Deutscher Perl Workshop 2015
Manuskript zum Vortrag

Carrot - Raus aus der Ein-Autor-Falle

Winfried Trümper <win@carrot-programming.org>

Der Rahmen

Die Entwicklung von Computer-Anwendungen ist in den vergangenen Jahrzehnten immer aufwändiger geworden. Gestern genügte das Stanzen einer Lochkarte mit einem mathematischen Algorithmus. Heute müssen Anwendungen vernetzt, multimedial und hochgradig interaktiv sein. Der Entwicklungsaufwand ist enorm gestiegen und erfordert Teams, die sich über verschiedene Fachdisziplinen erstrecken.

Beispiel: Zum Erstellen einer Website wird ein Team aus zwei Personen benötigt: Grafiker, Kommunikations- und HTML-Designer, (no?)SQL-Experte, Systemadministrator, Programmierer. Ups, so schnell sind es mehr als zwei. Und außerdem vielleicht drei Programmierer gleichzeitig, damit das Projekt schneller fertig wird. Außerdem könnte sein, dass ein Programmierer bald wechselt. Und so weiter.

Wird die Vergrößerung eines Teams durch einen Effekt erschwert, so spricht man von einem negativen Skalierungseffekt. Wird die Vergrößerung erleichtert, dann von einem Positiven.

Im Perl-Giftschrank findet man einige negative Skalierungseffekte. Allen voran das DWIM-Prinzip: "mach was ich meine". Je mehr Programmierer, desto mehr unterschiedliche Meinungen kommen zusammen. Wer hat wann was womit "gemeint"? Sich davon ernährende Bugs sind schwer aufzuspüren. Carrot vermeidet DWIM und setzt beispielsweise auf ausdrückliche Langnamen für Bezeichner wie etwa $localized_messages.

Zu den positiven Skalierungseffekten kann man erstens die Objektorientierung und zweitens problemspezifische Dateien zählen.

Die Objektorientierung erlaubt eine Aufsplittung von Programmcode in Einheiten, die dann von mehreren Programmierern gleichzeitig bearbeitet werden können. Insbesondere erlaubt sie das Beifügen zu Existierendem. Ein Autor kann auf der Arbeit eines anderen Aufbauen. Carrot bringt Objektorientierung für Perl 5.

Jpeg ist eine problemspezifische Datei. Der Grafiker kann also unabhängig vom Programmierer arbeiten. Das ist nicht selbstverständlich, wenn man sich die Mischung verschiedener Codes in einer PHP-Datei ansieht.

Carrot ermöglicht jeder Klasse die Anbindung problemspezifischer Dateien. Beispielsweise hat jede Klasse eine Konfiguration bestehend aus Definition, Voreinstellungen und aktuellem Wert. Die Konfiguration der Anwendung ist damit eine potentielle Summe der Einzel-Konfigurationen. Skaliert prima.

Braucht man das? Antwort: Die Ein-Autor-Falle hat kein Auslöse-Geräusch. Und sie ist in ihrer Funktionsweise nicht ortsgebunden. Man kann sie aber am Köder erkennen: "Wie soll denn ein Pixelfehler ein Großprojekt attackieren?" Absolute Argumente sind ein prima Köder für die Ein-Autor-Falle. Was man im ersten Moment noch abschätzig als Pixelfehler bezeichnet, kann ich im nächsten Moment - also nach Skalierung - als ganzer Absatz Text entpuppen.

Eine Perspektive auf Carrot: es unterstützt die Skalierung von Programmier-Projekten. Und zwar ohne Rücksicht auf den Perl 5 Mainstream, aber unter Berücksichtigung der Einschränkungen von Perl 5.

Code-Beispiele

Im ersten Beispiel wird das Carrot-Plugin für Block Modifier vorgestellt. Block Modifier befinden sich zwischen Blocknamen und öffnender Klammer. Sie sind durch Kommentarzeichen geschützt. Nach dem Schrägstrich folgt der Name des implementierenden Unter-Plugins. Zwei Schrägstriche erlauben eine mehrzeilige Argument-Liste.

Beispiel 1

```
package Carrot::Individuality::Controlled::Class_Names::Monad
# /type class
# /capability "A managed monad for handling of class names"
# //parent_classes
#       ::Individuality::Controlled::_Corporate::Monad
# //parameters
#       inheritance  ::Modularity::Object::Inheritance::ISA_Occupancy
{
        ...
}
```

Im obigem Beispiel 1 wird der Hauptblock „package“ zu einer Klasse. Er erhält eine Kurzbeschreibung mit /capability, Eltern-Klassen mit /parent_classes und schließlich wird ein Übergabeparameter vereinbart. Das könnte man auch alles anders erreichen. Aber in Carrot wird es gleichförmig erreicht, wie beim Unterblock „sub“ zu erkennen ist:

Beispiel 2

```
sub attribute_construction
# /type method
# /effect "Constructs the attribute(s) of a newly created instance"
# //parameters
#       meta_monad  +required  ::Meta::Monad
#       mapping  +required
# //returns
{
        ...
}
```

Im obigen Beispiel 2 wird der Unterblock „sub“ zu einer Methode. Er erhält die Kurzbeschreibung mit /effect, und er werden Übergabe- und Rückgabeparameter vereinbart. Auch hier könnte man das auf wilde Art anders erreichen. Aber in Carrot gelten die Regeln für Block Modifier universell: package, sub, foreach, while, usw. werden gleich gehandhabt. Das skaliert gut. Sowohl für die Plugin-Autoren, wie für die Plugin-Nutzer. Autoren können sich voll auf den Mehrwert ihrer Plugins konzentrieren, weil sie durch die einfachen Regeln von anderen Autoren unabhängig sind. Es ist attraktiv, ein weiteres Plugin zu schreiben, solange die Idee überzeugt.

