Eine Zustandsmaschine für Perl/Tk-Applikationen

Das Implementieren von graphischen Oberflächen kann mit dem Einsatz von Zustandsmaschinen wesentlich vereinfacht werden. Das gilt besonders für Perl/Tk, stellt diese Bibliothek kaum solche Steuerungsmechanismen zur Verfügung.
Doch Perl hat alle notwendigen Mitteln, um relativ einfach effektive Zustandsmaschinen zu realisieren. Ferner bietet Perl/Tk mit seinen virtual events eine gute Möglichkeit Zustandsübergänge zu steuern. Wie man das machen kann, lesen Sie im folgenden Artikel.

von Marco Marazzi

Einführung

Wer mit Perl/Tk grössere Applikationen entwickelt, wird früher oder später mit der Aufgabe konfrontiert, den Zustand der vielen Widgets zu steuern. In der Tat Perl/Tk bietet in diesem Gebiet nicht viel.
Während zur Definition von solchen graphischen Applikationen die Zustandsdiagramme und die Sequenzdiagramme breite Anwendung finden, wird für die Implementation der Perl/Tk Applikationen kein einheitliches Modell angewandt. Die meisten veröffentlichten Skripten verwenden zwar composite widgets, setzen aber keine zusätzliche Schicht zwischen Applikation und graphische Präsentation ein. Die Folge davon ist, dass der Applikationscode mit einer Unzahl von Tk – Botschaften durchsetzt ist, welche der Übersichtlichkeit nicht unbedingt förderlich sind.
Wenn bei einfachen Applikationen dieses Modell sicher optimal ist, kann es bei komplexeren Applikationen bösen Folgen für die Wartbarkeit und Qualität haben. Bei wirklich grossen Systemen führt es mit grosser Wahrscheinlichkeit zur Katastrophe.

Im Folgendem wollen wir drei Modelle verschiedener Komplexität skizzieren, welche helfen sollen, solche unangenehme Situationen zu meiden, in dem man eine zusätzliche Schicht zwischen Applikation und Präsentation entwickelt, welche die Steuerung der graphischen Elemente übernimmt.
Diese Schicht basiert einerseits auf einer Zustandsmaschine, und andererseits auf einem Perl/Tk spezifischen Kommunikationsmechanismus. Die Zustandsmaschine sorgt für die Verarbeitung des Zustandsbaum, während das Kommunikationsmechanismus die Umsetzung und Weiterleitung der Ereignissen an die Widgets übernimmt.

Zuerst aber ein Paar Worte über die Ausgangslage.

Die Ausgangslage

In unseren Betrachtungen gehen wir von einer Applikationstruktur aus, welche aus dem main package und aus mehreren selbständige Dialoge (sogenannte Toplevel) besteht.
Der main package hat einen Menu und den main loop, die Dialoge nehmen Eingaben an, zeigen Daten an, und lösen Funktionen oder Prozesse aus.
Weiter gehen wir davon aus, dass die Applikation sicherstellen muss, dass zu jedem Zeitpunkt tatsächlich nur diejenigen Widgets aktiv sind, welche zur Ausführung der zu dem Zeitpunkt erlaubten Funktionen auch notwendig sind.
Weiter setzen wir auch voraus, dass unsere Applikation aus einer Folge von eindeutig definierbaren Zuständen ist, welche in einem Zustandsbaum festgehalten wird.
Wir wissen auch, dass bestehende Widgets können und dürfen vom Applikationscode aus erkannt, und laufend modifiziert werden , was uns erlaubt Zustandsübergangsroutinen zu definieren, welche dann zum richtigen Zeitpunkt ausgelöst werden.

Haben wir also den Zustandsbaum in allen seinen Bestandteilen definiert, so sind wir in der Lage die passende Zustandsmaschine zu implementieren. Kennen wir alle Zustandsbäume, die wir unterstützen müssen, so können wir versuchen eine Klasse vom Typ ‚Zustandsmaschine' zu implementieren.

Als Nächstes befassen wir uns mit den Implementationsmodellen der Zustandsmaschine.

