Drag and Drop mit Perl/Tk

'Drag and Drop' wird in Perl/Tk durch eine Kollaboration von Objekten der Klassen DragDrop and DropSite. Instanzen dieser Klassen können ziemlich frei kombiniert werden, um Daten zwischen praktisch allen Widgets eines Prozesses auszutauschen .
Im Folgenden wird es gezeigt wie man das realisiert.

von Marco Marazzi

Einführung

Obwohl ‚Drag and Drop' (im weiteren Text DnD genannt) eine sehr wichtige Funktionalität von GUI – Systemen ist, wird es in der Perl/Tk Welt etwas stiefmütterlich behandelt. Es gibt zwar einige Artikel darüber, doch es scheint demgegenüber eine gewisse Abneigung zu herrschen. Vielleicht darum, weil einerseits DnD eine ziemlich komplexe Sache ist, und anderseits weil die existierenden Klassen wenig bis überhaupt nicht dokumentiert sind. Demzufolge erscheint der Einstiegsaufwand vielen Programmierern schlicht zu hoch. Dass dem nicht so sein muss, wollen wir im Folgenden zeigen.

DnD kann in folgenden Teilen unterteilt werden

- das DnD - System in seinen statischen Bestandteilen aufsetzen,
- eine Reihe von DnD - Sessionen bearbeiten:
  - die DnD - Session eröffnen (Drag Prozess, Dragged Object festlegen),
  - die DnD - Session visualisieren (Dragging Prozess),
  - die DnD - Session abschliessen (Drop Prozess, Dragged Object verarbeiten),
- das DnD - System beenden.

Eine Session wird definiert als der Vorgang ein bestimmtes Objekt (Dragged Object) von einer Quelle zu einer Senke zu transportieren. Sie schliesst sowohl sämtliche Visualisierungsaufgaben als auch die notwendigen Datenzugriffen ein.
Eine Session beginnt also in dem Augenblick, wenn ein Objekt von der Quelle erschlossen wird, und endet in dem Moment wenn entweder das gleiche Objekt bei einer Senke abgelegt wird, oder unterwegs fallengelassen wird.
Es ist eine Frage der Konzeption ob das Verarbeiten des Objekten bei der Senke auch als Bestandteil der Session betrachtet wird. Diese Frage darf nicht leichtfertig behandelt werde, denn sie birgt eine Folge von Problemen vor allem in Bezug auf die Anforderungen and das DnD – Systems selbst. Als Beispiel sei die simple Aufgabe des Kopierens von Dateien: soll das Kopieren synchron während der DnD - Session erfolgen, oder soll nur der notwendige Command so gestartet, dass es dann asynchron ablaufen kann?
Die Session ist eine Beziehung zwischen einer Quelle und einer oder mehreren Senken. Diese Beziehung muss nicht statisch sein, denn es ist nicht notwendig, dass die Senke bereits vor dem Beginn der Session bekannt ist. Man kann, und manchmal ist es ratsam, erst bei Beginn der Session die möglichen Senken festlegen. Welche dann zum Zuge kommt, wird erst beim darüber fahren mit der Maus (Dragging) erkannt, und beim anschliessenden ‚Fallenlassen' (Drop) ausgewählt.

DnD lebt nur innerhalb eines GUI - Systems und hat primär nur mit Widgets zu tun. Das heisst, alles was passiert, ist unter der Kontrolle des GUI - Systems und der

Zugriff auf Daten erfolgt entweder durch Methoden der als Quelle oder Senke definierten Widgets oder durch importierten Callbacks, welche dem Datenmodell gehören.

In einer Session sehen wir also zwei voneinander getrennten Ebenen :
- die Visualisierung des DND-Vorgangs und
- die notwendigen Datenzugriffe und Manipulationen um den Datentransport zu bewerkstelligen.

Einerseits hat DnD als GUI - Funktion eigentlich mit dem Datenmodell nicht viel zu tun, und man ist auch gut beraten die zwei Ebenen möglichst strikte zu trennen . Andererseits verlangt DnD, dass die Daten eindeutig identifiziert und mittels definierbaren Methoden verfügbar sind. Das klingt sehr trivial, ist es aber in der Praxis ganz und gar nicht, denn die Daten werden zu verschiedenen Zeitpunkten, und während einer gewissen Zeit in Anspruch genommen.
Das Ganze wird noch komplizierter, wenn man bedenkt, dass nicht selten in GUI - Systemen, und PerlTk gehört wohl zu diesen, zwei Datenmengen gleichzeitig behandelt werden müssen, nämlich die ursprünglichen Daten, zum Beispiel eine HASH-Struktur, und ihrer graphischen Abbildung, zum Beispiel ein HList-Widget.

