 geflossen.

Es wurden viele CPAN-Module, die mit dem Perl-Kern ausgeliefert werden aktualisiert. Die komplette Liste der Module ist in der Dokumenation perldelta5120.pod zu finden.

Es gibt auch ein paar Fehler, die bekannt sind. Solche Fehler in 5.12.0 werden in einer "Errata" gesammelt. Diese ist in Jesse Vincents github-Account zu finden: http://github.com/obra/perl-5.12.0-errata

Renée Bäcker

Perl Scopes Tutorial - Teil 4

Im letzten Teil dieses Tutorials betrachten wir zunächst lexikalische Pragmas, da auch diese in ihrer Wirkung mit Scoping zusammenhängen. Danach wenden wir uns den Anweisungen zu, die zur lexikalischen Deklaration von Variablen und deren Benutzung verwenden werden: `my`, `our` *und* `state`.

Lexikalische Pragmas

Die Beschreibung von `local` und seinen Auswirkungen wäre unvollständig ohne die Betrachtung lexikalischer Pragmas.

Unter *Pragmas* versteht man Anweisungen an den Compiler - sie werden wie jedes andere Modul mit `use` eingebunden, laden aber (meistens) keinen aufrufbaren Code, sondern steuern durch spezielle Befehle das Verhalten des Compilers. Man erkennt sie an ihren zur Gänze in Kleinbuchstaben gehaltenen Namen.

Manche Pragmas haben globale Auswirkungen auf das ganze Programm, andere hingegen sind in ihrer Wirkung auf Scopes beschränkt. Globale Pragmas beeinflussen oft Symboltabellen (`vars`, `subs`) - da diese Tabellen stets im ganzen Programm sichtbar sind, sind es auch die durch diese Pragmas hervorgerufenen Änderungen. Pragmas mit auf Scopes reduziertem Geltungsbereich hingegen wirken nur innerhalb des Scopes, in dem sie vorkommen. Man erkennt sie leicht daran, dass man sie für jeden Scope aktivieren oder deaktivieren kann:

```
# Aktiviert <pragma> im aktuellen Scope
use <pragma>;

# Deaktiviert <pragma> im aktuellen Scope
no <pragma>;
```

Solche Pragmas bezeichnet man auch als *lexikalische Pragmas*, da bei ihnen - ähnlich wie bei `my` Deklarationen - die Position im Code, an der sie vorkommen, wesentlich ist - ihre Wirkung entfalten sie in Code, der lexikalisch unterhalb ihres Auftretens aufscheint, bis zum Ende des aktuellen Scopes [1].

Wie sind solche Pragmas implementiert?

Perl besitzt zwei spezielle Variable, die das Verhalten des Compilers steuern: `$^H` und `%^H` (sie werden oft - angelehnt an ihre Bezeichner - als *Hintvariable* bezeichnet). Erstere ist ein Bitvektor - jedes Bit steuert eine bestimmte Compilerfunktion; die entsprechenden Pragmas setzen bzw. löschen die zugehörigen Bits. Letztere (die erst seit Perl 5.6 existiert) ist ein Hash, dessen Keys beliebige Werte aufnehmen können. Diese Variable ist auch für künftige Erweiterungen (neue lexikalische Pragmas) vorgesehen.

Das Bemerkenswerte an diesen beiden Variablen ist, dass sie *automatisch* für jeden Scope lokalisiert und mit den aktuellen Werten initialisiert werden [2]. Es wird also an *jedem* Scopebeginn ein

```
{
  local ($^H, %^H) = ($^H, %^H);
  ...
}
```

ausgeführt. Durch das implizite Lokalisieren bleiben die durch die Pragmas an den Variablen hervorgerufenen Änderungen auf den jeweiligen Scope beschränkt. Den Code für ein Pragma wie `integer` kann man sich somit kurz und bündig als

```
package integer;

my $integer_bit = 1;

sub import {
  $^H |= $integer_bit;
}
sub unimport {
  $^H &= ~$integer_bit;
}
1;
```

vorstellen: ein `use integer` veranlasst den Aufruf der `import` Funktion, die das betreffende Bit in `$^H` setzt, durch `no integer` wird die `unimport` Funktion aktiviert, die es wieder löscht. Der Compiler überprüft bei jedem Auffinden einer arithmetischen Operation den Zustand des Bits und generiert den entsprechenden Code für Ganzzahl- oder Gleitkomma-Operationen.

Da es sich bei `$^H` und `%^H` um normale Perl-Variable handelt, kann man auch selbst darauf zugreifen. Dabei muss man sich jedoch vor Augen halten, dass diese Variable nur *während des Kompilierens* verwendet werden und auch die implizite Lokalisierung nur zu diesem Zeitpunkt besteht. Ein Ansprechen macht daher nur in Pragmacode, der durch `use` oder `no` aufgerufen wird, oder von Code in `BEGIN` Blöcken Sinn [3].

In Perl 5.6 kam mit dem `warnings`-Pragma auch noch die Variable `${^WARNING_BITS}` hinzu, die für jede Warn-Kategorie ein entsprechendes Bit enthält. Auch sie wird am Beginn jedes Scopes implizit lokalisiert, daher kann mit dem `warnings`-Pragma für jeden Scope individuell eingestellt werden, welche Meldungen ausgegeben und welche unterdrückt werden sollen. Im Unterschied zu den Hintvariablen beeinflusst `${^WARNING_BITS}` jedoch auch das Verhalten des Interpreters (da ja etliche Meldungen erst während des Programmlaufs ausgegeben werden), daher wird hier die implizite Lokalisierung auch zur Laufzeit durchgeführt.

Beachte, dass der Effekt der impliziten Lokalisierung bei der vor Perl 5.6 zur Steuerung der Ausgabe von Warnungsmeldungen verwendeten Variable `$^W` nicht vorhanden ist - diese Variable musste man in Scopes selber lokalisieren:

```
#!/usr/bin/perl -w # Warnungen einschalten

$^W = 1;           # Geht auch so
...
{
  local $^W = 0;   # Jetzt keine Warnungen
  ...
}          # Warnungen wieder wie vorher
```

Hier handelt es sich jedoch um eine *Laufzeitanweisung*, die auf das Verhalten des Compilers keinen Einfluss mehr hat, daher konnte man damit nur zur Laufzeit generierte Warnungsmeldungen beeinflussen. Für das Steuern der Ausgabe von Meldungen beim Kompilieren mussten Zuweisungen an `$^W` in `BEGIN` Blöcke gesetzt werden. Das Aktivieren von `$^W` im eigenen Programm führte dadurch manchmal zu unerwarteten Meldungen, wenn hinzugeladener Modulcode unsauber programmiert war.

Durch das `warnings` Pragma und die `${^WARNING_BITS}` Variable wurde dieser Mangel beseitigt - da jedes Modul in einem eigenen Scope - seinem Dateiscope - kompiliert wird, übernimmt `${^WARNING_BITS}` zunächst die Einstellungen jenes Codes, der das Modul geladen hat. Ändert das Modul daran nichts, wird es mit diesen Einstellungen kompiliert. Werden aber durch den Aufruf von Pragmas Einstellungen modifiziert, so hat das nur Einfluss auf den Kompilierprozess des Modulcodes; durch das implizite Lokalisieren werden die Einstellungen des Hauptprogramms nicht beeinflusst und nach dem Kompilieren des Moduls weiterverwendet.

Lexikalische Variable deklarieren mit `my`

Lexikalische Variable und damit `my` und statisches Scoping wurden in Perl 5 eingeführt.

Lexikalische Variable werden mit `my` deklariert - diese Deklaration hat Einfluss auf das Verhalten des Compilers, der ansonsten beim Antreffen von Bezeichnern im Code von globalen Variablen ausgeht.

Durch `my` wird in Perl *statisches Scoping* implementiert: die Bindung einer Variable an ihre Umgebung (Scope) wird beim Kompilieren eingerichtet und kann zur Laufzeit nicht mehr geändert werden.

`my` besitzt jedoch auch einen *Laufzeiteffekt*: die Variable wird beim **Verlassen** des Scopes, in dem sie deklariert wurde, **automatisch gelöscht** - wird daher der Scope neu betreten (z.B. bei Schleifen oder Funktionsaufrufen), ist die Variable wieder undefiniert. Ist dies nicht gewünscht, muss die Variable *außerhalb* der Schleife oder Funktion deklariert, oder es muss `state` verwendet werden.

Mehrere lexikalische Variable können gleichzeitig durch Angabe einer Liste deklariert und es können ihnen dabei auch Werte zugewiesen werden:

```
my ($x, $y, $z) = (1, 2, 3);
```

Dabei ist jedoch zu beachten, dass die Zuweisung erst zur Laufzeit ausgeführt wird. Daher macht eine Anweisung wie

```
my $x if $r == 2;
```

wenig Sinn, da sie erst zur Laufzeit den aktuellen Wert der lexikalischen Variable `$x` (der zu diesem Zeitpunkt immer `undef` ist) bereitstellt, wenn `$r` den Wert zwei hat. Auf die Compilerdirektive (d.h. auf das Einrichten der lexikalischen Variable) hat diese Bedingung keinen Einfluss.

Beachte, dass bei Anweisungen wie

```
my $x = $x;
```

die lexikalische Variable erst in der der Deklaration folgenden Zeile ansprechbar ist - das `$x` auf der rechten Seite ist noch eine globale (oder in einem äußeren Scope deklarierte) Variable.

Die Deklaration wird häufig gleich mit dem Befehl, in dem die Variable erstmals verwendet wird, verbunden:

```
open my $fh, ...
for (my $i = 0; ...)
while (my $var = ...)
bless \my %obj, ...
myfunc(\my @array);
```

Generell darf `my` überall dort verwendet werden, wo auch ein Variablename (ein *lvalue*, d.h. ein in einer Zuweisung links stehender Ausdruck) vorkommen kann.

Das letzte Beispiel zeigt den Aufruf einer Funktion, der die Referenz einer neu deklarierten lexikalischen Variable übergeben wird. Die Funktion kann dann über die Referenz auf die Variable zugreifen:

```
sub myfunc {
  # Hol (hier) Referenz auf @array
  my $ref = shift;
  push @$ref, 1;      # und setzt Wert
}
```

`my` kann auch zur Deklaration von Typen und Attributen verwendet werden:

```
# Deklariere $var vom Typ "Test"
my Test $var;
# Deklariere %hash mit Attribut "Attrib"
my %hash:Attrib;
```

Darauf wird in den Beschreibungen der `fields` und `attribute` Pragmas, die Schnittstellen zur Typen- und Attributverarbeitung bereitstellen, eingegangen.

Scratchpads

In diesem Tutorial wurde schon an verschiedener Stelle der Begriff *Scratchpad* erwähnt, sodass dieser abschließend näher erläutert werden soll.

Jeder Scope besitzt ein Scratchpad (*Zwischenablage*). Neben anderen Daten (temporären Werten, anonymen Funktionen, Literalen u.a.) werden auch lexikalische Variable in Scratchpads abgelegt.

Stößt der Compiler auf eine Deklaration wie

```
my $var;
```

so reserviert er im Scratchpad des gerade aktuellen Scopes einen Eintrag für die Variable. Stößt er später wieder auf diesen Namen, z.B. in

```
print $var;
```

so sieht er zunächst im Scratchpad des aktuellen Scopes nach, ob die Variable dort existiert. Ist dies der Fall, generiert der Compiler den Zugriff auf den Eintrag im Scratchpad. Ansonsten wird im Scratchpad des nächst-äußeren Scopes nachgesehen und, falls dort vorhanden, auf dessen Eintrag zugegriffen. Falls die Variable auch dort nicht zu finden ist, wird wieder im nächst-äußeren Scope Ausschau gehalten, solange bis der äußerste Scope (Dateiscope) erreicht ist. Wird die Variable auch in dessen Scratchpad nicht gefunden, wird sie als global angesehen und es wird vom Compiler der Zugriff auf die jeweilige Symboltabelle (Paketvariable) generiert.

