Ausgabe 2/2009 – www.foo-magazin.de - 6,00 €

$foo

PERL MAGAZIN

Interview mit Richard Dice
über "The Perl Foundation"

GUI-Tests mit Perl
Linux GUI's automatisch testen

Test::Class Best Practices
Testen leicht gemacht

Nr 10

http://yapceurope2009.org/

YAPC::EU::2009

Corporate Perl

YAPC::EU::2009, the tenth edition of the largest **European Perl Conference**.

Join us for three days of talks, workshops, and more, among hundreds of Perl programmers from all over Europe, in sunny **Lisbon, Portugal**.

Whether you're a Perl Beginner or a Perl Expert, a developer or a sysadmin, whether your interests are in Web Development, Databases, Object Oriented Perl, MVCs, Perl 6, or something else that is Perl related this conference has something for you.

Come and see what everybody else is doing.

Come and learn what everybody else is using.

Share the experiences and make your life better.

Learn. Evolve.

3 to 5 August - Lisbon, Portugal

* The YAPC::EU::2009 Logo celebrates YAPC's second year in Portugal. It includes a Perly version of the Portuguese coat of arms, replacing the five "quinas" with the five sigils of Perl ($, @, %, &, *), and the seven castles with camels.

Design: Alberto Simões

Ist Perl tot?

Immer wieder hört man diese Frage oder den gleichen Satz mit Ausrufezeichen. Die einen sagen "Ja" und führen verschiedene Statistiken wie den TIOBE-Index oder die neuesten Zahlen von O'Reilly an (die Links dazu sind auf unserem delicious-Account zu finden). Andere sagen "Nein". Doch wer hat Recht?

Ich denke, dass eine objektive Aussage darüber nicht zu treffen ist. Sicherlich hat Perl ein Image-Problem. Vieles bleibt innerhalb der Perl-Community und nach außen (z.B. Manager, Programmierneulinge) dringt kaum etwas. Aber das hat ja im Prinzip nichts damit zu tun, ob Perl wirklich tot ist.

Die Zahlen von O'Reilly sind Fakten, auch wenn man natürlich die Randbedingungen wie "Wurden Perl-Bücher im Katalog aufgeführt?", "Gab es Neuerscheinungen?" betrachten muss. Der TIOBE-Index ist mehr als fragwürdig, wird aber häufig zitiert und führt wiederum zu der öffentlichen Meinung dass Perl tot sei. Und Heise hat in seinem neuesten Webspecial Perl nicht mal erwähnt, hat aber einen sehr guten Artikel von Susanne Schmidt über die Zukunft von Perl online gestellt.

Auf der anderen Seite gibt es bei den Perl 5 Porters ziemlich viel Aktivität, an Rakudo (Perl 6 auf Parrot) wird fleißig gearbeitet, die Anzahl der jährlichen Perl-Events steigt stetig und die Aktivität auf CPAN hat sich seit 2006 verdreifacht. Auch die Nachrichtenlage bei Perl-Nachrichten.de zeigt deutlich, dass Perl nicht tot sein kann.

Meiner Meinung nach ist vieles eine Sache des Standpunktes. Sicherlich hat der Anteil an Webseiten, die in Perl programmiert werden, prozentual abgenommen, aber sind nur Programmiersprachen für Webanwendungen vital?

Ich versuche durch vielfältige Aktivitäten zu zeigen, dass Perl noch lange nicht tot ist. Und wie heißt es so schön? "Totgesagte leben länger".

Jeder Perl-Interessierte kann seinen Teil dazu beitragen, dass der Spruch wahr ist.

Renée Bäcker

Die Codebeispiele können mit dem Code

3i8gj7

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/issue10.

IMPRESSUM

Herausgeber:	Smart Websolutions Windolph und Bäcker GbR Maria-Montessori-Str. 13 D-64584 Biebesheim
Redaktion:	Renée Bäcker, Katrin Blechschmidt, André Windolph
Anzeigen:	Katrin Blechschmidt
Layout:	//SEIBERT/MEDIA
Auflage:	500 Exemplare
Druck:	powerdruck Druck- & VerlagsgesmbH Wienerstraße 116 A-2483 Ebreichsdorf
ISSN Print:	1864-7537
ISSN Online:	1864-7545

INHALTSVERZEICHNIS

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

Renée Bäcker

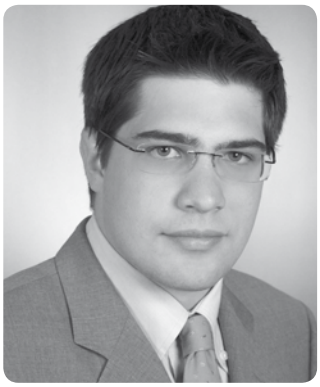
Seit 2002 begeisterter Perl-Programmierer und seit 2003 selbständig. Auf der Suche nach einem Perl-Magazin ist er nicht fündig geworden und hat so diese Zeitschrift herausgebracht. In der Perl-Community ist Renée recht aktiv – als Moderator bei Perl-Community.de, als Organisator des kleinen Frankfurt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

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.

Marcus Holland-Moritz

Geboren 1977 in Thüringen und seit Ende der 80er Jahre dem Programmieren verfallen (damals noch auf einem C16). Die Jahrtausendwende hat ihn in den Raum Stuttgart verschlagen, wo er seitdem Software für Patientenmonitore in C und C++ entwickelt. Perl hat er dabei eher zufällig (und zuerst widerwillig) entdeckt; mittlerweile schreibt er jedoch die meisten Tools, die er zum Arbeiten braucht, in Perl. Er ist Autor mehrerer CPAN-Module und auch bei den perl5-porters aktiv.



Marc Koderer

Marc Koderer (26) ist frisch gebackener M.Sc. der Informationstechnik und hatte den ersten Kontakt mit Perl als Linux Server Admin. Seit dem ist er begeisterter Perl Programmierer. Er ist Autor des Perl Moduls `X11::GUITest::record` und beschäftigt sich vorzugsweise mit dem Software Testing im Linux Bereich (von GUIs bis zum Linux Kernel).

Theo Ohnsorge

Theo Ohnsorge ist geschäftsführender Gesellschafter der Exelution GmbH mit Sitz in München. Er zeichnet für die technische Leitung verantwortlich. Die Exelution GmbH betreut seit 2005 ihre Kunden in dem Bereich Performance-Marketing (SEM, SEO, Affiliates, Portal-Kooperationen, Produktsuchmaschinen) und berät ihre Kunden strategisch im Bereich der effizienten Nutzung von Marketing-Spendings im Internet. Für die Steuerung aller Marketing-Aktivitäten im Internet wurde eine vollständig auf Perl basierende Applikation sowohl für Back- und Frontend entwickelt.

Curtis 'Ovid' Poe

Curtis "Ovid" Poe ist ein sehr bekannter Perl-Programmierer und der Originalautor des neuen `Test::Harness`. Er ist außerdem Mitglied des Perl Foundation Grant Committee und repräsentiert die BBC beim Parrot Foundation Advisory Board. Er spricht häufig auf Konferenzen und hält Schulungen über automatisiertes Testen und solides OO Design.

INTERVIEW

Richard Dice: Update über "The Perl Foundation"

Renée Bäcker (RB): Hallo Richard, es freut mich Dich wieder zu sehen. Seit unserem letztem Interview 2007 ist viel passiert. Ich möchte gerne auf 2008 zurückblicken und einen Ausblick auf das Jahr 2009 geben. Fangen wir mit dem Rückblick an. Denkst Du, dass 2008 ein gutes Jahr für Perl war?

Richard Dice (RD): Hallo Renée, schön, wieder mit Dir zu sprechen.

War 2008 ein gutes Jahr für Perl? Ich möchte das gerne in drei Teilen beantworten: Perl 5, Perl 6 und The Perl Foundation.

Das Jahr begann gut für Perl 5, beginnend mit dem Release von Perl 5.10.0 Ende 2007. Im November 2008 wurde das Release von Perl 5.10.1 beschlossen. David Mitchell wird mit Mitteln aus einem Perl Foundation Grant von Dijkmat BV in den Niederlanden unterstützt. Dies ist ein bedeutender Schritt für den Core von Perl 5.

Ich bin auch sehr darauf gespannt, was mit Perl 5 in Sachen "Module" passiert. Catalyst beginnt eine wohldurchdachte, reife Plattform zu werden und wird immer mehr angenommen (Viele Stellenangebote, die 2008 auf jobs.perl.org in 2008 veröffentlicht wurden, listen Erfahrungen mit Catalyst auf. Dies war in den letzten Jahren nicht der Fall.). Moose ist ein herrliches Objekt-System und beginnt auch Schwung aufzunehmen (TPF unterstützt im Moment ein Projekt im Moose Bereich – *http://www.perlfoundation.org/dave_rolsky_moose_docs*).

Perl 6 hat definitiv die Kurve bekommen. Rakudo, die Perl 6 Implementation von Parrot VM, begann 2008 mit Patrick Michaud, dem leitenden Programmierer. Er wurde von TPF und der Mozilla Foundation unterstützt. Dies hilft dem Projekt enorm. Später in 2008 unterstützten wir Patrick und weitere Rakudo Entwickler mit Hague Grants. Ich bemerke, dass sich der Ton der Diskussionen über Perl 6 ändert - dank des Vorstoßes von Rakudo in den letzten Jahren und des Entwicklerschwungs. Es sieht so aus als gäbe es nicht mehr so viele "Perl 6 will never happen" Nein-Sager. Diejenigen, die es noch sagen, tendieren dazu anonyme Feiglinge und Trolle zu sein.

Für die Perl Foundation war 2008 auch ein gutes Jahr. Wie bereits oben erwähnt, war es uns möglich Schlüsselprojekte in Perl 5 und Perl 6 mit bedeutenden Grants zu unterstützen. Es war unser bestes Jahr an Mittelbeschaffung, das wir je hatten. Unsere Prozesse sind so effizient, wie sie noch nie waren. Dies macht es für Leute in der Community jetzt einfacher TPF zu kontaktieren und mit TPF zu sprechen. Wir wurden von der IT Industrie und der Firmenpresse kontaktiert und nach Erläuterungen und Beratungen gefragt. Wir haben unsere Perspektive dargestellt. Auch unsere Kommunikation mit der Perl Community ist so gut wie noch nie. Hier hat unsere Webseite *http://news.perlfoundation.org/* sehr geholfen.

RB: Im Dezember 2007 wurde Perl 5.10 veröffentlicht und das Medieninteresse war groß. Viele News-Seiten haben Ankündigungen veröffentlicht und viele Blogger schrieben über ihre ersten Erfahrungen mit Perl 5.10. Denkst Du, dass diese Berichte die Aufmerksamkeit auf Perl gelenkt haben? Hat die Perl Foundation die Möglichkeit solche Dinge zu messen? Was denkst Du über Kennziffern wie den TIOBE-Index?

RD: Die weit verbreitete Ankündigung des Perl 5.10.0 Releases war eine gemeinsame Leistung vom Perl 5 Porters 5.10 Release Team und The Perl Foundation. Es war für Perl's 20. "Geburtstag" (18. Dezember 2007) geplant. Dies war eine sehr gute Gelegenheit, die nicht verpasst werden durfte.

Ich weiß nicht, ob das Perl 5.10 Release oder die begleitende Pressekampagne die Pressetexte, Blogposts usw. über Perl im Jahr 2008 hervorgerufen hat, aber ich stimme zu, dass Perl im letzten Jahr mehr Aufmerksamkeit bekommen hat als in den letzten Jahren zuvor. Die Perl Foundaton "misst" das ganze genauso wie jeder andere es könnte - lesen von Blogs und Artikeln in Webmagazinen und durch Setzen von Lesezeichen. Ich denke, dass wir vielleicht mehr als andere Leute das Ganze in einen größeren Kontext setzen. Ich habe 2007 Vorträge auf der YAPC::NA und der YAPC:EU über Perl's Platz in der IT Welt, durch Analyse von öffentlichen Daten und Anwendung strategischer Modell auf die gefundenen Daten belegt, gehalten.

Der TIOBE Index ähnelt einem Puzzle. Ich habe kein Problem mit Leuten, die versuchen Rankings über Programmierspracheneinsatz, -verbreitung, -trends oder -verwendung machen. Während einige Personen der Perl Community den Nutzen dieser Bemühungen bestreiten, bin ich gerne dazu Bereit den Leser des Rankingsystems nach Sachdienlichkeit filtern zu lassen. Für die Leser, die nichts über die speziellen Zweifel von vielen Perl-Leuten gegenüber dem TIOBE-Index wissen, sei gesagt, dass es viel mit dem Perlranking der letzten Jahre und dem erheblichen Rückgang in 2008 zu tun hat. Außerdem sind einige der TIOBE Rankingdaten nicht sehr einleuchtend (Delphi??), was einen natürlichen Spielraum für Zweifel mit sich bringt. Dass Tim Bunce, ein führendes Mitglied der Perl Community, die TIOBE Ergebnisse mit ihren veröffentlichen Rankingdaten nachzustellen versucht hat - er es aber nicht konnte, macht TIOBE zu einem Puzzle.

Ich will schon seit einiger Zeit mal mit den Personen hinter dem TIOBE Index in Kontakt kommen, um aus erster Hand zu erfahren wie der TIOBE Index funktioniert und wie Perl zu dem Ranking kommt, das es hat. Aufgrund des wenig intuitiven Ergebnisses von Perl im Ranking ist es möglich, dass es ein systematischer Fehler ist. Beispielsweise passierte so etwas ähnliches mit Perl und der automatischen Datensammlung von Ohloh.net. Die besagte, dass Perl ein wenig aktives Projekt ist. Das machte für Perl Community Mitglieder keinen Sinn. Es stellte sich heraus, dass Ohloh.net den p5p's Perforce Source Code Repository nicht prüfte. Jetzt, wo Perl's Quellen und Geschichte verwaltet wird, sollte Perl's Einordnung in Ohloh.net sich ein wenig verändern. Etwas Ähnliches kann mit TIOBE passieren, wenn wir mit ihnen sprechen.

RB: 2008 hat die Perl Foundation zwei große Geldsummen bekommen: US$ 200.000 von Ian Hague und US$ 50.000 von Booking.com. Kannst Du uns etwas über diese Mittel sagen und wie ihr (TPF) das Geld ausgegeben habt?

RD: Die Spende von US$ 50.000 von booking.com bekamen wir ohne feste Vorgaben. booking.com hat aber zum Ausdruck gebracht, dass sie speziell Perl 5.10 unterstützen wollen. Dies ist sicherlich auch ein Anliegen der TPF und wir werden nach Möglichkeiten Ausschau halten, dies zu tun. Sobald wir sie finden, wird ihre großzügige Spende sicherlich hilfreich sein, um sie zu unterstützen. Wie bereits vorhin erwähnt, ist das offensichtlichste was Perl 5.10 braucht, das Release von Perl 5.10.1. Dies wird bereits von einem Grant der TPF unterstützt. Wenn das Release von 5.10.2 Unterstützung der TPF benötigt, werden die Geldmittel von booking.com sicherlich hierfür verwendet.

Die US$ 200.000 Spende von Ian Hague war wesentlich stärker gegliedert. Sie ist speziell für die Perl 6 Unterstützung gedacht. In Vereinbarung mit Hr. Hague haben wir die Spende in zwei Hälften geteilt. Eine Hälfte ist für die direkte Unterstützung der Perl 6 Entwicklung und der Entwicklungsbemühungen gedacht. Zurzeit haben wir verschiedene Entwickler, die das nutzen: Daniel Ruoso arbeitet an SMOP, und weitere Entwickler, wie Patrick Michaud, Jerry Gay und Jonathan Worthington, entwickeln Rakudo. Die zweite Hälfte des Geldes wird von der TPF für interne Entwicklungen genutzt. Das passiert vor dem Hintergrund, dass TPF mehr und bessere Sachen tun muss, um Perl einer größeren Welt zu repräsentieren, als es im Moment möglich ist. Dies beinhaltet Ausgaben für Angestellte, ausgegliederte Services (z.B. Marketing-Berater, Buchhaltung), Büro- und Lieferungsausgaben, IT und Ausgaben, um an Konferenzen und Geschäftsmeetings teilzunehmen.

RB: Gabor Szabo hat vor einiger Zeit eine Diskussion über die Transparenz der TPF gestartet. Ich denke viele Leute - auch Perlentwickler - wissen nicht, was TPF macht. Das führt zu einem weiteren Thema, das Gabor ansprach. Er sagte, dass die TPF jährlich einen "fund drive" (Spendenaufruf) wie Wikipedia machen sollte. Was denkst Du über diese Idee? Würde eine "sichtbarere" TPF helfen, um mehr Geld zu bekommen?

RD: Eine Liste der Dinge, die TPF macht, ist sehr einfach:
• Wir tragen die rechtliche Verantwortung für verschiedene Markenzeichen und Copyrights,
• Wir unterstützen Perl Projekte mit Geldspenden, inklusive Perl 5 und Perl 6 Entwicklung,
• Wir betreiben die YAPC:NA Konferenz und wir unterstützen verschiedene Perlkonferenzen und Hackathons,
• Wir repräsentieren Perl im IT-Geschäftsumfeld und für Perlkunden.

Ich bin sehr offen für die Dinge, die Gabor aufbringt. Jeder möchte, dass die TPF Erfolg hat und mehr im Namen der Community macht. Wenn gute Ideen wie diese vorgebracht werden, aber nicht gleich umgesetzt werden, denken manche, dass das versteckte Ablehnung bedeutet. Das bedeutet es aber nicht. Es zeigt nur, dass die Mitglieder der TPF sehr beschäftigte Leute sind. Viele von uns haben Managementaufgaben in unseren jeweiligen Firmen, viele haben Babys oder kleine Kinder und Ehepartner, und einen Haushalt, um den sie sich kümmern müssen. Daher bleibt nur wenig Zeit für TPF Arbeit. In einer guten Woche kann ich persönlich ca. 5 Stunden für die TPF arbeiten. Gelegentlich mache ich eine Reise im Namen der TPF, z.B. war ich letzten Monat in New York um Ian Hague für die Spende zu danken und ihn darüber zu informieren, wofür das Geld genutzt wird. Das waren zwei fast volle Tage, die ich der TPF "gespendet" habe. Das ist mehr als die Zeit, die ich freiwillig jeden Monat aufbringen kann. Und ich tue es trotzdem.

TPF ist keine Dikussionsgesellschaft, die von der Öffentlichkeit abgeschirmt ist. Jede Person arbeitet über Monate still an einem Projekt und nimmt nur Kontakt mit anderen Personen auf, wenn Hilfe oder Beratung benötigt wird, was unregelmäßig vorkommt. Es gibt 3 Aspekte zum Thema Transparenz:
• Kommunikation über das, was die TPF macht,
• Einblick in den Prozess, wie es gemacht wird,
• Möglichkeit auf die TPF für Diskussionen zuzukommen.

Unser Kommunikationslevel mit der Perl Communitiy ist höher als je zuvor - dank des *http://news.perlfoundation.org/ Blogs*. Es gibt dort vielleicht zwei Postings im Monat - ich denke, dass ist eine vernünftige Repräsentation über die Dinge, die die TPF im Laufe eines Monats macht. Zum Thema Sichtbarkeit gibt es nicht viel zu sagen. Es gibt ungefähr zehn E-Mails in der TPF Steering Committee Liste im Monat, und diese beziehen sich mehr oder weniger auf interne Geschäfte und Prozesse: viel langweiliges Zeug, wie z.B. Diskussionen über das Kaufen von Büromaterial, über die Lieferantenauswahl für die T-Shirts der YAPC:NA, oder die Personalbesetzung für den TPF Messestand auf der OSCON. Manchmal geht es auch darum, neue Freiwillige zu finden, die Postings in der TPF machen. Wenn dies passiert, geht es meist schnell an die Öffentlichkeit über den news.perlfoundation.org Blog.

Bezüglich der Idee, wie z.B. Wikipedia zu Geldspenden aufzurufen - ja, durchaus, wir müssen hier ein Programm starten. Zurzeit arbeiten wir daran unser ganzes Spendensystem zu verbessern. Jim Brandt, der TPF Vice President, arbeitet bereits seit einigen Monaten daran. Wir hoffen, dass es in einigen weiteren Monaten fertig ist. Eine Menge Arbeit, die dort gemacht wird, ist juristisch und beinhaltet Vertragsverhandlungen mit unserem Spendensystemanbieter, der in unserem Buchhaltungssystem integriert wird. Eine "American 501©(3) Company" zu sein, bedeutet, dass es viele rechtliche Voraussetzungen zu beachten gibt, es ist nicht einfach nur Programmierung. Wir benötigen Sachkundige im juristischen Bereich und im Finanzbereich, die uns helfen, die Organisation zu führen und Verträge zu verhandeln. Ihre Rechnungen zu bezahlen bedarf großer Anstrengung.

Letztendlich möchten wir ein Programm starten, indem lokale Repräsentanten in ihren Ländern (oder sogar Städten) in der ganzen Welt Kontakt zu den lokalen Firmen herstellen, die Perl nutzen. So könnten wir direkt Spenden von Firmen bekommen. Mit Ausnahme der Hague-Spende, kamen die größten Spenden in der Vergangenheit von Betrieben. Sie sind so groß, dass eine große Firmenspende (wie die Spende von booking.com) mit vielen über Monate zusammengerechneten Spenden einzelner Personen gleichkommt oder übertrifft. Lokale Repräsentanten zu haben, scheint ein guter Weg zu sein, mehr Firmen zu erreichen. Aber das Netzwerk zum Laufen zu bekommen, es mit dem notwendigen Material zu versorgen und Backend-Systeme in der TPF fertig zu stellen, wird mit unserer jetzigen Anzahl an Freiwilligen Monate dauern. Aber es wird irgendwann einmal fertig.

RB: 2008 wurde die Parrot Foundation gegründet. Einige Mitglieder der Parrot Foundation sind auch Mitglieder der TPF und da Perl 6 (wie Rakudo) die "Haupt-"sprache in Parrot ist (jedenfalls im Moment), sollte klar sein, dass die beiden Organisationen zusammenarbeiten. Ist derartiges geplant?

RD: Die Parrot Foundation entstand aus einer Konversation zwischen Allison Randal (frühere TPF Präsidentin, z.Z. Board of Directors Mitglied der TPF, Vorsitzende des "Board of Parrot Foundation" und technische Architekturleiterin der Parrot VM) und mir. Anfang 2008 wollte Allison "Parrot" als ein Firmenname für "Yet Another Society" eintragen lassen. (Aus historischen Gründen ist "The Perl Foundation" ein Firmenname der Gesellschaft "Yet Another Society". Es ist erlaubt, mehr als einen registrierten Firmennamen für eine einzige Gesellschaft zu haben.) Zu Beginn dieses Prozesses fragte sie mich, was ich davon halte. Ich sagte ihr, dass es besser wäre, wenn die Parrot Foundation eigenständig sei. Dies würde der TPF ermöglichen Entscheidungen für Perl zu treffen, ohne über Situationen nachzudenken, in denen das Beste für Parrot nicht das beste für Perl sein könnte. Und ich dachte, dass Parrot als eigene Einheit besser wäre, weil es schon Beziehungen zu anderen Sprachen aufgebaut hat.

