

## FIT-Tests Kochbuch

Im weiteren wird die Nutzung des FIT-Frameworks für Akzeptanztests, genauer, des Perl-Ports erläutert. Fokus steht bei der Geschäftslogik.

Ein FIT-Test ist ein (HTML-) Dokument, welches die Testdaten sowie weiteren, erklärenden oder erläuternden Text enthält. Die Testdaten werden in Tabellen erfasst, die erläuternde Text darf überall, ausser in den Tabellen vorkommen.

Die Idee hinter FIT-Tests ist, dass man VOR einer Realisierung mit dem Kunden das Sollverhalten bestimmter („kritischer“) Komponenten beschreibt und dass dieses Dokument, so wie es ist, als Testfall abgearbeitet wird. Sie können es auch selbst schreiben, initial, oder gar vollständig, den Nutzen haben Sie trotzdem!

Die Verarbeitung eines solchen Dokumentes sieht so aus:

```
perl perl -MTest::C2FIT -e file_runner eingangsdokument.html
ausgangsdokument.html
```

Das FIT-Modul parst das Eingangsdokument, ermittelt welcher Code ausgeführt werden soll, führt diesen aus, prüft die Ergebnisse mit dem Dokument, erstellt das Ausgangsdokument. Jetzt kommt das wichtige: FIT-Modul erkennt, was womit geprüft werden soll. Stimmen die Inhalte überein, wird die entsprechende Tabellenzelle grün unterlegt. Stimmen die Inhalte nicht überein, wird die Zelle rot unterlegt. Ist ein Fehler aufgetreten, wird gelb verwendet.

Sehen wir uns ein einfaches Beispiel an:

Umrechnung von Frames in Sekunden. Runden auf ganze Sekunden. Ein Fernsehbild wird auf Frames aufgebaut, in Europa werden 25 Frames pro Sekunde verwendet.

<i>ff2sec</i>	
frames	getSekunden()
25	1
26	1
49	2
62	2
63	3

Es ist eine einfache „Geschäftslogik“, das, was dieser Test aussagen soll, ist, wie gerundet wird. In Java würde man sowas eher mit einem UnitTest machen (weil Eclipse-Unterstützung). In Perl hat man diesen IDE Support nicht, der Unterschied zw. Test::Unit und Test::C2FIT ist marginal. Interessant wird es erst dann, wenn der Kunde sowas schreiben oder zumindest prüfen soll.

Interessant ist die erste Zeile, in dieser steht der Name eines Perl-Packages (einer Java-Klasse), die die Verarbeitung der Daten vornimmt. Der Perl-Quellcode dazu sieht so aus:

```

package ff2sec;
use base 'Test::C2FIT::ColumnFixture';
use strict;

sub new {
    my $pkg = shift;
    return $pkg->SUPER::new( frames => undef );
}
sub getSekunden {
    my $self = shift;
    my $ff = $self->{frames};
    return int(($ff+12)/25);
}
1;

```

Der Code ist in einer Datei namens ff2sec.pm abgelegt. (Wird über den use xxx Mechanismus geladen). Die Anweisung use base 'Test::C2FIT::ColumnFixture' sagt, dass mit diesem Package eine abgeleitete Klasse von Test::C2FIT::ColumnFixture (mehr zu Fixtures weiter unten).

Sub new ist der Konstruktor, in diesem wird eine Instanzvariable, „frames“ deklariert.

Das FIT-Modul erkennt anhand der Spaltenüberschrift in der Tabelle, ob mit einer Zelle ein Feld oder eine Methode gemeint ist – eine Methode hat nämlich Klammern. Methoden liefern Werte, mit Feldern werden Werte gesetzt.

**Hinweis:** Perl ist typenlos, in Java müssten entsprechend typisierte Felder deklariert werden!

Pro Zeile wird einmal der Wert von frames gesetzt und einmal der Wert von getSekunden abgefragt. So aufgebaut, enthalten einzelne Zeilen separate, von einander unabhängige Tests.

