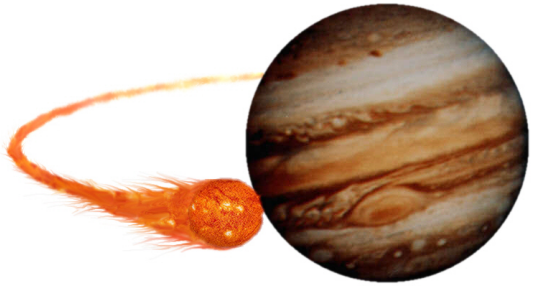
Safer Perl

Datenwäsche mit Perl

Ralf Peine www.jupiter-programs.de



1. Motivation

Wozu validieren ?

```
sub add {
   my $sum = 0;
   for (@_) { $sum += $_; }
   return $sum;
}
```

Wozu validieren ? (2)

```
$values = [];
$sum    = add($values);
```

Was passiert ?

- **Kein "die"**
- **Kein "warning"**
- **Ein Wert als Ergebnis**

Und dann …

```
$sum = 3452678;
```

!

Wollt ihr das wirklich ?

Wer bezahlt

3,452,678.00 Euro

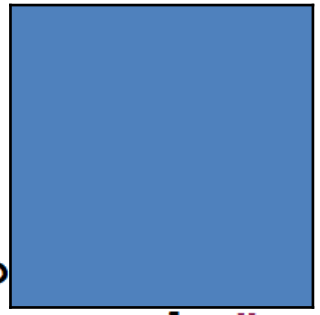
?!?!?

Für nichts ?

Lösung 1

if / unless Anweisungen

Lösung 1: if/unless Anweisungen verwenden



```
sub add
    my
    map
        croak "value is empty"
            unless defined $_ and $_ ne '';
        croak "value is not a Scalar: '$_'"
            unless ref($_) eq '';
        croak "value '$_' is not a Float"
            unless /^[\+\-]?\d+(\.\d+)?([Ee][\+-]?\d+)?$/;
        $sum += $_;
    } @_;
    return $sum;
}
```

Lösung 1: if/unless Anweisungen verwenden

- **Viel Arbeit**
- **Fehleranfällig**
- **Nichts vergessen ?**

Lösung 2

CPAN-Module zur Validierung nutzen

Lösung 2: CPAN Validerungsmodul nutzen

- **keine Arbeit**
- **einfach anwenden**
- **sichere Validierung**
- **keine Validierungs-**
  -Funktionalität mehr im Code

```
use Scalar::Validation  qw(par);

sub add {
   my $sum = 0;
   for (@_) {
      $sum += par(add => Float => $_);
   }
   return $sum;
}
```

Code ist jetzt sicher!

Perl-Script "dies" mit

```
… value 'ARRAY(0xd62a68)`
    is not a float at …
```

2. Anwendung

Sub mit Positions-Parametern

```
# create person by positional parameters
sub create_person_pos {
    my $trouble_level = p_start;

    my $prename = par prename => Filled    => shift;
    my $surname = par surname => Filled    => shift;
    my $birth   = par birth   => BirthDate => shift;

    p_end \@_;

    # fire exit, if validation does not die
    return undef if validation_trouble($trouble_level);

    # --- run sub ------------------------------------

    return {
        prename => $prename,
        surname => $surname,
        birth   => $birth
    };
}
```

Sub mit benannten Parametern

```
sub create_person_named {
    local ($Scalar::Validation::trouble_level) = 0;

    my %pars = convert_to_named_params \@_;

    my $person = {};

    $person->{prename} = npar -prename => Filled => \%pars;
    $person->{surname} = npar -surname => Filled => \%pars;
    $person->{birth}   = npar -birth   => BirthDate   => \%pars;

    p_end \%pars;

    return undef if validation_trouble();

    # --- run sub ----------------------------------------

    return $person;
}
```

Methode mit Positions-Parametern

```
sub add_date_positional {
    my $trouble_level = p_start;

    my $self     = par self => $is_self  => shift;
    my $iso_date = par date => IsoDate   => shift;

    p_end (\@_);

    return undef if validation_trouble $trouble_level;

    # --- run sub ------------------------------------

    $self->{date} = $iso_date;
}
```

Methode mit Positions-Parametern
Regel zum Testen von `$self`

```
use Scalar::Validation qw(:all);
use MyValidation;

my ($is_self) = is_a __PACKAGE__; # parentesis are needed!!
```

