Ausgabe 3/2010 – www.foo-magazin.de - 6,00 €

$foo

PERL MAGAZIN

AI::CBR
Case-Based Reasoning

OTRS::CiCS
eine Erweiterung für OTRS

Moose Tutorial
Teil 1

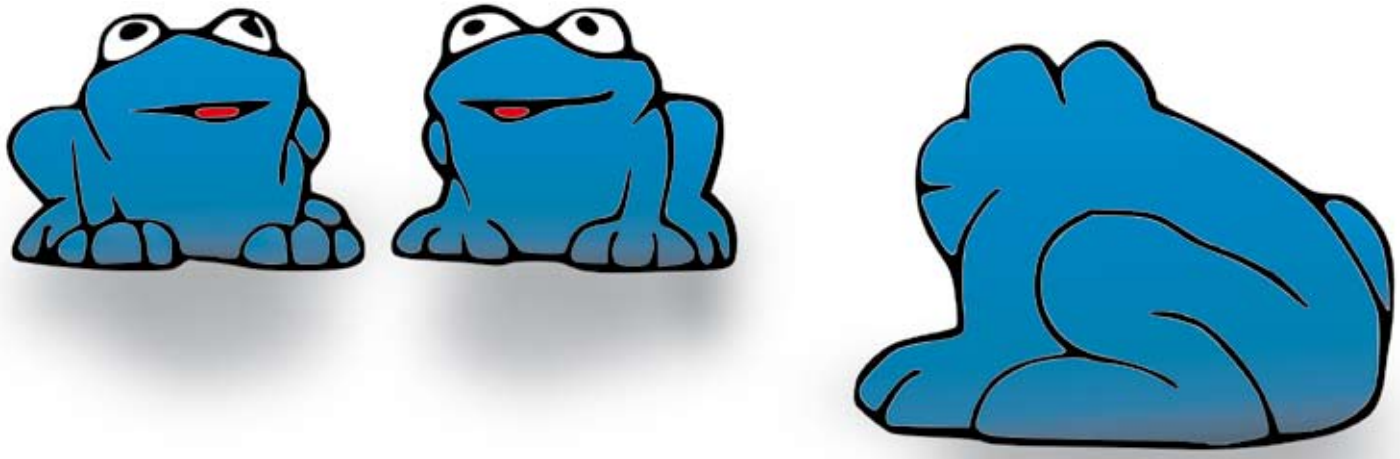
Nr. 15

FrOSCon?

Free and Open Source Software Conference

Klar, am 21. & 22.08.2010 in Sankt Augustin!

Call for Papers 12.4. bis 23.5.



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

Highlights dieses Jahr sind:

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

Weitere Infos auf www.froscon.de und auf twitter

Daten über Perl-Programmierer/innen

2007 gab es eine Umfrage unter den Perl-Programmierern. Schon damals sind interessante Informationen herausgekommen. Kieren Diement wurde von der TPF unterstützt, um eine weitere Umfrage durchzuführen.

In diesem Jahr war es soweit. Rund 3.500 Programmierer haben daran teilgenommen - und das in nur 3 Wochen, wobei die ersten 1.000 Antworten innerhalb von 24 Stunden eingingen. Im Unterschied zur letzten Umfrage, wurden die Methoden und Fragen komplett überarbeitet. Aber jetzt ein wenig aus dem "Plauderkasten".

Kieren Diement schätzt auf Grund der Umfrage-Teilnehmer die Anzahl der Perl-Programmierer auf 250.000.

Besonders interessant sind meiner Meinung nach die Demographischen Werte: 96% der Teilnehmer waren männlich und nur 3% weiblich (1% hat keine Angaben gemacht). Ist diese Diskrepanz ein generelles Problem in der IT oder ist das etwas Perl-Spezifisches?

Die Altersstruktur zeigt, dass der Großteil zwischen 30 und 39 Jahren alt ist (41%) oder zwischen 25 und 30 (22%). Um auch wieder etwas mehr junge Programmierer für Perl begeistern zu können, müsste vielleicht etwas an Schulen und Universitäten gemacht werden. Hier fände ich eine Initiative von Perlmonger-Gruppen sehr interessant. Ich selbst habe mit einer Uni und mit einer Schule gesprochen. Mal schauen was daraus wird. Auch die Auftritte bei "Nicht-Perl-Ereignissen" helfen sicherlich. Mal nach 3 Jahren solcher Initiativen so eine Umfrage zu machen, wäre sicherlich interessant.

Mein persönliches Gefühl wurde durch die Umfrage bestätigt: Deutschland ist ein starkes Perl-Land - nach den USA das Land mit dem größten Anteil bei den Teilnehmern. Man sieht bei den Zahlen auch, dass die Mobilität sehr groß ist. 8.6% der Teilnehmer aus Deutschland sind aus dem Ausland immigriert.

Es gibt noch jede Menge anderer interessanter Daten, die man aus der Umfrage herausziehen kann. Kieren Diement hat alles auf Github gestellt.

Viel Spaß bei dieser Ausgabe.

Renée Bäcker

Die Codebeispiele können mit dem Code

ngfx2m29

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/issue15

IMPRESSUM

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

INHALTSVERZEICHNIS

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

Renée Bäcker

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

Herbert Breunung

Ein perlbegeisteter Programmierer aus dem ruhigen Osten, der eine Zeit lang auch Computervisualistik studiert hat, aber auch schon vorher ganz passabel programmieren konnte. Er ist vor allem geistigem Wissen, den schönen Künsten, sowie elektronischer und handgemachter Tanzmusik zugetan. Seit einigen Jahren schreibt er an Kephra, einem Texteditor in Perl. Er war auch am Aufbau der Wikipedia-Kategorie: "Programmiersprache Perl" beteiligt, versucht aber derzeit eher ein umfassendes Perl 6-Tutorial in diesem Stil zu schaffen.

Thomas Fahle

Perl-Programmierer und Sysadmin seit 1996.
Websites:

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

Mario Kemper

Mario Kemper studiert Wirtschaftsinformatik an der Universität zu Köln. Parallel zum Studium arbeitet er in der Qualitätssicherung eines Kölner Softwareunternehmens. Seit zwei Jahren investiert er einen großen Teil seiner Freizeit in „Shutter“, eine Gtk2-Perl-Applikation zur Erstellung von Bildschirmfotos.

Torsten Thau

Torsten Thau studierte Informatik (Dipl.) an der TU Chemnitz und der University of Delaware und arbeitete im Anschluss an das Studium für das Systemhaus eines großen deutschen Telekommunikationsunternehmens. Seit 2003 arbeitet er mit Perl - insbesondere mit der Perl-API zu BMC/Remedy ARS. Kaum ein Jahr später erweckte das Interesse für OTRS, welches auch schnell produktiven Einsatz fand. 2006 gründete er mit drei weiteren Partnern die c.a.p.e. IT GmbH (http://www.cape-it.de), die in Mitteldeutschland zu den wenigen Dienstleistern und Anbietern von Open-Source- und proprietären Service-Management- und IT-Infrastruktur-Produkten zählt. Die c.a.p.e. IT GmbH stellt eine Reihe von OTRS-Erweiterungen kostenfrei zur Verfügung und bearbeitet selbst den Großteil der geschäftlichen Korrespondenz und Projektabwicklung in OTRS(::CiCS).

Darko Obradovic

Darko Obradovic ist Wissenschaftlicher Mitarbeiter am Deutschen Forschungszentrum für Künstliche Intelligenz und beschäftigt sich dort aktuell als Doktorand vorwiegend mit Analysen von Sozialen Medien im Web 2.0. Darüberhinaus ist er auch auf den Gebieten des Wissensmanagements und der Spiele-KI aktiv, und Autor von internationalen wissenschaftlichen Veröffentlichungen. Für fast alle Projekte benutzt er seit über zehn Jahren Perl, ist Gründer von Kaiserslautern.pm, Autor einiger CPAN-Module und war Sprecher bei den europäischen YAPCs 2007 bis 2009.

OTRS::CiCS – eine Erweiterung für OTRS

OTRS

Die Open Source Welt kennt eine Reihe von Troble-Ticket-Systemen (TTS), die z.T. bereits auf eine lange Historie zurückblicken können. Neben Request Tracker (RT), welches seit 1996 existiert, findet seit 2002 das Open-Ticket-Request-System (OTRS) immer größere Verbreitung. Mit geschätzten 70.000 Installationen weltweit kann es durchaus zu den Big-Playern unter den TTS gezählt werden. Ein Grund für die weite Verbreitung ist natürlich die Lizenzkostenfreiheit, aber sicherlich auch die freie Verfügbarkeit vieler Erweiterungspakete, welche die Einsatzszenarien von OTRS über die eines einfachen TTS hinaus erweitern. Mittels Zusatzmodulen ist so z.B. eine projektbezogene Zeiterfassung (Paket "TimeAccounting"), der Aufbau einer Wissensdatenbank (Paket "FAQ"), die Integration von System-Monitoring-Tools wie z.B. Nagios (Paket "SystemMonitoring") oder die ITIL-konforme Abbildung von IT Service Management Prozessen (OTRS::ITSM) möglich.

OTRS, mittlerweile bei Version 2.4.7 angekommen, muss sich oft den Vorwurf einer anachronistischen Nutzeroberfläche, der fehlenden Selbsterklärbarkeit oder der z.T. schleppenden Umsetzung von Feature-Requests gefallen lassen und wird so häufig als "von IT-lern für IT-ler gemacht" verstanden. Da die Einsatzszenarien jedoch weit über IT Service hinausgehen und gerade im Kundenservice häufig Mitarbeiter ohne einen IT-lastigen Hintergrund eingesetzt werden, war die Verbesserung der OTRS-Bedienbarkeit eine der Hauptmotivationen für die Umsetzung des OTRS-Erweiterungpaketes OTRS::CiCS (Customer Information and Communication System). Im Laufe der Zeit sind zu diesen rein oberflächlichen Anpassungen auch tiefgreifende Funktionserweiterungen hinzugekommen. Einige der Erweiterungen aus OTRS::CiCS haben bereits ihren Weg in den Hauptzweig von OTRS gefunden, wie z.B. ein FAQ-Genehmigungsprozess, die Möglichkeit einzelne Artikel zu Drucken, kompaktere Queueansichten bzw. Ticketübersichten, Schnellzugriff auf perönliche Suchvorlagen und andere.

Einige OTRS::CiCS Funktionen

Die offensichtlichsten Anpassungen, die mit der OTRS-Erweiterung OTRS::CiCS bzw. ihrem ITSM-Pendant OTRS::CiCS::ITSM ins Auge fallen, sind Änderungen an der GUI. Diese wurden z.T. in Kooperation mit dem Lehrstuhl für Arbeitswissenschaften der TU Chemnitz erarbeitet. Doch die Erweiterungen gehen wesentlich tiefer als für den GUI-Betrachter sofort ersichtlich. Nachfolgend sollen einige der Erweiterungen kurz beschrieben werden.

Eine zusätzliche Queueansicht erlaubt ebenfalls die Sortierung der Tickets und ermöglicht nach Aufklappen zusätzlich die Voransicht der Tickets und den sofortigen Zugriff auf die Anhänge des angezeigten Artikels. Diese Anzeige stellt somit einen Kompromiss zwischen der neuen, sehr übersichtlichen kurzen Ticketliste und der Voransichtsliste dar. Des Weiteren gibt es die Möglichkeit auch den in der Voransicht angezeigten Artikeltyp zu konfigurieren. Üblicherweise ist dies der letzte Artikel der vom Kunden verfasst wurde, was nicht in jedem Anwendungsfall die beste Lösung ist.

Erweiterte persönliche Einstellungen des Agenten erleichtern OTRS-Neulingen das Verständnis für gesperrte Tickets. Gesperrte Tickets werden üblicherweise nicht, bzw. erst nach Klick auf "alle", in der jeweiligen Queueansicht dargestellt. Die persönliche Einstellung "Anzeige aller Tickets" ermöglicht nun eine persistente Anzeige aller, d.h. auch gesperrter, Tickets. Eine weitere Einstellung "Maske nach Ticketabschluss" ermöglicht einen Wechsel in ein gerade

geschlossenes Ticket anstelle des Standardverhaltens. Dies ist insbesondere bei automatisierten Statuswechseln nach Erreichen eines Status vom Typ "geschlossen" sinnvoll und ermöglicht den Agenten noch einen abschließenden, prüfenden Blick auf seine Arbeit.

Die in bzw. zu OTRS::CiCS zusätzlich erhältlichen OTRS-Themen bieten eine übersichtlichere Anzeige der OTRS-Module und der im aktuellen Modul vorhandenen Aktionen (vertikale Navigationsleisten). Das jüngste Thema "KixOrange-Dropping" (zukünftig "MenuDropping")" ermöglicht direkten Zugriff auf alle Modulaktionen aus jedem OTRS-Modul heraus. Der Nutzer muss nicht erst aus dem Ticket-Bereich nach FAQ wechseln, um die Möglichkeit zu erhalten, einen FAQ-Eintrag anzulegen oder zu suchen. Die Wichtigkeit einer alternativen Darstellung ist spätestens dann klar, wenn Nutzer mit relativ vielen Zugriffsrechten auf OTRS-Installationen arbeiten, auf welchen viele Erweiterungspakete installiert sind.

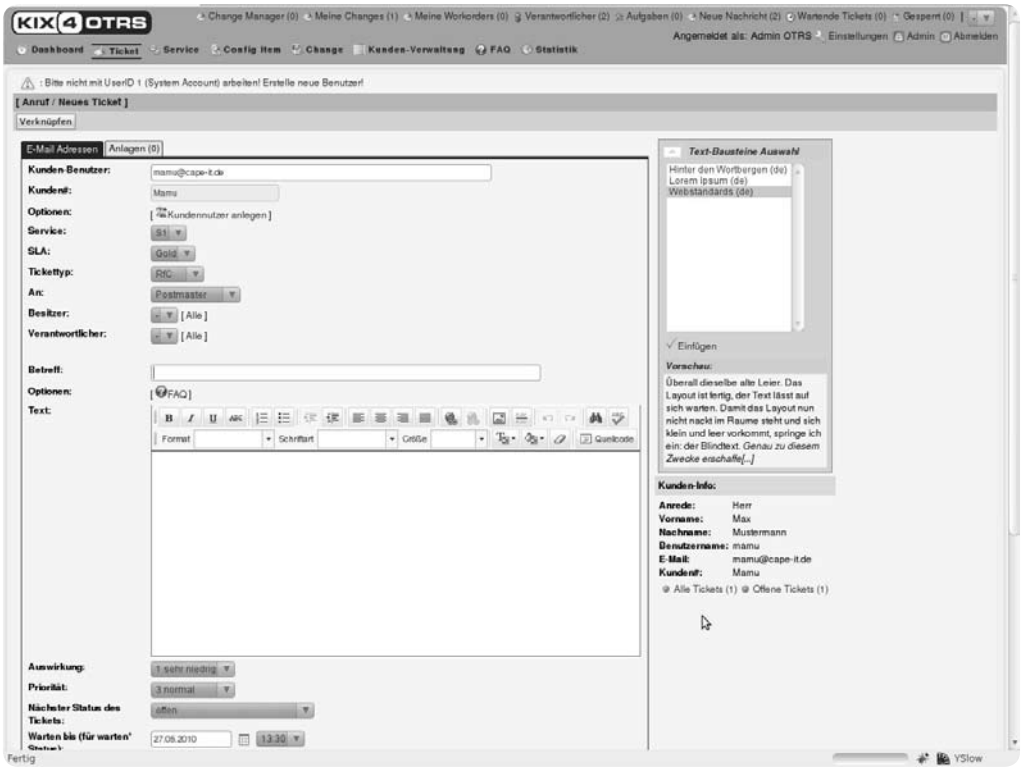


Abbildung 1: Erstellung Telefonticket ohne Verwendung Tab-Darstellung

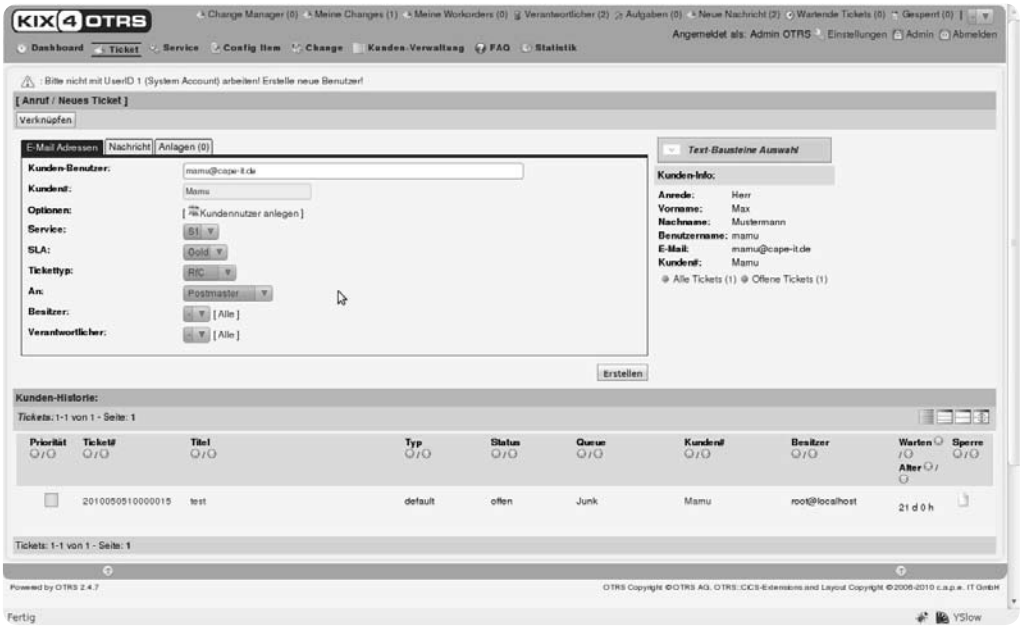


Abbildung 2: Erstellung Telefonticket mit Verwendung Tab-Darstellung

Um die Ticket-Bearbeitungsmasken übersichtlicher zu gestalten wurden diese auf konfigurierbare Tab-/Reiter-Darstellung umgestellt. Dies ist insbesondere dann hilfreich, wenn ein Scrollen vermieden werden soll. Die Idee für die Bearbeitung ist hier, dass anstelle eine Maske von oben nach unten durchzuarbeiten, bei Bedarf der jeweilige Reiter ausgewählt und mit Eingabedaten versehen werden kann. Ein zusätzlicher Reiter zeigt bei bestehenden Tickets die aktuellen Ticketdaten sowie den letzten Artikel an. So kann man das Öffnen eines zusätzlichen Browsertabs sparen wenn während des Bearbeitungsschrittes noch Einsicht auf die Ticketdaten benötigt wird.

Die Ticketinhaltsansicht wurde vollständig überarbeitet und ermöglicht nun eine konfigurierbare Anzeige der Kerndaten, wie Ticketstatus, Queue- oder Servicenamen (bei langen Bezeichnern umgebrochen und nicht abgekürzt). Angaben zu den für das Ticket relevanten Personen werden zunächst verkürzt dargestellt. Bei Bedarf können alle Daten in einem Pop-Up-Dialog angezeigt werden. Grundsätzlich werden alle Ticketdaten auf mehrere Reiter verteilt dargestellt, um die initiale Ansicht eines Tickets übersichtlicher und leichter erfassbar zu halten. Daher gibt es je einen Reiter für die bisherige Kommunikation zum Ticket, am Ticket gesammelte Kontakte, Ansicht aller verlinkten Objekte, Anzeige der Freitext- und -zeitfelder und weitere. Ein standardmäßig inaktiver Ticketreiter ermöglicht das Anlegen von Notizen, Status-, Bearbeiter- oder Verantwortlicherwechsel ohne den Aufruf einer zusätzlichen Bearbeitungsmaske und illustriert die Möglichkeiten dieser Ticketansicht. Die einzublendenden Reiter sind konfigurierbar und generieren ihren Inhalt unabhängig vom Ticketansichtsmodul. Diese grundlegende Änderung des Aufbaus der Ticketinhaltsansicht ermöglicht die unproblematische Integration weiterer, externer Daten in die Ticketbearbeitung. Dazu gehören z.B. die Anzeige komplexer Verrechnungsschlüssel oder Kundeninformationen die bei Ticketansicht aus ERP-Systemen wie z.B. Navision abgefragt werden.

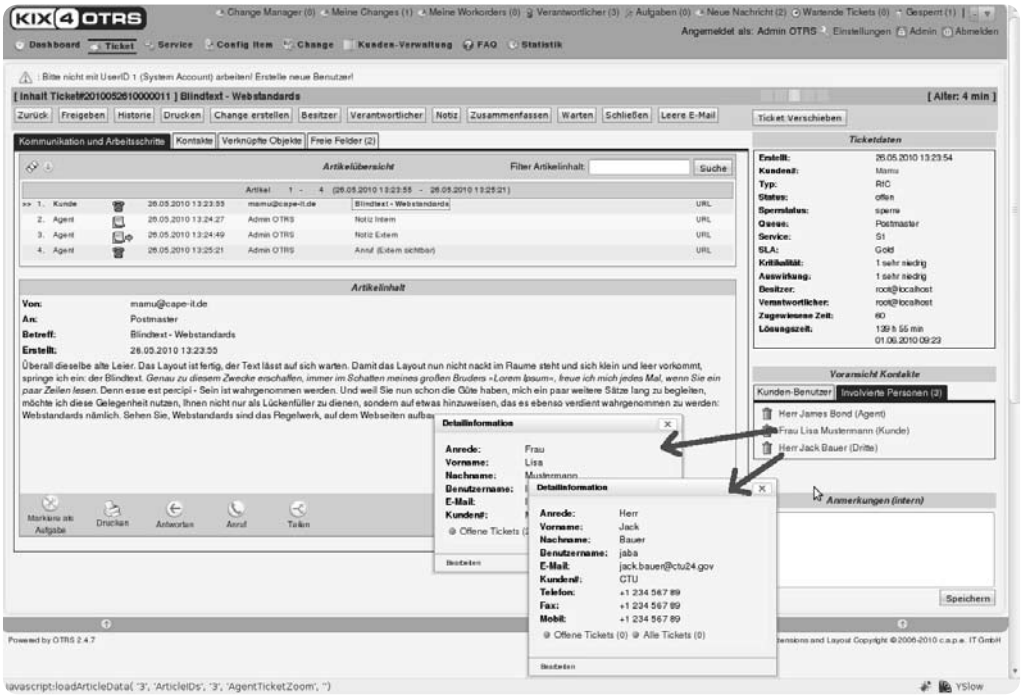


Abbildung 3: Ticketansicht mit aufgeblendeten Kontaktdetails zu involvierten Personen

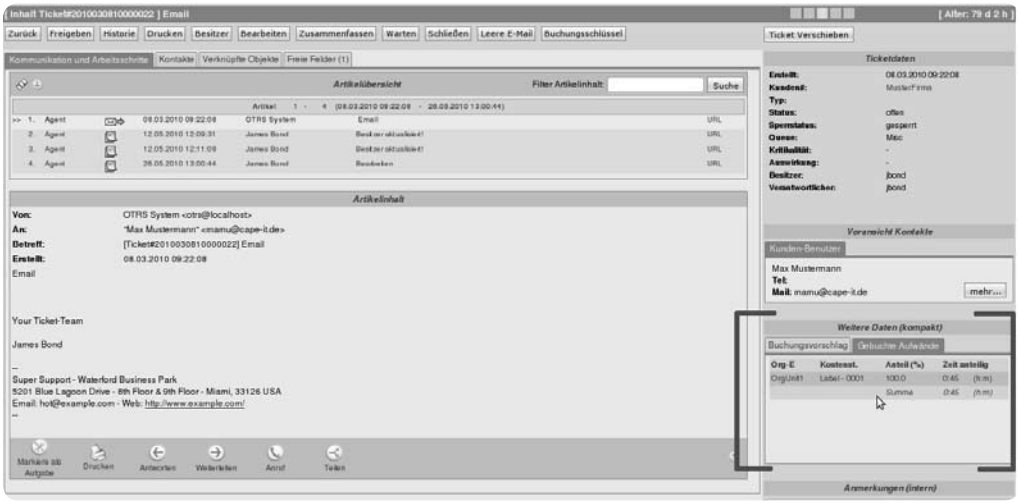


Abbildung 4: Ticketansicht mit kompakter Darstellung erweiterter Ticketdaten (hier Buchungsschlüssel)

Im Reiter "Kommunikation" wird die Übersicht der Artikel kompakter dargestellt. Mittels eines Suchtext-Artikelfilters können nur die Artikel angezeigt werden, die einen bestimmten Suchstring enthalten; der OTRS-eigene Artikelfilter nach Artikeltypen ist zusätzlich anwendbar.

