

Fehlerbehandlung mit Hilfe von Ausnahmen in Perl Skripten

Die Behandlung von Fehlern mit Hilfe von Ausnahmen ist ein sehr eleganter Lösungsansatz, um fehlertolerante Perl Skripte zu implementieren, der in ziemlich alle Anwendungen vorteilhaft anwendbar ist. Obwohl der nötige grundsätzliche Perl Konstrukt ziemlich einfach und allgemein bekannt ist, gibt es doch abgeleitete kompliziertere Konstruktionen, welche genau untersucht werden sollten, will man sich böse Überraschungen sparen. In diesem Artikel möchte ich verschiedene solche Konstruktionen beschreiben, diskutieren und mit Beispielen belegen.

von Marco Marazzi*

Unter dem Begriff 'fehlertolerantes Perl Skript' verstehen wir ein Skript, das in der Lage ist, nach einem Fehler so weiter zu arbeiten oder zu terminieren, dass

- sinnvolle Informationen aufgezeichnet und/oder angezeigt werden,
- keinen unkontrollierten Datenverlust entsteht,
- keine unbekannte Datenbeschädigung entsteht,
- einen korrekten Zustand seiner Variablen wiederhergestellt wird.

Ein Beispiel: wird mit einem Skript aus etlichen Dokumenten eine Übersicht für ein HTML-Dokument zusammengestellt, so soll auch dann eine brauchbare Übersicht erstellt werden, wenn noch nicht alle Dokumente verfügbar sind. In diesem Falle sollen Hinweise eingesetzt werden, welche auf die fehlenden Dokumenten hinweisen. Das bedeutet, die Datenfehler führen zu keiner fehlerhaften Ausgabe, in diesem Falle eine unvollständige Übersicht, die Fehler selbst werden ausgewiesen, und die Verarbeitung wird ordentlich abgeschlossen.

Zweifellos können wir das auch ohne den Rückgriff auf Ausnahmen implementieren, werden aber spätestens dann, wenn die Dokumenten in komplizierteren und womöglich rekursiven Strukturen gesucht werden müssen, feststellen, dass Ausnahmen doch verdammt nützlich sein können.

Sucht man im Web nach dem Begriff 'Perl error+handling', so findet man vor allem nur Angaben über anwendungsspezifischen Fehlerbehandlungen.

Das ist eigentlich verständlich, einerseits weil der wesentlichste Teil einer Fehlerbehandlung eines Prozesses deren anwendungsspezifischen Teil ist, und andererseits weil Perl selbst ziemlich fehlertolerant ist, so dass seine eigene Fehlerbehandlung in vielen Fällen weitgehend genügt.

Auch die vielen auf Perl spezialisierten Web Seiten beschränken sich im allgemeinen auf der Beschreibung der grundlegenden Strukturen. Selten wagen sich in die Tiefe, so wie zum Beispiel Stas Bekman und Matt Sergeant in [8] und [9].

Perl.Skripte mit Datenbankanbindung delegieren die Fehlerbehandlung dem entsprechenden Datenbanksystem, brauchen also praktisch nur zu entscheiden wann ein Rollback ausgeführt werden muss. Alles andere erledigt das Datenbanksystem selbst.

Somit bleibt der Kreis der Perl-Anwendungen, die tatsächlich eine eigene Fehlerbehandlung benötigen, auf wenigen Anwendungstypen beschränkt, nämlich solche welche grosse Mengen wichtiger Daten verwalten, die nur sehr schwer von Hand nach einem unsanften Abbruch repariert werden können.

Grundlagen

Zur Behandlung von Ausnahmen lässt Perl grundsätzlich folgende drei Möglichkeiten zu:

- Ausnahme auslösen,
- Ausnahme behandeln und
- den Wirkungsbereich der ausgelösten Ausnahme einschränken.