Fasst man also ein DnD - System nicht bloss als eine GUI - Funktionalität, sondern als eine Funktionalität, um real existierende Daten eines Modells zu manipulieren auf, so sieht man schnell, dass eine Menge DnD - Systeme existieren, welche hauptsächlich von der Beschaffung des darunter liegenden Datenmodells und seiner grafischen Darstellung abhängig sind.
Es gibt also zum Beispiel ein DnD – System, um Daten von einer Liste in einer anderen Liste zu transportieren, und ein Anderes, um Daten aus einem Baum in eine Liste zu extrahieren. Man kann sich sogar auch DnD - Systeme mit gleichem Datenmodell aber mit verschiedener Qualität vorstellen, zum Beispiel solche, welche mehr oder weniger prüfen, ob die behandelte Daten korrekt sind, oder weitere welche mehr oder weniger ausgereiftere grafische Effekte anwenden.

Diese Überlegungen führen dazu, dass es zweckmässig ist DnD - Systeme systematisch zu untersuchen, um einerseits allgemeine Entwurfsmodelle und anderseits die verschiedenen DnD-Typen in Klassen zu definieren, welche zusammen DnD als einbaubare Komponente realisieren. Dies kann innerhalb eines grösseren Projekts durchaus von Vorteil sein.

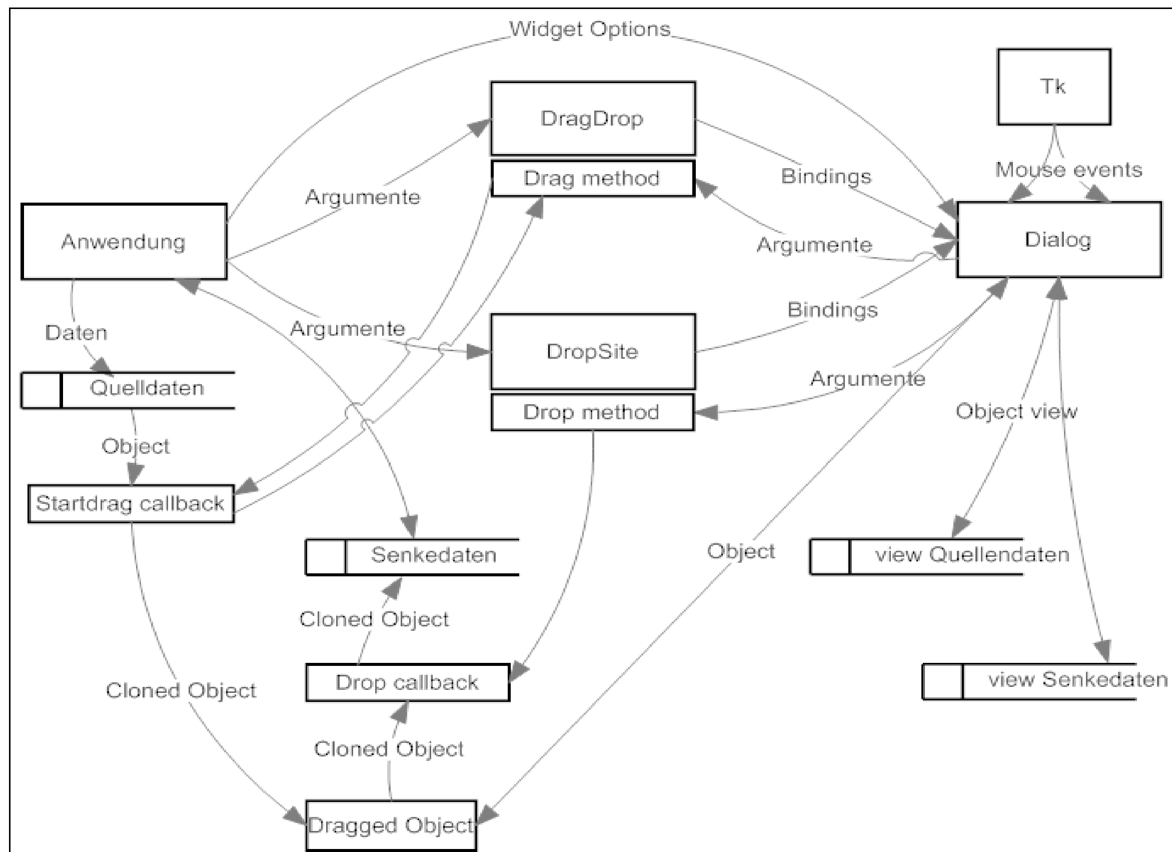

Abbildung 1 Datenflussplan des DnD in Perl/Tk

Mit Perl/Tk kann man ein solches Vorhaben mit vertretbaren Aufwand recht gut realisieren. Im folgenden Text stellen wir stellvertretend für alle auf der CD mitgelieferten Beispielskripte zwei verschiedene Ansätze vor, um die DnD - Funktionalität in Perl/Tk - Applikationen einzubauen. Die Erste wendet die DnD-Klassen unmittelbar in der Datenebene, die Zweite kapselt DnD mit Hilfe von composite Widgets in der VisualisierungsEbene.

DnD mit den Klassen DragDrop und Dropsite

Die Klasse DragDrop modelliert die Quelle, die Klasse DropSite modelliert die Senke eines DnD - Systems.

