Perl vs SAP im Firmenalltag

Steffen Ullrich `<steffen@genua.de>`
GeNUA mbH
`http://www.genua.de`

Wer sind wir (GeNUA)

- 12 Jahre alte Firma aus Kirchheim bei München
- Hersteller der GeNUGate Firewalls, Consulting, Systemadministration
- ca 70 Mitarbeiter, davon ca. 15 Perl-Entwickler
- 100% BSD/Linux intern

Was ist TNT

- CRM, Trouble-Ticket-System, Auftragsverwaltung, Projektmanagment, Zeiterfassung u.v.m
- Geld in Eigenentwicklung statt in SAP Consultants gesteckt
- 1996 Rewrite von Tcl/Tk nach Perl/Tk
- PostgreSQL als Backend

Was ist TNT

- inzwischen 100000 Zeilen Perl
- **unternehmskritisch**
- z.Z. ca. 0.5 Mann mit Wartung und Weiterentwicklung beschäftigt

wie sieht es aus

Kostentraeger(tnt2:steffen@localhost:9015) (Level: 0-0)

Mittwoch 2.2.2005 - 08:27:20

Person(tnt2:steffen@localhost:9015) (Level: 0-0)

Projektplan(tnt2:steffen@localhost:9015) (Level: 0-0)

Arbeitspaketübersicht für Projekt 70

Maschinenverwaltung(tnt2:steffen@localhost:9015) (Level: 0-0)

Information: 1108 Einträge aus Tabelle Maschine angezeigt

wie geht es weiter im Vortrag....

- vor 3 Jahren übernahm ich grossen Teil der Weiterentwicklung.
- gezeigt wird eine Auswahl der erlebten Probleme
- und der dazugehörigen Lösunganansätze

- Migration zu `use strict`
- Bugbehandlung
- Zugriffskontrolle auf Daten
- Eindämmung des Resourcenverbrauches
- Performanceverbesserungen
- Strategien und Tools der Entwicklung

Am Anfang war das Chaos

- Codebasis stammt von 1996
- zusammengehackt von einem einzelnen Entwickler
- keinerlei `use strict` benutzt
- Zusammenhänge im Code schwer zu erkennen durch mal zuviel und mal zuwenig Variablendeklarationen

Migration zu `use strict`

- erste Aktion nach Übernahme TNT war daher die wichtigsten Module zu `use strict` zu migrieren
- aber es reicht nicht alle Variablen zu deklarieren
- Code musste definitiv weiterhin laufen, da unternehmenskritisch

sanfte Migration

- Idee: sanfte, schrittweise Migration, auch nicht alle Anwender auf einmal
- aber: alle sollten die gleich Codebasis haben

Module `mystrict`

- `strict` mode abschaltbar über Environmentvariable

```
package mystrict;
use strict;
my $do_strict = $ENV{ TNT_STRICT } ? 1:0;

sub import { goto &strict::import if $do_strict }
sub unimport { goto &strict::unimport  if $do_strict }
1;
```

- könnte man noch flexibler machen (per Nutzer oder Module)

Resultate der Migration

- Migration geriet wie geplant soft
- es wurde einiges an Fehlern entdeckt
- der Code wurde verständlicher
- der Chef (d.h. der andere Entwickler) sah sich ebenfalls gezwungen nur noch `strict` zu schreiben

die Message wird erhört

- andere wurden von den offensichtlichen Vorteilen angesteckt
- der GeNUGate Code wurde auch strictified
- und inzwischen auch teilweise mit `use warnings` versehen
- `mywarnings` führt zu Abbruch in Testumgebung und zum Ignorieren der Warnung in Produktivumgebungen

Module `mywarnings`

- in Testumgebung alle
  `use mywarnings args`
  zu
  `use warnings ’FATAL’ args`
  umwandeln
- in Produktivumgebung alle `use mywarnings` ignorieren
- ansonsten verhalten wie `use warnings`

mywarnings

```
package mywarnings;
use strict;
require warnings;
our @ISA = ( 'warnings' );

sub import {
    return if ( $GGLib::showwarn eq 'ignore' ); # ignorieren
    if ( $GGLib::showwarn eq 'fatal' ) { # FATAL dazusetzen
        if ( @_ == 1 ) {
            splice(@_,1,0,'FATAL','all' );
        } else {
            splice(@_,1,0,'FATAL' );
        }
    } # else passthru
    goto &warnings::import;
};
1;
```