Ausnahmen auslösen

Das Auslösen von Ausnahmen wird in [2] Kapitel 13 'Error handling' ausführlich beschrieben. Hier folgen also nur zusätzliche Anmerkungen.

- Die Identifikation und Charakterisierung der Ausnahmen ist alleinige Sache des Programmierers. Perl selbst hat keine Systematik um Ausnahmen zu identifizieren und/oder klassifizieren.

- Ausnahmen werden durch ein einziges Argument, das ein Scalar sein muss, bezeichnet. Bei komplexeren Anwendungen kann dies eine Referenz zu einer zweckmässigen Datenstruktur sein, welche die Kontextinformation speichert. Mit dem CPAN Modul Fatal.pm kann man Perl Routinen dazu führen, dass sie Ausnahmen anstelle von Rückwerten des Typs undef verwenden, um missglückten Funktionen anzuzeigen. Mit dem CPAN Modul Carp kann man auch Ausnahmen auslösen, und zwar mit der exportierten Subroutine croak. Ob mit dem Perl-Befehl ‚die‘, indirekt mit Fatal oder mit Carp::croak alle Ausnahmen werden von Perl gleichermassen behandelt. (Siehe Scripts b0.pl, b0x.pl, b0xx.pl und b0xxx.pl)

Ausnahme behandeln

Werden Callbacks den Ausnahmen zugewiesen, muss Folgendes beachtet werden

- Callbacks können den Ablauf der Ausnahme nicht ändern.
- Bei deren Installation kann nur eine Referenz zu einem Code Block angegeben werden. Eine Argumentliste, wie im Perl/Tk üblich, ist nicht möglich, und führt sogar zur Auslösung der Ausnahme 'Not a subroutine reference' bei der Ausführung des Skriptes..

Solche Callbacks werden in der Fachsprache entweder ‚Error Handler‘ oder auch ‚Cleanup Routine‘ genannt. Wir werden in diesem Text die erste Bezeichnung verwenden, zuerst einmal um eine gemeinsame Sprachregelung mit [2] anzuwenden, und dann weil die zweite eher eine Spezialisierung bezeichnet, nämlich ein Stück Code welches sich ausdrücklich mit dem Aufräumen von Resources und dem Abschliessen von Prozessen beschäftigt. Siehe Script b6.pl.

Den Wirkungskreis der ausgelösten Ausnahmen einschränken

Zur Einschränkung des Wirkungskreises der ausgelösten Ausnahme gibt es grundsätzlich drei Methoden:

- Ein lokales Error-Handler deklarieren.
- Der Perl-Befehl eval anwenden, um damit das Code block, welches die Ausnahmen auslöst, isoliert auszuführen.
- Die Ausführung in Subprozessen auslagern.

Ein lokales Error-Handler deklarieren.

Die Deklaration des Error-Handlers einer Ausnahmeklasse kann lokal definiert werden:

```
sub mySub {
    my $cb = $SIG{__DIE__}; ## save ...
    local $SIG{__DIE__} = sub {
        ## for die exc. issued in mySub
        &$cb(@_); ## percolate!
    };
#
# do some stuff which may issue die
#
}
```

Das Restaurieren der Callback-Deklaration ist beim Verlassen des Code Blocks nicht notwendig, wird sie doch automatisch von Perl selbst eliminiert.

Der Perl Befehl eval anwenden.

Das Evaluieren eines Code Blocks bewirkt, dass die dorthin ausgelösten Ausnahmen nur den evaluierten Code Block terminieren. Der aufrufende Prozess geht weiter, und erfährt von der Ausnahme, indem er die Variable \$@ untersucht. Wird diese Inspektion unterlassen, so bleibt auch die allfällige Ausnahme unbemerkt.

```
sub foo {
    die 'missing mandatory arglist'
    unless @_;
    ##
    ## process arglist
    ##
}

eval {&foo()};
if ($@) {
    ## process exception
} else {}
## OK, go on
```

Alle publizierten Module, welche ‚recoverable exceptions‘ unterstützen, greifen in der einen oder anderen Weise auf diese Möglichkeit zurück. Der zu schützende Quellcode wird mehr oder weniger elegant evaluiert.