Implementationsmodelle der Zustandsmaschine

Als Modell wollen wir ein bestimmter Lösungsansatz für die Zustandsmaschine bezeichnen. Zudem ein Modell darf auch eine limitierte Zielsetzung, und deshalb eine limitierte Funktionalität aufweisen. Das ist für die Praxis sehr zweckmässig, denn während einfache Skripte eine ausgebaute Zustandsmaschine nicht unbedingt brauchen, können die Komplexeren beide gut gebrauchen!

- **Modell I** ist der einfachste Ansatz, und dient der Steuerung einzelner Widgets, wie zum Beispiel eine Menuleiste.

- **Model II** ist eine Maschine, welche von einem Prozess die Einhaltung eines bestimmten Zustandsbaums gewährleisten kann.

- **Modell III** ist eine vollständige Zustandsmaschine, welche einen Strom von Ereignissen verarbeiten kann.

Modell I

Wir wenden dieses Modell an, wenn ein bestimmtes Widget auf Grund des Zustandes einiger wenigen Variablen gesteuert werden kann.
In einer Tabelle werden die möglichen Zustände und die entsprechenden Zustandsänderungen festgehalten.
Die Variablen werden zu geeigneten Zeitpunkten analysiert, und deren Zustand in einem Tabellenindex transformiert. Dann wird die mit dem berechneten Index in der Tabelle gespeicherte Zustandsänderung ausgeführt.
Diese simple Maschine ist für die Steuerung einzelner Widgets, wie zum Beispiel eine Menuleiste, eine Statuszeile oder eine Gruppe Buttons sehr geeignet.

Siehe dazu Beispiel I **Listing_1.txt**

Dieses Modell führt nur Zustandsänderungen aus, soweit sie bekannt sind, hat aber nur eine sehr beschränkte Kontrollfunktion.

Modell II

Wir wenden dieses Modell an, wenn die Applikation einen komplexeren Zustandsbaum hat, und wir wollen sicherstellen, dass es eingehalten wird. Die Kontrolle der Ausführung bleibt aber beim Applikationscode und die Zustandsmaschine hat eher eine passive Rolle, nämlich die der Überwachung und Auslösung von Zustandsänderungen. Zu diesem Modell gehört nur eine einfache Tabelle der Zuständen und ihren Zustandsänderungen, welche nichts anderes sind als eine Folge von Botschaften zu den relevanten Widgets.
Im Gegensatz zu Modell I wird der Programmzustand nicht errechnet, sondern ergibt sich aus den ausgelösten Ereignissen. Das heisst, die Zustandsmaschine reagiert auf einen Ereignis, in dem sie das Ereignis in den Zustandsbaum für den aktuellen Zustand sucht, und falls gefunden auch ausführt.

Siehe dazu Beispiel II **Listing_2.txt**

Modell III

In diesem Modell spielt die Zustandsmaschine eine aktive Rolle, in dem Sie den im Zustandsbaum definierten Prozess ausgehend von einem Start-Zustand beginnend steuert. In diesem Fall nicht nur der Zustand der Widgets, sondern auch der Ablauf des Prozesses selbst wird kontrolliert. Das bedeutet , die Prozesse sind eine Folge von Aktionen, welche durch die Zustandsmaschine auf Grund der erhaltenen Ereignisse der Reihe nach ausgelöst werden. Werden von der Aktionen keine Ereignisse mehr erzeugt, so terminiert die Zustandsmaschine, und mit ihr auch der Prozess.

Siehe dazu Beispiel III **Listing_3.txt**

Die Funktionen der Zustandsmaschine.