Beide Klassen werden bei Perl von activeState mitgeliefert, müssen sonst von CPAN installiert werden.

Diese Klassen übernehmen die grafischen Aufgaben des DND - Systems und rufen die Callbacks des Datenmodells, um den Dragged Object zu behandeln.
Diese Callbacks werden als Argumente bei der Instanzierung der Quelle bzw. der Senke angegeben. Das Dragged Object selbst wird hingegen vom Datenmodell während der Session in einer globalen Variablen gespeichert und verwaltet. Die Klasseninstanzen kümmern sich kaum darum. Diese globale Variable muss die gleiche Lebensdauer wie das entsprechende instanzierte DND - Systems haben.

Eine Instanz vom Typ DragDrop definiert eine spezifische existierende Quelle.
Auf die Senke bezogen gilt das Gleiche für DropSite.

Weiter definieren diese Instanzen nur mögliche sozusagen abstrakte Quellen und Senken. Die real existierenden Quellen und Senken werden erst bei der Ausführung der DnD-Session, und zwar auf Grund des aktiven Widgets auf der Quellenseite und auf Grund der Mausposition auf der Seite der Senke.
Interessanterweise sind die Instanzen sowohl untereinander als auch von den gebrauchten Widgets unabhängig. Das bedeutet, einerseits können sie während der Ausführung beliebig kombiniert werden, und andererseits ihre Lebensdauer ist von den Widgets nicht bestimmt.

Das eröffnet sehr interessante Möglichkeiten, macht die Sache aber nicht einfach, denn die notwendigen Steuerfunktionen werden der Anwendungsebene delegiert, das heisst zum Beispiel, dass jede DropSite fähig sein muss, zu erkennen, ob sie den Dragged Object verarbeiten kann und darf. Es ist in der Tat möglich, eine DnD - Session mit mehreren Senken zu definieren, um dann während des Draggings die Richtige wählen zu lassen.

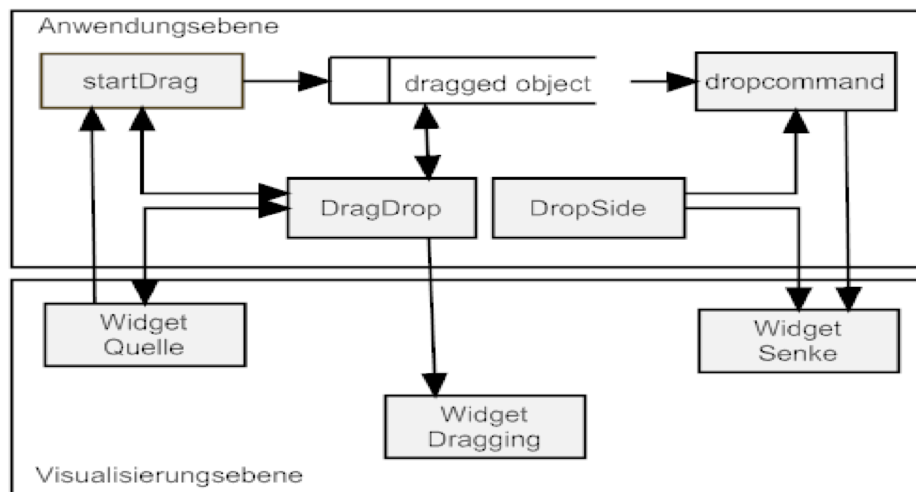

Abbildung 2 DND mit den Klassen DragDrop and DropSite.

