

Redesigning tktetris, von PERL nach OO-PERL.

Scripts sind meistens flache Programmstrukturen mit wenigen Abstraktionsebenen.

Erreicht ein solches Script eine gewisse Komplexität, so muss es zuerst einem Redesign unterzogen werden, um überhaupt noch weitere Anpassungen resp. Erweiterungen vernünftig einbauen zu können. Ein solches Vorgang wollen wir am Beispiel des Scripts tktetris.pl schildern, welcher systematisch von einer klassischen prozeduralen Perl-Implementation nach OO-Perl umgestaltet wird.

Problemstellung und Eingrenzung

Dabei ist die Art, wie man ein solches Redesign umsetzt, nicht unwichtig, denn allzu gewagte Umstellungen Risiken bergen, welche nicht zu unterschätzen sind, vor allem dann wenn die Zeit, die Systemlogik bis ins letzte Detail zu studieren, einfach nicht da ist, und man muss mit dem Vorhandenen Wissen dennoch anfangen!

Das bedeutet, zuerst die Implementation selbst analysieren, und so anzupassen, dass eine Strukturierung entsteht, ohne dass dabei die Funktionalität und die Arbeitsweise des Systems geändert wird. Dann die erkannte Strukturierung in einem OO-System stufenweise umwandeln, und schlussendlich Verbesserungen anbringen. Dabei soll das System nach jedem Schritt immer voll funktionsfähig bleiben.

Die erste Stufe basiert grundsätzlich auf die Analyse des Quellcodes. Es wird zuerst versucht, Quellcode-Sequenzen die bestimmte klar zu erkennende Funktionalitäten aufweisen, unverändert in Subroutinen zu verlagern. Dann werden die Subroutinen in Packages gruppiert, so dass bereits eine Modularisierung entsteht.

In der zweiten Stufe werden die Subroutinen zu Methoden umgeschrieben, und die Packages zu instanzierbaren Klassen ausgebaut. In dieser Phase werden die Konstanten und Variablen, die bis anhin immer noch global waren, zweckmässig den verschiedenen Klassen zugeteilt, so dass jede Klasse ausschliesslich für seine Aufgabe verantwortlich ist, und alles dazu Notwendige kapselt. Hand in Hand mit diesem Vorgang geht die Anpassung des Client codes, wo die Instanzierung der Klassen eingefügt, und die Subroutinenaufrufen in Botschaften zu den Klasseninstanzen umgestaltet werden. Das ist die heikelste Aufgabe, denn einerseits der

Quellcode bedeutend angepasst werden muss, und andererseits weil die Richtigkeit der Umgestaltung nur durch Tests überprüft werden kann.

In der dritten und letzten Stufe werden funktionale Verbesserungen implementiert, welche die erkannten Schwachpunkte des Systems beheben, und nicht zwingende proprietäre Lösungen durch standard Module ersetzen.

Von PERL nach OO-PERL

Nun wie kann man prozedurales Code stufenweise in OO-code umwandeln?

Für die Umgestaltung des Scripts *tktetris* habe ich folgende Methodik angewandt:

- Subroutinen im Package main bilden. Quellcode-Sequenzen welche eindeutig eine bestimmte und klar erkennbare Funktion erfüllen, werden unverändert in Subroutinen des Package *main* ausgelagert.
- Zugriff auf globalen Variablen minimieren. Globale Variablen in den Subroutinen werden durch lokalen Variablen mit dem gleichen Namen ersetzt, welche mittels Werte aus der Argumentenliste initialisiert werden. Diejenigen globale Variablen, welche nicht lokalisiert werden, werden mit ihren Package-Name qualifiziert. Damit wird der Subroutinencode weitgehend package-unabhängig.
- Packages bilden. Subroutinen des Package *main* werden möglichst unverändert in Packages verteilt. Hier geht es nur darum eine erste Modularisierung zu schaffen, wobei die Subroutinen noch als Klassenmethoden aufgerufen werden.

- Globale Variablen verteilen. Globale Variablen des Package *main* werden zweckmässig auf die Packages verteilt. Hier geht es darum den Packages klare Verantwortlichkeiten zuzuweisen.
- Klassen bilden. Packages werden mit Konstruktoren und accessors versehen. Aufrufe der Klassenmethoden werden in Botschaften zu den Klasseninstanzen umgestaltet. Diese Stufe schliesst die eigentliche Umgestaltung und bildet die Basis für allfällige Erweiterungen.
- Weitere Klassen bilden. Spezifische Funktionalitäten der Klassen werden in eigenständigen spezialisierten Klassen ausgelagert.

Beispiele

Ein möglichst einfaches Beispiel soll die Methodik demonstrieren.

