

4 GTK+ – Grundlagen

4.1 Was ist GTK+?

GTK+ ist ein *Toolkit* zum Programmieren grafischer Benutzeroberflächen. In grauer Frühzeit des X-Fenstersystems war das einzige Toolkit, das brauchbar war und sich einiger Popularität erfreute, *Motif*. Als nun im August 1995 Spencer Kimball und Peter Mattis, zwei kalifornische Informatikstudenten, die Idee hatten, ein Bildbearbeitungsprogramm zu schreiben, lag es nahe, dafür Motif zu verwenden.

Nun war dieses Programm, das später als ›The GIMP‹ einige Berühmtheit erlangen sollte, freie Software. Motif war es nicht. Dies wurde zum Hindernis für seine Verbreitung, und so schrieben sich Kimball und Mattis irgendwann ihr eigenes Toolkit namens GTK – ›The GIMP Toolkit‹. Das sollte sich als eine ihrer besten Ideen herausstellen. GTK erblickte im Juli 1996 offiziell das Licht der Welt [8]. Anfangs bestand das Toolkit aus insgesamt drei Bibliotheken: GLib als Fundament, *GDK* als Schnittstelle zu X11 und GTK obendrauf.

Irgendwann wurde GTK dann objektorientiert: Widgets konnten von anderen abgeleitet werden, und das prinzipiell noch heute vorhandene Signalsystem wurde eingeführt. Die Entwickler fühlten sich berechtigt, GTK in GTK+ umzubenennen, um dies zu honorieren [14].

Mit der Version 2.0 wurde das Objektsystem herausgetrennt und als GObject ausgelagert. Auch wurde GTK+ plattformunabhängig, da GDK nun verschiedene Backends unterstützte. Zwei neue Komponenten traten hinzu: *Pango*, eine mächtige Bibliothek zur Textdarstellung, und *ATK*, das Subsystem für Barrierefreiheit. Spätestens seit dieser Version braucht GTK+ den Vergleich mit keinem anderen GUI-Toolkit zu scheuen.

GTK+ ist von jeher freie Software im Sinne der GNU LGPL, weswegen sich das GNOME-Projekt auch entschied, GTK+ als Toolkit zu verwenden. Dieses Kapitel bespricht die Grundlagen von GTK+, bevor es im Rest des Buches um komplexere Anwendungsentwicklung und die Integration ins Gesamtsystem GNOME geht. Bei allen Überlegungen werden die vom GNOME Usability Project erstellten Richtlini-

en zur Benutzeroberflächengestaltung von GNOME-Anwendungen [5] unmittelbar einbezogen.

GUI-Programmierung hat, ob sie nun mit GTK+ oder einem anderen Werkzeug geschieht, immer das gleiche Grundprinzip: *Widgets* werden als Objekte angelegt und mit Hilfe von Eigenschaften bearbeitet, dann in *Behältern* angeordnet, und Code (das heißt Callback-Funktionen) wird mit *Ereignissen* verknüpft, die den Widgets widerfahren können; das bedeutet in GTK+, Signalhandler an entsprechende Signale zu binden.

Hinweis

Beachten Sie bei den folgenden Ausführungen, dass ich versuche, möglichst wenige API-Funktionen zu besprechen, nämlich nur jene, die einen Zweck erfüllen, der nicht durch Bearbeiten von Eigenschaften erreichbar ist. GTK+ ist voller Zugriffsfunktionen, die nichts anderes tun, als bestimmte Eigenschaften von Objekten zu manipulieren, was ja auch einfach durch die GObject-API passieren kann.

4.1.1 Widgets und Behälter

Behälter sind Widgets, die ihrerseits andere Widgets enthalten können; sie sind für das Layout eines Programms zuständig. Die offensichtlichsten Behälter sind *Fenster* – in den meisten Fällen befinden sich alle Widgets eines Programms in Fenstern.

Andere Behälter sind zum Beispiel Box- oder Tabellenwidgets, die mehrere andere Widgets aufnehmen und in einer bestimmten Lage zueinander anordnen. Durch das Ineinanderschachteln von Behältern erhält ein Anwendungs- oder Dialogfenster Form und Gestalt. Ein Widget in einem Behälter platzieren nennt man *packen*, und ein in einen Behälter gepacktes Widget bezeichnet man als dessen *Kind*. Die Hierarchie von ineinandergepackten Behältern und Widgets bildet den *Widgetbaum*.

Hinweis

Kein Widget kann gleichzeitig in mehreren Behältern enthalten sein. Anders ausgedrückt: Kein Widget kann gleichzeitig Kind mehrerer Behälter sein.

Widgets existieren, wenn sie erstellt werden, zunächst im Verborgenen; erst auf explizites Verlangen werden sie angezeigt. Sie können bearbeitet und zusammengefügt werden, ohne dass sie auf dem Bildschirm zu sehen sind. Um zum Beispiel ein Fenster voller Widgets anzuzeigen, ist es keine gute Vorgehensweise, das Fenster darzustellen und dann die Widgets hineinzubasteln; besser ist es, das Fenster komplett im Verborgenen zusammenzubauen und dann mit einem Schlag darzustellen.

Es ist im Übrigen auch möglich, Widgets wieder zu verbergen, das heißt, sie vom Bildschirm zu holen, ohne dass sie aus dem Speicher gelöscht werden. Kandidaten hierfür sind zum Beispiel Werkzeugpaletten, Eigenschaftsfenster oder Teile von Dialogfenstern, die nur auf Wunsch dargestellt werden.

Wie Widgets genau aussehen, hängt vom aktiven GTK+-Thema ab. Ein Thema kann so ziemlich alles sein – von einem Satz neuer Farben bis hin zu einem Codemodul, das völlig neue Routinen zum Zeichnen der Widgets implementiert. In Abbildung 4-1ff. sehen Sie, wie die Themeneinstellung das Aussehen von Widgets beeinflussen kann.

Das Vorgabethema von GTK+ ist sehr schlicht, und es hat in den vergangenen Jahren einige Versuche gegeben, es zu ersetzen. Unabhängig davon haben mittlerweile die meisten GNOME-Distributoren ein anderes Thema vorgegeben; das sehr populäre Ubuntu benutzt beispielsweise »Human«. Die Bildschirmfotos in diesem Buch sind entsprechend der Empfehlungen für GNOME-Dokumentation mit dem Thema »Clearlooks« aufgenommen.

Hinweis



Abbildung 4-1
GTK+-Thema
»Clearlooks«

4.1.2 Ereignisgetriebene Programmierung

Im Gegensatz zu klassischen, »von oben nach unten« ihren Code abarbeitenden Programmen, wie sie auf der Befehlszeile üblich sind, sind grafische Programme generell eine Ansammlung von Widgets (und anderen Objekten), die auf eine Benutzeraktion oder ein sonstiges Ereignis hin kurze Codestücke ausführen, die meistens direkt auf andere Objekte einwirken.

Hierfür wurde der Begriff der *ereignisgetriebenen Programmierung* geprägt. Ereignisse wie Tastendrucke und Mausklicks werden dem Pro-

Abbildung 4-2
GTK+-Thema »ThinIce«

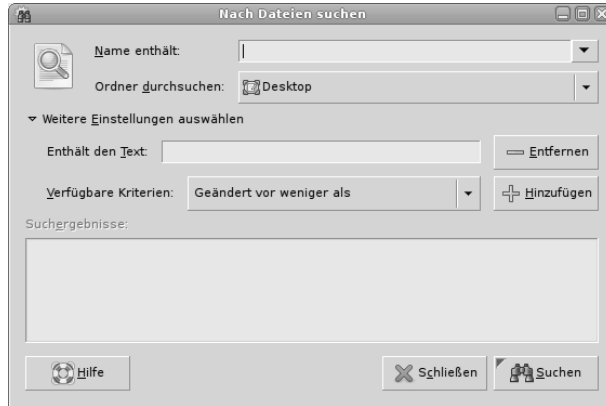
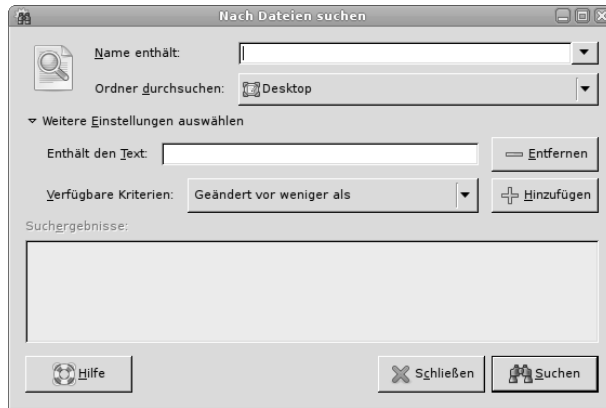


Abbildung 4-3
GTK+-Vorgabethema
»Raleigh«



gramm vom Fenstersystem überreicht und in der *Hauptschleife* verarbeitet: GTK+ ermittelt, welchem Widget das Ereignis widerfahren ist, und emittiert gegebenenfalls die entsprechenden Signale. Der Mechanismus, durch den dies geschieht, ist das in Abschnitt 3.5 besprochene Signalsystem GSignal.

Wichtig ist hier der Unterschied zwischen Ereignissen und Signalen. Ein Ereignis ist etwas von außen (das heißt durch die Hauptschleife) an ein Widget Herangetragenenes; ein Signal ist programmintern. Allerdings führt normalerweise jedes Ereignis auch zur Emission eines oder mehrerer Signale. Spezielle Signale, sogenannte Ereignissignale, die an der Endung `-event` zu erkennen sind, werden unmittelbar auf ein Ereignis hin emittiert; ihre Handler geben einen Wahrheitswert zurück: Ist dieser TRUE, wird die Signalemission abgebrochen.

Beachten Sie auch, dass Sie jede Änderung irgendeiner Objekteigenschaft als Auslöser für einen Signalhandler nutzen können, indem Sie das Signal `notify` mit dem Eigenschaftsnamen als Detail binden.

4.1.3 Ein einfaches Beispiel

GTK+ lebt hauptsächlich in einer Bibliothek, die je nach Backend einen unterschiedlichen Namen hat; auf X11 heißt sie *libgtk-x11-2.0*. Allerdings sind noch weitere Bibliotheken erforderlich, da GTK+ auf einer Reihe von Subsystemen beruht. Benutzen Sie `pkg-config` (siehe Abschnitt B), um Ihre Compilerbefehlszeile zusammenzustellen, und Sie sind auf der sicheren Seite. Der einzige Header, den Sie einbinden müssen, ist *gtk/gtk.h*, was auch alle Include-Dateien für die GLib, GObject und das restliche Zubehör anzieht.

gtk/gtk.h

In einem Buch über C-Programmierung zuallerst ein Programm zu schreiben, das »hello, world« ausgibt, ist Tradition, und Tradition ist es auch, Einführungen in GTK+ mit einem Programm ähnlich epischen Ausmaßes zu beginnen. Betrachten Sie *gtxhallo.c*.

gtxhallo.c

Zeichencodierung

Alle Zeichenketten, die Pango verarbeitet, müssen in UTF-8 codiert sein; ansonsten reichen die Ergebnisse von unschön bis katastrophal. Daher ist es sinnvoll, die Programmdateien gleich in dieser Codierung zu erstellen.

Warnung

Kompilierung

Kompilieren können Sie dieses Programm erwartungsgemäß folgendermaßen:

```
gcc -o gtxhallo gtxhallo.c `pkg-config gtk+-2.0 --cflags --libs`
```

Programmverhalten

Das Verhalten des Programms ist einfach zu beschreiben: Es wird ein Fenster geöffnet, in dem sich ein einziger, großer Knopf mit der Beschriftung »Hallo, Welt! Hier draufdrücken!« befindet. Klickt man diesen Knopf, wird »Hallo, Welt!« auf der Konsole ausgegeben und das Programm beendet. Zudem kann das Fenster auch über den *Fenstermanager*, das heißt also in der Regel über einen Knopf in der Titelleiste, geschlossen werden. Ein Bild des Ganzen sehen Sie in Abbildung 4-4.

Aufbauen der Anwendung

Der Aufruf von `gtk_init()` erledigt sämtliche Initialisierungsaufgaben für GTK+; diese Funktion ruft unter anderem `g_type_init()` für Sie auf. Sie erhält Zeiger auf `argc` und `argv`, um der Befehlszeile gewisse

gtk_init()

Abbildung 4-4Resultat von `gtkhallo.c`

Standardparameter zu entnehmen. Diese Parameter werden aus der Befehlszeile entfernt; falls Sie `argc` und `argv` selber weiterverarbeiten wollen, können Sie dies problemlos hinterher tun, ohne auf die GTK+-Optionen Rücksicht nehmen zu müssen.

Anschließend wird mit `g_object_new()` ein Objekt fenster vom Typ `GtkWindow` erzeugt. Den Eigenschaften »`default-width`«, »`default-height`«, »`border-width`« und »`title`« werden gleich Werte zugewiesen, um das Fenster anfänglich 200 Pixel lang und breit zu machen, ihm einen 12 Pixel breiten Rand und den Titel »`GtkHallo`« zu geben.

An die Signale »`delete-event`« und »`destroy`« des Fensters werden jeweils Handler (die Funktionen `delete_event()` und `ende()`) gebunden.

