Qualitätssicherung

bei Softwareentwicklung, Automation und Skripting – oder: wie sichert man sich 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

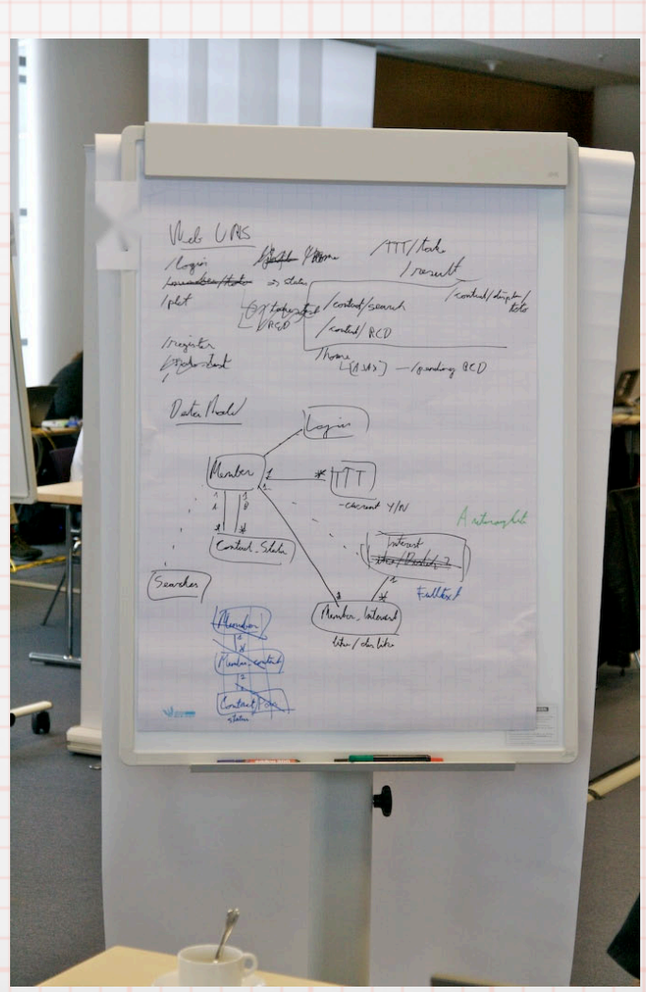
- Viel Code besteht aus Altlasten
  - Skripting im Bereich der Systemadministration macht da keine Ausnahme, oft besonders viele Altlasten
- Qualitätssicherung wird oft nur als Kostenfaktor gesehen: „Wieso, funktioniert doch!“
  - Dabei spart sie schon während der Entwicklung viel Geld

Arbeitsweisen

Saubere Arbeitsweisen sorgen für besseren Code

Code Planen

- Erst ausführlich planen, nicht draufloshacken
  - Wer drauflos tippt 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 Code/Module anlegen, mit Dokumentation 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

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

Schreiben Sie erst die Tests!

- Schreiben Sie Tests / 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

Objektorientierung nutzen

- Entwickeln Sie Objektorientiert *und* Prozedural
  - Nutzen Sie die Möglichkeiten von Skriptsprachen, beides zu mischen
- Objektorientierung ist flexibler und mittelfristig einfacher
- Kleine Hilfsfunktionen können aber prozedural sein
  - Dann auch leichter testbar

Code Review

- Führen Sie 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

Community Stil verwenden

* Verwenden Sie die Techniken, die in der jeweiligen Community üblich sind
  * Bei Perl:
    CPAN verwenden, CPAN-Techniken verwenden
    Code so schreiben, dass er auch aufs CPAN hochgeladen werden könnte
  * Vorteil: etablierte Standards; alte und neue Entwickler finden sich schneller zurecht

Der Psychopath . . .

. . . mit der Kettensäge

Vorsicht vor dem Psychopathen

* Damian Conway, Autor von *Perl Best Practices*, 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!

* Nutzen Sie die passenden Werkzeuge und Techniken gegen diesen freundlichen Herren

Schreiben Sie einfachen Code

* Einfacher Code ist lesbarer, wartbarer und hat weniger Fehler
* Vermeiden Sie tief verschachtelte Schleifen und dutzende If-Abfragen in einer Subroutine
* *McCabe-Metrik:* auf zu viele verschiedene Ausführungs-Pfade pro Funktion/Methode achten
    * 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 }
```

Nutzen Sie *Perl::Tidy*

- Perl-Entwickler: Formatieren Sie Ihren 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 )
```

Keine globalen Variablen

* Nutzen Sie keine globalen Variablen
  * Meist überflüssig
  * Auf gar keinen Fall zur Parameterübergabe an Funktionen oder Methoden nutzen!
  * Machen den Code unübersichtlich und bergen Gefahr von Fehlern
* Paket-globale Variablen können manchmal sinnvoll sein

Ausführlich Kommentieren

* Kommentieren Sie ausführlich, insbesondere was *warum* gemacht wird.
  * Jedes Modul und jede Funktion bzw. Methode sollte so dokumentiert sein, dass daraus eine HTML/PDF-Dokumentation erstellt werden kann (Perl: POD)
    * *Test::Pod::Coverage* testet, ob jede Funktion/Methode dokumentiert ist
    * *Module::Starter* legt entsprechende Tests an

Namenskonventionen

- Benennen Sie Variablen, Packages und Dateien nach den üblichen Regeln
  - Perl: 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.

