

Programmieren

Einführung

mit

Ruby

Teil 2: Grafische Benutzeroberflächen mit Ruby/GTK



Verbesserungen, Ergänzungen, Vervielfältigung erwünscht!

20. August 2006

Franz Burgmann

f.burgmann@gmail.com

Grafische Oberflächen mit Ruby

Nachdem du dir im ersten Teil der Ruby-Einführung die wesentlichen Grundlagen der Programmierung erarbeitet hast, wird, so hoffe ich, der Ruf nach mehr laut - nach grafischen Benutzeroberflächen, die heute aus der Anwendungsprogrammierung nicht mehr weg zu denken sind.

Ruby kann mittels so genannter Bindings (Binding = Anbindung) auf eine Vielzahl grafischer Bibliotheken zugreifen: GTK+, Tk, wxWidgets, Fox und Qt sind einige davon.

Wir werden uns mit der Bibliothek GTK+ (<http://www.gtk.org/>) und den entsprechenden Ruby-Bindings Ruby/GTK beschäftigen. GTK+ ist freie Software und unterliegt keinen Lizenzgebühren. GTK+ ist für Linux, Mac OS X und MS Windows verfügbar. Damit laufen auch deine Ruby/GTK-Programme auf diesen Plattformen.

Du findest im Zusammenhang mit der Bibliothek GTK+ häufig die Begriffe Gnome, Gnome2 und GTK2. Gnome ist ein Desktopmanager für Linux, der auf der Bibliothek GTK+ basiert und aktuell in der Version 2 (deshalb auch Gnome2) vorliegt. GTK2 drückt aus, dass die aktuelle Version 2 der Bibliothek GTK+ gemeint ist.

Der Name des Gesamtpaketes, Ruby-GNOME2, rührt daher, dass hier zusätzlich zur GTK+-Anbindung (Ruby/GTK) auch noch Gnome-Spezialitäten verfügbar sind. Diese jedoch sind nicht plattformunabhängig und werden hier nicht behandelt. Mehr dazu findest du auf den offiziellen Seiten von Ruby-GNOME2: <http://ruby-gnome2.sourceforge.jp/>.

Installation der Ruby/GTK-Bindings

Voraussetzung ist ein installierter Ruby-Interpreter. Die Installationsschritte für Linux, MS Windows und Mac OS X wurden bereits erläutert (Programmieren - Einführung mit Ruby, Teil1: Grundlagen).

Installation in Linux

Für die Installation unter Gentoo-Linux reicht der Befehl „emerge ruby-gnome2“, in Ubuntu-Linux lautet er „apt-get install ruby-gnome2“.

Für SUSE-Linux und Fedora-Linux stehen Binärpakete bereit (<http://franz.hob->

bruneck.info/downloads). Siehe dazu auch die Installationsanleitungen auf der offiziellen Ruby-GNOME2-Seite.

Du kannst die Bindings auch selbst kompilieren. Dazu müssen allerdings die Gnome- und Ruby-Entwicklerpakete installiert sein. Lade die Quellen von der Ruby-GNOME2-Seite und führe im entpackten Ordner die folgenden Befehle aus:

```
ruby extconf.rb
make
make install
```

Zum Installieren (make install) benötigst du Root-Rechte.

Installation in Microsoft Windows

Während für Linux das gesamte Paket der Ruby-GNOME2-Bindings installiert werden kann, ist dies für Windows nicht möglich, da für Windows der Desktopmanager Gnome nicht verfügbar ist. Für die Beispiele in diesem Skriptum reicht jedoch, wie bereits erwähnt, die Anbindung an GTK+.

Verwende dazu für Windows das Paket ruby-gtk2-<version>-i386-msvcrt-1.8.zip von der offiziellen Ruby-GNOME2-Seite und verschiebe den Inhalt des entpackten Ordners in das Ruby-Installationsverzeichnis (standardmäßig c:\ruby).

Hinweis: häufig wird der Ordner selbst (ruby-gtk2-<version>-i386-msvcrt-1.8) nach c:\ruby kopiert, deshalb erlaube ich mir die Wiederholung: damit die Bindings funktionieren, den Inhalt dieses Ordners, nicht den Ordner selbst kopieren. Beim (korrekten) Vorgang wirst du darauf aufmerksam gemacht, dass evt. Dateien überschrieben werden, dem kannst du getrost zustimmen.

Für Windows wird weiters die GTK+-Entwicklungsumgebung (Gtk+/Win32 Development Environment) benötigt, sie nennt sich gtk-win32-devel-<version>.exe, du findest sie etwa hier: <http://gladewin32.sourceforge.net/>.

Unter Windows waren in der Vergangenheit manche Kombinationen aus Ruby-Installer, Bindings und GTK+-Runtime nicht zur Zusammenarbeit zu bewegen, erfolgreich getestet wurden die augenblicklich aktuellsten Versionen der jeweiligen Pakete:

Ruby-One-Click Installer: ruby184-20.exe

Ruby/GTK: ruby-gtk2-0.15.0-1-i386-msvcrt-1.zip

GTK+-Entwicklungsumgebung: gtk-win32-devel-2.8.18-rc1.exe

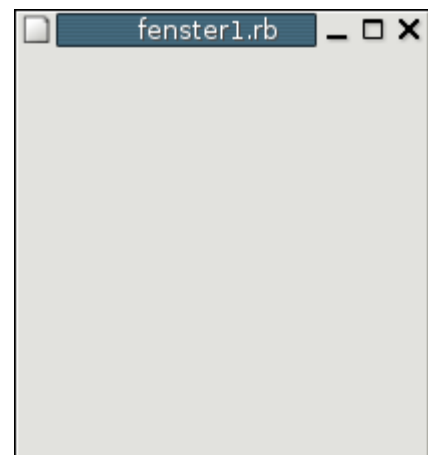
Installation in Mac OS X

Eine automatisierte Installation der Ruby/GTK-Bindings inklusive aller Abhängigkeiten ermöglicht DarwinPorts (<http://darwinports.opendarwin.org/>). Du installierst die Bindings mit „sudo port install rb-gnome“. Ruby/GTK-Programme erscheinen in Mac OS X jedoch nicht im Aqua-Look, da die Bibliothek GTK+ (noch) nicht nativ auf Mac OS X läuft. Es gab jedoch immer wieder Ansätze dafür, ein aktuell viel versprechender ist hier zu finden: <http://developer.imendio.com/projects/gtk-macosx>.

Ein einfaches Fenster

```
fenster1.rb
(1) require 'gtk2'
(2)
(3) Gtk::init
(4) fenster = Gtk::Window.new
(5) fenster.show
(6) Gtk::main
```

Diese fünf Zeilen zaubern bereits ein Fenster auf den Schirm!



Sollte dich einiges an diesem Code befremdlich anmuten, werden wir dem abhelfen:

```
require 'gtk2'
```

Diese Zeile bindet die Bibliothek GTK+ (Version 2) ein.

```
Gtk::
```

Dieser Ausdruck kommt sogar drei mal vor. „Gtk“ stellt einen Namensraum dar, die zwei aufeinander folgenden Doppelpunkte „::“ zeigen an, dass die Klasse „Window“ in dem genannten Namensraum zu finden ist.

Durch die Zeile „include Gtk“ kannst du dir in weiterer Folge etwas Tipparbeit sparen und förderst zudem die Übersichtlichkeit:

```
fenster2.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) init
(5) fenster = Window.new
(6) fenster.show
(7) main
```

Eine Erklärung bin ich dir noch schuldig: „init“ in Zeile 4 initialisiert die grafische Bibliothek und „main“ startet die Programm-Hauptschleife. Diese beiden Methoden werden in jedem GTK+-Programm benötigt. Hinweis: ab Version 0.15.0 der Bindings wird der Aufruf der Methode Gtk::init von den Bindings veranlasst und ist im Programm selbst deshalb nicht mehr nötig.

Ohne die Methode „fenster.show“ (Zeile 6) würde das Fenster erstellt, aber nicht angezeigt werden. Die Methode „show“ musst du für jede grafische Komponente (Widget), welche du sehen möchtest, aufrufen. Alternativ kannst du auf das Fenster „show_all“ anwenden, diese Methode zeigt das Fenster inklusive aller zu diesem Zeitpunkt eingefügten Widgets an.

Dasselbe noch mal, diesmal erstellen wir die Klasse „Fenster“, welche von Gtk::Window abgeleitet ist und damit alle Eigenschaften und Fähigkeiten dieser Klasse erbt:

```
fenster3.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)     def initialize
(7)         super
(8)         self.show
(9)     end
(10)
(11)end
(12)
(13)init
(14)Fenster.new
(15)main
```

Hier ist der zum Fenster gehörende Code in eine Klasse eingebettet . Bei umfangreichen Projekten wird sich eine solche Vorgehensweise bezahlt machen. Du solltest dieses Modell jenem vorziehen.

„super“ (Zeile 7) ruft den Konstruktor der Superklasse (Window) auf, dies ist nötig, um Systemvariablen zu initialisieren. Das Schlüsselwort „self“ in Zeile 8 verdeutlicht, dass sich die Methode show auf die Klasse selbst, eigentlich die erzeugte Instanz der Klasse, das Objekt, bezieht.

Selbstverständlich kannst du die neu erstellte Klasse auch innerhalb jeder anderen Klasse verwenden:

```

fenster3a.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)     def initialize
(7)         super
(8)         self.show
(9)     end
(10)
(11)end

```



```

main_klasse.rb
(1) require 'fenster3a'
(1)
(2) init
(3) Fenster.new
(4) main

```

Im nächsten Beispiel sehen wir uns einige Methoden der Klasse Window an:

```

fenster4.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)     def initialize
(7)         super
(8)
(9)         self.set_title( "mein Fenster" )
(10)        self.set_size_request( 300, 300 )
(11)        self.set_resizable( false )
(12)        self.set_window_position( Position::MOUSE )
(13)        self.show_all
(14)    end
(15)
(16)end

```

In Zeile 10 wird die Standard- und Mindestgröße des Fensters festgelegt, in Zeile 11 verhindern wir, dass der Benutzer die Größe des Fensters ändern kann und in Zeile 12 schließlich wird dafür gesorgt, dass das Fenster beim Starten des Programms immer genau an der Stelle des Bildschirms erscheint, wo sich der Mauszeiger befindet.

Du kannst die Felder eines Widgets auch direkt lesen und schreiben, indem du etwa den Fenstertitel so definierst:

```

self.title = "mein Fenster..."

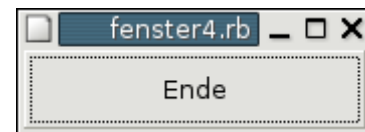
```

Eine Komponente ins Fenster einfügen

Einem inhaltslosen Fenster fehlt es an Nutzwert; ich werde dir jetzt zeigen, wie du grafische Komponenten (Widgets) ins Fenster einfügen kannst.

