

---

# 1 Gtk+2 - ein erweiterbares GUI-Toolkit

## Abstract

Gtk+2 (das **GIMP ToolKit**, Version 2) durchlief bei der Entwicklung zu Version 2 tiefgreifende Änderungen. Das Typsystem wurde stark abstrahiert, so daß sich Schnittstellen zu anderen Sprachen im wesentlichen auf das Bereitstellen des Objektsystems beschränken.

In diesem Dokument wird das Toolkit gtk+2 kurz vorgestellt, wobei Erfahrung mit GUI-Programmierung vorausgesetzt wird. Im Wesentlichen geht es darum, einige besondere Techniken vorzustellen sowie das Toolkit von Perl und C aus selbst zu erweitern.

## Gtk2 und Glib, eine Übersicht

### Übersicht über die G-Module

Es gibt eine Menge Bibliotheken, deren Namen mit G anfängt und im Dunstkreis von Gtk+ stehen. Vielleicht wäre es sinnvoll gewesen, eine neue G-Hierarchie in Perl zu begründen, so muß man sich mit einer teilweise nicht sehr logischen Namensgebung zu rechtzufinden.

### Glib

Glib ist die Schnittstelle zur `libglib` (sowie zu `libgobject`, `libgmodule` und `libgthread`). Es bildet die Grundlage für das Objekt- und Typ- und Signalsystem, bietet Hilfsfunktionen zum Umwandeln von Dateinamen, Logging usw.

### Gtk2

Dieses Modul ist eine Sammelstelle für diverse Bibliotheken: neben `libgtk+` auch `libgdk`, `libgdk_pixbuf` und `libpango`.

Sie reicht für fast alle Anwendungen aus.

### Gtk2::GLExt, Gtk2::GladeXML, Gtk2::TrayIcon und Gtk2::Spell

Bieten Integration von OpenGL, GladeXML, EggTrayIcon und Spell-checking für Text-Widgets.

### Gnome2

Schnittstelle zur `libgnome` und vielem mehr.

... muß man nicht benutzen, kann mir aber vorstellen, daß sie für die Erstellung von Gnome-Applikationen ganz nützlich sind :)

### Gnome2::Canvas, Gnome2::GConf, Gnome2::PanelApplet, Gnome2::Print, Gnome2::Rsvg, Gnome2::VFS, Gnome2::Vte, Gnome2::Wnck.

Jede Menge Gnome-Zeugs.

## Die Widgets von Gtk2 sind GObject

Wie jedes andere Toolkit auch, bietet Gtk2 eine Menge vorgefertigter *Widgets*: *Buttons*, *Labels*, *Frames* und andere "einfache" Widgets, aber auch komplexe, wie das *Text*- (komplett in Unicode, einbettbare Objekte) oder das *TreeView*- (Listen und Bäume im MVC-Modell) Widget.



Alle diese Widgets sind sogenannte *GObjects* (davon abgeleitet sind in Gtk2 dann sogenannte *Widgets*, GUI-Objekte), und haben folgende Features:

### Eine Klasse

Jedes GObject ist eine Instanz einer Klasse (*GObjectClass*), eine Datenstruktur, die Informationen über Methoden u.ä. speichert.

Eine *GObjectClass* ist allerdings nur ein Spezialfall eines *GTypes*: Die Glib bildet ein komplettes Typsystem mit Klassen, Aufzählungen, Flags, einfachen Datentypen wie Integer oder Fließkommazahlen oder auch C-structs an. All dies ist zur Laufzeit in einer Form verfügbar, in der sich auch neue - zur Compilezeit nicht bekannte - Datentypen verwenden lassen.

Schnittstellen für andere Sprachen werden dadurch einfach, da man "lediglich" die Konzepte "Datentyp", "Konverter", "KlasseC", "Methode", "Bitset" usw. umsetzen muß. Konkrete Instanzen dieser Typen funktionieren zur Laufzeit automatisch.

---

Bei einer gut geschriebenen C-Bibliothek, die auf Glib bzw. GObject aufsetzt, muß man eigentlich nur die Initialisierungsfunktion aufrufen, die die Datentypen registriert, und kann dann sofort von Perl darauf zugreifen - ohne eine einzige XS-Funktion geschrieben zu haben.

## Methoden

Jedes GObject hat mindestens Methoden zum Erzeugen (**constructor**) und Zerstören (**finalize**), Setzen und Lesen von Properties und eine Spezialität von GObject - **dispose**. Letzteres ist eine Art Anfrage an ein Objekt, sich freiwillig aufzulösen.

Gtk2 fügt noch einige weitere hinzu, wie die **destroy**-Methode, die aktiv versucht, ein Objekt zu zerstören: In GUI-Programmen ist es manchmal schwer, alle Referenzen im richtigen Moment aufzulösen, da sie häufig zirkulär sind. Manche Objekte "leben" auch ganz eigenständig, z.B. wird ein Toplevel-Fenster im allgemeinen vom Benutzer "referenziert", was sich aber nicht notwendigerweise im Referenz-Zähler niederschlägt.

Um solche Objekte loszuwerden, kann man sie bitten, sich zu *destroyen*. Dabei gibt es alle Referenzen auf andere Objekte frei und bittet diese, sich ebenfalls zu destroyen. Dadurch werden zirkuläre Referenzen effektiv aufgebrochen.

Methoden kann man in Perl nicht direkt implementieren, dafür aber eigene Signale, die beliebige Parameter oder Rückgabewerte besitzen können und mehr oder weniger direkt von C oder anderen Sprachen aus aufgerufen werden können.

## Signale

Signale ähneln Methoden, Callbacks oder sogenannten "Delegates". Wie Methoden sind sie fest an eine Klasse gebunden und rufen (in C) eine Methode der Klasse auf, sobald sie ausgelöst werden.

Ein Signal kann von jedem ausgelöst werden, insbesondere lösen aber Ereignisse (Events) in Gtk2 Signale aus. Der Signal-Handler eines Objektes verarbeitet das Ereignis (und beendet die Signalauslieferung) oder tut nichts, woraufhin Gtk2 ein anderes Objekt als Empfänger sucht.