Zum Beispiel könnte die Python Community zögern Parrot als "first-class" VM-Plattform für Python zu nutzen, wenn Parrot Alleineigentum der TPF wäre. Mit diesen Ideen starteten wir die Strategie eine Parrot Foundation zu gründen, die unabhängig von der TPF verantwortlich für Parrot VM ist.

Wenn man das alles bedenkt, ja, dann arbeiten die TPF und die Parrot Foundation eng zusammen. TPF hat ein großes strategisches Interesse am Erfolg von Parrot weil Rakudo Perl 6 davon abhängig ist. Wir möchten, dass Parrot erfolgreich ist, weil das eine Voraussetzung für den Erfolg von Rakudo ist. Und es gibt natürlich auch enge persönliche Beziehungen zwischen Leuten der Parrot Foundation und den Leuten der TPF, auch wenn Allsion die einzige ist, von der ich weiß, dass sie wirklich Mitglied von beiden ist. Viele der Hague Perl 6 Geldmittel gehen an die Rakudo Perl 6 Implementation. Dies ist aber keine Vorliebe in Richtung Rakudo oder Parrot. Es ist nur, weil Rakudo Perl Mitglieder offensiver die TPF um Unterstützung gebeten haben, und weil TPF sehr beeindruckt von der Professionalität, dem Einsatz und der Software Engineering Disziplin aufweist. Wir wären sehr glücklich darüber jegliche Perl 6 Implementationanstrengung zu unterstützen, die diesen Standards entspricht.

RB: Richard, vielen Dank für das Interview. Es war wirklich interessant zu sehen, was die TPF im letzten Jahr gemacht hat und ich freue mich auf 2009.

Internationalisierung oder Lokalisierung?

Dies ist eine Art philosophische und damit im Kontext Perl gut aufgehobene Frage. Die Fragestellung behandelt im Kern den Prozess, ein beliebiges Ding von einer lokalen Ebene auf eine globale zu heben; oder eben genau das Gegenteil zu erreichen. Ein Beispiel für den ersten Fall - die Internationalisierung - kann Werbung multinationaler Konzerne sein. Hier ist das Ziel möglicherweise, eine über Ländergrenzen hinaus einheitliche Wahrnehmung für ein lokales Produkt zu kreieren. Für den zweiten Fall - die Lokalisierung - muss dann beispielsweise die Bedienungsanleitung des erwähnten Produktes für jedes Land in die jeweilige Sprache übersetzt werden.

Beide hier beschriebenen Fälle müssen in der Softwareentwicklung berücksichtigt werden, wenn es darum geht Programmausgaben in mehreren menschlichen Sprachen zu machen. Der erste Schritt, die Internationalisierung, erfolgt beim Erstellen der Software mit dem "auszeichnen" der zu übersetzenden Zeichenketten. Der zweite Schritt, die Lokalisierung, erfolgt durch die Identifikation und Übersetzung dieser Zeichenketten und der richtigen Ausgabe pro Sprache wenn die Anwendung läuft.

Der folgende Artikel zeigt anhand frei verfügbarer GNU-Tools und einigen Perl-Skripte, wie diese Aufgabe erledigt werden kann und wie diese Vorgehensweise auch für das beliebte Framework Catalyst genutzt werden kann.

An dieser Stelle noch ein Wort zu den beiden in der Literatur, Foren und sonst wo auftauchenden Kürzeln *I18N* und *L10N*. Beides sind Kurzschreibweisen (jeweils der Anfangs- und Endbuchstabe und dazwischen die Anzahl der ausgelassenen Buchstaben als Zahl) für die beiden englischen Begriffe "Internationalization" und "Localization".

Alle benutzten GNU-Werkzeuge sind in dem Paket "gettext" enthalten und können zum Beispiel in Ubuntu einfach mit `sudo apt-get install gettext` installiert werden. Dieses Paket enthält die für uns relevanten Programme `xgettext`, `msginit` und `msgmerge` die nun genauer vorgestellt werden.

xgettext

Mit diesem Programm werden aus allen vom Nutzer festgelegten Dateien oder Verzeichnissen gekennzeichnete ("ausgezeichnete") Textstrings ausgelesen und in das Quelllexikon "message.pot" geschrieben. Wird ein Verzeichnis angegeben, so wird dieses rekursiv abgearbeitet. Es können mehrere Verzeichnisse angegeben werden. Bei diesem Prozess werden Textdubletten erkannt und unter Angabe vom Fundort (Datei, Zeile) zusammengefasst. Die Dateinamenserweiterung bedeutet im Übrigen "GNU Gettext Portable Object Base Translation". Ein typischer Aufruf sieht in etwa folgendermaßen aus:

```
xgettext --directory=/dir1
    --directory=/dir2 --ouput=message.pot
```

Alle definierten Quelldateien in den Verzeichnissen `/dir1` und `/dir2` werden nach Textstrings durchsucht. Das Ergebnis wird dann in der Datei message.pot gespeichert. Diese Datei enthält dann Einträge in dieser Form:

```
# Quelldatei:Zeilennummer
Msgid "Originaltext"
Msgstr ""
```

Standardmäßig geht `xgettext` davon aus, dass es sich bei den zu durchsuchenden Dateien um C-Quelltextdateien handelt. Mit der Option `-L` (`--language=`) kann eine andere vordefinierte Programmiersprache angemeldet werden - xgettext sucht dann nach den entsprechenden Ausgabemustern für diese Programmiersprache. Es können mit dem Parameter `-k` (`--keywords=`) auch eigene Muster an `xgettext` übergeben werden. So extrahiert der folgende Ausdruck alle Zeichenketten in den `print` Statements.

```
xgettext --directory=/mydir
    --keyword=print --ouput=message.pot
```

Wer alle Strings aus seinen Quelltexten extrahieren möchte, bedient sich der Option `-a`.

Damit die Zeichenketten zur Laufzeit des Programms in der richtigen Sprache ausgegeben werden können, müssen diese über eine Lokalisierungsfunktion, welche gleichzeitig die Auszeichnung der betreffenden Texte übernimmt, aufgerufen werden. Hier ein Perl-Beispiel flightcontrol.pl (das Modul `Locale::gettext` muss installiert sein):

```
#!/usr/bin/perl

use warnings;
use strict;
use Locale::gettext;

print gettext("Launch rocket."), "\n";
printf "%6s%.2f%4s%2s", gettext("Speed: "),
   19000.434343, gettext("mph"),  "\n";
warn gettext("Call Houston, Texas."), "\n";
die gettext("Apollo 13 Scenario:
   Stack overflow."), "\n";

exit;
```

Mit der Funktion `gettext()` wird `xgettext` mitgeteilt, dass die nachfolgende Zeichenkette in die Datei message.pot aufgenommen werden soll. Alle Steuerzeichen, wie hier die Newlines, sollten sich nicht in der Zeichenkette befinden, da der Übersetzer diese gegebenenfalls vergisst, oder nicht versteht. Auch die Benutzung von `printf` ist fraglich und wird an dieser Stelle nur der Möglichkeit wegen gezeigt, da die Längen von Zeichenketten für Wörter in den Sprachen variieren - aus Speed wird zum Beispiel Geschwindigkeit. Die zu obigem Beispiel gehörende message.pot wird durch den folgenden Aufruf erstellt:

```
xgettext flightcontrol.pl
    -output=message.pot
```

Dabei werden unter anderem die folgenden Zeilen in der Datei message.pot erzeugt:

```
#: flightcontrol.pl:7
msgid "Launch rocket."
msgstr ""

#: flightcontrol.pl:8
msgid "Speed: "
msgstr ""

#: flightcontrol.pl:8
msgid "mph"
msgstr ""

#: flightcontrol.pl:9
msgid "Call Houston, Texas."
msgstr ""

#: flightcontrol.pl:10
msgid "Apollo 13 Scenario: Stack overflow."
msgstr ""
```

msginit

Durch dieses Programm werden aus dem Quelllexikon alle gewünschten Sprachlexika initial erzeugt. Es wird wie folgt aufgerufen:

```
msginit -i message.pot -o de.po -l de_DE
```

Dabei wird über den Parameter `-i` (input) das Quelllexikon festgelegt, über den Parameter `-o` (output) die Ausgabedatei definiert und mit dem Parameter `-l` (locale) der Sprachraum angegeben.

Das so erzeugte Sprachlexikon für deutsch de.po, enthält analog zu der Datei message.pot Einträge in dieser Form:

```
# Quelldatei:Zeilennummer
Msgid "Originaltext"
Msgstr ""
```

In dieser Datei können alle Zeichenketten in den Zeilen "Msgstr" übersetzt werden.

Die im Abschnitt xgettext erzeugte Datei message.pot kann nun wie folgt für die Deutsche Sprache initialisiert werden:

```
msginit -i message.pot -o de.po -l de_DE
```

Dabei werden die folgenden Zeilen erstellt, welche bereits übersetzt wurden.

```
#: flightcontrol.pl:7
msgid "Launch rocket."
msgstr "Starte Rakete."

#: flightcontrol.pl:8
msgid "Speed: "
msgstr "Geschwindigkeit"

#: flightcontrol.pl:8
msgid "mph"
msgstr "kmh"

#: flightcontrol.pl:9
msgid "Call Houston, Texas."
msgstr "Rufe Houston, Texas an."

#: flightcontrol.pl:10
msgid "Apollo 13 Scenario: Stack overflow."
msgstr "Apollo 13 Szenario: Stapelüberlauf"
```

msgmerge

In der Praxis wird es häufig vorkommen, dass während der Softwareentwicklung aber auch später Änderungen an den Textausgaben vorgenommen werden. Diese Änderungen können den einzelnen Sprachlexika durch das Programm msgmerge bekannt gemacht werden. Bevor es aufgerufen wird sollte `xgettext` ausgeführt werden, damit in der Datei message.pot der aktuelle Stand der zu lokalisierenden Zeichenketten enthalten ist. Der Aufruf für `msgmerge` sieht folgendermaßen aus:

```
msgmerge -U de.po message.pot
```

Dadurch wird das Sprachlexikon de.po mit dem Quelllexikon message.pot abgeglichen und aktualisiert. Alle geänderten Einträge werden mit dem Attribut "fuzzy" versehen. Daran kann der Übersetzer erkennen, dass diese Zeichenkette zu überprüfen ist. Ein solcher geänderter Eintrag sieht wie folgt aus:

```
# Quelldatei:Zeilennummer
#, fuzzy
Msgid "Originaltext"
Msgstr "Übersetzter Text"
```

Bis hierher wurde der Weg gezeigt, wie aus Programmcode zu übersetzende Zeichenketten extrahiert, strukturiert und aktualisiert werden können, um einer Lokalisierung zugeführt zu werden. Die Prämisse dieser Form der Lokalisierung ist es, jede einzelne Zeichenkette und sei sie noch so ähnlich einzeln zu übersetzen. Dies erscheint auf den ersten Blick redundant und unnötig, wer jedoch einmal versucht parametrisierte Aussagen wie "Es wurden n Dateien von m Dateien gelöscht." in nur eine andere Sprache zu übersetzen, wird schnell erkennen, dass das zu einem "Schreckensszenario" werden kann. Dieses wird vor allen Dingen dadurch verursacht, dass es zwischen den einzelnen Sprachen auf einzelnen Ebenen grammatikalisch häufig keine Übereinstimmungen gibt. So kennen zum Beispiel einige Sprachen neben dem Singular und Plural auch den Dual. Bei der Formulierung der Ausgabe macht es dort im Gegensatz zum Deutschen zum Beispiel einen Unterschied ob eine, zwei oder mehrere Personen angesprochen werden. Zur Veranschaulichung der Komplexität dieses Themas gibt es unter TPJ13 Artikel einen sehr schönen Artikel aus dem Perl Journal mit dem vielsagenden Absatz: "A Localisation Horror Story: It Could Happen To You".

I18N und L10N mit Catalyst

Im folgenden Abschnitt werden die notwendigen Schritte gezeigt, welche für die Anpassung von den oben vorgestellten GNU-Tools benötigt werden, um diese für eine Lokalisierung unter Catalyst zu benutzen. Dabei wird zunächst gezeigt, wie die entsprechenden Lexikon-Dateien auf Grundlage von Templatedateien (`Template::Toolkit`) erzeugt werden und wie diese später bei der Ausgabe eingesetzt werden.

Für die Benutzung des Werkzeuges `xgettext` mit Catalyst, genauer dem `Template::Toolkit`, gibt es die beiden Module:

- `Locale::Maketext::Extract::Run`
- `Locale::Maketext::Extract::Plugin::TT2`

Das folgende Skript untersucht nun alle Templatefiles tt, tt2, html nach den Zeichenketten:

- `[% l(text, args) %]`
- `[% loc(text, args) %]`

Bei der Funktion `l` bzw. `loc` handelt es sich um ein zu erstellendes Macro, welches weiter unten erklärt wird.

```
#!/usr/bin/perl

use warnings;
use strict;
use Locale::Maketext::Extract::Run
                                qw(xgettext);
use Locale::Maketext::Extract::Plugin::TT2;

xgettext(@ARGV);

exit(0);
```

Das Skript wird mit den Parametern für `xgettext` aufgerufen:

```
./gettext.pl --directory=root/
     --output=message.pot
```

Die einzelnen Sprachlexika können nun, wie bereits oben beschrieben wurde, erzeugt und aktualisiert werden. Des Weiteren müssen alle po-Dateien in dem zu erstellenden Ordner `MyApp/lib/MyApp/I18N` liegen.

In Catalyst selber müssen die beiden Plugins

- I18N (`Catalyst::Plugin::I18N`)
- Unicode (`Catalyst::Plugin::Unicode`)

in `MyApp/lib/MyApp.pm` eingebunden werden, welche über CPAN bezogen werden können.

```
use Catalyst qw /
  -Debug
  -Stats=1
  Static::Simple
  ...
  I18N
  Unicode
/;
```

Für alle Templates sollte an zentraler Stelle ein Macro erstellt werden, welches für die Lokalisierung zuständig ist und die Funktion `l(text, args)` bzw. `loc(text, args)` definiert. Ein guter Ort hierfür ist ein Template, welches bei jedem Seitenabruf aufgerufen wird, wie z.B. ein HTML-Kopf-Template in welchem der DOCTYPE und der HTML-Header untergebracht sind. Dieses Macro könnte wie folgt aussehen:

```
[% MACRO l(text, args) BLOCK;
    c.localize(text, args);
END; %]
```

Die `localize`-Funktion wird dem Catalyst-Kontext `c` über das Plugin l18N zugefügt.

Alle zu lokalisierenden Zeichenketten innerhalb der Templates müssen nun dieses Macro aufrufen:

```
<h1>[% l('Hello User!') %]</h1>
```

Diese Zeile erzeugt nach dem Aufruf des Perl-Skripts `gettext.pl` dann die folgenden Zeilen in dem Quelllexikon und dem Sprachlexikon:

```
message.pot
#: root/index.tt:10
msgid "Hello User!"
msgstr ""

de.po
#: root/index.tt:10
msgid "Hello User!"
msgstr "Hallo Nutzer!"
```

Da Nachrichten von Programmen in der Regel nicht ausschließlich aus statischen Zeichenketten wie im obigen Beispiel beschrieben bestehen, gibt es die Möglichkeit eine Liste von Argumenten an die Methode `c.localize` zu übergeben.

```
[% username = 'Michael' %]
<h1>[% l('Hello [_1]!', username) %]</h1>
```

Diese Zeile erzeugt nach dem Aufruf des Perl-Skripts `gettext.pl` dann die folgenden Zeilen in dem Quelllexikon und der Sprachdatei:

```
message.pot
#: root/index.tt:10
msgid "Hello %1!"
msgstr ""

de.po
#: root/index.tt:10
msgid "Hello %1!"
msgstr "Hallo %1!"
```

Die Funktion `c.localize` wird später beim Rendern der Ausgabe aus dem Platzhalter `[_1]` den im Catalyst-Kontext bekannten Wert der Variable "username" machen.

Mehr zum PO Datei Format und dessen detaillierter Nutzung (Singular, Plural etc.) findet ihr hier http://www.gnu.org/software/gettext/manual/gettext.html#PO-Files.

Jetzt braucht Catalyst nur noch zu wissen, welche Sprache ausgegeben werden soll. Dies wird mit der folgenden Methode, am besten in *auto* oder *begin* im Root-Controller (Root.pm) aufgerufen, festgelegt:

```
sub auto : Private
{
    my ($self, $c) = @_;

    ..

    my $locale = 'de';
    $c->languages($locale);

    ..
}
```

Anstelle der statischen Festlegung auf nur eine Sprache mit `$locale = 'de'`, empfiehlt es sich an dieser Stelle entweder die vom Browser gelieferte Spracheinstellung des Users zu benutzen (bei unpersonalisierten Webseiten, siehe `$c->request->header('accept-language'))` oder auf die in einer Datenbank abgespeicherten Nutzereinstellungen zurückzugreifen (z.B. bei Webseiten bei denen man sich einloggen muss).

Kann die localize-Funktion kein Substitut finden, gibt sie die übergebene Quellzeichenkette aus.

Theo Ohnsorge

FrOSCon

Free and Open Source Software Conference - 2009

22.-23. August 2009
Bonn - St. Augustin

- Über 20 Projekte und Aussteller
- Über 60 Vorträge in 5 Hörsälen
- Java-Subkonferenz
- LPI Prüfung
- Hüpfburg

Call for Papers bis 23.5.09
Call for Projects bis 23.6.09

kontakt@froscon.de - www.froscon.de

Hochschule Bonn-Rhein-Sieg, Grantham-Allee 20, 53757 Sankt Augustin

Merkwürdigkeiten in Perl - Indirekte Notation

An einigen Stellen findet man auch die indirekte Notation, also `my $obj = new Klasse`. Allerdings kann das zu Fehlern führen. Besser ist es, `my $obj = Klasse->new` zu verwenden. Wer auf der ganz sicheren Seite sein will, schreibt `my $obj = 'Klasse'->new` oder noch besser `my $obj = Klasse::->new`. Aber was kann jetzt bei der indirekten Notation alles schief gehen?

Vieles. In den folgenden Abschnitten werden ein paar Beispiele gezeigt, die Ergebnisse bringen, wie sie vermutlich nur von wenigen erwartet werden.

Allgemein kann man erstmal festhalten, dass Perl zwei gültige Art und Weisen kennt, den Ausdruck `my $obj = new Klasse` zu parsen.

- `my $obj = 'Klasse'->new;`
- `my $obj = new( Klasse() )`

Auch wenn die folgenden Beispiele scheinbar "willkürliche" Ergebnisse liefern, kann man vorhersagen, wann was gemacht wird.

Die zwei Arten zu parsen bedeuten ganz unterschiedliche Dinge: Die erste Version ruft die Methode `new` im Package `Klasse` auf, die zweite ruft erst die Funktion `Klasse` im aktuellen Package auf und dann die Funktion `new` im aktuellen Package.

Um jetzt entscheiden zu können was passiert, müssen wir die folgenden drei Faktoren betrachten:

- Methode `new` im Package `Klasse` existiert
- Funktion `new` im aktuellen Package existiert
- Funktion `Klasse` existiert im aktuellen Package

Beispiel 1

Dieses Beispiel verhält sich so wie man es erwartet, die Methode `new` von `T1` wird aufgerufen:

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

use strict;

{
    package T1;
    sub new { print "T1::new\n" }
}

sub new { print "main::new\n"; }

new T1;
```

Als Ausgabe bekommt man also:

```
C:\>perl listing1.pl
T1::new
```

Also alles klar?! Die folgenden Beispiele werden aber verdeutlichen, dass es nicht unbedingt so klar ist.

Beispiel 2

Jetzt verhält sich das Skript schon nicht mehr so wie man es sich vorstellt, aber das Programm läuft durch:

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

use strict;

{
    package T1;
    sub new { print "T1::new\n" }
}

sub T1  { print "main::T1\n";  }

sub new { print "main::new\n"; }

new T1;
```

In diesem Fall wird erst die Funktion `T1` aufgerufen und dann die Funktion `new` aus dem `main`-Package.

Daher bekommt man folgende Ausgabe:

```
C:\>perl listing2.pl
main::T1
main::new
```

Beispiel 3

Dieses Beispiel wird erst gar nicht kompiliert, weil Perl einen Fehler vermutet.

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

use strict;

{
    package T1;
    sub new { print "T1::new\n" }
}

sub T1 { print "main::T1\n"; }

new T1;
```

Perl kann keine Subroutine `new` in `main` finden, versucht dann aber nicht `T1::new` zu finden, weil es das 'T1' als Subroutinenaufruf von `main::T1` erkennt. Perl behandelt das 'new' also als Bareword:

```
C:\>perl listing3.pl
Bareword found where operator expected at
  listing3.pl line 12, near "new T1"
(Do you need to predeclare new?)
syntax error at listing3.pl line 12,
  near "new T1"
Execution of listing3.pl aborted due to
  compilation errors.
```

Beispiel 4

Hier verhält sich Perl wieder so wie es jeder erwarten würde: Es wird `T1::new` aufgerufen. Also funktioniert alles...

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

use strict;

{
    package T1;
    sub new { print "T1::new\n" }
}

new T1;
```

Diese Beispiel zeigen, dass die indirekte Notation eher nachteilig ist. Vor allem wenn man "fremden" Code verwendet und man nicht mit Sicherheit sagen kann, dass sich an den Methoden etwas ändert. Oder man denkt nicht mehr an die Seiteneffekte, baut eine neue Funktion ein und schon funktioniert das ganze Programm nicht mehr.

Insgesamt kann man anhand der folgenden Übersicht in Tabelle 1 bestimmen, welches Verhalten zu erwarten ist.

Auf CPAN gibt es mittlerweile ein Modul, mit dem man sich bei indirekten Methodenaufrufen warnen lassen kann: `indirect`

```
no indirect;
my $x = new T1 1, 2, 3; # Warnung
{
 use indirect;
 my $y = new T2; # ok
}
no indirect ':fatal';
# stirbt -> beachte Typo
if (defied $foo) { ... }
```

Renée Bäcker

	main::new + main::T1	nur main::new	Nur main::T1	weder noch
T1::new existiert	new(T1())	'T1'->new	ERROR	'T1'->new
T1:: new existiert nicht	new(T1())	'T1'->new	ERROR	'T1'->new

Tabelle 1: Übersicht Verhalten

MODULE

Linux GUI's automatisch testen

Das automatisierte Testen im Allgemeinen ist eine sehr aufwändige Angelegenheit, da meist mehr Zeit zur Korrektur des Testskriptes als mit dem eigentlichen Testen verbracht wird. Aus diesem Grund sind Schnittstellen zum Überprüfen des aktuellen Teststadiums elementar wichtig.

Dieser Artikel beschreibt wie man Linux GUI's automatisiert testet und dabei den Datenstrom zwischen X-Server und X-Client zum Synchronisieren nutzt. Die eingesetzten Perl Module sind `X11::GUITest` und `X11::GUITest::record`.

Das X Window System

Das X Windows System ist in erster Linie ein Netzwerkprotokoll, das es ermöglicht, mittels einer Client/Server-Architektur grafische Oberflächen darzustellen. Es wird vor allem in den Unix/Linux Betriebssystemen eingesetzt.

Ein einfacher Testfall

Als Testfall dient eine einfache Applikation: xmessage (siehe Bild 1).