Im nächsten Beispiel werden wir einen Button einfügen. Wird auf den Button geklickt, soll das Programm beendet und eine entsprechende Meldung ausgegeben werden.

```
fenster5.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)   def initialize
(7)     super
(8)
(9)     button = Button.new( "Ende" )
(10)    button.show
(11)
(12)    self.add( button )
(13)    self.show
(14)  end
(15)
(16)end
(17)
(18)
```



Optisch entspricht das Ergebnis den Erwartungen, nur wird das Programm beim Klicken auf den Button nicht beendet.

Nun, alleine die Aufschrift führt natürlich noch nicht zur gewünschten Funktionalität. Der Programmierer muss vielmehr für jedes Ereignis (Event) den entsprechenden Code selbst schreiben.

Wenn ein Ereignis ausgelöst wird (z.B. Klicken auf einen Button), wird ein Signal erzeugt. Und dieses Signal kann vom Programm abgefangen und vom so genannten Signal-Handler ausgewertet werden.

Signal-Handler

GUI-Programme (Programme mit grafischer Benutzerschnittstelle) brauchen im Gegensatz zu Konsolenprogrammen keine zusätzliche Schleife, die das Programm am Leben hält, dafür ist standardmäßig die main-Schleife zuständig

(Gtk::main).

GUI-Programme werden als „Ereignis-Orientiert“ bezeichnet. Das Programm wartet ständig darauf, dass ein Ereignis (event) ausgelöst wird.

Was sind nun solche Ereignisse? Bleiben wir beim Button: der Benutzer kann den Mauszeiger auf den Button bewegen -> ein Ereignis (enter). Der Benutzer kann mit dem Mauszeiger den Button verlassen -> wieder ein Ereignis (leave). Weiters kann eine Maustaste gedrückt -> (pressed), losgelassen -> (released) oder geklickt, das heißt gedrückt und losgelassen werden -> (clicked).

Keine Sorge: wenn du beim Klicken eines Buttons eine Aktion auslösen möchtest, reicht die Aktion „clicked“, wie das fertig gestellte Beispiel zeigt:

```
fenster6.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)     #Membervariablen
(7)     @btn_ende
(8)
(9)     def initialize
(10)         super
(11)
(12)         @btn_ende = Button.new( "Ende" )
(13)         @btn_ende.signal_connect( "clicked" ) do
(14)             on_btn_ende_clicked
(15)         end
(16)
(17)         self.add( @btn_ende )
(18)         self.show_all
(19)     end
(20)
(21)     def on_btn_ende_clicked
(22)         puts "Programm wird beendet..."
(23)         main_quit
(24)     end
(25)
(26) end
```

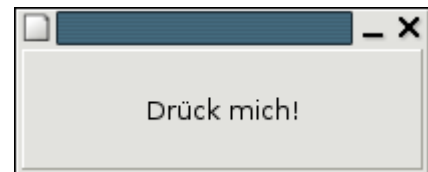
In Zeile 13 wird dem Button mit der Methode „signal_connect“ ein Signal-Handler (Event-Handler) hinzugefügt, der beim Auslösen des Ereignisses „clicked“ die angeführte Methode, hier „on_button_clicked“, aufruft. Diese spezielle Methode wird auch „Signal-Handler-Methode“ oder, korrekter, „Callback-Methode“ (oder auch nur Callback) genannt.

Im nächsten Beispiel wird auf das Ereignis „enter“ reagiert, wenn auch so, dass der Benutzer nicht sehr begeistert sein könnte...


```

drueck_mich.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)     #Membervariablen
(7)     @button
(8)
(9)     def initialize
(10)         super
(11)
(12)         self.set_title( "" )
(13)         self.set_size_request( 200, 60 )
(14)         self.resizable = false
(15)         self.set_window_position( Position::CENTER )
(16)
(17)         @button = Button.new( "Drück mich!" )
(18)         @button.signal_connect( "enter" ) do
(19)             on_button_enter
(20)         end
(21)
(22)         self.add( @button )
(23)         self.show_all
(24)     end
(25)
(26)     def on_button_enter
(27)         x = rand * 1000
(28)         y = rand * 700
(29)         self.move( x, y )
(30)     end
(31)
(32)end

```



Hinweise:

- Unter Windows XP scheint das Ereignis „enter“ manchmal ein wenig zu haken, der Grund dafür ist mir unbekannt.
- Die Aufschrift auf dem Button beinhaltet ein Sonderzeichen („ü“), dieses kann in manchen Fällen nicht korrekt angezeigt werden. Eine Lösung ist, den String explizit in die UTF-8-Kodierung zu transformieren, Zeile 17 etwa würde dann so lauten:

```
@button = Button.new( GLib.locale_to_utf8( "Drück mich!" ) )
```

Arbeitsauftrag: erzeuge ein Fenster mit einem Button und registriere alle fünf Ereignisse. Wenn eines ausgelöst wird, soll eine entsprechende Meldung auf der Konsole ausgegeben werden.

Dokumentation

Nachdem wir jetzt einen ersten Blick auf einige grafische Komponenten geworfen haben, sollte unser nächster Schritt sein, die Ruby/GTK-Klassendokumentation ins Auge zu fassen. Dies ist wichtig, damit du die unzähligen Methoden, die jede einzelne Klasse zur Verfügung stellt, nachschlagen und benutzen kannst.

Dokumentation Online

Du findest die Dokumentation online unter der URL <http://ruby-gnome2.sourceforge.jp/hiki.cgi?Ruby-GNOME2+API+Reference>

Ruby Browser

Sehr empfehlenswert ist der Dokumentationsbrowser „rbbr“ (**R**uby **B**rowser), du kannst ihn hier herunterladen:

<http://ruby-gnome2.sourceforge.jp/hiki.cgi?rbbr>.

Aktuell ist das Paket „rbbr-0.6.0-withapi.tar.gz“.

Installieren

Der Ruby Browser setzt ein installiertes Ruby mit Ruby/GTK-Bindings voraus. In Ubuntu-Linux kannst du den Ruby-Browser mit „apt-get install rbbr“ installieren, in Gentoo-Linux lautet der Befehl „emerge rbbr“

Für Windows und Linux-Distributionen, für die kein Paket bereit gestellt wird, muss die Installation „zu Fuß“ erledigt werden:

Entpacke dazu zunächst den rbbr-Tarball (rbbr-0.6.0-withapi.tar.gz) in ein temporäres Verzeichnis. Wechsle auf der Konsole dorthin und führe die folgenden Befehle aus:

```
ruby install.rb config
ruby install.rb setup
ruby install.rb install
```

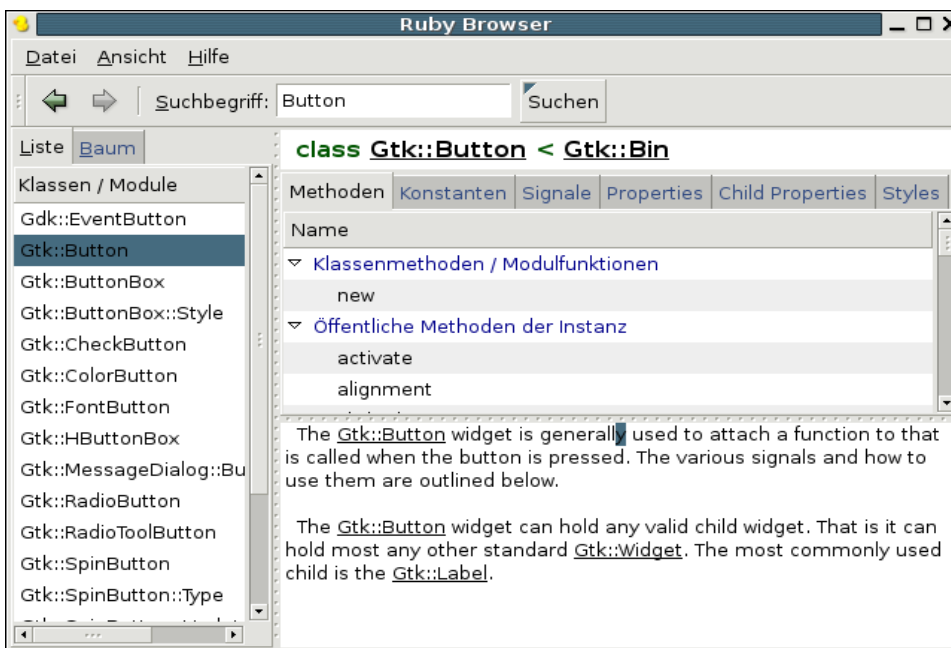
Aktualisieren

Weiters empfiehlt sich noch das Aktualisieren der Ruby/GTK-Dokumentation, du findest ein ständig aktualisiertes Paket unter derselben URL wie die Online-Dokumentation als so genannten „Nightly tarball“.

Verschiebe den Inhalt des entpackten Ordners unter Linux in den Ordner „/usr/share/rbbr/rd/“, unter Windows XP nach „c:\ruby\share\rbbr\rd“.

Starten

Du kannst den Dokumentations-Browser in Windows durch Eingabe von „c:\ruby\bin\ruby.exe c:\ruby\bin\rbbr.rbw“ starten, in Linux schlicht durch Eingabe von „rbbr“. Du kannst natürlich auch eine Verknüpfung mit diesem Ziel anlegen.



Da Ruby/GTK eine Anbindung an die Bibliothek GTK+ ist, kann in manchen Fällen die Original-Dokumentation für GTK+ eine Lücke bei der Ruby/GTK-Dokumentation stopfen helfen. Du findest die GTK+-Dokumentation hier: <http://www.gtk.org/api/>.

Beispielprogramme

Eine weitere sehr informative Anlaufstelle sind die zahlreichen Beispielprogramme, welche im Zuge der Installation von Ruby/GTK den Weg auf die Platte finden: in Windows findest du sie unter C:\ruby\samples\ruby-gnome2, in Ubuntu-Linux im Ordner /usr/share/doc/libgtk2-ruby/, dort im Ordner sample(s) oder example(s).

Mehrere Komponenten in dasselbe Fenster einfügen

Natürlich möchtest du häufig mehrere grafische Komponenten in dasselbe Fenster einfügen können. Dies geht nicht ohne weiteres. Wenn du versuchst, in ein Fenster bei einem bereits eingefügten Widget ein zusätzliches hinzuzufügen,

```
self.add( Entry.new )
self.add( Button.new )
```

wirst du eine Fehlermeldung erhalten:

```
Gtk-WARNING **: Attempting to add a widget with type GtkButton to a GtkWindow,
but as a GtkBin subclass a GtkWindow can only contain one widget at a time; it
already contains a widget of type GtkEntry
```

An der Tatsache, dass nur ein Widget das Fenster „bevölkern“ darf, kommst du nicht vorbei.