Die Ausführung in Subprozessen auslagern

Mit Hilfe von ‚fork‘ oder ‚system‘ kann man ganze Skripte in selbständigen Subprozessen ablaufen lassen, wobei die dort ausgelösten Ausnahmen keinen Bezug auf dem auslagernden Skript haben. Dieser kann lediglich herausfinden,

ob der Subprozess durch eine Ausnahme terminiert wurde. Dazu gibt es allgemein bekannten Methoden, die wir aber aus Platzgründen nicht weiter besprechen wollen. Wir weisen hier auf die Literatur hin.

Beschreibung einiger Perl-Konstrukte

Mit diesen wenigen aber ausreichenden grundlegenden Kenntnissen können wir nun versuchen, Konstrukte für die spezifische Behandlung von Ausnahmen in komplexeren Skripte zu beschreiben.

Im Folgendem wollen wir also einige besondere Perl-Konstrukte, welche eine eingehendere Betrachtung verdienen, und sozusagen Modell-Charakter aufweisen, untersuchen. Das sind

- BEGIN,CHECK, INIT und END Blocks,
- Package code,
- Callbacks,
- verschachtelte Ausnahmen,
- Rekursionen,
- Iteratoren,
- endlose Schleifen,
- mit memoize gepufferte Subroutinen und
- mit Tie verbundene Strukturen.

BEGIN, CHECK, INIT und END Blocks

BEGIN Blocks sind Klassenkonstruktoren, welche aber bereits während der Kompilierungsphase durchlaufen werden.

CHECK Blocks werden am Ende der Kompilierungsphase (nach dem BEGIN Block des main package) ausgeführt. Da sind aber lustige Ausnahmen:

- Wird ein Modul in einem evaluierten Code Block mit `use` geladen, so reagiert Perl mit der Warnung *'Too late to run CHECK block at module line nn'* und der CHECK Block wird nicht ausgeführt.
- Die gleiche Warnung erscheint wenn ein Modul mit `,require`, geladen wird.
- CHECK Blocks werden während der Debugging Sessions ignoriert. Das gilt sowohl für `ptkdb` (Version 1.1091) als auch für den debugger von `ActiveState` (Version 6.0).

INIT Blocks werden gerade vor dem Zeitpunkt, wenn der Perl-Interpreter seinen Lauf nimmt, ausgeführt. Sie werden also bei der Kompilierung nicht ausgeführt.

END Blocks sind Klassendestruktoren, und werden nur bei der Terminierung des Perl-Interpreters durchlaufen.

Wird aber in einem BEGIN oder CHECK Block während der Kompilierung eine Ausnahme ausgelöst, so werden die entsprechenden END Blocks nicht ausgeführt. Der Grund liegt wohl darin, dass im Falle einer Ausnahme in den obengenannten Blocks der Perl-Interpreter gar nicht gestartet wird.

Siehe Script `b2x.pl`.

Wird hingegen ein Modul mit dem Perl-Befehl `,require'` geladen, und wird eine Ausnahme in seinem BEGIN oder CHECK Block ausgelöst, dann werden alle END Blocks ausser sein Eigenes ausgeführt. In diesem Falle ist der Perl-Interpreter sehr wohl aktiv, kann und soll bei der Terminierung alle bekannten END Blocks ausführen.

Siehe Script `b2ty.pl`

Beispiel: Das Package `inspectRegistry` soll nur in einer Windows-Umgebung zugelassen werden. Daher wird bereits im CHECK Block die Variable `$^O` geprüft. Wenn die Identifikation der Plattform nicht zutrifft, dann wird eine Ausnahme ausgelöst.

```
package inspectRegistry;
CHECK {
    die "Unsupported platform $^O ."
        unless ($^O =~ /win32/i);
    ## OK, go on
}
END {}
```

Siehe Scripts `b2*.pl`

Package Code

Wir bezeichnen die Folge der Perl-Befehle eines Packages, die in keinem Block (Subroutine) enthalten ist, als Package Code.