Auch wenn dieses Beispiel sehr simpel erscheint, wird man feststellen, dass die zu automatisierenden Aktionen gar nicht so einfach umsetzbar sind. Dieser Artikel beschränkt sich auf das automatisierte Testen folgender Aktionen:

Bild1: xmessage

- Öffnet sich das Fenster?
- Wird der Text im Fenster richtig angezeigt?
- Schließt der "okay"-Button auch das Fenster?
- Ist das Fenster verschiebbar?

Mausbewegungen und Klicks

Durch das Perlmodul `X11::GUITest` ist es möglich, Mausbewegungen und Tastatureingaben vollständig zu automatisieren. Weiter bietet das Modul viele hilfreiche Funktionen, um zum Beispiel Fenster zu finden, Fenster zu schließen oder zu bewegen.

Die erste Testautomation würde aussehen, wie in Listing 1 dargestellt.

Nach dem Erstellen des Fensters muss die Window-ID gesucht werden. Hier bietet `X11::GUITest` die Funktion `FindWindowLike`. Diese durchsucht den X-Fensterbaum nach einem Fenstertitel und gibt alle passenden Fenster als Array zurück. Wird die Window-ID gefunden, fragt das Skript mit `GetWindowPos` die Position des Fensters ab und bewegt das Fenster mittels `MoveWindow`. Mausbewegungen, die ein Fenster betreffen, werden am besten relativ von der Position des Fensters angegeben. Durch `PressMouseButton` sowie `ReleaseMouseButton` wird der eigentliche Klick auf "okay" durchführt.

Die X-Server Record Extension zur Synchronisierung nutzen

Der erste Schritt zum automatisierten Testen ist somit getan. Jedoch sind noch ein paar Schwierigkeiten zu überbrü-

cken. Problematisch ist das Verifizieren der ausgeführten Aktion: Wird beim Drücken auf den "okay"-Button wirklich das Fenster geschlossen?

Lösung bietet das Modul `X11::GUITest::record`. Es implementiert die so genannte *X11 Record Extension*, die den kompletten Datenverkehr des X-Server und den X-Clients verfügbar macht. So können Mausbewebungen, Tastatureingaben, aber auch erstellte Fenster und Fensterinhalte ausgewertet werden (siehe Listing 2).

Um genau einzuschränken, welche Daten man vom X-System benötigt, legt man mit `SetRecordContext` alle Requests oder Events fest, die aufgezeichnet werden sollen. Ein Request ist hierbei der Datenstrom vom X-Client zum X-Server und ein Event umgekehrt vom X-Server zum X-Client.

Für das Beispiel reichen entstehende Fenster (X_CreateWindow), schließende Fenster (X_DestroyWindow) und Text (X_PolyText8) aus. Durch GetRecordInfo erhält man alle Daten in einer Hash-Referenz nacheinander, dem zeitlichen Ablauf

```
# guitest_1.pl
use X11::GUITest qw /:ALL/;
system ("xmessage Das ist ein Test &");
sleep 2;

my ($WinID) = FindWindowLike ('xmessage');
my ($x, $y) = GetWindowPos ($WinID);

MoveWindow($WinID, ($x + 50), ($y + 50));
sleep 1;
my ($new_x, $new_y) = GetWindowPos ($WinID);

print "Window is movable\n" if (($new_x > $x) &&
                                ($new_y > $y));
MoveMouseAbs ($new_x + 35 , $new_y + 45);
PressMouseButton (M_LEFT);
ReleaseMouseButton (M_LEFT);
sleep 1
```

Listing 1

```
# guitest_2.pl
use X11::GUITest::record qw /:ALL :CONST/;

SetRecordContext (X_CreateWindow, X_DestroyWindow,  X_PolyText8);
EnableRecordContext();
system ("xmessage -timeout 1 Das ist ein Test" );
sleep 1;
DisableRecordContext();

while (my $data = GetRecordInfo){
  print $data ->{TxtType} ." ";
  print "WinID: ". $data -> {WinID} if (exists ($data -> {WinID}));
  print "Text: ". $data -> {Text}   if ($data -> {TxtType} eq "X_PolyText8");
  print "\n";
}
__END__
#> perl guitest_2.pl
X_CreateWindow WinID: 6302137
X_CreateWindow WinID: 6302138
X_CreateWindow WinID: 6302139
X_CreateWindow WinID: 6302167
X_PolyText8 Text: Das ist ein Test
X_PolyText8 Text: okay
X_DestroyWindow WinID: 6302137
X_DestroyWindow WinID: 6302138
X_DestroyWindow WinID: 6302139
X_DestroyWindow WinID: 6302167
```

Listing 2

getreu. Als Ausgabe werden alle Window-ID's von geöffneten und geschlossenen Fenstern ausgegeben und der Text des Fensterinhaltes und des Buttons. Es verwundert zunächst, warum es zu mehreren CreateWindow-Requests kommt. Dies erklärt sich, wenn man weiß, dass der jeweilige Window Manager zu einem Fenster Rahmen, Titel und weitere Funktionalitäten (wie minimieren, maximieren und schließen) ergänzt. Dies alles sind für das X-System eigenständige Fenster, auch wenn der Benutzer dies nicht bemerkt.

Die Mischung macht's

Mischt man nun die beiden Module (siehe Listing 3), so können alle zuvor festgelegten Fragestellungen beantwortet werden.

Angemerkt sollte noch werden, dass Fensterinhalte bei komplexeren GUI-Toolkits wie GTK+ oder QT nicht über einen X_ PolyText8-Request übertragen werden. Hier bekommt man bei dieser Schnittstelle nur X_PolyLine-Requests mit. Diese kann dann nur anhand der Anzahl ausgewertet werden.

Marc Koderer

```
# guitest_3.pl
use X11::GUITest::record qw /:ALL :CONST/;
use X11::GUITest qw /:ALL/;

SetRecordContext (X_CreateWindow, X_PolyText8);
EnableRecordContext ();
system ("xmessage Das ist ein Test &");
sleep 2;
DisableRecordContext();

# "Öffnet sich das Fenster" und "Wird der Text angezeigt"
while (my $data = GetRecordInfo){
  print "Text ok\n" if ($data -> {Text} eq "Das ist ein Test");
  print "Fenster geoeffnet\n" if ($data -> {TxtType} eq "X_CreateWindow");
}

SetRecordContext (X_DestroyWindow);
EnableRecordContext ();

my ($WinID) = FindWindowLike ('xmes');
my ($x, $y) = GetWindowPos ($WinID);

MoveMouseAbs ($x + 35, $y + 45);
PressMouseButton (M_LEFT);
ReleaseMouseButton (M_LEFT);
sleep 2;
DisableRecordContext ();

# Werden Fenster durch "okay"-Button geschlossen?
while (my $info = GetRecordInfo){
  print "Fenster geschlossen\n";
}

__END__
perl guitest_3.pl
Fenster geoeffnet
Fenster geoeffnet
Fenster geoeffnet
Text ok
Fenster geschlossen
Fenster geschlossen
Fenster geschlossen
```

Listing 3

111% DBIx::Class

In dieser Ausgabe gibt es den zweiten Teil der Mini-Serie "100% DBIx::Class", in der es um ein paar Tricks geht, mit denen man die Möglichkeiten von `DBIx::Class` besser nutzen kann.

In dieser Folge wird es darum gehen wie man den Suchpfad für bestimmte Klassen ändern kann. Weiterhin geht es um so genannte "Virtuelle Views". Dabei kann man das Ergebnis einer Abfrage als "Tabelle" nehmen. Im letzten Abschnitt geht es um Profiling und wie man damit Statistiken über die Laufzeit von SQL-Befehlen erstellen kann.

load_namespaces

`load_namespaces` ist eine Alternative zu `load_classes` und ist dann nützlich wenn man eigene Result- oder ResultSet-Klassen hat, die nicht im Suchpfad liegen. Im Suchpfad liegen alle Klassen aus den `DBIx::Class::Result::*`- bzw. `DBIx::Class::ResultSet::*`-Namensräumen. Manchmal ist es aber erforderlich, dass die selbst geschriebenen Klassen außerhalb des Suchpfades liegen. Dann kommt `load_namespaces` ins Spiel.

Wie eigene ResultSet-Klassen geschrieben werden, wurde im letzten Teil von "100% DBIx::Class" gezeigt.

Hier drei Beispiele für die Verwendung von `load_namespaces`. Im ersten Beispiel liegen die Klassen alle unterhalb von `My::Schema`. Da die Klassen in `::Result::*` bzw. `::ResultSet::*` liegen - was die Standardnamen in `DBIx::Class` sind - braucht man `load_namespaces` keine Parameter zu übergeben.

```
# lädt My::Schema::Result::User,
# My::Schema::ResultSet::User etc.
My::Schema->load_namespaces;
```

Listing 1

Im zweiten Beispiel wird gezeigt, wie man von den typischen `DBIx::Class`-Namen abweichen kann. Die Result-Klassen liegen nicht in `::Result::*` und die ResultSet-Klassen nicht in `::ResultSet::*`, sondern in `::MyResults::*` bzw. in `::MyResultSets::*`. Aber auch diese Klassen liegen unterhalb von `My::Schema`.

```
# lädt My::Schema::MyResults::User,
# MySchema::MyResultSets::User etc.
My::Schema->load_namespaces(
  result_namespace => 'MyResults',
  resultset_namespace => 'MyResultSets',
);
```

Listing 2

Das dritte Beispiel zeigt, wie man vorgeht, wenn die Klassen auch nicht mehr unter `My::Schema` liegen. Soll `DBIx::Class` mitgeteilt werden, dass der Namensraum ein voll qualifizierter Name ist, muss ein `+` vorangestellt werden.

```
My::Schema->load_namespaces(
  result_namespace => '+Namespace::Results',
  resultset_namespace => '+Namespace::Sets',
);
```

Listing 3

Virtuelle Views

Eine View ist eine logische Relation in einem Datenbanksystem. Diese logische Relation wird über eine gespeicherte Abfrage definiert. In der Anwendung kann die View wie eine "normale" Tabelle behandelt werden und immer wenn diese View verwendet wird, wird die dahinterliegende Abfrage ausgeführt. Die gespeicherte Abfrage ist also schon eine Vorselektion der Daten.

Dies wird häufig verwendet, wenn man Nutzern nur bestimmte Daten zur Verfügung stellen will, weil ihnen z.B. Administrationsrechte fehlen. Eine andere Anwendung ist das Zusammenstellen von Informationen aus verschiedenen Tabellen.

Die folgenden Code-Fragmente zeigen, wie man mit `DBIx::Class` so eine View erstellen kann. Als erstes muss eine Klasse erstellt werden, die die View darstellt (Lising 4). Diese Klasse muss von `DBIx::Class::Core` erben. Der Tabellenname, der der Methode `table` übergeben wird, darf nicht schon anderweitig verwendet werden. Wie bei anderen Klassen, die "normale" Tabellen repräsentieren, werden die Spalten definiert, die in der View existieren (mit `add_columns`).

Dem ResultSource-Objekt, das man mittels `result_source_instance` erhält, teilt man dann mit `name` mit, welche Abfrage hinter der View liegt. Dabei muss es sich um eine Referenz auf einen String handeln.

```
package DB::UserView;
use base qw/DBIx::Class::Core/;

my $stmt = qq~( SELECT username FROM
                usertabelle
                WHERE print_abo = 1 )~;

__PACKAGE__->table('my_user_view');
__PACKAGE__->add_columns(qw/username/);
__PACKAGE__->result_source_instance
           ->name(\$stmt);
```

Listing 4

In diesem Beispiel werden also alle User vorselektiert, die ein Print-Abonnement haben. In der Anwendung kann man sich in Zukunft die Bedingung sparen und stattdessen direkt auf die View zugreifen.

In der Schema-Klasse, die man auch vom Standard-Einsatz von `DBIx::Class` kennt, muss man die View-Klasse noch bekannt machen (Listing 5).

```
package MySchema;
use base qw/DBIx::Class::Schema/;

use DB::UserView;

__PACKAGE__->register_class(
  PrintAbo => 'DB::UserView'
);
```

Listing 5

Danach kann man in der Anwendung auf die View zugreifen wie auf jede andere Tabelle auch (siehe Listing 6).

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

my $schema = MySchema->connect( ... );

my ($r) = $schema-> resultset('PrintAbo')
                 ->all;

print $r->username, "\n";
```

Listing 6

Die runden Klammern beim SQL-Befehl sind wichtig, da das ein Subselect ist. Betrachtet man das SQL-Statement, was bei der Verwendung abgesetzt wird, erkennt man warum das so ist:

```
SELECT me.username, me.print_abo FROM (
     SELECT username, print_abo
          FROM usertabelle )
     WHERE print_abo = 1 me
```

Listing 7

Ohne die Klammern würde es einen Syntax-Fehler geben.

Die Verwendung unterscheidet sich allerdings etwas, wenn in der View `?` verwendet werden. Wenn der Befehl also z.B. so aussieht:

```
my $stmt = qq~SELECT username
   FROM usertabelle
      WHERE age > ?~;
```

Listing 8

Dann muss bei der Abfrage in der Anwendung mit einem `bind`-Parameter gearbeitet werden:

```
my ($r) = $schema->resultset('PrintAbo')
     ->search( {}, {bind => [ 18 ]});
```

Listing 9

Profiling

Profiling wird immer dann betrieben, wenn die Anwendung zu langsam ist. Dabei geht es um die Suche der langsamsten Code-Stellen. Wenn man diese identifiziert hat, kann man sich hinsetzen und überlegen wie man etwas anders machen kann.

In diesem Abschnitt wird gezeigt, wie man SQL-Queries mit `DBIx::Class` profilen kann.

In `DBIx::Class` gibt es die Möglichkeit, SQL-Queries zu debuggen. Dazu muss das Debugging in `DBIx::Class::Storage` angeschaltet werden:

```
$dbic->storage->debug( 1 );
```

Das muss auch für das Profiling angeschaltet werden. Zusätzlich kann man eine Subklasse von `DBIx::Class::Storage::Statistics` erstellen, die eigene Profiling Mechanismen implementieren. Eine Beispielklasse ist in Listing 10 gezeigt.

```
package DB::Profiler;

use strict;
use warnings;
use DBIx::Class::Storage::Statistics;

our @ISA = qw(
         DBIx::Class::Storage::Statistics);
use Time::HiRes qw(time);

my $start;

sub query_start {
    $start = time;
}

sub query_end {
    my $diff = time - $start;
    my ($self,$sql,@params) = @_;

    print "Statement: $sql\n",
          "Parameters: @params\n",
          "Execution time: $diff\n\n";
    $start = undef;
}
```

Listing 10

In dem Beispiel werden die beiden Methoden `query_start` und `query_end` überschrieben. Wie der Name schon andeutet, werden die Methoden vor- bzw. nach dem Ausführen der Abfrage aufgerufen. Soll das Profiling auf Transaktionsebene stattfinden, so kann man die Methoden `txn_begin`, `txn_rollback` und `txn_commit` überschreiben.

Im Anwendungscode muss das Profiling noch aktiviert werden. Wie oben beschrieben muss zum einen der Debug-Modus eingeschaltet werden und zum anderen muss das Debugging-Objekt an das Storage-Objekt übergeben werden:

```
__PACKAGE__->storage
           ->debugobj(
               DB::Profiler->new
);
__PACKAGE__->storage->debug(1);
```

Listing 11

Sollen mit dem Profiling noch mehr Statistiken erstellt werden, so kann man das auf diese Weise machen:

```
sub query_end {
  my ($self,$sql,@params) = @_;

  my $elapsed = time - $start;
  push(@{ $calls{$sql} }, {
      params => \@params,
      elapsed => $elapsed
  });
}
```

Listing 12

Damit kann man dann für jeden SQL-Befehl statistische Daten wie Maximum-, Minimum- und Mittelwert berechnen und so feststellen, ob ein Befehl bei bestimmten Parametern besonders langsam ist. Eine weitere Empfehlung ist `DBIx::Class::QueryLog`.

Renée Bäcker

Test::Class Best Practices

Wenn man mit großen Test-Suites arbeitet, wird die Verwendung von Prozeduralen Tests für Objektorientierten Code mit der Zeit schwerfällig. Das ist der Punkt in dem `Test::Class` glänzt. Leider kämpfen viele Programmierer damit, dieses Modul kennenzulernen oder nutzen nicht die kompletten Möglichkeiten.

Beachte bitte, dass dieser Artikel eine grundlegende Kenntnis von Objektorientiertem Perl und Testen mit Perl voraussetzt. Auch sind einige Klassen nach den Standards vieler OO-Programmierer (den Autor eingeschlossen) nicht einwandfrei, aber sie wurden eher für Klarheit als für "Reinheit" geschrieben.

Module und Versionen

Dieser Artikel basiert auf folgenden Modulen und Versionen:

- `Test::Class` Version 0.31
- `Test::Most` Version 0.21
- `Test::Harness` Version 3.15
- `Moose` Version 0.7
- `Class::Data::Inheritable` Version 0.08

Du kannst auch niedrigere Versionen der Module benutzen (und mit OO selbst statt mit Moose schreiben). Aber beachte, dass Du dann unter Umständen ein etwas anderes Verhalten beobachten kannst.

Hinweise zum Code

Beachte, dass `Moose`-Packages *generell* mit

```
__PACKAGE__->meta->make_immutable;
no Moose;
```

enden sollen.

Das lassen wir bei unseren Beispielen weg. Wir lassen auch `use strict` und `use warnings` weg, aber wir gehen davon aus, dass das im Code steht (`strict` und `warnings` werden automatisch eingeschaltet, wenn man `use Moose` verwendet). Der Code wird auch so laufen. Wir machen das lediglich, um den Fokus auf die Core-Features des Codes zu legen.

Natürlich musst Du unter Umständen die Shebang Zeile (`#!/usr/bin/env perl -T`) an Dein System anpassen.

Entwicklung eines Perl-Programmierers

Es gibt viele Wege, die ein Programmierer in seiner Entwicklung gehen kann. Aber typischerweise scheint es folgender Weg zu sein:

1. Starten mit dem Schreiben einfacher prozeduraler Programme.
2. Schreiben von Modulen, wenn die Wiederverwendung von Code sinnvoll erscheint.
3. Starten mit der Verwendung von Objekten, wenn sie eine bessere Abstraktion benötigen.
4. Beginnen mit dem Schreiben von Tests.

Schön wäre es, wenn Leute mit dem Schreiben von Tests schon an Tag 1 beginnen würden - die Realität ist leider eine andere. Aber wie sehen die Tests aus wenn sie damit beginnen? Nunja, sie sind richtig prozedurale Tests wie dieser:

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

use strict;
use warnings;

use Test::More tests => 3;

use_ok 'List::Util', 'sum' or die;

ok defined &sum,
 'sum() should be exported to our namespace';
is sum(1,2,3), 6,
  '... and it should sum lists correctly';
```

Es gibt nichts an prozeduralen Tests auszusetzen. Sie sind z.B. sehr gut für Nicht-OO Code. Bei den meisten Projekten machen sie alles was man braucht und wenn man die meisten Module auf CPAN herunterlädt, findet man ihre Tests – wenn sie denn welche haben – im prozeduralen Stil geschrieben. Wie auch immer; wenn man beginnt mit einer größeren Code-Basis zu arbeiten, wird ein einfaches `t/`-Verzeichnis mit 317 Testskripten lästig. Wo ist der Test den Du gerade brauchst? Es wird schwierig, sich alle Testnamen merken zu wollen oder sich durch die Tests zu wühlen, um den einen Test zu finden, der den Code testet an dem Du gerade arbeitest. An dieser Stelle kann Adrian Howards `Test::Class` helfen.

Verwendung von Test::Class

Einfache Testklasse erzeugen

Lass uns jetzt mit `Test::Class` beginnen. Ich bin ein großer Freund davon, direkt in das Thema einzutauchen. Deshalb werden wir zuerst viel Theorie weglassen und schauen, wie die Dinge funktionieren. Auch wenn ich häufig Test-Driven-Development (TDD) betreibe, werde ich den Prozess hier umkehren, sodass Du wirklich siehst was getestet wird. `Test::Class` hat einige verschiedene Features, wobei ich einige hier nicht erklären werde. Für mehr Informationen schau' Dir bitte die Dokumentation an.

Als erstes werden wir eine einfache `Person`-Klasse erzeugen. Weil ich es nicht mag, einfache Methoden immer wieder auszuschreiben, werden wir `Moose` [1] verwenden. Es erledigt einiges an Routinearbeit für uns.

```
package Person;

use Moose;

has first_name => ( is => 'rw',
                    isa => 'Str' );
has last_name  => ( is => 'rw',
                    isa => 'Str' );

sub full_name {
    my $self = shift;
    return $self->first_name . ' ' .
           $self->last_name;
}

1;
```

Das gibt uns den Konstruktor sowie die `first_name`-, `last_name`- und `full_name`-Methode.

Lass uns jetzt ein einfaches `Test::Class` Programm dafür schreiben. Um das zu tun, brauchen wir einen Ort, an dem wir die Tests speichern können. Weiterhin müssen wir den Package-Namen sorgfältig wählen, um Namespace-Kollisionen zu vermeiden. Ich bevorzuge es, meinen Test-Klassen ein `Test::` voranzustellen. So stelle ich sicher, dass es keine Mehrdeutigkeiten gibt. In diesem Fall werde ich die `Test::Class` Tests in `t/tests/` speichern und unsere erste Klasse wird `Test::Person` heißen. Wir gehen von folgender Verzeichnis-Struktur aus:

```
lib/
lib/Person.pm
t/
t/tests/
t/tests/Test
t/tests/Test/Person.pm
```

Und die Testklasse wird so aussehen:

```
package Test::Person;

use Test::Most;
use base 'Test::Class';

sub class { 'Person' }

sub startup : Tests(startup => 1) {
    my $test = shift;
    use_ok $test->class;
}
```

Fortsetzung siehe Seite 28

```
sub constructor : Tests(3) {
    my $test  = shift;
    my $class = $test->class;
    can_ok $class, 'new';
    ok my $person = $class->new,
       '... and the constructor should
            succeed';
    isa_ok $person, $class,
       '... and the object it returns';
}

1;
```

Fortsetzung von Seite 27

Beachte: Wir verwenden `Test::Most` anstelle von `Test::More`. Die Vorteile der `Test::Most`-Features werden wir später verwenden. Die Methoden der Klasse sollten wirklich 'ro' (read-only) sein, da wir das Objekt in einem inkonsistenten Zustand lassen können. Das ist Teil dessen, was ich mit "richtigem" OO-Code meinte. Aber noch einmal: das ist nur zu Darstellungszwecken so geschrieben.

Bevor wir uns anschauen was das alles bedeutet, lass uns weitermachen und den Code laufen lassen. Um das zu tun, speichere das folgende Programm als run.t in unserem t/-Verzeichnis.