Jedoch ist es möglich, für dieses eine Widget einen Container zu verwenden, der imstande ist, weitere Widgets zu beheimaten. Der Container selbst bleibt dabei unsichtbar. Solche Container sind unter anderen VBox, HBox, Table und Layout.

VBox

Eine VBox ordnet die Widgets in vertikaler Ausrichtung an, es gibt auch die Möglichkeit, die Größe der Widgets zu beeinflussen, ich verweise dich dazu auf die Dokumentation und zeige hier ein einfaches Beispiel:

```
vbox.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)     #Membervariablen
(7)     @entry
(8)     @button
(9)
(10)    def initialize
(11)        super
(12)
(13)        self.set_title( "" )
(14)
(15)        vbox = VBox.new( true, 0 )
(16)
(17)        @entry = Entry.new
(18)        vbox.add( @entry )
(19)
(20)        @button = Button.new( "Start" )
(21)        @button.signal_connect( "clicked" ) do
(22)            on_button_clicked
```



```

(23)      end
(24)      vbox.add( @button )
(25)
(26)      self.add( vbox )
(27)      self.show_all
(28)  end
(29)
(30)  def on_button_clicked
(31)    set_title( @entry.text )
(32)  end
(33)
(34)end

```

Dem Konstruktor von VBox werden hier zwei Argumente übergeben: „true“ steht dafür, dass alle eingefügten Widgets die gleiche Größe haben, „0“, dass zwischen den Widgets kein Freiraum bleibt.

Die Methode „add“ (Zeilen 18 und 24) fügt die Elemente von oben beginnend in die VBox ein, die Methode „pack_start“ macht übrigens dasselbe, „pack_end“ fügt die Komponenten von unten beginnend ein.

Die Klasse „Entry“ stellt das standardmäßige Text-Eingabefeld dar. Ein Entry eignet sich für einzeilige Texteingaben.

Arbeitsauftrag: Schreibe ein Programm mit einem Entry, einem Button und einem Label. Beim Drücken des Buttons soll der Inhalt des Entry ins Label geschrieben werden.

HBox

Eine HBox wird ebenso gehandhabt, mit dem Unterschied, dass die eingefügten Widgets nebeneinander angeordnet werden.

```

hbox.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)   #Memvervariablen
(7)   @btn_eins
(8)   @btn_zwei
(9)
(10)  def initialize
(11)    super
(12)
(13)    hbox = HBox.new
(14)
(15)    @btn_eins = Button.new( "eins" )

```



```

(16)     @btn_eins.signal_connect( "clicked" ) do
(17)         on_btn_eins_clicked
(18)     end
(19)     hbox.add( @btn_eins )
(20)
(21)     @btn_zwei = Button.new( "zwei" )
(22)     @btn_zwei.signal_connect( "clicked" ) do
(23)         on_btn_zwei_clicked
(24)     end
(25)     hbox.add( @btn_zwei )
(26)
(27)     self.add( hbox )
(28)     self.show_all
(29) end
(30)
(31) def on_btn_eins_clicked
(32)     puts "Button eins wurde gedrueckt."
(33) end
(34)
(35) def on_btn_zwei_clicked
(36)     puts "Button zwei wurde gedrueckt."
(37) end
(38)
(39)end

```

Table

Ein Table kombiniert die Möglichkeiten von HBox und VBox, du kannst Komponenten gleichzeitig neben- und untereinander anordnen:

```

table.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)     def initialize
(7)         super
(8)
(9)         self.set_border_width( 3 )
(10)
(11)         tab = Table.new( 3, 3, true )
(12)         tab.set_column_spacings( 3 )
(13)         tab.set_row_spacings( 3 )
(14)
(15)         k = 1
(16)         for i in 0...3
(17)             for j in 0...3
(18)                 button = Button.new( k.to_s )
(19)                 button.signal_connect( "clicked" ) do
(20)                     | widget |
(21)                     on_button_clicked( widget )
(22)                 end

```



```

(23)         tab.attach( button, j, j + 1, i, i + 1 )
(24)         k += 1 #dasselbe wie k = k + 1
(25)     end
(26) end
(27)
(28) self.add( tab )
(29) self.set_title( "" )
(30) self.show_all
(31) end
(32)
(33) def on_button_clicked( w )
(34)     puts "Button " + w.label
(35) end
(36)
(37)end

```

In Zeile 11 wird ein Table mit drei Zeilen und ebenso vielen Spalten erzeugt.

Die Zeilen 9, 12 und 13 definieren die Abstände: „set_border_width“ wird aufs Window angewandt und sorgt für die Abstände der im Fenster enthaltenen Widgets zum Fensterrand. Die Methoden „set_column_spacings“ und „set_row_spacings“ definieren die Abstände zwischen den Spalten und Zeilen der Tabelle.

In Zeile 23 werden die Buttons dem Table mit der Methode „attach“ hinzugefügt. „attach“ verlangt 5 Argumente: zunächst das einzufügende Widget, dann folgen vier Koordinaten, wobei die ersten beiden für die horizontale, die beiden letzten für die vertikale Ausdehnung des Widgets verantwortlich zeichnen.

Ich erkläre das am Beispiel: 0, 1, 1, 2 sind die Koordinaten des Button mit der Aufschrift „4“, 2, 3, 1, 2 sind jene der „6“, 0, 3, 2, 3 wären die Koordinaten eines Widgets, welches über den Bereich der Buttons 7-9 aufgespannt würde.

Arbeitsauftrag: Programmieren Sie das bekannte Spiel Tic-Tac-Toe, es sollen zwei (menschliche) Spieler gegeneinander spielen können. Der Computer soll eine Meldung ausgeben, sobald das Spiel beendet ist.

Layout

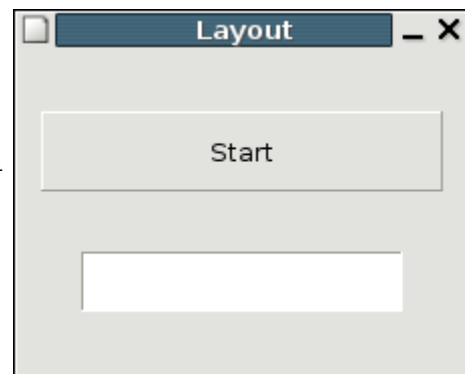
Eine weitere Möglichkeit, Widgets zu positionieren, bietet der Container Layout. Hier entscheidet der Programmierer punktgenau, wo jede Komponente

innerhalb des Fensters platziert wird.

```
layout.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)   def initialize
(7)     super
(8)
(9)     self.set_title( "Layout" )
(10)    self.set_size_request( 220, 160 )
(11)    self.resizable = false
(12)    self.set_window_position( Position::CENTER )
(13)
(14)    layout = Layout.new
(15)
(16)    button = Button.new( "Start" )
(17)    button.set_size_request( 200, 40 )
(18)    layout.put( button, 10, 30 )
(19)
(20)    entry = Entry.new
(21)    entry.set_size_request( 160, 30 )
(22)    layout.put( entry, 30, 100 )
(23)
(24)    self.add( layout )
(25)    self.show_all
(26)  end
(27)
(28)end
```

Die Widgets werden mit der Methode „put“ (Zeilen 18 und 22) ins Layout eingefügt, das zweite und dritte Argument der Methode definieren die Position des Widgets relativ zum linken oberen Eckpunkt des Fensters.

Arbeitsauftrag: Schreibe eine grafische Oberfläche mit neun Entry's, sodass die Elemente einer 3x3-Matrix eingegeben werden können. Verwende als Container das eben behandelte Layout.



Eine Callback-Methode für mehrere Signal-Handler

Im Beispiel vorhin (hbox.rb) wurde den beiden Buttons jeweils eine Methode für

das Ereignis „clicked“ zugeordnet.

Es ist jedoch auch möglich, beliebig viele Widgets auf die selbe Methode zu verweisen:

```
signale.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)   #Memvervariablen
(7)   @btn_eins
(8)   @btn_zwei
(9)
(10)  def initialize
(11)    super
(12)
(13)    hbox = HBox.new
(14)
(15)    @btn_eins = Button.new( "eins" )
(16)    @btn_eins.signal_connect( "clicked" ) do
(17)      on_btn_clicked
(18)    end
(19)    hbox.add( @btn_eins )
(20)
(21)    @btn_zwei = Button.new( "zwei" )
(22)    @btn_zwei.signal_connect( "clicked" ) do
(23)      on_btn_clicked
(24)    end
(25)    hbox.add( @btn_zwei )
(26)
(27)    self.add( hbox )
(28)    self.show_all
(29)  end
(30)
(31)  def on_btn_clicked
(32)    puts "Button ??? wurde gedrueckt."
(33)  end
(34)
(35)end
```

Verständlicherweise kann auf diese Weise innerhalb der Methode „on_btn_clicked“ nicht mehr nachvollzogen werden, welcher der beiden Buttons der Auslöser war, die Ausgabe ist also immer folgende, egal welcher der beiden Buttons gedrückt worden ist:

```
Button ??? wurde gedrueckt.
```

Es gibt allerdings die Möglichkeit, beim Eintreten eines Ereignisses (z.B. clicked) den Verursacher (hier Button eins oder zwei) festzuhalten und der Methode „on_btn_clicked“ zu übergeben:

```

signale2.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)     #Memvervariablen
(7)     @btn_eins
(8)     @btn_zwei
(9)
(10)    def initialize
(11)        super
(12)
(13)        hbox = HBox.new
(14)
(15)        @btn_eins = Button.new( "eins" )
(16)        @btn_eins.signal_connect( "clicked" ) do
(17)            | widget |
(18)            on_btn_clicked( widget )
(19)        end
(20)        hbox.add( @btn_eins )
(21)
(22)        @btn_zwei = Button.new( "zwei" )
(23)        @btn_zwei.signal_connect( "clicked" ) do
(24)            | widget |
(25)            on_btn_clicked( widget )
(26)        end
(27)        hbox.add( @btn_zwei )
(28)
(29)        self.add( hbox )
(30)        self.show_all
(31)    end
(32)
(33)    def on_btn_clicked( w )
(34)        puts "Button " + w.label + " wurde gedrueckt."
(35)    end
(36)
(37)end

```

Die Möglichkeit, das das Ereignis auslösende Widget „abzufangen“ (Zeilen 17 und 24) und es an die Methode zu übergeben (Zeilen 18 und 25), ermöglicht es, so manches Programm kürzer und übersichtlicher zu gestalten, da eine Methode auf diese Weise mehrere Widgets „bedienen“ kann.

Fenster schließen und Schließen verhindern

Es ist dir bei den bisherigen Beispielen vielleicht aufgefallen, dass das Fenster beim Schließen (Alt+F4, Drücken des Schließen-Kreuzes) zwar verschwindet,

das Programm aber trotzdem aktiv bleibt (Konsole wird nicht freigegeben).