Scopes werden also von *innen nach außen* durchsucht, woraus folgt, dass eine in einem äußeren Scope deklarierte Variable von einem inneren Scope aus nicht mehr ansprechbar ist, sobald in diesem Scope eine gleichnamige lexikalische Variable deklariert wurde. Weiter folgt daraus, dass bei einer gleichnamigen lexikalischen und globalen Variable vom Compiler immer die Lexikalische gefunden wird; auf die Globale kann dann nur mehr entweder durch Vorsetzen des Paketnamens oder über ihren Typeglob zugegriffen werden:

```
# Global (lex. $x noch nicht deklariert)
$x = 1;
my $x = 2;  # Aber jetzt
# Gibt 2 aus, lexikalische Variable
print $x;
# Gibt 1 aus, globale Variable
print $main::x;
# Dasselbe, allgemeiner
print ${__PACKAGE__.'::x'};
# Ebenso, ueber Typeglob
print ${*x};
# Ebenso, mit Liebe zum Detail
print ${*x{SCALAR}};
# Fuer ganz Gruendliche
print ${*main::x{SCALAR}};
```

Es ist zu erkennen, dass lexikalische und globale Variable vollkommen unterschiedlich implementiert sind (das liegt

an den verschiedenen Zeitpunkten ihrer Einführung). Daher gibt es im Umgang mit lexikalischen Variablen einige Besonderheiten, die man kennen sollte:

• In Scratchpads können nur Variable (Skalare, Array, Hashes) abgelegt werden, keine der anderen Wertetypen wie Funktionen [4], Handles oder Formate. Arrays und Hashes müssen als Ganzes deklariert werden, die Deklaration einzelner Elemente

```
my $array[6];            # Nicht erlaubt
my $hash{'hallo'};       # Detto
```

ist nicht erlaubt.

Perls Interpunktionsvariable (`$_ $! @_` usw.) sind *immer* global und demzufolge ist eine Deklaration mit `my` unzulässig [5].

• Typeglobs sind Bestandteile globaler Variable, daher können sie in Scratchpads nicht vorkommen. Eine Deklaration wie

```
my *hallo;         # Nicht erlaubt
```

macht daher keinen Sinn. Ebenso verboten ist

```
my %stash::;             # Nicht erlaubt
```

da auch Symboltabellen nur in globalen Variablen abgelegt werden dürfen. Außerdem fühlt sich der Parser von den Doppelpunkten in Verbindung mit `my` gestört.

• Das bei globalen Variablen mit Hilfe von Typeglobzuweisungen unterstützte *Aliasing*

```
*package1::var = *package2::var;
```

ist mit lexikalischen Variablen, da Scratchpads keine Typeglobs verwenden, nicht möglich [6].

• Scratchpads gehören Scopes und keinen Packages an. Daher sind Paketnamen in Bezeichnern lexikalischer Variable nicht erlaubt:

```
my $Hugo::var;     # Nicht erlaubt!
```

• Da auch symbolische Referenzen über Symboltabellenzugriffe (und nicht über Scratchpads) abgebildet werden, können lexikalische Variable hierfür nicht verwendet werden. Aber Achtung: die Variable, *mittels derer* die Referenz gebildet wird, kann sehr wohl lexikalisch sein:

```
my $name = 'hugo'; # Kann lexikalisch sein
$hugo = 'Hallo';   # MUSS global sein
print ${$name};    # Gibt 'Hallo' aus
```

Im `print` Befehl wird zunächst der Name der symbolischen Referenz ermittelt. Das ist ein ganz normaler Zugriff, der auf lexikalische wie auf globale Variable funktioniert. Dann wird mittels des `${...}` Konstrukts dieser Name zum Zugriff auf die Symboltabelle (nicht auf ein Scratchpad!) verwendet. Daher muss die auf diese Art referenzierte Variable global sein.

• Die durch Symboltabellen und Typeglobs vorhandene Möglichkeit der *Introspektion* globaler Variable (das Untersuchen von Datenstrukturen mit vorhandenen Sprachmitteln) ist bei Scratchpads nicht gegeben. Perl verfügt über keine Pseudo-Arrays oder Pseudo-Hashes, die auf Scratchpads abgebildet sind und direkt von Perlcode angesprochen werden könnten. Es stehen allerdings mit

```
PadWalker
Lexical::Alias
Lexical::Util
B::Showlex
B::LexInfo
```

(um nur einige zu nennen) etliche CPAN-Module zur Manipulation von Scratchpads und lexikalischen Variablen zur Verfügung. Diese Module verwenden hierfür entweder XS-Code oder den Backend-Compiler `B` (wie am Namen erkennbar).

Scratchpads bieten eine sehr effiziente Variablenverwaltung: da sie intern als Arrays (*Padarrays*) implementiert sind, ist der Zugriff wesentlich schneller als über Typeglobs. Wenn der Compiler für eine lexikalische Variable einen Eintrag im Scratchpad anlegt, dann merkt er sich den Index innerhalb des Padarrays. Wird später auf die Variable zugegriffen, geschieht dies nur über diesen Index; der Variablename wird zur Laufzeit nicht mehr benötigt.

Die Einschränkung, dass man in Scratchpads keine Funktions- oder Handlewerte ablegen kann, ist kein Problem, weil sich *Referenzen* darauf sehr wohl abspeichern lassen:

```
my $codref = \&code; # Funktionsreferenz
# Typeglob (Handle) Referenz
my $fhref = \*FH;
&{$codref}; # Dereferenzierung (Aufruf)
*{$fhref};   # Dereferenzierung (Handle)
```

Das Verwenden von Globreferenzen ist aber gar nicht notwendig, da seit Perl 5.6 alle Funktionen, die Handles als Ar-

gumente verwenden, die benötigten Typeglobs bei der ersten Benutzung automatisch anlegen, wenn ihnen ein Skalar mit dem Wert `undef` übergeben wird (*Auto-Vivifying*):

```
open my $fh, ...;
```

Im Skalar wird dann eine Referenz auf den angelegten Typeglob abgespeichert. Ein Seiteneffekt davon ist, dass die geöffnete Datei automatisch geschlossen wird, sobald der Scope, in dem `$fh` angelegt wurde, verlassen wird. Dies ist seither der bevorzugte Weg, mit Filehandles zu arbeiten.

Die Verwendung von Referenzen ist in keiner Richtung eingeschränkt: eine lexikalische Variable kann eine Referenz auf eine globale Variable enthalten und eine Globale eine Referenz auf eine Lexikalische. In diesem Fall bleibt der Wert der lexikalischen Variable auch nach Beendigung des Scopes erhalten, in dem sie deklariert wurde:

```
our $x;             # Globale Variable
{
  my $y = 3;        # Lexikalische Variable
  $x = \$y;         # Referenz auf $y
}                   # $y ist jetzt weg
# Aber sein Wert nicht; gibt 3 aus
print $$x;
```

Das innere Scratchpad wird nach Beendigung des Scopes bereinigt, aber der Wert der darin deklarierten Variable bleibt erhalten, da er über die äußere Referenz immer noch angesprochen wird.

Globale Variable deklarieren mit `our`

`our` und damit die Möglichkeit, *globale* Variable lexikalisch zu deklarieren, wurde in Perl 5.6 eingeführt.

Auch `our` implementiert *statisches Scoping*, d.h. das Binden von Variablen an Scopes. Der Unterschied ist lediglich, dass nun globale Variable angesprochen werden.

Durch eine solche Deklaration wird in Verbindung mit `strict` sichergestellt, dass eine Variable nur innerhalb des oder der vorgesehenen Scopes verwendet wird. Wird solcher Code später geändert und dadurch auf die Variable an einer ursprünglich nicht vorgesehenen Stelle zugegriffen, so wird dies sofort durch den Compiler gemeldet. Auch falsch geschriebene Variablenamen werden dadurch schnell entdeckt.

Wie `my` ist auch `our` eine Compilerdirektive und verwendet eine ähnliche Syntax. Eine Deklaration wie

```
our ($x, $y, $z);
```

teilt dem Compiler mit, dass die drei genannten Variablen im aktuellen Scope (und in untergeordneten Scopes) verwendet werden sollen. Beachte, dass diese - im Unterschied zu lexikalischen Variablen - durchaus schon vorher Werte besessen haben können, sie werden durch die Deklaration *nicht* initialisiert:

```
use strict 'vars';
{
  our ($x, $y, $z) = (1, 2, 3);
}
# Jetzt beginnt ein neuer Scope
{
  # Wir wollen $x, $y, $z wieder verwenden
  our ($x, $y, $z);
  # Gibt 1, 2, 3 aus
  print "$x, $y, $z\n";
}
```

Da es sich um *globale* Variable handelt, behalten sie natürlich auch außerhalb der Scopes, in denen sie deklariert wurden, ihren Wert. Die Deklaration teilt dem Compiler lediglich mit, dass ein *Ansprechen* im jeweiligen Scope vorgesehen ist. Werte können dabei gleichzeitig zugewiesen werden, aber dies ist nicht notwendig; die Variablen behalten ansonsten ihren vorigen Wert. Der bei `my` vorhandene Effekt, dass Variable am Scopebeginn stets undefiniert sind, ist hier nicht gegeben; soll eine mit `our` deklarierte Variable beim Betreten eines Scopes undefiniert sein, so muss sie händisch initialisiert werden. Die Möglichkeit einer solchen Initialisierung bei gleichzeitigem Wegsichern des alten Wertes ergibt sich durch die gemeinsame Verwendung von `local` und `our` (in dieser Reihenfolge!):

```
{
  local our $y;
  ...
}
```

Die `local` Anweisung rettet den vorigen Wert der Variable und setzt diese auf `undef`. Dadurch ist sie - wie auch jede lexikalische Variable - am Scopebeginn undefiniert. Die `our` Deklaration teilt dem Compiler mit, dass die Variable in diesem Scope angesprochen werden soll (was durch die `local` Anweisung bereits geschieht). Beachte, dass `our` nur während des Kompilierens, `local` hingegen nur zur Laufzeit Bedeutung hat.

Natürlich kann man im obigen Beispiel der Variable auch gleich einen neuen Wert zuweisen.

Tatsächlich deklariert man mit `our` keine Variable, sondern nur *die Absicht, sie anzusprechen*. `our` hat somit auch keinen Laufzeiteffekt, sondern ist nur beim Kompilieren - in Verbindung mit `strict vars` - von Bedeutung.

Für die Verwendung von `our` gilt sinngemäß dasselbe wie für `my` - wird eine Variable zum erstenmal damit deklariert, kann die Deklaration auch mit dem Befehl, der sie verwendet, zusammengefasst werden:

```
open our $fh, ...
for (our $i = 0; ...)
while (our $var = ....)
bless \our $obj, ...
myfunc(\our @array);
```

sind alles erlaubte Konstrukte - ob sie (speziell in Schleifen) sinnvoll sind, ist eine andere Frage.

Das Verschachteln von `our` Direktiven derselben Variablen macht hingegen keinen Sinn, da eine Deklaration bekanntlich auch in allen untergeordneten Scopes gültig bleibt und im Falle von `our` jedesmal dieselbe globale Variable angesprochen wird. `our` implementiert *kein* dynamisches Scoping; möchte man den Wert einer globalen Variable innerhalb eines Scopes temporär ändern, muss man `local` (siehe perlfoo #13) verwenden.

Wie `my` kann auch `our` zur Deklaration von Typen und Attributen - diesmal mit globalen Variablen - verwendet werden:

```
our Test $var;
our @array:Attrib;
```

Die Typendeklaration mit `our` hat in den aktuellen Perl-Versionen keine Bedeutung; die Attributdeklaration kann dagegen sinnvoll sein (siehe die Beschreibung des `attributes` Pragmas).

Scratchpads und `our`

Um das Verhalten und die Auswirkungen von `our` besser zu verstehen, ist es hilfreich, die Beziehung solcher Deklarationen zu Scratchpads zu kennen.