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

use lib 't/tests';
use Test::Person;

Test::Class->runtests;
```

Das kleine Programm setzt den Pfad zu unseren Testklassen, lädt sie und startet die Tests. Jetzt kannst Du das Programm mit dem prove-Tool laufen lassen:

```
prove -lv --merge t/run.t
```

Tipp: Das `--merge` teilt `prove` mit, `STDOUT` und `STDERR` zu vereinen. Das verhindert Synchronisationsprobleme, die passieren können, falls `STDERR` nicht immer mit `STDOUT` synchron ist. Es ist empfohlen, dass nicht zu verwenden, solange die Tests nicht im *verbose* Modus laufen. Das hängt damit zusammen, dass Fehler-Beschreibungen dann an `STDOUT` gesendet werden und `TAP::Harness` Zeilen von `STDOUT` wegwirft, die mit '#' beginnen - falls die Tests nicht in *verbose* Modus laufen.

Und wir bekommen eine Ausgabe, die folgender ähnlich ist:

```
t/run.t ..
1..4
ok 1 - use Person;
#
# Test::Person->constructor
ok 2 - Person->can('new')
ok 3 - ... and the constructor should succeed
ok 4 - ... and the object it
           returns isa Person
ok
All tests successful.
Files=1, Tests=4,  0 wallclock secs
     ( 0.03 usr  0.00 sys +
       0.43 cusr  0.02 csys =  0.48 CPU)
Result: PASS
```

Du wirst bemerken, dass die Ausgabe des Tests ("Test Anything Protocol" oder "TAP" genannt wenn es Dich interessiert) für die constructor-Methode mit der folgenden Diagnose-Ausgabe beginnt.

```
01: package Test::Person;
02:
03: use Test::Most;
04: use base 'Test::Class';
05:
06: sub class { 'Person' }
07:
08: sub startup : Tests(startup => 1) {
09:     my $test = shift;
10:     use_ok $test->class;
11: }
12:
13: sub constructor : Tests(3) {
14:     my $test  = shift;
15:     my $class = $test->class;
16:     can_ok $class, 'new';
17:     ok my $person = $class->new,
18:     '... and the constructor should succeed';
19:     isa_ok $person, $class, '... and the object it returns';
20: }
21:
22: 1;
```

Listing 1

```
# Test::Person->constructor
```

Das taucht vor jeder Ausgabe einer Test-Methode auf und macht es sehr einfach einen fehlgeschlagenen Test zu finden.

Nun lass uns einen Blick auf unseren Test werfen und schauen was da passiert - siehe Listing 1.

Die Zeilen 1 bis 4 sind ziemlich eindeutig. Zeile 4 erbt von `Test::Class` und das ist alles Notwendige, um `Test::Class` für die Tests zu verwenden. Zeile 6 definiert eine `class`-Methode, die unsere Tests verwenden, damit sie wissen welche Klasse sie testen. Es ist sehr wichtig das so zu tun und nicht den Klassennamen direkt in die Testmethoden zu schreiben. Das ist gute OO-Schreibweise – im Allgemeinen – und später werden wir noch sehen wie uns das weiterhilft.

Die `startup`-Methode hat ein Attribut - 'Tests' - das die Argumente `startup` und `1` hat. Jede Methode, die als `startup`-Methode gekennzeichnet ist, wird einmal ausgeführt bevor irgendeine andere Methode ausgeführt wird. Die `1` (eins) im Attribut besagt "wir werden einen Test in dieser Methode durchführen". Wenn Du keinen Test in der `startup`-Methode machst, lass die Zahl weg:

```
sub load_db : Tests(startup) {
    my $test = shift;
    $test->_create_database;
}

sub _create_database {
    ...
}
```

Tipp: Wie Du am Code oben sehen kannst, musst Du die `startup`-Methode nicht `startup` nennen. Ich empfehle, den Namen des zu testenden Attributs zu geben (aus Gründen, die wir später besprechen werden).

Das wird nur ein einziges Mal für jede Testklasse ausgeführt. Weil die `_create_database`-Methode keine Attribute hat, kannst Du es sicher aufrufen und `Test::Class` wird nicht versuchen, die Methode als Test aufzurufen.

Natürlich gibt es ein dazugehöriges `shutdown`.

```
sub shutdown_db : Tests(shutdown) {
    my $test = shift;
    $test->_shutdown_database;
}
```

Das ermöglicht, eine unberührte Testumgebug für jede Testklasse auf- und abzubauen, ohne dass eine andere Testklasse sich mit den aktuellen Tests behindert. Natürlich bedeutet das, dass Test vielleicht nicht parallel ausgeführt werden können und es gibt Wege das zu umgehen. Aber das geht über diesen Artikel hinaus.

Wie schon erwähnt hat unsere `startup`-Methode einen zweiten Parameter, der `Test::Class` mitteilt, dass wir einen Test in dieser `startup`-Methode laufen lassen. Das ist absolut optional. Hier nutzen wir das, um sicher zu testen, ob wir die `Person`-Klasse laden können. Als zusätzliches Feature nimmt `Test::Class` bei einem fehlgeschlagen Test an, dass es keinen Grund gibt, die restlichen Tests laufen zu lassen. Also lässt es die übrigen Tests der Klasse aus.

Tipp: Lass keine Tests in Deiner `startup`-Methode laufen. Warum klären wir gleich. Im Moment ist es besser, es so zu machen:

```
sub startup : Tests(startup) {
    my $test  = shift;
    my $class = $test->class;
    eval "use $class";
    die $@ if $@;
}
```

Wie auch immer, wir werden den Test in der `startup`-Methode noch eine Weil behalten, nur damit Du sehen kannst wie alles funktioniert.

Jetzt lass uns einen genaueren Blick auf die `constructor`-Methode werfen (siehe Listing 2).

Tipp: Wir haben den Konstruktor der Tests nicht `new` genannt, weil das eine Methode von `Test::Class` ist und das Überschreiben würde unsere Tests zerstören.

Unser `Tests`-Attribut zählt die Anzahl der Tests auf '3', aber wenn wir nicht genau wissen wieviele Tests wir haben werden, können wir weiterhin `no_plan` verwenden.

```
sub constructor : Tests(no_plan) { ... }
```

```
13: sub constructor : Tests(3) {
14:     my $test  = shift;
15:     my $class = $test->class;
16:     can_ok $class, 'new';
17:     ok my $person = $class->new,
18:         '... and the constructor should succeed';
19:     isa_ok $person, $class, '... and the object it returns';
20: }
```

Listing 2

```
sub first_name : Tests {
    my $test   = shift;
    my $person = $test->class->new;
    can_ok $person, 'first_name';
    ok !defined $person->first_name,
      '... and first_name should start out undefined';
    $person->first_name('John');
    is $person->first_name, 'John',
      '... and setting its value should succeed';
}

sub last_name : Tests {
    my $test   = shift;
    my $person = $test->class->new;
    can_ok $person, 'last_name';
    ok !defined $person->last_name,
      '... and last_name should start out undefined';
    $person->last_name('Public');
    is $person->last_name, 'Public',
      '... and setting its value should succeed';
}

sub full_name : Tests {
    my $test   = shift;
    my $person = $test->class->new;
    can_ok $person, 'full_name';
    ok !defined $person->full_name,
      '... and full_name should start out undefined';
    $person->first_name('John');
    $person->last_name('Public');
    is $person->full_name, 'John Public',
      '... and setting its value should succeed';
}
```

Listing 3

Als Shortcut können wir die Parameter weglassen, was auch `no_plan` bedeutet:

```
sub constructor : Tests { ... }
```

Die `my $test = shift` Zeile ist äquivalent zu `my $self = shift`. Ich mag es, in meinen Testklassen `$self` in `$test` umzubenennen, aber das ist eine persönliche Vorliebe.

Das `$test` Objekt ist eine leere Hashreferenz. Das erlaubt, dort Daten zu verstecken. Zum Beispiel:

```
sub startup : Tests(startup) {
    my $test = shift;
    my $pid  = $test
         ->_start_process or die
           "Could not start process: $?";
    $test->{pid} = $pid;
}
```

```
sub run : Tests(no_plan) {
    my $test = shift;
    my $process = $test
         ->_get_process($test->{pid});
    ...
}
```

Der Rest der Testmethode ist selbsterklärend, wenn man sich mit `Test::More` auskennt.

Wir hatten auch noch `first_name`, `last_name` und `full_name`. Also lass uns dafür Tests schreiben. Weil wir im "development mode" sind, werden wir diese Tests als `no_plan` belassen. Aber vergiss nicht, die Anzahl der Tests zu setzen wenn Du fertig bist (Listing 3).

Tipp: Wenn möglich, solltest Du die Testmethoden nach den Methoden, die Du testest, benennen. Das macht das Auffinden der Tests viel einfacher. Du kannst sogar Tools für Editoren schreiben, um automatisch dorthin zu springen. Natürlich werden nicht alle Testmethoden in dieses Muster passen - aber viele werden es.

Bei den Tests für `first_name` und `last_name` können wahrscheinlich viele gemeinsame Elemene herausgelöst werden, aber im Augenblick ist es so in Ordnung. Lass uns anschauen was passiert wenn wir die Tests laufen lassen (Warnung sind ausgelassen) - Listing 4.

Ohje. Wir können sehen, dass `full_name` sich nicht so verhält, wie wir es erwartet haben. Nehmen wir an, wir wollen `croak` verwenden, wenn entweder der Vor- oder Nachname nicht gesetzt wurde. Um es einfach zu halten, nehmen wir an, dass weder `first_name` noch `last_name` auf einen "falschen" Wert gesetzt werden darf.

```
sub full_name {
    my $self = shift;

    unless ( $self
       ->first_name && $self->last_name ) {
        Carp::croak("Both first
              and last names must be set");
    }

    return $self
       ->first_name . ' ' . $self->last_name;
}
```

Das sollte soweit klar sein. Jetzt lass uns den neuen Test anschauen. Wir werden den `throws_ok`-Test aus `Test::Exception` nehmen, um das `Carp::croak()` zu testen. Weil wir `Test::Most` anstatt `Test::More` verwenden, können wir diese Testfunktion verwenden ohne explizit `Test::Exception` zu verwenden (siehe Listing 5).

```
t/run.t ..
ok 1 - use Person;
#
# Test::Person->constructor
ok 2 - Person->can('new')
ok 3 - ... and the constructor should succeed
ok 4 - ... and the object it returns isa Person
#
# Test::Person->first_name
ok 5 - Person->can('first_name')
ok 6 - ... and first_name should start out undefined
ok 7 - ... and setting its value should succeed
#
# Test::Person->full_name
ok 8 - Person->can('full_name')
not ok 9 - ... and full_name should start out undefined

#   Failed test '... and full_name should start out undefined'
#   at t/tests/Test/Person.pm line 48.
#   (in Test::Person->full_name)
ok 10 - ... and setting its value should succeed
#
# Test::Person->last_name
ok 11 - Person->can('last_name')
ok 12 - ... and last_name should start out undefined
ok 13 - ... and setting its value should succeed
1..13
# Looks like you failed 1 test of 13.
Dubious, test returned 1 (wstat 256, 0x100)
Failed 1/13 subtests

Test Summary Report
-------------------
t/run.t (Wstat: 256 Tests: 13 Failed: 1)
  Failed test:  9
  Non-zero exit status: 1
Files=1, Tests=13,  0 wallclock secs ( 0.03 usr 0.00 sys + 0.42 cusr 0.02 csys = 0.47 CPU)
Result: FAIL
```

Listing 4

```
sub full_name : Tests(no_plan) {
     my $test   = shift;
     my $person = $test->class->new;
     can_ok $person, 'full_name';

     throws_ok { $person->full_name }
         qr/^Both first and last names must be set/,
         '... and full_name() should croak() if the either name is not set';

     $person->first_name('John');

     throws_ok { $person->full_name }
         qr/^Both first and last names must be set/,
         '... and full_name() should croak() if the either name is not set';

     $person->last_name('Public');
     is $person->full_name, 'John Public',
       '... and setting its value should succeed';
 }
```

Listing 5

```
 package Person::Employee;

 use Moose;
 extends 'Person';

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

 1;
```

Und die Testklasse dafür:

```
 package Test::Person::Employee;

 use Test::Most;
 use base 'Test::Person';

 sub class {'Person::Employee'}

 sub employee_number : Tests(3) {
     my $test = shift;
     my $employee = $test->class->new;
     can_ok $employee, 'employee_number';
     ok !defined $employee->employee_number,
         '... and employee_number should not start out defined';
     $employee->employee_number(4);
     is $employee->employee_number, 4,
         '... but we should be able to set its value';
 }

 1;
```

Listing 6

Und jetzt sind alle Tests erfolgreich und wir können zurückkehren und die geplante Anzahl der Tests setzen:

```
All tests successful.
Files=1, Tests=14,  0 wallclock secs ( 0.03
usr  0.00 sys +  0.47 cusr  0.02 csys =  0.52
CPU)
Result: PASS
```

Tests erben

Bis jetzt wirst Du vielleicht auf den Code schauen und sagen "das ist jede Menge Arbeit nur um eine Klasse zu testen". Wenn das alles wäre, hättest Du absolut recht wenn Du `Test::Class` vergessen würdest. Aber lass uns ansehen wie `Test::Class` glänzt, indem wir eine Subklasse von `Person` schreiben, die wir `Person::Employee` nennen. Wir halten es einfach, indem wir nur eine Methode `employee_number`

schreiben, aber Du wirst schnell die Vorzüge verstehen (siehe Listing 6).

Beachte, dass wir nicht von `Test::Class` erben, sondern von `Test::Person`, genauso wie `Person::Employee` von `Person` erbt. Wir haben noch die `class`-Methode überschrieben, um sicherzustellen, dass die Tests wissen welche Klasse sie verwenden.

Jetzt müssen wir noch `Test::Person::Employee` zu `t/run.t` hinzufügen:

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

use lib 't/tests';

use Test::Person;
use Test::Person::Employee;

Test::Class->runtests;
```

Und wenn wir `t/run.t` laufen lassen:

```
All tests successful.
Files=1, Tests=31,  1 wallclock secs (
        0.25 cusr +  0.06 csys =  0.31 CPU)
```

Wow! Einen Moment. Wir haben nur drei Tests hinzugefügt. Wir haben mit 14 begonnen. Wie können wir jetzt 31 haben?

Weil `Test::Person::Employee` die Tests von `Test::Person` *geerbt* hat. Das bedeutet, dass die ursprünglichen 14 Tests plus 14 geerbten Tests und die 3 hinzugefügten Tests die 31 Tests ergeben! Aber das sind keine überflüssigen Tests. Schau Dir die neuen Testausgabe an in Listing 7 an.

Weil wir den Klassennamen nicht direkt in den Tests verwendet haben und weil `Test::Person::Employee` die `class`-Methode überschrieben hat, laufen diese neuen Tests gegen Instanzen von `Person::Employee` und nicht `Person`. Das zeigt uns, dass wir das vererbte Verhalten nicht kaputt gemacht haben. Wenn wir das Verhalten einer dieser Methoden ändern müssen – wie wir es in Objektorientiertem Code erwarten können – müssen wir nur die dazugehörige Testmethode überschreiben. Was, wenn z.B. Angestellte ihren vollen Namen im Format "Nachname, Vorname" ausgeben müssen?

```
sub full_name {
    my $self = shift;

    unless ( $self->
        first_name && $self->last_name ) {
        Carp::croak("Both first and
                    last names must be set");
    }

    return $self->
        last_name . ', ' . $self->first_name;
}
```

```
# Test::Person::Employee->constructor
ok 16 - Person::Employee->can('new')
ok 17 - ... and the constructor should succeed
ok 18 - ... and the object it returns isa Person::Employee
#
# Test::Person::Employee->employee_number
ok 19 - Person::Employee->can('employee_number')
ok 20 - ... and employee_number should not start out defined
ok 21 - ... but we should be able to set its value
#
# Test::Person::Employee->first_name
ok 22 - Person::Employee->can('first_name')
ok 23 - ... and first_name should start out undefined
ok 24 - ... and setting its value should succeed
#
# Test::Person::Employee->full_name
ok 25 - Person::Employee->can('full_name')
ok 26 - ... and full_name() should croak() if the either name is not set
ok 27 - ... and full_name() should croak() if the either name is not set
ok 28 - ... and setting its value should succeed
#
# Test::Person::Employee->last_name
ok 29 - Person::Employee->can('last_name')
ok 30 - ... and last_name should start out undefined
ok 31 - ... and setting its value should succeed
```

Listing 7

Die passende Testmethode in `Test::Person::Employee` könnte aussehen, wie in Listing 8 dargestellt:

Mach diese Änderungen und alle Tests werden bestehen. `Test::Person::Employee` wird seine eigene `full_name` Testmethode aufrufen und nicht die ihrer Elternklasse.

Refaktorierung von Testklassen

Refaktorierung mit Methoden

Es gibt einige Dubletten im `full_name` Test, die ausgelagert werden sollten. In unserer `Test::Person`-Klasse könnte das so aussehen - siehe Listing 9.

Und in `Test::Person::Employee`:

```
sub full_name : Tests(no_plan)
    my $test   = shift;
    $test->_full_name_validation;
    my $person = $test->class->new(
        first_name => 'Mary',
        last_name  => 'Jones',
    );
    is $person->full_name,
      'Jones, Mary',
      'The employee name should render
                                    correctly';
}
```

Wie in jedem anderen OO-Code, erben wir die `_full_name_validation`-Methode und können sie mit unseren Subklassen teilen.

```
 sub full_name : Tests(no_plan) {
     my $test   = shift;
     my $person = $test->class->new;
     can_ok $person, 'full_name';

     throws_ok { $person->full_name }
     qr/^Both first and last names must be set/,
       '... and full_name() should croak() if the either name is not set';

     $person->first_name('John');

     throws_ok { $person->full_name }
     qr/^Both first and last names must be set/,
       '... and full_name() should croak() if the either name is not set';

     $person->last_name('Public');
     is $person->full_name, 'Public, John',
       '... and setting its value should succeed';
 }
```

Listing 8

```
sub full_name : Tests(no_plan)
     my $test   = shift;
     $test->_full_name_validation;
     my $person = $test->class->new(
         first_name => 'John',
         last_name  => 'Public',
     );
     is $person->full_name, 'John Public',
       'The name of a person should render correctly';
 }

 sub _full_name_validation {
     my ( $test, $person ) = @_;
     my $person = $test->class->new;
     can_ok $person, 'full_name';

     throws_ok { $person->full_name }
         qr/^Both first and last names must be set/,
         '... and full_name() should croak() if the either name is not set';

     $person->first_name('John');

     throws_ok { $person->full_name }
         qr/^Both first and last names must be set/,
         '... and full_name() should croak() if the either name is not set';
 }
```

Listing 9

Refakturierung mit "fixtures"

Beim Schreiben der Testklassen sind die `startup` und die `shutdown` Methoden sehr praktisch, aber sie laufen nur zu Beginn und am Ende der Testklasse. Aber manchmal braucht man Code, der vor und nach jeder Testmethode läuft. Zum Beispiel in unserem Code von oben haben viele der Testmethoden die folgende Zeile Code:

```
my $person = $test->class->new;
```

Jetzt willst Du das sicherlich nicht jedes Mal neu schreiben. Also kannst Du etwas nutzen, das als *fixture* bekannt ist. Ein *fixture* ist "gesicherter Status" für Deine Tests, gegen den sie laufen. Das erlaubt es Dir, einigen duplizierten Code aus Deinen Tests zu entfernen und eine kontrollierte Umgebung zu haben. Du kannst so etwas machen:

```
sub setup : Tests(setup) {
    my $test   = shift;
    my $class  = $test->class;
    $test->{person} = $class->new;
}
```

Oder wenn Du mit einem bestimmten Satz an Daten beginnen willst:

```
sub setup : Tests(setup) {
    my $test   = shift;
    my $class  = $test->class;
    $test->{person} = $class->new(
        first_name => 'John',
        last_name  => 'Public',
    );
}
```

Jetzt kannst Du in den Testmethoden einfach `$test->{person}` verwenden (wenn Du willst, kannst Du daraus eine Methode machen), um eine neue Instanz der zu testenden Klasse zu bekommen, ohne ständig den Code zu wiederholen.

Die entsprechende `teardown`-Methode ist nützlich wenn Du pro Test aufräumen musst. Wir werden später weitere dieser Methoden behandeln.

Unser Testerleben vereinfachen

Auto-discovering der Testklassen

Zum jetzigen Zeitpunk fängt Du vielleicht an zu verstehen wie `Test::Class` das managen einer großen Code-Basis vereinfachen kann. Aber was ist damit `Test::Class` Tests zu erleichtern? Das erste Problem ist unser Hilfsskript - `t/run.t`:

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

use lib 't/tests';

use Test::Person;
use Test::Person::Employee;

Test::Class->runtests;
```

Bis jetzt sieht das nicht so schlimm aus, aber wenn wir beginnen immer mehr Klassen hinzuzufügen, wird das unhandlich. Was passiert, wenn Du vergisst eine Testklasse hinzuzufügen? Deine Klasse könnte kaputt sein, aber da die Testklasse nicht aufgerufen wird, woher willst Du das wissen? Also lass uns das so fixen, dass unsere Tests automatisch erkannt werden.

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

use Test::Class::Load qw<t/tests>;
Test::Class->runtests;
```

Teile `Test::Class::Load` (kommt mit `Test::Class` mit) einfach mit, in welchen Verzeichnissen Deine Testklassen sind und es wird sie finden. Das macht es, indem es versucht alle Dateien mit der Endung `.pm` zu laden. Wenn Du also "Hilfsmodule" hast, die keine `Test::Class` Tests sind, speichere sie in einem extra Verzeichnis.

Eine gemeinsame Basisklasse nutzen

Da das hier Programmierung ist, wollen wie gemeinsamen Code auslagern. Wir haben das zum Teil schon gemacht, aber da ist Platz für mehr. Du wirst sehen, dass beide Testklassen eine Methode haben, die den Namen der zu testenden Klasse liefert. Warum sollten wir das nicht in eine Basisklasse schieben, wo wir den Namen dieser Klasse doch "berechnen" können? Wir werden das in `t/tests/My/Test/Class.pm` schieben (siehe Listing 10.)

In `Person::Employee` müssen wir die `class`-Methode einfach löschen. In `Person` löschen wir die `class`- und die `startup`-Methode und erben sie von `My::Test::Class` anstatt von `Test::Class`. Jetzt liefert `class` immer die aktuelle Klasse, die wir testen und es ist garantiert, dass es geladen ist wenn die Klasse benutzt wird. Die neue `Test::Person`-Klasse sieht aus, wie in Listing 11 dargestellt.

```
package My::Test::Class;

use Test::Most;
use base qw<Test::Class Class::Data::Inheritable>;

BEGIN {
    __PACKAGE__->mk_classdata('class');
}

sub startup : Tests( startup => 1 ) {
    my $test = shift;
    ( my $class = ref $test ) =~ s/^Test::/;
    return ok 1, "$class loaded" if $class eq __PACKAGE__;
    use_ok $class or die;
    $test->class($class);
}

1;
```

Listing 10

Und die Testergebnisse für `Test::Person::Employee`:

```
All tests successful.
Files=1, Tests=32,  1 wallclock secs (
         0.33 cusr +  0.08 csys =  0.41 CPU)
```

Jetzt haben wir einen extra Test, aber das kommt durch das `ok 1` in der `My::Test::Class::startup`-Methode. Das wird zusätzlich beim Laden von `My::Test::Class` aufgerufen.

Tipp: Wenn die Klasse zu `BEGIN`-Zeit geladen werden muss, überschreibe die `startup`-Methode in der Test-Klasse. Aber stelle sicher, eine `class`-Methode zu haben.

