42 Goldene Regeln für Perl-Applikationen

Oder: Wie sichert sich ein Perl-Entwickler vor dem Psychopathen mit der Kettensäge

Über den Autor

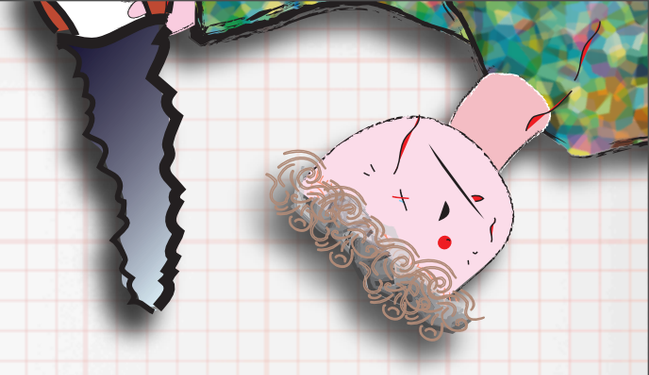
* Alvar C.H. Freude
  * Freiberuflicher Trainer, Software-Entwickler, Berater
  * Diplom-Kommunikations-Designer (FH)
  * http://www.perl-blog.de/
  * http://alvar.a-blast.org/
  * alvar@a-blast.org

Intro

* Viel Code besteht aus Altlasten
  * Perl-Code macht da keine Ausnahme
* Qualitätssicherung wird oft nur als Kostenfaktor gesehen: „Wieso, funktioniert doch!“
  * Dabei spart sie schon während der Entwicklung viel Geld

Goldene Regeln

* Ziel ist:
  * moderner Code
  * wartungsfreundlicher Code
  * robuster Code
  * Argumentationshilfe für den Chef oder Kunden
* Diese Regeln sind nur eine Anregung und nur bei wenigen erlaube ich mir Unfehlbarkeit festzustellen . . .



Vorbereitung

Perl-Version und Entwicklungsumgebung

Regel 1: Aktuelles, eigenes Perl

* Für neue Projekte bietet sich Perl 5.10.x an
* Unter Linux/Unix (inkl. OS X) ein eigenes Perl installieren
  * kollidiert nicht mit dem System (CPAN-Module)
  * anpassbar (Compiler-Switches)
* Beispielsweise unter `/usr/local/perl/myperl`

Regel 2: Erdbeeren vor die Fenster

* Wer schon mit Windows gestraft ist, braucht sich nicht auch noch mit ActivePerl bestrafen!
* Strawberry-Perl bietet:
  * weniger Gefrickel
  * volle CPAN-Integration
  * C-Compiler und so weiter – alles mit dabei

Regel 3: Entwicklungsumgebung

* Nutze eine Entwicklungsumgebung!
  * Kommandozeilenfreaks: vim oder emacs zur IDE aufrüsten (siehe vim-Vortrag)
  * GUI-Krempel meist einfacher
    * Eclipse+EPIC (Perl Plugin), Komodo oder Padre
* Den meisten Editoren wie UltraEdit fehlen viele Funktionen wie Syntaxcheck, Perl::Tidy-Integration, …

Regel 4: Versionskontrollsystem

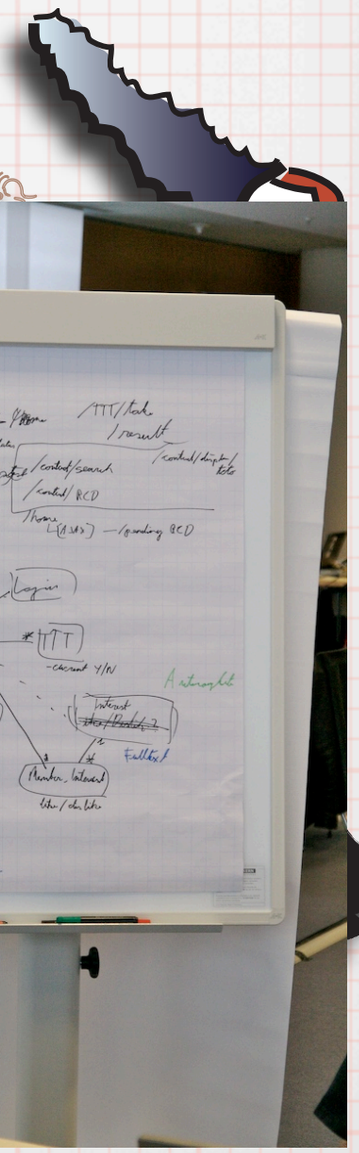
- Nutze ein Versionskontrollsystem!
  - Pflicht im Team und bei mehreren beteiligten Entwicklern!
  - Auch alleine Vorteile
    - weniger Konflikte bei Nutzung mehrerer Rechner
    - Versioniertes Backup
- Subversion, Git, SVK, …