Bugs

Bughunting

- Bugs sind normal
- brauchen nicht schlimm zu sein, wenn man vorbereitet ist (in unserem Umfeld!)
- benötigen gute Bugreports

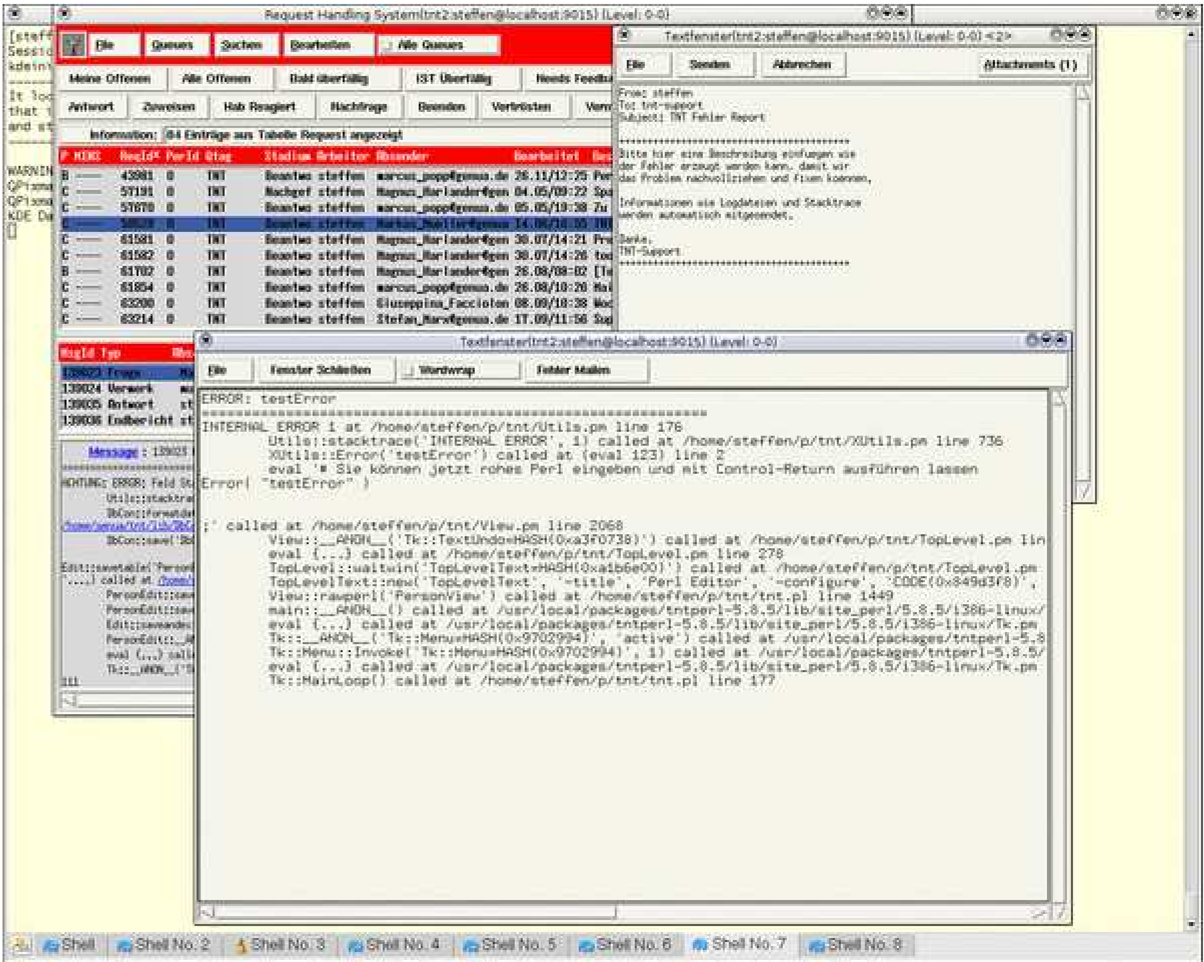
A Bugs Live

- kritische Probleme sollten direkt erkannt werden und zu **kontrolliertem** Abbruch der Aktion führen
- periodische Konsistenzchecks finden unerkannte Bugs im Nachhinein

geordneter Rückzug

- kritische Teile (Rechnungserstellung) sind gespickt mit Konsistenzchecks
- Konsistenzprobleme führen zu Bugreport
- Bugreport umfasst automatisch alle Logfiles, Stacktrace, Versionen...
- Bugreports landen im Trouble-Ticket-System mit einer Reaktionszeit von 4 Stunden

Fehlerfenster



Sanitäter!!!