Zusätzlich kann man vor oder nach dem eigentlichen Signal-Handler des Objektes weitere Callbacks einklinken. Z.B. löst das Text-Widget beim Mausklick rechts ein **populate-menu**-Signal aus, das normalerweise ein Popup-Menü generiert. Das kann man verändern oder ganz überschreiben. Und wenn man Mausklicks abfangen möchte, kann man das **button-press-event**-Signal überschreiben und so Mausklicks vom Text-Widget fernhalten.

## Properties (Eigenschaften)

Properties sind von der jeweiligen Klasse definierte Eigenschaften, die das Objektverhalten beeinflussen.

Sie werden von der Klasse festgelegt und auch von ihr benutzt.

### Schlüssel-Wert-Paare

Damit andere Benutzer mit einem Objekt Daten verknüpfen können, kann man an ein GObject noch beliebige Schlüssel-Wert-Paare (sogar mit Destruktor) packen, wobei sich dies von Perl aus auf Integer beschränkt, da das Glib-Modul jedes GObject automatisch mit einem Perl-Hash verknüpft und diesen auf Perl-Ebene zur Speicherung zur Verfügung stellt.

### Informationen über Typen und Klassen zur Laufzeit abfragen

Die Typinformationen, wie Werte von Aufzählungen, Namen von Datentypen, Objekt-Signale, -Properties uvm., sind alle zur Laufzeit abfragbar.

Dadurch wird das Schreiben von Klassenbrowsern oder RAD-Tools sehr einfach. Aber auch beim Entwickeln von Programmen ist das nützlich, da man nicht immer die Dokumentation zu Rate ziehen muss.

Z.B. liefert `$obj->list_properties` (oder `Glib::Type::list_properties` Klassenname) eine Liste von Arrayrefs mit Name, Typ, Flags und Beschreibung zurück, die man sich z.B. mit folgender Funktion ausgeben lassen kann:

```
use Gtk2;

sub gobj_props {
    for ($_[0]->list_properties) {
        printf "%-16s %-24s %-24s %s\n", $_->{name}, $_->{type},
            (join ":", @{$_->{flags}}), $_->{descr};
    }
}
```

Der Aufruf `gobj_props "Gtk2::Widget"` (oder mit einem `Gtk2::Window-Object` als Argument) liefert folgende Properties, die alle Gtk2-Widgets besitzen:

<code>user-data</code>	<code>gpointer</code>	<code>readable:writable</code>
Anonymous User Data Pointer		
<code>name</code>	<code>Glib::String</code>	<code>readable:writable</code>
The name of the widget		
<code>parent</code>	<code>Gtk2::Container</code>	<code>readable:writable</code>
The parent widget of this widget. Must be a Container widget		
<code>width-request</code>	<code>Glib::Int</code>	<code>readable:writable</code>
Override for width request of the widget,		

---

```

    or -1 if natural request should be used
    ...
extension-events Gtk2::Gdk::ExtensionMode readable:writable
    The mask that decides what kind of extension events this widget gets
no-show-all      Glib::Boolean             readable:writable
    Whether gtk_widget_show_all() should not affect this widget

```

Signale lassen sich ebenso abfragen:

```

sub gobj_signals($) {
    # Signale haben Argumente und Parameter, daher Dumpvalue
    use Dumpvalue;
    Dumpvalue->new (bareStringify => 0)->dumpValue ($_)
        for Glib::Type->list_signals ($_[0]);
}

gobj_signals "Gtk2::Widget";

...
'signal_id' => 23
'signal_name' => 'button-press-event'
'itype' => 'Gtk2::Widget'
'param_types' => ARRAY(0x83b4f28)
    0 'Gtk2::Gdk::Event'
'return_type' => 'Glib::Boolean'
'signal_flags' => [ run-last ]
    -> 2
...

```

Das `button-press-event`-Signal möchte also ein Event als einziges Argument und liefert einen Wahrheitswert.

Die Abstammung von `Gtk2::Window` ist schnell gefunden:

```

warn join " ", Glib::Type->list_ancestors ("Gtk2::Window");
Gtk2::Window Gtk2::Bin Gtk2::Container Gtk2::Widget
    Gtk2::Object Glib::Object at gtk.pl line xx.

# und list_interfaces wollte ich auch erwähnen:
warn join " ", Glib::Type->list_interfaces ("Gtk2::Window");
Gtk2::Atk::ImplementorIface at gtk.pl line xx.

```

Wenn man `gtk+` von C her kennt (oder eben die C-Dokumentation benutzt), kann man aus dem C-Typnamen den Namen in Perl erfahren:

```
warn Glib::Type->package_from_cname ("GtkWindowPosition");
Gtk2::WindowPosition at gtk.pl line xx.
```

```
warn Glib::Type->package_from_cname ("GdkPixbuf");
Gtk2::Gdk::Pixbuf at gtk.pl line xx.
```

Aufzählungen und Flags kann man ebenso hinterfragen:

```
sub g_vals($) {
    for (Glib::Type->list_values ($_[0])) {
        printf "%-30s %s\n", $_->{name}, $_->{nick};
    }
}

g_vals "Gtk2::Gdk::EventMask";

GDK_EXPOSURE_MASK           exposure-mask
GDK_POINTER_MOTION_MASK     pointer-motion-mask
...
GDK_SCROLL_MASK             scroll-mask
GDK_ALL_EVENTS_MASK         all-events-mask
```

**Aufzählungen und Flags** Natürlich kann man zur Laufzeit auch eigene Typen hinzufügen. Z.B. einen neuen Flag-Typ:

```
Glib::Type->register_flags (Meine::Flags =>
    'eins',           # implizit 1<<0
    'zwei',          # implizit 1<<1
    ['drei' => 15 ], # explizit 15
    ['vier' => 35 ], # explizit 35
    'fuenf',        # implizit 1<<4
);
```

Diese neuen Typen sind gleichwertig zu den vorhandenen, können also von C oder anderen Programmiersprachen aus benutzt werden.

