Gute Kritiken - ein Traum für Code

In vielen Unternehmen gibt es Programmierrichtlinien, viele Programmierer halten sich an die Vorschläge aus Damian Conway's Buch "Perl Best Practices". Doch wie kann sichergestellt werden, dass die einzelnen Richtlinien eingehalten werden?
Das Modul `Perl::Critic` ist genau dafür gedacht!

Jeffrey Ryan Thalhammer - der Autor von `Perl::Critic` - bezeichnet sein Modul als ein erweiterbares Framework um Programmierstandards auf Perl-Code anzuwenden.
Es analysiert den Quellcode mit Hilfe des Moduls `PPI` von Adam Kennedy.

Die Programmierstandards, die standardmäßig von `Perl::Critic` angewendet werden, sind in "Perl Best Practices" von Damian Conway beschrieben, allerdings lassen sich die Regeln auch ein- und ausschalten. Wer nicht immer mit den Regeln von Conway einverstanden ist, kann sich `Perl::Critic` so konfigurieren, dass es auf eigene Regeln passt.

Auch komplett eigene Standards lassen sich implementieren, wofür aber Kenntnisse von `PPI` notwendig sind. Wie eigene Standards implementiert werden, wird in der nächsten Ausgabe von `$foo` gezeigt.

In diesem Artikel geht es darum, wie `Perl::Critic` verwendet werden kann und wie man die Bordmittel weitgehend für eigene Bedürfnisse anpassen kann. Das ganze passiert an einem Beispielskript (Listing 1).

Zur Überprüfung wird das Kommandozeilentool `perlcritic`, das mit dem Modul mitgeliefert wird, verwendet. Es wird einfach mit `perlcritic skript.pl` gestartet.

```
 1 #!/usr/bin/perl
 2
 3 print "Dies ist ein tolles Programm\n";
 4
 5 use strict;
 6 use warnings;
 7
 8 my $number = calc();
 9 unless($number == 42){
10     print "falsche Antwort!\n";
11 }
12 else{
13     print "42 ist die Antwort auf alle Fragen\n";
14 }
15
16 sub calc{
17     print "Term eingeben (Form: 3 + 4 * 12): ";
18     my $expression = <STDIN>;
19     chomp $expression;
20
21     my $res;
22     my $string = '$res = $expression';
23     eval $string;
24
25     return undef unless $string > 0;
26     return $res;
27 }
```

Listing 1

In der Ausgabe (Listing 2) ist ersichtlich, dass nicht alles nach den "Best Practices" programmiert wurde.
Hier wird angemerkt, dass es Code vor der Einbindung von `strict` gibt. Das sollte relativ klar sein, warum das unerwünscht ist. Der nächste Kritikpunkt des Tools ist der String-`eval`, denn es wird immer wieder kompiliert.
Aber warum soll kein `undef` explizit zurückgegeben werden?

Wer keine Ausgabe von "Perl Best Practices" hat, braucht nicht lange zu grübeln.
Mit dem Parameter `--verbose` kann `perlcritic` dazu gebracht werden, etwas geschwätziger zu werden (siehe Listing 3).

Mit diesem `--verbose`-Wert von 10 wird genau erklärt, warum eine Regel existiert und wie man das verbessern kann.

Policies

Die Regeln sind in den sogenannten **Policies** festgehalten. `Perl::Critic` hat die Regeln nach dem Inhaltsverzeichnis von "Perl Best Practices" organisiert. So finden sich die

```
C:\>perlcritic step0.pl
Code before strictures are enabled at line 3, column 1.
  See page 429 of PBP.  (Severity: 5)
Expression form of "eval" at line 23, column 5.  See page 161 of PBP.
  (Severity: 5)
"return" statement with explicit "undef" at line 25, column 5.  See
  page 199 of PBP.  (Severity: 5)
```

Listing 2

Regeln zur Dokumentation im Namensraum Perl::Critic::Policy::Documentation

Die Module selbst sind sprechend benannt. Das Modul, dass leere Strings (durch Anführungszeichen begrenzt) anmerkt, heißt dementsprechend ProhibitEmptyQuotes und gehört zum Namensraum P::C::Policy::ValuesAndExpressions Wie man schon am Namen der Regel erkennen kann, sind leere Strings unerwünscht.