- jeden Tag läuft ein Konsistenzcheck auf der Datenbank
- Probleme werden damit schnell erkannt
- und anhand der Logfiles kann man meist die Ursache finden
- danach Fixen der Datenbank und des Programms.
- und notfalls gibt es noch Backups :)

Zugriffskontrolle

Zugriffskontrolle - Aufgaben

- normale Nutzer sollen nicht Zugriff auf die Zeiterfassung anderer Nutzer haben
- einige Dokumente sind nicht für alle zum Lesen bestimmt
- alle Nutzeraktionen sollen geloggt werden können

Zugriffskontrolle - Lösungen

- Dokumente werden bei Bedarf PGP verschlüsselt in DB abgelegt
- Logging und Beschränkungen mittels zusätzlichem, forciertem Layer unter Nutzung von `DBI::Proxy` und `DBI::Proxyserver` realisiert:

Migration zu Proxy

- zuerst Migration von `Pg.pm` nach `DBI::Pg`
- anschließend Migration von `DBI::Pg` nach `DBI::Proxy`
- Nutzung eines eigenen Proxyservers (basierend auf `DBI::Proxyserver`), der das Umschreiben von Statements und Loggen übernimmt
- Proxyprozess zeigt in ps-Ausgabe u.a. Name des Nutzers und Version des Clients an

Resourcenverbrauch

Die Zähmung des Monsters

- Applikation wurde immer umfangreicher
- 90% der Leute brauchten nur 10% der Funktionalität
- Lösung: Nachladen bei Bedarf

Ansätze zum Nachladen

- `AutoLoader`: nur unabhängige Funktionen, kein Syntaxcheck vorab
- `Class::Loader`: nur für Objekte
- daher was eigenes

Module `Demand` - Aufgaben

- Änderungen im Code sollen minimal sein
- automatisches Nachladen bei Aufruf einer Methode aus dem Module
- abschaltbar bei Problemen bzw. für Syntaxcheck

Module `Demand` - Implementation

- Basispackage `Demand` mit `AUTOLOAD` Funktion
- für alle nachzuladenden Klassen wird Proxypackage erstellt, das von `Demand` abgeleitet ist
- Beim Aufruf einer Funktion in Proxypackage wird Nachladen getriggert:
  - `@ISA` wird gelöscht
  - das richtige Package wird geladen
  - und die Zielfunktion darin aufgerufen

Module `Demand` - Code 1/3

```
package Demand;
use vars qw( $AUTOLOAD $__NO_STRICT );
# Code vereinfacht fuer Fall, das jedes Module in eigenem File

# use Demand 'Klasse'
# deklariere Klasse und setze @ISA von Klasse auf Demand Module
sub import {
    shift;
    foreach my $pkg (@_) {
      @{ "${pkg}::ISA" } = qw( Demand );
    }
}
```

Module `Demand` - Code 2/3

```
# Funktion aus dem Modul wird aufgerufen:
# lade das richtige Modul und rufe dort die Methode auf
sub AUTOLOAD {
    my ($pkg,$f) = $AUTOLOAD =~m|(.*)::(.*)|;
    __loadModule__( $pkg ) || return;
    my $goto = UNIVERSAL::can($pkg,$f )
        || UNIVERSAL::can($pkg,'AUTOLOAD' )
        || die;
    goto &$goto
}
```

Module `Demand` - Code 3/3

```
# lade das Modul und mache es unabhaengig von Demand
sub __loadModule__ {
    my $pkg = shift;
    @{ "${pkg}::ISA" } = (); # Demand rausnehmen
    eval "require $pkg";     # richtiges Modul laden
    die $@ if($@);
}
```

Module `Demand` - Resultate

- keine Modifikation der nachzuladenden Module nötig
- einmalige Definition nötig, was nachladbar ist:

```
use Demand qw( Bibliothek Reisekosten Schulung Zeiterfassung ... );
use Demand qw( AP<PM> Atgliste<Auftrag> AuftragView<Auftrag> ... );
```

- richtiges Modul noch etwas komplizierter um Vorkehrungen für mehrere Packages per File und gegen Endlosloops bei kaputten Files zu treffen
- geht nicht wenn Module etwas exportiert

