

GUI design mit Perl/Tk

In diesem Beitrag wollen wir einige wesentliche Aspekte der Implementation von GUI in Perl/Tk ausführlich beschreiben. Dabei beziehen wir uns auf die Implementation des GUI eines etwas vereinfachten Dokumenten – Explorers.

Ein erster Aspekt ist die Organisation der Implementation, die wegen der Besonderheiten von Perl/Tk eine spezielle Struktur aufweist. Weitere Aspekte sind die Implementation der eigentlichen GUI – Funktionalitäten. Aus Letzteren werden im Beitrag die Folgenden beschrieben: das Auslagern von Widgets, die dynamische Anpassung der Widget - Dimensionen und die laufende Anpassung der Widget – Zustände

von Marco Marazzi

Wie die Entwicklung von GUI Applikationen organisiert werden kann, wird in [1] bis ins letzte Detail ausgezeichnet dargestellt. Wie die Aufgaben zwischen Anwendung und GUI aufgeteilt werden, und welches Entwurfsmuster angewandt werden kann, wird in [2] ausführlich beschrieben.

Das Datenmodell, welches unser Explorer unterstützen soll, besteht aus einer Menge Dokumente, welche als selbständige Dateien gespeichert sind. Jedes Dokument besteht aus einer beliebigen Anzahl Komponenten, welche eindeutig erkannt und einzeln wohl - formatiert angezeigt werden sollen. Der Benutzer soll mit dem GUI ein Dokument wählen können, die Liste der Komponenten navigieren und die Komponenten einzeln betrachten können. Dabei soll er mehrere davon gleichzeitig anzeigen und betrachten können.

Unser GUI soll also folgende Merkmale aufweisen:

- ein Menu zur Aktivierung der Prozesse für die Bearbeitung des gewählten Dokuments,
- ein Komponenten-Frame bestehend aus einer Selektionsliste der erkannten Komponenten, und einem Frame für die aktuelle Komponente, welcher einerseits die wohl formatierte Anzeige der eigentlichen Komponentendaten und andererseits eine Liste von Aktionsknöpfe aufweist,
- ein Log der ausgeführten Prozesse und
- und eine Statuszeile, welche den aktuellen Stand anzeigt.

Selbstverständlich soll eine zeitgemäße Qualität des GUI hinsichtlich Graphik und Funktionalität gewährleistet werden.

Dabei sollen die drei wesentlichen Bereiche Komponenten-Frame, Selektionsliste, aktuelle Komponente und Log in ihren Dimensionen den momentanen Bedürfnissen anpassbar sein. Zusätzlich sollen die Komponenten in selbständige Dialoge einzeln ausgelagert werden können, damit sie gleichzeitig betrachtet, verglichen und aus ihnen Daten entnommen werden können.

Daraus ergeben sich für die Implementation des GUI folgende Funktionalitäten:

- der Status der Widgets dem aktuellen Zustand anpassen,
- die Dimensionen der Widgets dynamisch anpassen,
- die Widget ein- und auslagern,
- die ausgelagerten Widgets als Singletons definieren.

Ein GUI dieser Komplexität lässt sich nicht leicht in einem einzigen Schritt implementieren, vor allem weil man das Verhalten und die Stabilität aller Funktionalitäten im Voraus nicht genau kennt, und diese also zuerst einmal

ausgetestet werden sollten. Das gilt auch und vor allem für Module aus dem CPAN.

Das bedeutet, man muß die Implementation in einer Folge von Teilimplementationen zerlegen, wobei jeder Teil ein klar definiertes Ziel hat, und auch sicher getestet werden kann.

So geht man immer von einem zwar unvollständigen aber dennoch brauchbaren Zustand zum nächsten bis alle geforderten Ziele erreicht sind.

Der Anfang stellt ein Dialog, welche alle geforderten Bestandteile, aber noch kein dynamisches Verhalten aufweist. Das kann man entweder von Hand zusammenschrauben, oder mit einem GUI - Generator erstellen lassen (siehe dazu [3]).