Aufgabe von `our` war es, das Ansprechen globaler Variable mit einfachen (unqualifizierten) Namen auch mit aktivem `strict 'vars'` zu ermöglichen - und zwar im Unterschied zum `vars` Pragma nicht global, sondern auf den jeweiligen Scope eingeschränkt (als *lexikalische Alternative* zu `vars`, das auf Symboltabellen einwirkt, um seine Aufgabe zu erfüllen).

Um das zu bewerkstelligen, wurde ein bereits bestehender Weg nur ein Stück weitergegangen: Man hat einfach ausgenutzt, dass der Compiler beim Antreffen eines Variablenamens wie

```
$hugo
```

zunächst immer alle Scratchpads - von innen nach außen - durchsucht. Wird die Variable in einem der Scratchpads gefunden, so gibt sich der Compiler damit zufrieden - ein `strict 'vars'` kommt nicht zum Tragen. Daher war es naheliegend, diesen Mechanismus auch für globale Variable zu verwenden - und genau das ist es, was `our` macht: es legt einen **lexikalischen Alias auf eine globale Variable** an. Wie bei `my` wird im Scratchpad des aktuellen Scopes ein Eintrag für die Variable geschaffen - dieser Eintrag repräsentiert aber keinen Wert, sondern es ist ein *Zeiger* auf jenes Package, das die gleichnamige globale Variable enthält.

Stößt der Compiler später auf einen unqualifizierten Variablenamen, so werden zunächst die Scratchpads durchsucht und es wird der durch `our` angelegte Eintrag gefunden - genug, um `strict 'vars'` zufriedenzustellen. Anschließend erkennt der Compiler, dass es sich um einen Alias handelt - und daraufhin generiert er den Zugriff auf die globale Variable genauso, wie er das ohne `our` Deklaration getan hätte. Der zur Laufzeit ausgeführte Code zum Zugriff auf eine mit `our` deklarierte Variable ist identisch zu solchem ohne eine derartige Deklaration [7] - daher hat, wie schon weiter oben erwähnt, `our` zur Laufzeit keine Bedeutung mehr.

Aus dem Umstand, dass `our` genau wie `my` Scratchpads manipuliert, ergeben sich einige Konsequenzen, derer man sich bei seiner Verwendung bewusst sein sollte.

• Da Scratchpads nur ganze Aggregate (Arrays, Hashes) aufnehmen können, ist eine Deklaration einzelner Elemente unzulässig:

```
our $array[6];          # Nicht erlaubt!
our $hash{'hallo'};     # Nicht erlaubt!
```

• Symboltabellen und Typeglobs gelten nicht als Variable (im Sinne von `strict 'vars'`) und daher sind die folgenden Deklarationen unnötig (und nicht erlaubt):

```
our %symtab::;          # Nicht erlaubt!
our *glob;              # Nicht erlaubt!
```

• `our` deklariert zwar globale Variable (Paketvariable), aber es wirkt auf Scratchpads, und daher sind auch nur einfache Namen erlaubt:

```
our $Hugo::Var;    # Nicht erlaubt!
```

• Hingegen ist die Nutzung von mit `our` deklarierten Variablen in symbolischen Referenzen durchaus zulässig, da es sich bei diesen ja auch um globale Variable, die in Symboltabellen abgelegt sind, handelt.

• Innerhalb eines Scopes darf eine Variable nur mit `my` *oder* mit `our` deklariert werden, da pro Scratchpad ein Eintrag nur einmal existieren kann.

Die Tatsache, dass `our` lexikalische Einträge für globale Variable (Paketvariable) erzeugt, führt manchmal zu unerwartetem Verhalten:

```
package Alpha;
our $var = 1;

package Beta;
our $var = 2;

package Alpha;
print $var;        # Gibt 2 aus
```

Hier würde man erwarten, den Wert 1 zu erhalten, da `$Alpha::var` dieser Wert zugewiesen wurde. `package` wirkt aber auf den aktuellen Scope; es stellt den für Paketvariable verwendeten Namensraum ein, wenn keiner explizit im Bezeichner angegeben ist. Der Scope bleibt aber derselbe, d.h. der gesamte obige Code läuft innerhalb eines Scopes ab, und daher beeinflussen die beiden `our` Direktiven auch dasselbe Scratchpad. Es wird daher zuerst für die Variable `$Alpha::var` im Scratchpad ein lexikalischer Alias angelegt, der in der Folge durch die zweite Deklaration **überschrieben** wird und danach auf die Variable `$Beta::var` zeigt. Die dritte `package` Direktive ändert an diesem Umstand nichts, und daher wird der Compiler beim folgenden Ansprechen von `$var` auf den überschriebenen Alias stoßen und den Code für den Zugriff auf `$Beta::var` generieren.

Die Variable im Package `Alpha` kann dann nur mehr mit

```
print $Alpha::var;          # Gibt 1 aus
```

oder

```
print ${*var};        # Detto
```

angesprochen werden. Beachte, dass im zweiten Fall die letzte `package` Direktive dafür sorgt, dass über den Typeglob wieder die Variable im Package `Alpha` angesprochen wird.

Perl gibt für den obigen Code keine Warnung aus, da beide `our` Direktiven zwar im selben Scope, aber auf unterschiedliche Variable einwirken. Daher sollte man mit solchen Konstrukten vorsichtig sein; es empfiehlt sich generell, pro Scope nur eine `package` Direktive - und diese zweckmäßigerweise gleich am Scopebeginn - zu verwenden. Wird dann innerhalb dieses Scopes eine Variable irrtümlich zweimal deklariert, so erhält man eine Warnungsmeldung.

`our` wird oft als Nachfolger für das `vars` Pragma verkauft. In vielen Fällen ist das richtig - sowohl `our` als auch `vars` ermöglichen das Ansprechen globaler Variable ohne Packagenamen bei aktivem `strict 'vars'`. Es ist aber wichtig, sich deren vollkommen unterschiedliche Arbeitsweise vor Augen zu halten: `our` legt im aktuellen Scratchpad einen Alias an und ist daher in seinen Auswirkungen an Scopegrenzen (und somit auch an Dateigrenzen!) gebunden. `vars` manipuliert die Symboltabelle des jeweiligen Packages; seine Auswirkungen sind nicht an Scopegrenzen gebunden und von *überall* sichtbar - auch von Code aus anderen Dateien. Das kann ein Vorteil oder Nachteil sein - und muss vom Programmierer von mal zu mal selbst entschieden werden.

Persistente Variable deklarieren mit `state`

Wie bereits mehrfach erwähnt, gibt es ab Perl 5.10 eine weitere Möglichkeit zum Deklarieren lexikalischer Variable: mittels des Schlüsselwortes `state` ist es möglich, solche Variable von vorneherein als persistent zu kennzeichnen, wodurch man sich den Umweg über die `my` Deklaration in einem übergeordneten Scope sparen kann.

Achtung: auf die Verwendung von `state` muss der Compiler durch

```
use feature 'state';     # oder
use 5.10.0;              # kuerzer
```

erst vorbereitet werden.

Wie mit `my` werden auch mit `state` lexikalische Variable deklariert, allerdings sind diese *persistent*, d.h. sie behalten ihren Wert, auch wenn der Scope, in dem sie deklariert wurden, zwischenzeitlich verlassen wird. Das folgende Beispiel veranschaulicht den Unterschied:

```
sub my_inc {
  my $x;
  return ++$x;
}
sub state_inc {
 state $x;
 return ++$x;
}

# Gibt 1 1 1 1 1 aus
print my_inc() for 1..5;
# Gibt 1 2 3 4 5 aus
print state_inc() for 1..5;
```

Während die mit `my` deklarierte Variable bei jedem Aufruf der Funktion uninitialisiert ist und daher jedesmal nach dem Inkrementieren der Wert 1 zurückgegeben wird, ist dies bei der mit `state` deklarierten Variable nur beim erstenmal der Fall, dann bleibt der Wert erhalten und wird bei jedem weiteren Aufruf um eins erhöht.

Technisch gesprochen unterdrückt `state` den Laufzeiteffekt von `my`, vor dem Verlassen eines Scopes eine darin deklarierte lexikalische Variable auf `undef` zu setzen.

Wie mit `my` und `our` können auch mit `state` mehrere Variable gleichzeitig deklariert (und initialisiert) werden:

```
state ($x, $y, $z) = (100, 200, 300);
```

Wird einer solchen Variable bei der Deklaration ein Wert zugewiesen, so geschieht diese Zuweisung *nur einmal*:

```
sub inc {
  state $x = 50;
  return ++$x;
}
print inc() for 1..5;
# Gibt 51 52 53 54 55 aus
```

Beachte aber, dass der Ausdruck auf der rechten Seite der Zuweisung dennoch *jedesmal* ausgewertet wird, auch wenn die Zuweisung selbst unterbleibt:

```
sub f {
  print "f() wurde aufgerufen!\n";
  return 20;
}
sub inc {
 state $x = f();
 return ++$x;
}
print inc() for 1..5;
```

In diesem Beispiel wird neben der Zahlenreihe 21...25 auch fünfmal `f() wurde aufgerufen!` ausgegeben, da die Funktion bei jedem Aufruf von `inc()` ausgeführt wird. Sie gibt auch fünfmal den Wert 20 zurück, dieser wird aber nur beim ersten Aufruf zugewiesen, die weiteren Male wird er verworfen.

Beachte, dass in Perl 5.10 solche einmaligen Zuweisungen **nur bei Skalaren** unterstützt sind, für Arrays und Hashes sind sie nicht erlaubt und werden mit der Fehlermeldung

```
Initialization of state variables in
  list context currently forbidden
```

zurückgewiesen. Hier wird man die Variable wie vor Perl 5.10 außerhalb des Funktions- oder Schleifenscopes deklarieren und ihr gleichzeitig einen Wert zuweisen (siehe Abschnitt *Persistente lexikalische Variable*).

Deklarationen in Schleifenköpfen sind von einer solchen Einmal-Zuweisung ebenfalls ausgenommen. In Code wie

```
while (state $input = <STDIN>)
```

wird der Variable - wie mit `my` - bei *jedem* Schleifendurchlauf ein neuer Wert zugewiesen. Hier ist eine Deklaration mit `state` daher sinnlos.

Die Deklaration von Typen und Attributen ist identisch zu der mit `my`:

```
state Test $var;
state @array:Attrib;
```

Allerdings ignoriert der Parser derzeit Attributdeklarationen mit `state` und interpretiert sie als mit `my` deklariert.

Ferry Bolhár-Nordenkampf

[1] Derzeit (Perl 5.10.1) werden folgende lexikalische Pragmas mit Perl mitgeliefert:

```
autodie
bytes
charname
feature
filetest
integer
local
open
overload
re 'taint'
re 'eval'
sort
strict
utf8
vmsish
warnings
```

Auch das nur mit mit Perl 5.9 ausgelieferte `assertions` Pragma gehört zu dieser Gruppe.

[2] Mit Ausnahme des durch die Pragmadatei bereitgestellten Scopes (sonst hätte die Änderung ja nur innerhalb des Pragmacodes Wirkung).

`%^H` wird nur lokalisiert, wenn zumindest eines seiner Hashelemente angelegt oder modifiziert wird.

[3] Die Bedeutung der einzelnen Bits von `$^H` kann den jeweiligen Pragmadateien entnommen werden. Allerdings können sich diese Bits von Perl-Version zu Perl-Version ändern. Daher sollte man keinesfalls die Bits direkt ansprechen, sondern immer die jeweiligen Pragmas, die von den Modulautoren aktuell gehalten werden, verwenden.

Perl 5.10 unterstützt das Schreiben eigener lexikalischer Pragmas (unter Nutzung von `%^H` und einer erweiterten `caller` Funktion, die eine Referenz auf `%^H` an Modulcode weiterreicht). Siehe `perldoc perlpragma`.

[4] Obwohl Perl anonyme Funktionen sehr wohl in Scratchpads ablegt. Darauf hat man als Programmierer allerdings keinen Einfluss.

[5] Seit Perl 5.10 kann man `$_` auch als lexikalische Variable (`my $_;`) deklarieren.