Der Grund für dieses Verhalten ist, dass das Fenster nur in den Hintergrund gelegt, die Programmschleife (Gtk::main) jedoch nicht abgebrochen wird.

Im nächsten Beispiel sorgen wir dafür, dass Fenster und Programm geschlossen werden:

```
schliessen.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)   def initialize
(7)     super
(8)
(9)     self.signal_connect( "destroy" ) do
(10)       on_window_destroy
(11)     end
(12)
(13)     self.set_title( "" )
(14)     self.show
(15)   end
(16)
(17)   def on_window_destroy
(18)     main_quit
(19)   end
(20)
(21)end
```

„destroy“ (Zeile 9) ist das Ereignis, das beim Schließen des Fensters aufgerufen wird, im Beispiel wird beim Auslösen dieses Ereignisses die Methode „on_window_destroy“ aufgerufen, dort wird das Programm mit „main_quit“ beendet.

Dem Ereignis „destroy“ geht noch ein anderes Ereignis voraus, „delete_event“. Wird dort false zurückgegeben, nimmt der Vorgang den gewohnten Lauf (wenn du das Ereignis „delete_event“ nicht implementierst, wird standardmäßig false zurückgegeben), „true“ bricht das Schließen des Fensters ab, das Ereignis „destroy“ wird dann nicht ausgelöst.

In schliessen2.rb wird das Schließen des Fensters abgebrochen, das Programm kann nicht auf konventionelle Weise beendet werden:

```
schliessen2.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
```

```

(6)    def initialize
(7)      super
(8)
(9)      self.set_title( "" )
(10)     self.signal_connect( "delete_event" ) do
(11)       on_window_delete
(12)     end
(13)
(14)     self.show
(15)   end
(16)
(17)   def on_window_delete
(18)     return true
(19)   end
(20)
(21)end

```

Im zweiten Beispiel zu den MessageDialogen weiter unten (Seite 31) kannst du sehen, wie beim Beenden des Programms ein Bestätigungs-Dialog eingeblendet werden kann.

Etwas Farbe bitte

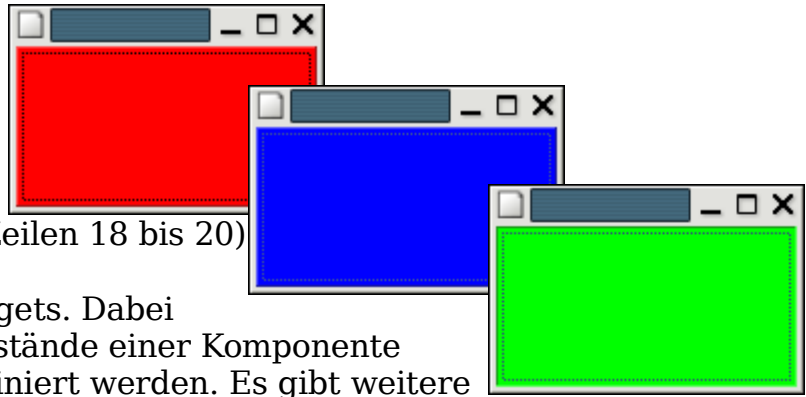
Du kannst deine Programme auch farbenprächtig gestalten und deine künstlerische Ader zum Vorschein bringen:

```

farben.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)   def initialize
(7)     super
(8)
(9)     self.set_title( "" )
(10)    self.set_size_request( 150, 80 )
(11)    self.signal_connect( "destroy" ) { main_quit }
(12)
(13)    rot = Gdk::Color.new( 65535, 0, 0 )
(14)    gruen = Gdk::Color.new( 0, 65535, 0 )
(15)    blau = Gdk::Color.new( 0, 0, 65535 )
(16)
(17)    button = Button.new
(18)    button.modify_bg( StateType::NORMAL, rot )
(19)    button.modify_bg( STATE_ACTIVE, gruen )
(20)    button.modify_bg( 2, blau )
(21)
(22)    self.add( button )
(23)    self.show_all
(24)  end
(25)

```

(26)end



Die Methode „modify_bg“ (Zeilen 18 bis 20) ermöglicht das Ändern der Hintergrundfarbe eines Widgets. Dabei können für verschiedene Zustände einer Komponente unterschiedliche Farben definiert werden. Es gibt weitere Methoden, um das Aussehen der Widgets zu manipulieren, mehr dazu in der Dokumentation.

Wie du der Dokumentation entnehmen kannst, gibt es insgesamt fünf StateTypes, also mögliche Zustände eines Widgets:

STATE_NORMAL	=	StateType::NORMAL	=	0
STATE_ACTIVE	=	StateType::ACTIVE	=	1
STATE_PRELIGHT	=	StateType::PRELIGHT	=	2
STATE_SELECTED	=	StateType::SELECTED	=	3
STATE_INSENSITIVE	=	StateType::INSENSITIVE	=	4

„STATE_NORMAL“ beispielsweise steht für dasselbe wie „StateType::NORMAL“, ebenso die „0“. Du kannst also, wie im Programm oben geschehen, jede der drei Konstanten äquivalent verwenden.

Hinweis: Bei Verwendung einiger Desktop-Themen kann es vorkommen, dass die Farbe einiger Widgets vom Thema überlagert und somit nicht angezeigt wird. Für DrawingAreas etwa, die weiter unten behandelt werden, gilt diese Einschränkung nicht. Unter Windows kannst du Abhilfe schaffen, indem du mit dem Theme-Selector von Gtk+ für Gtk-Programme ein von „MS-Windows“ abweichendes Thema einstellst.

Weitere Widgets

Die Grafikbibliothek GTK+ stellt neben den bereits behandelten Widgets Gtk::Window, Gtk::Button, Gtk::Label, Gtk::Entry und Gdtk::Color viele weitere Komponenten zur Verfügung, die sich für unterschiedliche Aufgaben anbieten.

Ich werde hier nicht alle vorstellen, sondern beschränke mich auf jene, die am gebräuchlichsten sind. Für eine vollständige Auflistung und Beschreibung verweise ich dich einmal mehr auf die Dokumentation oder auch die Tutorials auf den offiziellen Seiten von Ruby-GNOME2.

Zu Beginn wirst du dich vielleicht des öfteren fragen, welche Komponenten für deine ganz speziellen Wünsche am besten geeignet sind.

Dazu zwei Ansätze: zum einen wirst du mit etwas Übung sehr schnell über die nötige Erfahrung verfügen und zum zweiten ist es auch für einen geübten Codeschaufler immer wieder sehr hilfreich, bestehende Programme aus der Perspektive des neugierigen Programmierers zu betrachten.

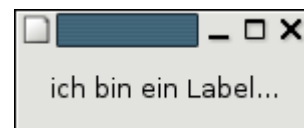
Eines noch vorne weg: ich erkläre den Code in den Beipielprogrammen nur in Auszügen, zum Teil ist der Code innerhalb des Quellcodes kommentiert, aber zum Großteil bist du selbst gefordert.

Dies ist von Vorteil: du lernst sehr viel, wenn du den Code durchforstest und zudem eigene, abgewandelte Programme schreibst.

Label

Labels dienen vor allem der Ausgabe von Text.

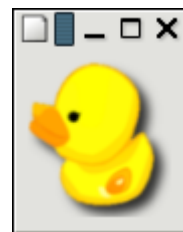
```
label.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)   def initialize
(7)     super
(8)
(9)     self.set_title( "" )
(10)    self.signal_connect( "destroy" ) do
(11)      main_quit
(12)    end
(13)
(14)    label = Label.new
(15)    label.set_label( "ich bin ein Label..." )
(16)
(17)    self.add( label )
(18)    self.show_all
(19)  end
(20)
(21)end
```



Image

Ein Image-Objekt kann ein Bild anzeigen:

```
bild.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)     def initialize
(7)         super
(8)
(9)         self.set_title( "" )
(10)        self.signal_connect( "destroy" ) { main_quit }
(11)
(12)        image = Image.new( "duck.png" )
(13)
(14)        self.add( image )
(15)        self.show_all
(16)    end
(17)
(18)end
```



Wenn die Größe eines Bildes geändert werden soll, kannst du es unter Zuhilfenahme von `Gdk::Pixbuf` skalieren, das Seitenverhältnis bleibt dabei unangetastet (die kleinere Zahl zählt, wenn das Seitenverhältnis des Bildes nicht dem Verhältnis der übergebenen Werte entspricht).

```
pixbuf = Gdk::Pixbuf.new( "duck.png", 50, 40 )
icon = Image.new( pixbuf )
hbox.pack_start( icon, false, false, 10 )
```

Du kannst allerdings auch Größe und Seiterverhältnis ändern:

```
pixbuf = Gdk::Pixbuf.new( "duck.png" )
pixbuf = pixbuf.scale( 50, 80 )
```

Weiters ist es möglich, einen Ausschnitt eines Bildes zu „extrahieren“:

```
pixbuf_quelle = Gdk::Pixbuf.new( "duck.png" )
pixbuf_ausschnitt = Gdk::Pixbuf.new( pixbuf_quelle, 0, 0, 20, 30 )
```

Dieser Konstruktor nimmt zunächst ein `Gdk::Pixbuf`-Objekt entgegen. Das zweite und dritte Argument sind der Startpunkt des Ausschnittes (x- und y-Richtung), die beiden weiteren Werte stellen die Größe des Ausschnitts dar.

EventBox

Es gibt Widgets, welche über keine Signale verfügen, das heißt, es kann nicht auf Ereignisse (Events) wie einen Mausklick reagiert werden. Davon betroffen sind neben anderen die gerade angesprochenen Labels und Images und die etwas später behandelte DrawingArea.

Da die Verfügbarkeit von Signalen in Zusammenhang mit den davon betroffenen Widgets dennoch häufig wünschenswert ist, können solche Widgets in eine EventBox eingebettet werden.