Auf Grund ihrer Komplexität kann man sich auch vorstellen, ob es sinnvoll ist, das DnD - Modell mit einer zusätzlichen Klasse zur Modellierung der DnD - Session, welche die Speicherung und Verwaltung des Dragged Object während der Session übernimmt. Analog dazu könnte man sich auch eine weitere Klasse zur Modellierung des Dragged Object selbst vorstellen.

-sitetypes	‚Local’
-startcommand	callback für den Drag – Prozess
-endcommand	Callback
-predropcommand	Callback
-postdropcommand	Callback
-delta	Bewegungschrittweite für das dragging Prozess
-cursor	Cursor shape während des Draggings
-event	Auslösender Ereignis der Session, normalerweise ‚<B1-Motion>’

Optionen für die Klasse DragDrop

-droptypes	‚Local' oder Bezeichnung des externen Systems (win32,KDE,SUN)
-dropcommand	Callback für den Drop - Prozess
-entercommand	Callback
-motioncommand	Callback

Optionen für die Klasse DropSite

Im folgenden Beispiel wird ein Dialog mit Labels erstellt, welche zwei Dateien mittels DnD zu verschiedenen Ausführungen gewählt werden können. Es ist ein klassisches Beispiel für eine M:N Beziehung zwischen Quelle und Senke, wobei alle mögliche Objekte vom gleichen Typ sind und alle Senken jederzeit zugänglich sind. Somit können alle Instanzen zum Zeitpunkt der Dialoginstanzierung angelegt werden. Weiter alle Instanzen brauchen die gleiche globale variable $bag um das aktuelle Dragged object zu speichern und zuzugreifen. Diese globale Variable ist das eigentliche Bindeglied zwischen Quelle und Senke. Sie muss beim Session start initialisiert und bei Session ende wieder gelöscht werden.

```
use Tk;
use Tk::Frame;
use Tk::Label;
my $bag;

sub simpleDnD {
      my $hwnd = shift;
      my (%args) = @_;
      my $rv;
      my $mw = $hwnd->Toplevel();
      $mw->configure(-title=> (exists $args{-title})? $args{-title}:'');
      $mw->protocol('WM_DELETE_WINDOW',sub{1});
      use Tk::DragDrop;
      use TK::DropSite;
      my $wr_001=$mw->Frame()->pack();
      my $wr_002=$mw->Frame()->pack();
      my $wr_004=$wr_001->Label(…)->pack(…);
      my $wr_007=$wr_002->Label(…)->pack(…);
      my $wr_005=$wr_001->Label(…)->pack(…);
      my $wr_008=$wr_002->Label(…)->pack(…);
      my $wr_010=$wr_002->Label(…)->pack(…);

      main::setupDnD($mw,
            -source,[$wr_004,$wr_005],
            -target , [$wr_007,$wr_008,$wr_010]
            );
      $rv = $mw;
      return $rv;

} ## end of simpleDnD

sub setupDnD {
      my ($tl,%args) = @_;
      my @source = @{$args{-source}};
      my @target = @{$args{-target}};
      map {
            $_->DragDrop(
                        -sitetypes => [qw(Local)],
                        -startcommand => [\&startDrag,$_],
                        -endcommand => [\&endDrag,$_]
```

```
                        );
        } @source;
        map {
                $_->DropSite(
                        -droptypes => [qw(Local)], ## must be defined
                        -dropcommand  => [\&drop,$_],
                        );

        } @target;
}

sub startDrag {
        my ($w) = @_;
        $bag = $w->cget(-text);
        return undef;
}

sub endDrag {
        my ($w) = @_;
        $bag = undef;
}

sub drop {
        my ($w) = @_;
        $w->Busy();
        if (main::executeDropProcess($w, $bag,…)) {
                $pb->destroy();
                print "\n Done '$bag'";
        } else {
                print "\n Cancelled '$bag'";
        }
        $w->Unbusy();
        return undef
        }
```

Wie man sieht, werden die DnD-Instanzen als Kinder der entsprechenden Widget angelegt, was damit begründet wird, dass die DnD seine Bindings für diese Widgets definiert. Das sind aber gerade die einzigen Abhängigkeiten!
Interessant ist es, dass die Perl/Tk Optionen der Widgets und diejenigen der DnD-Objekten völlig unabhängig sind. Das hat nicht nur Vorteile, sondern auch den Nachteil, dass die Bindung Quellen - Widget und entsprechendes Startdrag – und Drop - Callback ziemlich lose ist, was zu unerfreuliche Fehler führen kann.