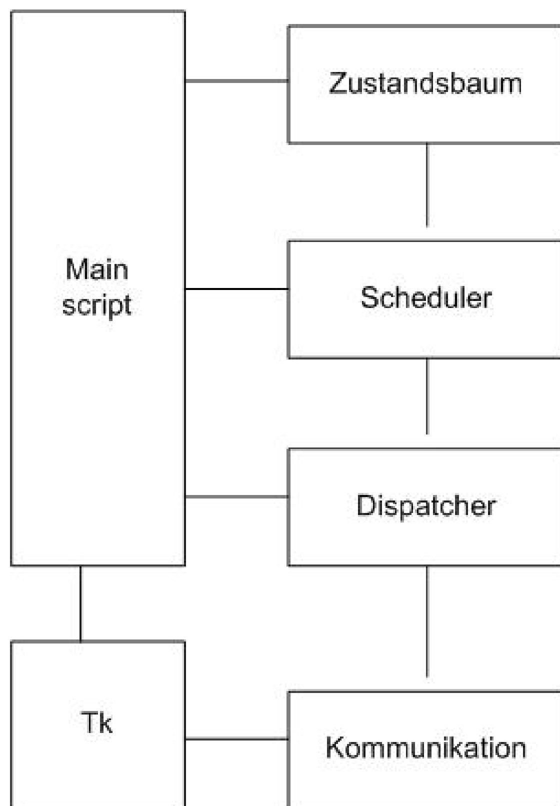
Wir wollen nun versuchen eine Zustandsmaschine zu entwerfen. Dabei beschränken wir uns auf den Typ Modell III, auch weil die anderen nicht viel Anderes als vereinfachte Versionen dieses Modells sind.

Die Maschine soll aus folgenden logischen Funktionen bestehen

- ‚Zustandsbaum',

- ‚Scheduler',
- ‚Dispatcher' und
- ‚Kommunikation'.

Die Funktion ‚Zustandsbaum' soll die Definition des Zustandsdiagramms realisieren, das heisst die Speicherung des Baums und die Zugriffe zu deren Elementen.
Die Funktion ‚Scheduler' soll die Ereignisse entgegennehmen, und für die Ausführung ordnen und bereithalten.
Die Funktion ‚Dispatcher' soll die Zustandsänderungen und die Aktionen auslösen.
Die Funktion ‚Kommunikation' soll die Verbindung zu den Widgets besorgen, das heisst im Wesentlichen die Ereignissen in Botschaften zu den widgets umwandeln.



Die Funktion ‚Zustandsbaum'

Ein Zustandsbaum besteht aus Zuständen, Ereignissen, Zustandsänderungen und Aktionen.

Die Speicherung eines solchen Baums kann bestens mit einer Tabelle des Typs HASH realisiert werden. Die Elemente werden durch die Zustandsidentifikationen indexiert.

Jeder Zustand ist seinerseits auch ein HASH mit zwei Elementen :

- die Zustandsänderung und
- die Liste der erwarteten Ereignissen.

Jedes Ereignis ist wiederum ein HASH mit den Elementen

- Dispatch (Aktionen)
- NextState (Identifikation des nächsten Zustand nach der Aktion)
- NextEvent (Ereignis welcher am Ende der Aktion automatisch ausgesetzt wird (Substates)
- Mode (modal oder nicht modal)

Die Funktion ‚Scheduler'

Diese Funktion empfängt die Ereignisse und ordnet sie für die Ausführung.
Normalerweise ist ein FIFO Stapel völlig ausreichend, auch wenn priorisierte Warteschlagen manchmal angebracht sind.
Der Scheduler ist notwendig, denn das Auslösen der Ereignisse und deren Aktionen nicht synchron sind. Ereignisse werden meistens am Ende einer Aktion ausgesetzt um das Resultat der Verarbeitung zu markieren. Dann muss zuerst die Aktion terminiert werden. Erst dann kann das nächste Ereignis verarbeitet werden.
Dennoch ist der Scheduler nicht immer notwendig, denn es gibt Prozesse welche das synchrone Abarbeiten der Ereignisse verlangen.

Die Funktion ‚Dispatcher'

Der Dispatcher führt den angegebenen Ereignis aus. Dabei muss er zuerst die Zustandsänderung und die Aktion auslösen, das Resultat der Aktion prüfen und die Fortsetzung der Arbeit bestimmen: entweder abbrechen oder den nächsten Zustand einleiten.