Individuelle Test-Klassen laufen lassen

Wenn ich Tests laufen lasse, hasse ich es, meinen Editor zu verlassen nur um Tests von der Kommandozeile aus zu starten. Um das zu vermeiden, habe ich etwas ähnliches wie dieses Mapping in meiner `.vimrc`-Datei:

```
noremap ,t :!prove --merge -lv %<CR>
```

Danach brauche ich beim Tests-schreiben nur `,t` zu drücken und meine Tests laufen. Aber das funktioniert nicht in einer Testklasse. Die Testklasse wird zwar geladen, aber die Tests laufen nicht. Ich könnte einfach ein neues Mapping hinzufügen:

```
noremap ,T  :!prove -lv --merge t/run.t<CR>
```

Das Problem ist, dass dies *alle* meine Testklassen aufrufen würde. Wenn ich hunderte Tests hätte, möchte ich nicht die ganze Ausgabe durchgehen um die fehlgeschlagenen Tests zu sehen. Ich möchte stattdessen nur eine einzelne Testklasse laufen lassen. Um das zu erreichen, ändere ich das Mapping so, dass es den Pfad zu meinen Testklassen einbindet.

```
noremap ,t
     :!prove -lv --merge -It/tests %<CR>
```

Ich lösche die `Test::Class->runtests`-Zeile von `t/run.t` (sonst werden meine Tests zweimal ausgeführt wenn ich die ganze Testsuite laufen lasse). Da ich jetzt eine gemeinsame Basisklasse habe, füge ich die folgende Zeile in `My::Test::Class` ein:

```
INIT { Test::Class->runtests }
```

Jetzt kann ich mit `,t` die Tests der Datei, die ich ändere, laufen lassen - egal ob ich in einem Standard `Test::Most` Programm bin oder in einer meiner neuen Testklassen.

Wenn Du die Tests für `Test::Person::Employee` laufen lässt, siehst alle 32 Tests weil `Test::Class` alle Tests der aktuellen Klasse laufen lässt und auch die der Klassen von der sie erbt. Wenn Du die Tests von `Test::Person` laufen lässt, siehst Du nur 15 Tests, was das gewünschte Verhalten ist.

Wenn Du Emacs bevorzugst, kannst Du das in Deiner `~/.emacs` Datei einfügen (siehe Listing 12).

Das bindet es an `C-c t` und Du kannst so tun als wärst Du so cool wie `vim`-Nutzer (ich mache nur Spaß! Stoppe die Hassmail).

```
package Test::Person;

use Test::Most;
use base 'My::Test::Class';

sub constructor : Tests(3) {
    my $test  = shift;
    my $class = $test->class;
    can_ok $class, 'new';
    ok my $person = $class->new, '... and the constructor should succeed';
    isa_ok $person, $class, '... and the object it returns';
}

sub first_name : Tests(3) {
    my $test   = shift;
    my $person = $test->class->new;
    can_ok $person, 'first_name';
    ok !defined $person->first_name,
      '... and first_name should start out undefined';
    $person->first_name('John');
    is $person->first_name, 'John', '... and setting its value should succeed';
}

sub last_name : Tests(3) {
    my $test   = shift;
    my $person = $test->class->new;
    can_ok $person, 'last_name';
    ok !defined $person->last_name,
      '... and last_name should start out undefined';
    $person->last_name('Public');
    is $person->last_name, 'Public', '... and setting its value should succeed';
}

sub full_name : Tests(4) {
    my $test   = shift;
    $test->_full_name_validation;
    my $person = $test->class->new(
        first_name => 'John',
        last_name  => 'Public',
    );
    is $person->full_name, 'John Public',
      '... and setting its value should succeed';
}

sub _full_name_validation {
    my ( $test, $person ) = @_;
    my $person = $test->class->new;
    can_ok $person, 'full_name';

    throws_ok { $person->full_name }
        qr/^Both first and last names must be set/,
        '... and full_name() should croak() if the either name is not set';

    $person->first_name('John');

    throws_ok { $person->full_name }
        qr/^Both first and last names must be set/,
        '... and full_name() should croak() if the either name is not set';
}

1;
```

Listing 11

```
(eval-after-load "cperl-mode"
    '(add-hook 'cperl-mode-hook
        (lambda () (local-set-key "\C-ct" 'cperl-prove))))

(defun cperl-prove ()
    "Run the current test."
    (interactive)
    (shell-command (concat "prove -lv --merge -It/tests "
        (shell-quote-argument (buffer-file-name)))))
```

Listing 12

Handhabung der Startup/Setup/Teardown/Shutdown Methoden

Oft stellt sich heraus, dass wir speziellen Code am Anfang und Ende einer Klasse und am Anfang und Ende jeder Testmethode laufen lassen müssen. Das kann nützlich sein, um eine Datenbankverbindung aufzubauen, temp-Dateien zu löschen, Testbedingungen herzustellen usw. `Test::Class` kann uns dabei helfen.

Der Einfachheit halber, werden wir die `Test::Class`-Methoden, die das können, als *test control*-Methoden bezeichen.

`Test::Class` bietet vier solche Methoden:

- `startup`: Diese Methode wird *einmal* für jede Klasse aufgerufen – bevor irgendein Test läuft.
- `shutdown`: Diese Methode läuft *einmal* für jede Klasse nachdem alle Tests gelaufen sind.
- `setup`: Diese Methode wird vor jedem Test aufgerufen.
- `teardown`: Diese Methode wird nach jeder Testmethode aufgerufen.

`startup` und `shutdown`

Eine übliche Funktion für die `startup` und `shutdown` Methoden ist es, eine Datenbank auf- und abzubauen:

```
package Tests::My::Resultset::Customer;

use base 'My::Test::Class';

sub startup : Tests(startup) {
    my $test = shift;
    $test->_connect_to_database;
}

sub shutdown : Tests(shutdown) {
    my $test = shift;
    $test->_disconnect_from_database;
}

...
```

Wenn eine Klasse geladen wird, ist der erste Code, der läuft, die startup Methode. Am Ende des Tests wird die shutdown Methode aufgerufen und die Datenbankverbindung abgebaut. Merke, dass der Rest der Tests nicht gestartet wird, falls die startup Methode Tests enthält und einer davon fehlschlägt. Aber alle Tests von Superklassen werden weiterhin gestartet.

```
sub startup : Tests(startup) {
    ok 0;   # the test class will abort here
}
```

Wenn das passiert, wird die `shutdown` Methode *nicht* aufgerufen.

`setup` und `teardown`

Natürlich müssen wir auch ggf. Code vor und nach jeder Testmethode laufen lassen. So macht man das:

```
sub setup : Tests(setup) {
    my $test = shift;
    $test->_start_db_transaction;
}

sub check_privileges : Tests(no_plan) {
    my $test = shift;
    $test->_load_priviledge_fixture;
    ...
}

sub teardown : Tests(teardown) {
    my $test = shift;
    $test->_rollback_db_transaction;
}
```

Der obige Code startet eine Datenbanktransaktion vor jeder Testmethode. Die `check_privileges` Methode lädt ihre eigenen Testbedingungen und die `teardown` Methode macht die Transaktion rückgängig um sicherzustellen, dass der nächste Test eine unverfälschte Datenbank hat. Beachte, dass bei einem fehlgeschlagenen Test in der `setup` Methode die `teardown` Methode dennoch aufgerufen wird. Das unterscheidet sich von der `startup` Methode, da `Test::Class` zum nächsten Test übergeht und annimmt, dass Du weitermachen willst.

Überschreiben von Testkontroll-Methoden

User, die neu im Umgang mit `Test::Class` sind, haben typischerweise zwei Probleme: Sie denken, dass mehr Testkontroll-Methoden laufen als sie es erwartet haben oder ihre Testkontroll-Methoden laufen in einer Reihenfolge, die sie nicht erwartet haben. Als Beispiel nehmen wir an, dass wir folgendes in der Basis-Testklasse haben:

Ausführungsreihenfolge kontrollieren

```
sub connect_to_db : Tests(startup) {
    my $test = shift;
    $test->_connect_to_db;
}
```

Und in einer abgeleiteten Testklasse:

```
sub assert_db : Tests(startup => 1) {
    my $test = shift;
    ok $test->_is_connected_to_db,
       'We still have a database connection';
}
```

Das wird wahrscheinlich fehlschlagen und Deine Tests werden nicht laufen. Warum? Weil `Test::Class` die Tests in einer Testklasse in alphabetischer Reihenfolge laufen lässt. Weil das *geerbte* Tests in Deiner Testklasse mit einschließt - `connect_to_db` wurde geerbt - und weil das *nach* `assert_db` einsortiert wird, wird es danach ausgeführt. Demzufolge prüfst Du die Datenbankverbindung *bevor* sie existiert.

Das Problem hier ist, dass es OO-Code ist und Du Dich nicht auf die Ausführungsreihenfolge verlassen solltest. Der Fix ist einfach. Benenne beide startup-Methoden in `startup` und rufe in der Kindklasse die Methode der Superklasse auf:

```
sub startup : Tests(startup) {
    my $test = shift;
    $test->SUPER::startup;
    die unless $test->_is_connected_to_db,
      'We still have a database connection';
}
```

Das funktioniert, weil `Test::Class` weiß, dass Du die Methode überschrieben hast und Du kannst sie ganz einfach selbst aufrufen.

Warnung: Beachte, dass wir jetzt `die` in der startup-Methode verwenden und keinen Test machen, weil `Test::Class` nicht wissen kann, ob Du wirklich die Methode der Superklasse aufrufst oder nicht. Das führt dazu, dass es nicht die wirkliche Anzahl der Tests kennen kann. Also nutzen wir `die` anstatt auf einen Fehler des Tests zu vertrauen, um die `startup` Methode zu beenden.

Tipp: Wegen den oben genannten Gründen solltest Du keine Tests in den Testkontroll-Methoden haben.

Überprüfen was ausgeführt wird

Nehmen wir an, Du hast eine Webseite, die Informationen anzeigt. Authentifizierte Nutzer haben zusätzliche Features. Du könntest das so testen:

```
sub unauthenticated_startup : Test(startup) {
    my $test = shift;
    $test->_connect_as_unauthenticated;
}
```

Und in Deiner "authenticated" Subklasse:

```
sub authenticated_startup : Test(startup) {
    my $test = shift;
    $test->_connect_as_authenticated;
}
```

Und auch hier werden die Tests wahrscheinlich fehlschlagen. Und zwar, weil `authenticated_startup` vor `unauthenticated_startup` ausgeführt wird und Du wahrscheinlich als nicht-authentifizierter User in der "authenticated" Subklasse agierst. Wie auch immer, Du wirst diesmal `unauthenticated_startup` wahrscheinlich gar nicht laufen lassen müssen. Gib auch diesmal den Tests den gleichen Namen, aber rufe die Methode der Elternklasse *nicht* auf.

```
sub startup : Test(startup) {
    my $test = shift;
    $test->_connect_as_authenticated;
}
```

Beachte, dass wir keine Tests in der Kontrollmethode haben. Wenn die Verbindung nicht klappt, wirf einen Fehler.

Performance

Mit `Test::Class::Load` kannst Du alle Deine Tests in einem Prozess ausführen:

```
use Test::Class::Load qw<path/to/tests>;
```

Das lädt alle Tests und alle Module, die getestet werden, auf einmal. Es kann eine große Performanzverbesserung bedeuten wenn Du große Module wie `Catalyst` oder `DBIx::Class` lädst. Aber bedenke, dass Du jetzt alle Klassen in einem einzigen Prozess lädst und es potenzielle Nachteile geben kann. Zum Beispiel, wenn eine Deiner Klassen ein Singleton oder eine globale Variable von der eine andere Klasse abhängt ändert, kannst Du unerwartete Ergebnisse bekommen. Auch laden viele Klassen Module, die das Verhalten von Perl global ändern. Du kannst die CPAN-Module nach `UNIVERSAL::` oder `CORE::GLOBAL::` durchsuchen, nur um mal zu sehen wie viele Klassen das tun.

Bugs, die eine globale Änderung beinhalten, sind sehr schwierig aufzufinden. Du musst selbst entscheiden, ob die Vorteile von `Test::Class` den Nachteilen überwiegen. Meine Erfahrung ist, dass diese Bugs sehr mühsam zu lösen sind, aber bei der Suche finde ich hin und wieder Probleme in meiner Code-Basis, die ich sonst nicht gefunden hätte. Für mich ist `Test::Class` ein Gewinn, trotz des gelegentlichen Frusts.

Diejenigen, die nicht den ganzen Code in einem einzigen Prozess laufen lassen wollen, erstellen oft "Treiber" dafür:

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

use Test::Person;
Test::Class->runtests;
```

und:

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

use Test::Person::Employee;
Test::Class->runtests;
```

Natürlich solltest Du den Aufruf von `runtests` weglassen wenn Du das in der Basisklasse im `INIT`-Block eingefügt hast.

Verhalten Deiner Testklassen wie `xUnit`-Klassen

In xUnit-artigen Tests ist dies ein kompletter Test:

```
sub first_name : Tests(tests => 3) {
    my $test   = shift;
    my $person = $test->class->new;
    can_ok $person, 'first_name';
    ok !defined $person->first_name,
      '... and first_name should start out
                                undefined';
    $person->first_name('John');
    is $person->first_name, 'John', '... and
          setting its value should succeed';
}
```

In der TAP-Welt würden wir das als drei Tests ansehen, aber xUnit sagt, dass wir drei Zusicherungen haben um ein Feature zu validieren - also haben wir einen Test. Jetzt ist es ein weiter Weg von TAP-basierten Tests bevor es für xUnit-Nutzer funktioniert, aber es gibt eine Sache, die wir tun können. Nehmen wir an, wir haben einen Test mit 30 Zusicherungen und die vierte Zusicherung schlägt fehl. Viele xUnit-Programmierer argumentieren wenn einmal eine Zusicherung fehlschlägt ist der Rest der Informationen in dem Test unzuverlässig. Demzufolge sollten die Tests blockiert werden. Egal ob Du dem zustimmst oder nicht (Ich hasse die Tatsache, dass jUnit zum Beispiel den Stop der Testmethode verlangt), kannst Du dieses Verhalten mit `Test::Class` bekommen. Benutze einfach `Test::Most` anstelle von `Test::More` und füge das in die Test-Basisklasse ein:

```
BEGIN { $ENV{DIE_ON_FAIL} = 1 }
```

Weil jede Testmethode in `Test::Class` in einem `eval` ist, das diese Testmethode stoppen wird, wird die passende `teardown` Methode (wenn vorhanden) ausgeführt und die Tests werden mit der nächsten Testmethode weitermachen.

Ich bin kein großer Fan davon, aber Du kannst anderer Meinung sein.

Fazit

Während für viele Projekte gut `Test::More`-Programme nutzen können, können größere Projekt mit Skalierungsproblemen enden. `Test::Class` bietet bessere Möglichkeiten für das Managen von Tests, Auslagern von gemeinsamen Code und für Tests, die den Produktivcode besser widerspiegeln.

Hier eine schnelle Zusammenfassung der oben aufgeführten Tipps:

- Benenne Testklassen konsistent nach den Klassen, die sie testen.
- Wenn möglich, mache das Gleiche mit den Testmethoden.
- Verwende keinen Konstruktortest mit dem Namen `new`.
- Erstelle Dir eine eigene `Test::Class` Basisklasse.
- Abstrahiere den Namen der zu testenden Klasse in eine `class`-Methode in der Basisklasse.
- Benenne Testkontroll-Methoden nach ihrem Attribut.
- Entscheide für jeden Fall einzeln, ob die Kontrollmethode der Elternklasse aufgerufen werden muss.
- Mache keine Tests in den Testkontroll-Methoden.

Danksagung

Dank an Adrian Howard für `Test::Class` und für die Tipps wie es einfacher zu nutzen ist. Dank auch an David Wheeler für die nützlichen Kommentare auch wenn das für den ersten Entwurf war, der für eine paar Jahren geschrieben wurde. I frage mich, ob er sich daran erinnert? :)

[1] `Moose` wurde in $foo Nr. 4 (Winter 2007) vorgestellt

Curtis 'Ovid' Poe

WHO TO BLAME?

ctgetreports

Als Autor von mehreren CPAN-Modulen ist man froh, dass es CPAN-Testers gibt: Man bekommt kostenlos seine Module in allen möglichen Konstellationen getestet. Von Perl 5.005 auf FreeBSD bis zu Perl 5.11.0 auf Windows ist alles dabei. Leider funktionieren einige Module nicht gleich auf allen Systemen und man bekommt eine Mail zugeschickt, die ungefähr so aussieht:

```
Dear Renee Baecker,

 ...

 Win32-FileNotify-0.2:
 - MSWin32-x86-multi-thread / 5.10.0:
   - FAIL http://nntp.x.perl.org/group/
          perl.cpan.testers/3227029

 ...

 Thanks,
 The CPAN Testers
```

Manchmal gibt es aber auch Fehler, die sich nicht erklären lassen. Wenn zum Beispiel 999 mal mit Perl 5.8.8 auf Ubuntu alles gut ging und das 1000. Mal geht etwas schief. Dann ist es natürlich ganz praktisch, wenn man weiß an wen man sich wenden muss. Dazu kann man das Programm `ctgetreports` aus dem Modul `CPAN::Testers::ParseReport` nehmen:

```
C:\>ctgetreports --report 3227029
FAIL  3227029 meta:perl[5.10.0]
   conf:archname[MSWin32-x86-multi-thread]
  conf:usethreads[] conf:optimize[-s -O2]
  meta:writer[CPAN-Reporter-1.1705]
  meta:from[dagolden@cpan.org (DAGOLDEN)]
```

Man kann sich natürlich noch mehr Informationen mit dem Programm holen. Insgesamt sehr nützlich!

Renée Bäcker

XS – Perl mit C erweitern

Teil 3: Fortgeschrittene XS-Programmierung 2

Dieser letzte Teil des XS-Tutorials wird einige spezielle Aspekte der XS-Programmierung beleuchten. Als erstes wäre da das Thema Portabilität, das spätestens dann interessant wird, wenn wir darüber nachdenken, unser XS-Modul ins CPAN zu stellen. Dann muss der Code nicht nur mit unserer Perl-Version funktionieren, sondern mit hunderten anderer Versionen.

Portabilität von XS-Code

Es gibt verschiedene Komponenten, die Einfluss auf die Portabilität eines XS-Moduls haben:

• Der Perl-Interpreter, und die Optionen, mit denen er gebaut wurde. So kann es beispielsweise passieren, dass sich ein XS-Modul mit einem Perl ohne ithreads prima bauen lässt, mit einem Perl mit ithreads aber nicht. Wesentlich häufiger passiert es sogar, dass man sein XS-Modul mit einer relativ neuen Perl-Version entwickelt, es aber dann mit älteren Perl-Versionen nicht kompiliert.

• Das Betriebssystem, auf dem der Perl-Interpreter läuft. Sobald man auf XS- oder C-Ebene programmiert, gibt es nur noch eine relativ kleine Menge an Sprachfunktionalität, auf die man sich bei allen Betriebssystemen verlassen kann. Verwendet man also Unix-spezifische Funktionen, darf man sich nicht wundern, wenn das Modul nicht unter Windows funktioniert.

• Der Compiler, mit dem unser XS-Modul übersetzt wird. Dieser Punkt wird gerne übersehen, aber verschiedene C-Compiler verhalten sich teilweise vollkommen unterschiedlich, sobald man sich abseits der im C-Standard beschriebenen Sprachdefinition bewegt. So unterstützt beispielsweise der GCC Spracherweiterungen, die von keinem anderen Compiler akzeptiert werden.

Bei den beiden letzten Punkten ist es im Allgemeinen hilfreich, sich beim Schreiben von XS-Code auf Konstrukte zu beschränken, die entweder im C-Standard beschrieben sind oder von der Perl-API angeboten werden. Denn schließlich hat Perl selbst die gleichen Probleme: es muss sich auf den verschiedensten Plattformen und mit diversen Compilern bauen lassen. Demzufolge gibt es Unmengen von bereits geschriebenem (und getestetem) portablen C-Code, auf den man als XS-Entwickler bevorzugt zurückgreifen sollte.