Nun wird ein Knopf `knopf` vom Typ `GtkButton` erzeugt; die Eigenschaft »`label`« und damit seine Beschriftung wird als »`Hallo, Welt!`« initialisiert, die Eigenschaft »`tooltip-text`« und damit seine Minihilfe als »`Hier draufdrücken`«. An das Signal »`clicked`« von `knopf` werden anschließend zwei Handler gebunden: zum einen die Funktion `hallo()` und zum andern die GTK+-Funktion `gtk_widget_destroy()`. Bei letzterer geschieht dies mittels `g_signal_connect_swapped()`, so dass sie beim Aufruf das per Datenzeiger übergebene Objekt `fenster` als Argument erhält.

Schließlich sind nur noch drei Dinge zu erledigen: Mit `gtk_container_add()` kann ein Widget in einen Behälter gepackt werden; das Programm stopft hier den Knopf `knopf` in den Behälter `fenster`. (Beachten Sie die entsprechenden Casts.) Dann wird das Fenster mit seinem sämtlichen Inhalt mittels `gtk_widget_show_all()` angezeigt. Und zu guter Letzt, jetzt, da das Programm fertig zur Benutzerinteraktion ist, wird mit `gtk_main()` die GTK+-Hauptschleife angeworfen.

Code binden

Mit den vorhin gebundenen Signalhandlern hat es folgende Bewandnis: `delete_event()` ist an das (fast) gleichnamige Signal¹ von `fenster` gebunden. Ein `GtkWindow` emittiert dann das Signal »`delete-event`«, wenn der Fenstermanager beantragt hat, das Fenster zu entfernen. Dieses Ereignissignal muss einen Wahrheitswert zurückgeben (siehe auch Abschnitt 4.1.2); nur wenn dieser `FALSE` ist, wird die Emission fortgesetzt und `fenster` zerstört, das heißt, das »`destroy`«-Signal durch `fenster` emittiert. Der Handler `ende()` ist an eben dieses Signal gebunden und hat keine andere Funktion, als die GTK+-Hauptschleife mittels `gtk_main_quit()` abubrechen und damit das Programm zu beenden.

Durch die Mimik mit dem »`delete-event`«-Signal können Sie entscheiden, ob ein Klick auf den Schließknopf des Fensters zum Abbruch führt oder nicht. Sie könnten im Handler dieses Signals zum Beispiel feststellen, ob noch geänderte Dateien offen sind, und beim Benutzer anfragen, ob er das Programm wirklich beenden will. Zudem stellt GTK+ die Funktion `gtk_widget_hide_on_delete()` zur Verfügung, die dazu gedacht ist, als Handler für das Signal »`delete-event`« gebunden zu werden. In diesem Fall führt ein Klick auf den Schließknopf dazu, dass das Fenster nur verborgen, nicht jedoch zerstört wird. Für Werkzeugpaletten und ähnliche Fenster kann dies sinnvoll sein.

Hinweis

Der Handler `hallo()` ist an das Signal »`clicked`« von `knopf` gebunden und wird naheliegenderweise dann ausgeführt, wenn der Knopf geklickt wurde. Er gibt »`Hallo, Welt!`« auf der Konsole aus und sonst nichts. Interessant ist dann die Verwendung von `gtk_widget_destroy()` mit dem Argument `fenster` als zweitem Handler für »`clicked`«: Bei Signalemission führt dies zur Zerstörung des Programmfensters.

Mittelbar bedeutet dies natürlich auch die Emission von »`destroy`« durch `fenster` und den Aufruf von `ende()`. Trotzdem wäre es unsauber gewesen, `ende()` direkt an das Signal »`clicked`« des Knopfes zu binden, denn ein GUI-Programm sollte sich immer erst verabschieden, wenn es alle seine Widgets ordnungsgemäß entsorgt hat. Somit werden keine »`destroy`«-Handler übergangen, und es bleibt kein Müll am Strand zurück.

Hinweis

Hoffentlich haben Sie das Beispielprogramm und die Erläuterungen dazu verstanden und sind nun mit mir einer Meinung, dass GTK+-

¹Sie können in Signalnamen auch Unterstriche statt Bindestrichen benutzen.

Programmierung eine recht saubere Sache ist, bei der mit klaren Konzepten hantiert wird. Alles weitere ist eigentlich nur noch Detailwissen.

4.2 Widgets: Grundlagen

GtkWidget Alle GTK+-Widgets sind Objekte der Klasse **GtkWidget** und damit auch Objekte der Klasse **GtkObject**, von der alle GTK+-Klassen abstammen.

Das objekttechnisch Besondere an **GtkObjects** und damit auch **Widgets** ist, dass ihr Referenzzähler etwas anders funktioniert als der normaler **GObjects**. Legen Sie ein **Widget** an, dann hat es keine normale, sondern eine sogenannte *schwebende Referenz*, die beim Packen des **Widgets** vom Behälterwidget übernommen wird. Dies stellt sicher, dass beim Zerstören des Behälters auch das **Widget** abgeräumt wird. (Schwebende Referenzen werden in der G-Welt übrigens realisiert durch Ableiten der Klasse von **GInitiallyUnowned**, einer Unterklasse von **GObject**.)

GInitiallyUnowned

Hinweis

Das Prinzip der schwebenden Referenz sorgt auch dafür, dass Sie beispielsweise den Aufruf einer Funktion, die ein **GtkWidget*** zurückgibt, gefahrlos in die Parameterliste eines Funktionsaufrufs, der ein solches entgegennimmt, einschachteln können. Obwohl Sie hierbei keinen Zeiger auf das neue **Widget** zurückbehalten, entsteht kein Speicherleck, da spätestens, wenn das **Widget** irgendwo gepackt wird, der Behälter die schwebende Referenz übernimmt.

Vergleichen Sie hiermit den Umgang mit anderen dynamisch angelegten Strukturen wie zum Beispiel Zeichenketten, auf die immer irgendwo ein Zeiger zurückbehalten werden muss, damit sie nach Ende ihrer Lebensdauer freigegeben werden können.

Die Klasse **GtkWidget** hat als Basisklasse aller sichtbaren Bedienelemente in GTK+ eine Menge Methoden, Eigenschaften und Signale. Ich möchte hier nur die wichtigsten besprechen.

gtk_widget_show()*

Zum Anzeigen vorgemerkt wird ein **Widget** mit **gtk_widget_show()**. Meistens werden Sie aber eine andere Funktion benutzen wollen, nämlich **gtk_widget_show_all()**, was auch alle Kinder des übergebenen **Widgets** vormerkt. Wirklich angezeigt wird ein **Widget** nur dann, wenn nicht nur es selber, sondern auch alle seine Vorfahren im **Widgetbaum** sichtbar sind. Dies muss übrigens nicht sofort geschehen, sondern wird von der Hauptschleife dann erledigt, wenn Zeit dafür ist.

Wenn es Ihnen um diese Hundertstel- oder Zehntelsekunden wirklich geht, verwenden Sie **gtk_widget_show_now()**; diese Funktion macht ein **Widget** nicht nur sichtbar, sondern kehrt erst dann vom Aufruf zurück, wenn es wirklich dargestellt ist.

In dieser Zeit läuft die Hauptschleife allerdings weiter, und es werden Ereignisse verteilt und Signale emittiert.

Warnung

Das Verbergen von Widgets funktioniert analog zu `gtk_widget_show()` mit `gtk_widget_hide()`.

Vernichtet wird ein Widget zu guter Letzt mit `gtk_widget_destroy()`, was Sie auch aus den Beispielen kennen dürften.

Eine weitere noch zu nennende nützliche Funktion ist `gtk_widget_activate()`, die ein `GtkWidget` als einziges Argument entgegennimmt. Sie versucht, das übergebene Widget zu **aktivieren** (zu drücken, auszuwählen oder Vergleichbares). Der Rückgabewert dieser Funktion hat den Typ `gboolean`; `FALSE` wird dann zurückgegeben, wenn das Widget, das zu aktivieren versucht wurde, gar nicht aktivierbar ist.

Ein `GtkWidget` hat unter anderem folgende Eigenschaften (soweit nicht anders erwähnt, vom Typ `gboolean`):

- »**visible**« Ist dies `TRUE`, ist das Widget sichtbar.
- »**sensitive**« Ist dies `FALSE`, ist das Widget inaktiv (grau geschaltet), das heißt, es reagiert nicht auf Eingaben und wird als inaktiv dargestellt, das heißt normalerweise in Grau.
- »**can-focus**« Ist dies `TRUE`, ist das Widget in der Lage, den Eingabefokus zu erhalten.
- »**has-focus**« Ist dies `TRUE`, hat das Widget den Eingabefokus.
- »**can-default**« Ist dies `TRUE`, ist das Widget in der Lage, das Vorgabewidget zu sein.
- »**has-default**« Ist dies `TRUE`, ist das Widget das Vorgabewidget.
- »**receives-default**« Ist dies `TRUE`, erhält das Widget die Vorgabeaktion, wenn es den Fokus hat.
- »**tooltip-text**« (Typ `GCharArray`) Eine Minihilfe für das Widget, die angezeigt wird, wenn man den Mauszeiger längere Zeit darüberhält. Minihilfen sollten nur für Widgets eingerichtet werden, die auch tatsächlich angeklickt oder anderweitig bedient werden können.
- »**tooltip-markup**« Dito, allerdings nicht mit einfachem Text, sondern mit Pango-Markup (siehe Abschnitt 4.4.1)

Als Vorgabewidget sollte jenes Widget in einem Fenster markiert werden, das zu aktivieren am sinnvollsten beziehungsweise am ungefährlichsten ist. Bei Dialogfenstern ist dies normalerweise der Knopf ganz unten rechts. Sinn des Ganzen ist unter anderem bessere Tastaturbedienung: Das Vorgabewidget wird zum Beispiel meistens dann aktiviert, wenn man in einem Fenster die Eingabetaste drückt.

```
gtk_widget_hide()
gtk_widget_
destroy()
gtk_widget_
activate()
```

GtkWidget emittiert zahlreiche Signale; die allermeisten davon werden nur dann gebraucht, wenn Sie ein neues Widget von **GtkWidget** ableiten. Von alltäglichem Nutzen sind die folgenden Signale:

- »**delete-event**« (Prototyp `gboolean handler(GtkWidget *widget, GdkEvent *ereignis, gpointer daten)`) Dieses Signal ist vor allem bei Fenstern wichtig; es wird emittiert, wenn ein Widget vom Fenstermanager ein Löschereignis erhält. Wie jedes Ereignissignal reicht es an den Handler ein `GdkEvent` durch, was Sie getrost ignorieren können, und hat den Rückgabetypen `gboolean`, damit Sie durch Rückgabe von `TRUE` die Signalemission abbrechen können.
- »**show**« Wird emittiert, wenn ein Widget angezeigt wird
- »**hide**« Wird emittiert, wenn ein Widget verborgen wird

Hinweis

Hier und in Zukunft werden Signalprototypen nur dann erwähnt, wenn sie von der üblichen Standardform `void handler(typ *widget, gpointer data)` abweichen.

4.3 Fenster

GtkWindow

Im Beispielprogramm in *gtkhallo.c* kommt ein Fenster vor; Sie können sehen, dass es sich um ein Behälterwidget der Klasse **GtkWindow** handelt.

Die Eigenschaft »**title**« enthält den Titel des Fensters als Zeichenkette.

Hinweis

Fenstertitel sollten dem Benutzer dazu dienen, Fenster in der Fensterliste möglichst leicht voneinander zu unterscheiden. Idealerweise sollte bei Anwendungsfenstern ein Fenstertitel aus dem Namen des Dokuments bestehen, das im Fenster angezeigt wird, also aus einem Dateinamen, einem Verzeichnisnamen oder Ähnlichem. Nur, falls ansonsten zwei Fenster der Anwendung denselben Titel hätten, kann der Dateiname zum Beispiel durch einen Pfad ergänzt werden. Neue Dokumente sollten einen Titel wie »Ungespeichertes Dokument« erhalten; ein Fenster, das noch nicht gespeicherte Änderungen enthält, sollte seinem Titel ein Sternchen voranstellen. All dies können Sie bei regelkonformen GNOME-Anwendungen gut beobachten. Nicht direkt inhaltsrelevante Informationen wie Anwendungsname oder Programmversionen sind in einem Fenstertitel fehl am Platze.

Bei Fenstern, die kein irgendwie geartetes Dokument anzeigen, sind die Namenskonventionen entsprechend anders:

Anwendungsfenster *Name der Anwendung* (»Lavalampe«)

Eigenschaftsfenster *Name des Objekts* – Eigenschaften (»Tabelle 3.1 – Eigenschaften«)

Einstellungsfenster *Name der Anwendung* – Einstellungen (»Wunder-Text – Einstellungen«)

Warnungen Kein Titel. Fenster mit Warnungen sollen ihre Information durch den kurzen Text in ihrem Inneren übertragen; dieselbe Information noch einmal in den Titel zu stellen, wäre doppelt gemoppelt und sieht verwirrend aus. Dies ist jedenfalls die Meinung der Richtlinie. *Meiner* Meinung nach schadet ein kurzer Fenstertitel wie »Warnung« oder »Information« niemandem, vor allem, da titellose Fenster in der Fensterliste als »namenloses Fenster« oder dergleichen erscheinen – und das verwirrt nun erst recht. In Abschnitt 4.11 finden Sie mehr zu Dialogfenstern und auch zu den in gewissen Fällen von GTK+ automatisch vergebenen Titeln.

Assistenten *Name des Assistenten* (»Kaffeemaschine konfigurieren«; zu Assistenten siehe Abschnitt 5.6)

Weitere wichtige Eigenschaften von **GtkWindow**:

»**resizable**« (Typ `gboolean`) Wenn dies `TRUE` ist, ist es möglich, die Größe des Fensters zu ändern.