Aufzählungen und Flags sind in Perl überladen. Das ist besonders hilfreich in Event-Handlern:

```
$widget->signal_connect (button_press_event => sub {
    if ($_[1]->type eq "button-press") {
        if ($_[1]->button == 1) {
            if ($_[1]->state == "shift-mask") {
```

---

```

    # Umschalt-Klick
    if ($_[1]->state * "shift-mask") { # oder '&'
        # Umschalt-Klick (eventuell plus anderen Modifiern)
        if ($_[1]->state * ["control-mask", "shift-mask"]) { # oder '&'
            # Klick + Umschalt ODER Strg (eventuell plus anderen Modifiern)
            if ($_[1]->state >= ["control-mask", "shift-mask"]) {
                # Klick + Umschalt UND Strg (Übermenge)
            }
        }
    }

```

Oder beim Manipulieren von Event-Masken:

```

$widget->set (events =>
    $widget->get ('events') - [qw(key_press_mask button-motion-mask)]
);

```

Auch eigene Klassen sind möglich, siehe Teil 2 :)

### Beispiel: Session-Management

Häufig gibt es in C sogenannte *convenience methods* (“Bequemlichkeitsfunktionen”). Sofern sie sinnvoll sind, wurden sie bei Glib und Gtk2 umgesetzt. So gibt es z.B. eine `set_events`- und eine `add_events`-Methode, die bei Gtk2-Widgets die Event-Maske verändern. Doch für alle Properties kann man die generischen `set`- und `get`-Methoden verwenden.

Damit läßt sich ein einfaches Session-Management schreiben:

```

use Scalar::Util;

my %get = (
    window_size    => sub { [ ($_[0]->allocation->values)[2,3] ] },
    column_size    => sub { $_[0]->get("width") || $_[0]->get("fixed_width") },
    modelsortorder => sub { [ $_[0]->get_sort_column_id ] },
);

my %set = (
    window_size    => sub { $_[0]->set_default_size (@{$_[1]}) },
    column_size    => sub { $_[0]->set (fixed_width => $_[1]) },
    modelsortorder => sub { $_[0]->set_sort_column_id (@{$_[1]}) },
);

my $state = Storable::retrieve "session.dat"; # Alte Session-Daten laden
my %widget;

```

```
sub state {
  my ($widget, $class, %attr) = @_;

  while (my ($k, $v) = each %attr) {
    $v = $state->{$class}{$k}
    if exists $state->{$class} && exists $state->{$class}{$k};

    $set{$k} ? $set{$k}->($widget, $v) : $widget->set ($k => $v);
  }

  $widget{$widget} = [$widget, $class, \%attr];
  Scalar::Util::weaken $widget{$widget}[0];
}
```

Naja, *gaanz* so einfach ist es nicht: Man kann damit schon recht viel anfangen, und es sind zugegebenermaßen mehr als drei Zeilen.

Was macht der Code? Es gibt Eigenschaften, die man über eine Session hinweg retten möchte, aber keine Properties sind. Für diese Eigenschaften (z.B. `window_size`) definiere ich eigene `set`- und `get`-Funktionen.

Die eigentliche Schnittstelle ist die `state`-Funktion. Sie wird so benutzt:

```
my $window = new Gtk2::Window;
state $window, "game::window", window_size => [600, 500];

my $hpane = new Gtk2::HPaned;
state $hpane, "game::hpane", position => 500;
```

Diese beiden Widgets werden in einem (nicht sehr fiktiven) Go-Programm benutzt um ein Spiel darzustellen. Der erste Aufruf von `state` sagt: Beim Widget vom Typ `game::window` (ein willkürlicher Name) soll die Property `window_size` von einer Session in die nächste Übertragen werden. Falls keine Sessiondaten existieren, nimm als Default `[600, 500]`. Der zweite Aufruf macht das gleiche für die `Position`-Property des `Gtk2::HPaned`-Widgets.

Beides zusammen sorgt dafür, daß neue Spielfenster in der gleichen Größe erscheinen wie bei der letzten Benutzung und daß die horizontale Unterteilung (links Spielbrett, rechts Chat-Bereich) ebenfalls erhalten bleibt.

In der Realität benutze ich eine etwas erweiterte `state`-Funktion mit einem `$instance`-Argument: manche Benutzer möchten z.B. verschiedene Chat-Räume in verschieden großen Fenstern darstellen. Das zusätzliche Argument dient der Unterscheidung: wird ein neuer Raum betreten, werden die Werte des Raum-Typs benutzt; wird ein bekannter Raum betreten, werden dessen bekannte Werte benutzt.

---

Die Funktion liest übrigens nur Werte aus den Sessiondaten `$state` aus, aber irgendwie müssen sie auch dort abgelegt werden. Dies geschieht in meinen Programmen meistens durch Benutzeraufforderung: der Benutzer stellt sein gewünschtes Layout zusammen und drückt dann `Layout Speichern` o.ä., worauf dann folgende Funktion aufgerufen wird.

```
sub save_state {
  for (grep $_, values %widget) {
    my ($widget, $class, $attr) = @$_;

    next unless $widget; # widget schon tot...
    $widget->realize if $widget->can ("realize");

    while (my ($k, $v) = each %$attr) {

      $state->{$class}{$k} =
        $get{$k} ? $get{$k}->($widget) : $widget->get ($k);
    }
  }

  Storable::nstore $state, "session.dat";
}
```

Sie geht alle Widgets durch, die mit `state` behandelt wurden, prüft, ob sie noch existieren, sorgt (mit `realize`) dafür, daß die Eigenschaften auch mit sinnvollen Werten gefüllt sind, und speichert dann die entsprechenden Eigenschaften in `$state`.

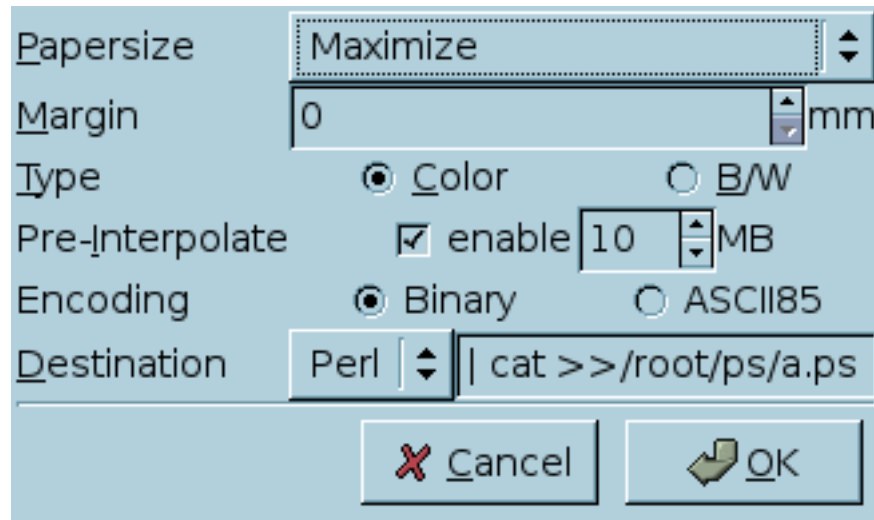
### Beispiel: Rapid Application Development mit Glade – mein Weg

Ich will es nicht verschweigen: Ich *HASSE* (hoffentlich ist das deutlich genug) GUI-Programmierung. Nichts ist langweiliger bzw. nerviger, als Dialogboxen zusammenzustöpseln oder Widgets in der Hierarchie zu verschieben.