Verlässt man sich nun aber auf die Perl-API, ist man wieder beim ersten Punkt: Da sich die API mit der Zeit ändert, ist es schwierig, die Portabilität zu alten Perl-Versionen sicherzustellen. Es gibt aber glücklicherweise ein Perl-Modul, das einem hierbei unter die Arme greift: `Devel::PPPort`. Einzige Aufgabe von `Devel::PPPort` ist die Erzeugung der Datei *ppport.h*:

```
$ perl -MDevel::PPPort \
  -eDevel::PPPort::WriteFile
```

ppport.h ist in erster Linie eine Headerdatei, die per `#include` in einer XS-Datei benutzt werden kann. Dies sieht dann typischerweise so aus wie in unserem Beispiel aus dem ersten Teil:

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

#include "ppport.h"


MODULE = Hello          PACKAGE = Hello
```

Aber was genau macht nun *ppport.h*? Es stellt primär einen Großteil der aktuellen Perl-API zur Verfügung, wenn ein XS-

Modul mit einer alten Perl-Version gebaut werden soll. Als Basis dient dabei die jeweils aktuelle Entwicklungsversion von Perl, die API wird dann von *ppport.h*, soweit sinnvoll möglich, bis Perl 5.003 zurückportiert. Somit kann man sich fast bedenkenlos aus der neuesten Perl-API bedienen; *ppport.h* sorgt dann dafür, dass der Code auch mit möglichst vielen Perl-Versionen läuft. Wer noch eine ältere Perl-Version installiert hat (5.8.4 oder früher), kann gerne einmal versuchen, das `AdvancedXS`-Modul aus dem zweiten Teil ohne *ppport.h* zu bauen. Dazu muss man lediglich die Zeile

```
#include "ppport.h"
```

entfernen. Übersetzt man das Modul dann erneut und versucht, die Funktion `fibonacci3` aufzurufen, passiert in etwa folgendes:

```
$ perl5.8.4 -Mblib -MAdvancedXS -le \
  'print join ", ", fibonacci3(10)'
/tmp/perl/install/default/perl-5.8.4/bin/per
l5.8.4: symbol lookup error:
/home/mhx/src/perl/xstutorial/2008/advanced/
AdvancedXS/blib/arch/auto/AdvancedXS/Advance
dXS.so: undefined symbol: mXPUSHu
```

Offensichtlich gab es also das Makro `mXPUSHu` damals noch nicht.

Aber *ppport.h* kann noch mehr. Es ist nicht nur eine C-Headerdatei, sondern gleichzeitig auch ein Perl-Skript. So gibt

```
$ perl ppport.h --help
```

wie gewohnt eine Übersicht der Optionen, und `perldoc ppport.h` zeigt die vollständige Dokumentation an. Mit Hilfe des Skripts kann man sich zum Beispiel Informationen zur Perl-API anzeigen lassen:

```
$ perl ppport.h --api-info mXPUSHu
=== mXPUSHu ===

Supported at least starting from perl-5.9.2.
Support by ppport.h provided back to
perl-5.003.
Depends on: EXTEND, PUSHmortal, STMT_END,
STMT_START, sv_setuv.
```

Wir sehen also, dass `mXPUSHu` mit Perl 5.9.2 eingeführt wurde. (Es wurde dann von dort auch in Perl 5.8.5 übernommen.) Auch wenn diese Funktion recht nützlich ist, wäre es doch lästig, jeden Teil der verwendeten Perl-API auf diese Art zu überprüfen. Das kann *ppport.h* aber zum Glück selbst: die Hauptaufgabe des Skripts ist es, sich den Quelltext eines XS-Moduls anzuschauen, auf potenzielle Probleme zu überprüfen, und gegebenenfalls den Code sogar anzupassen.

Schauen wir uns dazu folgendes Beispiel an:

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

int is_hello(SV *sv)
{
  char *str = SvPV_nolen(sv);
  return strEQ(str, "hello");
}

MODULE = PPPortTest     PACKAGE = PPPortTest

void
xs_crap(sv)
    SV *sv
  CODE:
    if (!is_hello(sv))  // check
      Perl_croak("not friendly");
```

Das Beispiel ist zwar sinnlos, sieht aber auf den ersten Blick gar nicht so furchtbar aus und wird sehr wahrscheinlich auch in den meisten Fällen funktionieren. Lassen wir nun *ppport.h* laufen:

```
$ perl ppport.h
Scanning ./PPPortTest.xs ...
=== Analyzing ./PPPortTest.xs ===
*** Doesn't pass interpreter argument aTHX
to Perl_croak
Uses SvPV_nolen, which depends on SV_GMAGIC,
SvPVX, sv_2pv_flags, sv_2pv
File needs sv_2pv_flags, adding static
request
Needs to include 'ppport.h'
*** Uses 1 C++ style comment, which is not
portable
Analysis completed
Suggested changes:
--- ./PPPortTest.xs     2009-01-18
13:15:33.000000000 +0100
+++ ./PPPortTest.xs.patched     2009-01-25
20:34:20.000000000 +0100
@@ -1,6 +1,8 @@
 #include "EXTERN.h"
 #include "perl.h"
 #include "XSUB.h"
+#define NEED_sv_2pv_flags
+#include "ppport.h"

 int is_hello(SV *sv)
 {
@@ -14,5 +16,5 @@
 xs_crap(sv)
     SV *sv
   CODE:
-    if (!is_hello(sv))  // check
-      Perl_croak("not friendly");
+    if (!is_hello(sv))  /* check */
+      Perl_croak(aTHX_ "not friendly");
```

Als erstes fällt auf, dass *ppport.h* von selbst nach Dateien sucht. Der Hintergrund ist, dass für eine genaue Analyse alle Dateien eines Moduls untersucht werden müssen. Man kann die zu untersuchenden Dateien aber auch explizit als

Parameter übergeben. Anschließend analysiert *ppport.h* die gefundenen Dateien. Es bemängelt, dass beim Aufruf von `Perl_croak` das Interpreterargument `aTHX` nicht übergeben wurde. Ein typischer Fehler, der erst dann auffällt, wenn man das Modul mit einem Perl mit ithreads testet. Die Funktion `SvPV_nolen` ist abhängig von `sv_2pv_flags`, diese wiederum wird von *ppport.h* nur bereitgestellt, wenn man sie explizit per `NEED_sv_2pv_flags` anfordert. An diesem Punkt merkt *ppport.h* auch, dass es sich selbst in den Quelltext einbinden muss. Zu guter Letzt wird im Quelltext auch noch ein C++-Kommentar verwendet, der von sehr alten C-Compilern nicht unterstützt wird und durch einen C-Kommentar ersetzt werden sollte.

Danach erzeugt *ppport.h* dann einen Diff mit den Änderungen, die sich aus der Analyse ergeben. Diesen sollte man dann noch kurz überprüfen, in den meisten Fällen kann man ihn jedoch bedenkenlos in den eigenen Code einpflegen.

Manchmal kann es aber sinnvoll sein, das Verhalten von *ppport.h* anzupassen. Wenn zum Beispiel ein XS-Modul ohnehin nur bis zu einer bestimmten Perl-Version rückwärtskompatibel sein soll, kann man diese Version über die Option `--compat-version` angeben. Ist sichergestellt, dass immer ein C++- oder ein moderner C-Compiler verwendet wird, kann man die Option `--cplusplus` setzten:

```
$ perl ppport.h --compat-version=5.8.5 \
  --cplusplus
Scanning ./PPPortTest.xs ...
=== Analyzing ./PPPortTest.xs ===
*** Doesn't pass interpreter argument aTHX
to Perl_croak
Analysis completed
Suggested changes:
--- ./PPPortTest.xs     2009-01-18
13:15:33.000000000 +0100
+++ ./PPPortTest.xs.patched     2009-01-25
20:34:21.000000000 +0100
@@ -15,4 +15,4 @@
     SV *sv
   CODE:
     if (!is_hello(sv))  // check
-      Perl_croak("not friendly");
+      Perl_croak(aTHX_ "not friendly");
```

ppport.h passt also sein Verhalten an und schlägt lediglich vor, das Problem mit `Perl_croak` zu beheben. Es ist in diesem Fall nicht einmal notwendig, *ppport.h* mit einzubinden.

All das hat natürlich seinen Preis: Die Datei *ppport.h* ist ein ziemlich dicker Brocken. Aktuell hat sie etwa 170kB, Tendenz steigend. Es gibt jedoch eine Möglichkeit, die Größe etwas zu reduzieren. Startet man das Skript mit der Option `--strip`, schrumpft die Datei auf weniger als die Hälfte ihrer ursprünglichen Größe zusammen. Damit geht allerdings die Skriptfunktionalität komplett verloren. Diese lässt sich aber mit der Option `--unstrip` bei Bedarf wieder herstellen.

Es ist im Übrigen keine gute Idee, die Datei *ppport.h* aus dem *Makefile.PL* heraus zu generieren, denn es ist durchaus möglich, dass zukünftige Versionen von `Devel::PPPort` nicht völlig kompatibel sind. Daher sollte *ppport.h* immer direkt in eine Distribution mit aufgenommen werden.

Ein kleine Portabilitätsfalle soll zum Abschluss nicht unerwähnt bleiben. `h2xs` erzeugt im Normalfall eine Modul-Vorlage, die lediglich zu der Perl-Version rückwärtskompatibel ist, mit der `h2xs` ausgeführt wurde. Das ist jedoch meistens **nicht** das gewünschte Verhalten. Daher sollte man immer mit der Option `-b` bzw. `--compat-version` die kleinste Perl-Version spezifizieren, welche noch von dem neuen Modul unterstützt werden soll.

```
$ h2xs -b 5.5.0 -An FooBar
```

Exception Handling

Widmen wir uns nun einem völlig anderen Problem: Ressourcenlecks. Ein Ressourcenleck tritt immer dann auf, wenn eine Ressource reserviert, aber unbeabsichtigt nicht wieder freigegeben wird. Die bekannteste Form von Ressourcenlecks sind sicher Speicherlecks, aber auch geöffnete Dateien können zu Ressourcenlecks führen, wenn sie nicht wieder geschlossen werden. Sehen wir uns folgendes Beispiel an:

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

#include "ppport.h"

void fill_buffer(char *buffer, int buflen)
{
  if (buflen > 2048)
    Perl_croak(aTHX_ "buffer too large");
  memset(buffer, 'X', buflen);
}

MODULE = Exception      PACKAGE = Exception

SV *
fill(int size = 32)
  CODE:
    RETVAL = newSV(size);
    fill_buffer(SvPVX(RETVAL), size);
    SvCUR(RETVAL) = size;
    SvPOK_on(RETVAL);
  OUTPUT:
    RETVAL
```

Die XSUB `fill` erzeugt einen neuen `SV` und ruft danach `fill_buffer` auf, um den `PV`-Slot des `SV` mit X-Zeichen zu füllen. Anschließend werden noch die Länge und das `POK`-Flag gesetzt und der `SV` wird zurückgegeben. Die ganze Sache hat nur einen Haken: `fill_buffer` kann fehlschlagen. In diesem Fall würde der Aufruf von `Perl_croak` dafür sorgen, dass nicht wieder in die XSUB `fill` zurückgesprungen wird. Der `SV` würde nicht auf den Stack gelegt und wäre damit nicht mehr referenziert, wir hätten den `SV` und seinen Stringpuffer verloren – ein klassisches Speicherleck. Mit

```
$ perl -Mblib -MException \
  -e'while(){eval{fill(4096)}}'
```

können wir uns dieses Speicherleck in Aktion ansehen. Binnen Sekunden wird das Programm unseren gesamten Speicher belegt haben.

Eine Möglichkeit, dieses Speicherleck zu vermeiden, haben wir bereits kennen gelernt: wir können den `SV` vor dem Aufruf von `fill_buffer` sterblich machen. Das funktioniert zwar gut für `SV`s, nicht aber für andere Ressourcen. Wir wollen uns daher eine generellere Methode anschauen:

```
#include "EXTERN.h"
#include "perl.h"

#define NO_XSLOCKS
#include "XSUB.h"

#include "ppport.h"

void fill_buffer(char *buffer, int buflen)
{
  if (buflen > 2048)
    Perl_croak(aTHX_ "buffer too large");
  memset(buffer, 'X', buflen);
}

MODULE = Exception      PACKAGE = Exception

SV *
fill(int size = 32)
  PREINIT:
    dXCPT;
  CODE:
    RETVAL = newSV(size);
    XCPT_TRY_START {
      fill_buffer(SvPVX(RETVAL), size);
    } XCPT_TRY_END
    XCPT_CATCH {
      SvREFCNT_dec(RETVAL);
      XCPT_RETHROW;
    }
    SvCUR(RETVAL) = size;
    SvPOK_on(RETVAL);
  OUTPUT:
    RETVAL
```

Um die verschiedenen Exception-Handling-Makros benutzen zu können, muss unbedingt vor dem Einbinden der Datei *XSUB.h* `NO_XSLOCKS` definiert werden. In dem Abschnitt, der eine Exception fangen will, muss zudem mit `dXCPT` eine Zustandsvariable deklariert werden. Anschließend kann der Teil, innerhalb dessen potenziell eine Exception geworfen wird, durch `XCPT_TRY_START` und `XCPT_TRY_END` eingeklammert werden. Der Code nach diesem Block wird dann in jedem Fall ausgeführt, auch dann, wenn eine Exception geworfen wurde. Der `XCPT_CATCH`-Block wird dagegen nur im Fall einer Exception ausgeführt. Mittels `XCPT_RETHROW` wird die Exception weitergeworfen. Dies ist übrigens zwingend erforderlich. Es ist momentan innerhalb von XS-Code nicht ohne weiteres möglich, eine Exception komplett zu ignorieren.

Typemaps Reloaded: Objektorientiertes XS

Eine Perl-Klasse in XS zu implementieren ist erstaunlich einfach. Der Trick besteht einzig darin, eine passende *typemap*-Datei anzulegen. Nehmen wir an, wir wollen eine einfache Klasse `Color` zur Farbverwaltung schreiben. Der XS-Code dazu könnte zum Beispiel so aussehen:

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

#include "ppport.h"

typedef struct _color {
  int c_blue;
} color;

MODULE = Color  PACKAGE = Color

PROTOTYPES: DISABLE

color *
color::new()
  CODE:
    Newxz(RETVAL, 1, color);
  OUTPUT:
    RETVAL

void
color::DESTROY()
  CODE:
    Safefree(THIS);

int
color::blue()
  CODE:
    RETVAL = THIS->c_blue;
  OUTPUT:
    RETVAL

void
color::set_blue(val)
  int val
  CODE:
    THIS->c_blue = val;
```

Mit `struct` wird in C ein Aggregat spezifiziert. Ein solches Aggregat ist im Grunde vergleichbar mit einem Hash in Perl, allerdings sind die Elemente eines C-`struct`s geordnet und nicht dynamisch. In unserem Beispiel besteht das Aggregat nur aus einem Element, `c_blue`. Mittels `typedef` wird aus dem Aggregat `_color` der echte C-Typ `color`.

Dieser Typ speichert die einzelnen Farbkomponenten (im konkreten Fall nur blau) – die Attribute unserer Klasse – und bildet die Basis für alle Methoden der Perl-Klasse `Color`. XS-Methoden unterscheiden sich von XS-Funktionen dadurch, dass ihnen der Name eines Typs vorangestellt ist.

Jede XS-Methode bekommt implizit als erstes Argument `THIS` übergeben, innerhalb des Konstruktors gibt es den Parameter `CLASS`. Der Konstruktor legt mit `Newxz` ein neues, mit Nullen initialisiertes C-Objekt vom Typ `color` an. In `RETVAL` steht nun ein Zeiger auf dieses C-Objekt. Der Destruktor, wie alle anderen Methoden auch, bekommt diesen Zeiger in `THIS` übergeben. Mit `Safefree` kann der über `Newxz` bereitgestellte Speicher wieder freigegeben werden. Die Methoden `blue` und `set_blue` sind trivial.

Die Zeile `PROTOTYPES: DISABLE` weist im Übrigen den `xsubpp` an, für unsere XSUBs keine Prototypen zu generieren, da Prototypen bei Methoden wenig sinnvoll sind.

Bleibt nur noch die Frage zu klären, wie denn jetzt aus dem C-Objekt ein Perl-Objekt wird – und umgekehrt. Wie oben schon angedeutet ist dies Aufgabe der *typemap*. Bevor wir uns diese ansehen, wollen wir kurz einen Blick auf einige der Perl-Variablen werfen, die innerhalb des Template-Teils einer Typemap-Datei verwendet werden:

- In `$var` steht der Name der C-Variablen aus der XS-Datei, z.B. `RETVAL`.
- In `$type` steht der Typ dieser Variablen, z.B. `SV *`.
- In `$arg` steht das Argument auf dem Perl-Stack, z.B. `ST(0)`.
- In `$Package` steht der Name des aktuelles Packages, z.B. `Color`.
- In `$func_name` steht der Name der aktuellen XSUB bzw. XS-Methode, z.B. `set_blue`.

Sehen wir uns jetzt einmal eine *typemap*-Datei an, die zu unserem Beispiel passt:

```
TYPEMAP
color *      O_OBJECT

OUTPUT
O_OBJECT
  sv_setref_pv($arg, CLASS, (void*) $var);

INPUT
O_OBJECT
  if (sv_derived_from($arg, \"$Package\") &&
      SvTYPE(SvRV($arg)) == SVt_PVMG)
    $var = ($type) SvIV((SV*) SvRV($arg));
  else {
    warn(\"${Package}::$func_name(): $var\"
         \" is not of type $Package\");
    XSRETURN_EMPTY;
  }
```

Wir bringen dem `xsubpp` also bei, dass ein Zeiger auf `color` wie ein `O_OBJECT` zu behandeln ist. Für `O_OBJECT` werden dann `INPUT`- und `OUTPUT`-Regeln definiert. Die `OUTPUT`-Regel, die beim Konstruktor zum tragen kommt, ruft die Funktion `sv_setref_pv` auf, die den Zeiger in `$var` in einem neuen `SV` ablegt. `CLASS` gibt dabei die Klasse des neuen `SV` an. `$arg`, im Falle des Konstruktors die Variable `RETVAL`, wird zu einer Referenz auf den neuen `SV`. Eine ganze Menge also, was diese eine Funktion macht.

Die `INPUT`-Regel, die bei allen anderen Methoden dafür sorgt, dass in `THIS` wieder unser C-Zeiger steht, überprüft als erstes, ob es sich bei dem Perl-Objekt um eine Instanz der Klasse `Color` oder einer von ihr abgeleiteten Klasse handelt. Ist dies der Fall, wird aus der Objektreferenz der C-Zeiger zurückgewonnen. Anderenfalls wird eine Warnung ausgegeben und die Methode durch `XSRETURN_EMPTY` beendet.

Die *typemap* ist also eindeutig der komplizierteste Teil, wenn man ein objektorientiertes XS-Modul schreiben will. Zum Glück ist es aber auch der Teil, an dem man nur sehr selten etwas ändern muss.

Um herauszufinden, ob unsere XS-Klasse auch richtig funktioniert, kann man das Angenehme mit dem Nützlichen verbinden und gleich einen Test schreiben:

```
use Test::More tests => 3;
use strict;

use_ok('Color');

my $c = Color->new;
is($c->blue, 0);

$c->set_blue(42);
is($c->blue, 42);
__END__
```

Fortsetzung S. 47

```
$ make test
t/Color....ok
All tests successful.
Files=1, Tests=3,  1 wallclock secs ( 0.01
usr  0.00 sys +  0.05 cusr  0.00 csys =
0.06 CPU)
Result: PASS
```

Fortsetzung von S. 46

Ties in XS

Selbstverständlich lassen sich auch Tie-Klassen in XS implementieren. Eine Tie-Klasse ist ja im Grunde eine ganz normale Klasse, die lediglich einige spezielle Methoden implementiert. Wir haben die notwendigen Werkzeuge also bereits im vorigen Abschnitt kennengelernt.

Wir werden also kurzerhand unsere `Color`-Klasse um die Fähigkeit erweitern, sich an einen Hash binden zu lassen. Dazu müssen wir als erstes die `TIEHASH`-Methode implementieren:

```
static color *
color::TIEHASH()
  CODE:
    Newxz(RETVAL, 1, color);
  OUTPUT:
    RETVAL
```

Diese ist prinzipiell identisch mit unserem Konstruktor. Allerdings kümmert sich der `xsubpp` hier nicht automatisch darum, das erste Argument – den Namen der Klasse – in `CLASS` bereitzustellen. Wir müssen ihm mit dem Schlüsselwort `static` auf die Sprünge helfen. Der geneigte Perl-Programmierer fragt sich nun sicher: Wieso denn ausgerechnet `static`? Der Ursprung liegt in der Sprache C++, in der es sich bei `static` deklarierten Methoden um Klassenmethoden (und nicht um Objektmethoden) handelt.

Aber zurück zu XS! Wir müssen noch mindestens eine Tie-Methode für unsere Klasse implementieren. Nehmen wir als einfaches Beispiel die `FETCH`-Methode:

```
int
color::FETCH(key)
  char *key
  CODE:
    if (strEQ(key, "blue"))
      RETVAL = THIS->c_blue;
    else
      XSRETURN_UNDEF;
  OUTPUT:
    RETVAL
```

Die `FETCH`-Methode bekommt, wie in Perl auch, als Argument den Schlüssel des Hash-Zugriffs übergeben. Diesen können wir mit Hilfe von `strEQ` auf den String `"blue"` testen. Bei Gleichheit geben wir einfach den in unserem Objekt gespeicherten Blauwert zurück, sonst `undef`. Noch ein paar Tests, und wir können unsere Tie-Klasse in Aktion sehen:

```
use Test::More tests => 5;
use strict;

use_ok('Color');

my $tied = tie my %chash, 'Color';
is($tied->blue, 0);
is($chash{blue}, 0);

$tied->set_blue(255);
is($tied->blue, 255);
is($chash{blue}, 255);
__END__

$ make test
t/Color....ok
t/Tie......ok
All tests successful.
Files=2, Tests=8,  0 wallclock secs ( 0.00
usr  0.02 sys +  0.10 cusr  0.00 csys =
0.12 CPU)
Result: PASS
```

Natürlich ist diese Tie-Klasse so noch nicht wirklich brauchbar, dazu müssen noch Methoden wie beispielsweise `EXISTS`, `FIRSTKEY` oder `NEXTKEY` implementiert werden. Da dies aber keine grundlegend neuen Erkenntnisse für uns bereithält, möchte ich statt dessen als eispiel auf das CPAN-Modul `Tie::Hash::Indexed` verweisen, welches eine komplette Tie-Klasse in XS implementiert.

Trotzdem noch eine Kleinigkeit. Wer sich noch an den letzten Teil des Tutorials erinnert, hätte bei der `TIEHASH`-Methode eigentlich kurz stutzig werden müssen. Immerhin haben wir diese Methode praktisch direkt vom Konstruktor kopiert. Und wer Quelltext kopiert, macht üblicherweise etwas falsch. Es geht natürlich auch anders, wir können `TIEHASH` als `ALIAS` von `new` implementieren:

```
color *
color::new()
  ALIAS:
    TIEHASH = 1
  CODE:
    Newxz(RETVAL, 1, color);
  OUTPUT:
    RETVAL
```

XS und C++

Das größte Problem bei der Verwendung von XS und C++ ist die Portabilität. Denn nicht überall ist ein C++-Compiler installiert und nicht jeder C++-Compiler ist gleich. Das gilt natürlich auch für C-Compiler, aber typischerweise gibt es bei C++ wesentlich mehr Probleme.