Arbeitsweisen

Saubere Arbeitsweisen sorgen für besseren Code

Regel 5: Code Planen

* Erst ausführlich planen, nicht draufloshacken
  * Wer drauflostippt, wird vieles ungeschickt lösen
* Daher: vorher planen
  * bei größeren Projekten auch mal mehrere Tage
  * Planung aufzeichnen
  * am besten im Team!



Code planen (2)

* Nach der Grobplanung:
  * Am Rechner Module anlegen, mit POD versehen, die Synopsis befüllen
  * Keine Angst vor dem Ändern des Konzeptes!
  * Ausreichend Gedanken machen zur Benennung von Modulen und Funktionen/Methoden. Konsistenz!
  * Anwendung in kleine Teile zerlegen, um monolitischen Code zu vermeiden

Regel 6: Architektur überprüfen

- Die geplante Architektur überprüfen
  - Insbesondere bei großen Projekten einen Architektur-Review durchführen
  - Einfachste Form: einem Kollegen alles erklären und durch ihn überprüfen lassen

Regel 7: Schreibe erst die Tests!

- Schreibe Tests bzw. betreibe Testgetriebene Entwicklung
  - Im Idealfall: vor dem Ausprogrammieren den Test schreiben
  - Mit dem Test die Funktionsfähigkeit des Codes überprüfen
  - Zum Testen *nicht* die Anwendung laufen lassen, sondern Tests schreiben

Regel 8: Objektorientierung nutzen

* Entwickle Objektorientiert *und* Prozedural
  * Nutze die Möglichkeiten von Perl, beides zu mischen
    * Objektorientierung ist flexibler und mittelfristig einfacher
    * Kleine Hilfsfunktionen können aber prozedural sein
      * Dann auch leichter testbar

Regel 9: Code Review

- Führe Code-Reviews durch
  - Wichtiger Bestandteil der Qualitätskontrolle
    - Code mit Kollegen zusammen durchgehen und Schwachstellen oder unverständliche Konstrukte suchen
  - Dies schon bei der Zeitplanung berücksichtigen
    => Der Chef hat es dann schon abgesegnet

CPAN-Tauglichkeit

Schreibe den Code so, als würde er aufs CPAN kommen

Regel 10: Schreibe keine Skripte

* Schreibe keine Skripte, außer zum Starten vom Code in Modulen
  * Wir haben:
    * Einzeiler
    * Skripte
    * Applikationen

Schreibe keine Skripte (2)

- Es ist möglich, 10 000 Zeilen Code in eine Datei zu packen
  - Aber keine gute Idee!
  - Modularer Code ist übersichtlicher, besser wartbar, überhaupt erst testbar, erweiterbar
    - beispielsweise Web-Interface statt CLI

Schreibe keine Skripte (3)

* Einfaches Beispiel-Skript – damit steckt aller relevanter Code in Modulen

```
#!/usr/bin/perl

use strict;
use warnings;

use Project::Tool::App;

Project::Tool::App->run(@ARGV);
```

Regel 11: Nutze *Module::Starter*

* Nutze *Module::Starter* oder ähnliches zum Anlegen von Projekten
  * Erstellt Grundgerüst für ganze Distributionen
    * *Module::Starter::Smart* erlaubt späteres hinzufügen neuer Module
    * *Module::Starter::PBP* erlaubt Templates
  * Ein eigenes *create-module.pl* kann hilfreich sein

Regel 12: Nutze *Module::Build*

- Nutze ein Build-System, beispielsweise *Module::Build*
  - Integration von Test, Code-Coverage-Testing und Packaging
  - Der eigene Code ist CPAN-Kompatibel
  - Neue Entwickler finden sich schnell zurecht

Der Psychopath ...