Es ist fast nie eine gute Idee, `FALSE` in diese Eigenschaft einzutragen und es dem Benutzer damit unmöglich zu machen, sich das Fenster selbst zurechtzulegen. Falls das Behälterlayout in einem Fenster richtig konfiguriert ist, bleibt der Inhalt auch bei starkem Vergrößern noch sauber angeordnet und bedienbar.

Hinweis

»**modal**« (Typ `gboolean`) Ist dies `TRUE`, ist das Fenster *modal*, das heißt, das Fenster, dem es als Dialog zugeordnet ist, ist stillgelegt, während dieses existiert.

Modale Fenster sind heutzutage in den meisten Fällen völlig vermeidbar. Sie sollten versuchen, ohne modale Fenster auszukommen, da es beispielsweise den Benutzer verwirren kann, wenn ein Fenster die Reaktion auf seine Eingaben verweigert, weil irgendwo noch ein verdeckter Dialog offen ist.

Warnung

»**window-position**« Ein Aufzählungswert, der die Fensterplatzierung bestimmt; mögliche Werte sind die folgenden:

GTK_WIN_POS_NONE Der Fenstermanager platziert das Fenster nach seinen eigenen Vorstellungen.

GTK_WIN_POS_CENTER_ON_PARENT Zentriert über dem Elternfenster.

GTK_WIN_POS_CENTER Möglichst zentriert in der Bildschirmmitte.

GTK_WIN_POS_CENTER_ALWAYS Zentriert, und zwar so, dass das Fenster auch bei Größenänderungen zentriert verbleibt.

GTK_WIN_POS_MOUSE Möglichst nahe dem Mauszeiger

All dies funktioniert nur mit einem entsprechend kooperierenden Fenstermanager, das heißt einem, der die standardisierten **NET_WM-Hints** [37] versteht.

»**default-width**« (Typ `gint`) Die Vorgabebreite, wie im Beispiel schon gesehen

»**default-height**« (Typ `gint`) Entsprechend: Die Vorgabehöhe.

Hinweis

Normalerweise sollten Sie den Fensterinhalt die Größe eines Fensters bestimmen lassen. Wenn Sie selber eine Fenstergröße festlegen, die nicht von äußeren Zwängen bestimmt wird, wählen Sie angenehme Abmessungen, zum Beispiel ein Seitenverhältnis von etwa 1 zu 1,6 (eine Annäherung an den Goldenen Schnitt, ein in Kunst und Typographie sehr beliebtes Teilungsverhältnis).

»**destroy-with-parent**« (Typ `gboolean`) Ist dies `TRUE`, wird das Fenster mit abgeräumt, wenn sein Elternfenster zerstört wird.

»**decorated**« (Typ `gboolean`) Ist dies entgegen der Vorgabe `FALSE`, wird das Fenster vom Fenstermanager nicht dekoriert, also nicht mit Rahmen und Titelleiste versehen.

Hinweis

Es gibt nur sehr wenige Fälle, in denen dies sinnvoll ist; in erster Linie sind dies Werkzeuge, die permanent angezeigt werden und möglich wenig Platz wegnehmen sollen.

»**icon-name**« (Typ `gchararray`) Der Name des Symbols für das Fenster. Er wird verwendet, um das entsprechende Symbol im aktuellen Thema aufzufinden (siehe Abschnitt 4.4.2).

»**icon**« Ein `GdkPixbuf` (siehe auch Abschnitt 4.4.2), der das Symbol für das Fenster enthält.

4.4 Anzeigeelemente

Die einfachsten Widgets sind jene, die einfach nur etwas darstellen. GTK+ hat drei davon, zur Darstellung von Text, Bildern und Abläufen:

Beschriftungen Eine Beschriftung ist, wie der Name schon sagt, ein (meist kurzer) Text, der dazu dient, Text an oder neben Widgets

anzubringen. Pango sei Dank können Beschriftungen auch sehr leicht mit Formatierungen versehen werden. Zum Anzeigen größerer Textmengen sind Beschriftungen nicht geeignet; verwenden Sie hierzu einen Textpuffer und eine Textansicht (siehe Abschnitt 4.13).

Bilder Das Gegenstück zur Beschriftung: das Bild. Ein Widget, das in Zusammenarbeit mit GdkPixbuf, der GDK-Schnittstelle zu Bilddateien, Bilder so ziemlich jeder Art darstellen kann.

Fortschrittsbalken Letztes der reinen Anzeigeelemente ist der Fortschrittsbalken. Er kann entweder veranschaulichen, zu welchem Anteil ein bestimmter Vorgang abgeschlossen ist, oder er kann signalisieren, dass sich irgendetwas abspielt, dessen Endzeitpunkt aber noch nicht abzusehen ist. Letzteres nennt sich ›Aktivitätsmodus‹.

Das Programm *anzeige.c* demonstriert diverse Anzeigeelemente. Das Bildschirmfoto in 4-5 ist auch dementsprechend groß.

anzeige.c



Abbildung 4-5
Resultat von *anzeige.c*

4.4.1 Beschriftungen und Pango-Markup

GtkLabel Um eine Beschriftung anzulegen, muss das `GtkLabel`-Widget einfach nur den Text in seiner »`label`«-Eigenschaft erhalten.

Ein anderer Fall ist da schon `schrift_markup`. Hier soll der Text der Beschriftung formatiert sein – ziemlich wild formatiert sogar. Hierzu wird die Eigenschaft »`use-markup`« auf `TRUE` gesetzt, worauf der übergebene Text in einer *Markup-Sprache* aufgefasst wird.

Diese Sprache kommt mit wenigen Tags aus. Das wichtigste ist ` ... `. Es kann beliebigen Text einschließen und auch verschachtelt werden; an sich hat es keine Wirkung und ist deswegen nur im Zusammenhang mit mindestens einem der folgenden Attribute sinnvoll:

`font_desc=` **font_desc=** Schließt Text ein, der nach der angegebenen *Schriftbeschreibung* formatiert wird. Eine solche vollständige Beschreibung ist beispielsweise »Times Italic 12«. Andere Attribute des Tags haben Vorrang vor dem mit `font_desc=` Angegebenen. Schriftbeschreibungen können Sie aus jedem GTK+-Schriftwähler ersehen.

`font_family=` **font_family=** Wählt eine Schrift, zum Beispiel »Times« oder »Sans«

`face=` **face=** Funktionsgleich mit `font_family=`.

`size=` **size=** Regelt die Schriftgröße. Es gibt mehrere mögliche Werte:

Ein **Zahlenwert** Entspricht der absoluten Größe in 1024stel von Punkten, normale Schriftgröße also einem Wert um 10240.²

xx-small Extrem klein

x-small Sehr klein

small Klein

medium Mittel

large Groß

x-large Sehr groß

xx-large Extrem groß

smaller Kleiner als der umschließende Text

larger Größer als der umschließende Text

`style=` **style=** Regelt die Schriftlage; möglich sind die Schlüsselwörter `normal`, `oblique` (geschrägt) und `italic` (kursiv). Nicht alle Schriften bieten schräge Schnitte; wenn sie es tun, dann sind diese entweder alle geschrägt oder alle kursiv; beides in einer Schrift ist selten.

²Pango verwendet ein Maßsystem, das Einheiten (bei Schriftangaben stets Punkt) in eine Anzahl Untereinheiten teilt, die im globalen Makro `PANGO_SCALE` festgeschrieben ist. Der derzeitige Wert 1024 ist also nicht in Stein gemeißelt.

weight= Regelt die Schriftstärke; mögliche Werte sind: *weight=*

ultralight extraleicht

light mager

normal Buch

bold fett

ultrabold ultrafett

heavy heavy

Ein Zahlenwert gibt die genaue Abstufung der Schriftstärke an; zur Orientierung: Ultraleicht ist Stärke 200, Buch Stärke 400, ultrafett Stärke 800 und heavy Stärke 900.

Die wenigsten Schriften bieten wirklich alle diese Stärken.

variant= Wählt die Schriftvariante, entweder `normal` oder `smallcaps` für Kapitälchen. *variant=*

stretch= Regelt die Laufweite, das heißt den horizontalen Zeichenabstand; mögliche Werte sind: *stretch=*

ultracondensed ultrakompres; die Zeichen kleben fast aneinander

extracondensed extrakompres

condensed kompres

semicondensed halbkompres

normal

semiexpanded halbsplendid

expanded splendid

extraexpanded extrasplendid

ultraexpanded ultrasplendid; die Zeichen stehen mit extrem großem Abstand

Auch hier gilt: Nicht alle Schriften bieten all diese Laufweiten an.

background= Wählt die Hintergrundfarbe des Texts. Dies kann eine übliche RGB-Farbsequenz wie "#AACC40" sein oder ein X11-Farbname wie "midnightblue". Die komplette Liste von Farbnamen finden Sie auf einem Unix-System in `/usr/X11R6/lib/X11/rgb.txt`.³ *background=*

foreground= Wählt analog zu `background=` die Textfarbe *foreground=*

underline= Wählt die Unterstreichung des Texts, mögliche Werte sind: *underline=*

single einfach

double doppelt

³Eine angenehme Lektüre – unter den X11-Farbnamen sind so poetische wie `honeydew`, `rosy brown` oder `medium spring green`.

low tief (unterhalb der Unterlängen)

none gar nicht

rise= **rise=** Bestimmt den Grundlinienversatz (den Abstand des Texts zur Grundlinie der Zeile; kann auch negativ sein) in 1024steln von Punkten.

strikethrough= **strikethrough=** Dieses Attribut erlaubt nur die Werte true oder false; ist es true, wird der eingeschlossene Text durchgestrichen.

lang= **lang=** Gibt an, in welcher Sprache der eingeschlossene Text gehalten ist. Der Wert ist ein Sprachcode nach RFC 3066 und ISO 639, also beispielsweise de für Deutsch, fy für Friesisch oder i-klingon für Klingonisch. Genutzt wird dieser Wert vor allem, um bei nicht westeuropäischen Sprachen, bei denen Zeichen-, Wort- und Zeilengrenzen anders definiert sind, die entsprechenden Umbruchverfahren auszuwählen.

Weiterhin gibt es einige Kurzformen von Tags:

` ... ` ** ... ** Kurzform für ` ... `

`<big> ... </big>` **<big> ... </big>** Kurzform für ` ... `

`<i> ... </i>` **<i> ... </i>** Kurzform für ` ... `

`<s> ... </s>` **<s> ... </s>** Kurzform für ` ... `

`_{...}` **_{...}** Schließt tiefgestellten, kleinen Text ein (Indizes); entspricht ` ... `.

`^{...}` **^{...}** Schließt hochgestellten, kleinen Text ein (Exponenten); entspricht ` ... `.

`<small> ... </small>` **<small> ... </small>** Kurzform für ` ... `

`<tt> ... </tt>` **<tt> ... </tt>** Schließt Text in dicktengleicher (nicht proportionaler) Schrift (Teletype) ein.

`<u> ... </u>` **<u> ... </u>** Kurzform für ` ... `

Hinweis

Trotz all dieser Möglichkeiten sollten Sie es mit dem Formatieren nicht übertreiben. Verwenden Sie fette oder kursive Darstellung für Hervorhebungen, wenn nötig; benutzen Sie relative Schriftgrößeneinstellungen, um Überschriften oder Warnhinweise größer darzustellen (siehe auch die Richtlinie für Dialogfenster: Abschnitt 4.11); aber verwenden Sie nach Möglichkeit keine absoluten Schriftgrößenangaben und werfen Sie nicht mit verschiedenen Schriften um sich.

Zum Schluss seien noch die restlichen wichtigen Eigenschaften von `GtkLabel` genannt:

- »**use-underline**« (Typ `gboolean`) Ist dies `TRUE`, kann ein Unterstrich verwendet werden, um im Text das nächste Zeichen als Abkürzungsbuchstabe zu markieren.
- »**mnemonic-widget**« (Typ `GtkWidget`) Das Widget, das aktiviert werden soll, wenn der Abkürzungsbuchstabe der Beschriftung gedrückt wird.
- »**justify**« Dies bestimmt die Ausrichtung der Zeilen der Beschriftung gegeneinander. Mögliche Werte sind:
 - GTK_JUSTIFY_LEFT** linksbündig
 - GTK_JUSTIFY_RIGHT** rechtsbündig
 - GTK_JUSTIFY_CENTER** zentriert
 - GTK_JUSTIFY_FILL** Blocksatz
- »**wrap**« (Typ `gboolean`) Ist dies `TRUE`, werden zu lange Zeilen umgebrochen.
- »**wrap-mode**« Bestimmt die Art des etwaigen Umbruchs; mögliche Werte sind:
 - PANGO_WRAP_WORD** wortweise (Vorgabe)
 - PANGO_WRAP_CHAR** zeichenweise
 - PANGO_WRAP_WORD_CHAR** wortweise, solange möglich, zeichenweise, wenn ein Wort zu lang für die Zeile ist
- »**single-line-mode**« (Typ `gboolean`) Ist dies `TRUE`, wird unabhängig von der Länge des Texts immer höchstens eine Zeile dargestellt.
- »**ellipsize**« Ob und wie eine Beschriftung bei Überlänge durch Auslassungspunkte gekürzt werden soll; mögliche Werte sind:
 - PANGO_ELLIPSIZE_NONE** gar nicht (Vorgabe)
 - PANGO_ELLIPSIZE_START** am Anfang
 - PANGO_ELLIPSIZE_MIDDLE** in der Mitte
 - PANGO_ELLIPSIZE_END** am Ende

Ist Kürzen durch Auslassungspunkte eingeschaltet, beansprucht eine Beschriftung, der Sie auf andere Weise keine Breite vorgeben, nur die Breite von drei Punkten. Achten Sie also darauf, dass solche Beschriftungen durch andere Eigenschaften oder durch Packoptionen genügend Raum erhalten.