Methode mit Positions-Parametern
Regel zum Testen von `ArticleId`

```
declare_rule (
    ArticleId =>
    -as          => Scalar =>
    -where       => sub { /^ID\d+$/ },
    -message     => sub { "$_ is not an article ID like ID1234" },
    -description => '$_ has to be a scalar like ID1234'
    );
```

3. Test

Regeln können isoliert getestet werden

Isolierter Test der Regel ArticleId in MyValidation.t

```
diag ("--- ArticleId ----------------------------------------

lives_ok {validate article_id => ArticleId => 'ID0'},
    " ArticleId => ID0";
lives_ok {validate article_id => ArticleId => 'ID1'},
    " ArticleId => ID1";
lives_ok {validate article_id => ArticleId => 'ID1234567890'},
    " ArticleId => ID1234567890";

throws_ok {validate article_id => ArticleId => undef }
    qr/<undef> is not an article ID like/,
    "!ArticleId => undef";
throws_ok {validate article_id => ArticleId => bla; }
    qr/'bla' is not an article ID like/
```

Isolierter Test der Regel `ArticleId` Ergebnisse

```
# --- ArticleId ----------------------
ok 53 -  ArticleId => ID0
ok 54 -  ArticleId => ID1
ok 55 -  ArticleId => ID1234567890
ok 56 - !ArticleId => undef
ok 57 - !ArticleId => bla
ok 58 - !ArticleId => id123
ok 59 - !ArticleId => [ ID123 ID456 ]
```

Kompliziertere Regeln: IsoDate

```
{ # make month_days private
    my @month_days = (
        0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
        );

    declare_rule (
        IsoDate =>
        -as      => Filled =>    # a filled string
        -where   => sub {
            my ($year, $month, $day) = /^(\d{4})-([01]\d)-([0123]\d)$/;
            return 0 unless defined ($year);
            return 0 if $month == 0 || $month > 12;
            return 0 if $day   == 0 || $day   > $month_days[$month];
            return 1 if     $month !=  2;
            return 1 if     $day   <= 28;
            return 0 if     $year  %   4;
            return 1 unless $year  % 400;
            return 0 unless $year  % 100;
            return 1;
        },
        -owner       => MyValidation =>
        -message     => sub { "value $_ is not an ISO 8601 date like 2014-08-30" },
        -description => 'This rule checks if $_ is an ISO 8601 date like 2014-08-30'
        );
}
```

47 Tests minimal benötigt für Regel

IsoDate

```
ok 1 -  IsoDate 0000-01-01
ok 2 -  IsoDate 1000-01-31
ok 3 -  IsoDate 1000-02-28
ok 4 -  IsoDate 1000-03-31
ok 5 -  IsoDate 1000-04-30
ok 6 -  IsoDate 1000-05-31
ok 7 -  IsoDate 1000-06-30
ok 8 -  IsoDate 1000-07-31
ok 9 -  IsoDate 1000-08-31
ok 10 -  IsoDate 1000-09-30
ok 11 -  IsoDate 1000-10-31
ok 12 -  IsoDate 1000-11-30
ok 13 -  IsoDate 1000-12-31
ok 14 -  IsoDate 2017-01-17
ok 15 -  IsoDate 1965-12-29
```

```
ok 16 - !IsoDate undef
ok 17 - !IsoDate 0000-00-00
ok 18 - !IsoDate 0000-01-00
ok 19 - !IsoDate 0000-00-01
ok 20 - !IsoDate 2000-02-30
ok 21 - !IsoDate 2000-02-31
ok 22 - !IsoDate 2000-04-31
ok 23 - !IsoDate 2000-06-31
ok 24 - !IsoDate 2000-09-31
ok 25 - !IsoDate 2000-11-31
ok 26 - !IsoDate 2000-01-32
ok 27 - !IsoDate 2000-02-32
ok 28 - !IsoDate 2000-03-32
ok 29 - !IsoDate 2000-04-32
ok 30 - !IsoDate 2000-05-32
ok 31 - !IsoDate 2000-06-32
ok 32 - !IsoDate 2000-07-32
ok 33 - !IsoDate 2000-08-32
ok 34 - !IsoDate 2000-09-32
ok 35 - !IsoDate 2000-10-32
ok 36 - !IsoDate 2000-11-32
ok 37 - !IsoDate 2000-12-32
ok 38 - !IsoDate 2000-13-01
```

```
# --- Schaltjahre ------------
ok 39 -  IsoDate 1004-02-29
ok 40 -  IsoDate 1200-02-29
ok 41 -  IsoDate 1204-02-29
ok 42 -  IsoDate 2000-02-29
ok 43 -  IsoDate 9200-02-29
ok 44 - !IsoDate 1001-02-29
ok 45 - !IsoDate 1002-02-29
ok 46 - !IsoDate 1003-02-29
ok 47 - !IsoDate 1900-02-29
```