... mit der Kettensäge

Vorsicht vor dem Psychopathen

* Damian Conway sagt:

  Codieren Sie immer so, als wäre der Typ, der den Code pflegen muss, ein gewaltbereiter Psychopath, der weiß, wo Sie wohnen.

* Zusatz: es handelt sich bei dem gewaltbereiten Psychopathen um den mit der Kettensäge!

* Perl bietet die passenden Werkzeuge gegen diesen freundlichen Herren

Regel 13: Schreibe einfachen Code

* Einfacher Code ist lesbarer, wartbarer und hat weniger Fehler
* Vermeide tief verschachtelte Schleifen und dutzende If-Abfragen in einer Subroutine
* *McCabe-Metrik:* achte auf zu viele verschiedene Ausführungs-Pfade pro Funktion/Methode
  * Testen mit *Perl::Critic* bzw. *Test::Perl::Critic*

Was macht dieser Code?

```
sub MV1IND {my $rc=0; my $err=""; my $mv1pos=0; my $ph="MV1";
my ($out,$xmvkey); my ($mv1bt);
if (not open(MV1,'<'.$mv1dat)) {$rc=2; $err="$ph** open-Error $mv1dat\n"; print PROT $err; print $err}
else { binmode(MV1); ($mv1lm,$mv1n,$mv1ns,$mv1nd)=(1600,0,0,0);
  seek(MV1,$mv1pos,0); $mv1l=read(MV1,$out,$mv1lm); $mv1lm=index($out,"\cJ");
  if ($mv1lm<913) {$rc=2; $err="$ph** Satz 1: MV1-Laenge $mv1lm ist < 913\n"; print PROT $err; print $err;}
  else {$out=substr($out,0,$mv1lm); $mv1l=$mv1lm; $mv1bt=$mv1lm+1; if ($tt) {print PROT "$ph: in Satz 1 ermittelte
MV1-Laenge: $mv1lm\n"}}
  while ($mv1l==$mv1lm) { $mv1n++;
    if (substr($out,0,3) eq $ph) {
      if    ($mvart eq "mv1") {if (substr($out,$pmva,$lmva) ne '000000') {
        $xmvkey=""; for ($v=0;$v<=$#mvxkp;$v++) {$xmvkey=$xmvkey.substr($out,$mvxkp[$v],$mvxkl[$v])}
        if (not defined($mv1{$xmvkey})) {$mv1ns++; $mv1{$xmvkey}=$mv1pos}
        else {$mv1nd++; if ($tt) {print PROT "Key $xmvkey:Satz $mv1n ueberschreibt ".($mv1{$xmvkey}/$mv1bt+1)."\n"}
          $mv1{$xmvkey}=$mv1pos; }}}
      elsif ($mvart eq "mv2") {if (substr($out,$phist,$lhist) eq 'A ') {
        $xmvkey=""; for ($v=0;$v<=$#mvxkp;$v++) {$xmvkey=$xmvkey.substr($out,$mvxkp[$v],$mvxkl[$v])}
        if (not defined($mv1{$xmvkey})) {$mv1ns++; $mv1{$xmvkey}=$mv1pos}
        else {$mv1nd++; if ($tt) {print PROT "Key $xmvkey:Satz $mv1n ueberschreibt ".($mv1{$xmvkey}/$mv1bt+1)."\n"}
          $mv1{$xmvkey}=$mv1pos; }}}
      elsif ($mvart eq "mv3") {if (substr($out,$pgrb,$lgrb) eq '001' and substr($out,$phist,$lhist) eq 'A ') {
        $xmvkey=""; for ($v=0;$v<=$#mvxkp;$v++) {$xmvkey=$xmvkey.substr($out,$mvxkp[$v],$mvxkl[$v])}
        if (not defined($mv1{$xmvkey})) {$mv1ns++; $mv1{$xmvkey}=$mv1pos}
        else {$mv1nd++; if ($tt) {print PROT "Key $xmvkey:Satz $mv1n ueberschreibt ".($mv1{$xmvkey}/$mv1bt+1)."\n"}
          $mv1{$xmvkey}=$mv1pos; }}}}
    else {$rc=2; $err="$ph** Satz $mv1n: beginnt nicht mit $ph\n"; if ($tt>=2) {print "$ph: $mv1n="; printf '<
%*vX>',":","$out"} print PROT $err; print $err;}
    $mv1pos+=$mv1bt;
    seek(MV1,$mv1pos,0); $mv1l=read(MV1,$out,$mv1lm);
  } close MV1;
  if ($mv1l > 0) { $rc=2; $err="$ph** Datei MV1 endet mit Satzfragment ".($mv1n+1)." der Laenge $mv1l<>$mv1lm\n";
print PROT $err; print $err}
  if ($tt) {$v=keys(%mv1); print "$ph-$mvart: $mv1n Zeilen, $v=$mv1ns Keys, $mv1nd ueberschrieben\n";
    print PROT "$ph-$mvart: $mv1n Zeilen, $v=$mv1ns Keys, $mv1nd ueberschrieben\n"};
  if ($v<1) {$rc=2; $err="$ph-$mvart: keine Keys ermittelt bei Satzlaenge $mv1lm. Abbruch\n"; print PROT $err; print
$err; exit $rc};
  if ($tt>=3) {foreach $v (keys(%mv1)) {print PROT "$v:$mv1{$v} "} print PROT "\n"};
}
if ($tt>=2) {@time=reverse(localtime(time)); print PROT "$ph: @time\n"}
return $rc }
```

