GMD Report
94

GMD –
Forschungszentrum
Informationstechnik
GmbH

Jürgen Christoffel, Jost Krieger,
Alexander Sigel (Hrsg.)

Zweiter Deutscher Perl-Workshop

8. - 10 März 2000
Fachhochschule Rhein-Sieg
Sankt Augustin

März 2000

GMD –
Forschungszentrum Informationstechnik GmbH
Schloß Birlinghoven
D-53754 Sankt Augustin
Germany
Telefon +49 -2241 -14 -0
Telefax +49 -2241 -14 -2618
http://www.gmd.de

Anschriften der Herausgeber/Addresses of the editors:
Jürgen Christoffel
Port25 GmbH
D-53757 Sankt Augustin

Jost Krieger
Ruhr-Universität Bochum
Rechenzentrum
D-44780 Bochum

Alexander Sigel
Informationszentrum Sozialwissenschaften
Lennéstraße 30
D-53113 Bonn

ISSN 1435-2702

Abstract

Contributions to the Second German Perl-Workshop – Aim of the workshop was to further the collaboration between German speaking developers of Perl modules and advanced users.

Keywords: Perl, XML, Client Server Programming, CGI Programming, Dynamic Websites

Zusammenfassung

Beiträge zum Zweiten Deutschen Perl-Workshop – Ziel des Workshops war, den Austausch und die Vernetzung zwischen den Entwicklern von Perl-Modulen und fortgeschrittenen Anwendern im deutschsprachigen Raum zu fördern.

Schlüsselworte: Perl, XML, Client-Server-Programmierung, CGI-Programmierung, dynamische Websites

Programmkomitee/Chairs

Jürgen Christoffel
Norbert Grüner
Christian Kirsch
Jost Krieger
Marc Lehmann
Jörn Reder
Gerald Richter
Susanne Schmidt
Alexander Sigel

Deutscher Perl-Workshop 2.0

Inhalt

Perl und Mail
Übersicht und Beispiele eines Postmasters

Jost Krieger
Jost.Krieger+perl@ruhr-uni-bochum.de
Ruhr-Universität Bochum, Rechenzentrum

20. Februar 2000

Inhaltsverzeichnis

Während es an Programmen zum Versenden, Empfangen und Bearbeiten von Mail im Internet nicht mangelt, gibt es doch immer wieder spezielle Bedürfnisse, für die das *Rapid Prototyping* und, noch wichtiger, *Continuous Development* von Perl sich viel mehr anbieten als die Benutzung eines fertigen Produktes.

Hier werden die verschiedenen Schnittstellen zur Internet Mail (SMTP (RFC 821), Mail (RFC 822) usw.) angesprochen und einige Möglichkeiten gezeigt, von Perl aus auf diesen Schnittstellen aufzusetzen. In einigen Bereichen kann man sich anhand von praktischen Beispielen von den Möglichkeiten der einzelnen Wege überzeugen.

Als Postmaster hat man häufig mit Mengenproblemen zu kämpfen, die den Einsatz von dedizierten Perl-Programmen anregen. Es werden also einige Beispiele aus der Postmasterarbeit vorgestellt, insbesondere aus dem Bereich der Spambehandlung.

1 Einführung

Während große Teile der Versendung und Bearbeitung von Mail im Internet mit bereits vorhandenen (teils ausgefeilten) Programmen erfolgen, die auf die schnelle Bewältigung des Massebgeschäfts ausgelegt sind, gibt es auch Spezialaufgaben, die es sich anbietet, mit Hilfe von Perl zu automatisieren.

Zunächst teilen sich die automatisierbaren Bereiche der Mail-Verarbeitung in drei große Bereiche:

Mail-Tansport Übertragung der Mail durchs Internet nach RFC 821.

Mail-Zugriff Zugriff auf ausgelieferte Mail über POP3 (RFC 1725) oder IMAP (RFC 2060).

Mail-Bearbeitung Verarbeitung ankommender (oder bereits eingetroffener) Mail im Format von RFC 822 bzw. im MIME-Format (RFC 2045 ff.)

Im Abschnitt 2 gibt es eine Übersicht über die Module, die zur Behandlung dieser Aufgaben zur Verfügung stehen.

Meine persönlichen Erfahrungen erstrecken sich besonders auf den Postmaster-Bereich, deshalb folgt eine kleine Übersicht häufiger Aufgaben im Abschnitt 3.

Es folgen Beispiele aus der automatisierbaren Postmaster-Arbeit, zunächst ein einfaches Mailbearbeitungs-Beispiel (Abschnitt 4: vacation), ein Hinweis auf kombinierte Anwendungen (Abschnitt 5: Webmail), weitere Beispiele aus der Mail-Fehlerbehandlung (Abschnitte 6 und 7) und schließlich dann ein Beispiel für den Mail-Transport (Abschnitt 8: Relaytest).

2 Mail-Module

Im CPAN[1] stehen zahlreiche Module zur Verfügung, die bei der Automatisierung der Mail helfen können. Ich empfehle *sehr*, diese Module auch zu benutzen, statt das Rad neu zu erfinden, wenn nicht Wichtiges dagegen spricht. Die zu realisierenden Protokolle in den RFCs sind nicht immer sehr einfach, und nichtkonforme Mail-Programme gibt es schon genug.

Tabelle 1 zeigt die gebräuchlichsten Module in den verschiedenen Bereichen. Diese Module findet man alle im CPAN.

Im Augenblick scheint es keinen in Perl geschriebenen konkurrenzfähigen SMTP-Server für den generellen Gebrauch zu geben, ähnliches gilt für POP- und IMAP-Server.

3 Postmaster-Arbeit

Der Postmaster ist seit Beginn der Gechichte der Email der Ansprechpartner für Mail-Probleme (und manche andere), seine Rolle und Mailadresse ist sogar im RFC fixiert. Genausolange handelt es sich um eine typischen Teilzeit- bzw. Nebenaufgabe.

Die anstehenden Aufgaben lassen sich leicht in manuelle und automatisierbare aufteilen: Benutzeranfragen wie „Ich kann nur noch Mail versenden, aber keine mehr empfangen, was soll ich tun?“ müssen bis jetzt noch zunächst von Menschen verstanden werden, da hilft nur eine gutes Mail-Programm und eine Standard-Antwort-Bibliothek.

Falls man aber Postmaster für Tausende von Empfängern ist, fallen doch Aufgaben an, die man automatisieren (oder unerledigt lassen) muss.

Die auffälligsten Bereiche sind hier unzustellbare Mails (*bounces*), unerwünschte Werbe-Mails (*Spam*) und Überschreitungen der Speicherberechtigungen (*quota*). Das dritte Problem bietet sich zwar auch zur Perl-Bearbeitung an, hat aber weniger mit Mail direkt zu tun. Die beiden anderen Probleme werden uns weiter beschäftigen. Weiterhin gibt es beliebig viele Überwachungs- und Statistikprobleme, bei deren Auswertung Perl das natürliche Hilfsmittel ist.

Außerdem gibt es eine ganze Reihe Dienste, die man den Benutzern zur Verwaltung Ihrer Mail anbieten kann. Auch hier ist Perl häufig das Mittel der Wahl.

[1] `http://www.cpan.org`

Anwendung	*Client*	*Searver*
Mail-Transport (RFC 821)		
Standard Spezialanwendung	Net::SMTP	 NetServer::SMTP Net::SMTP::Server
Mail-Zugriff (POP3, IMAP)		
POP3 IMAP IMAP Verwaltung kombiniert	Mail::POP3Client Net::POP3 Mail::IMAPClient Net::IMAP Net::IMAP::Simple IMAPget IMAP::Admin Mail::Cclient	
Mail-Bearbeitung (RFC 822, MIME)		
einfach MIME	Mail::Internet (Mail-Tools) Mail::Folder MIME::Tools MIME::Lite	
Mail-Versendung kombiniert (RFC 822 / RFC 821 oder lokal)		
	Mail::Send Mail::Sender Mail::Sendmail Mail::Bulkmail	

Tabelle 1: Mail-Module

Die Umgebung, aus der die hier geschilderten Erfahrungen und Beispiele stammen, ist ein Universitätsrechenzentrum mit ca. 50000 potenziellen, ca. 40000 eingetragenen und ca. 12000 aktiven Benutzern. Alle Mails für die Universität (nicht nur für die eigenen Benutzer) werden über zwei Rechner geleitet (den zentralen Mailhost, eine Sun Ultra 2, sowie eine Sparc 5 als Backup zur Ausfallüberbrückung). Täglich werden ca. 90000 Mails transportiert. Als Transportsoftware (*MTA*) wird `qmail 1.03` eingesetzt.

Die verwendeten Perl-Skripts sind natürlich auf diese Umgebung abgestimmt, aber Algorithmen und Ideen sollten unabhängig von der Konfiguration sein.

4 Beispiel: vacation

Als Hilfsmittel für Benutzer (die dies über ein Web-Interface einstellen können), steht eine Perl-Variante des klassischen Vacation-Programms zur Verfügung, die ursprünglich von Larry Wall und Tom Christiansen stammt. Dieses Programm sendet (z.B. im Urlaubsfall) eine Benachrichtigung an den Absender eines ankommenden Briefes zurück.

Hier wurde nur der nicht benötigte Interaktiv-Modus abgeschaltet und einige Änderungen für die lokalen Gegebenheiten vorgenommen. Insbesondere wird die bevorzugte Mailadresse aus der Datei `.name` eingelesen (sie stimmt bei uns nicht mit der LoginID überein).

Inhaltlich wichtig ist an diesem Beispiel:

1. Die Urlaubsmeldung wird nur verschickt, wenn der Empfänger unter To: oder Cc: aufgeführt ist.
2. Jeder Absender erhält die Meldung nicht mehr als einmal in einem bestimmten Zeitraum (standardmäßig eine Woche).
3. Als Umschlagabsender (Return-Path) wird die leere Adresse eingesetzt. Dies ist sehr wichtig, um unbeabsichtigte Mail-Schleifen zu vermeiden. Bei allen Mails, die automatisch als Reaktion auf andere Mails erzeugt werden, sollte man so vorgehen, wenn kein guter Grund dagegenspricht.
4. Die Meldung wird an die Umschlagabsender-Adresse zurückgechickt, nicht an die From-Adresse. Auch dies ist eine Konvention, die bei allen Benachrichtigungen an den Absender einzuhalten ist. Sie vermeidet die lästigen automatischen Mails, die man manchmal beim Posten in eine Mailingliste erhält, selbst wenn diese Liste die Empfänger in das To:-Feld schreibt (was sie nicht sollte).

Technisch ist noch zu bemerken, dass die Mail-Bearbeitung höchst einfach ist. Der Header wird als ein Absatz (bis zur Leerzeile) eingelesen, umgebrochene Zeilen werden „entfaltet“, und dann werden benötigte Headerzeilen mit regulären Ausdrücken der Form `/^Feldname.*/im` untersucht. Der Quelltext folgt.

Listing 1: vacation

```
#!/usr/local/bin/perl -w –
#
# vacation program
# Larry Wall <lwall
#
# updates by Tom Christiansen <tchrist
#
# RUB adaption by Jost Krieger

use strict;                                                                    10

sub exitwarn;

my ($vacation, $user, $realname, $default_msg, $timeout, $opt_j);
my ($home, $header, $ok, $to, $cc, $from, $subject, $nsubject, $sender);
my ($edit, $msg, $name, $now, $lastdate);
my (@ignores, @aliases, %scale, %VAC);

$vacation = $0;
```

```
$vacation = '/var/qmail/bin/vacation' unless $vacation =~ m#^/#;

$user = $ENV{'USER'} || $ENV{'LOGNAME'} || getlogin || (getpwuid($>))[0];

$default_msg = <<'EOF';
I will not be reading my mail for a while.
Your mail regarding "$SUBJECT" will be read when I return.
EOF


@ignores = (
        'daemon',
         'postmaster',
         'mailer-daemon',
         'mailer',
         'root',
     );

# set-up time scale suffix ratios
%scale = (
        's', 1,
        'm', 60,
        'h', 60 * 60,
        'd', 24 * 60 * 60,
        'w', 7 * 24 * 60 * 60,
    );


while (@ARGV && $ARGV[0] =~ /^-/) {
    $_ = shift;
    if (/^-I/i) {      # eric allman's source has both cases
        chdir;
        &initialize;
        exit;
    } elsif (/^-j/) {
        $opt_j++;
    } elsif (/^-f(.*)/) {
        &save_file($1 ? $1 : shift);
    } elsif (/^-a(.*)/) {
        &save_alias($1 ? $1 : shift);
    } elsif (/^-t([\d.]*)([smhdw])/) {
        $timeout = $1;
        $timeout *= $scale{$2} if $2;
    } else {
        die "Unrecognized switch: $_\n";
    }
}

if (@ARGV) {
    $user = shift || $user;
}


push(@ignores, $user);
push(@aliases, $user);
die "Usage: vacation [username]\n" if $user eq '' || @ARGV;

$home = (getpwnam($user))[7];
die "No home directory for user $user\n" unless $home;
chdir $home || die "Can't chdir to $home: $!\n";

$timeout = 7 * 24 * 60 * 60 unless $timeout;

if (open(NAME, ".name")) {
    $realname=<NAME>;
    chomp($realname);
```

```
    close(NAME);
}
$realname ||= $user;

push(@ignores, $realname);
push(@aliases, $realname);

dbmopen(%VAC, ".vacation", 0666) || die "Can't open vacation dbm files: $!\n";


$/ = '';                        # paragraph mode
$header = <>;
$header =~ s/\n\s+/ /g;         # fix continuation lines

exitwarn "bulk" if $header =~ /^Precedence:\s*(bulk|junk)/im;
exitwarn "request" if $header =~ /^From.*-REQUEST@/im;

for (@ignores) {
    exitwarn "ignored" if $header =~ /^From.*\b$_\b/im;
}

($from) = ($ENV{RPLINE} =~ /^Return-Path: <(.*)>$/);
($from) = ($header =~ /^From\s+(\S+)/m) unless $from;  # that's the Unix-style From line
die "No RPLINE or \"From\" line!!!!\n" unless $from;

($sender) = ($header =~ /^From:\s+(.*)/m); # real from line
$sender =~ s/\s+$//;
$sender ||= $from;

($subject) = ($header =~ /Subject:\s+(.*)/m);
$subject = "(No subject)" unless $subject;
$subject =~ s/\s+$//;

$nsubject="Subject: This is a recording... [Re: $subject]";

($to) = ($header =~ /^To:\s+(.*)/m);
($cc) = ($header =~ /^Cc:\s+(.*)/m);
$to .= ', '.$cc if $cc;

#print STDERR "$from $sender $to $subject\n";
if (open(MSG,'.vacation.msg')) {
    undef $/;
    $msg = <MSG>;
    close MSG;
} else {
    $msg = $default_msg;
}
$msg =~ s/\$SUBJECT/$subject/g;         # Sun's vacation does this
$msg =~ s/\$SENDER/$sender/g;           # Sun's vacation does this

if ($msg =~ /^(Subject: .*)\n/) {
   $msg=$';
   $nsubject=$1;
}

unless ($opt_j) {
    foreach $name (@aliases) {
        $ok++ if $to =~ /\b\Q$name\E\b/i;
    }
    exitwarn "no alias found" unless $ok;
}

$lastdate = $VAC{$from};
$now = time;
if (defined($lastdate) && $lastdate ne '') {
    ($lastdate) = unpack("L",$lastdate);
```

```
    exitwarn "improper lastdate" unless $lastdate;
    exitwarn "too young" if $now < $lastdate + $timeout;
}

$VAC{$from} = pack("L", $now);
dbmclose(%VAC);


$ENV{QMAILUSER}=$realname;                                                    160
$ENV{QMAILHOST}="ruhr-uni-bochum.de";

open(MAIL, "|datemail $from") || die "Can't run datemail: $!\n";

print MAIL <<EOF;
Return-Path: <>
To: $from
$nsubject
Precedence: junk
                                                                              170
EOF
print MAIL $msg;
close MAIL;
exitwarn "mailed $from";

sub initialize {
    &zero('.vacation.pag');
    &zero('.vacation.dir');
    open(FOR, ">.forward") || die "Can't create .forward: $!\n";
    print FOR "\\$user, \"|$vacation $user\"\n";                               180
    close FOR;
    $edit=0;
    &make_default;
}

sub zero {
    my($FILE) = @_;
    open (FILE, ">$FILE") || die "can't creat $FILE: $!";
    close FILE;
}                                                                             190


sub save_file {
    my($FILE) = @_;
    local($_);

    open (FILE, $FILE) || die "can't open $FILE: $!";

    while (<FILE>) {
        push(@ignores, split);                                                200
    }
    close FILE;
}

sub make_default {
    return if $edit;
    open(MSG, ">.vacation.msg") || die "Can't create .vacation.msg: $!\n";
    print MSG $default_msg;
    close MSG;
}                                                                             210

sub save_alias {
    push @aliases, shift;
}

sub exitwarn {
    warn "vacation: $_[0]\n";
```

```
    while(<>){ }
    exit;
}
```

5 Webmail

Viele Neu-Benutzer scheinen sich mit einem Web-Browser erheblich wohler zu fühlen als mit einem Mail-Programm. Hier gibt es eine ganze Auswahl von Perl-Programmen, die ein Interface vom klassischen Perl-Bereich CGI zur Mail darstellen. Im Augenblick ist bei uns WWW-Mail im Einsatz. Dieses Programm leidet allerdings unter seinem Alter und benutzt nicht genügend der oben erwähnten Module. Dies macht sich dann in Feinheiten bemerkbar, z.B. funktionierte es nicht, wenn der Benutzer Sonderzeichen im Passwort hatte.

Eine Auswahl von Perl-Programmen zu diesem Zweck findet man z.B. in der Dokumentation von `acmemail`[2] unter *competition*.

6 Bounces

Eines der großen Probleme des Postmasters sind nicht zustellbare Mails, die zu sogenannten Bounces führen. Ursachen sind vielfältig:

- Tippfehler des Absenders
- Falsch weitergegebene Adressen
- Volle Mailboxen
- Nicht erreichbare Zielrechner
- usw.

Schon aus Datenschutzgründen sollte der Postmaster sich um solche (häufigen) bounces nicht kümmern müssen, schließlich wird ja der Absender benachrichtigt.

Etwas ganz anderes sind sogenannte Double-Bounces, also Mails, bei denen weder Empfänger noch Absender erreichbar sind. Dies deutet immer auf ein Problem hin, das den Eingriff des Postmasters benötigt. Ursachen sind hier z.B.:

- Tippfehler des Absenders, insbesondere bei Webmail
- Der Absender kennt seine Mail-Adresse nicht oder hat sie falsch in sein Mailprogramm eingetragen
- Der Absender hat eine volle Mailbox
- Der Absender verschickt riesige Mails, die werder in die Mailbox des Empfängers noch (als Bounce) in seine eigene Mailbox passen
- Das Mailsystem des Absenders ist fehlkonfiguriert
- Der Absender ist ein Spammer und hat absichtlich eine falsche Adresse angegeben

Natürlich kann man alle diese Fälle manuell behandeln, aber glücklicherweise lässt sich hier einiges automatisieren. Bei uns landen alle diese Mails in einem Perl-Skript, dass folgende Maßnahmen ergreift:

1. Ist die Mail groß, so wird versucht, sie auf 200 Zeilen abzuschneiden und sie noch einmal an den ursprünglichen Absender zu versenden.

[2] `http://www.astray.com/acmemail`

2. Ist der ursprüngliche Absender ein (vermutlich) lokaler Benutzer (erkenntlich am Host-Teil der Adresse), wird versucht, typische Fehler (z.B. Nicht-Ascii-Zeichen oder Blanks) in der Adresse zu korrigieren (mit Hilfe der aktuellen Liste der gültigen Mail-Adressen), und die Mail mit einem Vermerk über die Korrektur an die korrigierte Adresse weiterzuleiten.

3. Handelt es sich um einen lokalen, nicht existierenden Benutzer oder um einen nicht-lokalen Benutzer, so wird die (ja offensichtlich fehlerhafte) Adresse in eine Sperrliste aufgenommen, so dass dieser Absender einige Zeit lang kann keinerlei Mails mehr an unsere Hosts versenden kann. Diese Maßnahme führt bei einem gutwilligen Bochumer Absender dazu, dass er wenigstens die Fehlermeldung zu Gesicht bekommt, die man ja nicht zurückschicken konnte. Bei einem Spammer wir zumindest der Strom von Müllmail mit dem gleichen Absender unterbrochen.

Alle nicht weitergeleiteten Double-Bounces werden auf jeden Fall zur manuellen Inspektion (und Spambehandlung) aufgehoben.

7 Spam

Ein weiterer großer Problembereich sind unerwünschte Werbe- und Massenmails, bekannt als Unsolicited Commercial bzw. Bulk Email (UCE bzw. UBE), oder kollektiv (in Übertragung aus dem News-Bereich) als Spam.

Unsere Benutzer können sich wünschen (per WWW), vor solchen Mails möglichst geschützt zu werden. Die Realisierung ist hier relativ komplex und bedarf laufender Weiterentwicklung.

7.1 Entwicklung

Der ganze Bereich des Spam nimmt in den letzten Jahren nicht nur mengenmäßig zu (laut US-Schätzung sind dort 30% aller Mails Spam), sondern es findet auch ein technisches Wettrüsten zwischen Spammern und Spambekämpfern statt.

Hier ein kurzer Aufriss der technischen Entwicklung des Spamming:

- Zuerst verschicken Spammer Mails mit Ihrer eigenen Mailadresse als Absender und die Empfänger löschen einfach die (seltenen) unerwünschten Mails.

- Da die Anzahl der erfolgreichen Werbungen wohl nicht befriedigend ist, versenden die Spammer ihre Mails an immer mehr Empfänger (hauptsächlich an aus den News gesammelte Adressen). Dies hat mindestens zwei Effekte: Viele Adressen stimmen nicht, so dass viele Fehlermeldungen zurückkamen, und es wird lästig, so dass die Empfänger sich beschweren.

- Die Spammer gehen dazu über, als Absender Fantasieadressen zu verwenden, auch mit Fantasie-Domainnamen. Kontaktpunkte stehen in der Mail (häufig Web-Adressen oder Telefonnummern). Als Reaktion nehmen viele Empfänger keine Mails von nicht existierenden Domains mehr an. Außerdem gehen die Beschwerden jetzt an die Provider der Spammer, die diese daraufhin, soweit möglich, sperren, und von denen einige Spamschutz in Ihre Mailserver einbauen (wie Obergrenzen für die Anzahl der verschickten Mails pro Zeiteinheit).

- Spammer verschicken daraufhin ihre Mails mit Absendern existierender Domains und direkt von ihrer Anwählleitung aus (meist mit dazu gekaufter spezieller Spammersoftware). Viele Empfänger nehmen daraufhin keine Mails direkt von den (über Online-Listen bekannten) Anwählleitungen großer Provider mehr an. Einige Provider sperren in Ihren Routern den SMTP-Port 25 für abgehende Mails und zwingen damit alle Ihre Benutzer, über Ihre Mailserver zu gehen.

- Spammer entdecken verstärkt die *offenen Relays*, Mailserver in der ganzen Welt, die für beliebige Absender Mails an beliebige Empfänger weiterleiten. Auf diese Weise kommen die Mails aus anderen Richtungen, und der Spammer braucht seine Leitung nicht zu belasten (das Relay erhält eine Mail mit hundert Empfängern und erledigt die Arbeit). Jetzt gibt es auch Online-Listen im DNS (wie `MAPS`

RBL[3], ORBS[4] und MAPS RSS[5]), die offene Relays verzeichnen, um dagegen anzugehen. Außerdem verwenden belästigte Spam-Emfänger Software wie Sam Spade[6] oder Online-Dienste wie SpamCop[7], die helfen, mit Hilfe der Received-Header der Mail die wahre Quelle zu entdecken und sich beim Provider zu beschweren.

- Spammer gehen dazu über, Relays zu suchen, die nicht nur offen sind, sondern auch den Absender nicht korrekt (oder gar nicht) aufzuzeichnen. Außerdem wird versucht, die Empfänger so über die Relays zu streuen, dass verschiedene Kopien der Mail aus verschiedenen Richtungen kommen, verschieden absender haben und evtl. auch noch verschieden Subject-Zeilen. Die Empfänger versuchen, offene Relays möglichts schnell zu erkennen und möglichst viele davon zu schließen. Das ist ungefähr der heutige Stand.

Gleichzeitig gibt es Bestrebungen zur rechtlichen Bekämpfung des Spammings, die aber im internationalen Internet nicht schnell zu Ergebnissen führen werden. (Wenn ein amerikanischer Spammer über ein anonymes ostasiatisches Relay einen Empfänger in Deutschland belästigt, dürfte diesem selbst bei eindeutiger Rechtslage juristisch nicht zu helfen sein, und das ist ein typischer Fall.)

7.2 Realisierung: Überblick

Der Spambekämpfung dienen mehrere Maßnahmen

- Wie bereits geschildert, werden Mails von durch Double-Bounces bekannt falschen Absendern einige Zeit zurückgewiesen (als formal inkorrekt).

- Ebenso werden keine Mails aus falschen Domains angenommen.

Sämtliche weiteren Maßnahmen liegen unter der Kontrolle der Empfänger, insofern werden die Mails zunächst nur markiert (*Spam-Tagging*).

- Die Header der Mails werden gegen eine manuell gepflegte Liste von Spam-Signaturen geprüft, die bestimmte Spams auszeichnen (ob z.B. im Subject `University Diploma` vorkommt) oder die auf bestimmte Spam-Software hinweisen (einige Formen von falschen Datumsangaben). Dabei werden gleich auch einige Viren erkannt und ähnlich behandelt. Solche Mails werden sofort als verdächtig markiert.

- Die Absender-Felder werden gegen eine lokale Liste bekannter Spam-Absender (*Blacklist*) geprüft, die auch schon mal ganze Domains enthalten kann, z.B. `excte.com`, sowie gegen die aufgezeichnete Liste von ungültigen, bouncenden Adressen. Auch diese Mails sind unmittelbar verdächtig.

- Gleichzeitig werden die Absender-Felder gegen eine *Whitelist* geprüft, die aus manuell als unverdächtigt markierten Absendern (die schon mal in falschen Verdacht geraten waren) und aus Beantwortern der Relayanfragen (s.u.) besteht.

- Alle Received-Zeilen werden überprüft, ob IP-Adressen in der Online-Liste MAPS RSS von bereits missbrauchten offenen Relays vorkommen. Die erste externe Received-Zeile wird auch in der MAPS RBL und der MAPS DUL gesucht. Für solche Mails wird eine Warnung an den Absender zurückgeschickt, die ihn auf das offene Relay bzw. auf die Direkteinlieferung hinweist. Der Empfänger dieser Warnbriefe wird mittels `VERP` (Abschnitt 9) sowohl im Umschlag-Absender wie im From-Feld festgehalten und mit MD5 gesichert, um bei Bounces und Antworten entsprechend reagieren zu können. In diesen Fällen erfolgt eine *Hold*-Markierung.

[3] `http://www.mail-abuse.org/rbl`
[4] `http://www.orbs.org/`
[5] `http://www.mail-abuse.org/rss`
[6] `http://www.samspade.org`
[7] `http.//spamcop.net`

Die Markierungen werden als Header-Zeilen in die Mail eingefügt und haben folgende Form (künstlicher Umbruch):

```
X-Spam-Tag: type relay field Received origin relays.mail-abuse.org
            trigger 192.41.172.77 action hold issuer sunu450.rz.ruhr-uni-bochum.de
X-Spam-Tag: type signature trigger ADV-Spam action complain
            issuer sunu450.rz.ruhr-uni-bochum.de
```

Mit Absicht wurde auf die Analyse des Mail-Inhalts verzichtet, da dem sowohl Aufwands- wie Datenschutzbedenken entgegenstehen.

Nach dieser Markierung werden die Mails sofort an den Empfänger weitergeleitet. Dieser kann mit den Mails natürlich verfahren, wie er will, z.B. sie mit Filtern in seinem Mailklienten aussortieren oder die Spam-Markierung völlig ignorieren.

Unsere direkten Benutzer können sich (per WWW) folgende Standardbehandlung wünschen:

- *Verdächtige* Mails werden in eine spezielle Box des Postmasters gesandt, der dann die immer wieder auftauchenden *false positives* mit Vermerk weiterleitet und in die *Whitelist* aufnimmt, Zweifelsfälle ebenso mit einer Anfrage weiterleitet, und bei echten Spams je nach Fall einen *Blacklist*-Eintrag vornimmt, ein Beschwerdeverfahren einleitet, oder auch einen Relaycheck durchführt (siehe Abschnitt 8).

- Bei Mails in *hold* wird bis zu einem Tag abgewartet, ob eine Antwort oder ein bounce auf die automatsiche Anfrage eintrifft und die Mail damit entweder weißgewaschen oder schwarz gestrichen wird. Nach einem Tag ohne Reaktion wird von der Gutartigkeit des Absenders ausgegangen (wegen der sonst zu hohen *false positive*-Rate).

7.3 Technik des Spam-Tagging

Da **alle** ankommende Mail durch das Spam-Tag-Skript `ucecheck` läuft, wurde hier versucht, den Aufwand minimal zu halten, es werden noch nicht einmal die Mailtools verwendet. Die Black- und Whitelist werden in einer `dbm`-Datei gehalten, hauptsächlich um mit unserer Perl-Standard-Implementierung auszukommen. Im Prinzip wäre hier das CDF-Format vorzuziehen.

Hier ein Stück der inneren Schleife des Skripts.

Listing 2: ucecheck (Auszug)

```
my $line = '';
my $need_received=1;

while(<>) {
    chomp;
    last if /^$/;
    if (/^\s/) {
        s/^\s+/ /;
        $line .= $_;
    } else {                                                                    10
        handle_line($line);
        $line = $_;
    }
}
handle_line($line) if $line;

sub handle_line {
    my ($line) = @_;
    if($line =~ /^Received:\s+from\s+.*[\[(@](\d+\.\d+\.\d+\.\d+)[\])].*\s+by\s/i and $1 ne '134.147.32.36') {
        if($need_received) {                                                    20
            $need_received=0;
            $ip=$1;
        } else {
```

```
            my $tip=$1;
            if($tip && !$exempt_ips{$tip}) {
                for (keys %rlists) {
                    (&tag_spam(type => 'relay', field => 'Received', action => 'hold',
                     origin => $rlists{$_}, trigger => $tip), $relay=[$tip,$rlists{$_}]) if inlist($tip, $rlists{$_});
                }
            }
        }
    }
    if ($field=&addressline($line)) {
        if (($list, $reason)=&main::isspam($line)) {
            &tag_spam(type => 'blacklist', action => 'complain', field => $field, origin => $list, trigger => $reason);
            $white=1 if $list eq $whitehat;
        }
        if (lc($field) eq 'from') {
            $from=$1 if $line =~ /^from\s*:\s+(.*\S)\s*$/i;
            $from =~ s/\(.*?\)//g;
            $from=$1 if $from =~ /<(.*)>/;
        }
    }

    $bounce=0 if $line =~ /^from:.*(postmaster|mailer[-_]?daemon|listserv)/i;

    &tag_pat($line);
}
```

Und hier ein Teil der Spam-Pattern, man sieht auch die Gefahr von *false positives*, schließlich könnte ja wirklich jemand über *Prosperous Future* schreiben.

Listing 3: spampats (Auszug)

```
^X-Spanska: Yes virus   Happy99-Virus
^Subject:\s+Check this\s*$      virus   VBS/Freelink-Virus
^(?i)To:\s+friend@public.com    complain        Public-Spam
^Date: .* Pacific Daylight Time complain        FunnyDate-Spam
^(?i)From: \b[0-9]\S*@hotmail.com       complain        FakeHotmail-Spam
^Subject: Prosperous Future     complain        Prosperous-Spam
^(?i)Subject:.*\s+UNIVERSITY\s+DIPLOMA  complain        Diploma-Spam
^(?i)Subject:\s+Royalebank[A-Z] complain        Royalebank-Spam
Approved-By: aleph1@SECURITYFOCUS.COM   whitehat        BUGTRAQ-Moderation
```

Das Skript liest nur den Header der Mail, faltet dabei Folgezeilen nach RFC822 zusammen und analysiert nur die interessanten Zeilen. Dabei werden alle Spam-Tag-Zeilen gepuffert und erst komplett ausgegeben, wenn die Analyse beendet ist (schon wegen möglicher Whitelist-Einträge).

7.4 Technik des Benutzer-Filters

Bei jedem Benutzer, der den Spam-Filter eingeschaltet hat, läuft ein Skript `block` über die Mail, dieses liest noch einmal die Header, wertet die Spam-Tags aus sowie die persönliche White- und Blacklist des Benutzers. Auch die Adressfelder werden noch einmal überprüft. Danach wird je nach der stärksten Einstufung in folgender Priorität verfahren:

1. Die *Whitelist* des Benutzers (wenn er Mails will, die alle anderen als Spam empfinden, aber nicht ganz auf den Filter verzichten).

2. Die *Blacklist* des Benutzers. Solche Mails werden mit Fehlermeldung zurückgesandt, es muss sich ja nicht unbedingt um Spam handeln.

3. *Verdächtig*-Markierungen. Diese Mails werden in die Spambox des Postmasters umgeleitet.

4. *Hold*-Markierungen. Wenn dies die stärkste Markierung ist wird geprüft, ob inzwischen eines der Absender-Felder in der *Blacklist* oder der *Whitelist* aufgetaucht ist. In diesem Fall wird entprechend verfahren. Ansonsten wird die Mail nach Ablauf eines Tages zugestellt (demnächst mit einer Text-Markierung für den Empfänger).

5. Keine Markierung (der Normalfall). Die Mail landet unbesehen in der Empfänger-Mailbox. Hierzu zählen auch die durch die globale Whitelist reingewaschenen Mails.

Hier das Gerüst des `block`-Skripts:

Listing 4: block (Auszug)

```
$gaction='pass';
$msgid='';

&setgaction('bounce', 'blacklist', $reason) if $sender and $reason=&main::islspam($sender);

$line = '';
while(<>) {
    last if /^$/;
    if (/^\s/) {
        s/^\s+/ /;
        $line .= $_;
    } else {
        &handle_spam_tag($line) if $line =~ /^X-Spam-Tag:/;
        $msgid=$1 if /^Message-ID:\s*<(.*)>/i;
        &setgaction('bounce', 'blacklist', $reason) if &addressline($line) and $reason=&main::islspam($line);
        &reclassify($line) if      $gaction eq 'hold' and &addressline($line);
        $line =$_;
    }
}

&handle_spam_tag($line) if $line =~ /^X-Spam-Tag:/;
$msgid=$1 if /^Message-ID:\s*<(.*)>/i;
&setgaction('bounce', 'blacklist', $reason) if &addressline($line) and $reason=&main::islspam($line);
&reclassify($line) if      $gaction eq 'hold' and &addressline($line);

&reclassify($sender) if     $gaction eq 'hold' and $sender;

dbmclose(%db) if $dbisopen;

#    check if too old
$gaction='pass' if $gaction eq 'hold' and $currentnotify;

&setgaction('complain', 'holdexpire', $gtrigger) if $gaction eq 'hold' and -M STDIN >= $holdtime;

$gaction='pass' if $gaction eq 'inspect';

if ($gaction ne 'pass' and $gaction ne 'hold') {
  open (LOG, '>>block.log');
  print LOG "$tstamp\t$sender\t$recp\t$msgid\t$gaction\t$gtype\t$gtrigger\n";
  close (LOG);
}

if ($gaction eq 'hold') {
    seek (STDIN, 0, 0);
    exit 111;
}

if ($gaction eq 'drop') {
    seek (STDIN, 0, 0);
    exit 99;
```

```
}

if ($gaction eq 'bounce') {
    seek (STDIN, 0, 0);
    system "bouncesaying 'The addressed user does not wish to receive this mail'";

    exit 99;
}

if ($gaction eq 'complain') {                                                    60
    seek (STDIN, 0, 0);
    system 'forward caughtspam';
    exit 99;
}

while(<>) {};

exit 0;
```

8 Beispiel: Relaytest und -beschwerde

Wenn der Postmaster seine Spambox durchgesehen (und *false positives* weitergeleitet hat), bleibt ihm ein Haufen an Spam zurück. Viele dieser Spams sind schon markiert als *Relay-Spam*, andere lassen sich manuell als solche erkennen. Für diese Fälle gibt es ein Tool `relaycrash`, dass über die Kommandozeile eine IP-Adresse (oder einen Hostnamen) und eine Datei mit dem Spam erhält und dann den angegebenen Host testet, ob er wirklich ein offenes Relay ist. Hierzu wird mittels Net::SMTP eine direkte Verbindung zu dem Host aufgebaut und versucht eine Mail (mit einer Anmerkung und dem ursprünglichen Spam) durch diesen Host an den eigenen Rechner zu relayen. Wenn dies zu gelingen scheint, wird auch gleich versucht, diese Meldung auf demselben Weg an ORBS und MAPS RSS weiterzuleiten (falls der Host nicht schon dort eingetragen ist). Es gibt hier mehrere Komplikationen:

1. Nur weil ein Host eine Mail annimmt, heißt das noch nicht, dass er relayt. Man muss schon auf die Ankunft der Mail warten.

2. Es gibt viele Möglichkeiten des Relayens. Bestimmte Sendmail-Versionen weigern sich z.B., Mails an `Meine.Adresse@ruhr-uni-bochum.de` weiterzuleiten, nicht aber bei der Adresse `"Meine.Adresse@ruhr-uni-bochum.de"`. Sehr viele Mailer haben aus historischen Gründen den *Prozent-Hack* aktiviert, der erlaubt, Mails mittels `Meine.Adresse%ruhr-uni-bochum.de@domain.des.relays` weiterzuleiten. Daher ist das Skript `relaycrash` mit einer Tabelle ausgestattet, die es auch erlaubt, solche Kombinationen nachzurüsten, wenn sie neu bekannt werden.

3. Die beiden vorigen Punkte haben zur Folge, dass das Skript solange testet, bis eine Mail angenommen wird. Da dies ja ein falscher Alarm sein kann, gibt es eine Restart-Möglichkeit.

Wieder ein Code-Auszug:

Listing 5: relaycrash (Auszug)

```
#
# ...
#
# personal configuration

my $ident  = "Jost.Krieger";
my $domain = "ruhr-uni-bochum.de";
my $fromfix = "+relaytest";
my $infix   = "+relaytest-in-";
my $outfix  = "+relaytest-out-";                                                 10
```

```
# end configuration

# test configuration

my @pattern = (
               ['$sender\@$sdomain', '$recip\@$rdomain'],
               ['$sender\@$sdomain', '\"$recip\@$rdomain\"'],
               ['$sender', '$recip\@$rdomain'],
               ['', '$recip\@$rdomain'],                                    20
               ['$sender\@$sdomain', '$recip%$rdomain\@$qdomain'],
               ['$sender\@$sdomain', '$recip\@$rdomain\@$qdomain'],
               ['$sender\@$sdomain', '\@$qdomain:$recip\@$rdomain'],
               ['$sdomain!$sender', '$rdomain!$recip'],
               ['$sender', '$rdomain!$recip'],
               ['$sender\@$qdomain', '$recip\@$rdomain'],
              );
# end test configuration
# reporting configuration
                                                                            30
my %lists=(
           MAPS_RBL => [ "rbl.maps.vix.com", "",     "",              CHECK ],
           ORBS     => [ "relays.orbs.org",  "relays", "orbs.org",    REPORT ],
           MAPS_RSS => [ "relays.mail-abuse.org", "Jost.Krieger+b2rss", "ruhr-uni-bochum.de", REPORT ],
          );

# end reporting configuration

use Net::SMTP;
use Socket;                                                                 40

#
# ...
#

my $smtp = Net::SMTP->new($relayip, Timeout => 120, Hello => "randomhost.$domain", Debug => $opt_d);
die "failed: $@\n" unless $smtp;

push @log, $smtp->banner;
warn $smtp->banner;                                                         50

push @log, $smtp->message;
warn $smtp->message;

addrelaynames ( $smtp->domain);

my $sender="$ident$infix";
my $sdomain=$domain;
my $recip="$ident$outfix";
my $rdomain=$domain;                                                        60

my ($envs, $envr);

my $done=0;
my $first=1;

trial:
for $pat (@pattern) {
  my ($qptr, $qdomain);
                                                                            70
  if (${$pat}[0] =~ /qdomain/ || ${$pat}[1] =~ /qdomain/ ) {
    $qptr=\@relaynames;
  } else {
    $qptr = [ $relay ];
  }

  for $qdomain (@{$qptr}) {
```

```
    sleep(2) unless $first||$search;
    $first=0;

    $res=$smtp->reset;

    $envs = eval "\"${$pat}[0]\"";

    $res=$smtp->mail($envs);
    push @log, $smtp->message;
    warn $smtp->message;
    (warn "failed: mail\n"), next unless $res;

    $envr = eval "\"${$pat}[1]\"";
    $searchn--, next if $searchn && defined($search) && $search eq $envr;
    next if $search && $searchn;
    $res=$smtp->to($envr);
    push @log, $smtp->message;
    warn $smtp->message;
    next unless $res;

    if ($report) {
      for (keys %lists) {
        if (${$lists{$_}}[3]>1 && ${$lists{$_}}[1]) {
          warn "Reporting to $_.\n";
          my $recip=${$lists{$_}}[1];
          my $rdomain=${$lists{$_}}[2];
          $res=$smtp->to(eval "\"${$pat}[1]\"");
          push @log, $smtp->message;
          warn $smtp->message;
          next trial unless $res;
        }
      }
    }
    send_mail(), $done=1 if $res;
    last trial if $res && !$opt_F;
  }
}


$res=$smtp->quit;
warn $smtp->message;

die "failed: rcpt\n" unless $done;

exit 0;
```

Wenn nun ein Relay als solches bekannt ist und in die Listen eingetragen wird, wäre es natürlich schön, wenn das Problem behoben würde.

Dazu werden alle alle zurückgekommenen Relaytests ausgewertet und versucht, den Zuständigen des Relays zu benachrichtigen mit einem Brief, der Hinweise, den Relaytest und den Spam enthält. Das Relay selbst ist mittels `VERP` (siehe Abschnitt 9) im Absender des Relaytests kodiert, an die Information über den Zuständigen wird versucht, mittels DNS, Whois und abuse.net zu gelangen.

9 Variable Envelope Return Path (`VERP`)

`VERP` ist eine Technik, die das Problem umgehen hilft, dass Zustellfehlermeldungen normalerweise nicht in einem Standardformat ankommen und oft genug keinen Hinweis auf die ursprünglich unzustellbare Adresse enthalten.

Zu diesem Zweck kodiert man den Empfänger in der Umschlagabsender-Adresse z.B. in der Form `Absender.Adresse+Empfaenger.Adresse=Empfaenger.Domain@Absender.Domain`,

wenn der eigentliche Absender die Adresse `Absender.Adresse@Absender.Domain` und er Empfänger die Adresse `Empfaenger.Adresse@Empfaenger.Domain` hat.

Solche plussed Adressen wurden wohl an der CMU erfunden und sind heute weit verbereitet, werden auch z.B. von aktuellen Sendmail-Versionen unterstützt.

Die häufigste Anwendung findet das Verfahren bei Mailinglisten, da es eine automatische Pflege der Listenteilnehmer möglich macht, wie z.B. bei `ezmlm` implementiert.

Das Verfahren ist aber immer dann von Vorteil, wenn es auf eine automatische Erkennung von Bounces ankommt, wie bei den Probebriefen der Spambehandlung.

10 Ausblick

Im Mailbetrieb treten immer wieder neue unerwartete Effekte auf, und als Postmaster erlebt man immer wieder, dass Vorgänge zunächst einmal, dann mehrfach und schließlich häufig auftreten, die sich automatisieren lassen. Häufig führt dies zu den berühmten Perl-Einzeilern (Zeilenlänge bis ca. 300), die sich dann zu täglich gebrauchten Skripts entwickeln.

Für die meisten dieser Probleme ist typisch, dass sich die Anforderungen ungefähr gleich schnell verändern wie das Programm zur Lösung entwickelt werden kann. Der schnelle Entwicklungszyklus macht Perl zum idealen Hilfsmittel, um nicht ins Hintertreffen zu geraten.

XS - Einführung und esoterische Anwendungen

Marc Lehmann
pcg@opengroup.org

12. Februar 2000

Inhaltsverzeichnis

XS, die C/C++-Schnittstelle von Perl ist einfach in der Benutzung und dennoch sehr mächtig. Der erste Teil dieses Vortrags stellt Schritt für Schritt die Erstellung einer Schnittstelle zum Uniforum-Katalog-Standard („gettext“).

Der zweite Teil stellt spezielle Probleme vor, die recht häufig vorkommen und gibt Lösungsvorschläge, sowie einige Tips & Tricks. Das Ziel dieses Teiles ist es, aufzuzeigen, welche Lösungen existieren, und wo man nachschauen muß, falls man auf ein Problem stößt, das schon von anderen gelöst wurde…

1. XS - Wozu, wenn's doch auch `system` gibt!

Ehrlich gesagt, mir kam diese Frage (oder auch eine ähnliche) nie in den Sinn. Tatsächlich aber ist sie durchaus angebracht: Perl wurde als Klebstoff entwickelt, um bestehende Programme miteinander verbinden zu können (naja, zumindest kann Perl das sehr gut, ob es speziell dafür entwickelt wurde…).

Man kann die meisten Probleme *durchaus* ohne XS-Programmierung lösen, ja, im Idealfall hat sich schon jemand anderer die Arbeit gemacht, und man muß nur ein paar Zeilen in der CPAN-Shell (tolles Tool!) eingeben, um es zu installieren (plus vielleicht ein zwei Stunden Anpassungen an sein System ;)

Ich sehe zwei Hauptgründe, für die XS-Programmierung:

- „RAW SPEED“. `system()` und Konsorten sind denkbar langsam. Manche Dinge lösen sich in C (C++..) einfach von selbst, und vor allem schnell. Reguläre Ausdrücke und die ArithmetikOperatoren von Perl sind nicht grundlos in C geschrieben.

- FAKTORISIERUNG. Das Wort, das sonst nur Forth-Programmierer verwenden. Man könnte es auch FLEXIBILITÄT nennen. Ein externes Programm läßt sich über Config-Dateien, über Kommandozeilenswitches und STDIN füttern und lenken. Den Programmablauf selbst kann man nur sehr schlecht steuern (etwa so, als hätte man nur eine einzige Funktion mit vielen Parametern).

Mit XS kann man einen Vorgang (z.B. dekodieren von Binaries in Mail oder News) in viele kleine Arbeitsschritte zerlegen. Das erlaubt große Flexibilität. auf der Programmiererseite: Nichts ist schlimmer als ein Programm, daß *im Prinzip* funktioniert, sich aber nicht genügend fernsteuern läßt.

Darüberhinaus sind XS-Funktionen so weit in Perl integriert, daß sie sich wie eingebaute Funktionen verhalten. So kann man sich eine „Sprache“ für ein spezielles Problem maßschneidern. Das ist eleganter und schöner als irgendwelche Hacks mit externen Programmen, die zu einem Pflege-Horror werden.

Außerdem gibt es Dinge, die man einfach nicht mit externen Programmen tun kann - GUI-Toolkits programmieren zum Beispiel. Klar, es geht schon - wenn man sich durch das Labyrinth einer unangepaßten API wühlen möchte…

1.1. Übersicht über die Möglichkeiten mit XS

Mit XS kann man grundsätzlich alles machen, was man auf der Perl-Ebene auch kann: Skalare, Hashes, Arrays…Perl-Objekte, Überladen (mehr fällt mir im Moment nicht ein). Wenn etwas fehlt, kein Problem, man kann immer Perl selbst aufrufen.

Darüberhinaus hat man vollen Zugriff auf die Sprache C (bzw. C++) und alle Bibliotheken, die dafür verfügbar sind (eine Menge). Es ist praktisch die einzige Möglichkeit, auf Bibliotheken in anderen Sprachen zurückzugreifen.

Ein häufig vorkommendes Problem ist beispielsweise die Schnittstelle zu einer Bibliothek, die einen internen Zustand speichert, z.B. `Compress::Zlib` oder `Digest::MD5`. Je nach Aufwand, den man investieren will und der Flexibilität, die man benötigt, bieten sich drei Wege an:

- Der interne Zustand (meist eine `struct` oder eine C++-Klasse) wird „einfach“ in einen Skalar verpackt - entweder direkt oder nur der Pointer. Wenn man dieses Objekt (z.B. auf der Perl-Ebene) mit `bless` in ein Objekt verwandelt, kann man sogar ein sehr schönes Interface basteln und häßliche organisatorische Aufrufe (wie das Freigeben) in der `DESTROY`-Methode verstecken.

- Der interne Zustand wird ebenfalls einfach verpackt. Zusätzlich werden Zugriffsmethoden („Mutatoren“) für alle internen Felder definiert.

- Das kann man auch alles auf C bzw. XS-Ebene. Das erfordert etwas mehr Wissen, ist aber vom Aufwand in etwa gleich.

2. Integration von `gettext` in Perl

gettext ist eine weitverbreitete Methode zur Lokalisierung von Texten in Programmen. In C ist die Anwendung denkbar einfach. Statt:

```
printf ("The output is %d%s\n", weight,
        metric ? "kg" : "st");
```

Schreibt man einfach:

```
printf (gettext("The output is %d%s\n"), weight,
        metric ? gettext("kg") : gettext("st"));
```

d.h. man ruft bei jeder String-Ausgabe die Funktion gettext auf und verwendet den Rückgabewert. Je nach ausgewählter (Ziel-) Sprache gibt `gettext` entweder den String selbst zurück (wenn er nicht übersetzt werden konnte) oder eine entsprechende Übersetzte Variante.

Vor der Benutzung von `gettext` muß das ganze System noch initialisiert werden, damit das System weiß, welchen Katalog es benutzen soll und wo die Tabellen gespeichert sind:

```
setlocale (LC_ALL, "");
textdomain ("meine-domain");
bindtextdomain ("meine-domain", "/usr/local/share/locale");
```

Das Beispiel läßt sich übrigens noch schöner schreiben, wenn man ein:

```
#define _(s) gettext(s)
```

verwendet. _ ist ein ungewöhnlicher Name, aber für diesen Zwecke sehr gut geeignet, und vor allem schon „Standard“:

```
printf (_("The output is %d%s\n"), weight,
        metric ? _("kg") : _("st"));
```

Nun gibt es schon ein (relativ spartanisches) Perl-Modul `Locale::gettext`, das ein Interface für `gettext` zur Verfügung stellt. Gäbe es dieses Modul jedoch nicht, wäre man praktisch aufgeschmissen: `gettext` kommt mit vielen Programmen daher, die das Management von „Katalogen“ (Übersetzungstabellen) vereinfachen. Würde man sich eine „nur-perl“ Lösung einfallen lassen, könnte man diese vorhandenen Hilfen nicht benutzen.

Ich weiß nicht, wie das `Locale::gettext`-Modul entstand. Aber ich weiß, das es kein großer Aufwand ist, wenn man die vorhandenen Hilfen in Perl ausschöpft.

2.1. Ein CPAN Modul besteht aus. . . ?

Wenn man ein Modul von CPAN holt, wird einem auffallen, daß die meisten Module einem recht strengen Standard folgen: Es gibt eigentlich immer die Dateien `Makefile.PL`, ein `MANIFEST`, ein `Changes` ein Test-Skript und eine Perl-Moduldatei (Endung `.pm`) und im Falle von XS-Modulen auch eine Datei `*.xs`.

Der Grund ist, daß man seine Module nicht selbst „von Grund auf“ schreibt, sondern als ersten Schritt das Programm `h2xs` („header-nach-xs“) verwendet: Als Eingabe nimmt es eine C-Header-Datei und erzeugt ein komplettes Verzeichnis mit allen notwendigen Dateien - schon installationsfähig.

Das Headerfile ist in meinem Falle `/usr/include/libintl.h`. Es enthält (fast) alle Funktionsdefinitionen, auf die es ankommt. Probieren wir's (das Modul `C::Scan` muss unbedingt installiert sein!):

```
cerebro:~/src# h2xs -x -A -n Locale::gettext /usr/include/libintl.h
Scanning typemaps...
 Scanning /usr/app/lib/perl5/ExtUtils/typemap
Scanning /usr/include/libintl.h for functions...
Writing Locale/gettext/gettext.pm
Writing Locale/gettext/gettext.xs
Writing Locale/gettext/Makefile.PL
Writing Locale/gettext/test.pl
Writing Locale/gettext/Changes
Writing Locale/gettext/MANIFEST
```

Falls es hier Probleme gibt, ist meistens C::Scan dafür verantwortlich. Es ist bei der Installation etwas bockig (`make test` läuft oft nicht durch) und erkennt viele Konstrukte nicht: Wenn es nicht richtig klappt (so furchtbare Dinge wie C__const> oder auch `const` verwirren `C::Scan` leider sehr), sollte man den Header kopieren und ein bißchen editieren (oder umbenennen, da h2xs nur mit .h-Dateien funktioniert).

Je nachdem, wie „verständlich“ das Headerfile geschrieben war, erzeugt h2xs (mit C::Scan) für einige oder alle Funktionen eine Perl-Schnittstelle.

2.1.1. Woran erkennt man, ob es geklappt hat?

Schauen wir die neuen Dateien einmal an: `Changes` ist sehr einfach:

```
Revision history for Perl extension Locale::gettext.

0.01  Sat Feb  5 22:26:58 2000
        - original version; created by h2xs 1.19
```

Auch `MANIFEST` hat nicht viel zu bieten:

```
Changes
MANIFEST
Makefile.PL
gettext.pm
gettext.xs
test.pl
```

Was in Ordnung ist: `MANIFEST` enthält nur eine Liste der Dateien, die Teil des Paketes sind. Dadurch kann bei der Installation festgestellt werden, ob Dateien fehlen. Vor allem aber kann man einfach `make dist` verwenden, um sein Perl-Modul in eine versendbare Form zu bringen.

`test.pl` ist ein einfaches Test-Skript, das bei einem `make test` ausgeführt wird und sicherstellt, daß das Erweiterungsmodul geladen wird.

Das erzeuge `Makefile.PL` enthält auch nicht viel, daß angepaßt werden müßte:

```
use ExtUtils::MakeMaker;
# See lib/ExtUtils/MakeMaker.pm for details of how to influence
# the contents of the Makefile that is written.
WriteMakefile(
    'NAME'   => 'Locale::gettext',
    'VERSION_FROM' => 'gettext.pm', # finds $VERSION
    'LIBS'   => [''],   # e.g., '-lm'
    'DEFINE' => '',     # e.g., '-DHAVE_SOMETHING'
    'INC'    => '',     # e.g., '-I/usr/include/other'
);
```

Einzig und allein das Attribut `LIBS` könnte auf den Wert `['-lintl']` gesetzt werden, da sich die `gettext`-Funktionen auf vielen Systemen in dieser Bibliothek verbergen.

Nun zu `gettext.pm`: Hier sollte man hier sofort Hand anlegen, vor allem bei der Dokumentation (das Listing befindet sich im Anhang). In diesem Falle sollten zu `@EXPORT` außerdem noch einige Funktionen, z.B. `textdomain` und `bindtextdomain`, hinzugefügt werden, damit sie bequem importiert werden können.

Perl-Module kann aber jeder... das eigentlich Interessante verbirgt sich in `gettext.xs`:

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

#include <libintl.h>


MODULE = Locale::gettext                PACKAGE = Locale::gettext


char *
gettext(msgid)
        const char * msgid

char *
dgettext(domainname, msgid)
        const char * domainname
        const char * msgid

char *
textdomain(domainname)
        const char * domainname

char *
bindtextdomain(domainname, dirname)
        const char * domainname
        const char * dirname
```

Hmm.. sieht eigentlich nicht viel anders als ein C-Headerfile aus. Normalerweise muß man hier einige kleine Korrekturen vornehmen (Position des Headerfiles, löschen überflüssiger Funktionen etc..).

2.2. Benutzung

Probieren geht über studieren. So, wie das Modul aussieht, könnte es glatt funktionieren... probieren wir mal:

```
cerebro:~/src/Locale/gettext# perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Locale::gettext
cerebro:~/src/Locale/gettext# make
/usr/bin/perl -I/usr/app/lib/perl5/i686-linux -I/usr/app/lib/perl5
/usr/app/lib/perl5/ExtUtils/xsubpp  -typemap
/usr/app/lib/perl5/ExtUtils/typemap gettext.xs > gettext.xsc && mv
gettext.xsc gettext.c
Error: 'const char *' not in typemap in gettext.xs, line 13
Error: 'const char *' not in typemap in gettext.xs, line 17
Error: 'const char *' not in typemap in gettext.xs, line 18
Error: 'const char *' not in typemap in gettext.xs, line 22
Error: 'const char *' not in typemap in gettext.xs, line 26
Error: 'const char *' not in typemap in gettext.xs, line 27
Please specify prototyping behavior for gettext.xs (see perlxs manual)
make: *** [gettext.c] Error 1
```

Nun gut. Perl kennt „const char *“ nicht. Aber wer braucht das auch? ;) Nach dem Entfernen aller „const“'s aus `gettext.xs` funktioniert es:

```
cerebro:~/src/Locale/gettext# make
[viele Zeilen Ausgabe, aber keine Fehlermeldung]
cerebro:~/src/Locale/gettext# make test
PERL_DL_NONLAZY=1 /usr/bin/perl -Iblib/arch -Iblib/lib  \
-I/usr/app/lib/perl5/i686-linux -I/usr/app/lib/perl5 test.pl
1..1
ok 1
```

So weit, so gut:

```
cerebro:~/src/Locale/gettext# make install
Installing /usr/app/lib/i686-linux/auto/Locale/gettext/gettext.so
Installing /usr/app/lib/i686-linux/auto/Locale/gettext/gettext.bs
Installing /usr/app/lib/i686-linux/Locale/gettext.pm
Installing /usr/app/man/man3/Locale::gettext.3
```

Und nun der erste, echte, Test. Auf meinem System gibt es eine textdomain namens `libc`, die unter anderem die Übersetzung von Fehlermeldungen enthält. Probieren wir aus:

```
use Locale::gettext;

# initialisierung

textdomain "libc";

print gettext("write incomplete"), "\n";
```

Das Beispiel geht davon aus, daß `gettext` und `textdomain` im `@EXPORT`-Array eingetragen wurden. Ausprobieren:

```
$ perl first-test
write incomplete
$ LANG=de perl first-test
Der 'Write' wurde nur unvollstandig ausgefuhrt
```

```
$ LANG=es perl first-test
escritura incompleta
$ LANG=sv perl first-test
ofullstanding skrivning
```

Ich finde, die deutsche Übersetzung ist am besten ;) Aber davon abgesehen: bisher musste keine einzige Zeile Perl oder C geschrieben werden (von Schönheitskorrekturen mal abgesehen).

Grundsätzlich funktioniert das Modul aber ... Zeit, für Verbesserunge!

2.2.1. `setlocale` (I)

Das der erste Versuch klappt, liegt daran, daß Perl `setlocale` aufruft - zumindest in meiner Version von Perl. Der Standardaufruf (der nicht ohne gewichtige Gründe anders ausfallen sollte) dafür lautet normalerweise:

```
setlocale(LC_ALL, "");
```

Das überschreibt aber alle Einstellungen, und die Nutzer unseres Moduls sind aber erst einmal an Texten (LC_MESSAGES) interessiert. Es wäre also sehr sinnvoll, wenn beim ersten Laden der Erweiterung automatisch ein `setlocale(LC_MESSAGES, „")` ausgeführt würde. Das erreicht man mit der `BOOT:`-Anweisung, die man irgendwo in den XS-Teil der Datei `gettext.xs` unterbringt (der erste Teil der besteht aus normalem C, erst nach der `MODULE`-Anweisung beginnt der eigentliche XS-Teil):

```
BOOT:
        setlocale (LC_MESSAGES, "");
```

Dies wird ausgeführt, wenn das Modul geladen wird (genauer, wenn in `gettext.pm` die Methode `bootstrap` aufgerufen wird), also normalerweise beim ersten `use/require Locale::gettext`.

Am Anfang der Datei (bei den include-statements muß natürlich noch der header `locale.h` „included“ werden).

Übrigens reagiert der XS-Parser von Perl sehr allergisch auf (manche) Leerzeilen, z.B. wird

```
BOOT:
        xxx;

#ifdef LC_MESSAGES
        setlocale(LC_MESSAGES, ...);
#endif
```

Garantiert *nicht* richtig geparsed. Um solchen Fällen zu entgehen, sollte man immer einen Block ({}) verwenden, wnen man sich nicht ganz sicher ist:

```
BOOT:
{
        xxx;

#ifdef LC_MESSAGES
        setlocale(LC_MESSAGES, ...);
#endif
}
```

2.2.2. `make test`

`make test` (oder das GNU-Äquivalent `make check`) bieten neuerdings immer mehr Pakete (in anderen Sprachen) - Perl-Module dagegen bieten so etwas fast alle an, und schon seit langer Zeit. Grund ist auch hier die gute Standardisierung: `h2xs` legt automatisch ein `test.pl` an, und der MakeMaker bindest das auch automatich ins Makefile ein:

```
# Before 'make install' is performed this script should be runnable with
# 'make test'. After 'make install' it should work as 'perl test.pl'

######################### We start with some black magic to print on failure.