4. Doc

Dokumentation
Eigene Regeln

```
my $rules_ref = get_rules();
```

↓ + Report::Porf

Rule Name	Parent Rule	Quelle	Beschreibung	Enum
ArticleId	Scalar	MyValidation	$_ has to be a scalar like ID1234	
BirthDate	IsoDate	MyValidation	This rule checks if $_ is an ISO birth date and in the past	
IsoDate	Filled	MyValidation	This rule checks if $_ is an ISO 8601 date like 2014-08-30	
LivingBirthDate	BirthDate	MyValidation	This rule checks if $_ is an ISO birth date of a living person	

Dokumentation
Alle Regeln

Rule Name	Parent Rule	Quelle	Beschreibung	Enum
ArrayRef		CPAN	Value is an Array reference.	
ArticleId	Scalar	MyValidation	$_ has to be a scalar like ID1234	
BirthDate	IsoDate	MyValidation	This rule checks if $_ is an ISO birth date and in the past	
Bool		CPAN	Value is a Scalar, all values including undef allowed	
Class		CPAN	Value is a class reference and not a scalar.	
CodeRef		CPAN	Value is a Code reference.	
Defined		CPAN	Value is defined	
Empty		CPAN	Value is not defined or ''	
Even	Int	CPAN	Value is an even integer ($_ % 2 == 0)	
ExistingDir	Filled	CPAN	Directory with given name has to exist	
ExistingFile	Filled	CPAN	File with given name has to exist	
FileHandle		CPAN	Value is a file handle and not a scalar.	
Filled		CPAN	Value is Scalar and defined and not empty ('')	
Float	Filled	CPAN	Value is a floating number with optional exponent	
HashRef		CPAN	Value is a Hash reference.	
Int	Filled	CPAN	Value is an integer	
IsoDate	Filled	MyValidation	This rule checks if $_ is an ISO 8601 date like 2014-08-30	
LivingBirthDate	BirthDate	MyValidation	This rule checks if $_ is an ISO birth date of a living person	
ModuleName	Filled	CPAN	Value is a module name like A::Ba::Cba.	
NegativeFloat	Float	CPAN	Value is a negative floating number with optional exponent	

Dokumentation
API - Meta-Modell

- Meta-Modell kann extrahiert werden
- Geringer Aufwand
- Details siehe Beispiele und Man Page

Dokumentation
API - Beispiel

Module Name	Sub Name	Pos./Named	Parameter Name	Rule(s)
MyClass	my_next_sub			
MyClass	my_sub	Positional	first_par	Int
MyClass	my_sub	Named	-first_named	PositiveFloat
ShoppingCart	_init	Positional	self	IsClass 'ShoppingCart'
ShoppingCart	_init	Named	-ID	-Optional [PositiveInt]
ShoppingCart	put_in	Positional	self	IsClass 'ShoppingCart'
ShoppingCart	put_in	Positional	article	ArticleId

5. Vergleich

Scalar::Validation
Eigenschaften

- Gut lesbar
- Keine Verwechslung von pos. Parametern möglich
- Strict Spartanic: Kein Compiler/Make notwendig
- Regeln
  - dynamisch, z.B. aus XSD generierbar
  - private Regeln (z.B. pro login/user)
- mittlere Geschwindigkeit
- API-Dokumenation generierbar
- Prüfmodi: error / warning / silent / off

Vergleich mit

- `Params::Validate (Moose)`
- `Type::Params`
- `Kavorka`
- `Perl 5.20`

Moose

MooseX::Params::Validate
Benannte Parameter

```
package Foo;
use Moose;
use MooseX::Params::Validate;

sub foo {
    my ( $self, %params ) = validated_hash(
        \@_,
        bar => { isa => 'Str', default => 'Moose' },
    );
    return "Hooray for $params{bar}!";
}
```