Regel 14: Keine globalen Variablen

* Nutze keine globalen Variablen
  * Meist überflüssig
  * Auf gar keinen Fall zur Parameterübergabe an Funktionen oder Methoden nutzen!
* Paket-globale Variablen können manchmal sinnvoll sein

Regel 15: Nutze *Perl::Tidy*

* Formatiere den Code mit Perl::Tidy
  * Dies bietet saubere Code-Formatierung nach vorgegebenen Regeln
* Die Regeln aus *Perl Best Practices* bieten sich an
  * Bei Bedarf auch eigene
* In Eclipse: *Shift-Command-F*

Erahnbar, was der Code macht

```
sub MV1IND
   {
   my $rc     = 0;
   my $err    = "";
   my $mv1pos = 0;
   my $ph     = "MV1";
   my ( $out, $xmvkey );
   my ($mv1bt);

   if ( not open( MV1, '<' . $mv1dat ) )
      {
      $rc  = 2;
      $err = "$ph** open-Error $mv1dat\n";
      print PROT $err;
      print $err;
      }
   else
      {
      binmode(MV1);
      ( $mv1lm, $mv1n, $mv1ns, $mv1nd ) = ( 1600, 0, 0, 0 );
      seek( MV1, $mv1pos, 0 );
      $mv1l = read( MV1, $out, $mv1lm );
      $mv1lm = index( $out, "\cJ" );

      if ( $mv1lm < 913 )
         {
         $rc  = 2;
         $err = "$ph** Satz 1: MV1-Laenge $mv1lm ist < 913\n";
         print PROT $err;
         print $err;
         }
      else
         {
         $out   = substr( $out, 0, $mv1lm );
         $mv1l  = $mv1lm;
         $mv1bt = $mv1lm + 1;
         if ($tt) { print PROT "$ph: in Satz 1 ermittelte MV1-Laenge: $mv1lm\n" }
         }
      while ( $mv1l == $mv1lm )
         {
         $mv1n++;
         if ( substr( $out, 0, 3 ) eq $ph )
```

Regel 16: Kommentiere ausführlich

* Kommentiere ausführlich, insbesondere was *warum* gemacht wird.
  * Jedes Modul und jede Funktion bzw. Methode sollte mit POD dokumentiert sein
    * *Test::Pod* testet die korrekte Syntax
    * *Test::Pod::Coverage* testet, ob jede Funktion/Methode dokumentiert ist
    * *Module::Starter* legt entsprechende Tests an

Regel 17: Namenskonventionen

* Benenne Variablen, Packages und Dateien nach den üblichen Perl-Regeln
  * Variablen und Subroutinen klein schreiben, Unterstrich als Wort-Trenner
  * *$i*, *$x* und *$xnr* sind nicht aussagekräftig
  * Den Typ der Variable nicht in den Name packen:
    * $*StrName* sagt nicht mehr als $*name*

Namenskonventionen (2)

* Packages und Module sollten immer den gleichen Namen haben, nicht sowas:

```
use Altlast::User;

[...]

my $user = User->new;
```

* Das Package im Modul sollte also *Altlast::User* heißen.

