Perl [BP]est Practices

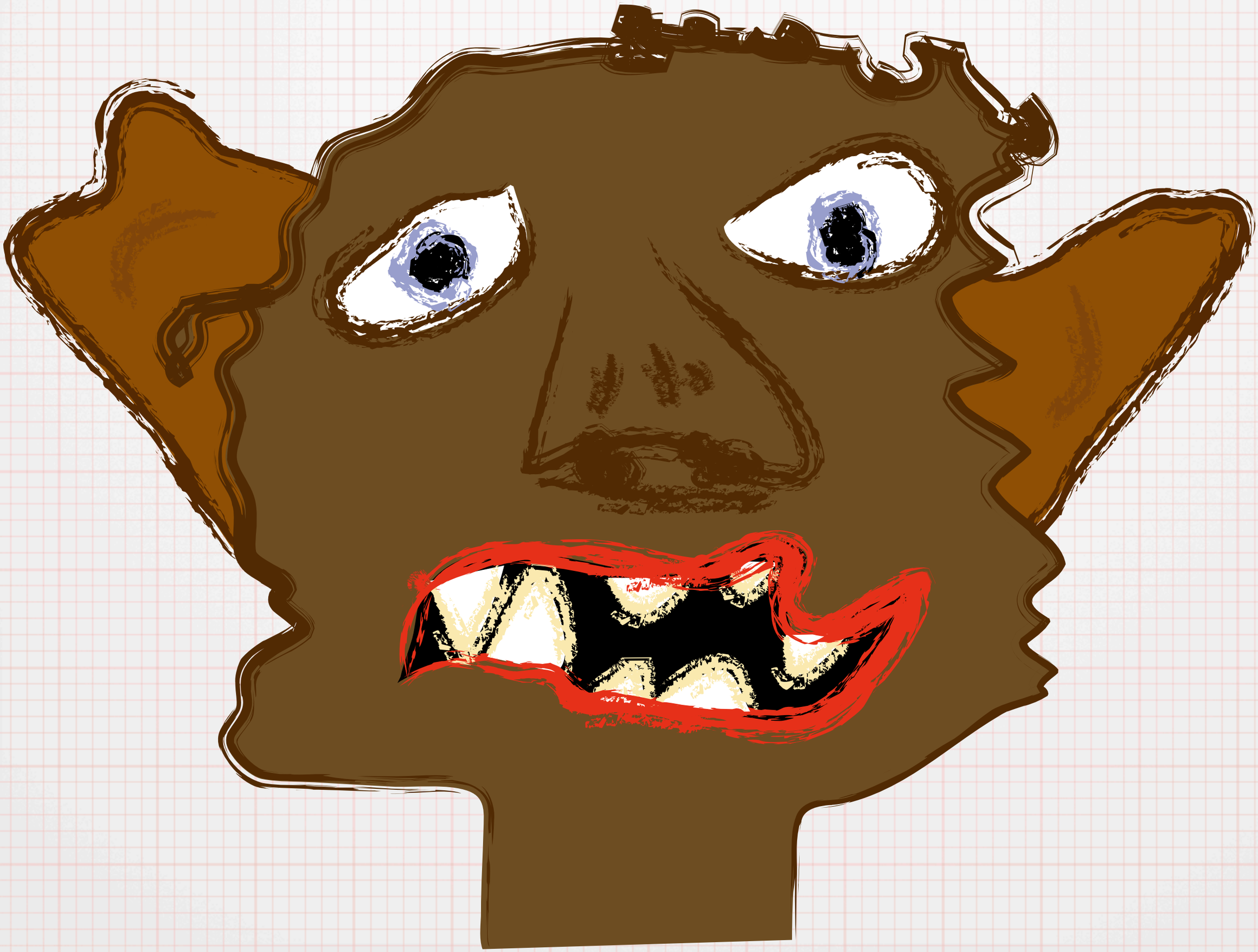
Vorschläge für gute Programmierpraktiken
für wartbaren, effizienten und robusten Code

Deuscher Perl-Workshop 9.0 / 2007

Alvar C.H. Freude

Perl-Code ist häßlich und unleserlich, nicht wartbar und konfus …

* Das wird zumindest gerne behauptet
* Wir Perl-Nutzer wissen zwar, dass das nicht stimmt
* Aber dennoch ertappen wir uns hin und wieder, selbst unschönen Code zu produzieren
* Oder wir müssen fremden Code warten, der von jemandem entwickelt wurde, der in jeder Sprache komisches Zeug schreiben würde



Zwei Typen von schlechtem Code

* A) der generell schlechte
* B) Perl-spezifischer schlechter Code
* Variante B sieht man zum Glück seltener, denn meist sind es eigenwillige Entwickler, die komischen Code schreiben – und der ist dann eben auch in Perl entsprechend komisch

Besserer Code zeichnet sich aus durch

- Robustheit
  - Probleme verhindern (_ref); Verwendung einfach erweiterbarer Konstrukte; Fehlerbehandlung
- Effektivität
- Wartbarkeit
  - Boehms Gesetz: Wartungskosten vier mal höher als Entwicklungskosten
  - Ist der Code nach sechs Monaten noch verständlich?