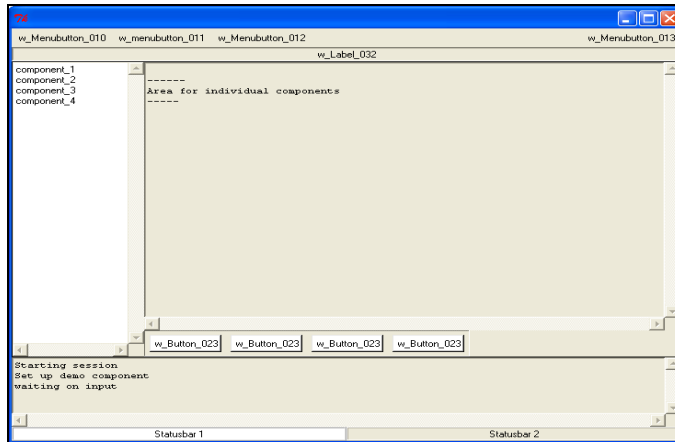


Abbildung 1 Prototyp, mit einem GUI - Generator erstellt.

Dieser Schritt ist insofern wichtig, weil es erlaubt mit wenigen Testdaten den GUI - Entwurf zumindest in seinen statischen Elementen frühzeitig zu überprüfen, und allfällige Korrekturen anzubringen. Das ist in der Praxis ein kaum zu unterschätzende Vorteil. Ist einmal dieser Schritt erfolgreich vollzogen, so kann man sich daran wagen, Leben ins GUI einzuhauchen.

Dazu muss man sich zuerst einmal ein klares Konzept anlegen, das heißt sowohl über die Art, wie man die geforderten Bestandteile implementiert, als auch mit welchen Mitteln und in welcher Reihenfolge.