# Change 1..1 below to 1..last_test_to_print .
# (It may become useful if the test is moved to ./t subdirectory.)

BEGIN { $| = 1; print "1..1\n"; }
END {print "not ok 1\n" unless $loaded;}
use Locale::gettext;
$loaded = 1;
print "ok 1\n";

######################### End of black magic.
```

So, wie es dasteht, prüft es lediglich, ob das Modul geladen werden kann. Das an sich ist schon ein guter Test, der die meisten Link-Probleme abfängt. Auch ist die Ausgabe von `make test` nicht sonderlich schön:

```
cerebro:~/src/Locale/gettext# make test
PERL_DL_NONLAZY=1 /usr/bin/perl -Iblib/arch ...
1..1
ok 1
```

Besser ist es, wenn man ein eigenes Verzeichnis für test verwendet:

```
$ mkdir t
$ mv test.pl t/01_basic.t
```

(Nach solchen chirurgischen Eingriffen sollte man unbedingt das `MANIFEST` ändern und das Makefile (mit `perl Makefile.PL`) neu generieren).

Ruft man nun `make test` auf, sieht es schon viel hübscher aus:

```
cerebro:~/src/Locale/gettext# make test
PERL_DL_NONLAZY=1 /usr/bin/perl -Iblib/arch ...
t/01_basic.........ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs ( 0.05 cusr +  0.00 csys =  0.05 CPU)
```

Hier werden alle Datein im Verzeichnis `t`, deren Name auf `.t` endet, der Reihe nach (alphabetisch) aufgerufen. Ihre Ausgabe wird überprüft und neben einer Statistik wird ein beruhigendes „All tests successful.“ ausgegeben.

Dies ist auch der Grund für die merkwürdige Ausgabe des Test-Skriptes: Zuerst wird ausgegeben, welche Tests ausgeführt werden (hier: `1..1`, also nur einer). Jeder Test sollte dann ein „`ok` Nummer“ ausgeben - oder ein `not ok`.

Einen guten Test für dieses Modul zu schreiben, ist garnicht mal so einfach, da dazu eine Übersetzungstabelle installiert werden müßte, was relativ systemabhängig ist. Ein kleiner Test sollte es aber schon sein (die geänderten Stelle wurden hervorgehoben):

```
BEGIN { $| = 1; print "1..2\n"; }
END {print "not ok 1\n" unless $loaded;}
use Locale::gettext;
$loaded = 1;
print "ok 1\n";

print gettext("fantasy test text") eq "fantasy test text" ? "ok 2" : "not ok 2";
```

Wenn man viele Tests schreibt, sollte man sich unbedingt das Modul `Test` ansehen. Es hilft bei der Planung und nimmt einem das Zählen der Tests ab.

2.3. Eine neue Funktion: __

Immer `gettext` zu schreiben wird auf Dauer langweilig. Eine schönere Syntax muß her. Zuerst dachte ich daran, eine Unterstrich zu verwenden: `print _„Hello, World!\n“`, aber das scheiter daran, daß der Unterstrich für Perl einfach „zu magisch“ ist.

Der (für mich) nächstliegendere Name waren *zwei* Unterstriche „__“. Da in Perl Klammern nahezu überflüssig sind, ist dieser längere Name immer noch kürzer als sein Äquivalent in C:

```
C:    print _("Text");
Perl: print __"Text";
```

Solche „Kleinigkeiten“ sind in Perl sehr wichtig...

Aber nun denn, schreiten wir zur Implementation. Im Gegensatz zu `gettext` müssen wir hier die Funktion selbst implementieren:

```
char *
__(msgid)
        char *       msgid
        CODE:
        RETVAL = gettext (msgid);
        OUTPUT:
        RETVAL
```

Hmm.. eine Menge neues Holz. Hinzugekommen sind zwei neue XS-Anweisungen: `CODE:` (gefolgt von C-Anweisungen> und `OUTPUT:` (um Rückgabewerte zu definieren). Es ginge auch etwas einfacher (z.B. mit einem `define`), aber obiges klappt mit jeder Funktion.

Was passiert hier? Zuersteinmal, alles, was hinter `CODE:` folgt ist im wesentlichen normales C (oder C++), man könnte also auch schreiben:

```
        CODE:
        printf ("DEBUG: __(\"%s\") ", msgid);
        RETVAL = gettext (msgid);
        printf ("=> \"%s\"\n", RETVAL);
```

(`stdio.h` nicht vergessen!) Der einzige Unterschied ist die Variable `RETVAL`: Sie wird von Perl erzeugt, um das Ergebnis aufzunehmen. Was man in `RETVAL` ablegt, wird auf der Perl-Ebene als Ergbnis zurückgeliefert.

Damit Perl auch weiß, das diese Variable geschrieben wird, muß man das in einer `OUTPUT:`-Anweisung festlegen. Meiner Meinung nach ist das überflüssig: Wenn Perl schon weiß, daß es `RETVAL` bereitstellen muß, könnte das *auch* automatisch passieren. Aber `h2xs` und der XS-Compiler haben einige archaische Angewohnheiten, oder anders gesagt: es könnte *noch* besser sein.

2.3.1. `PROTOTYPE:`

Ein kleiner kosmetischer Fehler ist in der obigen Version aber noch enthalten. Der Aufruf:

```
print __"Text1", "\n";
```

geht in die Hose, da __ beide Argumente an sich zieht, damit nicht zurecht kommt und eine Fehlermeldung ausgibt. Deshalb sollte man für __ (und die anderen funktionen) noch einen Prototyp festlegen:

```
        PROTOTYPE: $
```

Das setzt den Prototyp auf „$“ (__ nimmt also nur einen Skalar an). Es geht übrigens noch einfacher. Man kann global (ähnlich wie `BOOT:`) festlegen, daß der XS-Compiler automatisch einen (möglichst passenden) Prototyp generieren soll:

```
PROTOTYPES: ENABLE
```

Das ist meistens (aber nicht immer) das Richtige.

Die Manpage zu `perlxs` enthält noch jede Menge weiterer, lustiger Anweisungen...

2.4. Perl-Datenstrukturen in C

Mit dem bisherigen Rüstzeug kann man schon viel machen. Die Intraktion von Perl und C ist aber irgendwie auf dem C-level, da die Funktionen kein `undef`, keine Referenzen oder anderen Perl-Spezialitäten kennen.

Greifen wir nun in die unerschöpfliche Trickkiste der an den Haaren herbeigezogenen Beispiele und fischen eine `gettext`-Variante heraus, die sich verhält wie das Original, es sei denn, man füttert sie mit `undef` - in diesem Fall soll ein Fehler ausgegeben werden.

```
gettext($string)        # => Ubersetzung
gettext()               # => die "Fehler!"
```

Dazu muß man auf Perl-Datenstrukturen zugreifen (keine Angst, ist ganz einfach). Die wichtigste ist ein „`SV`“ (Scalar Value oder Skalarwert). Da man in Perl nur Skalare an Funktionen übergeben kann (ja, egal, was man schreibt, ob Hash, Referenz, Array oder Konstante, übergeben wird immer ein oder mehrere Skalare), ist der Typ auf der C-Ebene einfach ein „`SV *`“, ein Zeiger auf einen Skalarwert. Statt `char *` kann man also einfach `SV *` schreiben:

```
char *
gettext(msgid)
           SV * msgid
```

Auf die Struktur selbst darf man nicht zugreifen, dafür gibt es jede Menge Makros (die in der Manpage zu `perlguts` beschrieben sind). Die wichtigsten sind: **Funktion Beschreibung** SvOK(SV *) Ist der Skalar überhaupt definiert? SvIOK(SV *) Ist der Skalar eine gültige Integer-Zahl? SvNOK(SV *) Ist der Skalar eine gültige Fließkomma-Zahl? SvPOK(SV *) Ist der Skalar ein gültiger String? SvIV(SV *) Gib den Skalar als `IV` (integer-value) zurück SvNV(SV *) Gib den Skalar als `NV` (numeric-value) zurück SvPV(SV *, STRLEN len) Gib den Skalar als String zurück (und schreibe die Länge in `len`) SvPV_nolen(SV *) Das gleiche, aber ohne Länge (erst ab Perl 5.005) SvTRUE(SV *) Behandele den Skalar als Wahrheitswert und gib diesen C-gerecht zurück SvRV(Sv *) Liefert das Objekt, auf das eine Referenz zeigt newSViv(IV) Erzeuge Skalar aus einem IV newSVnv / NV) Erzeuge Skalar aus einem NV newSVpv(const char *, int) Erzeuge neuen Skalar aus einem String (oder anderen Daten)

Wenn man ein bißchen damit arbeitet, wird man bald feststellen daß alle Datentypen sehr vernünftig benannt sind. Ein SV / NV / IV / AV / HV / PV usw. ist immer ein bestimmter Wert (Value), und die Buchstaben S / N / I / A / H / P usw. in Funktionsnamen deuten immer darauf hin, welche Datentypen die Funktion zurückgibt (oder erwartet): Datentypen **SV Ein Perl-Skalar** AV Ein Perl-Array HV Ein Perl-Hash IV Ein Integer (es gibt auch `I16`, `I32` usw.. für Integer mit einer bestimmten Größe) UV Ein „unsigned-integer“ (ohne Vorzeichen) auch davon gibt es `U32` ... Varianten NV Eine Fließkommazahl (numeric value) PV Ein „Pointer-Wert“, also ganz allgemein eine Datenstruktur, meistens aber ein C-String (in Perl ist es bekanntlich möglich, beliebige Inhalte in Skalaren zu speichern). RV Eine Referenz

2.4.1. Beispiele

Um einen Fehler auszulösen, falls `undef` übergeben wird, kann man in der `gettext`-Funktion einfach schreiben:

```
        if (!SvOK (msgid))
          croak ("cannot translate undefined values");

        RETVAL = gettext (SvPV_nolen (msgid));
```

`croak` verhält sich wie sein Perl-Äquivalent im Modul `Carp`. Wollen wir auch Referenzen auf Strings zulassen (um ein völlig abgedrehtes Beispiel zu bringen), müßte man schreiben:

```
        if (SvROK (msgid))
          {
```

```
    if (SvTYPE (SvRV (msgid)) == SVt_PV)
      msgid = SvRV (msgid);
    else
      croak ("only references to scalar values allowed");
  }
```

Mit `SvROK` wird geprüft, ob eine Referenz vorliegt, wenn ja, wird mit `SvRV` das Objekt geholt, auf das die Referenz zeigt, und dessen Typ (den man mit `SvTYPE` bekommt) bestimmt. Ist er `SVt_PV` (ein „PV“), haben wir den String zum Übersetzen gefunden, alles andere ist ein Fehler.

Dieses - zugegeben gemeine Beispiel - zeigt, daß das wichtigste immer ein zweites Terminal mit der Manpage zu `perlguts` ist, in der alle diese Funktionen erklärt werden. Ich mußte für das obige Beispiel zwei Dinge nachschauen: ob `SvTYPE` auch wirklich `SvTYPE` heißt, und was der Typ für einen PV ist (nämlich `SVt_PV`).

Also: Mit Manpage = einfach. Ohne Manpage = Guru / Blöd.

2.4.2. Haben Strings ein Nullbyte am Ende?

Laut Dokumentation darf man sich nicht darauf verlassen (sie drückt sich extrem schwammig aus, wohl, weil es ursprünglich geplant war, Strings ohne Nullbyte zu speichern). In der Praxis funktioniert kaum ein Modul, wenn das nicht der Fall wäre. Deshalb kann man davon ausgehen, das Strings immer mit einem Nullbyte enden, und muß dafür sorgen, daß Strings, die man selbst erzeugt, ebenfalls mit einem Nullbyte enden.

2.4.3. Rückgabewerte

Da auch die Rückgabewerte SV's sind, kann man auch hier die Typenumwandlung selbst in die Hand nehmen:

```
SV *
gettext(msgid)
           SV * msgid
           CODE:
           ...
           OUTPUT:
           RETVAL
```

Jetzt kann man das Resultat mit Hilfe der Funktion `newSvPV` von einem C-String in einen SV umwandeln:

```
        const char *translated = gettext (SvPV_nolen (msgid));
        RETVAL = newSvPV (translated, 0);
        RETVAL = sv_2mortal (RETVAL);
```

Böse wie XS ist, lauert hier noch ein kleiner Stolperstein, doch zuerst die Funktion `newSvPV`: Ihr erstes Argument ist der Zeiger auf den C-String (`translated`). Das zweite Argument ist die Länge. Übergibt man 0, berechnet Perl diese freundlicherweise für uns.

Als Ergebnis winkt ein ganz normaler `SV *`. Würden wir diesen direkt zurückgeben, hätten wir aber ein Speicherproblem. Warum das so ist, liegt an der Speicherverwaltung von Perl:

Jedes Objekt hat einen Referenzzähler, also eine Zahl, die angibt, von wie vielen „Benutzern“ dieses Objekt gerade benutzt wird. Sinkt dieser auf 0, wird es freigegeben, sonst nicht. Auf der Perl-Ebene wird der Referenzzähler z.B. erhöht, wenn man eine Referenz auf einen Skalar erzeugt, und verringert, wenn diese Referenz gelöscht wird, oder das Programm den Gültigkeitsbereich der Variable verläßt:

```
    {
       my $sv = 5;    # SvREFcnt($sv)  == 1
```

```
    my $rv = \$sv;          # SvREFcnt($sv)  == 2
    undef $rv;     # SvREFcnt($sv)  == 1
}                      # SvREFcnt($sv)  == 0 => LOSCHEN
```

Das Problem ist: wenn ein neuer Skalar erzeugt wird, hat dieser den Referenzzähler 1. Von selbst verringert sich dieser nicht, und selbst darf man ihn auch nicht verringern, sonst würde der Skalar ja sofort gelöscht. Deshalb gibt es das Konzept der „Sterblichkeit“, was nichts anderes ist, als ein verzögertes herunterzählen des Referenzzählers.

Wenn ein Skalar als „sterblich“ (eine ausgesprochen gute Bezeichnung) markiert wurde (mit `sv_2mortal`), wird sein Referenzzähler etwas später erniedrigt, günstigerweise dann, wenn der Aufrufer die Möglichkeit hatte, den Wert in einer Variablen zu speichern (und damit eine Referenz zu erzeugen).

2.5. Was fehlt noch zur Profi-Version?

Nun sind die wichtigsten Schritte, um ein XS-Modul zu erzeugen, besprochen worden. Alles, was dasrüber hinaus geht, wird nur „bei Bedarf“ benötigt, und dann ist es immer gut, sich eine Manpage (`perlxs` und `perlguts`) zu schnappen und dort nach einer Lösung zu suchen.

Wenn man auf ein Problem stößt (z.B. „wie übergibt man beliebig viele Argumente“), das man nicht aus der Dokumentation lösen kann, dann ist es *sehr* hilfreich, eine Weile nach einem anderen Modul zu suchen, wo das gleiche Problem auftrat (z.B. in `Image::Magick`), um dort „abzugucken“. Das gilt nicht nur für XS-Code, sondern auch für das `Makefile.PL` oder andere Spezialitäten (wie die `typemap`). Mir persönlich hat das mehr geholfen als die gesamte Perl-Dokumentation. Klar - wer viele Module benutzt hat hier einen Vorteil.

Um noch mal auf das spezielle gettext-Beispiel zurückzukommen. Es ist so noch nicht wirklich einsatzbereit (genausowenig wie das ursprüngliche `Locale::gettext` auf CPAN), da es kaum Hilfen gibt, um aus Perl-Code die Texte zu extrahieren und in eine `.pot`-Datei zu schreiben, die ein Übersetzer bearbeiten kann. Wer sich dafür interessiert, findet im `Gimp`-Modul ein Programm names `pxgettext` (für „perl-xgettext“), das das normale `xgettext`-Programm ersetzt.

3. XS und Du und Ich

Der folgende, zweite Teil ist eher wie eine Art Kochbuch aufgebaut. Niemand benötigt alle folgenden „Tricks“, um ein Modul zu schreiben. Deshalb geht es mir nicht darum, im Detail die Lösung zu vermitteln, sondern vielmehr einige interessante Probleme und ihre Lösungen zu beschreiben.

Mein Ziel ist es, zu zeigen, *was* möglich ist. Im Idealfall wird irgendwann jemand ein XS-Modul schreiben und auf ein Problem stossen, und sich dann daran erinnern, daß er das schon mal irgendwo gehört hat.

Wichtig ist, daß man weiß, *daß* etwas möglich ist, und man „nur“ in der Dokumentation suchen braucht. Sonst werkelt man vielleicht an einer schlechten Lösung herum, ohne zu wissen, daß es besser geht.

3.1. Konstanten

Konstanten (z.B. `LCMESSAGES`) habe ich bewußt aus dem Beispiel im ersten Teil ausgeklammert. Die traditionelle Methode, C-Konstanten in den Perl-Interpreter zu befördern ist der `AUTOLOAD/constant`-Mechanismus, den `h2xs` normalerweise benutzt (der `-A`-Schalter hindert `h2xs` daran). Dabei wird ein Xs-funktion namens `constant` definiert, die anhand des Konstantennamens den Wert zurückliefert:

```
static int
constant(char *name)
{
    errno = 0;
    switch (*name) {
    case 'A':
```

```
#ifdef UUACT_COPYING
        if (strEQ(name, "ACT_COPYING")) return ACT_COPYING;
#else
        goto not_there;
#endif
    case 'B':
    case 'C':
        /* .. usw... */
    case 'Z':
    }
    errno = EINVAL;
    return 0;

not_there:
    errno = ENOENT;
    return 0;
}
```

Im Perl-Teil wird ein `AUTOLOAD` benutzt, um noch nicht definierte Konstanten zu „laden“:

```
sub AUTOLOAD {
    # This AUTOLOAD is used to 'autoload' constants from the constant()
    # XS function.  If a constant is not found then control is passed
    # to the AUTOLOAD in AutoLoader.

    my $constname;
    ($constname = $AUTOLOAD) =~ s/.*::///;
    croak "& not defined" if $constname eq 'constant';
    my $val = constant($constname, @_ ? $_[0] : 0);
    if ($! != 0) {
        if ($! =~ /Invalid/) {
            $AutoLoader::AUTOLOAD = $AUTOLOAD;
            goto &AutoLoader::AUTOLOAD;
        }
        else {
                croak "Your vendor has not defined Locale::gettext macro $constname";
        }
    }
    no strict 'refs';
    *$AUTOLOAD = sub () { $val };
    goto &$AUTOLOAD;
}
```

Aus irgendeinem Grund verwendet keines meiner Module diesen Mechanismus. Am Anfang fand ich diesen Mechanismus nur unheimlich, inzwischen bin ich über einige unangenehme Nebeneffekte gestolpert und weiß, weshalb ich ihn nicht verwende:

- `h2xs` funktioniert nur einmal (ein großes Manko). Die `constant`-Funktion upzudaten ist nicht die angenehmste Aufgabe.

- Der Prototyp der Konstante (also „`()`“) ist erst nach der Benutzung definiert, d.h. folgendes Programmfragment:

```
$res == UUERR_NONE ? „ok“ :  „not ok“;
```

Wird nur korrekt geparsed, falls UUERR_NONE schoneinmal verwendet wurde (z.B. in einem anderen Programmteil. Ist dies nicht der Fall, wird die Funktion (!) `UUERR_NONE` aufgerufen mit dem Resultat

eines Pattern-Matching (?-Operator), der nicht geschlossen wird. Besonders böse wird dieser Fall, wenn man das ganze noch kommentiert:

```
$res == UUERR_NONE ? „ok“ :  „not ok“; # did it work?
```

So etwas passiert *mir* dauernd... solche Debugging-Sessions können Stunden dauern...

- Es ist langsam, die Konstanten werden nicht ge-inlined. Dies sollte an sich kein Grund sein, den Mechanismus abzulehnen, aber es ist ja nicht das einzige Problem.
- Die Methode, das vorhandensein einer Konstanten per `#ifdef` zu prüfen, ist ja ganz nett, aber in heutigen Zeiten ist das ein echtes Kompatibilitätsproblem. Perl selbst konnte deshalb auf einigen System (Linux!) nicht übersetzt werden, bis eine bessere Methode gefunden wurde.

Natürlich muß es einen besseren Weg geben. Den gibt es auch, zumindest ab perl5.005: `newCONSTSUB`. Dies ist eine C-API-Funktion von Perl und wird normalerweise so eingesetzt:

```
BOOT:
{
   HV *stash = gv_stashpvn ("Gimp", 4, TRUE);

   newCONSTSUB (stash, "PARAM_BOUNDARY",newSViv(PARAM_BOUNDARY));
   newCONSTSUB (stash, "PARAM_CHANNEL", newSViv(PARAM_CHANNEL) );
   newCONSTSUB (stash, "PARAM_COLOR",   newSViv(PARAM_COLOR)   );
   newCONSTSUB (stash, "PARAM_DISPLAY", newSViv(PARAM_DISPLAY) );
   newCONSTSUB (stash, "PARAM_DRAWABLE",newSViv(PARAM_DRAWABLE));
   newCONSTSUB (stash, "PARAM_END",     newSViv(PARAM_END)     );
}
```

Ein „stash“ entspricht in etwa einem Perl-Package (und ist im wesentlichen ein Hash der alle Package-globalen Objekte enthält). Die Funktion `newCONSTSUB` erzeugt eine neue Konstante aus einem Skalar (den man schnell mit `newSViv` o.ä. erzeugen kann). Diese Methode hat keinen der Nachteile von `constant`, und erlaubt ein angenehmeres Konstanten-Management (Konstanten, die zusammengehören, sind auch im Quellcode nahe beieinander; das Sortieren entfällt; man kann sie einfacher Programmatisch erzeugen).

Die Nachteile sind ein höherer Speicherverbrauch (ein Perl-`sub {}` ist relativ fett) und eine längere Ladezeit.

3.1.1. `bootstrap` und `BEGIN`

Ein ähnliches Problem wie `constant` gibt es auch beim Bootstrappen des Moduls. Wenn Perl z.B. `Locale::gettext` lädt, wird nur die `.pm`-Datei geladen. Erst der Aufruf `bootstrap Locale::gettext` lädt die XS-Funktionen und definiert diese auf der Perl-Ebene.

Da dies zur Laufzeit geschieht, bedeutet daß, das alle *Benutzer* eines Moduls die Funktionen (und vor allem die Prototypen) „sehen“ können, außer dem Modul selbst.

Freunde des übermäßigen Klammernverbrauchs werden das nicht bemerken. Genaugenommen bemerkt niemand etwas, denn Perl kann mögliche Fehler durch falsche Parameterzahl etc. garnicht diagnostizieren. Abhilfe schafft ein `BEGIN`. Statt „normal“ `bootstrap` aufzurufen, wickelt man es in ein `BEGIN`:

```
BEGIN { bootstrap Locale::gettext $VERSION }
```

In PDL z.B. liefen einige Module nach dieser Änderung nicht mehr, weil sie XS-Funktionen aufriefen und dabei falsche Parameter verwendeten.

3.2. Callbacks

Relativ häufig sind sog. „Callbacks“, also Aufrufe von Perl-Funktionen aus einer C-Bibliothek heraus. Dabei treten zwei Probleme auf:

1. „Was ist ein Callback“: Eine Codereferenz oder ein Funktionsname, der irgendwie aufbewahrt werden muß.

2. „Wie wird er aufgerufen“: Wie sieht ein Perl-Funktionsaufruf in C aus.

Ersteres ist nicht schwer, es gibt jedoch eine Menge Stolperfallen. Es hat sich bewährt, eine globale Variable für den Callback zu definieren, und ihn auf `undef` zu setzen:

```
static SV *my_callback;

MODULE = ...     PACKAGE = ...

INIT: /* vor perl 5.6 mus es BOOT: sein! */
        my_callback = newSVsv (&PL_sv_undef);

void
set_my_callback(callback=0)
        SV *    callback
        CODE:
        sv_setsv (my_callback, callback);
```

Auf diese Weise erspart man sich jede Menge Spielchen mit dem Referenzzähler! `set_my_callback` kann man auf viele Weise benutzen:

```
set_my_callback sub { print "callback called!\n" }; # ein anonymous sub
set_my_callback \&my_perl_callback;                 # Funktionsreferenz
set_my_callback "my_perl_callback";                 # (nicht so gut!)
set_my_callback undef;                              # deaktivieren
set_my_callback;                                    # ditto
```

Der Aufruf sieht schon etwas komplizierter aus (Die `perlcall`-manpage enthält nicht wenige Hinweise...), gleicht aber immer diesem Beispiel:

```
static int
my_callback(int timestamp, char *infostring)
{
  dSP;
  int count;
  int retval;

  ENTER; SAVETMPS; PUSHMARK (SP);

  EXTEND (SP, 2);
  PUSHi (sv_2mortal (newSViv (timestamp)));
  PUSHs (sv_2mortal (newSVpv (infostring, 0)));

  PUTBACK;
  count = perl_call_sv ((SV *)cb, G_SCALAR);
  SPAGAIN;

  if (count != 1)
    croak ("perl-my_callback returned more than one argument");

  retval = POPi;
```

```
  PUTBACK; FREETMPS; LEAVE;

  return retval;
}
```

`ENTER; SAVETMPS; PUSHMARK (SP);` entspricht in etwa einer öffnenden `geschweiften Klammer`, `PUTBACK; FREETMPS; LEAVE;` schließt sie wieder.

Mit `EXTEND (SP, 2)` wird Raum für zwei Argumente geschaffen, die zwei folgenden `PUSHs` legen ein `int` und einen String als Argument auf den Perl-Stack.

Den eigentlichen Aufruf erledigt die Kombination `PUTBACK/perl_call_sv/SPAGAIN`. Das `POPi` nimmt den (einzigen) Rückgabewert vom Stack (den man *immer* aufräumen muß!).

`perlcall` enthält noch viele weitere Details, mit denen man sich belasten kann.

3.3. INCLUDE:

Eine nützliche Anweisung ist auch `INCLUDE:`. Klar ist, daß man mit ihr XS-Code „includen“ kann. Ein eher unbeachtetes Feature (vieler Perl-Module!) ist, daß man auch andere Dinge als nur Dateien als Quelle benutzen kann. In einem meiner Module (`Video::Capture::V4l`) erzeuge ich z.B. die Mutator-Methoden für alle verwendeten Objekte durch ein Perl-Skript namens `genacc`:

```
INCLUDE: ./genacc |
```

Das „magische“ `open` von Perl sorgt dafür, daß hier die Ausgabe des Programmes eingefügt wird. Damit kann man Dinge machen, von denen C-Programmierer normalerweise zurückschrecken, nämlich, ganz neue Sprachen entwicklen ;)

3.4. Aliase

Ein nettes Feature von XS sind die sogenannten Aliase. Wenn man eine XS-Funktion (ein sogenanntes „XSUB“) definiert, kann man beliebig viele weitere Namen angeben, unter denen dieses XSUB in Perl erscheint. Das ist besonders nützlich für Funktionen, die eine recht aufwendige Umsetzung erfordern, sich aber sonst stark ähneln. In Gimp z.B. gibt es zwei Funktionen, `gimp_install_procedure` und `gimp_install_temp_proc`, die sich nur in ihrer Semantik, nicht in den Parametern unterscheiden.

Die Definition sieht so aus:

```
void
gimp_install_procedure(name, blurb, usw....)
        char *  name
        char *  blurb
        ...
        ALIAS:
                gimp_install_temp_proc = 1
        CODE:
        ... aufwendige initialisierungen ...
        if (ix)
          gimp_install_temp_proc (name, blurb, ...);
        else
          gimp_install_procedure (name, blurb, ...);
```

Man gibt jedem „Alias“ eine Nummer. Beim Aufruf kann man dann in der Variablen `ix` nachsehen, welcher Alias aufgerufen wurde. Ist sie 0, wurde kein Alias (sondern die eigentliche Funktion) aufgerufen.

3.5. C_ARGS

Ein weiteres kleines Feature, das erst in Version 5.6 von Perl erscheinen wird, ist C_ARGS. Wenn man keinen CODE:-Abschnitt benutzt, erzeugt Perl selbst einen. Das spart mindestens 4 Zeilen ein. Vor Version 5.6 konnte man aber Argumente nicht umstellen, oder erzeugen: Manchmal gibt es Funktionen, die man in Perl „einfach anders“ aufrufen würde, oder Funktionen, die Parameter verlangen, die man in Perl einfach berechnen kann.

Ein einfaches (nicht sonderlich sinnvolles) Beispiel ist memset: Es erwartet Adresse und Länge eines Speicherblocks. Möchte man eine Schnittstelle zu memset, die einen Perl-Skalar „setzt“, kann man mit C_ARGS die Parameter für den Aufruf setzen, ohne den Aufruf selbst schreiben zu müssen:

```
void
memset(data, content)
        SV *    data
        int  content
        C_ARGS:
          SvPV_nolen (data), content, SvCUR (data)
```

3.6. Typemaps

Typemaps sind eines der arbeitssparendsten Werkzeuge, die XS zu bieten hat. Im gettext-Beispiel wurden C-Datentypen wie char * oder int benutzt, als wären sie „eingebaut“ tatsächlich sind sie genausowenig in den XS-Compiler eingebaut wie SV * oder ein anderer Perl-Datentyp.

Stattdessen konsultiert der XS-Compiler (der übrigens xsubpp heißt, also in etwa „XSUBPräprozessor“) jedesmal, wenn er über einen Typennamen stolpert, eine Tabelle, die ihm sagt, wie man

1. diesen C-Typ aus einem Perl-Skalar erzeugt (INPUT)
2. aus diesem C-Typ einen Perl-Skalar konstruiert (OUTPUT)

Hier ist eine einfache Typemap (die in etwas ausführlicherer Form mit perl mitgeliefert wird):

```
int             T_IV

INPUT

T_IV
        $var = SvIV ($arg);

OUTPUT

T_IV
        sv_setiv ($arg, $var);
```

Der erste Teil ordnet jedem C-Typ eine Art Typkennung zu. Diese kann man frei erfinden (z.B. „Stefans_Object“). Der INPUT-Abschnitt legt fest, wie die Perl-Argumente in das XSUB „hineinwandern“, der OUTPUT-Abschnitt wird benutzt, um C-Datentypen wieder an perl „auszugeben“.

Die Definition ist in „ganz normalem“ C geschrieben. $var und $arg sind Platzhalter und stehen für die C-*VAR*iable und das Perl-*ARG*ument. Außerdem gibt es noch den Platzhalter $type, der den Namen des C-Datentyps (z.B. int) bereithält, sowie $ntype, der den Typnamen in Perl-Form enthält (bei dem Unterstriche durch :: ersetzt wurden). Doppelte Anführungszeichen (") und andere Perl-Sonderzeichen ($, @, \) müssen außerdem gequoted werden.

Wenn man sich einmal die Typemap (bzw. einige *der} Typemaps*) ansieht, die zu Perl schon mitgeliefert wird (sie ist etwas gekürzt im Anhang abgedruckt), findet man einige interessante Datentypen (die im übrigen *alle* und *vollkommen vollständig* undokumentiert sind). Zwei sehr hilfreiche sind T_PTROBJ und T_PTRREF. Beide können sehr komplexe Typen aufnehmen (Zeiger auf structs, classen u.ä.).

`T_PTRREF` erzeugt dafür einfach eine Referenz auf einen Skalar (der den Zeiger enthält), während `T_PTROBJ` ein echtes Perl-Objekt erzeugt. Als Beispiel soll folgende Typemap dienen:

```
Locale_gettext_state    T_PTROBJ
```

So kurz kann es sein. Schreibt man jetzt im C-Teil noch folgendes typedef:

```
typedef struct my_state Locale_gettext_state;
```

so kann man den neuen Datentyp `Locale_gettext_state` verwenden, der in Perl als Objekt vom Typ(!) `Locale::gettext::state` existiert (das wird mit `$ntype` erreicht). Steckt man jetzt noch ein paar Methoden in dieses Modul:

```
MODULE = Locale::gettext                PACKAGE = Locale::gettext::state

void
set_state_arg(state, arg)
        Locale_gettext_state state
        int  arg
        CODE:
        state->arg = arg;

void
DESTROY(state)
        Locale_gettext_state state
        CODE:
        free (state);
```

Kann man Objekte komplett und vor allem angenehm in XS implementieren, wobei der Typemap-Eintrag für das `state`-Objekt sogar eine Typprüfung für uns übernimmt. „No need to use perl anymore ;)“.

3.6.1. Typemaps mit Haaren

Die Umsetzung von `Locale_gettext_state` zu `Locale::gettext::state` ist hilfreich, aber wie macht man das in einem Typemap-Eintrag? Einen kleinen Hinweis bekommt man, wenn man sich die Ähnlichkeit eines Typemap-Eintrages mit einem Perl-String vor Augen hält:

```
        sv_setiv ($arg, $var);
```

Das ganze ist nichts anderes, als ein String, in dem `$arg` und `$var` interpoliert wird! Den String kann man jederzeit beenden:

```
        sv_setiv ($arg, ". sin(2) ." + $var);
```

Der Ausdruck `sin(2)` wird in Perl (und vor allem während der Übersetzungszeit) ausgeführt. Auf diese Weise kann man sehr komplexe Typen (z.B. `T_PTROBJ`) erzeugen, die in ihrer Mächtigkeit sehr ähnlich zu generischen Typen (wie C++-`templates` oder Haskells Typsystem) sind.

Die `INPUT`-Definition für `T_SimpleVal` aus dem Gtk-Modul sieht z.B. so aus:

```
T_SimplePtr
        $var = Sv" . ($foo=$ntype, $foo=~s/://g, $foo=~s/^GtkGdk/Gdk/, $foo) . "($arg,0)
```

Er dient dazu, aus Gtk-Typnamen die entsprechenden Zugriffmakros zu erzeugen:

Datentyp	**Makro**
Gtk::Gdk::Event	SvGdkEvent
Gtk::Window	SvGtkWindow

Hier wird noch ein weiteres Feature von XS verwendet: Der XS-Compiler ersetzt Doppelpunkte in Typennamen durch Unterstriche, wenn er die Typnamen aus der Typemap im C-Programm (jede XS-Datei wird in eine normale C-Datei umgewandelt). Gtk muß deshalb noch ein paar `typedefs` benutzen:

```
typedef GdkEvent *   Gtk__Gdk__Event;
typedef GTwWindow *  Gtk__Window;
```

Und schon kann man XS-Funktionen „fast wie in Perl“ schreiben:

```
void
set_title(self, title)
        Gtk::Window     self
        char *          title
        CODE:
        gtk_window_set_title(self, title);
```

3.7. `Makefile`-Tips

Sie betreten jetzt die Typemap Chill Out Zone..... Die folgenden Tips sind eher praktischer Natur und drehen sich um das `Makefile.PL`.

3.7.1. Ein README generieren

Diesen Tip erhielt ich von Andreas König, nachdem ich mein allererstes Modul auf CPAN veröffentlicht habe, wie üblich ohne README. Andreas meinte, ich könnte aus meiner Moduldokumentation automatisch ein README generieren lassen, indem ich den `dist`-Parameter von `WriteMakefile` verwende:

```
WriteMakefile(
    'dist'   => {
                  PREOP       => 'pod2text Gimp.pm >README',
                 },
    ...
```

`PREOP` gibt einen Shell-Befehl an, der vor dem Verpacken als Archiv aufgerufen wird (z.B. von `make dist`). Dieses README ist nicht so toll wie ein von Hand verfasstes sein könnte, aber es ist wesentlich informativer als viele READMEs auf CPAN, die einem nicht einmal sagen, *was* das Modul macht. Und vor allem ist es besser als gar keines.

Wenn wir schonmal dabei sind: man könnte auch gleich die Zugriffrechte der Dateien auf vernünftige Werte setzen, und gzip-Komprimierung schadet ebenfalls nicht:

```
    'dist'   => {
                  PREOP       => 'pod2text Gimp.pm >README && chmod -R u=rwX,go=rX .',
                  COMPRESS    => 'gzip -9v',
                  SUFFIX      => '.gz',
                 },
```

3.7.2. `.pm`-Dateien und ihr Zielverzeichnis

`ExtUtils::MakeMaker` ist geradezu extrem unflexibel, wenn man mehrere Module in einem Paket vereinigt und noch dazu unverschämte Ansprüche stellt, wie z.B. bestimmte Module nur gegen bestimmte Bibliotheken linken möchte.

Prinzipiell muß man dann jedes Modul (also `.pm`+`.xs`-Dateien) in ein eigenes Unterverzeichnis packen, wobei man dann manuell angeben muß, welchen „Perl-Namen“ ein Perl-Modul erhält, indem man das `PM`-Argument von `WriteMakefile` verwendet:

```
    'PM'     => {
                  'Gimp.pm'                 => '$(INST_LIBDIR)/Gimp.pm',
                  'Net/Net.pm'      => '$(INST_LIBDIR)/Gimp/Net.pm',
                  'UI/UI.pm'                => '$(INST_LIBDIR)/Gimp/UI.pm',
                 },
```

3.7.3. Abhängigkeiten zu anderen Modulen

Schon seit langem hat man die Möglichkeit, Abhängigkeiten zu anderen Modulen im `Makefile.PL` anzugeben:

```
'PREREQ_PM'      => {
                  Gtk                 => 0.5,
                  PDL                 => 1.99,
                  Data::Dumper        => 2.0,
                  Parse::RecDescent   => 1.6,
                 },
```

Früher gab das `Makefile.PL` beim Erzeugen des `Makefiles` nur ein paar Warnungen aus. Neuere Versionen von `CPAN` (warum heutzutage, wo es doch das CPAN gibt, immer noch Steinzeitmethoden wie *manuelles ftp* zur Installation von Modulen verwendet werden, ist mir schleierhaft ;), neuere Versionen von CPAN also erkennen diese Abhängigkeiten und installieren automatisch alle benötigten Module, so daß man von `PREREQ_PM` regen Gebrauch machen sollte.

Ein Problem kann das aber nicht lösen: viele Abhängigkeiten sind optional: Es wäre schön, wenn das entsprechende Modul vorhanden wäre, wenn es sich aber nicht übersetzen läßt ist es nicht so schlimm. Dies läßt sich leider (noch) nicht angeben.

3.7.4. Eigene Erweiterungen für das `Makefile`

Sehr häufig möchte/muß man eigene Regeln in das erzeugte `Makefile` aufnehmen. Dazu definiert man eine Funktion namens `MY::postamble` (irgendwie logisch... oder auch nicht). Ihr Rückgabewert wird ohne Änderung in das `Makefile` aufgenommen:

```
sub MY::postamble {
   <<PA
acc.c: genacc
        genacc >acc.c
PA
}
```

3.8. `Devel:PPPort` - Perl/Pollution/Portability

`SvPV_nolen` gibt es nicht in Perl 5.005, `newCONSTSUB` gibt es nicht in Perl 5.004, ... und wenn man ein neues Modul schreibt, weiß man das natürlich nicht, und die Benutzer älterer Perl-Versionen beschweren sich, weil es bei ihnen nicht läuft.

Als ich mit Kenneth Albanowski über dieses Problem sprach, meinte er, dies ginge ihm schon seit längerem durch den Kopf, und er schreibt (für das Gtk-Modul) einen Kompatibilitätsheader, der aber „zu gefährlich“ für CPAN wäre. Nun ja, das war schnell ausgeräumt und nun gibt es `Devel::PPPort`, ein Modul mit einem untippbaren Namen, das sich noch nicht einmal installieren läßt.

Es besteht im wesentlichen aus einer C-Header-Datei, `ppport.h`, die sich auch als Perl-Skript ausführen läßt.

Die Benutzung geschieht normalerweise in den folgenden Schritten:

1. neueste Perl-Version installieren.
2. neueste `Devel::PPPort`-Version holen.
3. `ppport.h` in sein Modulverzeichnis kopieren.
4. `ppport.h` *durchlesen* und ausführen:

   ```
   perl -x ppport.h *.c *.h *.xs foo/*.c etc..
   ```

   `ppport.h` gibt hilfreiche Tips aus.

1. Tips umsetzen und `ppport.h` in seinen Datein includen.

Viele Leute glauben, man sollte einfach `Devel::PPPort` installieren und bekäme automatisch Portabilität zu älteren Modulen. Das ist der Grund, weshalb Kenneth es für „gefährlich“ hielt.

Vielmehr erlaubt es einem Modul-Programmierer, sein Modul an die jeweils aktuelle Perl-Version anzupassen, und dann eine größtmögliche Kompatibilität zu älteren Perl-Installationen zu bekommen.

Das klappt natürlich nicht mit allen Features, das Modul fängt aber die meisten Probleme ab, so daß man relativ ungehindert Programmieren kann.

3.9. „magic“

Die meisten Variablen in Perl verhalten sich ihrem Typ (Skalar, Array...) entsprechend. Einige jedoch (z.B. `$!`, `%ENV` oder jedes `getiete` Objekt) verhalten sich anders. Bei einem `getieten` Skalar beispielsweise wird bei jedem Lese- oder Schreibzugriff eine benutzerdefinierte Funktion aufgerufen.

Diese Objekte heißen „magisch“, weil man ihnen ein Stück „magic“ angeklebt hat. Magic ist, ganz handfest, eine Struktur, die bestimmte Zugriffe regelt und an ein Objekt angehängt werden kann.

Jede Art von „magic“ wird durch ein Zeichen repräsentiert: ’E’ z.B. ist die `%ENV`-Magic, ’S’ dagegen die `%SIG`-Magic. Für XS-Module sind zwei Typen interessant: ’U’ (Variablenzugriffe abfangen) und ’~’ (beliebig).

3.9.1. ’U’-Magic

Für den Typ ’U’ muß man eine `struct ufuncs` bereitstellen, die drei Felder enthält: `uf_val` (wird bei lesenden Zugriffen aufgerufen), `if_set` (wird bei schreibenden Zugriffen aufgerufen) und `uf_index` (kann eine beliebige Zahl enthalten, die den Funktionen übergeben wird. Beispiel:

```
/* C-Teil */
static I32
my_get_function (IV index, SV *scalar)
{
  printf ("variable is being read\n");
  sv_setiv (scalar, rand());
}

static I32
my_set_function (IV index, SV *scalar)
{
  printf ("variable was set to new value\n");
  srand (SvIV (scalar));
}

/* XS-Teil */
void
attach_randomize_magic(sv)
        SV *    sv
        CODE:
        struct ufuncs uf;

        uf.uf_val = &my_get_function;
        uf.uf_set = &my_set_function;
        uf.uf_indef = 0;
        sv_magic (sv, 0, 'U', (char*)&uf, sizeof (uf));
```

Die XS-Funktion erwartet einen Skalar, und macht ihn „magisch“. Danach wird für jede Zuweisung die C-Funktion `my_set_function` aufgerufen, für jeden Lesezugriff stattdessen die Funktion `my_get_function`. Danach verhält sich der Skalar wie ein Zufallszahlengenerator: Jeder Lesezugriff gibt einen neue Zufallszahl zurück, und jeder Schreibzugriff setzt den Startwert des Generators.

3.9.2. '~'-Magic

'~'-Magic ist etwas komplizierter. An sich macht dieser Typ überhaupt nichts, er kann dazu verwendet werden, Datenstrukturen an Skalar anzuhängen, die von Perl aus nicht angesprochen (und vor allem nicht zerstört) werden können.

Viel schöner ist aber die Möglichkeit, alle Arten von Zugriffen abfangen zu können, indem man die „Magic Virtual Table“ (das ist so etwas wie die virtuelle Methodentabelle eines C++-Objektes). Dazu definiert man am besten eine Variable vom Typ `MGVTBL`, die (ähnlich wie `struct ufuncs`) fünf Zeiger auf Funktionen enthält: **Prototyp Beschreibung** int (*svt_get)(SV* sv, MAGIC* mg); Wird *nach* dem Lesen einer Variablen aufgerufen int (*svt_set)(SV* sv, MAGIC* mg); Wird nach dem Setzen einer Variablen aufgerufen U32 (*svt_len)(SV* sv, MAGIC* mg); Soll die Länge der Variablen zurückgeben int (*svt_clear)(SV* sv, MAGIC* mg); Wird bei `undef $var` u.ä. aufgerufen int (*svt_free)(SV* sv, MAGIC* mg); Wird aufgerufen, wnen die Variable zerstört wird

In Gimp beispielsweise verwende ich '~'-Magic, um eine C-Struktur wieder freizugeben, die nur so lange in Benutzung zu bleiben braucht, wie die entsprechende Variable in Perl genutzt wird. Das Erzeugen der '~' sieht folgendermaßen aus:

```
static MGVTBL vtbl_gdrawable = {0, 0, 0, 0, gdrawable_free};

static SV *new_gdrawable (GDrawable *gdr)
{
  SV *sv = newSViv ((IV) gdr);

  sv_magic (sv, 0, '~', 0, 0);
  mg_find (sv, '~')->mg_virtual = &vtbl_gdrawable;

  return sv_bless (newRV_noinc (sv), gdrawable_stash);
}
```

Bis auf `svt_free` sind alle Zeiger auf `0` gesetzt, d.h. die zugehörigen Funktionen werden nicht aufgerufen. Wenn das erzeugte `GDrawable`-Objekt zerstört wird (z.B. wenn der Programmablauf den Gültigkeitsbereich der Variable verläßt), wird die C-Funktion `gdrawable_free` aufgerufen:

```
{
   my $gdr = new_gdrawable (...)
   ...
} # <- hier wird C<gdrawable_free> aufgerufen
```

`gdrawable_free` gibt das `GDrawable` wieder frei (in Wirklichkeit wird es noch in einem Hash gecached und anderes mehr):

```
/* magic stuff.  literally.  */
static int gdrawable_free (SV *obj, MAGIC *mg)
{
  GDrawable *gdr = (GDrawable *)SvIV(obj);

  gimp_drawable_detach (gdr);

  return 0;
}
```

3.10. Austauschen von Informationen zwischen Modulen

Manchmal möchte man neben einer Schnittstelle auf der Perl-Ebene auch eine Schnittstelle auf der C-Ebene bereitstellen. PDL z.B. „exportiert“ einige interne Funktionen für affine Transformationen, die auch rege benutzt werden (z.B. von Gimp oder PDL::Audio). Das Time::HiRes-Modul exportiert zwei Funktionen, mit denen die aktuelle Zeit abgefragt werden kann. Auch Event stellt seine gesamte API auch auf der C-Ebene zur Verfügung.

PDL z.B. macht dies, indem die Adresse einer struct mit Funktionszeigern in der Variablen $PDL::SHARE hinterlegt wird. Wird sie verändert kann man sein Programm abschreiben.

Um eine sichere Methode zu schaffen, um globale Werte hinterlegen zu können, wurde in Perl 5.005 eine globale Variable, HV *PL_modglobal eingeführt, in denen Module beliebeige Werte setzen können, die aber auf der Perl-Ebene nicht sichtbar ist.

PDL könnte z.B. folgende Methode benutzen, um den Zeiger auf die Funktionstabelle zu speichern:

```
/* Der Hash-Key (sollte den Modulnamen enthalten!) */
#define PDL_SHARE "PDL::SHARE"

/* Abspeichern */
hv_store (PL_modglobal, PDL_SHARE, strlen (PDL_SHARE), newSViv ((IV)func_ptr), 0);

/* Wiederfinden */
struct pdl_functions *share;
SV **share_ptr = hv_fetch (PL_modglobal, PDL_SHARE, strlen (PDL_SHARE), 0);

if (!share_ptr)
  croak ("PDL shared data block not found, please load the PDL module first!");

share = (struct pdl_functions *)SvIV (share);
```

3.11. INIT und CHECK

In Perl 5.6 wird der Compiler (B), der als experimentelles Feature schon in Version 5.005_03 dabei war, erstmals vollständig unterstützt. Für Modul-Programmierer (sowohl Perl und XS) wirft dies neue Probleme auf: Module, die in einem Programm geladen werden, führt der Compiler nur einmal aus: während der Compilierung.

Werden in der BOOT:-Sektion (oder auch im der .pm-Datei) Initialisierungen durchgeführt, die der Compiler nicht sehen kann (z.B. globale Variablen in C), gehen diese natürlich verloren.

Deshalb wurden zwei neue Funktionsnamen reserviert: Neben BEGIN und END werden auch die beiden Funktionen CHECK und INIT automatisch von Perl aufgerufen: Alle CHECK-Funktionen werden aufgerufen, wenn Perl die Compilierung beendet hat, aber bevor der Objektcode gespeichert wird. Die CHECK-Routine hat also die Möglichkeit, den internen Zustand in einer Perl-Variablen zu speichern.

Wird das (compilierte) Programm aufgerufen, werden all INIT-Funktionen aufgerufen. Dort können dann entweder die nötigen Initialisierungen wiederholt werden, oder es können Perl-Variablen (die in CHECK erzeugt wurden) dazu verwendet werden, den internen Zustand wiederherzustellen.

Ein Beispiel:

```
BEGIN        { print "begin\n" }
CHECK   { print "check\n" }
INIT    { print "init\n"  }
END  { print "end\n"   }

print "runtime\n";
```

Wird dieses Programm übersetzt, erscheint die folgende Ausgabe:

```
begin
check
```

Wenn das übersetzte Programm dann ausgeführt wird, erscheint:

```
init
runtime
end
```

Wird das Programm dagegen „nur“ interpretiert, werden alle Funktionen aufgerufen:

```
begin
check
init
runtime
end
```

Wenn man also Perl 5.6 voraussetzen kann (das ist der eigentlich Knackpunkt), so kann man die gesamte Initialisierung in ein `INIT` (oder in einen `INIT:`-Abschnitt im XS-Teil) packen.

A. Listing - `gettext.pm`

```
package Locale::gettext;

use strict;
use vars qw($VERSION @ISA @EXPORT @EXPORT_OK);

require Exporter;
require DynaLoader;

@ISA = qw(Exporter DynaLoader);

# Items to export into callers namespace by default. Note: do not export
# names by default without a very good reason. Use EXPORT_OK instead.
# Do not simply export all your public functions/methods/constants.
@EXPORT = qw(

);
$VERSION = '0.01';

bootstrap Locale::gettext $VERSION;

# Preloaded methods go here.

1;
__END__
# Below is the stub of documentation for your module. You better edit it!

=head1 NAME

Locale::gettext - Perl extension for blah blah blah

=head1 SYNOPSIS

  use Locale::gettext;
  blah blah blah

=head1 DESCRIPTION

Stub documentation for Locale::gettext was created by h2xs. It looks like the
```

```
author of the extension was negligent enough to leave the stub
unedited.

Blah blah blah.

=head1 Exported functions

  extern char *gettext  (const char *msgid)    ;
  extern char *dgettext  (const char *domainname,
                          const char *msgid)    ;
  extern char *textdomain  (const char *domainname)    ;
  extern char *bindtextdomain  (const char *domainname,
                                const char *dirname)    ;


=head1 AUTHOR

A. U. Thor, a.u.thor@a.galaxy.far.far.away

=head1 SEE ALSO

perl(1).

=cut
```

B. Die leicht gekürzte ExtUtils-Typemap

```
# basic C types
int                     T_IV
unsigned                T_UV
caddr_t                 T_PV
unsigned long *         T_OPAQUEPTR
char **                 T_PACKED
void *                  T_PTR
Time_t *                T_PV
SV *                    T_SV
SVREF                   T_SVREF
AV *                    T_AVREF

I32                     T_IV
I16                     T_IV
I8                      T_IV

############################################################################
INPUT
T_SV
        $var = $arg
T_SVREF
        if (sv_isa($arg, \"${ntype}\"))
            $var = (SV*)SvRV($arg);
        else
            croak(\"$var is not of type ${ntype}\")
T_UV
        $var = ($type)SvUV($arg)
T_IV
        $var = ($type)SvIV($arg)
T_INT
```

```
        $var = (int)SvIV($arg)
T_ENUM
        $var = ($type)SvIV($arg)
T_BOOL
        $var = (int)SvIV($arg)
T_NV
        $var = ($type)SvNV($arg)
T_DOUBLE
        $var = (double)SvNV($arg)
T_PV
        $var = ($type)SvPV($arg,PL_na)
T_PTR
        $var = INT2PTR($type,SvIV($arg))
T_PTRREF
        if (SvROK($arg)) {
            IV tmp = SvIV((SV*)SvRV($arg));
            $var = INT2PTR($type,tmp);
        }
        else
            croak(\"$var is not a reference\")
T_REF_IV_REF
        if (sv_isa($arg, \"${type}\")) {
            IV tmp = SvIV((SV*)SvRV($arg));
            $var = *($type *) tmp;
        }
        else
            croak(\"$var is not of type ${ntype}\")
T_REF_IV_PTR
        if (sv_isa($arg, \"${type}\")) {
            IV tmp = SvIV((SV*)SvRV($arg));
            $var = ($type) tmp;
        }
        else
            croak(\"$var is not of type ${ntype}\")
T_PTROBJ
        if (sv_derived_from($arg, \"${ntype}\")) {
            IV tmp = SvIV((SV*)SvRV($arg));
            $var = INT2PTR($type,tmp);
        }
        else
            croak(\"$var is not of type ${ntype}\")
T_PTRDESC
        if (sv_isa($arg, \"${ntype}\")) {
            IV tmp = SvIV((SV*)SvRV($arg));
            ${type}_desc = (\U${type}_DESC\E*) tmp;
            $var = ${type}_desc->ptr;
        }
        else
            croak(\"$var is not of type ${ntype}\")
T_REFREF
        if (SvROK($arg)) {
            IV tmp = SvIV((SV*)SvRV($arg));
            $var = *INT2PTR($type,tmp);
        }
        else
            croak(\"$var is not a reference\")
T_REFOBJ
```

```
        if (sv_isa($arg, \"${ntype}\")) {
            IV tmp = SvIV((SV*)SvRV($arg));
            $var = *INT2PTR($type,tmp);
        }
        else
            croak(\"$var is not of type ${ntype}\")
T_OPAQUE
        $var NOT IMPLEMENTED
T_OPAQUEPTR
        $var = ($type)SvPV($arg,PL_na)
T_PACKED
        $var = XS_unpack_$ntype($arg)
T_PACKEDARRAY
        $var = XS_unpack_$ntype($arg)
T_CALLBACK
        $var = make_perl_cb_$type($arg)
T_ARRAY
        $var = $ntype(items -= $argoff);
        U32 ix_$var = $argoff;
        while (items--) {
            DO_ARRAY_ELEM;
        }
T_IN
        $var = IoIFP(sv_2io($arg))
T_INOUT
        $var = IoIFP(sv_2io($arg))
T_OUT
        $var = IoOFP(sv_2io($arg))
################################################################################
OUTPUT
T_SV
        $arg = $var;
T_SVREF
        $arg = newRV((SV*)$var);
T_IV
        sv_setiv($arg, (IV)$var);
T_UV
        sv_setuv($arg, (UV)$var);
T_INT
        sv_setiv($arg, (IV)$var);
T_SYSRET
        if ($var != -1) {
            if ($var == 0)
                sv_setpvn($arg, "0 but true", 10);
            else
                sv_setiv($arg, (IV)$var);
        }
T_ENUM
        sv_setiv($arg, (IV)$var);
T_BOOL
        $arg = boolSV($var);
T_DOUBLE
        sv_setnv($arg, (double)$var);
T_PTR
        sv_setiv($arg, (IV)$var);
T_PTRREF
        sv_setref_pv($arg, Nullch, (void*)$var);
```

```
T_REF_IV_REF
        sv_setref_pv($arg, \"${ntype}\", (void*)new $ntype($var));
T_REF_IV_PTR
        sv_setref_pv($arg, \"${ntype}\", (void*)$var);
T_PTROBJ
        sv_setref_pv($arg, \"${ntype}\", (void*)$var);
T_PTRDESC
        sv_setref_pv($arg, \"${ntype}\", (void*)new\U${type}_DESC\E($var));
T_REFREF
        sv_setrefref($arg, \"${ntype}\", XS_service_$ntype,
                    ($var ? (void*)new $ntype($var) : 0));
T_REFOBJ
        NOT IMPLEMENTED
T_OPAQUE
        sv_setpvn($arg, (char *)&$var, sizeof($var));
T_OPAQUEPTR
        sv_setpvn($arg, (char *)$var, sizeof(*$var));
T_PACKED
        XS_pack_$ntype($arg, $var);
T_PACKEDARRAY
        XS_pack_$ntype($arg, $var, count_$ntype);
T_DATAUNIT
        sv_setpvn($arg, $var.chp(), $var.size());
T_CALLBACK
        sv_setpvn($arg, $var.context.value().chp(),
                $var.context.value().size());
T_ARRAY
        ST_EXTEND($var.size);
        for (U32 ix_$var = 0; ix_$var < $var.size; ix_$var++) {
                ST(ix_$var) = sv_newmortal();
        DO_ARRAY_ELEM
        }
        SP += $var.size - 1;
T_IN
        {
            GV *gv = newGVgen("$Package");
            if ( do_open(gv, "<&", 2, FALSE, 0, 0, $var) )
                sv_setsv($arg, sv_bless(newRV((SV*)gv), gv_stashpv("$Package",1)));
            else
                $arg = &PL_sv_undef;
        }
T_INOUT
        {
            GV *gv = newGVgen("$Package");
            if ( do_open(gv, "+<&", 3, FALSE, 0, 0, $var) )
                sv_setsv($arg, sv_bless(newRV((SV*)gv), gv_stashpv("$Package",1)));
            else
                $arg = &PL_sv_undef;
        }
T_OUT
        {
            GV *gv = newGVgen("$Package");
            if ( do_open(gv, "+>&", 3, FALSE, 0, 0, $var) )
                sv_setsv($arg, sv_bless(newRV((SV*)gv), gv_stashpv("$Package",1)));
            else
                $arg = &PL_sv_undef;
        }
```

C. Referenzen

Die Dokumentation zu *allen* Modulen auf CPAN kann man sich im Web unter der Addresse http://theoryx5.uwinnipeg.ca/CPAN/ ansehen.

- `perlxs` - XS language reference manual.
- `perlguts` - Introduction to the Perl API.
- `perlapi` - autogenerated documentation for the perl public API.
- `Locale::gettext` - message handling functions.
- `C::Scan` - scan C language files for easily recognized constructs.
- `Test` - provides a simple framework for writing test scripts.
- `PDL` - The Perl Data Language.
- `PDL::Audio` - Some PDL functions intended for audio processing.
- `Video::Capture::V4l`- Perl interface to the Video4linux framegrabber interface.
- `Time::HiRes` - High resolution ualarm, usleep, and gettimeofday.
- `Gimp` - Perl extension for writing Gimp Extensions/Plug-ins/Load & Save-Handlers.
- `Event` - Event loop processing.
- `B` - The Perl Compiler.
- `Devel::PPPort` - Perl/Pollution/Portability.
- `Gtk` - The GNU ToolKit.

Einführung in Event::

Marc Lehmann
pcg@opengroup.org

24. Januar 2000

Inhaltsverzeichnis

Wenn viele Jobs parallel ausgeführt werden sollen, eignet sich das bekannte fork-Paradigma von Unix nicht mehr: Die Interprozeßkommunikation und der Mehraufwand an Speicher und Ressourcen überwiegt dir Vorteile der einfacheren Programmstruktur bei weitem. Diese kurze Einführung in die Ereignisgesteuerte Programmierung in Perl zeigt an einem konkreten Beispiel (News-Scanner), wie einfach sich selbst komplexe Strukturen in Perl realisieren lassen.

1. `Event` in der Praxis — oder wie man 500 Newsserver gleichzeitig scannt.

1.1. Ereignis-gesteuerte Programmierung?

Zur Lösung paralleler ablaufender Prozesse sind heute drei Ansätze gebräuchlich:

- Prozesse mit getrenntem Adressraum (z.B. mit `fork`)
- eng gekoppelte Prozesse mit gemeinsamen Adreßraum (z.B. mit *pthreads*)
- Ereignis-gesteuerte Prozesse

Jeder dieser Ansätze hat verschiedene Vor- und Nachteile: Das `fork`-Modell ist sehr einfach zu programmieren und eignet sich besonders für einfache Probleme, die sozusagen in kleine „Stückzahlen“ anfallen. Durch die Abschottung der Prozesse wird eine einfach Parallelisierung möglich, da die Prozesse (z.B.) auf unterschiedlichen Rechnern arbeiten können. Größter Nachteil ist die relative aufwendige Interprozeßkommunikation, die einen großen Overhead nach sich ziehen kann.

Threads werden vielfach als das Mittel der Wahl angesehen. Der größte Vorteil von Threads ist das Vorhandensein mehrere Ablauf-Instanzen, die getrennt blockieren können. Leider werden Threads in den meisten Fällen nur dazu mißbraucht, das Blockieren des gesamten Prozesses zu verhindern (z.B. wenn Daten nicht sofort zur Verfügung stehen), werden also effektiv nur als Krücke für asynchrone-EA verwendet. Diesen Vorteil erkauft man sich durch eine zwar schnelle aber dafür extrem komplizierte Synchronisation innerhalb der Threads. *Threads sind in in den seltensten Fällen die richtige Wahl für ein Problem.*

Ereignis-gesteuerte Programmierung beruht auf dem *Callback*-Prinzip: Eine zentrale Anlaufstelle innerhalb des Prozesses wartet auf Ereignisse (engl. „Events“, also z.B. „Daten angekommen“, „Zeit abgelaufen“ etc...). Je nach Ereignis werden entsprechende Callback-Funktionen aufgerufen. Der Vorteil dieses Ansatzes ist eine übersichtliche Programmstruktur, eine extreme schnelle Kommunikation (nur ein Prozeß) und ein ressourcenschonendes Endprodukt. Auch dieser Ansatz hat seine Nachteile. Der größte ist, daß man bei vielen Problemen „Umdenken“ muß, da sich Callbacks eben keine lineare Programmstruktur verwirklichen läßt (*Closures* können dabei jedoch helfen). Außerdem muß man sich bewußt sein, das ein blockierender Funktionsaufruf (z.B. `read`) das gesamte Programm anhält.

1.2. Das Problem...

...ist oberflächlich betrachtet, recht einfach: Eine (kleine) Menge von Usenet-Servern soll nach Newsgruppen abgesucht werden. Das kann auf faire Weise geschehen: man öffnet eine NNTP-Verbindung und schickt Requests. Dies läßt sich durch Pipelining (senden mehrere Befehle gleichzeitig) beschleunigen. Durch die Zeiten, die der News-Server benötigt um Artikel zu suchen, wird die Datenrate in der Praxis allerdings drastisch beschränkt.

Also die unfaire Weise: statt einer öffnet man 5, 10 oder gleich mehrere hundert Verbindungen zu einem (oder mehreren) Servern und verteilt so die Verbindungslatenz und die Antwortzeit.

1.3. Die Planung

Die (für mich) naheligende Idee, dies mit mehreren Scanprozessen zu implementieren, scheiterte an zwei Problemen:

- Die Scanprozesse müssen sich untereinander absprechen, um Duplikate zu vermeiden. Dies ist zwangsläufig Interprozeßkommunikation (z.B. über eine SQL-Datenbank), die sehr aufwendig zu implementieren ist. Hinzu kommt, das einzelne Jobs zuerst markiert werden müssen („in Arbeit“), damit sie nicht von mehreren Prozessen gleichzeitig bearbeitet werden, was jedoch sehr schwierig ist, wenn man Wert darauf legt, Prozesse beliebig abbrechen zu können, ohne Artikel zu verlieren.

- Das Zielsystem, ein Pentium-166-System, hat weder unendlich Rechen- noch Speicherressourcen. Da Perl von beidem gerne viel nimmt, wäre die Sättigung schon bei relativ wenigen Verbindungen erreicht. Stichwort Speicher: jeder Prozeß benötigt einen Interpreter, eine Kopie der libc-Variablen, eine eigene Kopie des Scanprogramms und seine eigene SQL-Anbindung.

Die Lösung (klar!) lag im Event-Modul. Da alle Verbindungen von einem Prozeß bearbeitet werden, gibt es keine Synchronisationsprobleme. Der Overhead pro Verbindung beschränkt sich ebenfalls auf einen Hash, und das umschalten von Prozessen entfällt ebenfalls (schneller).

2. Die Implementation

Die folgenden Abschnitte stellen die wichtigsten „Knotenpunkte“ des Scanprogrammes vor. Jedesmal wird kurz das Problem erläutert und die Lösung mit Hilfe des `Event`-Moduls diskutiert.

2.1. Der „Scheduler“

Der komplizierteste Teil des Programmes ist der Scheduler: Er verteilt einzelne Jobs auf die Scanner, bzw. beendet das Programm, wenn alle Jobs abgearbeitet wurden. Es gibt nur zwei Typen von „Jobs“:

- 'S': *S*canne eine Gruppe. Der Scanner sucht eine bestimmte Newsgruppe auf dem Server und stellt mit Hilfe einer SQL-Tabelle fest, welche Artikel(-nummern) noch nicht gescannt wurden.
- 'A': *A*rtikel holen. Da Gruppen Tausende von Artikeln enthalten können, wird nur ein „Job“ pro Gruppe erzeugt. Ein Scanner sucht sich eine Artikelnummer aus, bearbeitet sie und legt die restlichen wieder zurück in die Warteschlange.

Ein „Scanner“ ist dabei kein Prozeß, sondern nur eine Instanz der `Scanner`-Klasse, in der im wesentlichen der Zustand einer Verbindung gespeichert wird (Server, Port, aktuelle Gruppe…). Für jede potentielle Verbindung wird ein solches Objekt erzeugt. Für hundert Verbindungen sieht das z.B. so aus:

```
new Scanner for 1..100;
```

Die Objekte reihen sich automatisch in die `idle`-Warteschlange ein.

Beim Programmstart werden alle Server- und Gruppen aus einer Datei gelesen und in die Job-Warteschlange eingefügt. Dann wird in die Hauptschleife gesprungen:

```
Scanner::loop();        # Hauptschleife
sub loop {
   while (@queue || @idle < $scanners) {
      runq;
      Event::loop;
   }
}
```

Dabei stehen die zu bearbeitenden Jobs in `@queue` und die verfügbaren Scanner-Objekte in `@idle`. Solange noch Jobs vorhanden sind (`@queue != 0`) und nicht alle (`$scanners`) Scanner idlen, wird `runq` aufgerufen und in die Hauptschleife von `Event` gesprungen.

`runq` (das steht für „run queue“) nimmt Jobs aus der Warteschlange und teilt sie verfügbaren Scannern zu. Der Algorithmus ist sehr primitiv (FCFS) und könnte wesentlich verbessert werden. Wichtig ist, daß die Lastverteilung in diesen wenigen Zeilen stattfindet und sehr gut lokalisiert und damit sehr einfach änderbar ist.