Hinweis

- »**max-width-chars**« (Typ `gint`) Die maximale Breite in Zeichen. Die Vorgabe `-1` bedeutet automatische Berechnung.
- »**width-chars**« Dito, bestimmt allerdings nicht die maximale, sondern die aktuelle Breite, unabhängig von der wirklichen Länge des Texts. Die Werte `0` bis `2` werden als `3` interpretiert.

- »**selectable**« (Typ `gboolean`) Ist dies `TRUE`, kann der Text der Beschriftung mit der Maus markiert werden.
- »**cursor-position**« (Typ `gint`, schreibgeschützt) Die Position des Anfangs der Markierung, in Zeichen vom Anfang der Beschriftung her gezählt
- »**selection-bound**« (Typ `gint`, schreibgeschützt) Die Position des Endes der Markierung, in Zeichen von ihrem Anfang her gezählt
- »**angle**« (Typ `gdouble`) Der Winkel des Texts zur Horizontalen in Grad gegen den Uhrzeigersinn (0,0 bis 360,0). Ein anderer Wert als die Vorgabe 0,0 hat nur eine Wirkung, wenn »**selectable**« und »**wrap**« `FALSE` sind sowie »**ellipsize**« gleich `PANGO_ELLIPSIZE_NONE` ist.

Verwenden Sie Beschriftungen in möglichst einheitlicher und eindeutiger Weise, um die Bedienelemente Ihrer Anwendung zu beschriften. Dabei sollten Sie folgende Grundsätze beachten:

- Normalerweise sollten Beschriftungen linksbündig sein.
- Der in einer Beschriftung festgehaltene Name für ein Bedienelement sollte aussagekräftig genug sein, damit zum Beispiel ein blinder Benutzer, der diesen Namen über seinen Bildschirmvorleser hört, sich etwas darunter vorstellen kann, ohne den gesamten Rest eines Dialogfensters gehört zu haben.
- Bei großen Symbolen und sonstigen Bildern sollte eine etwaige Beschriftung unterhalb der Grafik stehen (analog zu Bildunterschriften in Printmedien).
- Bei kleinen Symbolen sollte die Beschriftung direkt rechts anschließen (das Symbol fungiert als ‘Aufzählungspunkt’).
- Bei größeren Bedienelementen wie Listen oder Knopfgruppen sollte die Beschriftung oberhalb stehen (analog zu Überschriften über Text).
- Kleinere Bedienelemente wie Zahlenfelder, kleine Textfelder und Ähnliches sollten direkt rechts an eine Beschriftung anschließen.
- Beschriftungen, die in der Leserichtung vor (das heißt für die meisten Europäer: über oder links von) dem Bedienelement stehen, das sie bezeichnen, sollten in einem Doppelpunkt enden, um die Beziehung zu diesem Element herzustellen.
- Es sollte nicht vorkommen, dass mehrere Elemente im selben Fenster mit gleichen oder sehr gleichklingenden Namen beschriftet sind.

4.4.2 Bilder und Pixbufs

Wie Sie aus dem Beispiel in *anzeige.c* ersehen, ist es simpel, ein Bild darzustellen, das in einer Datei vorliegt. Es reicht, das Widget (Klasse

GtkImage) anzulegen und dabei in der Eigenschaft »file« den Pfad einer Bilddatei zu übergeben.

GtkImage

Verarbeitung und Anzeige des Bildes geschehen über ein **GdkPixbuf**-Objekt, GDKs grundlegende Datenstruktur für Bilddaten, die mit nahezu allen Grafikformaten umgehen kann.

Sie können einem Bild auch einen **GdkPixbuf** direkt zuweisen oder ein Bild aus dem Repertoire entnehmen (siehe Abschnitt 4.4.2). All dies passiert über folgende Eigenschaften:

»**pixbuf**« (Typ **GdkPixbuf**) Gegebenenfalls der im Bild enthaltene **Pixbuf**
 »**file**« (Typ **gchararray**, nicht auslesbar) Eine Datei, aus der das Bild erstellt werden soll

»**stock-id**« (Typ **gchararray**) Gegebenenfalls die Repertoire-Kennung des anzuzeigenden Bildes

»**icon-size**« Die Größe des aus dem Repertoire oder Symbolthema entnommenen Bildes; mögliche Werte sind:

GTK_ICON_SIZE_MENU für Menüeinträge (16x16 Pixel)

GTK_ICON_SIZE_SMALL_TOOLBAR für Einträge in schmalen Werkzeugleisten (18x18 Pixel)

GTK_ICON_SIZE_LARGE_TOOLBAR für Einträge in breiten Werkzeugleisten (24x24 Pixel)

GTK_ICON_SIZE_BUTTON für Symbole auf Knöpfen (20x20 Pixel)

GTK_ICON_SIZE_DND für Symbole, die Drag-and-Drop-Objekte darstellen (32x32 Pixel)

GTK_ICON_SIZE_DIALOG für Dialogfenster-Symbole (48x48 Pixel)

»**icon-name**« (Typ **gchararray**) Gegebenenfalls der standardisierte Name des Symbols im Symbolthema

»**storage-type**« (schreibgeschützt) Die Art des enthaltenen Bildes, mögliche Werte sind unter anderem:

GTK_IMAGE_EMPTY Das Bild ist noch leer.

GTK_IMAGE_PIXBUF **Pixbuf** oder aus einer Datei entnommenes Bild

GTK_IMAGE_STOCK Das Bild stammt aus dem Repertoire.

GTK_IMAGE_ICON_NAME Das Bild wurde per Name aus dem Symbolthema entnommen.

GdkPixbuf

GtkImage benutzt **GdkPixbuf**, um Bilder darzustellen. Falls Sie ein Bild nicht einfach nur auf den Bildschirm bringen, sondern verändern wollen, können Sie ihm einen **GdkPixbuf** entnehmen und bearbeiten.

GdkPixbuf

Hinweis

Ganz gleich, wie Sie einem `GtkImage` seinen Inhalt zugewiesen haben – es speichert die Bilddaten intern. Das heißt zum Beispiel, dass das Bild sich nicht verändert, wenn Sie den `GdkPixbuf`, den Sie beim Erstellen hineingeschrieben haben, bearbeiten. Sie müssen den bearbeiteten `GdkPixbuf` nochmals in das Bild eintragen, wie dies beispielsweise die Funktion `bild_aktualisieren()` in `pixbufdemo.c` tut.

Folgende wichtige Funktionen arbeiten auf `GdkPixbufs`:

<code>gdk_pixbuf_new_from_file()</code>	GdkPixbuf* gdk_pixbuf_new_from_file(const gchar *dateiname, GError **fehler) Diese Funktion lädt die durch <i>dateiname</i> bezeichnete Datei in einen Pixbuf und gibt diesen zurück. Falls etwas dabei schiefgeht, wird <code>NULL</code> zurückgegeben, und der <code>GError fehler</code> dient dazu, Fehlerinformationen zurückzugeben (Fehlerklasse entweder <code>GDK_PIXBUF_ERROR</code> oder <code>G_FILE_ERROR</code>).
<code>gdk_pixbuf_new_from_file_at_size(scale())</code>	GdkPixbuf* gdk_pixbuf_new_from_file_at_size(dateiname, int breite, int hoehe, fehler) Dito, skaliert das Bild aber bereits beim Laden auf die Abmessungen <i>hoehe</i> und <i>breite</i>
<code>gdk_pixbuf_get_width/height()</code>	GdkPixbuf* gdk_pixbuf_new_from_file_at_scale(dateiname, breite, hoehe, gboolean proportional, fehler) Dito, behält aber, sofern <i>proportional</i> auf <code>TRUE</code> gesetzt ist, das Seitenverhältnis des Bildes bei. Es ist daher nicht garantiert, dass der Pixbuf nachher genau die angegebenen Abmessungen hat. int gdk_pixbuf_get_width(const GdkPixbuf *pixbuf) Gibt die Breite von <i>pixbuf</i> in Pixel zurück int gdk_pixbuf_get_height(pixbuf) Gibt die Höhe von <i>pixbuf</i> in Pixel zurück
<code>gdk_pixbuf_copy()</code>	GdkPixbuf* gdk_pixbuf_copy(pixbuf) Gibt eine Kopie von <i>pixbuf</i> zurück, also nicht nur eine neue Referenz, sondern einen Zeiger auf eine Kopie der Pixeldaten
<code>gdk_pixbuf_flip()</code>	GdkPixbuf* gdk_pixbuf_flip(pixbuf, gboolean horizontal) Spiegelt <i>pixbuf</i> in der Horizontalen, sofern <i>horizontal</i> <code>TRUE</code> ist, ansonsten in der Vertikalen.
<code>gdk_pixbuf_rotate_simple()</code>	GdkPixbuf* gdk_pixbuf_rotate_simple(pixbuf, GdkPixbufRotation winkel) Dreht <i>pixbuf</i> um ein durch <i>winkel</i> bestimmtes Vielfaches von 90 Grad gegen den Uhrzeigersinn; möglich sind: GDK_PIXBUF_ROTATE_NONE 0 Grad GDK_PIXBUF_ROTATE_COUNTERCLOCKWISE 90 Grad GDK_PIXBUF_ROTATE_UPSIDEDOWN 180 Grad GDK_PIXBUF_ROTATE_CLOCKWISE 270 Grad

- GdkPixbuf* gdk_pixbuf_copy_area(const GdkPixbuf *quelle, int quell_x, int quell_y, int breite, int hoehe, GdkPixbuf *ziel, int ziel_x, int ziel_y)** Kopiert eine Fläche der Breite *breite* und der Höhe *hoehe* von der durch *quell_x* und *quell_y* bezeichneten Position im Pixbuf *quelle* an die durch *ziel_x* und *ziel_y* bezeichnete Position im Pixbuf *ziel*. Der Koordinatenursprung (0|0) liegt dabei links oben, und die Koordinatenpaare bezeichnen ebenfalls jeweils die obere linke Ecke des fraglichen Rechtecks. *gdk_pixbuf_copy_area()*
- GdkPixbuf* gdk_pixbuf_scale_simple(pixbuf, breite, hoehe, GdkInterpType interpolation)** Gibt eine Kopie des Pixbuf *pixbuf* zurück, die auf die Abmessungen *hoehe* und *breite* skaliert ist. Dazu wird das durch *interpolation* angegebene Interpolationsverfahren verwendet. Möglich sind: *gdk_pixbuf_scale*()*
- GDK_INTERP_NEAREST** Nächster-Nachbar-Interpolation. Dieser Algorithmus ist der schnellste, aber bietet die schlechteste Qualität. Zum Verkleinern ist er unbrauchbar, beim Vergrößern mag das Ergebnis zufriedenstellend sein.
- GDK_INTERP_TILES** Kachel-Interpolation. Diese Skalierungsmethode wird von PostScript verwendet. Die Ergebnisse sind etwas besser als bei der vorigen Methode; dieser Algorithmus ist zum Verkleinern recht gut geeignet.
- GDK_INTERP_BILINEAR** Bilineare Interpolation. Liefert das beste Preis-Leistungs- beziehungsweise Rechenzeit-Qualitäts-Verhältnis.
- GDK_INTERP_HYPER** Hyperbolische Interpolation. Sehr hohe Qualität bei immensem Rechenaufwand.
- void gdk_pixbuf_scale(quelle, ziel, ziel_x, ziel_y, breite, hoehe, double x_versatz, double y_versatz, double x_faktor, double y_faktor, interpolation)** Für Fälle, in denen Sie mit der vorigen Funktion nicht auskommen, gibt es diese. Sie skaliert den Pixbuf *quelle* in x- beziehungsweise y-Richtung um die Faktoren *x_faktor* und *y_faktor*, verschiebt ihn dann in x-Richtung um *x_versatz* und in y-Richtung um *y_versatz* und schreibt schließlich den durch *ziel_x*, *ziel_y*, *breite* und *hoehe* angegebenen Bereich dieses resultierenden Bildes in den Pixbuf *ziel*. Dessen Inhalt wird dadurch ersetzt. Das Interpolationsverfahren wird in bekannter Weise durch den Wert von *interpolation* bestimmt.
- void gdk_pixbuf_composite(quelle, ziel, ziel_x, ziel_y, breite, hoehe, x_versatz, y_versatz, x_faktor, y_faktor, interpolation, int alpha)** Diese Funktion arbeitet wie die vorige, ersetzt den Inhalt von *ziel* aber nicht durch das neue Bild, sondern überlagert es damit. Dies hat natürlich nur dann Sinn, wenn das überlagernde *gdk_pixbuf_composite()*

Bild irgendwo transparente Stellen hat und/oder Sie den Gesamt-Transparenzwert *alpha* (Bereich 0 bis 255) für das Quellbild auf einen Wert kleiner 255 eingestellt haben.