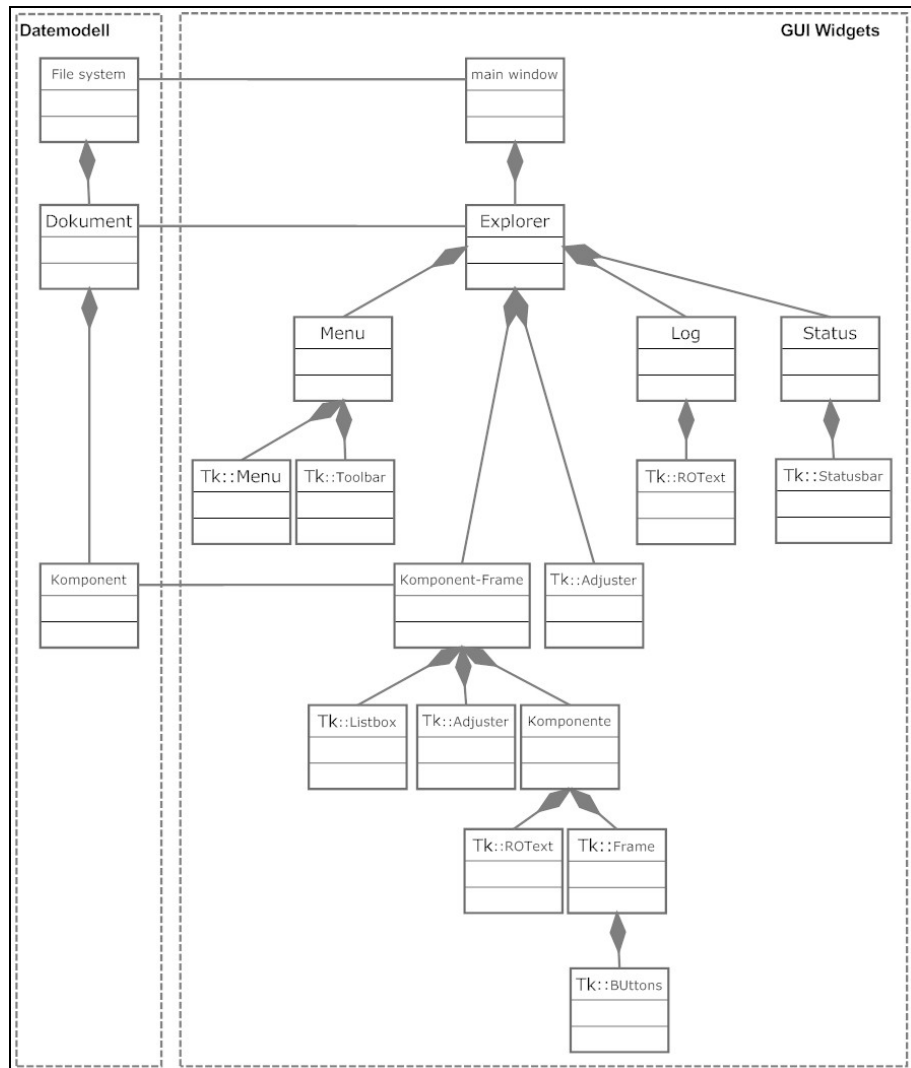


Bild 1: Klassendiagramm des Dialogs Explorer

Mit diesen Kenntnissen kann man sich dann im CPAN umsehen, passende Module auswählen, runterladen, installieren und testen. Damit gewinnt man langsam ein Gefühl für die zu erwartenden Schwierigkeiten, und man kann rechtzeitig Korrekturen anbringen.

Vor allem werden in dieser Phase allfällige in dem vorgesehenen Rahmen unerfüllbare Ziele erkannt, die entweder fallengelassen oder den konkreten Möglichkeiten angepasst werden. In unserem Falle hat man die individualisierte graphische Darstellung der Komponentendaten mit Tk::TreeGraph ganz fallengelassen, und zuerst einmal alle Komponenten als Textbausteine implementiert.

In unserem Falle sind Module in Tabelle 1 relevant.

Modul	Zweck
Tk::Adjuster	Dynamisches Anpassen der Widget-Dimensionen.
Tk::Panedwindow	Erlaubt Frames mit dynamischen Subframes, analog Adjuster.
Tk::Pane	Erlaubt scrollable Frames.
Tk::Multi	Erlaubt die Bildung von Dialoghierarchien, analog Panedwindow.
Tk::Toolbar	Implementiert eine Werkzeugleiste.
Tk::Statusbar	Erlaubt eine vollständige Anzeige des Prozeßstatus.

Tabelle 1 Die für das GUI benötigten Module

Hat man sich nun mal mit allen Bestandteilen familiarisiert, so kann man sich daran machen diese auch mal der Reihe nach zusammenzubauen.

In unserem Falle werden folgende Schritte nacheinander durchgeführt, wobei bei jedem Schritt nur eine bestimmte Funktionalität implementiert wird:

- Die Höhe des Komponenten-Frame und die Breite der aktuellen Komponente werden mit dem Module Tk::Adjuster dynamisch gesteuert,
- das Auslagern von Komponenten-Frame wird implementiert,
- der Status der Widget wird dem momentanen Zustand laufend angepaßt, und
- schließlich werden die Module Tk::Toolbar und Tk::Statusbar anstelle von eigenen einfacheren Konstrukten eingesetzt.

Selbstverständlich ist es manchmal notwendig, nach einem erfolgreichen Schritt den Quellcode zu refaktorisieren, um eine saubere Basis für die nächste Stufe zu legen.

Das ist vor allem wichtig, wenn Composite Widgets gebildet werden. Dieses äußert mächtige Werkzeug von Perl/Tk hat seine Tücken, welche schwer zu verstehende Fehler erzeugen können. Diese wiederum können nach meiner Erfahrung den Zeitaufwand der GUI - Implementierung entscheidend beeinflussen.

Es ist also eine gute Angewohnheit, mit diesem Werkzeug sehr vorsichtig umzugehen, das heißt, wenn immer möglich, die Composite Widgets zuerst in einer dedizierten Umgebung entwickeln und austesten, um sie dann als fertige Module im Projekt einzusetzen.

I. Prototyp, statische Widgets	<code>perl -w explorer_test_0.pl</code>
II. Tk::Adjustereingebaut, die widgets expandieren nicht wie erwartet	<code>perl -w explorer_test.pl</code>
III. <configure> callbacks eingeführt.	<code>perl-w explorer_test_zx.pl</code>
IV. Mehrere Explorer bzw. docComponent implementiert, wobei docComponent1 ausgelagert werden kann	<code>perl -w explorer_test_zxx.pl</code>
V. ToolBar, Statusbar eingesetzt, Menu erweitert, docComponent1 kann ausgelagert werden via Button.	<code>cd ../xExplorer && perl -w explorer_test_zxx.pl</code>

Tabelle 2 Die wichtigsten Entwicklungsschritte und ihre Beispielskripte im subdir ./Explorer .