```
sub runq {
   while (@queue && @idle) {
      my $c = pop @queue;
      my $s = pop @idle;
      $s->run(@$c);
   }
   Event::unloop_all unless @queue || @idle < $scanners;
}
```

Der Aufruf von `unloop_all` beendet alle Event-Schleifen, wenn alle Jobs abgearbeitet wurden.

2.2. Job Management & Rescheduling

Um neue Jobs in das System einzufügen, gibt die Funktion `add_job`:

```
sub add_job {
   push @queue, [@_];
   $reschedule->start if @idle;
}
```

Die wichtigste Teil ist der Aufruf von `$reschedule->start`: Wenn ein Scanner verfügbar ist (`@idle` nicht leer ist), muß der Scheduler aufgerufen werden. Da der Aufruf von `add_job` sehr häufig ist, und der Scheduler (`loop`) eine Rekursion bedeutet, wird er nicht direkt aufgerufen, sondern nur, wenn sonst keine Ereignisse anliegen. Dies wird mit einem `idle`-Event-Handler erreicht, der in der globalen Variable `$reschedule` steht:

```
my $reschedule = Event->idle(
   desc => "reschedule hook",
   max => 5,
   cb => sub {
      $_[0]->w->stop;
      Event::unloop;
   }
);
$reschedule->stop;
```

`Event->idle` ist der *Konstruktor*, der einen Ereignis-Handler vom Typ „idle“ erzeugt. Die einzelnen Attribute bedeuten:

Attribut	**Beschreibung**
desc	Eine Beschreibung, z.B. für das `NetServer::ProcessTop`-Modul.
max	Zeit (in Sekunden) nach dem der Callback *auf jeden Fall* ausgeführt wird.
cb	Die Callback-Funktion, die aufgerufen wird.

Übertragen auf den `$rescheduler` bedeutet dies, daß aus der Event-Schleife gesprungen wird, wenn gerade kein Datentransfer oder sonstige Aufgaben anliegen, *oder nach fünf Sekunden*, je nachdem, was früher eintrit. Diese Einschränkung verhindert, das ein schnell eintreffender Artikel den gesamten Prozeß „am Laufen hält“ und damit verhindert, das freie (idle) Scanner nicht mit neuen Jobs versorgt werden.

Wenn der Callback angesprungen wird, bekommt er ein *Ereignis-Objekt* übergeben (ähnlich wie ein `XEvent`). Als erstes sucht er darüber (`$_[0]`) das eigentliche Objekt (`$_[0]->w`, „w“ steht für „watcher“) und ruft die `stop`-Methode auf. Damit wird verhindert, daß der Callback nicht mehr aufgerufen wird, bis er das nächste mal gestartet wird (z.B. in `add_job`). `$_[0]->w->stop` ist übrigens das gleiche wie `$rescheduler->stop`, die Variable `$rescheduler` ist wegen `my` jedoch erst *nach* dem Aufruf des Konstruktors sichtbar.

Das zweite (und wichtigste) was der Callback unternimmt, ist, den eigentlichen Scheduler wieder anzuspringen `loop`. In `loop` wurde die Hauptschleife des Event-Moduls aufgerufen (`Event::loop`): `unloop` ist das Gegenstück dazu und springt aus dieser Schleife heraus, so daß der Scheduler neue Jobs verteilen kann.

2.2.1. Beendigung eines Jobs

Wenn ein Scanner-Objekt einen Job verarbeitet hat, muß es sich wieder in die `@idle`-Queue eintragen:

```
sub idle {
   my $self = shift;
   push @idle, $self;
   $reschedule->start;
}
```

der Aufbau gleicht `add_job`.

2.3. Die Jobschleife

Für die Abarbeitung der Jobs ist die Methode `run` zuständig. Sie hat mindestens drei Parameter: `self` (das Scanner-Objekt), `host` (der NNTP-Server, inkl. Port) und `cmd` (der Jobtyp).

Da das NNTP-Protokoll „stateful“ ist, muß der aktuelle NNTP-Server und die aktuelle Gruppe gespeichert werden. Gilt der neue Job für denselben Rechner und dieselbe Gruppe (der Normalfall) passiert nichts, ansonsten wird die Verbindung zum NNTP-Server neu aufgebaut, bzw. die Gruppe gewechselt.

Das Aufbauen der NNTP-Verbindung ist ein Problem für den Event-Ansatz: ein `connect`-Aufruf *blockiert* den Prozeß, bis entweder die Verbindung steht oder ein Fehler passiert. Da ein solcher `connect` einige Sekunden benötigen kann (bei Netzwerkproblemen auch wesentlich länger), müssen sog. „non-blocking-calls“ verwendet werden.

Das ist auch der Grund, weshalb das Programm auf Standardmodule wie `IO::Socket` oder `Net::NNTP` verzichten muß: Unterstützung für nicht-blockierende Aufrufe ist kaum oder überhaupt nicht vorhanden. Das `Net::NNTP`-Modul ist in dieser Hinsicht besonders schlecht, dnen man kann die entsprechende Methoden nicht einfach in einer Subklasse überschreiben.

Der schwierigste Teil war der Aufruf von `connect`, der ebenfalls nicht blockieren sollte:

```
if (socket $fd, PF_INET, SOCK_STREAM, getprotobyname 'tcp') {
   sub TCP_NODELAY(){1} sub SOL_TCP(){6}; # linux-2.2
   setsockopt $fd, SOL_TCP, TCP_NODELAY, 1;
   fcntl $fd, F_SETFL, O_NONBLOCK;
   connect $fd, sockaddr_in $port, inet_aton($ip);
   fcntl $fd, F_SETFL, 0;
} else {
   undef $fd;
}
```

Einige Konstanten (z.B. `SOL_TCP`) sind in Perl nicht einfach zu bekommen. Da das Script mehr ein Hack als eine professionelle Anwendung ist, wurden sie einfach hardcodiert.

Wenn der Server gewechselt wird, wechselt auch der Filehandle, so daß eine neuer Event-Watcher erzeugt werden muß:

```
($self->{w} = Event->io(fd => fileno $fd, poll => 'r'))->stop;
```

2.4. NNTP-Befehle

Das NNTP-Protokoll ist sehr einfach: Kommandos bestehen aus einer einzelnen Textzeile, antworten aus einem Zifferncode und einer beschreibenden Textzeile. Artikel werden als Textblock übertragen, wobei die letzte Zeile einen einzelnen Punkt als Endekennung enthält.

Das Absetzen eines Befehls geschieht über die Methode `rcb`. Ihr werden zwei Argumente übergeben, das Kommando (ohne Zeilenende) und eine *Callback*-Funktion. Das Kommando wird an den NNTP-Server geschickt, die Callback-Funktion wird aufgerufen, wenn die erste Zeile der Antwort angekommen ist (mit dem Statuscode).

Dies wird erriecht, indem der Event-Watcher für die NNTP-Verbindung gefüttert und gestatrtet wird:

```
sub rcb {
   my $self = shift;
   my $cmd = shift;
   my $cb = shift;
   if ($cmd) {
      $self->command($cmd);
   } else {
      $cmd = "<anonymous command>";
   }

   $self->{w}->desc($cmd);
   $self->{w}->cb(sub {
      $self->{w}->stop;
```

```
      $cb->($self);
   });
   $self->{w}->start;
}
```

Mit `desc` wird die Beschreibung gesetzt, der eigentliche Callback stoppt den Watcher und ruft die ursprüngliche Callback-Funktion auf.

2.5. Lesen der Antwort

Der schwierigste Teil des Skriptes ist das zeilenweise Lesen, das vom NNTP-Protokoll vorausgesetzt wird. Da Perl von sich aus (noch) keinerlei Support dafür anbietet, mußte das Zusammensetzen der Zeilen selbst implementiert werden.

Grundlage dafür ist die Methode `refill`, die alle Zeichen liest, die angekommen sind (ohne zu blockieren) und sie in einem Puffer ablegt:

```
sub refill {
   my $self = shift;
   my $wait = shift;
   my $fd = $self->{fd};
   fcntl $fd, F_SETFL, O_NONBLOCK;
   for(;;) {
      my $r = sysread $fd, $self->{buff}, 32768, length $self->{buff};
      if ($r>0) {
         last;
      } elsif (!defined $r && $! == EAGAIN) {
         last unless $wait;
         $self->{w}->cb(sub { $self->{w}->stop; Event::unloop });
         $self->{w}->start;
         Event::loop();
      } else {
         $self->{buff} = "500 I/O error: $!\015\012.\015\012";
         delete $self->{host};
         last;
      }
   }
   fcntl $fd, F_SETFL, 0;
}
```

Das Argument `$wait` bestimmt, ob auf jeden Fall gewartet werden soll, oder ob `refill` zurückkehren soll, auch wenn keine neuen Daten verfügbar sind. Letzteres ist außerst selten der Fall und wurde entsprechend ineffizient implementiert, indem ein „leerer“ Watcher gestartet wird und dann auf dessen Unloop gewartet wird.

Als nächstes in der Hierarchy steht `getline`, das einfach die nächste Zeile liefert, notfalls durch Warten:

```
sub getline {
   my $self = shift;
   $self->refill(1) while $self->{buff} !~ s/^([^\015\012]*)\015\012//o;
   $1;
}
```

Sie ist sehr einfach: gibt es schon eine ganze Zeile im Puffer, dann schneide sie heraus und gib sie zurück. Nicht sehr effizient, aber einfach zu benutzen.

Sie wird benutzt von `response`, wo die Zeile in ihre beiden Kompomenten (Statuscode, Meldung) zerlegt wird, und die erste Ziffer des Statuscodes zurückgegeben wird (der für das weitere Vorgehen am entscheidensten ist).

```
sub response {
   my $self = shift;
   @{$self}{'code','message'} = split m/ /, $self->getline, 2;
   substr $self->{code}, 0, 1;
}
```

2.6. Scannen einer Gruppe

Um herauszufinden, welche Artikel seit dem letzten Mal neu hinzugekommen sind, wird die Statusmeldung ausgewertet, die der Server beim Wechsel in eine Gruppe liefert:

```
BEFEHL   GROUP comp.lang.perl.moderated
ANTWORT  211 125 4886 5010 comp.lang.perl.moderated group selected
```

`211` ist der Statuscode für „O.K.“, `125` ist die Zahl der Artikel, `4886` ist die erste und `5010` die letzte Artikelnummer.

Dies ist eine ideale Anwendung für `rcb`:

```
$self->rcb("GROUP $group", sub {
   if ($self->response == 2 && $self->{message} =~ /(\d+)\s+(\d+)\s+(\d+)/) {
      my($count, $first, $last, $name) = ($1, $2, $3, $3);
      if ($count) {
         $self->slog("selected group $group");
         $self->{group} = $group;
         $self->{first} = $first;
         $self->{last} = $last;
         $cb->($self);
         return;
      } else {
         $self->slog("SKIPPED empty group $group: ", substr($self->{message},0,-1));
      }
   } else {
      $self->slog("SKIPPED bogus group $group on ".$self->{host}[0].": ", substr($self-
>{message},0,-1));
   }
   $self->idle;
});
```

Etwas später wird diese Information (`first` und `last`) mit den Daten aus der SQL-Datenbank verglichen:

```
sub group_scan {
   my $self = shift;
   my $group = $self->{group};
   my $todo = new Set::IntSpan $self->{first}."-".$self->{last};
   $todo = $todo->intersect($self->gs_done->complement);
   if ($todo->empty) {
      $self->slog("[no new articles in $group]");
   } else {
      $self->slog("scanning group $group: ", $todo->run_list);
      add_job($self->{host},'A',$group,$todo);
   }
   $self->idle;
}
```

Das `Set::IntSpan`-Modul wird dazu benutzt, um aus der Menge der vorhandenen Artikel die bereits gescannten (die von `gs_done` zurückgegeben werden) zu entfernen. Ist die resultierende Menge nicht leer, wird ein neuer Job („hole alle diese Artikel“) erzeugt.

2.7. Holen eines Artikels

Das Holen geschieht in zwei Stufen. Zuerst wird die *Message-Id* mit einem `STAT`-Befehl ausgewertet. Damit wird außerdem festgestellt, ob ein bestimmter Artikel überhaupt existiert.

```
$self->rcb("STAT ".$self->{num}, \&got_stat);
```

Ein Protokollbeispiel:

```
BEFEHL  STAT 5010
ANTWORT 223 5010 <85j7jc$68n@junior.apk.net> article retrieved - request text separately
BEFEHL  STAT 4977
ANTWORT 430 No such article: 4977
```

Der Callback `got_stat` wertet diese Information aus:

```
sub got_stat {
   my $self = shift;
   my $r = $self->response;
   $self->mark_article_done;

   ($self->{mid}) = $self->{message} =~ /<([^>]+)>/g;

   if ($r == 2) {
      my $aid = sql_fetch("select count(*) from art where mid=? limit 1", "".$self->{mid});
      $self->mark_article_present;
      if ($aid) {
         sql_exec("replace into lnk values (?,?)", $self->gid, $aid);
         $self->idle;
      } else {
         $busy{$self->{mid}}++;
         $stat_article++;
         $self->rcb_dot("ARTICLE ".$self->{num}, \&got_article);
      }
   } else {
      $self->idle;
   }
}
```

Existiert der Artikel nicht, ist der Job beendet und es wird in den idle-Modus gegangen. Wurde er schon einmal geholt (z.B. in einer anderen Gruppe) wird er nicht noch einmal geholt, sondern lediglich in die Gruppe „gelinkt“ (Artikel können sehr groß werden).

Ansonsten wird ein `ARTICLE`-Befehl abgesetzt, mit dem der gesamte Artikel geholt wird.

```
BEFEHL  ARTICLE 5010
ANTWORT 220 5010 <85j7jc$68n@junior.apk.net> article retrieved - text follows
ANTWORT From: allbery@apk.net (Brandon S. Allbery KF8NH)
ANTWORT Newsgroups: comp.lang.perl.moderated
ANTWORT Subject: Re: Usefulness of Pseudo Hashes
ANTWORT Message-ID: <85j7jc$68n@junior.apk.net>
ANTWORT
ANTWORT Also sprach Alex Rhomberg <rhomberg@ife.ee.ethz.ch>
        (<384E39B8.D8635949@ife.ee.ethz.ch>):
ANTWORT +-----
ANTWORT | I wonder why pseudo hashes were invented
ANTWORT +--->8
ANTWORT
ANTWORT Sometimes you need an ordered list (so you can't use hashes) with keyed access
ANTWORT to the list (so lists/arrays are slow and a pain in the butt to use).  Pseudo
ANTWORT hashes are a better solution than the usual hack of maintaining duplicate
```

```
ANTWORT information in a hash and an array/list.
ANTWORT
ANTWORT --
ANTWORT brandon s. allbery      [os/2][linux][solaris][japh]     allbery@kf8nh.apk.net
ANTWORT system administrator         [WAY too many hats]           allbery@ece.cmu.edu
ANTWORT carnegie mellon / electrical and computer engineering                   KF8NH
ANTWORT                           Kiss my bits, Billy-boy.
ANTWORT .
```

Hierbei tritt das Problem auf, daß nach der Statuszeile ein Artikel folgt. Deshalb wird statt `rcb` die Methode `rcb_dot` benutzt (das steht für „read callback + data read until dot“):

```
sub rcb_dot {
   my $self = shift;
   my $cmd = shift;
   $self->{rcb_cb} = shift;
   delete $self->{body};
   $self->rcb($cmd, sub {
      if ($self->response == 2) {
         $self->{w}->cb([$self, 'rcb_cb']);
         $self->{w}->start;
         $self->rcb_cb;
      } else {
         $self->{rcb_cb}->($self);
      }
   });
}

sub rcb_cb {
   my $self = shift;
   $self->refill(0);
   if ($self->{buff} =~ s/^\.\015\012|^(.*?)\015\012\.\015\012//s) {
      $self->{body} .= $1;
      $self->{w}->stop;
      $self->{body} =~ s/\015\012/\n/g;
      $self->{rcb_cb}->($self, delete $self->{body});
   } elsif ($self->{buff} =~ s/^(.*\015\012)//s) {
      $self->{body} .= $1;
   }
}
```

Der komplizierteste Teil ist `rcb_cb`, in der die Artikeldaten akkumuliert werden, wozu furchtbare regexes benutzt wurden. Im Gegensatz zu vielen anderen Stellen wurden die Callbacks nicht durch Closures implementiert, da Event+Closures im allgemeinen ein großes Memory-Leak ist (soll ab Event-0.59 besser sein, aber man kann sichs nicht immer ausssuchen).

2.8. Updaten von SQL-Tabellen

Die Aufrufe `mark_article_done` und `mark_article_present` markieren einen Artikel in der Datenbank als bearbeitet bzw. vorhanden. Sie setzen einfach ein Element in der entsprechenden `Set::IntSpan`-Menge.

Diese Mengen werden in einer SQL-Tabelle gespeichert. Da sie relativ groß sind (einige Kilobytes), serh häufig geändetr werden (bis zu 100 mal pro Sekunde) und der Zielrechner sehr langsam ist, sollten die Tabellen nicht bei jeder Änderung gespeichert werden. Dies wird mit einem `idle`-Watcher erreicht, der jedesmal gestartet wird, wenn sich die Daten ändern:

```
my $save_gs = Event->idle(
                      desc => "groupstatus saver",
                      max => 60,
                      cb => sub {
                         $_[0]->w->stop;
                         # zuruckschreiben der Tabellen
                      }
                  );
$save_gs->stop;

sub mark_article_done {
   my $self = shift;
   $gs{$self->hid,$self->gid}[0]->insert($self->{num});
   $save_gs->start;
}
```

Sollte der Draht so richtig dampfen, sorgt der Timeout von 60 Sekunden dafür, daß bei einem Absturz maximal die letzte Minute fehlt. In der Praxis wird er viel häufiger aufgerufen, nämlich dann, wenn alle einkommenden Verbindungen einmal bedient wurden und noch keine weiteren Daten angekommen sind.

2.9. Künstliche „Lastsimulation“

Da der Test-Server auf der lokalen Maschine lief, mußte künstlich Last erzeugt werden, um einigermaßen wirklichkeitsnahe Ergebnisse zu erhalten. Die größten Zeitfaktoren bei NNTP sind die Latenz zum Server (abhängig von der Entfernung) und die Bandbreite.

Um eine künstliche Latenz einzuführen, wird die `command`-Funktion leicht abgeändert:

```
sub command {
   my ($self, $cmd) = @_;
   Event->timer(after => rand, cb => sub {
      $_[0]->w->cancel;
      syswrite $self->{fd}, "$cmd\015\012";
   });
}
```

Statt das Kommando sofort zu verschicken, wird ein kurzer Timer gestartet. Die Verzögerung liegt zwischen 0 und 1 Sekunde (`rand`) und sorgt für eine Streuung. Ohne diese zufällige Verzögerung würde ein unerwünschtes Bearbietungsmuster entstehen, bei dem effektiv nur ein Scan-Vorgang gleichzeitig stattfindet.

Die obige Version von `command` schneidet in ihrer Kürze recht gut gegen die „normale“ Version ab:

```
sub command {
   my ($self, $cmd) = @_;
   syswrite $self->{fd}, "$cmd\015\012";
}
```

2.10. `NetServer::ProcessTop`

Ein recht interessantes Modul ist `NetServer::ProcessTop`. Wird es benutzt, bindet es sich auf einen TCP-Port, den man per `telnet` ansprechen kann, um ein `top`-artiges Listing der Event-Watcher zu bekommen, Außerdem kann man die Watcher edieren.

Die Benutzung ist denkbar einfach:

```
eval {
   require NetServer::ProcessTop;
```

```
   NetServer::ProcessTop->new(7000);
};
```

Ein `telnet localhost 7000` erzeugt dann dieses Bild:

```
get PID=3407 @ cerebro                                          | 14:26:46 [  60s]
10 events; load averages: 0.75, 0.73, 0.00; lag  0%

  EID PRI STATE   RAN  TIME   CPU TYPE DESCRIPTION                          P1
    0   7         912  0:00 26.6%  sys idle
    3   4 zomb    227  0:00 16.9%   io ARTICLE 273573
    6   4 zomb    236  0:00 16.6%   io ARTICLE 273572
    4   4 sleep   232  0:00 16.4%   io ARTICLE 273575
    5   4 sleep   221  0:00 16.0%   io ARTICLE 273574
    9   4 wait    117  0:00  7.3% idle groupstatus saver
   10   4 wait    180  0:00  0.3% idle reschedule hook
    2   3 sleep     1  0:00  0.0% time Event::Stats
    1   3 cpu       0  0:00  0.0%   io NetServer::ProcessTop::Client localhost
    7   3 sleep     0  0:00  0.0%   io NetServer::ProcessTop
    8   4 sleep     0  0:00  0.0%   io user input
    0  -1           0  0:00  0.0%  sys other processes

%
```

Weil das Modul aber ein potentielles Sicherheitsproblem sein kann, sollte es nur zum Debuggen / Erfreuen verwendet werden.

A. Der Quellcode

```
#!/usr/app/bin/perl

# this goody scans newsgroups on any number of servers
# using more than one connection

# (c)1999 Marc Alexander Lehmann <pcg@goof.com>

$::scanners = 4; # the number of scanner "processes" to use
$::max_data = 1e6 * 30;

package Scanner;

use Event;
use Socket;
use Fcntl;
use Errno qw(EAGAIN);
use Set::IntSpan;
BEGIN { eval "use Time::HiRes 'time'" }

use sex_lib;

my $scanners;
my @idle;
my @queue;
```

```
# statistics
my $stat_article =
my $stat_stat =
my $stat_bread = 0;
my $stat_start = time;

# cmds
# S group     scan the newsgroup
# A group anum        scan the article

# utliity methods and functions

sub slog {
   my $self = shift;
   printf "%2d [%2d,%2d]: ", $self->{identifier}, scalar@queue, scalar@idle;
   print @_, "\n";
}

my $reschedule = Event->idle(
   desc => "reschedule hook",
   max => 5,
   cb => sub {
      $_[0]->w->stop;
      Event::unloop;
   }
);
$reschedule->stop;

sub runq {
   if ($stat_bread > $::max_data) {
      print "max data size exceeded, stopping...\n";
      @queue = ();
   }
   while (@queue && @idle) {
      my $c = pop @queue;
      my $s = pop @idle;
      #printf "JJJ%d @$c\n", $s->{identifier};#d#
      $s->run(@$c);
   }
   Event::unloop_all unless @queue || @idle < $scanners;
}

sub loop {
   while (@queue || @idle < $scanners) {
      #printf "main loop %d < %d\n", scalar @queue, scalar @idle;
      runq;
      Event::loop;
   }
}

sub idle {
   my $self = shift;
   push @idle, $self;
```

```
   $reschedule->start;
}

sub add_job {
   push @queue, [@_];
   #print "adding job [@_]\n";
   $reschedule->start if @idle;
}

sub shuffle_jobs {
   srand time;
   for my $i (0..$#queue) {
      my $j = $#queue - int rand $i;
      my $d = $queue[$i]; $queue[$i]=$queue[$j]; $queue[$j]=$d;
   }
}

# I/O handling -- Net::NNTP is too dumb and too slow

sub command {
   my ($self, $cmd) = @_;
   if (1) {
      syswrite $self->{fd}, "$cmd\015\012";
   } else {
      # simulate light load
      Event->timer(after => rand, cb => sub {
         $_[0]->w->cancel;
         syswrite $self->{fd}, "$cmd\015\012";
      });
   }
}

sub refill {
   my $self = shift;
   my $wait = shift;
   my $fd = $self->{fd};
   fcntl $fd, F_SETFL, O_NONBLOCK;
   for(;;) {
      my $r = sysread $fd, $self->{buff}, 32768, length $self->{buff};
      if ($r>0) {
         last;
      } elsif (!defined $r && $! == EAGAIN) {
         last unless $wait;
         $self->{w}->cb(sub { $self->{w}->stop; Event::unloop });
         $self->{w}->start;
         Event::loop();
      } else {
         $self->{buff} = "500 I/O error: $!\015\012.\015\012";
         delete $self->{host};
         last;
      }
   }
   fcntl $fd, F_SETFL, 0;
```

```
}

sub getline {
   my $self = shift;
   $self->refill(1) while $self->{buff} !~ s/^([^\015\012]*)\015\012//o;
   $1;
}

sub response {
   my $self = shift;
   @{$self}{'code','message'} = split m/ /, $self->getline, 2;
   substr $self->{code}, 0, 1;
}

sub rcb {
   my $self = shift;
   my $cmd = shift;
   my $cb = shift;
   if ($cmd) {
      $self->command($cmd);
   } else {
      $cmd = "<anonymous command>";
   }
   if ($self->{buff} =~ /\015\012/) {
      # cannot not happen normally, as this would indicate that
      # we already had a reply to a command we just send, before
      # we actually sent it.
      $cb->($self);
   } else {
      $self->{w}->desc($cmd);
      $self->{w}->cb(sub {
         $self->{w}->stop;
         $cb->($self);
      });
      $self->{w}->start;
   }
}

sub rcb_cb {
   my $self = shift;
   $self->refill(0);
   if ($self->{buff} =~ s/^\.\015\012|^(.*?)\015\012\.\015\012//s) {
      $self->{body} .= $1;
      $self->{w}->stop;
      $self->{body} =~ s/\015\012/\n/g;
      $self->{rcb_cb}->($self, delete $self->{body});
   } elsif ($self->{buff} =~ s/^(.*\015\012)//s) {
      $self->{body} .= $1;
   }
}

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

```
   my $cmd = shift;
   $self->{rcb_cb} = shift;
   delete $self->{body};
   $self->rcb($cmd, sub {
      if ($self->response == 2) {
         $self->{w}->cb([$self, 'rcb_cb']);
         $self->{w}->start;
         $self->rcb_cb;
      } else {
         $self->{rcb_cb}->($self);
      }
   });
}

# main state machine and program logic

sub hid {
   my $self = shift;
   my $hid = $hid{$self->{host}[0]} || sql_fetch("select hid from host where na-
me=?", $self->{host}[0]);
   unless (defined $hid) {
      sql_exec("insert into host (name) values (?)", $self->{host}[0]);
      $hid = sql_insertid;
   }
   $hid{$self->{host}[0]} = $hid;
}

sub gid {
   my $self = shift;
   my $gid = $gid{$self->{group}} ||
             sql_fetch("select gid from grp where name=?", $self->{group});
   unless (defined $gid) {
      sql_exec("insert into grp (name) values (?)", $self->{group});
      $gid = sql_insertid;
   }
   $gid{$self->{group}} = $gid;
}

my %gs;
my %save_gs;
my $save_gs = Event->idle(
                     desc => "groupstatus saver",
                     max => 60,
                     cb => sub {
                        $_[0]->w->stop;
                        while (my($k,$self)=each %save_gs) {
                           my($hid,$gid)=split /\0/, $k;
                           my($d,$p)=@{$gs{$hid,$gid}};
                           #print "\nSGS $d,$p $hid,$gid\n";
                           sql_exec("replace into grpstat values (?,?,?,?)",
                                    $hid, $gid, $d->run_list, $p->run_list);
                        }
                        %save_gs = ();
```

```
                    }
                );
$save_gs->stop;

sub gs_done {
   my $self = shift;
   unless (exists $gs{$self->hid,$self->gid}) {
      my ($d, $p) =
         sql_fetch("select done,present from grpstat where hid=? and gid=?",
                   $self->hid, $self->gid);
      $d = new Set::IntSpan $d;
      $p = new Set::IntSpan $p;
      $gs{$self->hid,$self->gid} = [$d, $p];
   }
   $gs{$self->hid,$self->gid}[0];
}

sub mark_article_done {
   my $self = shift;
   $self->gs_done;
   $gs{$self->hid,$self->gid}[0]->insert($self->{num});
   $save_gs{$self->hid."\0".$self->gid} = 1;
   $save_gs->start;
}

sub mark_article_present {
   my $self = shift;
   $self->gs_done;
   $gs{$self->hid,$self->gid}[1]->insert($self->{num});
   $save_gs{$self->hid."\0".$self->gid} = 1;
   $save_gs->start;
}

sub group {
   my ($self, $group, $cb) = @_;
   if ($self->{group} eq $group) {
      $cb->($self);
   } else {
      $self->rcb("GROUP $group", sub {
         if ($self->response == 2 && $self->{message} =~ /(\d+)\s+(\d+)\s+(\d+)/) {
            my($count, $first, $last, $name) = ($1, $2, $3, $3);
            if ($count) {
               $self->slog("selected group $group");
               $self->{group} = $group;
               $self->{first} = $first;
               $self->{last} = $last;
               $cb->($self);
               return;
            } else {
               $self->slog("SKIPPED empty group $group: ",
                           substr($self->{message},0,-1));
            }
         } else {
```

```
            $self->slog("SKIPPED bogus group $group on ".$self->{host}[0].": ",
                        substr($self->{message},0,-1));
         }
         $self->idle;
      });
   }
}

my %busy;

# I had to serialize this into many different subs, otherwise
# we get biiiig memory leaks in anonymous subs

sub got_article {
   my $self = shift;
   if (@_) {
      $stat_bread += length $_[0];
      sql_exec("insert into art (mid,mtime) values (?,?)", $self->{mid}, time);
      my $aid = sql_insertid;
      my $fh = storage_create $aid;
      print $fh $_[0];
      sql_exec("replace into lnk values (?,?)", $self->gid, $aid);
      print "*";
   } else {
      $self->slog($self->{mid}.": STAT yes, but no ARTICLE??");
   }
   delete $busy{$self->{mid}};
   $self->idle;
}

sub got_stat {
   my $self = shift;
   my $r = $self->response;
   $self->mark_article_done;
   ($self->{mid}) = $self->{message} =~ /<([^>]+)>/g;

   if ($r == 2 && !exists $busy{$self->{mid}}) {
      my $aid = sql_fetch("select count(*) from art where mid=? limit 1",
                          "".$self->{mid});
      $self->mark_article_present;
      if ($aid) {
         print "[-]";
         sql_exec("replace into lnk values (?,?)", $self->gid, $aid);
         $self->idle;
      } else {
         $busy{$self->{mid}}++;
         $stat_article++;
         $self->rcb_dot("ARTICLE ".$self->{num}, \&got_article);
      }
   } else {
      print "-";
      $self->idle;
   }
```

```
}

sub article_get {
   my $self = shift;
   $stat_stat++;
   $self->rcb("STAT ".$self->{num}, \&got_stat);
}

sub group_scan {
   my $self = shift;
   my $group = $self->{group};
   my $todo = new Set::IntSpan $self->{first}."-".$self->{last};
   $todo = $todo->intersect($self->gs_done->complement);
   if ($todo->empty) {
      $self->slog("[no new articles in $group]");
   } else {
      $self->slog("scanning group $group: ", $todo->run_list);
      add_job($self->{host},'A',$group,$todo);
   }
   $self->idle;
}

# start a single command action
sub run {
   my ($self,$host,$cmd,@args) = @_;
   if ($self->{host}[0] ne $host->[0]) {
      my ($hostname, $user, $pass) = @$host;
      delete $self->{group};
      eval { $self->{w}->cancel };
      my ($ip,$port) = $hostname =~ /^([^:]+)(?::(\d+))?/g;
      $port ||= 119;
      $self->slog("connecting to $ip ($port)");
      my $fd = $self->{fd} = local *HOST;

      if (socket $fd, PF_INET, SOCK_STREAM, getprotobyname 'tcp') {
         sub TCP_NODELAY(){1} sub SOL_TCP(){6}; # linux-2.2
         setsockopt $fd, SOL_TCP, TCP_NODELAY, 1;
         fcntl $fd, F_SETFL, O_NONBLOCK;
         connect $fd, sockaddr_in $port, inet_aton($ip);
         fcntl $fd, F_SETFL, 0;
      } else {
         undef $fd;
      }

      ($self->{w} = Event->io(fd => fileno $fd, poll => 'r'))->stop;
      if ($fd && $self->response == 2) {
         if ($user) {
            $self->command("AUTHINFO USER $user"); $self->response;
            $self->command("AUTHINFO PASS $pass"); $self->response;
         }
         $self->{host} = $host;
      } else {
         $self->slog("SKIPPED host $host->[0]: $!");
```

```
         delete $self->{host};
         $self->idle;
         return;
      }
   }
   if ($cmd eq 'S') {
      my ($group) = @args;
      $self->group($group, \&group_scan);
   } elsif ($cmd eq 'A') {
      my ($group,$todo) = @args;
      $self->{num} = min $todo;
      $todo->remove($self->{num});
      add_job($self->{host},'A',$group,$todo) unless $todo->empty;
      $self->group($group, \&article_get);
   } else {
      die "unknown command $cmd (@args)";
   }
}

sub new {
   my $class = shift;
   my @cmd = @_;
   my $self = bless {}, $class;
   $self->{identifier} = ++$scanners;
   $self->idle;
   $self;
}

sub statistics {
   my $time = time-$stat_start;
   print "$stat_stat STATs (~",
         int($stat_stat/$time)/10,
         "/s), $stat_article ARTICLEs, $stat_bread newsbytes read (~",
         int($stat_bread/($time*102.4))/10,"kb/s)\n";
}

if (-t STDIN) {
   Event->io(fd => 0, poll => "r", desc => "user input", cb => sub {
      my $cmd = <STDIN>;
      print "\nUSER: $cmd";
      statistics;
      if ($cmd =~ /quit/) {
         print "\07flushed queue\n";
         @queue = ();
      }
   });
}

package main;

# it does so many things, so I called it just "get"

use lib '.';
```

```
$|=1;

use Event;

eval {
   require NetServer::ProcessTop;
   NetServer::ProcessTop->new(7000);
};

print "starting $::scanners workers...\n";
new Scanner for 1..$::scanners;

print "feeding seed commands...\n";
my $host;
while (<>) {
   chomp;
   s/\s*#.*$//;
   next unless /\S/;
   if (/^server (\S+?)(?::(\S+?):(\S+?))?$/) {
      push @host, [$1, $2, $3];
   } else {
      for my $host (@host) {
         Scanner::add_job($host, 'S', $_);
      }
   }
}

#Scanner::shuffle_jobs;

print "looping...\n";
Scanner::loop();
print "done...\n";
Scanner::statistics;
```

A.1. Mehr!

Die folgenden Module/Programme/RFCs wurden für das Projekt verwendet.

- `Event` - Event loop processing. http://www.cpan.org/[1]
- `Set::IntSpan` - Manages sets of integers. http://www.cpan.org/[2]
- `RFC-977` Network News Transfer Protocol. ftp://ftp.isi.edu/in-notes/rfc977.txt
- `NetServer::ProcessTop` - Make event loop statistics easily available. http://www.cpan.org/[3]
- `Time::HiRes` - High resolution ualarm, usleep, and gettimeofday. http://www.cpan.org/[4]
- `Socket` - load the C socket.h defines and structure manipulators. (Teil der Perl-Distribution).
- `DBI` - Database independent interface for Perl
- `MySQL` SQL-Datebank. http://www.mysql.com[5].

[1] See URL http://www.cpan.org/
[2] See URL http://www.cpan.org/
[3] See URL http://www.cpan.org/
[4] See URL http://www.cpan.org/
[5] See URL http://www.mysql.com

Ein 'Hello-World!' in Gtk+

Marc Lehmann
pcg@opengroup.org

20. Februar 2000

Inhaltsverzeichnis

Gtk+ (das GNU *T*ool*K*it) ist ein relativ junges UI-ToolKit mit einem objektorientierten Design. Gtk+ selbst ist in C geschrieben, es existieren jedoch Bindungen für C++, Perl, Python und einige andere Sprachen. In diesem Vortrag wird Schritt für Schritt ein einfaches ¨Hello-World!¨-Programm erzeugt, um das ¨look&feel¨ von Gtk+ zu erfahren

1 Gtk+

Gtk+ (das *G*NU *T*ool*K*it) ist ein relativ junges UI-ToolKit mit einem objektorientiertem Design. Gtk+ selbst ist in C geschrieben, es existieren jedoch Bindings für C++, Perl, Python und einige andere Sprachen. Das Perl-Interface ist noch nicht vollkommen ausgereift, ist aber schon hervorragend benutzbar und wird immer mehr eingesetzt (z.B. für Administrationsfrontends in Debian GNU/Linux). Die Vorteile von Gtk+ gegenüber Tk und anderen Toolkits ist die hervorragende Integration: Perl-Widgets können von anderen Sprachen benutzt werden (z.B. von C) und natürlich umgekehrt. Eigene Widgets können sehr einfach erzeugt werden (im Gegensatz zu Tk), und die Gtk+-Bibliothek ist frei (im Sinne der LGPL) – auch auf anderen Betriebssystemen als UNIX.

2 Gtk

`Gtk` ist das Perl-Modul, das die Schnittstelle zur Gtk+-Bibliothek bildet. Da die meisten Menschen ¨Gtk¨ sagen, wenn sie Gtk+ meinen, wird die Perl-Schnittstelle meistens explizit ¨Gtk-Perl¨ genannt (genauso, wie das Gimp-Modul meistens Gimp-Perl genannt wird).

Zur Zeit bietet das Gtk-Modul auch Bindungen für Gnome und einige andere Bibliotheken an, möglicherweise (das ist nicht sicher) werden diese Module aber aufgetrennt werden: Gtk+ war ursprünglich das *G*imp-*T*ool*K*it, und sowohl Gtk+ als auch Gimp sind *kein* Teil des Gnome-Projektes und können deshalb ohne Gnome verwendet werden.

2.1 Ein einfaches Programm

Hier ist ein ganz einfaches Gtk-Programm. Nichtsdestotrotz öffnet es schon ein Fenster:

```
use Gtk;

Gtk->init;

$main = new Gtk::Window;

$main->show;
Gtk->main;
```

`use Gtk;` sollte klar sein. Eine Eigenheit von Gtk+ ist es, daß es zuerst explizit initialisiert werden muß (in einem Gnome-Programm würde statt `Gtk->init` ein `Gnome->init` stehen), sonst gibt es jede Menge Fehler.

Der Aufruf `new Gtk::Window` ist *ein* Weg, ein neues Fenster zu erzeugen. Neue Gtk-*Widgets* sind Anfangs aber noch unsichtbar. Damit sie angezeigt werden, muß zuerst die `show`-Methode aufgerufen werden (es gibt auch das entsprechende `hide`).

Das letzte, was unser Programm macht, ist, in die Hauptschleife von Gtk+ zu springen. Die Hauptschleife zeigt das Fenster und und – wartet auf Ereignisse:

Das Fenster ist natürlich leer und es wurden auch keine Reaktionen auf Ereignisse. Die einzige Möglichkeit, das Programm zu beenden, ist es, es zu ¨killen¨ (entweder `xkill` oder z.B. `SIGINT`).

2.2 Das Signalsystem von Gtk+

Gtk+ setzt intern auf einer Bibliothek namens `Gdk` (GNU Drawing ToolKit) auf, die eine Schnittstelle auf niedriger Ebene zum jeweiligen Fenstersystem (X, Win32, BeOS...) bildet. Diese Schnittstelle erzeugt Ereignisobjekte und reicht sie an Gtk+ weiter.

Gtk+ verteilt diese Ereignisse an seine Widgets, indem es diesen Signale schickt. Ein Signal ist z.B. eine Mausklick (`clicked`) oder das anklicken des ¨Fenster-Schließen¨-Knopfes (`delete_event`). Gtk+ kennt aber auch eigene Signal, z.B. wenn ein Objekt zerstört wird (`destroy`), was einem Destruktor entspricht.

Jeder Signal-Handler kann das Signal konsumieren, an andere Widgets weiterreichen oder einfach ignorieren (wodurch es z.B. an das Vaterwidget gelangt).

Um eine Aktion an ein Signal zu binden, kann man die `signal_connect`-Methode aufrufen:

```
$main = new Gtk::Window;
```

```
$main->signal_connect(delete_event => sub { Gtk->main_quit });
```

Nun kann man den Schließknopf des Fensters betätigen, und das ausgelöste Signal (`delete_event`) führt zu einem Aufruf von `Gtk->main_quit`. Das Beendet die Hauptschleife und damit das Programm.

2.3 Der erste Button

Nun zu etwas mehr Aktion. Ein ¨Hello-World!¨-Button muß her. Dies ist ganz einfach:

```
$button = new Gtk::Button "Hello, World!";
$main->add($button);

$button->show;
```

Das einzig neue (neben der Erkenntnis, daß es auch `Gtk::Buttons` gibt), ist der Aufruf von `$main->add`. `Gtk::Window` *ist ein* `Gtk::Container`, und erbt von diesem einige Methoden (unter anderem `add`).

Das Fenster sieht nun so aus (nicht schön, aber es wird):

Nun besetzen wir noch das `clicked`-Ereignis mit einem überflüssigen Signal-Handler:

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

2.4 Vereinfachungen

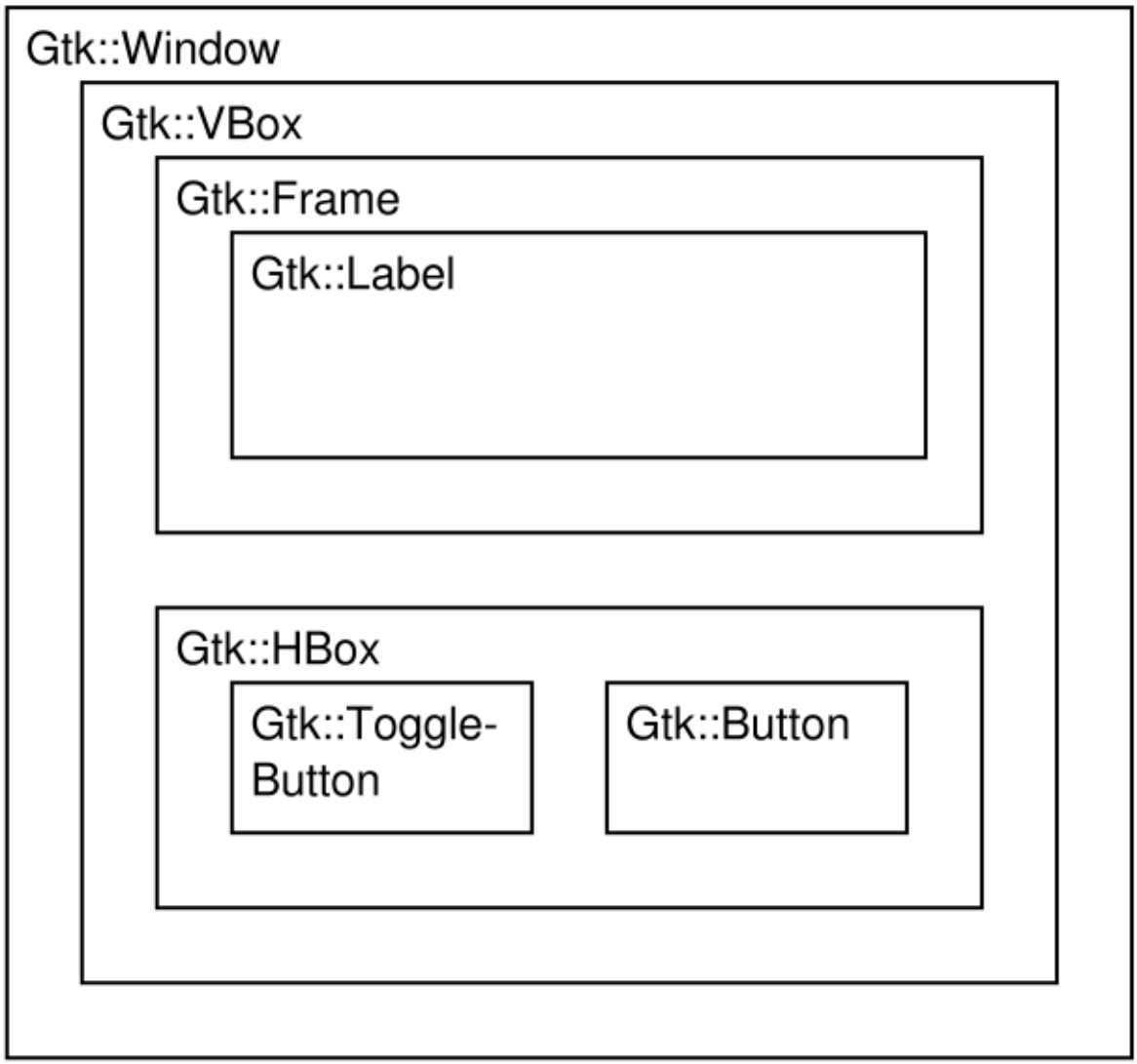
Das ganze Programm kann man (wenn man will) noch etwas einfacher haben. Zuersteinmal: mir geht dieses dauernde `$xxx->show` auf die Nerven. Weil es anderen wohl auch so ging, gibt es die Methode `show_all`. Sie ruft `show` für das Widget und, rekursiv, für alle seine Kinder auf. Man kann also alle (naja, zwei) `show`-Aufrufe durch ein einzelnes `$main->show_all` ersetzen.

Die zweite Vereinfachung ist eher Geschmackssache: Man kann die Erzeugung des Widgets und alle Initialisierungen (z.B. Signale) in einem Aufruf erledigen:

```
use Gtk;

init Gtk;

$main = new Gtk::Widget "Gtk::Window",
            -signal::delete_event => sub { Gtk->main_quit };

add $main (new Gtk::Widget "Gtk::Button",
               -label => "Hello, World!",
               -signal::clicked => sub { print "I was here!\n" });

$main->show_all;

Gtk->main;
```

2.5 Verschönerungen

Um noch ein paar Features vorzustellen sollen noch ein paar Knöpfe hinzukommen. Der ¨Hello-World¨-Knopf soll in einen Rahmen. Unter den Rahmen sollen zwei Knöpfe: der erste soll den Text an- und abschalten, der zweite soll das Programm beenden:



Die Technik, Widgets an- und abzuschalten wird gerne für Dialoge in der Art ¨Advanced Options >>¨ verwendet, um dem Benutzer mehr Optionen anzubieten.

Gtk+ bietet viele Arten von Layout-Managern. Grundsätzlich möchte es aber das Layouten des Fensters gerne selbst übernehmen. Man *kann* die Größe und Lage von Widgets selbst bestimmen, aber Gtk+ arbeitet wesentlich besser, wenn man nur Hinweise gibt. Ein solcher Hinweis kann in Form der `Gtk::VBox` und `Gtk::HBox`-Container geschehen. Diese sind Rechtecke, die mehrere Widgets enthalten können, die vertikal oder horizontal ausgerichtet werden.

Das Hauptelement des Fensters soll also eine `Gtk::VBox` werden. Oben soll eine `Gtk::Frame` stehen, und im unteren Teil soll eine `Gtk::HBox` mit den ¨Bedienelementen¨ sein (Eine `Gtk::Box` kann durchaus auch mehr als zwei Widgets enthalten). Die VBox wird einfach erzeugt mit:

```
add $main (my $vbox = new Gtk::Widget "Gtk::VBox",
                      homogeneous => 0,
                      spacing => 5,
                      border_width => 5);
```

Eine `Gtk::Box` ist `homogeneous`, wenn alle Widgets die gleiche Größe haben. Unsere VBox soll das *nicht* sein, die `Gtk::HBox` dagegen schon. Dann wird eine `Gtk::Frame` erzeugt, in die das `Gtk::Label` mit dem ¨Hello, World!¨ gepackt wird (der Layoutvorgang wird auch ¨packen¨ genannt, da die Widgets dabei in den verfügbaren Raum ¨eingepackt werden¨).

```
# Message-Frame:
add $vbox (my $frame = new Gtk::Frame "A Message:");

# Message-Label:
add $frame (my $label = new Gtk::Label "Hello, World!");
```

Die horizontale Box für die Knöpfe wird mit einem vereinfachten Konstruktor (statt dem allgemeinen `Gtk::Widget->new`) erzeugt, der nur zwei Argumente (¨homogen¨ und ¨Abstand¨) akzeptiert.

```
add $vbox (my $hbox = new Gtk::HBox 1,5);
```

Das Interessanteste an dem Programm ist zweifellos der Umschaltknopf, der die Nachricht an- und abschaltet. Dazu bedienen wir uns eines `Gtk::ToggleButtons`, der zwei Zustände (aktiv und inaktiv) besitzt:

```
add $hbox (new Gtk::Widget "Gtk::ToggleButton",
           label => "Message Shown",
           active => 1,
           signal::clicked => sub {
              $label->visible
                 ? $label->hide
                 : $label->show;
           });
```

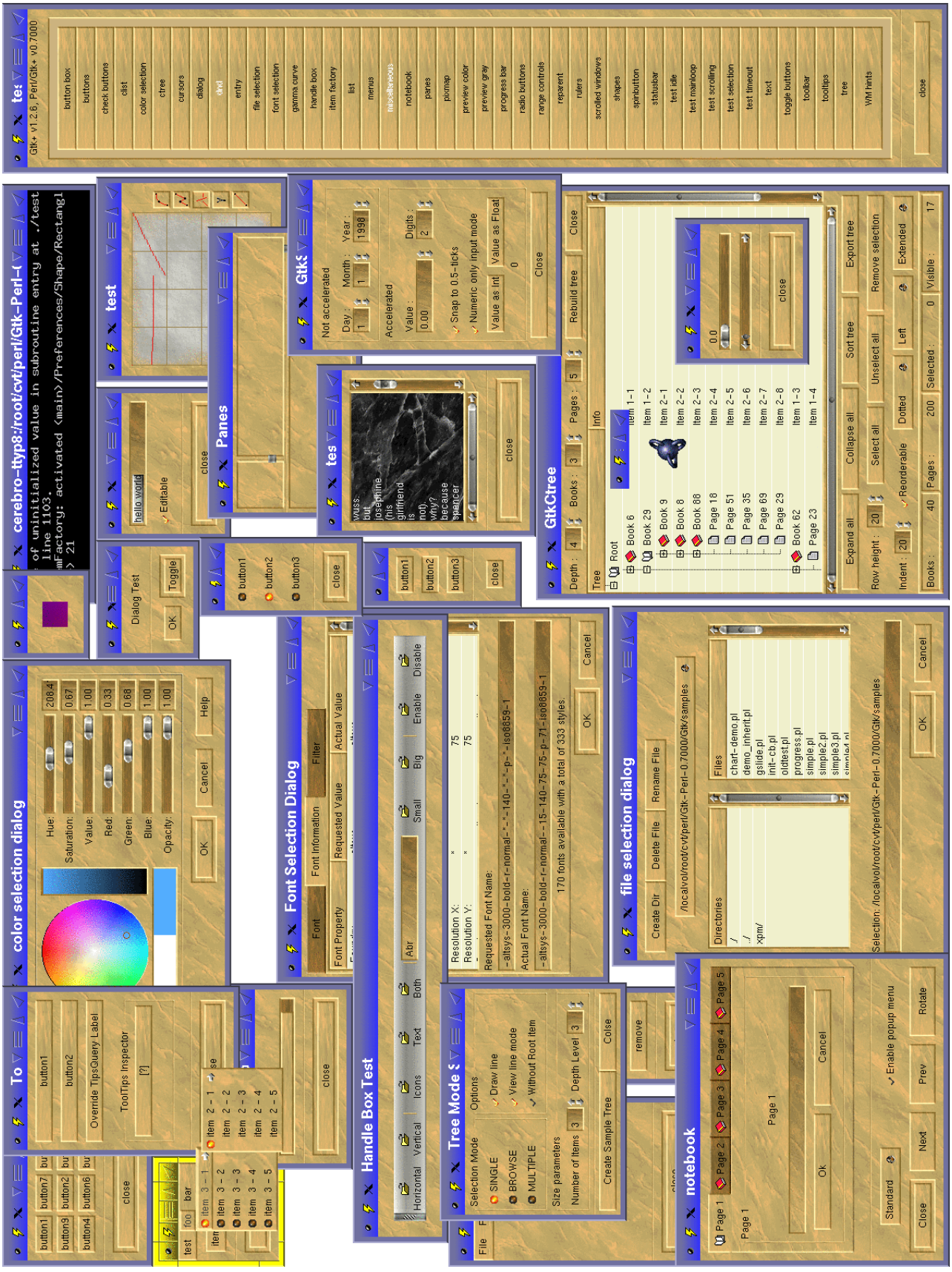
Hier ist das gesamte Programm:

```
use Gtk;

init Gtk;

$main = new Gtk::Widget "Gtk::Window",
            -signal::delete_event => sub { Gtk->main_quit };

add $main (my $vbox = new Gtk::Widget "Gtk::VBox",
                      homogeneous => 0,
                      spacing => 5,
                      border_width => 5);

# Message-Frame:
add $vbox (my $frame = new Gtk::Frame "A Message:");

# Message-Label:
add $frame (my $label = new Gtk::Label "Hello, World!");

# Button-Bar
add $vbox (my $hbox = new Gtk::HBox 1,5);

# Toggle-Button
add $hbox (new Gtk::Widget "Gtk::ToggleButton",
           label => "Message Shown",
           active => 1,
           signal::clicked => sub {
              $label->visible
                 ? $label->hide
                 : $label->show;
           });

# Close Button
add $hbox (new Gtk::Widget "Gtk::Button",
           label => "OK",
           signal::clicked => sub { main_quit Gtk });

$main->show_all;

Gtk->main;
```

3 Eine Widget-Demo

Als Abschluß folgt noch ein Screenshot, bei dem ich mich bemüht habe, die wichtigsten (schon verfügbaren) Widgets alle auf einen Bildschirm zu bekommen (das Gtk-Perl-Demoprogramm, `Gtk/samples/test.pl` war sehr hilfreich ;)

Natürlich kann man jederzeit auch eigene Widgets programmieren...



Design und Implementierung eines betriebssystem– und datenbankunabhängigen Informationssystems

Hartmut Börner
haboe@tzv.fal.de
Eildert Groeneveld
eg@tzv.fal.de
Helmut Lichtenberg
heli@tzv.fal.de

Institut für Tierzucht und Tierverhalten
(Bundesforschungsanstalt für Landwirtschaft)
Forschungsbereich Genetik und genetische Ressourcen
31535 Neustadt / Mariensee

27. Januar 2000

Inhaltsverzeichnis

1 Ausgangslage

In der Tierzucht werden ständig große Mengen von Daten erhoben bei denen unter züchterischen Aspekten die Speicherung der Verwandtschaften eine besondere Rolle spielt. Viele bestehende Datenbanken in zahlreichen Ländern erfüllen wesentliche Voraussetzungen bezüglich der Datenintegrität nicht.

Bestehende Lösungen basieren in der Regel auf proprietärer Software, v.a. der Datenbank. Einkommende Massendaten (Dateien) werden meist mit Lademodulen der jeweiligen Datenbank verarbeitet. Die Integritätsregeln für die Daten sind – soweit überhaupt vorhanden – üblicherweise in nicht genormten Erweiterungen des jeweiligen SQL-Dialekts geschrieben. Programme wie SQL*Forms (Oracle) stellen komfortable Werkzeuge zur Maskenerstellung zur Verfügung – innerhalb des jeweiligen proprietären Umfeldes. Die Nutzung proprietärer Werkzeuge und eine fehlende Generalisierbarkeit der abzubildenen Strukturen

führt dazu, daß einmal entwickelte Systeme in einem anderen Umfeld nicht verwendbar sind, was zu vielen (fast) parallelen Neuentwicklungen führt.

Vor etwa 2 Jahren begannen wir in einem kleinen Versuch, eine generische Lösung für diese Probleme zu finden und verglichen dazu zwei mögliche Entwicklungsumgebungen: die GNUstep Database Library (GDL) und Perl mit den verfügbaren Modulen.

Voraussetzung war der Einsatz einer SQL-Datenbank, die Möglichkeit zur Implementierung datenbankunabhängiger Business Rules und zur Verarbeitung verschiedener Datenströme (Dateien, Masken, Browser). Die Herstellerunabhängigkeit bezieht sich sowohl auf Datenbank wie auch Betriebssystem.

Verschiedene Gründe führten dazu, daß wir seitdem das Projekt in Perl realisieren.

2 Design

Wir gehen von 3 Schichten aus:

1. **Datenbank** (SQL-RDBMS): Diese wird im wesentlichen nur zum Speichern und Abrufen der Daten eingesetzt. Es sollen möglichst keine datenbankspezifischen Erweiterungen eingesetzt werden.

2. **Prüfschicht** (Business Rules, Constraints, Trigger): Alle Daten, die in die Datenbank kommen, müssen zur Absicherung der Datenintegrität und ggf. zur Initiierung weiterer Aktionen diesen Layer passieren. Er wird zentral durch eine einzige Konfigurationsdatei gesteuert und ist in seinem ausführbaren Code generisch, d.h. unabhängig von der jeweiligen Konfiguration und dem Inhalt der Daten. Auch die bestehende Datenbasis kann z.B. bei Änderung der Integritätsregeln jederzeit überprüft werden.

3. **Anwenderschnittstellen** (Dateien, Eingabemasken, Browser, Reports): Auch die Schnittstellen zu den Datenströmen sollen durch zentrale Parameterdateien konfiguriert werden.

Durch die völlig generische Implementierung dieses Designs ist es auch außerhalb des Bereichs der Tierzucht universell einsetzbar, ein generelles Werkzeug zum einfachen Erzeugen umfassender Informationssysteme. Definition der Datenstrukturen und Integritätsregeln, Maskenerzeugung, Datenextraktion und Reports – diese Schritte sollen für den Anwendungsentwickler und den Anwender konzentriert an einer Stelle erfolgen und ihm dabei die Freiheit der Wahl der Datenbank und des Betriebssystems lassen.

In diesem Zusammenhang sind weitere Aspekte von Bedeutung:

- Internationalisierung
  Benutzerführung und Fehlermeldungen erfolgen in der jeweiligen Landessprache. Neue Sprachen lassen sich problemlos ergänzen.

- Skalierbarkeit
  Das ganze System muß sowohl auf üblichen PC als auch im zentralen Multiuser-Rechenzentrumsbetrieb laufen.

- leichte Änderbarkeit
  Die vorhandenen Regeln / Constraints sind an zentraler Stelle leicht und ohne Programmierkenntnisse auszutauschen. Neue Regeln sind einfach zu implementieren.

3 Implementierung

Hauptentwicklungsumgebung ist Perl 5.00X auf Linux mit PostgreSQL. Erste Erfahrungen haben wir mit Sybase und Oracle sowie mit Win95 / 98 gesammelt. Im wesentlichen setzen wir folgende Perlmodule ein: DBI, DBD::Pg, DBD::Sybase, DBD::CSV, SQL::Statement, Tk, CGI, IniConf.

Modelldatei

Zentrale Rolle des gesamten Systems spielt die Modelldatei. Sie umfaßt sowohl die normalisierte Datenstruktur der Datenbank als auch den kompletten Satz aller Business Rules. Die Constraints werden nur hier definiert und sind für die gesamte Prüfschicht gültig. Neben spaltenbezogenen Constraints lassen sich auch solche für die gesamte Tabelle sowie Trigger definieren.

Die eigentliche Prüfschicht sichert völlig generisch auf Grundlage der Definitionen in der Konfigurationsdatei die Datenintegrität ab. Beim Laden werden fehlerhafte Datensätze zurückgewiesen und mit entsprechenden Fehlermeldungen versehen.

Auf Basis der Konfigurationsdatei werden die SQL-DDL-Anweisungen zum Anlegen der Datenbanktabellen ebenso wie eine grafische Darstellung der Datenbankstruktur (incl. der Fremdschlüssel) als xfig-Datei erzeugt.

Sie ist momentan eine Zusammenstellung komplexer Perl-Datentypen (Hashes mit Arrays und Subhashes). Wahrscheinlich werden die Modelldatei und die Konfigurationsdateien für Masken und Reports auf XML umgestellt, um Inhalt und Form mit vorhandenen Werkzeugen leichter überprüfen zu können.

Masken und Reports

Ähnlich wie bei der zentralen Modelldatei sollen auch die Masken und Reports an einer Stelle parametrisiert werden. Eine Implementierung liegt bereits für Eingabemasken vor.

Neben generellen Eigenschaften der Maske (Größe, Schriftarten, Titel, etc.) können auch die Eigenschaften jedes einzelnen Eingabefeldes definiert werden (editierbar / nicht editierbar, BrowseEntry bei Foreign-Key-Feldern, frei programmierbare Funktionen). Nach dem automatischen Erzeugen einer Standardmaske als 1:1-Abbildung der definierten Tabellen können diese durch Verschieben der einzelnen Elemente ansprechender und ergonomischer angeordnet werden.

Ein generischer Reportgenerator muß noch entwickelt werden, ggf. können externe Tools einbezogen werden.

Referenzdatenbank

Zur Entwicklung und Überprüfung der Module ist es notwendig, eine gemeinsame Testbasis mit allen wesentlichen Prüfkriterien zu haben.

4 Ausblick

Unser Projekt hat gegenwärtig den Arbeitstitel *pdbl* (Perl DataBase Layer) und wurde von uns unter die GNU General Public Licence (2. Version, 1991) gestellt. Neben dem Institut für Tierzucht und Tierverhalten (FAL) in Mariensee (bei Hannover) sind Tierzüchter aus Griechenland, der Slowakei, Slowenien, Tschechien, Lettland, Südafrika und Sachsen bisher an dem Projekt beteiligt. Leider fehlt es bei vielen Partnern oft an Programmiererfahrungen, v.a. bezüglich Perl. Es würde uns freuen, wenn sich weitere Entwickler beteiligen würden. Da das gesamte Projekt nicht auf die Tierzucht begrenzt und stark modularisiert ist, lassen sich gut einzelne Teilbereiche bearbeiten. Anregungen jeder Art sind uns willkommen.

Folgende Ziele werden kurz– bzw. mittelfristig angestrebt:

- Umstellung der Konfigurationsdateien auf eine einheitliche Syntax, wahrscheinlich XML.
- Entwicklung eines generischen Reportgenerators zum Erstellen, Speichern und Abrufen von eigenen Reports. Ähnlich wie bei den Masken soll die Reportbeschreibung auch in (XML-)Konfigurationsdateien erfolgen.
- Erstellung eines generischen Web-Interfaces, abgeleitet aus den Maskenkonfigurationsdateien.
- Schaffung eines Konfigurationsinterfaces für externe Massendaten aus Dateien.