Wenn die Ausführung der Zustandsänderung meistens eine Folge von Botschaften zu den Widgets ist, kann das Auslösen der Aktion mehrere Formen annehmen:

- das Aufrufen eines anonymous Code-blocks,
- das Aufsetzen eines Unterzustands (substate),
- das simple auslösen eines weiteren Ereignisses.

Diese Komponente wird entweder von der Zustandsmaschine für jeden Element des Ereignisstapels aufgerufen, oder von der Aktion selbst, wenn das angegebene Ereignis unmittelbar und synchron prozessiert werden soll.

```
      Define Zustandsbaum
      Set start-Ereignis and run
      While (Ereignis im
Ereignisstapel) do begin
            Locate Zustand,
Ereignis
            Ausführe
Zustandsänderung
            Ausführe Aktion    /*
setzt mindestens ein Ereignis aus
*/
            Setzte nextState
            end
```

Die Funktion ‚Kommunikation'

Diese Funktion soll den Ereignisstrom zwischen Zustandsmaschine und Tk – Widgets steuern:
einerseits Ereignisse der Zustandsmaschine zu den Widgets führen, und andererseits Ereignisse aus den Tk-Dialogen der Zustandsmaschine zuführen.

Beide Funktionen setzen also einen Mechanismus voraus, welches zuerst einmal die Widgets kennt, und mit ihnen kommunizieren kann. Weiter muss er fähig sein, die Ereignissen des Applikationscodes entgegenzunehmen, und in Botschaften zu den Widgets umzusetzen. Und schlussendlich Tk-Ereignisse in Ereignisse der Zustandsmaschine umwandeln.

- Ereignisse der Zustandsmaschine zu den Widgets führen.

  Das kann unterschiedlich implementiert werden:

  - Der Kommunikationsmechanismus wird in dem Zustandsbaum eingebettet, in dem die **Zustandsänderungen selbst** die Widgets mit Botschaften beliefern.
  - Es wird eine **Umsetzungstabelle** eingerichtet, welche die Umsetzung der Ereignissen des Zustandsbaums in Botschaften zu den Widgets definiert. Der Dispatcher wird so ausgebaut, dass die Tabelle bei jeder Zustandsänderung berücksichtigt wird.

  Botschaften zu den Widget können im einfacheren Fall explizit, in komplexeren Fällen hingegen indirekt mittels virtual events ausgelöst werden.

- Tk-Ereignisse in Ereignisse der Zustandsmaschine umwandeln.

  Das kann ebenfalls verschieden gelöst werden:

  - die callbacks der relevanten Ereignissen , wie z.B. Buttons, rufen den Dispatcher oder senden ein Ereignis zum Scheduler der Zustandsmaschine.
  - Die Widgets bilden einen Mega-Widget welches von der Zustandsmaschine abgeleitet wird. Bei der Realisierung wird die Instanz der Zustandsmaschine angegeben. So kann der Mega-Widget mit der Zustandsmaschine kommunizieren und ihr in der gerade beschriebenen Art Ereignisse übergeben.
  - Die Zustandsmaschine richtet für die main window virtual events, welche von den Widgets beim Auftreten von relevanten Tk-Ereignisse, erzeugt werden. Die callbacks dieser virtual events sind nichts anderes als Aufrufe des Scheduler oder Dispatcher der Zustandsmaschine.

Das wichtigste Merkmal dieses Konzeptes ist , dass für die Applikation das Ereignis an keine Resource des Präsentationslayers gebunden ist. Sogar gegenüber der Zustandsmaschine selbst ist seine Bindung sehr schwach, denn besteht ein Ereignis aus einer einfachen Identifikation des Typs scalar. Auch die Abhängigkeit der Zustandsmaschine gegenüber dem Präsentationslayer ist schwach, denn alles was sie wissen muss, ist die Liste der Widgets, besteht doch der virtual event praktisch aus

seiner Identifikation. Somit ist die Definition der Zustandsänderungen voll und ganz Bestandteil des Präsentationslayers und liegt nur in seinem Gültigkeits- und Verantwortungsbereich.