Regel 18: Altlasten

* Lege Altlasten in einem anderen Suchpfad ab
  * Alte Module, die weitergenutzt werden müssen, in einem anderen Suchpfad ablegen, und diesen mit *use lib "$Bin/../../lib-altlast";* einbinden
    * Können bei *Test::Perl::Critic* etc. übersprungen werden
* Alternativ: ganz außerhalb des Projektes lagern

Regel 19: Perl Best Practices

* Lese Perl Best Practices
  * Damian Conway stellt darin viele Regeln für guten Code auf
  * Nicht alle sind Pflicht, manche Quatsch und andere veraltet
  * Aber: gut als Anregung für eigene Regeln

Regel 20: *Test::Perl::Critic*

* Nutze Perl::Critic und Test::Perl::Critic
  * Testet rudimentär die Code-Qualität
  * Mahnt gefährliche Konstrukte an
  * Neuer Code: Mindestens mittlere Strenge (Severity 3)
  * Einzelne Tests (beispielsweise weil andere POD-Überschriften verwendet werden) lassen sich ausschalten

Regel 21: Schreibe Tests!

* Schreibe Tests, um den Code und Teile zu testen
  * Für den Einstieg: *Perl Testing: A Developers Notebook* von Ian Langworth und chromatic
  * Es gibt viele gute Test-Module auf dem CPAN, beispielsweise *Test::Exception* oder *Test::MockModule*
  * Mit automatischen Tests spart man Zeit!

Regel 22: Teste klein und groß

* Teste einzelne kleine Teile und das Große Ganze
  * Kleine Teile testen ist einfach
    * Daher: teile den Code in kleine Unterfunktionen auf
  * Dennoch: auch den Gesamtablauf einer Applikation testen
    * Zum Beispiel: Ein Array aufbauen mit Eingabedaten, Soll-Wert, Test-Name; dann dies einzeln testen

Beispiel für Test-Eingabe-Daten

```
my @blast_tests = (
    {
        text => "Hier ein Test.",
        soll => q{Hier ein <a href="/test/">Test</a>.},
        info => "Ein einzelner einfacher Link",
    },
    {
        text => "Hier Doppel-Wort-Versuch.",
        soll => q{Hier <a href="/doppel-wort/">Doppel-Wort</a>-Versuch.},
        info => "Mehr-Wort-Stichwort",
    }
 );


foreach my $test (@blast_tests)
    {
    local $TODO   = $test->{TODO} if $test->{TODO};
    my    $result = $blaster->blast( $test->{text} );
    is( $result, $test->{soll}, "Blast: $test->{info}" );
    }
```

Regel 23: Kein *printf*-Debugging

* Schreibe Tests, anstatt die Applikation bei jeder Änderung laufen zu lassen
  * *print* bzw. *printf*-Debugging sieht einfach aus, aber:
    * ist sehr mühselig, fehleranfällig, zeitaufwendig
    * verlangt eine komplette Umgebung
  * Tests sind sinnvoller, leichter erweiterbar, finden Fehler bei nachträglicher Code-Änderung usw.

Regel 24: Nutze *Devel::Cover*

* Nutze Devel::Cover, um zu schauen wieviel vom Code getestet wird
  * Zeigt, wieviel vom Code getestet wurde
    * Inkl. Verzweigungen *(if)*, Bedingungen *(and/or)* usw.
* Integration in *Module::Build:*

```
./Build testcover
```

Devel::Cover-Ausgabe

* Beispiel-Ausgabe von *Devel::Cover*

Coverage Summary

file:///Users/alvar/Documents/svn/blaster/Blaster

Google

Coverage Summary

Database: /Users/alvar/Documents/svn/blaster/Blaster3/cover_db

file	stmt	bran	cond	sub	pod	time	total
blib/lib/Blaster3.pm	59.6	7.1	0.0	81.8	100.0	3.0	52.6
blib/lib/Blaster3/BlastEngine.pm	100.0	100.0	100.0	100.0	100.0	97.0	100.0
Total	84.9	50.0	50.0	91.3	100.0	100.0	80.1

Das CPAN gehört zu Perl dazu!

Regel 25: Nutze das CPAN

* Gibt es eine Lösung für mein Problem auf dem CPAN?
  * Das CPAN hat stand 15. Januar 2009 16822 Distributionen und 64628 Module. Am 24. Februar waren es schon 17112 Distributionen und 65718 Module.
* Vieles ohne CPAN gar nicht möglich, DBI zum Beispiel
* Nutze es!