<i>gtk_pixbuf_get_has_alpha()</i>	gboolean gdk_pixbuf_get_has_alpha(<i>pixbuf</i>) Gibt dann TRUE zurück, wenn der Pixbuf <i>pixbuf</i> einen Alpha-Kanal (Transparenzinformationen) enthält
<i>gdk_pixbuf_add_alpha()</i>	GdkPixbuf* gdk_pixbuf_add_alpha(<i>pixbuf</i>, gboolean <i>ausschneiden</i>, gchar <i>r</i>, gchar <i>g</i>, gchar <i>b</i>) Kopiert die Daten von <i>pixbuf</i> , fügt dabei, falls notwendig, einen Alpha-Kanal hinzu und gibt das Ergebnis zurück. Falls <i>ausschneiden</i> dabei TRUE ist, werden Pixel mit der durch die Rot-Grün-Blau-Werte <i>r</i> , <i>g</i> und <i>b</i> (Bereich jeweils 0 bis 255) angegebenen Farbe dabei transparent gemacht. Alle anderen Pixel (falls <i>ausschneiden</i> FALSE ist, alle) erhalten den Transparenzwert 255 (völlig opak).
<i>gdk_pixbuf_saturate_and_pixelate()</i>	void gdk_pixbuf_saturate_and_pixelate(<i>quelle</i>, <i>ziel</i>, gfloat <i>saettigung</i>, gboolean <i>verpixeln</i>) Ändert die Sättigung (den Farbkontrast) des Pixbufs <i>quelle</i> , vergrößert die Auflösung des Ergebnisses, sofern <i>verpixeln</i> TRUE ist, und schreibt das neue Bild in den Pixbuf <i>ziel</i> . Format und Menge der Pixeldaten muss bei <i>quelle</i> und <i>ziel</i> exakt gleich sein, weshalb es sich empfiehlt, entweder bei beidem den gleichen Pixbuf anzugeben oder als <i>ziel</i> eine Kopie von <i>quelle</i> zu verwenden.
<i>gdk_pixbuf_fill()</i>	void gdk_pixbuf_fill(<i>pixbuf</i>, guint32 <i>farbwert</i>) Füllt den Pixbuf <i>pixbuf</i> mit dem in <i>farbwert</i> angegebenen Rot-Grün-Blau-Alpha-Wert aus (beispielsweise 0xcedeceff für undurchsichtiges Grau, 0x00000000 für transparentes Schwarz, 0xfffff80 für halbdurchsichtiges Weiß und so weiter).

Ein einfaches Beispiel für den Umgang mit **GdkPixbuf** finden Sie in *pixbufdemo.c*

Das Programm stellt ein Bild in einem GtkImage dar, dazu zwei Schieberegler für Zoom und Farbsättigung (siehe Abschnitt 4.7.5). Wird einer dieser beiden Regler bewegt, führt dies dazu, dass ein Signalhandler den neuen Wert in eine globale Variable niederschreibt und die Funktion `bild_aktualisieren()` aufgerufen wird.

Diese erledigt die eigentliche Arbeit: Beim ersten Aufruf sichert sie das Ausgangsbild als Pixbuf. Bei allen Aufrufen erzeugt sie dann jeweils aus diesem am Anfang gesicherten Pixbuf einen neuen mit anhand des Zoomwerts berechneten Abmessungen. Danach wird noch die Farbsättigung angepasst und der neue Pixbuf in das GtkImage geschrieben.

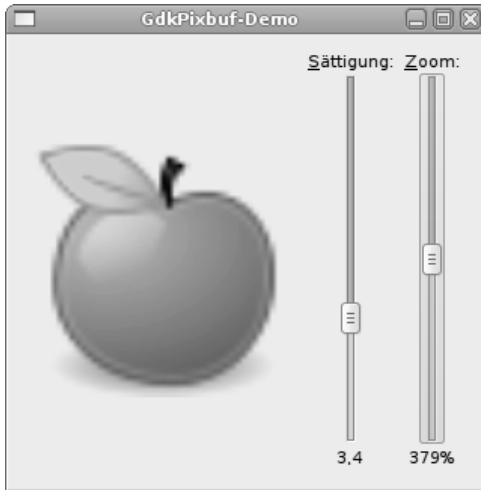


Abbildung 4-6
Resultat von
`pixbufdemo.c`

Anschließend wird der neue Pixbuf mit `g_object_unref()` gelöscht, da er ja ins `GtkImage` kopiert worden ist. Es ist immer wichtig, nicht mehr benötigte Objekte freizugeben, aber hier noch wichtiger als sonst: Gäbe die Funktion den Pixbuf nicht frei, dann würde ein mehrmaliges kräftiges Hin- und Herschieben des Zoomreglers reichen, um dieses winzige Demonstrationsprogramm zu einem Speicherverbrauch von an die 50 M anschwellen zu lassen.

Warnung

Falls Sie noch mehr wissen wollen, schauen Sie sich den Quellcode von Eye Of GNOME (`eog`), dem GNOME-Bildbetrachter, an, der seit jeher auch Prüfstand und Demonstrationsanwendung für `GdkPixbuf` gewesen ist.

Repertoire und Symbolthemen

Zu vielen für GUI-Anwendungen typischen Aktionen gehören in GTK+ standardisierte Beschriftungs-Zeichenketten. Diese enthalten auch standardisierte Abkürzungsbuchstaben sowie Tastenkombinationen und sind als sogenannte ‘Repertoire-Einträge’ in die Bibliotheken selber eingekompiliert.

Der Grund dafür, dies hier zu besprechen, ist die Hauptmotivation, Repertoire-Einträge zu verwenden: Mit diesen Einträgen sind standardisierte Symbole verknüpft. So ist es zum Beispiel möglich, einen Knopf für eine Standardaktion wie »OK« oder »Abbrechen« durch einfachen Verweis auf einen Repertoire-Eintrag sowohl mit einem Symbol als auch mit einer fertig lokalisierten Beschriftung zu versehen.

Ein Repertoire-Eintrag wird mit seiner Kennung angesprochen; diese ist zwar eine Zeichenkette, da aber für jede solche Kennung ein Makro definiert ist, das zu dieser Zeichenkette expandiert, sollte normalerweise dieses Makro verwendet werden.

Eine Übersicht über alle Repertoire-Einträge, die mit GTK+ mitgeliefert werden, finden Sie in Anhang C.

Neben den Symbolen aus dem Repertoire gibt es auch noch einen Mechanismus, um solche, die nicht unbedingt zu standardisierten Aktionen gehören, anhand spezieller Namen zu laden. Wie dies genau zu funktionieren hat, ist in einem XDG-Standard festgelegt [21]. Unter der Haube wird dieser Mechanismus auch benutzt, um die meisten Repertoiresymbole zu implementieren.

Der Vorteil, vorinstallierte Symbole über Namen anzusprechen statt sie aus Dateinamen zu laden, ist einmal der, dass man sich nicht bemühen muss, die entsprechenden Pfade zusammensuchen, vor allem aber, dass die Darstellung gegebenenfalls dem aktuellen Symbolthema angepasst wird. Solche Themen werden, da sie einen plattformübergreifenden Standard implementieren, unabhängig von GTK+-Themen verwaltet. Sofern Sie ein Symbol durch Zuweisen eines Namens an »icon-name« in ein GtkImage geladen haben, wird dieses gegebenenfalls sogar während der Laufzeit Ihres Programms angepasst, wenn das Symbolthema sich zwischenzeitlich ändert.

Warnung

Der Themenstandard definiert leider nicht, welche Symbole grundsätzlich vorhanden sein müssen. Das Vorhandensein der Repertoiresymbole ist hingegen immer garantiert.

Dazu, »von Hand« Symbole aus Symbolthemen zu laden, dient die Klasse **GtkIconTheme**. Ein Aufruf von `gtk_icon_theme_get_default()` liefert ein Objekt dieser Klasse zurück, das dem derzeit aktiven Thema entspricht. Sie können dieses Objekt dann den folgenden Funktionen übergeben:

GtkIconTheme	
<code>gtk_icon_theme_get_default()</code>	
<code>gtk_icon_theme_has_icon()</code>	gboolean <code>gtk_icon_theme_has_icon(GtkIconTheme *thema, const gchar *name)</code> Gibt genau dann TRUE zurück, wenn das Thema ein Symbol namens <i>name</i> enthält.
<code>gtk_icon_theme_get_icon_sizes()</code>	gint* <code>gtk_icon_theme_get_icon_sizes(thema, name)</code> Ein neu angelegtes, mit 0 abgeschlossenes Feld mit allen Größen (sprich Kantenlängen in Pixel), in denen das Symbol <i>name</i> verfügbar ist, zurückgeben. Eine Größe von -1 bedeutet, dass das Symbol in einem Vektorformat vorliegt und daher in jeder beliebigen Größe erhältlich ist.
<code>gtk_icon_theme_load_icon()</code>	GdkPixbuf* <code>gtk_icon_theme_load_icon(thema, name, gint groesse, GtkIconLookupFlags flags, GError **fehler)</code> Lädt das Symbol <i>name</i> im Thema <i>thema</i> , mit einer Kantenlänge von <i>groesse</i> Pixel (sofern

möglich) und gibt es als Pixbuf zurück. Dieser Pixbuf darf nicht geändert und muss nach Gebrauch freigegeben werden; machen Sie, wenn es sich irgendwie bewerkstelligen lässt, eine Kopie davon, damit Sie ihn sofort freigeben können, da GTK+ ansonsten eventuell unnötig einen ganzen Symbolsatz im Speicher halten muss. Beim Heraussuchen des Symbols werden die Optionen in *flags* berücksichtigt; mögliche Flags sind:

- GTK_ICON_LOOKUP_NO_SVG** In keinem Fall SVG-Symbole zurückgeben
- GTK_ICON_LOOKUP_FORCE_SVG** In jedem Fall SVG-Symbole zurückgeben
- GTK_ICON_LOOKUP_USE_BUILTIN** Auch eingebaute Symbole, die nicht aus Dateien geladen werden, berücksichtigen (hierzu gehören die Repertoiresymbole)
- GTK_ICON_LOOKUP_GENERIC_FALLBACK** Falls ein Symbol nicht gefunden wird, sukzessive alle durch Bindestrich abgegrenzten Namens- teile abtrennen und es noch einmal mit dem jeweils verkürzten Namen versuchen

Die beiden möglichen Fehler in der Klasse `GTK_ICON_THEME_ERROR` sind `GTK_ICON_THEME_NOT_FOUND`, falls ein Symbol nicht gefunden werden konnte, und `GTK_ICON_THEME_FAILED` für alle anderen Abbruchgründe.

Ein einfaches Beispiel für die Verwendung von Themensymbolen finden Sie in *iconviewdemo.c*.

4.4.3 Fortschrittsbalken

Zu guter Letzt noch der Fortschrittsbalken. So ein `GtkProgressBar` hat folgende Eigenschaften:

GtkProgressBar

- »**fraction**« (Typ `gdouble`) Der Bruchteil, zu dem der Vorgang abgeschlossen ist
- »**pulse-step**« (Typ `gdouble`) Der Bruchteil des Balkens, um den der Pulsator sich im Aktivitätsmodus bei jedem Puls verschieben soll
- »**orientation**« Ausrichtung und Wachstumsrichtung des Balkens; mögliche Werte sind:
 - GTK_PROGRESS_LEFT_TO_RIGHT** horizontal von links nach rechts
 - GTK_PROGRESS_RIGHT_TO_LEFT** horizontal von rechts nach links
 - GTK_PROGRESS_BOTTOM_TO_TOP** vertikal von unten nach oben
 - GTK_PROGRESS_TOP_TO_BOTTOM** vertikal von oben nach unten
- »**text**« (Typ `GCharArray`) Im Balken anzuzeigender Text

Wie im Beispiel in *anzeige.c* zu sehen, ist der ›Betrieb‹ des Balkens im normalen Verlaufsmodus einfach: Der Füllstand wird über die Eigenschaft ›fraction‹ angepasst, der Text über ›text‹ eingestellt.

Hinweis

Der Füllstand eines Fortschrittsbalkens darf nie größer oder gleich 1,0 sein – dies führt zu unschönen Warnungen auf der Konsole. Im Beispiel ist dieses Problem pragmatisch gelöst, indem der hochgezählte Bruchwert mit `CLAMP()` zwischen 0,0 und 0,999 festgesetzt wird.

`gtk_progress_bar_pulse()`

Im Aktivitätsmodus wird der Balken durch Aufrufe von `gtk_progress_bar_pulse()` ›angetrieben‹. Ein Einsatzgebiet für den Aktivitätsmodus ist jeder Vorgang, dessen Ende nicht absehbar ist. Bei Lichte betrachtet gibt es davon nicht viele. Ein gutes Beispiel ist das Durchsuchen eines Verzeichnisbaumes: dabei ist im Voraus natürlich nicht bekannt, wie viele Dateien gefunden werden. Der Aktivitätsanzeiger könnte also alle 100 gefundene Dateien gepulst werden, um dem Benutzer anzuzeigen, dass das Programm beschäftigt ist.

Das Beispiel zeigt, dass ein und derselbe Fortschrittsbalken problemlos zwischen Verlaufs- und Aktivitätsmodus hin und her wechseln kann. Dies dürfte Ihnen vielleicht von den Fortschrittsbalken so manches Web-Browsers her bekannt vorkommen.⁴

4.5 Behälter und Layout

Ohne Behälterwidgets wären alle anderen ziemlich nutzlos; das Layout eines Fensters kommt zustande, indem man Behälter mit den richtigen Parametern in richtiger Weise ineinanderpackt.