Die artikelbezogenen Aktionen wie Um- und Weiterleiten, Antwort verfassen, Anruf dokumentieren, Ticket teilen, Drucken, usw. sind in OTRS::CiCS durch Veränderung der Konfiguration und ohne Bearbeitung des Quellcodes zu beliebigen Artikeltypen zuordenbar. So steht die Funktion "Weiterleiten" standardmäßig nur für Email-Artikel zur Verfügung. Sie kann aber durchaus auch bei Notiz-Artikeln hilfreich sein. Dies ist insbesondere dann relevant wenn eigene Artikeltypen verwendet werden sollen. Eine zusätzliche, aber kritisch zu betrachtende, artikelbezogene Funktion ist "Artikel

kopieren/verschieben". Diese ermöglicht das Kopieren bzw. Verschieben von Artikeln von einem Ticket A zu einem Ticket B. Sie ist standardmäßig inaktiv und nur Administratoren vorbehalten.

Ein einfaches Off-The-Record-Notizfeld am Ticket ermöglicht den Agenten das Eintragen von kurzen Notizen, die nicht als Artikel am Ticket hinterlegt werden sollen. Über dieses Feld erfolgt keine Historisierung. Es dient als eine Art "Schmierzettel".

Das Vorschlagen eines FAQ-Eintrages durch jeden Agenten verbessert die FAQ-Integration. Durch Setzen eines Flags "FAQ-Eintrag erstellen" kann jeder Ticketbearbeiter seinen aktuell angelegten Artikel als FAQ-Eintrag vorschlagen. Dabei wird der FAQ-Eintrag aus dem ersten Artikel des Tickets (der meist die Symptomatik, die Problembeschreibung enthält) und dem markierten Artikel als Lösungsbeschreibung erstellt. Der so erstellte FAQ-Eintrag wird in eine spezielle FAQ-Kategorie eingeordnet, da er noch einiger redaktioneller Bearbeitung bedarf.

Die Funktion "Involvierte Personen" ermöglicht das Hinterlegen von relevanten Personendaten am Ticket. Dies kann manuell geschehen, erfolgt aber auch automatisiert durch Analyse aller Empfänger an ein-/ausgehenden Emails und der Besitzerwechsel an einem Ticket. Dabei werden die Personen nach den Typen "Kunde", "Agent" oder "Dritte" kategorisiert. Voraussetzung für die Eintragung eines Kontakts ist das Vorhandensein eines entsprechenden Eintrages in den Kunden- oder Agentenbackends.

Ein zusätzliches Frontendmodul bietet die Möglichkeit zur Suche nach Kundennutzern und der anschließenden Suche nach für diesen Kontakt relevanten Tickets. Dabei kann der Kundennutzer sowohl als Kunde aber auch als weitere am Ticket involvierte Person hinterlegt sein. In erster Linie stellt dieses Modul aber die Anzeige der Kundendaten und der zu diesen im System vorhandenen Tickets zur Verfügung.

Eine Erweiterung zur Abwesenheitsfunktion des OTRS ermöglicht das Hinterlegen eines Stellvertreters, der alle vom System an den abwesenden Agenten versendeten Nachrichten in Kopie erhält. Bei wichtigen Kundenrückmeldungen kann dieser sofort handeln. Vertretungsbedingte Bearbeitungslücken werden so minimiert.

OTRS bietet die Möglichkeit verschiedene Datenquellen für die Haltung und Pflege von Kundendaten zu verwenden. Unter diesen Optionen befindet sich auch die Verwendung von LDAP-fähigen Verzeichnisdiensten. Diese sind jedoch im Standard-OTRS auf Leseaktionen beschränkt und erlauben keinen schreibenden Zugriff auf den Verzeichnisdienst. OTRS::CiCS bringt die Möglichkeit zum Schreiben von Kundendaten in das angebundene LDAP-Kundennutzerbackend mit. Eine Kopieren-Funktion vereinfacht ferner das Anlegen neuer Kundennutzereinträge als veränderte Kopie eines bereits bestehenden.

Eine einfache DMS-Funktionalität (Dokumentenlink) ermöglicht das Verknüpfen von Dokumenten, welche in einem auf dem OTRS-System eingehängten Filesystem liegen. Dabei verfolgt das System Änderungen, wie Umbenennungen oder Verschieben der verknüpften Dateien und bietet auf OTRS-Gruppen oder -Rollen basierendene Mechanismen zur Zugriffsbeschränkung. Die so mit einem Ticket verknüpften Dokumente können über die Weboberfläche heruntergeladen werden, ohne sich direkten Zugriff auf das Filesystem verschaffen zu müssen. Diese Funktion ist in Verbindung mit dem OTRS-Zusatzpaket "FileManager" sehr gut einsetzbar.

Ein einfaches Skript (otrs.CreateQGR.pl) ermöglicht das Anlegen und Aktualisieren von Queues, Gruppen, Rollen sowie Rollen-Gruppen-Zuordnungen basierend auf einem CSV-File. Dadurch wird das Pflegen, Versionieren und Dokumentieren von komplexen Rollen- und Queuestrukturen vereinfacht und der Administrator umgeht "Klick-Orgien" in der OTRS-Oberfläche.

Auf Basis der bereits in OTRS vorhandenen Access-Control-Lists (ACLs) kann durch kleinere Entwicklungen ein Workflow mit einer bestimmten Statusfolge vorgegeben werden. Da dies eine häufig angefragte Anforderung ist, bietet es sich an eine einfache Konfigurationsmöglichkeit zur Verfügung zu stellen. Diese sollte ohne Entwicklungsaufwand und nur durch Konfiguration derartige Prozesse definieren können. Ein solcher Mechanismus ist nun ebenfalls in OTRS::CiCS vorhanden und ermöglicht eine Tickettyp-bezogene Definition von Workflows sowie automatische Statuswechsel und einfache automatische Aktionen wie z.B. Queuewechsel, Tickettyp- oder Status-Update bei Erreichen eines bestimmten Status. Die Erweiterung betrifft natürlich auch die Ermittlung von zulässigen Folgestatus bei Eingang einer Follow-Up-Nachricht.

Mit OTRS::ITSM ab der Version 2.0.x steht nun auch im Hauptzweig von OTRS ein Event-Mechanismus für Config Items zur Verfügung. Dieser ist jedoch derzeit noch auf Post-Action-Events, also Mechanismen, die nach der Umsetzung einer bestimmten Aktion, wie z.B. das Anlegen einer neuen Config Item-Version, ausgeführt werden, beschränkt. So wird beispielsweise die Historisierung von Änderungen an Config Items über einen solchen Event-Handler abgebildet. Die CiCS-Erweiterung für OTRS::ITSM bietet hier auch die Möglichkeit Pre-Action-Events zu nutzen. Diese werden vor der eigentlichen Aktion ausgeführt und ermöglichen somit z.B. die automatische Manipulation der anzulegenden Config Item-Versionen oder Plausibilitätsprüfungen. Ein klassischer Anwendungsfall hierfür ist das Prüfen ob eine bestimmte IP oder Name bereits in der CMDB enthalten und somit belegt ist. Die Integration der Rückgabewerte dieser Pre-Action-Events in die Frontend-Module und Skripte von OTRS stellt sicher, dass der Nutzer auch mit entsprechenden Fehlermeldungen versorgt werden kann. Diese Funktion wird u.a. dadurch umgesetzt, dass auch in Klasse Kernel::System::ConfigItem die Option zur Definition von eigenen Superklassen eingefügt wurde, wie sie von Klasse Kernel::System:Ticket (SysConfig-Schlüssel "Ticket::CustomModule") her bekannt ist.

Eine weitere OTRS::ITSM bezogene Erweiterung stellt der Config Item Baum dar. Er visualisiert einen Suchbaum, resultierend aus dem Graph der verknüpften Configuration Items. Dies erleichtert die Nachvollziehbarkeit der Auswirkungen von Störungen verknüpfter Config Items und ist jeweils auf einen ausgewählten Verknüpfungstyp bezogen.

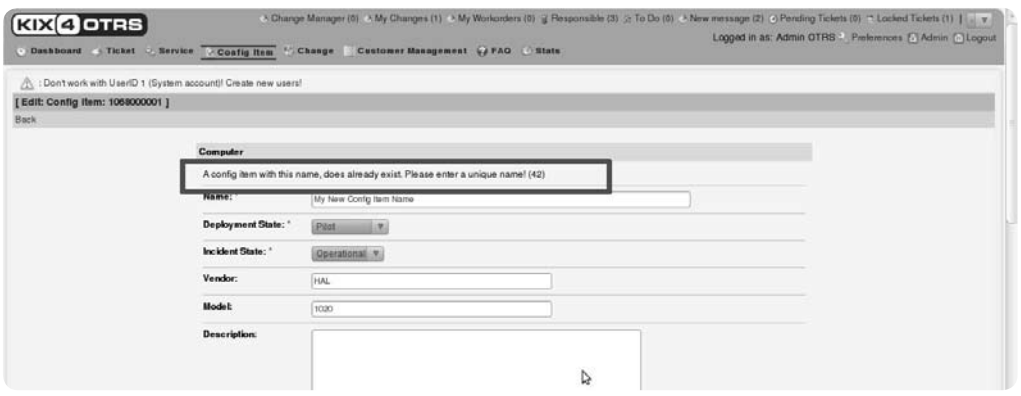


Abbildung 5: Pre-Event-Beispiel - Plausibilitätsprüfung meldet Verstoß gegen Eindeutigkeitsgebot bei Bearbeitung eines Config Items

Funktionsweise der OTRS-Paketverwaltung

OTRS bringt eine eigene und recht mächtige Paketverwaltung mit. Diese kann sowohl über die Web-GUI (Admin-Bereich Unterpunkt Paket Management), als auch über Kommandozeile (<OTRS_HOME>/bin/opm.pl) bedient werden. Das Kommandozeilenscript dient hierbei auch zum Erstellen eigener Pakete. In OTRS-Paketen sind neben den zum Paket gehörenden Dateien (Base-64 codiert) auch Anweisungen, wie z.B. während des Installationsvorgangs auszuführenden Perl-Code oder die anzulegenden Datenbankstrukturen oder -inhalte, beschrieben. Installiert eine OTRS-Erweiterung nun Dateien, die bereits in der Standard-OTRS-Installation vorhanden sind, werden die Original-Dateien zuvor gesichert und dann mit der Dateiversion aus dem Paket ersetzt.

Grenzen der OTRS-Paketverwaltung

Wer zu spät kommt, den bestraft die Paketverwaltung - so praktisch wie die OTRS-Paketverwaltung ist, so schnell sind auch ihre Grenzen erreicht. So ist das Überschreiben von Dateien in bereits installierten Zusatzpaketen nicht möglich. Die Paketinstallation ermittelt die zu installierenden Dateien und prüft diese gegen bereits vorhandene Pakete. Enthalten zwei Pakete identisch benannte Dateien, verweigert die Paketverwaltung die Installation des zweiten Paketes. Der naive Ansatz wäre hier ein entsprechendes Gegenstück zum bereits installierten Paket zu einwickeln, welches sowohl die komplette Funktion des Original-Paketes, als auch die neuen und veränderten Funktionen der Erweiterung enthält. Das Original-Paket ist dann zu deinstallieren und durch die erweiterte Version zu ersetzen. Davon abgesehen, dass bei der Deinstallation von Paketen auch die damit verbundenen Datenbankstrukturen und -inhalte verloren gehen, erhöht ein solches Vorgehen den Pflegeaufwand für Erweiterungen um ein Vielfaches. Nun wäre es theoretisch möglich während der Installation des zweiten Paketes

die Original-Dateien des ersten Paketes durch ausgeführten Perl-Code zu ersetzen, jedoch entdeckt die Paketverwaltung diese Manipulation später und empfiehlt die Reinstallation des korrupten Paktes. Alle so durchgeführten Änderungen gingen dann verloren. Benötigt wird also ein Mechanismus der das Ersetzen von Teilen von vorhandenen Paketen ermöglicht, ohne diese tatsächlich zu überschreiben.

Erweiterungsmöglichkeiten in ausgewählten Kernmodulen

Neben dem Überschreiben von OTRS-Original-Dateien gibt es noch eine relativ begrenzte Möglichkeit eigene Kernfunktionen z.B. zur Ticket-Klasse hinzuzufügen. Mittels eines SysConfig-Parameters kann ein zusätzliches Perl-Paket angegeben werden, welches die der Klasse Ticket die verfügbaren Methoden um eigene erweitert. Leider ist es damit nicht möglich bestehende Methoden, die in Kernel::System::Ticket selbst enthalten sind, zu überschreiben. Hintergrund ist, dass OTRS hier eine Mehrfachvererbung abbildet. Konkret bedeutet dies, dass Kernel::System::Ticket mehrere Super- klassen, wie Kernel::System::Ticket::Article, den konfigurierten Ticketnummerngenerator, Artikelspeicher- und Suchindexbackend sowie ein frei konfigurierbares Paket, hat und somit deren Methoden erbt. Die in package Kernel::System::Ticket selbst definierten Methoden überschreiben aber die der Superklassen. Existiert eine Funktion im aktuellen package wird nicht in den in @ISA enthaltenen Superklassen nach einer solchen Funktion gesucht. Diese Einschränkung wird dadurch noch verstärkt, als dass dieser Mechanismus zum einen nicht in allen Kernmodulen vorgesehen und zum anderen in Frontendmodulen gar nicht eingeplant ist.

Ein grundlegender Erweiterungsansatz für OTRS

Um die zuvor genannten Grenzen der OTRS-Paketverwaltung und die strikten Einschränkungen der Erweiterungsmöglichkeiten zu entgehen, greift die CiCS-Erweiterung auf die Manipulation der @INC Variable zurück. In dieser Variable sind alle für die Ausführung des aktuellen Skriptes relevanten Pfade hinterlegt. Dies spielt insbesondere bei der Verwendung von use-Anweisungen eine entscheidende Rolle. Um eine use-Anweisung umzusetzen sucht der Perl-Interpreter am Anfang beginnend in den in @INC enthaltenenen Pfaden nach dem in der gerade betrachteten use-Anweisung enthaltenen Paket. Mittels einer separaten Konfiguration in der Datei <OTRS_HOME>/CiCS_Package wird in den für die Verwendung von OTRS relevanten Skripten diese @INC-Variable um eigene Verzeichnise erweitert. Diese Erweiterungsverzeichnisse liegen im OTRS-Verzeichnis selbst. Sie enthalten eine dem OTRS-Verzeichnis identische Struktur, wobei nur die geänderten bzw. überschriebenen Dateien enthalten sind. Die Datei CiCS_Package ist dabei so aufgebaut, dass sie pro Zeile eine Regsitrierung eines solchen Erweiterungsverzeichnisses enthält. Weiter unten registrierte Erweiterungspakete haben eine höhere Priorität, d.h. deren Funktionen überschreiben Funktionen von Erweiterungen niederer Priorität oder des Standard-OTRS..

Um nun die geänderten Pfade in @INC in OTRS zu nutzen, müssen einige Dateien geändert werden. Dabei ist die Konfigurationsdatei <OTRS_HOME>/Kernel/Config.pm von besonderer Bedeutung, da sie in der überwältigenden Mehrheit aller OTRS-Skripte und -Pakete als erste use-Anweisung enthalten ist. Dazu wird in Config.pm lediglich das in Listing 1 (Code-Snipplet zur Manipulation von @INC) dargestellte Code-Stückchen am Ende hinzugefügt.

```
#----------- BEGIN required for CiCS-Tsunami framework ---------------------
my %Config;
&Kernel::Config::Load(\%Config);
if( open ( my $Handle, "$Config{Home}/CiCS_Packages" ) ) {
while ( <$Handle> ){
  chomp;
  if ( $_ && (length($_) > 0) ) {
  unshift(@INC, "$Config{Home}/$_");
  }
}
close $Handle;
}
#----------- END required for CiCS-Tsunami framework -----------------------
```

Listing 1

Da die zentrale Konfigurationsdatei leider nicht in allen Skripten die erste Anweisung vor der Nutzung von weiteren use-Klauseln ist, müssen noch weitere Dateien angepasst werden. Dazu gehören vor allem index.pl, customer.pl, public.pl und xmlrpc.pl in <OTRS_HOME>/bin/cgi-bin. Die Anpassung dieser Dateien geschieht nicht durch eine Auslieferung mit dem CiCS-Paket, sondern durch Einlesen und erneutes Schreiben der geänderten Dateien während des Installationsvorgangs. Dabei werden entsprechende Sicherungsdateien angelegt.

Die Verwendung von angepassten Layouts sollte sich den Prioritäten der CiCS-Erweiterungspaket entsprechend verhalten. Dafür ist eine Anpassung an der zentralen Layout-Funktion notwendig. Hier musste ein Algorithmus eingebaut werden, der zunächst alle CiCS-Erweiterungen nach entsprechenden Template-Dateien (*.dtl) durchsucht und bei Auffinden verwendet. Ist keine spezifische Template-Datei vorhanden, soll auf die entsprechende Standard-OTRS-Datei ausgewichen werden. Hier galt es auch einen Fallback für nicht im jeweiligen Thema vorhandene Template-Dateien gemäß der Erweiterungsverzeichnis-Priorität abzubilden.

OTRS::CiCS und otrs.org

Oftmals wurde die Frage nach dem Zweck der OTRS::CiCS-Erweiterungen gestellt und ob diese nicht das OTRS-Projekt gefährden bzw. warum diese Anpassungen nicht in eben jenes einfließen. Hier stellen sich zwei grundsätzliche Probleme. Es sind ohne Zweifel viele, aber nicht jede der CiCS-Erweiterungen sinnvoll im Standard-OTRS aufgehoben. In jedem Fall erfordert eine Aufnahme in den Hauptzweig von OTRS aber einen erheblichen Koordinierungs- und Diskussionsaufwand mit der OTRS AG. Dies führt auch schon zum zweiten Problem, der zeitlichen Umsetzung. Am Alter einiger offener Feature-Requests sieht man sehr gut, wie lange eine Bearbeitung einer solchen Anfrage dauern kann, bis sie letztlich in OTRS aufgenommen oder abgelehnt wird. Die Alternative zur zeitnahen Umsetzung liegt somit auf der Hand und kann nur eine eigene OTRS-Erweiterung sein. Wenn diese öffentlich und frei zugänglich ist, ist dies noch besser, denn auch so finden die Verbesserungen ihren Weg zurück in das Projekt bzw. die Community.

Fazit und Ausblick

Mit der sich abzeichnenden OTRS Version 3.0 wird das Standard-OTRS über eine aktuelle, zeitgemäße Weboberfläche verfügen und auch weitere funktionale Verbesserungen mit sich bringen. Dennoch ist bereits abzusehen, dass auch hier spezifische Erweiterungen, die für mehr als eine Handvoll Anwendungsfälle relevant sind, möglich und notwendig sein werden. Dies betrifft auch bzw. insbesondere erweiterte Kernfunktionen wie Pre-Events, die weitere Integration externer Daten, konfigurierbare Workflows, Textbausteine, Formularentwurf und -verwendung, CMDB-Anbindung an Inventarisierungstools, etc. Diese Liste ließe sich allein auf Grund der vielfältigen Einsatzszenarien für OTRS noch beliebig fortführen. Festzuhalten bleibt, OTRS ist ein, wenn nicht gar das beste, frei erhältliche Open-Source Servicedesk-Tool und hat seine Grenzen auch jenseits von ITIL und IT Service Management. Es bietet mit seinen vielfältigen Support- und Erweiterungsmöglichkeiten eine interessante Alternative zu den wesentlich weniger offenen Big-Playern in diesem weiten Feld.

Torsten Thau

Shutter – Ein funktionsreiches Werkzeug zur Erstellung von Bildschirmfotos

Bildschirmfotos – eine (fast) alltägliche Aufgabe

Bildschirmfotos, oft auch als Screenshot oder Hardcopy bezeichnet, existieren schon seit es grafische Benutzeroberflächen gibt. Die gängigen Betriebssysteme bringen meist entsprechende Werkzeuge mit, die mit einfachen Tastenkürzeln, z.B. *Druck* und *Alt+Druck*, aufgerufen werden können. So kann entweder der gesamte Desktop oder das aktuelle Fenster aufgenommen werden. Oft landet die Aufnahme danach in der Zwischenablage und kann anschließend weiterverarbeitet werden.

Motivation

Es stellt sich nun die Frage, warum man eine spezielle Anwendung benötigt, um diese Aufgabe zu erledigen. Welche Funktionen würden einen solchen Workflow beschleunigen?

Aus der Not geboren

Parallel zu meinem Studium der Wirtschaftsinformatik an der Universität zu Köln arbeite ich in der Qualitätssicherung eines Kölner Softwareunternehmens. Die Dokumentation und Weiterleitung der Fehler, die während der Testphasen entdeckt werden, nehmen hier einen Großteil der Zeit ein. Je besser ein Fehlerfall dokumentiert ist, desto leichter fällt es anschließend den Entwicklern, das Problem nachzustellen und den Fehler zu beheben. Da Bilder oft mehr sagen als tausend Worte, spielen Bildschirmfotos hier eine wichtige Rolle.

Um diese besonders aussagekräftig zu gestalten, werden die Aufnahmen oft durch Texte, Pfeile oder Ähnliches aufgewertet. Für den Fall, dass das Bildschirmfoto auch öffentlich zugänglich sein soll, müssen sensible Daten aus der Aufnahme entfernt werden.

Ich setze während der Arbeit Ubuntu Linux ein, und ein Werkzeug, welches all diese Anforderungen erfüllt, war hier schlichtweg nicht zu finden. Der Workflow bestand also in der Regel aus mehreren Einzelschritten: Bildschirmfoto aufnehmen, mit GIMP öffnen, Bild aufwendig bearbeiten, sinnvoll benennen und abspeichern. Im Laufe eines Tages sammeln sich so viele Dateien an, so dass man leicht den Überblick verliert - von der Tatsache, dass GIMP sicherlich nicht als intuitives Werkzeug für den Einsteiger bezeichnet werden kann, einmal abgesehen.