Für den Produktiveinsatz würde man den Quellcode, der die Geschäftslogik enthält nicht in die Fit-Klasse reinprogrammieren, vermutlich würde man ein Package, nennen wir es tcode.pm, erstellen und darin eine Methode frames2sec anbieten. Dann sähe unser FIT-Test-Code so aus:

```

package ff2sec;
use base 'Test::C2FIT::ColumnFixture';
use tcode;
use strict;

sub new {
    my $pkg = shift;
    return $pkg->SUPER::new( frames => undef );
}
sub getSekunden {
    my $self = shift;
    my $ff = $self->{frames};
    return tcode::frames2sec($ff);
}
1;

```

Wenn wir uns gut fühlen, und weitere zwei Zeilen Quellcode sparen wollen, dann schreiben wir getSekunden um:

```

sub getSekunden {
    tcode::frames2sec($_[0]->{frames});
}

```

Was aber, wenn unsere Geschäftslogik Zugriff auf Stammdaten benötigt?

Zuerst sollte überlegt werden, wie groß das zu testende Modul sein soll. Braucht man **alle** Stammdaten eines Systems, ist das Modul zu groß gefasst. Reichen eine Handvoll Daten aus, kann man in erster Näherung davon ausgehen, dass alles ok ist. Wo bringt man diese unter?

Am besten, direkt im FIT-Test-Dokument, und zwar vor den eigentlichen Testfällen.

Sehen wir und dazu das folgende Beispiel an:

Kalkulatorische Lohnkosten werden in Abhängigkeit des Kalenderjahres und der Leistungsart abgelegt. Das sind unsere Stammdaten. Also erfassen wir sie so:

<i>stammdaten</i>		
jahr	leistungsart	kosten
2004	24	102.5
2005	24	105
2006	24	106
2004	34	54
2005	34	56
2006	40	7.5

Danach erfassen wir unsere Testfälle, wieder in einer Tabelle. Nehmen wir an, es handelt sich um Daten wie Projektnummer, Datum (=wann fällt die Leistung an), Anzahl Stunden, vielleicht noch weitere Daten. Ermittelt werden soll die Summe pro Projektnummer.

Bevor wir damit fortfahren, ein Paar Anmerkungen. Die Stammdaten enthalten als Attribut das Jahr, es handelt sich um eine Nummer. Die Testdaten enthalten ein Datum, wir brauchen also ein Stück Code was ein Datum in ein Jahr umrechnet. Perl ist typenlos, d.h. es liegt an uns eine geeignete Form dafür zu finden. Möglicherweise wäre es eine gute Idee, hierfür einen separaten Test zu machen (den muss der Kunde nicht sehen, weil es für ihn selbstverständlich ist, zu welchem Jahr der 14. Juni 2006 gehört...).

**Ausser** - nicht das Kalenderdatum, sonder das Plandatum ist in den Daten enthalten, oder es gibt abstruse Regeln, wann es zum einen, wann zum anderen Tag gehört...

So hier unsere Daten:

<i>kosten1</i>			
datum	leistung	anzahlStunden	kosten()
11.02.2004	24	4	410
11.02.2004	34	10	540
04.04.2006	40	2	15

Wie stammdaten.pm und kosten1.pm aussehen müssten, das können wir uns vorstellen. Wie aber

teilt man kosten1 mit, dass die Daten von stammdaten.pm zu verwenden sind?

Anwort: Es muss etwas globales her!

In Java würde man so etwas mit einem Singleton machen. Einen Singleton kann man auch in Perl realisieren, in unserem Beispiel kommen wir aber mit einer globalen Variablen aus.