- Umstellung der Netzwerkfähigkeit auf Grundlage von DBD::Proxy. Damit könnte z.B. die Datenbank unter Unix im LAN betrieben werden während Clients unter MS-Windows darauf zugreifen. Gegenwärtig kann z.B. nur über einen lokalen PostgreSQL-Client auf eine entfernte Datenbank zugegriffen werden.
- Synchronisation entfernter Datenbanken, automatische Software-Updates. Einbindung weiterer Datenbanken.
- Entwicklung eines Sicherheitskonzept mit Bereichen wie Benutzerverwaltung, Zugriffsrechte auf Datenbank und Modelldatei.
- Steigerung der Performance. Viele Teile des Quellcodes wurden erstellt, um generell die Funktionsfähigkeit des Konzeptes zu überprüfen. Es besteht noch ein weites Feld für Optimierungen.
- Erweiterung der Prüfregeln (z.B. konditionale Constraints), Verbesserung der Fehlerbehandlung bei abgewiesenen Datensätzen.
- Erstellung von speziellen Eingabemasken für professionelle Eingabe von Massendaten.
- Dokumentation für Benutzer und Entwickler
- Zusammenstellung lauffähiger Versionen, auch auf CD. Dabei müssen die zahlreichen Perl-Module eingebunden sowie Installationsroutinen entwickelt werden.

Das *pdbl*-Programmpaket kann von **ftp://ftp.tzv.fal.de/pub/pdbl** heruntergeladen werden.

Ein datenbank- und plattformunabhängiges Perl-basiertes Redaktionssystem für Intra- und Internet

Jörn Reder
joern@zyn.de
Softwareentwickler, dimedis GmbH

9. Februar 2000

1 Ein datenbank- und plattformunabhängiges Perl-basiertes Redaktionssystem für Intra- und Internet

Was macht eine Gruppe motivierter Softwareentwickler in folgender Situation?

Gegeben ist ein für einen Kunden erstelltes Redaktionssystem, welches auf einer Oracle Datenbank aufsetzt und unter Solaris läuft. Man beschließt aufgrund des Projekterfolges, diese Software als Basis für ähnliche kundenspezifische Projekte heranzuziehen. Erfreulicherweise läßt der nächste Interessent nicht lange auf sich warten, dem wirtschaftlichen Erfolg scheint nichts mehr im Wege zu stehen! Aber... der Kunde wünscht nicht Oracle auf Solaris sondern Informix auf Windows NT! Diese Kombination wird als strategische Plattform im Unternehmen des Kunden eingesetzt - eine Missionierung auf die bislang unterstützte Oracle / Solaris Plattform ist vollkommen zwecklos!

Die Antwort auf die am Anfang gestellte Frage lautet natürlich: **PORTIEREN!**

Eine Portierung kann aber unangenehme Folgen für die Softwareentwicklung haben, insbesondere wenn es sicht nicht vermeiden läßt, den Source Code auf die unterstützten Plattformen zu spezialisieren. Wenn so mehrere Versionsstände erzeugt werden, bedeutet das einen hohen Aufwand bei Entwicklung und Wartung, da viele Arbeiten u.U. doppelt gemacht, oder aufwendig in die übrigen Versionen eingepflochten werden müssen.

Doch die Gruppe motivierter Softwareentwickler hatte Glück: die Software war zu 100% in Perl geschrieben! Und genau für die gerade gestellte Aufgabe bietet diese Programmiersprache die richtigen Konzepte: sie ist unabhängig vom Betriebssystem und besitzt eine datenbankunabhängige Schnittstelle namens DBI. Prima, DBI wurde auch schon für den Zugriff auf die Oracle Datenbank verwendet, da konnte doch gar nichts mehr schief gehen...

Vom vieldiskutierten Unterschied zwischen Theorie und Praxis handelt nun dieser Artikel.

1.1 Theorie und Praxis einer Software Portierung

Das hochgesteckte Ziel lautete also: das Redaktionssystem, im übrigen @it getauft (sprich: edit), soll ohne die Erzeugung unterschiedlicher Versionsstände betriebssystem- und datenbankunabhängig sein. Unterschiedliche Fassungen der auszuliefernden Programmdateien sollen hierbei wenn möglich ebenfalls vermieden werden, so daß tatsächlich alle Plattformen aus einer einzigen Quelle heraus unterstüzt werden.

@it wurde mit der hauseigenen Entwicklungsumgebung new.spirit entwickelt, welche sich der HTML Embedding Sprache CIPP bedient. CIPP generiert CGI Programme. Das gesamte Redaktionssystem basiert

also auf dem CGI Konzept und wird vollständig über einen Webbrowser bedient. Voraussetzung auf Serverseite ist demnach ein CGI fähiger Webserver. Der Kunde setzt den Netscape Webserver ein, hier sollte es eigentlich keine Probleme geben, obwohl bislang meist der Apache verwendet wurde.

1.2 Perl und die Entwicklungsumgebung auf Windows NT

Zunächst mußte im Hause eine der Kunden-Plattform entsprechende Entwicklungsumgebung auf Windows NT eingerichtet werden. Hierzu waren noch einige Portierungsarbeiten an new.spirit und CIPP erforderlich, damit sich diese tatsächlich produktiv unter Windows NT und mit dem Netscape Webserver einsetzen ließen.

Die Entscheidung bezüglich der Perl Version fiel zugunsten ActiveState aus, da sich diese am unproblematischsten installieren ließ, weit verbreitet war und über ein großes Repository von binär installierbaren Perl Modulen verfügte.

Die Verwaltung des Source Codes erfolgte wie immer über CVS, welches auch unter NT verfügbar ist und sich glücklicherweise der Konvertierung des ASCII Formates annimmt. Probleme mit den unterschiedlichen Zeilenende Codierungen von Unix und Windows wurden hierdurch von vorne herein vermieden.

1.3 Die Praxis: sind Perl Programme unabhängig vom Betriebssystem?

Nun, Perl selbst und seine Standardbibliothek sind weitgehend unabhängig vom Betriebssystem, d.h. aber noch lange nicht, daß jedes in Perl geschriebene Programm dies ebenfalls ist. Probleme machen zunächst diese ganz offensichtlichen Unterschiede zwischen Unix und Windows, über die sich ein reiner Unix Entwickler aber in der Regel wenig Gedanken macht:

Laufwerksbuchstaben Das Konzept von Laufwerksbuchstaben ist Unix völlig fremd. Bei absoluten Pfaden ohne Laufwerksangabe (wie sie ein Unix Entwickler ganz automatisch produziert) muß beachtet werden, daß man über diese auf Laufwerk C: zugreift.

Generell empfiehlt es sich, Pfade zu Verzeichnissen und Dateien in Konfigurationsdateien abzulegen, so daß diese sich leicht für ein bestimmtes Betriebssystem anpassen lassen und eine Änderung des Quelltextes unnötig ist.

Slash vs. Backslash Diese Problematik löst Perl zum Glück weitgehend selbständig, da die mit Slash geschriebenen Pfade auch unter Windows NT wie gewohnt funktionieren, d.h. Perl kümmert sich intern um die Umsetzung auf die Backslash Schreibweise.

Probleme treten aber dort auf, wo Perl selbst nicht mehr für die Pfade verantwortlich ist, z.B. bei system() Aufrufen. Diese sind natürlich also solche schon problematisch, da z.B. so etwas unter Windows i.d.R. **nicht** funktioniert:

`system ('grep joern /etc/passwd | cut -d: -f5')` Das Voraussetzen unter Unix üblicher Kommandozeilen Tools verbietet sich von selbst.

/dev/null vs. NUL Auch Windows kennt ein NULL Device, hier heißt es allerdings schlicht 'NUL'. Die Datei '/dev/null' ist Windows unbekannt. Dies ist also auch ein Kandidat für die plattformspezifische Konfigurationsdatei.

/tmp vs. C:\TEMP Unter Unix kann man die Existenz von '/tmp' als Verzeichnis für temporäre Dateien eigentlich voraussetzen. Viele Betriebssysteme weichen hiervon aber ab, z.B. auch das von Perl unterstützte VMS. Abhilfe schafft auch hier ein Eintrag in die plattformspezifische Konfigurationsdatei, oder die Verwendung des Moduls TmpDir, welches über CPAN verfügbar ist. Ab Perl Version 5.6 wird die entsprechende Funktionalität zur Standard Perl Library gehören:

```
use File::Spec::Functions 'tmpdir';
my $tmpdir = tmpdir();
```

Binäre Dateien Dem Unix Entwickler ebenfalls völlig fremd sind Unterschiede in der Behandlung von Binär- und Textdateien. Unter Windows ist es aber notwendig, ein FileHandle, in welches man binäre Daten zu schreiben gedenkt, vor der ersten Verwendung mit der Perl Funktion binmode zu behandeln:

```
use FileHandle;
my $fh = new FileHandle;
binmode $fh;
```

Die shebang Zeile Das Konzept der shebang Zeile, die unter Unix anzeigt welcher Interpreter für die Ausführung einer Datei verantwortlich zeichnet, ist unter Windows NT unbekannt. Die Kommandozeilen Shell von Windows erkennt nur .bat und .exe Dateien als ausführbar an. Eine Möglichkeit ist es nun, den Perl Code in eine .bat Datei zu kapseln, welches bei einer großen Anzahl von Programmen sehr mühsam ist und vor allem inkompatibel zu Unix ist.

Durch den Einsatz von Cygwin, welches die wichtigsten GNU Shell Tools unter Windows verfügbar macht, z.B. auch die bash, konnte dieses Problem grundlegend gelöst werden. Der verwendete Netscape Webserver kümmert sich ebenfalls wenig um die shebang Zeile sondern bekommt den Perl Interpreter im Rahmen der CGI Konfiguration zugewiesen. Dem Apache Webserver ist im übrigen hier nicht so leicht beizukommen, da dieser die shebang Zeile sehr wohl auswertet. Hier muß also dafür gesorgt werden, daß der Perl Interpreter auch auf dem Windows System an der gewohnten Stelle (z.B. C:\USR\LOCAL\BIN\PERL) verfügbar ist.

1.4 Die Tücke liegt im Detail

Nun, neben diesen eher offensichtlichen Problemen gab es auch noch einige unaufälligere Unterschiede zwischen den Plattformen, oder auch Bugs in der ActiveSate Implementierung, die teilweise erst im Laufe der Entwicklung zu tage traten:

Dup'ing von FileHandles ActiveState Perl hatte bis zur Version 516 Probleme mit dem dup'ing von FileHandles, wenn dies in einem als Subprozess gestartetenen Programm erfolgte. Hiervon war insbesondere die Entwicklungsumgebung new.spirit betroffen, da diese Subprozesse startet, z.B. um einen Perl Syntaxcheck durchzuführen. Eine Zeit lang, mußte auf dieses Feature bei der Entwicklung verzichtet werden. Seit Build 518 ist dieses Problem aber grundlegend gelöst.

flock() auf tied hash Dateien Um konkurrierende Zugriffe auf tied hash Dateien zu kontrollieren wurde unter Unix ein flock() auf die zusätzlich im append mode geöffnete Datei gemacht, welches dort auch wunderbar funktionierte. Unter Windows versagte dieser Mechanismus, so daß die Erstellung einer eigenen .lck Datei pro tied hash erforderlich war.

crypt() Die crypt() Funktion war lange Zeit unter Windows nicht verfügbar. Er seit neueren Builds (in jedem Fall ab 516) konnte crypt() verwendet werden.

DBD::Informix Nach wie vor ist aufgrund der Lizenzpolitik von Informix keine Binär Version des Moduls verfügbar, so daß dies stets von Hand übersetzt werden muß.

Zum Zeitpunkt der Entwicklung (August 1999) gab es zudem nur experimentellen Support von DBD::Informix unter Windows NT. Das Modul ließ sich nur nach einiger Handarbeit am Makefile.PL und einiger Source Dateien übersetzen, funktionierte dann aber tadellos.

Unicode::String Auch dieses Modul bedurfte einiger Änderungen, da hier Variablen verwendet wurden, die mit einigen von ActiveState verwendeten Funktionen kollidierten (ein häufiges Problem beim Portieren von Perl XS Modulen auf ActiveState Perl). Nach dem Umbenennen der betroffenen Variablen (ein Kandidat war z.B. die Variable 'new') und dem Setzen einiger explizierter Casts, ließ sich auch dieses Modul installieren und funktionierte einwandfrei.

Mozilla::LDAP Im Rahmen des Projektes mußte zusätzlich eine Anbindung an einen bestehenden Netscape Directory Server vorgenommen werden. Die Wahl des Perl Moduls fiel auf Mozilla::LDAP, da dieses direkt auf dem LDAP SDK von Netscape aufsetzt und sehr schnell ist. Außerdem ließ es sich problemlos auch unter Windows NT installieren.

Es gab aber auch eine ganze Reihe von Perl Modulen, vor allem die der Standard Library, die ohne weiteres Zutun einwandfrei auch unter Windows NT funktionierten. Hier eine Liste der wichtigsten Module:

- SDBM_File
- FileHandle
- Text::*
- File::*
- Data::Dumper
- Date::Manip
- Storable
- DBI
- DBD::Oracle

Nach Lösung der oben beschriebenen Probleme und durch den Einsatz der von hause aus plattformunabhängigen Module lief sowohl die Entwicklungsumgebung als auch das Redaktionssystem unter Windows NT (wobei letzteres zunächst über das Netzwerk auf eine Solaris Oracle Datenbank zugriff).

Die Änderungen an @it waren hierbei sehr moderat ausgefallen, da dieses kaum Zugriffe auf das Filesystem durchführt, sondern fast ausschließlich auf der Datenbank aufsetzt. Insgesamt kann man sagen: je weniger Filesystem basierter Code vorhanden ist und je mehr Standard Perl Module zur Ausführung verschiedenster Funktionen eingesetzt werden, desto leichter fällt die Programmierung eines plattformübergreifenden Systems.

1.5 DBI und die Datenbankunabhängigkeit

Nun sollte @it aber nicht auf Oracle aufsetzen, sondern auf einer Informix Datenbank, für die es glücklicherweise umfassenden DBI Support gibt - zumindest bei Informix Versionsnummern <= 7.x. Die neueren Datentypen der Informix Universal Data Option, Versionnummern >= 9.x, werden nach wie vor nur spartanisch unterstützt. In dem Projekt wurde eine Informix 7.x Datenbank eingesetzt, hier waren also keine Probleme zu erwarten.

Das Perl Modul DBI, zusammen mit den datenbankspezifischen DBD Modulen, ist aber nur in einer Hinsicht wirklich datenbankunabhängig: es vereinheitlicht die Kommunikation zwischen Applikation und Datenbank, greift dabei aber kaum in den Inhalt der Kommunikation ein. Konkret bedeutet dies: ein SELECT Statement an eine Datenbank absetzen und das Ergebnis verarbeiten, ist aus Applikationssicht durch die Verwendung von DBI absolut einheitlich. Ob jede Datenbank das von der Applikation gesendete SELECT Statement aber auch versteht, steht auf einem ganz anderen Blatt.

De facto gibt es keinen in der Praxis tatsächlich brauchbaren SQL Standard, der von allen Datenbankherstellern gleichermaßen unterstützt wird. Die unterschiedlichen ANSI Spezifikationen sind nur ungenau umgesetzt, es gibt viele Inkompatibilitäten im Detail. Nicht von ANSI erfaßte Features sind in der Regel vollkommen unterschiedlich implementiert. Dieser Problematik nimmt sich DBI aber per Definition nicht an, es regelt eben nur die Kommunikation.

Etwas neues mußte also her, eine abstraktere Zwischenschicht, die die verschiedensten Inkompatibilitäten glättet. Zunächst wurde das CPAN befragt, ob es hier bereits Lösungen gibt, diese Recherche fiel allerdings negativ aus. Einzig das Modul DBIx::RecordSet schien in diese Richtung zu gehen. Dieses greift aber sehr stark in die Art der Datenbankprogrammierung ein. Da zu diesem Zeitpunkt bereits fast 60.000 Zeilen Code existierten, war an eine solche Umstellung nicht zu denken. Hinzu kam der fehlende offizielle Informix Support, so daß das Modul erst hätte erweitert werden müssen.

1.6 Dimedis::Sql

Aber auch ein neu entwickeltes Modul mußte dergestalt sein, daß es nicht die Anpassung der gesamten Applikation erforderlich gemacht hätte. DBI nimmt einem ja schon viel Arbeit ab, und nur etwa 20-30% der SQL Statements waren von den Inkompatibilitäten tatsächlich betroffen. So fiel die Entscheidung auf eine Architektur, die direkte DBI Zugriffe erlaubt, solange die dort verwendeten SQL Befehle von so einfacher Natur sind, daß sie überall gleichermaßen funktionieren. Das neue Modul sollte nur die Fälle übernehmen, wo die Unterschiede definitiv zu groß sind. So wurde das Modul Dimedis::Sql geschaffen, welches sich der folgenden Probleme annimmt:

- Erzeugung eindeutiger Schlüsselwerte
- Einheitliches Format für Datumsfelder
- Lesen und Schreiben von Blobs und Clobs
- Case insensitiver Vergleich von Strings
- Volltextsuche
- Outer Joins

Im folgenden werden drei der wichtigsten Funktionen von Dimedis::Sql etwas genauer beschrieben, um die Philosophie zu vermitteln, die sich dahinter verbirgt.

1.7 Erzeugung eindeutiger Schlüsselwerte

Diese elementare Funktion wird fast immer benötigt, wenn Datensätze erzeugt werden. Leider unterscheiden sich die Implementierungen der einzelnen Datenbankhersteller massiv.

Dimedis::Sql übernimmt diese Aufgabe völlig transparent. Es gibt zum Einfügen und Updaten von Daten Methoden, die so abtrakt gehalten sind, daß das Erzeugen der Schlüsselwerte kein Problem mehr darstellt.

Hier ein sehr einfaches Beispiel, welches die Anwendung von Dimedis::Sql zum Einfügen eines Datensatzen illustriert:

```
# Die Variable $dbh mus ein gultiges DBI Datenbankhandle enthalten

use Dimedis::Sql;
my $sqlh = new Dimedis::Sql ( dbh => $dbh );

my $host_id = $sqlh->insert (
    table => "hosts",
    data  => {
        id => undef,
        hostname => "www.perl.org",
    },
    type  => {
        id => 'serial'
    }
);
```

Dimedis::Sql setzt eine bestehende Datenbankverbindung voraus, und erwartet deren DBI Handle beim Erzeugen einer Dimedis::Sql Objektinstanz. Anhand des Datenbankhandles wird zur Laufzeit die verwendete Datenbankplattform erkannt und das entsprechende spezielle Dimedis::Sql Datenbankmodul geladen, welches die Dimedis::Sql für die geforderte Datenbank implementiert. Dieser Vorgang bleibt gegenüber der Applikation aber transparent, da diese nur mit dem Handle von Dimedis::Sql arbeitet.

Die hier gezeigte Insert Methode erwartet im wesentlichen drei Informationen:

1. Den Namen der Tabelle
2. Ein Hash mit den einzufügenden Daten
3. Ein entsprechendes Hash, welches besondere Spalten typisiert

Über die Deklaration der 'id' Spalte als 'serial' wird beim Einfügen durch Dimedis::Sql ein eindeutiger Schlüssel generiert. Bei einer Oracle Datenbank werden hierzu Sequences verwendet. Informix verfügt über einen serial Mechanismus, der beim Anlegen der Tabelle festlegt, daß eine Spalte einen eindeutig generierten Schlüssel erhalten soll. Auch Sybase verfügt über einen ähnlichen Mechanismus, dieser funktioniert aufgrund von Einschränkungen der Client Library aber leider nicht im Zusammenhang mit Platzhaltern. Deshalb implementiert Dimedis::Sql für Sybase einen eigenen Mechanismus, der die Schlüsselwerte in einer eigenen speziellen Tabelle verwaltet.

Die insert Methode gibt die gerade vergebene ID zurück, ein Feature auf das man nie verzichten kann, da diese ID oft anschließend als Fremdschlüssel in eine andere Tabelle eingetragen werden soll. Hierfür bietet DBI keine einheitliche Schnittstelle.

1.8 Schreiben und Lesen von Blobs

Dieses Problem könnte eigentlich auch DBI lösen, leider ist der Blob Support hier noch nicht soweit gediehen, daß er gleichermaßen auf allen Datenbankplattformen unterstützt wird.

Das Schreiben von Blobs erfolgt ebenfalls über die insert und update Methoden von Dimedis::Sql.

```
my $face_id = $sqlh->insert (
    table => "Faces",
    data  => {
        id => undef,
        face => "/tmp/face.jpg"
    },
    type  => {
        id => 'serial',
        face => 'blob'
    }
);
```

Durch die Typisierung 'blob' behandelt Dimedis::Sql die 'face' Spalte entsprechend. Wird hier ein Skalar übergeben, wird dieses als Dateiname interpretiert und der Inhalt der Datei wird als Blob in die Datenbank geschrieben. An dieser Stelle kann auch eine Skalarreferenz übergeben werden. In diesem Fall wird das referenzierte Skalar als Blob gespeichert.

Auch beim Lesen von Blobs unterscheiden sich die verschiedenen Datenbankarchitekturen stark, deshalb gibt es hierfür eine eigene Methode.

```
my $blob_sref = $sqlh->blob_read (
    table  => "Faces",
    col    => "face",
    where  => "id=?",
    params => [ $face_id ]
);
```

Hier wird der gerade erzeugte Blob in den Speicher gelesen und eine Referenz darauf zurückgegeben. Alternativ kann an die blob_read Methode auch ein Dateiname oder ein Filehandle übergeben werden, in das der gelesene Blob geschrieben wird. Die Ausgabe eines Blobs direkt aus der Datenbank heraus, beispielsweise auf STDOUT, gerät so zum Kinderspiel:

```
my $blob_sref = $sqlh->blob_read (
    table  => "Faces",
```

```
    col    => "face",
    where  => "id=?",
    params => [ $face_id ],
    filehandle => \*STDOUT
);
```

1.9 Outer Joins

Die Vereinheitlichung von Outer Joins war die bei weitem undankbarste Aufgabe. Die Implementierungen gehen bei Outer Joins unverständlich viele Wege. Auch werden nicht alle Outer Join Typen von allen Datenbanken unterstützt. Man kann drei Typen unterscheiden (wobei hier nur Left Joins betrachtet werden, da sich ein Right Join immer auf einen solchen zurückführen läßt):

Simple Left Outer Join Hier werden ein oder mehrere rechte Tabellen gegen eine linke Tabelle gejoined, wobei fehlende Elemente der rechten Tabellen mit NULL Werten aufgefüllt werden.

Nested Left Outer Join Hier erfolgt ein verschachtelter Outer Join mindestens dreier Tabellen A, B und C, wobei zuerst B und C left outer gejoined werden und das Ergebnis dieser Operation gegen A left outer gejoined wird.

Left Outer Join eines Simple Join Auch hier werden mindestens drei Tabellen gejoined. Zunächst werden B und C normal gejoined (d.h. mit Wegfall von Ergebniszeilen), und dieses Ergebnis wird left gegen A outer gejoined.

Um es kurz zu machen: nur der erste Typ wird von allen großen Datenbanken unterstützt, Sybase ist hier der Spielverderber. Oracle kennt noch die ersten beiden, Informix zeigt sich am flexibelsten und beherscht alle drei. Dimedis::Sql unterstüzt aber nur die ersten beiden Typen, da diese von den derzeit wichtigsten Systemen Oracle und Informix unterstützt werden. Für Sybase muß Typ 2 per Hand auf Applikationsebene aufgelöst werden, was dank der elementaren Hashes von Perl auch nicht zu schwierig ist.

Die Unterschiede in der Syntax sind massiv, gezeigt am Beispiel eines Simple Left Outer Joins:

```
# Oracle
select *
from   A, B
where  A.foo = B.foo (+)

# Informix
select *
from   A, outer B

# Sybase
select *
from   A, B
where  A.foo *= B.foo
```

Die left_outer_join Methode von Dimedis::Sql liefert zwei Werte zurück: $from, $where. Diese müssen an den entsprechenden Stellen im SELECT eingebaut werden, um den Outer Join zu realisieren. Hier ein Beispiel eines Simple Left Outer Joins.

```
my ($where, $from) = $sqlh->left_outer_join (
    "tableA A", ["tableB B"], "A.x = B.x"
);
```

Die genaue Beschreibung der Parameter-Liste würde hier zu weit führen, deshalb sei diese hier nur grob erklärt: die als Listenreferenz übergebene Tabelle `"tableB B"` wird an die Tabelle `"tableA A"` outer

gejoined, wobei als Join Bedingung die Gleichheit der Spalte x gilt. Durch die Verwendung dieser Aufrufparameter lassen sich auch leicht verschachtelte Joins formulieren, indem in die Listenreferenz weitere Listen geschachtelt werden.

1.10 Die Philosophie hinter Dimedis::Sql

Wie bereits erwähnt ist der wichtigste Grundsatz bei Dimedis::Sql: es nimmt sich nur der unbedingt nötigen Features an. Probleme, die rein durch die Verwendung von DBI gelöst werden können, werden von Dimedis::Sql nicht erfaßt. Dies setzt natürlich genaue Kenntnisse der zu unterstützenden Datenbankplattformen voraus, um entscheiden zu können, ob dieses oder jenes SQL Statement so von allen Datenbanken unterstützt wird.

Ein weiterer wichtiger Punkt ist: Dimedis::Sql kennt keine Methode, um ein SELECT Statement an die Datenbank abzusetzen. Eine abstrakte SELECT Schnittstelle wäre zu aufwendig, da es hier etliche verschiedene Varianten gibt. Der Aufwand all diese Spielarten über eine abstrakte Schnittstelle zu übergeben stünde in keinem Verhältnis zum Ergebnis. Vielmehr generiert Dimedis::Sql wo nötig lediglich Bestandteile von SELECT Befehlen (wie z.B. beim Left Outer Join), die dann in das eigentliche SELECT Statement eingefügt werden.

Darüber hinaus setzt Dimedis::Sql viele Konventionen bei der Definition der Tabellen voraus. Z.B. können Blobs nur verwendet werden, wenn diese über eine eindeutige WHERE Bedingung erfaßt werden können. Auch datenbankspezifische Datentypen, die nicht von Dimedis::Sql erfaßt werden, sind absolut tabu.

In der Praxis hat es sich allerdings als genau das richtige Werkzeug zur Lösung der anstehenden Probleme erwiesen: der Port von dem ursprünglich rein Oracle basiertem System zum datenbankunabhängigen hat durch Dimedis::Sql weniger als zwei Mannwochen benötigt. Zusammen mit zwei Mannwochen für die Windows NT Anpassungen konnte die Software so mit einem Zeitaufwand von nur vier Mannwochen auf der geforderten Plattform zur Verfügung gestellt werden - nicht zuletzt weil hier zu 100% auf Perl Technologie gesetzt wurde und viele hervorragende Module (insbesondere DBI) einen Großteil der Probleme schon von hause aus lösten.

Dimedis::Sql unterstützt mittlerweile auch die MySQL Datenbank, wobei man hier auf Transaktionen verzichten muß, was einen produktiven Einsatz in kritischen Bereichen verbietet. Für eine kostengünstige und trotzdem leistungsfähige Lösung ist diese Datenbank aber keine schlechte Wahl, vor allem, da diese wirklich alle von Dimedis::Sql erfaßten Feature 100%tig unterstützt, im Gegensatz zu manch anderer Datenbank der großen Hersteller! :)

Das Modul Dimedis::Sql steht unter Artistic und GNU License und kann über den Webserver der dimedis GmbH heruntergeladen werden. (http://www.dimedis.de/).

Realisierung datenbankbasierter Websites mit CIPP

Jörn Reder
joern@zyn.de
Softwareentwickler, dimedis GmbH

9. Februar 2000

Inhaltsverzeichnis

1 Realisierung datenbankbasierter Websites mit CIPP

1.1 Was ist CIPP?

CIPP (CgI Perl Preprocessor) ist ein Perl-Modul, welches eine leistungsfähige Makro-Sprache definiert, die in HTML eingebettet wird, und so die Programmierung dynamischer Webseiten erheblich vereinfacht.

Dabei werden die meisten bei der CGI Programmierung immer wiederkehrenden Aufgaben auf ein Minimum reduziert. Dazu gehören z.B. das Handling von Formularen und CGI Eingabeparametern, inkl. dem Upload von Dateien und eine leistungsfähige Datenbankschnittstelle, die natürlich auf DBI aufsetzt und somit die Kommunikation mit allen unterstützten Datenbanken vereinheitlicht. Es kann auch beliebiger Perl-Code auf einfache Weise in HTML Code eingebettet werden und umgekehrt.

Um große Applikationen sinnvoll zu strukturieren gibt es das Konzept von Includes, anhand derer, unter voller Kontrolle der Schnittstellen, Untereinheiten gebildet werden können. Darüber hinaus können sehr einfach echte objektorientierte Module mit CIPP programmiert werden, die 1:1 auf das Modulkonzept von Perl aufsetzen, so daß der in objektorienter Programmierung erfahrene Perl Programmierer hier nichts vermißt.

1.2 Syntaktisches Grundprinzip: HTML Embedding

CIPP Befehle, oder auch CIPP Tags, werden in HTML eingebettet und beginnen mit der Zeichenfolge <? und unterscheiden sich so von den HTML Tags, die mit einem enfachen < Zeichen beginnen. Geschlossen werden die CIPP Tags, wie HTML Tags auch, mit einem > Zeichen. Dabei existieren - wie auch in HTML - CIPP Tags, die nicht geschlossen werden, sowie Container Tags, die einen Block umschließen, und somit auch wieder geschlossen werden müssen.

Optionen werden bei CIPP Tags auf dieselbe Art und Weise angegeben, wie bei HTML, indem einer Option ihr aktueller Wert mit dem = Zeichen zugewiesen wird. Wenn der Wert Leerzeichen enthält, muß dieser in Anführungszeichen geschrieben werden, ansonsten können diese weggelassen werden.

```
<?SINGLETAG OPTION="VALUE" ...>
<?CONTAINERTAG OPTION="VALUE" ...>
...
<?/CONTAINTERTAG>
```

Abbildung 1: CIPP Tags

1.3 Ein Beispiel

Anhand eines nicht ganz trivialen Beispiels läßt sich die generelle Funktionsweise des Embedding Prinzips effizient erläutern. Ebenso zeigt ein Blick auf die Syntax, daß es für einen Perl Programmierer ein leichtes ist, CIPP zu erlernen, da wo möglich auf Perl Syntax zurückgegriffen wird, bzw. diese tatsächlich von CIPP unverändert eingebunden wird.

Die selbstgestellte Aufgabe: es soll ein kleines Frontend angeboten werden, um ein Bild zusammen mit einem Namen in einer MySQL Datenbank zu speichern. Das Frontend besteht aus drei Seiten und somit aus drei CIPP Programmen, plus einer Include Datei, die für ein einheitlicheres Aussehen sorgt:

1. Formular zur Eingabe der Daten: ask.cipp
2. Ein Include für einen einheitlichen Header: header.inc
3. Programm zum Speichern der Daten: save.cipp
4. Programm zur Ausgabe des Bildes direkt aus der Datenbank heraus: image.cipp

1.4 Formular zur Eingabe der Daten: ask.cipp

Diese Seite ist sehr einfach und enthält auch noch nicht viele CIPP Befehle. Wie man sieht können CIPP und HTML Tags beliebig kombiniert werden. Einige Tags können ihre Verwandschaft zu HTML nicht verbergen, andere haben mehr Erklärungsbedarf. Generell gilt: um die Ausgabe von HTML Code muß sich der Programmierer nicht mehr kümmern. Dies geschieht durch das Einbetten des CIPP und Perl Codes in HTML sozusagen automatisch.

Die erste Zeile z.B. deklariert einen optionalen Eingabeparameter, d.h. der CGI Parameter 'name' kann an das Programm übergeben werden (Beispiel: http://localhost/ask.cipp?name=Foo), muß aber nicht. Die Variable $name wird an dieser Stelle implizit mit my deklariert. CIPP Programme laufen mit dem Compiler Pragma 'use strict' und verlangen deshalb die Deklaration aller Variablen. Mit dem `<?INTERFACE>` Befehl können auch zwingend notwendige Eingabeparameter definiert werden (`<?INTERFACE INPUT="$foo, $bar">`). Fehlen diese, erzeugt CIPP automatisch eine entsprechende Exception samt Fehlermeldung.

Der `<?INCLUDE>` Befehl bindet die angegebene Datei ein, wobei als Eingabeparameter ein Titel übergeben wird. Der Inhalt und die Funktionsweise der Include Datei wird weiter unten erläutert.

Die beiden `<?INPUT>` Befehle erzeugen die entsprechenden INPUT HTML Tags, wobei eine gegebenes VALUE Option automatisch richtig gequotet wird. HTML Fehler durch Variablen-Inhalte, mit denen der Entwickler nicht gerechnet hat (z.B. doppelte Anführungszeichen) gehören hiermit der Vergangenheit an.

```
<?INTERFACE OPTIONAL="$name">

<html>
<head><title>Deutscher Perl Workshop 2000 - CIPP Beispiel</title></head>
<body>

<?INCLUDE NAME="header.inc" title="CIPP Beispiel">

<FORM ACTION="save.cipp" ENCTYPE="multipart/form-data">

  <p>
  Name:
  <?INPUT TYPE=TEXT NAME=name VALUE=$name>

  <p>
  GIF Datei:
  <?INPUT TYPE=FILE NAME=image>

  <p>
  <?INPUT TYPE=SUBMIT VALUE=" Send ">

</FORM>

</body>
</html>
```

Abbildung 2: Das CIPP Programm ask.cipp fragt nach einigen Informationen

1.5 Funktionsweise von CIPP

Zum besseren Verständnis der folgenden Beispiele sei kurz die grundliegende Funktionsweise von CIPP erläutert.

Wie zu Anfang erwähnt, handelt es bei CIPP um einen Präprozessor. Das Modul verarbeitet CIPP Source Code Datei zu reinem Perl Code, der anschließend von Perl ohne weiteres Zutun von CIPP direkt ausgeführt werden kann. Hierbei sind unterschiedliche Einsatzgebiete möglich, die von reinem CGI bis zum Einsatz als Apache Modul reichen. Die unterschiedlichen Umgebungen für CIPP werden in einem eigenen Kapitel noch ausführlich erläutert.

Das Grundprinzip von CIPP ist, für jeden Teil des CIPP Programms den korrespondierenden Perl Code zu generieren. D.h. aus den reinen HTML Blöcken werden print() Anweisungen generiert, im Perl Code zum <?INPUT> Befehl verbirgt sich der Aufruf zum Quoten des VALUEs usw. Bei den später gezeigten Datenbankbefehlen wird deutlich, wieviel Arbeit dem Programmierer durch die Generierung des Perl Codes abgenommen wird.

1.6 Das Header Include: header.inc

```
<?INCINTERFACE INPUT="$title">

<h1>Deutscher Perl Workshop 2000: $title</h1>
<hr>
```

Abbildung 3: Das CIPP Include header.inc gibt eine einheitliche Überschrift aus

Includes bieten eine Schnittstelle über den <?INCINTERFACE> Befehl an, der einen ähnlichen Aufbau wie der bereits beschriebene <?INTERFACE> Befehl hat (welcher allerdings die Schnittstelle eines CGI Programms festlegt). Auch hier können notwendige und optionale Parameter deklariert werden. Zusätzlich gibt es noch die Möglichkeit, über benannte Ausgabeparameter auch Werte wieder zurückzugeben. Über die Ein / Ausgabeschnittstellen von Includes können alle Variablentypen von Perl, also Skalare, Listen und Hashes übergeben werden, so daß es hierbei keinerlei Einschränkungen gibt.

Dieses Include empfängt also den notwendigen Parameter $title und erzeugt aus diesem eine einfache einheitliche HTML Überschrift. Dabei wird die Variable $title einfach innerhalb des HTML Kontextes verwendet, zum Einbinden von Variablen bedarf es also keiner besonderen Syntax.

1.7 Programm zum Speichern der Daten: save.cipp

Anhand dieses Programms wird vor allem das leistungsfähige Exception Handling sowie die Datenbankschnittstelle von CIPP deutlich. Es ist deshalb auch etwas länger und komplizierter als die bisherigen Beispiele. Deshalb erfolgt die Listung und Erklärung in zwei Teilen, zudem sind die Zeilen durchnummeriert, um bei den Erklärungen auf den Quelltext Bezug nehmen zu können.

Die folgenden neuen Befehle werden hier verwendet:

<?MY>**, Zeile 3** Dieser Befehl deklariert die aufgelisteten Variablen mit my. Wie bereits erwähnt müssen alle verwendeten Variablen vor ihrer Verwendung deklariert werden, da CIPP das Compiler Pragma 'use strict' verwendet.

Kommentar, Zeile 5 Zeilen, die mit einem # Zeichen (nach optionalem Whitspace) beginnen, werden von CIPP als Kommentarzeilen ignoriert. Das Setzen von Kommentaren mit # hinter einem HTML oder CIPP Tag ist nicht möglich, Kommentare müssen am Anfang der Zeile stehen.

Diese Kommentare werden vom Präprozessor vollständig eliminiert, sie erscheinen weder im generierten Perl Programm noch in dem von diesem erzeugten HTML Code.

```
 1  <?INTERFACE OPTIONAL="$name, $image">
 2
 3  <?MY $error_message>
 4
 5  # Sind Bild und Name angegeben worden?
 6
 7  <?IF COND="$image ne '' and $name ne ''">
 8    <?TRY>
 9      # Bild im Filesystem ablegen
10      <?SAVEFILE VAR="$image" FILENAME="/tmp/image">
11
12      # Bild in Speicher laden
13      <?MY $blob>
14      <?PERL>
15        open (IN, "/tmp/image")
16            or die "open\tcan't read /tmp/image";
17        $blob = join ('', <IN>);
18        close IN;
19      <?/PERL>
20
21      # Daten in Datenbank ablegen
22      <?SQL SQL="insert into Faces (name, image)
23                 values (?, ?)"
24            PARAMS="$name, $blob">
25      <?/SQL>
26
27    <?/TRY>
28
29    <?CATCH MY EXCVAR="$exc" MSGVAR="$msg">
30      <?VAR NAME=$error_message>
31        Es ist ein Fehler aufgetreten: $exc $msg
32      <?/VAR>
33    <?/CATCH>
34
35    <?PERL>
36      # temp. Datei loschen
37      unlink "/tmp/image";
38    <?/PERL>
39
40  <?ELSE>
41    <?VAR NAME=$error_message>
42      Bitte Namen und Bild angeben
43    <?/VAR>
44  <?/IF>
```

Abbildung 4: Speichern von Name und Bild in der Datenbank, erster Teil von ask.cipp

<?IF> <?ELSE>**, Zeilen 7 und 40** Die Bedingung für den <?IF> Befehl wird als Perl Bedingung übergeben. CIPP macht hierbei nichts anderes, als eine Perl if () {} Konstruktion zu erzeugen, die die gegebene Bedingung enthält.

<TRY> <?CATCH>**, Zeilen 8 und 27** Über diesen Mechanismus wird das Exception Handling implementiert. Dabei werden innerhalb des <?TRY> Blocks auftretende Ausnahmen abgefangen. Der nachfolgende <?CATCH> Block wird im Falle einer Ausnahme ausgeführt, dabei erhalten hier die Variablen $exc und $msg die Bezeichnung der Exception (i.d.R. der Name des CIPP Befehls, der die Ausnahme erzeugt hat) und eine ausführliche Fehlermeldung.

Durch das Kapseln der Dateisystem- und Datenbankoperationen in den <?TRY> Block, werden hier folgende Fehlerfälle abgefangen:

1. Primary Key Verletzung bei der „name“ Spalte der Tabelle Faces. Die Namen müssen eindeutig sein.
2. Fehler beim Empfangen der Bilddatei.
3. Fehler beim Schreiben der temp. Bilddatei in das Dateisystem.

Dieser Befehl speichert die über einen Client Fileupload empfangene Datei in das Dateisystem. CIPP führt hier eine sehr genaue Erfolgskontrolle durch und erzeugt Ausnahmen, wenn Fehler beim Empfangen oder Schreiben der Datei auftreten.

<?SAVEFILE>**, Zeile 10** <?PERL>**, Zeile 14** Für CIPP natürlich ein Leichtes: der <?PERL> Befehl bindet beliebigen Perl Code ein, d.h. CIPP übernimmt diesen einfach ohne Änderungen in den generierten Code. Über den Befehl <?HTML> ist es im übrigen jederzeit möglich, wieder HTML Code innerhalb von einem <?PERL> Block erzeugen zu lassen. Überhaupt jeder CIPP Befehl kann innerhalb eines <?PERL> Blocks verwendet werden, so daß häufige Wechsel zwischen Perl und HTML Kontext vermieden werden.

<?SQL>**, Zeile 22** Ganz klar eine Stärke von CIPP: die Ausführung von SQL Befehlen. An dieser Stelle wird ein INSERT durchgeführt, der entsprechende CIPP <?SQL> ist in diesem Fall sehr einfach. Über den Parameter SQL wird der eigentliche SQL Code übergeben, der in diesem Fall Platzhalter enthält. Die dazugehörigen Parameter werden mit der PARAMS Option gelistet. Im zweiten Teil des Programms wird ein SELECT durchgeführt, für den weitere Optionen notwendig sind. Dazu mehr im zweiten Tail von save.cipp.

Der aufgeweckte Leser wird sich wahrscheinlich wundern, wie das Absetzen von SQL Befehlen ohne einen Datenbank-Connect funktionieren kann. Die Antwort hierauf ist: der entsprechende Programmcode zum Datenbank-Connect wird von CIPP automatisch generiert, wenn das Programm SQL Befehle enthält. Die hierzu erforderlichen Zugangsdaten kommen aus einer globalen Konfiguration. Mehr dazu im Kapitel „Einsatzumgebungen von CIPP“.

<?VAR>**, Zeile 30** Mit dem <?VAR> Befehl wird eine Variable definiert, dabei muß sich der Entwickler nicht um das Quoting von Sonderzeichen kümmern. Unter Angabe des optionalen Parameters MY kann dabei auch gleichzeitig eine Variable deklariert werden.

Der erste Teil des Programms war für die Speicherung der Daten verantwortlich. Der zweite Teil gibt eine Antwortseite aus, die darüber hinaus alle in der Datenbank enthaltenen Namen und Bilder ausgibt.

Hier gibt es die folgenden neuen Befehle:

<?GETURL>**, Zeilen 56 und 70** Dieser Befehl erzeut eine korrekte URL inkl. Übergabeparametern. Die Formatierung und das Quoting der Parameter werden dem Programmierer hierbei abgenommen.

In Zeile 56 soll im Falle eines Fehlers ein Link auf die Startseite erzeugt werden, der den Parameter für den Namen schon vorbelegt hat.

Der <?GETURL> Befehl in Zeile 70 erzeugt eine Bild URL auf das CIPP Programm image.cipp. Auch dieses bekommt als Parameter den Namen des Bildes übergeben. image.cipp wird dann das Bild direkt aus der Datenbank heraus ausgeben.

```
46  <html>
47  <head><title>Deutscher Perl Workshop 2000</title></head>
48  <body>
49
50  <?INCLUDE NAME="header.inc" title="Bild Speichern">
51
52  <?IF COND="$error_message">
53    <p>
54    Oha, da ist ein Fehler aufgetreten:<br>$error_message
55
56    <?GETURL NAME="ask.cipp" PARAMS="$name" MY URLVAR="$url">
57
58    <p>
59    <a HREF="$url">Nochmal versuchen</a>
60
61  <?ELSE>
62
63    <p>
64    Liste aller in der Datenbank enthaltenen Bilder:
65
66    <?SQL SQL="select name
67               from   Faces
68               order  by name"
69          VAR="$name">
70      <?GETURL NAME="image.cipp" MY URLVAR="$url"
71               PARAMS="$name" >
72      <p>
73      $name<br>
74      <img src="$url">
75    <?/SQL>
76
77    <p>
78    <a HREF="ask.cipp">Zuruck zum Anfang</a>
79
80  <?/IF>
81
82  </body>
83  </html>
```

Abbildung 5: Ausgabe eine Liste aller Bilder aus der Datenbank, zweiter Teil von save.cipp

<?SQL>, Zeile 66 Dies ist ein Beispiel für ein SQL SELECT Statement. In diesem Fall hat der SQL Befehl keine Platzhalter, so daß auf die Option PARAMS verzichtet werden kann. Die Option VAR nimmt eine Liste von Variablen auf, in die das Ergebnis des SELECT Statements geschrieben werden soll.

Der Inhalt des <?SQL> Blocks wird hierbei für jede gelieferte Zeile wiederholt, wobei die in VAR gelisteten Variablen die entsprechenden Werte für Zeile erhalten. Innerhalb des Blocks kann über diese Variablen also auf das Ergebnis des SELECT Statements zugegriffen werden.

1.8 Ausgabe eines Bildes direkt aus der Datenbank heraus: image.cipp

Zu guter letzt fehlt nun noch das Programm image.cipp, welches die eigentliche Ausgabe eines Bildes aus der Datenbank erledigt.

```
<?AUTOPRINT OFF>

<?INTERFACE INPUT="$name">

<?PERL>
  <?SQL SQL="select image
             from   Faces
             where  name=?"
        PARAMS="$name"
        MY VAR="$image">
  <?/SQL>

  print "Content-type: image/gif\n";
  print "Content-length: ", length($image), "\n\n";
  print $image;
<?/PERL>
```

Abbildung 6: Ausgabe eines Bildes direkt aus der Datenbank: image.cipp

Da dieses Programm keine anderen Daten als den HTTP Header und das Bild selbst ausgeben darf, wird der CIPP Präprozessor gleich in der ersten Zeile angewiesen, selbständig keine print() Befehle mehr zu erzeugen. Nach der Verwendung von <?AUTOPRINT OFF> ist der Entwickler nun für alle Ausgaben selbst zuständig. Ausgaben müssen also explizit mit einem print() Befehl innerhalb eines <?PERL> Blocks durchgeführt werden.

Der innerhalb des <?PERL> Blocks eingebettete <?SQL> Befehl liest das Bild aus der Datenbank aus und schreibt es in die Variable $image. Wir setzen voraus, daß es sich bei den Bildern um GIFs handelt, deshalb wird ein entsprechender HTTP Header erzeugt. Das Feld „Content-length“ kann leicht über die Größe der Bild Variablen ermittelt werden. Schließlich wird das Bild selbst einfach ausgegeben.

1.9 Einsatzumgebungen von CIPP

CIPP ist sehr flexibel bezüglich seiner möglichen Einsatzumgebungen. Es gibt drei deutlich voneinander verschiedene Architekturen:

Erzeugung von ausführbaren CGI Programmen Die Perl Entwicklungsumgebung new.spirit setzt CIPP in dieser Form ein. Ursprünglich war CIPP ein fester Bestandteil von new.spirit, erst später wurden die weiteren Module entwickelt.

Der Vorteil dieser Vorgehensweise liegt vor allem darin, daß der originale CIPP Quellcode beim Entwickler verbleiben kann und nur die generierten CGI Programme auf dem Produktivsystem installiert werden müssen. Ein bei der Entwicklung kommerzieller Programme nicht ganz unwichtiger Aspekt.

Die so von CIPP generierten CGI Programme sind mod_perl fest, d.h. können ohne Änderungen direkt mit mod_perl und seinem Apache::Registry Modul eingesetzt werden.

Ausführung über einen CGI Wrapper Diese Funktionalität wird durch das Perl Module CGI::CIPP bereitgestellt. Der Entwickler muß nur ein kleines Wrapper CGI schreiben, welches genau eine Funktion von CGI::CIPP aufruft, zusammen mit einigen Konfigurationsparametern. Dieser CGI Wrapper wird dann als Handler für die CIPP Dateien im Webserver konfiguriert und führt so transparent .cipp Dateien aus, die im htdocs Verzeichnis des Webservers liegen.

Dieser Wrapper läßt sich hervorragend mit dem Perl Modul CGI::SpeedyCGI kombinieren. Dieses ermöglicht, ähnlich wie das Apache Modul mod_perl, eine persistente Ausführung von Perl Programmen. So müssen die von CIPP generierten Perl Programme nur beim ersten Aufruf von Perl compiliert werden, bei Folgeaufrufen sinken die Antwortzeiten signifikant, da diese Requests nun ohne diesen Overhead verarbeitet werden können.

Integration in den Webserver Apache über mod_perl Eigens hierfür wurde das Perl Modul Apache::CIPP entwickelt, welches einen Apache Request Handler definiert. Das Apache Modul mod_perl kann damit so konfiguriert werden, daß z.B. Dateien mit der Endung .cipp im htdocs Verzeichnis über diesen Request Handler abgewickelt werden. Auch hierbei ist weiterhin nur die Angabe einiger Konfigurationsparameter notwendig, diesmal in der Apache Konfigurationsdatei.

Bezüglich der Performancevorteile gilt hier dasselbe, was im vorigen Abschnitt zum Thema SpeedyCGI angemerkt wurde. Durch die Persistenz des Perl Interpreters im Apache Kern entfällt der Overhead des erneuten Compilierens für jeden Request.

1.10 Der CGI Wrapper unter Einsatz von CGI::CIPP

Es folgt ein Beispiel für einen CGI::CIPP Wrapper, der zusammen mit CGI::SpeedyCGI eingesetzt wird. Anschließend wird kurz erläutert, wie ein Apache Webserver konfiguriert werden muß, damit dieser Wrapper transparent die Ausführung von CIPP Seiten erledigt.

Über die shebang line wird die Ausführung unter CGI::SpeedyCGI aktiviert. Dabei bedeuten die Parameter, daß das Programm maximal 100 Requests beantwortet bzw. sich nach einer idle time von 60 Sekunden selbst beendet. Das hier unter /usr/local/bin/speedy abgelegte Binary ist Bestandteil der CGI::SpeedyCGI Distribution.

Das Modul CGI::CIPP definiert die Klassenmethode CGI::CIPP->request, die die gesamte Ausführung eines einzelnen CIPP Requests abwickelt. Die Methode erwartet ein Hash mit den folgenden Eingabeparametern:

document_root In diesem Verzeichnis liegen alle Dateien des Webservers, bzw. insbesondere die CIPP Dateien. Es ist am einfachsten, hier die Document Root des Webservers zu wählen, um Verwirrungen bei den Pfaden zu vermeiden.

wrapper_url Die URL unter der der CGI Wrapper selbst zu erreichen ist.

directory_index Welche Datei soll als Directory Index herangezogen werden, wenn auf einen Ordner zugegriffen wird?

cache_dir Dieses Verzeichnis verwendet CGI::CIPP um einmal übersetzte CIPP Programme zu cachen. Der Präprozessor Lauf wird also nur dann durchgeführt, wenn sich die entsprechende CIPP Quelldatei seit der letzten Übersetzung auch geändert hat.

Beim Einsatz mit CGI::SpeedyCGI oder als Modul im Apache wird noch ein weiterer Cache aktiviert: der generierte Perl Code wird als Subroutine im Speicher gehalten und muß so bei Folgeaufrufen

```
#!/usr/local/perl/bin/speedy -- -r100 -t60

use strict;

use CGI::CIPP;

CGI::CIPP->request (
        document_root => "/home/joern/projects/ZAS/htdocs",
        wrapper_url => "/cgi-bin/cipp",
        directory_index => "index.cipp",
        cache_dir => "/tmp/speedycipp",
        databases => {
                zas => {
                        data_source => 'dbi:mysql:zas',
                        user => 'joern',
                        password => 'topsecret',
                        auto_commit => 1
                }
        },
        default_database => "zas",
        lang => 'EN'
);
```

Abbildung 7: Ein CGI::CIPP Wrapper unter Einsatz von CGI::SpeedyCGI

nicht erneut von Perl compiliert werden. Dies ähnelt dem vom Apache Modul Apache::Registry verfolgen Ansatz und beschleunigt die Ausführung von CIPP Seiten enorm.

databases Dieses Hash definiert die Datenbanken, auf die die CIPP Programme zugreifen. Dabei hat jede Datenbank einen Identifier, dem die Zugangsdaten zugeordnet werden.

CIPP Programme können beliebig viele verschiedene Datenbanken im Zugriff haben. Unter Bezug auf den hier vergebenen Namen können die unterschiedlichen Datenbanken angesprochen werden.

default_database Diese Datenbank wird angesprochen, wenn bei SQL Befehlen keine Datenbank explizit angegeben wurde.

lang Die CIPP Fehlermeldungen sind derzeit in Englisch und Deutsch verfügbar. Hier kann demzufolge 'EN' und 'DE' angegeben werden.

Im Apache Konfigurationsfile müssen nun folgende Einstellungen vorgenommen werden, durch die der CGI Wrapper für Ausführung von .cipp Dateien herangezogen wird:

```
ScriptAlias /cgi-bin /path/to/cgi-bin

AddType x-cipp-execute cipp
Action x-cipp-execute /cgi-bin/cipp
```

Abbildung 8: Apache Konfiguration für den CGI Wrapper

In diesem Fall heißt der Wrapper also schlicht 'cipp' und liegt in '/path/to/cgi-bin'. Über die ScriptAlias Direktive wird dieses Verzeichnis als CGI ausführbar deklariert. Die AddType Direktive weist an, daß Dateien mit der Endung .cipp über den Handler x-cipp-execute ausgeführt werden sollen, welcher anhand der Action Direktive auf /cgi-bin/cipp gemapped wird. Voila! :)

1.11 Dokumentation

Die CIPP Dokumentation liegt in englischer Sprache vor. Diese kann als besonders gut druckbares PDF vom CPAN heruntergeladen werden. Darüber hinaus gibt es das POD Modul CIPP::Manual, welches zum Standardumfang der CIPP Distribution gehört, und mit perldoc in einer Kommando Shell betrachtet werden kann. Dabei sind die PDF Version und CIPP::Manual stets auf demselben Stand, da beide aus ein und derselben Datenquelle generiert werden.

1.12 Verfügbarkeit und Referenzen

CIPP wird bei der dimedis GmbH seit fast drei Jahren im Rahmen der Entwicklungsumgebung new.spirit produktiv eingesetzt. Seit Ende 1998 steht CIPP unter GNU und Artistic License und kann vom CPAN heruntergeladen werden:

http://www.perl.com/CPAN/modules/by-module/CIPP

Die Firma dimedis GmbH hat zahlreiche Projekte unter Einsatz von new.spirit und CIPP realisiert, hier eine Auswahl aktueller URLs:

- http://www.moebelmesse.de/
- http://www.emimusic.de/
- http://www.mwmtv.nrw.de/
- http://www.fuetex.mwmtv.nrw.de/

Darüber hinaus basiert seit dem Jahreswechsel 2000 das freie deutsche Internet Satire Magazin ZYN! (http://www.zyn.de/) auf dem Redaktionssystem ZAS (ZYN! Autoren System), welches vollständig mit CIPP realisiert wurde, auf einer MySQL Datenbank basiert und produktiv unter Einsatz von CGI::SpeedyCGI läuft.

new.spirit - webbasierte Perl Entwicklungsumgebung

Jörn Reder
joern@zyn.de
Softwareentwickler, dimedis GmbH

9. Februar 2000

Inhaltsverzeichnis

1 new.spirit - webbasierte Perl Entwicklungsumgebung

Spätestens wenn mehr als drei Entwickler gemeinsam an einem größeren Projekt arbeiten, muß man sich Gedanken über eine einheitliche Entwicklungsumgebung machen. Der konkurrierende Zugriff auf die Quelldateien muß sinnvoll kontrolliert werden, die Einbindung von Versionskontrollsystemen wie CVS ohne viel Aufwand möglich sein, Abhängigkeiten zwischen Quelldateien müssen wenn möglich automatisch verwaltet werden und vieles mehr.

Wenn es zusätzlich noch um die Entwicklung von Online Informationssystemen geht, was liegt da näher, als eine Entwicklungsumgebung auf eben dieses Format zu bringen? Der Zugriff auf Projektdaten und Quelldateien über den Browser bringt hier dieselben Vorteile, wie der Zugriff auf ein Informationssystem über das Web, z.B.: Unabhängigkeit vom Ort des Zugriffs und Verzicht auf plattformspezifische und zusätzlich zu entwickelnde Client Software (hierzu werden ja die auf allen relevanten Plattformen vorhandenen Webbrowser verwendet).

Oftmals erlauben Kunden nicht den vollen Zugriff auf die Maschine, so daß z.B. ein zur Wartung des Systems eigentlich unverzichtbarer Telnet Zugang fehlt. Wenn die Wartung der Software über das Web und somit über HTTP möglich ist, kann dies ein entscheidender Vorteil sein.

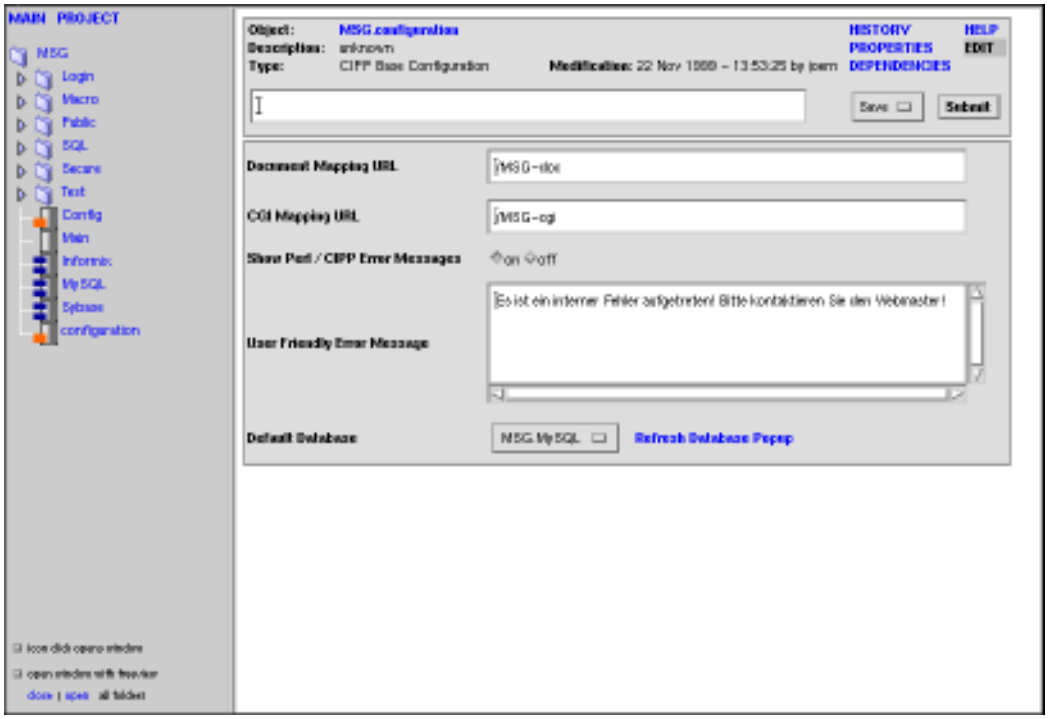
new.spirit gehört genau zu dieser Gattung von Entwicklungsumgebung. Es läßt sich vollständig über den Browser bedienen, bietet eine leistungsfähige Schnittstelle zu CVS, ist aber trotzdem so offen, daß Quelldateien auch nach wie vor mit dem Lieblingseditor des Entwicklers bearbeitet werden können.

new.spirit wurde mit dem Ziel entwickelt, alle zur Entwicklung benötigten Werkzeuge und Daten unter der Browser-Oberfläche zu vereinen. So sind z.B. auch Zugriffe auf Datenbanken direkt über den Browser möglich, so daß auf das sonst nötige Shell Fenster zum Datenbankserver verzichtet werden kann. Bei

der Verwendung gemischter Datenbankarchitekturen hat dies auch den Vorteil, daß man sich nicht mit der Bedienung und den Besonderheiten der unterschiedlichen Client Tools der Datenbankhersteller herumschlagen muß.

Dieser Vortrag soll einen praktischen Überblick über die Möglichkeiten von new.spirit vermitteln und wird deshalb in Form einer Life Performance anhand eines kleinen Beispielprojektes durchgeführt. Dieses Dokument wird lediglich einige wichtige Bestandteile von new.spirit zur Orientierung anhand von Screenshots kommentieren, um einen Eindruck von der Philosophie hinter new.spirit zu vermitteln.

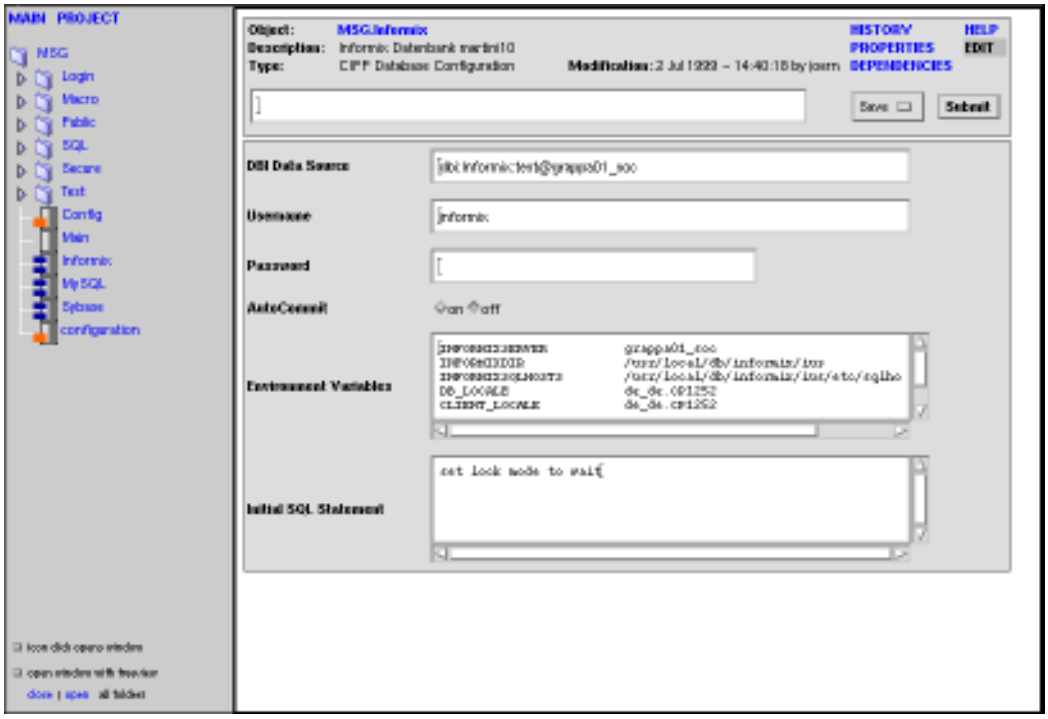
1.1 Projektbrowser und Grundeinstellungen



Der Hauptbildschirm von new.spirit. Links der Projekt Browser, der alle Quelldateien anhand von typspezifischen Icons darstellt. Rechts die Seite zum Editieren der Grundeinstellungen eines Projektes. Hier werden insbesondere Dokumenten- und CGI Mappings des Webservers eingestellt, über die die generierten Seiten und Programme abrufbar sein sollen.