Glücklicherweise gibt es RAD-Tools, mit denen man sich seine Applikation zusammenklicken kann. Was mich an RAD-Tools allerdings stört ist ihre Starrheit: meistens wollen sie gleich den Quelltext schreiben, inklusive des Hauptprogramms. Und häufig ist es schwierig, generelle Klassen zu definieren, z.B. ein Chat-Room-Widget, von dem es zur Laufzeit beliebig viele geben kann.

Auch *Glade* – ein (das!) Gtk+ und Gtk+2 RAD-Tool – stellt hier keine Ausnahme dar. Zum Glück kann man Glade auch etwas anders zum Entwickeln benutzen, und diesen anderen Weg (ich kenne niemanden, der Glade so benutzt), möchte ich hier beschreiben.

Zuallererst ist es notwendig, die Widget oder die UI in Glade zu erstellen und als GladeXML-Datei zu speichern. Z.B. habe ich für ein Bilder-Anzeige-Programm (`Gtk2:CV`, reiner Beta-Code) einen Ausdrucken-Dialog erstellt:



Die Beschreibung ist in der XML-Datei `cv.glade` gespeichert. Darin habe ich den einzelnen Widgets Namen gegeben, z.B. heißt das Dialogfenster `PrintDialog`, das Option-Menü zum Auswählen der Papiergröße `papersize` usw.

Den ganzen Dialog kapselte ich in einem Perl-Modul namens `Gtk2::CV::PrintDialog` ab, mit insgesamt 70 Zeilen und zwei Methoden. Die kürzeste davon benutzt ein anderes Modul `Gtk2::CV::PostScript` zum Ausdrucken und ist völlig uninteressant :) Die längste ist die `new`-Methode, die ein neues `PrintDialog`-Widget erzeugt. Zuerst der Standardkopf:

```
use Gtk2;
use Gtk2::GladeXML;
# uvm.

sub new {
    my $class = shift;
    my $self = bless { @_ }, $class;
```

Als nächstes wird die GladeXML-Datei geladen:

```
$self->{dialog} = my $d
    = new Gtk2::GladeXML ".../cv.glade", "PrintDialog";
```

Mit `PrintDialog` wird ein Teilbaum aus dem Widget-Baum geladen, und zwar der mit dem `PrintDialog`-Namen. So kann man viele verschiedene Dialoge in einer Datei ablegen. Das Laden ist nicht unbedingt eine schnelle Angelegenheit, ist aber in den meisten Fällen flott genug, um bei jedem Aufruf von neuem gemacht zu werden (und man hat sowieso keine Wahl :)

---

Das Ergebnis ist *kein* Widget, sondern ein `Gtk2::GladeXML`-Objekt, das die Methode `get_widget` besitzt, mit der man anhand des Namens einzelne Widgets “herausgreifen” kann.

Damit fülle ich den Dialog mit Leben:

```
$d->get_widget ("destination")->set (
    text => $ENV{CV_PRINT_DESTINATION} || "| lpr");

my $menu = $d->get_widget ("papersize")->get_menu;
for (Gtk2::CV::PostScript->papersizes) {
    my ($code, $name, $w, $h) = @$_;
    $menu->append (my $item = new Gtk2::MenuItem $name);
    $item->set_name ($code);
}
$menu->show_all;
$d->get_widget ("papersize")->set_history (0);

$d->get_widget ("PrintDialog")->signal_connect (close => sub {
    $_[0]->destroy;
});
$d->get_widget ("PrintDialog")->signal_connect (response => sub {
    if ($_[1] eq "ok") {
        $self->print (
            size          => (Gtk2::CV::PostScript->papersizes)
                          [ $d->get_widget ("papersize")->get_history ],
            margin        => $d->get_widget ("margin")->get_value,
            ...
            destination   => $d->get_widget ("destination")->get ("text"),
            binary        => $d->get_widget ("encoding_binary")->get ("active"),
        );
    }
    $_[0]->destroy;
});

return $self; # Ende
}
```

Jetzt wird klar, warum das Ding `$d` heißt und nicht einen längeren Namen trägt.

Wer das GladeXML-Modul kennt, weiß, daß es noch viel “einfacher” geht, z.B. mit `signal_autoconnect_from_package`. Leider weiß man in den Callbacks, die diese Methode mit den Widgets verbindet, nichts mehr über das ursprüngliche `PrintDialog`-Objekt,

so daß sie mir recht nutzlos erscheint, wenn man nicht alles in globalen Variablen ablegen will (was häufig ausreicht, aber nicht gerade sauber ist).

Daher habe ich die Funktion `signal_autoconnect_all` geschrieben. Sie geht einen etwas anderen Weg und erlaubt es, die Signal-Callbacks als Closures anzugeben. Damit wäre das Namensraum-Problem gelöst. Leider muss man dazu jedem Signal-Handler einen Namen geben, und das ist so umständlich, daß ich die Funktion selbst nie benutzt habe und es einfach "per Hand" anhand der Widget-Namen mache.

Dies alles heißt nicht, daß man es nicht einfacher anders besser machen könnte. Es ist einfach "mein Weg", Glade zu benutzen. Und wenn ich noch etwas darüber nachdenke, muß ich sagen, daß es mit Glade nicht wirklich einfacher ist, Widgets umzustellen oder GUIs zu designen.

\*Seufz\*.