Wird ein Module mit `perl -c <modulname>` kompiliert, so wird das Package Code nicht ausgeführt. Wird hingegen ein Modul mit dem Perl-Befehl `use` in einem Skript eingebunden, so wird der Package Code gerade nach seinem BEGIN Block, und vor seinem CHECK Block ausgeführt.

Das bedeutet,

- CHECK wird nur dann ausgeführt, wenn keine Ausnahmen im BEGIN Block und in dem Package Code auftreten.
- Die entsprechenden END Blocks werden nicht durch Ausnahmen unterdrückt, die im Package Code ausgelöst werden.

Ausnahmen in Callbacks

Callbacks werden, wie der Name selbst suggeriert, an Prozessen weitergegeben mit dem Ziel später dann unter ganz anderen aber im Voraus gekannten Umständen aufgerufen zu werden. Aber anders als Subroutinen, werden sie in der Umgebung ausgeführt, wo sie einmal definiert wurden. Man kann sich also fragen, was passiert, wenn dieser Callback in einem anderen Kontext eine Ausnahme auslöst.

```
my $dieCB = sub {
    my $e = shift;
    print "\n exception caught :$e"
};
my $cb = sub {
    local $SIG{__DIE__} = $dieCB;
    die "Cannot divide by 0"
    unless (@_ == 2 && $_[1] != 0);
    return $_[0] / $_[1];
}

$SIG{__DIE__} = sub {
    ## not used by &$cb
};
my $x = &$cb(2,4);

## or even

eval {$x = &$cb(2,0)}
die "Could not compute \$x because of '$@'" if ($@);
```

Siehe Scripts b3.pl, b3ee.pl und b3eee.pl

Die Antwort ist sehr erfreulich und der Sache sehr zuträglich. Die Gültigkeitsregeln der Variablen werden angewandt. Das bedeutet, Error-Handler werden behandelt wie übliche Callbacks.

Ein wichtige Anwendung von Callbacks sind Iteratoren. Diese Schnittstellen zu Listen aller Art sind sehr ausführlich in [1] Kapitel 4 ‚Iterators‘ beschrieben.

Ausnahmen können in Iteratoren ausgelöst werden, nicht nur um Fehler auszuweisen, sondern auch um echte Navigations- und Steuerungsfunktionen zu implementieren. In diesem Fall muss der Iterator diese Funktionen evaluieren, sodass die Ausnahme auf keinem Fall propagiert wird. Der mit ‚eval‘ verbundenen Overhead kann für einen Iterator einen gewichtigen Nachteil sein, der die Eleganz der Implementation zu Nichte macht.

Siehe Script bb.pl

Verschachtelte Ausnahmen

Zuerst einmal was ist eine solche Ausnahme? Besteht eine Anwendung aus mehreren Schichten, welche gegebenenfalls in mehreren Skripten realisiert werden, die zu gegebener Zeit mit require geladen werden, ist es sinnvoll für jede Schicht, die ein eigenes Error-Handler benötigt, Folgendes vorzusehen:

- den aktuellen Error-Handler in einer lokalen Variable retten,
 - den eigenen lokalen Error-Handler setzen,
 - den eigenen Code ausführen.
- Wird eine Ausnahme ausgelöst, so führt der lokale Error-Handler folgende Schritte aus:
- zuerst die Fehlerbehandlung seiner Schicht ausführen, dann
 - das gerettete Error-Handler aufrufen.

Eine verschachtelte Ausnahme wird also jedem Error-Handler der aktiven Schichten zur Verarbeitung weitergegeben. Die Ausnahme wird also propagiert gemäss der momentanen Verschachtelung der Schichten. Das erlaubt, den Error-Handling genau so wie die übrige anwendungsspezifische Verarbeitung zu kapseln.