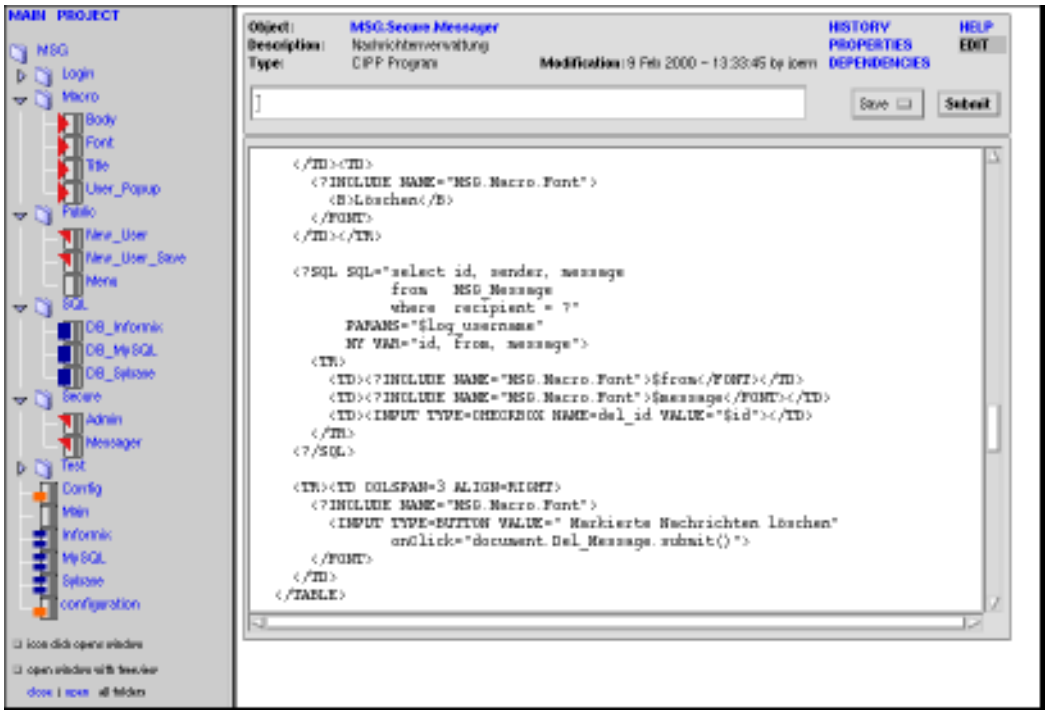
Es ist möglich mehrere Grundeinstellungs-Objekte anzulegen, und so verschiedene Produktivumgebungen (z.B. bei unterschiedlichen Kunden) zu verwalten. new.spirit enthält Werkzeuge zum automatischen Update von Produktivinstallationen, die sich dieser Grundeinstellungen bedienen.

1.2 Konfiguration für den Datenbankzugriff



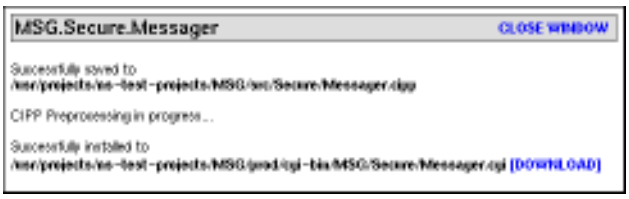
Damit die CGI Programme auf eine Datenbank zugreifen können, muß hier lediglich einmal die Konfiguration hinterlegt werden. Es ist möglich beliebig viele Datenbank Konfigurationen zu erstellen und auf entsprechend viele Datenbanken aus einem Programm heraus zuzugreifen.

1.3 Editieren einer CIPP Seite



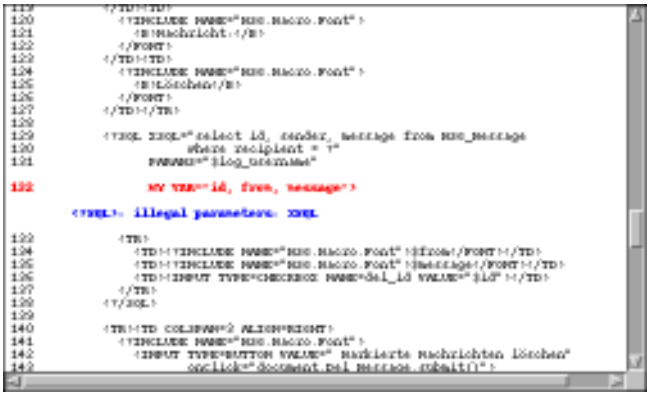
new.spirit setzt bei der Generierung von CGI Programmen auf dem Perl Modul CIPP auf, welches die Einbettung von Perl und SQL in HTML auf einfache Weise erlaubt.

1.4 Speichern Fenster



Wenn ein Quellobjekt gespeichert wird, erscheint ein neues Fenster mit den entsprechenden Ergebnissen. Beim Speichern einer CIPP Seite wird dieses sofort durch den CIPP Präprozessor zu einem CGI verarbeitet, dies wird im Speichern Fenster dokumentiert.

1.5 Fehler Report

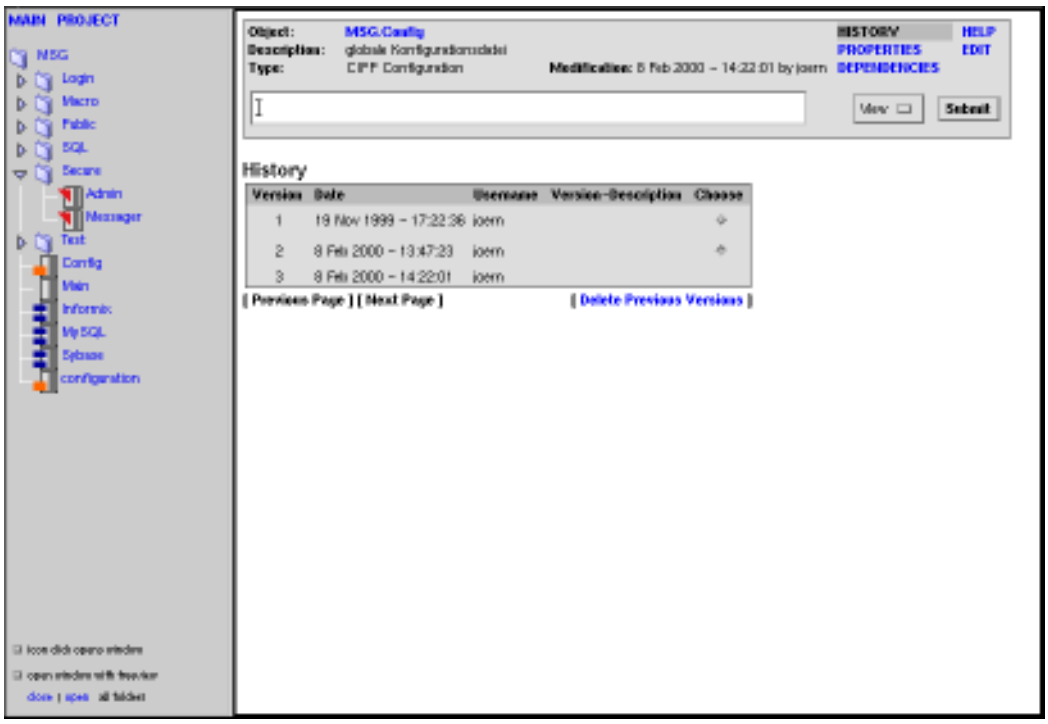


Treten beim Speichern Fehler auf (z.B. CIPP oder Perl Syntaxfehler), so werden diese anhand eines gehighlighteten Quelltextes angezeigt. Am Ende der Seite erscheint eine übersichtliche Zusammenfassung aller Fehler, die über Hyperlinks mit den entsprechenden Stellen im Code verbunden sind.

1.6 Kontrolle der Abhängigkeiten

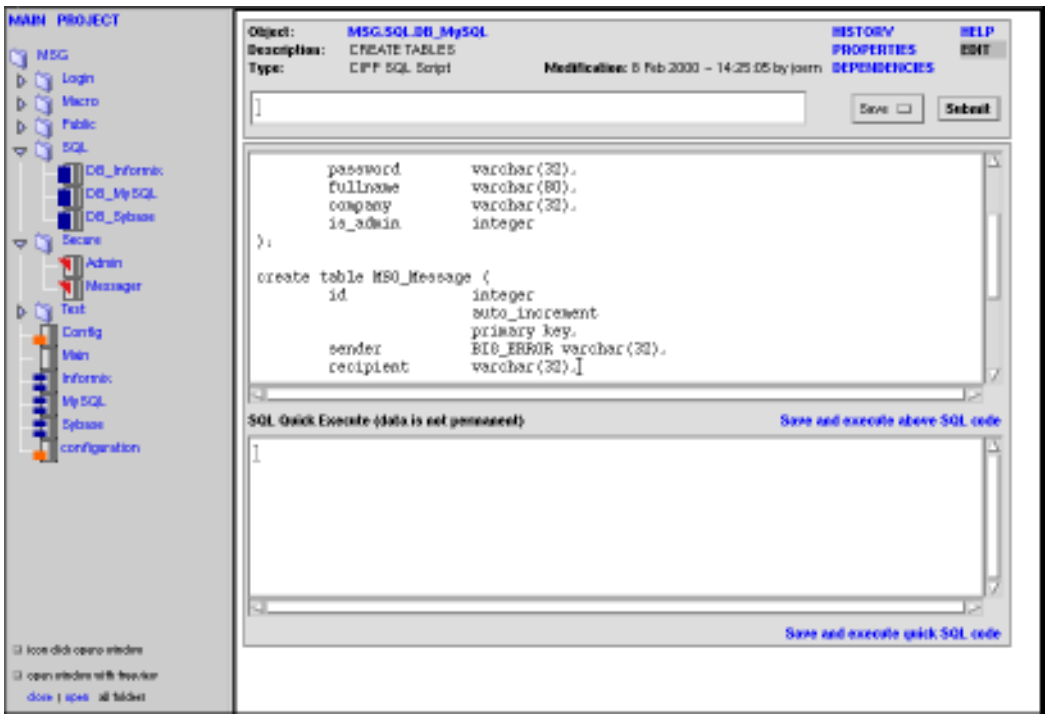
Anhand dieser Übersicht werden Abhängigkeiten zwischen den einzelnen Quellobjekten aufgezeigt. Bei der Speicherung von Objekten, deren Abhängigkeiten sich zur Compilierzeit auswirken, werden automatisch die abhängigen Objekte neu übersetzt. Die Pflege der Abhängigkeitsinformationen übernimmt new.spirit vollautomatisch.

1.7 History von Quelldatei Versionen



Wenn gewünscht, hält new.spirit alte Versionen von Quellobjekten nach. Über die History Funktion können diese beliebig nach Ansicht jederzeit wiederhergestellt werden.

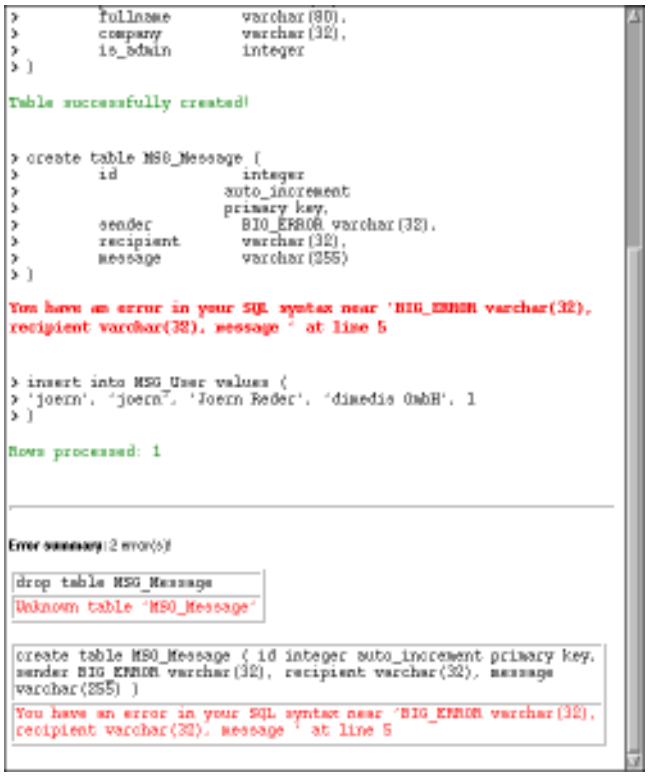
1.8 Absetzen von SQL Befehlen über den Browser



SQL Dateien sind eigenständige Objekte in new.spirit. Hiermit kann beliebiger SQL Code verwaltet und

ausgeführt werden. Die Ausführung erfolgt direkt über DBI und ist somit für alle Datenbanken verfügbar, die von DBI unterstützt werden.

1.9 Fehlerzusammenfassung der Ausführung von SQL Dateien



Beim Ausführen einer SQL Datei erscheinen die Ausgaben in einem eigenen Fenster. Dabei werden Fehler besonders hervorgehoben und am Ende der Seite nochmals zusammengefaßt. Die Meldungen sind mit den entsprechenden Fehlerstellen im SQL Code über Hyperlinks verbunden.

Apache/mod_perl/Embperl: Was gibt es Neues?

Gerald Richter
richter@ecos.de

21. Februar 2000

Inhaltsverzeichnis

1 Was ist Embperl?

1.1 Perl Code in HTML Dokumente einfügen

Die Hauptanwendung von HTML::Embperl ist Perlcode in HTML Dokumente einzufügen. Embperl kann zwar ebenfalls mit nicht HTML Dokumenten benutzt werden, hat jedoch einige Features speziell für HTML.

1.2 Zusätzliche HTML Features

Einer der Vorteile von Embperl ist, daß es speziell auf HTML zugeschnitten ist. Es stellt u.a. Funktionen zur Formularbehandlung und für HTML Tabellen zur Verfügung, einhergehend mit der Fähigkeit Logdateien und Fehlerseiten in HTML darzustellen. Ebenso erledigt es die HTML und URL Kodierung. Dies verhindet jedoch nicht das Embperl mit allen Arten von Textdateien umgehen kann.

1.3 Integration mit Apache und mod_perl

Embperl kann offline (als normales CGI Skript oder als Modul dessen Funktionen sich von anderem Perlprogrammen / -modulen aufrufen lassen) benutzt werden, aber die meisten Möglichkeiten und beste Performance entwickelt es unter mod_perl und Apache. Dort werden direkt die Funktionen der Apache API genutzt und mod_perl erlaubt es den Code vorzukompilieren, um dadurch den Compilierungsvorgang bei jedem weiterem Request einzusparen.

1.4 Embperl arbeitet mit HTML Editoren

Embperl ist entworfen worden um direkt mit dem von HTML Editoren erzeugten Code zu arbeiten. Der Perlcode wird dabei als normaler Text eingeben. Es ist nicht nötig, das der HTML Editor spezielle HTML Tags kennt, noch müssen diese über umständliche Dialoge eingegeben werden. Embperl kümmert sich darum, z.B. ein vom HTML Editor erzeugtes < in < umzuwandeln, bevor es dem Perlinterpreter übergeben wird. Außerdem entfernt es unerwünschte HTML Tags, z.B. ein
, das der Editor eingefügt hat, weil man eine neue Zeile anfängt, aus dem Perlcode.

2 Perl Code in HTML Dokumente einfügen

Perlcode kann auf drei Arten eingebettet werden:

2.1 [- . . . -] Führt den Code aus

```
[- $a = 5 -]  [- $b = 6 if ($a == 5) -]
```

Der Code zwischen [- und -] wird ausgeführt, dabei wird keine Ausgabe erzeugt. Diese Form eignet sich für Zuweisungen, Funktionsaufrufe, Datenbankanfrage, usw.

2.2 [+ . . . +] Das Ergebnis ausgeben

```
[+ $a +]  [+ $array[$b] +] [+ "A is $a" +]
```

Der Code zwischen dem [+ und dem +] wird ausgeführt und der Rückgabewert (der Wert des letzten Perlausdruckes welcher berechnet wurde) wird ausgegeben (zum Browser gesandt)

2.3 [! . . . !] Code nur einmal ausführen

```
[! sub foo { my ($a, $b) = @_ ; $a * $b + 7 } !]
```

Genauso wie [- . . . -], der Code wird jedoch nur einmal, für den ersten Request, ausgeführt. Dies ist hauptsächlich für Funktionsdefinitionen und einmalige Initialisierungen.

3 Meta-Commands

Embperl unterstützt einige Meta-Commands um dem „Programmablauf“ innerhalb des Embperldokuments zu steuern. Dies kann mit einem Preprozessor in C verglichen werden. Die Meta-Commands haben folgende Form:

```
[$ <cmd> <arg> $]
```

3.1 if, elsif, else, endif

Der if Befehl hat die selben Auswirkungen wie in Perl. Er kann genutzt werden um Teile des Dokuments nur unter bestimmten Bedingungen auszugeben/auszuführen. Beispiel:

```
  [$ if $ENV{REQUEST_METHOD} eq 'GET' $]
    <p>Dies ist ein GET Request</p>
 [$ elsif $ENV{REQUEST_METHOD} eq 'POST' $]
    <p>Dies ist ein POST Request</p>
 [$ else $]
    <p>Dies ist weder ein GET noch ein POST Request</p>
 [$ endif $]
```

Dieses Beispiel gibt eine der drei Absätze in Abhänigkeit von dem Wert von $ENV{REQUEST_METHOD} aus.

3.2 while, endwhile

Der while Befehl wird dazu benutzt, um eine Schleife innerhalb des HTML Dokuments zu erzeugen. Beispiel:

```
  [$ while ($k, $v) = each (%ENV) $]
    [+ $k +] = [+ $v +] <BR>
 [$ endwhile $]
```

Das Beispiel zeigt alle Environementvariablen, jede abgeschlossen mit einem Zeilenumbruch (
).

3.3 do, until

`do until` erzeugt ebenso eine Schleife, jedoch mit der Bedingung am Ende. Beispiel:

```
  [- @arr = (3, 5, 7); $i = 0 -]
 [$ do $]
    [+ $arr[ $i++ ] +]
 [$ until $i > $#arr $]
```

3.4 foreach, endforeach

Erzeugt eine Schleife, die über jedes Element einer Liste / Arrays iteriert. Beispiel:

```
  [$ foreach $v (1..10) $]
    [+ $v +]
 [$ endforeach $]
```

3.5 var <var1> <var2> ...

Standartmäßig ist es nicht nötig irgenwelche Variablen innerhalb einer Embperlseite zu deklarieren. Embperl kümmert sich darum nach jedem Request wieder aufzuräumen. Manchmal möchte man jedoch die zu benutzenden Variablen explizit deklarieren. Dies ist mit var möglich:

```
  [$ var $a @b %c $]
```

Hat den selben Effekt wie der Perlcode:

```
 use strict ; use vars qw {$a @b %c} ;
```

3.6 hidden

hidden ermöglicht es versteckte Formularfelder zu erzeugen und wird weiter unten im Abschnitt über Formularfelder beschrieben.

4 Dynamische Tabellen

Ein sehr leistungsfähiges Feature von Embperl ist das Erzeugen von dynamischen Tabellen. Am einfachsten lassen sich auf diesem Weg Perlarrays in Tabellen umwandeln (ein- oder zweidimensional, gleich- und ungleichmäßige), aber auch andere Datenquellen sind möglich.

4.1 Anzeigen eines Perlarrays

```
  [- @a = ( 'A', 'B', 'C') ; -]
<TABLE BORDER=1>
  <TR>
       <TD> [+ $a[$row] +] </TD>
  </TR>
</TABLE>
```

Das obige Beispiel gibt einfach eine Tabelle mit drei Zeilen, welche A, B und C enthalten aus.

Der Trick dabei ist die Benutzung der magischen Variable **$row**, welche die Zeilennummer innerhalb der Tabelle enthält und automatisch für jede Zeile um eins erhöht wird. Die Tabelle ist zu Ende, wenn der Block, in dem **$row** auftaucht, **undef** zurückgibt. Das funktioniert auch mit **$col** für Spalten und **$cnt** kann benutzt werden, wenn die Elemente, nach einer bestimmten Anzahl, in die nächste Reihe rutschen sollen.

Dies funktioniert ebenso mit `table/select/menu/ol/dl/dir`

4.2 Einfaches DBI Beispiel

Hier ist ein einfaches DBI Beispiel, welches das Ergebnis einer Anfrage in einer zwei dimensionalen Tabelle anzeigt, mit den Feldnamen als Überschrift in der ersten Zeile:

```
  [-
# Verbinden mit Datenbank
 $dbh = DBI->connect($DSN) ;

 # SQL Select vorbereiten
$sth = $dbh -> prepare ("SELECT * from $table") ;

 # Datenbankanfrage ausfuhren
$sth -> execute ;

 # $head erhalt die Feldnamen fur die Tabellenuberschrift
$head = $sth -> {NAME} ;

 # $dat erhalt die Datensatze
$dat = $sth -> fetchall_arrayref ;
-]

 <table>
   <tr><th>[+ $head->[$col] +]</th></tr>
   <tr><td>[+ $dat -> [$row][$col] +]</td></tr>
</table>
```

5 Formularfelder

5.1 Gesendete Formulardaten sind in %fdat/@ffld verfügbar

Der Hash **%fdat** enthält alle Werte der Formularfelder. Das Array **@ffld** enthält die Namen in der Reihenfolge wie sie gesendet wurden.

5.2 Input/Textarea/Select tags erhalten ihre Werte aus %fdat

Wenn innerhalb des HTML Codes kein Wert für ein Inputtag angegeben ist und Daten in **%fdat** dafür verfügbar sind, fügt Embperl automatisch den Wert aus **%fdat** ein. Dies ist ähnlich dem Verhalten von CGI.pm. Das bedeutet, daß wenn man die Daten eines Formular (in einer Embperlseite) an sich selbst schickt, automatisch die Daten wieder angezeigt werden, die gerade eingegeben wurden.

5.3 [$ hidden $]

[$ hidden $] erzeugt versteckte Formularfelder für alle Werte aus **%fdat**, die bis dahin nicht in einem anderem Formularfeld ausgegeben wurden. Dies ist hilfreich, wenn Daten über mehere Formulare hinweg transportiert werden müssen.

5.4 Ein einfaches Texteingabe/Bestätigungs Formular

Das folgende Beispiel zeigt viele der Möglichkeiten von Embperl. Es ist ein einfaches Formular, in dem man seinen Namen, seine Email Adresse, sowie eine Nachricht eingeben kann. Wenn man es absendet, werden die Daten zunächst noch einmal angezeigt. Von dort kann man zum vorherigen Formular zurückkehren, um die Daten zu korrigieren oder der Benutzer bestätigt die Daten, wodurch sie zu einer vordefinierten Email Adresse gesandt werden. Das Beispiel zeigt auch wie eine Fehlerüberprüfung implementiert werden kann. Wenn der Name oder die Email Adresse weggelassen wird, wird eine entsprechende Fehlermeldung angezeigt und das Eingabeformular erscheint wieder.

Der erste Teil ist die Fehlerüberprüfung; der zweite Teil die Bestätigungsseite; der dritte Teil versendet die Email, wenn die Eingaben bestätigt wurden und der letzte Teil ist das Eingabeformular.

In Abhängigkeit der Werte von `$fdat{check}`, `$fdat{send}` und ob `$fdat{name}` und `$fdat{email}` Daten enthalten, entscheidet das Dokument welcher Teil zur Ausführung kommt.

```
  [-  $MailTo = 'richter\@ecos.de' ;

   @errors = () ;
  if (defined($fdat{check}) || defined($fdat{send}))
    {
    push @errors, "**Bitte Namen eingeben" if (!$fdat{name}) ;
    push @errors, "**Bitte E-Mail Adresse eingeben" if (!$fdat{email}) ;
    }
 -]

  [$if (defined($fdat{check}) and $#errors == -1)$]
 [-
  delete $fdat{input} ;
  delete $fdat{check} ;
  delete $fdat{send}
 -]

  <hr><h3> Sie haben folgende Daten eingegeben:</h3>
 <table>
  <tr><td><b>Name</b></td><td>[+$fdat{name}+]</td></tr>
  <tr><td><b>E-Mail</b></td><td>[+$fdat{email}+]</td></tr>
  <tr><td><b>Nachricht</b></td><td>[+$fdat{msg}+]</td></tr>
  <tr><td align="center" colspan="2">
     <form action="input.htm" method="GET">
       <input type="submit" name="send"
              value="Send to [+ $MailTo +]">
       <input type="submit" name="input" value="Daten abandern">
       [$hidden$]
    </form>
    </td></tr>
 </table>

  [$elsif defined($fdat{send}) and $#errors == -1$]
```

```
  [- MailFormTo ($MailTo,'Formdata','email') -]
<hr><h3>Ihre Nachricht wurde abgeschickt</h3>

  [$else$]

  <hr><h3>Bitte geben Sie Ihre Daten ein</h3>

  <form action="input.htm" method="GET">
  <table>
    [$if $#errors != -1 $]
      <tr><td colspan="2">
      <table>
    <tr><td>[+$errors[$row]+]</td></tr>
      </table>
      </td></tr>
    [$endif$]
    <tr><td><b>Name</b></td> <td><input type="text"
                                        name="name"></td></tr>
    <tr><td><b>E-Mail</b></td> <td><input type="text"
                                          name="email"></td></tr>
    <tr><td><b>Nachricht</b></td> <td><input type="text"
                                           name="msg"></td></tr>
    <tr><td colspan=2><input type="submit"
                             name="check" value="Send"></td></tr>  </table>
</form>

  [$endif$]
```

6 Persistente Daten (Sessions)

(Embperl 1.2 oder neuer)

Während versteckte Felder gut innerhalb Formularen einsetzbar sind, ist es oft notwendig **Daten persistent** auf eine allgemeinere Art und Weise zu speichern. Embperl benutzt *Apache::Session* um dies durchzuführen. *Apache::Session* ermöglicht die Daten im Speicher, in einem Textfile oder in einer Datenbank abzuspeichern. Weitere Speichermöglichkeiten sind für die Zukunft zu erwarten. Man kann zwar einfach *Apache::Session* aus Embperl Seiten herausaufrufen, aber Embperl ist in der Lage dies für den Benutzer transparent durchzuführen. Es genügt einfach seine Daten in dem Hash **%udat** abzuspeichern, sobald der selbe Benutzer wieder eine Embperl Seite aufruft, stehen in `%udat` wieder die selben Daten. Dies ermöglicht auf eine einfache Art und Weise Zustandsinformationen für einen Benutzer zu speichern. In Abhängigkeit vom Ablaufzeitpunkt können so Benutzerspezifische Daten auch über einen längeren Zeitraum hinweg gespeichert werden. Ein zweiter Hash, **%mdat**, dient dazu, Daten, die zu einer bestimmten Seite gehören, zu speichern. Ein einfaches Beispiel ist z.B. ein Zähler der Anzahl der Seitenaufrufe:

```
 Die Seite wurde seit dem [+ $mdat{date} ||= localtime +]
[+ $mdat{counter}++ +] mal abgerufen
```

Das obige Beispiel zählt die Anzahl der Abrufe und zeigt die Zeit, wann die Seite zum ersten Mal aufgerufen wurde. Embperl sorgt dafür, dass die Daten nur dann wieder abgespeichert werden, wenn sie auch geändert wurden.

7 Aufteilen des Codes in mehrere Komponenten

(Embperl 1.2 oder neuer)

7.1 Funktionen

Wächst ein Programm, teilt man es in mehere Funktionen auf. Dies ist mit Embperlseiten ebenfalls möglich. Folgendes Beispiel zeigt dies an Hand von beschrifteten Texteingabefeldern:

```
  [$ sub textinput $]
    [- ($label, $name) = @_ -]
    [+ $label +]<input type=text name=[+ $name +]>
 [$ endsub $]

  <form>
    [- textinput ('Nachname', 'lname')  -]<p>
    [- textinput ('Vorname', 'fname') -]<p>
 </form>
```

Das `sub` Meta-Command kennzeichnet den Anfang der Funktion und die Parameter werden im Array `@_` übergeben. Man kann innerhalb der Funktion alles tun, was auch in einer normalen *Embperl* Seite möglich ist. Aufgerufen wird die Funktion, wie jede andere Perlfunktion auch, einfach durch Schreiben des Namens und ggf. der Parameterliste.

7.2 Execute

Wenn man an einer ganzen Website arbeitet, kommt es meistens vor, daß es Elemente gibt, die in jeder oder vielen Seiten immer wieder vorkommen. Anstatt den Quellencode nun in jede Seite zu kopieren, ist es möglich **Embperl Module** in die Seite einzufügen, so daß der Quellencode nur einmal existieren muß. So ein Modul könnte z.B. ein Kopf, ein Fuß, eine Navigationsleiste usw. sein. Es können dabei nicht nur Teile einer Seite eingefügt, sondern auch, ähnlich einem Unterprogramm, Argumente übergeben werden - z.B. um der Navigationsleiste mitzuteilen, welches Element hervorzuheben ist.

Beispiel für eine einfache Navigationsleiste

```
  [- @buttons = ('Index', 'Infos', 'Suchen') -]
 <table><tr><td>
     [$if $buttons[$col] eq $param[0]$] <bold> [$endif$]
     <a href="[+ $buttons[$col] +].html"> [+ $buttons[$col] +] </a>
     [$if $buttons[$col] eq $param[0]$] </bold> [$endif$]
 </td></tr></table>
 <hr>
```

Wenn man nun auf der Info-Seite ist, kann die Navigationsleiste wie folgt eingefügt werden:

```
  [- Execute ('navbar.html', 'Infos') -]
```

Dies fügt die Navigationsleiste, welche in der Datei navbar.html gespeichert ist, an entsprechender Stelle ein und übergibt ihr als Parameter die Zeichenkette 'Infos'. Das Navigationsleistenmodul selbst benutzt eine dynamische Tabelle um die Spalten anzuzeigen, welche den Text und einen entsprechenden Link enthalten. Die Texte werden dabei dem Array `@buttons` entnommen. Wenn der Text gleich dem übergebenen Parameter ist, wird er fett dargestellt. Weiterhin gibt es noch eine ausführliche Form des Executeaufrufes, welche es erlaubt sehr detailiert die Ausführung des Moduls zu kontrollieren.

7.3 Erstellen von Komponenten Libraries

Statt eine extra Datei für jedes bischen HTML Code zu erstellen, welches in eine andere Seite eingefügt werden soll, ist es möglich diesen in eine HTML Datei zusammenzufassen. Um dies zuerreichen muß jedes einzelne Codestück eine eigene *Embperl* Funktion sein. Mittels des `import` Parameters der `Execute`

Funktion können nun alle *Embperl* Funktionen in den Namensraum der aktuellen Seite importiert werden und fortan wie normale Perlfunktionen aufgerufen werden.

Weiterhin ist es möglich die *Embperl* Funktionen (zusammen mit normalen Perl Code) als ein Perl Modul (.pm Datei) zu installieren. Dadurch stehen sie systemweit zur Verfügung und können wie jedes andere Perl Modul mittels `use` genutzt werden.

8 EmbperlObject

`(ab Embperl 1.3)`

Einen Schritt weiter als das einfache Einbetten von anderen Dateien mittels `Execute` geht *EmbperlObject*. *EmbperlObject* ist ein *mod_perl* handler, der es erlaubt eine Website in konsistenter Weise aus einzelnen Komponenten zusammenzusetzen. Dabei definiert man ein Rahmenlayout, welches "Platzhalter" für einzelne Elemente der Site (z.B. Kopf, Fuß, Navigation etc.) enthält. Diese "Platzhalter" können nun für unterschiedliche Bereiche (Unterverzeichnisse) der Site mit verschiedenen Inhalten gefüllt werden. Definiert ein Bereich (Unterverzeichnis) keinen eigenen Inhalt, wird automatisch der Inhalt des übergeordneten Verzeichnisses eingefügt. Konkret heißt das, man identifiziert Bereiche, die auf allen / vielen Seiten gleich aussehen sollen, macht daraus eine eigenständige Komponente (HTML Datei) und fügt diese dann nur noch an passender Stelle ein. Es leuchtet ein, dass dies das Design und Änderungen wesentlich vereinfacht, da eine Änderung in der Komponente sich auf alle Seiten auswirkt. Hier ein einfaches Beispiel, um zu verdeutlichen wie EmbperlObject arbeitet; dabei definert base.htm das Rahmenlayout, head.htm enthält den Kopf und foot.htm den Fuß für die Seite: **Anordnung der Dateien:**

```
 /foo/base.htm
/foo/head.htm
/foo/foot.htm
/foo/page1.htm
/foo/sub/head.htm
/foo/sub/page2.htm
```

/foo/base.htm:

```
 <html>
<head>
<title>Beispiel</title>
</head>
<body>
[- Execute ('head.htm') -]
[- Execute ('*') -]
[- Execute ('foot.htm') -]
</body>
</html>
```

/foo/head.htm:

```
<h1>Kopf aus foo</h1>
```

/foo/sub/head.htm:

```
<h1>Hier ein anderer Kopf aus dem Verzeichnis sub</h1>
```

/foo/foot.htm:

```
<hr> Fuszeile <hr>
```

/foo/page1.htm:

```
Hier steht der Inhalt von Seite 1
```

/foo/sub/page2.htm:

```
Hier steht der Inhalt von Seite 2
```

/foo/sub/index.htm:

```
Index im Verzeichnis /foo/sub
```

Der Request **http://host/foo/page1.htm** führt dann zu folgender Seite:

```
<html>
<head>
<title>Beispiel</title>
</head>
<body>
<h1>Kopf aus foo</h1>
Hier steht der Inhalt von Seite 1
<hr> Fuszeile <hr>
</body>
</html>
```

Der Request **http://host/foo/sub/page2.htm** führt dann zu folgender Seite:

```
<html>
<head>
<title>Beispiel</title>
</head>
<body>
<h1>Hier ein anderer Kopf aus dem Verzeichnis sub</h1>
Hier steht der Inhalt von Seite 2
<hr> Fuszeile <hr>
</body>
</html>
```

Der Request **http://host/foo/sub/** führt dann zu folgender Seite:

```
<html>
<head>
<title>Beispiel</title>
</head>
<body>
<h1>Hier ein anderer Kopf aus dem Verzeichnis sub</h1>
Index im Verzeichnis /foo/sub
<hr> Fuszeile <hr>
</body>
</html>
```

9 Debugging

9.1 Embperl Logdatei

Das Logfile ist die Hauptinformationsquelle zum Debuggen. Es zeichnet auf, was mit der Seite geschieht, während sie von Embperl bearbeitet wird. In Abhängigkeit von den Debugflags, logged Embperl folgende Dinge:

- **Quellencode**
- **Umgebungsvariablen**
- **Formular daten**
- **Perlcode (Quelle + Ergebnis)**
- **Tabellenbearbeitung**
- **Eingabe-Tag-Bearbeitung**
- **HTTP headers**

9.2 Embperl Logdatei kann direkt im Browser angezeigt werden

Zur Fehlersuche kann Embperl veranlasst werden, an jedem Seitenanfang einen Link zur Logdatei anzuzeigen. Wenn man dem Link folgt, wird der Teil der Logdatei, welcher zu dem entsprechenden Request gehört angezeigt. Dabei werden unterschiedliche Einträge zur leichteren Orientierung verschiedenfarbig dargestellt.

9.3 Embperl Fehlerseite enthält Links zum Logfile

Wenn die Links zur Logdatei freigeschaltet sind, werden auch in jeder Fehlerseite die Fehler direkt als Link dargestellt, die direkt auf die richtige Position im Logfile verweisen. So läßt sich einfach feststellen, was an dieser Stelle schief gelaufen ist.

10 Datenbankzugriff

10.1 DBI

Dies ist ein weiteres Beispiel für den Datenbankzugriff mittels DBI. Im Gegensatz zum vorhergehenden Beispiel arbeitet es aber mit expliziten Schleifen.

```
  [-
# Mit der Datenbank verbinden
$dbh = DBI->connect($DSN) ;
# Vorbereiten des SQL Select
$sth = $dbh -> prepare ("SELECT * from $table") ;

 # Abfrage ausfuhren
$sth -> execute ;

 # Ermitteln der Feldnamen fur die Uberschrift in $head
$head = $sth -> {NAME} ;
-]
```

```
  <table>
    <tr>
    [$ foreach $h @$head $]
        <th>[+ $h +]</th>
    [$ endforeach $]
    </tr>
    [$ while $dat = $sth -> fetchrow_arrayref $]
        <tr>
            [$ foreach $v @$dat $]
                <td>[+ $v +]</td>
            [$ endforeach $]
        </tr>
    [$ endwhile $]
 </table>
```

10.2 DBIx::Recordset

DBIx::Recordset ist ein Modul welches den Datenbankzugriff vereinfachen soll. Eine weiterführende Einführung zu DBIx::Recordset und Embperl findet sich in der iX 9/1999 unter http://www.heise.de/ix/artikel/1999/09/137/ .

10.3 Datenbankabfrage Beispiel

```
  [-*set = DBIx::Recordset -> Search ({%fdat,
                                      ('!DataSource'   => $DSN,
                                       '!Table' => $table,
                                       '$max'   => 5,)}) ; -]
 <table>
  <tr><th>ID</th><th>NAME</th></tr>
  <tr>
    <td>[+ $set[$row]{id} +]</td>
    <td>[+ $set[$row]{name} +]</td>
  </tr>
 </table>
 [+ $set -> PrevNextForm ('Previous Records',
                          'Next Records',
                          \%fdat) +]
```

10.4 Search erzeugt ein Recordsetobjekt

Search nimmt die Werte aus `%fdat` und benutzt diese um einen SQL WHERE Ausdruck zu erzeugen. Auf diese Weise hängt es davon ab, was an das Dokument für Daten gesandt werden, welche Anfrage ausgeführt wird. z.B. wenn man das Dokument mit http://host/mydoc.html?id=5 aufruft, werden alle Datensätze deren Feld id den Wert 5 enthält angezeigt.

10.5 Die Daten können als Array oder mittels eines aktuellen Datensatzzeigers angesprochen werden

Das Ergebnis der Abfrage kann wie ein Array angesprochen werden (was nicht heißt, daß das ganze Array auch tatsächlich von der Datenbank angefordert wird). Alternativ können die Felder des aktuellen Record angesprochen werden.

```
    set[5]{id}   Zugriff auf das Feld 'id' des sechsten gefundenen Datensatzes
   set{id}      Zugriff auf das Feld 'id' des aktuellen Datensatzes
```

10.6 Felder können mit ihren Namen angesprochen werden

Während bei DBI Feldinhalte hauptsächlich über ihre Spaltennummern angesprochen werden, benutzt DBIx::Recordset Spaltennamen. Dies macht das Programm einfacher zu schreiben, leichter verständlich und unabhäniger von Veränderungen in der Datenbankstruktur.

10.7 PrevNextForm erzeugt keinen/einen/zwei Schaltflächen je nachdem ob weitere Datensätze angezeigt werden müssen

Die PrevNextButtons Funktion kann dazu benutzt werden um Schaltflächen zum Anzeigen der vorhergehenden bzw. folgenden Datensätze zu erzeugen. `PrevNextForm` generiert ein kleines Formular welches alle nötigen Daten als versteckte Felder enthält.

10.8 Wie fürs Suchen, gibt es auch Funktionen für Insert/Update/Delete

Beispiel für Insert

Wenn `%fdat` die Daten für einen neuen Datensatz enthält, fügt der folgende Code einen diesen der angegebenen Tabelle hinzu.

```
[-*set = DBIx::Recordset -> Insert ({%fdat,
                                     ('!DataSource'   => $DSN,
                                      '!Table' => $table)}) ; -]
```

10.9 Datenbanktabellen können ebenso an einen Hash gebunden werden

DBIx::Recordset kann ebenfalls eine Datenbanktabelle an einen Hash binden. Man muß lediglich den Primärschlüssel der Tabelle angeben und kann dann auf die Tabelle mittels eines Perl Hashs zugreifen.

```
$set{5}{name}    Zugriff auf Feld 'name' mit id=5
                (id ist Primarschlussel)
```

10.10 Arbeiten mit mehreren Tabellen

DBIx::Recordset bietet zahlreiche Möglichkeiten um einfach mit mehreren Tabellen umgehen zu können. DBIx::Recordset versucht auf Grund der Namen innerhalb der Datenbank selbstständig Zusammenhänge zwischen Tabellen zu erkennen. Weitere Zusammenhänge können manuell angegeben werden. Mit diesen Informationen kann DBIx::Recordset automatisch Unterobjekte erzeugen, die die zum entsprechenden Datensatz zugehörigen Datensätze der verbundenen Tabelle enthalten. Ebenso ist es möglich das DBIx::Recordset einer Abfrage automatisch Felder hinzufügt, die den referenzierten Datensatz beschreiben. So ist es z.B. möglich, wenn in einer Tabelle die Kundennr enthalten ist, aus dem Kundenstammsatz automatisch den Namen des Kunden hinzuzufügen, ohne das diese jedesmal explizit angeben werden müßte.

10.11 Safe namespaces

Deshalb kann Embperl Safe.pm nutzen, um den Zugriff auf alle Namensräume außerhalb des eigentlichen Skripts zu unterbinden. Dadurch wird es z.B. möglich, Berechnungen innerhalb eines Perlmoduls durchzuführen und die Ergebnisse an ein Embperl Dokument zu übergeben. Wenn dieses in einem sicheren Namensraum läuft, kann es diese Ergebnisse darstellen, jedoch auf keine anderen Daten zugreifen. Dadurch wird es sicher, verschiedene Personen am Layout arbeiteten zu lassen.

10.12 Operatoren Einschränkungen

Safe.pm erlaubt es dem Administrator jeden Perl Opcode zu sperren. Dadurch wird es möglich zu kontrollieren, welche Perl Opcodes innerhalb der Seiten genutzt werden dürfen.

11 Escaping/Unescaping

11.1 Quellendaten: Unescaping

(sperren mit optRawInput)

```
- konvertiert HTML escapes zu Zeichen (z.B. &lt; zu <)

- entfernt HTML tags aus dem Perlcode (z.B. <br> welches durch einen
 HTML Editor eingefugt wurde)
```

11.2 Ausgabe: Escaping

(sperren mit escmode)

```
- konvertiert Sonderzeichen nach HTML (z.B. < zu &lt;)
```

Emacs & Vim: some tips for editing perl scripts

Stephen Riehm
http://www.bigfoot.com/~stephen.riehm/

Norbert Grüner
http://www.MPA-Garching.MPG.DE/~nog/

February 22, 2000

Contents

1 Editor tips for Perl Programmers (Emacs & Vim)

- A developer spends most of his time working in a text editor
- Editors are almost never properly taught in Universities etc.
- Most developers don't know how to get the most out of their editor
- This presentation will hopefully help fill this educational hole, with some tips and tricks for:

  vim: 

  [1] as well known as vi is, it is still largely misunderstood and its powers underestimated. **vim** has taken the basic vi concepts and extended them remarkably.

  emacs: 

  [2]also well known. **emacs** has the unfortunate problem of providing too much. Most users don't take the time to find the things they need, here we will provide some insights which might be useful to perl programmers.

- There are many other editors, with their own strengths and weaknesses, however they are not the focus of this presentation. We are not trying to sell either vim or emacs!

  Please don't start any flame wars over this!

2 Vi (Vim) as a perl editor

- fast
- powerful
- most people can't use it properly :-(

3 vi: Moded editing

Vi has several modes, namely:

input mode In this mode, you can insert text into your document. Most *WYSIWYG* word processors etc. only have this mode - how boring :-)

command mode This is the mode used for moving around in the text, and making changes to the text. (like deleting, or replacing the text with new text)

[1] See URL http://www.vim.org
[2] See URL http://www.emacs.org

ex mode This mode is similar to command mode, however, the commands *can* effect the whole file, and not just the part of the file where the cursor is.

visual mode *(vim only)*

This mode is similar to highlighting the text with a mouse, and then performing some operation on the highlighted text. Of course, you don't really have to use the mouse.

Some people have conceptual problems with using a moded editor like vi. Having many modes means that typing a key or some text can have completely different effects, depending on the mode you are currently in. For some this appears to be inconsistent, confusing or awkward. For others its a great way to increase the amount of functionality you can get for each key on the keyboard.

Personally, I love using a moded editor! - *Steve*

4 vi: Some tips about insert mode

- Use the Escape key to get into command mode. (From there you can quit, save your file etc)
- Don't use the cursor keys.

  Newer versions of vi and most of the clones now handle cursor keys quite well now, however, they have traditionally caused a lot of headaches.

  Besides, you are in *insert* mode, do you really want to *insert cursor keys?*
- **imap**s and **abbrev**iations can be quite helpful.

  abbreviations **abbreviation** only exchange the text you type, with some other text. Also, the abbreviation must be typed as a single word (ie: preceded *and* followed by either white-space or punctuation)

  If you have an abbreviation called `me`, then it would be expanded to `Stephen Riehm` if you typed *please send me an email*, but it wouldn't if you typed: *don't be mean*.

  imaps (insert mappings) will be expanded as soon as they have been typed. You can also do much more with an imap, as you can map any commands you like (rather than just specifying some simple text to expand)

5 vi: Command mode tips

- A typical vi user will spend *most* of their time in command mode when editing files.
- commands have a very consistent structure
- mappings, as with imaps, can be used to perform all sorts of useful tasks.

6 vi: Command mode structure

Most commands from command mode have the following structure:

```
<multiplier> <action> <multiplier> <motion>
```

multiplier The *multiplier* is a simple integer. The command you type will be performed this many times. (by default a command is only run once)

action The action is the thing that vi should do, ie: **d** is for *delete*, **c** is for *change*, > is for *indenting* and so on.

motion Is a typical motion command for vi. i.e.: **w** to move to the next *word*, **}** moves to the *next blank line*, **%** moves to the *matching bracket*(the cursor must be near a *bracket*, *brace* or *curly bracket* to do this).

There is also a special case where the command should effect the entire current line. For this cases, the command is simply doubled (ie: the command letter is typed twice, and there is no *motion*)

7 vi: Typical motions

The following is a short list of the most useful motion commands. (In the authors humble opinion) - for a full list, please check the ≪≫.

/*pattern* Search for a pattern. This is one of the quickest ways to get around a file. **?** searches in the reverse direction (upwards).

Tip: I find that using *case-insensitive* searches is much better - partly because I am lazy, and partly because I never know if what I'm looking for has been mis-spelled. Vim also has a *smartcase* option, which will automatically do a case-sensitive search if the pattern contains upper case letters.

Tip: you might also like to set the *wrapscan* option. This means that if the pattern you are looking for doesn't appear before the end of the file, then the search will continue at the top of the file. Don't worry, vi is clever enough not to search past the current location of the cursor again.

} The curly brackets can be used to jump to the next blank line in the file. **{** will jump to the previous blank line.

If the text is well structured, with blank lines separating logic blocks of text (functions, paragraphs etc) then this is a great way to *page* around a bit.

% The percent key tries to find a matching bracket for the bracket currently under (or to the right of) the cursor. If you are wondering if your brackets are matched, this is the way to check it!

f and t these two commands search along the current line for the next occurrence of the character typed directly after. The **f** command places the cursor *on* the character found, whereas the **t** command places the cursor one place to the left of the character. (ie: "**f**ind" and "up **t**o")

The uppercase letters **F** and **T** do the same as their lowercase counterparts, but in the other direction.

Note: These commands only search the current line! For bigger searches, use the **/** command.

w The **w** command moves the cursor to the first character of the next word. This command will jump to the next line if needed. The uppercase **W** does the same, however, the next word is defined to be the text following the next white space (new line, space or tab). The lowercase **w** also stops at any kind of punctuation.

To jump backwards to previous words, use **b** and **B**.

Tip: The **e** and **E** commands also move the cursor a word at a time, however, they stop at the end of the current word!

and * *(vim only)*

These commands use the word currently under the cursor as a search pattern. This is very effective when looking for variables or subroutines.

Ctrl-] This command uses the word currently under the cursor as the name of a *tag*, and then moves the cursor to the destination of that tag.

Tags are fantastic for writing programs!

You'll see more about tags later on.

Note: This command is unfortunately almost impossible to type on a German keyboard.

0, ^ and $** **0** moves the cursor to the start of the line. **^** moves the cursor to the first non-whitespace character on the line, and **$ jumps to the end of the line.

h j k l Anyone who has learned vi will have learned these keys. They are useful for moving the last few positions if you didn't quite hit your goal, however, they are useless if you are editing a 5,000 line file. What do they do? The move the cursor one position **left**, **down**, **up** and **right** respectively.

8 vi: Typical actions

d delete text. The editor is left in *command* mode.

c change text. The text is deleted, and the editor is left in *insert* mode.

y yank text into a cut buffer. This is the *copy* in *copy and paste*. The **p** command is the *paste* - however the p command isn't an *action* which you would be likely to use with a motion.

> This command indents the text. The amount of indenting is determined by the *shiftwidth* option. The **>** command will mix tabs and spaces as needed. The **<** command *outdents* the text by the same amount.

~ *(tilde)* toggle *case* of the text. ie: *UPPERCASE* letters become *lowercase*, and vise-versa.

9 vi: Some examples

So - this is where vi starts becoming *fun*. You can combine any of the actions and motions from the previous slides, to perform all sorts of wonderful things. Here are some combinations you might like to try:

dw delete up to the next word

5dw delete the next 5 words

cw change the contents of the current word. (This is like highlighting a word in a word processor and then typing a new word to take its place)

Note: this command was used *wrongly* so often that it has been hacked to work like the command **ce** - which preserves the space before and after the current word! Theory would predict that cw would also *change* the whitespace between the current word and the next word - it doesn't!

>% indent the current block of code. The cursor must be on either the opening or closing bracket of the block.

ct" change the text between the cursor and the next `"` on the current line. Other variations which are quite useful include: **ct)** or **ct]**

3df) delete everything up to the third closing brace on the current line. This can be useful for cleaning up complex `if` statements.

y3/fred yank everything from the current cursor position to the third *fred*. The text matched by the pattern is *NOT* included! This is important when deleting!

10 Emacs as a perl editor

- GNU Emacs from FSF (Richard M. Stallman)
- Practical experience with versions **19.29, 19.34, 20.4,** and **20.5**

11 Emacs: General Features

- Graphical Interface
- Context sensitive editing
- Syntax highlighting
- Extensive online help

For further information, see the extensive online help ;-)

12 Emacs: Major Modes

- Context sensitive editing via so-called **Major Modes**
- Emacs provides a **Major Mode** for just about every programming language
- Each window can be in exactly one Major Mode
- Each Major Mode has its own individual keyboard mapping
- Changing the current Major Mode also changes the current keyboard mapping
- Certain keys will perform special functions for the selected language
- The keys that are changed frequently are `TAB`, `DEL` and `LFD`

13 Emacs: available Perl Modes

- Perl-Mode, standard perl mode distributed with emacs

```
perl-mode.el --- Perl editing commands for GNU Emacs
Author: William F. Mann
Current Version: unknown
```

- CPerl-Mode, available since 1991, distributed for the first time with emacs 20.5

```
perl-mode.el --- Perl code editing commands for Emacs
Author: Ilya Zakharevich and Bob Olson
Current Version: CPerl Version 4.24 from August 1999
```

14 vi / emacs: Tags

- tags are like hypertext for programs

vi – use **ptags** to generate tags files for perl scripts
– use `Ctrl-]` or `:ta <tagname>` to resolve a tag

emacs – use **etags** to generate tags files
– use `Meta-.` to resolve a tag

15 vim: Some specialities in comparison to vi

Vim has improved on the original vi by adding an incredible amount of functionality, some of which includes:

- multiple undoes
- online help
- syntax highlighting
- automatic indenting (for formatting C code (works great for **perl** too!))
- text highlighting (visual mode)
- a GUI
- functions (in addition to mappings and abbreviations)
- the ability to remember actions between sessions

For more information, please see the vim homepage[3].

16 vim: Bracketing macros - a further extension

OK - how about something a little closer to perl?

For the past 10 years I have been using a set of macros which helps me keep track of bracketing in all of my text files.

I have a complete description[4] available on the web, where you can also download the latest version of the macros.

The advantages of the macros are:

- opening and closing brackets and quotes *always* match up
- wrapping text in brackets is greatly simplified
- you can create *mini forms* and then simply fill in the gaps
- you don't have to do *any* indenting at all!
- You can prepare templates which you simply fill out like a form
- they make use of quite a few vim features, however, they can be modified to work in plain vi

17 vim: Bracketing - the first steps

Preposition Whenever you type an opening bracket or quote, you are very probably going to want a closing bracket or quote to match.

This is exactly what my macros do!

- The **Meta** key is used to make remembering the macros easy
- Plain-text input is not effected (important for *cut & paste*)
- The macros can be used in insert mode and in visual mode
- Special markers are used to save you having to *find you way out* again.

[3] See URL http://www.vim.org
[4] See URL http://www.bigfoot.com/~stephen.riehm/vim

18 Emacs: CPerl Features

- The various brackets are almost always automatically paired. ie: {}, (), [] and sometimes . When you open a bracket, you automatically get the closing bracket as well.
- Provides expansion of the Perl control constructs:

```
if, else, elsif, unless, while, until, continue, do,
for, foreach, formy and foreachmy.
```

- Provides expansion of POD directives
- Has a builtin list of one-line explanations for perl constructs
- Lineup vertically “middles” of rows, like ‘=’ or ‘=>’
- Can run program, check syntax, start debugger
- Can insert spaces where this improves readability
- Perl statements are automatically indented
- Can switch to different indentation styles by one command
- Has support for **imenu**, including:
  - Separate unordered list of “interesting places”;
  - Separate TOC of POD sections;
  - Separate list of packages;
  - Hierarchical view of methods in (sub)packages;
  - and functions (by the full name - with package);
- Has 6 different ways to generate TAGS

19 Bracketing - learning by doing

Here is an overview of the macros provided:

Example	Description	vim Macro	emacs (CPerl)
(the marker is replaced by the cursor)	Jump to next jump point marker	**DEL**	**C-j**
«»	Insert a new jump point manually	**M-DEL**	
'text'	Single quotes	**M-'**	
"text"	Double quotes	**M-"**	
`text`	Back-quotes	**M-`**	
(text**)**	Braces, no padding	**M-(**	**(**
(text**)**	Braces, with padding	**M-)**	
[text]	Brackets, no padding	**M-**[	[
[text]	Brackets, with padding	**M-**]	
{text**}**	Curlies, no padding	**M-{**	**{**
{ text }	A new block, formatted *correctly*	**M-}**	
<text>	Angle brackets, no padding	**M-**<	<
< text >	Angle brackets, with padding	**M-**>	
(text**);**	shortcut M-) with trailing ;	**M-;**	
(text1) { text2 }	short cut for **M-)M-}**	**M-**\	*perl command*

20 Before and After

The macros can be used while writing new text, or when changing the text again afterwards.

While generating text:

- A jump marker is left outside the brackets to easy navigation when you want to continue past the brackets
- the **Delete** key is used to jump to the next marker (it wasn't being used for anything else useful :-)

When editing text:

- use vim's *visual mode* to highlight the text you wish to wrap in brackets.
- use the macro you would have used in insert mode
- the brackets will be added to the text, but no jump marker will be set.

  PS: There are exceptions to this, like **M-**\ which creates a new block which you can then turn into an `if` or `while` block.

21 Using the bracketing macros to create templates

As a special side effect of the jump markers, you can insert a text label between them, which will be displayed on the command line when you jump to that marker. This way, you can create pre-fabricated templates, with hints as to what should be filled in where.

Here is an example template:

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

```
#
#       --  description
#
###############################################################################
#  Usage:
#
#        M-fM-s  [-d]  options
#
#  Arguments:
#       -d      activates debugging
#
#
###############################################################################
#  Description:
#
#   M-fM-s
#        full description
#
###############################################################################
#  Administration:
#
#    Author:            Stephen Riehm, PC-Plus Computing, Germany
#    Maintainer:        Author
#    Creation date:      M-dM-e
#    Version date:      %E% %U%
#
###############################################################################
#  "@(#) %W%, %I%"

use strict;

( $program = $0 ) =~ s,.*/,,;

sub usage
{
    print <<_EOUSAGE;
Usage: $program
_EOUSAGE
    exit( 1 );
}

 you write your bit here
```

22 Making mini-forms

- When editing text which is very repetitive, try typing the parts which don't change, and insert jump points for the parts which do change.

Example:

```
if defined( $var1 )          then print "var1 is set to $var1\n";
if defined( $var2 )          then print "var2 is set to $var2\n";
```

you'll be mad to type this line for line if you were going to query 20 variables (probably doing something different for each one).

Create a mini-form like this:

```
if defined( ${  } ) then print "   is set to ${  }\n";
```

Yank it into the cut buffer, and paste it 20 times (`yy20p` for vi users), now all you have to do is fill in this missing bits, using `DEL` to jump between the jump points.

```
if defined( ${var1} )       then print "var1 is set to ${var1}\n";
if defined( ${options} )    then print "options is set to ${options}\n";
if defined( ${  } ) then print "   is set to ${  }\n";
if defined( ${  } ) then print "   is set to ${  }\n";
if defined( ${  } ) then print "   is set to ${  }\n";
if defined( ${  } ) then print "   is set to ${  }\n";
if defined( ${  } ) then print "   is set to ${  }\n";
if defined( ${  } ) then print "   is set to ${  }\n";
if defined( ${  } ) then print "   is set to ${  }\n";
if defined( ${  } ) then print "   is set to ${  }\n";
if defined( ${  } ) then print "   is set to ${  }\n";
if defined( ${  } ) then print "   is set to ${  }\n";
if defined( ${  } ) then print "   is set to ${  }\n";
if defined( ${  } ) then print "   is set to ${  }\n";
if defined( ${  } ) then print "   is set to ${  }\n";
if defined( ${  } ) then print "   is set to ${  }\n";
if defined( ${  } ) then print "   is set to ${  }\n";
if defined( ${  } ) then print "   is set to ${  }\n";
if defined( ${  } ) then print "   is set to ${  }\n";
```

23 References

vim: 

[5]

- **Homepage:** http://www.vim.org
- **Sources:** http://www.vim.org/download.html
- **Bracketing macros:** http://www.bigfoot.com/~stephen.riehm/vim

emacs: 

[6]

- **Homepage:** http://www.emacs.org
- **Sources:** ftp://ftp.gnu.org/pub/gnu/emacs/emacs-20.5a.tar.gz or ftp://ftp.gnu.org
- **Cperl-mode:** ftp://ftp.math.ohio-state.edu/pub/users/ilya/emacs and ftp://ftp.math.ohio-state.edu/pub/users/ilya/perl

[5] See URL http://www.vim.org
[6] See URL http://www.emacs.org

Sumdiary: evaluating a personal diary with fuzzy features

Stephen Riehm
http://www.bigfoot.com/~stephen.riehm

February 22, 2000

Contents

1 Sumdiary - Time Tracking

- Admin wants to know how long we work, and what we do.
- We want to know what we've done, and perhaps learn from our own mistakes :-)
- Why do everything twice?

2 What options are open?

Xtimex: A simple, very easy to use time tracking program.

A typical Xtimex window

Excel: The time sheets used at varetis[1] are nothing more than M$ Excel sheets. Many people open the sheet and type in the times as they remember them, each day. This method only serves the administrators, the developers etc. have no benefits from this information.

Other software: There are some other time tracking systems around, but none have interfaces which make it possible to export the time information to the Excel sheets.

A simple diary: I decided to keep a simple, *almost* free text diary, originally as a part of the PSP. It was then just a matter of writing a script to extract the desired information from the diary file. Thus came sumdiary into being.

3 An example of the diary file

a snapshot of my diary file

- colourisation is available for vim and emacs
- a couple of very simple macros have been provided for adding new timestamps at the bottom of the file.