Ein Gegenbeispiel ist ein neues globales Schlüsselwort für ein Plugin, welches auch noch konfliktfrei importiert werden muss. „Neue (Pseudo-)Syntax“ ist ein oft bereuter Standard-Ansatz in Perl, weil er nur Autoren mit enormen Ressourcen zur Verfügung steht. Wie sich hinterher herausstellt. An diesem Beispiel sieht man, dass man sich für starke Skalierung vom Perl 5 Mainstream lösen muss.

Viele der Plugins für die Plugins für das Block-Modifier-Plugin basieren auf einer Filterung oder Generierung von Source Code. Aber es gibt noch viele andere Plugins in Carrot, die nur zur Laufzeit aktiv sind. Daher kann man Carrot nicht zutreffend als System der Source Code Filterung bezeichnen.

Den Paketnamen kommt bei der Skalierung eine besondere Bedeutung zu. Denn große Computerprogramme bestehen aus vielen Teilen. Und viele Teile führen zu vielen Namen aus sich wiederholenden Komponenten. Im ersten Teil des folgenden Beispiels sind die redundanten Abschnitte grau markiert. Im zweiten Teil sind die Abschnitte durch einen Anker und relative Namen ersetzt. Beide machen die Verwendung von vielen Namen attraktiver.

Beispiel 3

```
Carrot::Modularity::Constant::Familiar::Explicit
Carrot::Modularity::Constant::Familiar::Ordered_Attributes
Carrot::Modularity::Constant::Local::Subroutine_Parameters


::Modularity::Constant::
        ::Familiar::Inheritable_Constant
        ::Familiar::Ordered_Attributes
        ::Local::Subroutine_Parameters
```

Relative Paketnamen erkennt man an dem führenden Separator :: und Anker erkennt man am hängenden Separator. Relative Anker sind möglich. Der Standard-Anker ist Carrot::. Carrot versteht außerdem die Joker * für die Namen der ersten Unter-Ebene und ** für die rekursive Suche nach Paketnamen.

Man muss auch hier wieder betonen, dass Skalierung die Absicht hinter der Notation ist. Denn im Extremfall hat man mehr Namen als Anweisungen mit Nutzlast. Beispiel: Mit Name1 wird Objekt1 erzeugt und mit Name2 Objekt2. Bis hierhin ist noch nicht viel passiert, aber es wurden schon zwei Namen benötigt. Schließlich verwendet Objekt1 das Objekt2 oder umgekehrt. Fazit: Eine Anweisung, zwei Namen. Das kommt in Carrot in ähnlicher Form häufig vor.

[Drei untergeordnete Code-Beispiele aus Platzgründen gelöscht]

Monaden sind ein zentraler Baustein von Carrot. Es handelt sich dabei um Konstrukte mit begrenzter Individualität. Bekannt sind Monaden aus der funktionalen Programmierung, von globalen Variablen oder als Ergebnis von C++/Java Factories. Carrot stellt jedem Paket eine Meta-Monade bereit, mit der weitere Monaden heraufbeschworen werden können. Beispiel:

Beispiel 4

```
my $expressiveness = Carrot::individuality;
$expressiveness->provide(
        my $customized_settings = '::Individuality::Controlled::Customized_Settings');

$customized_settings->provide_plain_value(
        my $timeout_build_response = 'timeout_build_response',
        my $context_name = 'context_name');
```

Die Meta-Monade befindet sich in $expressiveness. Sie ist für jedes Paket eindeutig. *Sie ist das Paket.* In Objektform lässt sich das Paket sehr komfortabel herumreichen. Natürlich kostet dieser Komfort erhebliche Ressourcen im Vergleich zum Paketnamen, der klassischerweise herumgereicht wird.

Die Beschwörung von $customized_settings ist etwas langatmig. Daher darf man den grau hinterlegten Teil auch weglassen, wenn er zu Beginn eines Pakets steht. Ein anderes Plugin ergänzt die Beschwörung gebräuchlicher Monaden automatisch. Oder man nimmt den Weg über den konfigurierbaren Alias `$expressiveness->customized_settings->...`

Durch $customized_settings erhält jedes Paket eine eigene Konfigurationsdatei. Genau genommen erfordert $customized_settings eine Definitionsdatei mit Voreinstellungen (Beispiel 5). Die Setzungen können dann entweder in einer Konfigurationsdatei mit Namen Paketname.cfg vorgenommen werden. Oder im Abschnitt [Paketname] der Konfigurationsdatei *.cfg (Beispiel 6). Diese Dateien werden in einem konfigurierbaren Suchpfad erwartet. Im Falle mehrer Dateien überschreiben oder akkumulieren sich die Setzungen je nach Typ.