Um diesen Arbeitsablauf zu vereinfachen, habe ich 2008 begonnen Shutter zu entwickeln. Aufgrund meiner Vorliebe für Perl stand die Programmiersprache schnell fest - eine GUI Bibliothek musste noch gefunden werden. Der Artikel *WxPerl Tutorial - Teil 1 (Ausgabe Winter 2009)* geht ausführlich auf die Unterschiede der verschiedenen Bibliotheken ein, weshalb ich an dieser Stelle darauf verzichte. Ich entschied mich für `Gtk2`, die Perl-Anbindung and das GIMP-Toolkit (abgekürzt: Gtk+).

Funktionsumfang

Shutter bietet zur Zeit deutlich mehr Funktionen als ich mir in den Anfangstagen erhofft hatte. Die Anwendung erlaubt die Aufnahme des gesamten Desktops, eines bestimmten Bereiches, einzelner Fenster, Fensterinhalten (z.B. eines

Buttons), Menüs, Tooltips und die Aufnahme von Webseiten. Letztere werden in ihrer nativen Größe aufgenommen, ohne dass ein Browser geöffnet werden muss - die Eingabe der URL genügt (siehe Abbildung 1).

Des Weiteren wird ein eingebauter Editor (siehe Abbildung 2) mitgeliefert, der neben den üblichen geometrischen Formen (u.a. Rechtecke, Ellipsen, Pfeile) auch spezielle Funktionen bietet, die bei der Verarbeitung von Bildschirmfotos sinnvoll sind (z.B. ein Highlighter und ein Werkzeug, um sensible Daten unkenntlich zu machen).

Da sich das digitale Leben oft in Webforen oder sozialen Netzwerken abspielt, bietet Shutter die Möglichkeit, die Aufnahmen direkt zu einem Bilderdienst (z.B. imageshack.us) oder per FTP hochzuladen. Hierbei werden automatisch hilfreiche URLs (z.B. BBCode) generiert, die per Copy & Paste verwendet werden können (siehe Abbildung 3).

Abbildung 1: Shutter in Version 0.86.2

Abbildung 2: Eingebauter Editor

Gtk+ und Perl?

Gtk+ ist eine plattformunabhängige API, zur Erstellung von grafischen Benutzeroberflächen. Schaltfächen, Fenster, Schieberegler etc. werden allgemein als *Widgets* bezeichnet. Die low-level Methoden, um diese Widgets zu zeichnen, werden in der GDK-Bibliothek (Gtk+ Drawing Kit) zusammengefasst. Um die Tatsache, dass Gtk+ in der Programmiersprache C implementiert ist, muss man sich als Perl-Entwickler im Allgemeinen nicht kümmern. Das Speichermanagement wird innerhalb der Bindings durchgeführt, so dass man sein Programmierverhalten nicht umstellen muss.

Die erste Anwendung mit Gtk+ und Perl

In Bezug auf die Entwicklung grafischer Anwendungen stellt die ereignisgesteuerte Architektur meist die größte Hürde dar. Das heißt, es existiert kein gewöhnlicher linearer Kontrollfluss - die Ereignisse (*events*) werden, quasi zufällig, durch den Benutzer ausgelöst und von einem Dispatcher an die jeweiligen Funktionen (*callbacks*) weitergeleitet. Widgets können in der Regel auf unterschiedliche Ereignisse reagieren. Zum Beispiel empfängt eine Schaltfläche (`Gtk2::Button`) das Ereignis `clicked`. Um auf diese Ereignisse reagieren zu können, muss das Ereignis mit einer Funktion verknüpft werden.

Das Beispiel in Listing 1 illustriert die angesprochenen Konzepte.

`use Gtk2 '-init';` bindet das Gtk2 Modul ein und startet gleichzeit den Initialisierungsprozess. Die Initialisierung muss zwangsläufig vor dem Aufruf der anderen Funktionen

Abbildung 3: Ergebnis eines Dateiuploads

```
use strict;
use warnings;

#Initialisierung des Perl Moduls Gtk2
#!muss ausgeführt werden bevor
#andere Funktionen des Moduls verwendet werden!
use Gtk2 '-init';

#Erstellung eines neuen Fensters
my $window = Gtk2::Window->new;

#Erstelle einen neuen Button
my $button = Gtk2::Button->new ('Bildschirmfoto aufnehmen');
$button->signal_connect (clicked => \&button_callback);

#Erstelle einen Container mit vertikaler Ausrichtung
my $vbox = Gtk2::VBox->new;

#Packe den Button in die VBox
$vbox->pack_start_defaults($button);

#Füge nun die VBox zum Fenster hinzu
$window->add ($vbox);

#Zeichne das Fenster und den Inhalt (rekursiv)
$window->show_all;

#Starte die Ereignisschleife (event loop)
Gtk2->main;

#Diese Funktion wird aufgerufen, wenn der Button
#das Ereignis 'clicked' erhält
sub button_callback {
   print "button was clicked\n";
}
```

Listing 1

erfolgen. Nachdem in Zeile 13 eine neue Schaltfläche erstellt wurde, wird in Zeile 14 das Ereignis `clicked` mit der Subroutine `button_callback` verknüpft. `signal_connect` verlangt hier nach einer Codereferenz, welches auch die Verwendung einer anonymen Subroutine ermöglicht. So könnte man diesen Abschnitt auch folgendermaßen implementieren:

```
$button->signal_connect (clicked =>
   sub { print "button was clicked\n"; });
```

Die Widgets innerhalb eines Fensters werden hierarchisch angeordnet. Dies geschieht mittels spezifischer Widgets, z.B. `Gtk2::Box`. Abgeleitet vom Typ `Gtk2::Box` sind die Widgets

Abbildung 4: Die erste Anwendung mit Gtk+

`Gtk2::VBox` und `Gtk2::HBox`, die weitere Widgets entweder in vertikaler oder horizontaler Richtung aufnehmen. Mit `Gtk2::VBox->new;` wird eine solche vertikale Box angelegt. Durch den Aufruf `pack_start_defaults($button)` wird die soeben erstellte Schaltfläche der Box hinzugefügt. Würde man nun weitere Widgets hinzufügen, so würden diese zur Laufzeit gemäß der Aufrufreihenfolge angezeigt. Das Widget `Gtk2::Window` leitet sich von der Klasse `Gtk2::Container` ab und kann durch die Methode `add()` lediglich ein weiteres Widget aufnehmen. In Zeile 23 fügen wir also die `Gtk2::VBox` dem Fenster hinzu. `show_all()` in Zeile 26 sorgt dafür, dass alle (die Hierarchie wird hierbei rekursiv durchlaufen) Widgets auf dem Bildschirm gezeichnet werden. Anschließend leitet `Gtk2->main;` die Ereignisschleife ein.

Führen wir den Beispielcode aus, so erscheint ein simples Fenster mit einer Schaltfläche. Klickt man diese an, so wird "button was clicked" auf STDOUT ausgegeben (siehe Abbildung 4).

Der Weg vom Fenster zum Bildschirmfoto

Die GDK-Bibliothek stellt, wie bereits erwähnt, die low-level Funktionalitäten zum Zeichnen der Widgets zur Verfügung. Gleichzeitig abstrahiert sie das zugrunde liegende Windowing-System (z.B. das X Window System bei den meisten Unixderivaten). Da Shutter momentan nur auf Unixderivaten lauffähig ist, werde ich in diesem Absatz näher auf das X Window System eingehen. Ein Fenster, so wie es im allgemeinen Sprachgebrauch verwendet wird, wird in der Terminologie des X Window Systems als *top-level window* bezeichnet. Ein *top-level window* besteht in der Regel aus mehreren untergeordneten Fenstern, den sogenannten *subwindows*. So stellt im Grunde jedes Widget der Gtk+ Bibliothek ein eigenes Fenster dar. Alle Fenster stehen also in einer hierarchischen Beziehung zueinander. Die *top-level windows* werden einem Wurzelfenster (*root window*) zugewiesen, welches durch den X Server selbst angelegt wird. Das *root window* entspricht der Größe des Bildschirms. Um ein einfaches Bildschirmfoto zu erstellen, sollte es also genügen, an den Inhalt des Wurzelfensters zu gelangen. Wir ergänzen unsere Subroutine `button_callback` also um die folgenden Zeilen:

```
my $root =
   Gtk2::Gdk->get_default_root_window;
my ( $x, $y, $w, $h ) =
   $root->get_geometry;
my $pixbuf =
   Gtk2::Gdk::Pixbuf->get_from_drawable(
     $root, undef, $x, $y, 0, 0, $w, $h
   );
```

`Gtk2::Gdk->get_default_root_window;` liefert ein Objekt vom Typ `Gtk2::Gdk::Window` zurück, welches das Wurzelfenster repräsentiert. Mit `$root->get_geometry;` können Ursprung (x- und y-Koordinate) und Abmessung (Breite und Höhe) abgefragt werden. Die Bibliothek bzw. das Modul `Gtk2::Gdk::Pixbuf` stellt verschiedene Funktionen bereit, um Rastergrafiken zu verwalten und zu manipulieren. Beispielsweise könnte man eine lokale Bilddatei mit dem Aufruf `$pixbuf = Gtk2::Gdk::Pixbuf->new_from_file($filename)` laden und innerhalb der Applikation verwenden, z.B. um diese Datei in einem entsprechenden Widget (`Gtk2::Image`) anzuzeigen. Es besteht allerdings auch die Möglichkeit, direkt den Bildinhalt eines *Drawables* in einer Instanz vom Typ `Gtk2::Gdk::Pixbuf` zu speichern. Ein *Drawable* ist, wie der Name vermuten lässt, ein Objekt, auf welches per GDK gezeichnet werden kann, z.B. ein Objekt vom Typ `Gtk2::Gdk::Window`. Der Aufruf `Gtk2::Gdk::Pixbuf->get_from_drawable` eignet sich also hervorragend, um den Bildinhalt des Wurzelfensters abzufragen.

Der Inhalt befindet sich nun in der Variable `$pixbuf` und könnte wie folgt in eine lokale Datei abgespeichert werden:

```
$pixbuf->save($filename, 'png');
```

Um die Aufnahme direkt im Anwendungsfenster anzuzeigen, verwenden wir das Widget `Gtk2::Image`. `$image = Gtk2::Image->new` erstellt das neue Widget, welches anschließend mittels `pack_start_defaults($image)` der `Gtk2::VBox` zugewiesen werden kann. Ergänzt man nun die Subroutine `button_callback` um die folgende Zeile, so wird der Bildschirminhalt direkt im obigen Widget angezeigt:

```
$image->set_from_pixbuf($pixbuf);
#Will man das Bild direkt skalieren,
#so genügt folgendes
#$image->set_from_pixbuf($pixbuf->scale_
#simple(640, 480, 'bilinear'));
```

Aufnahme eines bestimmten Bereiches – Vorbereitungen

Die Aufnahme des gesamten Desktops gehört zum Standardrepertoire einer jeden Anwendung, welche die Erstellung von Bildschirmfotos zum Ziel hat. Oft benötigt man als Anwender aber gar nicht den gesamten Bildschirminhalt, sondern nur einen bestimmten Bereich. Das Bildschirmfoto müsste also erst entsprechend zugeschnitten werden, nachdem man die Aufnahme gemacht hat. Schöner wäre es, wenn man den Bereich also selbst auswählen könnte. Der aufmerksame Leser wird bereits festgestellt haben, dass der Aufruf `Gtk2::Gdk::Pixbuf->get_from_drawable` bereits die entsprechenden Parameter (x, y, Breite, Höhe) zur Verfügung stellt. Es sollte also genügen diese Parameter mit Benutzereingaben zu füllen, um zum Ziel zu kommen. Wie aber sollte der Benutzer die Eingaben vornehmen? Sicherlich könnte man vier Eingabefelder zur Verfügung stellen, die diese Werte entgegennehmen, aber diese Vorgehensweise würde nur in den seltensten Fällen zu einem zufriedenstellenden Ergebnis führen, da der Anwender die Abmessungen selbst abschätzen müsste. Intuitiver wäre hier die Auswahl eines

Bereiches mit der Maus - so wie man es aus vielen anderen Anwendungen bereits gewohnt ist.

Um dies zu implementieren, müssen wir zunächst eine Möglichkeit finden, die aktuelle Position der Maus zu bestimmen. Eine besondere Schwierigkeit ist hier, dass gewöhnlicherweise nur die Events der eigenen Anwendung abgefangen werden können. Auch Tastatureingaben werden nur entgegengenommen, wenn das eigene Fenster den Fokus hat.

Eine Lösung könnte aussehen, wie in Listing 2 dargestellt.

`Gtk2::Gdk->pointer_grab` und `Gtk2::Gdk->keyboard_grab` ermöglichen es, die volle Kontrolle über das Zeigegerät, üblicherweise die Maus, bzw. die Tastatur zu übernehmen. Alle Events werden nun an das Fenster weitergeleitet, welches man durch den ersten Parameter festgelegt hat - in diesem Fall das Wurzelfenster. Des Weiteren besteht die Möglichkeit, die Events zu filtern (*event mask*). Wir interessieren uns vorerst nur für die Bewegung des Mauszeigers und setzen an dieser Stelle `pointer-motion-mask` ein. Mit `Gtk2::Gdk->pointer_is_grabbed` kann anschließend abgefragt werden, ob diese Aktion erfolgreich war oder nicht. Sollte der Zeiger bereits durch eine andere Applikation blockiert sein, so muss dieser erst wieder freigegeben werden. Die eingehenden Ereignisse verknüpfen wir mittels `Gtk2::Gdk::Event->handler_set` mit einer anonymen Subroutine, in der wir die einzelnen Ereignisse abfragen. Der Übergabeparameter `$event` liefert ein Objekt vom Typ `Gtk2::Gdk::Event` zurück. Mit `$event->type` kann nun der Typ des Events abgefragt werden. Die einzelnen Typen verfügen über spezifische Methoden - zum Beipiel stellt das Ereignis vom Typ `Gtk2::Gdk::Event::Motion` die Funktionen `$event->x` und `$event->y` zur Verfügung, welche die Maus-Koordinaten zum Zeitpunkt des Ereignisses beinhalten. Andere Fenster, die sich aktuell auf dem Desktop befinden, werden nun nicht mehr über Tastatur- und Mauseingaben informiert. Um das Programm beenden zu können, fangen wir zusätzlich die Ereignisse vom Typ `Gtk2::Gdk::Event::Key` ab. Mit `$event->keyval == $Gtk2::Gdk::Keysyms{Escape}` stellen wir fest, ob die Escape-Taste gedrückt wurde. Anschließend machen wir `Gtk2::Gdk->pointer_grab` und `Gtk2::Gdk->keyboard_grab` rückgängig. `Gtk2::Gdk::Event->handler_set( undef, undef );` deaktiviert zusätzlich den Ereignishandler, so dass unsere Anwendung wieder wie gewohnt reagiert.

```
my $root = Gtk2::Gdk->get_default_root_window;

Gtk2::Gdk->pointer_grab( $root, 0,
   [qw/pointer-motion-mask/],
   undef, undef, Gtk2->get_current_event_time
);

Gtk2::Gdk->keyboard_grab( $root, 1, Gtk2->get_current_event_time );

if ( Gtk2::Gdk->pointer_is_grabbed ) {

   Gtk2::Gdk::Event->handler_set(sub {
      my $event = shift;
      return 0 unless defined $event;

      if ( $event->type eq 'key-press' ) {
         if ( $event->keyval == $Gtk2::Gdk::Keysyms{Escape} ) {
            Gtk2::Gdk->pointer_ungrab( Gtk2->get_current_event_time );
            Gtk2::Gdk->keyboard_ungrab( Gtk2->get_current_event_time );
            Gtk2::Gdk::Event->handler_set( undef, undef );
         }
      } elsif ( $event->type eq 'motion-notify' ) {
         print "X:",$event->x," Y:",$event->y,"\n";
      }

   });

}
```

Listing 2

Aufnahme eines bestimmten Bereiches – Zeichnen mit GDK

Wir sind nun also in der Lage, die Mausbewegungen des Anwenders zu verfolgen und die aktuellen Koordinaten auf STDOUT auszugeben. Damit haben wir die Grundlage für die Auswahl eines rechteckigen Bereiches geschaffen. In diesem Absatz erweitern wir die obige Funktionalität, indem wir das Zeichnen des Rechteckes implementieren. Der Anwender soll mit gedrückter linker Maustaste ein Rechteck aufziehen können - lässt er die Taste wieder los, so beenden wir die Funktion wieder und nehmen den gewählten Bereich in Form eines Bildschirmfotos auf. In einem ersten Schritt ergänzen wir also den Aufruf von `Gtk2::Gdk->pointer_grab` durch die entsprechenden Ereignisfilter.

```
Gtk2::Gdk->pointer_grab( $root, 0,
   [qw/
      pointer-motion-mask
      button-press-mask
      button-release-mask
   /],
   undef, undef, Gtk2->
                    get_current_event_time
);
```

`button-press-mask` steht hier für das Drücken einer Maustaste und `button-release-mask` für das Loslassen der Maustaste. Anschließend ergänzen wir `Gtk2::Gdk::Event->handler_set`, indem wir die entsprechenden Ereignisse (`$event->type eq 'button-press'` und `$event->type eq 'button-release'`) behandeln.

Betrachten wir nun den vollständigen Quelltext in Listing 3. Zunächst werden einige Variablen deklariert. In der Variable `$btn_pressed` halten wir den Status der linken Maustaste fest - wird diese betätigt, so nimmt die Variable den Wert `1` an. Dieser Zeitpunkt markiert den Beginn des Rechteckes. Anschließend legen wir die Variablen für die Startkoordinaten und die Abmessungend es Rechteckes fest (`$rx, $ry, $rect_x, $rect_y, $rect_w, $rect_h`). Die Zeichenfunktionen werden durch das Modul `Gtk2::Gdk::Drawable` zur Verfügung gestellt (in unserem Fall verwenden wir `draw_rectangle`). Da `Gtk2::Gdk::Window` vom Typ `Gtk2::Gdk::Drawable` abgeleitet ist, können wir diese Funktion direkt mit der Objektinstanz des Wurzelfensters aufrufen (`$root->draw_rectangle($gc, 0, $rect_x, $rect_y, $rect_w, $rect_h)`).

In Form des ersten Parameters verlangt der Aufruf nach einem *graphics context* (GC). Dieser bündelt alle Informationen, die zum Zeichnen benötigt werden. `$gc->set_line_attributes` bietet u.a. die Möglichkeit, die Breite der Linie (1 Pixel) und den Linienstil (gestrichelt) zu beeinflussen. Die Vordergrundfarbe wird mittels `$gc->set_rgb_fg_color` auf die Farbe Weiß gesetzt. In unserem Fall ist die Wahl des korrekten *GdkSubwindowModes'* besonders wichtig. Dieser legt fest, ob sich die Zeichnung auf einem Fenster bzw. einem Objekt vom Typ *Drawable* auch auf die Unterfenster (*subwindows*) auswirkt. Dieser Modus kann auf die Werte `'clip-by-children'` oder `'include-inferiors'` gesetzt werden. Da wir auf das Wurzelfenster zeichnen wollen und gewährleistet sein muss, dass unsere Zeichnung (das Rechteck) immer sichtbar ist, wählen wir `'include-inferiors'`, d.h. `$gc->set_subwindow('include-inferiors');`. Einen kleinen Trick wenden wir nun durch die Wahl des Zeichenmodus' an. Dieser legt fest, wie die Pixel des Wurzelfensters (`src`) mit den zu zeichnenden Pixeln (`dest`) verbunden werden. Mit `$gc->set_function('xor');` wird diesbezüglich folgende Regel angewendet: `dst = src XOR dst`. Dies bringt den Vorteil

```
if ( Gtk2::Gdk->pointer_is_grabbed ) {

   #Hilfsvariable, um den Status der Maustaste zu bestimmen
   my $btn_pressed = 0;

   #Startkoordinaten und Abmessungen des Rechtecks
   my ( $rx, $ry, $rect_x, $rect_y, $rect_w, $rect_h ) = ( 0, 0, 0, 0, 0, 0 );

   #GC definieren
   my $gc = Gtk2::Gdk::GC->new( $root, undef );
   $gc->set_line_attributes( 1, 'double-dash', 'butt', 'round' );
   $gc->set_rgb_fg_color(Gtk2::Gdk::Color->new( 65535, 65535, 65535 ));
   $gc->set_subwindow('include-inferiors');
   $gc->set_function('xor');
```

Listing 3

mit sich, dass wir das gezeichnete Rechteck wieder entfernen können, indem wir die Funktion (`draw_rectangle`) mit identischen Parametern erneut aufrufen.

Kommen wir nun also zur Ablauflogik. Wird die linke Maustaste gedrückt, so wird ein Ereignis vom Typ `'button-press'` ausgelöst. Diese Tatsache halten wir in der Variable

```
    Gtk2::Gdk::Event->handler_set(sub {
       my $event = shift;

       if ( $event->type eq 'key-press' ) {
          #ESC-Taste wird gedrückt => beenden
          if ( $event->keyval == $Gtk2::Gdk::Keysyms{Escape} ) {
             Gtk2::Gdk->pointer_ungrab( Gtk2->get_current_event_time );
             Gtk2::Gdk->keyboard_ungrab( Gtk2->get_current_event_time );
             Gtk2::Gdk::Event->handler_set( undef, undef );
          }
       } elsif ( $event->type eq 'motion-notify' ) {
          #Wenn Maustaste gedrückt, dann wird das Rechteck gezeichnet
          if ($btn_pressed) {
             #Das letzte gezeichnete Rechteck wieder entfernen
             #siehe $gc->set_function('xor');
             if ( $rect_w > 0 ) {
                $root->draw_rectangle( $gc, 0, $rect_x, $rect_y, $rect_w, $rect_h );
             }

             #Abmessungen des neuen Rechtecks bestimmen
             $rect_x = $rx;
             $rect_y = $ry;
             $rect_w = $event->x - $rect_x;
             $rect_h = $event->y - $rect_y;
             if ( $rect_w < 0 ) {
                $rect_x += $rect_w;
                $rect_w = -$rect_w;
             }
             if ( $rect_h < 0 ) {
                $rect_y += $rect_h;
                $rect_h = -$rect_h;
             }

             #Rechteck zeichnen
             if ( $rect_w != 0 ) {
                 $root->draw_rectangle( $gc, 0, $rect_x, $rect_y, $rect_w, $rect_h );
             }
          }
       } elsif ( $event->type eq 'button-press' ) {
          #Maustaste wurde gedrückt
          $btn_pressed = 1;
          #Koordinaten festhalten
          $rx = $event->x;
          $ry = $event->y;
       } elsif ( $event->type eq 'button-release' ) {
          if ( $rect_w > 1 ) {
             #Das letzte gezeichnete Rechteck wieder entfernen
             #siehe $gc->set_function('xor');
             $root->draw_rectangle( $gc, 0, $rect_x, $rect_y, $rect_w, $rect_h );

             Gtk2::Gdk->pointer_ungrab( Gtk2->get_current_event_time );
             Gtk2::Gdk->keyboard_ungrab( Gtk2->get_current_event_time );
             Gtk2::Gdk::Event->handler_set( undef, undef );

             my $pixbuf = Gtk2::Gdk::Pixbuf->get_from_drawable(
                           $root, undef, $rect_x, $rect_y, 0, 0, $rect_w, $rect_h
                         );
             $image->set_from_pixbuf($pixbuf);
          }
       }

    });
}
```

Listing 3 (Fortsetzung)

`$btn_pressed` fest und setzen `$rx` und `$ry` auf die x- bzw. y-Koordinate des Ereignisses. Bewegt der Anwender, bei gedrückter linker Maustaste, nun den Cursor über den Bildschirm, so werden fortwährend Ereignisse vom Typ `'motion-notify'` ausgelöst. Hier werden jetzt, gemäß den aktuellen Koordinaten des Ereignisses, die Werte für das zu zeichnende Rechteck bestimmt (`$rect_x`, `$rect_y`, `$rect_w`, `$rect_h`) und die Zeichenoperation durchgeführt.

```
root->draw_rectangle(
 $gc, 0, $rect_x, $rect_y, $rect_w, $rect_h)
```

Das "alte" Rechteck wird gegebenenfalls zuvor wieder entfernt (Bedingung `if ( $rect_w > 0 )`). So wird sichergestellt, dass sich immer nur ein Rechteck auf dem Bildschirm zu sehen ist, welches die aktuelle Auswahl repräsentiert. Wird die linke Maustaste losgelassen, so tritt das Ereignis `'button-release'` ein. Erneut wird das letzte gezeichnete Rechteck entfernt, Maus und Tastatur werden wieder freigegeben und der Ereignishandler wird deaktiviert. Anschließend wird, wie bereits im Kapitel *Der Weg vom Fensser zum Bildschirmfoto* geschildert, das Bildschirmfoto aufgenommen.