4 Advantages of using a diary

- The diary contains all sorts of information, and quickly becomes a free-text database of names, commands, activities, things to do, problems and solutions.
- Tracking the time spent on tasks becomes implicit in the daily workflow.
- Diary can be kept on any machine (Unix, Windows)
- Simple scripts can be used to extract all sorts of information.
  - Generate a to-do list
  - Summarise time information for any month, any year
  - Create a list of customers, contact names etc
  - Generate a tags file to simplify the finding of information

5 The macros available

new entry
- vim: **M-n**
- emacs: **Keypad-1**

entry details
- vim: `Del`
- emacs: **Keypad-2**

logout
- vim: **M-l**
- emacs: **F11**

[1] See URL http://www.varetis.de

6 The sumdiary script

- Generates time summaries for any month (by default, the current month - when exporting, the last month)
- Exports the summary in a form which can be imported directly into the Excel sheet.
- An example summary (for an incomplete month)

```
summarising February 2000
Setting today's logout time to 2000-02-09 16:48

                          Total| Mon    Tue    Wed    Thu    Fri    Sat    Sun
                 2000 CW 5     |31.01.  1.02.  2.02.  3.02.  4.02.  5.02.  6.02.
                          -----+------ ------ ------ ------ ------ ------ -
-----
                        Org  9:43|         1:04   7:12   1:06   0:21
         PerlWS Preparation 11:11|                2:31   1:04   7:36
  PerlWorkshop Preparation  7:02|         7:02
                    Support  0:10|                0:10
                      Tools 10:00|         4:14          5:46
                          -----+------ ------ ------ ------ ------ ------ -
-----
                       Start     |        11:13  11:19  11:12  11:07
                       Pause     |         1:58   2:07   0:30   0:24
                        Stop     |        25:31  23:19         19:28
               Total Worked 18:28|        12:20   9:53 -11:18   7:57

                          Total| Mon    Tue    Wed    Thu    Fri    Sat    Sun
                 2000 CW 6     | 7.02.  8.02.  9.02. 10.02. 11.02. 12.02. 13.02.
                          -----+------ ------ ------ ------ ------ ------ -
-----
          CMVC - Nachfolger  0:35|         0:35
                        Cmvc  1:13|                1:13
                         Org  6:01|  4:06   1:45   0:10
         PerlWS Preparation  3:27|  0:53   2:05   0:29
             Self Education  0:30|  0:30
                     Support  0:23|         0:08   0:15
          Support - ZB / PM  0:32|         0:32
                       Tools  4:48|  1:22   0:28   2:58
                         Web  0:33|         0:33
                          -----+------ ------ ------ ------ ------ ------ -
-----
                       Start     | 11:12  11:00  11:09
                       Pause     |  1:02   1:08   0:34
                        Stop     | 19:05  18:14  16:48
               Total Worked 18:02|  6:51   6:06   5:05

Overall Summary:
================
                        Task  Total  %'ge
                        ---- ------ -------
                         Org  15:44  28.03%
                       Tools  14:48  26.37%
```

```
       PerlWS Preparation  14:38  26.07%
 PerlWorkshop Preparation   7:02  12.53%
                    Cmvc   1:13   2.17%
       CMVC - Nachfolger   0:35   1.04%
                     Web   0:33   0.98%
                 Support   0:33   0.98%
       Support - ZB / PM   0:32   0.95%
          Self Education   0:30   0.89%
                   ----- ------
                   Total  56:08

Error Messages and Warnings:
          You forgot to logout on 2000-02-03 (Line 15067).
```

7 Some interesting dark corners from sumdiary

- Fuzzy pattern recognition (we're parsing free text after all)
- Time calculations
- Duplicate reduction

Note: sumdiary was written before the Date:: modules became available - the only modules used by sumdiary are:

- **FindBin**
- **POSIX**
- **Getopt::Long**
- **Term::Query**
- **Data::Dumper** (only used for debugging)

8 Fuzzy pattern recognition

Problem: The official project names are too much to be typed every time you change tasks. The user is free to define their own labels for the tasks they perform. i.e.:

```
Org
Tools
PerlWS Preparation
PerlWorkshop Preparation
```

The official names for these labels are:

```
(p) allg. Org., Meetings
(p) Tools, Installation und Wartung
(p) Schulungsvorbereitung
```

In the example above, **PerlWorkshop Preparation** and **PerlWS Preparation** are both booked as **(p) Schulungsvorbereitung**

9 How does sumdiary do that?

The `guess()` routine uses the label from the diary to search the official list of known projects. The following steps are taken:

1. check for a match in the preferences file (read during initialisation)
2. prepare the labels from the diary for use as a regular expression
   - escape special characters like `.`, `+`, `?` and `*`)
   - separate the words of the search pattern with `|`'s - this way, each word can match with the words in the official labels. (The words are bound to the start of the word, so: **vorb** will match **vorbereitung**, but not **Schulungsvorbereitung**)
3. see if the entire string matches exactly
4. see if the entire string matches with several possibilities, if so, sort based on the number of words which matched (the most likely guess moves to the top of the list)
5. prompt the user with the list of probably matches
6. store the user's response in a preference file for next time. (a simple hash is used to convert the user's labels into the official ones)

10 Code snippits from the `guess()` routine

- routine to escape special characters

```
sub string2re
    {
    my @re_list = @_;
    grep( s/[\/\?\+\*\[\]\$\(\)\|]/\\$&/g, @re_list );
    return wantarray ? @re_list : $re_list[0];
    }
```

- preparation of the diary labels for use as a regular expression

```
# The expression will have the form: word1|word2|word3
$guess_pattern = join( "|", string2re( @guess_list ) );
```

- see if the pattern matches any of the official labels (in `@selection_list`)

```
if( @matches = grep( /\b($guess_pattern)/i, @selection_list ) )
```

- sort the list of matches based on the number of words matched

```
@matches = sort {
        ( @b = ( $b =~ /\b($guess_pattern)/gi ) )
        <=>
        ( @a = ( $a =~ /\b($guess_pattern)/gi ) )
        }
    @matches;
```

- if `@matches` has more than one element, then prompt the user. (Simply use `@matches` in the scalar context to check how many labels matched)

```
$selection = chose_from(
        -prompt => "Which $object do you want for '$guess_string'",
        -list => \@matches,
        -mandatory => $mandatory
        );
```

11 Time Calculations

- timestamp recognition is performed by a simple set of regular expressions (of course)

```
while( <DIARY> )
    {
    s/[\n\r]*$//;           # remove DOS's CRNL crap if needed
    next    if /^\s*$/;     # skip blank lines
    next    if $wait_for_login && ! /^(\d{4}-\d{2}-\d{2})\s+/oi;

    #
    # the regexp's below are ordered in order of occurrence for speed
    # i.e.: a simple hh:mm timestamp occurs more often than a
    # login or logout timestamp
    #
    if ( /^\s+(\d{1,2}:\d{2})\s*(.*)$/o )
        {
        entry( $1, $2 );
        }
    elsif ( /^\s+(\d+)\s+Interruption:\s+(.*)/o )
        {
        interruption( $1 );
        }
    elsif ( /^(\d{4}-\d{2}-\d{2})\s+(\d{1,2}:\d{2})\s+login/oi )
        {
        daystart( $1, $2 );
        }
    elsif ( /^(\d{4}-\d{2}-\d{2})(\s+(\.\.)?\s*((\d{4}-)?\d{2}-\d{2}))?\s+
(urlaub|feiertag|krank|sick|holiday)/oi )
        {
        dayoff( $1, $6, $4 );
        }
    elsif ( /^(\d{4}-\d{2}-\d{2})\s+(\d{1,2}:\d{2})\s+logout/oi )
        {
        dayende( $1, $2 );
        }

    # everything else is uninteresting
    }
```

- all timestamps are stored as seconds since 1970, time durations are stored as minutes (for convenience).
- use `POSIX::mktime` to convert *text date-stamps* into *seconds since the epoch* timestamps

```
# accept ISO date-stamps, ie: YYYY-MM-DD
( $yyyy, $mm, $dd ) = $day =~ /(\d+)-(\d+)-(\d+)/;
```

```
$retime = POSIX::mktime( 0, 0, 0, $dd, $mm-1, $yyyy-1900 );
@retime = localtime( $retime );
$weekday = $retime[6] || 7; # move Sunday to the end of the week
$yearday = $retime[7] + 1;  # get the day of the year
```

- the calendar week is used to know when to reset the weekly calculations

```
# work out the calendar week
$thisweek = int( ( $yearday - $weekday + 10 ) / 7 );
```

- the `hhmm2mins` routine is used to convert timestamps from the diary into the number of minutes after midnight.

```
sub hhmm2mins
{
    my $hhmm = shift;
    my $hh;
    my $mm;
    my $mins;

    ( $hh, $mm ) = ( $hhmm =~ /(\d+):(\d+)/o );
    $mins = $hh * 60 + $mm;

    return( $mins );
}
```

- The `mins2hhmm()` routine is used at the last minute to produce something human readable.
  Note: `mins2hhmm()` converts lists of numbers, useful for printing entire lists in one command.

```
sub mins2hhmm
    {
    my @mins = @_;
    my $mins;
    my $hh;
    my $mm;
    my $dd;
    my @times = ();

    foreach $mins ( @mins )
        {
        if ( $mins )
            {
            $hh = int( $mins / 60 );
            $mm = $mins % 60;

            push( @times, sprintf( "%2d:%2.2d", $hh, $mm ) );
            }
        else
            {
            push( @times, "" );
            }
        }
```

```
    return( @times );
    }
```

12 Duplicate reduction

- use normalised representation

```
#
# strip- leading and training blanks, and make the first letter capital
#
$project =~ s/^\s*(.*?)\s*$/\1/;
$project =~ s/\w+/\u$&/g;
```

- always search for existing projects using case-insensitive searches. If a project name doesn't match, or has multiple matches, then it is added as a new project.

```
#
# see if the project already exists (possibly with different
# upper/lower case spelling)
#
my @similar_projects = ();
my $project_re = string2re( $project );
if( ( @similar_projects = grep( /^$project_re$/i, @projects ) ) == 1 )
    {
    $project = $similar_projects[0];
    }
else
    {
    push( @projects, $project );
    }
```

- check for special cases - the time is counted, but not included in the daily total

```
#
# check for special entries
#
$project = "daypause"       if $project =~ /^(pause|lunch|mittag)$/io;
```

13 more to come...

Installer: keeping /usr/local/ under control

Stephen Riehm
http://www.bigfoot.com/~stephen.riehm/

February 22, 2000

Contents

1. Challenge: Clean up and manage /usr/local/

That should be easy!

All you have to do is download, compile and install third party software.

No Problem!

2. No Problem? ... well almost

No central point of administration `/usr/local/` is a local file system on each host.

High maintenence costs Installation of software must be performed locally on each host.

No synchronisation No mechanisms exist to help synchronise the contents of directories distributed around the network. Changes made to one host have no effect on any other host.

No consistent user environment Users aren't given any consistency accross the network. This means that login scripts need to be specialised for each host being used, that the user can only use programs on the hosts where they happen to be installed etc.

Possible loss of data Files overwritten during installation are *lost forever*, often *without warning*.

Removal of software is impossible Since there is no way of finding out which files belong to a given program, there is no way to remove a program and all of its support files.

`/usr/local/` **is full of *Junk*** `/usr/local/` is full of files which no-one knows anything about. Many programs never get used, many are outdated, and many files are *left over* from previous versions of programs which have since been upgraded.

3. A few known solutions

(as of 1.10.1999)

My Old Installer System In 1996 I developed a set of Ksh scripts to help. They were never published, however, they have been in use at pc-plus (now: varetis AG[1]) ever since.

GenOpt Written by Steffen Beyer[2].
GenOpt is essentially identical to my old system. Based on bourne shell scripts - it is slow and limited, but works everywhere.

Lude Home Page[3]
Lude doesn't centralise C, but bundles pre-compiled versions of the software for simple distribution. Installation still needs to be carried out on each individual host.

opt_depot Home Page[4]
Also similar to my old system, but written in C - not portable.

In short - all of these systems weren't what I was looking for...

4. General Approach

- Use a single NFS directory for each host platform (based on Operating System and Hardware type)
- Each program is installed in its own directory (á la `/opt/` on some systems)
- Softlinks used to make programs *publically available*
- Scripts etc. help maintain links
- Supply ways to find out what is installed

[1] See URL http://www.varetis.de
[2] See URL http://www.engelschall.com/u/sb/
[3] See URL http://www.iro.umontreal.ca/lude2/lude2.toc.html
[4] See URL http://encap.cso.uiuc.edu/

5. Benefits of this approach

- Central point of administration (per host platform)
- Lower maintenence costs
- Automatic Synchronisation of hosts (per host platform)
- Consistent user environment
- Data is never overwritten (at most a link)
- De-installation of software is trivial
- Junk can be identified and cleaned up easily

6. My old System

- Ksh scripts
- Very portable
- **Very slow**
- **Difficult to use**
- **No cross-platform operations**
- **No cross-platform information**

7. Approach using perl

- Using perl is *quick*
- Ease of use can be increased (with a GUI if need be)
- Client / Server architecture allows cross platform operations
- Client / Server architecture allows cross platform information
- Easier to maintain
- **Probably won't run** *out of the box* :-(

8. Instant Success !

- First prototypes showed an **enormous** speed increase!
  The shell scripts would take minutes to configure and set up the links for a program like perl.
  The perl script installed *all the applications in /usr/local/ in about the same time!*.
- Better consistency checks were possible (and *easy*)

9. Object Orientation

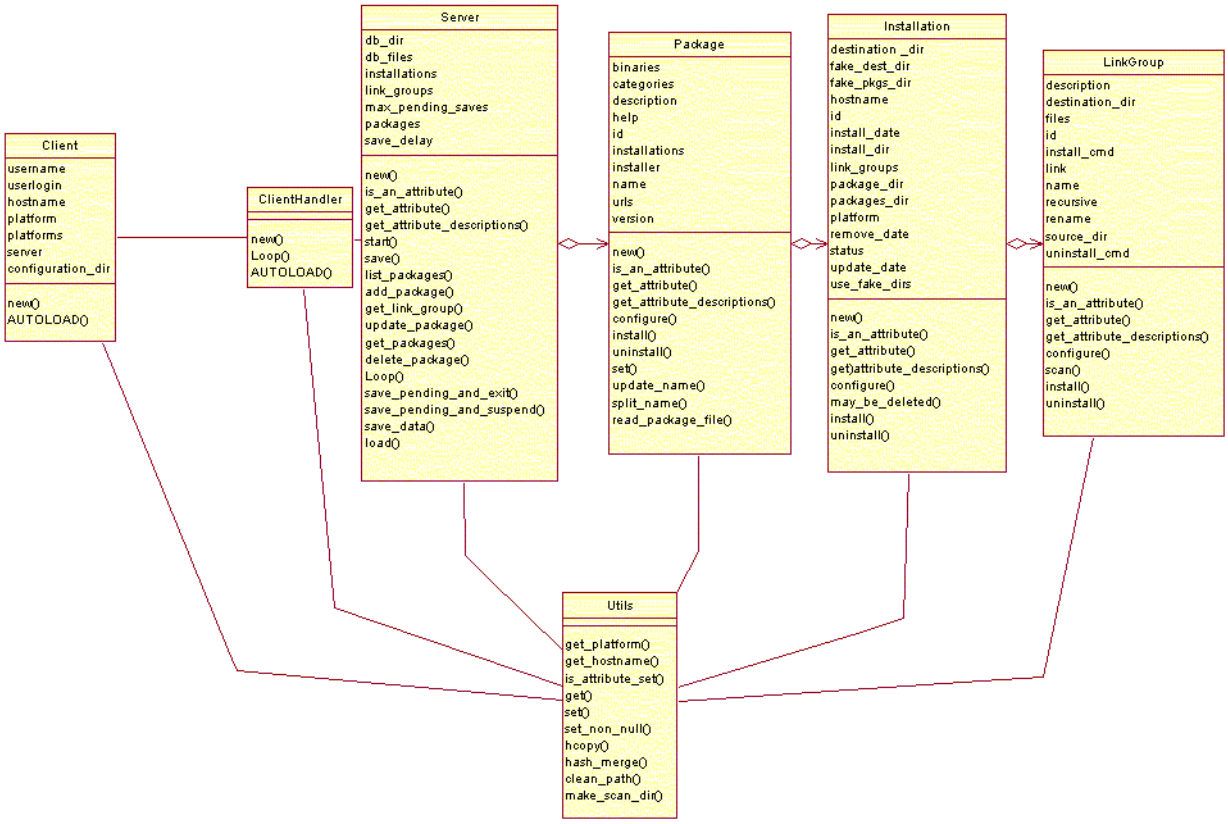
The gap between theory and practice in Object Orientation has always caused me problems. The idea of treating things as objects appeals to me greatly, but the fun and games I've had trying to program (in C, C++ and Perl) my concepts has always left me with a sour aftertaste.

Data Inheritence of classes is very difficult to implement - merging the contents of several hashes is not an easily automatable task.

Checking for the correct use of class attributes is possible, but when combined with data inheritence, it becomes a lot more difficult and cumbersome.

I got around these problems by using *containers* instead of inheritence.

10. The Class Diargram for Installer



11. Problems with Classes

Inheritance The idea of using a base class to define the basic object operations didn't work. i.e.: attribute checking etc. The concept of inheriting a HASH and adding to it's contents, and the idea of using a class hash for the definition of the attributes (which attributes are allowed, their default values etc.) doesn't work well with the `@ISA` construction.

Without this kind of attribute mechanism, incorrect use of an object cannot be detected. (no error messages, no indication that anything is wrong - except that perhaps a new attribute has been created - which the object knows nothing about)

Updating information in an Object When changing information in a container object (ie: `Package`) then the changes have to be reflected on the server. To do this I had to find a way to *recursively merge* multiple hashes.

12. hash_merge

- recursively assign the values each key in a hash to the destination hash
- need to differentiate between `HASH` and other data types.
- **Objects are not HASH's** according to `ref`.
- How to properly merge the contents of an `ARRAY`?

13. hash_merge posted on UseNet

```
sub hcopy(\%\%)
    {
    my($h1, $h2) = @_;

    foreach my $k (keys %$h2)
        {
        if ( ref( $h2->{$k} ) eq 'HASH')
            {
            $h1->{$k} ||= {} ;
            hcopy( $h1->{$k}, $h2->{$k});
            }
        else
            {
            $h1->{$k} = $h2->{$k};
            }
        }
    }
```

- **Problem:** doesn't handle Objects!

14. My hash_merge (work in progress)

- checks for Objects
- *can* restrict which stuff gets merged (ie: only pre-defined attributes)
- *can* call routines to set / get attribute data, if routines are defined.
- *can* remove data from the destination hash (ie: if the key no longer exists in the source hash)
- detects cyclic links
- is probably far too complicated
- still doesn't quite work the way I would like it to

  The source code is in the appendix.

15. Client Server database

Client / Server database I wanted to store the information about the installed software in a central database, however, I didn't want to use a commercial or full blown database (too big), and the simple databases don't cater for complex data structures (hashes of hashes of ...)

Jochen Wiedmann's[5] **PlRPC V0.2001** modules were the solution, however they are not included in the standard perl distribution, and *can* cause problems when installing on "*exotic*" machines.

The `RPC::PlServer` module is an object which listens for connections from clients. When a client calls, the RPC::PlServer object clones itself, the clone then talks to the client, while the original object continues listening for new connections.

This is a problem if you want a process to have control over the database. For this reason, the Server class looks after the database, and starts a ClientHandler. The ClientHandler has a reference to the Server object, and uses an AUTOLOAD function to pass on calls to the Server object.

16. Problems that remain to be solved

Bootstrapping How to write installer in such a way that it can be installed on a nacked unix system. i.e.:

1. configure the required directories
2. install perl (in a temporary directory? in the directories which will be used by installer? ie: `/opt/perl-5.005_03`)
3. install required modules (via CPAN?)
4. find a server on the network (if one is available)

Automatic Configuration If installer is installed on a single machine system - can it be configured to NOT look in the net for a server?

If a server already exists, get the general configuration information from the server.

Server downtime Obviously, if the server is down, the clients wont work - is there a way for the clients to start the server remotely?

A. `hash_merge` code

Last updated: Fri 04 Feb, 2000

```
#
# routine to merge the contents of two hash's
# h1 is the target and one of the sources - the idea is not to
# duplicate data, but to really merge the data if possible
#
# used to prevent circular links in hash tables
my %seen_hashes = ();
sub hash_merge
    {
    my %parameters = @_;
#     print "hash_merge params: ", Dumper( \%parameters );
    my @hashes = @{$parameters{'hashes'}};
    my $target = shift(@hashes);
    my $defaults = $parameters{'defaults'} || $target;
    # the defaults for the following flags are set below
```

[5] See URL mailto:joe@ispsoft.de

```
    my $restrict = $parameters{'restrict'};
    my $execute_subs = $parameters{'execute_subs'};
    my $strip_undefs = $parameters{'strip_undefs'};
    my $rejects = $parameters{'unknown_attrs'};
    my $source;

    if( $trace )
        {
        print "trace: merging:\n";
        foreach( @hashes )
            {
            print ref( $_ ), "\n";
            }
        }

    #
    # set default values
    # checking with || doesn't work - as 0 fails :-(
    #
    $restrict = 1   unless defined( $restrict );
    $execute_subs = 0       unless defined( $execute_subs );
    $strip_undefs = 0       unless defined( $strip_undefs );

    $trace && $restrict && print "trace: restrictive merge\n";
    $trace && $execute_subs && print "trace: will execute subs\n";
    $trace && $strip_undefs && print "trace: strip undefs\n";

    $trace && print "trace: non-hash - return early\n"
        if ref( $target ) =~ /ARRAY|SCALAR|REF|CODE|GLOB/;
    return undef    if ref( $target ) =~ /ARRAY|SCALAR|REF|CODE|GLOB/;

    if( (caller( 1 ))[3] ne 'Installer::Utils::hash_merge' )
        {
        %seen_hashes = ();
#   print "reset\n";
        }

    foreach $source ( $defaults, @hashes )
        {
        $trace && print "trace: merging $source with $target\n";
        $trace && print "trace: ", ref( $source ), "\n";
        #
        # ensure that the source of the new data is also a hash or an
        # object (which is also a hash :-/)
        #
        if( ref( $source ) !~ /^ARRAY|SCALAR|REF|CODE|GLOB$/ )
            {
            my $key;
            foreach $key ( keys( %{$source} ) )
                {
                #
                # if restrict is on, only merge values defined in the
                # defaults table
```

```
                #
                if( $restrict && ! exists ( $defaults->{$key} ) )
                    {
                    if( $rejects )
                        {
                        $rejects->{$key} = $source->{$key};
                        }
                    else
                        {
                        cluck( "undefined ",
                            ( ref( $target ) eq 'HASH' )
                                ? "option" : "attribute",
                            " '$key'" );
                        }
                    next;
                    }
                else
                    {
                    #
                    # merge a value from the source table into the
                    # target table
                    #
#               print "source $key: $source->{$key}\n"
#                   if( ref( $target->{$key} ) eq 'HASH' )
#                       && ( ref( $source->{$key} ) eq 'HASH' );
#           $trace && print "skipping $key - already seen $source->{$key}\n"
#                   if $seen_hashes{$source->{$key}};
                    if( ( ref( $target->{$key} ) eq 'HASH' )
                            && ( ref( $source->{$key} ) eq 'HASH' )
                            && ! $seen_hashes{$source->{$key}}++
                            )
                        {
                        #
                        # call recursively to merge hashes in hashes.
                        # use restrict - 1 to allow restrictions down to a
                        # given level of the hash tree.
                        #
                        $trace && print "trace: recursive hash merge\n";
                        hash_merge(
                            'hashes' => [ $target->{$key}, $source->{$key} ],
                            'default' => $defaults->{$key},
                            'restrict' => ( $restrict - 1 )
                            );
                        }
                    elsif( $execute_subs
                            && ( ref( $source->{$key} ) eq 'CODE' ) )
                        {
                        #
                        # run a routine to get the value for the
                        # target
                        #
                        $trace && print "trace: running code to set $key\n";
                        $target->{$key} = &{$source->{$key}}();
```

```
                        }
                    elsif( defined( $source->{$key} )
                            || ! exists( $target->{$key} ) )
                        {
                        # print "setting $key to $source->{$key}\n";
                        if( ref( $source->{$key} ) eq 'HASH' )
                            {
                            $trace && print "trace: cloning $key\n";
                            $target->{$key} = dclone( $source->{$key} );
                            }
                        else
                            {
                            $trace && print "trace: setting $key\n";
                            $target->{$key} = $source->{$key};
                            }
                        }
                    else
                        {
#                       print "don't know what to do!\n";
#                       print "source:", Dumper( $source );
#                       print "target:", Dumper( $target );
                        }
                    }
                }
            }
        else
            {
            print "broken hash_merge call in", join( " ", caller() ), "\n";
            }
        }

    if( $strip_undefs )
        {
        $trace && print "trace: stripping undef's\n";
        foreach( keys( %{$target} ) )
            {
            delete $target->{$_}    unless defined( $target->{$_} );
            }
        }

    # print "hash_merge returning:\n", dumper( $target );
    return $target;
    }
```

Entwicklung größerer Webanwendungen mit XML und Tamino

Jochen Wiedmann
jochen.wiedmann@softwareag.com

8. Februar 2000

Inhaltsverzeichnis

1 Einführung

Vor kurzem hatte ich die Aufgabe, mir über die Entwicklung von Webanwendungen mit der Datenbank **Tamino** zu beschätigen. Tamino ist die erste native XML-Datenbank und ein Produkt meines Arbeitgebers, der **Software AG**. Dabei versuchte ich zunächst, die klassischen Technologien einzusetzen, d.h. Templatesysteme wie **EmpPerl** oder **HTML::EP**. Wie gewohnt ließ sich damit ein rascher Erfolg erzielen. Nachdenklich wurde ich aber, als ich anschließend die schnell entwickelte Anwendung von **HTML** auf **WML**, d.h. die Anwendung mit **WAP**-Browsern portierte: Der dabei entstandene Aufwand war vergleichbar mit dem für die Neuentwicklung.

Noch nachdenklicher wurde ich, als ich mir klarmachte, daß sich nicht nur mein eigener Arbeitsaufwand verdoppelt hatte: Ich hatte wie üblich nackte HTML- bzw. WML-Seiten entwickelt. In einer professionellen Anwendung würden nun diese nackten Seiten an einen Designer übergeben werden, der die immer gleichen Arbeitsschritte für alle HTML- bzw. WML-Seiten wiederholen würde. Dabei würde er wie üblich einige kleinere Fehler begehen, etwa Nichtbeachtung von Sonderzeichen, d.h. einige Seiten zur Korrektur wieder an mich zurückgeben.

Die Vorstellung, dieselbe Verdopplung des Aufwands nicht nur bei einer kleinen, sondern gar bei einer größeren Anwendung zu erleben, fand ich bestürzend. Durch Diskussion mit Kollegen (vor allem **Martin Bauer**) stieß ich auf die **Servlet**-Technologie, auf die Trennung von Daten, Logik, auf XML als internes

Darstellungsformat und auf Einsatz von XSL-Stylesheets als Mittel der Präsentation. Offensichtlich boten diese Prinzipien enorme Vorteile, speziell die Trennung in mehrere logische Arbeitsschritte.

Umgekehrt fand ich aber die Entwicklung von Servlets relativ mühsam. Für das Lesen der XML-Dateien aus Tamino fehlte ein Framework. XSL-Stylesheets konnten nur clientseitig eingesetzt werden und auch da nur mit dem Internet Explorer 5. (Meine Erfahrungen mit dem Einsatz von clientseitiger Software sind deutlich schlecht.) In einem gewissen Sinn fühlte ich mich damit an die Anfänge der Web-Entwicklung mit CGI-Binaries zurückgeworfen: Für jede darzustellende HTML- oder WML-Seite mußte im Prinzip ein eigenes Programm entwickelt werden. Damit waren meines Erachtens die Vorteile der traditionellen Templates verloren. Erneut begann ich unzufrieden zu werden.

Langsam begannen sich in mir neue Ideen zu regen, die allerdings noch reichlich verschwommen waren. Und an dieser Stelle stieß ich auf **Cocoon**, ein Apache-Projekt. Der Cocoon-**Produzent** kann mit beliebigen Datenquellen (Datei, Tamino, SQL-Datenbanken, ...) arbeiten. Die generischen **Prozessoren** ermöglichen ein Arbeiten mit Templates. Und die **Präsentatoren** sind das Framework für die Browser-abhängige Präsentation der Daten. Abschließend ist Cocoon ein Servlet und erbt damit die meisten Vorteile dieser Technologie.

Leider ist Cocoon eine reine Java-Lösung. Also begann ich nach ähnlichen Ideen in der Perl-Welt zu suchen - und fand nichts. Das ist der Grund für diesen Vortrag: Die Anregung, diese Ideen zu übernehmen und zu verbessern.

2 Traditionelle Systeme

Für die traditionelle Web-Entwicklung haben sich bislang zwei völlig unterschiedliche Systeme durchgesetzt. Dies sind zum einen die Template-Systeme (s. Abschnitt 2) und zum anderen die Servlets (s. Abschnitt 2.1).

2.1 Template-Systeme

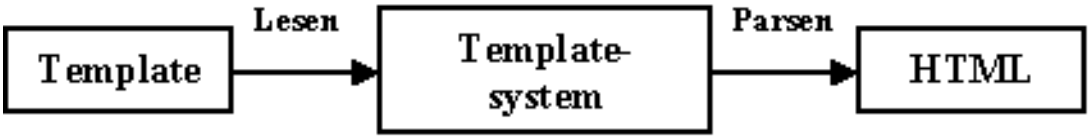
Das Prinzip der Template-Systeme ist ein sehr einfaches. Zur Generierung der dynamischen HTML-Seite wird ein Template verwendet, d.h. eine Datei, die im wesentlichen HTML enthält und dadurch auch für Nichtprogrammierer, vor allem Designer, einigermaßen leicht verständlich ist. In das Template werden spezielle Pattern eingebettet, die etwa so aussehen:

```
<p>Die aktuelle Uhrzeit: Es ist jetzt
<perl>scalar(localtime())</perl></p>
```

Ein spezieller Präprozessor, das **Templatesystem** liest die Datei, durchsucht sie nach Pattern und ersetzt diese. Im Beispiel würde etwa die Perl-Funktion

```
localtime()
```

aufgerufen und Ihre Ausgabe in die HTML-Seite eingesetzt.



Daß das Prinzip der Templatesysteme sinnvoll und gut ist, scheint mir schon alleine dadurch belegt, daß es eine ganze Reihe von unterschiedlichen Systemen gibt, die sich völlig unabhängig voneinander durchgesetzt haben. Die bekanntesten sind:

- Active Server Pages (ASP) (s. Abschnitt 6) sind eine Entwicklung von Microsoft für ihren Internet Information Server (s. Abschnitt 6). Der Leistungsumfang von ASP ist eher eingeschränkt, was wohl auch am größten Vorzug der ASP liegt: Der Sprachunabhängigkeit. Man kann ASP unter anderem

mit Visual Basic (VBScript), JavaScript und Perl (PerlScript) programmieren. Eine plattformunabhängige, aber auf Perl eingeschränkte Version der ASP ist Apache::ASP (s. Abschnitt 6). Dank des Microsoft-Marketing ist ASP vermutlich die weltweit bekannteste Lösung.

- PHP (PHP Hypertext Processor, ein rekursives Akronym im Stil von GNU) ist ein Templatesystem, das seine eigene Skriptsprache mitbringt. PHP wird oft als "lightweight Perl" bezeichnet, was daran liegt, daß speziell die Linux-Distributionen und die meisten professionellen Anbieter von Webspace PHP fertig in den Webserver installiert mitbringen - dadurch kann man PHP-Entwicklung praktisch überall und ohne Installationsaufwand beginnen. Bei kleineren Anwendungen kommen dadurch die Nachteile der Sprache PHP gegenüber Perl (fehlende Modularität, praktisch kein OO, kein einheitliches Datenbankinterface) nicht so deutlich zur Geltung. PHP ist fast ebenso unabhängig von Betriebssystem und Webserver wie der Einsatz von Perl.

- Eine Reihe von Perl-basierenden Lösungen stehen zur Verfügung, z.B. CIPP (s. Abschnitt 6) von Jörn Reder, EmbPerl von Gerald Richter, HTML::EP (s. Abschnitt 6) vom Autor dieses Vortrags oder Mason (s. Abschnitt 6) von Jonathan Swartz. Dank der Mächtigkeit von Perl sind diese Systeme meist besonders einfach und elegant, konnten sich aber leider auch nie auf eine gewisse Konvergenz einigen. Dadurch wurde ihre Entwicklung nie so vorangetrieben wie z.B. die von PHP und sie sind deshalb in der Öffentlichkeit weitgehend unbekannt.

- Während die bisher vorgestellten Systeme auf Skriptsprachen basieren, beruhen die Java Server Pages (JSP) (s. Abschnitt 6) von Sun überraschenderweise auf der Compilersprache Java. Demzufolge wird eine JSP-Seite auch quasi compiliert und glänzt so durch gute Performance: Das Parsen der Seite findet nur einmal statt.

Dies sind die Eigenschaften der Templatesysteme:
+ RAD durch niedrige Turnaround-Zeiten + Sehr einfache Anwendung auch schwieriger Technologien wie Zugriff auf SQL-Datenbanken oder HTTP-Sessions. + Die Systeme sind meist ausgereift, bewährt und stabil. - Keinerlei Trennung von Logik und Präsentation - Schwieriger Einsatz von Cachetechnologien; dadurch oft schlechte Performance

2.2 Servlets

Ein ganz anderes Prinzip verkörpern Servlets (s. Abschnitt 6). Das prinzipielle Vorgehen sieht hier so aus: Eine statische HTML-Seite wird von einem Parser gelesen. Ein Codegenerator erzeugt daraus Java-Quelltext, der ein diese HTML-Seite repräsentierendes Java-Objekt aufbaut, anschließend in einen String umwandelt und an den Browser verschickt. Die Aufgabe des Programmieres besteht nun darin, Java-Quelltext einzubauen, der das Java-Objekt modifiziert.

Das compilierte Programm wird von der Servlet-Engine ausgeführt: Meist ein separates Programm, in dem die Java Virtual Machine läuft und das über Sockets mit dem Webserver kommuniziert. Der offensichtliche Vorteil der Servlet-Engine ist die sehr einfache Verwendung von Threads mit ihrer simplen Kommunikationsmöglichkeiten. Das macht sich etwa bei persistenten Datenbankverbindungen sehr vorteilhaft bemerkbar.

Die Eigenschaften der Servlets unterscheiden sich drastisch von denen der Templates:
+ Einfache Interprozeßkommunikation, persistente Datenbankverbindungen + Kein Parsen zur Laufzeit + Ausgezeichnete Portabilität, dank der Akzeptanz von Java in der komerziellen Welt - Keine Trennung von Logik und Präsentation - Hohe Turnaroundzeiten; dadurch kaum RAD

3 Neue Prinzipien

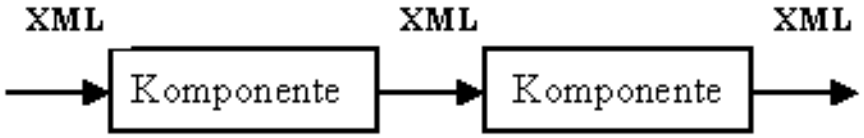
Im folgenden sollen nun die Prinzipien der XML-basierenden Entwicklung von Web-Anwendungen erläutert werden.

3.1 Komponentenbildung

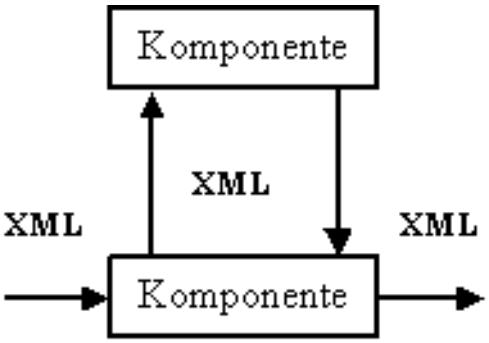
Was in der Softwareentwicklung meist schon lange selbstverständlich ist, wird bei der Webentwicklung noch kaum genützt: Die Komponentenbildung. Gemeint ist die Entwicklung von kleinen Bausteinen, die Komponenten und die Entwicklung der Software durch die Kombination derselben.

Minimale Anforderungen für den Einsatz von Komponenten sind:

Chaining Verkettung von Komponenten bedeutet, daß die Ausgabe einer Komponente einer anderen Komponente als Eingabe zur Verfügung gestellt wird. Dieses Prinzip ist von den Unix-Pipes bekannt und hat sich dort ausgezeichnet bewährt.



Ableitung Aus der OO-Programmierung ist die Ableitung einer Subklasse bekannt. In unserem Fall kann bedeutet dies, daß eine Komponente intern eine andere Komponente aufruft. Das ist etwa dann sinnvoll, wenn eine vorhandene Komponente dem gewünschten Verhalten sehr nahe kommt.



3.2 XML als Kommunikationsmittel

Die Frage ist nun, wie diese Komponenten intern miteinander kommunizieren sollen. Im Falle der bereits erwähnten Pipes ist das sehr naheliegend und einfach: Über einen Strom von Bytes. Die naheliegende Idee ist es, dabei XML zu verwenden. XML ist in diesem Fall der kleinste gemeinsame Nenner der gewünschten Ausgabeformate, z.B. HTML oder WML.

Aus Effizienzgründen darf dabei natürlich nicht XML als Strom von Bytes übergeben werden, sondern als DOM-Tree: DOM (Document Object Model) ist ein vom W3C (s. Abschnitt 6) standardisiertes API.

3.3 XSL-Stylesheets

XSL-Stylesheets sind im wesentlichen nichts anderes als unsere altbekannten Templatesysteme. Vergleicht man sie mit den Templatesystemen, haben sie sogar starke Nachteile, insbesondere sind sie erheblich unhandlicher. Sie haben aber andererseits große Vorteile:

- XSL-Stylesheets erhalten einen XML-Baum als Eingabe und generieren als Ausgabe ebenfalls XML, optional aber auch HTML oder WML. Damit sind sie ideale Komponenten im Sinne der vorigen Punkte.
- XSL unterliegt einer Standardisierung durch das W3C. Damit ist eine breite Akzeptanz zu erwarten.
- Bereits in der Entwicklung sind XSL-Stylesheet-Editoren: Diese werden ähnlich einfach bedienbar sein wie HTML-Editoren.

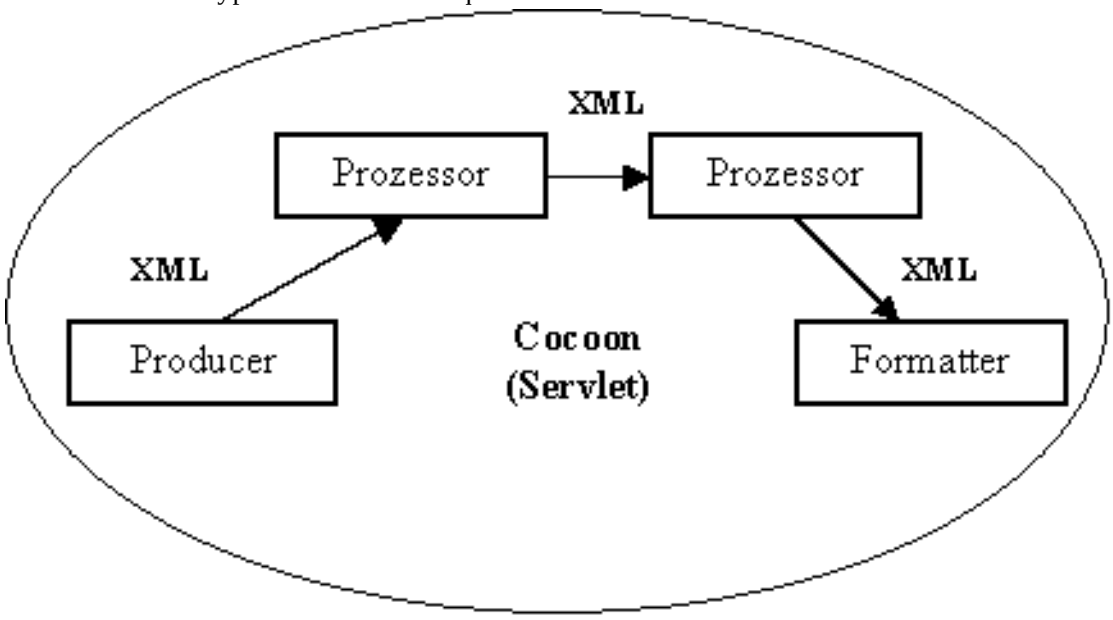
Zusammenfassend sind XSL-Stylesheets das Mittel der Wahl für die Präsentation der Daten.

4 Cocoon

Cocoon (s. Abschnitt 6) ist ein spezielles Servlet, das im Prinzip ein Framework für den Einsatz der oben genannten Prinzipien darstellt. Die Architektur von Cocoon unterscheidet

- **Produzenten (Producer)**, d.h. Komponenten, die einen XML-Baum erzeugen, indem sie z.B. eine Datei lesen, ein Tamino-Dokument lesen.
- **Prozessoren (Processors)**, d.h. Komponenten, die in die Mitte einer Pipeline von Komponenten gestellt werden. Typische Prozessoren sind etwa ein SQL-Prozessor, der SQL-Queries durchführt und die Ergebnisse als XML-Fragment umwandelt, ein analoger Tamino-Prozessor und ein LDAP-Prozessor, sowie Prozessoren im Stil der Java Server Pages, die Java- bzw. JavaScript-Fähigkeiten integrieren. Ferner stehen XSL-Stylesheet-Prozessoren zur Verfügung.
- **Präsentatoren (Formatter)**, d.h. Komponenten, die den fertigen XML-Baum in seine endgültige Form (HTML, WML, ...) umwandeln und an den Browser schicken. Dabei ist Cocoon in der Lage, einen für den speziellen Browser geeignetes Stylesheet auszuwählen.

Der Ablauf eines typischen Cocoon-Requests ist etwa so:

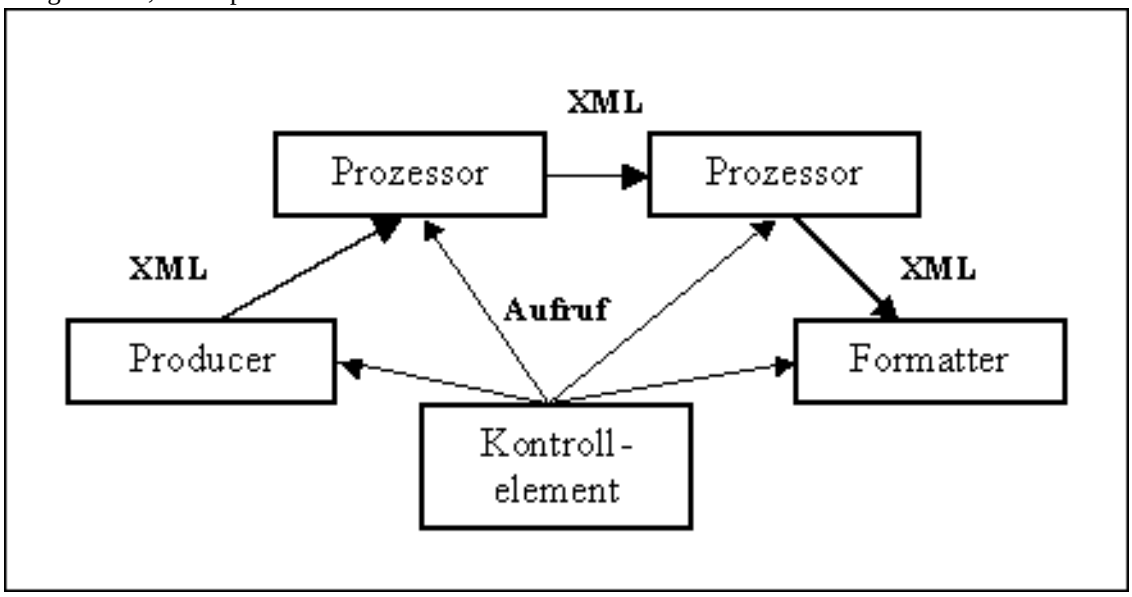


Arbeitet man mit Cocoon, so stellt man aber auch rasch Defizite fest: Die Auswahl der Pipeline-Komponenten erfolgt stets über PI's (Processing Instructions) im XML-Baum, den der Produzent generiert hat. Es fehlt Support für Stylesheets, die vom virtuellen Host oder dem Directory abhängen. Insbesondere sind keine zentralen Einstellungen vorgesehen.

5 XML::EP

Abschließend soll nun ein Vorschlag für eine Perl-basierende Lösung vorgestellt werden. Diese erhält den Arbeitstitel **XML::EP** (**E**mbedded **P**erl). Ihre Architektur ähnelt weitgehend der von Cocoon, als Neuheit wird ein sogenanntes Kontrollelement eingeführt. Das Kontrollelement hat als Aufgabe die Bildung der Komponentenpipeline. Dabei werden der virtuelle Host, das Verzeichnis, der Dateiname und der Browser

des Clients berücksichtigt. Wie bei Cocoon haben die Produzenten, aber auch die Prozessoren nach wie vor die Möglichkeit, die Pipeline zu erweitern.



Die meisten Bestandteile dieser Software sind bereits vorhanden, insbesondere Produzenten (dank XML::DOM (s. Abschnitt 6)) und Präsentatoren (dank XML::XSL (s. Abschnitt 6)). Was vor allem fehlt, sind Standardprozessoren im Stil von EmbPerl, Mason und CIPP. Hier sind die Autoren dieser Pakete aufgerufen: Das neue ist ja nun, daß es keinerlei Problem mehr darstellt, mehrere Systeme zu integrieren. Ferner fehlt viel Arbeit zur Steigerung der Effizienz: Insbesondere sind die Cache-Mechanismen von Cocoon nachzubilden: Z.B. ist Cocoon in der Lage, einen Produzenten mit nachgeschaltetem Prozessor als gemeinsamen Produzenten zu cachen, sofern beide Komponenten dies vorsehen.

Es gibt viel zu tun: Packen wir's an!

6 Referenzen und Links

- Apache::ASP: ftp://ftp.funet.fi//pub/languages/perl/CPAN/modules/by-module/Apache/
- ASP: http://msdn.microsoft.com/workshop/server/asp/ASPover.asp
- CIPP: ftp://ftp.funet.fi//pub/languages/perl/CPAN/modules/by-module/HTML/
- Cocoon: http://xml.apache.org/cocoon/
- EmbPerl: ftp://ftp.funet.fi//pub/languages/perl/CPAN/modules/by-module/HTML/
- HTML::EP: ftp://ftp.funet.fi//pub/languages/perl/CPAN/modules/by-module/HTML/
- ePerl: ftp://ftp.funet.fi/pub/languages/perl/CPAN/authors/id/RSE/
- Internet Information Server: http://www.microsoft.com/ntserver/web/
- Java Server Pages: http://java.sun.com/products/jsp/
- Mason: ftp://ftp.funet.fi//pub/languages/perl/CPAN/modules/by-module/HTML/
- Software AG: http://www.softwareag.com/
- Servlets: http://java.sun.com/products/servlet/

- Tamino, die native XML-Datenbank der Software AG: http://www.softwareag.com/tamino/
- W3C (World Wide Web Consortium): http://www.w3.org/
- XML::DOM, ein DOM-Parser für XML: ftp://ftp.funet.fi/pub/languages/perl/CPAN/modules/by-module/XML/
- XML::XSL, ein Stylesheet-Prozessor: ftp://ftp.funet.fi/pub/languages/perl/CPAN/modules/by-module/XML/

Tao Te Perl: Stringverarbeitung und ostasiatische Sprachen

Jens Ohlig
jo@devcon.net

16. Februar 2000

Inhaltsverzeichnis

Im Jahr des Drachens, das hierzulande eher als Jahr 2000 bekannt ist, sollten wir einen kurzen Blick auf Tatsache werfen, dass für es für den grössten Teil der Erdbevölkerung mehr als ASCII oder ein paar Umlaute gibt. In dem Vortrag sollen Probleme vorgestellt werden, die bei der Verarbeitung von Texten aus China, Japan und Korea (CJK-Texte) und – im eingeschränkten Maße – in vietnamesischen Texten auftauchen und wie diese mit Perl zu lösen sind.

1 Nicht-elektronische Probleme: Warum CJK-Schriftsysteme so komplex sind

Die chinesische Schrift entstand vor ca. 4300 Jahren. Ihr Ursprung liegt weitgehend im Dunkeln. Der Legende nach wurde die Schrift von einem Gelehrten erfunden, der sie aus den Spuren von Vögeln und wilden Tieren entwickelte. Wahscheinlicher ist jedoch, dass sie ursprünglich religiösen Zwecken diente und für Weissagungen und Orakel benutzt wurde.

Im Westen weitverbreitet ist der Irrtum, dass es sich bei der chinesischen Schrift um eine reine Bilderschrift handelt. Tatsächlich gibt es einige bildhafte Zeichen, die meisten Wörter werden jedoch aus grundlegenden Zeichen (den Radikalen) zusammengesetzt und bestehen aus zwei (oder mehr) Zeichen, bei denen eins meist bedeutungstragend ist und das andere auf die Aussprache hinweist.

Vor der Einführung der chinesischen Schrift besassen weder Korea noch Japan eigene Schriftsysteme. Die Übernahme eines so komplexen Schriftsystems wie des Chinesischen erklärt sich daraus, dass in der Antike China die führende Hochkultur im ostasiatischen Raum darstellte; nicht nur in der Literatur, sondern auch in Bereichen wie Politik und Religion prägte der “grosse Bruder” China die beiden Nachbarnationen. Vieles, was heute typisch japanisch scheint, wie der Zen-Buddhismus oder typisch koreanisch, wie die konfuzianische Ethik im Miteinander, hat seinen Ursprung in China. Trotzdem ist es wichtig anzumerken, dass weder die koreanische noch die japanische Sprache mit dem Chinesischen verwandt sind und die chinesischen Zeichen somit quasi einen Fremkörper in beiden Sprachen bilden: Japanisch ist eine isolierte Sprache und gehört keiner bekannten Sprachfamilie an. Eine Verwandtschaft mit dem Koreanischen wird von einigen Linguisten aufgrund der sehr ähnlichen Grammatik angenommen, konnte aber bisher nicht belegt werden. Koreanisch gehört nach vorherrschender Lehrmeinung zur Gruppe der altaischen Sprachen, zu der z.B. auch Mongolisch, Ungarisch und Finnisch gehören.

Korea übernahm als erstes die chinesische Schrift in seiner Gesamtheit zum damaligen Entwicklungsstand, weshalb sich im Koreanischen die Zeichen in ihrer traditionellen Form, anders als in der Volksrepublik China, erhalten haben. In Japan erfolgte die Übernahme der Zeichen in Schüben mit erheblichem zeitlichen Abstand, teils direkt aus China, teils über den Umweg der "Kulturbrücke" Korea. Die japanische Schriftsprache weist daher chinesische Zeichen aus verschiedenen Stadien der Entwicklung auf, die z.T. auch völlig unterschiedliche Lesungen haben: eine rein japanische Lesung (Kun) und eine sinojapanische Lesung (On).

Wie oben erwähnt, sind Japanisch und Koreanisch eigentlich nicht besonders der chinesischen Schrift angepasst. Dazu kommt, dass das komplexe Schriftsystem der chinesischen Zeichen dazu führte, dass nur eine kleine Gruppe von Gelehrten überhaupt lesen und schreiben konnten. Beide Sprachen entwickelten daher eigene Schriften, was im Japanischen dazu führte, dass heutzutage drei verschiedene Schriftsysteme in einem Satz auftauchen können (vier, wenn die lateinische Schrift mitgezählt wird). Die japanische Silbenschrift Hiragana war ursprünglich eine Schrift für Frauen. Sie ist heutzutage die normale Schrift für grammatische Partikel (Genitivendung etc.) und Wörter, die keine chinesischen Zeichen besitzen. Die zweite Silbenschrift im Japanischen heisst Katakana und wird heutzutage für Fremdwörter (z.B. ri-nu-ku-su für Linux und po-su-to-su-ku-ri-pu-to für PostScript) oder zur Hervorhebung (ähnlich unserer Kursivschrift) benutzt.

In Korea dauerte es wesentlich länger, bis sich ein eigenes Schriftsystem neben dem chinesischen entwickelte. Am 9. Oktober 1446 erliess König Sejong der Große eine eigene koreanische Schrift, die nach Katalogisierung aller möglichen phonetischen Elemente im Koreanischen von Grund auf neu entwickelt worden war. Diese Volksschrift, die seit ca. 1910 "Hangul" genannt wird, ist eine Alphabetschrift, die nur auf den ersten Blick den chinesischen Zeichen ähnelt und aus 24 Einzelzeichen besteht. Sie gilt Linguisten als eine der logischsten und wissenschaftlichsten Schriften überhaupt. Heute ist Hangul in Korea die übliche Schrift. Die Verwendung der chinesischen Zeichen im Koreanischen geht tendenziell stark zurück, ihre Benutzung ist jedoch von der untersuchten Textsorte abhängig: In geschichtlichen oder philosophische Texten kann der Anteil an chinesischen Zeichen bis zu 50% betragen, in Modezeitschriften oder Kinderbüchern findet man gewöhnlich kein einziges chinesisches Zeichen.

Dem Perl-Programmierer ist dieses Nebeneinander verschiedener Schriften vielleicht nicht aus seiner Muttersprache bekannt, in Perl kennt er jedoch das Übernehmen von Notationen aus fremden "Kulturbereichen". Die regulären Ausdrücke in Perl und die Stringverarbeitung gehen auf sed und awk zurück, andere Notationen wurden aus C oder sogar aus Basic entlehnt.

2 Elektronische Probleme: Warum ein Byte nicht ein Zeichen ist

Da die Verarbeitung von Zeichen ein grundlegender Bestandteil von Perl-Programmen ist, treten hier verschiedene Probleme auf. Anders als bei den meisten europäischen Sprachen reichen hier für die Darstellung im Computer 8 Bit für ein Zeichen nicht aus: Der chinesische Zeichensatz umfasst mehr als 8.000 einzelne Zeichen. Dagegen wirkt die Zahl von 256 Zeichen, die in vielen europäischen Zeichensätzen zur Verfügung stehen, fast ärmlich.

Die Lösung für dieses Problem ist bestechend einfach: Ein Zeichen (zumindest in einer der ostasiatischen Schriften) ist nicht mehr ein Byte lang, sondern zwei Byte. Dadurch muss der Perl-Programmierer in einigen wichtigen Punkten umdenken. Perl nimmt an, dass ein Zeichen immer ein Byte lang ist, der idomatische reguläre Ausdruck `/./` trifft jetzt also nicht mehr auf genau ein Zeichen zu, ebenso wie für `split` explizit ein regulärer Ausdruck definiert werden muss, wenn wir einen String in ein Array von Zeichen zerlegen wollen. Ein Beispiel hierzu: Wir wollen unsere koreanische Rezeptsammlung auf das Vorkommen von Kimtschi untersuchen. Kimtschi (in der Transkription nach McCune-Reischauer: "Kim Ch'i") ist eine Art Sauerkraut, hergestellt aus in Chilipulver eingelegtem Chinakohl. Die Zeichen "Kim" und "Ch'i" haben im Zeichensatz KS X 1001:1992 die Positionen 0x31 0x68 (Kim) und 0x44 0x21 (Ch'i). Eine naheliegende Lösung wäre etwa folgender Perl-Code:

```
while(<>) {
  print if (m#\x31\x68\x44\x21#); # auf der Suche nach Kimtschi
}
```

Diese Strategie kann jedoch nicht aufgehen. Die Sequenz `\x31\x68` würde auch auf die beiden ASCII-Zeichen "1h" zutreffen. Damit hier wirklich 16-Bit breite Zeichen ("wide characters") und nicht zwei 8-Bit breite Zeichen vom pattern matching erfasst werden, müssen wir uns genauer überlegen, welche Kriterien Texte in den ostasiatischen Zeichensätzen erfüllen können.

Ähnlich wie bei Perl gilt auch hier "There's more than one way to do it". Koreanisch kann mit drei verschiedenen Zeichensätzen verarbeitet werden. Neben dem oben erwähnten südkoreanischen Standard KS X 1001:1992 gibt es noch einen sehr ähnlichen Standard namens KPS 9566-97, der in Nordkorea verwendet wird. Neben Unterschieden in der Sortierung des koreanischen Alphabets und in der Behandlung von chinesischen Zeichen in diesem Standard dürfte am Rande interessant sein, dass in KPS 9566-97 die beiden Namen "Kim Il-Sung" und "Kim Jong-Il" als eigene Zeichen eine feste Position einnehmen; ein sprechendes Beispiel für die Allgegenwärtigkeit des Personenkults in Nordkorea, der offensichtlich bis in EDV-Standards hineinreicht. Ausserdem gibt es noch einen von der chinesischen Standardbehörde definierten Zeichensatz, der für die starke koreanische Minderheit in der VR China geschaffen wurde. Der Einfachheit halber beschränken sich die weiteren Ausführungen auf den südkoreanischen Standard. Koreanisch ist übrigens nicht die einzige CJK-Sprache mit so vielen Zeichensätzen zur Auswahl: Chinesisch hat unterschiedliche Standards in der VR China, in Taiwan und in der Sonderwirtschaftszone Hongkong der VR China.

Den zweiten Punkt, den wir über ostasiatische Textdaten wissen müssen, um einen passenden regulären Ausdruck zu formen, betrifft das sogenannte Encoding. Da ASCII eine Untermenge aller CJK-Zeichensätze ist (ansonsten wäre es schwierig, E-Mails oder HTML-Dokumente in diesen Sprachen zu verfassen), gibt es ein Nebeneinander von 8-Bit-Zeichen und 16-Bit-Zeichen. Um vom 8-Bit-Modus in den "Wide-Character"-Mode zu springen, gibt es verschiedene Methoden der Kodierung. Bespielhaft werden hier ISO-2022 und EUC (Extended Unix Code) dargestellt.

ISO-2022 ist für alle CJK-Sprachen definiert, entsprechend heissen die Standards ISO-2022-CN, ISO-2022-TW (Taiwan), ISO-2022-JP und ISO-2022-KR. Ein Text in ISO-2022-KR beginnt mit der Sequenz "<ESC>) C" (in ISO-2022-JP steht an letzter Stelle der Sequenz ein z.B. "(D"), oder in hexadezimaler Notation (KS X 1001:1992) "1B 24 29 43". Der Sprung von 8 auf 16 Bit wird mit dem Zeichen "<SI>" bzw. "<SO>" (0x0F bzw 0xOE) gekennzeichnet. In dieser Kodierungsmethode wird der Kimtschi zur Sequenz "0x1B 0x24 0x29 0x43 0x0E 0x31 0x68 0x44 0x21 0x0F". Hat man das Pech, diesen Text per E-Mail über einen Mailer-Daemon zu bekommmen, der alle Control-Zeichen (wie Escape oder Shift-In und Shift-Out) herausfiltert, ist der Text unlesbar. Menschen, die regelmässig E-Mail aus Ostasien bekommen, kennen dieses Problem, weshalb sich für elektronische Post auch ein anderer Kodierungsmechanismus namens EUC etabliert hat.

Die EUC-Kodierung (entsprechend EUC-JP, EUC-KR etc.) heisst zwar ausgeschrieben "Extended Unix Code", ist aber nicht auf dieses Betriebsystem beschränkt, sondern ist z.B. auch unter MacOS die Kodierung der Wahl. Hier werden einfach verschiedene Code-Sets festgelegt. In EUC-KR ist ein Zeichen von 0x21 bis 0x7E immer ASCII (oder genauer KS-Roman), danach beginnen die doppelt so breiten koreanischen Zeichen. Der Kimtschi wird hier zu "0xB1 0xE8 0xC4 0xA1".

3 Perl hilft: Welche Möglichkeiten bietet Perl dem CJK-Programmierer

Nachdem jetzt das Zusammenspiel von Zeichensatz und Kodierung dærgestellt wurden, gehen wir zurück zu unserem Ausgangsbeispiel, in dem wir die Zeichenfolge "Kim Ch'i" mit einem regulärem Ausdruck gesucht haben. Zunächst definieren wir ein Muster, das auf EUC-KR passt:

```
my $euckr = q(
   [\x00-\x7E] # ASCII bzw. KS-Roman: 8-Bit-breite lateinische Zeichen
      | [\xA1-\xFE][\xA1-\xFE] # KS X 1001:1992 16-Bit-breite koreanische Zeichen
      );
```

Jetzt nutzen wir das non-greedy matching mit `*?` aus, um unsere Zeichenfolge zu finden:

```
print if (/^ (?: $euckr )*? \xB1\xE8 \xC4\xA1 /ox);
```

Mit `(?: $euckr )*?` springen wir jetzt nicht zeichenweise (oder vielmehr, was Perl für zeichenweise hält, eher byte-weise) durch den Text, sondern sozusagen EUC-KR-weise. Damit die hübschen Kommentare in `$euckr` bleiben können, modifizieren wir das Matching mit `x` und mit `o` legen wir fest, dass `$euckr` nicht bei jedem Durchlauf neu kompiliert wird.

Oben wurde ausgeführt, dass wir es bei ostasiatischen Texten mit unterschiedlichen Kodierungsverfahren (*encodings*) zu tun haben. Bei den beiden gebräuchlisten Verfahren ISO-2022 und EUC ist es zum Glück einfach, zwischen beiden Repräsentationen zu wandeln. Der Algorithmus hierfür ist denkbar einfach: P1_ISO und P2_ISO seien die beiden Bytes eines 16-Bit-Zeichen in ISO-Kodierung. In EUC-Kodierung gilt nun P1_EUC = P1_ISO - 128 und P2_EUC = P1_ISO - 128. Die Umwandlung von EUC nach ISO erfolgt entsprechend. In Perl ausgedrückt:

```
 sub iso2euc () {
  my $first_byte = shift;
  my $second_byte = shift;
  map($_ - 128, ($first_byte, $second_byte));
}

 sub euc2iso () {
  my $first_byte = shift;
  my $second_byte = shift;
  map($_ + 128, ($first_byte, $second_byte));
}

 print ("EUC-KR 0xB1 0xE8 0xC4 0xA1 lautet als dezimaler ISO-Kimtschi: ");
@char = &euc2iso(0xB1, 0xE8);
print("$char[0] $char[1] ");
@char = &euc2iso(0xC4, 0xA1);
print("$char[0] $char[1]\n");
```

Um das durch die verschiedenen Kodierungsmethoden und Zeichensätze verursachte Chaos zu umgehen, gibt es heute schon eine Lösung. Der Zeichensatz Unicode (http://www.unicode.org/) wurde geschaffen, um wenigstens die gängigsten Sprachen der Welt in einem Zeichensatz zusammenzufassen. Neben den CJK-Sprachen und den auf lateinischen und kyrillischer Schrift basierenden Sprachen werden Arabisch, Hebräisch und verschiedene indische und süd-ostasiatische Schriften integriert. Unicode hat das Potenzial, viele Probleme zu lösen, die im Moment in Bezug auf die Verarbeitung von CJK-Sprachen bestehen. Die Version 3.0 von Unicode wurde im Februar 2000 verabschiedet und die Tatsache, dass Windows NT und Windows 2000 diesen Zeichensatz intern verarbeiten, wird für die notwendige Verbreitung sorgen (sicher werden sich auch Systeme wie MacOS oder Linux, die Unicode unvollständig oder gar nicht unterstützen, diesem Trend rasch anpassen).

Unicode-Unterstützung in Perl ist geplant und wird wohl in absehbarer Zeit kommen. Es ist wichtig, dass die Unterstützung von Unicode direkt in den Sprachumfang von Perl integriert wird, um endlich funktionierende reguläre Ausdrücke und Kommandos wie `split` und `tr` für *wide-characters* zu haben. “Hacks” wie JPerl (eine gepatchte Version von Perl, die japanische Zeichen in regulären Ausdrücken verarbeiten kann) werden dann wohl der Vergangenheit angehören. Aber auch bei Unicode stellt sich die Frage der Konvertierung. Der Textkorpus, der z.B. durch E-Mails und Web-Seiten zur Verfügung steht, ist eben heutzutage nicht in Unicode verfasst.

Abhilfe bieten gleich zwei CPAN-Module namens Unicode::Map8 (von Gisle Aas) und Unicode::Map (von Martin Schwartz). In Zukunft wird nur noch das erstgenannte Modul weiterentwickelt werden. Unicode::Map8 stellt Möglichkeiten zur Verfügung, verschiedene 8-Bit-Zeichensätze (wie ASCII) in 16-Bit-Zeichensätze (wie Unicode oder die CJK-Zeichensätze) zu wandeln. Das mag bei europäischen, vor allem skandinavischen Schriften interessant sein, bei CJK-Texten hilft es uns aber in den meisten Fällen nicht. Dazu kommt, dass das Modul zwar eine beeindruckende Anzahl von japanischen Zeichensätzen unterstützt, an koreanischen und chinesischen Zeichensätzen stehen aber jeweils nur einer zur Auswahl. Allerdings ist es leicht möglich, eigene Mapping-Tabellen dem Modul hinzuzufügen. Nicht beachtet wurden hier die

encodings. Die zu wandelnden Zeichen müssen direkt aus dem Zeichensatz kommen. Eine EUC-kodierte Datei ist also in die ISO-Kodierung zu wandeln, der Vorspann wegzulassen und die Shift-In- und Shift-Out-Zeichen zu entfernen, bevor die Zeichen den Methoden des Moduls übergeben werden. Ein Beispiel zur Benutzung von Unicode::Map8 (entnommen der Dokumentation des Moduls):

```
 require Unicode::Map8;
my $no_map = Unicode::Map8->new("ISO646-NO") || die;
my $l1_map = Unicode::Map8->new("latin1")    || die;

my $ustr = $no_map->to16("V}re norske tegn b|r {res\n");
my $lstr = $l1_map->to8($ustr);
print $lstr;

print $no_map->tou("V}re norske tegn b|r {res\n")->utf8;
```

Wie könnte die Zukunft von Perl in Bezug auf die Bearbeitung chinesischer, japanischer und koreanischer Texte aussehen? Es ist ratsam, sich hier Inspiration von einer Sprache zu holen, die von Anfang an Unicode-basiert ist und viele Probleme schon gelöst hat: Java. In Java ist die Wandlung von EUC-JP nach Unicode ein Kinderspiel:

```
 File i = new File("input");
FileInputStream instream = new FileInputStream(i);
BufferedReader in =
  new BufferedReader(new InputStreamReader(instream, "EUC_JP");
```

Java unterstützt neben allen gängigen Zeichensätzen für ostasiatische Sprachen auch eine riesige Menge von Kodierungsmethoden direkt (neben EUC- und ISO-Kodierung z.B. auch Big Five für Chinesisch und Shift-JIS für Japanisch). Ein verbessertes Modul Unicode::Map8, verbunden mit einem Umstieg auf Unicode als Perl-Zeichensatz könnte diesen Komfort von Java auch in die Perl-Welt bringen.

Ein weiterer Punkt auf der Wunschliste des Perl-CJK-Programmierers wäre eine Ergänzung der Regular-Expression-Engine. Der Editor Emacs bietet bereits eine sinnvolle Auswahl an Meta-Zeichen für reguläre Ausdrücke. So matcht `\ch` ein beliebiges koreanisches Zeichen, `\cH` trifft auf ein japanisches Hiragana-Zeichen zu und so weiter.

Perl ist sicher wegen der einfachen Möglichkeiten zur Bearbeitung von Texten die erste Wahl eines Entwicklers, der sich mit der verwirrenden Welt ostasiatischer Sprachen und deren Repräsentation im Computer beschäftigt. Trotzdem zeigt der Vergleich mit Java, dass Perl hier noch besser werden kann.