Im nächsten Beispiel soll das Image die Position wechseln, wenn darauf geklickt wird:

bild2.rb

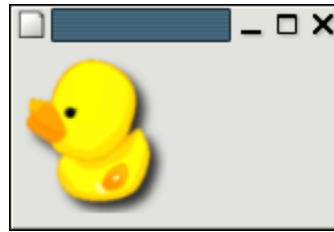
```
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)   #Membervariablen
(7)   @eb1
(8)   @eb2
(9)   @image
(10)  @pos #aktuelle Position des Bildes
(11)
(12)  def initialize
(13)    super
(14)
(15)    self.set_title( "" )
(16)    self.signal_connect( "destroy" ) { main_quit }
(17)
(18)    @image = Image.new( "duck.png" )
(19)    @pos = "eb1"
(20)
(21)    hbox = HBox.new( true, 0 )
(22)
(23)    @eb1 = EventBox.new
(24)    @eb1.set_name( "eb1" )
(25)    @eb1.signal_connect( "button_press_event" ) do
(26)      | widget, event |
(27)        on_eb_button_press_event( widget )
(28)    end
(29)    hbox.pack_start( @eb1 )
(30)
(31)    @eb2 = EventBox.new
(32)    @eb2.set_name( "eb2" )
(33)    @eb2.signal_connect( "button_press_event" ) do
(34)      | widget, event |
(35)        on_eb_button_press_event( widget )
(36)    end
(37)    hbox.pack_start( @eb2 )
(38)
(39)    @eb1.add( @image )
(40)
```



```

(41)         self.add( hbox )
(42)         self.show_all
(43)     end
(44)
(45)     def on_eb_button_press_event( w )
(46)         if @pos == w.name
(47)             if w.name == "eb1"
(48)                 @eb1.remove( @image )
(49)                 @eb2.add( @image )
(50)                 @pos = "eb2"
(51)             else
(52)                 @eb2.remove( @image )
(53)                 @eb1.add( @image )
(54)                 @pos = "eb1"
(55)             end
(56)         end
(57)     end
(58)
(59) end

```



In der vom Ereignis „button_press_event“ ausgelösten Methode „on_eb_button_press_event“ wird zunächst geprüft, ob auf die EventBox mit dem Image geklickt worden ist (Zeile 46).

Dazu trägt die Instanzvariable „@pos“ als Wert immer den Namen jener EventBox, welche das Bild enthält, zu Beginn ist der Inhalt von @pos also „eb1“. Auch die EventBoxes tragen ihren Namen (Zeilen 24 und 32) zu genau diesem Zweck.

Für den Fall, dass auf die leere EventBox geklickt wurde, wird kein weiterer Code ausgeführt. Wird hingegen auf die EventBox mit dem Image geklickt, wird überprüft, welche EventBox das Ereignis ausgelöst hat (Zeilen 47 bzw. 51). Es folgen das Löschen des Images aus der einen und das Einfügen in die andere Eventbox. Der Inhalt von „@pos“ wird auf den neuen Wert gesetzt.

Arbeitsauftrag: Schreibe das Programm so um (oder schreibe es besser neu), dass neun Positionen entstehen und das Image zufällig seine Position wechselt, wenn der Mauszeiger darauf positioniert wird.

RadioButton

radiobutton.rb

```

(1) require 'gtk2'
(2) include Gtk
(3)
(4)
(5) class Fenster < Window
(6)
(7)     #Membervariablen
(8)     @rb1
(9)     @rb2
(10)    @rb3
(11)
(12)    def initialize
(13)        super
(14)
(15)        self.set_title( "" )
(16)        self.signal_connect( "destroy" ) { main_quit }
(17)
(18)        #VBox mit RadioButtons
(19)        vbox = VBox.new( false, 0 )
(20)        vbox.set_border_width( 20 )
(21)
(22)        @rb1 = RadioButton.new( "klein" )
(23)        vbox.pack_start( @rb1 )
(24)
(25)        @rb2 = RadioButton.new( @rb1, "mittel" )
(26)        @rb2.set_active( true )
(27)        vbox.pack_start( @rb2 )
(28)
(29)        @rb3 = RadioButton.new( @rb1, "groß" )
(30)        vbox.pack_start( @rb3 )
(31)
(32)        #HBox mit Buttons
(33)        hbox = HBox.new( true, 10 )
(34)        hbox.set_border_width( 10 )
(35)
(36)        btn_start = Button.new( Stock::EXECUTE )
(37)        btn_start.signal_connect( "clicked" ) do
(38)            on_btn_start_clicked
(39)        end
(40)        hbox.pack_start( btn_start )
(41)
(42)        btn_stop = Button.new( Stock::QUIT )
(43)        btn_stop.signal_connect( "clicked" ) { main_quit }
(44)        hbox.pack_start( btn_stop )
(45)
(46)        vbox.pack_start( hbox )
(47)
(48)        self.add( vbox )
(49)        self.show_all
(50)    end
(51)
(52)    def on_btn_start_clicked
(53)        if @rb1.active?
(54)            puts "klein"
(55)        elsif @rb2.active?
(56)            puts "mittel"
(57)        else
(58)            puts "gross"
(59)        end
(60)    end

```



```
(61)
(62)end
```

Im Konstruktor des ersten RadioButtons (Zeile 22) steht nur der Bezeichner des RadioButtons, die weiteren RadioButtons (Zeilen 25 und 29) enthalten zusätzlich noch den Bezug zum ersten RadioButton. Dies ist nötig, damit nur einer von den dreien gleichzeitig aktiviert sein kann.

Bei den Buttons (Zeilen 36 und 42) siehst du neben der Beschriftung noch ein Symbol. Es gibt eine große Anzahl solcher vorgefertigter Buttons, mehr dazu in der Dokumentation (Gtk::Stock).

Es können jedoch auch ganz individuelle Icons eingefügt werden, dazu gleich mehr.

Widgets mit individuellem Icon versehen

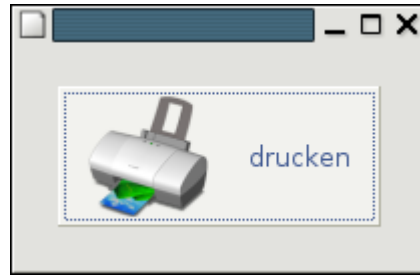
Du kannst Widgets wie beispielsweise Buttons eine eigene Note verleihen, indem du beliebige Bilder (Icons) einfügst:

```
drucken.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)   def initialize
(7)     super
(8)
(9)     self.set_title( "" )
(10)    self.set_border_width( 20 )
(11)    self.set_window_position( Position::MOUSE )
(12)    self.signal_connect( "destroy" ) { main_quit }
(13)
(14)    hbox = HBox.new( false, 0 )
(15)
(16)    icon = Image.new( "printer.png" )
(17)    hbox.pack_start( icon, false, false, 10 )
(18)
(19)    label = Label.new( "drucken" )
(20)    hbox.pack_start( label, false, false, 10 )
(21)
(22)    button = Button.new
(23)    button.add( hbox )
(24)
(25)    self.add( button )
```

```

(26)     self.show_all
(27) end
(28)
(29)end

```



Der Button dient hier als Container: es wird eine HBox mit einem Image und einem Label vorbereitet, die HBox wird schließlich auf den Button gelegt (Zeile 23).

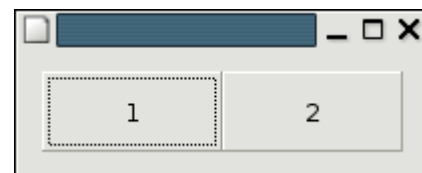
Tooltip

Ein Tooltip erscheint, wenn der Mauszeiger kurz auf einem Widget verweilt. Tooltips dienen dazu, zusätzliche Informationen anzuzeigen.

```

tooltip.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)   def initialize
(7)     super
(8)
(9)     self.set_title( "" )
(10)    self.set_size_request( 200, 60 )
(11)    self.signal_connect( "destroy" ) { main_quit }
(12)    #Abstand der Widgets zum Fenster
(13)    self.set_border_width( 10 )
(14)
(15)    hbox = HBox.new( true, 0 )
(16)    tooltips = Tooltips.new
(17)
(18)    btn1 = Button.new( "1" )
(19)    tooltips.set_tip( btn1, "erster Button", "" )
(20)    hbox.pack_start( btn1 )
(21)
(22)    btn2 = Button.new( "2" )
(23)    tooltips.set_tip( btn2, "zweiter Button", "" )
(24)    hbox.pack_start( btn2 )
(25)
(26)    self.add( hbox )
(27)    self.show_all
(28)  end
(29)
(30)end

```



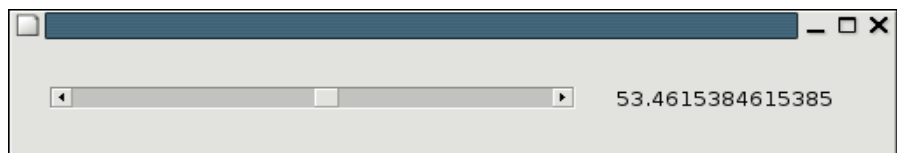
Das dritte Argument der Methode `set_tip` (Zeilen 19 und 23) stellt einen privaten Tip dar, über dessen Nutzen ich im Unklaren geblieben bin. Du kannst anstatt des leeren Strings auch „nil“ schreiben.

Scrollbar

Es gibt HScrollbars und VScrollbars, beide unterscheiden sich lediglich in ihrer Ausrichtung (horizontal und vertikal).

scrollbar.rb

```
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)     #Membervariablen
(7)     @lbl_anzeige
(8)     @hsb
(9)
(10)     def initialize
(11)         super
(12)
(13)         self.set_title( "" )
(14)         self.signal_connect( "destroy" ) { main_quit }
(15)
(16)         hbox = HBox.new( false, 10 )
(17)         hbox.set_border_width( 20 )
(18)
(19)         @hsb = HScrollbar.new
(20)         @hsb.set_size_request( 300, 30 )
(21)         @hsb.set_range( 0, 100 )
(22)         @hsb.set_increments( 1, 10 )
(23)         @hsb.set_value( 30 )
(24)         @hsb.signal_connect( "value_changed" ) { on_hsb_value_changed }
(25)         hbox.pack_start( @hsb )
(26)
(27)         @lbl_anzeige = Label.new( "30.0" )
(28)         @lbl_anzeige.set_size_request( 150, 30 )
(29)         hbox.pack_start( @lbl_anzeige )
(30)
(31)         self.add( hbox )
(32)         self.show_all
(33)     end
(34)
(35)     def on_hsb_value_changed
(36)         @lbl_anzeige.set_label( @hsb.value.to_s )
(37)     end
(38)
(39) end
```



Die Methode

„set_increments“ (Zeile 22) dient dazu, dass der Scrollbar seine Position um den als erstes Argument übergebenen Wert verändert, wenn einer der Pfeile des Scrollbars gedrückt wird. Das zweite Argument gibt den Wert vor, um den sich der Scrollbar seine Position verändert, wenn man hineinklickt.