Das main package sieht dann so aus

```
require "simpleDnD.pl";

my $mw=MainWindow->new(-title=>'Demo simple DnD');
my $dnd = main::simpleDnD($mw, -title , 'Simple DnD');
$dnd->focus() if defined $dnd;
MainLoop;
```

Man merke hierzu, dass die globale variable $bag und die Callbacks dort definiert wurden, wo das Tk - Dialog, also das Referenzobjekt für DnD, definiert wurde.

DnD als (vererbte) Funktionalität

DnD kann aber auch als eine Funktionalität einer bestimmten Komponentearchitektur betrachten, und entsprechend entwerfen. Das heisst, DnD existiert in der Anwendungsebene als selbständige instanzierte Objekten nicht, sondern es existieren Widgets, welche die Fähigkeit haben, als Quelle oder Senke einer DnD Session aufzutreten. Ob diese Funktionalität in einer Klasse gekapselt und vererbt wird, ist es eigentlich unwesentlich. Wichtig an dem Konzept ist, dass keine eigenständige DnD - Konstrukte in der Anwendungsebene existieren.

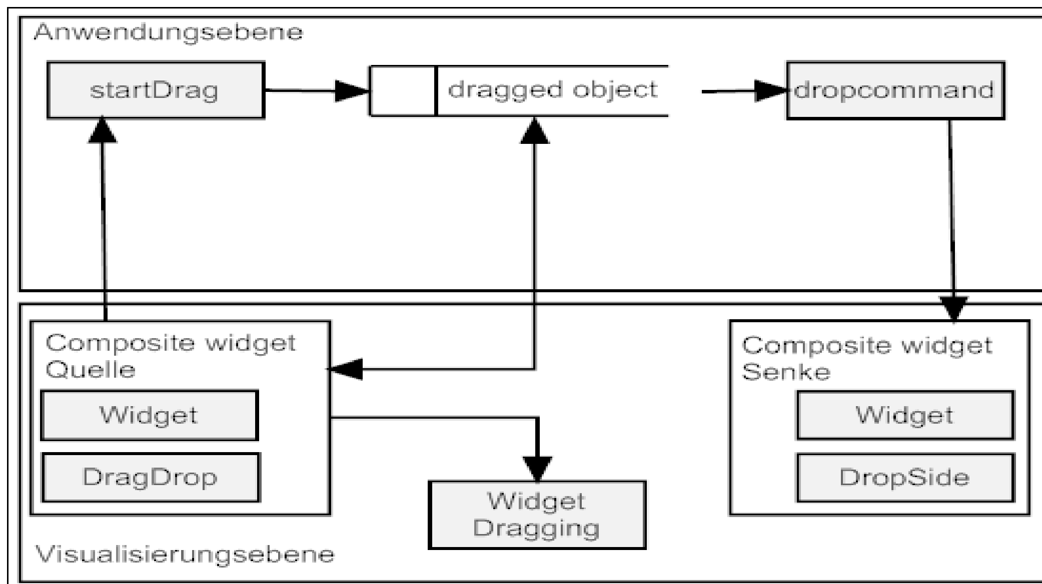

Abbildung 3 DnD mit composite widgets.