Einige Anmerkungen

- Der Event loop : die Zustandsmaschine setzt kein eigenes event loop auf, vielmehr sie nutzt entweder denjenigen von Perl/Tk oder sie verarbeitet kontinuerlich die Ereignissen, welche von den Aktionen dem Scheduler übergeben werden.

- Ereignisse sind für den Applikationscode einfache Identifikationen welche durch einen Scalar identifizert werden. Sie brauchen nur im Gültigkeitsbereich des entsprechenden Zustandsbaums eindeutig zu sein.

- Die drei Modelle können im gleichen Prozess gut nebeneinander leben, wobei jeder eine Spezifische Aufgabe übernimmt.

- Modell III ist aus dem Bedürfnis und Hoffnung entstanden, einen Prozess vollständig mit der Definition seines Zustandsbaumes steuern zu können. Das ist in vielen Fällen auch gelungen, aber es blieben gewichtige Ausnahmen. So mussten ganz pragmatisch Kompromisse akzeptiert werden, welche den Dispatcher mit Methoden ergänzten, die es den Aktionen erlaubten direkt den Ablauf des Prozesses zu steuern, was heisst, die Definitionen des Zustandsbaums zu überspielen.

Die Zustandsmaschine in der Applikation einsetzen

Der Einsatz der Zustandsmaschine in der Perl/Tk Applikation sieht im einfachsten Fall folgenden Rahmen vor:

- Die Mainline setzt den Menu , die globale Variablen, uses und requires modules, initialisiert den Zustandsbaum der Applikation selbst, initialisiert den Kommunikationsmechanismus zu den Widgets, instanziiert die Zustandsmaschine und startet den Perl/Tk mainloop.

- Die Funktionen des Menus rufen jeweils eine subroutine des main packages, welche Folgendes ausführen muss :

  - den Zustandsbaum des Prozesses definieren,
  - das composite Widget instanziieren, das den Prozess visualisiert,
  - den Kommunikationsmechanismus zu den Widgets initialisieren,

  - die Zustandsmaschine instanziieren,
  - die Ausführung der Zustandsmaschine auslösen,

  - die Aktionen definieren.

- Die Zustandsmaschine selbst wird durch die Klasse easyDispatcher und easyEvent implementiert. Die Klasse eventDispatcher erweitert easyDispatcher mit dem Kommunikationsmechanismus.

Siehe dazu Beispiel IV **Listing_4_1.txt**

Dialoge implementieren

Abhängig von der Komplexität der zu realisierenden Funktionalität der einzelnen Dialoge werden GUI-Dialoge entweder in subroutinen, in required modules , oder in composite Widgets, auch Mega Widgets genannt, implementiert.

Gleichwohl welche der soeben erwähnten Implementierungsform gewählt wird, folgende Funktionalitäten müssen für die Instanzierung des Dialogs vorgesehen werden:

- die Widgets selbst instanziieren,
- die Zustandsänderungen mit bind-Botschaften den virtual event zuordnen,
- die Referenz zum Toplevel der Zustandsmaschine bekannt machen.

Wird ein Dialog mit einem Mega-Widget implementiert, so kann es vorteilhaft sein, die Funktionalität der Zustandsmaschine zu vererben.

Siehe dazu Beispiel V **Listing_4_2.txt**

Spezialisierte Zustandsmaschinen bilden

Spezialisierte Zustandsmaschinen werden gebildet, wenn applikatorische Eigenschaften in die Funktionalität der Zustandsmaschine vorteilhaft eingebettet werden können. Das ist dann der Fall, wenn besonders strenge Sicherheitsauflagen eingehalten werden sollen, wenn der Aufruf von externen Aktionen besondere Vorbereitungen im Dispatcher nötig machen, oder wenn der Ereignisstrom inkompatibel ist mit der Funktion ‚Scheduler'.