Je nach Präferenz des Programmierers wird für die Speicherung solcher Daten, wie in der Tabelle stammdaten angegeben, entweder ein Array oder ein Hash-Array (Arrayref bzw. ein Hashref) verwendet. Wie die Zelleninhalte einer Zeile in das ColumnFixture-Objekt gelangen, haben wir gezeigt. Was wir brauchen, ist eine Methode, die dann aufgerufen wird, wenn das Fit-Framework am Ende der Zeile angelangt ist. Genauer genommen, zwei Methoden wären besser: am Anfang und am Ende einer Zeile. FIT hat so etwas vorgesehen. Was wir tun müssen, ist, die Methoden reset (Anfang einer Zeile) und execute (Ende einer Zeile **oder** vor dem ersten Methodenzugriff) zu implementieren.

```
package stammdaten;
use base 'Test::C2FIT::ColumnFixture';
use strict;

sub new {
    my $pkg = shift;
    return $pkg->SUPER::new();
}
sub reset {
    my $self = shift;
    $self->{jahr} = undef;
    $self->{leistungsart} = undef;
    $self->{kosten} = undef;
}

sub execute {
    my $self = shift;
    die unless defined($self->{jahr});
    die unless defined($self->{leistungsart});
    die unless defined($self->{kosten});

    $::stammdaten = {} unless ref($::stammdaten); # glob. Variable
erzeugen
    $::stammdaten->{$self->{jahr}}->{$self->{leistungsart}} = $self-
>{kosten};
}
1;
```

Die Berechnungsvorschrift, die in bei kosten1 zum tragen kommt greift entweder auf die globale Variable \$::stammdaten zu, oder bekommt diese als Parameter übergeben.

Sollte die Methode kosten() nicht nur die Summe über eine Zeile, sondern über alle Zeilen erstellen, kann die benötigte Zwischensumme als Instanzvariable in der Berechnungsvorschrift abgelegt werden.

Was aber, wenn die Berechnungsvorschrift komplizierter ist? Bzw. wenn die Ergebnisse gänzlich anders aussehen, als die Eingangsdaten?

Schauen wir uns das nächste Beispiel an:

Hier soll eine Summe pro Kostenträger gebildet werden.

Wir gehen davon aus, dass die gleiche stammdaten Tabelle benötigt wird. Unsere Eingangsdaten sehen dann so aus:

<i>kostenI</i>			
datum	leistung	anzahlStunden	kostentraeger
11.02.2004	24	4	156110
11.02.2004	34	10	158777
04.04.2006	40	2	156110

Es ist kein Tippfehler, dass als Klasse wieder *kostenI* verwendet wird! Wir können die gleiche Klasse/Package nutzen, mit zum Teil anderen Feldern. In unserem ersten Beispiel war *kostentraeger* für den Testfall - die Berechnung *Jahr/Leistungsart/anzahlStunden* - nicht relevant. Daher konnten wir diesen weglassen. Jetzt interessiert uns *kosten()* nicht, weil wir hier nur unsere "Eingangsdaten" angeben.

Kommen wir zu der Festlegung der Ausgangsdaten:

<i>kostenProKtr</i>	
kostentraeger	gesamtbetrag
156110	425
158777	540

Ob das Ergebnis unserer Berechnung stimmt, das prüft das FIT-Framework für uns. Wie stellen wir ihm aber unsere berechneten Daten zur Verfügung? Sehen wir uns den Quellcode an:

```

package kostenProKtr;
use base 'Test::C2FIT::RowFixture';
use strict;

sub new {
    my $pkg = shift;
    return $pkg->SUPER::new();
}
sub query {
    my $self = shift;
    return [
        { kostentraeger => '156100', gesamtbetrag => 425 },
        { kostentraeger => '158777', gesamtbetrag => 540 }
    ];
}
1;

```

Man sieht sofort, ich war faul :-). Ich berechne keine Daten, sondern liefere das zurück, was ich in meinem Dokument angegeben habe.

(Im Grunde genommen bin ich nicht faul, ich will zunächst sicherstellen, dass das ganze drumherum stimmt, bevor ich mich an die Berechnungsvorschrift ranmache und dann womöglich an völlig falscher Stelle Fehler suche...)

Wichtig sind zwei Dinge: kostenProKtr ist eine abgeleitete Klasse von RowFixture (mehr zu den Fixtures weiter unten), diese Klasse implementiert die Methode query, welche wiederum die gewünschten Daten als Arrayref liefert. Inhalt des Arrayrefs sind Hashrefs, es könnten aber auch Instanzen von anderen Klassen sein, dann würde ich (mit großer Wahrscheinlichkeit) in der Tabelle kostenProKtr Methodennamen verwendet (also getKostentraeger() etwa an Stelle von kostentraeger)

Nehmen wir an, unsere Geschäftslogik unterstützt weitere Parameter - oder lässt sich weiter konfigurieren - und diese Parameter sind nicht intuitiv in einer Tabelle untergebracht.