Das ursprüngliche Quellcode zur Berechnung des Umfangs und der Fläche von Kreisen mit Radius 1 bis 9 sieht wie folgt aus:

```
package main;
my ($radius,$umfang,$flaeche);
my $PI = 3.14;
foreach (qw/1 2 3 4 5 6 7 8 9/) {
    $radius = $_;
    $umfang = 2 * $PI * $radius;
    $flaeche = $PI * $radius * $radius;
    print $radius, $umfang, $flaeche
}
```

Zuerst werden die Berechnungen und das Drucken unverändert in Subroutinen des gleichen Package *main* verlagert:

```
package main;
my ($radius,$umfang,$flaeche);
my $PI = 3.14;
foreach (qw/1 2 3 4 5 6 7 8 9/) {
    $radius = $_;
    main::umfang;
    main::flaeche;
    main::drucken
}

sub umfang {
    $umfang = 2 * $PI * $radius
}
sub flaeche {
    $flaeche = $PI * $radius *
    $radius
}
sub drucken {
```

```
}
    print $radius, $umfang, $flaeche
}
```

Dann werden die Subroutinen in einem separaten Package namens *kreis* verlagert. Die Variablen *\$PI*, *\$umfang* und *\$flaeche* werden global deklariert und qualifiziert accessiert. Die Variable *\$radius* wird hingegen als Argument den Subroutinen übergeben.

```
package kreis;
{
sub umfang {
    my ($radius) = @_;
    $main::umfang = 2 * $main::PI * $radius
}
sub flaeche {
    my ($radius) = @_;
    $main::flaeche = $main::PI * $radius *
    $radius
}
sub drucken {
    my ($radius) = @_;
    print $radius, $main::umfang,
    $main::flaeche
}
}
package main;
my $radius;
our ($umfang,$flaeche);
our $PI = 3.14;
foreach (qw/1 2 3 4 5 6 7 8 9/) {
    $radius = $_;
    kreis::umfang($radius);
    kreis::flaeche($radius);
    kreis::drucken($radius)
}
}
```

Nicht zwingenden globalen Variablen werden nun eliminiert, oder dort verlegt wo sie tatsächlich gebraucht werden. Die Variable *\$PI* wird im package *kreis* deklariert. Die Subroutinen *umfang* und *flaeche* werden als Funktionen umgestaltet. Die Subroutine *drucken* wird so ausgebaut, dass sie die von *\$radius* abhängige Werte Umfang und Fläche selbst ausrechnet.

```
package kreis;
{
my $PI = 3.14;
sub umfang {
    my ($radius) = @_;
    return 2 * $PI * $radius
}
sub flaeche {
    my ($radius) = @_;
    return $PI * $radius * $radius
}
sub drucken {
    my ($radius) = @_;
    print $radius, umfang($radius),
    flaeche($radius)
}
}

package main;
my ($umfang,$flaeche);
foreach (qw/1 2 3 4 5 6 7 8 9/) {
    $radius = $_;
```

```

    $umfang = kreis::umfang($radius);
    $flaeche = kreis::flaeche($radius);
    kreis::drucken($radius)
  }
}

```

Perl erlaubt es, durch simples Kompilieren mit dem Schalter `-w` die Korrektheit dieser Umgestaltungen zu überprüfen. Baut man keine semantische Anpassungen, so ist die Chance sehr gross, dass der neue Code auch korrekt funktioniert.

Ein weiteres Beispiel diesmal aus dem Script *tktetris* selbst: das Redesign der Subroutine `main::get_next_block`.

```

my $next_block;

sub get_next_block {
    $next_block_nr = int(rand()*$nmbr_blks);
    $next_dir      = int(rand()*4);
    $next_block    = $block->[$next_block_nr];
    for (0 .. $next_dir) {
        turn($next_block_nr, $next_block);
    }
}

```

Dieselbe Subroutine nun im Package *tetris*

```

my $next_block;

sub get_next_block {
    $next_block_nr = int(rand()*$nmbr_blks);
    $next_dir      = int(rand()*4);
    $next_block    = block->new(-block =>
    $block->[$next_block_nr]);
    for (0 .. $next_dir) {
        block::turn($next_block_nr, $next_block)
    }
}

```

Und so wird sie aufgerufen

```

sub action {
    unless (defined $active_block_nr) {
        tetris::get_next_block() unless
        (defined $next_block_nr);
        $active_block_nr =
        $next_block_nr;
        $main::blocks++;
        $active_block = [];
        block::copyblock($next_block,
        $active_block);
        $active_dir = $next_dir;
        get_next_block();

        block::draw_next_block($next_block);
    }
    ...
}

```

Schliesslich sieht die Methode in der Klasse *tetris* so aus:

```

sub get_next_block {