(siehe dazu auch .\Explorer\demoexplorer.bat oder ./Explorer/demoexplorer.sh)

Komponenten-Frame mit Adjustern steuern.

Das Module Tk::Adjuster kann als verschiebbare Trennung zweier benachbarten Widgets eingesetzt werden.

Dabei wird nur eine Seite, der sogenannte managed Widget, aktiv manipuliert, während die andere Seite dem geometry manager überlassen wird. Diese Manipulation bringt nicht immer das, was man sich wünscht, so dass etwas nachgeholfen werden muss (siehe dazu Tabelle 1, Schritt II).

Eine Methode besteht darin, je einen dazu passende Callback dem <configure>-Ereignis beider benachbarten Widgets zuzuweisen.

Dieser soll die Optionen, in unserem Falle die Größe, der im jeweiligen Widget enthaltenen Elemente den aktuellen Gegebenheiten anpassen. In der Tat wird dieser Ereignis immer nur dann ausgelöst, wenn die Geometrie der Widgets angepasst werden muss. Der jeweilige geometry manager entscheidet das, und man kann dies nur durch Anpassungen seiner Optionen zur Zeit der Widgetsanzeige, das heißt meistens beim Konstruktor, beeinflussen.

Besteht der Widget aus einer Aggregation von Widgets, wie in unserem Falle ein Frame, so wird der <configure>-Ereignis für jedes Element der Aggregation propagiert. Weil der Callback als erstes Argument immer die Reference zu dem Widget erhält, kann er die entsprechende Verarbeitung selektieren.

In unserem Falle soll der Callback die Höhe und Breite der Komponenten an die aktuelle Höhe und Breite des Komponenten-Frame anpassen, und sieht wie folgt aus:

```
$wr_018->bind('<Configure>' => [sub {  
  my $frame = shift;  
  my ($ROText,$actions) = @_;  
  my $xe = $frame->XEvent;  
  my ($frameW,$frameH) = ($xe->w, $xe->h);  
  my $actionsH = $actions->reqheight;  
  my $actualHeight = $frameH - $actionsH;  
  return 0 if ($actualHeight == $ROText->Height &&  
              $frameW == $ROText->Width);  
  $ROText->GeometryRequest($frameW, $actualHeight);  
  return 1  
}],$wr_015,$w_Action_021]  
);
```

Abbildung 2 Der <configure> Callback.

Interessant an diesem Code ist Folgendes :

- Der Inhalt der XEvent-Instanz ist in der Dokumentation des Module Tk::Event beschrieben. Allerdings ist sie ziemlich dürftig. Im Extremfall kann man die Analyse des Quellcodes oder ausgedehnte Tests nicht umgehen.
- die Höhe und Breite der veränderlichen Widgets wird mit den Messages Width und Height bestimmt, diejenige der unveränderlichen Widgets hingegen mit den Messages reqwidth() resp. Reqheight(), welche die notwendige Größe ergeben.
- Die Argumentenliste wird auch dann von der Reference des Widgets angeführt, für welchen das Ereignis ausgelöst wurde, wenn der Bind-Befehl mit einer Argumentenliste versehen ist.
- Es scheint, daß der Callback erst dann ausgelöst wird, wenn das Tk wieder in idle-Status ist, und alle Widget-Optionen nachgeführt wurden. Das ist nicht immer so, denn wenn der Callback-Quellcode in einem anderen Zusammenhang wieder verwendet wird, muss er von einem Message ,update' eingeleitet werden, um eben das Nachführen der Optionen und den idle-Modus sicherzustellen.

Als Alternative zu den eigens auskodierte Frames und Adjustern könnte man die Module Tk::Panedwindow oder Tk::Multi einsetzen. Beide

verwenden `Tk::Adjuster` um ihre Slaves zu dimensionieren, und weisen also die gleichen oben besprochenen Probleme auf.

Beide Module bringen also für unser GUI im Vergleich mit der oben skizzierten Lösung keine entscheidende Vorteile, welche ihr Einsatz mit den dazu gekoppelten Aufwänden rechtfertigen. Selbstverständlich sind beide Module in anderen Fällen auf Grund ihrer guten Qualität sehr brauchbar.

Das Auslagern des Komponenten-Frame

Das Auslagern von Widgets aus einem Dialog nach einen Toplevel besteht aus zwei Momenten:

das Anlegen der neuen Instanz und das Löschen der bestehenden Instanz.