Substates

Es ist möglich, und ja erwünscht, innerhalb eines Zustandsbaum weitere Zustandsbäume , sogenannte Substates, zu gebrauchen, um komplexere Aktionen zu steuern. Die Klasse easyDispatcher lässt dies zu, Einzelheiten dazu sind in der POD section der Klasse selbst zu lesen.
Dennoch ist es manchmal sinnvoll, dies in der Aktion selbst zu kodieren. In diesem Fall muss neben der Definition und Instanziierung des Zustandsbaums auch noch den Scheduler mit einer Botschaft run(startEvent) oder den Dispatcher mit dem ersten Ereignis lanciert werden.

```
sub action {
 my $self = shift;
 my $dTable = {};## definiere den Baum
 my $dispatcher = easyDispatcher->new(
       -dTable => $dTable,
        -start => 'start',
        -debug => $debug);
 $dispatcher->scheduleEvent('convert');
 $dispatcher->run();
 return 1
}
```

Aktionen evaluieren

Prozesse welche Ausnahmen auslösen können, sollten evaluiert werden, damit die Ausnahme nicht zu einem Abbruch der Zustandsmaschine führt.

```
sub action {
 my $self = shift;
 require extScript;
 eval 'Interface2extScript(@_)';
 if ($@) {
       $self->ScheduleEvent(,ERROR');
 } else
       ## weiter machen
 }
 return 1
}
```

Externe Prozesse auslösen

Während solcher Prozesse ist es vorteilhaft die Dateneingabe aller Widgets zu sperren, und wenn möglich den Fortschritt anzuzeigen. Dies kann am besten mit einem normierten Zustand ‚SERIAL', der zwei Ereignisse ‚SPERREN' und ‚ENTSPERREN' vorsieht, und wenigen Botschaften implementiert werden, wie folgende Abbildung zeigt.

```
sub action {
 my $self = shift;
 my @output;
 local *LS;
 my $state = $self->state();
 $self->changeStateAndDispatchEvent
        (,',SPERREN');
 open (LS,"ls -al *.pl|";
 @out = <LS>;
 close LS;
 $self->dispatchEvent(,ENTSPERREN');
 $self->changeState ($state);
 ## do something with @out …
 return 1
}
```

Bemerkenswert ist es, dass das Sperren und Entsperren keine Zustandsänderungen, sondern Aktionen sein müssen, denn die Zustandsmaschine führt Zustandsänderungen nur beim Betreten und nicht beim Verlassen eines Zustandes aus.

Zeitabhängige Applikationen

Zeitabhängige Applikationen, wie zum Beispiel Simulationen oder Spiele, können mit easyDispatcher implementiert werden, wobei jeweils eine Zustandsmachineninstanz einem Zeitgeber zugeteilt wird, welche deren Ereignissen entgegennimmt. Die Aktionen ihrerseits können auch weitere Zustandsmaschinen instanziieren und ablaufen lassen.
In diesem Falle Modell I oder II ist angebracht, denn der Ablauf selbst eine zeitliche Folge von Aktionen ist, welche von den Zeitgebern und der externen Ereignissen gegeben ist.

Implementationsdetails

Die POD Sections der Klassen enthalten detaillierte Angaben über der Parametrisierung und der Anwendung der Klassen, welche die Zustandsmaschine bilden. Dennoch wollen wir ein Paar Einzelheiten etwas näher anschauen.

Die Klasse **easyEvent** modelliert das Ereignis der Zustandsmaschine.

Die Klasse **easyDispatcher** modelliert die Zustandsmaschine beschrieben in Modell III, jedoch ohne Funktion ‚Kommunikation'. Dies aus praktischen Gründen, denn sie wird auch für Aktionen verwendet, welche keine Verbindung zu graphischen Elementen aufweisen.

Die Klasse **eventDispatcher** erweitert easyDispatcher mit der Funktion ‚Kommunikation',
die Ereignisse in einer Sequenz virtual event zu den widget umzuwandeln.