Mit dem folgenden Programm kannst du eine der drei Grundfarben auswählen und deren Intensität verändern:

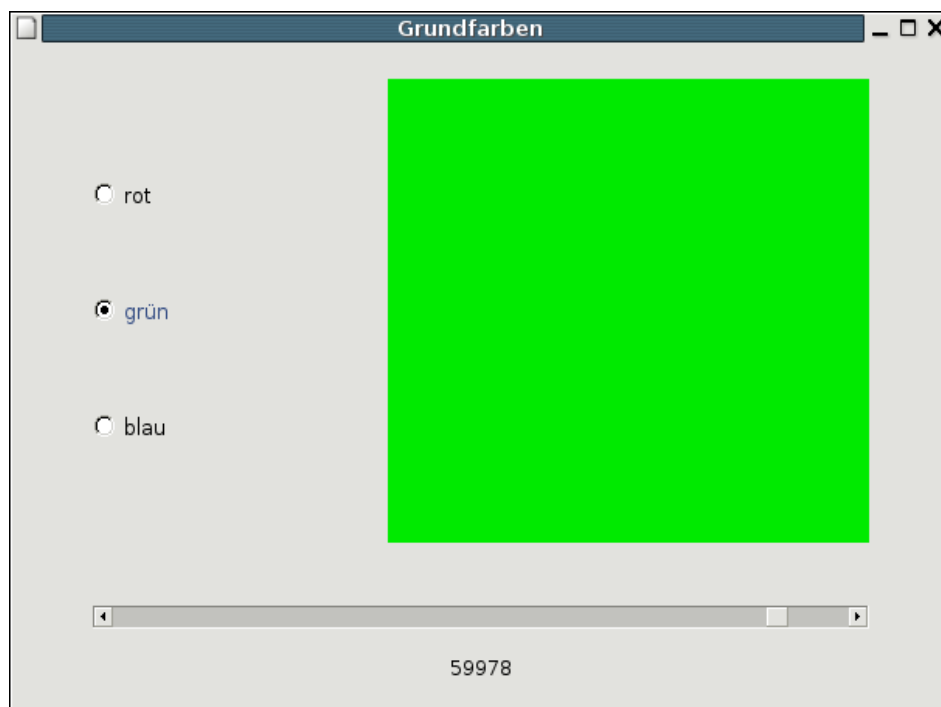
grundfarben.rb

```
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)   #Membervariablen
(7)   @rb_rot; @rb_gruen; @rb_blau #RadioButtons
(8)   @hsb #ScrollBar
(9)   @da #DrawingArea
(10)  @lbl_ausgabe #Label
(11)
(12)  def initialize
(13)    super
(14)
(15)    self.set_title( "Grundfarben" )
(16)    self.set_size_request( 600, 400 )
(17)    self.signal_connect( "destroy" ) { main_quit }
(18)
(19)    layout = Layout.new
(20)
(21)    #RadioButtons
(22)    @rb_rot = RadioButton.new( "rot" )
(23)    @rb_rot.signal_connect( "toggled" ) { on_rb_toggled }
(24)    @rb_rot.set_active( true )
(25)    layout.put( @rb_rot, 50, 80 )
(26)
(27)    @rb_gruen = RadioButton.new( @rb_rot, "grün" )
(28)    @rb_gruen.signal_connect( "toggled" ) { on_rb_toggled }
(29)    layout.put( @rb_gruen, 50, 150 )
(30)
(31)    @rb_blau = RadioButton.new( @rb_rot, "blau" )
(32)    @rb_blau.signal_connect( "toggled" ) { on_rb_toggled }
(33)    layout.put( @rb_blau, 50, 220 )
(34)
(35)    #ScrollBar
(36)    @hsb = HScrollbar.new
(37)    @hsb.set_size_request( 500, 30 )
(38)    @hsb.set_range( 0, 65535 )
(39)    @hsb.set_increments( 100, 1000 )
(40)    @hsb.set_value( 0 )
(41)    @hsb.signal_connect( "value_changed" ) { on_hsb_value_changed }
(42)    layout.put( @hsb, 50, 330 )
(43)
(44)    #DrawingArea
(45)    @da = DrawingArea.new
(46)    @da.set_size_request( 310, 280 )
(47)    @da.modify_bg( STATE_NORMAL, Gdk::Color.new( 0, 0, 0 ) )
(48)    layout.put( @da, 240, 20 )
(49)
(50)    #Label
(51)    @lbl_ausgabe = Label.new( "0.0" )
(52)    @lbl_ausgabe.set_size_request( 500, 50 )
(53)    layout.put( @lbl_ausgabe, 50, 350 )
(54)
(55)    self.add( layout )
(56)    self.show_all
(57)  end
(58)
```

```

(59) def on_hsb_value_changed
(60)     #Wert des Scrollbars abfragen
(61)     wert = @hsb.value.to_i
(62)     #Pruefen, welcher RadioButton aktiv ist
(63)     #und Farbe entsprechend setzen
(64)     if @rb_rot.active?
(65)         farbe = Gdk::Color.new( wert, 0, 0 )
(66)     elsif @rb_gruen.active?
(67)         farbe = Gdk::Color.new( 0, wert, 0 )
(68)     else
(69)         farbe = Gdk::Color.new( 0, 0, wert )
(70)     end
(71)     #Farbe der DrawingArea festsetzen
(72)     @da.modify_bg( STATE_NORMAL, farbe )
(73)     #Wert des Scrollbars ins Label schreiben
(74)     @lbl_ausgabe.set_label( wert.to_s )
(75) end
(76)
(77) def on_rb_toggled #anderer RadioButton gewaehlt
(78)     @hsb.set_value( 0 )
(79) end
(80)
(81)end

```



Arbeitsauftrag: Schreibe ein Programm, mit dem die drei Grundfarben mit drei Scrollbars gemischt werden können. Gib das Resultat in einer DrawingArea aus.

Dialoge

Dialoge im Allgemeinen haben den Zweck, zwischen Benutzer und Programm einen Informationsaustausch zu ermöglichen. Dabei bieten sich für spezielle Vorhaben unterschiedliche Dialoge an.

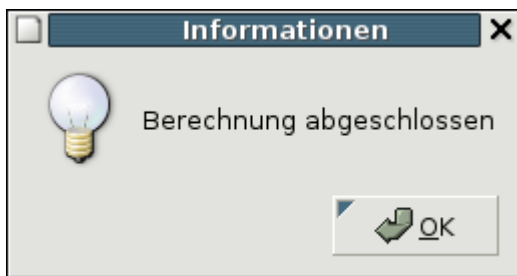
MessageDialog

MessageDialogs dienen vorwiegend dem Ausgeben von Informationen, Warnungen oder Fehlermeldungen, können aber auch für das Einlesen von Benutzereingaben zurechtgebogen werden (siehe Beispiel 3).

Der Konstruktor eines MessageDialogs verlangt die Übergabe von einer Hand voll Parametern:

```
Gtk::MessageDialog.new( parent, flags, type, buttons, message )
```

A) „parent“ definiert, zu welchem Window ein Dialog gehört. Wird der MessageDialog aus einer Klasse heraus aufgerufen, kann hier „self“ verwendet werden, „nil“, wenn keine Zuordnung gewünscht wird.



B) „flags“: dieser Wert beeinflusst das Verhalten des Dialogs in Bezug auf das Elternfenster.

Als Flag sind die folgenden Werte möglich:

Dialog::Flags::MODAL = 0

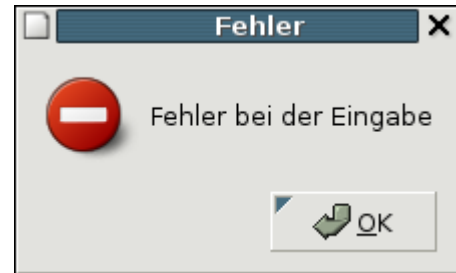
Dialog::Flags::DESTROY_WITH_PARENT = 1

Dialog::Flags::NO_SEPARATOR = 2

Anstelle dieser nicht gerade kurzen Ausdrücke kannst du die Konstanten auch in Zahlenform schreiben: „0“ etwa drückt dasselbe aus wie „Dialog::Flags::MODAL“. Dieses Flag veranlasst, dass das Hauptfenster nicht reagiert, solange der Dialog geöffnet ist.

C) „type“ gibt den Typ des MessageDialogs an:

MessageDialog::Type::INFO = 0
MessageDialog::Type::WARNING = 1
MessageDialog::Type::QUESTION = 2
MessageDialog::Type::ERROR = 3



D) „buttons“:



MessageDialog::ButtonType::NONE = 0
MessageDialog::ButtonType::OK = 1
MessageDialog::ButtonType::CLOSE = 2
MessageDialog::ButtonType::CANCEL = 3
MessageDialog::ButtonType::YES_NO = 4
MessageDialog::ButtonType::OK_CANCEL = 5

Rückgabewerte der Buttons (Response-ID):

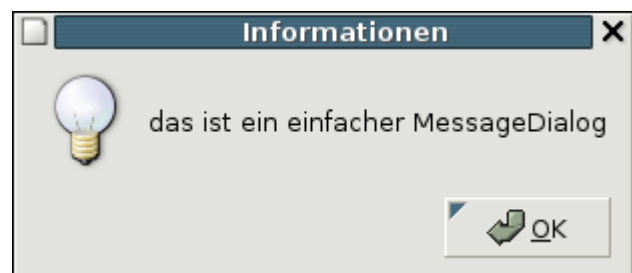
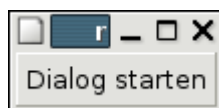
YES_NO: -8 (YES) und -9 (NO)
OK_CANCEL: -5 (OK) und -6 (CANCEL)

E) „message“: Text, welcher im MessageDialog erscheinen soll.

Erstes Beispiel: in einem MessageDialog Informationen anzeigen

message_dialog.rb

```
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)   def initialize
(7)     super
(8)
(9)     button = Button.new( "Dialog starten" )
(10)    button.signal_connect( "clicked" ) { on_button_clicked }
(11)    self.add( button )
(12)
```



```

(13)         self.show_all
(14)
(15)     end
(16)
(17)     def on_button_clicked
(18)         msg = "das ist ein einfacher MessageDialog"
(19)         dialog = MessageDialog.new( self, 0, 0, 1, msg )
(20)         dialog.run
(21)         dialog.destroy
(22)     end
(23)
(24)end

```

Wie du siehst, hält sich der Programmieraufwand, um einen simplen Ausgabedialog zu erstellen, denkbar gering (Zeilen 18 bis 21). Beim Klicken auf „OK“, wird der Dialog automatisch geschlossen.

Zweites Beispiel: in einem MessageDialog eine Auswahl präsentieren

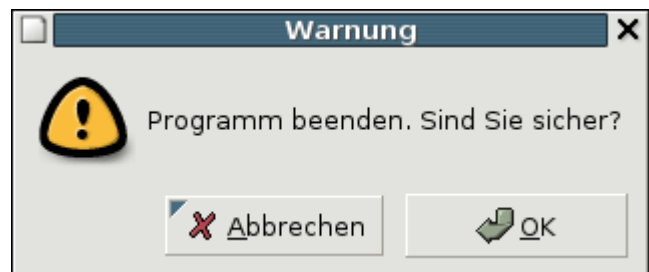
Wenn statt einem zwei Buttons, „OK“ und „Abbrechen“ eingefügt werden, sollte ausgewertet werden können, welcher von den beiden gedrückt wurde.