## Teil 2 - Gtk+ als solide Grundlage für Eigenentwicklungen

Gtk+ eignet sich hervorragend für Eigenentwicklungen - auch größere. Belegt wird dies dadurch, daß man es vollständig von Perl aus erweitern kann und nicht auf einige vorgefertigte Klassen beschränkt ist. Außerdem kann man Widgets aus beliebigen Sprachen miteinander integrieren und sogar Teile in Perl und einer anderen Sprache schreiben, was große Anwendungen einfacher macht: zeitkritische Teile und Widgets werden (z.B.) in C implementiert, während Perl bequem für die GUI-Logik benutzt werden kann.

### Perl-Objekte vs. Glib-Objekte

Meiner Meinung nach ist der Hauptgrund für das typisch "skripthafte" Aussehen vieler Perl/Tk (oder Perl/Gtk oder Perl/Qt oder auch Tcl/Tk) Programme die ausschließliche Verwendung von Standard-Widgets: Die Programme sehen aus wie aus dem Baukasten, weil sie nur aus Baukastenteilen bestehen. "Richtige" Anwendungen (wie z.B. Gimp) benutzen aber eigene Widgets, sofern das Sinn macht.

Viele Toolkits machen es einem sehr schwer, eigene Widgets zu programmieren. Bei Gtk2 kann man zwischen einfachen Ableiten und "echten" Gtk2-Widgets wählen.

**Die Perl-Methode** Die "Perl-Methode" besteht darin, eine neue Klasse von einem bestehenden Widget abzuleiten. Das ist einfach und erfordert keinerlei Umdenken:

```
package IntegerWidget;

use base Gtk2::Entry;

sub new {
    my ($class, %prop) = @_;
    my $self = $class->SUPER::new;
```

---

```

$self->set_text ($prop{text});

$self->signal_connect (insert_text => sub {
    my ($self, $text, $length, $position) = @_;
    $text =~ y/0-9//cd;
    ($text, $position);
});

$self;
}

...

$container->add (new IntegerWidget text => 50);

```

Das geht mit fast jedem Toolkit. Aber ein richtig neues Widget kommt dabei nicht heraus. Trotzdem ist dieses Vorgehen meistens nützlich, wenn man eine große Menge an Widgets benötigt, die mehr oder weniger eine speziell konfigurierte Version eines Standard-Widgets ist.

Einige mehr oder minder wichtige Dinge kann man damit allerdings nicht implementieren: eigene Signale und Eigenschaften. Sicher kann man eigene `set_xxx`-Methoden schreiben, aber sie verhalten sich nicht wie Glib-Properties (das Session-Management-Beispiel müßte speziell abgeändert werden). Und Signale kann man durch Callbacks implementieren, aber der Nutzer des Widgets muß dann auf `signal_connect` verzichten und das Widget verhält sich insgesamt nicht wie andere Widgets,

Und schließlich kann man nur sehr schlecht selbst zeichnen und auf Ereignisse (z.B. `Expose`) reagieren.

**Die Glib-Methode** Mit der “Glib-Methode” wird keine normale Perl-Klasse erstellt, sondern eine echte Glib-Klasse (und zusätzlich eine Perl-Klasse, aber eben keine ganz normale).

Dabei hilft einem das `Glib::Object::Subclass`-Modul, das ähnlich wie ein `Pragma` wirkt:

```

package IntegerWidget;

use Glib::Object::Subclass
    Gtk2::Entry,
    signals => {
        insert_text => sub {
            my ($self, $text, $length, $position) = @_;
            $text =~ y/0-9//cd;

```

```
        $self->signal_chain_from_overridden ($self, $text, $length, $position);
    },
};
```

`Gtk2::Entry` ist die Basisklasse und mit `signals` werden vorhandene Signale überschrieben bzw. neue deklariert. Überschreibt man ein vorhandenes Signal (wie hier) kann man das ursprüngliche Signal mit `signal_chain_from_overridden` aufrufen, was einem `SUPER`-Aufruf in Perl entspricht.

Das Modul setzt `@ISA` und liefert auch gleich einige Methoden, z.B. `new`. Der Grund liegt darin, daß die `new`-Methoden der existierenden `Gtk2`-Widgets das Klassenargument ignorieren, wie auch die C-Funktionen, die sie aufrufen, kein Klassenargument besitzen. Bei der Umsetzung von `gtk+` in Perl dachte man, die normalen Konstruktoren seien wichtiger, da sie häufiger aufgerufen werden.

Die von `Glib::Object::Subclass` gelieferte `new`-Methode ruft lediglich `Glib::Object::new` auf. Das ist der generische Objekt-Konstruktor, mit dem man jedes `Glib`-Objekt (bzw. auch davon abgeleitete) erzeugen kann. Er akzeptiert `property => wert`-Paare, was für dieses Beispiel ausreicht, man muss also nichts weiter implementieren.

Interessant wird es, wenn man zusätzliche Properties und Signale registriert:

```
package IntegerWidget;

use Glib::Object::Subclass
    Gtk2::Entry,
    signals => {
        # neues signal, range-exception
        range_exception => {
            class_closure => sub { <default-implementierung> },
            flags          => [qw(run-first)],
            param_types   => [Glib::Integer],
        },
        # vorhandene Signale überschreiben
        insert_text => sub {
            ...
        },
        changed => sub {
            my ($self) = @_;
            my $text = $self->get_text;
            $self->signal_emit (range_exception => $text)
                if $text < $self->{min} || $text > $self->{max};
        },
    },
```

---

```

},
properties => [
  Glib::ParamSpec->string (
    min => "Minimum",
    "Values smaller than this value cause a range_exception",
    0, [qw(readable writable)]
  ),
  Glib::ParamSpec->string (
    max => "Maximum",
    "Values larger than this value cause a range_exception",
    0, [qw(readable writable)]
  ),
],

```

Hier wird ein neues Signal `range_exception` eingeführt (das standardmäßig garnichts macht). Es wird ausgelöst, wenn die eingegebene Zahl kleiner oder größer als die Properties `min` oder `max` ist, was in der `changed`-Methode festgestellt wird.