Beispiel : Folgendes Skript enthält drei Schichten: die Erste ist das package code des package main, die Zweite ist die subroutine doThis und die Dritte ist die subroutine doThat. Jede Schicht hat seinen eigenen Error-Handler. Erfolgt eine Ausnahme in einer Schicht, so werden nur die Error-Handler der aktiven Schichten ausgeführt. In dem Beispiel werden also die Error-Handler aller Schichten nacheinander ausgeführt.

```
sub doThat {
    my $arg = shift;
    my $dieCallback = $SIG{__DIE__};
    local $SIG{__DIE__} = sub {
        my $e = shift;
        print "\n$e";
        &$dieCallback($e)
        if defined $dieCallback;
    };
    die 'doThis, missing mandatory arg'
    unless defined $arg;
    return uc $arg
}
sub doThis {
    my $rv = '';
    my $dieCallback = $SIG{__DIE__};
    local $SIG{__DIE__} = sub {
        my $e = shift;
        print "\n$e";
        &$dieCallback($e)
        if defined $dieCallback;
    };
```

```
$rv = main->doThat();
}

my $dieCallback = $SIG{__DIE__};
local $SIG{__DIE__} = sub {
    my $e = shift;
    print "\n$e";
    &$dieCallback($e)
    if defined $dieCallback;};
}

main::doThis();
```

Siehe Script b4.pl

Note: Wird die Verarbeitung von doThis evaluiert, so ändert sich die Behandlung der Ausnahme erwartungsgemäss nicht. Der Skriptablauf selbst aber wird anders, weil die Ausnahme nicht zum Abbruch des main packages führt. Siehe Script b4x.pl

Rekursionen

Die Ausnahmebehandlung in Rekursionen ist eigentlich ein Sonderfall von verschachtelten Ausnahmen. Alle involvierten Stufen verwenden, auch wenn selbstverständlich mit verschiedenen Daten, den gleichen Error-Handler. Siehe Script b5.pl

Endlose Schleifen

Erstaunlicherweise enthält Perl keine explizite Einrichtung, um endlose Schleifen zu bearbeiten. Man muss also selber Hand anlegen und die suspekten Schleifen in einem Code Block einpacken und evaluieren. Die Ausführung wird durch einen ‚alarm‘ mit seinem lokalen Callback überwacht. Das könnte in etwa so aussehen:

```
eval {
    my $loop;
    local $SIG{ALRM} = sub { die"loop" };
    alarm 2;
    eval {
        my $j = 0;
        for (my $i = 1;
            $i < 10000 ;
            $i++) {
            eval "print ++$j unless(\$i%100";
        }## long job
        kill ALRM,$$;
    };
    $loop = $@;
    alarm 0;
};
alarm 0;
warn "endless loop"
```

```
if ($loop && $loop =~ /^loop/i);
```

In analoger Weise können auch Timeouts behandelt werden, wie Sriram Srinivasan in [3] Kapitel 5, Absatz ‚Using eval for time-outs‘ beschreibt.

Das Problem liegt darin, dass nicht alle Plattformen ‚alarm‘ unterstützen, was uns eine höchst unwillkommene Plattformabhängigkeit beschert. Siehe Script b7.pl

Mit memoize gepufferte Subroutinen

Wir unterscheiden zwischen folgenden zwei Möglichkeiten:

- die Ausnahme trifft in der memoizierten Subroutine selbst auf oder
- die Ausnahme trifft im memoize Mechanismus selbst auf.

Im ersten Fall soll die Fehlerbehandlung des client codes die Ausnahme behandeln, und die memoization soll davon unberührt bleiben. Im zweiten Fall hingegen, soll die Ausnahme zuerst vom memoization code behandelt werden und dann der Fehlerbehandlung des clients code weitergegeben werden. Eine weitere Möglichkeit wäre, nur die memoization zu eliminieren und weiterverarbeiten, so als ob die Funktion nicht mehr memoiziert wäre. Das ist allerdings während einer früheren Testphase angebracht, sonst aber eher zu riskant.