[6] Die Module `Devel::Lexalias` und `Lexical::Alias` erlauben das Anlegen von Aliases auf lexikalische Variable.

[7] Fast identisch. Der entsprechende *Opcode* (der vom Perl-Interpreter ausgeführte Befehl zum Zugriff auf die Variable) wird als mit `our` deklariert gekennzeichnet - aber nur, damit später wieder ein korrektes Rückparsen (wenn aus dem kompilierten Binärcode wieder Skript-Anweisungen erzeugt werden, z.B. mit dem Modul `B::Deparse`) möglich ist!

Eine Shell für Perl
Devel::REPL

Oft möchte man eine Kleinigkeit in Perl ausprobieren, einfach mal testen, ob der Befehl überhaupt das gewünschte Ergebnis bringt oder nicht. Und dann jedesmal einen Einzeiler mit "perl -wle '...'" daraus machen ist zu viel Tipparbeit. Hier wäre eine Perl-Shell ganz nützlich. Ich möchte hier mal ein paar Möglichkeiten aufzeigen, wobei ich den Schwerpunkt auf `Devel::REPL` legen werde.

Das, den Modulen/Skripten zugrunde liegende, Prinzip nennt sich "REPL" und bedeutet "read-eval-print-loop". Diese lange Bezeichnung sagt schon ganz gut aus, was das Prinzip macht: Lese die Benutzereingaben, evaluiere sie, gebe das Ergebnis aus und mache das ganze immer wieder (in einer Schleife).

Für Perl gibt es - wie es für Perl typisch ist - mehrere Module/Skripte, die das leisten:

- PerlConsole
- ein selbst geschriebenes Skript wie unter http://sedition.com/perl/perl-shell.html
- der Perl-Debugger
- Devel::REPL

Vermutlich gibt es noch mehr Lösungen, aber ich möchte hier nur auf die letzten drei Sachen eingehen.

Ein selbstgeschriebenes Skript hat den Charme, dass man es selbst gemacht hat, ist also gut für das persönliche Empfinden. Ansonsten sollte man sich aber an andere Lösungen halten. Denn selbstgeschriebene Lösungen müssen selbst erweitert und gewartet werden.

Perl-Debugger

Der Perl-Debugger ist ganz nützlich für das Debugging von Programmen. In der Ausgabe 7 von $foo gab es auch schon eine Einführung von Thomas Fahle. Diese REPL kann man ganz einfach mit `perl -d 1` starten. Allerdings sollte man dabei auch beachten, dass der Debugger sich in einigen Punkten anders verhält als der normale Perl-Interpreter. Das sind zum Teil Bugs, zum Teil Features. Der geneigte Leser kann sich die Bug-Liste unter http://rt.perl.org anschauen. Für einfache Sachen ist der Debugger aber durchaus zu gebrauchen - neben den typischen Debugging-Aufgaben. So kann man einfache Befehle einfach mal testen:

```
perl -d -e 0

  DB<1> print "Hallo Foo-Magazin"
Hallo Foo-Magazin

  DB<2> print 6 * 7; print "\n"; print 6 x 7
42
6666666
```

Das Zeilenende wird als Anweisungsende aufgefasst, das übliche Semikolon am Zeilenende kann daher entfallen. Mehrere Anweisungen in einer Zeile müssen allerdings durch ein Semikolon getrennt werden.

Näher will ich hier gar nicht auf den Debugger eingehen.

Devel::REPL

Ich persönlich bevorzuge `Devel::REPL`, weil es die weitestgehenden Möglichkeiten bietet. So ist es auch möglich, eigene Plugins zu schreiben, um das Verhalten des Moduls zu verändern. Unter ActivePerl war die Installation bei meinem letzten Versuch nicht wirklich erfolgreich, weil ein paar Mo-

dule nicht in den PPM-Repositories zu finden waren. Das Modul verwendet `Moose`, so dass die Liste an Abhängigkeiten nicht gerade kurz ist.

Nach der Installation macht das Modul aber Spaß. Der einfachste Start ist der Aufruf von `re.pl`, das bei der Installation mitgeliefert wird.

```
C:\>perl re.pl

$ my $a = 3;
3
$ $a + 2;
5
$
```

Das Semikolon kann hier auch ausgelassen werden, da `Devel::REPL` die Befehle in einem Block ausführt. Also ist auch das hier gültig:

```
$ $a + 3
6
```

Das Modul kann aber noch mehr. `Devel::REPL` verfügt über ein Plugin-System, mit dem man das Verhalten der "Shell" noch erweitern kann. Ohne Plugins ist es nicht möglich, Befehle über mehrere Zeilen zu schreiben:

```
$ sub test {
Compile error: Missing right curly or square
bracket at (eval 234) line 5, at end of line
syntax error at (eval 234) line 5, at EOF

$
```

Um mehrzeilige Befehle zu ermöglichen, gibt es das Plugin `Devel::REPL::Plugin::MultiLine::PPI`. In der Standard `re.pl` wird das Plugin automatisch geladen. Schreibt man sich eine eigene REPL mit `Devel::REPL`, so kann man das Plugin laden:

```
use Devel::REPL;
my $repl = Devel::REPL->new;
$repl->load_plugin( 'MultiLine::PPI' );
$repl->run;
```

Wenn jetzt das Skript gestartet wird, kann man auch mehrzeilige Befehle eingeben:

```
C:\>perl repl.pl

$ sub test {

>  print "perl-magazin.de"

> }

$ test()
1
$ perl-magazin.de
```

Es werden schon etliche Plugins mit der Distribution mitgeliefert. Zwei für mich sehr interessante Plugins sind `Devel::REPL::Plugin::History`, mit dem man auf Befehle zurückgreifen kann, die man innerhalb der Session (dem aktuellen Programmlauf) ausgeführt hat.

```
C:\>perl repl.pl

$ my $a = 'test';
test
$ !1
my $a = 'test';
test
```

Und `Devel::REPL::Plugin::Nopaste`, mit dem der Inhalt der aktuellen Sitzung in einem Nopaste-Bereich der Öffentlichkeit zugänglich gemacht wird. Trifft man auf Probleme oder Fragen während man einige Sachen austestet, kann man mit einem einfachen Befehl die eigenen Tests veröffentlichen und in Foren und/oder IRC einfach auf die Nopaste-Seite verweisen. Die Veröffentlichung kann man mit dem Befehl `#nopaste` anstoßen (siehe Listing 1).

Trotz der Fehlermeldung ist der Code jetzt auf pastie.org zu finden.

Profile und RC-Dateien

Bei `Devel::REPL` kann man auch mit Profilen arbeiten. In diesen Profilen kann gespeichert werden, welche Plugins geladen werden sollen. Die Profile werden dann als Perl-Modul abgespeichert. Ein Beispiel-Profil ist in Listing 2 zu sehen.

```
C:\>perl repl.pl

$ my $a = 3
3
$ my $a = 3
3
$ #nopaste
There is no form numbered 2 at C:/strawberry/perl/site/lib/App/Nopaste/Service/P
astie.pm line 13
http://pastie.org/836669
$
```

Listing 1

```
package Devel::REPL::Profile::FooMagazin;

use Moose;
use namespace::clean -except => [ 'meta' ];

with Devel::REPL::Profile;

sub plugins {
    return qw/MultiLine::PPI Done RegexExplain/;
}

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

    $repl->load_plugin( $_ ) for $self->plugins;
}

1;
```

Listing 2

Das Profil wird dann mittels

```
C:\>perl re.pl --profile FooMagazin
```

eingebunden.

Die RC-Dateien (*Run Control*) werden üblicherweise dafür benutzt, um noch mehr Vorbereitungsarbeit zu erledigen. So kann man darin Beispielsweise Datenbank-Verbindungen aufbauen oder schon bestimmte Module oder auch REPL-Plugins laden. Die RC-Dateien werden standardmäßig in `$HOME/.re.pl/repl.rc` gesucht. Aber man kann eine solche Datei auch beim Start der REPL angeben:

```
C:\>perl re.pl --rcfile projektname.rc
```

Wie so eine Datei aussehen kann, wird hier gezeigt.

```
use CGI;
use DBI;

sub dbi_connect {
 my $dbh = DBI->connect( 'DBI:SQLite:test' )
 or die $DBI::errstr; $dbh
}
```

Und warum gibt es RC-Dateien und Profile? Mit den RC-Dateien kann man sehr gut auf Projektebene arbeiten. So kann man für jedes Perl-Projekt eine eigene RC-Datei anlegen, in der dann die spezifischen Module geladen werden. Beim Start der REPL gibt man dann nur noch an, welche RC-Datei geladen werden soll und schon hat man eine funktionierende Umgebung für das Projekt zur Verfügung. In den Profilen werden dann die persönlichen Einstellungen für die REPL gespeichert. So kann man für unterschiedliche Entwickler die gleiche RC-Datei verwenden, jeder hat aber sein eigenes Profil.

Ein eigenes Plugin schreiben

Weiter oben bin ich schon auf Plugins eingegangen. In diesem Abschnitt möchte ich zeigen, dass es sehr einfach ist, eigene Plugins zu schreiben. Allerdings sollte man schonmal etwas von `Moose` gehört haben. `Devel::REPL` ist mit `Moose` implementiert und dort gibt es die Methoden-Modifikatoren `before`, `around` und `after`. Damit ist es möglich, mit einfachen Mitteln vor und/oder nach einer Methode noch weiteren Code auszuführen.

Ein kurzes Beispiel in Listing 3.

```
#!/usr/bin/perl

{
    package Foo;

    use Moose;
    use LWP::Simple;

    my $url = 'http://perl-magazin.de';

    sub url {
        print $url;
    }

    after 'url' => sub {
        print "\n",
        join "\n",
        grep{ defined }head( $url );
    };
}

Foo->url;

C:\>perl after.pl
http://perl-magazin.de
text/html; charset=ISO-8859-1
Apache/2.2.3 (Debian)
C:\>
```

Listing 3

Als Beispiel Plugins werde ich zuerst ein Plugin schreiben, das nach der Ausführung den gerade ausgeführten Befehl ausgibt.

```
#!/usr/bin/perl

package Devel::REPL::Plugin::Done;
use Devel::REPL::Plugin;

use Data::Dumper;

use namespace::clean -except => [ 'meta' ];

my $code;

before 'eval' => sub { $code = $_[1] };

after 'print' => sub {
    my $self = shift;
    my $fh   = $self->out_fh;
    print $fh "\n" . $code;
};

__PACKAGE__
```

Hier sieht man die Anwendung von `before` und `after`. Das `eval` und `print` sind keine Perl-Built-in-Funktionen, sondern Methoden aus `Devel::REPL`. In der Methode Da man bei `print` nicht den Befehl übergeben bekommt, muss der Befehl abgefangen werden, bevor er ausgeführt wird (deswegen das `before 'eval'`). Die Ausgabe des evals wird in der `print`-Methode ausgegeben. Da der Befehl nach der Ausgabe angezeigt werden soll, muss hier das `after 'print'` verwendet werden.

```
C:\>perl repl.pl

$ #explain ^d{4}$
The regular expression:

(?-imsx:^d{4}$)

matches as follows:

NODE         EXPLANATION
-----------------------------------------------
(?-imsx:     group, but do not capture
             (case-sensitive)(with ^ and $
             matching normally) (with . not
             matching \n) (matching white-
             space and # normally):
-----------------------------------------------
  ^          the beginning of the string
-----------------------------------------------
  d{4}       'd' (4 times)
-----------------------------------------------
  $          before an optional \n, and the
             end of the string
-----------------------------------------------
)            end of grouping
-----------------------------------------------

$
```

Listing 4

Bei den ersten eigenen Plugins muss man gegebenenfalls einfach mal ausprobieren vor oder nach welcher Methode der eigene Code ausgeführt werden soll.