Einteilung der Fehler

Die Abweichungen vom Programmierstandard werden in 5 verschiedene Grade eingeteilt. Die schwerwiegenden Abweichungen sind dabei vom Level "5", während kleine optische Abweich-ungen vom Level "1" sind. Standardmäßig werden nur die Meldungen vom Level "5" angezeigt.
Auch die Ausgabe aus dem Testlauf zeigen nur die "Fehler", die sehr schwerwiegend sind.

Die Einteilung in 5 Level ist sehr sinnvoll, da man sich nach und nach an die kleinen Abweichungen vom Programmierstandard annähern kann. Wenn standardmäßig **alle** Meldungen angezeigt bekommen würde, wäre man von der Masse an Meldungen regelrecht erschlagen. Man sollte sich also von den oberen Level nach unten durcharbeiten.
In der folgenden Tabelle kann man gut sehen, dass auch bei einem relativ kurzen Programm die Meldungen der unteren Level in der Überzahl sind. Welche Meldungen angezeigt werden sollen, kann man über den `--severity`-Parameter des Konsolenprogramms einstellen.
Wenn man sich die Meldungen der unteren Level anschaut, dann

Level	Anzahl Meldungen
5	3
4	1
3	0
2	6
1	2

merkt man schnell, dass hier wirklich nur optische Dinge angemeckert werden.

Regeln anpassen

Die Regeln, die Damian Conway in seinem Buch aufstellt sind seine persönliche Meinung. Bei den meisten Regeln muss man ihm Recht geben -wie zum Beispiel die Verwendung von `strict`, andere Regeln sind diskussionswürdig. Bevor man seine eigenen Standards ändert und man sich nicht wohl fühlt, sollte man lieber Conways Regeln anpassen.

critic-Pseudopragma

Im Prinzip hat Conway Recht, wenn er sagt, dass man keine Namen von built-in-Funktionen für eigene Subroutinen verwenden sollte. Wenn aber eine built-in-Funktion tatsächlich überschrieben werden soll - was aus den unterschiedlichsten Gründen

```
C:\>perlcritic --verbose=10 step0.pl
Code before strictures are enabled at line 3, column 1.
  TestingAndDebugging::RequireUseStrict (Severity: 5)
      Using strictures is probably the single most effective way to improve
      the quality of your code. This policy requires that the `'use strict''
      statement must come before any other statements except `package',
      `require', and other `use' statements. Thus, all the code in the entire
      package will be affected.
Expression form of "eval" at line 23, column 5.
  BuiltinFunctions::ProhibitStringyEval (Severity: 5)
      The string form of `eval' is recompiled every time it is executed,
# und so weiter
```

Listing 3

vorkommen kann, kann man diese Methode vor Perl::Critic schützen.

Das geht mit einem "Pseudopragma". Das Pseudopragma ist kein richtiges Pragma, da es nur ein erweiterter Kommentar ist. Mit ##no critic kann Perl::Critic temporär ausgeschalten werden. Code innerhalb der Subroutine nicht.

Der "no critic"-Anweisung kann man auch Argumente übergeben. So kann man auch einzelne Regeln ausschließen. Die Argumente werden als Reguläre Ausdrücke betrachtet, so dass man auch sehr schnell eine ganze Gruppe von Perl::Critic nach einer Datei mit dem Namen .perlcriticrc.

Die Konfigurationsdatei ist wie eine .ini-Datei formatiert, wobei der Sektionsname dem Namen des Moduls für die Regel entspricht (siehe Listing 5).

```
##no critic (EmptyQuotes)
#
# Das Argument von no critict matcht auf die Regel
# Perl::Critic::Policy::ValuesAndExpressions::ProhibitEmptyQuotes
my $foo = "";
```

Listing 4

Steht das ##no critic in der gleichen Zeile wie ein Statement, ignoriert Perl::Critic nur diese Zeile. Mit

```
sub format{ ##no critic
    # Code der Subroutine
  }
```

wird nur der Name der Subroutine ignoriert, der Code danach aber nicht.

Wenn ein größerer Bereich nicht analysiert werden soll, dann kann man diesen Bereich in einen ##no critic / ##use critic-Block stellen.

```
##no critic
sub format{
  # Code der Subroutine
}
##use critic
```

Jetzt wird die komplette Subroutine von Perl::Critic ignoriert.