MooseX::Params::Validate Positions Parameter

```
sub bar {
    my $self = shift;
    my ( $foo, $baz, $gorch ) = validated_list(
        \@_,
        foo   => { isa => 'Foo' },
        baz   => { isa => 'ArrayRef | HashRef', optional => 1 },
        gorch => { isa => 'ArrayRef[Int]', optional => 1 }
    );
    [ $foo, $baz, $gorch ];
}
```

MooseX::Params::Validate Eigenschaften / Offene Punkte

- Verwechslung von pos. Parametern möglich
- Moose benötigt Compiler
- Regeln sind fix

- Definition neuer Regeln?
- Verhältnis von Typen und Regeln?
- Geschwindigkeit?

Type::Params

Type::Params
Positions Parameter mit state Variable

```perl
use Type::Params qw( compile );
use Types::Standard qw( slurpy Str ArrayRef Num );

sub deposit_monies
{
   state $check = compile( Str, Str, slurpy ArrayRef[Num] );
   my ($sort_code, $account_number, $monies) = $check->(@_);

   my $account = Local::BankAccount->new($sort_code, $account_number);
   $account->deposit($_) for @$monies;
}
```

Type::Params
Benannte Parameter mit state Variable

```
sub dump
{
   state $check = compile(
      slurpy Dict[
         var    => Ref,
         limit  => Optional[Int],
      ],
   );
   my ($arg) = $check->(@_);

   local $Data::Dumper::Maxdepth = $arg->{limit};
   print Data::Dumper::Dumper($arg->{var});
}
```

Type::Params
Für ältere Perl-Versionen ohne "state"

```perl
my $deposit_monies_check;
sub deposit_monies
{
   $deposit_monies_check ||= compile( Str, Str, slurpy ArrayRef[Num] );
   my ($sort_code, $account_number, $monies) = $check->(@_);

   ...;
}
```

Type::Params
Eigenschaften

- Sehr schnell
- Kein Compiler
- CPAN shell empfohlen
- Verwechslung von pos. Parametern möglich
- Regeln sind fix
- Verwendet Type::Tiny
- Erstellung neuer Regeln:
  - Aufwendig
  - nicht in Manpage beschrieben

Kavorka

Kavorka
Mit eigener Syntax: fun ...

```
use Kavorka;

fun maxnum (Num @numbers) {
    my $max = shift @numbers;
    for (@numbers) {
        $max = $_ if $max < $_;
    }
    return $max;
}

my $biggest = maxnum(42, 3.14159, 666);
```

Kavorka
Überlangerung von Funktionen/Methoden

```
1  use v5.14;
2  use Kavorka 0.004 qw( multi fun );
3
4  multi fun fib ( Int $i where { $_ <= 1 } ) {
5          return $i;
6  }
7
8  multi fun fib ( Int $i ) {
9          return fib($i-1) + fib($i-2);
10 }
11
12 say fib($_) for 0..9;
13
```

Kavorka
Eigenschaften / Offene Punkte

- Early Stage: Inkompatible Änderungen möglich
- Eigene Syntax
- Benötigt Compiler
- Typen sind fix

- Definition neuer Regeln?
- Verhältnis von Typen und Regeln?
- Geschwindigkeit?

Perl 5.20
Signatures

Perl 5.20 Signatures
Nur Bennenung von Parametern!

```
sub foo ($left, $right) {
    return $left + $right;
}
```

Perl 5.20 Signatures
Eigenschaften / Offene Punkte

- Experimental: Keine Garantie für nichts
- Eigene Syntax (Kernel)
- Mitgeliefert
- Nur Namen
- Typen durch Prototypes

- Keine eigenen Regeln

Fazit

Vergleich
Fazit (1)

Für Moose

- MooseX::Params::Validate
- Type::Params

Vergleich
Fazit (2)

Für

- Speed
- Reine Typprüfung
- Moo / OO
- Funktionale Progammierung

- Type::Params

Vergleich
Fazit (3)

Für

- Sicherheit
- Variabilität/dynamische Regeln
- Schwierige Laufzeitumgebungen
- API-Docs
- Moo / OO
- Funktionale Progammierung

- Scalar::Validation

Vergleich
Fazit (4)

Für experimentierfreudige

- Kavorka

Nicht empfohlen

- Perl 5.20 Signatures

Daten-Validierung

Danke!