Es gibt folgende Hauptarten von Behältern:

horizontale und vertikale Boxen Dies sind die wichtigsten und grundlegendsten Behälter. Überall, wo es darum geht, mehrere Widgets neben- oder übereinander anzuordnen, wird eine Box verwendet. Es gibt sie horizontal (kurz ›HBox‹, Klasse `GtkHBox`) und vertikal (kurz ›VBox‹, Klasse `GtkVBox`). Beide Klassen sind abgeleitet von der abstrakten Oberklasse `GtkBox`.

Abgeleitet von diesen Boxen sind spezielle Boxen für Knopfrahmen. Sie hören auf den Namen `GtkHButtonBox` (horizontal) und

⁴Ein Fortschrittsbalken im Aktivitätsmodus ist für den ›Zuschauer‹ recht unbefriedigend. Es gibt daher Anwendungen, die es vorziehen, auch dann, wenn sie nicht wissen, wann es weitergeht, den Balken im Verlaufsmodus langsam weiterlaufen zu lassen. Dies ist gewissermaßen eine Vorspiegelung falscher Tatsachen, und Sie sollten so fair zu Ihrem Benutzer sein, den Balken, wenn nötig, in den Aktivitätsmodus zu schalten.

GtkVButtonBox (vertikal) und sind abgeleitet von der abstrakten Oberklasse **GtkButtonBox**. Sie benehmen sich genau wie jede andere Box, bis auf die Tatsache, dass allen Widgets dieser Klasse ein konsistentes Grundlayout gemeinsam ist, das gegebenenfalls auch zentral umgestellt werden kann. Verwenden Sie **Gtk*ButtonBox** also möglichst überall dort, wo Knöpfe anzuordnen sind, aber für keinen anderen Zweck.

horizontale und vertikale Aufteilungen Eine Aufteilung besteht aus zwei Behältern mit einer Schiebeleiste dazwischen. Ein typischer Verwendungszweck einer horizontalen Aufteilung (Klasse **GtkHPaned**) ist es, das Fenster eines Dateimanagers in eine Hauptansicht und eine Seitenleiste zu gliedern. Eine vertikale Aufteilung (Klasse **GtkVPaned**) wird zum Beispiel eingesetzt, um in einem Mailprogramm oder Newsreader die Anzeigefläche für die Kopfzeilen von jener für die Artikelrümpfe zu trennen. Horizontale und vertikale Aufteilungen stammen ebenfalls von einer gemeinsamen abstrakten Oberklasse ab, in diesem Fall von **GtkPaned**.

Notizbücher Diese Art Behälter ist, vor allem in der Windows-Welt, auch als ›Register‹ bekannt. Er besteht aus mehreren Seiten, die übereinander zu liegen scheinen. Jede Seite hat einen Reiter, wie sie aus Notizbüchern und Aktenschränken bekannt sind, und kann über diesen aufgerufen werden. Typischerweise werden Notizbücher (Klasse **GtkNotebook**) verwendet, um Eigenschaftsdialoge und Ähnliches aufgeräumt zu organisieren oder auch zwischen verschiedenen Ansichten eines Objektes umzuschalten.

Notizbücher mit zu vielen Seiten sind eine Plage der modernen GUI-Entwicklung. Versuchen Sie bitte, Ihre Notizbücher übersichtlich zu halten.

Hinweis

Tabellen Ein Behälter, um Widgets in Zeilen und Spalten anzuordnen. Es gibt hierfür mehr Anwendungsgebiete, als es scheinen mag. Der typische Anwendungsfall für eine Tabelle (Klasse **GtkTable**) sind Matrizen aus gleichen Elementen, wie zum Beispiel das Spielfeld bei einem Tic-Tac-Toe-Spiel oder die Ankreuzfelder zum Einstellen der Zugriffsberechtigungen in Nautilus' Eigenschaftsdialog für Dateien; jedoch kann man Widgets in einer Tabelle auch mehrere Zeilen und Spalten überspannen lassen, womit sich, wenn die richtigen Packoptionen eingestellt sind, sehr gut Layouts für vielfältige Anwendungsfälle bauen lassen.

Ausrichtung Dieser einfache Behälter (Klasse **GtkAlignment**) kann genau ein Widget aufnehmen und dessen Ausrichtung und Größe relativ zu seinen Rändern kontrollieren.

Ausklapper Ein Ausklapper (Klasse `GtkExpander`) kann genau ein Widget aufnehmen; durch Klicken darauf deckt er dieses auf oder verbirgt es. Ein zugeklappter Aufklapper sieht normalerweise aus wie eine Beschriftung mit einem nach rechts zeigenden Dreieck daneben.

Ein Beispiel mit Aufteilungen, einer `GtkButtonBox`, einem Notizbuch, einem Aufklapper und einer Tabelle finden Sie in `behaelter.c`. Als Demonstration für `GtkBox` eignet sich so gut wie jedes andere Beispielprogramm in diesem Kapitel.

Abbildung 4-7
Resultat von
`behaelter.c`



GtkContainer

`gtk_container_`
`add()`

Alle Behälter haben die gemeinsame Oberklasse `GtkContainer`. Schon im allerersten Beispiel (siehe `gtkhallo.c`) haben Sie die Funktion `gtk_container_add()` kennengelernt, die als erstes Argument einen `GtkContainer` und als zweites ein `GtkWidget` entgegennimmt und das Widget sodann in den Behälter packt.

Hinweis

Verwenden Sie `gtk_container_add()` nur mit einfachen Behältern wie `GtkWindow` oder `GtkFrame` (siehe Abschnitt 4.8.1). Bei anderen Behältern sollten Sie die dafür vorgesehenen, spezifischeren Packfunktionen verwenden.

`gtk_container_`
`remove()`

Von der Aufrufsyntax her gleich ist `gtk_container_remove()`, das ein Widget aus einem Behälter entfernt.

Warnung

Meistens zerstört dies das Widget, da der Behälter zu diesem Zeitpunkt normalerweise die letzte Referenz darauf hält.

`gtk_container_`
`foreach()`
`GtkCallback`

Schließlich gibt es noch die Möglichkeit, über alle Kinder eines Behälters zu iterieren. Dazu dient `gtk_container_foreach()`; diese Funktion nimmt als erstes Argument den Behälter, als zweites eine Funktion vom Typen `GtkCallback` und als drittes einen Datenzeiger. Die übergebene Funktion wird auf allen Kindern des Containers aufgerufen.

```
/* aus $(PREFIX)/include/gtk-2.0/gtk/gtkwidget.h */
typedef void (*GtkCallback) (GtkWidget *widget, gpointer data);
```

Jeder `GtkContainer` hat des Weiteren die beiden folgenden Eigenschaften:

- »**border-width**« (Typ `gint`) Die Randbreite des Behälters in Pixel. Ist sie null, füllen die hineingepackten Widgets ihn bis zum Rand aus.
- »**child**« (Typ `GtkWidget`, nicht auslesbar) Ein Widget in diese Eigenschaft zu schreiben, packt es in den Behälter. Der Aufruf `g_object_set(behaelter, "child", widget, NULL)` bewirkt also dasselbe wie `gtk_container_add(behaelter, widget)`.

Ein wichtiger Aspekt von Containern ist noch, dass sie für ihre Kinder sogenannte *Kindeigenschaften* verwalten. Dabei handelt es sich nicht um normale Objekteigenschaften des Kindes; sie sind ausschließlich über den Behälter zugänglich. Hierzu dienen die Funktionen `gtk_container_child_get/set()`, die wie `g_object_get/set()` arbeiten, aber vor der Schlüssel-Wert-Liste *zwei* Argumente verlangen, nämlich als erstes den Container und als zweites das Kind, auf dessen Kindeigenschaften zugegriffen werden soll.

`gtk_container_`
`child_get/set()`

Als Hilfsfunktion, um Code und Aufrufe zu sparen, gibt es noch die Packfunktion `gtk_container_add_with_properties()`. Nach den ersten beiden Argumenten, dem Behälter und dem Widget, nimmt diese Funktion ähnlich `g_object_set()` noch eine mit `NULL` abgeschlossene Liste von Kindeigenschaften für das Widget entgegen.

`gtk_container_add_`
`with_properties()`

4.5.1 Boxen

Ob es jetzt eine vertikale Box (`GtkHBox`) oder eine horizontale Box (`GtkVBox`) ist – geschaffen wurden Boxen zu dem Zweck, Widgets hineinzu packen. Dies kann an deren Anfang oder deren Ende geschehen. Zu ersterem dient `gtk_box_pack_start()`, zu zweiterem `gtk_box_pack_end()`.

GtkHBox
GtkVBox

`gtk_box_pack_`
`start/end()`

Gehen Sie nicht davon aus, bei HBoxen sei der Anfang immer links und das Ende immer rechts. Für Benutzer in Sprachumgebungen, wo von rechts nach links gelesen wird, sind nämlich eventuell auch die Widgets dementsprechend gespiegelt.

Hinweis

Beide Funktionen haben dieselbe Folge von Parametern:

1. Die aufnehmende **Box**
2. Das zu packende **Widget**

3. **Ausdehnen?** Ein Wahrheitswert, der darüber entscheidet, ob das Widget sich ausdehnen soll: Der noch verfügbare Freiraum in der Box wird zwischen allen Widgets aufgeteilt, bei denen dieser Parameter beim Packen TRUE war.
4. **Ausfüllen?** Ein weiterer Wahrheitswert, der nur wichtig ist, wenn der vorgenannte Parameter TRUE ist. Ist er TRUE, dann füllt das Widget den ihm nach der genannten Methode zugewiesenen zusätzlichen Freiraum vollständig aus, statt sich nur darin zu zentrieren.
5. **Polsterung** Dieser `guint`-Wert gibt an, mit wie vielen Pixel Leerraum das Widget an den beiden Seiten, mit denen es auf seine Nachbarwidgets beziehungsweise das Ende der Box stößt, jeweils ausgepolstert wird.

`gtk_box_pack_start/end_defaults()`

Wenn Sie diese Werte nicht festlegen möchten, können Sie die vereinfachten Formen der Packfunktionen verwenden, nämlich `gtk_box_pack_start_defaults()` beziehungsweise `gtk_box_pack_end_defaults()`. Diese funktionieren genauso wie die Funktionen ohne `_defaults`, nehmen aber nur zwei Argumente, nämlich die Box und das Widget, entgegen. Ausdehnen und Ausfüllen werden von diesen Funktionen eingeschaltet und die Polsterung auf 0 Pixel festgesetzt.

Eine `GtkBox` hat zwei Eigenschaften:

»**spacing**« (Typ `gint`) Der Abstand zwischen den hineingepackten Widgets in Pixel.

Hinweis

Beim Packen widgetweise zugegebene Polsterung kommt zu diesem Abstand hinzu.

»**homogeneous**« (Typ `gboolean`) Ist dies TRUE, werden und bleiben alle hineingepackten Widgets gleich groß.

Ein Widget gewinnt beim Hineinpacken in eine `GtkBox` folgende Kindeigenschaften, über die es auch nachträglich noch in der Box umhergeschubst und umformatiert werden kann:

»**expand**« (Typ `gboolean`) Entscheidet über das Ausdehnen und entspricht dem dritten Parameter der Packfunktionen.

»**fill**« (Typ `gboolean`) Entscheidet über das Ausfüllen und entspricht dem vierten Parameter der Packfunktionen.

»**padding**« (Typ `guint`) Die Polsterung in Pixel; entspricht dem fünften Parameter der Packfunktionen.

»**pack-type**« Die Seite, auf die das Widget gepackt wurde; mögliche Werte sind `GTK_PACK_START` für den Anfang und `GTK_PACK_END` für das Ende.

»**position**« (Typ `gint`) Die Position des Widgets in der Box, von 0 an vom Anfang her gezählt.

GtkHButtonBox und **GtkVButtonBox** benehmen sich wie normale Boxen, haben aber noch eine zusätzliche Eigenschaft »`layout-style`«. Diese entscheidet darüber, wie die Knöpfe in der Box angeordnet werden, und kann einen der folgenden fünf Werte annehmen:

GtkH/VButtonBox

GTK_BUTTONBOX_DEFAULT_STYLE die gegebene Voreinstellung

GTK_BUTTONBOX_SPREAD Knöpfe werden gleichmäßig in der Box verteilt.

GTK_BUTTONBOX_EDGE Knöpfe werden an den Enden der Box platziert.

GTK_BUTTONBOX_START Knöpfe werden am Anfang der Box gruppiert.

GTK_BUTTONBOX_CENTER Knöpfe werden in der Box zentriert.

GTK_BUTTONBOX_END Knöpfe werden am Ende der Box gruppiert.

Kinder einer **GtkButtonBox** gewinnen zudem die Kindeigenschaft »`secondary`« hinzu. Ist dieser Wahrheitswert `TRUE`, wird das entsprechende Kind als sekundärer Knopf behandelt und unter Umständen anders dargestellt. Hilfsknöpfe sind typische Beispiele für sekundäre Knöpfe.

4.5.2 Tabellen

Eine **GtkTable** ist aufzufassen wie eine zweidimensionale Box. Sie hat, wie im Beispiel in *behaelter.c* zu sehen, zwei Eigenschaften »`n-rows`« und »`n-columns`«, die die Zeilen- beziehungsweise Spaltenanzahl aufnehmen.

GtkTable

Ist eine Tabelle angelegt, können Sie sie als Behälter verwenden. Natürlich reicht es hier nicht, einfach irgendwie die Widgets hineinzupacken; es muss ja festgelegt werden, an welcher Stelle sie landen. Da sie auch mehrere Spalten oder Zeilen überspannen können, müssen eine rechte Spalte, eine linke Spalte, eine obere Zeile und eine untere Zeile angegeben werden: Die Differenz zwischen linkem und rechtem beziehungsweise oberem und unterem Wert entspricht der Breite des Widgets in der Tabelle in Zeilen beziehungsweise Spalten.