Wird eine Property gelesen, ruft Glib die `GET_PROPERTY`-Funktion auf. Sie ist in Grossbuchstaben geschrieben, weil sie nur von Glib und nie vom eigenen Code aus aufgerufen wird, ähnlich wie `AUTOLOAD`. Genauso wird beim Verändern die `SET_PROPERTY`-Funktion aufgerufen. `Glib::Object::Subclass` liefert zwei Default-Implementierung, die die Werte aus `$self->{name}>` lesen bzw. dort ablegen. Man kann diese Funktionen natürlich überschreiben:

```

sub SET_PROPERTY {
  my ($self, $pspec, $newval) = @_;
  $self->{$pspec->get_name} = $newval;
  # nicht so sinnvoll:
  warn "max wurde geändert!!!" if $pspec->get_name eq "max";
  # sehr sinnvoll:
  $self->emit_signal ("changed");
}

```

Wenn `min` oder `max` geändert werden, löst diese Implementierung ein `changed`-Signal aus, um die neuen Grenzen zu prüfen.

Die großgeschriebenen Funktionen sind übrigens genau das: Funktionen. Sie sollten niemals ihre SUPER-Implementierung aufrufen, das erledigt Glib automatisch.

Die beiden anderen Funktionen, die Glib aufruft sind `INIT_INSTANCE` und `FINALIZE_INSTANCE`. Sie sollten das Objekt initialisieren bzw. auflösen. *Niemals* sollte man `DESTROY` implementieren, da `DESTROY` intern von Glib benutzt wird und mehrmals aufgerufen werden kann.

**Referenzprobleme** Jetzt zu einem kleinen aber feinen Problem. Was ist der Unterschied der beiden folgenden `INIT_INSTANCE`-Funktionen?

```
use Glib::Object::Subclass
    Gtk2::Frame;

sub INIT_INSTANCE {
    my ($self) = @_;
    $self->add ($self->{entry} = new Gtk2::Entry);
    $self->{entry}->signal_connect (changed => sub {
        warn $self->{entry}->get_text;
    });
}

sub INIT_INSTANCE {
    my ($self) = @_;

    $self->add ($self->{entry} = new Gtk2::Entry);
    $self->{entry}->signal_connect (changed => sub {
        warn $_[0]->get_text;
    });
}
```

Nun, die obere Version erzeugt eine neue zirkuläre Referenz: Die Closure im ersten Fall referenziert das äußere `$self`. Wie löst man diese Referenzen auf?

Das größte Problem neuer Widgets sind zusätzliche Referenzen, die das Widget daran hindern, sich aufzulösen.

Gtk2 löst dieses Problem durch ein explizites `destroy`-Signal: Wird es ausgelöst, sollte das Objekt möglichst alle Referenzen auf andere Objekte freigeben. Fällt dabei der Referenz-Zähler auf 0 ab, wird das Objekt aufgelöst.

Container lösen dabei rekursiv das `destroy`-Signal auf die in ihnen enthaltenen Objekte auf. Dabei werden normalerweise alle Referenzen aufgelöst, da z.B. Signal-Handler ebenfalls entfernt werden, die Selbstreferenz im obigen Beispiel ist dabei also eigentlich kein Problem.

Viele Perl-Widgets sind aber versteckte Container, da man häufig zusammengesetzte Widgets erzeugt, wie im obigen Beispiel. Dann kann es passieren, das auch nach einem `destroy` ein eingebettetes Widget eine Referenz auf den Vater enthält, und der hält gleichermaßen eine Referenz (`$self->{entry}`).

Dies kann man umgehen, indem man bei einem `destroy` einfach `$self` löscht:

```
$self->signal_connect (destroy => sub { $_[0] = () });
```

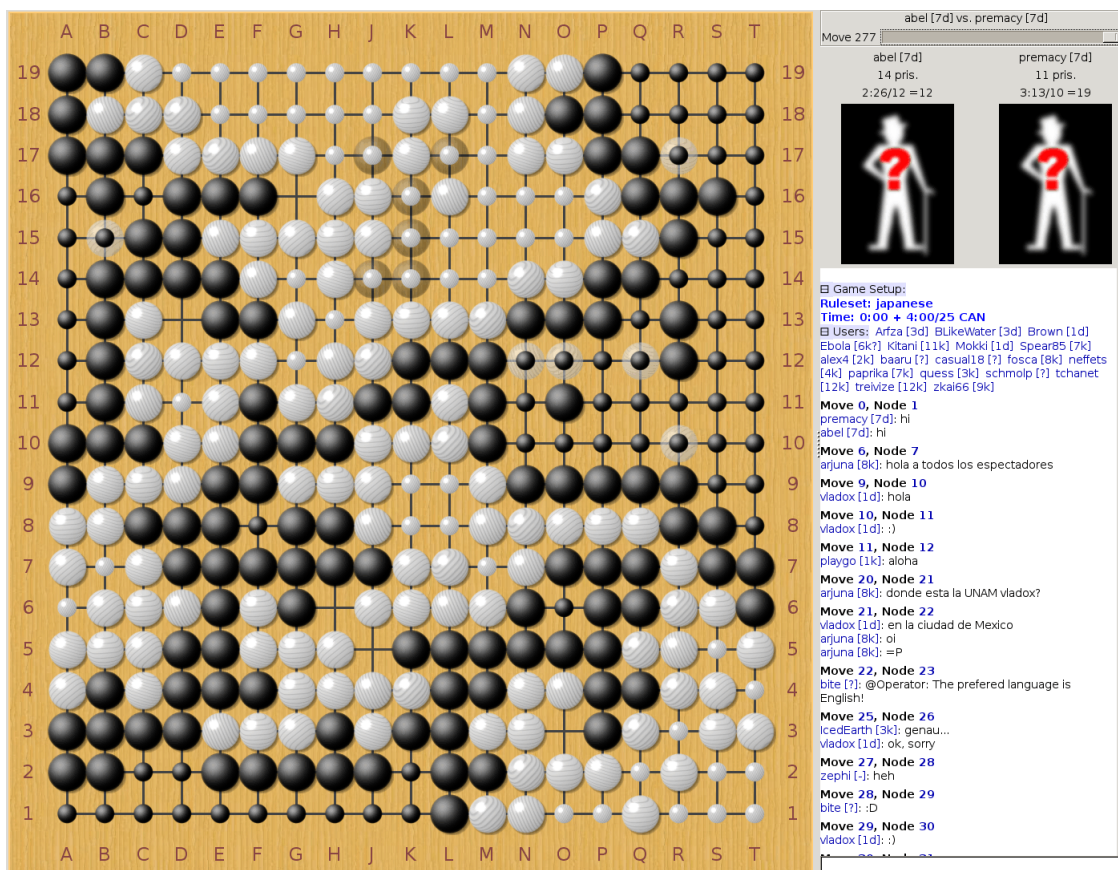
Das ist etwas radikal, daher gibt es zwei sauberere Lösungen. Einmal kann man bei einem `destroy` wirklich sauber aufräumen:

```
$self->signal_connect (destroy => sub {  
    (delete $self->{entry})->destroy;  
    # etc. für andere Widgets  
});
```

Oder man verhindert zusätzliche Referenzen auf Perl-Seite, was natürlich nur bei einfachen Referenzen geht:

```
use Scalar::Util ();  
Scalar::Util::weaken $self->{entry};
```

**Beispiel: ein Go-Brett** Für einen Go-Server-Client benötige ich ein Widget, das ein Go-Spielbrett darstellt (das Bild zeigt es beim Auszählen der Punkte eines Spiels).