Ereignisgesteuert programmieren mit Event

Jochen Stenzel
perl@jochen.stenzel.de

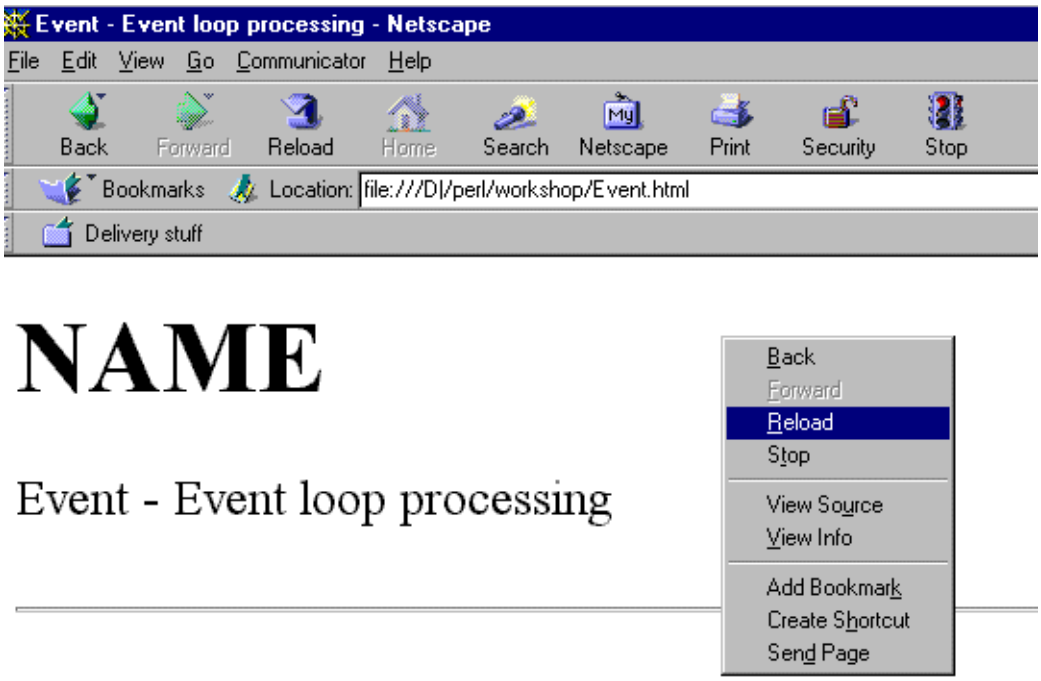
22. Februar 2000

Inhaltsverzeichnis

1. Einführung

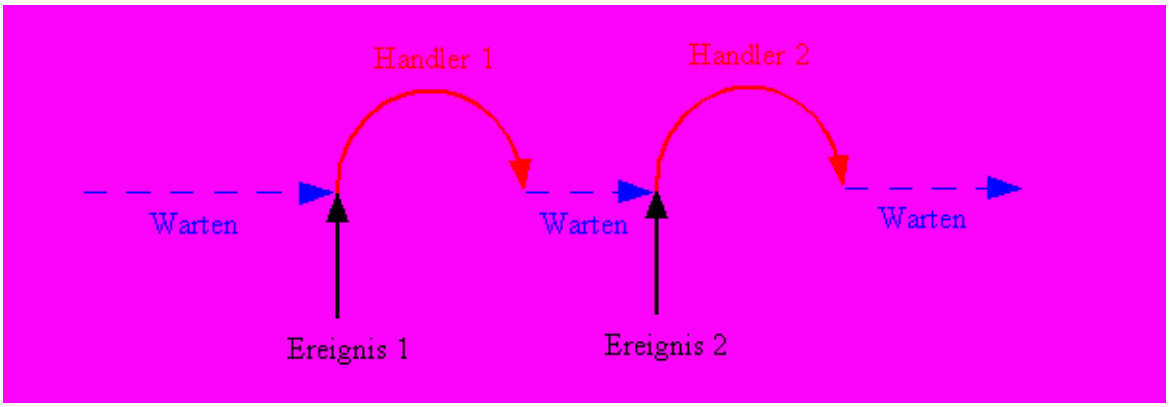
Jeder erlebt täglich Programme wie dieses:

Wenn ich irgendwo klicke, geschieht etwas. Klicke ich an einer anderen Stelle, geschieht (üblicherweise) etwas anderes. Ist mein Programm ein Browser und trifft eine angeforderte Webseite von einem Server ein,



kann der Browser sie entgegennehmen und sie entsprechend darstellen, bleibt aber währenddessen so bedienbar, daß ich den Vorgang auch jederzeit abbrechen kann.

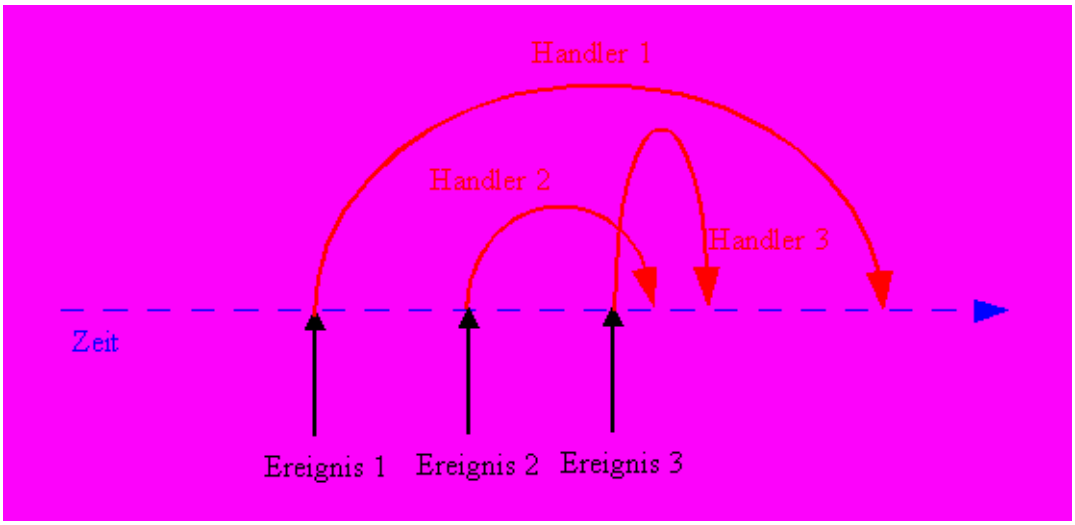
Klicken, Servernachrichten und Menüeingaben sind *Ereignisse*. Das Grundprinzip eines solchen Programms besteht darin, in Abhängigkeit von Ereignissen zu arbeiten. Und egal, wann und in welcher Frequenz Ereignisse eintreten, es wird immer die entsprechende Bearbeitungsfunktion dafür ausgeführt. Man könnte das so darstellen:

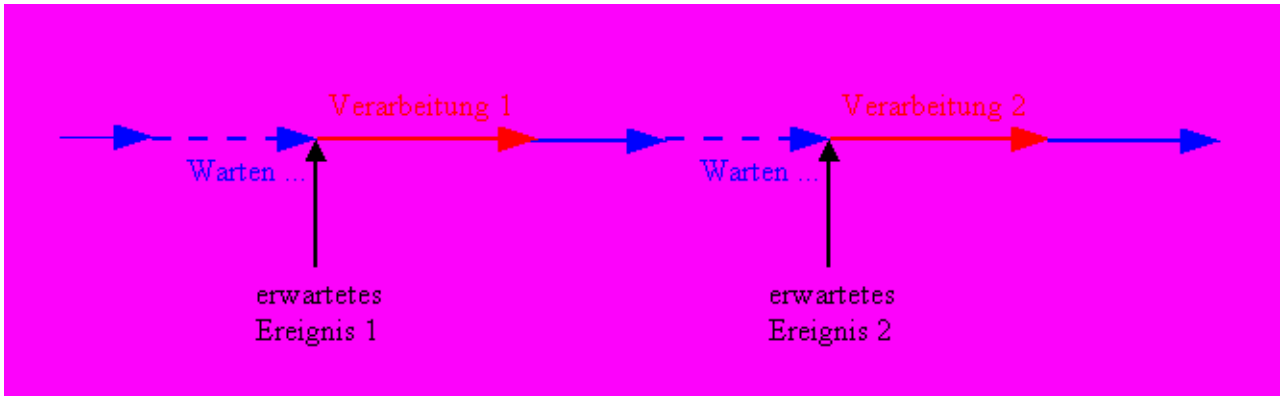


Gute grafische Oberflächen arbeiten darüber hinaus üblicherweise auch noch *asynchron*: die Bearbeitung früher auftretender Ereignisse kann später Ergebnisse liefern als die erst danach festgestellter Events.

> Wenn ich also eine Website anfordere und danach mit demselben Programm meine Mails abhole, können die Mails unter Umständen noch vor der Website eintreffen.

Im Gegensatz dazu arbeiten Programme ohne grafisches Interface traditionell meist *synchron* und reagieren auf Ereignisse nur dann, wenn sie darauf vorbereitet sind:


Handler 1
Handler 2
Handler 3
Zeit
Ereignis 1
Ereignis 2
Ereignis 3


Verarbeitung 1
Verarbeitung 2
Warten ...
Warten ...
erwartetes
Ereignis 1
erwartetes
Ereignis 2

Die Reaktion auf das erste registrierte Ereignis wird immer vor der auf das zweite erfolgen. (Unter Perl und C verletzen Signalhandler dieses Schema in begrenztem Rahmen).

> Eine Shell (ohne Job Control) kann nicht auf Eingaben reagieren, wahrend sie ein Kommando abarbeitet; bietet sie dagegen einen Prompt an, kann sie bis zum Empfang eines neuen Kommandos nichts weiter tun als auf den Anwender zu warten.

Die Beschränkung des ereignisgesteuerten, asynchronen Ansatzes auf grafische Oberflächen ist jedoch keineswegs zwangsläufig (auch wenn umgekehrt synchrone GUI's nur schwer vorstellbar sind).

> Eine asynchrone Shell würde es dem Anwender gestatten, dem ersten Kommando auf Wunsch bereits ein zweites folgen zu lassen, während der erste Auftrag noch lauft. Die Shell würde beide Auftrage parallel bearbeiten und das zuerst verfügbare Ergebnis unabhängig vom Auftragseingang zuerst anbieten.

Der ereignisgesteuerte Ansatz ist immer dann eine Erwägung wert, wenn das Programm mehrere „Handlungsstränge" hat, die nicht streng aufeinander aufbauen, die bestimmter Auslöser bedürfen und die (ganz oder in geeigneten Teilen) schnell genug durchlaufen werden können, daß sie sich gegenseitig nicht blockieren.

> Unabhängige Tasks könnten z.B. das Ausführen von Berechnungen sein oder
>
> - das Vorbereiten bestimmter Daten, um sie bei einer Anforderung ohne Verzögerung liefern zu können; - die Anzeige von Arbeitsfortschritten, während gleichzeitig an der Aufgabe weitergearbeitet wird; - die Annahme von Eingaben; - der Datenempfang von anderen Prozessen; - die Koordination mehrerer IPC-Verbindungen (IRC); - Signalverarbeitung; - das Überwachen von Terminen; - ...

2. Implementierungsansätze

2.1. Asynchrone Programme

Für die Realisierung asynchroner Programme gibt es mehrere Möglichkeiten. Sehr beliebt sind Mehrprozeßsysteme auf der Basis von `fork()` (das übliche Servermodell) oder Threads. Als dritte Variante gesellt sich die *Ereignissteuerung* dazu. Darunter soll von nun an ein System verstanden werden, das eintretende Ereignisse innerhalb eines Prozesses auf der Basis eines *Loops* bearbeitet. Jedes dieser Systeme hat Vor- und Nachteile:

Methode	Kosten	Datenaustausch	Parallelität	Bemerkungen
fork()	relativ hoch (systemabhängig)	schwierig	(theoretisch) echt	System begrenzt Prozeßanzahl
Threads	relativ niedrig	unter Umständen *erheblicher* Synchronisationsaufwand	(theoretisch) echt	mit Perl 5.005 nicht wirklich einsetzbar
Ereignissteuerung	niedrig	einfach	Serialisierung	langlaufende Antworten müssen zerlegt oder delegiert werden

Allen Ansätzen ist gemeinsam, daß das Gesamtproblem in „Portionen" aufgeteilt werden muß - wenn auch auf verschiedene Weise.

Ereignissteuerung erweist sich in diesem Vergleich als ernstzunehmende Alternative!

2.2. Ereignissteuerung unter Perl

Ereignissteuerung in rudimentärer Form bietet `%SIG`. Diese Schnittstelle ist sehr einfach aufgebaut. Ihre Einsatzmöglichkeiten sind unter unserem Aspekt entsprechend beschränkt. Außerdem kann sie lediglich mit Signalen umgehen.

Mit `select()` bringt Perl aber bereits eine echte Basisfunktion für Ereignissteuerung mit. Die Funktion übergibt die Kontrolle solange an das Betriebssystem, bis eins der beim Aufruf angegebenen Handles Bereitschaft zum Lesen oder Schreiben oder auch einen Fehler signalisiert. Zusätzlich kann ein Timeout angegeben werden. Handles und Auslösebedingungen werden über Vektoren festgelegt, das Modul **IO::Select** vereinfacht die Handhabung (und die Lesbarkeit des Codes) erheblich.

IO::Select eignet sich jedoch nur für sehr überschaubare Anwendungsfälle und ist auf Handles und *einen* Timeout beschränkt. Andere Ereignistypen lassen sich damit nicht abfangen. Auch wenn man durchaus universellere Eventloops auf der Basis dieses Moduls programmieren kann, wird man dabei doch schnell an seine Grenzen stoßen.

Dieser Aufwand ist jedoch gar nicht nötig: **Event von J. N. Pritikin (CPAN-ID JPRIT)** bietet einen sehr mächtigen, schnellen, flexiblen und skalierbaren Loop mit einem relativ einfachen Interface für *verschiedenste* Ereignistypen an, mit dem man schnell Erfolgserlebnisse hat. **Event**-Code ist darüber hinaus gut lesbar.

3. Event: Ein Überblick

In ereignisgesteuerten Modellen finden alle für die Ereignisbehandlung wesentlichen Vorgänge im gleichen Prozeß statt. Sie beruhen darum auf einem *Eventloop* – einer zentralen Steuerfunktion, die alle anderen Abläufe einbettet und in geeigneter Weise serialisiert. Dabei sind zwei Aufgaben immer wieder neu zu erledigen: Ereignisse müssen erkannt und die damit verknüpften Handler aufgerufen werden.

Die Serialisierung der Ereignishandler wird in **Event** wie allgemein üblich über eine *Queue* realisiert. Zur Ereigni*serkennung* bedient sich das Modul spezieller Objekte, die Watcher genannt werden.

3.1. Das Watcher-Konzept

Event arbeitet objektorientiert. Seine Haupt-Objekte sind sogenannte *Watcher* - Wächter also, die auf das Eintreffen erwarteter Ereignisse vorbereitet sind und entsprechende Reaktionen organisieren können.

Diese Wächter sind sehr spezialisiert. Jeder von ihnen ist für die Erkennung eines bestimmten Ereignistyps zuständig. Einige erkennen *IO*-Events, andere das Ablaufen von *Timern*, andere *Signale*, einer ist für die Überwachung von *Variablen* zuständig und einer hat sich aufs Nichtstun verlegt - er wird immer dann hellhörig, wenn das System eine *Pause* macht. Sogar *Kollegen* lassen sich beschatten.

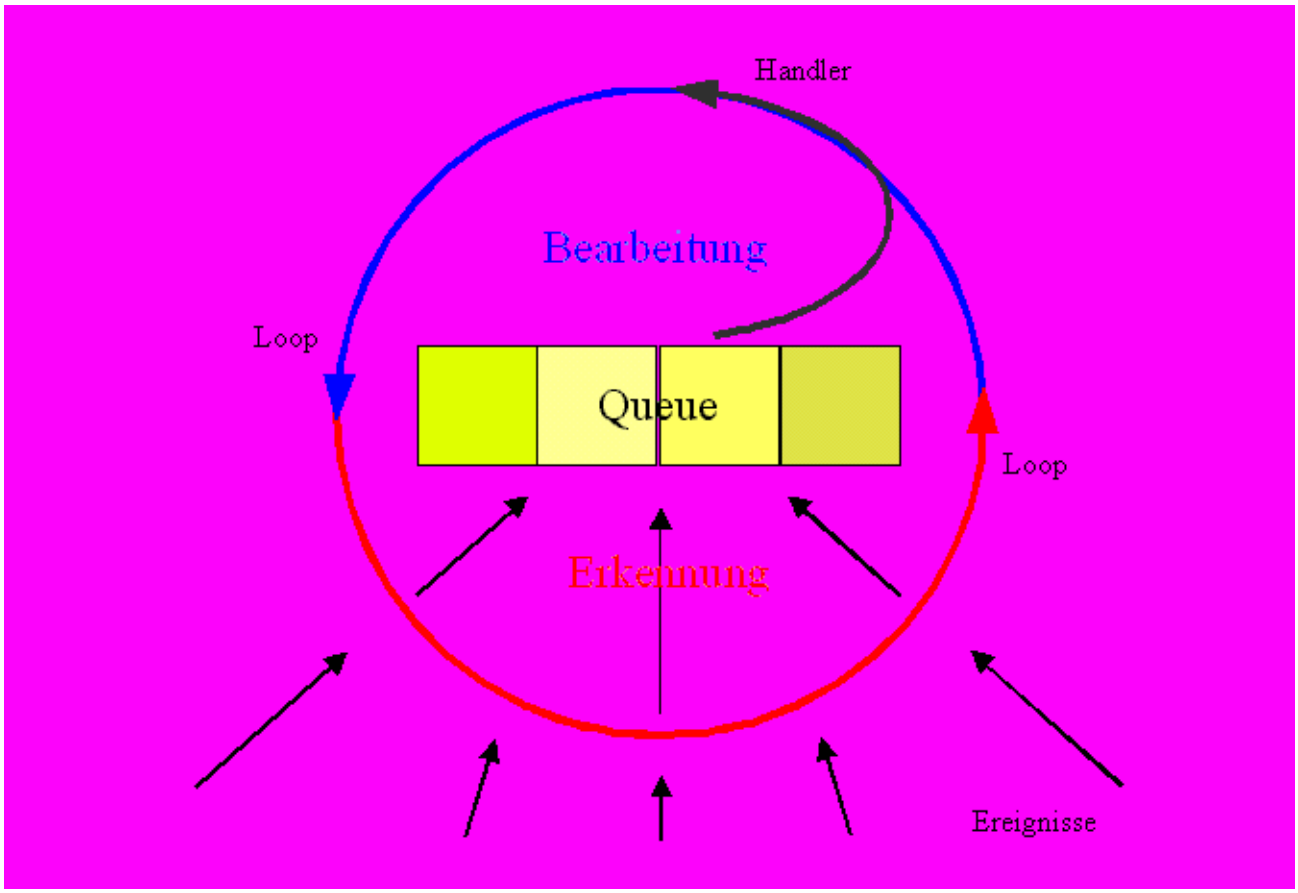
Ereignistyp	**Watcher**
I/O	**io**
Timer	**timer**
Signale	**signal**
kein Ereignis aufgetreten	**idle**
Variable ändert sich	**var**
andere(r) Watcher aktiv	**group**

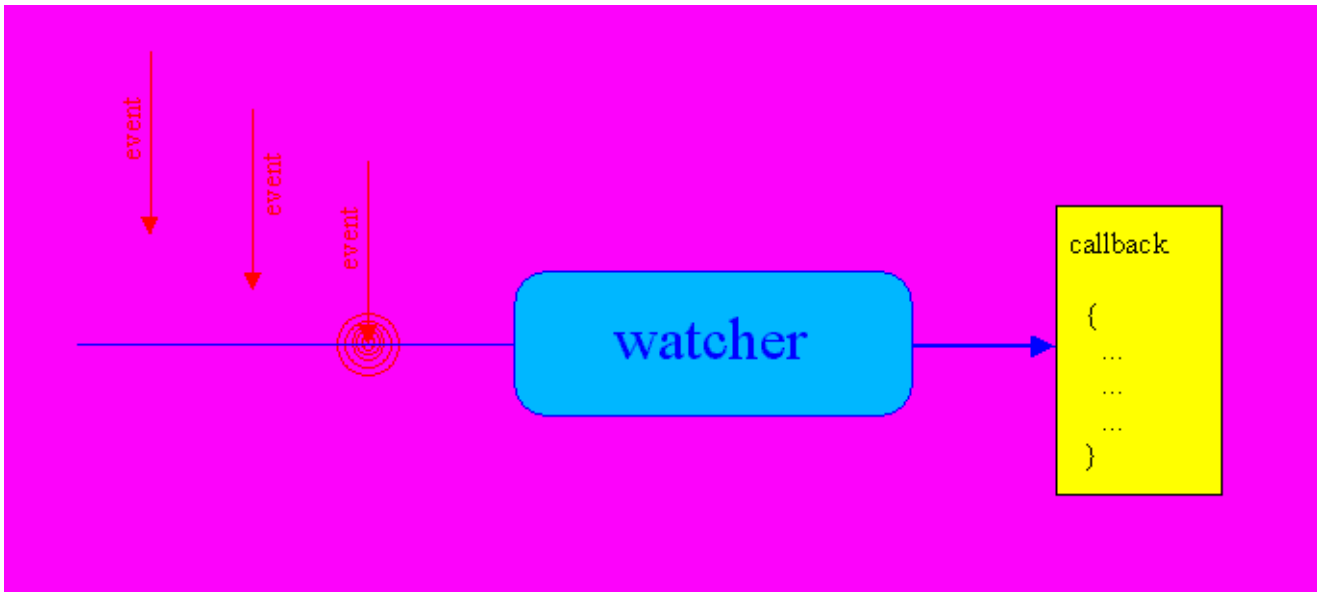
Einige weitere Wächtertypen, etwa für Semaphoren, sind bereits geplant.

Technisch betrachtet sind diese verschiedenen Watcher Objekte verschiedener von **Event** abgeleiteter Klassen (**Event::io**, **Event::timer** usw.). Sie vermitteln zwischen bestimmten *Ereignissen* und *Funktionen*.

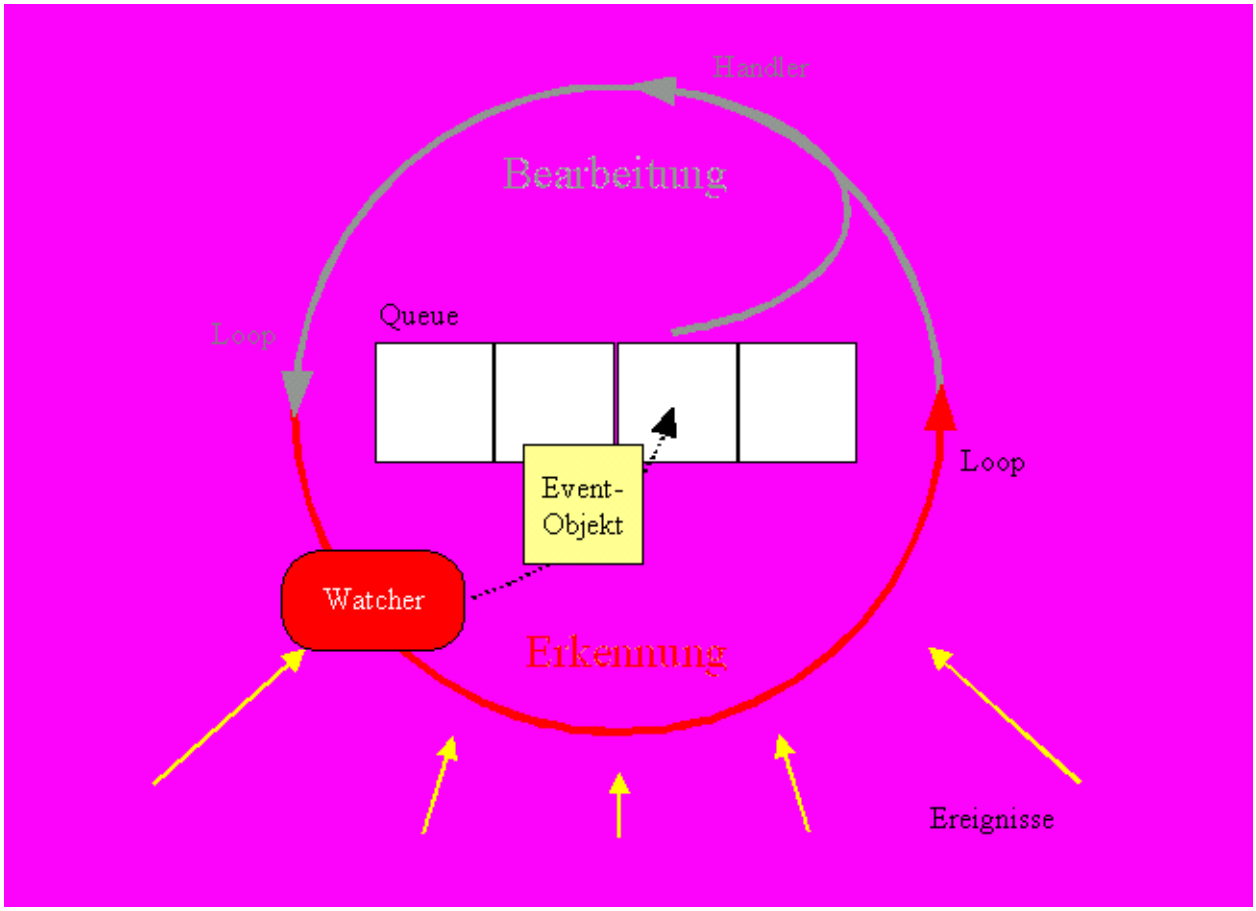
Sobald ein Watcher das Auftreten seines Zielereignisses bemerkt, löst er also *im Prinzip* den Aufruf seiner Callbackfunktion aus.

Der tatsächliche Ablauf ist allerdings etwas komplexer: um die Zusammenarbeit mit anderen Watchern zu gewährleisten, bedient sich **Event** bei der Vermittlung zwischen Ereignis und Handler eines Zwi-


Handler
Bearbeitung
Loop
Queue
Loop
Erkennung
Ereignisse


event
event
event
watcher
callback
{
...
...
...
}

schenschritts. Nach der Erkennung des Events generiert ein Watcher zunächst ein *neues Objekt* der Klasse `Event::Event` (bzw. `Event::Event::Io`) und hinterlegt es in der Queue. (Ich werde von nun an ausschließlich den Klassennamen `Event::Event` verwenden, um Verwirrung zu vermeiden.) Dieses Objekt repräsentiert einen Arbeitsauftrag und enthält alle Informationen zum aktuellen Ereignis. Für den Watcher ist der Fall damit erledigt, und er kann sich sofort wieder der Erkennung weiterer Events widmen.



Das erzeugte `Event::Event`-Objekt dagegen bleibt in der Queue, bis es in der Abarbeitungsphase des Loops abgeholt werden kann. Der Loop ruft dann den darin enthaltenen Callback als Eventhandler auf.

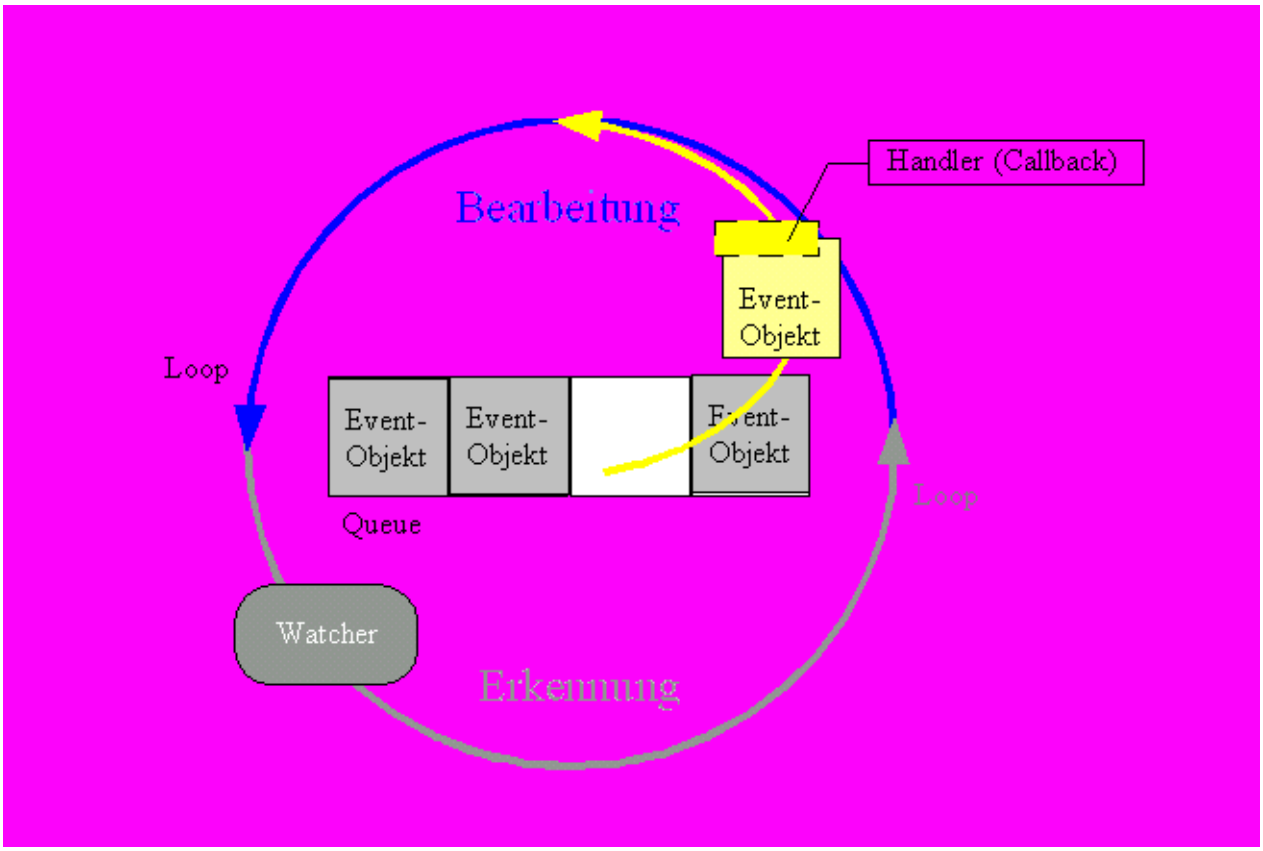
Nach der Ausführung des Auftrages wird das Auftragsobjekt durch den Loop zerstört.

Ein in der Queue hinterlegter Auftrag wird nicht mehr durch seinen „Vater"-Watcher beeinflußt. Das `Event::Event`-Objekt enthält lediglich eine Referenz auf diesen Watcher. Zu einem gegebenen Zeitpunkt können sich daher auch mehrere durch denselben Watcher erzeugte `Event::Event`-Objekte in der Queue befinden, ihre Anzahl läßt sich mit der Watcher-Methode `pending()` ermitteln:

```
# get the still pending orders
@pendingOrders=$watcher->pending;
# How many still unhandled tasks did the watcher produce?
print $watcher->desc, ": ", scalar(@pendingOrders), " tasks\n";
```

`pending()` liefert in skalarem Kontext einen wahren Wert, wenn durch den Watcher generierte Aufträge in der Queue stehen, im Listenkontext liefert sie sogar die entsprechenden Objekte zurück.

Wenn man die Betrachtungen zu **Event** an dieser Stelle zusammenfaßt, kann man bereits drei wesentliche Teile des Modells erkennen: *Watcher* für die Erkennung von Ereignissen, (in `Event::Event`-Objekten


Handler (Callback)
Bearbeitung
Event-
Objekt
Loop
Event-
Objekt
Event-
Objekt
Event-
Objekt
Loop
Queue
Watcher
Erkennung

verpackte) *Callbacks* für ihre Bearbeitung und den *Loop*, der den Ablauf mit Hilfe einer Queue koordiniert. Watcher, Loop und Callbacks sind darum auch die Basiselemente der Event-Programmierung.

3.2. Einen Watcher erzeugen

Ein **Event**-Loop ohne Watcher wäre eine leere Endlosschleife und würde daher sofort beendet werden. Darum erzeugt man noch vor dem Start des Loops mindestens einen Watcher:

```
# I/O-Watcher fur STDIN installieren, der bei
# Eingaben callback() aufruft
Event->io(fd=>\*STDIN, cb=>\&callback);
```

Die Konstruktoren der einzelnen Watchertypen sind nach dem jeweiligen Typ benannt, Watcher anderer Typen werden also durch analoge Aufrufe erzeugt (`Event->timer()`, `Event->var()` usf.).

Die Eigenschaften eines Watchers bestimmen seine *Objektattribute*. Die an den Konstruktor übergebenen Parameter sind daher schlicht und einfach Attributeinstellungen, die das Verhalten des Watchers ganz nach Wunsch konfigurieren. Üblicherweise nutzt man dabei nur das jeweils interessante Subset der verfügbaren Attribute und verläßt sich in den restlichen Fällen auf die eingebauten Defaults. Im Beispiel oben wird ein I/O-Watcher erzeugt, der bei Ereignissen am Handle `STDIN` eine Funktion `callback()` aufrufen soll.

Die Anzahl erzeugbarer Watcher ist nicht begrenzt.

Hier noch ein anderes Beispiel, das einen Timer registriert:

```
# Wecker installieren, der nach 5 Minuten
# einmal an die fertige Pizza erinnnert.
Event->timer(
             repeat   => 0,
             interval => 300,
             cb       => sub {warn "Herd abstellen!\n"},
            );
```

Die in den Beispielen erzeugten Watcher werden sofort aktiv und warten auf Ereignisse; damit sie jedoch wirklich darauf reagieren können, muß der Eventloop gestartet werden.

3.3. Den Loop starten

Nachdem mindestens ein Watcher registriert ist, kann man den *Eventloop* starten. Dies geschieht durch die naheliegende Anweisung `loop()`:

```
Event::loop;
```

Und damit läuft das Script ereignisgesteuert! Der lineare Programmablauf wird verlassen. Alle vielleicht noch folgenden Anweisungen kommen erst dann zum Zuge, wenn der gestartete Loop wieder beendet wird.

Der gesamte weitere Programmablauf wird jetzt durch die installierten Watcher, ihre Callbacks und natürlich durch die auftretenden Ereignisse bestimmt.

Den laufenden Loop kann man jederzeit durch Aufruf der Klassenmethode `unloop()` beenden:

```
Event::unloop();
```

Dieser Aufruf führt zum Abbruch der Ereignisschleife, berührt die installierten Watcher aber *nicht*. (Man könnte die Ereignisverarbeitung durch einen neuen `loop()`-Aufruf also wieder aufnehmen.) Aus verständlichen Gründen muß man ihn in einem Watcher-Callback implementieren.

3.4. Ein vollständiges Beispiel

Um den Einsatz von **Event** noch einmal im Zusammenhang zu demonstrieren, hier ein komplettes Beispiel.

```
# set pragma
use strict;

# load module
use Event qw(loop unloop);

# install initial watcher
Event->io(fd=>\*STDIN, poll=>'r', cb=>\&io);

# start loop
loop;


# FUNCTIONS ###################################################################

# io handler
sub io
 {
  # read line
  my $cmd=<STDIN>;
  chomp $cmd;

  # stop processing, if necessary
  unloop, return if uc($cmd) eq 'QUIT';

  # get alarm data
  warn("[Error] Wrong format in \"$cmd\".\n"), return unless $cmd=~/^(\d+)\s+(.+)/;
  my ($period, $msg)=($1, $2);

  # install a new one shot alarm timer
  Event->timer(
               prio   => 2,                             # before IO;
               at     => time+$period,                  # set alarm;
               cb     => sub                            # callback;
                         {
                              warn "[Alarm] $msg\n";        # inform
                          $_[0]->w->cancel;             # clean up
                         },
               repeat => 0,                             # one shot;
              );

  # display a message
  warn '[Info] Your timer "', $msg, '" is running.', "\n";
 }
```

Dieses Script realisiert einen einfachen Reminder: es nimmt Termine entgegen und erinnert zu gegebener Zeit einmal daran.

> Man notiert also beispielsweise 17:20, noch ganz in sein Projekt vertieft, den blitzartig auftauchenden Gedanken an die Heimfahrt:

```
3600 Letzte S-Bahn!
[Info] Your timer "Letzte S-Bahn!" is running.
```

Wenige Minuten später startet man in Gedanken die Kaffemaschine und hält vorsichtshalber fest:

```
300 Kaffee
[Info] Your timer "Kaffee" is running.
```

Jetzt ist der Abend gerettet. 5 Minuten später erscheint die Meldung

```
[Alarm] Kaffee
```

, und pünktlich 18:20 wird man mit

```
[Alarm] Letzte S-Bahn!
```

an den Aufbruch erinnert.

Nicht schwierig, oder?

4. Event im Detail

Dieses Kapitel stellt die Einzelheiten der Event-Programierung vor, die sich in der Darstellung der grundlegenden **Event**-Konzepte naturgemäß nicht alle unterbringen ließen.

4.1. Watcher-Attribute

Die Eigenschaften eines Watchers werden durch seine *Objektattribute* bestimmt. Man setzt sie im Konstruktoraufruf oder stellt sie später nach Bedarf ein. Einige *Basisattribute* sind dabei für Watcher aller Typen gleich.

Seit Version 0.60 steht darüber hinaus das Basisattribut `data` zur Verfügung, das nicht der Konfiguration, sondern dem Hinterlegen eigener Daten im Watcherobjekt dient.

Jeder Watchertyp besitzt neben den allgemeinen Basisattributen noch weitere, spezielle Einstellungsmöglichkeiten.

Attribute werden im Konstruktor eines Watchers durch gleichnamige benannte Parameter initialisiert:

```
# register a timer
Event->timer(interval=>32, hard=>1, cb=>\&callback);
```

Attribute können aber auch während der gesamten Lebensdauer eines Watchers über gleichnamige Methoden abgefragt und gegebenenfalls verändert werden, beispielsweise so:

```
# modify a watchers description
$timerWatcher->desc("Ist es wirklich schon so weit?");

# report callback runtime
print "Letzter Callback-Start um ",
      POSIX::strftime("%c\n", localtime($w->cbtime));
```

Attribut	Bedeutung
voller Zugriff:	
cb	Callback-Routine, die beim Auftreten des Ereignisses aufgerufen werden soll
debug	Schalter für die Spurverfolgung
desc	eine Beschreibung des Watchers, über dieses Attribut läßt er sich später gut wiederfinden
max_cb_tm	Zeitrahmen für Callbacks, länger laufende Callbacks werden abgebrochen
prio	Priorität
reentrant	ein Flag, das die verschachtelte Ausführung des Callbacks erlaubt
repeat	legt fest, ob der Watcher nach dem ersten auftretenden Ereignis auf dem Posten bleiben soll
lesender Zugriff:	
cbtime	letzter Aufruf des Callbacks
is_running	Anzahl gerade laufender Callbacks
nur im Konstruktor:	
async	legt fest, daß Ereignisse *sofort* (ohne Rücksicht auf andere Watcher) zu bearbeiten sind, wird durch `prio` überschrieben
parked	verhindert, daß der neue Watcher aktiv wird, er reagiert also noch *nicht* auf Ereignisse
nice	Priorität als Offset zum Default, wird durch `prio` und `async` überschrieben

Tabelle 1: Basisattribute

4.2. Objektverwaltung

Die bisher gezeigten Konstruktoraufrufe für einen Watcher wirkten vielleicht etwas ungewöhnlich. Normalerweise läßt man sich von einem Konstruktor ja das erzeugte Objekt liefern:

```
my $watcher=Event->io(fd=>\*STDIN, cb=>\&callback);
```

Stattdessen sieht man in **Event**-Scripts oft die beschriebene vereinfachte Form:

```
Event->io(fd=>\*STDIN, cb=>\&callback);
```

Man kann hier auf die Rückgabe des Objekts und dessen eigene Verwaltung verzichten, weil **Event** alle Watcher bereits *intern* verwaltet. Wenn man später selbst wieder explizit Zugriff auf einen bestimmten Watcher benötigt, kann man ihn jederzeit durch Klassenmethoden wiederfinden:

Die interne, automatische Verwaltung eines Watchers durch **Event** hat den Nebeneffekt, daß der Referenzzähler des Watcherobjekts durch modulinterne Operationen beeinflußt wird. Darum bleibt in

```
{
 my $watcher=Event->io(fd=>\*STDIN, parked=>1);
}
```

der neue Watcher auch *nach* dem Verlassen des Blocks am Leben und aktiv.

Umgekehrt *kann* man natürlich ein vom Konstruktor geliefertes Watcherobjekt durchaus auch in eigenen Datenstrukturen verwalten. Man muß dann „nur“ dafür sorgen, daß man auf ein solches Objekt nicht mehr verändernd zugreift, falls **Event** es (mit der Methode *cancel()*) bereits aus seiner internen Liste gestrichen hat, ansonsten wird eine Exception ausgelöst. Der aktuelle Watcher-Zustand läßt sich mit der Methode `is_cancelled()` überprüfen.

Der folgende Abschnitt geht näher auf die einzelnen Zustände eines Watchers ein.

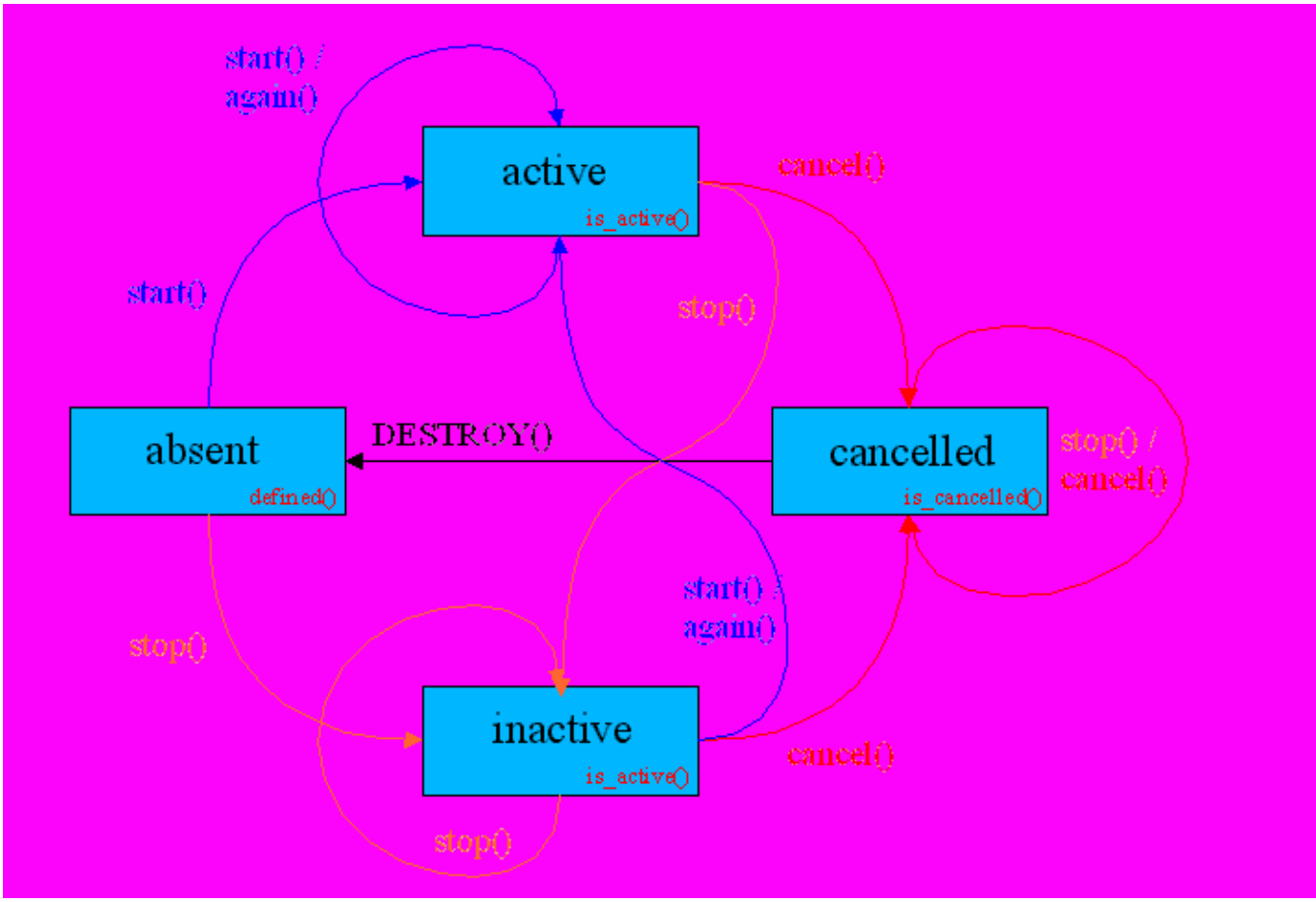
Attribut	Beschreibung
io:	
fd	das überwachte Handle
poll	legt genauer fest, welche Events von Interesse sind: lesende oder schreibende Zugriffe, Fehler oder Timeouts (bzw. beliebige Kombinationen)
timeout	Zeit, nach deren Ablauf der Callback spätestens ausgeführt wird (auch ohne Ereignis)
timeout_cb	läuft ein gesetzter Timeout ab, wird der hier hinterlegte Code anstelle des „normalen“ Callbacks aufgerufen (optional)
hard	legt bei gesetztem Timeout fest, ob die Zeit am Anfang oder Ende eines Callback-Aufrufs neu zu laufen beginnt
timer:	
at	Zielzeitpunkt
interval	Ablaufzeit
hard	legt bei Vorgabe eines Intervalls fest, ob die Zeit am Anfang oder Ende eines Callback-Aufrufs neu zu laufen beginnt
signal:	
signal	das überwachte Signal
idle:	
max	Zeit, nach deren Ablauf der Callback spätestens ausgeführt wird (auch ohne Ereignis)
min	Mindestzeit zwischen zwei Callback-Aufrufen (selbst bei zwischenzeitlichen Events)
var:	
var	eine Referenz auf die beobachtete Variable
poll	legt genauer fest, ob lesende oder / und schreibende Zugriffe überwacht werden sollen
group:	
timeout	Zeit, nach deren Ablauf der Callback spätestens ausgeführt wird (auch ohne Ereignis)
add	verweist auf ein Watcher-Objekt, dessen Aktivität das erwartete Ereignis darstellt (man kann das Attribut beliebig oft belegen, als Event gilt dann die *erste* Aktivität eines beliebigen Watchers der so gebildeten Gruppe). Mit der Methode `del()` kann man einen beobachteten Watcher wieder aus der Gruppe entfernen.

Tabelle 2: Spezifische Attribute

Methode	Beschreibung
all_watchers()	liefert alle registrierten Watcher
all_running()	gibt alle Watcher zurück, deren Callbacks gerade laufen
all_idle()	bietet eine Liste von `idle`-Watchern an, die aufgrund höherpriorisierter Ereignisse gerade auf ihre Ausführung warten müssen

4.3. Der Lebenszyklus eines Watchers

Ein Watcher durchläuft während seines Lebens mehrere *Zustände*, in denen er sich verschieden verhält. Er läßt sich also nicht nur erzeugen und freigeben, sondern bei Bedarf beispielsweise auch deaktivieren. Die einzelnen Zustandsübergänge können *explizit durch Methodenaufrufe* des Watchers-Objektes erzwungen werden, oder sie finden *implizit beim Vorliegen entsprechender Voraussetzungen* statt. Der aktuelle Zustand läßt sich durch entsprechende Funktionen oder Methoden erfragen.



Der Zustand eines Watchers spiegelt zum einen seine *Aktivität*, zum anderen seine *Registrierung* wieder. Die Aktivität drückt aus, ob er auf Ereignisse wartet und sie erkennt. Die Registrierung beschreibt, ob der Loop den Watcher kennt, so daß dieser entsprechende Aufträge in der Queue ablegen kann.

Zustände

ABSENT: Der Ausgangszustand jeder Variablen. Der Watcher wurde noch nicht erzeugt, oder das Watcher-Objekt wurde bereits wieder freigegeben. Ein solcher Watcher ist weder aktiv noch registriert.

Wie bei jeder Variablen kann man diesen Zustand mit `defined()` erkennen.

ACTIVE: Der Watcher ist aktiv und registriert, kann auftretende Ereignisse also erkennen und entsprechende Aufträge in der Queue ablegen.

`is_active()` liefert im aktiven Zustand einen wahren Wert.

INACTIVE: Der Watcher ist „beurlaubt“. Seine Einstellungen bleiben erhalten, er ignoriert jedoch alle Events und generiert damit auch keine Aufträge. Dem Loop ist er weiter bekannt.

`is_active()` gibt in diesem Zustand folgerichtig einen „falschen“ Wert zurück.

CANCELLED: Der Watcher achtet nicht länger auf Ereignisse und ist nicht länger beim Loop registriert.

Aus diesem Grund kann er auch nicht mehr in die Zustände **ACTIVE** oder **INACTIVE** zurückversetzt werden. Lesende Operationen (etwa zum Abfragen bestimmter Attribute) sind noch möglich, jeder andere Zugriff löst jedoch eine Exception aus.

Mit der Methode `is_cancelled()` läßt sich darum vor einem Zugriff prüfen, ob sich der Watcher im Zustand *CANCELLED* befindet. Er wird nur dann erreicht, wenn man die Verwaltung eines erzeugten Watcher-Objektes nicht **Event** überläßt, sondern das Objekt auch in eigenen Daten referenziert, oder wenn ein Watcher bereits gelöscht wurde, die Queue aber noch einen seiner Aufträge enthält. In diesen Fällen ist also besondere Vorsicht geboten.

Zustandsübergänge

Implizite Übergänge: Generell gilt, daß ein Watcher nur dann den Zustand **ACTIVE** erreichen oder beibehalten kann, wenn seine Attribute einen sinnvollen aktiven Betrieb zulassen.

> Ein var-Watcher ohne Variablenreferenz kann beispielsweise kaum sinnvolle Aufträge generieren.

Sobald eine Operation diese Vorgabe verletzt, wird der Watcher aus **ACTIVE** automatisch in **INACTIVE** versetzt. Aktive Watcher müssen *immer* einen Callback besitzen. Die folgende Tabelle enthält die übrigen Mindestanforderungen an einen aktiven Watcher:

Watchertyp	Mindestanforderungen
io	Timeout gesetzt oder gültiges Handle in `fd` hinterlegt
timer	Timeout gesetzt; wiederholte Ausführung nur mit gültigem Intervall
signal	gültiges Signal in `signal` eingetragen
idle	keine
var	Die Attribute `poll` und `var` müssen gesetzt sein. Die Überwachung schreibgeschützter Variablen wie `$1` ist unzulässig.
group	mindestens ein beobachteter Watcher mit `add` hinterlegt

Das folgende Beispiel demonstriert einen forcierten impliziten Übergang.

```
# deactivate an active watcher implicitly
# (demonstration only!)
my $w=Event->signal(signal=>'INT', cb=>\&cb);
print "Watcher gestartet.\n" if $w->is_active;
$w->signal('FUN');
print "Watcher deaktiviert.\n" unless $w->is_active;
```

Konstruktoraufrufe: Der Parameter `parked` weist den Konstruktor an, das neue Objekt durch einen Aufruf der Methode `stop()` sofort in den Zustand **INACTIVE** zu versetzen. Er wird auch dann eingenommen, wenn der Konstruktor nicht alle für eine Ereignisbearbeitung nötigen Attribute sinnvoll einstellt (s.o.). Standardmäßig ruft ein Konstruktor jedoch die Methode `start()` auf, nach deren Ausführung sich der neue Watcher im Zustand **ACTIVE** wiederfindet.

```
# new and active watcher
print Event->var(var=>\$var, cb=>\&cb)->is_active, "\n";

# similar, but explicitly deactivated
print Event->var(var=>\$var, cb=>\&cb, parked=>1)->is_active, "\n";

# insufficient attributes -> state INACTIVE
Event->io->is_active or die "[BUG] Insufficient watcher attributes!";
```

Das Konstruktorattribut `parked` wurde eingeführt, um das Anlegen von Watcher-Objekten „auf Vorrat“ zu ermöglichen. Das Erzeugen eines neuen Objekts ist nämlich theoretisch aufwendiger als die Reaktivierung eines beurlaubten Watchers, so daß sich in zeitkritischen Anwendungen das vorausschauende

Anlegen eines Pools lohnen kann. Es hängt erfahrungsgemäß allerdings sehr von der konkreten Anwendung ab, ob man aus dieser Strategie letztendlich wirklich Gewinn schlägt.

Deaktivierung: Mit Hilfe der Methode `stop()` kann man einen aktiven Watcher explizit von der Ereigniserkennung freistellen. Sie versetzt den Watcher in den Zustand **INACTIVE**. Bereits in der Queue hinterlegte Aufträge werden durch den Zustandsübergang nicht berührt. Ein deaktivierter Watcher geht durch einen Aufruf der Methode `again()` (oder auch `start()`) wieder in den Zustand **ACTIVE** über (wenn seine Attribute das erlauben).

```
# stop watcher temporarily ...
$w->stop;
...
# and reactivate it
$w->again;
```

Abmeldung: Um einen Watcher ganz aus dem Verkehr zu ziehen, gibt es die Methode `cancel()`. Sie gibt den Watcher aus der Sicht von **Event** als Objekt frei, so daß Perls Verweiszähler erniedrigt werden kann. Gibt es keine weitere Referenz auf den Watcher (in einem noch nicht bearbeiteten Auftrag oder in privaten Daten), kommt es darum umgehend zum Aufruf von `DESTROY()`, und das Objekt hört auf zu existieren. Sollten dagegen weitere Referenzen existieren, befindet es sich anschließend im Zustand **CANCELLED**.

```
# generate a cancelled watcher
my $cw=Event->io(parked=>1);
$cw->cancel;
print $cw->is_cancelled, "\n";
```

Bereits in der Queue hinterlegte Aufträge werden durch den Zustandsübergang nicht berührt.

4.4. Prioritäten

Wenn verschiedene Ereignisse gleichzeitig auftreten, soll das Starten der entsprechenden Handler unter Umständen nicht dem Zufall überlassen bleiben.

> Ein Signal sollte in den meisten Fällen schnell verarbeitet werden, auch wenn zufällig gerade der Timer fur die Mittagspause abläuft.

Mit *Prioritäten* läßt sich recht fein festlegen, wer in einem solchen Fall Vorrang haben und wer sich in der (Warte-)Schlange anstellen soll. **Event** stellt dazu derzeit acht verschiedene Ebenen zur Verfügung - von sofortiger Bearbeitung bis zur Erledigung „bei Gelegenheit“. Jeder Watcher-Typ wird von Haus aus mit einer Standardpriorität ausgestattet.

Ebene	Beschreibung	Default
-1	*asynchrone* Bearbeitung: der Watcher wird sofort aktiviert, die Queue wird umgangen	
0	höchste „reguläre“ Priorität	
1		
2	als Konstante `PRIO_HIGH` hinterlegt	signal
3		
4	als Konstante `PRIO_NORMAL` hinterlegt	idle, io, timer, var
5		
6	niedrigste Priorität	

Es steht natürlich jedem frei, seine eigene Hierarchie zu etablieren. Alle Watcher-Konstruktoren bieten dazu gleich drei Attribute an: `prio` wählt explizit eine bestimmte Priorität aus, `nice` gibt stattdessen einen Offset zur Standardpriorität des Watchers an, und `async` legt fest, daß mit -1 gearbeitet werden soll, die Queue also zu umgehen ist.

```
# a default signal watcher
$sigWatch=Event->signal(parked=>1);
print "Default: ", $sigWatch->prio, "\n";

# watcher with explicit priority setting
$sigWatch=Event->signal(parked=>1, prio=>1);
print "Prio 1: ", $sigWatch->prio, "\n";

# constructor using prio offset
$sigWatch=Event->signal(parked=>1, nice=>-2);
print "Default-2: ", $sigWatch->prio, "\n";

# signals should be served immediately
$sigWatch=Event->signal(parked=>1, async=>1);
print "Asynchron: ", $sigWatch->prio, "\n";
```

Falls man in einem Konstruktor übrigens versehentlich einmal mehrere dieser Prioritätsattribute verwendet, hat `prio` Vorrang vor `async`, und `async` vor `nice`.

Und natürlich läßt sich die Priorität auch jederzeit zur Laufzeit verändern, dann allerdings nur über die `prio()`-Methode (`async` und `nice` stehen nur im Konstruktor zur Verfügung).

```
# load module, import constant
use Event qw(PRIO_HIGH);

# signals should be served immediately
$sigWatch=Event->signal(parked=>1, async=>1);
print "Initial: ", $sigWatch->prio, "\n";

# oops, back to default priority
$sigWatch->prio(PRIO_HIGH);
print "Modified priority: ", $sigWatch->prio, "\n";
```

Bei der Wahl geeigneter Ebenen für verschiedene Watcher darf man nicht aus den Augen verlieren, daß auch die nicht extrem wichtigen Ereignisse letztendlich nach gewisser Zeit bearbeitet sein sollten. Eine Prioritätsebene darf darum nicht nur die *Bedeutung* eines Ereignisses bzw. die *Dringlichkeit* seiner Bearbeitung ausdrücken, sondern muß sich auch an der (wahrscheinlichen) *Häufigkeit* des jeweiligen Ereignisses orientieren. Wenn „wichtige“ Events zu oft auftreten, können sie alle übrigen Watcher blockieren. Optimal ist eine Einstellung, bei der sehr wichtige Ereignisse nur sehr selten stattfinden.

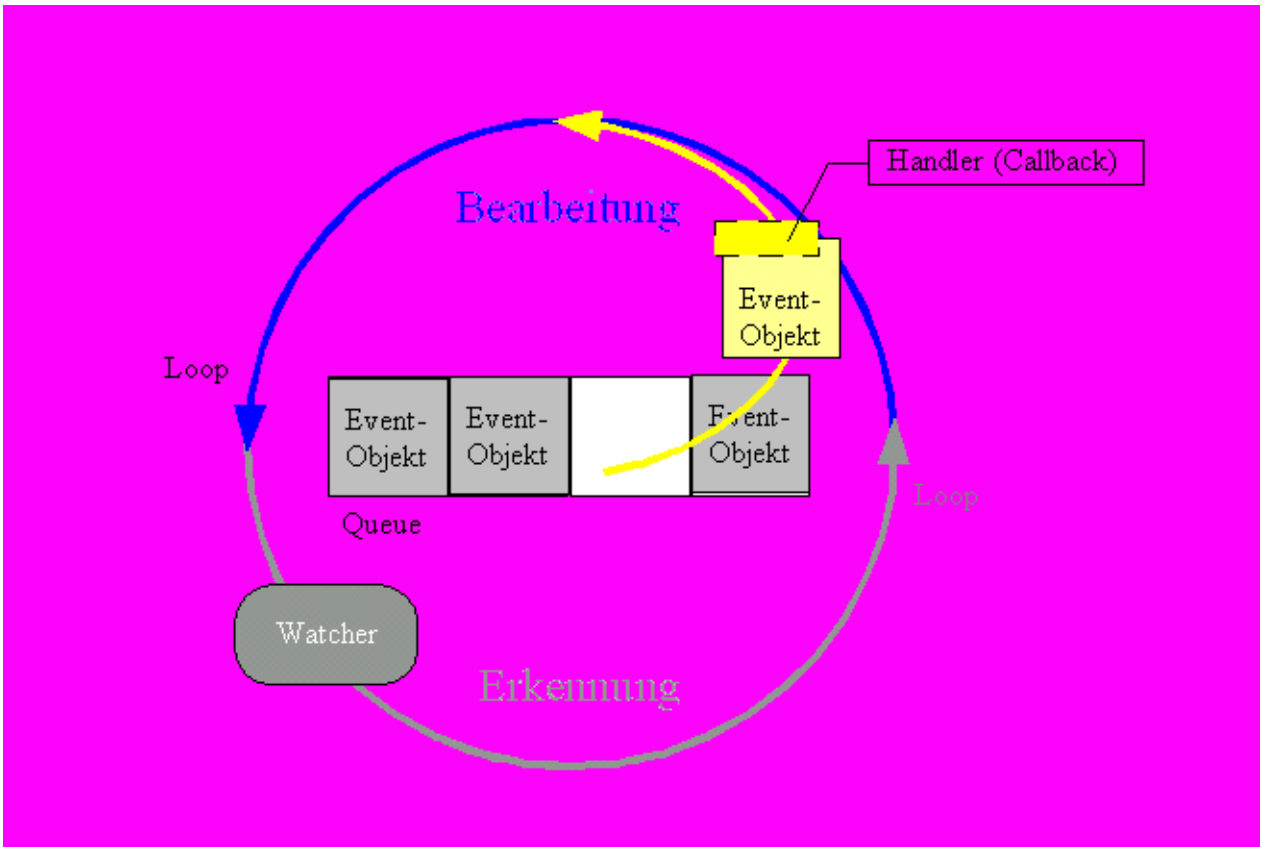
4.5. Beobachtungstrupps

Event erlaubt es ausdrücklich, daß beliebig viele Watcher auf ein bestimmtes Ereignis angesetzt werden. Im gegebenen Fall werden dann *alle* auf der Lauer liegenden Watcher (nacheinander) aktiv.

4.6. Wie schreibe ich einen Callback?

Jeder durch einen Watcher in der Queue hinterlegte Auftrag wird durch ein Objekt der Klasse `Event::Event` repräsentiert. Der Loop führt den Auftrag in der Bearbeitungsphase dann durch einen Aufruf des in diesem Objekt hinterlegten Callbacks aus.

Das verwendete `Event::Event`-Objekt wird dem Callback implizit als erster (und einziger) Parameter übergeben und dient ihm durch zusätzlich darin gespeicherte Informationen als Bindeglied zum Auftreten des auslösenden Ereignisses und zum Watcher, der dieses Ereignis erkannt hat. Nach Abschluß des Callback-Aufrufes wird es zerstört.


Handler (Callback)
Bearbeitung
Event-
Objekt
Loop
Event-
Objekt
Event-
Objekt
Event-
Objekt
Loop
Queue
Watcher
Erkennung

```
# callback taking the Event object
Event->io(..., cb=>sub {my ($event)=@_;});
```

Obwohl derart flüchtig, spielen `Event::Event`-Objekte also dennoch eine zentrale Rolle bei der Bearbeitung eines Ereignisses. Ein Event-Objekt ist ähnlich einem Watcher aufgebaut: es besitzt Attribute, auf die man durch gleichnamige Methoden Zugriff hat. `Event::Event`-Objekt-Attribute können nur gelesen werden.

Attribut	**Beschreibung**
got	ist nur verfügbar, wenn der zugehörige Watcher ein `poll`-Attribut besitzt: dann wird das auslösende Ereignis hier (im `poll`-Format) näher beschrieben
hits	hier hat der Watcher eine laufende Nummer für den erzeugten Auftrag hinterlegt, man erfährt also, wie oft das auslösende Ereignis bereits auftrat
prio	die Watcher-Priorität *zum Zeitpunkt des Events*
w	das Watcher-Objekt

Tabelle 3: Event::Event-Objekt-Attribute

Besonders der hier gebotene Zugriff auf das Watcher-Objekt wird gern genutzt, um dessen Konfiguration im Callback zu modifizieren oder seinen Zustand zu verändern, etwa so:

```
sub callback
 {
  # get Event object
  my ($event)=@_;

  ...
  # cancel the initial watcher, if possible
  $event->w->cancel if $expectedEventsArrived;
  ...
 }
```

Dabei ist allerdings zu beachten, daß der *Watcher* sich nicht mehr im gleichen Zustand wie beim Auftreten des Ereignisses befinden muß. Während ein `Event::Event`-Objekt den Ereigniszustand „einfriert“, arbeitet der damit verbundene Watcher ja weiter; und zwischen Event und Bearbeitung kann unter Umständen viel Zeit vergangen sein. Im Extremfall ist der Watcher schon im Zustand *CANCELLED*, so daß schreibende Zugriffe (etwa zum Ändern der Priorität) eine Exception auslösen würden. Es empfiehlt sich bei (schreibenden) Watcher-Zugriffen im Callback darum immer, zunächst den aktuellen Zustand des Watchers zu ermitteln.

```
# something seems to block us, we should act more often!
$event->w->prio($event->w->prio-1) unless $event->w->is_cancelled;
```

In vielen Fällen wird man allerdings auch ganz ohne Zugriff auf das übergebene `Event::Event`-Objekt auskommen, so daß man es ignorieren kann.