Widget können grundsätzlich nicht kopiert werden, sie müssen also unter dem neuen Parent neu erstellt werden.

Das Löschen der bestehenden Instanz kann zwei Formen annehmen:

der Widget wird unwiederbringlich mit einem Message `destroy()` gelöscht, weil er nicht mehr gebraucht wird, oder er wird nur aus dem Dialog vorübergehend mit einem Message `packForget()` weggewischt, um dann später mit einem Message `pack()` wieder angezeigt zu werden.

In diesem Zusammenhang muß man folgendes merken: erstens, daß der geometry manager die Widget-Optionen nicht persistent speichert, man muß sie also beim Einlagern nochmals angeben. Zweitens, daß man auch die Position wieder angeben muß, was entweder mit Optionen oder mit einem ‚embedding Frame‘ geschehen kann.

Nur einzelne Widgets können bei solchen Operationen behandelt werden. Will man also die Widgets eines Frame auslagern, so muß man jedes einzelne Widget für sich auslagern. Dasselbe gilt für das Löschen und das Zurücklagern.

```
sub moveToToplevel {
    my ($mw,$source) = @_;
    my $parent = $mw->Toplevel();
    main::populateFl($parent);
    map {
        main::geomForget($_);
    } $source->children();
    $parent->protocol('WM_DELETE_WINDOW',
        [\&main::moveBack,$parent])
}
```

Abbildung 3 Widget eines Frame auslagern.

Will man das vermeiden, so muß man diese Widgets in einem Composite verpacken. Dieser dann kann als ein Widget behandelt werden.

```
sub moveToToplevel { ##
    my $self = shift;
    my ($mw) = @_;
    my $rv;
    my $parent = $mw->Toplevel();
    my $class = ref $self;
    $rv = $parent->$class()->pack();
    $self->geomForget();
    $parent->protocol('WM_DELETE_WINDOW',
        [\&MoveableComposite::moveBack,$self,$parent]);
    return $rv
}
```

Abbildung 4 Ein Composite widget auslagern.

Selbstverständlich muß man beim Auslagern prüfen, daß eine Komponente nur einmal ausgelagert werden kann.

Wenn zu einer bestimmten Zeit nur eine Komponente aktuell sein kann, so kann man prüfen ob eine solche existiert und noch nicht ausgelagert wurde. Dazu kann man die Existenz des Komponenteninhalts prüfen

```
sub checkFrame {
    my ($frame) = @_;
    my $rv = 0;
    map {
        $rv++ if defined $_->manager()
    } $frame->children();
    main::Log("checkFrame $rv");
    return $rv;
}
```

Abbildung 5 Prüfen ob die Widget eines Frame bereits ausgelagert wurden.

Eine andere und elegantere Lösung besteht darin, die ausgelagerte Komponente als Singleton zu definieren.

Dazu muß man jeder möglichen Komponenten eine eindeutige Identifikation geben und den Toplevel der ausgelagerten Komponente mit der Funktionalität eines Singletons ausstatten.

Das bedeutet, man erstellt einen neuen Typ Toplevel, welches von Tk::Toplevel und Singleton abgeleitet wird.

```
package SingletonToplevel;
{
    use vars qw($VERSION);
    $VERSION = '1.01';
    require Tk::Toplevel;
    require Tk::Derived;
    require Singleton;
    <\>@SingletonToplevel::ISA =
        qw(Tk::Derived Tk::Toplevel Singleton);
    Construct Tk::Widget 'SingletonToplevel';
    sub ClassInit {
        my $self = shift;
        $self->SUPER::ClassInit(@_);
    }

    sub Populate {
        my ($self, $args) = @_;
        my $id = delete $args->{-id};
        $self->SUPER::Populate($args);
        $self->Singleton($id);
        return $self;
    }
}
```

Abbildung 6 Der Composite widget SingletonToplevel.

Der Status der Widgets anpassen

Da das Steuern von Widgets bereits in [4] ausführlich beschrieben ist, wollen wir in diesem Zusammenhang nur zwei interessante Aspekte besprechen: das Erkennen des aktuellen Zustands und das laufende Nachführen der Widgets.