Mit Tie verbundene Strukturen

Mit dem Modul Tie kann man einer Variablen eine individuelle Schnittstelle zuweisen. Diese kann anwendungsspezifische Aspekte einbeziehen, und somit unter Umständen auch die Notwendigkeit Ausnahmen auszulösen. Die Frage wie Perl diese Ausnahmen behandelt liegt demzufolge nahe. In der Literatur wird nicht auf eine Sonderbehandlung hingewiesen. Manche CPAN Tie-Modules, wie z.B. Attribute::Tie, lassen es dem Programmierer frei, einen eigenen Error-Handler zu spezifizieren, auch wenn die voreingestellte Option ist, eine Ausnahme auszulösen.

Try - blocks

Das CPAN Modul Error implementiert C/C++ ähnliche Konstrukte. Obwohl sie sehr hilfreich

sein können, weisen sie aber auch einige Nachteile auf:

- Das try-Konstrukt muss peinlich genau kodiert werden, will man nicht mit sehr lustigen Fehlern konfrontiert werden!
- Es wird ausgiebig von ‚eval‘ Gebrauch gemacht, was die Performance nicht unbedingt fördert.
- Die erzeugten Ausnahmen können entweder Objektinstanzen oder auch einfache Zeichenketten sein. Da dies in den Klauseln abgefragt werden muss, vergrößert sich die Menge notwendigem Quellcode.
- Ausnahmen müssen nach aussen in den Klauseln manuell propagiert werden.
- Wird in einer Klausel eine Ausnahme mit ‚die‘ abgesetzt, so werden die entsprechenden ‚otherwise‘ resp. ‚finally‘ Klausel nicht ausgeführt.
- Wird eine nicht definierte Ausnahme gezogen, so wird ein Syntax-Fehler induziert.
- Wird eine nicht definierte Ausnahme in einer catch - Klausel angegeben, so wird sie ignoriert.
- Wird in einer Klausel der Perl Befehl ‚next‘ benutzt, so muss er ein Label aufweisen.

```
use Error qw(:try);
use Error::Unhandled;

try {
    # the code block to be protected
    my $rv = &foo();
    die "error!" unless
$rv;
}
catch Error::IO with {
    # specialized handler
    my $E = shift;
    print STDERR "File ",
    $E->{'-file'},
    " had a problem\n";
}
except {
    # handler for all
other
    my $E = shift;
    my
$general_handler=sub {send_message
$E->{-description}};
    return {
    UserException1 =>
$general_handler,
    UserException2 =>
$general_handler
    };
}
otherwise {
    ## additional handler
```

```
        my $E = shift;
        print $E-
>stringify."\n";
    }
finally {
    ## always processed handler
    close_the_garage_door_already();
    # Should be reliable
}; # Do not forget this trailing ;

sub foo {
    throw Error::Unhandled(
        unhandled => sub {
            print "Not handled\n";
            exit}
        });
}
```

Zusammenfassend kann man sagen, dass das Modul für einfacheren Anwendungen gut ist. Für Anwendungen mit grossen Datenmengen oder sehr performanten Prozessen ist es jedoch nicht zu empfehlen .
Siehe Scripts b8x.pl, b8z.pl, b8u.pl b8uu.pl.

Perl/Tk

Perl Tk bietet einige zusätzliche Möglichkeiten in Zusammenhang mit Ausnahmen:

- Die Methoden Tk::catch() und Tk::break().
- Das Modul Tk::After.
- Die Methode Tk::Error und Tk::ErrorDialog.
- Callbacks werden evaluiert.