Diskussion

Einschränkungen

- Die externe Definition der Tabellen ist aufwendig und nicht sicher.
- Der Zustandsbaum wird nicht validiert.
- Die Fehlerbehandlung ist mangelhaft, da Aktionen nicht evaluiert, sondern aufgerufen werden.
- Die vorliegende Implementation unterstützt keine echte asynchrone Prozesse, welche mit dem Systemaufruf ‚fork' gestartet werden.
- Die Kommunikation kann nicht überprüft werden, d.h. die Zustandsmaschine weiss nicht, ob tatsächlich seine Ereignisse auch von den widgets abgenommen werden.

Refaktoring

Kann man einen Perl/Tk nachträglich mit einer Zustandsmaschine versehen?
Wenn man eher an die Anwendung einer der drei erwähnten Modellen denkt, als an den Einsatz einer von easyDispatcher spezialisierten Klasse, dann kann die Frage mit einem Ja beantwortet werden. Modell I lässt sich gut einbauen, denn es tangiert die applikatorische Logik kaum oder unbedeutend. Auch Modell II kann für eine Refaktorisierung in Betracht gezogen werden, während Modell III eindeutig nur im Rahmen eines Redesign-Prozesses zu verantworten ist.

Ausbaumöglichkeiten

- Klasse easyDispatcher mit Methoden zum Aufbau des Zustandsbaum versehen, damit die Tabelle in der Instanz gesichert wird und auch validiert wird.
- Zustandsänderung auch beim Verlassen des Zustandes auslösen.
- Applikationen, Aktionen und Prozesse als Klassen modellieren.
- Metasprache entwickeln : die Definition der Zustandsänderung ist sehr aufwendig. Eine Metasprache würde hier Abhilfe schaffen.
- Dispatcher mit beliebigem Ereignis starten.

Schlusswort

Ich gebe es zu, auf dem ersten Blick sieht der Aufwand beträchtlich gross, um in den meisten Perl/Tk Applikationen angewandt zu werden.
Trotz den erwähnten Einschränkungen sind die Erfahrungen mit dem geschilderten Modellen doch sehr positiv, vor allem weil der forcierte Formalismus zu tiefgehenderes Auseinadersetzen mit den zu lösenden Problemen verlangt.
Interpretative Sprachen wie perl/Tk verführen den Programmierer, auch scripts grösserer Komplexität inkrementell zu implementieren. Man beginnt bei einer einfacheren Version, und erweitert sie je nach Erfahrungen allmählich weiter. Auch die Anforderungen in Bezug auf Qualität und Sicherheit ändern sich . Wächst der Anwenderkreis, dann steigen die Anforderungen meistens entsprechend. Ein teueres Redesign ist die logische Folge. In jedem Falle ist es angebracht, Systeme zu verwenden, welche zu einer systematischen Implementationsweise führen.
Zustandsmaschinen im Allgemeinen, und easyDispatcher im Einzelnen gehören dazu.

Literaturangaben

- [1] Das UML-Benutzerhandbuch, 2006, Addison-Wesley
- [2] The Unified modeling language Reference manual, second edition, 2005, Addison Wesley Chapter 7 State machine view.
- Practical Statecharts in C/C++, Miro Samek 2002, Cmp Books
- [3] AI for games developers, 2004, O'Reilly Chapter 9 Finite state machines.
- [4] Mastering Perl/Tk Steve Lidie and Nancy Walsh.
- [5] The Algorithm design manual Chapter 8.7.7 Finite state machine minimization.

- [6] Introduction to algorithm, second edition
  Chapter 32.3 String matching with finite automata.

- [7] Mathematik für Informatiker, Willibald Dörfler
  Band 1 Finite Methoden und Algebra
  Kapitel 13.10 Petri-Netze und markierte Graphen.

Nützliche Links

- State Pattern at Perl Design Patterns Wiki :
  http://perldesignpatterns.com/?PerlDesignPatterns

- Finite automata in Perl :
  http://sedition.com/perl/automata.html

- DFASimple - A PERL module to implement simple Discrete Finite Automata :
  http://search.cpan.org/~randym/DFA-Simple-0.32/Simple.pm

- Perl_com Building a Finite State Machine Using DFASimple :
  http://www.perl.com/pub/a/2004/09/23/fsms.html