Wer sich davon (und von der Sprache C++) nicht abschrecken lässt, kann im folgenden sehen, wie einfach sich das Beispiel der `Color`-Klasse in C++ implementieren lässt. Praktischerweise erledigt der `xsubpp` in diesem Fall fast die gesamte Arbeit für uns:

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

#include "ppport.h"

class color
{
  private:
    int c_blue;

  public:
    color()
      : c_blue(0)
    { }

    ~color()
    { }

    int blue() const
    { return c_blue; }

    void set_blue(int val)
    { c_blue = val; }
};

MODULE = Color  PACKAGE = Color

PROTOTYPES: DISABLE

color *
color::new()

void
color::DESTROY()

int
color::blue()

void
color::set_blue(val)
     int val
```

Ich werde an dieser Stelle nicht den Versuch unternehmen, C++ und seine Konzepte erschöpfend zu erklären, dazu möchte ich auf die frei verfügbaren Bücher von Bruce Eckel verweisen (*http://mindview.net/Books/TICPP/ThinkingInCPP2e.html*).

Ich werde lediglich kurz die C++-Klasse `color` beschreiben. `c_blue` ist ein Attribut dieser Klasse. Es ist nur für die Klasse selbst sichtbar, da es in einem `private`-Abschnitt steht. Im `public`-Abschnitt finden wir den Konstruktor `color()`, den Destruktor `~color()`, sowie die Methoden `blue()` und `set_blue()`. Mit `: c_blue(0)` wird im Konstruktor `b_blue` auf `0` initialisiert. Das `const` hinter dem Kopf der `blue()`-Methode besagt, dass ein Aufruf dieser Methode das Objekt nicht verändert.

Werfen wir nun einen Blick auf den XS-Teil. Wir sehen, dass dort lediglich das Interface der Klasse `color` beschrieben wird. Die gesamte Funktionalität ist ja bereits in der C++-Klasse implementiert. Der `xsubpp` scheint also zu wissen, dass er in der XSUB `new` den Konstruktor bzw. in `DESTROY` den Destruktor der C++-Klasse aufrufen muss. Ganz so einfach ist es allerdings nicht, wir müssen ihm zumindest sagen, dass wir C++-Code generieren wollen. Dies geht mit Hilfe der `-C++`-Option, die man im *Makefile.PL* setzen kann:

```
use ExtUtils::MakeMaker;
WriteMakefile(
  NAME          => 'Color',
  VERSION_FROM => 'lib/Color.pm',
  CC            => 'g++',
  LD            => '$(CC)',
  XSOPT         => '-C++',
);
```

Desweiteren müssen wir im *Makefile.PL* den C++-Compiler und -Linker angeben, in unserem Fall `g++`. Die *typemap*-Datei und unsere Tests sind identisch mit denen der C-Implementierung der Klasse `Color`, die wir vorher kennen gelernt haben. Jetzt können wir das Modul kompilieren und testen:

```
$ perl Makefile.PL
$ make
$ make test
t/Color....ok
All tests successful.
Files=1, Tests=3,  0 wallclock secs ( 0.01
usr  0.00 sys +  0.02 cusr  0.00 csys =
0.03 CPU)
Result: PASS
```

Ich möchte nun noch kurz auf ein spezielles Problem (und dessen Lösung) bei der Verwendung von C++ und XS eingehen: Namespaces. Im Gegensatz zu C gibt es bei C++ die Möglichkeit, verschiedene Programmteile in unterschiedlichen Namensräumen zu implementieren. Dazu dient das Schlüsselwort `namespace`, wie im folgenden Beispiel zu sehen:

```
namespace image
{
  class color
  {
    private:
      int c_blue;

    public:
      color()
        : c_blue(0)
      { }

      ~color()
      { }

      int blue() const
      { return c_blue; }

      void set_blue(int val)
      { c_blue = val; }
  };
}
```

Da in C++, wie in Perl auch, Namensräume durch `::` abgetrennt werden, ist es intuitiv, sowohl in den XSUBs als auch in der *typemap* einfach `color` durch `image::color` zu ersetzen. Konstruktor und Destruktor sehen dann also so aus:

```
image::color *
image::color::new()

void
image::color::DESTROY()
```

Versuchen wir nun, das Modul zu kompilieren:

```
$ make
Color.c: In function 'void
XS_Color_new(CV*)':
Color.c:58: error: 'image__color' was not
declared in this scope
Color.c:58: error: 'RETVAL' was not declared
in this scope
Color.c: In function 'void
XS_Color_DESTROY(CV*)':
Color.c:85: error: 'image__color' was not
declared in this scope
Color.c:85: error: expected
primary-expression before ')' token
Color.c: In function 'void
XS_Color_blue(CV*)':
Color.c:116: error: 'image__color' was not
declared in this scope
Color.c:116: error: expected
primary-expression before ')' token
Color.c: In function 'void
XS_Color_set_blue(CV*)':
Color.c:147: error: 'image__color' was not
declared in this scope
Color.c:147: error: expected
primary-expression before ')' token
make[1]: *** [Color.o] Error 1
```

Das war leider nicht so erfolgreich. Irgendwie hat der `xsubpp` aus unserem `image::color` ein `image__color` gemacht. Leider ist das sein Standardverhalten, auch im C++-Modus. Die Lösung ist die `-hiertype`-Option: Sie aktiviert die Unterstützung für hierarchische Typen in C++, also Namensräume und verschachtelte Klassen. Passen wir also unser *Makefile.PL* an und versuchen noch einmal, das Modul zu kompilieren:

```
use ExtUtils::MakeMaker;
WriteMakefile(
  NAME         => 'Color',
  VERSION_FROM => 'lib/Color.pm',
  CC           => 'g++',
  LD           => '$(CC)',
  XSOPT        => '-C++ -hiertype',
);
__END__

$ make
$ make test
t/Color....ok
All tests successful.
Files=1, Tests=3,  0 wallclock secs ( 0.01
usr  0.00 sys +  0.05 cusr  0.00 csys =
0.06 CPU)
Result: PASS
```

Don't call us, we'll call you

Zum Abschluss möchte ich noch ein besonders interessantes Thema behandeln: Das Aufrufen von Perl-Subroutinen aus XS-Code. Im folgenden Beispiel wollen wir eine Subroutine per Referenz aufrufen, ihr Argumente übergeben und ihren Rückgabewert verarbeiten.

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

#include "ppport.h"

void do_call(pTHX_ SV *sub)
{
  int count;
  SV *rv;
  dSP;

  ENTER;
  SAVETMPS;

  PUSHMARK(SP);
  mXPUSHs(newSVpvs("Zaphod"));
  mXPUSHs(newSViv(42));
  PUTBACK;

  count = call_sv(sub, G_SCALAR);

  SPAGAIN;
```

Fortsetzung S. 50

```
  if (count != 1)
    Perl_croak(aTHX_ "failed (%d)", count);

  rv = POPs;

  sv_dump(rv);

  PUTBACK;

  FREETMPS;
  LEAVE;
}

MODULE = Call    PACKAGE = Call

void
call(rout)
  SV *rout
  PPCODE:
    do_call(aTHX_ rout);
```

Fortsetzung von S. 49

Die XSUB `call` ist schnell erklärt: Sie bekommt als Argument eine Referenz auf die Perl-Subroutine, die wir aufrufen möchten. Diese Referenz gibt die XSUB direkt an unsere C-Funktion `do_call` weiter. Bei `aTHX_` handelt es sich um ein Makro, das im Falle eines Perl-Binaries mit ithreads an dieser Stelle das Interpreterargument und ein Komma einfügt. Das Gegenstück ist `pTHX_` im Kopf von `do_call`. Auf diese Art kann das Interpreterargument von Funktion zu Funktion übergeben werden.

Nun aber zu `do_call`. Neben den beiden Variablen `count` und `rv` wird am Anfang mit `dSP` eine lokale Kopie des Stack Pointers angelegt. Der Rest der Funktion steht zwischen

```
ENTER;
SAVETMPS;
```

und

```
FREETMPS;
LEAVE;
```

`ENTER` und `LEAVE` bilden zusammen einen neuen Gültigkeitsbereich ("scope"), vergleichbar mit einem `{ }`-Block in Perl oder C. Innerhalb diese Bereichs angelegt Variablen verlieren beim Verlassen des Bereichs mit `LEAVE` ihre Gültigkeit. Perl besitzt zur Verwaltung der Gültigkeitsbereiche zwei Stacks: den Save Stack (`PL_savestack`), auf dem alle Werte abgelegt werden, die nach dem Verlassen des Gültigkeitsbereichs wiederhergestellt werden sollen, und den Scope Stack (`PL_scopestack`), der sich merkt, an welcher Stelle des Save Stacks die einzelnen Gültigkeitsbereiche enden. `ENTER` speichert also auf dem Scope Stack die aktuellen Stack Pointer des Save Stacks; `LEAVE` läuft dann auf dem Save Stack bis genau zu dieser Position zurück und stellt die in der Zwischenzeit überschriebenen Werte wieder her.

`SAVETMPS` und `FREETMPS` kümmern sich um temporäre Perl-Variablen, also Variablen, die sterblich gemacht wurden. An dieser Stelle kommt ein weiterer Stack ins Spiel, der Temporaries Stack (`PL_tmps_stack`). Dieser merkt sich bei jedem Aufruf von `sv_2mortal` den `SV`. `SAVETMPS` rettet den aktuellen "Boden" des Temporaries Stacks (`PL_tmps_floor`) auf den Save Stack und setzt ihn anschließend auf den aktuellen Stack Pointer des Temporaries Stacks. `FREETMPS` läuft dann auf dem Temporaries Stack bis auf den neuen Boden zurück und dekrementiert dabei den Reference Count von jedem `SV`. Die sterblichen `SV`s mit einem Reference Count von 1 werden dabei freigegeben.

Es ist gar nicht so einfach, bei den ganzen Stacks immer den Überblick zu behalten. In den meisten Fällen genügt es aber, sich zu merken, dass `ENTER/SAVETMPS` und `FREETMPS/LEAVE` einen Rahmen auf dem Callstack von Perl umschließen.

Die beiden Makros `PUSHMARK` und `PUTBACK` bilden den Rahmen für den Aufbau der Argumentliste unseres Aufrufs. Mit `PUSHMARK(SP)` wird die aktuelle Position des Stack Pointers gespeichert. `PUTBACK` kopiert, nachdem wir unsere beiden Argumente `"Zaphod"` und `42` auf den Perl-Stack gelegt haben, die lokale Kopie des Stack Pointers zurück in den globalen Stack Pointer, so dass er von `call_sv` gesehen werden kann.

Die Funktion `call_sv` übernimmt die eigentliche Arbeit und ruft die Perl-Subroutine auf. `call_sv` bekommt eine Referenz auf die Subroutine sowie den Kontext des Aufrufs übergeben und gibt die Anzahl der Rückgabewerte der aufgerufenen Perl-Subroutine zurück. `G_SCALAR` bewirkt, dass der Aufruf im Skalarkontext erfolgt, mit `G_VOID` und `G_ARRAY` kann man Routinen im Void- bzw. Listenkontext aufrufen. Der Kontext kann darüberhinaus mit verschiedenen Flags verodert werden, z.B. `G_EVAL` oder `G_METHOD`. Bei `G_EVAL` erfolgt der Aufruf so, als würde er innerhalb eines `eval { }`-Blocks stehen. Wird `G_METHOD` angegeben, erfolgt der Aufruf als Methodenaufruf; dabei muss als erstes Argument auf dem Stack entweder ein Objekt oder – im Falle eines Klassenmethodenaufrufs – der Name einer Klasse liegen.

Anschließend wird mit `SPAGAIN` unsere lokale Kopie des Stack Pointers aktualisiert. Nach einer kurzen Überprüfung der Anzahl der Rückgabewerte wird dann mit `POPs` der erste Wert vom Perl-Stack – der Rückgabewert der Perl-Subroutine – entfernt und in `rv` gespeichert. Die Funktion `sv_dump`, die wir bereits aus dem zweiten Teil des Tutorials kennen, gibt dann Informationen über diesen Rückgabewert aus. Da wir den Perl-Stack mit `POPs` erneut manipuliert haben, müssen wir nochmals den globalen Stack Pointer mit `PUTBACK` aktualisieren.

Das war nun wirklich nichts für schwache Nerven. Schauen wir einmal nach, ob unser XS-Modul funktioniert:

```
$ perl -Mblib -MCall \
  -le'call(sub{print STDERR for @_; "foo"})'
Zaphod
42
SV = PV(0x815a0d0) at 0x816d240
  REFCNT = 1
  FLAGS = (TEMP,POK,pPOK)
  PV = 0x8164c18 "foo"\0
  CUR = 3
  LEN = 4
```

Der Aufruf von `call` legt die beiden Argumente `"Zaphod"` und `42` auf den Stack und ruft danach unsere anonyme Subroutine auf, die die beiden Argumente ausgibt und anschließend `"foo"` zurückgibt. Der `SV` mit `"foo"` wird dann von `do_call` an `sv_dump` übergeben und dort ausgegeben.

Neben `call_sv` gibt es noch andere Funktionen, um Perl-Routinen aus XS aufzurufen. Diese werden in der Manpage `perlcall` ausführlich behandelt.

Und jetzt?

Jetzt möchte ich mich bei allen bedanken, die bis zum Ende tapfer durchgehalten haben. Ich hoffe, dass ich bei den XS-Neulingen etwas Interesse wecken konnte, und dass auch diejenigen, die bereits Erfahrung mit XS sammeln konnten, beim Lesen vielleicht noch das eine oder andere dazugelernt haben. Früher oder später sicher aufkommende Fragen stellt man übrigens am besten auf der *perl-xs* Mailingliste (*http://lists.cpan.org/showlist.cgi?name=perl-xs*). Und jetzt viel Spaß beim Hacken!

Marcus Holland-Moritz

Perl 6 Tutorial - Teil 7 : Text, Regeln und Grammatik

Willkommen zum siebenten Teil dieses poetischen Perl 6-Tutorials, das vor allem den Umgang mit Text, also Strings, zum Inhalt hat. Das klingt stark nach Regex, beinhaltet aber noch einiges mehr, wie *quoting*, Interpolation und Formate. Beginnen wir jedoch mit den einfachsten Grundlagen.

Es werde String

Wie im zweiten Kapitel beschrieben, ist für Perl 6, ebenso wie in Perl 5, String nur ein Kontext, in den beliebige Daten umgewandelt werden können. Dieser Kontext wird aber nicht nur mit Operatoren erzeugt die mit ~ beginnen, sondern auch wenn ein Text ganz einfach in Anführungszeichen in den Quellcode eingefügt wird.

Einfaches Quoting

Die schlichte Variante sind dabei (wie bekannt) die einfachen Anführungszeichen, innerhalb derer jedes Zeichen wörtlich (*literal*) verstanden wird und besondere Bedeutungen aufgehoben sind. Die einzigen beiden Ausnahmen sind `\'` und `\\`. Ersteres ist notwendig um ein einfaches Anführungszeichen innerhalb des Strings zu erhalten, zweiteres für einen umgekehrten Schrägstrich (*Backslash*), der normalerweise die besondere Bedeutung der Steuerzeichens (Metazeichens) aufhebt, was auch *escapen* genannt wird. Etwas in Anführungszeichen zu setzten wird auch *quoting* (Zitieren) bezeichnet, weswegen ein `q//` die generalisierte Form von `''` ist.

```
say  '\'';   # sagt: '
say q/\\/;   # sagt: \
# dito, beliebige Klammern sind möglich
say q["\\"];
```

Das Schöne an Perl 6 ist allerdings nicht nur dass es sich hier bequem an Perl 5 angleicht, sondern dass das Bekannte (oft) zur allgemeingültigen Grundregel wird. `''` hat überall die gleiche Bedeutung, auch innerhalb eines regulären Ausdrucks. Und `q//` ist die Basis sämtlicher *quoting*-Operatoren die lediglich `q//` mit verschiedenen Optionen aufrufen. Aber um wirklich ehrlich zu sein: auch das ist nur die halbe Wahrheit. Der echte Urvater aller *quoting*-Operatoren ist das `Q`, welches gar nichts interpoliert und `q//` ist ein *Alias* auf `Q :q //`. Der Doppelpunkt vor dem "q" markiert einen benannten Parameter. Die ganze Erklärung dazu steht in der vierten und fünften Folge dieses Tutorials.

Komplexeres Quoting

Auch wenn die Einfachen oft ausreichend sind und minimal Rechenzeit sparen, so sieht man öfter die doppelten Anführungszeichen. Ihre Bedeutung blieb auch im Wesentlichen unberührt. Innerhalb von `""` werden Variablen evaluiert (durch ihren Inhalt ersetzt) und *Escapesequenzen* wie `\n` (Zeilenende) oder `\t` (Tab) gegen entsprechende Sonder- oder Steuerzeichen ersetzt. Hinzu kam die an Ruby erinnernde Möglichkeit, neben Subroutinen (hier mit "&" schreiben) nun auch Blöcke (*Closures*) auszuführen.

```
say "Und die Zusatzzahl für $heute lautet:
  \t { int rand(47) }.";
```

Auch hier hat Larry konsequent vereinheitlicht. Blöcke werden nun mal (wenn sie kein Parameter sind) sofort ausgeführt. Und auch wer jetzt innerhalb einer Regex etwas ausführen möchte sollte sich dieses Mittels bedienen anstatt sich mit der Option "e" oder `(?{ ... })` einen Kompilererror einzufangen. Anstatt "ee" lässt sich ein `eval` in den Block einfügen (Ausführung des Ausführungsergebnisses).

Damit ist eindeutig klar: `""` oder alternativ auch `qq//` ist ein *Alias* auf `Q :s :a :h :f :c :b //`, oder? Verständlicher wird es, wenn die "langen Formen" der "Adverbien" verwendet werden (`Q :scalar :array :hash :function :closure :backslash //`). `:scalar` besagt das Skalare evaluiert werde, `:array` - Arrays ... und `:backslash` bezieht sich auf die Steuerzeichen. Das impliziert auch `:q` aka `:single`, dass hierfür nicht extra angegeben werden musste. `:qq` lautet ausgeschrieben entsprechend `:double`. Weitere *quote*-"Adverbien" sind `:x` aka `:exec`, was dem bekannten Perl 5-`qx` entspricht, nur das `qx` in Perl 6 logischerweise nicht interpoliert, da es gleich `q :x //` ist. `:w` aka `:words` ist ebenfalls altbekannt, wird aber in Perl 6 idiomatisch "< >" geschrieben (siehe dritte Folge). `:ww` aka `:quotewords` entspricht dem bereits bekannten "<< >>".

```
my $ich = 'Prinz';
my @n = 1 .. 5;
say q:a/$ich sage: in \@n ist @n./
# sagt: $ich sage: in @n ist 1 2 3 4 5.
```

Heredocs

Heredocs sind nun auch nichts besonderes (eigenständiges) mehr, nur noch normale *quotings*, deren Ende mit einer definierten Zeichenkette markiert ist. Diese kann nun beliebig eingerückt sein. *Heredocs* werden damit weit mächtiger, ohne erhöhten Lernaufwand.

```
my $letter = qq:to/END/
    Dear $recipient:
    Thanks!
    Sincerely,
    $me
    END
```

Die Langform des `:to`-Adverbs ist `:heredoc`.

Die neuen Regex

Es gibt von diesen Adverbien noch einige mehr, diese haben jedoch nur eine lange Form und nicht mehr direkt etwas mit Strings zu tun, da sie zu Code oder Regex evaluiert werden. Denn Reguläre Ausdrücke werden in Perl 6 nicht mehr als Strings angesehen, sondern als eigenständige Sprache innerhalb der Sprache (wie in Rebol). Diese Sprache ist kontextfrei, was rekursive Ausdrücke erlaubt, wie erst seit Perl 5.10 möglich. Und diese Sprache unterscheidet sich teilweise so stark von den Regex in Perl 5, so dass eine Zeit lang für sie ein neuer Begriff geprägt wurde (die Perl "rules"). Somit wandelte sich der Ausdruck "Perl rules" von einer Angeberei zur Erwähnung einer normalen Tatsache. Mittlerweile setzte sich aber auch für die neuen Regeln der Begriff Regex wieder durch.

Es war nötig sie von Grund auf neu zu gestalten, da der Regex-Syntax mit der Zeit zu unübersichtlich und widersprüchlich wurde. Dies war auch nur teilweise das Verschulden von Larry oder Ilya, da viele Unsauberheiten der UNIX-Kultur entstammen. Damals schien es noch eine gute Idee sich an die Standards zu halten und Perl leicht erlernbar zu machen. Mittlerweile hat Perl selbst den Standard gesetzt (PCRE) und es wird Zeit hier einiges gerade zu rücken.

Eine Regex kann mit drei Schreibweisen in einem Skalar gespeichert werden:

```
my $r = rx[abc];
my $r = q:regex[abc];
my $r = regex{abc};
```

Die dritte Variante ist aber nicht ganz das Selbe wie die ersten Beiden und kann daher nur ausschließlich mit geschweiften Klammern geschrieben werden. Mehr dazu wenn es um "Grammatiken" geht. Oft werden Regex nicht gespeichert sondern direkt angewendet, was immer noch mit `m/.../` oder `s/.../.../` oder den Sonderformen `mm/.../` und `ss/.../.../` getan wird. Auch `tr/.../.../` gibt es weiterhin. Diese Operationen werden wie in Perl 5.10 mit dem *Smartmatch* (~~) an einen String gebunden. Das Resultat ist allerdings hier ein *Matchobjekt*, das im boolschen Kontext zu `Bool::True` oder `Bool::False` evaluiert, damit `if`, `when`, `while` und `until`, die diesen Kontext erzwingen, weiterhin funktionieren. Das zuletzt im aktuellen Block erzeugte *Matchobjekt* liefert auch die Variable `$/`, die alle Detailinformationen enthält. Andere, bisherige Sondervariablen wurden abgeschafft. Allerdings sind `$0` bis `$9` *Aliase* auf `$/[0]` bis `$/[9]`. Als Trenner können weiterhin beinah beliebige Zeichen verwendet werden. Aber Optionen werden nun (wie beim *quoting*) als Adverbien vor die Trenner und hinter den Namen der Operation gesetzt.

Regex Optionen

```
my $text =
 "Ich schreibe gerne udn schnell udn viel.";
$text ~~ s:g/udn/und/;
```

Ansonst sollte die Option ":g" bekannt sein. Ohne sie wird nur einmal ersetzt, mit ihr, jedes Aufkommen. Auch die ":i"-Option, welche die Gross/Kleinschreibung ignorieren lässt ist unverändert geblieben. Ähnlich wie beim *quoting* gibt es auch hier für die ordentlichen Schönschreiber mit `:global` und `:ignorecase` eine lange Version dieser Optionsnamen. Neu ist das Adverb `:ii` aka `:samecase` welches ebenfalls die Groß/Kleinschreibung des Suchausdrucks nicht genau nimmt, ihn aber dann an die Schreibweise des Ersetzten anpasst.

```
my $text =
 "Ale Vögel sind schon da, alle ...";
$text ~~ s:g:ii/ale/alle/;
# $text eq "Alle Vögel sind schon da,
# alle ..."
```

In ähnlicher Weise gibt es nun auch `:a` aka `:ignoreaccent`, sowie `:aa` aka `:sameaccent` das Buchstaben als gleich erkennen und ersetzen können, egal ob sie einen Punkt, einen Strich, ein Dach oder nichts über sich haben. Dazu ignoriert der Interpreter alle "*mark characters*" zusammengesetzter *Unicode*-Zeichen.

Es kamen etliche weitere Adverbien hinzu aber es wurden auch viele abgeschafft. Neben den bereits erwähnten `:e` und `:ee` betrifft das auch `:m` und `:s`. `^` und `$` bedeuten nun immer Anfang und Ende des Strings. `^^` und `$$` stehen für Anfang und Ende einer Zeile: Perl verwendet natürlich das eigene `\n` um Zeilen zu unterscheiden, was immer erfolgreich sein sollte, egal auf welchem Betriebssystem das Programm gerade läuft oder woher die Daten kommen. Ein `.` bedeutet jetzt wirklich "ein beliebiges Zeichen". Die alte Bedeutung (ein beliebiges Zeichen außer \n) nennt sich jetzt konsequenterweise `\N`, da großgeschriebene Symbole für Zeichenklassen immer das Komplementär zu ihrem kleingeschriebenen Pendant sind.

Auch die Option `:x` ist weg, da Leerzeichen und Kommentare nun standardmäßig eingefügt werden können, soweit keine besondere Regel etwas anderes vorschreibt. Der Syntax von Kommentaren ist innerhalb der Regex der Selbe wie außerhalb. Eine Raute kommentiert bis zum Ende der Zeile, folgt der Raute beliebige Klammersymbole, gilt bis zum Schließen dieser Klammern der Inhalt der Regex als Kommentar.

Eine der Regeln die Leerzeichen besondere Bedeutung geben können ist `:s` aka `:sigspace`. Wird sie gewählt, zählt jedes Leerzeichen als "<.ws>", was grob mit `\s+` (mehrere Leerzeichen verschiedenster Art) übersetzt werden kann. Diese I<sigspace> findet Larry so bedeutend, dass `m:s/.../` gleichbedeutend mit `mm//` ist. Ein `ss/.../.../` entspricht allerdings einem `s:ss/.../.../`. `:ss` aka `:samespace` unterscheidet sich von `:sigspace` nur insoweit, dass auch zwischen dem zweiten und dritten Trenner jedes Leerzeichen als "<.ws>" gilt.

```
my $lied = "Der Mond\nist\taufgegangen ...";
ss/Mond ist aufgegangen/Plumssack geht im/;
# "Der Plumssack\ngeht\tim".
```

Sehr praktisch finde ich auch die Optionen `:c` aka `:continue` und `:p` aka `:pos`. Ersteres sucht ab einer Position, zweiteres nur an einer Position. Wird keine Zahl für die Position angegeben, so wird diese `$/` entnommen.

```
my $text =
  "Vom Eise befreit sind Strom und Bäche."
if $text ~~ m:c(3)/Eis/ {
    # wird ausgeführt
    ...
if $text ~~ m:p(3)/Eis/ {
    # wird nicht ausgeführt
    ...
```

Was genau gezählt werden soll, kann mit den Adverbien `:bytes`, `:codes` (*Unicode codepoints*), `:graphs` und `:chars` angegeben werden.