Als zweites Beispiel soll ein Plugin erstellt werden, das zu einem Regulären Ausdruck eine Erklärung ausgibt. Die Erläuterung liefert das Modul `YAPE::Regex::Explain`, auf das ich in Ausgabe 4 schonmal genauer eingegangen bin. Das Plugin soll den Befehl `#explain` in der REPL hinzufügen.

```
#!/usr/bin/perl

package Devel::REPL::Plugin::RegexExplain;
use Devel::REPL::Plugin;

use YAPE::Regex::Explain;

use namespace::clean -except => [ 'meta' ];

sub expr_command_explain {
  my ( $self, $eval, $code ) = @_;

  return YAPE::Regex::Explain
                    ->new($code)->explain
}

__PACKAGE__
```

Das Plugin heißt `Devel::REPL::Plugin::RegexExplain` und wird später mit `$repl->load_plugin( 'RegexExplain' );` geladen. Die Verwendung von `namespace::clean` ist nützlich, um den Namensraum des Moduls sauber zu halten. Alle importierten Funktionen - außer die `meta` Methode - werden aus dem Namensraum gelöscht. Die `meta`-Methode wird von MooseX::Object::Pluggable benötigt.

Der Subroutinenname `expr_command_explain` legt fest, dass man später mit `#explain` auf die Funktionalität des Plugins zugreifen kann. Das `expr_command_` ist hier der "Präfix", der `Devel::REPL` mitteilt, dass ein neues Kommando registriert wird. Und das `explain` ist das Kommando.

Mit diesem kurzen Plugin sind Erklärungen zu Regulären Ausdrücken über "explain" möglich - siehe Listing 4.

#Renée Bäcker

MODULE

Komplexe Datenstrukturen mit Data::Dumper sichtbar machen - Schluss mit HASH(0x814cc20)

Überblick

Der Inhalt von Datenstrukturen ist manchmal ein Rätsel, vor allem dann, wenn die Daten aus fremden Modulen eingelesen werden oder über weitverzweigte `if`-Konstrukte befüllt werden. Spätestens dann, wenn ein Hash Referenzen enthält bekommt man mit `print \%hash` nur noch die bekannte Hexausgabe `HASH(0x814cc20)` und man ist nicht viel schlauer als vorher. Hier kommt das Standardmodul `Data::Dumper` ins Spiel. Mit diesem Modul ist es ein Leichtes, komplexe Datenstrukturen vollständig und formatiert auszugeben. Es ist dabei egal, ob das Modul prozedural oder objektorientiert verwendet wird. Beide Möglichkeiten sind an den entsprechenden Stellen sehr hilfreich.

Die Methode Dumper

`Dumper` erzeugt eine formatierte Ausgabe. Zusammen mit `print` oder `die` können Programme leicht debugged werden. Skalare, Hashes und Arrays werden ihr als Liste in beliebiger Reihenfolge übergeben. In genau dieser Reihenfolge erfolgt die Ausgabe mit der Bezeichnung `$VARx`. Hashes und Arrays haben in Perl auch Listenstruktur. Daher ist es sinnvoll, diese als Referenz zu übergeben, damit die innere Struktur erhalten bleibt (siehe Listing 1).

```
use Data::Dumper;
my %hash = (miranda => 'Uranus V',
            ariel => 'Uranus I');
my @array = ('miranda', 'ariel', 'umbriel');
print Dumper \%hash, \@array;

$VAR1 = {
          'ariel' => 'Uranus I',
          'miranda' => 'Uranus V'
        };
$VAR2 = [
          'miranda',
          'ariel',
          'umbriel'
        ];
```

Listing 1

Da die Ausgabe mit vielen Variablen unübersichtlich werden kann, können diese auch selbst benannt werden. Dies geschieht am leichtesten in der objektorientierten Form. Die Daten werden in einer Arrayreferenz übergeben, die eigenen Namen ebenfalls. Da es sich um Referenzen handelt, werden die Daten per default als Skalare dargestellt. Übergibt man der Funktion Namen, denen ein * vorangestellt wurde, wird die Ausgabe angepasst, sodass Arrays mit dem @-Zeichen und Hashes mit dem %-Zeichen dargestellt werden (siehe Listing 2).

Formatierung der Ausgabe

Die Formatierung kann vielfältig angepasst werden. Die wichtigsten Methoden und Schalter werden kurz vorstellt. Es kann durchaus vorkommen, dass die Ausgabe eingeschränkt werden soll. In der Praxis gibt es meist Hashes mit einer sehr hohen Verschachtelungstiefe deren Keys sehr vie-

```
my %hash = (miranda => 'Uranus V',
            ariel => 'Uranus I');
my @array = ('miranda', 'ariel', 'umbriel');
print Data::Dumper->new([\%hash, \@array],
            [qw(myhash *myarray)])->Dump();

$myhash = {
            'ariel' => 'Uranus I',
            'miranda' => 'Uranus V'
          };
@myarray = (
             'miranda',
             'ariel',
             'umbriel'
           );
```

Listing 2

```
my $uranus = {
    miranda => {
        disc => 1948,
        name => { figure => 'the tempest'},
    }
};

$Data::Dumper::Maxdepth = 1;
print Dumper $uranus;

$Data::Dumper::Maxdepth = 2;
print Dumper $uranus;

$VAR1 = {
          'miranda' => 'HASH(0x800abc)'
        };
$VAR1 = {
          'miranda' => {
                'name' => 'HASH(0x800174)',
                'disc' => 1948
                }
        };
```

Listing 3

len Daten enthalten. Das macht das Ergebnis schnell unübersichtlich, wenn zum Beispiel nur die Struktur der obersten Hierarchieebene interessiert. Der Schalter `MaxDepth` bietet die Möglichkeit, die Verschachtelungstiefe einzuschränken (siehe Listing 3).

Enthält ein Hash dagegen sehr viele Keys, sollen diese möglicherweise sortiert ausgegeben werden. Mit der Funktion `Sortkeys` werden einerseits die Keys in einer Defaultorder angezeigt. Andererseits lässt sich hier auch eine Subroutine angeben, welche je Hash mit einer Referenz auf diesen aufgerufen wird. Zurückgegeben wird eine Arrayreferenz, welche die auszugebenden Keys und deren Reihenfolge enthält. Somit wird die Ausgabe nicht nur sortiert, sondern bei Bedarf auch gefiltert. Keys, deren Inhalt nicht interessiert, können also ausgeschlossen werden.

```
$Data::Dumper::Sortkeys = 1;
$Data::Dumper::Sortkeys =
                    \&my_filter_routine;
```

Cross-Referenzen werden in der Defaulteinstellung nicht aufgelöst. Demzufolge ist auch die Ausgabe unübersichtlich. Wird `Deepcopy` auf 1 gesetzt, werden die Cross-Referenzen ausgeschrieben und das Ergebnis ist besser lesbar (siehe Listing 4).

Weitere Schalter sind `Terse`, `Indent` und `Purity`. Wird `Terse` auf 1 gesetzt, werden keine Variablennamen ausgegeben, selbst dann nicht, wenn explizit welche angegeben wurden. `Indent` steuert wie "schön" die Strukturen dargestellt werden. Wird der Wert auf Null gesetzt, bleiben zwar die Namen erhalten, jedoch ist jegliche Formatierung ausgeschaltet, d.h. die gesamte Datenstruktur wird in einer Zeile ausgegeben. Liegt eine rekursive Datenstruktur vor, lässt sich diese mit `Purity` anzeigen.

Fazit

Die Erfahrung hat gezeigt, dass `Data::Dumper` in der Praxis ein nicht wegzudenkendes Tool ist. Einerseits lassen sich schnell und einfach komplexe Datenstrukturen darstellen. Andererseits bietet es auch Unterstützung beim Schreiben von POD. Beispielsweise bei der Beschreibung des Formats der Übergabeparameter von Methoden, können diese "gedumped" und dann als wiederverwendbarer Code in die POD eingefügt werden.

#Colin Hotzky

```
my $ring = { ring => 'yes'};
my $moon = { moon => 'yes'};
my $gas  = { gas => 'yes'};

my $solar = {
    earth => [$moon],
    saturn => [$ring, $moon, $gas],
    uranus => [$ring, $moon, $gas],
};

# ohne Deepcopy
print Data::Dumper->new([$solar], [qw(Solar)])->Dump();
# mit Deepcopy
$Data::Dumper::Deepcopy = 1
print Data::Dumper->new([$solar], [qw(Solar)])->Dump();

$Solar = {
           'earth' => [
                        {
                          'moon' => 'yes'
                        }
                      ],
           'saturn' => [
                         {
                           'ring' => 'yes'
                         },
                         $Solar->{'earth'}[0],
                         {
                           'gas' => 'yes'
                         }
                       ],
           'uranus' => [
                         $Solar->{'saturn'}[0],
                         $Solar->{'earth'}[0],
                         $Solar->{'saturn'}[2]
                       ]
         };
$Solar = {
           'earth' => [
                        {
                          'moon' => 'yes'
                        }
                      ],
           'saturn' => [
                         {
                           'ring' => 'yes'
                         },
                         {
                           'moon' => 'yes'
                         },
                         {
                           'gas' => 'yes'
                         }
                       ],
           'uranus' => [
                         {
                           'ring' => 'yes'
                         },
                         {
                           'moon' => 'yes'
                         },
                         {
                           'gas' => 'yes'
                         }
                       ]
         };
```

Listing 4

WxPerl Tutorial - Teil 3: Sizer

Übersicht

Thema der heutigen Folge sind *Sizer*, auch Layout- oder Geometriemanager genannt. Sie sind jene fleißigen Helfer, die meistens für eine schöne und saubere Anordnung der Bedienelemente auf einer Fläche eingesetzt werden. Da sie sehr vielseitig sind und WxWidgets einige von ihnen kennt, ist dies Stoff genug für die folgenden Seiten.

Layout ohne Sizer

Anfang der letzten Folge beschrieb ich, wie Programmierer starre Fenster erhalten. Denn ein Verändern der *Frame*-Größe hätte in diesem Beispiel ein unschönes Bild mit verdeckten Widgets oder ungleichmäßig verteiltem Freiraum gezeigt. Mit Hilfe des Ereignisses `EVT_SIZE`, das bei jeder Größenänderung ausgelöst wird, könnte man sich noch wie folgt helfen und die Größen der *Widgets* manuell anpassen (siehe Listing 1)..

Aber das ist sehr umständlich und zeigt deutlich wozu *Sizer* erfunden wurden. Es ist auch nicht empfehlenswert diese Art der Widgetplatzierung mit *Sizern* zu kombinieren, da selbstverwaltete Widgets für *Sizer* unsichtbar sind und es zu unschönen Überlagerungen von Widgets kommen kann.

```
use strict;
use warnings;

package ServusWelt;
use Wx qw/ :everything /;
use base qw(Wx::App);

sub OnInit {
    my $app   = shift;
    my $frame = Wx::Frame->new( undef, wxDEFAULT, 'Servus Welt',[-1,-1],[100,100]);
    my $panel = Wx::Panel->new( $frame, -1);
    my $knopf1 = Wx::Button->new($panel,-1,'a',[5, 5],[30,20]);
    my $knopf2 = Wx::Button->new($panel,-1,'b',[5,27],[30,20]);
    Wx::Event::EVT_SIZE( $panel, sub {
        my ($panel, $event) = @_;
        my $size = $panel->GetSize();
        $knopf2->SetSize($size->GetWidth-20, $size->GetHeight - 30);
    });

    $frame->Center();
    $frame->Show(1);
    $app->SetTopWindow($frame);
    1;
}

package main;
ServusWelt->new->MainLoop;
```

Listing 1

Der Pate Teil I

Sizer sind nicht sichtbare Objekte, die Position und Größe ihnen anvertrauter Widgets bei einem `EVT_SIZE` eines zugeordneten Widget regulieren. Daher sind es eher Paten. Direkte Eltern eines Widgets (erster Parameter bei `new`) bleiben die Panel oder ähnliche Flächen, auf denen die Widgets platziert werden.