Tk::catch(<block code>)

Erlaubt das Ausführen von Block-code, wobei allfällige Ausnahmen durch \$@ signalisiert werden.

Tk::break()

Diese Methode erlaubt das Abarbeiten eines Callbacks unmittelbar zu unterbrechen, alle hängige Tk-Ereignissen zu ignorieren, und zum Mainloop zurückzukehren. Tk::break funktioniert im wesentlichen wie eine ‚die‘-Ausnahme , ohne dass Fehlermeldungen herausgegeben werden, und ohne dass das Prozess terminiert.

Siehe Script b1.pl

Tk::after

Dieses Modul kann dazu missbraucht werden, um endlose Schleifen zu terminieren.

Siehe Script b8t.pl

Tk::Error und Tk::ErrorDialog

Will man nach einer Ausnahme in einem Tk-Callback noch eine spezifische Verarbeitung

ausführen lassen, so muss man sie in einer ‚cleanup routine‘ einbetten, wie folgendes Beispiel zeigt.

```
sub Tk::Error { ## cleanup code
my $mw = shift;
## insert here exception's handling
}

## or alternatively

use Tk::ErrorDialog;

my $error = ErrorDialog->new(-
cleanupcode => \&cleanup);

sub cleanup {
# here goes the application's
# cleanup code
}
```

Die wichtigste Aufgabe dieser Routine ist die Anwendung entweder wieder in einem bekannten korrekten Zustand zu bringen, so dass sie weiter arbeiten kann, oder mit möglichst kleinem Schaden abzuschliessen.

Siehe Script b8m.pl

Leider kann das Modul Tk::ErrorDialog den cleanupcode nur auf der Ebenen der Anwendung deklarieren, denn bei jeder Instanzierung lässt es die Methode Tk::Error neu kompilieren. Das hat auch den Nebeneffekt, dass ein lokaler cleanupcode ohne zusätzlicher programmatischer Hilfskonstruktion nicht deklariert werden kann. Siehe Script b8m.pl

Callbacks werden evaluiert

Perl/Tk evaluiert die Callbacks seiner Widgets, so dass darin ausgelöste Ausnahmen nur die Verarbeitung des Callbacks, und nicht die Anwendung terminieren. Will man nach einer Ausnahme die Anwendung auch noch terminieren, so muss es ausdrücklich implementiert werden. Das ist zu beachten, wenn ein Perl Skript zu einem Perl/Tk Skript umgewandelt wird, oder auch wenn einen Perl-Skript in einem Perl/Tk Callback, zum Beispiel als Subprozess in einem GUI, aufgerufen wird. Selbstverständlich lassen sich try-blocks in Tk-Callbacks einsetzen. Siehe Script b8y.pl .