Analog zur Subroutine `button_callback` könnte man den obigen Code nun in das Beispielprogramm integrieren.

Dokumentationen und Tutorials

Wer nun Lust bekommen hat, dem empfehle ich, die offizielle Gtk2-Perl-Seite unter *http://gtk2-perl.sourceforge.net/doc/* zu besuchen. Neben der vollständigen API-Dokumentation finden sich hier auch einige Links zu Tutorials in unterschiedlichen Sprachen. Wer seine Fragen lieber direkt an die Entwickler und die aktive Community richten möchte, der kann unter *http://mail.gnome.org/mailman/listinfo/gtk-perl-list* der Mailingliste des Projektes beitreten.

Ausblick

Obwohl Shutter nun bereits seit zwei Jahren in Entwicklung ist, sind die Arbeiten noch lange nicht abgeschlossen. Vor allem aus der Community werden kontinuierlich neue Ideen und Feature-Wünsche an das Projekt herangetragen. Zusätzlich sind wir auch stets bemüht, Shutter an neue Technologien anzupassen. Aktuell arbeite ich an einer Integration in Ubuntu One, einem Cloud Computing-Dienst der Firma Canonical. Wer Lust und Zeit hat, sich an unserem Projekt zu beteiligen, ist jederzeit herzlich willkommen. Die offizielle Projektseite ist unter *http://shutter-project.org* zu erreichen.

Mario Kemper

Regex Debugging

Debugging von Regulären Ausdrücken

In Ausgabe 7 hat Thomas Fahle schon eine Einführung in den Debugger für Perl-Programme gegeben – an dieser Stelle nun eine Einführung in das Debugging von Regulären Ausdrücken. Ein paar hilfreiche Module in Sachen "Reguläre Ausdrücke" wurden bereits in Ausgabe 4 vorgestellt.

Eine Sache die man nicht gerne macht, ist das Debuggen von Regulären Ausdrücken. Bei einfachen RegEx lässt sich das noch gut "per Hand" machen, aber sobald es etwas komplexer wird, ist man verloren.

Die RegEx-Engine von Perl hat einen Debug-Modus, mit dem man sich anschauen kann, was bei einem Regulären Ausdruck für einen Eingabestring passiert.

Allerdings muss man sagen, dass Regex-Debugging nicht die einfachste Aufgabe ist. Den Debugger kann man mit `-Mre=debug` im Aufruf des Perl-Interpreters anschalten:

```
perl -Mre=debug -e '"foobar" =~ /regex/'
```

Fangen wir ganz einfach an - Listing 1.

Gehen wir das ganze Schritt für Schritt durch:

Als erstes wird ein alter Reguläre Ausdruck verworfen, danach wird der zu testende Reguläre Ausdruck kompiliert. Die Regex-Engine ist sehr komplex. Für Perl 5.10 wurde die komplette Regex-Engine neu geschrieben und das Konzept dahinter wurde umgeworfen. Damit sind jetzt auch komplexere Reguläre Ausdrücke schneller.

```
'foobar' =~ /(.)b/; # ein beliebiges Zeichen vor 'b'

Freeing REx: `","'
Compiling REx `(.)b'
size 8 Got 68 bytes for offset annotations.
first at 3
   1: OPEN1(3)
   3:   REG_ANY(4)
   4: CLOSE1(6)
   6: EXACT <b>(8)
   8: END(0)
anchored "b" at 1 (checking anchored) minlen 2
Offsets: [8]
        1[1] 0[0] 2[1] 3[1] 0[0] 4[1] 0[0] 5[0]
Guessing start of match, REx "(.)b" against "foobar"...
Found anchored substr "b" at offset 3...
Starting position does not contradict /^/m...
Guessed: match at offset 2
Matching REx "(.)b" against "obar"
  Setting an EVAL scope, savestack=3
   2 <fo> <obar>          |  1:  OPEN1
   2 <fo> <obar>          |  3:  REG_ANY
   3 <foo> <bar>          |  4:  CLOSE1
   3 <foo> <bar>          |  6:  EXACT <b>
   4 <foob> <ar>          |  8:  END
Match successful!
Freeing REx: `"(.)b"'
```

Listing 1

Die Größenange zeigt, wie groß die kompilierte Form des Regulären Ausdrucks ist. In der Regel ist die Einheit 4-byte-Worte. Danach wird die Größe der Offset/Längen-Tabelle für den Regex ausgegeben. Auf diese Tabelle komme ich gleich noch einmal zu sprechen.

`first at 3` gibt an, welcher Knoten des Regulären Ausdrucks der erste Knoten ist, der einen Match versucht. Die Knoten werden dann in den Zeilen dargestellt:

```
1: OPEN1(3)
3:   REG_ANY(4)
4: CLOSE1(6)
6: EXACT <b>(8)
8: END(0)
```

Der Aufbau ist wie folgt:

```
ID: TYP(NÄCHSTER_KNOTEN)
```

Ich werde nicht auf alle Typen eingehen, die es gibt. Die sind in der Dokumentation `perldebguts` beschrieben. Auf das Beispielprogramm werde ich dennoch eingehen. `OPEN#` zeigt an, dass hier eine Gruppierung vorliegt, die die Treffer auch speichert. Die Nummer gibt an, die wievielte Gruppierung das ist. Würde eine "non-capturing group" verwendet werden, also `(?:.)` würde das `OPEN1` und das `CLOSE1` komplett wegfallen.

In der Klammer nach dem Typ wird die ID des Knotens gespeichert, zu dem gesprungen werden soll, wenn der Ursprungsknoten matcht. In diesem Beispiel ist das immer der nächste Knoten. Ich werde nachher noch ein Beispiel zeigen, bei dem das nicht der Fall ist.

`REG_ANY` bedeutet, dass ein beliebiges Zeichen gematcht werden soll. Nach dem Typ `EXACT` wird noch angegeben, welcher String literal gematcht werden soll.

Nach der Darstellung des kompilierten Programms kommen ein paar Angaben zu Optimierungen und Heuristiken - Listing 2.

In diesem Fall weiß die Regex-Engine, dass es ein literales `b` in dem String mit dem Offset 1 geben muss und dass der zu durchsuchende String mindestens 2 Zeichen lang sein muss.

Danach wird die Offset/Längen-Tabelle angezeigt. Zuerst wird die Anzahl der Element in der Tabelle angegeben; hier sind es 8 Elemente. Dann folgt der Inhalt der Tabelle. Die Zählung der Elemente fängt bei 1 an und der Index ist gleich der ID der Knoten im kompilierten Programm. Die Elemente werden in der Schreibweise `offset[länge]` notiert.

Ist ein Element `0[0]`, dann existiert kein Knoten dafür. Wir haben in der kompilierten Form gesehen, dass es keinen Knoten mit der ID 2 gibt, also ist das zweite Element der Tabelle ein `0[0]`.

Das erste Element (hier: `1[1]`) repräsentiert also den Knoten mit der ID 1 (`OPEN1`). Der Offset gibt an, an welcher Stelle im originalen (nicht-kompilierten) Regulären Ausdruck dieser Knoten beginnt und die Länge gibt an, wieviele Zeichen der Knoten repräsentiert. Ich denke, es ist klar ersichtlich, dass das "(" am Anfang des Regulären Ausdrucks (`(.)b`) steht und 1 Zeichen lang ist.

Nach dieser Tabelle werden hier noch angezeigt, dass die Regex-Engine versucht zu raten, wo der Reguläre Ausdruck in dem zu durchsuchenden String anfängt. Hier findet die Engine heraus, dass "b" der vierte Buchstabe des Strings ist ("offset 3") und rät, dass der Treffer beim dritten Buchstaben anfängt ("offset 2").

Bis zu dieser Stelle waren Ausgaben, die de Kompile-Zeit des Regulären Ausdrucks betreffen.

Ab jetzt folgen Ausgaben, die die Laufzeit des Regulären Ausdrucks betreffen. Sollte so eine Ausgabe fehlen, bedeutet das, dass der Reguläre Ausdruck erst gar nicht ausgeführt wird, z.B. weil der String zu kurz ist oder ein fester Substring erst gar nicht im String enthalten ist (z.B. bei `'hallo' =~ /es/`.

```
anchored "b" at 1 (checking anchored) minlen 2
Offsets: [8]
        1[1] 0[0] 2[1] 3[1] 0[0] 4[1] 0[0] 5[0]
Guessing start of match, REx "(.)b" against "foobar"...
Found anchored substr "b" at offset 3...
Starting position does not contradict /^/m...
Guessed: match at offset 2
```

Listing 2

Aber zurück zum Beispiel:

```
Matching REx "(.)b" against "obar"
  Setting an EVAL scope, savestack=3
   2 <fo> <obar>          |  1:  OPEN1
   2 <fo> <obar>          |  3:  REG_ANY
   3 <foo> <bar>          |  4:  CLOSE1
   3 <foo> <bar>          |  6:  EXACT <b>
   4 <foob> <ar>          |  8:  END
```

Es wird angezeigt, gegen welchen String der Regex ausgeführt wird. Hier sieht man auch, welche Auswirkungen das Raten der Engine hat. Es wird nämlich nicht "foobar" sondern "obar" genommen. Das spart Zeit.

Danach sieht man den Ablauf des Regulären Ausdrucks. Die Zeilen sind wie folgt aufgebaut:

```
STRING_OFFSET <PRE-STRING> <POST-STRING>
                                  | ID: TYP
```

`STRING_OFFSET` ist die Stelle im zu durchsuchenden String, `PRE-STRING` der schon abgearbeitete Teil des Strings und `POST-STRING` der noch zu durchsuchende Teil. Danach kommt die Angabe, welcher Knoten aus dem kompilierten Regulären Ausdrucks an dieser Stelle genommen wird.

Backtracking

Backtracking ist ein Thema, das nicht wirklich leicht ist und man kann leicht in die Backtracking-Falle tappen. Es ist am einfachsten, Backtracking an einem Beispiel zu zeigen. Dazu verwenden wir wieder das Debugging für Reguläre Ausdrücke.

Was ist Backtracking?

Am besten kann man Backtracking mit einem Bildlichen Vergleich beschreiben: Man stelle sich eine Schnitzeljagd durch die Gegend vor. Es geht von einem Punkt zum nächsten. Es gibt eine Beschreibung, wie es nach jedem Punkt weiter geht. Ein Regulärer Ausdruck ist diese Beschreibung. Bei machen Zwischenpunkten auf der Schnitzeljagd wird es nur eine Möglichkeit geben, wie man zu dem nächsten Zwischenziel kommt, bei anderen wird es verschiedene Möglichkeiten geben. Ziel ist es natürlich, bis ans Ende zu kommen. Befindet man sich an einem Zwischenziel mit mehreren Möglichkeiten, muss man sich erst für eine Möglichkeit entscheiden. Diesen Weg geht man dann. Unterwegs stellt man fest, dass man so nicht mehr weiter kommt. Also geht man zurück bis zum letzten erfolgreichen Zwischenziel und versucht die nächste Möglichkeit. Das nennt man Backtracking. Wenn man bei einem Zwischenziel alle Möglichkeiten ausgeschöpft hat und nicht mehr weiterkommt, muss man eventuell auch zwei oder mehr Zwischenschritte zurückgehen. Das kann Zeit kosten.

Als Beispiel

```
  "abcde" =~ /(abd|abc)(df|d|de)/;
```

1. Beginne mit dem ersten Buchstaben im String ('a')

2. Probiere die erste Alternative in der ersten Gruppe ('abd')

3. Matche 'a' gefolgt von einem 'b'. Soweit so gut.

4. Das 'd' im Regulärer Ausdruck matcht nicht das 'c' im String - eine Sackgasse. Also gehe zwei Buchstaben zurück (Backtracking) und probiere die zweite Alternative in der ersten Gruppe ('abc').

5. Matche 'a' gefolgt von einem 'b' gefolgt von einem 'c'. Wir sind im Plan und haben die erste Gruppe erledigt. Setze `$1` auf 'abc'.

6. Gehe zur zweiten Gruppe und probiere die erste Alternative ('df').

7. Matche das 'd'.

8. Das 'f' im Regulären Ausdruck match nicht das 'e' im String, also eine Sackgasse. Gehe einen Buchstaben zurück (Backtracking) und probiere die zweite Alternative in der zweiten Gruppe ('d').

9. Das 'd' matcht. Die zweite Gruppe ist erfolgreich, also setze `$2` auf 'd'.

10. Wir sind am Ende des Regulären Ausdrucks, also sind wir fertig! Wir haben 'abcd' im String 'abcde' gematcht.

Das Beispiel in Listing 3 zeigt nochmal in der Debug-Ausgabe, wobei die Regex-Engine dabei schon einiges optimiert und Strings zusammenfasst:

```
C:\>perl -Mre=debug -e "'abcde' =~ /(abd|abc)(df|d|de)/;"
Freeing REx: `","'
Compiling REx `(abd|abc)(df|d|de)'
size 24 Got 196 bytes for offset annotations.
first at 3
   1: OPEN1(3)
   3:   BRANCH(6)
   4:     EXACT <abd>(9)
   6:   BRANCH(9)
   7:     EXACT <abc>(9)
   9: CLOSE1(11)
  11: OPEN2(13)
  13:   BRANCH(16)
  14:     EXACT <df>(22)
  16:   BRANCH(19)
  17:     EXACT <d>(22)
  19:   BRANCH(22)
  20:     EXACT <de>(22)
  22: CLOSE2(24)
  24: END(0)
minlen 4
Offsets: [24]
        1[1] 0[0] 1[1] 2[3] 0[0] 5[1] 6[3] 0[0] 9[1] 0[0] 10[1] 0[0] 10[1] 11[2]
 0[0] 13[1] 14[1] 0[0] 15[1] 16[2] 0[0] 18[1] 0[0] 19[0]
Matching REx "(abd|abc)(df|d|de)" against "abcde"
  Setting an EVAL scope, savestack=3
   0 <> <abcde>           |  1:  OPEN1
   0 <> <abcde>           |  3:  BRANCH
  Setting an EVAL scope, savestack=14
   0 <> <abcde>           |  4:    EXACT <abd>
                              failed...
   0 <> <abcde>           |  7:    EXACT <abc>
   3 <abc> <de>           |  9:    CLOSE1
   3 <abc> <de>           | 11:    OPEN2
   3 <abc> <de>           | 13:    BRANCH
  Setting an EVAL scope, savestack=25
   3 <abc> <de>           | 14:      EXACT <df>
                                 failed...
   3 <abc> <de>           | 17:      EXACT <d>
   4 <abcd> <e>           | 22:      CLOSE2
   4 <abcd> <e>           | 24:      END
Match successful!
Freeing REx: `"(abd|abc)(df|d|de)"'
```

Listing 3

Hier sieht man schon, dass die Regex-Engine viel arbeiten muss. In diesem Beispiel war es noch ein relativ einfacher Regulärer Ausdruck.

Renée Bäcker

AI::CBR

Einführung

In diesem Artikel möchte ich einen Einblick in die Methode des Fallbasierten Schließens geben, im Englischen *Case-Based Reasoning* oder kurz CBR genannt. CBR ist eine Methode aus dem Gebiet der Künstlichen Inteligenz, welche zur ähnlichkeitsbasierten Suche, und weitergehend auch zur ähnlichkeitsbasierten Problemlösung verwendet werden kann.

Die theoretischen Grundlagen dieser Methode wurden Ende der 70er Jahre von Roger Schank gelegt, wissenschaftlich und technisch wurde CBR dann im Laufe der 90er sowohl in den USA als auch in Deutschland entwickelt, und ist auch heute noch ein wissenschaftlich sehr aktives Thema. Eine zentrale Rolle spielt hier der Begriff *Fall*, welcher ein neues Poblem, eine Situation oder eine Objektbeschreibung sein kann. Im Folgenden verwende ich hier immer nur den allgemeinen Begriff *Fall*.

Im Kern imitiert CBR die menschliche Fähigkeit, intuitiv Ähnlichkeiten zwischen Objekten oder Situationen erkennen zu können, und dies zum Transfer von früheren Erfahrungen auf neue Fälle verwenden zu können. Es gibt zahllose Beispiele von Ärzten, Ingenieuren und Support-Mitarbeitern, welche dank ihres Erfahrungsschatzes in der Lage sind, sich bei neuen Fällen an frühere ähnliche Fälle und deren erfolgreiche Lösung zu erinnern. Dies kann man dann nutzen, um den neuen Fall auf ähnliche Weise zu lösen.

Eine weitere Möglichkeit, auf welche ich in diesem Artikel hauptsächlich eingehen werde, ist die reine ähnlichkeitsbasierte Suche. Mittels dieser ist es möglich, einen Benutzer bei der Suche nach einem Produkt seine exakten Präferenzen angeben zu lassen, und ihm dann alle dazu ähnlichen Ergebnisse in sinnvoller Sortierung anzuzeigen.

Dieses CBR-Prinzip wurde in einem sehr generischen Framework für Perl umgesetzt, und ist unter dem Paketnamen `AI::CBR` auf CPAN zu finden. Dieser Artikel beschreibt, wie man mit Hilfe dieses Frameworks sehr einfach CBR-basierte Anwendungen entwickeln kann.

Der CBR-Zyklus

Ein CBR-System behandelt neue Fälle in mehreren Schritten.

1. Neuer Fall
Die Spezifikation eines neuen Falls liegt vor. Diese kann sowohl eine Problembeschreibung als auch die Beschreibung eines Wunschprodukts sein.

2. Retrieval
Die Gesamtmenge bisher bekannter und schon gelöster Fälle, die sogenannte *Fallbasis*, wird durchsucht, und die Ähnlichkeiten zum aktuellen Fall werden berechnet. Ergebnis des Retrievals ist eine sortierte Liste der ähnlichsten bekannten Fälle.

3. Reuse
Beim *Reuse*, zu Deutsch Wiederverwendung, werden der ähnlichste Fall oder die ähnlichsten Fälle zur Wiederverwendung herangezogen. Ob man hierfür einfach die Lösung des ähnlichsten Falls, oder eine Synthese aus den k ähnlichsten Fällen heranzieht, ist eine Frage der Systemanforderungen und der Implementierung. Im Falle der ähnlichkeitsbasierten Suche sind die ähnlichsten Treffer schon das Ergebnis, und der Prozess ist hier am Ende.

4. Adapt, Revise & Retain
Im Falle eines CBR-Systems zu Problemlösungszwecken werden die Lösungen der bekannten Fälle noch für einige weitere Schritte verwendet.

Zuerst muss die Lösung des bekannten Falls auf das neue Problem adaptiert werden (Adapt). Nach der Anwendung dieser adaptierten Lösung kann man diese entsprechend des Erfolgs oder Misserfolgs nochmal revidieren (Revise), und anschließend den neuen Fall zusammen mit der revidierten Lösung dann in der Fallbasis abspeichern (Retain). Dies realisiert dann ein lernendes System.

Hierauf wird dieser Artikel jedoch nicht weiter eingehen, da solche Systeme für produktive Umgebungen nur mit großem Aufwand realisiert werden können, und das `AI::CBR` Framework (noch) keine direkte Unterstützung für diese weiteren Schritte bietet.

Die Ausgangssituation

Der erste Schritt zur Entwicklung eines neuen CBR-Systems ist die Spezifikation der auftretenden Fälle. Diese sind sehr ähnlich zu Objekten, beinhalten jedoch über die Attribute hinaus noch Ähnlichkeitsfunktionen und -gewichte für die Attribute des Objekts.

Betrachten wir zuallererst einmal den Fall eines Patienten beim Arzt, mit einigen relevanten Attributen in Hash-Notation:

```
# Patient A beim Arzt
{
  age      => 40,
  gender   => 'male',
  job      => 'programmer',
  symptoms => ['headache', 'cough'],
}
```

Nun stellt sich die Frage, ob der Arzt allein auf Basis seiner bisherigen Erfahrungen eine oder mehrere mögliche Diagnosen vermuten könnte. Wichtigstes Werkzeug hierfür, und Kern jeder CBR-Anwendung, ist die Ähnlichkeitsberechnung zwischen Fällen.

Ähnlichkeitsberechnung

Die Ähnlichkeit zwischen zwei Objekten wird über den Durchschnitt der Ähnlichkeiten ihrer Attribute berechnet. Folglich sind die Ähnlichkeiten auf Attributebene das Kernstück der Ähnlichkeitsberechnung. Diese werden jeweils über eine Ähnlichkeitsfunktion bestimmt, welche abhängig vom Datentyp und von der Interpretation des Attributs ausgewählt werden muss.

Ähnlichkeitsfunktionen

Die Ähnlichkeit wird im Wertebereich von *0* für *grundverschieden*, bis *1.0* für *identisch* gemessen. Im `AI::CBR` Framework haben Ähnlichkeitsfunktionen immer mindestens zwei Parameter, nämlich die beiden Werte der zu vergleichenden Attribute. Optional kann es noch einen dritten Parameter für die Ähnlichkeitsfunktion geben, mit welchem sich deren Berechnung individuell steuern lässt.

Das Modul `CRR::AI::Sim` exportiert einige Ähnlichkeitsfunktionen, die häufig geeignet sind. Beispielsweise eine einfache Gleichheitsfunktion für enumerative String-Datentypen wie den Job einer Person:

```
sim_eq('programmer', 'programmer'); # 1.0
sim_eq('programmer', 'manager');    # 0.0
```

Für Zahlenwerte eignet sich oft die Funktion für das relative Verhältnis zweier Zahlen:

```
sim_frac(2, 4);   # 0.5
sim_frac(40, 30); # 0.75
```

Für Listen von Elementen eignet sich eine Funktion, welche die Größe der Schnittmenge durch die Größe der Vereinigungsmenge dividiert:

```
sim_set([qw(a b c)], ['a']);
# 1/3 = 0.33
sim_set([qw(a b c)], [qw(b c d)]);
# 2/4 = 0.5
```

Als Beispiel für eine Ähnlichkeitsfunktion mit einem dritten Parameter stelle ich noch die Ähnlichkeit zweier Zahlen relativ zu einem Maximalabstand, hier im Beispiel *10*, vor:

```
sim_dist(5, 8, 10);  # 3/10 = 0.3
sim_dist(5, 18, 10); # 0.0
```

Fallspezifikation

Ein Fall wird als Objekt der Klasse `AI::CBR::Case` unter Referenzierung der gewünschten Ähnlichkeitsfunktionen spezifiziert, wie das folgende Listing demonstriert:

```
use AI::CBR::Case;
use AI::CBR::Sim qw(
   sim_eq sim_frac sim_set
);
my $new_case = AI::CBR::Case->new(
  age       => { value => 40,
                 sim   => \&sim_frac },
  gender    => { value => 'male',
                 sim   => \&sim_eq   },
  job       => { value => 'programmer',
                 sim   => \&sim_eq   },
  symptoms  => { value =>
                 ['headache', 'cough'],
                 sim   => \&sim_set  },
);
```

Auf Basis dieser Spezifikation kann man nun die Ähnlichkeit zu jedem anderen Patienten exakt berechnen. Schauen wir uns die Attribute eines zweiten Patienten aus der Fallbasis mit bekannter Diagnose an:

```
# Patient B aus Fallbasis
{
  age       => 28,
  gender    => 'male',
  job       => 'programmer',
  symptoms  => ['headache'],
  diagnosis => 'hangover',
}
```

Für die vier Attribute ergeben sich mit den weiter oben im Fall spezifizierten Ähnlichkeitsfunktionen jeweils Ähnlichkeitswerte von *0.7*, *1.0*, *1.0* und *0.5*, was im Durchschnitt eine Ähnlichkeit der Patienten A und B von *3.2 / 4 = 0.8* ergibt.