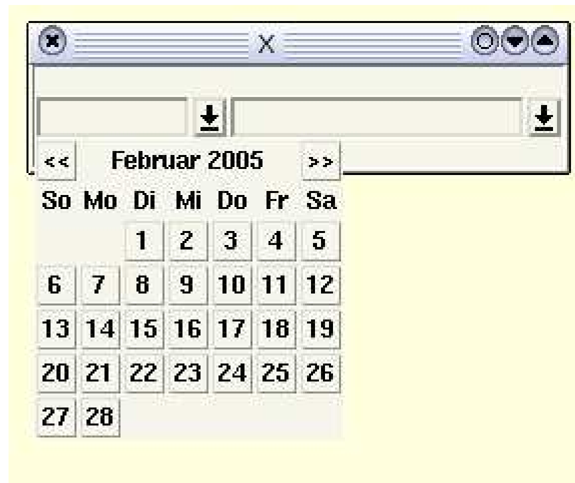
Performance

slow moving target

- Performance von perl/Tk i.a. kein Problem
- Änderung des Algorithmus bringt am meisten
- Optimierung SQL Abfragen

`TK::*Entry` Probleme

- `Tk::BrowseEntry` und `Tk::DateEntry` haben Entry und Button und Popup
- Popup wird erstellt zur gleichen Zeit wie Entry
- Performanceproblem, wenn in Massen (z.B. in Tabellen) eingesetzt



Ersatz für`Tk::*Entry`

- Versuch vorhandene Module abzuändern scheiterte
- Neuschreiben entsprechender Module unerwartet einfach
- neue Module sind nahezu interfacekompatibel mit kleinen Einschränkungen und kleinen Erweiterungen

all you can eat

all you can eat

- Applikation verschlang immer mehr Speicher und gab ihn nicht wieder frei
- zuviele Upgrades und Änderungen in letzter Zeit, sodaß nicht klar war, was die Ursache war

Memory leaks

- Perl räumt seinen Speicher selber auf, aber:
  - zirkuläre Referenzen
  - Code aus eingebundenen XS
- sehr schwer zu finden

Finden von Memory leaks

- Tagging von interessanten Objekten mit "Cookies"

```
$object->{__memcookie__} = MemCookie->new( 'Beschreibung' );
```

- Cookie schreibt raus, wann er erzeugt wurde
- wird `$object` zerstört wird Cookie auch zerstört und schreibt entsprechende Info
- Auswertung des Logfiles ergibt, welche Cookies nicht zerstört wurden und grenzt damit Suche ein

MemCookie - Implementation

```
package MemCookie;
my ($total,$activ);
sub new {
    my ($class,$id);
    $total++; $activ++;
    my $text = $id || "cookie#$total";
    debug( "create cookie $text $activ|$total" );
    return bless \$text,$class;
}
sub DESTROY {
    my $self = shift;
    $activ--;
    debug( "destroy cookie $$self $activ|$total" );
}
```

MemCookie - Resultat

- Bug in `Tk::Callback` (XS), der dazu führte, das in bestimmten Fällen der Refcounter falsch war (Tk800_024)
- Workaround war möglich (`[\&sub]` statt `\&sub` in callback)
- alle happy

Entwicklungsprozess

Entwicklungsstrategien

- Release often, Release early
- Produkt ständig in Beta-Qualität
- nur möglich durch wirksame Fehlerreaktionen
- die man unweigerlich öfter übt :)

Entwicklungstools

- Nachladen von gefixten Modulen in die laufende Applikation (beschränkt)
- beliebige Perl-Kommandos in der laufenden Applikation absetzbar
- Menü gibt Auskunft über die geladenen Module und welche Version (wird auch bei Bugreport mitgeschickt)
- Debugmode zur Laufzeit einschaltbar
- mehrere Testdatenbanken zum Spielen
- Trouble-Ticket-System, Bugzilla

abschließende Worte

menschlich

- ändern 2 Leute am Code
- unterschiedliche Programmierstil
- vi vs. emacs (Einrückungen, tabstop)
- **zuviele Wünsche, zuwenig Zeit**

und schließlich

- Ziel erreicht
- Anwendung macht was wir wollen, nicht wir, was SAP kann
- sehr große Flexibilität durch perl

Leinen los

- Code unter der Artistic License verfügbar
- allerdings ist (wie üblich) die Dokumentation mangelhaft
- und wir haben auch praktisch keine Zeit für Support
- aber ein paar Goodies sind trotzdem dabei

The End