```

```

    my $self = shift;
    $next_block_nr = int(rand()*
    scalar(@$block));
    $next_dir = int(rand()*4);

    my $newBlock = $block-
    >[$next_block_nr]->copy();

    for (0 .. $next_dir) {
        $newBlock->turn();
    }

    $next_block = 'nextblock'->new(-block
    => $newBlock,-canvas => $main::next-
    >Subwidget('canvas'));
}

```

Und so wird sie angewandt:

```

sub action {
    my $self = shift;
    unless (defined $active_block_nr) {
        $self->get_next_block() unless
        (defined $next_block_nr);
        $active_block_nr =
        $next_block_nr;
        $main::score->addBlocks();
        $active_block = $next_block-
        >block;

        $active_block-
        >canvas($main::field);
        $active_dir = $next_dir;
        $self->get_next_block();
        $next_block->draw();
    }
}

```

Redesigning script *tktetris*

Unter Verwendung der eingangs dargestellten Methodik habe ich die Umgestaltung des Scripts *tktetris* in folgenden Phasen unterteilt.

- Phase I Implementation analysieren und reorganisieren.
- Phase II Packages bilden.
- Phase III Presentation layer und core layer trennen.
- Phase IV Klassen bilden.
- Phase V Klassen weiterentwickeln

Die Stufen entsprechen im Wesentlichen denjenigen der Methodik, mit der Ausnahme von Stufe II und Stufe III.

Stufe II vereinigt die Stufen ‚Packages bilden‘ und ‚Globale Variablen verteilen‘, während Stufe III eigentlich nur ein Sonderfall von ‚Klassen bilden‘ ist.

Die einzelnen Phasen wurden noch in Schritten unterteilt, wobei jeder Schritt ein klares Ziel anstrebt, das einzeln erreicht und ausgetestet werden muss.

Die Phasen im Einzelnen

main	
------	--

Phase I - Implementation analysieren und reorganisieren

In dieser Phase wird zuerst analysiert welche Objekten agieren. Weiter wird der Code umgruppiert, wobei Code-Sequenzen möglichst unverändert in Subroutinen umgelagert werden. Variablennamen und deren Gültigkeitsbereich bleiben unverändert.

Angefangen wird mit eindeutig nebensächlichen Funktionalitäten, dann solche die man eindeutig identifizieren kann. Die wichtigsten Subroutinen bleiben in dieser Phase noch unverändert.

Beispiel:

- Das Aufstellen der main windows mit Widgets wird in der Subroutine `main::populateMainWindow` isoliert.
- Das Setzen der Bindings wird in der Subroutine `main::setBindings` verlegt.

Das Resultat ist eine voll funktionsfähige stark reduzierte *main line* und eine Serie von kleineren Subroutinen im package *main*.

Die einzelnen Schritten

- Analysis

<code>\$next_block,</code> <code>\$next_block_nr,</code> <code>\$next_dir</code>	Objekt block
<code>\$active_block</code> <code>\$help_block</code>	Objekt block
<code>\$points \$blocks, \$lines</code>	Objekt score
<code>@highscores</code>	Objekt highscore
<code>\$playfield</code>	Objekt playfield
<code>\$base_speed,</code> <code>\$base_level,</code> <code>\$level,\$speed, \$block</code>	Objekt tetris (das Spiel selbst)

- Subroutinen implementieren

code chunk in package main	<code>main::populateMainW</code> <code>indow</code>
code chunk in package	<code>main::set_bindings</code>

Phase II - Packages bilden

In dieser Phase werden Packages mit eigenen Gültigkeitsräumen gebildet, welche aber noch keine Klassen sind, so dass die Subroutinen als Klassenmethoden verwendet werden. Variablen werden möglichst lokal deklariert, wenn nötig über Argumentenlisten, oder mit dem entsprechenden Packagename voll qualifiziert.

Im Package *main* existieren immer noch die globalen Variablen, welche aber bereits vom neuem Code angesprochen werden.

Das Resultat ist ein main Script und eigenständige Packages, welche spezialisierte Funktionalitäten aufweisen.

Im Unterschied zu Phase I, bereits hier haben wir eine physische Isolierung der einzelnen Funktionen, welche aber noch auf globalen Variablen zugreifen.

Es werden also die Packages *block*, *tetris*, *nextblock* und *highscore* erstellt.

Die einzelnen Schritten

- Packages *block*, *tetris*, *nextblock* und *highscore* bilden,
- Globale Variablen in den Packages verteilen,
- Aufrufe qualifizieren,
- Argumentenlisten anpassen,
- Accessors implementieren.

Phase III - Presentation layer und core layer trennen.

In dieser Phase wird versucht, alles was TK-abhängig ist, in composite Widget zu isolieren.

Das Resultat ist eine erste voll funktionsfähige Version des Presentationslayers und des Core layers.