Retrieval

Um nun an die gesuchten ähnlichsten Fälle aus der Fallbasis zu kommen, benutzen wir das `AI::CBR::Retrieval` Modul. Dieses erwartet neben der Spezifikation des aktuellen Falls noch eine Liste der Objekte aus der Fallbasis, welche zum Beispiel vorher aus einer Datenbank geladen wurden:

```
use AI::CBR::Retrieval;
my $r = AI::CBR::Retrieval->new(
  $new_case, [@case_base]
);
$r->compute_sims();
```

Danach kann man verschiedene Methoden verwenden, um an den ähnlichsten Fall oder die *k* ähnlichsten Fälle zu kommen, zum Beispiel:

```
my $solution = $r->most_similar_candidate();
```

Hier sind auch fortgeschrittenere Retrieval-Algorithmen vorhanden, bzw. es können spezifische Verfahren einfach durch eine Methode in einer abgeleiteten Retrieval-Klasse ergänzt werden.

Intelligente Produktsuche

Wie zu Beginn erwähnt, kann ein CBR-System auch nur zur ähnlichkeitsbasierten Suche verwendet werden, ganz ohne den Problemlösungsaspekt. Hier gibt der Kunde einfach die Beschreibung seines Wunschobjekts an, das System liefert dann die passendsten Angebote zurück.

Gerade im kommerziellen Bereich gibt es hierfür sehr viele Anwendungsgebiete, bei denen heutige Angebote noch große Defizite aufweisen. Beispielsweise die Suche nach Gebrauchtwagen, bei der man meistens umständlich Suchintervalle für die Kilometerleistung oder den Preis angeben muss, obwohl dies durchweg fließende, und damit eigentlich ähnlichkeitsbasierte Suchkriterien sind. Weitere Beispiele sind Hotelsuchen, Suchen nach Reiseangeboten, usw.

Ein CBR-System für die intelligente ähnlichkeitsbasierte Produktsuche verwendet nur die ersten Schritte des CBR-Zyklus bis zum Retrieval. Das Ergebnis des Retrievals, die nach Ähnlichkeit sortierte Liste aus der Fallbasis, ist dann direkt das Suchergebnis.

In diesem Prozess kann man optional auch individuelle Gewichtungen von Attributen oder Festlegungen auf exakte Übereinstimmung von bestimmten Attributen ergänzen.

Ausblick

In diesem Artikel habe ich einen ersten Einblick in CBR-Systeme und eine kurze Einführung in das `AI::CBR` Framework

gegeben. CBR ist seit Jahren ein spannendes Feld der Wissenschaft, und gerade die späteren Schritte des Adaptierens von Problemlösungen und das anschließende Lernen bergen großes Potenzial, sind aber noch nicht reif für die breite Verwendung in Anwendungen.

Die vorgestellte Variante der intelligenten Produktsuche hingegen ist ein solides Werkzeug für viele Szenarien, und das vorgestellte `AI::CBR` Framework ermöglicht es jedem Programmierer nach kurzer Einarbeitung, ein solches System zu entwickeln.

Das Framework wurde bereits extensiv bei der Entwicklung einer Strategiespiel-KI auf *http://www.cosair.org* eingesetzt, wo die KI Entscheidungen von menschlichen Spielern in ähnlichen Situationen imitiert hat. Details hierzu finden sich bei Interesse in meiner Diplomarbeit.

Darko Obradovic

MODULE

Sauberer Namensraum mit namespace::clean

Gerade viele ältere Module auf CPAN und viele Module, die ich vor ein paar Jahren geschrieben habe, exportieren alle Subroutinen - ohne wenn und aber. Objektorientiertes Perl kannte ich zu dem Zeitpunkt noch nicht wirklich. Benutzt man diese Module, hat man gleich sämtliche Subroutinen im Namensraum. Wenn man viele solcher Module verwendet, kann man sich den Namensraum "zumüllen". Und wenn mehrere Module gleichnamige Subroutinen exportieren, kann es zu Konflikten kommen.

Perl Best Practices empfiehlt, Subroutinen nicht standardmäßig zu exportieren, sondern nur auf Verlangen. In Perl::Critic ist dies auch als Regel enthalten. Wenn man viele Subroutinen benötigt, kann das Anfordern der Subroutinen viel Schreibarbeit bedeuten (angedeutet in Listing 1 und 2).

Und setzt man Export-Tags ein, werden unter Umständen zu viele Subroutinen exportiert. So kann man mit `use CGI qw(:standard)` sehr viele Funktionen importieren. Vielleicht möchte man auch alle außer der Funktion `header` auch tatsächlich haben.

Wer die Funktionen aus `CGI` importiert und dann eine eigene Funktion `header` implementieren will, bekommt eine Warnung. Die kann man zwar mit `no warnings 'redefine';` ausschalten, aber das ist ja auch nicht unbedingt Sinn der Sache.

```
package ExportTest;

use base 'Exporter';

our @EXPORT_OK = qw(
    sub1 sub2 sub3 sub4 sub5 sub6 ... sub10
);
```

Listing 1

```
#!/usr/bin/perl

use ExportTest qw(sub1 sub2 sub5 sub7 sub9);

# ... weiterer Code ...
```

Listing 2

Wie kann man das verhindern? Man kann entweder darauf verzichten, Funktionen in den eigenen Namensraum zu importieren - und dann die Vollqualifizierten Funktionsnamen verwenden - oder man muss den Namensraum vor dem "Müll" schützen.

Um den Namensraum sauber zu halten, kann man `namespace::clean` verwenden. Programmierer, die `Moose` einsetzen, werden das Modul vielleicht kennen. Aber auch für alle anderen, kann dieses Modul sehr interessant sein.

Einsatz in den eigenen 4 Wänden

Bleiben wir bei dem Beispiel mit dem CGI-Modul. Nehmen wir an, wir wollen in einem Skript das Modul verwenden und importieren mal die Standard-Funktionen. Damit würden wir jede Menge Funktionen in unseren Namensraum bekommen. Wir brauchen aber nur `header` und `b`.

In diesem Fall können wir mit `namespace::clean` arbeiten - siehe Listing 3.

```
use strict;
use warnings;

{
    use CGI qw(:standard);
    use namespace::clean -except =>
                          ['header','b'];
}

print header, b('test'),p();

sub p {
    return "p\n";
}
```

Listing 3

Das `use CGI ...` und das Einbinden von `namespace::clean` erfolgt hier in einem extra Block. Warum das so ist, werde ich gleich noch erläutern. Zuerst werden die Funktionen von `CGI.pm` importiert und danach wird mittels `namespace::clean` gesagt, dass der Namensraum saubergehalten werden soll, nur die Funktionen `header` und `b` sollen erhalten bleiben.

Wird einfach nur `use namespace::clean;` gesagt, werden alle importierten Funktionsnamen entfernt. Sonst kann man die Liste auf zwei Arten einschränken: Erstens kann man `namespace::clean` sagen, welche Funktionsnamen aus dem Namensraum entfernt werden sollen (`use namespace::clean 'header'` entfernt nur `header`, alles andere bleibt erhalten) oder man gibt - wie in dem Beispielcode gezeigt - an, welche Funktionsnamen nicht entfernt werden sollen.

Die Arbeit nimmt `namespace::clean` dann am Ende des Scopes auf (Hinweis: In den $foo-Ausgaben 11 - 14 gab es ein Tutorial über Scopes von Ferry Bolhár-Nordenkampf). Das ist auch der Grund, warum in dem Beispielcode der extra Block verwendet wurde. `CGI.pm` definiert eine Funktion `p`, die auch mit dem Tag ':standard' importiert wird. Wird der extra Block nicht verwendet, so bekommt man eine Warnung, dass die Funktion "redefined" wird. Der Namensraum wäre in dem ganzen Skript voll mit den Funktionen aus `CGI.pm`. `namespace::clean` würde die Arbeit erst am Ende des Skriptes aufnehmen - also zu spät.

Durch den Extra-Block, werden die Funktionsnamen allerdings schon beim Schließen des Blocks entfernt. Wenn wir unsere eigene Funktion `p` definieren, ist der Name der aus `CGI.pm` importierten Funktion schon entfernt.

```
package KlasseMitDataDumper;

use strict;
use warnings;
use Data::Dumper;

sub new {
    my ($class) = @_;
    return bless {}, $class;
}

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

   print Dumper $var;
}

1;
```

Listing 4

Interessant für OO

Aber nicht nur in Skripten und einfachen Modulen ist `namespace::clean` sinnvoll. Auch - oder gerade - für Objektorientierte Module eignet sich das Modul. Nehmen wir als Beispiel eine Klasse, die mit `Data::Dumper` arbeitet (Listing 4). `Data::Dumper` exportiert standardmäßig die Funktion `Dumper`.

Mit einem kleinen Stückchen Code kann man sich anschauen, welche Funktionsnamen in einem Namensraum bekannt sind (Listing 5). Wenn man sich das Ergebnis anschaut, fällt auf, dass auch die Funktion `Dumper` aufgeführt wird, obwohl das nicht in der Klasse definiert wird.

In den seltensten Fällen möchte man aber dieses Ergebnis haben. Warum sollte ein Code außerhalb der Klasse auch nur annehmen dürfen, dass die Klasse die Methode `Dumper` implementiert.

Das ist so ein Fall, in dem `namespace::clean` hilfreich ist:

```
package KlasseMitDataDumper;

use strict;
use warnings;
use Data::Dumper;
use namespace::clean;

# ... weiterer Code wie oben...
```

```
use KlasseMitDataDumper;

my @subs = find_subs();
print join ", ", @subs;

sub find_subs {
    my @subs;

    no strict 'refs';

    for my $name (
       keys %{KlasseMitDataDumper::} ) {
        if(defined
          &{"KlasseMitDataDumper::$name"} )
        {
            push @subs, $name;
        }
    }

    return @subs;
}

__END__
Dumper, new, my_dump
```

Listing 5

Damit wird wieder am Ende des Scopes (hier: am Ende der Datei) aufgeräumt. Versucht jetzt ein Code `KlasseMitDataDumper->Dumper(...)` aufzurufen, gibt es eine Fehlermeldung:

```
Can't locate object method "Dumper" via
  package "KlasseMitDataDumper"
```

Warum funktioniert das?

Sobald `namespace::clean` mit `use` eingebunden wird, merkt sich das Pragma, welche Subroutinen bis zu diesem Zeitpunkt definiert wurden. Genauer gesagt, merkt es sich die Namen. Am Ende des Scopes werden diese Namen gelöscht. Wichtig ist hierbei die Trennung von Funktionsnamen und dem Code.

Der Code bleibt die ganze Zeit erhalten, nur der Name wird gelöscht. Und das Modul arbeitet mit Blick in die Vergangenheit. Das heißt, es werden nur Funktionsnamen berücksichtigt, die bis zu dem Einbinden definiert wurden. Funktionen, die erst später definiert werden, werden nicht berücksichtigt.

Man kann also allein durch die Position der `use`-Anweisung bestimmen, welche Funktionsnamen gelöscht werden sollen.

```
use strict;
use warnings;

{
    package MyTest;

    use Data::Dumper;
    use namespace::clean;

    sub public {
        _internal();
    }

    no namespace::clean;

    sub _internal {
        print Dumper [1];
    }

    use namespace::clean;
}

MyTest->public;
MyTest->_internal;
```

Listing 6

Listing 6 zeigt, wie man mit `namespace::clean` manche der eigenen Funktionsnamen am Ende des Scopes löschen kann, während andere behalten werden. Das importierte `Dumper` von `Data::Dumper` soll in anderen Scopes nicht zur Verfügung stehen. Eine "öffentliche" Methode soll eben öffentlich sein und aus anderen Scopes heraus aufrufbar sein. Interne Funktionen sollen dagegen nur der Klasse selbst bekannt sein.

Bei der Schachtelung ist zu berücksichtigen, dass `namespace::clean` rückwärtsgewandt arbeitet. Bei der ersten `use`-Anweisung werden die Funktionsnamen gemerkt, die bis zu diesem Zeitpunkt definiert/importiert wurden. Danach kommen Funktionsdefinitionen. Mit der `no`-Anweisung merkt sich das Pragma die Funktionsnamen, die zwischen der `use`-Anweisung und der `no`-Anweisung definiert wurden. Diese werden nicht gelöscht. Gab es noch keine `use namespace::clean`-Anweisung, werden die Definitionen vom Beginn des Scopes bis zur `no`-Anweisung gemerkt. Das `use` am Ende weist `namespace::clean` an, die Funktionsnamen - die zwischen dem `no` und dem `use` definiert wurden - am Ende des Scopes zu löschen.

Die Ausführung des Codes in Listing 6 bringt folgende Ausgabe:

```
Can't locate object method "_internal"
  via package "MyTest" at ....
$VAR1 = [
          1
        ];
```

Aber nochmal zurück zur Trennung von Code und Namen. Man kann sich das ungefähr so vorstellen, dass es wie bei Kartons ist, bei denen der Name oben drauf steht. In jedem Karton befindet sich Material (der Code). In manchen Kartons gibt es Zettel, auf denen ein Name eines anderen Kartons steht, in dem sich ähnliches Material befindet. Das ist im Prinzip der Funktionsaufruf.

Wenn ich die Kartons durchgehe und mir merke, an welcher Stelle welcher Karton steht (das ist der Compile-Prozess), dann brauche ich später nicht mehr alle Namen auf den Kartons zu prüfen, sondern kann direkt den richtigen Karton greifen.

Im Prinzip macht das Perl genauso. Perl weiß nach dem Compile-Prozess, wo welcher Code ist. `namespace::clean` ist hier der Edding, der die Namen auf den Kartons unkennt-

lich macht (die Kartons bleiben aber an der Stelle stehen). Ich kann dann immer noch den richtigen Karton finden, weil ich die Stelle kenne, an dem der Karton steht. Jemand von außerhalb (ein anderer Scope) kann aber ohne die Namen auf den Kartons den Karton nicht finden.

Vorsicht!

Man muss natürlich aufpassen, wenn man mit Strings und `can` arbeitet. Da perl erst zur Laufzeit erfährt, welche Subroutine aufgerufen werden soll, weiß perl nicht, wo der Code liegt. Die Problematik ist in Listing 7 dargestellt.

Wenn man diesen Code ausführt, bekommt man folgende Fehlermeldung:

```
Use of uninitialized value in subroutine
  entry at clean5.pl line 15.
Can't use string ("") as a subroutine ref
  while "strict refs" in use at clean5.pl
  line 15.
```

Renée Bäcker

```
use strict;
use warnings;

{
    package MyTest;

    use Data::Dumper;
    use namespace::clean;

    sub public {
        my $internal = '_internal';
        my $sub =
            __PACKAGE__->can( $internal );
        $sub->();
    }

    no namespace::clean;

    sub _internal {
        print Dumper [1];
    }

    use namespace::clean;

}

MyTest->public;
MyTest->_internal;
```

Listing 7

MODULE

WxPerl Tutorial - Teil 4: Flächen

Thema

Nach dem dritten Teil, der zeigte wie Widgets optisch angenehm angeordnet werden, geht es dieses mal darum, worauf sie angeordnet werden und wie man sie in Gruppen organisiert. Weitere Möglichkeiten der Eingabe und Dekoration werden vorgestellt. Der Titel ist absichtlich schlicht, um Lesern einen sorglosen Einstieg zu bieten. Nach Ende der Lektüre können Begriffe wie *virtuelle Flächen*, *Splitter* und B*ookCtrl* dann nicht mehr einschüchtern.

Noch ein Tip: unter *http://docs.wxwidgets.org/trunk/* gibt es die Dokumentation in einem besser lesbaren und besser navigierbarem Format. Zu jedem Widget sind dort auch Abbildungen für jede Plattform.

Dekoration

Die einfachsten Widgets sind sicher die `StaticLine` und der `StaticText`. Das *static* im Namen lässt erkennen: diese Widget können vom Programmbenutzer nicht verändert werden und dienen der Orientierung.

```
Wx::StaticLine->new($panel, -1, @pos,
                    @groesse, wxLI_HORIZONTAL);
# oder wxLI_VERTICAL
Wx::StaticText->new($panel, -1, $text,
                 @pos, @groesse);
```

Entsprechend heißt ein einfaches Bild `Wx::StaticBitmap`. Es wird aber erst in der nächste Folge behandelt, wenn auch der genaue Unterschied zwischen Bitmap, Image und Icon erklärt wird.

Eine `StaticBox` ist aber kein einfacher rechteckiger Rahmen. Den kann eine sehr breite `StaticLine` mimen oder man greift zu Rahmen-Stilen (*Style*) der Panel und Fenster. Eine `StaticBox` ist ein Rahmen mit eingebauter Überschrift. Sie wurde letztens erwähnt, da der `StaticBoxSizer` innerhalb des Rahmens die Widgets anordnet.

Mittel der Wahl

Oft werden solche Rahmen für Radiobuttons verwendet. Dazu gibt in Wx aber eine spezielle *Radiobox*, die am Ende dieses Absatzes vorgestellt wird. *Radiobutton*, falls nicht bekannt, sind kleine runden Kreisen mit *StaticText*, bei denen immer nur einer, der Auserwählte der Gruppe, einen dicken Punkt inne hat. Selbstverständlich gibt es neben `Wx::RadioButton` auch `Wx::CheckBox`, die kleinen Quadrate mit *StaticText* in die einzeln und unabhängig Häkchen gesetzt werden. Das folgende Beispiel demonstriert die Benutzung dieser Widget, aber auch 2,3 Kniffe, mit denen sich einige Zeilen sparen lassen. Um später die Eingaben abzufragen braucht es Referenzen auf alle nichtdekorative Widgets. Aber wenn man die Referenzen in Arrays oder Hashes speichert, lässt sich das Füllen des Sizers mit einer Schleife vereinfachen. Auch wiederkehrende Parameterfolgen können in Arrays zwischengespeichert werden. Den wiederkehrenden Programmteil zu Beginn und am Ende hab ich weggelassen (siehe Listing 1).

Dieses Beispiel zeigt eine komplexere Sizeranwendung, welche die letzte Folge vermissen ließ, aber auch einige unübliche Dinge wie: 2 Panel nebeneinander in einem Frame und `RadioButton` mit `CheckBox` in einer `StaticBox`. Dieser Grenzfall zeigt sehr schön die Logik und die Grenzen der Navigationstasten. Mit keiner Taste kann ein Hauptpanel (Kind des Frames) verlassen werden.

```
sub OnInit {
    my $app   = shift;
    my $frame = Wx::Frame->new( undef, -1, 'Panel Test');

    my $poben = Wx::Panel->new( $frame, -1);
    my $schachtel = Wx::StaticBoxSizer->new(
        Wx::StaticBox->new($poben,-1,'Schachtel'),
        wxVERTICAL
    );
    my @std = (0, wxALL, 5);
    my @rb = map {
        Wx::RadioButton->new
            ($poben,-1,'Radiostation '.$_)
    } 1..3;
    $schachtel->Add($_, @std) for @rb;
    $schachtel->Add(Wx::CheckBox->new
            ($poben,-1,'Haken'), @std);

    my $rechtersizer = Wx::BoxSizer->new( wxVERTICAL );
    my @cb = map {
        Wx::CheckBox->new($poben,-1,'Haken '.$_)
    } 1..4;
    $schachtel->Add($_, @std) for @cb;

    my $obensizer = Wx::BoxSizer->new( wxHORIZONTAL );
    $obensizer->Add($schachtel, @std);
    $obensizer->Add($rechtersizer, @std);
    $poben->SetSizer($obensizer);

    my $punten = Wx::Panel->new( $frame, -1);
    Wx::RadioButton->new($punten,-1, '22', [10,10]);
    Wx::RadioButton->new($punten,-1, '43', [10,40]);

    my $panelsizer = Wx::BoxSizer->new(wxVERTICAL);
    $panelsizer->Add($poben, 1, wxGROW, 0);
    $panelsizer->Add($punten, 1, wxGROW, 0);
    $frame->SetSizer($panelsizer);
    $frame->Show(1);
    $punten->SetFocus();

    $app->SetTopWindow($frame);
    1;
}
```

Listing 1

Hoch und *Links* sowie *Rechts* (zurück) und *Unten* (vor) tun in diesem Beispiel das Selbe, sie wechseln zum vorigen/nächsten Widget. Allerdings nur in die Radiogruppe hinein, niemals hinaus. Dazu nimmt man `Shift+Tab` (zurück), oder `Tab` (vor), die zwar in die `StaticBox` aber nicht zu einem `RadioButton` wechseln. Das lässt erkennen, dass die `StaticBox` wirklich nur Dekor ist und die `RadioButton` sich als Gruppe absondern. Es ließen sich auch mehrere Gruppen in eine `StaticBox` frachten, dazu müßte der erste `RadioButton` jeder Gruppe ein `wxRB_GROUP` im Stil-Parameter haben. Für gewöhnlich werden Radiogruppen aber mit der zugehörigen `StaticBox` als ein Widget erstellt:

```
my $rg = Wx::RadioBox->new(
    $panel, -1, 'Überschrift',
    @pos, @groesse, @optionen,
    $spalten, wxRA_SPECIFY_COLS
);
```

Das ist schneller geschrieben und man kann wirklich mit den Richtungstasten in der Gruppe navigieren (mit Links / Rechts zwischen den Spalten), mit Tab zwischen den Gruppen. Die Zahl der Spalten bitte immer angeben, da ein Fehlen der letzten Parameter die Optionen waagerecht anordnet, was sehr breit werden kann. Aber auch weil die Label der `RadioButton` breiter als sichtbar sind, ist hier die Benutzung von `Fit` (siehe letzte Folge) ratsam. Beim Abfragen gibt es natürlich auch Unterschiede:

```
# gewählte Position
$rg->GetSelection;
# Label der Auswahl
$rg->GetString($rg->GetSelection);
$rbutton->GetValue();
$rbutton->GetLabel();
```

GetValue gibt wie das IsChecked() einer CheckBox einen boolschen Wert zurück. Ähnlich der RadioBox fasst eine CheckListBox mehrere CheckBoxen mit Rahmen als ein Widget zusammen.

```
my $clb = Wx::CheckListBox->new(
    $panel, -1,
    @pos,@groesse,@optionen,
);
```

Die Parameterliste sagt alles: CheckListBoxen haben keine eingebaute Überschrift und reihen die Kästchen in nur einer Spalte auf. Die Abfrage per $clb->IsChecked($n) für die $n-te CheckBox ist einfach.