Beispiel 5

```
name    features
--------
table   ::Structure::Table::Format::Aggregated_MxOxN
source  ::Source::Here::Plain
        *---------------+------------------------*
        | program       | plugin                 |
        +===============+========================+
        *---------------+------------------------*
column  ::Valued::Raw
column  ::Valued::Perl::Package_Name::Wild_With_Parameters

name    context_name
--------
flat    ::Valued::Raw
source  ::Source::Here::Plain
        program
```

Im obigen Beispiel ist eine Definitionsdatei zitiert. Ohne die Details im einzelnen zu verstehen fällt sofort auf, dass Paketnamen eine zentrale Rolle spielen. Die Monade $customized_settings schlägt den Bogen zurück zur Wichtigkeit von Namen. Dies geht soweit, dass Programmierung auf eine Auflistung von Paketnamen reduziert wird. Statt eines main-Blocks in einer .pl-Datei schreibt man in Carrot eine Paketliste in einer .cfg- oder .ini-Datei.

[Zwei untergeordnete Code-Beispiele aus Platzgründen gelöscht]

Die vier gebräuchlichsten Monaden sind in Beispiel 6 gezeigt. Ihre Verwendung ist stets gleichförmig. Einer provide-Methode wird ein String in einer skalaren Variablen übergeben. Der String wird durch ein Objekt geeigneten Typs ersetzt (z.B. gleichnamig im Fall von Klassennamen). Nach dem Aufruf von provide_plain_value enthält **$timeout_build_response** einen Zahlwert.

Beispiel 6

```
$customized_settings->provide_plain_value(
        my $timeout_build_response = 'timeout_build_response',
        ...
$class_names->provide(...
$class_names->provide_instance(...
$distinguished_exceptions->provide(...
$localized_messages->provide(...
```

Das Package-Dot

Der Mangel von Tests ist eine verbreitete Diagnose und Läuterung wird auf jeder nächstbesseren Konferenz gepredigt. Schuld sind allerdings nicht die vielgescholtenen Programmierer, sondern die Idee des .t-Verzeichnisses: es skaliert nicht. Also ein Fall für Carrot.

Eine Kultur des Testens fällt nicht vom Himmel, sobald ein Programm fertiggestellt ist. Sondern die Kultur des Testens muss eine frühe Keimzelle haben. Möglichst schon beim ersten Prototypen. Aber wo speichert man ein proof-of-concept.pl? Im .t-Verzeichnis stört es nur.

In Carrot wird das Problem allgemein durch das Package-Dot-Verzeichnis gelöst. Siehe Beispiel 7. Zu jedem Perl-Modul gibt es also ein Verzeichnis, in dem man beispielsweise das proof-of-concept.pl speichern kann. Für einen Prototypen völlig ausreichend.

Das Package-Dot dient darüber hinaus zur Speicherung von temporären Dateien und übersetzten Dateien. So wird beispielsweise die default_settings.cdf in eine default_settings.pl übersetzt, um wertvolle Start-Zeit zu sparen.

Beispiel 7

```
Some_Module.pm
Some_Module./           notice the dot
        documentation/
        unit_tests/
        localized_messages/

        default_settings.cdf
        default_settings.pl
        default_settings.ini
        managed_modularity.pl
        manual_modularity.pl
        shadow-development.tmp.pm
        shadow-development.pm
```

Zugang zur Theorie

Carrot stellt Ausdruckskraft bereit. Ausdruckskraft setzt sich zusammen aus Diversität, Modularität und Individualität. Diese drei kann man hilfsweise zeitlich verstehen als die Phasen vor, während und nach dem Laden eines Moduls. Daher drei.

Durch die Assoziation externer Dateien demonstriert Carrot ein generisches Verständnis von Objektorientierung, welches über Methoden und Attribute hinausgeht.

Fragen der Teilnehmer

Ist Carrot eine neue Sprache? Im Moment ist Carrot ein Programmierstil für Perl 5.

Was war die Motivation für Carrot? Es ist ein Ableger des Mica Environment. Es enthält alle Teile, die nicht zur Kernfunktion von Mica zählen.

Warum werden die Monaden nicht Singletons genannt? Weil nur die Hälfte der Monaden in Carrot wirklich single sind.

Sind die Block Modifier verkappte Smart Comments? Beide eint die Absicht abseits der Perl-Syntax zu stehen. Allerdings beschränkt sich das Konzept der Block Modifier auf Blöcke, basiert auf Plugins und fordert einheitliche Regeln für die „Comments“. Das ist etwas näher am C Präprozessor.

Ist Carrot ein Template-System? Ist C ein Template-System für Maschinencode? Die Frage, was ein Template-System ist, sprengt den Rahmen des Vortrags.