Es werden also die Megawidgets *game*, *shadow*, *tnext* und *score* erstellt. Zudem wird die Klasse *timer* implementiert.

Die einzelnen Schritten

- Klasse *timer* implementieren,
- Megawidget *game*, *shadow*, *tnext* und *score* erstellen,

- Megawidgets in Package *main* aufrufen,
- Konstanten und Variablen zu den megawidgets zuweisen.

Phase IV - Klassen bilden

In dieser Phase werden die Packages zu instanzierbaren Klassen umgestaltet. Dabei werden die Konstruktoren ausgebaut und die Methodenaufrufen angepasst. Im Client Code mutieren die Variablenwerte zu Instanzen, die Subroutinenaufrufe zu Botschaften.

Das ist eine heikle Phase, denn der Ablauf angepasst werden muss. Die Klassen müssen wirklich eigenständig sein, das heisst, jede Klasse muss entweder die Angaben selbst enthalten oder sie bei der zuständigen Klasse abholen. Ferner kann diese Umgestaltung nicht vom Compiler verifiziert werden, denn der PERL-Compiler die Gültigkeit von Instanzvariablen und Botschaften nicht überprüfen kann. Diszipliniertes und methodisches Vorgehen ist hier von grösster Bedeutung.

Die Klassen *tetris*, *block* und *nextblock* werden erweitert, die abgeleiteten Klassen *block0* und *block1* werden implementiert.

Das Resultat ist wiederum eine voll funktionsfähige Kollaboration von Instanzen. Damit ist das eigentliche Ziel des Redesigns erreicht. Nun kann die eigentliche Weiterentwicklung eingeleitet werden.

Die einzelnen Schritten

- Konstruktoren implementieren.
- Instanzen kreieren.
- Accessors schreiben.
- Abläufe anpassen .

Phase V - Klassen weiterentwickeln

In dieser Phase werden grundsätzliche Verarbeitungen isoliert, und das Zusammenspiel der Instanzen weiter verbessert.

- Das neue Megawidget *tcanvas* übernimmt das Zeichnen der Blockbausteinen.
- Die neue Klasse *playfield* übernimmt das Steuern der Blöcke.
- Die Klasse *grid* übernimmt die grundlegenden Block-Operationen .

Die Bindings werden mit dem Ziel neu verteilt, die Steuerung des Spiels selbst von der Steuerung des Prozesses zu trennen.

Das Dispatching der block-Bewegungen wird vereinfacht, indem für jede Bewegung eine eigene Methode geschaffen wird, welche direkt angesprochen wird.

Das Resultat ist das angestrebte umgestaltete System.

Die einzelnen Schritten

- Klassen *grid* und *playfield* implementieren,
- Megawidget *tcanvas* implementieren,
- Ablauf verfeinern.

Diskussion

- Jede Phase lässt nur Aenderungen zu, die mit dem momentanen Wissensstand auch untermauert werden können.
- Der Wissenstand wächst von Phase zu Phase, wobei die Isolierung von zum Teil völlig unabhängigen Aufgaben das Verständnis wesentlich erleichtert.
- Vor allem in Phase IV lassen sich dedizierten Testumgebungen für die einzelnen Klassen anwenden, welche dann systematisch ausgetestet werden können, was die Qualität des gesamten Systems erheblich beeinflusst.
- Der bewusst klein gehaltene Umfang der Änderungen in jeder Phase, vermindert wesentlich die Gefahr von Fehlanalysen, und erlaubt rasches Korrigieren, wenn sie dennoch auftreten.
- Nach jeder Phase war ein voll funktionsfähiges System vorhanden.
- In keiner Phase waren grössere Problemen aufgetreten.
- Jede Phase beanspruchte wenig Zeit.
- Jede Phase liess mehrere Lösungsvarianten für die nächste Phase offen.

Schlusswort

Der phasenweise Ansatz der Umgestaltung und die PERL Konzeption der Packages und Klassen ermöglichen sinnvolle und weiche Transformation von prozeduraler nach objektorientierter Implementation.

Das stufenweise Redesign wird weitgehend mit einfachen Editierungsfunktionen durchgeführt, kann ja sogar semi-automatisiert werden, so zum Beispiel das Qualifizieren von Variablen und Subroutinenaufrufen sowie die Umwandlung der Subroutinenaufrufen in Botschaften.

Selbstverständlich ist eine gute Portion Erfahrung

und Selbstdisziplin notwendig, um sich einerseits nicht in unnötigen akademischen Übungen zu verlieren, und andererseits die Klippen allzu frecher Umgestaltungen sicher zu meistern. Denn, das zeigt die Praxis immer wieder, auch ein traumhaftes Redesign-Konzept kann sich plötzlich und unerwartet in einem Furcht erregenden Alptraum verwandeln!