In so einem Falle würde in unserem Testdokument etwa folgender Abschnitt kommen (vor den Ergebnisdaten!):

<i>fit.ActionFixture</i>		
start	meineOperationen	
check	anzahl	3
enter	setIgnoreLeistungsart	40
check	anzahl	2

Zugegebenermaßen, etwas an den Haaren herbeigezogen, das Beispiel, meine ich.

fit.ActionFixture ist ein Standard-Fixture des FIT-Frameworks. In der ersten Spalte steht bei diesem

Fixture ein "Befehl". ActionFixture kennt vier Befehle: start - Instanz einer Klasse wird erzeugt. check - eine Methode wird aufgerufen, der Rückgabewert wird geprüft. enter - eine Methode wird aufgerufen, keine Prüfung des Rückgabewertes, press - eine Methode wird aufgerufen, keine Parameter, keine Rückgabewerte. Wie sieht der Quellcode von meineOperationen aus? Sehen wir uns das Beispiel an:

```
package meineOperationen;
use strict;

sub new {
    my $pkg = shift;
    return bless {}, $pkg;
}

sub setIgnoreLeistungsart {
    my $self = shift;
    my $leistungsart = shift;
    # hier kommt der entsprechende Code
}

sub anzahl {
    my $self = shift;
    return scalar(@$::kosten);
}
1;
```

Als erstes fällt einem auf, dass es sich um keine abgeleitete Klasse handelt, zumindest muß keine Klasse des FIT-Framework verwendet werden. D.h. wir können in einem ActionFixture direkt den Produktivcode nutzen.

Mit den bisherigen Beispielen haben wir die wichtigsten Features des FIT-Framework kennengelernt. Nun können wir zu den Feinheiten übergehen. Zuerst aber eine Zusammenfassung der Fixtures.

## Die Fixtures

### ColumnFixture

**Verwendung:** Eine Zeile – Ein Testfall; Eine Zeile – Daten für weitere Tests (d.h. Der Testfall ist ausserhalb der ColumnFixture), Eine Zeile – Testfall bestehend aus dieser Zeile und den vorangegangenen.

Eigene Klasse erbt von Test::C2FIT::ColumnFixture. Wenn der Testfall eine Zeile umfasst, wird typischerweise eine Methode (beliebigen Namens) implementiert. Wenn Daten für andere Tests „eingegeben“ werden, dann wird die Methode execute implementiert.

### RowFixture

**Verwendung:** Enthält erwartete Ergebnisse, der gesamte Tabelleninhalt ist die erwartete Ergebnismenge.

Eigene Klasse erbt von `Test::C2FIT::RowFixture` und implementiert die Methode `query`.

Die Reihenfolge der Spalten in einer `RowFixture`-Tabelle ist wichtig! Die Spalten werden von links nach rechts als `Key`-Spalten verwendet, d.h. Die Reihenfolge der Spalten entscheidet darüber, ob FIT eine Differenz in einer Zeile als „Datensatz fehlt“ oder „Wert anders als erwartet“ ausgibt!

## ActionFixture

**Verwendung:** Sequenzen von Befehlen abarbeiten

## TypeAdapter

In unserer Logik verwenden wir häufig Abbildungen, die der Verarbeitung dienlich sind. "Lesbare Werte" werden i.d.R. erst ganz am Ende (d.h. ausserhalb des Scopes unserer Tests) erzeugt.

Dauer wird häufig intern als "Anzahl Sekunden" gehalten, ausgegeben wird aber so etwas wie "84:11" oder "01:24:11".

Für solche Fälle bietet das FIT-Framework auch eine Unterstützung, und diese heisst `TypeAdapter`.