Regel 26: Sei aktuell

* Halte Perl und die CPAN-Module aktuell
* Zwei Philosophien:
  * Einmal installieren, dann *never touch a running system*
  * Oder: regelmäßig aktualisieren und aktuell halten
* Ersteres lässt sich oft nicht durchhalten
  * z.B. neue Module und Ärger mit anderen veralteten
* Daher in der Praxis häufiges aktualisieren meist besser

Regel 27: Lokaler CPAN-Mirror

* Installiere bei Bedarf einen lokalen CPAN-Mirror
  * z.B. wenn wegen Firewall nicht jede Maschine raus darf
* *CPAN::Mini* macht den Mirror einfach
  * Mirror nutzbar z.B. via *file://* oder *http://*
* Obacht mit Virenscannern: offizieller Test-Virus im ClamAV-Modul

Moderne Techniken

Nutze moderne Perl-Techniken

Regel 28: Verteile Code mit PAR

* Nutze *PAR* zum Verteilen von Code
  * PAR kann nicht nur Binaries erstellen
  * Zum Deployment: PAR-Archiv auf Webserver

```
use PAR;
use lib qw(http://code.intern/non-prod/mein-code.par);
```

Regel 29: Tests und Staging

* Stages mittels PAR und Subversion nachbilden
  * Stages Devel, Test, Non-Prod, Prod
    * Ab Test Code in PAR-Archiv packen und via HTTP verteilen
  * Nach Freigabe in die nächste Stage

* Detail-Beschreibung im Tagungsband

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

=head1 NAME

Beispiel-Applikation

=head1 BESCHREIBUNG

Diese Applikation lädt je nach Stage den passenden Code (Module) und führt diese dann aus

$ENV{REPOSITORY_URL} kann zum Beispiel sein: "http://code-server.local/svn";

=cut

use strict;
use warnings;

# Schon zur Compile-Time ausführen, daher BEGIN-Block
BEGIN
{
   if ( $ENV{REPOSITORY_URL} )
      {
      # Wenn der Code aus dem Subversion via PAR kommt:
      my $stage = $ENV{STAGE} || die "Env STAGE nicht angegeben!\n";
      my $fetch_path = "$ENV{REPOSITORY_URL}/par/$stage/project-tool.par";

      # Zur Laufzeit ausführen wegen der URL, daher eval
      eval q{
              use PAR;
              use lib '$fetch_path';
              return 1;
              } or die "PAR nicht installiert?\n $@";
      }
   else
      {
      # Wenn der Code von lokal kommen soll, Verzeichnisse einbinden
      eval q{
              use FindBin qw($Bin);
              use lib "$Bin/../lib";
              use lib "$Bin/../../Altlasten/lib";
              };
      }
} ## end BEGIN

use Project::Tool::App;                    # Applikation laden; kommt aus dem PAR-Archiv oder von lokal
exit Project::Tool::App->run(@ARGV);       # Applikation starten und deren Rückgabewert zurückgeben
```

Regel 30: Nutze *Moose*

- *Moose* bietet ein modernes Objektsystem für Perl 5
  - Mit allem Schnickschnack
    - Einfach, komfortabel, mächtig
  - Basiert auf dem Perl 6 Objektsystem
  - Moose hat Attribute, Roles, Typ-Überprüfung, Methoden-Modifier, Delegation, …

Regel 31: Oder *Class::Accessor*

* . . . oder wenigstens Class::Accessor
  * Erleichtert die Erzeugung von Accessoren
    * AUTOLOAD funktioniert oft auch, macht aber Ärger
  * Lässt sich problemlos nachrüsten
* Alternative Inside-Out-Objekte, wenn der Nutzer vor dem Zugriff auf Internas geschützt werden muss

Regel 32: Damian Conways Module

* Nutze keine Module von Damian Conway ...
* ... oder nur nach ausgiebigen Tests.
  * Tolle Ideen, wichtiges Buch
  * Aber Module werden schlecht gepflegt
  * Offensichtliche Inkompatibilitäten
  * Keine Reaktion auf Patches

Regel 33: Nutze *DBIx::Class*

* Frickle Dir keinen eigenen ORM, nutze DBIx::Class (o.a.)
  * ORM-Wrapper sind hilfreich, um Datenbankabfragen objektorientiert zu kapseln
  * Das Handling kann praktisch sein, z.B. beim Weiterreichen im Code
  * Manuell ein ORM schreiben ist Wahnsinn
  * Komfort wird mit schlechterer Performance erkauft

Regel 34: Nutze *File::HomeDir*

* *File::HomeDir* ist praktisch, um systemunabhängig die korrekten Pfade für Dokumente, Einstellungen, Musik, Dokumente, … herauszufinden

* Manuelles Suchen keine gute Idee
  * Zum Beispiel ist nicht überall die Environment-Variable HOME vorhanden!

Regel 35: Nutze Perls Möglichkeiten

* Nutze die Möglichkeiten, die Perl bietet
  * Nicht Perl wie C oder die Bash nutzen.
    * Perl hat Hashes, komplexe Datenstrukturen, Reguläre Ausdrücke, Objektorientierung usw.
    * C-Style-Code ist schwer wartbar
  * Nicht die 100. Version von *join* oder den 1000. Kommandozeilen- oder Konfigurationsdateien-Parser bauen!

Regel 36: Temporäre Dateien

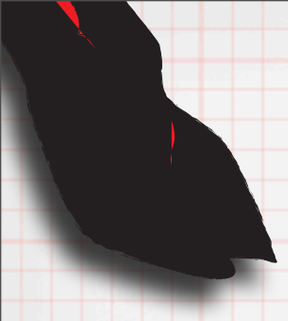
* Man kann sich prima unnötige Arbeit machen …
* => Lege temporäre Dateien nicht manuell an!
* *File::Temp* legt temporäre Dateien und/oder Ordner sicher und zuverlässig an