Soll eine Ersetzung nur dreimal ausgeführt werden so hilft `:3x` aka `:x(3)` und falls nur der fünfte Fund einer Suche interessiert, so steht dazu `:5th` aka `:nth(5)` bereit. `:1st`, `:2nd` und `:3rd` sind Sonderformen die eine natürliche Ausdrucksweise erlauben sollen und auch *Junctions* wie `nth(3|5|7)` sind möglich. Die Bedeutung runder Klammern hat sich nicht verändert. Sie umschließen nach wie vor Teilausdrücke, deren Fund bitte gespeichert werden soll. Teilausdrücke deren Fund nicht interessiert werden jetzt in eckige Klammern gesetzt anstatt in `(?: ... )`.

```
s:3rd/(\d+)/@data[$0]/;
# ist das selbe wie:
m/(\d+)/ && m:c/(\d+)/
   && s:c/(\d+)/@data[$0]/;
```

Die Regeln für die Nummerierung von `$0` bis `$9` haben sich auch geändert, da bisher kleine Änderungen in der Regex manchmal zu viele Nachbesserungen in dem Code forderte, der diese Variablen auswertet, da eine ungünstig gesetzte Klammer die Position aller anderen verändert. Sind Klammern hintereinander wie in `()()`, werden sie immer noch genauso gezählt. In einem Fall wie `(... (...) )` wäre der Fund der inneren Klammer in `$0[0]`.

Was mich ebenfalls begeisterte sind die die neuen Optionen `:ov` aka `:overlap` und `:ex` aka `:exhaustive` die am besten an einem Beispiel verstanden werden.

```
$str = "abracadabra";

if $str ~~ m:overlap/ a (.*) a / {
    @substrings = @@();
    # bracadabr cadabr dabr br
}
if $str ~~ m:exhaustive/ a (.*?) a / {
    say "@()";
    # br brac bracad bracadabr c cad
    # cadabr d dabr br
}
```

Da `$0` bis `$9` oft nicht reichen, gibt es `@()`, der alle Funde enthält. Um das Suchverhalten zu steuern, kann auch die neue Option `:ratchet` hilfreich sein, die jegliches *Backtracking* verhindert. Das bedeutet, dass Perl den zu durchsuchenden String nur ein einziges mal von links nach rechts überprüft und dabei niemals rückwärts geht, selbst wenn in der Regex Alternativen formuliert sind die erfolgreich sein könnten wenn die Suche noch einmal begonnen würde. Das `Backtracking` kann aber auch für einzelne Teilausdrücke abgestellt werden, indem ihm ein `:` nachgestellt wird (ehemals `(?> ...)`).

Quanto costa?

Die sogenannten *quantifier* haben sich kaum verändert. `?` steht immer noch dafür den Teilausdruck nicht oder nur einmal zu finden, `+` für einmal oder beliebig oft und `*` für keinmal oder beliebig oft. Auch die nicht gierigen *quantifier* (`??`, `*?`, `+?`) haben sich in Syntax und Semantik (Bedeutung) nicht verändert. Nur exakte *quantifier* werden anders geschrieben, da geschweifte klammern wie beschrieben immer Blöcke darstellen. Die neue Schreibweise, der aufmerksame Leser ahnt es bereits, entstammt dem allgemeinem Perl-Syntax. Wenn ein Multiplikationszeichen Wiederholungen bezeichnet, dann kann eine vierfache Wiederholung ja nur `**4` lauten. Und darf sich ein Teilausdruck nur ein bis dreimal wiederholen so wird dies `()**1..3` geschrieben.

Nichts ist illegal

Mit Perl 6 gibt sich Larry auch große Mühe Zweideutigkeiten zu vermeiden. Ein simples:

```
$text ~~ m/ /;
```

wirft jetzt einen wunderschönen Error in die Kommandozeile. Um die letztbenutzte Regex wiederzuverwenden schreibe man:

```
$text ~~ m/ <prior> /;
```

Oder falls tatsächlich nichts gesucht wird, gibt es die Schreibweisen `''` und `<?>`. Wird eine Alternative beim Programmieren vergessen wird dies nun sofort sichtbar. Ja auch das `|` hat sich in seiner Bedeutung nicht verändert.

```
$text ~~ m/ [<alpha>|_]+ /;
```

Diese Regex sucht Wörter die aus Buchstaben und Unterstrichen bestehen. Was vorher Dach war (Komplementärmenge an Zeichen) ist nun Minus. Oder anschaulich: Keine hexadezimale Zahlen, die man früher mit der Gruppe `[^0-9a-f]` erfasst hat, werden jetzt mit `<-[0-9a-f]>` gesucht. In diesem Falle würde es `<-hexdigit>` allerdings auch tun.

Wendehälse

Vorschau (lookahead) und Rückschau (lookbehind) haben ihre Schreibweise geändert und sind nun lesbarer. Angelehnt an logische Operatoren wie den Ternären Operator bezeichnet ein `?` immer eine positive und `!` eine negative Option. Folglich wurde `(?= ...)` zu `<?before ...>` und `(?! ...)` zu `<!before ...>`. Entsprechend änderte sich `(?<= ...)` zu `<?after ...>` und `(?<! ...)` zu `<!after ...>`. Mit solchen Teilausdrücken verlangt man welche Zeichen vor oder hinter dem gesuchten Teilstring gewünscht oder unerwünscht sind. Eine detaillierte Erklärung dazu hat das perlretut in den *perldocs*.

Wer spätestens jetzt bei all den Neuerungen und Änderungen die Lust verliert, kann mit `:P5` aka `:Perl5` die alten Regeln wieder einsetzen. Jedoch müssen Optionen auf jeden Fall vor die Regex oder in sie hinein in eine `(?m ...)` Gruppe.

Die große Perspektive

Auch wenn mit dem neuen Syntax viele Details klarer werden, können Regex immer noch schnell unleserlich werden, wenn komplexe Zusammenhänge eingelesen und aufgespalten werden sollen. (Wir bauen mal schnell einen HTML-Parser.) Hierfür bräuchte es Mittel wie lex und yacc, die ein stufenweises Aufspalten erlauben. Die dazu verwendeten Regeln wären verständlicher und einfacher wiederverwendbar. Genau diese Überlegungen stehen hinter den Perl 6-Grammatiken. Eine `grammar` ist in Wirklichkeit eine Klasse, deren Methoden Regex sind. Deshalb behandelt dieses Tutorial auch die Regex nach der OOP, obwohl Strings ein elementares Thema für Perl-Programmierer sind. Für `grammar` wurden 3 spezielle Typen von Routinen geschaffen. Dies wäre zum einen `regex`. Damit wird klar warum die Schreibweise `regex [name] {}` geschweifte Klammern verlangt. Eine `sub` wird ebenfalls nur mit geschweiften Klammern geschrieben. Um das Parsen einfacherer Regeln effektiver und das Kombinieren von Teilregeln optisch zu vereinfachen gibt es noch den Routinentyp `rule`. Der entspricht `regex :ratchet :sigspace {...}`. Der dritte Typ (`token`) wurde geschaffen um Grundbausteine zu definieren und entspricht `regex :ratchet { ... }`. Ein Grammatik um eine Fließkommazahl zu zerlegen sähe ungefähr so aus:

```
grammar Float {
    rule digits { \d+    }
    rule sign   { \+ | - }
    rule exp :i { e <sign>? <digits> }
    rule man    { \. <digits> | <digits>
                  [ \. <digits>?  ]?  }
    rule top    { <sign>? <man> <exp>? }
}
```

Einzelne `rules` definieren Teilausdrücke die spätere `rules` verwenden, indem sie den Namen der `rule` in spitze Klammern setzen. Spitze Klammern und Punkt markieren Teilausdrücke deren Fund nicht gespeichert wird. (wie das erwähnte `<.ws>`) Diese `rules` sind tatsächlich Routinen denn ich kann sie wie welche aufrufen.

```
my $result = Float.top("-6.02e+23");
```

In `$/{'man'}` oder `$result{'man'}` steht nun z.B. "6.02", in `$/{'exp'}{'sign'}` ein Pluszeichen, da dies das Vorzeichen des Exponenten ist. Aus Folge vier wissen wir das Hashschlüssel auch einfacher geschrieben werden können, z.B. "$/<sign>". Nun kann man erkennen wieviel Sinn es macht, benannte Teilausdrücke in spitze Klammern zu setzen, da die Hashschlüssel unter denen ihr Fund gespeichert ist, gleich aussehen. Ähnliches war innerhalb einer Regex ab Perl 5.10 nur mit `(?<name> ...)` erreichbar. Grammatiken sind allerdings wie (andere Klassen auch) ableitbar und können so einfach auf bereits bekannte Art kombiniert und erweitert werden. Dies ist nicht nur praktisch sondern erlaubt erst viele besondere Fähigkeiten von Perl 6, dass damit immer mehr zu einer Meta-Programmiersprache mutiert. Metaprogrammierung wird jedoch Thema der nächsten und letzten Folge sein.

Herbert Breunung

Win32 Tipps & Tricks

In dieser Ausgabe konzentriere ich mich nicht auf einen Tipp, sondern zeige ein paar Tricks für Windows-Nutzer in FAQ-Manier.

Ist die Datei "versteckt"?

Dazu am besten `Win32::File` verwenden. Hier ein Beispiel für die Verwendung des Moduls. Soll eines der anderen Attribute überprüft werden, muss die Konstante im Import und beim "&" natürlich ausgetauscht werden.

```
#!/usr/bin/perl -l

use strict;
use warnings;
use Win32::File qw(GetAttributes HIDDEN);

my $file = 'hidden.txt';

my $attribs;
GetAttributes( $file, $attribs );
print $file, " -> ", $attribs & HIDDEN;
```

"gekürzte" Pfade in lange Pfade wandeln

Wer kennt sie nicht, die alte Schreibweise (8.3 Dateiname / SFN) bei langen Namen: `C:\DOKUMENT~1\PRIVATER~1\Datei.txt`. Damit kann man unter Umständen nichts anfangen aber in manchen Fällen werden sie dennoch von Modulen zurückgeliefert. Hier zeige ich, wie man diese Pfade in die langen Pfade umwandelt:

```
#!/usr/bin/perl -l

use strict;
use warnings;
use Win32;

# hole den 8.3 Namen des Verzeichnisses
my $dir = Win32::GetShortPathName(
            Win32::GetCwd() );
print $dir;

# wandle in den langen Pfad um
my $long = Win32::GetLongPathName( $dir );
print $long;
```

Laufwerksbuchstaben auslesen

Zum Beispiel für einen Verzeichnisbaum á la Windows Explorer braucht man alle Laufwerksbuchstaben des Rechners. Diese bekommt man auf diese Weise:

```
#!/usr/bin/perl

use strict;
use warnings;

use Win32::API;

my $function = Win32::API->new(
    'kernel32', 'GetLogicalDriveStringsA',
    'NP', 'N' );

my $drivestr = ' 'x1024;
my $ret = $function->Call(
                        1024, $drivestr );

print "'$_'\n" for split "\0", substr(
  $drivestr, 0, $ret );
```

Zwischenablage auslesen

Wer sich für den Inhalt der Zwischenablage interessiert, der kommt so an den Text:

```
#!/usr/bin/perl

use strict;
use warnings;
use Win32::Clipboard;

print Win32::Clipboard::Get();
```

Renée Bäcker

CPAN News X

Module::Load::Conditional

`Module::Load::Conditional` eignet sich ganz gut für Programme, bei denen man keinen Einfluss auf die Umgebung hat und man vor dem Laden von Modulen (z.B. Plugins oder ähnliches) erstmal überprüfen möchte, ob das Modul existiert und wenn ja, ob es auch die richtige Mindestversion hat. Zusätzlich lässt sich mit diesem Modul noch herausfinden, welche anderen Module ein bestimmtes Modul benötigen.

Mit `check_install` kann man sich Informationen darüber holen, in welcher Version ein Modul vorliegt und wo es gespeichert ist. Mit `can_load` kann man ein Modul "sicher" laden, da bei einem nicht-installierten Modul das Programm nicht abbricht, sondern einen "unwahren" Rückgabewert hat.

```
#!/usr/bin/perl

use strict;
use warnings;
use Data::Dumper;
use Module::Load::Conditional
      qw/can_load check_install/;

my $check = {
    'DBIx::Class::Log4perl' => 0.01,
};

my $retval = can_load( modules => $check );
print $retval ?
  "Alle Module geladen\n" :
  "Modul kann nicht geladen werden\n";

my $hash = check_install(
    module  => 'CGI',
    version => '3.42',
  );

  print Dumper $hash;
```

Clone

Mit `Clone` können alle möglichen Datenstrukturen kopiert werden – auch Objekte und getiete Variablen. Bei der Kopie handelt es sich um eine echte Kopie, das heißt dass die Datenstruktur rekursiv durchgegangen wird und alles kopiert wird. Es befinden sich also keine Referenzen auf die ursprüngliche Datenstruktur in der Kopie.

```
#!/usr/bin/perl

use strict;
use warnings;
use Clone qw(clone);

my $nested = {
    key1 => [
        { key => 'value' },
        { key => 'value' },
    ],
    key2 => {
        key2_1 => 'test',
        key2_2 => [ 1, 2 ],
    }
};

my $copy = clone( $nested );
```

Module::CoreList

Ist das ein Standardmodul? Seit welcher Perl-Version ist es ein Standardmodul? Diese Fragen können mit `Module::CoreList` beantwortet werden. Das Modul liefert auch gleich ein Skript mit, mit dem man sehr einfach ein Modul überprüfen kann. Auf perlpunks.de gibt es auch ein Online-Formular. Zusätzlich kann man mit `Module::CoreList` nach Module suchen (`find_modules`).

```
C:\>corelist CGI

CGI was first released with perl 5.004

C:\>corelist NonExistant

NonExistant was not in CORE (or so I think)
```

```
#!/usr/bin/perl

use strict;
use warnings;
use Module::CoreList;

my $module = 'CGI';

print Module::CoreList->first_release(
                        $module ),"\n",
        Module::CoreList->first_release(
                        $module, 3.1 ),"\n";
```

Test::Strict

Mit `Test::Strict` kann man Tests schreiben, die überprüfen ob der Code auch wirklich `strict` und `warnings` verwendet. Mit `syntax_ok` wird ein Syntaxcheck mittels `perl -c $datei` durchgeführt.

```
#!/usr/bin/perl

use strict;
use warnings;
use Test::Strict tests => 3;

syntax_ok( $0 );
strict_ok( $0 );
warnings_ok( $0 );
```

Sub::Information

Mit diesem Modul kann man sich alle möglichen Informationen zu einer Subroutine ausgeben lassen. Dazu gehören Name, Adresse, Package und Code der Subroutine. `Sub::Information` fasst die Funktionalität mehrerer Module zusammen und vereinfacht so die Informationssammlung über Subroutinen. Beim Laden des Moduls kann man auch einen eigenen Namen angeben, unter dem die `inspect`-Methode in den eigenen Namensraum importiert werden soll. Damit kann man seinen Namensraum sauber halten.

```
#!/usr/bin/perl

use strict;
use warnings;
# use Sub::Information;
# importiert 'inspect'
use Sub::Information as => 'subinfo';

my $info = subinfo \&test;

print $info->package,"::",
      $info->name,"\n",
      $info->code;

sub test {
    print "hallo\n";
}
```

File::Find::Object

`File::Find` ist toll, aber das Modul hat auch gewisse Nachteile, die durch `File::Find::Object` behoben werden sollen. Mit `File::Find::Object` bekommt man einen Iterator, mit dem man sich durch die gefundenen Einträge "navigieren" kann.

```
#!/usr/bin/perl -l

use strict;
use warnings;
use File::Find::Object;

my $ffo = File::Find::Object->new(
                                    {}, '.' );

while( my $file = $ffo->next ){
    print $file;
}
```

Neues von TPF

Grants im 1. Quartal 2009

Im ersten Quartal 2009 wurden zwei neue Grants von der Perl Foundation genehmigt:

- Improving learn.perl.org von Eric Wilhelm und Tina Connolly
- Web.pm - a lightweight web framework for Perl 6 von Ilya Belikin, Carl Mäsak und Stephen Weeks

Updates zu laufenden Grants

Port pyYAML to Perl

Ingy hat YAML::Perl geladen, eine Pure Perl von PyYAML. Man kann damit mittels "ysh -MYAML::Perl" spielen. Ingy hat den Perl-Port von PyYAML zu 95% fertig. Bei den letzten 5% hängt er zwar ein wenig, aber das Release von YAML::Perl soll in den nächsten Tagen kommen. Das neue Modul soll einiges mehr können, als es YAML.pm bisher tat. Ingy will auch noch jede Menge Dokumentation schreiben.

Test::Builder 2

Nachdem in den letzten Monaten nichts passiert ist, fängt Michael wieder mit dem Programmieren an.

SMOP

Daniel hat eine lange Liste mit den ganzen Neuigkeiten erstellt. Sie ist auf der Seite der Perl Foundation zu finden

Archive::Zip

Alan hat weiter an Bugfixes gearbeitet, auch wenn diese Woche nicht sehr produktiv war. Probleme traten beim Unicode-Handling auf. Alan hat eine Entwicklerversion des Moduls auf CPAN geladen. Mittlerweile hat Alan Unicode-Unterstützung für extractTree und addTree unter Windows hinzugefügt. Alan hat auch die Unterstützung von Archiven mit Unicode-Dateinamen abgeschlossen. Jetzt arbeitet er daran, Archive zu erzeugen, die Dateien mit Unicode-Dateinamen beinhalten.

Extending BSDPAN

Keine Neuigkeiten

Perl Cross-Compilation for WinCE and Linux

Vadim arbeitet hauptsächlich an seinem anderen Grant.

Tcl/Tk Access in Rakudo

Vadim wird den Grant wahrscheinlich bald abschließen.

The Perl Survey

Kieren hat viel Vorarbeit geleistet und hat die Dumps der ursprünglichen Umfrage erhalten. Da Kieren aber einen neuen Job angefangen hat, genießt dieser erstmal Priorität.

The Mojo Documentation Project

Sebastian hat das erste Kapitel geplant, das eine allgemeine Einführung in die Webprogrammierung sein soll. Es wird ähnlich wie ein CGI-Tutorials sein, aber mit modernen Mojo-Beispielen. Nach einer Operation hat sich Sebastian bis 14. März eine Auszeit genommen.

Embeding Perl into C++ Applications

Es gibt ein wenig was Neues zu den Tests. Leon hat insgesamt einiges getan und die komplette Liste mit seinen Updates ist auf der Perl-Foundation-Seite zu finden. Unter anderem arbeitet er an den Bindings für Reguläre Ausdrücke.

Moose Docs

Dave hat mit der Moose-Dokumentation begonnen und hat eine erste Version der Dokumenation als Teil von Moose 0.66 veröffentlicht. Siehe auch sein use.perl.org-Journal. Mittlerweile hat er die ersten zwei Meilensteine des Grants erreicht und arbeitet weiter an den Beispielen.

Integrating Padre with Parrot and Rakudo

Gabor hat mit der Arbeit begonnen und bloggt über den Fortschritt in seinem Blog (http://szabgab.com/blog.html). Padre hat mittlerweile ein (langsames) Syntax-Highlighting für Perl 6. Gabor hat das Parrot-Plugin von Padre gefixt und ein vergefertigtes Paket mit Parrot, Rakudo, Padre und Strawberry Perl veröffentlicht. Weiterhin hat er Unit Tests für die API zum Einbetten von Parrot in Perl 5 geschrieben. Siehe auch http://www.szabgab.com/blog/2009/01/1232096454.html. Im März hat Gabor den Grant abgebrochen.

Perl 6 Microgrants: Buch in Wikibooks

Andrew Whitworth erhält einen "Perl 6 Microgrant" von der Perl Foundation, um ein Buch auf Wikibooks zum Thema "Perl 6" zu schreiben.

Er soll dabei Inhalte hinzufügen, die sich aus den Synopsen und den Tests ergeben, weiterhin soll Whitworth (Teile der) Perl 6 Dokumentation integrieren und neue Autoren und Leser anwerben.

Smart-Websolutions

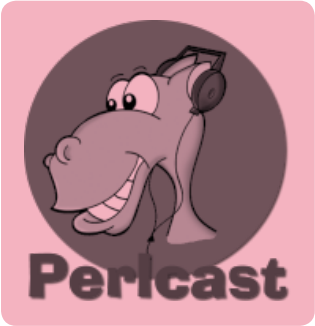
Windolph und Bäcker GbR

Perl-Programmierung

info@smart-websolutions.de

Neue Perl-Podcasts

Perlcast.com



brian d foy

Nach langer Pause gibt es eine neue Aufnahme bei Perlcast.com: brian d foy hat schon vor einiger Zeit über einen "typischen" Tag gesprochen. Diese Episode ist für jeden etwas, der sich über einen Tag von brian d foy amüsieren möchte.

Michel Rodriguez über XML::Twig

Während der OSCON 2008 wurde dieser Podcast aufgenommen, in dem der Autor von XML::Twig über den beliebten XML-Parser spricht.

Termine

Mai 2009

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

Juni 2009

01. Treffen Vienna.pm
02. Treffen Frankfurt.pm
04. Treffen Dresden.pm
08. Treffen Ruhr.pm
10. Treffen Hamburg.pm
11. Treffen Cologne.pm
12./13. Französischer Perl-Workshop
15. Treffen Erlangen.pm
17. Treffen Darmstadt.pm
22.-24. YAPC::NA in Pittsburgh
24. Treffen Berlin.pm
30. Treffen Bielefeld.pm

Juli 2009

02. Treffen Dresden.pm
06. Treffen Vienna.pm
07. Treffen Frankfurt.pm
08. Treffen Hamburg.pm
09. Treffen Cologne.pm
15. Treffen Darmstadt.pm
18. Treffen Ruhr.pm (Grillen in Essen)
20. Treffen Erlangen.pm
28. Treffen Bielefeld.pm
29. 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“

Warum Fachleute auf Schulungen gehen sollten

Externe Schulungen sind wie eine Studienfahrt mit Fremden, die am gleichen Thema arbeiten. 5 Tage lang 'mal nicht mit den eigenen Kollegen im immer gleichen Brei schwimmen. Es ist nämlich nicht richtig, alles im Selbststudium lernen zu wollen: Jede(r) von uns hat die Fundamente seines Könnens in Schulen und Universitäten gelernt, und gerade wer im Betrieb stark belastet ist, hat keine Chance, schwierige Themen „nebenbei“ am Arbeitsplatz zu erlernen oder neue Kollegen anzulernen.

Schauen Sie 'mal: www.Linuxhotel.de

Wer sich wirklich intensiv auf eine Arbeit einläßt, will auch Spaß dabei haben! Im Linuxhotel kombinieren wir deshalb ganz offen eine äußerst schöne und luxuriöse Umgebung mit höchst intensiven Schulungen, die oft bis in die späten Abendstunden zwanglos weitergehen. Natürlich freiwillig! Wer will, zieht sich zurück! Und weil unser Luxushotel ganz weitgehend per Selbstbedienung läuft, sind wir gleichzeitig auch noch sehr preiswert.