Schlussfolgerungen

Auf Grund der geschilderten Konstrukte können wir Folgendes festlegen

- Die breite Verwendung von Ausnahmen als Grundlage einer Fehlerbehandlung ist uneingeschränkt zu empfehlen. In der Tat unterstützt die Ausnahmebehandlung von Perl alle erdenklichen Skript- und Variablenkombinationen.
- Mit Hilfe von geschachtelten Ausnahmen kann die Fehlerbehandlung der Strukturierung der Anwendung angepasst werden.
- Die Möglichkeit Quellcode zu evaluieren lässt es zu, Anwendungsschichten isoliert zu behandeln, sodass wahlweise die Ausnahmen nur lokal behandelt oder weitergegeben werden.
- Die Möglichkeit dem Error-Handler Referenzen zu strukturierten Datenstrukturen weiterzugeben erlaubt es, Ausnahmen sehr effizient zu identifizieren und mit Kontextangaben zu ergänzen. Der Error-Handler kann eine anwendungsorientierte Verarbeitung ausführen, ohne auf globalen Variablen zurückzugreifen.
- Fehlerbehandlungen lassen sich problemlos refaktorisieren, wenn die Anwendung bereits einigermaßen strukturiert ist.
- Fehlerbehandlungen lassen sich nachträglich einbauen, wenn die existierende Schichtung der Anwendung es zulässt. Das bedeutet, jede Schicht hat seinen eigenen Error-Handler.
- Ausnahmen in den BEGIN, CHECK, INIT Blocks sind konzeptionell sehr elegant, aber praktisch wegen ihrem etwas seltsamen Verhalten problematisch.
- Ausnahmen im Package code sollten vermieden werden.
- Der Einsatz von try-Blocks kann Vorteile bringen, braucht aber ziemlich viel Mehraufwand. Auch seine schon erwähnte problematische Implementation lässt manche Zweifel zu. Siehe dazu die Anmerkungen in [10].
- Eine durchgezogene OO-orientierte Fehlerbehandlung ist nur in grösseren Anwendungen von Vorteil. In der Tat ändern die Klassen nichts an der grundsätzlichen Behandlung der Ausnahmen. Sie kleiden sie nur in einem OO-Kostüm ein. Von Vorteil ist aber sicher innerhalb einer Anwendung oder einer Anwendungsgruppe eine Standardisierung der Ausnahmenidentifikation, der Kontextinformation und des grundsätzlichen Aufbau eines verallgemeinerten Error-Handlers. Siehe dazu [11] und [12].

Einerseits bietet Perl alles was es braucht, um fehlertolerante Skripte mit einem vernünftigen Aufwand zu implementieren. Andererseits muss eine zweckmässige Strukturierung angewandt werden, damit überhaupt eine sinnvolle Fehlerbehandlung implementiert werden kann. Denn auch wenn die Perl-Konstrukte zur Fehlerbehandlung ziemlich einfach aussehen, so werden die anwendungsspezifischen Error-Handler schnell kompliziert und schwer zu testen. Die Möglichkeit, dass Fehler in diesen Error-Handler selber auftreten, wird mit steigender Komplexität ziemlich wahrscheinlich, sodass eine zusätzliche Fehlerbehandlungsschicht eingebaut werden muss. Und da ist der Programmierer gefordert. Denn gute Lösungen aus einem Guss sind eher die Ausnahme. Meistens kristallisiert sich eine gangbare Lösung nach mehreren Versuchen und einer gewissen Betriebszeit. Und natürlich auch einer gehörigen Portion Frust.

Literatur

- [1] 2005, Mark Jason Dominus: Higher Order Perl
- [2] 2005, D. Conway: Perl Best Practices
- [3] 1997, Sriram Srinivasan: Advanced Perl Programming

Links

- [5] Arun Udaya Shankar, 2002: Object Oriented Exception Handling in Perl.
<http://www.perl.com/pub/a/2002/11/14/exceptions.html>
- [6] Perl Design Patterns TinyWiki: Error Handling.
<http://perldesignpatterns.com/?ErrorHandling>
- [7] Grant McLean, 2006: Exception Handling in Perl.
<http://wellington.pm.org/archive/200610/exceptions/>
- [8] Stas Bekman, Matt Sergeant: Exception Handling for mod_perl.
http://perl.apache.org/docs/general/perl_reference/perl_reference.html#Exception_Handling_for_mod_perl
- [9] Stas Bekman, Matt Sergeant: Customized __DIE__ handler.
http://perl.apache.org/docs/general/perl_reference/perl_reference.html#Customized___DIE___handler

[e/perl_reference.html#Customized __DIE__ handler](http://perl.apache.org/docs/general/perl_reference/perl_reference.html#Customized___DIE___handler)

- [10] Nilson Santos Figueiredo Junior, 2007, Error::TryCatch, CPAN
- [11] Dave Rolsky, 2008, Exception::Class, CPAN
- [12] Dale M. Amon, 2008, Fault-1.0, CPAN

* Marco Marazzi ist Wirtschaftsinformatiker, lebt in Zürich und ist unter marazzimarco@bluewin.ch erreichbar.