Der Code für Board-Widget fängt so an:

```
use Glib::Object::Subclass
    Gtk2::AspectFrame,
    properties => [
        Glib::ParamSpec->IV (
            "size",
            "Board Size",
            "The Go Board size, 2..38",
            2, 38, 19,
            [qw(construct-only writable readable)],
        ),
        Glib::ParamSpec->IV (
            "cursor-mask",
            "cursor mask",
            "The mask used to show a cursor",
            0, 1<<30, 0,
            [qw(writable readable)],
        ),
        Glib::ParamSpec->IV (
            "cursor-value",
            "cursor value",
            "The value used to show a cursor",
            0, 1<<30, 0,
            [qw(writable readable)],
        ),
    ],
    signals => {
        "button-press" => {
            flags      => [qw/run-first/],
            return_type => undef, # void return
            param_types => [Glib::Int, Glib::Int, Glib::Int],
        },
        "button-release" => {
            flags      => [qw/run-first/],
            return_type => undef, # void return
            param_types => [Glib::Int, Glib::Int, Glib::Int],
        },
        destroy      => sub {
            $_[0]->signal_chain_from_overridden;
            %{$_[0]} = ();
        }
    }
}
```

---

```
    },  
};
```

Das Brett wird also von einer `Gtk2::AspectFrame` abgeleitet: Das Brett ist zwar nicht ganz quadratisch, hat aber immer dasselbe Seitenverhältnis von ca. 11:10, so daß sich ein `AspectFrame` anbietet.

Als erstes werden drei Properties definiert: die Größe (`size`), die die Anzahl der Linienschnittpunkte angibt. Sie kann nur beim Erzeugen gesetzt werden (Flag `construct-only`), nicht mehr später. Dies vereinfacht den Code etwas. Die beiden anderen Properties definieren das Aussehen des “Cursors”: wenn man mit der Maus über das Brett fährt, kann man damit z.B. einen halbdurchscheinenden Stein anzeigen.

Dann folgen zwei neue Signale, die ausgelöst werden, wenn man einen Mausknopf auf dem Brett drückt bzw. losläßt. Sie haben drei Integer-Parameter: Die Nummer des Mausknopfes und die Koordinaten des Klicks.

Das `destroy`-Signal wird überschrieben, das `$$self` brutal löscht. Da ich von `Gtk2::AspectFrame` ableite weiß ich, daß ich kein anderes Perl-Widget in der Hierarchie über mir stören kann, da `Gtk2`-Widgets selbst den Perl-Hash `$$self` nicht antasten.

Danach folgen einige Konstantendefinitionen für Steine, z.B. `MARK_B` oder `MARK_SQUARE`, die zusammenaddiert das Aussehen einer Brettposition festlegen oder die relative Größe von Steinen (schwarze Steine sind etwas größer) sowie Funktionen zum Laden von Bildern.

Interessanter wird dann die `INIT_INSTANCE`-Funktion:

```
sub INIT_INSTANCE {  
    my $self = shift;  
  
    @::black_img  
    or load_images;  
  
    $self->double_buffered (0);  
    $self->set (border_width => 0, shadow_type => 'none',  
              obey_child => 0, ratio => TRAD_RATIO);  
    $self->set(cursor_mask => MARK_B | MARK_W,  
              cursor_value => MARK_B | MARK_GRAYED);  
  
    $self->add ($self->{canvas} = new Gtk2::DrawingArea);  
  
    $self->{canvas}->signal_connect (  
        configure_event => sub { $self->configure_event ($_[1]) });  
    $self->{canvas}->signal_connect (  
        motion_notify_event => sub { $self->motion });  
    $self->{canvas}->signal_connect (  

```

```
        leave_notify_event => sub { $self->cursor (0) });
$self->{canvas}->signal_connect (
    button_press_event => sub { $self->button ("press", $_[1]) });
$self->{canvas}->signal_connect (
    button_release_event => sub { $self->button ("release", $_[1]) });

$self->{canvas}->set_events ([
    @{$self->{canvas}->get_events },
    'leave-notify-mask',
    'button-press-mask',
    'button-release-mask',
    'pointer-motion-mask',
    'pointer-motion-hint-mask'
]);
}
```

Hier wird eine `Gtk2::DrawingArea` eingebettet, die das eigentliche Bild darstellt. Eine `Gtk2::DrawingArea` ist eine Art leeres Widget, das selbst nichts zeichnet, aber an der Größenbelegung teilnimmt und eine hervorragende Grundlage für eigene Widgets ist.

Das Go-Widget verwendet einen Trick: Die Brett-Grafik befindet sich im X-Server (Windows: keine Ahnung) und wird automatisch vom Server gezeichnet, das Widget muss also keine `Expose`-Events behandeln, sondern "nur" zeichnen. Als Beispiel für ein Widget, das `Expose`-Events selbst behandelt, sei `Gtk2::CV::Schnauzer` genannt, das ein komplettes Widget inklusive `Size`-Allocation und `Expose` in Perl implementiert.

Als nächstes die `SET_PROPERTY`-Funktion:

```
sub SET_PROPERTY {
    my ($self, $pspec, $newval) = @_;

    $pspec = $pspec->get_name;

    $self->cursor (0) if $pspec =~ /^cursor/;
    $self->{$pspec} = $newval;
    $self->cursor (1) if $pspec =~ /^cursor/;
}
```

Sie macht das gleiche wie die normale `SET_PROPERTY`-Funktion, nur bei den `Cursor`-Attributen wird der `Cursor` neu gezeichnet (was die Funktion `cursor` erledigt).