Die Parameter 3 und 4 werden überflüssig, sollten aber auf `[-1,-1]` oder `wxDefaultPosition` und `wxDefaultSize` gesetzt werden und nicht auf `-1`, was ein **variable is not of type Wx::Size** provozieren würde. Natürlich können diese Angaben auch ganz weggelassen werden, nur folgt ihnen der oft entscheidende *Style*-Parameter. Da es umständlicher ist, mit `$widget->SetWindowStyle(...)` oder `$widget ->SetWindowStyleFlag(...)` und vielleicht noch abschließendem `$widget->Refresh()` das Verhalten und Aussehen eines Widget festzulegen, gibt man einfacher den Stil bei der Erzeugung an und füttert die beiden vorangehenden Parameter jeweils mit dem üblichen `[-1,-1]`.

Perlenkettensizer

Der einfachste *Sizer* ist der `Wx::BoxSizer`, der mit einem Faden vergleichbar, die Widgets in einer Linie von oben nach unten oder links nach rechts organisiert. Um die Richtung zu entscheiden, schreibt man entweder:

```
my $sizer = Wx::BoxSizer->new( wxVERTICAL );
```

oder

```
my $sizer = Wx::BoxSizer->new
                              ( wxHORIZONTAL );
```

Ein `$sizer->GetOrientation` kann diese Information jederzeit abfragen. Ist der *BoxSizer* einmal erstellt, können mit `Add` die Widgets fleißig aufgefädelt werden. Das ansonst funktionsgleiche `Prepend` fügt sie am jeweils anderen Ende (oben oder links) an. `Insert` nimmt einen ersten, zusätzlichen Parameter, der per Index oder Widgetreferenz angibt, welchem Widget das Neue vorangestellt wird. Ähnlich tauscht `Replace` Widgets aus. Ein kleines Hütchenspiel mit 4 Knöpfen ($button1 .. $button4) zur Probe:

```
$sizer->Add( $button1, ... );
# Anzeige: $button1
$sizer->Prepend( $button2, ... );
# Anzeige: $button2, $button1
$sizer->Insert( 1, $button3, ... );
# Anzeige: $button2, $button3, $button1
$sizer->Replace( 2, $button4, ... );
# Anzeige: $button2, $button3, button4
```

Die 4 Parameter, die alle diese 4 Befehle kennen, gehören zum Wichtigsten, was ein Anfänger verstehen sollte. Der erste ist eine Referenz auf das Widget, oder einen anderen *Sizer* der z.B. eine Teilgruppe von Widgets waagerecht verwaltet, wenn der übergeordnete mehrere Gruppen senkrecht anordnet.

Der zweite Parameter nennt sich *proportion* und sollte immer gesetzt werden, auch wenn er optional ist, da sonst mannigfaltige unbeabsichtigte Effekte auftreten werden. Er gibt an, in welchem Verhältnis sich die Widgets den zusätzlich verfügbaren Platz aufteilen. Das gilt aber nur für die Richtung (waagerecht oder senkrecht), die dieser `Boxsizer` verwaltet. Ist er auf 0 gesetzt, behält das Widget seine Größe unverändert, selbst wenn die Fenstergröße verändert wird. Ist der Parameter bei einem Widget 2 und bei einem anderen auf 3 gesetzt, so teilen sie sich den Platz im Verhältnis 2:3. Um zu verhindern, dass Widgets zu klein werden, sollten dem übergeordneten `Panel` oder `Frame` Mindestgrößen befohlen werden:

```
$frame->SetMinSize([$x, $y]);
```

Aber auch *Sizer* akzeptieren diesen Befehl und bei komplexeren Layouts sollten die verschachtelten *Sizer* auch jeweils eigene Mindestgrößen bekommen. Der dritte Parameter ist der *Style*, was 3 Dinge betrifft: Ausrichtung, Ränder und Wachstum. Die Konstantennamen `wxALIGN_LEFT`, `wxALIGN_RIGHT`, `wxALIGN_TOP`, `wxALIGN_BOTTOM`, `wxALIGN_CENTER_VERTICAL` und `wxALIGN_CENTER_HORIZONTAL` sagen alles. `wxALIGN_CENTER` fasst die letzten beiden zusammen und es gibt zu allen Zentrierungen noch welche mit der Schreibweise `CENTRE` anstatt `CENTER`. Ebenso selbsterklärend geben `wxLEFT`, `wxRIGHT`, `wxTOP`, `wxBOTTOM` oder zusammenfassend `wxALL` die Richtung an, in der der *Sizer* dem Widget zusätzlichen Rand gibt.

Der vierte Parameter sagt wie groß dieser Rand ist - in Pixel. Weiterhin gibt es noch die Stylekonstanten `wxGROW` und `wxSHAPED`. Mit ersterer oder auch `wxEXPAND` bekommt das Widget allen erhältlichen Platz, auch in der Richtung, die

nicht von *proportion* verwaltet wird. Mit zweiterem wird zusätzlich darauf geachtet, dass das Höhen-Breite-Verhältnis des Widget nicht verzerrt wird.

Manchmal ist es aber zu wenig einem Widget einfach in 3 Richtungen 10 Pixel Rand zu geben. Dann können mit

```
$sizer->AddSpacer( $px );
```

oder

```
$sizer->InsertSpacer( $pos, $px );
```

auch zusätzliche Ränder eingefügt werden. Wenn es ein gieriger Rand mit *proportion* sein soll, geht das mit:

```
$sizer->PrependStretchSpacer( $prop );
```

Es können auch jederzeit Widgets, Spacer und Subsizer einzeln mit `Detach` oder kollektiv mit `Clear` entfernt werden. `Clear` erhält einen Parameter der, falls positiv, den Abbau aller verwalteten Widgets erlaubt.

Ist der *Sizer* endlich zur Zufriedenheit ausgestattet, wird er mit `SetSizer` einem Widget oder Fenster zugeordnet, der dann sozusagen der Boss (caput di capi) dieses Paten wird, der mit `GetContainingWindow` einsehbar ist. Mit dem `SetSizer` wird gleichzeitig ein `SetAutoLayout(1)` ausgeführt. Ändert nun der Boss seine Größe, muss der *Sizer* reagieren und wird seine Schützlinge dirigieren. Dieses Verhalten kann jederzeit mit `SetAutoLayout(0)` ausgesetzt werden. Werden einzelne Widgets mit `Hide(1)` oder `Show(0)` unsichtbar, entsteht in der Raumaufteilung des Fensters ein Loch.

Trotz eines aktivierten `AutoLayout` muss der Befehl `Layout` manuell aufgerufen werden, damit dieses Loch verschwindet, da der *Sizer* auf keine `Show`-Befehle der Kinder reagiert, sondern nur auf den `EVT_SIZE` des Bosses.

Manchmal sollte aber der Boss auf die Kinder hören, also das Fenster oder Panel der minimal benötigten Größe der Widgets angepasst werden. Das erreicht:

```
$size = $sizer->Fit($frame);
```

welches leider nicht rekursiv arbeitet, also manuell von der untersten (Widgets) zur obersten Ebene (Frame) aufgerufen werden muss, wenn jedes Widget wirklich nur den benötigten Platz bekommen soll.

Weitere Lineare Sizer

Vom `StaticBoxSizer` gibt es noch die Variante mit Concilieri, also einen Rahmen, der die Kindwidgets umgibt. Da ein solcher Rahmen auch noch ein Label besitz, kann dies beim Erzeugen nach Orientierungsrichtung und Elternfenster angegeben werden. Es geht aber auch nachträglich mit `$sizer->GetStaticBox->SetLabel( $label );`.

Die vom *Sizer* mitverwaltete `StaticBox` bleibt nicht nur ein eigenständiges Objekt sondern sie kann auch vorher erzeugt werden und beim Erzeugen dem *Sizer* zugewiesen werden (`$sizer->new($staticBox, wxVERTICAL);`).

In älterem Code könnt ihr noch den `wxNotebookSizer` antreffen. Dieser achtete darauf, dass `NoteBook` (Karteikasten mit Reitern) immer groß genug für jede Seite sind. Da das ein `NoteBook` nun selber kann, darf auf den `Wx::Panel`, die den Reiterseite entsprechen, jeder beliebige `Sizer` verwendet werden. Dieser *Sizer* gilt daher als veraltet und seine Verwendung wird nicht empfohlen.

Neuer dagegen ist der `StdDialogButtonSizer`. Dessen Konstruktor erhält keine Parameter, da er immer waagerecht seine Knöpfe anordnet. Er wurde erfunden um vor allem Dialoge zu bauen, in denen die Knöpfe der wichtigen Befehlszeile (Ja, Nein,Abbrechen) in Reihenfolge und Abständen die Vorgaben des Betriebssystems einhalten.

Deshalb sollte die Fütterung per `AddButton` immer ein `Realize` abschließen. Auch ist darauf zu achten, dass die zugefügten Knöpfe eine der folgenden ID's haben: `wxID_OK`, `wxID_YES`, `wxID_SAVE`, `wxID_APPLY`, `wxID_CLOSE`, `wxID_NO`, `wxID_CANCEL`, `wxID_HELP`, `wxID_CONTEXT_HELP`. Ist dies nicht möglich, so kann ein bereits zugefügter Knopf mit `$sizer->SetCancelButton($knopf);` als Abbrech-Kopf markiert werden. Ähnlich ersetzt `SetAffirmativeButton` `wxID_YES` und `SetNegativeButton` `wxID_NO`.

Der Pate Teil II

Wie angedeutet können mehrere `Wx::BoxSizer` verschachtelt werden um komplexere Oberflächen zu aufzubauen. Oft geht das auch einfacher mit dem `Wx::GridSizer`, was einer

guten alten *HTML*-Tabelle ähnelt. Einzige Schwäche des Vergleiches: hier sind alle Zeilen und Spalten gleich groß. Genauer: das breiteste Widget bestimmte die Breite aller Spalten und das Höchste, die Höhe aller Zeilen. Alle Zeilen und alle Spalten haben jeweils auch den gleichen zusätzlichen Abstand untereinander (zustätzlich zum Abstand der Widgets zum Zellenrand).

Und jede Zelle kann nur ein Widget aufnehmen, denn es entspricht einer Position des Sizers. Diese Positionen lassen sich von links nach rechts und zweitranging von oben nach unten abzählen, was Bedeutung bekommt wenn man einzelne Widgets mit `$sizer->Insert($pos, $widget)` ins Regal tun will. Ansonsten kann es mit `Add` fortlaufend befüllt werden. Möchte man dabei ein Feld leer lassen, verwendet man `Add(0,1)`, was einem `AddSpacer(1)` entspricht. Falls ihr einmal ein seltenes `AddWindow` sehen solltet, dies ist nur eine Form von `Add` die nur Widgets zulässt. Da der `Wx::GridSizer` im Gegensatz zum `Wx::BoxSizer` den Platz in 2 Dimensionen verwaltet, hat der zweite Parameter von `Add` hier keine Bedeutung und im Dritten lässt `wxGROW` das Widget in beide Richtungen platzgierig sein.

Wer mehr Abwechslung bei der Platzeinteilung mag, der greife zum `Wx::FlexGridSizer`. Seine Erzeugung gleicht dem `Wx::GridSizer`. Die 4 Parameter geben Anzahl der Zeilen, Spalten, Zeilenabstand in Pixel und Spaltenabstand an. Diese Zeilen und Spalten werden sich auch wie ein `Wx::GridSizer` verhalten. Nur jene nachträglich mit `AddGrowableRow` und `AddGrowableCol` hinzugefügten, verhalten sich ähnlich den *Strechspacer* oder Widgets mit Angabe im zweiten Parameter (*proportion*). Doch da hier die gesamte Spalte nur eine Breite haben kann, wird die *proportion* als zweiter Parameter von `AddGrowableCol` angegeben, gleich nach der Position der Platzgierigen Spalte. Die andere Methode funktioniert entsprechend genauso. `RemoveGrowableRow` und `RemoveGrowableCol` können zur Laufzeit das Zugefügte wieder entnehmen. Nur ein abschließendes `$sizer->Layout();` nicht vergessen.