In `schliessen3.rb` wird dem Benutzer beim Schließen des Fensters mittels eines MessageDialogs die Möglichkeit geboten, seine Entscheidung zu revidieren.

```

schliessen3.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)     def initialize
(7)         super
(8)
(9)         self.set_title( "" )
(10)        self.set_size_request( 250, 200 )
(11)        self.signal_connect( "delete_event" ) { on_window_delete }
(12)        self.signal_connect( "destroy" ) { on_window_destroy }
(13)
(14)        self.show_all
(15)    end
(16)
(17)    def on_window_delete
(18)        msg = "Programm beenden. Sind Sie sicher?"
(19)        #MessageDialog
(20)        dialog = MessageDialog.new( self, 0, 1, 5, msg )
(21)        dialog.run do
(22)            |response|
(23)            if response == -5
(24)                dialog.destroy
(25)                return false

```



```

(26)         elsif response == -6
(27)             dialog.destroy
(28)             return true
(29)         end
(30)     end
(31)     #MessageDialog Ende
(32) end
(33)
(34) def on_window_destroy
(35)     main_quit
(36) end
(37)
(38)end

```

Das Klicken der Buttons „Abbrechen“ und „OK“ liefert unterschiedliche Response-IDs, „Abbrechen“ gibt den Wert „-6“, „OK“ gibt „-5“ zurück. Ab Zeile 23 wird dieser Wert abgefragt und entsprechend reagiert.

Im dritten Beispiel zu den MessageDialogs wollen wir noch der Frage nachgehen, wie der Benutzer über einen MessageDialog Daten eingeben kann:

Beim Starten des Programms, noch vor Erscheinen des Fensters, wird mittels eines MessageDialogs vom Benutzer der Titel des Fensters abgefragt:

```

titel_abfragen.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)     #Membervariablen
(7)     @titel
(8)
(9)     def initialize
(10)         super
(11)
(12)         self.set_title( "" )
(13)         self.set_size_request( 250, 200 )
(14)         self.signal_connect( "destroy" ) { main_quit }
(15)
(16)         #MessageDialog
(17)         msg = "Fenstertitel:"
(18)         dialog = MessageDialog.new( self, 0, 2, 5, msg )
(19)         entry = Entry.new
(20)         entry.show
(21)         dialog.vbox.pack_start( entry )
(22)         dialog.run do
(23)             | response |
(24)             if response == -5

```



```

(25)         @titel = entry.text
(26)     elsif response == -6
(27)         @titel = "kein Titel..."
(28)     end
(29) end
(30) dialog.destroy
(31) #MessageDialog Ende
(32)
(33) self.set_title( @titel )
(34) self.show_all
(35) end
(36)
(37)end

```



Dialog

MessageDialogs sind im wesentlichen spezialisierte Dialogs und sind einfacher und schneller zu erstellen als Dialogs.

Dafür können Dialogs besser an individuelle Bedürfnisse angepasst werden als MessageDialogs.

Ein Beispiel:

```

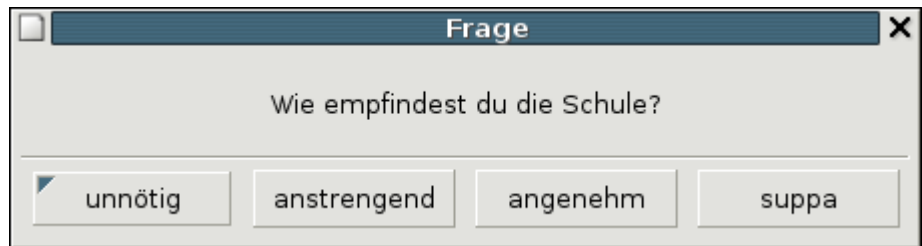
umfrage.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)     #Membervariablen
(7)     @label
(8)
(9)     def initialize
(10)         super
(11)
(12)         self.set_title( "Umfrage" )
(13)         self.set_size_request( 180, 70 )
(14)         self.signal_connect( "destroy" ) { main_quit }
(15)
(16)         vbox = VBox.new( true, 0 )
(17)
(18)         button = Button.new( "Frage" )
(19)         button.signal_connect( "clicked" ) { on_button_clicked }
(20)         vbox.pack_start( button )
(21)
(22)         @label = Label.new( "Auswertung" )
(23)         vbox.pack_start( @label )
(24)
(25)         self.add( vbox )
(26)         self.show_all
(27)     end
(28)

```

```

(29) def on_button_clicked
(30)     dialog = Dialog.new
(31)     dialog.set_title( "Frage" )
(32)     label = Label.new( "\nWie empfindest du die Schule?\n" )
(33)     label.show
(34)     dialog.vbox.pack_start( label )
(35)     dialog.add_button( "unnötig", 0 )
(36)     dialog.add_button( "anstrengend", 1 )
(37)     dialog.add_button( "angenehm", 2 )
(38)     dialog.add_button( "suppa", 3 )
(39)     dialog.run do
(40)         | response |
(41)         if response == 0
(42)             text = "Schule ist unnötig"
(43)         elsif response == 1
(44)             text = "Schule ist anstrengend"
(45)         elsif response == 2
(46)             text = "Schule ist angenehm"
(47)         elsif response == 3
(48)             text = "Schule ist suppa"
(49)         end
(50)         @label.set_text( text )
(51)         dialog.destroy
(52)     end
(53) end
(54)
(55)end

```



Ein Dialog besteht aus zwei Bereichen: wie bei den MessageDialogs einer integrierten VBox (namens „vbox“) im oberen Teil und einem Bereich für die Buttons unten.

Widgets, die im oberen Bereich angezeigt werden sollen, kannst du einfach der vorhandenen VBox hinzufügen:

```
dialog.vbox.pack_start( label )
```

Dem muss allerdings noch die Methode

```
label.show
```

vorausgehen, da die Widgets in der VBox sonst nicht angezeigt werden.

Die Buttons werden mittels

```
dialog.add_button( "suppa", 3 )
```

automatisch im unteren Bereich des Dialog platziert. Das erste Argument im Methodenaufruf stellt die Beschriftung des Buttons dar, das zweite die so genannte Response-ID, das ist jener Wert, der beim Drücken des Buttons zurückgegeben wird.

Jetzt ist der Dialog komplett und muss nur noch gestartet werden:

```
dialog.run
```

Diese Methode sorgt dafür, dass der Dialog angezeigt und der restliche Programmablauf gestoppt wird, solange der Dialog aktiv ist.

Mit der Methode `dialog.destroy` wird der Dialog beendet.

Wir fahren fort mit vier weiteren spezialisierten Dialogs: `AboutDialog`, `FileSelectionDialog`, `ColorSelectionDialog` und `FontSelectionDialog`.

AboutDialog

Ein `AboutDialog` ist ein vorgefertigter, spezialisierter Dialog, um Informationen zum Programm, zur Lizenz, zu Autor, Webseite, e-Mail-Adresse und weitere Informationen anzuzeigen.

info_dialog.rb

```
(1)#!/usr/bin/env ruby
(2)
(3)require 'gtk2'
(4)include Gtk
(5)
(6)class Fenster < Window
(7)
(8)  def initialize
(9)    super
(10)    self.signal_connect( "destroy" ) { main_quit }
(11)
(12)    btn_info = Button.new( "Info" )
(13)    btn_info.signal_connect( "clicked" ) { on_btn_info_clicked }
(14)    self.add( btn_info )
(15)
(16)    self.show_all
(17)
(18)  end
(19)
(20)  def on_btn_info_clicked
(21)    a = AboutDialog.new
(22)    a.artists = ["Grafiker 1 <no1@abc>", "Grafiker 2 <no2@abc>"]
(23)    a.authors = ["Programmautor 1 <no1@abc>", "Programmautor 2 <no2@abc>"]
(24)    a.comments = "Beispiel für einen AboutDialog"
(25)    a.copyright = "Copyright (C) 2005 Franz Burgmann"
(26)    a.documenters = ["Documenter 1 <no1@abc>", "Documenter 2 <no2@abc>"]
(27)    a.license = "Lizenzbestimmungen"
(28)    a.logo = Gdk::Pixbuf.new( "bild.png" )
(29)    a.name = "AboutDialog-Beispiel"
(30)    a.translator_credits = "Übersetzer 1 <no1@abc>\nÜbersetzer 2 <no2@abc>"
(31)    a.version = "0.1"
(32)    a.website = "http://meine_webseite.abc"
```



```

(33)     a.website_label = "Titel der Webseite"
(34)
(35)     a.set_modal( true )
(36)     a.show_all
(37) end
(38)
(39)end

```



FileSelectionDialog

FileSelectionDialog stellt den betriebssystemspezifischen Datei-Dialog dar. Er wird zum Öffnen und Speichern von Dateien benutzt:

```

file_selection.rb
(1) #!/usr/bin/env ruby
(2)
(3) require 'gtk2'
(4) include Gtk
(5)
(6) class Fenster < Window
(7)
(8)   def initialize
(9)     super
(10)
(11)     self.set_title( "" )
(12)     self.set_border_width( 20 )
(13)     self.set_window_position( Position::MOUSE )
(14)     self.signal_connect( "destroy" ) { main_quit }
(15)
(16)     button = Button.new( "Datei öffnen" )
(17)     button.signal_connect( "clicked" ) { on_button_clicked }
(18)
(19)     self.add( button )
(20)     self.show_all
(21)   end
(22)
(23)   def on_button_clicked
(24)     fs = FileSelection.new( "Datei öffnen" )
(25)     fs.set_modal( self )
(26)     fs.ok_button.signal_connect( "clicked" ) do
(27)       puts "gewählte Datei: #{ fs.filename }"
(28)     end
(29)     fs.cancel_button.signal_connect( "clicked" ) do
(30)       fs.destroy

```

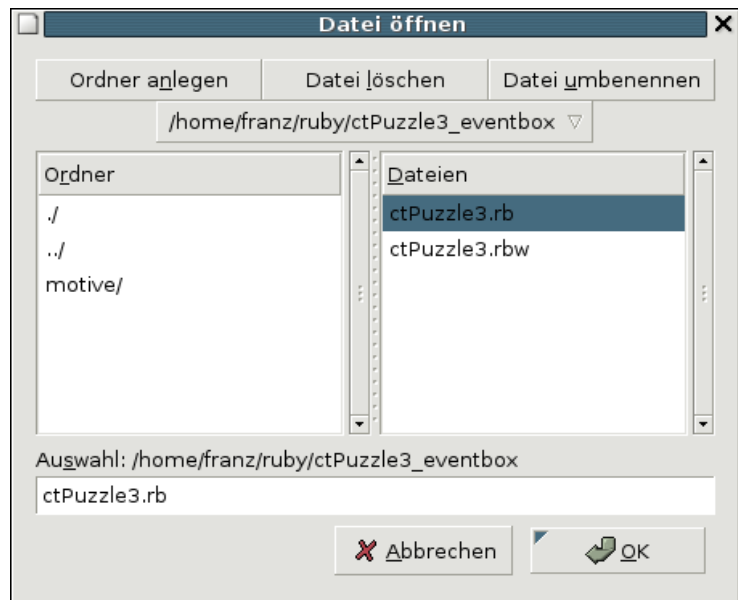
```

(31)      end
(32)      fs.show_all
(33)  end
(34)
(35)end

```

In Zeile 27 kommt innerhalb des Strings der Ausdruck „#{ fs.filename }“ vor. Ausdrücke, die sich zwischen „#{“ und „}“ befinden, werden vor dem Anzeigen ausgewertet.