Neben der Parameterschnittstelle gibt es in Callbacks nur noch eins zu beachten: sie sollten vor allem ***schnell*** sein. Während der Callback läuft, sind alle anderen Aufträge und natürlich auch die registrierten Watcher blockiert. Langlaufende Callbacks sollten darum geeignet „verkürzt“ werden. Das kann man durch ihre *Zerlegung in Teilaufgaben* mit Hilfe einer State Machine erreichen: jeder Callback-Aufruf bearbeitet dann beispielsweise einen Zustand. Alternativ dazu kann man Aufgaben an parallel laufende Prozesse *delegieren*, die im eigenen (Threads oder Tochterprozesse) oder einem fremden Prozeßraum (auf einem Server) laufen. Als dritte Methode bietet sich *Kooperation* an: man kann mit Hilfe der Klassenmethode `sweep()` aus einem Callback heraus die Erkennung und Bearbeitung inzwischen aufgelaufener Ereignisse veranlassen. Auf diese und andere Klassenmethoden zum Loop-Management geht der folgende Abschnitt ein.

4.7. Loop-Management

Neben Watchern und Callbacks ist der Eventloop selbst das dritte tragende Element in **Event**. Er wird durch Klassenmethoden gesteuert, von denen man in vielen Fällen sicher nur `loop()` und `unloop()` benötigt.

Ein interessanter Aspekt des Event-Designs ist die Tatsache, daß sich Loops auch verschachteln lassen. Die Methode `loop()` kann also auch innerhalb eines Callbacks nochmals aufgerufen werden und erzeugt dann eine neue Ebene, in der allerdings alle registrierten Watcher weiterarbeiten.

Methode	Beschreibung
loop()	startet einen (weiteren) Loop. Loops beenden sich automatisch, falls nicht mindestens ein aktiver Watcher registriert ist.
unloop()	beendet den *innersten* Loop, die angemeldeten Watcher bleiben davon unberührt
unloop_all()	beendet *alle* Loops (ohne Einfluß auf die Watcher)
sweep()	ermöglicht es einem Callback, inzwischen aufgelaufene Ereignisse erkennen und bearbeiten zu lassen (die Ereignisverarbeitung läßt sich durch Übergabe einer Mindestpriorität einschränken). Die Methode kehrt anschließend sofort zurück.

Tabelle 4: Loop-Steuerung

Da sich ein Loop ohne aktive Watcher selbst beendet, benutzt man alternativ zu `unloop()` (bzw. genauer `unloop_all()`) auch gern die folgende kompakte Konstruktion zum Aufräumen:

```
# stop all loops AND deregister all watchers
$_->cancel foreach Event::all_watchers;
```

5. Fortgeschrittene Anwendung

5.1. Watching Watchers

J. N. Pritikin hat **Event** mit umfangreichen Möglichkeiten für Debugging, Fehleranalyse und Tuning ausgestattet. So bietet das Modul bereits in der Standardinstallation eine Klassenvariable `$Event::DebugLevel` und für jeden Watcher ein `debug`-Attribut, über die sich Trace-Messages aktivieren lassen. Bei einer Installation mit `-DEVENT_MEMORY_DEBUG` steht zusätzlich eine Klassenmethode zur Verfügung, mit deren Hilfe man sich jederzeit über die gerade installierte Watcherzahl informieren kann.

```
# display installed watchers
warn "[Trace] Watchers: ", join('-', Event::_memory_counters), "\n";

# This displays something like
# "1-29509-0-0-0-0-5-0-3-0-8-0-0-0-0-0-0-0-0", where
# each slot is a certain event or watcher counter.
```

Mit Hilfe des Zusatmoduls **Event::Stats** kann man darüber hinaus auch Informationen zu jedem einzelnen Watcher und seinem Laufzeitverhalten erfragen. Mit **NetServer::ProcessTop** schließlich installiert man in seiner Anwendung mit einer einzigen Zeile einen kleinen Telnet-Server, der alle laufenden Watcher live in der Tradition des UNIX-Utilities `top` darstellt. Es ist faszinierend, dem Eventloop so über die Schulter schauen zu können! Der Clou des ganzen ist die Möglichkeit, über diesen Server dynamisch die verschiedensten Watcherattribute und -zustände zu variieren.

```
Watcher bei der Arbeit: NetServer::ProcessTop
```

```
serviceStatvfs PID=10012 @ redbull                          | 15:57:33 [  60s]
14 events; load averages: 0.97, 0.98, 0.00; lag  0%

  EID PRI STATE   RAN  TIME   CPU TYPE
DESCRIPTION                          P1
   10   4 sleep    84  0:46 86.2%   io action registration socket
    5   3 sleep     1  0:05  9.9%   io NetServer::ProcessTop
    0   7         150  0:00  1.6%  sys idle
    7   3 wait     70  0:00  1.6% idle idle process
    3   3 sleep    51  0:00  0.4%   io interface connection to s8a8263 via port
   16   3 cpu      11  0:00  0.2%   io NetServer::ProcessTop::Client s8a8263
    2   3 sleep    13  0:00  0.0% time Event::Stats
    9   4 sleep     0  0:00  0.0%   io more restricted interface registration s
    8   4 sleep     0  0:00  0.0%   io less restricted interface registration s
   11   2 sleep     0  0:00  0.0% time controler host list update timer
   12   2 sleep     0  0:00  0.0% time action host list update timer
   13   1 sleep     0  0:00  0.0% sign signal handler for HUP
   14   1 sleep     0  0:00  0.0% sign signal handler for INT
   15   1 sleep     0  0:00  0.0%   io controler socket
    6   6 sleep     0  0:00  0.0% time system check timer: actions
    0  -1           0  0:00  0.0%  sys other processes
```

NetServer::ProcessTop soll in zukünftigen Versionen um Symboltabelleninspektion ergänzt werden.

5.2. Watcher suspendieren

Jeder Watcher kann in einen speziellen *Modus* **SUSPENDED** gebracht werden. Auf den ersten Blick handelt sich dabei um einen weiteren *Zustand*, bei genauerer Betrachtung erweisen sich Suspendierungen aber als sehr verschieden davon. **SUSPENDED** wurde *für Entwicklung, Tuning und Fehlersuche* implementiert und wirkt sich besonders auf die *Aktivität* des Watchers aus.

Ein suspendierter Watcher verhält sich auf den ersten Blick wie im Zustand **INACTIVE**, ohne daß man seinen wirklichen Zustand dafür jedoch ändern muß: er registriert keine Ereignisse und generiert deshalb auch keine Aufträge. Genauer betrachtet *friert* **SUSPENDED** *den Watcher jedoch sozusagen ein*, so daß man ihn beispielsweise ungeachtet vielleicht gesetzter Timeouts beliebig lange studieren (und dabei natürlich auch seinen Zustand ändern) kann.

Ebenso wie die einzelnen Watcherzustände verrät sich auch **SUSPENDED** durch eine spezielle Methode, die in diesem Fall `is_suspended()` heißt.

Das „Einfrieren“ eines Watchers wird ausschließlich über das Attribut `suspend` bzw. die Methode `suspend()` gesteuert. Ein wahrer Attributwert setzt die Ereigniserkennung und Auftragsgenerierung so lange aus, bis er zurückgenommen wird. Bereits in der Queue hinterlegte Einträge werden davon ebensowenig beeinflußt wie der eigentliche Zustand des Watchers, denn *Suspendierungen wurden ja als Entwicklungs-Hilfe entworfen*. Ein Watcher kann also gleichzeitig **ACTIVE** und **SUSPENDED** sein. Suspendierungen sollten darum **nicht** in die eigentliche Anwendung eines Watchers integriert werden - dazu ist `stop()` besser geeignet.

```
# build a new active watcher
my $w=Event->var(var=>$object, cb=>\&cb);
print "Watcher gestartet.\n" if $w->is_active;

# suspend the watcher, check its state
$w->suspend(1);
print "Watcher ist weiter aktiv ...\n" if $w->is_active;
print "... aber supendiert.\n" if $w->is_suspended;
```

```
# cancel suspension
$w->suspend(0);
```

Das im vorhergehenden Abschnitt vorgestellte Modul **NetServer::ProcessTop** bietet zusätzlich die Möglichkeit, einen Watcher auch *remote* zu suspendieren.

Die besondere Zielsetzung dieses Modus' wird auch im folgenden Beispiel deutlich. Es zeigt, daß **SUSPENDED** den Watcher in eine besondere Umgebung einbettet, in der beispielsweise sonst verbotene Dinge möglich werden. Wird **SUSPENDED** verlassen, stellt **Event** jedoch wieder einen sinnvollen Zustand her.

```
# In diesem Beispiel wird ein Watcher mit
# unzureichenden Attributen in den Status
# ACTIVE versetzt. Diesen Übergang würde
# Event normalerweise verhindern, läßt ihn
# im Zustand SUSPENDED jedoch zu. Wird
# SUSPENDED verlassen, stellt Event wieder
# einen plausiblen Zustand her.

use strict;
use Event;

# make proband
my $w=Event->io(fd=>\*STDIN, parked=>1);

# check
state($w);
switch($w, 'suspend', 1);
switch($w, 'again');
switch($w, 'stop');
switch($w, 'start');
switch($w, 'suspend', 0);
switch($w, 'suspend', 1);
switch($w, 'again');
switch($w, 'stop');
switch($w, 'start');
switch($w, 'cb', sub {});
switch($w, 'suspend', 0);
switch($w, 'cancel');
state($w);

sub switch
 {
  # get operation
  my ($w, $op, @par)=@_;

  # get current state
  my @prev=($w->is_active(), $w->is_suspended(), $w->is_cancelled());

  # perform operation
  eval {$w->$op(@par);};
  die $@ if $@;

  # check new state, prepare message
  my ($msg, $diff)=("$op(@par): ", 0);
```

```
   $msg=join('', $msg, $diff?', ':'',
             "activity: $prev[0] ==> ", $w->is_active
            ),
     $diff=1 if $prev[0] ne $w->is_active;

   $msg=join('', $msg, $diff?', ':'',
             "cancellation: $prev[2] ==> ", $w->is_cancelled
            ),
     $diff=1 if $prev[2] ne $w->is_cancelled;

   $msg=join('', $msg, $diff?', ':'',
             "suspension: $prev[1] ==> ", $w->is_suspended
            ),
     $diff=1 if $prev[1] ne $w->is_suspended;

   # report changes
   print "$msg.\n";
  }

 sub state
  {
   # get operation
   my ($w)=@_;

   # report state
   print "STATE: active=>", $w->is_active,
         ", cancelled=>", $w->is_cancelled,
         ", suspended=>", $w->is_suspended, ".\n";
  }
```

5.3. Zusätzliche Möglichkeiten

Event bietet eine Fülle weiterer Features, die hier nur angedeutet werden können.

Auftretende Exceptions werden sicher abgefangen und als Meldung dargestellt. Der Loop läuft unbeeinträchtigt weiter. Diese Standardreaktion kann jedoch durch eine eigene Routine ersetzt werden.

Wichtige Stellen des internen Modulkerns lassen sich über Hooks durch eigene Routinen erweitern oder ersetzen.

Eine spezielle API ermöglicht die Einbindung schneller **C**-Callbacks.

5.4. Zusammenarbeit mit anderen Loops

Perl/Tk bringt seine eigenen Module zur Ereignissteuerung mit und kann deshalb derzeit meines Wissens nicht zusammen mit **Event** eingesetzt werden. Für **Event**-Programme gilt also nicht mehr die übliche Aussage, daß sie sich (wie andere Perlscripts) durch ein wenig Zusatzarbeit leicht mit einer grafischen Oberfläche ausstatten lassen. Nick-Ing Simmons verfolgt **Event** aber mit wachen Augen. Vielleicht gibt es eines Tages einen gemeinsamen Loop - das ist aber erst ein Wunschtraum.

Ähnliches gilt leider auch für **gtk+**, das wiederum ein eigenes Modell der Ereignissteuerung (auf der Basis von glibc) nutzt. Möglicherweise bieten die von **Event** angebotenen Hooks eine Möglichkeit der Integration, ich weiß jedoch von keinem entsprechenden Versuch.

Im Gegensatz dazu soll es aber sehr einfach sein, **Event** und **PerlQt** miteinander zu kombinieren.

```
Dieses Beispiel von J. N. Pritikin
demonstriert den gemeinsamen Einsatz
```

```
von Event und Qt.

use Qt 2.0;
use Event;

package MyMainWindow;

use base 'Qt::MainWindow';
use Qt::slots 'quit()';

sub quit {Event::unloop(0);}

package main;

import Qt::app;

Event->io(
          desc    => 'Qt',
          fd      => Qt::xfd(),
          timeout => .25,
          cb      => sub {
                          $app->processEvents(3000);  #read
                          $app->flushX();             #write
                         }
         );

my $w=MyMainWindow->new;

my $file=Qt::PopupMenu->new;
$file->insertItem("Quit", $w, 'quit()');
my $mb=$w->menuBar;
$mb->insertItem("File", $file);

my $at=1000;
my $label=Qt::Label->new("$at", $w);
$w->setCentralWidget($label);

Event->timer(
             interval => .25,
             cb => sub {
                        --$at;
                        $label->setText($at);
                       }
            );

$w->resize(200, 200);
$w->show;

$app->setMainWidget($w);
exit Event::loop();
```

Auf den ersten Blick schwierig ist die Zusammenarbeit von **Event** und anderen Modulen, die ebenfalls eine Art Ereignissteuerung implementieren, wie z.B. **Term::ReadLine::Gnu**, das entsprechend konfiguriert

jede Eingabe in STDIN abfängt, um beispielsweise eine automatische Vervollständigung von Dateinamen anbieten zu können. In Zusammenarbeit mit den Modulautoren hat sich aber gezeigt, daß beide Module durchaus kombinierbar sind. Die **Event**-Distribution enthält ein entsprechendes Beispiel.

6. Einsatzbeispiele

In der **Event**-Mailingliste wurde über folgende Projekte auf der Basis des Moduls berichtet:

Anwendung
Trading-System einer Bank
Die **State Machine-Library** *POE*, die im CPAN zu finden ist, soll in einer neuen Version auf **Event** aufsetzen.
Eine **Systemüberwachung** für Logfiles, Ports, Netzwerk, Prozesse u.ä. Beim Auftreten eines Ereignisses wird ein entsprechender Alarm versandt, z.B. per Mail oder an Systeme wie Tivoli. Das System ist voll konfigurierbar und auf der Basis nutzerdefinierter Agenten modular aufgebaut. Der Autor schreibt: "The Event module allows me to service all agents in a controlled & timely manner ... Watching a few active log files & testing each record against 20-30 regex's, checking the process list & netstat every minute, opening a couple of application ports & passing some info from time-to-time, & keeping an eye on paging – in total, costs about a minute of CPU a day. The benefits are many."
Datenbankfrontends
Webbackends
Client/Server-Architekturen

7. Ausblick

Event wird ständig weiterentwickelt und verbessert. J. N. Pritikin reagiert sehr schnell auf Anfragen und stellt oft in wenigen Stunden Fixes oder Patches zur Verfügung. Vor kurzem hat er einen beruflichen Aufgabenwechsel ins Windows-Umfeld angekündigt und dabei auch in Aussicht gestellt, daß **Event**'s Portabilität möglicherweise davon profitieren könnte.

In der Perl-Entwicklergruppe wird bereits darüber diskutiert, wie die Sprache selbst um Möglichkeiten zur Ereignissteuerung erweitert werden kann. Ob diese Erweiterung auf **Event** basieren wird, ist derzeit nicht klar. Ungeachtet dieser Diskussion ist es mit **Event** heute schon möglich, unter Perl ereignisgesteuert zu programmieren.

A. Technische Daten

Stichwort	**Details**
Name	**Event.pm** von J. N. Pritikin (JPRIT).
Version	Diese Einführung bezieht sich auf Version 0.66.
Originaldokumentation	*http://theoryx5.uwinnipeg.ca/CPAN/data/Event/Event.html*
Implementierung	größter Teil in *C*, um maximale Performance zu erreichen. Viel „Perl-Magic“.
Einschränkungen	**Event**-Loops sind threadsafe, *solange Event nur in einem Thread verwendet wird*, Threads sollen in Zukunft aber besser unterstützt werden.
Bekannte Bugs	Beim Beenden eines **Event**-Programms kann es in seltenen Fällen zu Speicherfreigabefehlern kommen, die sich aber nur in einigen erstaunlichen Fehlermeldungen der Laufzeitumgebung auswirken. Vorsicht ist lediglich bei der Einbettung in andere Programme geboten, die einen Returncode auswerten wollen. Die Ursachen sind noch nicht ganz klar, stehen aber offenbar im Zusammenhang mit einem entsprechenden Perl-Bug.
Plattformen	UNIX (verschiedene Derivate), ein Windows-Port erscheint inzwischen möglich.
Support und Austausch	Mailingliste *perl-loop@perl.org*.

Workshop: GUI-Programmierung in Perl

Slaven Rezic
slshape eserte@cs.tu-berlin.de

Im Rahmen des Workshops wuensche ich mir ein organisiertes Zusammentreffen, mit Demos und Erfahrungsaustausch zum Thema: „GUI-Applikationen mit Perl“.

Die Fragen, die mir dazu einfallen, sind:

- Ist die Erstellung „großer“ Anwendungen mit (grafischen) Benutzeroberflächen unter Perl möglich und sinnvoll? Welche Erfahrungen haben die Benutzer dabei gemacht, z.B. in Hinblick auf Portabilität, Entwicklungszeit und Performance?
- Ist Perl dabei eine Alternative zu Java, Tcl / Tk und Visual Basic?
- Wie kann man Perl-Programme am besten verteilen (CD-ROMs, Internet)?

Da ich mich im Bereich Perl / Tk recht gut auskenne, bin ich an Erfahrungsberichten bei der Nutzung anderer Toolkits (Qt, Gtk) interessiert.

Marc Lehmann wird voraussichtlich Gtk+ kurz vorstellen.

Danach werde ich anhand meines Fahrradroutenplaners BBBike (`http://pub.cs.tu-berlin.de/src/BBBike`) meine Erfahrungen bei der Verwendung von Perl / Tk und bei Portierungsdetails zwischen Unix und Windows vorstellen sowie einigen Studien zur Distributionserstellung mit den anderen Teilnehmern teilen.

Im Anschluss daran sind alle Interessenten herzlich eingeladen, sich selbst einzubringen.

Noch nicht ausformulierte Stichpunkte zu BBBike:

- Warum (und wie) GUI-Anwendungen mit Perl?
  - Vergrößerung der Akzeptanz bei Benutzern
  - Erweiterte Funktionalität
  - Konkurrenz von anderen Skript- / Interpretersprachen (Tcl, Java, VB...)
  - Standalone-Anwendungen vs. Perl-CGI-Anwendungen

  - Tk / Gtk / Qt / Curses / ...
  - Perl-Plugin

Ab hier beziehen sich die meisten Punkte auf Perl / Tk:

- Performance / Speicherverbrauch
  - hoher Speicherverbrauch
  - Startzeit bei großen Programmen lang
  - Reaktionszeit im interaktiven Umgang meist OK
  - teilweise Möglichkeit der Verwendung von XS zur Optimierung
- Anwendungsentwicklung
  - Zeit zum Parsen ist bei größeren Applikationen signifikant
  - (Lösung durch Verwendung von bytecompilierten Modulen?)
  - strikte Modularisierung und verzögertes Laden, AutoLoader / autouse
  - oder manuelles Laden per require+import bei Bedarf
  - Debugger (ptkdb und Standard-) verlangsamen die Ausführung
  - teilweise merklich
  - ptksh: interaktive Perl-Shell, kann leicht in Anwendungen zum
  - Debuggen eingefügt werden
  - „Reload modules“, um das erneute Starten der Anwendung während der
  - Entwicklung zu vermeiden
  - SpecPerl?
  - Allgemeines Lob für die Perl- und Tk-Entwickler. Mit Perl / Tk lassen
  - sich komplexe Anwendungen leicht und schnell entwickeln.
- Kompatibilität
  - leider keine Mac-Version von Perl / Tk (Freiwillige?)
  - kaum Probleme bei der Verwendung von Tk zwischen Win32 und X11
  - Unterschiede Win32 <=> Unix
    - Home-Directory ($HOME vs. Registry-Einträge)
    - fork
    - Ausführen von externen Programmen
    - case-sensitivity bei Dateinamen

    - locale bei Win32 nicht unterstützt
    - Extra-Features, RecentDoc
    - Problematisch: Druck (Ausweg: Ghostscript / view auch bei Win32 benutzen)
    - CD-ROM-Laufwerke
    - für Sonstiges: pod perlport

- Installation / Erstellung von Distributionen
  - Standards bei Unix: RPM (Linux), Ports / Packages (*BSD)
  - Source kann so ausgelegt werden, dass eine Installation nicht
  - notwendig ist (Verwendung von FindBin / lib für relative Pfade)
  - Ist die Installation per „perl Makefile.PL“ mit ExtUtils für
  - nicht-Module praktikabel?
  - Win32: Distributionen, die von der CD-ROM laufen, sind möglich
  - (langsam beim Start)
  - Automatisierung der Distributionserstellung: aufwendiger, wenn
  - die CD Versionen für mehrere Betriebssysteme enthalten soll.
  - Soll die CD ein komplettes Perl-Paket oder nur die benötigten
  - Module enthalten? Installation auf Festplatte: soll Perl getrennt
  - von der Anwendung installiert werden?
  - Entwicklung von jar-ähnlichen Modul-Archiven?
  - Alternativen: Perl2Exe
- Referenz mit existenten Perl / Tk- (Gtk,Qt?) Anwendungen

Die wunderbare Welt des CPAN (nur Abstract)

Andreas König
Andreas.Koenig@anima.de

Für die Effizienz des Einsatzes einer Programmiersprache und für die Qualität der erzielten Lösungen ist es von hohem Nutzen, wenn ein zentrales Open-Source Repository existiert, in das Autoren Perl-Module und -Skripte einstellen können und aus dem Anwender geeignete Bausteine installieren und damit wiederverwenden können.

Genau dies leistet CPAN (http://www.cpan.org/ und seine mehr als 100 Spiegelungen weltweit), das Comprehensive Perl Archive Network, also Perls umfassendes Archiv-Netzwerk, das im September 1999 bereits auf über 760 Megabyte angewachsen ist.

In diesem Beitrag werde ich, ähnlich wie beim Deutschen Perl-Workshop 1.0 oder der ersten Perl-Konferenz in San Jose, „meine“ CPAN-Shell und die Vielfalt ihrer arbeitssparender Anwendungen präsentieren.

The DNAPLOT Server: Ein spezialisierter Internet-Server zur Analyse von DNA-Sequenzen

Werner Müller
W.Mueller@uni-koeln.de

Hans-Helmar Althaus
Althaus@dnaplot.de

Der Server http://www.dnaplot.de ist auf die Analyse von immunologisch relevanten Sequenzen, vorzugsweise vom Menschen, spezialisiert. Die Sequenzanalyse wird von einem selbsterstellten Sequenzanalyse-Programm DNAPLOT (ANSI-C) vorgenommen. Die Steuerung des Programms und die Schnittstelle zum Internet werden durch Perl-Skripte realisiert. Durch den Einsatz des Moduls CGI.pm konnte sehr einfach erreicht werden, daß die einheitliche Perl-Schnittstelle über einen Internetbrowser oder Web-Agenten interaktiv angesprochen sowie über lokale Skripte gesteuert werden kann.

Zur einfacheren Einbindung des C-Programmes DNAPLOT in Perl-Skripte wurde ein einfaches DNAPLOT.pm-Modul entwickelt. Der Implementierung liegt die Idee zugrunde, zeitraubende Zeichenvergleiche in das C-Programm einzubauen und die weniger zeitkritischen Routinen, z.B. Aufbereitung der Web-Ausgabe, als Perl-Routinen zu realisieren.

Für jede Gruppe von Sequenzen wurden spezifische Sequenzdatenbanken erstellt, die mit Hilfe von Skripten autmatisch erneuert werden können. Das Programm DNAPLOT erlaubt eine automatische Annotierung von Sequenzen. In Zukunft soll das Programmpaket für weitere Genfamilien ausgebaut werden.

XML::Edifact
(nur Abstract)

Michael Koehne
kraehe@copyleft.de

EDIFACT ist ein wichtiger Industriestandard zum Austausch von Formularen in einem „papierlosen Büro“, der in über 2.700 Seiten festgeschrieben ist (United Nations Standard).

XML::Edifact ist ein GNU-lizenziertes, freies Perl-Modul, das einen offenen Migrationspfad zwischen XML und UN/EDIFACT bietet. Es setzt jede wohlgeformte UN/EDIFACT-Mitteilung in für Menschen lesbares XML um und umgekehrt. Dazu benutzt es den Original-Wortlaut der UN/EDIFACT-Stapelverzeichnisse als Auszeichnung und das definierende Dokument als Namensraum.

Da die meisten Teilnehmer Perl kennen, werde ich voraussichtlich mehr über den Standard denn über Perl-Interna sagen.

Weitere Informationen finden sich unter: http://www.xml-edifact.org/

Einführendes Tutorial: Was bieten Perl und dieser Workshop? (nur Abstract)

Alexander Sigel
sigel@bonn.iz-soz.de

Jürgen Christoffel
jc@port25.com

1. März 2000

Zielgruppe:

Das einführende Tutorial richtet sich an Interessenten mit keinen oder ersten Vorkenntnissen in Perl, die eine Orientierungshilfe suchen, um sich danach selbständig weiter einarbeiten zu können. Vornehmlich ist dabei an Studierende der FH Rhein-Sieg gedacht, potentiell aber auch an Berufspraktiker.

Auch kurzfristig und ohne Voranmeldung ist die Teilnahme noch möglich. Das Tutorial ist im Tagungsbeitrag enthalten.

Inhalt:

Das einführende Tutorial richtet sich an Interessenten mit keinen oder

In etwas mehr als 2 Stunden wollen wir eine Übersicht darüber vermitteln:

1. Was bietet Perl?
   - Zentrale, attraktive Konzepte
   - Bedeutung und wichtige Einsatzbereiche
   - existierende Lösungen, Wiederverwendung
2. Was gibt es für mich auf dem Workshop?
   - Leichtverständliche Erläuterung, um was es in den Beiträgen grob geht und wie sie zusammenhängen
   - Bewertungshilfe, was für die eigene Arbeit eher relevant

Es handelt sich nicht um einen Programmierkurs – das kann ohnehin in der Kürze der Zeit nicht geleistet werden – sondern lediglich um einen Appetithappen, der Lust auf mehr wecken soll.

podquovadis - Pod quo vadis?

Marek Rouchal
marek.rouchal@hl.siemens.de
Infineon Techologies AG
Postfach 80 09 49
81609 München
Tel 089 234 25849
Fax 089 234 24477

21. Februar 2000

Inhaltsverzeichnis

Auf CPAN tauchen immer mehr POD Konverter auf, mit stark unterschiedlichen Fähigkeiten und in sehr durchwachsener Qualität. Die These dazu ist: Die Dokumentation des POD-Formats (`perlpod`) ist zu lax und läßt zum Teil unterschiedliche Interpretationen zu. Viele Konverter stricken daher ihre eigenen Erweiterungen (hidden features!) um `perlpod` herum.

Aber es ist Licht am Horizont zu erkennen: Brad Appleton's `Pod::Parser` Modul wird ab 5.6 integraler Bestandteil des Perl-Cores und damit Fundament für alle Arten von Konvertern. Zusammen mit einer überarbeiteten und detaillierteren Dokumentation und einem Verifikationswerkzeug (`Pod::Checker`) ist damit der Grundstein gelegt, um eine allgemeine Vereinheitlichung anzugehen. Der erste Beitrag meinerseits zu diesem Thema ist der `Pod::Checker` und (demnächst) `Pod::HTML`.

1 Motivation

Seit einiger Zeit setzt Infineon DAT (Design Automation) POD zur Dokumentation der Perlskripte und weiterer Teile des IC-Designflows ein. Mark Pease's **pod2fm** dient zur Konversion nach FrameMaker, mit dem wiederum die gedruckte Dokumentation sowie PDF erstellt wird.

pod2fm hat allerdings einige Schwächen, die ich beheben wollte. Dabei stieß ich relativ bald an die Grenzen des Skripts, dessen Design im wesentlichen dem ursprünglichen **pod2html** ähnelt und damit nicht besonders robust und wartbar ist. Also entstand die Idee, das Problem an der Wurzel, d.h. beim Parsen des POD-Formats, zu packen.

2 Der Schlüssel: `Pod::Parser`

Nach kurzem Suchen auf CPAN wurde ich fündig: `Pod::Parser` von Brad Appleton bietet das Rüstzeug, um POD elegant zu verdauen. Ein einfacher `Pod::Checker` war bereits auch dabei. Diese Gelegenheit habe ich am Schopf ergriffen und `Pod::Checker` erweitert, um schnell die Fehler meiner Kollegen (die leider noch nicht fließend POD sprachen) zu finden. Brad hat diese Erweiterungen kritisch, aber auch dankbar entgegengenommen. `Pod::Checker` tut seitdem zuverlässig seinen Dienst und meckert eine Vielzahl von Fehlern an. Unsauberkeiten, die zwar keine Fehler darstellen, aber dem einen oder anderen POD-Konverter Probleme bereiten können, werden als Warnung gemeldet.

3 Intermezzo, das sich zum Projekt ausgewachsen hat: `Pod::HTML`

Auf Kundenanfrage hin wurde ich auf das **pod2html** Thema gestoßen und erlebte den ersten Einbruch der heilen POD-as-single-source-Welt: Das im Perl-Core enthaltene **pod2html** scheitert bereits an zwei verschachtelten Sequenzen á la `S<[ B<-force> ]>`! Mit `Pod::Parser` im Hinterkopf war schnell die Idee geboren, sozusagen zum Aufwärmen für das **pod2fm**-Projekt mal eben schnell einen `Pod::HTML` zu hacken. Gesagt, getan, Hals über Kopf habe ich drauf losgehackt. Ein Prototyp war schnell fertig, in der gleichen Geschwindigkeit wuchs jedoch die TODO-Liste. Höchste Zeit, um innezuhalten und CPAN zu konsultieren. Und siehe da, hätte ich das mal eher gemacht: `PodToHtml-0.04` von Nick Ing-Simmons wartete nur darauf, heruntergeladen zu werden. Leider zeigte sich bald - siehe auch an der Versionsnummer - daß der Autor an dem Paket nicht mit Hochdruck weiterentwickelte.

4 Der Plan

ist simpel: Basierend auf den vorhandenen Modulen und Skripten einen neuen `Pod::HTML` implementieren und mit der Erfahrung daraus `Pod::Maker` anzugehen. Nebenbei fiel mir auf, daß Russ Alberry beeits `Pod::Text` und `Pod::Man` auf der Basis von `Pod::Parser` reimplementiert hat - großartig, und absolut komplentär zu meinen Anstrengungen. Das spornte mich an, folgende Ziele fuer `Pod::HTML` zu setzen:

*** Robustheit**
: Der Konverter soll POD wirklich verstehen, d.h. einen echten Parser benutzen - und der ist ja schon da!

*** Modularität**
: Nick Ing-Simmons hat es vorgemacht: `Pod::Parser`, `HTML::Element` und `HTML::FormatPS` werden eingesetzt, das sollte so bleiben. Dazu noch `HTML::Parser` für die `=begin html` Abschnitte.

*** Allgemeinheit**
: `Pod::HTML` sollte zwar die Perl-eigene Dokumentation ordentlich umsetzen, aber nicht in erster Linie dafür optimiert sein.

*** Konfigurierbarkeit**

Durch die Ausnutzung von OO-Features sollte es sehr leicht möglich sein, den HTML-Output lokalen Anforderungen anzupassen.

*** Wiederverwendbarkeit**

Teile, die auch für andere Konverter sinnvoll sind, sollen als Module implementiert werden.

*** Kompatibilität**

Soweit möglich, sollte das **pod2html**-Skript ähnliche Kommandozeilenargumente haben wie die existierenden Skripte.

5 Die Konkurrenz: `Pod::Tree::Html`

Bei einem weiteren Trip ins CPAN fand ich dann PodTree-1.00 von Steven McDougall. Das ist einer dieser bedauerlichen Fälle, wo zwei fähige Programmierer Spaß an der selben Sache gefunden haben. Soweit ich erkennen kann, bietet PodTree nichts (außer einer etwas konsequenteren Implementierung des Syntax-Baums), was PodParser nicht hätte, und PodParser war zuerst da. Damit liegt meine Sympathie bei letzterem. Spätestens jetzt ist es Zeit für einen Vergleich der Features der existierenden und des Statuses des werdenden **pod2html** (leider z.T. unvollständig).

Ja, funktioniert j: ja, mit Einschränkungen
Nein, wird nicht unterstützt n: nein, funktioniert nicht richtig
P, geplant, aber noch nicht umgesetzt

Kategorie	*pod2html*	*PodToHtml*	*PodTree*	*PodHTML*
Umsetzung der Standard POD syntax	n	J	J	J
Rekursives Abarbeiten von Verzeichnissen	j	?	?	J
Relative Links im HTML	n	?	?	J
Postscript-Option	N	j	N	J
Einfache Anpassung an lokale Stile	N	N	N	J
Verwendung von CSS	N	N	N	P
Links zu Bildern	j	N	N	P
Erzeugung von Inhaltsverzeichnis und Index	N	J	?	j
Angabe von „Bibliotheks-PODs“	J	N	N	J
Automatisches Highlighting von URLs, Mailadressen, Perl Variablen ...	J	N	N	p
Checken der Syntax, aussagekräftige Fehlermeldungen	N	?	?	J
Modularität	N	j	N	J
Erkennung und Validierung der Hyperlinks	n	j	?	J
Unterstützung der Entities	j	n	n	JP
Konfigurierbarkeit der HRs	N	N	J	j
C<XE<lt>...E<gt>> als Link-Ziele	?	?	J	?
Linkziele als Cache-File	j	N	N	N
Navigation zwischen den Seiten	N	N	N	J
Optionale lokale Navigation (TOC)	j	J	J	J

6 Status

PodHTML (Version 0.03, sucht noch einen Platz auf CPAN) funktioniert schon recht gut. Folgende Arbeiten stehen noch aus:

*** Splitten des Indexes auf mehrere Files**

Momentan kommt ein 850k Monolith heraus!

*** Einführung von stylesheet CLASSes**

um die Konfiguration der Darstellung von logischen Einheiten zu ermöglichen (z.B. `H2` oder `DT`)

*** Nachziehen der Features/Kommandozeilenargumente der anderen Konverter**
*** Kritik einer breiteren Öffentlichkeit/p5p einholen.**
*** `perlpod` anpassen**

Einige Features von bestimmten Konvertern werden inzwischen so häufig eingesetzt, daß sie in `perlpod` dokumentiert werden sollten.

*** Tabellen**

Eine rudimentäre Unterstützung von Tabellen ist überfällig.

7 Schmankerl

Eigenlob stinkt, also nennen wir's einfach die Features, die den neuen Konverter von den existierenden abheben.

*** Verwendung von `Pod::Checker`**

Das ist natürlich gemein, aber Microsoft macht's ja genauso: Ich habe in den `Pod::Checker` Funktionalität eingebaut, um die in einem POD-file vorhandenen Linkziele zu extrahieren; das wird von `Pod::HTML` benutzt. Wie auch immer, `Pod::Checker` steht schließlich allen zur Verfügung.

Im Ernst: Um die Links, bessergesagt die Ziele zu finden, müssen alle PODs zweimal geparst werden. Für den ersten Lauf wird `Pod::Checker` verwendet, so daß man nicht nur die Link-Ziele erhält, sondern auch die Information über alle Fehler und Unsauberkeiten. Und man hat die Garantie, daß die Link-Ziele sowie die Links selbst auf die gleiche Weise verarbeitet werden, so daß maximale Konsistenz sichergestellt ist.

*** `Pod::Hyperlink`**

Diese Klasse aus dem `Pod::ParseUtils` Modul parst Links auf sehr aufwendige Weise, so daß die größtmögliche Zahl der z.B. in den Perl-Modulen vorkommenden Links korrekt erkannt werden. Das geht etwas zu Lasten der Eleganz des Codes, aber es ist noch wartbar.

*** `=begin html`**

Auf diese Blöcke wird `HTML::TreeBuild` losgelassen, welches wiederum die HTML-Syntax dieser Einschübe prüft. Weiterer Vorteil: Auch der Inhalt dieser Einschübe erscheint dann - soweit von `HTML::FormatPS` unterstützt - im PostScript-Output.

*** „schönes" HTML**

Bei der Erzeugung / Ausgabe des HTML-codes wurde darauf geachtet, daß auch dieser Code „lesbar" ist, d.h. an bestimmten logischen Punkten neue Zeilen begonnen werden. Das ganze dient im wesentlichen dem Debugging, nicht etwa einer Vereinfachung der Nachbearbeitung, denn diese macht man geschickterweise mit den...

*** OO-Features zur Anpassung an lokale Anforderungen**

Nach der Umsetzung des eigentlichen POD-files wird eine Methode „`customize`" aufgerufen. Per default fügt diese eine Überschrift / Unterschrift hinzu sowie die (optionale) Navigation. Diese Stelle kann man nun mittels den Perl-OO Methoden gezielt verändern, um so lokale Anpassungen (Logos, Navigation, Farben, ...) vorzunehmen.

8 Ausblick

Ich hoffe, die Arbeiten im ersten Quartal 2000 abschließen zu können. Falls noch Zeit übrigbleiben sollte, habe ich die Unterstützung von Navigationsframes (auf Directory-Ebene) im Hinterkopf, so daß man quasi per Knopfdruck ein benutzerfreundlich verlinktes Perl-Referenzmanual erhält.

Die Entwicklungsarbeit an `Pod::Checker`, `Pod::ParseUtils` und anderen wird dann in die Überarbeitung / Neuimplementierung von `Pod::Frame` fließen. Parallel dazu möchte ich mich an einer Überarbeitung von `perlpod` beteiligen. Damit sollte der Weg geebnet sein um andere Klassen von Konvertern zu entwickeln. Hat da jemand `Pod::XML` gerufen?

Siehe auch

*** Brad Appleton's `PodParser-1.092`**
*** Nick Ing-Simmon's `PodToHtml-0.04`**
*** Steven McDougall's `PodTree-1.00`**
*** Russ Allbery's `podlators-0.08`**
*** Tom Christiansen's `pod2html` und `Pod::Html` (im Perl core)**

Using Parse::RecDescent

Craig Smith
smith@gandalf.uni-trier.de

February 28, 2000

Contents

1 Introduction

Parsing text is a common component of programming problems. Usually we represent information in a nice human readable format. However, this information is not easily manipulatable by a program. The information needs to be checked to be sure it contains valid characters, that they form valid keywords (syntactically correct) and the keywords are in a particular order. Usually actions are associated with a valid ordering either to associate meaning with the original text, or to perhaps just collect information from the text. Performing these tasks is what parsing is about.

An example of parsing you should be familiar with is your trusty Perl interpretor! It examines the submitted Perl script by first checking to be sure it consists of permitted Perl characters (this is almost all the characters on the keyboard!) which build valid Perl keywords. The order of these keywords is important too–so the parsing phase of the interpretor also checks if the keywords are in a valid order (the Perl code `"my for my $j (0..10)"` consists of valid Perl keywords, yet is it is not semantically correct). After the input is deemed valid, actions are taken to convert the Perl program to an internal format.

`task-over.gif` [1] For problems of this general ilk it would be nice if there existed a general tool to lex an input stream (convert input stream to keywords), specify what a valid ordering of

these keywords are and then associated actions with these accepted orderings. A tool of this type has existed for a long time: lex and yacc. These tools existed long before Perl so naturally, because the state of the art keeps advancing, there is a better tool available for the Perl community: Damian Conway's wonderful Parse::RecDescent module available at a CPAN site near you.

A general tool for parsing must allow the user to specify what are valid tokens (keywords), what a valid ordering of those tokens are and what actions to take when we have found a valid ordering. This is easy using a grammar.

2 Grammar

A grammar consists of rules that specify what are valid tokens and the order of those tokens. It is easier to illustrate with an example. Let's progressively build a prefix notation calculator that will allow variables and complex expressions. Here is a sample input to the calculator:

```
+ 10 2
=> 12
```

Now that we have it in use, what is a valid addition operation? It is a plus sign followed by two numbers.

```
addition_expr: '+' number number
               { print "successfully matched addition_expr rule\n"; }
number: /-?\d+/
```

Above is a rule that specifies the tokens and the valid ordering of those tokens. Some grammar terminology is necessary to continue. The text to the left of the colon is the name of the rule. In this example there are two rules: *addition_expr* and *number*. To the right of the colon is a list of items that comprise the production–the ordering necessary for this rule to be satisfied. The items in the production can be terminals or non-terminals. A terminal specifies what a token is comprised of: in the case of addition_expr the plus sign ('+') is a terminal. A non-terminal item (or subrule) refers to another rule: in addition_expr `"number"` is a non-terminal. The text inside the braces, after the last item of the production, is the actions that are executed when the production succeeds. The actions are familiar perl code. In this case the action merely prints out that the rule successfully matched. Later, this will be replaced with an action that evaluates the prefix expression.

The number rule is more interesting. It contains merely one item–a terminal. This terminal defines legal token by a regular expression, one that matches negative or positive integral numbers.

Here is how we would use this grammar in a perl code:

```
1  use Parse::RecDescent;
2  my $grammar = <<'EOG';
3  addition_expr: '+' number number
4                 { print "successfully matched addition_expr rule\n"; }
5               | <error: illegal expression>
6  number: /-?\d+/
7  EOG
8  my @lines = <DATA>
```

```
 9  my $text = join('', @lines);
10  my $parserRef = new Parse::RecDescent($grammar);
11  my $ret = $parserRef->addition_expr($text);
12  die $@ if $@;
```

Lines 2-7 define the grammar and store it in the variable $grammar. The RecDescent module cannot read directly from STDIN, so input must be gathered into a string and then presented for parsing (lines 8-9). The parser for the defined grammar is created on line 10 and a reference to the object stored in $parserRef. Parsing is started by calling a rule defined in the grammar–in this case addition_expr (line 11). If this rule returns a value, it is placed in $ret. The return value of a rule is explain later.

3 Example

Let's build up the calculator grammar example I mentioned earlier, however without variables for now. Most of the work is building the grammar. A few more operators would be useful: subtraction, multiplication and division:

```
start:  expression
expression: addition_expr         |
            subtraction_expr      |
            multiplication_expr   |
            division_expr

addition_expr:       '+' number number
subtraction_expr:    '-' number number
multiplication_expr: '*' number number
division_expr:       '/' number number
number: /-?\d+/
```

We'll define the actions later. There are couple of new things here:

- I've inserted a rule at the top called start. It is useful to clearly identify the starting point of the grammar.
- The '|' symbol in the grammar means "or". So the expression rule now can be fulfilled by the subrules addition_expr, subtraction_expr, multiplication_expr or division_expr.

However, this grammar doesn't accept nested expressions (i.e. *+ 3 - 4 3*). The problem is that the second operand to the addition_expr must be a number (as the first operand). A '-' occurring in the middle of the addition_expr rule is not allowed so it fails. We need either operand of any operator to be an expression, including a number. Let's try again:

```
start:  expression
expression: addition_expr         |
            subtraction_expr      |
            multiplication_expr   |
```

```
            division_expr         |
            number

addition_expr:       '+' expression expression
subtraction_expr:    '-' expression expression
multiplication_expr: '*' expression expression
division_expr:       '/' expression expression
number: /-?\d+/
```

This looks better! Now either the first or second operand to any operator accepts an expression. This expression can be a number or any operator! A nice recursive definition. Let's add actions to this to evaluate the expression and print out the answer:

```
start:      expression
            { print "$item[1]\n"; }
expression: addition_expr         |
            subtraction_expr      |
            multiplication_expr   |
            division_expr         |
            number                |
            <error: unrecognized expression>
addition_expr:       '+' expression expression
              { $return = $item[2] + $item[3]; }
subtraction_expr:    '-' expression expression
              { $return = $item[2] - $item[3]; }
multiplication_expr: '*' expression expression
              { $return = $item[2] * $item[3]; }
division_expr:       '/' expression expression
              { $return = $item[2] * $item[3]; }
number: /-?\d+/
```

There is a lot here.If no part of the currently parsed text matches the subrules of expression, then an unknown expression was entered and the it will trigger the <error:> directive. The <error: string> directive causes the parser to terminate and print the string. However, the most important new idea is that items (terminals and non-terminals), like perl subroutines, can return a value back to the rule that invoked it. The return value is passed back to the caller by setting the special $return variable. The default action is to set $return to the *last* item in the production (hence there are no actions for the productions in the expression rule). Access to the return value of each item in the current production is through a specially prepared array called 'item'. The first value of the array, $item[0], always contains the name of the current rule being executed and the items in the production list occupy the elements thereafter.

4 Extended Example

Let's add to the calculator variables and assignments (i.e. = a 2):

```
start:      expression
     { print "$item[1]\n"; }
expression: bin_op
          | number
          | assign
          | rval
          | <error: unrecognized expression>
bin_op:     bin_operators expression expression
      { eval('$return = ' . "$item[2] $item[1] $item[3];"); }
bin_operators: '+' | '-' | '*' | '/'
number: /-?\d+/
assign: '=' lval expression
      { ${$item[2]} = $item[3];
        $return = ${$item[2]}; }
rval: variable
    { my $ref = $item[1];
      if ( not defined($$ref) ) {
        $$ref = 0;
      }
      $return = $$ref;
      1; }
lval: variable
variable: /[a-zA-z]+/
        { no strict 'refs';
          my $var = "varspace::$item[1]";
          $return = \${$var}; }
```

Ouch! That hurts! Well, it is not **that** bad. I've compressed the binary operators in the previous example down to one rule by converting the expression into Perl code and evaluating it (this of course depends on the binary operators being the same as Perl's). The rules that implement variables and assignments are below the number rule. Variables can be used in two contexts, using terminology from C, as a rval or lval. An rval is the contents of the variable and the lval is the location of the variable. The first argument to an assignment must be an lval so the second argument can be stored. Looking closely at the grammar, an lvar evaluates to a reference to a variable in the 'varspace' package. This reference, in the assign rule, is followed to the referent which is set to the value of the expression non-terminal. The rval rule retrieves a reference to the variable via the variable rule (as in the lval rule). However, the rval rule shelters the caller of rval by setting the referent to zero if it is not defined (Perl, on the other hand, converts to zero but does not set the variable).

Viola! We have the grammar for an infix calculator with variables and nestable prefix notation expressions. This is actually the beginnings of an interpretor for a language: it has variables and expressions, it just needs control flow. The complete code, including the reading of input from STDIN and invoking the parser, is left for the reader to do at home. I'll give you a suggestion: invoke the parser on each line separately so the user can see the results for each command.

5 Advanced Features and Gotchas

There are many many features I cannot discuss in detail because of space limitations. They are important so I'll briefly mention them:

- Item Quantifiers: One can specify how many times a item must occur in a production, i.e. zero or one time, one or more. For instance a rule for plus in prefix notation that accepts two or more expressions:

```
addition_expr:      '+' expression(2..)
```

- Non-terminal Arguments: It is possible to pass data, via arguments, from the caller to the subrule. This, with other features, makes it possible to create generic rules that are configured by the arguments:

```
genlist:     <matchrule:$arg{rule}>[ %arg ]  taillist[%arg]
                { $return = [ $item[1], @{$item[2]} ] }
taillist:    /$arg{sep}/ genlist[%arg]
        |    ...!/$arg{sep}/ { $return = [] }
```

  The arguments are passed to the subrule by following the item with list in side []'s. In the subrule, the arguments can be accessed in the @arg array or the %arg hash. In the future I forsee an extensive library of utility rules.

- Negative Lookahed: This feature is used in the example above and therefore I am obliged to explain it. The token '...!' preceeding the item /$arg{sep}/ means that for the production to successfully match it cannot start with the separator token. That is, the negative lookahead feature specifies what is not permitted occur next (I suppose somewhat akin to '!~').

- Indirect Subrule Invocation: Because the grammar is interpreted it is possible to specify a non-terminal in a string. In the genlist rule above, the user passes in the name of the rule for an element in the list. This rule is invoked by using the <matchrule: > directive.

- Trace Mode: Setting the global variable $::RD_TRACE runs the parser in verbose mode. The parsing actions taken by RecDescent are clearly displayed and is easy to follow with your grammar in another window. Invaluable for finding logic errors.

- Left Recursive Rules: Due to the nature of recursive descent parsers, rules that contain left recursion are not permitted because they cause infinite recursion:

```
A: A '1' | '1'
```

  Rewriting this to be compliant is easy and therefore left as an exercise for the reader. :-)

- Accidently Failing a Rule: A block of actions is considered an item in a production. It can therefore be followed by other items. For the block to match, the last expression in the block must evaluate to a defined value. Don't be surprised to see, as above in the rval rule, an action block ending with "1;" just to make sure the block evaluates to a defined value.

- Speed: RecDescent is not fast. If speed is necessary consider writing the parser in C with lex and yacc.

6 Conclusion

Programming is hard enough, even more difficult when faced with a parsing problem. As Larry says, be lazy. Therefore, write it once. Write it such that it is flexible. Write it with the proper tool: RecDescent.

7 Bibliography

- Conway, Damian. "The man(1) of descent". In *The Perl Journal*, Issue #12 1998. 46-58.
- Conway, Damian. RecDescent(1) man page.

8 Acknowledgements

Special thanks to my brother, Eric Smith, a Perl nut like myself made many useful suggestions. Thanks also to Damian Conway for answering my emails with humor and zest.

Mit `Paula` distributionsunabhängig Linux „einfacher“ administrieren

Axel Miesen
miesen@id-pro.de

25. Februar 2000

Inhaltsverzeichnis

`Paula` ist eine Software zur „einfachen“ Linux-Administration. Der entscheidende Unterschied zu `YaST`, `Linuxconf`, usw. ist die Distributionsunabhängigkeit, was konkret heißt, daß `Paula` dank eines modularen Konzepts jederzeit dahingehend erweitert werden kann, daß es mit jedem Linux zurechtkommt.

`Paula` ist Client / Server-basiert. Der erste Client wird ein Web-Frontend sein, denkbar sind auch ein Perl / Tk-Frontend oder ein Shell-artiges Frontend (so ähnlich wie bei mysql, gnuplot, ...).

Der Server ist ein CIM Object Manager, realisiert als Apache-Modul (mod_perl). CIM heißt „Common Information Model“ und ist ein objektorientiertes Modell, mit dem man Management-Informationen beschreiben kann. (Ausführliche Infos dazu gibt's bei http://www.dmtf.org.)

1 Warum ist distributionsunabhängige Linux-Konfigurationsverwaltung ein Problem?

Sie ist genaugenommen kein Problem, zumindest kein großes. Es bedeutet lediglich einen gewissen Mehraufwand, die maschinelle Durchführung von Administrationsaufgaben für verschiedene Distributionen vorzusehen. Dazu schwebt uns folgende Lösungsidee vor: In einer systemweiten Konfigurationsdatei, etwa `/etc/paula/paula.conf`, findet sich z.B. die Zeile:

```
Distribution: RedHat6.1
```

Aufgrund dieser Zeile wird dann der `@INC`-Pfad erweitert, so daß das Modul-Set für RedHat 6.1 verwendet wird, in welchem die „Low-Level“-Ausführung der jeweiligen Administrationsaufgabe implementiert ist.

2 Unsere Lösungsidee

2.1 Konzeption/Architektur

Wie eingangs schon erwähnt, ist `Paula` Client / Server-basiert. Client und Server kommunizieren miteinander über HTTP / XML / CIM, was wir kurz an einem Beispiel verdeutlichen wollen:

Angenommen, das Frontend braucht im Unterpunkt „Benutzerverwaltung“ eine Liste aller Benutzernamen im System. Das Frontend sagt also zum Server: „Gib mir alle Benutzernamen im System.“

In objektorientierte Sprache übersetzt, heißt dies: „Gib mir alle Instanzen(namen) der Klasse ’User’“, oder, schon etwas maschinennäher: `EnumerateInstanceNames('User')`

Im Dokument *Specification for CIM Operations over HTTP* ist spezifiziert, wie nun das „on-the-wire-encoding“ geschehen soll, nämlich mit der Hilfe von XML:

```
<?xml version="1.0" encoding="utf-8" ?>
 <CIM CIMVERSION="2.0" DTDVERSION="2.0">
  <MESSAGE ID="42" PROTOCOLVERSION="1.0">
   <SIMPLEREQ>
    <IMETHODCALL NAME="EnumerateInstanceNames">
     <LOCALNAMESPACEPATH>
      <NAMESPACE NAME="root"/>
      <NAMESPACE NAME="paula"/>
     </LOCALNAMESPACEPATH>
     <IPARAMVALUE NAME="ClassName">
      <CLASSNAME NAME="User"/>
     </IPARAMVALUE>
    </IMETHODCALL>
   </SIMPLEREQ>
  </MESSAGE>
 </CIM>
```

Das Ganze wird jetzt noch mit einem passenden Header versehen und dann zum Server, d.h. zum CIM Object Manager geschickt. Dieser parst das ganze, generiert eine passende Antwort (an geeigneter Stelle steht dann zum Beispiel `<IMETHODRESPONSE>`, schickt sie zurück zum Client; dieser zieht sich seine Informationen aus dem XML-Dokument und stellt sie für den Benutzer dar.

Am Vortragstermin werde ich etwas tiefer auf CIM eingehen.

2.2 Verwendete Software bzw. Module

- Apache / mod_perl
- LWP
- XML::DOM
- nsgmls

3 Offene Punkte

Da gibt’s noch einige ...

Die Implementierung des CIM Object Managers steht noch ziemlich am Anfang, nicht zuletzt deswegen, weil einige Konzepte im CIM noch relativ schlecht verstanden sind. Auch werden nirgends wirklich handfeste Tips zur konkreten Implementation gegeben, da heißt es z.B. nur: “CIM is not bound to a particular implementation, and this flexibility allows different systems and applications to exchange and interpret managment information.”

Auf der Client-, d.h. Frontendseite gibt’s auch noch Unklarheiten. Daß hinter dem ganzen ein (wie wir finden) gutes Konzept steht, interessiert denjenigen, der `Paula` nutzt, letztlich wenig; es soll hauptsächlich „intuitiv“ bedienbar sein und funktionieren.

Auch dazu mehr am Vortragstermin...

Reverse Engineering von Obfuscated Perl: Ein einfaches Beispiel: Meine Y2k-Email-Signatur

Jörn Reder
joern@zyn.de

1. März 2000

Inhaltsverzeichnis

1 Was tut folgendes Programm und warum?

```
  &_,s;(.);chr(1^ord($1));eg;sub _{$_=q;%}<0:%^<uhld,857790311:gns)::;.
'(zAu<mnb`muhld)%^(:qshoug!#x3j>!ruhmm!`mhwd !$ei-!$el-!$er!]s#-
%^.'.
q+2711-%uZ0\-%uZ1\:**%^:rmddq!0|+}eval,"++$_-> http://www.zyn.de/y2k"
```

Auf den ersten Blick sieht dies hier nach einem schwer verdaubaren Stück Perl Code aus, und die Aussage "Nur Perl kann Perl parsen!" scheint mal wieder bestätigt zu werden. Aber eben nur auf den ersten Blick ... ;)

Auf Email-Signature kompatible Größe (nicht mehr als 4 Zeilen, weniger als 70 Zeichen Breite) wurde hier ein sehr unübersichtliches Programm gepresst, welches darüber Auskunft gibt, wie lange wir den angeblichen und so umfangreichen Y2K Crash nun schon überleben. Die Ausgabe sieht in etwa so aus:

```
y2k?  still alive!  711h, 57m, 58s
```

wobei die Zeit im Sekundentakt hochgezählt wird.

2 Wie funktioniert das?

Im folgenden wird der obige Code Stück für Stück analysiert und man wird schnell feststellen, daß die Idee dahinter relativ simpel ist und nur durch möglichst viele syntaktische Besonderheiten von Perl verschleiert wird.

Im wesentlichen läßt sich der Code in fünf Bestandteile zerlegen, die sauber aufgeschrieben so aussehen:

1. `&_`
2. `s;(.);chr(1^ord($1));eg;`
3. `sub _ { ... }`
4. `eval`
5. `,"++$_-> http://www.zyn.de/y2k"`

Es macht Sinn mit dem dritten Schritt zu beginnen: hier wird eine Subroutine definiert, die die lineare Abarbeitung des Codes unterbricht, was schon ein erstes Mittel der Verwirrung darstellt.

Aufgerufen wird die Subroutine mit der ersten Zeile, anhand der seit Perl 5 nicht mehr üblichen Schreibweise mit vorangestelltem & Zeichen. Ein Vorteil: ohne Verwendung eines Prototypen können so die sonst bei diesem Funktionsaufruf notwendigen leeren Klammern weggelassen werden: das spart Platz, und das Programm soll ja schließlich so kurz wie eben möglich sein.

Betrachten wir also zunächst den Inhalt der Subroutine mit dem spartanischen Namen _. Die Wahl dieses Namens dient ebenfalls der Verwirrung, da hier eine so schöne Verwechslungsgefahr mit der allgegenwärtigen globalen Perl Variablen $_ besteht.

Der Inhalt der Subroutine sieht sauber aufgeschrieben so aus:

```
  $_ = q;%}<0:%^<uhld,857790311:gns)::;.
      '(zAu<mnb`muhld)%^(:qshoug!#x3j>!ruhmm!`mhwd !$ei-!$el-
!$er!]s#-%^.'.
      q+2711-%uZ0\-%uZ1\:**%^:rmddq!0|+
```

Hier wird also lediglich der globalen Variablen $_ ein reichlich unübersichtlicher Zeichenmüll zugewiesen, wobei diese Zuweisung in drei Substrings zerlegt ist, die mit dem Perl . Operator konkateniert werden.

Bei der Definition der drei Substrings wird von den flexiblen Quoting- Mechanismen von Perl Gebrauch gemacht, d.h. die Wahl des Delimiters für ein Quoting ist durch die Verwendung des q Operators weitgehend frei. In der ersten Zeile wird das ; als Delimiter verwendet, in der letzten das + Zeichen. Die zweite Zeile verwendet zur

Abwechslung das einfache Hochkomma, welches ja als Quoting-Delimiter durchaus üblich ist und bekannt sein sollte.

Dieser Zeichenmüll wird nun vom oben aufgeführten 2. Programmblock bearbeitet. Der RegEx Operator s///, ohne Variablenbezug angewendet, wirkt auf die globale Variable $_, die ja durch den Aufruf der Subroutine nun definiert ist. Auch hier ist die Wahl der Delimiter frei, d.h. vom / wird abgewichen, statdessen wird das ; verwendet. Auch dies dient der Verwirrung, da das ; ja normalerweise einen Perl Befehl abschließt.

Verständlicher aufgeschrieben, sieht diese Substitutions-RegEx so aus:

```
s/(.)/chr(1^ord($1))/eg;
```

Auf jedes Zeichen des Strings wird der Ausdruck chr(1^ord($1)) angewendet, d.h. das niederwertigste Bit des ASCII Codes aller Zeichen wird gekippt. Hierbei handelt es sich also um einen sehr primitiven Verschlüsselungsalgorithus (XOR mit konstantem Schlüssel 1), der aber für die hier erwünschte Verschleierung vollkommen ausreicht. (Zudem hat dieses Verfahren den Vorteil, daß es sowohl zur Ver- als auch zur Entschlüsselung angewendet werden kann, womit klar sein sollte, wie der Erschaffer dieses Programms vorgegangen ist ... ;)

Der reguläre Ausdruck verwandelt den Zeichenmüll in die folgende schon viel besser zu lesende Zeichenkette:

```
  $|=1;$_=time-946681200;for(;;){@t=localtime($_); printf
"y2k?  still alive!  %dh, %dm, %ds \r",$_/3600,$t[1], $t[0];++$_;sleep
1}
```

Na also, da haben wir unser eigentliches Perl Programm. Sauber notiert sieht man: hier werden keine Wunder vollbracht:

```
  $| = 1; $_ = time-946681200; for(;;) { @t = localtime($_);
printf "y2k?  still alive!  %dh, %dm, %ds \r", $_/3600,$t[1],
$t[0]; ++$_; sleep 1 }
```

Das eval am Ende des Programms (Schritt 4) führt diese Perl Code nun aus, da sich eval - ohne Angabe eines Parameters - der globalen Variablen $_ bedient. Voila!

Kurzer Hinweis zu der Zeitberechnung:

Am 01.01.2000 00:00:00 Uhr waren seit dem 1. Januar 1970 exakt 946681200 Sekunden vergangen. Diese Zahl muß also von dem aktuell gelieferten Wert der time() Funktion abgezogen werden, um die Anzahl der vergangenen Sekunden im Jahr 2000 zu erhalten.

Aber da ist doch noch was: die oben als Punkt 5 geführte Zeile!

Sie hat überhaupt keine Funktion, außer mit der URL auf die Y2K Seiten des freien deutschen Internet Satire Magazins ZYN! aufmerksam zu machen. Es handelt sich

hierbei schlicht um einen String-Ausdruck, der anscheinend als zweiter Parameter an das eval übergeben wird und überflüssigerweise auch noch Zeichen enthält, die wie Perl Syntax aussehen.

Die eval Funktion nimmt aber nur *einen* skalaren Parameter entgegen, somit wird das , hier als der Perl Komma Operator interpretiert. Dieser hat in diesem Kontext aber keine Auswirkung, u.a. weil das Programm ja in der Endlosschleife innerhalb des eval's verweilt.

3 Kurze Zusammenfassung: Die Idee

Wie schon anfangs erwähnt: die Idee ist sehr simpel. Das eigentliche Perl Programm wird durch die XOR Verschlüsselung und die komplizierte Definition der Speichervariablen verschleiert. Durch eine einfache RegEx wird das Programm im Speicher dekodiert und anschließend über eval ausgeführt.

Wo möglich, wird von der globalen Variablen $_ Gebrauch gemacht, da diese bei vielen Perl Funktionen als Default Parameter fungiert, und somit zumindest unerfahrenen Perl Programmierern das Verständnis erschwert.

Durch die variablen Quoting-Mechanismen sowohl bei der String-Zuweisung also auch bei der Ausführung des regulären Ausdrucks, war es leicht diesen Vorgang so zu verschleiern, daß der wahre Programmablauf nicht auf den ersten Blick erkennbar ist.

Keine Magie, also! ;)

4 Ein kleiner Tipp zum Schluß

Die Funktionsweise des Programms wird sehr schnell deutlich, wenn man den eval Befehl durch den print Befehl ersetzt. In diesem Fall kann man die komplizierte Verfahrensweise zur Definition des eigentlichen Programmcodes vernachlässigen und betrachtet ohne weiteren Aufwand schlicht deren Ergebnis! ;)