Wer bei all dem das gute alte `HTML`-artige *colspan/rowspan* vermisst, braucht den `Wx::GridBagSizer`. Er ist quasi ein `wxFlexGridSizer`, dessen `Add`-Methode etwas mächtiger ist. Die aus `Wx::BoxSizer` bekannten Parameter 1 (Widget), 3(*Style*) und 4 (Randbreite) belegen hier die Plätze 1, 4 und 5. Auf 2 und 3 sind Position und Ausdehnung des Widget. Das ist etwas umständlich gelöst, da beides ein extra Objekt ist, dessen Erzeugung sich aber in eine *sub* mit besonders kurzem Namen auslagern lässt. Gerade bei mehrmaligem Gebrauch lohnt sich anstatt:

```
$sizer->Add(
    $knopf[1], Wx::GBPosition->new(0,0),
    Wx::GBSpan->new(1,2), wxGROW|wxALL, 15
);
```

es so zu lösen:

```
sub gbpos  { Wx::GBPosition->new(@_) }
sub gbsize { Wx::GBSpan->new(@_)     }
$sizer->Add($knopf[1], gbpos(0,0),
            gbsize(1,2), wxGROW|wxALL, 15);
```

Diese Zeilen lassen sich auch durch Wiederverwendung der Positions- und Größenobjekte kürzen. Und nicht vergessen: die Positionen beginnen bei (0,0) aber die Spanne muss immer mindestens (1,1) sein.

Alle vorgestellten Arten von Tabellen haben den Nachteil gemein: nur ein Widget je Zelle ist erlaubt. Sollen es mehrere werden, muß man schachteln, wie zu Anfang dieses Kapitels angedeuted. Dazu übergibt man die Widgets einem *Sizer*, der darauf mit `Add()` oder Ähnlichem dem übergeordneten *GridSizer* unterstellt wird.

In der nächsten Folge wird das auch an einem Beispiel demonstriert, denn die nächste Folge wird sich mit den Unterlagen der Sizer beschäftigen. Dies können einfache `Wx::Panel` sein, aber auch verschiedenste Fenster. Und Panel lassen sich auch zu Reiterleisten und vielem mehr organisieren. Dabei werden neben einfachen Knöpfen noch weitere Widgets in Erscheinung treten.

Herbert Breunung

App::cpanminus - Erweiterbarer Zero Conf Installer für CPAN-Module

Zur Installation von CPAN-Modulen werden üblicherweise die bekannten und bewährten Module *CPAN* oder *CPANPLUS* verwendet. Für weniger erfahrene Perl-User ist die Vielzahl der Konfigurationseinstellungen allerdings manchmal verwirrend.

Das noch sehr junge *App::cpanminus* von Tatsuhiko Miyagawa vereinfacht die Installation von CPAN-Modulen erheblich, indem es automatisch eine sinnvolle Konfiguration wählt und Abhängigkeiten ebenfalls automatisch auflöst.

Durch ein flexibles Plugin-System lässt sich *App::cpanminus* darüber hinaus bequem an die eigenen Bedürfnisse anpassen.

Installation cpanminus

cpanminus lässt sich via CPAN/CPANPLUS oder bevorzugt direkt aus dem Sourcecode-Repository installieren:

```
$ git clone \
    git://github.com/miyagawa/cpanminus.git
$ cd cpanminus
$ perl Makefile.PL
$ sudo make install
```

CPAN-Module mit cpanm installieren

CPAN-Module lassen sich nun über das mitgelieferte Kommandozeilenprogramm `cpanm` installieren - siehe Listing 1.

Der gesamte Installationsprozess wird in die Datei *build.log* im Verzeichnis *~/.cpanm/* geloggt. *cpanminus* zeigt auf der Konsole nur die wichtigsten Informationen an und stellt keine weiteren Fragen.

So einfach kann es sein.

Weitere Optionen und Handbuchseiten erhält man via `perldoc App::cpanminus` oder `cpanm --help`.

local-lib - CPAN-Module in ein eigenes Verzeichnis installieren

Ebenfalls sehr einfach ist die Installation von CPAN-Modulen in ein eigenes Verzeichnis - auch ohne administrative Rechte - über den Schalter *--local-lib* - siehe Listing 2.

```
$ sudo cpanm CGI

Fetching http://search.cpan.org/CPAN/authors/id/L/LD/LDS/CGI.pm-3.49.tar.gz ... OK
Configuring CGI.pm-3.49 ... OK
==> Found dependencies: Test::More, FCGI
Fetching http://search.cpan.org/CPAN/authors/id/M/MS/MSCHWERN/Test-Simple-0.94.tar.gz ... OK
Configuring Test-Simple-0.94 ... OK
Building and testing Test-Simple-0.94 for Test::More ... OK
Successfully installed Test-Simple-0.94 (upgraded from 0.62)
Fetching http://search.cpan.org/CPAN/authors/id/M/MS/MSTROUT/FCGI-0.69.tar.gz ... OK
Configuring FCGI-0.69 ... OK
Building and testing FCGI-0.69 for FCGI ... OK
Successfully installed FCGI-0.69
Building and testing CGI.pm-3.49 for CGI ... OK
Successfully installed CGI.pm-3.49 (upgraded from 3.15)
```

Listing 1

Die so installierten Module lassen sich wie gewohnt durch das Pragma *lib*

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

use lib '/home/tf/project/lib/perl5';

print "\@INC:\n";
print join( "\n", @INC ), "\n";
```

oder über die Umgebungsvariable *PERL5LIB* einbinden.

```
$ export PERL5LIB=\
               "/home/tf/project/lib/perl5"
```

Wenn `cpanm` ohne administrative Rechte und ohne Angabe von *--local-lib* aufgerufen wird, wird automatisch *~/perl5* als Installationsverzeichnis verwendet - so wie man es auch schon von *local::lib* kennt.

Plugins - App::cpanminus erweitern

cpanminus lässt sich über Plugins einfach erweitern. Die Dokumentation ist noch recht spärlich, aber man kann sich ja den Sourcecode der mitgelieferten Beispiele ansehen. Plugins werden einfach in das plugin-Verzeichnis *~/.cpanm/plugins/* kopiert. Zur Aktivierung der so installierten Plugins muss derzeit noch die Umgebungsvariable `PERL_CPANM_DEV` gesetzt werden.

```
$ export PERL_CPANM_DEV=1
```

Eine Übersicht der installierten Plugins liefert die Option *--list-plugins*

```
$ cpanm --local-lib=/home/tf/projekt
        --list-plugins
Using git_site_perl
git_site_perl -
         Hooks to keep site_perl in git
minicpan -
         fallback to local minicpan when
         a dist is not found
```

plugin minicpan

Dieses Plugin bindet ein bereits vorhandenes `CPAN::Mini`-Repository ein. Eine aufwendige Konfiguration entfällt, da *minicpan* die benötigten Information aus der vorhandenen *.minicpanrc*-Datei im Heimatverzeichnis des aktuellen Nutzers ausliest.

```
$ cpanm --local-lib=/home/tf/project CGI

Attempting to create directory /home/tf/project
Attempting to create file /home/tf/project/.modulebuildrc
==> Found dependencies: ExtUtils::MakeMaker, ExtUtils::Install
Fetching http://cpan.hexten.net/authors/id/M/MS/MSCHWERN/ExtUtils-MakeMaker-6.56.tar.gz ... OK
Configuring ExtUtils-MakeMaker-6.56 ... OK
Building and testing ExtUtils-MakeMaker-6.56 for ExtUtils::MakeMaker ... OK
Successfully installed ExtUtils-MakeMaker-6.56 (upgraded from 6.30_01)
Fetching http://cpan.hexten.net/authors/id/Y/YV/YVES/ExtUtils-Install-1.54.tar.gz ... OK
Configuring ExtUtils-Install-1.54 ... OK
Building and testing ExtUtils-Install-1.54 for ExtUtils::Install ... OK
Successfully installed ExtUtils-Install-1.54 (upgraded from 1.33)
Fetching http://cpan.hexten.net/authors/id/L/LD/LDS/CGI.pm-3.49.tar.gz ... OK
Configuring CGI.pm-3.49 ... OK
Building and testing CGI.pm-3.49 for CGI ... OK
Successfully reinstalled CGI.pm-3.49
```

Listing 2

```
$ export PERL_CPANM_DEV=1
$ export PERL_MM_OPT="INSTALL_BASE=/home/tf/project"
$ cpanm --local-lib=/home/tf/project CGI
Using git_site_perl
Initializing git repository for /home/tf/project
Fetching http://cpan.hexten.net/authors/id/L/LD/LDS/CGI.pm-3.49.tar.gz ... OK
Configuring CGI.pm-3.49 ... OK
Building and testing CGI.pm-3.49 for CGI ... OK
Successfully reinstalled CGI.pm-3.49
Committing updates in /home/tf/project to git...
```

Listing 3

plugin git_site_perl

Mein persönliches Lieblingsplugin *git_site_perl* ermöglicht die automatische Verwendung der Versionsverwaltungssoftware *git* für die zusätzlich installierten CPAN-Module in *site_perl*. Ein eigenes *site_perl* lässt sich über die Umgebungsvariable *PERL_MM_OPT* angeben - siehe Listing 3.

Die automatisch erzeugten Commit-Messages überzeugen durch durch ihre schlichte Klarheit (Listing 4).

END { }

Für produktive Systeme kann ich *cpanminus* derzeit noch nicht uneingeschränkt empfehlen, dazu ist das Projekt noch zu jung.

Die bestechende Einfachheit und die umfangreiche Unterstützung durch erfahrene und bekannte Perl-Entwickler lassen mich vermuten, dass *cpanminus* der kommende CPAN-Installer-Superstar sein wird.

Thomas Fahle

```
$ cd project/
$ git log
commit 3c73d13d59315a355855126b140b766be19fcd79
Author: Thomas Fahle <info@thomas-fahle.de>
Date:   Sat Mar 20 10:46:24 2010 +0100

        CGI.pm 3.49 (CGI)

commit ...
```

Listing 4

Merkwürdigkeiten in Perl - $1 und Co.

Das Speichern von Treffern in Regulären Ausdrücken ist ein wichtiges Feature, denn so kann man gewisse Daten aus einem String herausziehen, ohne den String zu verändern. Was gespeichert wird, wird mit runden Klammern "()" im Regulären Ausdruck festgelegt.