Wir gehen davon aus daß der Programmzustand hinreichend von einer Gruppe Programmvariablen charakterisiert wird. Jedesmal wenn mindestens eine solche Variable verändert wird, ist es notwendig zu überprüfen, ob die Widgets auch nachgeführt werden müssen.

Eine elegante Methode besteht darin, diese Variablen mittels Perl's 'tie' zu überwachen. Wenn eine Wertzuweisung erfolgt, so wird das entsprechende STORE - Message generiert, welche zu einer Anpassung des Zustands führt. Zusätzlich wird ein idle-callback generiert, welcher beim Erreichen des nächsten idle-Status von Tk selbsttätig ausgeführt wird.

Dieser Callback inspiziert den Zustand und führt die Anpassungen der Widgets aus.

```

use strict;
use lib './';
our $p1; our $p2; our $p3; our $state;
use Tk;

package state ;
{
    sub STORE {
        my ($this, $value) = @_;
        $$this = $value;
        $main::state = ($main::p1 & $main::p2 & $main::p3);
    }
}

package main;
my $debug = 1;

tie($main::p1,'state');
tie($main::p2,'state');
tie($main::p3,'state');

die "Could not tie p1, p2 or p3"
    unless(tied($p1) && tied($p2) && tied($p3));

my $mw = MainWindow->new();

...

sub computeState {
    ## dummy, the real computation is done in tie
    main::Log("state $state");
    return $main::state
}

sub dispatch { ## simple dispatcher
    $mw->Busy;
    my $rv;
    eval '$rv = &main::doTest($mw, @_);';
    if ($@) {
        $mw->Unbusy();
    } else {
        $mw->Unbusy();
        if (main::computeState()) {
            $mw->update();
            $mw->DoWhenIdle([\&main::updateState, $exit]);
            $mw->DoWhenIdle([\&main::updateState1, $exit]);
        }
    }
    return $rv
}

sub doTest {
    my $mw = shift;
    my ($rP) = @_;
    $rP = 1;    ## simulate a process which changes the program
state
    sleep 1;    ## ok, it takes a long time ...
    return 1;
}

```

Abbildung 7 Status der Widget anpassen mittels tie und Idle-callback.

Damit hat man eine non-invasive Implementation zur Hand, welche sehr unabhängig vom implementierten Quellcode ist, allerdings mit dem Nachteil, daß die relevanten Variablen global und `tied` sein müssen. Ob man diesen Nachteil immer in Kauf nehmen kann, bleibt fraglich.

Schlußfolgerungen

Auf Grund der Erfahrungen mit der Entwicklung der Beispielanwendung kann man folgende Punkte festhalten:

- Die Steuerung des GUI ist wichtiger und aufwendiger als die Definition bzw. Instanzierung der Widget selbst, wobei undokumentierte wichtige

Details der Module die Implementation erschweren, und den Aufwand wesentlich beeinflussen.

- Den Einsatz von Composite zur Implementation von Anwendungsdialoge erleichtern die Aufgabe, auch wenn die Familiarisierung etwas aufwendig ist.
- Die Anwendung von CPAN - Modules ist unersetzlich, auch dann wenn etwas nachkorrigiert werden muß.
- Die Qualität des Endproduktes ist für den ‚in house‘ - Einsatz mehr als ausreichend; für professionelle Ansprüche jedoch bleiben einige Zweifel.
- Für noch komplexeren Dialoge bleiben ebenfalls Zweifel, ob der Aufwand doch wegen der mangelhaften Dokumentation und versteckten Nebeneffekten bzw. Fehlern nicht unverhältnismäßig groß wird.

Es bleibt nur noch die Frage, ob vergleichbare Alternativen zu Perl/Tk überhaupt existieren.

Diese Frage wage ich zu verneinen, wenn man Wert auf Portabilität und Kosten legt, den immensen Leistungsumfang von Perl/Tk berücksichtigt, und den Rückgriff auf einem Browser nicht machen kann.

Literatur

[1] Alan Cooper und Robert Reimann: About Face 2.0. The Essential of Interaction Design. 2003

[2] Bastian Angerstein: Toolbox 4/2007: GUIs mit Perl.

[3] Graeme Geldenhuys: Modellierete GUIs. Toolbox 6'2008.

[4] Marco Marazzi: Eine Zustandsmaschine für Perl/Tk. Toolbox 3/2007.