Wenn die Analyse für einen Block ausgesetzt werden soll, kann man auch folgendes machen:

```
sub format{
  ##no critic
  # Code der Subroutine
}
```

Hier wird der Name der Subroutine noch analysiert, aber der Regeln ausschalten kann.

Listing 4 zeigt, wie Meldungen über "EmptyQuotes" verhindert werden.

Konfigurationsdatei

Um nicht in jedem Programm die gleichen Code-Fragmente mit den critic-Anweisungen von der Analyse ausnehmen zu müssen, kann man eine Konfigurationsdatei erstellen.
Ich werde - auch wenn Conway es anders vorschlägt - nicht auf Postfix-Bedingungen verzichten und unless-Blöcke werden auch in Zukunft im Code zu finden sein.

Da diese Meldungen global ausgeschaltet werden sollen, ist die Konfigurationsdatei die ideale Lösung. Die Datei muss im aktuellen Verzeichnis oder im HOME-Verzeichnis liegen - genau in der Reihenfolge sucht Perl::Critic nach einer Datei mit dem Namen .perlcriticrc.

Die Konfigurationsdatei ist wie eine .ini-Datei formatiert, wobei der Sektionsname dem Namen des Moduls für die Regel entspricht (siehe Listing 5).

Der Anfang des Modulnamens ("Perl::Critic::Policy") kann auch weggelassen werden. Perl::Critic geht davon aus, dass alle Policy-Module in diesem Namensraum liegen.

Um diese Regel komplett auszuschalten, reicht es ein "-" vor den Modulnamen zu schreiben, also

```
[-ValuesAndExpression
 ::ProhibitEmptyQuotes]
```

Nachdem alle Fehler beseitigt wurden und manche Regeln ausgeschaltet wurden, läuft das Programm durch und bestätigt, dass der Code den "Best Practices" der Perl-Programmierung entspricht.

Test::Perl::Critic

Bei der Entwicklung von Modulen ist es aber umständlich, immer wieder das Programm aufzurufen.

```
  [Perl::Critic::Policy::ValuesAndExpressions::Prohib
itEmptyQuotes]
  severity = 1
  arg1 = value1
  arg2 = value2
```

Listing 5

Da hilft es doch, wenn es ein Testskript dafür gibt. Das ganze kann auch sehr einfach über `Test::Perl::Critic` eingebunden werden.

Dieses einfache Skript in den t-Ordner zu den anderen Tests und danach wird bei jedem Testlauf auch auf die Einhaltung der Programmierstandards geprüft.

```
use Test::More;
use Test::Perl::Critic
    (-severity => 1);
all_critic_ok();
```

Natürlich kann der `severity`-Level höher gesetzt werden, je nach Wunsch, wie stark die Standards verwendet werden sollen.

Wenn die Standards über eine Konfigurationsdatei angepasst wurden, sollte diese mit in die Distribution gepackt werden und im Testskript benannt werden, damit die Installation auf einem anderen Rechner nicht fehlschlägt.

Das Pragma "criticism"

Die strengste Form der Überprüfung ist die Verwendung des `criticism`-Pragmas im Programmcode. Allerdings hat das auch einen erheblichen Geschwindigkeitsnachteil, weil bei jedem Compile-Vorgang der Code überprüft wird. Das ist eigentlich unnötig. Wenn der Testlauf mit dem Kommandozeilentool oder mit `Test::Perl::Critic` erfolgreich war und der Code nicht verändert wurde, dann entspricht er immer noch den "Best Practices".

Aber der Vollständigkeit halber wird hier gezeigt, wie die Verwendung des Pragmas aussieht.

```
1 use criticism;
2
3 $_ = "test";
4 print;
```

Wenn dieses Programm ausgeführt wird, wird gleich das fehlende strict angemeckert:

```
C:\>perl critic.pl
critic.pl:
 Code before strictures
    are enabled at line
    3, column 1. See
    page 429 of PBP.
test
```

Dieses Pragma kann man nicht deaktivieren wie zum Beispiel `strict` - es gibt also kein `no criticism`. Man kann nur mit dem oben genannten Pseudo-Pragma arbeiten. `criticism` ignoriert auch die Konfigurationsdateien.

Ausschau

In der nächsten Ausgabe wird gezeigt, wie man eigene Policies entwickeln kann.

Renée Bäcker

Werden Sie selbst zum Autor...

... wir freuen uns über Ihren Beitrag!

Info@foo-magazin.de