Wenn sofort bei Auswahl etwas geschehen soll, braucht es Events. Die heißen immer wie das Widgets und benötigt auch immer die gleichen 3 Parameter.

```
EVT_RADIOBUTTON
EVT_RADIOBOX
EVT_CHECKBOX
EVT_CHECKLISTBOX
```

Dies ist einer der wenigen Stellen an denen die Wx-Doku nicht zu gebrauchen ist. Es fehlt der erste Parameter, der in WxPerl immer das Widget ist, das den Event auslöst. Es folgt die EventID, die wir bisher mit -1 durchwinken und die Coderef auf den Callback - die Routine die bei Eintreten des Ereignisses ausgeführt wird.

```
EVT_RADIOBOX($rb, -1, sub{...});
```

Es gibt noch eine lange Liste weiterer Auswahlwidgets, die sogar zu Listen reicht, die mit HTML gestaltet werden. Deswegen nur noch eine ein Weiteres, dessen Name (Wx::Choice) nicht selbsterklärend ist. Mancher kennt den Namen ComboCtrl, ein umrahmter Schriftzug, der angeklickt ein Menü oder Liste nach unten öffnet. Der darin ausgewählte Text wird neuer Inhalt des Rahmens. Die Wx::Choice ist die kleine Schwester der Wx::ComboCtrl die keine Mehrfachwahl zulässt. Die Erzeugung gleicht der CheckListBox, Abfrage erfolgt mit ($rg->GetString($rg->GetSelection);) und der Event ist EVT_CHOICE.

Wx::Panel

Über die Panel selber lässt sich kaum etwas sagen, da es passive Elemente sind. Das Wichtigste ist hier der Fokus. Damit wird eine Hervorhebung bezeichnet, die das aktive Widget markiert, dass den Tastaturinput empfängt. Wie im ersten Listing kann $widget->SetFocus jedem Widget die Aufmerksamkeit des Nutzers schenken. Ein $panel->SetFocus delegiert das allerdings an sein erstes Kind, dass damit etwas anfangen kann. Um sicher zu gehen, einfach $panel->AcceptsFocus abfragen. Wenn es positiv ist, gibt es keine Kinder, die den Fokus empfangen können. Es gibt nämlich Kandidaten wie StaticText und StaticLine die den Fokus nicht bekommen würden. Selbst wenn sie ihn explizit mit SetFocus verpasst bekämen wäre keiner zu sehen. Das lässt sich nutzen um den Fokus zum Programmstart zu unterdrücken. Oft wirkt das angenehmer. Die feine Art, sichtbaren Fokus zu vermeiden, ist aber $panel->SetFocusIgnoringChildren.

Die Reiterei

Panel sind die wiederverwendbaren Einheiten für die GUI. Zwar lassen sich auch einzelne Sizer mit angehängten Widgets ein- und ausblenden, aber das einfügen an anderer Stelle ist wesentlich aufwendiger (aber mit Reparent möglich). Auch gibt es bereits Widgets die beim ein- und ausblenden der Panel helfen, wovon das Notebook aka Reiterleiste wohl das bekannteste ist. Der deutsche Name ist etwas irreführend, da zum Widget Wx::Notebook auch die Fläche gehört, die sich ändert, wenn die Reiter angeklickt werden. Die Erzeugung eines Notebook ist bei Sizerlayout trivial:

```
my $nb = Wx::Notebook->new( $panel, -1);
```

Auch das befüllen mit Seiten, die je einem Reiter entsprechen, ist sehr einfach

```
$nb->AddPage( $subpanel, 'label', 0 );
```

wenn man einmal begriffen hat, dass es die bekannten Panel (oder einzelne Widgets) sind, die hier eingefügt werden. Sie können auch erst später befüllt oder geändert werden. Einzig wichtig dabei: Das eingefügte muss ein Kind des Notebook sein. Falls nicht, kommt es zu schweren optischen Verwerfungen. Auf jeden Fall reichen diese 2 Zeilen für eine optisch funktionierende Reiterleiste. Den dritten Parameter setze ich nur auf 1, wenn diese Seite/Reiter die aktive sein soll. Die weitere API mit InsertPage, GetPage und DeletePage ist nicht schwer, auch der Umgang mit den Events sollte einfach von der Hand gehen, wenn man bedenkt, dass der

zweite Parameter and den Callback ein `Wx::NotebookEvent` ist und das während `EVT_NOTEBOOK_PAGE_CHANGING` `GetSelection` nichts Sinnvolles liefern kann. Es gibt vielleicht 2 Dinge die sich nicht sofort erschließen. Das `Notebook` lässt sich auf einem Tab einfrieren wenn man:

```
$nb->SetSelection($nr);
EVT_NOTEBOOK_PAGE_CHANGING($nb, -1, sub {
    my ($widget, $event) = @_;
    $event->GetSelection == $nr
        ? $event->Veto
        : $event->Skip;
});
```

Bitte das `Skip` nicht vergessen, nirgends, jeglicher Event wird sonst abgewürgt. In diesem Falle fände ein Reiterwechsel ohne Panelwechsel statt. Ich hatte auch das Problem Reiter wechseln zu wollen, ohne Ereignisse auszulösen, die lange Datenchecks in Gang gesetzt hätten. Dann war `ChangeSelection` die kürzeste Lösung. Fast jeden wurmt sicher: Wie bekomme ich Icons in die Reiter? Was unweigerlich zu der nächsten Frage führt: Wie komme ich schnell an Icons ran? Die Systemicons sind unter festgelegten Namen (Konstanten) auffindbar. Der Code in Listing 2 konvertiert sie und lädt sie in eine Bildliste.

Wenn diese Liste dem `Notebook` zugewiesen wird:

```
$nb->AssignImageList( $imagelist );
$nb->SetPageImage($_, $_) for 1..9;
```

lässt sich mit `SetPageImage` einem Reiter ein Icon geben, wobei die erste Zahl die Nummer des Reiters (beginnt mit 0), die zweite die Nummer des Bildes in der Liste ist.

Buchkontrolle

Neben der Reiterleiste (tabs) gibt es noch andere Widgets deren Name auf "book" ended und die ebenso einen Stapel Panels oder Widgets verwalten. Ihre API ist fast identisch mit dem Vorgestellten, nur die Events heißen entsprechend anders.

Das im vorletzten Absatz genannte `Wx::Choice` ist Teil eines `Wx::Choicebook`, eine senkrechte Liste (`Wx::ListCtrl`) ist die Auswahl im `Wx::Listbook`, eine `Wx::ToolBar` (Werkzeugleiste) steuert das `Wx::Toolbook` und im `Wx::Treebook` dient eine Baumstruktur der Panelübersicht. Die letzten 3 bieten sich vor allem für Konfigurationsdialoge an.

Bewegliche Teiler

Ein anderes wichtiges Mittel zur Raumgestaltung sind *Splitter*. Jene senkrechten oder waagerechten Balken, die dem Nutzer erlauben selber zu wählen, wieviel Platz Flächen bekommen. Auch hier ist es entscheidend zu verstehen, dass nicht nur der Balken, sondern auch die beiden verwalteten Flächen zum Widget dazugehören. Deshalb müssen die Flächenfüller Kinder des `Wx::SplitterWindow` sein.

```
$teiler = Wx::SplitterWindow->new
    ($panel,-1,[-1,-1],[-1,-1],
         wxSP_LIVE_UPDATE);
$teiler->SplitVertically(
          $links, $rechts, 200);
$teiler->SetSashGravity(0.3);
```

Bei Flickerproblemen ist es oft das Einfachste `wxSP_LIVE_UPDATE` wegzulassen. Dann wird statt des Balkens nur eine Markierung bewegt, auf die der Balken bei Loslassen verschoben wird. Da das obere bzw. linke Widget das Wichtigere ist, sagt der dritte Parameter in `SplitVertically` und `SplitHorizontally`, wieviel Pixel Platz es bekommt. `SetSashGravity` ähnelt dem *proportion* der letzten Folge und legte fest, wieviel Prozent (1 == 100%) des Platzes es bekommt, der beim Vergrößern des Fensters neu erhältlich wird. Wie bei den `BookCtrl` gibt es die Events `EVT_SPLITTER_SASH_POS_CHANGING` und `EVT_SPLITTER_SASH_POS_CHANGED`. `$teiler->Unsplit` lässt den Trenner verschwinden und nur das obere oder linke Widget bleibt über. Das geschieht auch bei Doppelklick auf den Balken. Den bisher einzigen Weg das zu verhindern fand ich in dieser Lösung:

```
Wx::Event::EVT_SPLITTER_DCLICK
    ($teiler, -1, sub { $_[1]->Veto });
```

```
my $bildliste = Wx::ImageList->new( 16, 16 );
for my $art ( qw(wxART_GO_BACK wxART_GO_FORWARD
                 wxART_GO_UP wxART_GO_DOWN
                 wxART_PRINT wxART_HELP wxART_TIP)){
    $bildliste->Add(
        Wx::ArtProvider::GetBitmap(
            $art, 'wxART_OTHER_C',[16, 16] );
  }
```

Listing 2

Übergrößen

Was aber tun, wenn die erwünschte Oberfläche nicht in ein Fenster passt? Dazu gibt es mit dem `Wx::ScrolledWindow` ein besonderes Panel. Wie der Name ahnen lässt, besitzt es Scrollbars. Diese erscheinen, sobald das Panel weniger Platz als eine bestimmte Mindestgröße einnimmt. Überschreitet es die Größe, wird es zu einem gewöhnlichen Panel. Diese kritische Größe definiert:

```
$panel->SetScrollbars
    (10, 5, 40, 80, 0, 1);
```

`GetVirtualSize()` würde `[400,400]` ergeben, da die X-Ausdehnung 40 Einheiten zu je 10 Pixel und Y 80 Einheiten zu 5 Pixel beträgt. `[0,1]` sind die Koordinaten der neuen linken, oberen Ecke, zu der gescrollt wird (in Scrolleinheiten, die auch der Befehl `Scrol` annimmt). Scrolleinheiten meint die Schrittgröße der Scrollbewegung in Pixel, die wie gezeigt mit `SetScrollbars` oder `SetScrollRate` gesetzt wird.

Da z.B. Mausevents mit die Koordinaten des sichtbaren Bereiches arbeiten, aber das `ScrolledWindow` nur mit Koordinaten, die auch den unsichtbaren Bereich einbeziehen, braucht es an dieser Stelle noch eine Umrechnung. Angenommen das vorher definierte Panel hat nur 200x200 Platz und wurde ganz nach rechts unten, (also (20,40) in Scrolleinheiten) gescrollt. Ein Textfeld, dass an die Koordinaten [300,300] des Panels gepflanzt wurde:

```
Wx::TextCtrl->new(
    $panel, -1, 'text',
    [300,300],[100,100],
    wxTE_MULTILINE
)
```

beginnt nun bei den Koordinaten 100x100. Zumindest bei Abfrage dieses Mausklicks

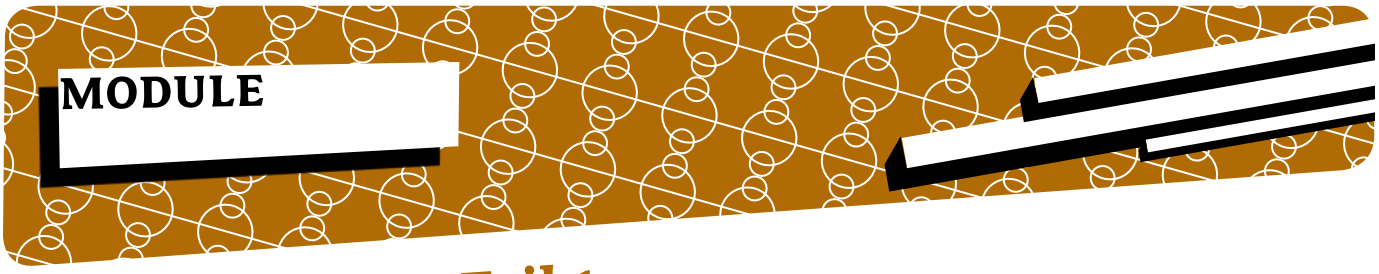
```
EVT_LEFT_DOWN($panel, sub{
    my ($panel, $event) = @_;
    my $x = $event->GetX;
    my $y = $event->GetY;
    my @true_xy $ =
     $panel->CalcUnscrolledPosition($x, $y);
});
```

müssen die erhaltenen X- und Y-Werte in Panel-Koordinaten konvertiert werden. `CalcScrolledPosition` ist der nah liegende Rückweg. Aber wem das zu viel ist, kann die Fläche jederzeit mit `FitInside` in ein einfaches Panel zurückverwandeln.

Ausblick

Der nächste Teil erzählt von Flächen anderer Art: den Fenstern. Neben *MDI* und *AUI* wird der Weg beschrieben, wie aus einem nackten `Wx::Frame` der Rahmen einer Applikation mit Menüleiste, Werkzeugleiste und Statuszeile wird.

Herbert Breunung



Moose Tutorial - Teil 1

In der 4. Ausgabe von $foo (Winter 2007) gab es schon einmal einen Artikel zu `Moose`, aber auf vielfachen Wunsch werde ich ein Tutorial zu dem Thema schreiben. Dabei werde ich ganz langsam Schritt für Schritt vorgehen und einzelne Punkte zu `Moose` beleuchten und erklären.

Wer schneller mit dem Einsatz von Moose beginnen will, kann sich den oben genannten Artikel zu Gemüte führen. Darin gibt Ronnie Neumann eine Schnelleinführung in das Modul.

In diesem ersten Teil soll es um ganz einfache Basisklassen gehen, die mit Moose umgesetzt werden. Ich werde dabei auch Vergleiche mit der Standard-Objektorientierung ziehen, damit die Unterschiede deutlicher werden.

Kamel vs. Elch?

`Moose` ist so ein Beispiel dafür, dass manche Entwicklung aus Perl 6 auch nach Perl 5 portiert wird. Das Modul setzt in etwa die Objektorientierung aus Perl 6 um. Einige Dinge, die in Perl 6 möglich werden, sind mit Perl 5 nicht umzusetzen (oder zumindest nicht so einfach). Außerdem bleibt die Syntax von Perl 5 innerhalb der Methoden erhalten.

Die Standard-Objektorientierung von Perl 5 wird immer wieder kritisiert - aus den unterschiedlichsten Gründen. Aber man sollte sie nicht unterschätzen. Sie ist mächtig und extrem flexibel. Allerdings muss man häufig Code schreiben, der mit Moose unnötig wird. Die Vergleiche werden das zeigen.

`Moose` hat aber zwei Probleme: Die Abhängigkeiten und die Performanz. Wer das Modul verwenden möchte, muss erstmal eine ganze Reihe an Modulen installieren. Sind diese aber erstmal installiert, steht dem Vergnügen nichts im Wege.

`Moose` ist beim Starten relativ langsam. Daher ist es für einfache CGI-Anwendungen sicher nicht geeignet. Wird das Modul aber unter mod_perl eingesetzt, wird ja nicht jedes Mal das Modul neu geladen. Und bei der Laufzeit sind Moose-Klassen nur minimal langsamer als "alter" Perl-Code.

Ein Riesenvorteil von `Moose` gegenüber anderem Perl-Code ist die Lesbarkeit und die "Codeeffektivität". Darunter verstehe ich die Anzahl der Zeilen um ein Ergebnis zu erreichen - wobei der Code lesbar bleiben muss. Mit Perl-Golfing lässt sich vielleicht ein Ziel in minimalem Code erreichen, aber die Lesbarkeit ist in der Regel dahin.

Ob man bei der Standard-Objektorientierung von Perl bleibt, oder Moose benutzt, hängt sicherlich von einigen Faktoren ab und es sollte jeder für sich entscheiden.

Minieinführung in Objektorientierung

Bevor es mit `Moose` richtig losgeht, möchte ich eine Minieinführung in Objektorientierung geben. Es sollen nur die wichtigsten Punkte in wenigen Sätzen beschrieben werden. Gerade in der Java-Welt gibt es sehr ausführliche Literatur zu dem Thema.

Die Objektorientierung versucht, alles in "Objekten" und "Klassen" abzubilden. Diese Sichtweise ist an das alltägliche Leben angelehnt: Alles um uns herum ist ein Objekt - sei es der Opel oder das Einfamilienhaus. Und diese Objekte haben einen Oberbegriff, der in der Objektorientierung als "Klasse" bezeichnet wird. Beim Opel wäre es die Klasse "Auto" und beim Einfamilienhaus das "Gebäude".

Was sind Klassen?

Als Klasse bezeichnet man den Oberbegriff von Objekten. Sie ist das "Baumuster" für die Objekte und wird mit Attributen und Methoden strukturiert. Als Beispiel kann die Klasse "Zeitschrift" sehen, die als Container für alle möglichen Zeitschriften dient.

Attribute

Ein Attribut ist eine Eigenschaft eines Objekts. Diese Beispielklasse "Zeitschrift" hat z.B. die Attribute "Titel", "Verlag" und "ISBN". Die Eigenschaften des Objekts werden in Variablen gespeichert, die von einem bestimmten Typ sind - das Attribut "Seitenzahl" ist z.B. ein Integer.

Methoden

Methoden spiegeln das "Verhalten" des Objekts wider. So kann ein Tier laufen, ein Fahrzeug fahren usw.

Was sind Objekte?

Ein Objekt ist die einzelne Ausprägung einer Klasse, so ist "$foo - Perl-Magazin" ein Objekt der Klasse "Zeitschrift" genauso wie "Schöner wohnen". In den Objekten sind die Attribute auch mit konkreten Werten belegt.

Vererbung

Aus Klassen können ganze Hierarchien aufgebaut werden. Die untergeordneten Klassen erben Methoden und auch Attribute der übergeordneten Klassen. Die so genannte "Superklasse" ist dabei ein Oberbegriff für die Subklassen. Ein Beispiel wäre "Fahrzeug". Jedes Fahrzeug hat eine "Antriebsart" und eine Methode "bewegen". Jetzt gibt es mehrere Klassen, die von "Fahrzeug" erben - nämlich "PKW", "LKW" und "Motorrad". Jede dieser drei Klassen hat automatisch eine Methode "bewegen". Alle weiteren Methoden und Attribute muss die Klasse selbst festlegen, denn für "LKW" sind andere Sachen wichtiger ("Größe Ladefläche") als für ein Motorrad.

Die Bedeutung dieser Begriffe wird im Laufe des Artikels durch die Umsetzung deutlicher.

Jetzt aber los!

Jetzt werden wir uns nach und nach an Moose herantasten. Die Beispiele basieren alle auf der Idee der Zeitschrift von weiter oben. In diesem Teil des Tutorials werden wir uns sehr ausführlich mit Attributen beschäftigen. Moose bietet da unzählige Möglichkeiten.

In Listing 1 wird eine ganz einfache Klasse "Zeitschrift" gezeigt.

```
package Zeitschrift;

use Moose;

has title     => ( is  => 'ro',
                   isa => 'Str' );
has issue     => ( is  => 'rw',
                   isa => 'Int' );
has publisher => ( is  => 'ro',
                   isa => 'Str' );

1;
```

Listing 1

Schon mit diesen wenigen Zeilen hat man einiges erreicht. Die Anweisung `use Moose;` importiert einige Schlüsselwörter, so z.B. `has`. Zusätzlich werden `strict` und `warnings` automatisch geladen.

Mit `has` werden Attribute erzeugt und damit auch Getter/Setter-Methoden (also Methoden, mit denen man die Attributwerte auslesen und setzen kann - siehe Listing 2). Durch den aussagekräftigen Code sieht man sehr schnell, welche Attribute ein Objekt der Klasse Zeitschrift haben wird.

```
my $foo = Zeitschrift->new(
    title     => '$foo - Perl-Magazin',
    publisher => 'Mein Verlag',
);

$foo->issue( 15 );

print sprintf "%s Nr. %s (%s)",
    $foo->title,
    $foo->issue,
    $foo->publisher;
```

Listing 2

Weiterhin legt man noch ein paar Metainformationen über das Attribut fest. In diesem Beispiel sind *title* und *publisher* Attribute, die "read-only" (`ro`) sind. Das bedeutet, dass die Werte nur beim Erzeugen eines Objekts gesetzt werden können. Würde man in Listing 2 noch die Zeile `$foo->publisher( 'neuer Verlag' );` einfügen, so würde das Skript an dieser Stelle beendet werden, da das Setzen des Verlags nicht erlaubt ist.

Dem entgegen stehen `rw`-Attribute wie *issue* im Beispiel. Solche Attribute können jederzeit geändert werden.

In dem Beispiel wird auch die Möglichkeit genutzt, Typen für die Attribute anzugeben. *title* und *publisher* sind vom Typ String (`Str`) und *issue* vom Typ Integer (`Int`). Moose

```
package ZeitschriftPlain;

use strict;
use warnings;

use Carp 'confess';

sub new {
    my $class = shift;
    my %args  = @_;
    my $self  = {};

    if (exists $args{issue}) {
        confess "Attribute (issue) does not pass the type constraint because: Validation
        failed for 'Int' with value $args{issue}"
            if ref($args{issue}) || $args{issue} !~ /\A[+-]?\d+\z/;
        $self->{issue} = $args{issue};
    }

    for my $attr ( qw/title publisher/ ) {
        if (exists $args{$attr}) {
            my $value = $args{$attr};
            confess "Attribute ($attr) does not pass the type constraint because: Validation
            failed for 'Str' with value $value"
                if ref($args{$attr});
            $self->{$attr} = $args{$attr};
        }
    }

    return bless $self, $class;
}

sub issue {
    my $self = shift;

    if (@_) {
        my $value = shift;
        confess "Attribute (issue) does not pass the type constraint because: Validation
        failed for 'Int' with value $value"
            if ref($value) || $value !~ /\A[+-]?\d+\z/;
        $self->{issue} = $value;
    }

    return $self->{issue};
}

sub title {
    my $self = shift;

    confess "Cannot assign a value to a read-only accessor" if @_;

    return $self->{title};
}

sub publisher {
    my $self = shift;

    confess "Cannot assign a value to a read-only accessor" if @_;

    return $self->{publisher};
}

1;
```

Listing 3

kennt einige Datentypen, aber man kann auch eigene Typen festlegen. Darauf werde ich später noch eingehen.

Wird so ein Typ festgelegt, wird auch automatisch eine Überprüfung durchgeführt sobald der Attributwert gesetzt wird. Für den Programmierer ist das sehr angenehm, da bei den gängigen Datentypen das nicht mehr selbst programmiert werden muss.

Schon allein an Listing 1 kann man erahnen, dass Moose viel Schreibarbeit erspart. Aber mal zum Vergleich ist die gleiche Klasse in Standard-OO von Perl in Listing 3 zu sehen.

Wie zu sehen ist, ist der Aufwand dabei ungleich größer. Vor allem die Umsetzung der "read-only" Attribute ist mit Standard-OO recht umständlich.

Mehr über Attribute bei Moose

Der in Listing 1 aufgeführte Code zeigt nur einen Bruchteil der Dinge, die für Attribute mit Moose möglich sind. In den nachfolgenden Abschnitten wird nach und nach immer etwas mehr gezeigt.

Neben den gezeigten `ro` und `rw` gibt es noch `bare`. Wird `bare` verwendet, wird weder eine Getter- noch eine Setter-Methode erzeugt. Versucht man also auf das Attribut zuzugreifen, wird ein Fehler ausgegeben.

```
# attribut der Moose-Klasse
has title => (
    is => 'bare',
);

# im Skript wirft das einen
# Fehler
$obj->title;
```

`is => 'ro'` kann auch durch `reader` ersetzt werden.