Die Funktion, die ein Widget an einer Tabelle festmacht, ist `gtk_table_attach()`. Ihre Argumente sind die folgenden:

`gtk_table_attach()`

1. **Tabelle**
2. **Neues Kindwidget**
3. **Linke Spalte** (Zählung von 0 an) (»`left-attach`«, Typ `guint16`)
4. **Rechte Spalte** (»`right-attach`«, Typ `guint16`)
5. **Obere Zeile** (»`top-attach`«, Typ `guint16`)
6. **Untere Zeile** (»`bottom-attach`«, Typ `guint16`)
7. Ein Bitfeld mit **Formatflags** für die **horizontale Ausrichtung** (»`x-options`«); mögliche Flags sind:

GTK_EXPAND Wenn TRUE, reißt das Widget zusätzlichen Platz an sich (entsprechend dem Verhalten in einer Box, siehe auch Abschnitt 4.5.1).

GTK_FILL Wenn TRUE], füllt das Widget diesen zusätzlichen Platz ganz aus (ebenfalls entsprechend zur Box).

GTK_SHRINK Wenn TRUE, versucht das Widget, auf den verfügbaren Platz zusammenzuschrumpfen.

8. Entsprechende Formatflags für die vertikale Ausrichtung (»y-options«)
9. Polsterung in Pixel zur Linken und Rechten des Widgets (»x-padding«, Typ guint16)
10. Dito über und unter dem Widget (»y-padding«, Typ guint16)

Die Werte werden dabei den in Klammern genannten Kindeigenschaften des Kindes zugewiesen.

gtk_table_attach_defaults()

Wie bei den Packfunktionen für Boxen gibt es auch hier eine vereinfachte Form, nämlich die Funktion `gtk_table_attach_defaults()`, die nur die ersten sechs genannten Argumente nimmt. Für beide Richtungen werden dabei die Flags auf den Wert `GTK_EXPAND|GTK_FILL` und die Polsterung auf 0 Pixel eingestellt.

Folgende Eigenschaften hat ein **GtkTable**-Widget außerdem noch:

»column-spacing« (Typ guint) Spaltenabstand in Pixel

»row-spacing« (Typ guint) Zeilenabstand in Pixel

»homogeneous« (Typ gboolean) Ist dies TRUE, werden und bleiben alle Tabellenspalten gleich breit und alle Zeilen gleich hoch.

4.5.3 Aufteilungen

GtkHVPaned

Eine Aufteilung ist ein Behälter, der zu beiden Seiten der Schiebeleiste je ein Widget aufnehmen kann. Bei **GtkHPaned** ist diese Leiste senkrecht, und die beiden Widgets liegen links beziehungsweise rechts davon; bei **GtkVPaned** ist die Leiste waagrecht, und die Widgets liegen darüber beziehungsweise darunter.

gtk_paned_add1/2()

Hierzu dienen die Funktionen `gtk_paned_add1()` und `gtk_paned_add2()`; beide nehmen jeweils ein `GtkPaned` als erstes und ein `GtkWidget` als zweites Argument. Dabei packt `gtk_paned_add1()` das Kindwidget linkerhand beziehungsweise über, `gtk_paned_add2()` es jedoch rechterhand beziehungsweise unter die Schiebeleiste.

gtk_paned_pack1/2()

Falls Sie das Größenänderungsverhalten der Kinder regeln möchten, bieten die ansonsten gleichwertigen Funktionen `gtk_paned_pack1()` und `gtk_paned_pack2()` hierzu die Möglichkeit. Sie nehmen noch zwei

weitere Argumente vom Typen `gboolean`. Ist das dritte Argument (später als Kindeigenschaft »`expand`« zu erreichen) `TRUE`, dann wirken sich Größenänderungen des Aufteilungswidget auf das Kind aus; es empfiehlt sich somit, zum Beispiel bei einer Seitenleiste, die ihre Breite auch beim Verändern der Fenstergröße behalten soll, dies auf `FALSE` und für das andere Kind auf `TRUE` zu setzen. Das vierte Argument (später Kindeigenschaft »`fill`«) bewirkt, wenn `TRUE`, dass das Kindwidget mit Hilfe der Schiebeleiste unter seine beantragte Minimalgröße geschrumpft werden kann.

4.5.4 Notizbücher

Ein Notizbuch (`GtkNotebook`) besteht aus mehreren Seiten, von denen jede einen Reiter hat. Eventuell kann ein Rechtsklick auf irgendeinen Reiter ein Kontextmenü aufrufen, das den Schnellzugriff auf beliebige Reiter erlaubt.

Ist ein Notizbuch erstellt, gibt es eine Reihe von Funktionen, um ihm Seiten hinzuzufügen. Die wichtigste ist zunächst `gtk_notebook_append_page_menu()`. Sie nimmt als erstes Argument das `GtkNotebook` und als zweites das Kindwidget; in unserem Beispiel sind die Kindwidgets eine Beschriftung, ein Ausklapper mit einem Bild darin und ein Knopf; normalerweise werden Sie aber hier ein Behälterwidget verwenden, um mehrere Widgets auf eine Notizbuchseite zu bringen. Das dritte Argument ist ein Beschriftungswidget für den Reiter.

Sie können also ein beliebiges Widget benutzen, um den Reiter zu beschriften. De facto ist es fast immer unsinnig, hier etwas anderes als ein `GtkLabel` zu verwenden.

Als viertes Argument können Sie ein weiteres Beschriftungswidget angeben, das dann für den Eintrag verwendet wird, unter dem das Schnellzugriffs-Menü den Reiter führt. Haben Sie aber, wie oben empfohlen, ein `GtkLabel` benutzt, um den Reiter zu beschriften, dann reicht es, hier `NULL` zu übergeben, und der Titel des Reiters wird übernommen.

Falls Sie kein solches Menü wünschen, können Sie Ihre Notizbuchseiten auch mit `gtk_notebook_append_page()` anhängen; diese Funktion verzichtet auf das letzte Argument und fügt keinen Menüeintrag ein.

Es gibt noch weitere Funktionen, um Seiten hinzuzufügen:

`gtk_notebook_prepend_page_menu()` Aufrufgleich zu `gtk_notebook_append_page_menu()`, fügt die Seite aber vorne statt hinten im Notizbuch und im Menü an.

`gtk_notebook_prepend_page()` Dito ohne Menüeintrag

GtkNotebook

`gtk_notebook_append_page_menu()`

Hinweis

`gtk_notebook_append_page()`

`gtk_notebook_prepend_page*()`

gtk_notebook_insert_page()* **gtk_notebook_insert_page_menu()** Wie `gtk_notebook_append_menu()`, nimmt aber als fünftes Argument noch einen `gint`-Wert, der die Position des Reiters bestimmt (0 = ganz vorne), und erlaubt so ein Einfügen von Seiten an beliebiger Stelle.

gtk_notebook_insert_page() Dito ohne Menüeintrag

gtk_notebook_remove_page() Ansonsten ist noch die Funktion `gtk_notebook_remove_page()` zu erwähnen; sie nimmt zwei Argumente, ein `GtkNotebook` sowie die Nummer einer Seite (Typ `gint`) und entfernt diese Seite aus dem Notizbuch.

Des Weiteren hat ein Notizbuch noch folgende Eigenschaften:

»**tab-pos**« Die Position der Reiter; mögliche Werte sind:

GTK_POS_LEFT links

GTK_POS_RIGHT rechts

GTK_POS_TOP oben

GTK_POS_BOTTOM unten

»**show-tabs**« (Typ `gboolean`) Ist dies `FALSE`, werden keine Reiter angezeigt.

»**show-border**« (Typ `gboolean`) Ist dies `FALSE`, wird kein Rand angezeigt.

»**scrollable**« (Typ `gboolean`) Ist dies `TRUE`, werden Blätterpfeile angezeigt, wenn mehr Reiter vorhanden sind, als angezeigt werden können.

»**tab-hborder**« (Typ `guint`) Die Breite des Randes über und unter den Reiterbeschriftungen in Pixel

»**tab-vborder**« (Typ `guint`) Die Breite des Randes links und rechts von den Reiterbeschriftungen in Pixel

»**tab-border**« (Typ `guint`, nicht auslesbar) Die Breite des Randes um die Reiterbeschriftungen in Pixel

»**page**« (Typ `gint`) Die Indexnummer der aktuellen Seite

»**enable-popup**« (Typ `gboolean`) Ist dies `TRUE`, kann durch Rechtsklick auf einen Reiter ein Kontextmenü zum direkten Anspringen der verschiedenen Reiter aufgerufen werden.

»**homogeneous**« (Typ `gboolean`) Ist dies `TRUE`, erhalten und behalten alle Reiter die gleiche Breite.

Jedes Kindwidget in einem Notizbuch erhält zudem folgende Kindeigenschaften:

»**tab-label**« (Typ `gchararray`) Beschriftung des Reiters

»**menu-label**« (Typ `gchararray`) Bezeichnung des Reiters im Menü

»**position**« (Typ `gint`) Die Position des Reiters im Notizbuch (ganz vorne = 0)

»**tab-expand**« (Typ `gboolean`) Ist dies `TRUE`, dehnt der Reiter sich aus.

- »**tab-fill**« (Typ `gboolean`) Ist dies `TRUE`, füllt sich der Reiter aus.
- »**tab-pack**« Bestimmt darüber, wie die Seite ins Notizbuch gepackt wird; mögliche Werte sind `GTK_PACK_START` (Anfang) und `GTK_PACK_END` (Ende).

4.5.5 Ausrichtungsbehälter

`GtkAlignment` ist ein sehr einfacher Behälter. Erzeugen Sie einen, stecken Sie ein Widget hinein und konfigurieren Sie folgende Eigenschaften:

GtkAlignment

- »**xalign**« (Typ `gfloat`) Die horizontale Ausrichtung des Kindwidgets
- »**yalign**« (Typ `gfloat`) Die vertikale Ausrichtung des Kindwidgets
- »**xscale**« (Typ `gfloat`) Der horizontale Maßstab des Kindwidgets im Falle, dass der Behälter mehr Fläche einnimmt als das Kind. 0,0 bedeutet, dass das Kind auf keinen Fall vergrößert werden, 1,0, dass es den gesamten zur Verfügung stehenden Platz einnehmen soll.
- »**yscale**« (Typ `gfloat`) Dito für den vertikalen Maßstab
- »**top-padding**« (Typ `guint`) Die Polsterung zur Oberseite hin in Pixel
- »**bottom-padding**« (Typ `guint`) Dito zur Unterseite hin
- »**left-padding**« (Typ `guint`) Dito nach links
- »**right-padding**« (Typ `guint`) Dito nach rechts

Eigenschaften mit Namen wie »**xalign**« und »**yalign**«, die ebenfalls durch einen Wert zwischen 0,0 und 1,0 die Ausrichtung eines Widgets kontrollieren, finden sich auch in vielen anderen Klassen.

Hinweis

4.5.6 Aufklapper

Auch ein `GtkExpander` nimmt genau ein Widget auf. Seine Beschriftung steht in der Eigenschaft »**label**«; entsprechend sind auch »**use-markup**« und »**use-underline**« vorhanden. In »**label-widget**« kann ein `GtkWidget` eingetragen werden, das an Stelle einer Beschriftung erscheint. »**spacing**« bestimmt hier den Abstand zwischen Ausklapper und Kind.

Die `gboolean`-Eigenschaft »**expanded**« ist `TRUE`, wenn der Ausklapper ausgeklappt ist. Benutzen Sie sie, um beim Anlegen zu kontrollieren, ob das Kind per Vorgabe sichtbar oder verborgen sein soll.

4.5.7 Bedienelemente sinnvoll anordnen

Nun wissen Sie, wie Sie Bedienelemente in Fenstern anordnen *können*. Dazu, wie Sie es tun *sollten*, um für Konsistenz, Klarheit und Lesefreundlichkeit zu sorgen, hat die GNOME-Richtlinie für Benutzeroberflächengestaltung einiges zu sagen.

- Das Layout sollte oben links beginnen und von oben nach unten und von links nach rechts fortschreiten. Dies gilt zumindest für Sprachumgebungen, in denen von links nach rechts gelesen wird.
- Abstände zwischen Bedienelementen sollten alle ein ganzzahliges Vielfaches von 6 Pixel betragen, beispielsweise:
 - 6 Pixel Abstand zwischen einem Symbol und seiner Beschriftung
 - 12 Pixel Abstand zwischen sonstigen Elementen und der zugehörigen Beschriftung
 - 12 Pixel freier Rand um den Inhalt eines Dialogfensters
 - 18 Pixel Breite für vertikale, der Gruppierung in ›Spalten‹ dienende Leerräume
- Ausrichten, Einrücken und Gruppieren durch Abstand sind lesefreundlicher als Rahmen, Trennlinien und andere ›Zierelemente‹. Statt beschrifteter Rahmen sollte eine fette, linksbündige Überschrift, unter der die zugehörigen Gruppenelemente 12 Pixel weit eingerückt sind, verwendet werden.

4.6 Knöpfe

Zu den einfachsten Bedienelementen gehören Knöpfe.