```
my $dir = tempdir( CLEANUP => 1 );   # Dir. mit Auto-Aufräumer

my $file_handle = tempfile();        # temporäres File-Handle
```

Regel 37: Nutze *Log::Log4perl*

- Schreibe Logs mit Log4perl
  - Ein *print* ist schnell hingeklatscht, aber nicht flexibel
  - Log4perl ist ein umfangreiches Logging-Framework
    - Verschiedene Log-Level
    - Verschiedene Log-Ziele wie Dateien, Screen, DBI, …
    - Sehr flexibel, gut erweiterbar



Web-Entwicklung

Perl ist eine ausgereifte Sprache mit vielen Möglichkeiten. Manche davon sind weniger gut …

Regel 38: Schreibe keine CGIs!

- Übliche CGIs werden schnell unübersichtlich und schwer wartbar
  - Beispielsweise durch Wiederholungen
- Schreibe zur Not ein CGI-Skript für den Start der Anwendung
  - Der Rest wird in Module gepackt

Regel 39: FastCGI zur Beschleunigung

* Nutze FastCGI zur einfachen Beschleunigung von CGIs
  * Startup-Overhead fällt weg, Bytecode bleibt im RAM
  * Globale Variablen vermeiden – außer es gibt einen Grund – die bleiben zwischen den Requests da
* Vorsicht mit Speicherverbrauch!

Regel 40: *mod_perl*

- Nutze mod_perl für hohe Performance und spezielle Anwendungen
  - CGI-Beschleunigung ist nur Abfallprodukt
  - Kompletter Zugriff aufs Apache-API
    - Beliebige Apache-Module in Perl
  - Bei Bedarf sehr hohe Performance mit Perl-Handlern
  - Erzwingt (teilweise) modulare Struktur

Regel 41: Kein HTML im Code

* Kein HTML im Code, nutze Template Engines
  * Weniger Arbeit durch Trennung von Code und Ausgabe
  * *HTML::Template::Compiled* ist sehr schnell unter persistenten Umgebungen
  * *Template::Toolkit* und *Template::Alloy* sind sehr mächtig
  * *HTML::Mason* und *HTML::Embperl* sind mehr als ein Template-System, verleiten aber auch dazu Code und Ausgabe zu mischen

Regel 42: Nutze Web-Frameworks

* Meist sinnvoll ein fertiges Web-Framework zu nutzen
  * z.B. *Catalyst, Gantry, Mojo, CGI::Application* usw.
    * Vorsicht: Einarbeitungszeit nötig
* Manchmal ist es sinnvoll, auf ein fertiges Framework zu verzichten. z.B. wegen hoher Performance-Anforderung
  * => Eigenes Mini-Framework erstellen, CPAN nutzen!

Schlusswort

Das Wort zum Ende