Im folgenden Beispiel wird DnD in den Label Widget eingebunden. Das Dialog selbst dann spezifiziert diese erweiterten Labels (composite widgets), und definiert nur noch die Callbacks für die Datenzugriffe. Damit wird die angestrebte schärfere Trennung der Visualisierungs – von der Anwendungsebene ein Schritt näher.
Die Subroutine setupDnd wäre eigentlich überflüssig, wird aber weiter aufgeführt, um die Tatsache zu zeigen, dass DnD als eine beliebige Option behandelt werden kann.

```
use Tk;
use Tk::Label;

package dndLabel;
use vars qw($VERSION);
$VERSION = '1.01';
require Tk::Frame;
require Tk::Derived;
@dndLabel::ISA = qw(Tk::Derived Tk::Frame);
Construct Tk::Widget 'dndLabel';

sub ClassInit {
      my $self = shift;
##
##    init class
##
      $self->SUPER::ClassInit(@_);
}
sub Populate {
```

```
        my ($self,$args) = @_;
        ##    move args to local variables)
        my $start = delete $args->{-startcommand} if exists $args->{-
startcommand};
        my $end = delete $args->{-endcommand} if exists $args->{-endcommand};
        my $drop = delete $args->{-dropcommand} if exists $args->{-
dropcommand};
        my %largs = ();
        foreach (keys %$args) {$largs {$_} = delete $args->{$_}}

        $self->SUPER::Populate($args);
        my $mw = $self;
        my $wr_001 = $mw -> Label (%largs) -> pack();
        my $wr_002= $wr_001->DragDrop(
                                -event => '<B1-Motion>',
                                -sitetypes => [qw(Local)],
                                -handlers => [],
                                -startcommand => [$start,$self],
                                -endcommand => [$end,$self]
                                ) if defined $start;

        my $wr_003 = $wr_001->DropSite(
                                -droptypes => [qw(Local)], ## must be defined
                                -dropcommand  => [$drop,$self],
                                ) if (defined $drop) ;

        ## ctk: public subwidgets
        $self->Advertise('label'=>$wr_001);
        $self->Advertise('drag'=>$wr_002) if defined ($wr_002);
        $self->Advertise('drop'=>$wr_003) if defined ($wr_003);
        return $self;
}
1;      ## make perl compiler happy...
```

```
sub simpleDnDx {
        my $hwnd = shift;
        my (%args) = @_;
        my $rv;
        my $mw = $hwnd->Toplevel();
        $mw->configure(-title=> (exists $args{-title})? $args{-title}:'');
        $mw->protocol('WM_DELETE_WINDOW',sub{1});

        use Tk::DragDrop;
        use TK::DropSite;

        use dndLabel 1.01;

        my ($wr_001 ,$wr_002 ,$wr_004 ,$wr_005 ,$wr_007 ,$wr_008 ,$wr_010) ;

        $wr_001 = $mw->Frame () -> pack();
        $wr_002 = $mw->Frame () -> pack();
        $wr_004 = $wr_001->dndLabel(-startcommand,sub{undef})->pack();
        $wr_007 = $wr_002->dndLabel(-dropcommand,\&main::drop)-> pack();
        $wr_005 = $wr_001->dndLabel (-startcommand,sub{undef})-> pack();
        $wr_008 = $wr_002->dndLabel (-dropcommand,\&main::drop)-> pack();
        $wr_010 = $wr_002->dndLabel ( -dropcommand,\&main::drop)-> pack();

        main::setupDnD($mw,
              -source,[$wr_004,$wr_005],
              -target , [$wr_007,$wr_008,$wr_010]
```

```
            );
      $rv = $mw;
      return $rv;

} ## end of simpleDnD

sub setupDnD {
      my ($tl,%args) = @_;
      my @source = @{$args{-source}};
      my @target = @{$args{-target}};
      map {
            $_->Subwidget('drag')->configure(
                        -startcommand => [\&startDrag,$_],
                        -endcommand => [\&endDrag,$_]
                        );
      } @source;
}
```

Selbstverständlich bleibt das main package praktisch unverändert.

```
require "simpleDnDx.pl";

my $mw=MainWindow->new(-title=>'Demo simple x DnD');
my $dnd = main::simpleDnDx($mw, -title , 'Simple x DnD');
$dnd->focus() if defined $dnd;
MainLoop;
```

Auf der zum Heft beiliegenden CD sind die vollständigen Skripte dieser Beispielen und einige Weitere, welche die geschilderten Eigenschaften von DnD mit Perl/Tk demonstrieren.

Überlegungen

- **Vor- und Nachteile:** Der grosse Vorteil der Klassen DragDrop und DropSite ist ihre Flexibilität in der Bildung der Sessionen. Ihr grosser Nachteil ist, dass sie der Anwendungsebene zu viele Aufgaben delegieren. Doieser Nachteil kann entschärft durch composite widgets und/oder eine Strukturierung der Anwendungsebenen, wie in den Beispielskripten vorgeschlagen wird.