4.6.1 Eigentliche Knöpfe

Knöpfe im engeren Sinne gibt es in drei verschiedenen Arten, wie sie auch von Elektrogeräten her bekannt sind:

GtkButton Knöpfe – `GtkButton` ›Taster‹ für die Elektriker unter Ihnen. Ein Knopf löst eine Aktion aus, wenn man ihn drückt. Dementsprechend sollte die Beschriftung eines Knopfes auch immer eine Infinitivkonstruktion der Art »Dies tun« oder »Das tun« sein. Ein Knopf ist Ihnen schon im ersten Beispiel in *gtkhallo.c* begegnet.

GtkCheckButton Ankreuzfelder – `GtkCheckButton` Elektrisch gesprochen: ›Druckschalter‹. Auch wenn Ankreuzfelder nicht nach Knöpfen aussehen,

sie benehmen sich genauso wie ein Druckschalter, der entweder an ist oder aus.⁵

Radioknöpfe – GtkRadioButton Radioknöpfe sind Radioknöpfe, auch wenn man sie mittlerweile eher von billigen Kassettenrecordern und Ventilatoren als von Radios her kennt. Im Gegensatz zu anderen Knöpfen tauchen sie nie einzeln, sondern immer in Gruppen auf; erst durch ihr Gruppensein wird das möglich, was ihr Wesen ausmacht, nämlich, dass immer nur ein Knopf in einer Gruppe niedergedrückt sein kann.

GtkRadioButton

Beispiele für alle drei Arten finden Sie in *knoepfe.c* (Bildschirmfoto in Abbildung 4-8). Die dargestellte Anwendung ist ein Automat für mehr oder weniger heiße Getränke – Sie können zwischen drei Temperaturen für das Getränk wählen, die sich natürlich gegenseitig ausschließen; hierzu wird ein Radioknopf verwendet; auch können drei Zutaten mit Ankreuzfeldern an- oder abgewählt werden. Zubereitet wird das Getränk dann durch Druck auf einen Knopf.⁶

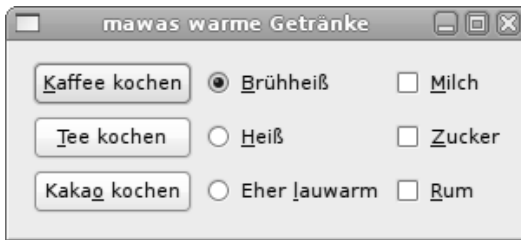


Abbildung 4-8

Resultat von *knoepfe.c*

Wie Sie sehen, werden alle drei Arten von Knöpfen auf ähnliche Weise erstellt. Sie alle tragen in der üblichen Eigenschaft »label« ihre Beschriftung und werden auf übliche Art in Behälter gepackt.

Allen Knöpfen gemeinsam sind auch die Eigenschaften »use-underline« und »use-stock«; beide können nur beim Erstellen des Widgets beschrieben werden und enthalten Wahrheitswerte. Ist »use-underline« gleich TRUE, werden durch Unterstriche markierte Abkürzungsbuchstaben in der Beschriftung berücksichtigt. Im

⁵Es gibt auch Umschaltknöpfe in GTK+; diese sind funktionell identisch mit Ankreuzfeldern, sehen aber aus wie ein richtiger Druckschalter. Sie sollten nicht verwendet werden, da sie Knöpfen zu ähnlich sind und man einem Ankreuzfeld seinen Status leichter ansehen kann.

⁶Der Verständlichkeit zuliebe wurde auf den umfangreichen Code zur Ansteuerung einer TCP/IP-Kaffeemaschine verzichtet; das Programm gibt einfach eine Meldung der Form »Es wird lauwarmer Kaffee mit Zucker gekocht.« auf der Konsole aus.

Beispiel haben die Aktionsknöpfe solche Abkürzungen, so dass Sie beispielsweise mit Alt-K Kaffee kochen können.

Um einen Knopf aus dem Repertoire darzustellen, setzen Sie »use-stock« auf TRUE. Der Inhalt von »label« wird dann nicht als Beschriftung interpretiert, sondern als Kennung eines vorgefertigten Knopfes aus dem Repertoire (siehe Abschnitt 4.4.2).

Wann immer ein Knopf übrigens neben Text noch ein Bild (wie das Repertoiresymbol) enthält, ist dieses ein Kindwidget des Knopfs und als `GtkWidget*` durch die Eigenschaft »image« zu erreichen. Prinzipiell sind somit beliebige Widgets in einem Knopf möglich, sinnvoll sind aber nur wenige, so zum Beispiel `GtkImage` oder `GtkArrow`.

Eine Besonderheit ist die Eigenschaft »group« bei `GtkRadioButton`. Sie nimmt ein `GObject` auf, und zwar einen anderen Radioknopf, in dessen Gruppe der betroffene Knopf eingeordnet werden soll. Wie im Beispiel ersichtlich ist es somit der einfachste Weg, eine Gruppe von Radioknöpfen zu erstellen, den ersten Knopf anzulegen und ihn dann bei allen weiteren Knöpfen in die »group«-Eigenschaft einzutragen.

`GtkCheckButton` und `GtkRadioButton` sind noch zwei Eigenschaften gemeinsam, nämlich »active« und »inconsistent«. Die für Sie normalerweise wichtigste ist »active«, ein Wahrheitswert, der die Information darüber enthält, ob der Knopf eingedrückt beziehungsweise das Feld angekreuzt ist. Ebenfalls ein Wahrheitswert ist »inconsistent« – ist diese Eigenschaft TRUE, wird der Knopf oder das Feld in einem ›Zwischenzustand‹ (mit Querbalken oder grau ausgefüllt) dargestellt, der signalisiert, dass sich keine klare Aussage darüber machen lässt, ob dieses Element ein- oder ausgeschaltet ist. (Eine sinnvolle Verwendung dieses Zustandes ist das Darstellen von nicht durchgängigen Formatierungen in einer markierten Textstrecke, wie zum Beispiel in *textdemo.c*.)

`GtkButton` verfügt außerdem über die Eigenschaften »xalign« und »yalign«, die wie üblich funktionieren, und über die `gboolean`-Eigenschaft »focus-on-click«. Setzt man letztere entgegen der Vorgabe auf FALSE, hat dies zur Folge, dass der Knopf, wenn er angeklickt wird, nicht den Tastaturfokus erhält. Dies kann beispielsweise sinnvoll sein bei Knöpfen, die als Zubehör zu Eingabefeldern, in denen mit der Tastatur gearbeitet wird, dienen.

Das einzige wichtige Signal in `GtkButton` ist »clicked«, das beim Klicken auf den Knopf emittiert wird.

4.6.2 Linkknöpfe

GtkLinkButton

Eine Unterklasse von `GtkButton` ist auch `GtkLinkButton`, ein Widget, das einen Hyperlink darstellt. Es ist bei üblichen Themeneinstellungen nicht ohne Weiteres als Knopf zu erkennen.

Gegenüber einem einfachen Knopf hat `GtkLinkButton` eine zusätzliche Eigenschaft »uri«, die die Linkadresse als Zeichenkette enthält. Damit Sie nicht für jeden Link denselben »clicked«-Handler binden müssen, bringt die Klasse eine Hilfsfunktion mit, um einen globalen Hook einzurichten; dessen Prototyp vom Typen `GtkLinkButtonUriFunc` sieht folgendermaßen aus:

```
/* aus $(PREFIX)/include/gtk-2.0/gtk/gtklinkbutton.h */
typedef void (*GtkLinkButtonUriFunc) (GtkLinkButton *button,
                                       const gchar *link_,
                                       gpointer user_data);
```

Um einen solchen Hook einzurichten, rufen Sie `gtk_link_button_set_uri_hook()` auf. Die Funktion erhält den Hook als erstes Argument, als zweites einen Datenzeiger und als drittes gegebenenfalls eine Freigabefunktion (`GDestroyNotify`), die auf dem Datenzeiger aufgerufen wird, wenn dieser nicht mehr gebraucht wird.

`gtk_link_button_set_uri_hook()`

Das Beispielprogramm `linkdemo.c` stellt einige Linkknöpfe dar. Der verwendete Hook ruft eine GIO-Funktion auf, um die angeklickten Adressen zu öffnen (hierzu siehe Abschnitt 6.2.5). Sie sehen auf dem Bildschirmfoto, dass einer der Links als Knopf dargestellt wird; dies liegt daran, dass der leider nicht mit abgebildete Mauszeiger darüber steht.



Abbildung 4-9
Resultat von
`linkdemo.c`

4.7 Dateneingabe

Zur Eingabe von komplexeren Daten, die sich nicht durch ein paar Knöpfe oder Kästchen repräsentieren lassen, in eine Anwendung dienen verschiedene Widgets:

Textfelder – `GtkEntry` Eine einzelne Zeile zur Texteingabe.

Textfelder sollten nur zur Eingabe von freiem Text verwendet werden, bei dem es keinerlei vorgegebene Antwortmöglichkeiten gibt. Für alles andere gibt es speziellere Widgets (siehe weiter unten).

Zur Eingabe von Zahlen sollten Schieberegler oder Zahlenfelder verwendet werden.

Comboboxen – GtkComboBox In ihrer einfachsten Form ist eine Combobox ein Feld mit einer aufklappbaren Liste, aus der ein Eintrag ausgewählt werden kann. Je nach GTK+-Thema kann dieses Feld im Ganzen als Knopf, also wie die unter Unix eingebürgerten Optionsmenüs, oder wie beispielsweise bei Windows als nicht veränderbares Textfeld mit Aufklapper daneben dargestellt werden.

Combo-Textfelder – GtkComboBoxEntry Dies ist gewissermaßen eine Kreuzung aus Textfeld und Combobox: ein Textfeld mit einem Aufklapper daneben, der eine Liste von Ausfüllvorschlägen öffnet. Comboboxen sind einem einfachen Textfeld dann vorzuziehen, wenn es eine sinnvolle Reihe solcher Vorschläge gibt.

Schieberegler – GtkScale Zum Einstellen von nach oben und unten begrenzten Stellgrößen, bei denen es hauptsächlich auf das relative Regeln (»Dreh mich insgesamt leiser, aber gib mir noch mehr Bass!«) ankommt und der eigentliche Wert nicht von überragender Bedeutung ist, gibt es Schieberegler.⁷

Es gibt sie als horizontale Regler (Klasse `GtkHScale`) und als vertikale Regler (Klasse `GtkVScale`), die beide abgeleitet sind von der abstrakten Oberklasse `GtkScale`.

Zahlenfelder – GtkSpinButton Geht es weniger um das Regeln als um das genaue Angeben eines Wertes für eine Stellgröße, und ist die Größe nach mindestens einer Seite unbegrenzt,⁸ ist ein Zahlenfeld gefragt – ein änderbares Feld mit einer Zahl darin und zwei Knöpfen, um diese Zahl um einen gewissen Festbetrag zu erhöhen oder zu vermindern.

dateneingabe.c All diese Widgets werden demonstriert in *dateneingabe.c*. Es wird dort ein Fenster angelegt, das sechs Widgets enthält: ein Textfeld, eine Combobox, ein Combo-Textfeld sowie zwei Schieberegler (je einer horizontal und vertikal) und ein Zahlenfeld.

Beide Regler und das Zahlenfeld sind mit derselben Stellgröße verbunden, so dass Sie, wenn Sie einen der Schieberegler bewegen oder

⁷Andere GUI-Toolkits haben auch Drehknöpfe, aber die sind bis auf ihren Dekorationswert eigentlich unbrauchbar, da sie nicht intuitiv mausbedienbar sind.

⁸Beziehungweise »praktisch unbegrenzt«, da in GTK+ jede Stellgröße einen definierten Wertebereich haben muss.

das Zahlenfeld bearbeiten, beobachten können, wie die jeweils anderen beiden Widgets die Änderung mitmachen.⁹



Abbildung 4-10
 Resultat von
dateneingabe.c

Zur Auswahl von drei sehr speziellen, im GUI-Alltag jedoch häufigen Arten von Eingabedaten dienen die folgenden, in *waehler.c* demonstrierten Widgets:

Farbwähler Ein Farbwähler ist ein ziemlich großes und mächtiges Widget zur Auswahl einer Farbe. Es enthält unter anderem ein Farbrad, Regler für die verschiedenen Farbkomponenten und eine Palette, die geladen und gespeichert werden kann. Das eigentliche Widget findet sich in der Klasse `GtkColorSelection`, ein Farbfeld in `GtkColorButton`; brauchen Sie einen Farbwahldialog, können Sie `GtkColorSelectionDialog` verwenden.

Schriftwähler Analog zum Farbwähler dient ein Schriftwähler zum Auswählen einer Schrift. Wiederum gibt es das reine Widget als Klasse `GtkFontSelection`, ein Schriftfeld in `GtkFontButton` und eine Klasse für Schriftwahldialoge (`GtkFontSelectionDialog`).

Dateiwähler Der Dateiwähler dient zum Auswählen von Dateien oder Verzeichnissen; das reine Widget hat hier die Klasse `GtkFileChooserWidget`, das Dateifeld `GtkFileChooserButton` und der Dialog `GtkFileChooserDialog`. Alle drei implementieren die Schnittstelle `GtkFileChooser`.

4.7.1 Textfelder

`GtkEntry` ist eines der am einfachsten zu verwendenden Widgets. Im Beispiel wird ein Textfeld einfach erstellt und gepackt; die Eigenschaft »text« dient dazu, ihm bei der Erzeugung einen Inhalt zuzuweisen.

GtkEntry

⁹In MVCesisch ausgedrückt ist die Stellgröße das Model, die drei Widgets stellen Views darauf dar und das von Ihnen beeinflusste Widget einen Controller dafür. Näheres zu MVC siehe Abschnitt 4.12.