```
'test' =~ /.(..)/;
```

Auf den gespeicherten Teilstring kann man dann mit `$1`, `$2`, ... zugreifen. Dabei gilt, dass die erste Gruppe in `$1` landet, die zweite in `$2` und so weiter.

Viele stolpern darüber, wann `$1` und Konsorten neu gesetzt werden. Denn immer wieder sieht man Fragen, warum bei einem Code wie

```
'test' =~ /t(es)t/;
print $1;
'test' =~ /h(al)lo/;
print $1;
```

zweimal "es" ausgegeben wird. Viele erwarten stattdessen, dass beim zweitenmal `$1` `undef` ist, weil der Reguläre Ausdruck ja nicht gematcht hat.

Durch diese Erwartungen trifft man häufiger mal auf Code wie

```
'test' =~ /t(es)t/;
# ganz viel Code
'test' =~ /h(al)lo/;

unless( $1 ) {
    print "hallo ist nicht enthalten";
}
```

In den meisten Fällen möchte der Programmierer damit aber ausdrücken "wenn 'test' nicht auf 'hallo' matcht und deshalb 'al' nicht in $1 landet, dann gehe in den unless-Block".

Da bei einem nicht-erfolgreichen Match `$1` nicht neu gesetzt wird, steht hier immer noch 'es' in `$1`. Deshalb sollte man besser damit arbeiten, die Treffer in "normalen" Variablen zu speichern:

```
'test' =~ /t(es)t/;
# ganz viel Code
my ($al) = 'test' =~ /h(al)lo/;

unless( $al ) {
    print "hallo ist nicht enthalten";
}
```

Hier wird die lexikalische Variable `$al` wirklich nur dann gesetzt wenn der Match erfolgreich ist.

Es gibt aber eine weitere Sache, die man dabei beachten muss: `$1` ist an den Regulären Ausdruck gebunden. Man liest immer wieder, dass `$1` an den Scope gebunden ist. Aber man sollte hierbei aber die Ausgabe von folgendem Code betrachten:

```
my $regex = qr/ie(.)/;
level1();

sub level1 {
    'sieben' =~ $regex;
    level2();
    print $1;
}

sub level2 {
    'vier' =~ $regex;
}
```

Hier wird ein Objekt von einem Regulärer Ausdruck erzeugt und dieser RegEx wird dann in zwei unterschiedlichen Subroutinen verwendet. Hier würde also jeweils ein neuer Scope erzeugt. Die Ausgabe ist aber nicht "b", sondern "r". `$1` wird hier also in der Subroutine `level2` belegt. Das zeigt, dass `$1` an die RegEx gebunden ist, nicht an den Scope.

Das zeigt, dass man bei der Verwendung von `$1` immer auf Auswirkungen an ganz anderen Codestellen achten muss.

Renée Bäcker

Neues von TPF

Grants abgebrochen: Tcl/Tk in Rakudo und Perl Cross-Compilation für WinCE und Linux

Da die Grants schon seit September 2008 laufen und Vadim Konovalov auf keine Mails mehr reagiert, wurden diese Grants geschlossen.

Grant-Update: Perl für Windows in Unternehmen (14.03.2010)

Es gibt Neuigkeiten zu dem Grant von Curtis Jewell. Wobei einige "Neuigkeiten" eigentlich schon älter sind:

• Eine Installationsdatei, die auch in anderen .msi-Installern eingebettet werden kann: Das wurde mit dem Januar 2010 Release von StrawberryPerl erreicht
• Installation in beliebiges Verzeichnis: Daran arbeitet Jewell gerade. Er versucht, das in das April 2010 Release mit einzubauen.
• Ein setup für local::lib inkl. Windows Registry: Einen Ansatz davon gibt es schon im SVN-Repository.

Grant Update: Perl-Umfrage (14.03.2010)

Kieren Diment arbeitet gerade an den letzten Feinheiten des Codes für die Perl-Umfrage.

Das System für die nächste Umfrage soll bald laufen und in eine Pilotphase starten.

Der Code ist auf Github verfügbar: http://github.com/singingfish/perl-survey-2009

Grant akzeptiert: Bugfixing am Perl 5 Kern

Die Perl Foundation hat den Grant Vorschlag von Dave Mitchell akzeptiert. Er wird 500 Stunden daran arbeiten, Bugs in Perl zu fixen. Als Pumpking von Perl 5.10.1 kennt er sich sehr gut in den Internas von Perl aus und kann auch komplexe Bugs identifizieren und fixen.

Grant abgeschlossen: Archive::Zip Bugfixing

Alan Haggais Arbeit an Archive::Zip ist abgeschlossen. Er hat unzählige Bugs gefixt und das Modul kann jetzt mit mehr Daten umgehen. Eine Entwicklerversion sollte bald auf CPAN verfügbar sein. Eine stabile Version sollte dann bald darauf folgen.

Ergebnisse der Grant-Vorschläge für das 1. Quartal 2010

Im Blog der Perl Foundation wurden die Ergebnisse der Abstimmung über die Grant-Vorschläge des 1. Quartals 2010 veröffentlicht:

Zwei Vorschläge wurden angenommen und 5 Vorschläge wurden abgelehnt: Der Perl Compiler Vorschlag von Reini Urban wurde erst in die Warteschlange verschoben, dann hat cPanel erklärt, diesen Grant zu unterstützen. Ricardo Signes wird an seinem Modul Dist::Zilla weiterarbeiten.

Grant abgebrochen: pyYAML nach Perl portieren

Ingy döt Net wollte pyYAML nach Perl portieren. Nachdem länger nichts mehr passiert ist, haben sich die TPF und Ingy darauf geeinigt, den Grant abzubrechen.

Die Hälfte des Geldes wurde ausgezahlt, da die Hälfte der Aufgaben erledigt wurde.

Hague Grant abgeschlossen

Jonathan Worthington hat seinen "Hague Grant" abgeschlossen. Im Zuge dieses Grants hat Worthington die Methodensignaturen verbessert.

Die Arbeit an dem Grant hat sich verzögert, weil Jonathan Worthington verschiedene Branches berücksichtigen musste.

Grant Manager gesucht

Die Perl Foundation ist auch der Suche nach einem oder mehreren Grant Managern.

Grant Manager sind für die Kommunikation zwischen Grant-Empfänger und Grant-Komitee zuständig. Sie müssen hin und wieder bei den Projekten nachfragen wie der aktuelle Stand ist.

Interessenten sollen sich an ambs@perlfoundation.org wenden.

CPAN News XIV

App::whichpm

Hin und wieder möchte man wissen, wo ein Modul liegt und in welcher Version es installiert ist. Man kann das schnell selbst programmieren oder `App::whichpm` nehmen.

```
#!/usr/bin/perl

use App::whichpm qw(which_pm);

my @modules = qw(Test::More Catalyst);
for my $module ( @modules ) {
    my ($file,$version) = which_pm(
                                $module );
    print "$file -> $version\n";
}
```

autovivification

Autovivification ist eine ganz nette Sache. Gerade Anfänger wundern sich aber über dieses Verhalten von Perl. Und es gibt durchaus Fälle, da ist autovivification nicht erwünscht. Für diese Fälle gibt es das gleichnamige Modul. Im ersten Teil wird der Schlüssel "test" im Hash angelegt. Nach dem "no autovivification" passiert das nicht mehr.

```
use Data::Dumper;

my %hash;
if ( $hash->{test}->{foo} ) {
    print 'foo';
}
print Dumper \%hash;

no autovivification;

my %info;
if ( $info->{test}->{foo} ) {
    print 'foo';
}
```

Acme::Base64

Auch aus der Acme::-Ecke gibt es mal wieder ein tolles Modul: Wer wollte nicht schon immer mal in Base64 programmieren? Das ist jetzt möglich. Natürlich ist das wie die ähnlichen Module `Acme::Bleach` und Konsorten nicht wirklich geeignet, um seinen Code zu schützen, aber so etwas ist doch immer wieder ganz nett anzuschauen.

```
use Acme::Base64;

cHJpbnQgIkhlbGxvIHdvcmxkIVxuIjsK
```

Log::Dispatch::Dir

Logging hilft bei der Suche nach Fehlern und ist während der Entwicklung sehr nützlich. Für Module wie`Log::Log4perl` gibt es unzählige "Backends" wohin die Logmeldungen gespeichert werden können. `Log::Dispatch::Dir` sorgt dafür, dass pro Meldung eine Datei erzeugt wird. Diese Dateien werden in Verzeichnissen gespeichert, die von dem Modul erzeugt werden. Für das Erstellen der Verzeichnisse gibt es auch einen Mechanismus für Log-Rotating. Nützlich kann das Modul sein, wenn die Meldungen nicht nur aus einer Zeile stehen, sondern z.B. aus kompletten HTTP-Requests.

```
use Log::Dispatch::Dir;

my $dir = Log::Dispatch::Dir->new(
    name => 'dir1',
    min_level => 'info',
    dirname => 'somedir.log',
    filename_pattern =>
          '%Y%m%d-%H%M%S.%{pid}.html',
);
$dir->log( level => 'info',
           message => 'your comment\n" );
```

Perl::PrereqScanner

Welche Abhängigkeiten hat mein Modul? Welche Module muss ich für meine Anwendung installieren? Mit `Perl::PrereqScanner` lässt sich Perl-Code auf diese Abhängigkeiten untersuchen. Man kann einen String, ein PPI-Dokument oder eine Datei angeben...

```
use Perl::PrereqScanner;
my $scan    = Perl::PrereqScanner->new;
my $prereqs = $scan->
    scan_ppi_document( $ppi_doc );
my $prereqs = $scan->
    scan_file( $file_path );
my $prereqs = $scan->
    scan_string( $perl_code );
```

Test::Vars

Man entwickelt und entwickelt und entwickelt. Dabei fällt einem gar nicht auf, dass man irgendwelche Variablen deklariert hat, die man dann gar nicht benutzt. Für globale Variablen gibt es Warnungen vom Perl-Interpreter, bei lexikalischen Variablen aber nicht. Abhilfe kann hier `Test::Vars` bringen.

```
use Test::Vars;

all_vars_ok(); # check libs in MANIFEST
```

Termine

Mai 2010

01.-02. Nordischer Perl-Workshop
04. Treffen Vienna.pm
Treffen Frankfurt.pm
06. Treffen Dresden.pm
10. Treffen Ruhr.pm
11. Treffen Stuttgart.pm
17. Treffen Erlangen.pm
19. Treffen Munich.pm
Treffen Darmstadt.pm
25. Treffen Bielefeld.pm
26. Treffen Berlin.pm

Juni 2010

01. Treffen Vienna.pm
03. Treffen Dresden.pm
04.-05. Portugiesischer Perl-Workshop
07.-09. Deutscher Perl-Workshop
08. Treffen Frankfurt.pm
Treffen Stuttgart.pm
15. Treffen Ruhr.pm
16. Treffen Darmstadt.pm
21. Treffen Erlangen.pm
26. Belgischer Perl-Workshop
29. Treffen Bielefeld.pm
30. Treffen Berlin.pm

Juli 2010

01. Treffen Dresden.pm
06. Treffen Frankfurt.pm
Treffen Vienna.pm
12. Treffen Ruhr.pm
13. Treffen Stuttgart.pm
19. Treffen Erlangen.pm
21. Treffen Darmstadt.pm
Treffen Munich.pm
27. Treffen Bielefeld.pm
28. Treffen Berlin.pm

Hier finden Sie alle Termine rund um Perl.

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

Uhrzeiten und weiterführende Links zu den einzelnen Veranstaltungen finden Sie unter

http://www.perlmongers.de

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

termine@foo-magazin.de

LINKS

http://www.perl-nachrichten.de

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

http://www.perl-community.de

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

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

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

http://www.perl-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.

// SEIBERT / MEDIA

BESSERE ATMOSPHÄRE? MEHR FREIRAUM?

Wir suchen erfahrene Perl-Programmierer/innen (Vollzeit)

//SEIBERT/MEDIA besteht aus den vier Kompetenzfeldern Consulting, Design, Technologies und Systems und gehört zu den erfahrenen und professionellen Multimedia-Agenturen in Deutschland. Wir entwickeln seit 1996 mit heute knapp 60 Mitarbeitern Intranets, Extranet-Systeme, Web-Portale aber auch klassische Internet-Seiten. Seit 2005 konzipiert unsere Designabteilung hochwertige Unternehmensauftritte. Beratungen im Bereich Online-Marketing und Usability runden das Leistungsportfolio ab.

Ihre Aufgabe wird sein, in unserer Entwicklungsabteilung im Team komplexe E-Business Applikationen zu entwickeln. Dabei ist objektorientiertes Denken genauso wichtig, wie das Auffinden individueller und innovativer Lösungsansätze, die gemeinsam realisiert werden.

Wir freuen uns auf Ihre Bewerbung unter www.seibert-media.net/jobs.
// SEIBERT / MEDIA GmbH, Rheingau Palais, Söhnleinstraße 8, 65201 Wiesbaden
T. +49 611 20570-0 / F. +49 611 20570-70, bewerbung@seibert-media.net

„Statt mit blumigen Worten umschreiben unsere Programmierer den Job so:

Apache, Catalyst, CGI, DBI, JSON, Log::Log4Perl, mod_perl, SOAP::Lite, XML::LibXML, YAML“

YAPC::Europe::2010

My One Hotel Galilei, Pisa, August 4-6

- Four tracks over three days!
- Dozens of talks!
- Special guests! *Damian Conway*, *Allison Randal*, *Dave Rolsky* and *Larry Wall*... meet them all!
- Pre- and post- training courses!
- Recruiting sessions!
- *...and legendary coffee breaks worth dueling for!*

The Renaissance of Perl

For more info: http://yapceurope.org/2010