Dann folgt der Handler für das `configure-notify-event`-Signal, das erzeugt wird, wenn sich Größe oder Position des Widgets ändern:

```
sub configure_event {
    my ($self, $event) = @_;
```

---

```

$self->{window} = $self->{canvas}->window;

my $drawable = $self->{window};

$drawable->set_back_pixmap (undef, 0);

$self->{idle} ||= add Glib::Idle sub {
    delete $self->{stack};

    $self->{width} = $self->{canvas}->allocation->width;
    $self->{height} = $self->{canvas}->allocation->height;
    $self->draw_background;

    $self->draw_board (delete $self->{board}, 0) if $self->{board};
    $self->{window}->clear_area (0, 0, $self->{width}, $self->{height});

    delete $self->{idle}; # Handler lief
    0; # Lösche Handler
};

1;
}

```

Sie löscht die Background-Pixmap, da sie nicht mehr gültig ist und neu gezeichnet wird. Da das initiale Zeichnen des Brettes clientseitig geschieht und das Zeichnen und der Transfer zum Server einige Zeit in Anspruch nimmt, geschieht es nicht synchron zum Signal, sondern später, in einem Idle-Handler. Dieser wird aufgerufen, wenn die Hauptschleife von Gtk2 alle anstehenden Ereignisse abgearbeitet hat.

Das Gdk-Fenster, das man mit `$self->{canvas}->window` erhält, kann erst hier abgefragt werden und nicht schon in `INIT_INSTANCE`, da zum Zeitpunkt von `INIT_INSTANCE` noch keine Gdk-Fenster für die Widgets erzeugt wurden sondern erst, nachdem sie das erste mal angezeigt wurden (bzw. etwas früher, wenn sie *realized* werden).

Im Idle-Handler werden noch andere Daten invalidiert (`$self->{stack}` speichert die vorberechneten Steine), die neue Größe abgefragt und der Bretthintergrund gezeichnet sowie die Steine gemalt und gesetzt.

Am Ende wird dem X-Server mit `clear_area` mitgeteilt, daß er die Grafik auffrischen soll.

Danach folgen einige hundert Zeilen Code, der die Grafik zeichnet (und dazu im wesentlichen `Gtk2::Gdk::Pixbuf` benutzt). Und dann:

```

sub do_button_press {
    my ($self, $button, $x, $y) = @_;
}

```

```
sub do_button_release {
    my ($self, $button, $x, $y) = @_;
}
```

Als der Code geschrieben wurde, gab es noch kein `class_closure`-Feld für Signale und Signale wurden immer als Perl-Methoden aufgerufen, mit einem vorangestellten `do_` ("weil Python das auch so macht" :).

Und abschließend noch die Methode, die die Mausklick-Signale auslöst:

```
sub button {
    my ($self, $type, $event) = @_;

    $self->motion;

    if ($self->{cursorpos}) {
        $self->signal_emit ("button-$type", $event->button, @{$self->{cursorpos}});
    }
}
```

`$self->motion` bewegt den Cursor, falls das notwendig ist, und, falls die Maus über einer gültigen Brettposition steht, wird ein Signal ausgelöst.

Wenn man so wollte, könnte man von C/C++/XS aus jetzt ein neues Go-Brett erzeugen:

```
GtkWidget *w = g_object_new (
    "Gtk2__GoBoard", # ':' wird zu '_'
    "size", 19
);
```

... und im wesentlichen so benutzen wie jedes andere Gtk2-Widget.

## Objekte aus anderen Sprachen nutzen

Obwohl es eine aufregende Idee wäre, Python-Widgets in Perl und umgekehrt zu benutzen, ist dies eher unrealistisch: man müßte zuerst Python und Perl im selben Prozess vereinigen, was wahrscheinlich möglich ist, aber kostspielig.

Deshalb beschränke ich mich auf Widgets/Objects aus C oder C++, die Vorgehensweise ist aber übertragbar auf andere Sprachen.

Dazu muß man zwei Schritte durchführen: Erstens die Bibliothek oder den Code in den Prozess linken, zweitens die `GTypes` beim Glib-Modul registrieren.

Am einfachsten (bzw. portabelsten) geschieht dies mit einem kleinen XS-Modul. Möchte man z.B. den `GimpColorButton` aus der `libgimpwidgets`, schreibt man ein kleines

---

Modul (oder benutzt `DynaLoader` und `Inline::C`), das man gegen die `libgimpwidgets` linkt.

In dessen `BOOT`-Section schreibt man:

```
BOOT:
    gperl_register_object (GIMP_TYPE_COLOR_BUTTON, "Gimp::ColorButton");
```

Damit ist die Verbindung zwischen C und Perl hergestellt, und man kann den `Gimp::ColorButton` sofort benutzen:

```
use MeinXSModul;

my $button = Gimp::ColorButton->Glib::Object::new (...);
```

Ähnlich verfährt man mit anderen Datentypen (z.B. Flags und Aufzählungen), die man registrieren möchte:

```
gperl_register_fundamental (GDK_TYPE_CAP_STYLE, "Gtk2::Gdk::CapStyle");
```

Sobald sie registriert sind, kann man sie wie gewohnt von Perl aus nutzen.

## Verweise

- <http://www.gtk.org/>  
Die Dokumentation zu Gtk+, Glib usw.
- `perldoc Glib`, `Glib::Object`, `Gtk2`, `Gtk2::<widgetname>`  
Das `Gtk2`-Modul hat für jedes Widget eine `perldoc`-Seite, die (kurz) die Hierarchiezugehörigkeit, Methoden, Signale und Properties auflistet.
- `apt-get install devhelp-book-gtk2 #` oder `devhelp-books`  
Installiert unter Debian *devhelp*, einen interaktiven Help-Browser und das `Gtk2`-Buch (dazu gehört auch `GObject`, `Glib`-Doku uvm.). Sehr hilfreich beim Programmieren.
- `Gtk2::PodViewer`, `Gtk2::GoBoard`, `Gtk2::CV::Schnauzer`  
Verschiedene `Gtk2`-Widgets in Perl.

## Danke

... an denjenigen, der `Latex` (a.k.a. "the bag of dirty broken hacks") endlich mal fixt.