Es liegt in der Natur der Sache, dass die Typunterstützung unter Java besser ausfällt als unter perl, weil perl ja typenlos ist. Mal angenommen, intern halten wir Frames, die Anzeige soll aber in der Form "HH:MI:SS:FF" erfolgen. Dazu definieren wir einen `TypeAdapter`, der Code sieht so aus:

```
package FpTypeAdapter;
use strict;
use base 'Test::C2FIT::TypeAdapter';

sub parse {
    my $self = shift;
    my($s) = @_;

    return undef unless defined($s);
    die "Unbekanntes Format $s" unless $s =~
/^(\d\d):(\d\d):(\d\d):(\d\d)$/;

    my $hh = $1;
    my $mm = $2;
    my $ss = $3;
    my $ff = $4;

    die "Zu viele Frames $ff" if $ff > 24;
    die "zu viele Sekunden $ss" if $ss > 59;
    die "zu viele Minuten $mm" if $mm > 59;
    #           wg. Folgetag 03:00:00:00 -> 27:00:00:00
    die "zu viele Stunden $hh" if $hh > 47;

    my $rv = $hh * 3600 * 25 + $mm * 60 * 25 + $ss * 25 + $ff;
    return $rv;
}
1;
```

Ein `TypeAdapter` ist also eine Klasse, die von `Test::C2FIT::TypeAdapter` erbt und eine Methode `parse` implementiert. Parameter dieser Methode ist der String, der im Dokument steht, Rückgabewert ist das, was in den Daten gehalten wird (betrifft sowohl Eingangs- als auch Ausgangsdaten)

Da Perl typenlos ist, muss man ein Paar Zeilen Code schreiben, um Felder (oder Methoden) an einen eigenen TypeAdapter zu binden. In einer Fixture-Klasse wird dies hinterlegt, es gibt mehrere Möglichkeiten. Sehen wir uns diese an:

```
sub new {
  my $pkg = shift;
  my $types = {
    frames => 'FpTypeAdapter',
  };
  return $pkg->SUPER::new( @_, fieldColumnTypeMap => $types );
}
```

In diesem Beispiel werden Spalteinhalte von „frames“ mit dem FpTypeAdapter geparkt. (FpTypeAdapter ist der vollständige Package-Name)

```
sub suggestFieldType {
  my ( $self, $name ) = @_;

  return 'NullFkWrapper' if $name =~ /id$/i || $name =~ /ref$/i;
  return $self->SUPER::suggestFieldType;
}
```

In diesem Beispiel werden alle Spalten, die mit „id“ oder „ref“ enden an den NullFkWrapper gebunden.

## Testen von globalen Subroutinen

(Also subroutinen, die nicht in in einem Package vorkommen)

Erstens: Es ist möglich.

Wrapper-Packages wird man i.d.R. Brauchen. Das Script, was die Subroutine enthält wird in fileRunner.pl mittels require geladen, im Wrapper Script erfolgt dann Zugriff unter Angabe des Main-Packages: &::meine\_sub(); oder &main::meine\_sub()

Was mache ich, wenn mein Script ein komplettes Programm enthält, das laden über require dazu führen würde, dass Code ausgeführt wird, der nicht ausgeführt werden soll.

Erstens: Es ist möglich.

Den Codeteil, der „main“ ist, wird in eine Subroutine Namens main gepackt. Die Subroutine wird im Code so „aufgerufen“:

```
&main unless define $ENV{FIT_TEST}
```

Und in fileRunner.pl steht dann vor dem require() ein:

```
$ENV{FIT_TEST} = 1;
```

## Das Kleingedruckte

Das hier beschriebene bezieht sich auf die Version 0.7, diese kann von CPAN heruntergeladen (<http://search.cpan.org/~tjbyrne/Test-C2FIT-0.07/>)

Im Gegensatz zu java gibt es kein cruisecontrol und keine Unterstützung in der Entwicklungsumgebung (was ist das??? ;-), daher muss man daran denken, die Tests laufen zu lassen.

Word 2000 erzeugt gutes html, Word97 eher mäßiges, OpenOffice muß man leider vergessen...

## Links

<http://fit.c2.com>