Altlasten

- Legen Sie 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

Perl Best Practices

* Lesen Sie 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

Nutzen Sie *Test::Perl::Critic*

* Nutzen Sie 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

Schreiben Sie Tests!

* Schreiben Sie Tests, um den Code und Teile zu testen
  * Für den Einstieg, Perl: *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!

Testen Sie „klein“ und „groß“

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

Kein *printf*-Debugging

- Schreiben Sie 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.

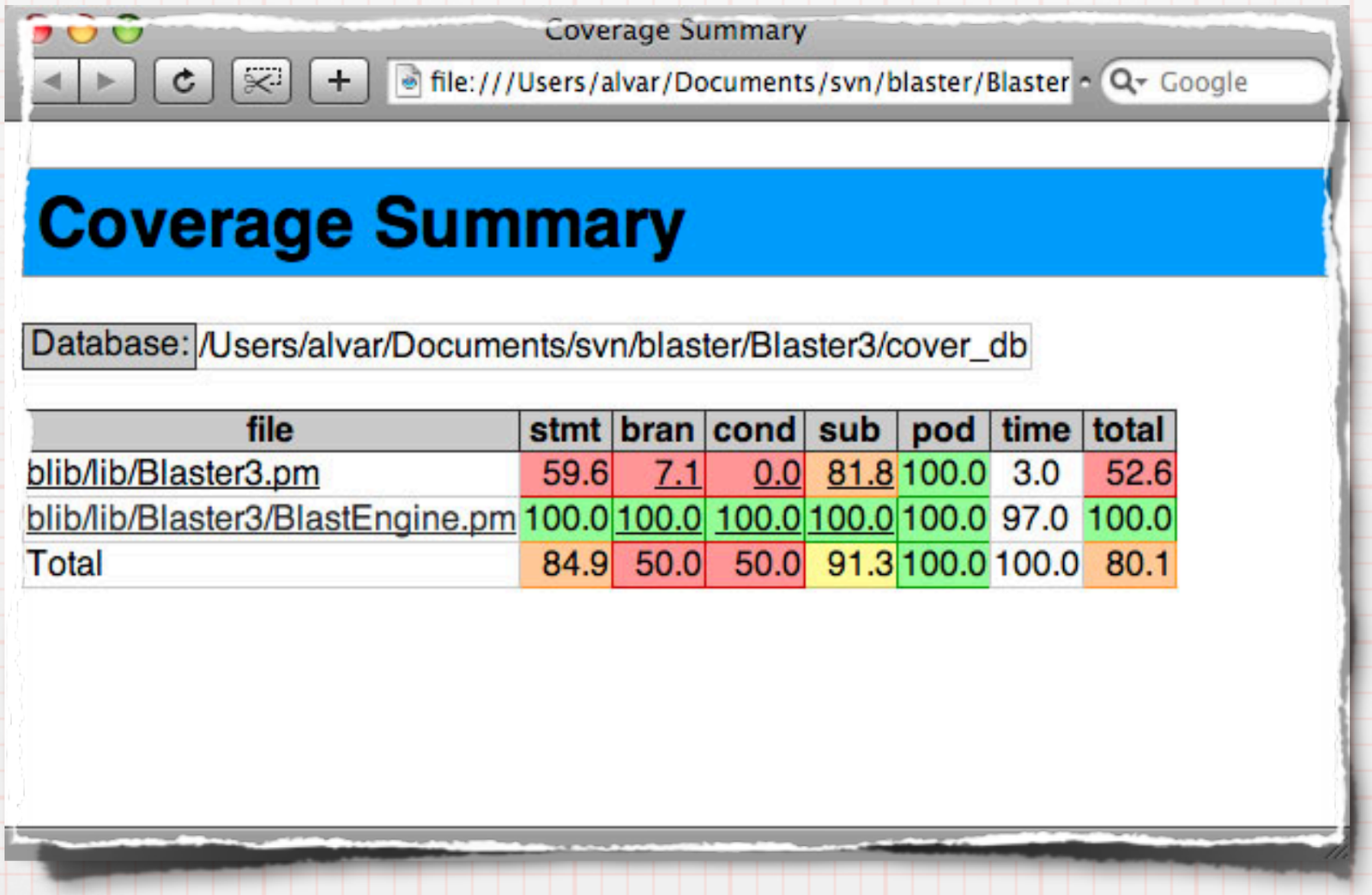
Nutzen Sie *Devel::Cover*

* 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*



Was ist hier falsch?

```
my $config_data = $self->dbh->resultset("config_delta")->search(
    {
    stage      => $app->stage   || die "Stagename fehlt\n",
    release    => $app->release || die "Anwendungs-Release fehlt\n",
    region     => $app->region  || die "Region fehlt\n",
    });
```

Das macht der Code:

```
my $config_data = $self->dbh->resultset("config_delta")->search(
    {
    stage => $app->stage || die
             (
             "Stagename fehlt\n",
             "release",
             $app->release || die
                (
                "Anwendungs-Release fehlt\n",
                "region",
                $app->region  || die "Region fehlt\n"
                )
             ),
});
```

So funktioniert es wie gewollt

```
my $config_data = $self->dbh->resultset("config_delta")->search(
   {
   stage      => ( $app->stage   || die "Stagename fehlt\n" ),
   release    => ( $app->release || die "Anwendungs-Release fehlt\n" ),
   region     => ( $app->region  || die "Region fehlt\n" ),
   });
```

Schlusswort

Das Wort zum Ende