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

entspricht

```
has title => (
  reader => 'title',
);
```

Entsprechend dem `reader` gibt es auch einen `writer`-Parameter. Damit kann man den Namen der Setter-Methode bestimmen. Möchte man sowohl `reader` als auch `writer` verwenden, muss man darauf achten, dass diese nicht den gleichen Namen haben.

```
has title => (
    reader => 'title',
    writer => 'title',
    isa    => 'Str',
);
```

Wirft erstmal keinen Fehler möchte man dann aber im Programm mit `$obj->title` den Titel bekommen, so erscheint der Fehler, dass `undef` nicht den Regeln entspricht (weil ein String erwartet wird).

`reader` und `writer` eigenen sicher aber dafür, für Getter- und Setter-Methode unterschiedliche Namen festzulegen. In einigen Unternehmen ist es in den Programmierrichtlinien vorgesehen, dass Setter mit "set_" und Getter mit "get_". Das kann man dann mit den beiden Parametern erreichen.

```
has title => (
    reader => 'get_title',
    writer => 'set_title',
);
```

Ein weiterer Parameter, der im obigen Beispielcode schon gezeigt wurde ist `isa`. `Moose` kennt schon einige Typen von Haus aus (Abbildung 1).

Die Standardtypen sind soweit selbsterklärend. Der Unterschied zwischen `Item` und `Any` ist nur ein konzeptioneller. `Item` ist die Wurzel der Typen.

Die Typen mit `[]` können Parametrisiert werden. D.h. ein Attribut mit Typ `ArrayRef[Int]` ist eine Arrayreferenz, deren Elemente Integer sind. Wenn ein beliebiger Klassenname

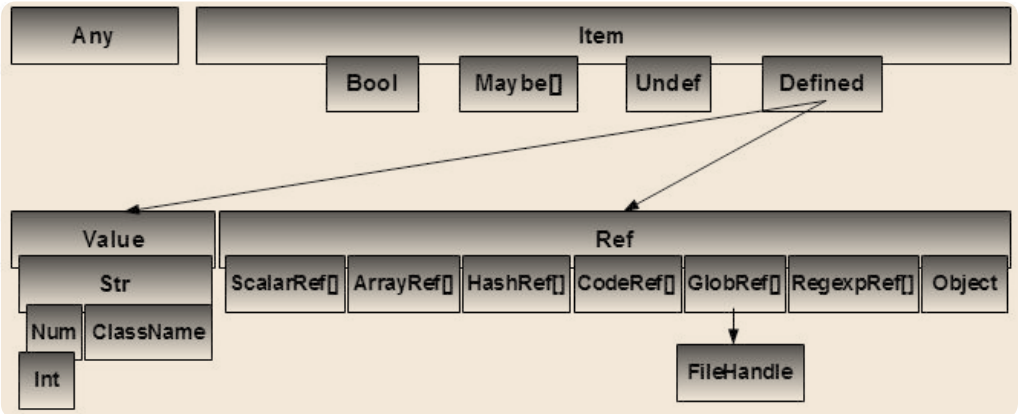


Abbildung 1

verwendet wird, muss darauf geachtet werden, dass diese Klasse vorher mit `use` eingebunden wird. Etwas später ist so ein Beispiel mit `DateTime` zu sehen.

Eine kurze Erläuterung zu den verschiedenen Typen. Typen wie `Any`, `Str`, `Num` und `Int` sind - denke ich - soweit sprechend. Auf `ArrayRef[]` bin ich auch schon etwas eingegangen.

- **Bool**

Dieser Wert darf `1` sein oder ein Wert, den Perl als "false" interpretiert.

- **Maybe[]**

Dieses Attribut darf entweder `undef` sein oder von dem Typ, mit dem `Maybe[]` parametrisiert wurde.

- **Undef**

Dieses Attribut ist `undef`.

- **Defined**

Dieses Attribut darf alles Mögliche sein, außer `undef`.

- **Value**
- **ClassName**

Dieses Attribut muss vom Typ *ClassName* sein.

- **Ref**

Attribute vom Typ `Ref` müssen eine Referenz als Wert haben. Dabei spielt es keine Rolle, ob dies eine Arrayreferenz, eine Hashreferenz oder sonst irgendeine Referenz ist.

- **ScalarRef[]**

Der Wert muss eine Referenz auf ein Skalar sein. Wie man an den `[]` erkennen kann, kann das noch näher bestimmt werden. Soll es eine Referenz auf ein Skalar sein, wobei der Inhalt der Skalarreferenz ein Integer sein soll, so kann man `ScalarRef[Int]` schreiben.

- **HashRef[]**

Eine Hashreferenz, bei der die Werte zu den Schlüsseln vom Typ des Parameters sein müssen. `HashRef[CodeRef]` ist also ein Mapping von Schlüsseln zu Codereferenzen.

- **CodeRef**

Eine Codereferenz.

- **GlobRef**

Referenz auf ein Glob.

- **FileHandle**

Wert muss vom Typ `IO::Handle` oder ein perl Built-in FileHandle sein.

- **RegexpRef**

Referenz auf ein Regexp-Objekt.

- **Object**

Bei dem Attributwert muss es sich ein Objekt handeln. Im Gegensatz zu `ClassName` ist dabei aber nicht wichtig, von welcher Klasse das Objekt ist. Hier wird überprüft, ob es eine Referenz ist und ob diese Referenz ge`bless`t ist.

Möchte man mehrere (Standard-)Typen erlauben, so können die verschiedenen Typen einfach mit `|` verbunden werden.

```
has mix => (
  is => 'rw',
  isa => 'ArrayRef[Str]|Str',
);
```

Damit kann das Attribut eine Arrayreferenz mit String-Elementen oder ein String sein. Möchte man mit einer Klasse und etwas anderem arbeiten, muss man einen Subtyp für die Klasse anlegen. Ein Beispiel dafür ist im Abschnitt "Subtypen" zu finden.

Bis jetzt waren die Attribute alle optional. Es war also egal, ob das Attribut gesetzt wurde oder nicht. Es gibt aber häufig auch Attribute die gesetzt werden müssen. Es gibt keine Person ohne Namen. Für solche "Muss"-Attribute gibt es den Parameter `required`. Wird dieser auf einen "wahren" Wert gesetzt, so muss ein Wert für dieses Attribut gesetzt werden.

```
has name => (
    is => 'rw',
    isa => 'Str',
    required => 1,
);
```

Vergisst man jetzt dem Konstruktor einen Namen zu übergeben, so bekommt man gleich die Fehlermeldung `"Attribute (title) is required"` angezeigt. Lässt man die `isa`-Angabe weg, so muss das Attribut zwar einen Wert haben, der darf aber auch `undef` sein.

Soll das Attribut ein Pflichtattribut sein, soll es aber freigestellt sein, ob dem Konstruktor ein Wert übergeben wird, so bietet `Moose` auch hierfür eine Lösung. Man kann über den Parameter `default` einen Wert setzen oder man gibt bei dem Parameter `builder` einen Subroutinen-Namen an. Diese Subroutine muss dann Standardwert zurückliefern. `default` und `builder` können aber auch ohne `required` verwendet werden.

```
has title => (
  is => 'rw',
  isa => 'Str',
  required => 1,
  builder  => 'standard_title',
);

sub standard_title {
    return 'hallo welt!';
}
```

Bei `default` kann man entweder eine Subroutinenreferenz angeben oder einen festen Wert. Soll dieser "feste Wert" allerdings eine irgendwie geartete Referenz sein, muss man diese in eine Subroutine packen, die diese Referenz liefert.

```
has pages => (
    is => 'rw',
    isa => 'Int',
    default => '3',
);

has publisher => (
    is => 'rw',
    isa => 'Str',
    default => sub{ return 'MeinVerlag' },
);

has test => (
    is => 'rw',
    isa => 'ArrayRef[Int]',
    # default => [], # FEHLER!!
    default => sub { return [ 1, 2 ] },
);
```

Aber worin liegt jetzt der Unterschied zwischen `default` und `builder`? Zum einen kann man die Subroutine, die bei `builder` angegeben ist für mehrere Attribute verwenden, wenn diese Attribute auf die gleiche Weise den Standardwert bekommen. Ein Beispiel wäre ein Objekt vom Typ "Arbeitsaufgabe", bei der Start und Ende standardmäßig die aktuelle Zeit ist. Das könnte dann so aussehen:

```
package Arbeitsaufgabe;

use Moose;
use DateTime;

has start => (
    is => 'rw',
    isa => 'DateTime',
    builder => 'now',
);

has end => (
    is => 'rw',
    isa => 'DateTime',
    builder => 'now',
);

sub now {
    return DateTime->now;
}
```

Das folgt dann wieder dem Motto "DRY" (Don't repeat yourself). Zusätzlich bietet die Sache mit `builder` noch einen weiteren Vorteil: Wenn man von der Klasse erbt, kann man diese Subroutine überschreiben.

```
package ArbeitsaufgabeGeerbt;

use Moose;
use DateTime;
extends 'Arbeitsaufgabe';

sub now {
    return DateTime->now->add( days => 2 );
}
```

Und schon beginnen die Aufgaben nicht sofort, sondern erst in zwei Tagen (siehe `script5.pl` in den Quellcodes). Das hier gezeigte `extends` bedeutet, dass diese Klasse die Klasse `Arbeitsaufgabe` erweitert (und davon erbt). Auf Vererbung werde ich in diesem Teil des Tutorials aber nicht näher eingehen. Das kommt in einer der nächsten Ausgaben.

Auf der Abbildung von den Standardtypen war schon zu sehen, dass es einige Typen gibt, die Referenzen sind (z.B. `ArrayRef[]`. Um nicht jedesmal dereferenzieren zu müssen wenn man die Attributwerte haben möchte, gibt es den Parameter `auto_deref`. Die folgenden Codes zeigen den Unterschied.

```
# in der Klasse
has list => (
    is => 'rw',
    isa => 'ArrayRef[Int]',
);

# im Skript
for my $value ( @{ $object->list } ) {
    print "$value\n";
}
```

gegenüber

```
# in der Klasse
has list => (
  is => 'rw',
  isa => 'ArrayRef[Int]',
  auto_deref => 1,
);


# im Skript
for my $value ( $object->list ) {
    print "$value\n";
}
```

`auto_deref` kann allerdings nur bei `ArrayRef`, `HashRef`, `ArrayRef[]` und `HashRef[]` eingesetzt werden. Wird dieser Parameter bei anderen Attributen eingesetzt, bekommt man eine Fehlermeldung.

Ein sehr flexibler Parameter ist `handles`, womit man Methoden an andere Module (das in `isa` angegebene Modul) delegieren kann. Der Parameter akzeptiert Sowohl eine Hashreferez, eine Arrayreferenz oder einen Regulären Ausdruck.

```
has published => (
    is => 'rw',
    isa => 'DateTime',
    handles => {
        year => 'year',
        date => 'ymd',
    },
);
```

Damit bekommt das Objekt neue Methoden: `year` und `date`. Diese beiden Methoden werden aber durch das Objekt in dem Attribut ausgeführt. Durch das `handles` ist jetzt folgender Code möglich:

```
print sprintf
  "Published: %s\nDate: %s\nJahr %s\n",
    $types->published,
    $types->date,
    $types->year;
```

Ohne das `handles` würde der Code folgendermaßen aussehen:

```
print sprintf
  "Published: %s\nDate: %s\nJahr %s\n",
    $types->published,
    $types->published->ymd,
    $types->published->year;
```

Sollen die delegierten Methoden den gleichen Namen wie die Methoden für das Objekt im Attribut (wie hier z.B. "year"), dann kann man auch eine Arrayreferenz übergeben:

```
has published => (
    is => 'rw',
    isa => 'DateTime',
    handles => ['year', 'ymd'],
);
```

Sollen alle Methoden, die einem bestimmten Muster entsprechen, delegiert werden, so kann man auch einen Regulären Ausdruck übergeben.

```
has published => (
    is => 'rw',
    isa => 'DateTime',
    handles => qr/^y/,
);
```

In diesem Fall werden alle Methoden, die mit 'y' beginnen delegiert. Sind in der Klasse aber solche Methoden jedoch implementiert, werden natürlich diese Funktionen aufgerufen.

Aufpassen muss man, wenn Methoden delegiert werden, die in mehreren Klassen auftauchen:

```
has test1 => (
    is => 'rw',
    isa => 'TestKlasse1',
    handles => qr/^t/,
);

has test2 => (
    is => 'rw',
    isa => 'TestKlasse2',
    handles => qr/^t/,
);
```

Sowohl `TestKlasse1` als auch `TestKlasse2` implementieren die Methode `test`. Dann bekommt man die Fehlermeldung `"You cannot overwrite a locally defined method (test) with a delegation"`. Das ist im ersten Moment verwirrend, wenn die eigene Klasse gar keine Methode `test` definiert. In so einem Fall muss man dann auf die Hash-Variante wechseln und bei einem Attribut eben einen anderen Namen bereitstellen.

```
has test2 => (
    is => 'rw',
    isa => 'TestKlasse2',
    handles => {
        tester => 'test',
    },
);
```

Eine vierte Variante gibt es noch: Man kann `handles` einen Namen einer Rolle übergeben. Aber auf Rollen gehe ich in einer späteren Folge des Moose Tutorials ein.

Wurde das Attribut schonmal gesetzt? Diese Frage stellt man sich hin und wieder. Mit dem Parameter `predicate` kann man einen Subroutinennamen vergeben.

```
has date => (
  is => 'rw',
  isa => 'Str|Undef',
  predicate => 'has_date',
);
```

Im Programm kann man dann einfach mit `$obj->has_date` abfragen, ob der Slot für das Attribut existiert oder nicht. Dass das nichts mit `undef` oder Definiertheit zu tun hat, zeigt das Beispiel.

```
use ZeitschriftPredicate;

my $var = ZeitschriftPredicate->new;
print "yes (1)\n" if $var->has_date;

$var->date( undef );
print "yes (2)\n" if $var->has_date;
```

Beim erstenmal wird nichts ausgegeben, da das Attribut bisher nicht besetzt wurde. Dann wird das Datum auf `undef` gesetzt, und erneut überprüft. Jetzt wird `yes (2)` ausgegeben.

Natürlich gibt es auch einen Parameter, mit der man eine Subroutine bestimmen kann, die das Attribut wieder zurücksetzt. Der Parameter heißt `clearer`.

```
has date => (
  is => 'rw',
  isa => 'Str|Undef',
  predicate => 'has_date',
  clearer   => 'clear_date',
);

$var->clear_date;
print "yes (3)\n" if $var->has_date;
```

Das `yes (3)` wird nicht ausgegeben, weil das Attribut jetzt nicht einfach auf `undef` gesetzt wird, sondern der Slot für das Attribut entfernt wird.

Sehr praktisch ist auch `trigger`. Bei diesem Parameter kann man eine Codereferenz angeben, die ausgeführt wird, sobald das Attribut gesetzt wird. Damit kann man z.B. eine nachträgliche Überprüfung von Werten oder Logging ermöglichen.

```
has date => (
  is => 'rw',
  isa => 'Str|Undef',
  trigger => \&date_trigger,
  default => 3,
);

sub date_trigger {
  my ($self,$value,$old) = @_;

  print "Neues Datum: $value\n";
}
```

Zu beachten ist hierbei, dass der Trigger nur aufgerufen wird, wenn ein Wert über den Konstruktor oder über den Setter gesetzt wird. Wird das Attribut mit `default`, `lazy_build` oder `builder` gesetzt wird, wird der Trigger nicht aufgerufen.

Als Parameter bekommt der Trigger das Objekt, den neuen Wert und den alten Wert übergeben.

Eben wurde schon `lazy_build` kurz erwähnt. Dann möchte ich jetzt noch kurz darauf und auf `lazy` eingehen. `lazy_build` vereinfacht das Erzeugen von Attributen, weil es einige Parameter automatisch setzt.

Ein

```
has name => ( lazy_build => 1 );
```

entspricht einem

```
has name => (
  lazy => 1,
  required => 1,
  clearer => 'clear_name',
  builder => '_build_name',
  predicate => 'has_name',
);
```

Auch hier taucht das `lazy` wieder auf. Dieser Parameter sorgt dafür, dass das Attribut erst beim ersten Zugriff existiert. Das macht gerade dann Sinn, wenn ein Attribut nicht immer gebraucht wird und/oder der Standardwert sehr aufwändig erzeugt werden muss.

Wenn `lazy` verwendet wird, muss ein `builder` oder ein `default` angegeben werden.

Soll für ein Attribut im Konstruktor ein anderer Name angewendet werden als bei den Getter/Setter-Methoden, so kann man den neuen Namen über den Parameter `init_arg` setzen.

```
has date => (
  is => 'rw',
  isa => 'Str',
  init_arg => 'published',
  predicate => 'has_date',
);
```

Wenn man jetzt dem Konstruktor einen Parameter `date` mitgibt, wird das Attribut nicht gesetzt (`has_date` liefert auch "false" zurück). Setzt man dagegen den Parameter `published`, wird das Attribut gesetzt und man kann sich danach mit `$obj->date` den Wert holen.

Viel nützlicher empfinde ich da die Möglichkeit, `init_arg` auf `undef` zu setzen. Damit kann man erreichen, dass der Parameter einfach ignoriert wird und das Attribut nicht über den Konstruktor gesetzt werden kann. Hierbei ist allerdings zu beachten, dass bei `required`-Attributen `init_arg` nicht auf `undef` gesetzt werden darf wenn kein `builder` oder `default` angegeben ist.

Zum Abschluss noch zwei weitere Parameter für die Attribute: `coerce` und `weak_ref`.

Mit coerce kann man erreichen, dass beim Setzen eines Attributwerts ein Objekt daraus erzeugt wird. Das ist z.B. sehr praktisch wenn man bei einem Datumsattribut einfach nur einen String übergeben möchte und dann ein `DateTime`-Objekt in dem Attribut gespeichert wird.

```
use Moose;
use Moose::Util::TypeConstraints;
use DateTime;

subtype 'DateTime' =>
  as 'Object' =>
  where { $_->isa( 'DateTime' ) };

coerce 'DateTime' =>
  from 'Str' =>
  via {
    my ($day,$month,$year) = split /\./, $_;
    my $obj = DateTime->new( year => $year,
              month => $month, day => $day );
    $obj;
  };

has published => (
  is => 'rw',
  isa => 'DateTime',
  coerce => 1,
  handles => [ 'year', 'month' ],
);
```

Damit ist es sehr angenehm, mehr aus diesem Attribut zu holen:

```
use Coerce;

my $var = Coerce->new(
   published => '01.08.2010',
);

print $var->year;
```

Wer `coerce` verwendet, kann kein `weak_ref` verwenden. `weak_ref` ist dafür da, dass der Referenzzähler für das Objekt in dem Attribut nicht hochgezählt wird. Das ist wichtig, wenn dadurch zirkuläre Referenzen entstehen könnten. Das würde zu Speicherlöchern führen.

Auf die Parameter `does` und `traits` werde ich an dieser Stelle nicht näher eingehen, da diese mit Rollen zu tun haben. Hier sei nur gesagt, dass man bei `does` eine Rolle angeben kann, die ein Verhalten für dieses Attribut implementiert.

Die beiden Option `initializer` und `documentation` existieren zwar, sollten aber nicht verwendet werden. `Moose` kann mit `documentation` nichts anfangen, außer die Information zu speichern. Und statt `initializer` sollte `builder` verwendet werden.

Subtypen

Es gibt Fälle, in denen die Standardtypen für Attribute nicht ausreichen. So kann z.B. die Anzahl der Seiten in einer Zeitschrift nicht negativ sein. Also reicht der Typ `Int` nicht aus. In diesem Fall kann man in der Klasse das so formulieren:

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

subtype positiveInt =>
    as 'Int' =>
    where { $_ > 0; }
;

has pages => (
  is => 'rw',
  isa => 'positiveInt',
);
```

Damit ist es nur noch möglich, positive Integer dem Attribut zuzuweisen. Versucht man jetzt eine negative Zahl zu übergeben, kommt die Fehlermeldung `"Attribute (pages) does not pass the type constraint because: Validation failed for 'positiveInt' failed with value -64 at ..."`.

Mit diesen Subtypen kann man beliebige eigene Typen erstellen.

Ich bin auch noch das Beispiel für "verknüpfte" Typen mit eigenen Subtypen schuldig. Hier ein Beispiel, wie man erreichen kann, dass ein Attribut entweder ein `DateTime`-Objekt oder ein String ist. Natürlich kann man auch das weiter aufbohren und einen Subtyp "datestring" machen, der überprüft ob der String auch tatsächlich im richtigen Format vorliegt.

```
subtype 'DateTime' =>
    as 'Object' =>
    where { ref( $_ ) eq 'DateTime'; };

has date => (
    is => 'rw',
    isa => 'DateTime|Str',
);
```

Hier wird gesagt, dass es einen Subtyp `DateTime` gibt, der von `Object` abgeleitet ist. `Moose` sorgt dan dafür, dass erst die übergeordnete Regel - also ob es ein Objekt ist - geprüft wird und danach erst die Regel des Subtypes.

Der Block, der bei `where` übergeben wird, muss einen "wahren" Wert zurückliefern damit die Regel als erfüllt angesehen wird.

Attribute in Subklassen verändern

In einem Beispiel weiter oben wurde schon gezeigt, wie man Vererbung mit `Moose` erreichen kann. Dabei werden auch die Attribute mit vererbt. Manchmal kommt es vor, dass diese Attribute für die erbende Klasse angepasst werden sollen.

In so einem Fall definiert man für die erbende Klasse das gleiche Attribut, stellt dem Namen aber noch ein "+" voran.