Perl Pest Practices

* Das Buch Perl Best Practices von Damian Conway ist jedem Perl-Entwickler zu empfehlen
  * Es gibt auch eine (gleichnamige) deutsche Version
* Viele Sachen betreffen nicht nur Perl
* Man muss sich nicht an alles halten, aber sollte sich Gedanken machen

Pflicht!

* use strict;
* use warnings;
* bei Bedarf: Taint-Mode
  * Code wird langsamer, aber zur Entwicklungszeit ist das erstmal egal
  * Manchmal schwierig, z.B. manchen Modulen

Dokumentation

* POD!
* Nach HTMLwandeln, z.B. mit *Pod::ProjectDocs*
* Syntax:
  * perldoc perlpod
  * perldoc perlpodspec
  * Testen mit: podchecker
  * Tests bei Module::Build/Module::Starter

Grundgerüst

* *Module::Starter* baut ein Grundgerüst für Perl-Projekte auf
  * OK, nur für Module – aber alles was mehr als ein paar Zeilen Code hat, sollte sowieso in Module wandern
* Zusammen mit *Module::Build* hat man dann auch gleich ein komfortables Build-System, z.B. auch
  * ./Build testcover
  * ...

Code-Layout

* Perl zwingt den Entwickler nicht, ein bestimmtes Code-layout einzuhalten
  * Das ist gut so – aber Freiheit bedeutet Verantwortung
* *Perltidy* hilft, den Code nach vorgegebenen Regeln automatisch zu formatieren
* (Details: siehe Tagungsband Seite 236)

Klammerung

```
if ($test) {
    $test_counter++;
    print "Testzähler: $test_counter\n";
}
```

* Damian Conway empfiehlt die K&R-Regel für Klammern
* Das spart eine Zeile gegenüber der BSD/GNU-Methode

```
if ($test)
{
    $test_counter++;
    print "Testzähler: $test_counter\n";
}
```

Klammerung nach BSD/GNU-Methode

Klammerung (3)

```
if ($test)
    {
    $test_counter++;
    print "Testzähler: $test_counter\n";
    }
```

- Eine andere Lösung ist, Klammern und Code auf die gleiche Ebene zu packen
- Oft übersichtlicher, insbesondere bei vielen Klammern
- Geschmackssache!

Zwischenkommentar

```
#
# Spezialbehandlung Blabla
# und Abspeichern der Zwischenwerte
#

foreach my $obst (@obst)
    {
    next if $obst eq 'Banane';


    $db_h->do('INSERT INTO obst (name) VALUES (?)', undef, $obst)


    verwurste_obst($obst) if $obsthash{lc($obst)};

    }
```

Kommentare an gleicher vertikelen Positiom

```
# bearbeite alle Obstsorten

# Herrn Müllers Banane ignorieren

# speichern; ID und Datum macht die DB selbst

# Nur Spezialobst darf verwurstet werden,
# Obacht: Updates d. Liste bei Kollege Mayer!
```

bei zu langem Code:
oben drüber

Auch mal in zwei Zeilen

Inhalt: kommentiere, *warum* was gemacht wird
und welche Besonderheiten es gibt

Positionierung von Kommentaren

```perl
my $status = write_bla_to_database
    (
    $db_h,
    'Blatext Blablablubb && Foo && Bar',
    $user_id,
    $bla,
    {},
    'default',
    1,
    );
```

Unterfunktionen mit vielen Parametern werden irgendwann verwirrend

```
my $status = write_bla_to_database
   ({
   db_h       => $db_h,
   blatext    => 'Blatext Blablablubb && Foo && Bar',
   user_id    => $user_id,
   bla        => $bla,
   super_flag => 1,
   });
```

Mit Hashes oder Hash-Referenzen ist es übersichtlicher und flexibler

```perl
sub write_bla_to_database
   {
   my $arg_ref = shift;
   my %params =
      (
      value_hash => {},            # Default-Wert für value_hash
      foo        => 'default',     # Default-Wert für foo
      bla        => 'defaultbla', # Default-Wert für bla
      %$arg_ref                    # alle anderen Werte; diese überschreiben
                                   # bei Bedarf die Default-Werte

      );

   [...]

   }
```

Paremeter in Funktion auslesen und mit Default-Werten versehen

Namenskonventionen

- Manchmal kann man beim Suchen von Namen für Variablen, Funktionen, Module/Klassen und so weiter viel Zeit aufwenden
- Anständige Namen sind zur Vermeidung von vielen Problemen sinnvoll, daher sollte man sich auch etwas Zeit dafür nehmen
- Aber es sollte auch einheitlich sein, denn eine Variable `$max_geschwindigkeit` und die nächste `$AbstandMax` oder `$mxabst` oder `$Xmaximal` zu nennen ist verwirrend

Beispiele für Namen (genauer im Tagungsband)