- **Extern DnD:** Perl/Tk unterstützt auch DnD - Sessionen mit anderen Applikationen, aber nur um Daten, im wesentlichen Dateinamen, in Perl/Tk – Anwendungen zu importieren. Zudem kann die DnD – Session nicht mit den Tk - Messages Busy / Unbusy synchronisiert werden. Lokales und externes DnD können problemlos nebeneinander existieren. Das geht so weit, dass sogar ein bestimmtes Widget gleichzeitig als externe und lokale Senke definiert werden kann.
  Siehe dazu auch Skript simpleExtDnD_unit_test.pl auf der CD.

- **Das event-Loop**: Die Ausführung des Drop – Callback kann auch einem Scheduler/Dispatcher übergeben werden. Ein Beispiel dazu wäre das Mainloop selbst, wie folgendes Beispiel zeigt.

```
use Tk ':eventtypes';

   …
```

```
sub drop {
   my ($w) = @_;
   print "\ndrop";
   $w->DoWhenIdle([\&main::_drop,$w]); ## callback will be called once
when TK is idle again
}
```

Siehe dazu auch Skript simpleDnDy_unit_test.pl auf der CD.

- **Das Callback dragcommand:** diese Subroutine wird beim Drag – Prozess und bei jedem Dragging Schritt aufgerufen, solange sie kein UNDEF als Return - value zurückgibt. Diese Return - value gilt als Zeichen, dass das Drag - Prozess erfolgreich war, und somit die DnD – Session gestartet wurde.

- **Dragged Object:** oft ist das Ziel der DnD - Session, eine Kopie des Quellenobjekts bei der Senke zu erstellen. Das bedeutet das Objekt muss vollständig parametrisierbar sein. Das ist nicht immer der Fall, wie zum Beispiel das Canvas - Item vom Typ „Polygon". In diesem Falle muss eine spezialisierte Klasse eingesetzt werden. Siehe dazu auch Skript guiDnD.pl auf der CD.

- **Error handling:** Ausnahmen bei der Ausführung von Drag – oder Drop – Callbacks können die entsprechende DnD – Instanzen vernichten, so dass sie neu instanziert werden müssen. Dafür ist die Anwendungsebene verantwortlich. Das bedeutet, dass Ausnahme – fähige Prozesse in eval - statement gekapselt ausgeführt werden sollten. Erfahrungsgemäss sind Diskrepanzen zwischen Widget, Dragged Object und DnD - Instanzen die wahrscheinlichste Ursache für Ausnahmen während DnD – Sessionen, welche ebenfalls die DnD - Instanzen zum verschwinden bringen.

- **Hausgemachtes DnD:** wer spezialisierte DnD-Klassen als alternative zu den vorhandenen Klassen implementieren möchte kann Rat bei diversen publizierten Anwendungen, wie zum Beispiel das Freeware System „Sprog!". Die verfügbaren Dokumentationen über DnD in Perl/Tk sind in dieser Hinsicht nicht sehr empfehlenswert.

- **Internals:** leider sind die beiden Klassen DragDrop und DropSite schlecht bis überhaupt nicht dokumentiert, und auch noch nicht so einfach zu entschlüsseln. Wer mehr wissen möchte, muss über gute Kenntnisse des Tk - Systems verfügen, und mit ziemlichem Zeitaufwand für systematischen Versuche rechnen.

Schlussfolgerungen

Perl/Tk gibt die Möglichkeit DnD für praktisch alle Widget - Klassen zu implementieren. Dennoch gibt es etliche Klippen zu überwinden, denn nicht alle Widgets sind zweckmässig beschrieben, und erfahrungsgemäss die böse Überraschung lauert immer um die Ecke. Also vor all zu kühnen Vorhaben sei hier gewarnt!
Die Probleme liegen aber eher auf der Seite der Datenebene und der Komplexität des Datenmodells. Auch schon die Datenmenge kann einem unvorsichteigen Entwickler schlaflose Nächte bescheren, denn eine Senke in einer Unmenge Widgets festzulegen, oder das Einfügen eines simplen Blattes in einem genug grossen Baum, kann auch den geduldigsten Benutzer zur Verzweiflung bringen!

Literatur

- [1] 2002, Steve Lidie & Nancy Walsh: Mastering Perl/Tk

Links

- [2] 2001, Steve Lidie: A Drag-and-Drop Primer for Perl/Tk
  http://www.perl.com/pub/a/2001/12/11/perltk.html
- [3] Sprog!
  http://sprog.sourceforge.net/index.html
- [4] 2002, Drag and Drop with Perl/Tk
  http://user.cs.tu-berlin.de/~eserte/src/PerlMonth/dnd/dnd.html