```
package ArbeitsaufgabeGeerbt;

use Moose;
use DateTime;

extends 'Arbeitsaufgabe';

has "+end" => (
    is => 'rw',
    isa => 'DateTime',
    builder => 'set_end',
);

sub set_end {
    DateTime->now->add( years => 20 );
}
```

Hier wurde jetzt die "builder"-Methode für das Attribut `end` geändert. Nicht alle Angaben kann man hierbei ändern, sondern nur:

- default
- coerce
- required
- documentation
- lazy
- isa
- handles
- builder
- metaclass
- traits

Ausblick

Dies war der erste Teil des Moose-Tutorials. Etliche Seiten nur über Attribute haben wohl einen ersten Einblick gegeben, wie mächtig Moose ist und wie flexibel die Attribute sind. Nicht alle Parameter werden immer gebraucht, aber man sollte um sie wissen. In der nächsten Ausgabe des Moose Tutorials wird es dann um Methoden gehen. Auch dort gibt es einiges, bei dem Programmierern Arbeit abgenommen wird.

Renée Bäcker

12. Deutscher Perl Workshop
Perler unter sich

Wolltet ihr schon immer einmal Gleichgesinnte treffen, die sich stundenlang eure Programmierprobleme und Überlegungen anhören können und euch von ihren neuesten Hacks und Entdeckungen erzählen? Vom 7. - 9. Juni gab es wie jedes Jahr Gelegenheit. Denn zum 12. deutsche Perlworkshop in Schorndorf bei Stuttgart versammelten sich etwa 80 Programmierer um Vorträgen zuzuhören, Ansichten auszutauschen und zusammen ein Bier oder Limonade zu trinken.

Die Vorträge

Bei den Vorträgen, die meist 20 oder 40 Minuten lang waren, ging es natürlich um die ständigen Themen wie Webprogrammierung, Datenbanken, `UTF`, `XML`, Automation und Testen, sowie um aktuelle Module wie `Moose`, `DBIx::Class` und `Plack` sowie *git*. In Sachen Perl 6 gab es wie erwartet Meinungsverschiedenheiten aber man konnte auch kompakte, praktische Lösungen wie `IO::All` kennen lernen oder wie man in KDE einfach Unicode-Zeichen mit Tastatur eingibt. Daneben wurden auch exotische Themen wie die Sprache Rebol oder *golden software devices*, auf Zuverlässigkeit optimierte Referenzimplementationen, präsentiert.

Aber fast immer stand hinter einem Vortrag ein momentanes Betätigungsfeld des Autors. Egal ob eines von der Arbeit (Firefox-Automatisation von Max Maischein) oder ein selbst gewähltes (Aufbau einer *benchmark suite* für Perl von Steffen Schwigon). Technisch professionell, witzig und mit eigenem Modul war wie immer Marc Lehmann, dieses mal mit `AnyEvent::MP`. Der lauteste und für viele unterhaltsamste Redner war sicher der aus England eingeflogene Matt Trout, der über `ExtUtils::MakeMaker`, `IO::All` und einige andere Inhalte sprach. Patrick Michaud konnte leider nicht kommen, aber der aus Australien gereiste Kieren Diment gab 2 sehr lange, aber dennoch interessante Vorträge über das Schreiben von guter Dokumentation, wozu er besonders weniger versierte Programmierer aufrief, da diese oft besser die Leser einer Doku verstehen. In seinem zweiten Vortrag wertete er die jüngste Perl-Umfrage aus, welche die Perl-Foundation in Auftrag gab. Diese kann mit solider Datenlage einige Vorurteile gegenüber Perl entkräften, aber auch der TPF wertvolle Hinweise geben, welche Module am besten in Zukunft mehr gefördert werden sollten. Noch mehr in Richtung Marketing ging Gabor Szabos Kurzvortrag. Er berichtete in Wort und Bild über seine jüngsten Aktionen auf Konferenzen wie FOSDEM und CeBIT um Bekanntheit und Wahrnehmung von Perl zu verbessern. Er verließ die Konferenz auch nach dem ersten Tag um zum LinuxTag nach Berlin zu fahren..

Die Organisation

Jedoch ging er erst nach dem *Social Event*, einer gemeinsamen Feier aller Teilnehmer die im günstigen Preis von 75 Euro für 3 Tage Konferenz inbegriffen war. Dafür gab es sogar in den Pausen noch Getränke, "gscheite Brezen" (Brezeln) oder "süsse Stückle" (Quark-Frucht-Taschen). Dies wurde aber von Sponsoren wie *XING* und *booking.com* bezahlt. Den größten Anteil (die Raummiete für die Barbara-Künkelin-Halle) bezahlte die *Wirtschaftsförderung Region Stuttgart*, die besonders um *Open Source* bemüht ist. Auch *Nestoria*, *ActiveState* und die *$foo* trugen bei und die Bücher zur abschließenden Tombola kamen wie immer von *O'Reilly*. Organisiert wurde die Konferenz vor allem von den freiwilligen Helfern der Stuttgart.pm um Rolf Schaufelberger.

Die Teilnehmer hatten wirklich eine gute, vor allem entspannte Zeit. Das Klima war sehr harmonisch. Alle Wege waren

sehr kurz (kleines Fachwerkstädtchen statt riesigem Betoncampus - ein Vorraum anstatt langer Flure), das Essen im Ristorante neben Gottlieb Daimlers Geburtshaus war sehr gut und die allermeisten Gesichter wohlvertraut.

Neue Besucher mit und ohne Perlerfahrung sind immer willkommen, jedoch wurde dieses Jahr weniger die Werbetrommel geschwungen und es gab auch einige technische Probleme z.B. mit der Act-Software. Damit war zum ersten mal der alt-ehrwürdige Deutsche Perlworkshop mit anderen Perlveranstaltungen vernetzt. Eine andere Neuerung waren die 2 parallelen Tracks, auch wenn es sie nur am ersten Vormittag gab.

Wo und ob im nächsten Jahr der Workshop stattfindet ist noch nicht festgelegt. Keine Mongers-Gruppe bewarb sich bisher offiziell und der Workshop könnte auch zu Gunsten der YAPC::EU (die die Frankfurt.pm anstrebt) auf 2012 verschoben werden.

Einen oder zwei Helfer sucht die WsOrga dennoch, denn einen nächsten Workshop wird es sicher geben!

Herbert Breunung

// 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“

HowTo

CPAN::Reporter - CPAN-Tester werden ist ganz einfach

Das *CPAN Testers Project* möchte möglichst viele CPAN-Module auf möglichst vielen Plattformen testen. Diese Testergebnisse bieten CPAN-Autoren und Nutzern wertvolles Feedback und helfen die Qualität der CPAN-Module zu verbessern und somit den Wert des CPAN zu erhöhen.

Dank des CPAN-Moduls CPAN::Reporter von David Golden kann jeder, der ein CPAN-Modul auf seinem Rechner installiert, einen Test-Report automatisch erzeugen und zu den CPAN-Testern übermitteln.

CPAN::Reporter installieren

CPAN::Reporter benötigt das Modul CPAN ab Version 1.9301 und weitere Module, wie z.B. Module::Build, in möglichst aktuellen Versionen.

```
cpan> install Bundle::CPAN
cpan> install CPAN::Reporter
cpan> reload CPAN
```

CPAN::Reporter konfigurieren

Die Konfiguration beschränkt sich meist auf die Beantwortung einiger weniger Fragen.

```
cpan> o conf init test_report
```

Im ersten Schritt wird festgelegt, ob die Testergebnisse zu den CPAN Testern versandt werden sollen.

```
<test_report>
Email test reports if CPAN::Reporter is
  installed (yes/no)? [no] yes

Would you like me configure CPAN::Reporter
  now? [yes] yes
```

Falls *ja* wird als nächstes die E-Mail Adresse des Test Reporters abgefragt. Wer über eine PAUSE-ID verfügt, trägt bitte die dort hinterlegte E-Mail-Adresse (*pauseid@cpan.org*) ein; ansonsten ist jede gültige Adresse okay.

```
email_from? [] "Vorname Nachname"
 <e-mail@adresse.tld>
```

Die Testergebnisse können entweder direkt mit *Net::SMTP* an die Mailserver von perl.org gesendet werden oder über einen eigenen Mailserver. Im ersten Fall lässt man das Feld einfach leer, im zweiten Fall wird die Adresse des eigenen Mailservers eingetragen.

```
smtp_server? [] mail.mycompany.tld
```

Auf die Konfiguration für den Versand über ISPs, die meist eine Authentifizierung verlangen, wird weiter unten in diesem Artikel eingegangen.

Die Testergebnisse können vor dem Versand noch mit einem Editor bearbeitet werden. Als Tester kann man hier noch Hinweise für den Author einfügen.

Die vorgeschlagenene Standardkonfiguration sendet die Reports im Erfolgsfall automatisch und fragt im Misserfolgsfall nach, ob der Report vor dem Versand editiert werden soll.

```
edit_report? [default:ask/no pass/na:no]
```

Nun muss die Konfiguration noch gespeichtert werden.

```
cpan> o conf commit
```

Die Konfigurationsdatei *config.ini* befindet sich im Heimatverzeichnis des aktuellen Benutzers im Unterverzeichnis *.cpanreporter* und kann mit einem beliebigen Text-Editor bearbeitet werden.

Konfiguration Versand über ISP

Wer die Testergebnisse über einen ISP versenden möchte, der SMTP-Auth in Verbindung mit STARTTLS verwendet, installiert zusätzlich das CPAN-Modul *Net::SMTP::TLS* und passt den Transportmechanismus (*transport=*) in der Konfigurationsdatei *config.ini* von Hand an.

Beispiel für den Versand via Strato AG:

```
edit_report=default:ask/no pass/na:no
email_from="Thomas Fahle"
             <cpan@thomas-fahle.de>
send_report=default:ask/yes pass/na:yes

transport=Net::SMTP::TLS User
cpan@thomas-fahle.de Password geheym Port 587

smtp_server=smtp.strato.de
```

Da in dieser Datei das Passwort im Klartext steht, sollte man zu mindestens die Rechte für den Zugriff auf die Datei und auf das Verzeichnis .cpanreporter einschränken, z.B. `chmod 600 config.ini`.

Alle Testergebnisse auf ein Mal senden

Die Übermittlung eines Testberichts kann je nach Anbindung zwischen drei und fünf Sekunden benötigen. Hier bietet sich an, die Reports lokal zwischen zu speichern und erst später zu senden. Dazu dient die Option *transport=File*.

Beispiel:

```
edit_report=default:ask/no pass/na:no
email_from="Thomas Fahle"
             <cpan@thomas-fahle.de>
send_report=default:ask/yes pass/na:yes
transport=File /home/tf/.cpanreporter/reports
```

Nun werden die Reports im Ordner *.cpanreporter/reports/*, den man zuvor von Hand anlegen muss, zwischengespeichert.

Die gespeicherten Berichte lassen sich mit dem Programm *x-perl-send-test-reports* von Pedro Melo bequem versenden:

```
$ perl ./x-perl-send-test-reports \
--from '"Thomas Fahle" \
   <cpan@thomas-fahle.de>' \
--transport "Net::SMTP::TLS User
    cpan@thomas-fahle.de \
    Password geheym Port 587" \
--clean \
--server smtp.strato.de \
/home/tf/.cpanreporter/reports
```

Hinweis zur Privatsphäre

Testberichte enthalten Informationen über das Testsystem, wie Umgebungsvariablen, Betriebssystem, installierte Bibliotheken usw. Wer sich Sorgen um die Privatsphäre macht, sollte die Berichte zuvor editieren. Die dazu passende Konfiguration

```
edit_report=default:ask/yes pass/na:yes
```

Werde CPAN-Tester

Es ist wirklich einfach CPAN-Test-Reporter zu sein. Je größer das Netzwerk der Tester, um so besser. Werde CPAN-Tester.

Links

- CPAN::Reporter http://search.cpan.org/dist/CPAN-Reporter/
- CPAN::Reporter::Config http://search.cpan.org/perldoc?CPAN::Reporter::Config
- CPAN Testers Project http://www.cpantesters.org/
- The CPAN Testers Wiki http://wiki.cpantesters.org/
- Pedro Melo - A faster configuration for CPAN::Reporter http://www.simplicidade.org/notes/archives/2009/10/a_faster_config.html
- Pedro Melo - x-perl-send-test-reports http://github.com/melo/scripts/blob/master/bin/x-perl-send-test-reports

Thomas Fahle

TPF News

Corporate Perl unter Windows

Curtis Jewell hat seinen Grant abgeschlossen. Dieser beinhaltete die folgenden Punkte:

• Ein Skript, dass die Speicherung der local::lib-Einstellungen unter Windows leicht ermöglicht Zur Zeit existiert dieses Skript im CPAN und im SVN-Repository.
• Ein Modul, das es anderen ermöglicht, Distributionen auf Basis von StrawberryPerl zu erstellen.
• Die Möglichkeit, den Installationsort von StrawberryPerl wählbar zu machen.

Dieser Punkt wurde durch einen Bug in Perl 5.10 verzögert. Perl 5.12 hat diesen Bug nicht mehr, so dass zukünftige Releases von StrawberryPerl an jeden Ort installiert werden kann.

Perl-Umfrage 2010

Kieren Diment hat die Perl-Umfrage 2010 abgeschlossen. Der Grant umfasste die Programmierung der Umfrage-Software, die Durchführung der Umfrage und die Auswertung der Ergebnisse. Die Umfrage wurde Ende Mai/Anfang Juni durchgeführt. Die erste Analyse der Daten wurde auf dem 12. Deutschen Perl-Workshop vorgestellt. Die Auswertung lag bei Redaktionsschluss für diese Ausgabe noch nicht vor.

Dist::Zilla Grant

Ricardo Signes hat seine Arbeiten für den Grant "Verbesserungen an Dist::Zilla" abgeschlossen. Während des Grants hat er an zusätzlichen Features gearbeitet aber auch den bestehenden Code überarbeitet. Neu hinzugekommen sind auch die Möglichkeiten des Loggings und die Tests wurden ausgebaut.

Dist::Zilla ist ein Programm/Modul, mit dem Releases und damit verbundene Aufgaben (z.B. Tagging im Code-Repository, Erzeugen der META.yml) automatisiert werden können.

Ein Tutorial zu dem Modul gibt es unter http://dzil.org/tutorial/start.html

Neue Grant-Manager

Das Grant-Komitee der Perl Foundation hat drei neue Grant-Manager:
• Tom Hukins
• Makoto Nozaki
• Alan Haggai Alavi

Gleichzeitig hören
• Adrian Horward
• Jeff Adam
als Grant-Manager auf.

Neue Regeln im Grant-Komitee

Der Prozess im Grant-Komitee der Perl Foundation wird ab sofort geändert. Zukünftig wird das Geld für Grants nur noch ausbezahlt, wenn der Grant auch abgeschlossen wird (Ausnahmen bedürfen einer Abstimmung im Komitee). Damit soll die aktuelle Situation verbessert werden, in der nur ein geringer Teil der Grants auch wirklich abgeschlossen werden. Zusätzlich gibt es jetzt klare Regeln, wann ein Grant vorzeitig abgebrochen wird.

Die neuen Regeln können unter http://news.perlfoundation.org/2010/05/new-grants-rules.html eingesehen werden.

Fixing Perl 5 Bugs

Mit seinem Grant möchte Mitchell 500 Stunden an Bugfixes für den Perl-Kern arbeiten.

Als ersten Schritt hat er die Bugreports im Bugtracker von Perl 5 getaggt und so ein leichtes Auffinden von Bugs nach Themen sortiert ermöglicht.

Als erste Gruppe hat Dave Mitchell an den "security/taint" markierten Bugs gearbeitet.

Insgesamt hat er über im März 74 Stunden an den Bugmeldungen gearbeitet und dabei 65 Bugmeldungen geschlossen - entweder nachdem er sie gefixt hat oder bei denen schon Patches vorlagen.

Dave hat im April über 52 Stunden an seinem Grant gearbeitet. Sehr viel Zeit hat er für einen Regex-Bug gebraucht. Auch seine bisherigen Fixes mussten noch leicht korrigiert werden. Es wurden 10 Bugs gefixt.

Mojo Dokumentation

Sebastian Riedel hatte in diesem Jahr gesundheitliche Probleme, die zu einer Pause bei dem Grant führten. Dennoch sollen die Arbeiten an der Mojo-Dokumentation weitergehen und Sebastian erwartet, dass er im Juni Ergebnisse abliefern kann.

Projekte für den Google Summer of Code

Google hat die Projekte für den Google Summer of Code 2010 bekanntgegeben. Auch die Perl Foundation ist in diesem Jahr wieder als Mentoren-Organisation dabei.

Diesmal wurden 10 Projekte der TPF angenommen:

- Daniel Arbelo Arrocha: NFG and single-representation strings for the Parrot Virtual Machine.
- John Harrison: Improvements to the NCI system and LLVM Stack Frame Builder
- Justin Hunter: Rework Catalyst framework's instance initialisation code to provide more flexible and extensible inversion of control
- Mirko Westermeier: Bulletproofing the Mojolicious unit and integration test suite
- Muhd Khairul Syamil Hashim: Implementing an Instrumentation Tool for Parrot VM for GSoC 2010
- Nat Tuck: Hybrid Threads for Parrot
- Pawel Murias: Releasing Mildew and SMOP on CPAN
- Ryan Jendoubi: Ctypes for Perl
- Tyler Curtis: A PAST Optimization Framework for Parrot
- Carl Masak: Adding support for binary data in Rakudo

Änderung des Perl 5 Optrees

Gerard Goossen hat den Optree bei der Kompilierung von Perl 5 Code in einen "Abstract Syntax Tree" umgewandelt.

Der Code ist unter http://github.com/ggoossen/perl/tree/codegen zu finden.

Zur Zeit sind die Änderungen noch nicht in den Perl-Kern eingeflossen und die Perl 5 Porters diskutieren ob und wie die Änderungen übernommen werden können.

Im Moment gibt es noch ein paar Tests, die fehlschlagen, aber Goossen arbeitet daran.

Bierdeckel

Mojo

Perl

Dancer

7 Stunden Vorträge

Perl-Magazin

Padre

round tuits

uvm.

am

22. August 2010

auf der

Veranstaltungsort der FrOSCon 2010:
Hochschule Bonn-Rhein-Sieg - Grantham-Allee 20 - 53757 Sankt Augustin - http://www.froscon.de

CPAN News XV

Test::XPath

Funktioniert meine HTML-Generierung? Das könnte die Frage sein, auf die `Test::XPath` die Antwort ist. Mit diesem Modul lassen sich solche Tests sehr einfach in die eigenen Unit-Tests integrieren. Mit dem Modul kann man überprüfen, ob bestimmte Elemente vorhanden sind oder ob diese Elemente auch den richtigen Wert haben. Der Zugriff auf die Elemente funktioniert - wie der Name des Moduls schon suggeriert - über XPath-Angaben.

```
use Test::More tests => 2;
use Test::XPath;

my $xml = <<'XML';
<html>
  <head>
    <title>Hello</title>
    <style type="text/css"
      src="foo.css">
    </style>
    <style type="text/css"
      src="bar.css">
    </style>
  </head>
  <body>
    <h1>Welcome to my lair.</h1>
  </body>
</html>
XML

my $tx = Test::XPath->new( xml => $xml );

$tx->ok( '/html/head',
     'There should be a head' );
$tx->is( '/html/head/title',
     'Hello',
     'The title should be correct' );
```

Try::Tiny

Auf ein "try-catch"-Konstrukt mittels `eval` bin ich in der Ausgabe 11 von $foo eingegangen. Eine schönere Möglichkeit, so eine Fehlerbehandlung zu machen, ist mit dem Modul `Try::Tiny`. Das Modul ist leichtgewichtig (hat keine komplexe Syntax und bis auf ein Core-Modul auch keine Abhängigkeiten). Durch Verwendung des Moduls bekommt man ein "sauberes" `eval` und man spart sich einige Zeilen.

```
# handle errors with a catch handler
try {
        die "foo";
} catch {
        warn "caught error: $_"; # not $@
};

# just silence errors
try {
        die "foo";
};
```

Image::Compare

Sind die zwei Bilder gleich? Handelt es sich um unterschiedliche Bilder? Eine Person kann sehen, ob zwei Bilder gleich sind. Für Perl-Programme gibt es `Image::Compare`. Für den Vergleich gibt es mehrere Verfahren und es kann festgelegt werden, in welchem Grad diese Bilder gleich sein müssen.

```
use strict;
use warnings;
use Image::Compare;

my($cmp) = Image::Compare->new();
$cmp->set_image1(
    img  => '/path/to/some/file.jpg',
    type => 'jpg',
);
$cmp->set_image2(
    img  =>
      'http://somesite.com/someimage.gif',
);
$cmp->set_method(
    method => &Image::Compare::THRESHOLD,
    args   => 25,
);
if ($cmp->compare()) {
    # The images are the same,
    # within the threshold
}
else {
    # The images differ beyond the threshold
}
```

Net::GitHub

Git wird auch unter Perl-Programmierern immer beliebter und auch die Quellen des Perl-Kerns werden mittlerweile in einem Git-Repository verwaltet. GitHub ist wohl die bekannteste Plattform für Git-Hosting. Mit `Net::GitHub` gibt es ein Modul, mit dem man aus einem Programm heraus auf ein Git-Repository zugreifen kann und so kann man sich zum Beispiel Informationen zu einem Commit holen:

```
use strict;
use warnings;
use Data::Dumper;
use Net::GitHub;

my $repo = Net::GitHub->new(
    owner => 'reneeb',
    repo  => 'BMATrainer',
);

my $info = $repo->commit->show(
'4960c586fb96376bf29a' );
print Dumper $info;
```

Text::Clip

Mit `Text::Clip` ist es möglich, Teile eines größeren Texts herauszuziehen. Natürlich kann man das auch selbst mit Regulären Ausdrücken machen, aber wenn man häufiger ab der selben Stelle einen Textausschnitt braucht, bietet `Text::Clip` eine angenehme Möglichkeit. Das Modul hat noch ein paar Schwächen; so wird immer die komplette Zeile mit dem Startmarker als Start erkannt. Hier ein Beispiel, wie man einen kleinen Teil aus einem POD-Text rausziehen kann:

```
use Text::Clip;

my $string = <<'POD';
=head1 Titel eines Artikels

hier kommt der Text des Artikels.
Der Artikel soll dann gedruckt
werden.

=head2 Subtitel
POD

my $marker = Text::Clip
             ->new(data => $string )
             ->find( qr/^=head1\s+/ );
my ($m2, $headline) = $marker
        ->find( qr/\r?\n/, slurp => '[)' );
print "Titel: $headline\n";

my ($m3, $abstract) = $marker
        ->find( qr/^=head2/, slurp => '[)' );
print "Abstract: ", $abstract, "\n";
```

GIS::Distance::Lite

Häufig geht es um die Frage "wie weit ist xy von yz entfernt?". Mit `GIS::Distance::Lite` kann diese Entfernung berechnet werden. Man braucht nur die Längen- und Breitenangaben im WGS84-Format und schon kann es losgehen. Diese Daten kann man z.B. bei Google Maps sehr einfach herausbekommen. Im Beispiel wird die (geographische) Entfernung zwischen dem Sitz des Bundespräsident und der Bundeskanzlerin berechnet.

```
use strict;
use warnings;
use GIS::Distance::Lite qw(distance);

my ($lat1,$lon1) = (52.51606939330561,
                    13.352422714233398);
my ($lat2,$lon2) = (52.52014324681717,
                    13.371005058288574);

my $distance = distance( $lat1, $lon1 =>
                         $lat2, $lon2 );

print $distance;
```

Termine

August 2010

03. Treffen Frankfurt.pm
Treffen Vienna.pm
04.-06. YAPC Europe in Pisa
05. Treffen Dresden.pm
09. Treffen Ruhr.pm
10. Treffen Stuttgart.pm
16. Treffen Erlangen.pm
18. Treffen Darmstadt.pm
21.-22. FrOSCon in St. Augustin
25. Treffen Berlin.pm
31. Treffen Bielefeld.pm

September 2010

02. Treffen Dresden.pm
07. Treffen Frankfurt.pm
Treffen Vienna.pm
13. Treffen Ruhr.pm
14. Treffen Stuttgart.pm
17.-18. FrOSCamp in Zürich
20. Treffen Erlangen.pm
28. Treffen Bielefeld.pm
29. Treffen Berlin.pm

Oktober 2010

01.-02. Kieler Linuxtage
05. Treffen Frankfurt.pm
Treffen Vienna.pm
07. Treffen Dresden.pm
11. Treffen Ruhr.pm
12. Treffen Stuttgart.pm
18. Treffen Erlangen.pm
20. Treffen Darmstadt.pm
26. Treffen Bielefeld.pm
27. Treffen Berlin.pm

Hier finden Sie alle Termine rund um Perl.

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

Uhrzeiten und weiterführende Links zu den einzelnen Veranstaltungen finden Sie unter

http://www.perlmongers.de

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

termine@foo-magazin.de

LINKS

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-nachrichten.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.perl-community.de

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

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

http://www.perl-workshop.de

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-foundation.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.

http://www.Perl.org

Smart-Websolutions

Windolph und Bäcker GbR

Perl-Programmierung

info@smart-websolutions.de

YAPC::Europe::2010

My One Hotel Galilei, Pisa, August 4-6

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

The Renaissance of Perl

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