* Disk::DVD::Rewritable
  HomeServer::Disk::DVD::Rewritable
* $naechser_client (nicht: $naechstes_elem)
  $voriger_termin (nicht: $voriges_elem)
* sub generiere_profil
  sub generiere_ausführungs_profil_mit
* generiere_ausführungs_profil_mit($naechter_client);

Perl hilft uns für übersichtlichen Code

* Quoting-Operatoren:

```
print "STATUS=\"$status\"\n";      # unübersichtlich

print 'STATUS="'.$status.'"'."\n"; # noch schlimmer

print qq{STATUS="$status"\n};      # übersichtlichste Variante mit qq
```

Reguläre Ausdrücke mit x

* Reguläre Ausdrücke, die nicht auf den ersten Blick klar sind, auf mehrere Zeilen verteilen:

```
my ($conf, $value) = m{      # Gefundene Stellen zuweisen
                     ^\s*          # Beginn: beliebig viele (auch keine) Whitespaces
                     ([\w\d]+)     # (1): Dann ein oder mehrere Buchstaben oder Zahlen
                     \s*           # Eventuelle Whitespace
                     :             # der Doppelpunkt
                     \s*           # eventuell wieder whitespaces
                     (.*?)?        # (2): beliebiger Text non greedy
                     \s*           # wieder evtl. whitespace
                     (:?\#.*)?     # eventueller Kommentar (nicht einfangend)
                     $             # Zeilenende
                     }x;           # x-modifier erlaubt Kommentare und mehrzeilige Regexe
```

Tests

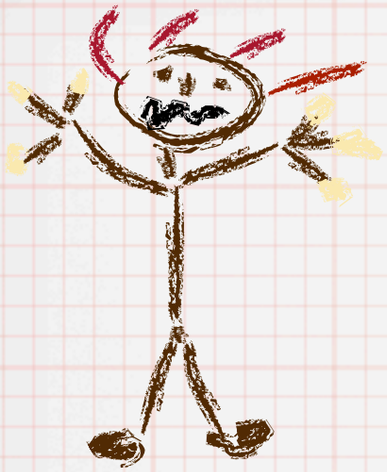
* Da hatten wir ja schon einen Vortrag:
* Macht Tests!
* Auch wenn es schwer fällt: Versucht es zumindest!

Kommandozeilen-Parameter

* Konsistente Parameter bei Kommandozeilen-Programmen nutzen
* *Getopt::Long* hilft bei den parametern
* *Pod::Usage* generiert automatisch aus der Dokumentation Hilfe und Manual

OO

* Damian empfiehlt *Class::Std*
* Aber: das ist gleich ein ganzer Haufen kram, nicht unumstritten
* Eine Alternative ist *Object::InsideOut*
* (Vorsicht: die große Weisheit:) Wichtig ab mittleren Projekten: gute Planung!



DBI

* Bind-Variablen verwenden und SQL-Injection vermeiden!

```
my ($user_id) = $db_h->fetchrow_array(
    "SELECT uid FROM user WHERE NAME = '$form{name}' AND passwd = '$passwd_md5'");
```

* Und wenn man im Formular als Name

```
admin' --
```

eingibt?

* Dann kommt als SQL raus:

```
SELECT uid FROM user WHERE NAME = 'admin' -- ' AND passwd = 'egal'
```

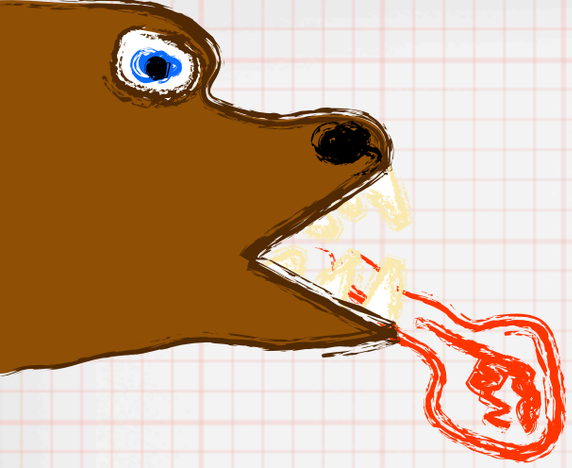
Die Lösung ist einfach

```
my ($user_id) = $db_h->fetchrow_array
    (
    'SELECT uid FROM user WHERE NAME = ? AND passwd = ?',
    undef,
    $form{name},
    $form{passwd},
    );
```

* Sonderzeichen werden automatisch maskiert
* SQL-Injection nicht mehr möglich
* Im schlimmsten Falle wird eben kein User gefunden

Obwohl einfach, auch in „professioneller“ Software ignoriert

* Beispiel: LX-Office ERP
  * kein use strict;
    kein use warnings;
  * Direkte Verwurstung von Formularparametern in SQL
  * => SQL-Injection ist trivial möglich!
  * Auch andere Parameter werden ungeprüft an *system* und *open* weitergereicht



Danke für guten Code!

(und natürlich fürs Zuhören)

* Langfassung des Vortrages ist im Tagungsband

Alvar Freude
http://alvar.a-blast.org/
alvar@a-blast.org