Die Methode „set_modal“ (Zeile 25) deaktiviert das Hauptfenster so lange, bis der Dialog geschlossen wird.



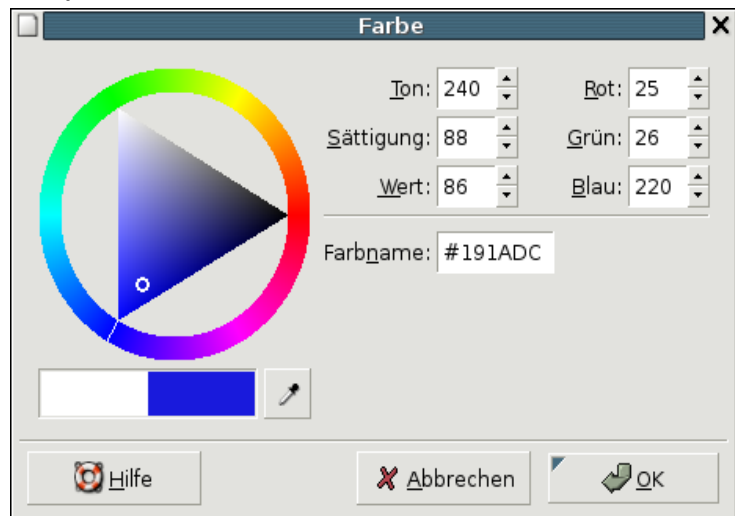
ColorSelectionDialog

ColorSelectionDialog ermöglicht das präzise Definieren von Farben:

```

color_selection.rb
(1) #!/usr/bin/env ruby
(2)
(3) require 'gtk2'
(4) include Gtk
(5)
(6) class Fenster < Window
(7)
(8)   #Membervariablen
(9)   @farbe
(10)  @button
(11)
(12)  def initialize
(13)    super
(14)
(15)    self.set_title( "" )
(16)    self.set_border_width( 20 )
(17)    self.set_window_position( Position::MOUSE )
(18)    self.signal_connect( "destroy" ) { main_quit }
(19)
(20)
(21)    @button = Button.new( "Farbe wählen" )
(22)    @button.signal_connect( "clicked" ) { on_button_clicked }
(23)
(24)    self.add( @button )
(25)    self.show_all
(26)  end
(27)
(28)  def on_button_clicked

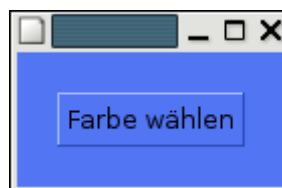
```




```

(29)     cs = Gtk::ColorSelectionDialog.new("Farbe")
(30)     cs.set_modal( self )
(31)     cs.ok_button.signal_connect( "clicked" ) do
(32)         @farbe = cs.colorsels.current_color
(33)         cs.destroy
(34)         #gesamtes Fenster einfaerben
(35)         @button.modify_bg( STATE_NORMAL, @farbe )
(36)         @button.modify_bg( STATE_ACTIVE, @farbe )
(37)         @button.modify_bg( STATE_PRELIGHT, @farbe )
(38)         self.modify_bg( STATE_NORMAL, @farbe )
(39)     end
(40)     cs.cancel_button.signal_connect( "clicked" ) do
(41)         cs.destroy
(42)     end
(43)     cs.help_button.signal_connect( "clicked" ) do
(44)         puts "help"
(45)     end
(46)     cs.show_all
(47) end
(48)
(49)end

```



FontSelectionDialog

Als letzten Dialog dieser Art siehst du hier jenen zur Auswahl von Schriftart, -grad und -größe:

```

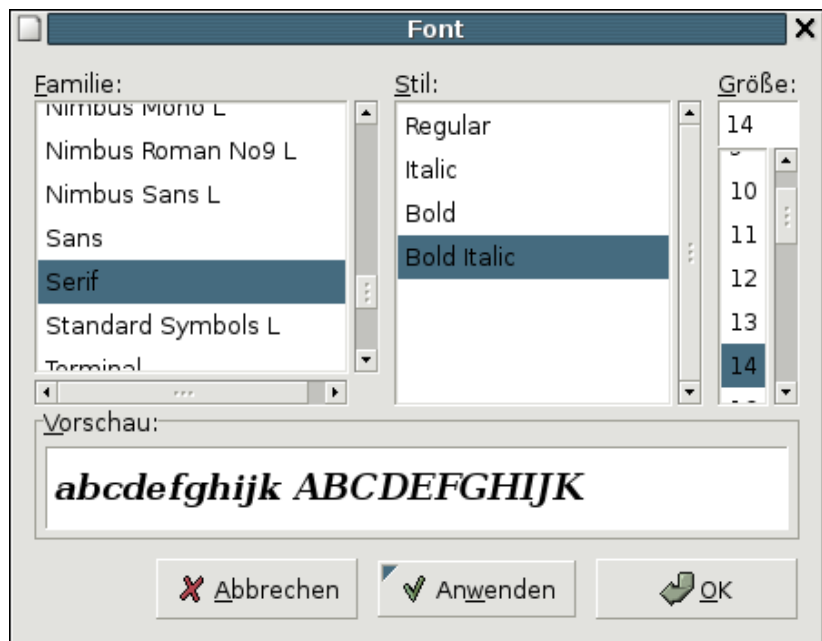
font_selection.rb
(1) #!/usr/bin/env ruby
(2)
(3) require 'gtk2'
(4) include Gtk
(5)
(6) class Fenster < Window
(7)
(8)     #Membervariablen
(9)     @btn_start
(10)    @entry
(11)    @font
(12)
(13)    def initialize
(14)        super
(15)
(16)        self.set_title( "Schriften" )
(17)        self.set_border_width( 20 )
(18)        self.set_window_position( Position::MOUSE )
(19)        self.signal_connect( "destroy" ) { main_quit }
(20)
(21)        vbox = VBox.new( true, 10 )
(22)
(23)        @btn_start = Button.new( "Font" )
(24)        @btn_start.set_size_request( 200, 30 )
(25)        @btn_start.signal_connect( "clicked" ) { on_btn_start_clicked }
(26)        vbox.pack_start( @btn_start )

```

```

(27)
(28)   @entry = Entry.new
(29)   vbox.pack_start( @entry )
(30)
(31)   self.add( vbox )
(32)   self.show_all
(33) end
(34)
(35) def on_btn_start_clicked
(36)   fs = FontSelectionDialog.new( "Font" )
(37)   fs.ok_button.signal_connect( "clicked" ) do
(38)     font_string = fs.font_name
(39)     @font = Pango::FontDescription.new( font_string )
(40)     @entry.modify_font( @font )
(41)     fs.destroy
(42)   end
(43)   fs.apply_button.signal_connect( "clicked" ) do
(44)     font_string = fs.font_name
(45)     @font = Pango::FontDescription.new( font_string )
(46)     @entry.modify_font( @font )
(47)   end
(48)   fs.cancel_button.signal_connect( "clicked" ) do
(49)     fs.destroy
(50)   end
(51)   fs.signal_connect( "destroy" ) do
(52)     fs.destroy
(53)   end
(54)   fs.show_all
(55) end
(56)
(57)end

```



Schrift direkt ändern

Du kannst Schriften auch ohne den Umweg über einen FontSelectionDialog definieren:

```

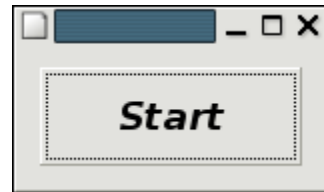
font.rb
(1) #!/usr/bin/env ruby
(2)
(3) require 'gtk2'

```

```

(4) include Gtk
(5)
(6) class Fenster < Window
(7)
(8)     #Membervariablen
(9)     @btn_start
(10)    @font
(11)
(12)    def initialize
(13)        super
(14)
(15)        self.set_title( "" )
(16)        self.set_border_width( 10 )
(17)        self.set_window_position( Position::MOUSE )
(18)        self.signal_connect( "destroy" ) { main_quit }
(19)
(20)        lbl_aufschrift = Label.new( "Start" )
(21)        @font = Pango::FontDescription.new( "Sans Bold Italic 14" )
(22)        lbl_aufschrift.modify_font( @font )
(23)
(24)        @btn_start = Button.new()
(25)        @btn_start.add( lbl_aufschrift )
(26)        @btn_start.signal_connect( "clicked" ) { on_btn_start_clicked }
(27)
(28)        self.add( @btn_start )
(29)        self.show_all
(30)    end
(31)
(32)    def on_btn_start_clicked
(33)        puts @font
(34)    end
(35)
(36)end

```

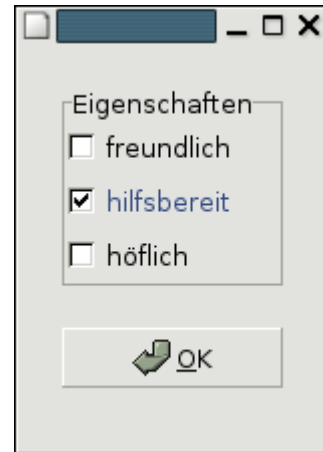


In Zeile 21 wird die Schrift festgelegt. Da sich die Schriftart der Aufschrift des Buttons selbst nicht ändern lässt, wird im Beispiel ein Label auf den Button gelegt (Zeile 25).

CheckBox und Frame

check_button.rb

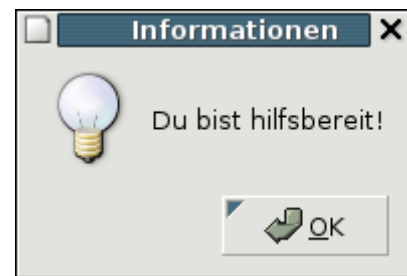
```
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)     #Membervariablen
(7)     @chk_freundlich
(8)     @chk_hilfsbereit
(9)     @chk_hoeflich
(10)
(11)     def initialize
(12)         super
(13)
(14)         self.set_title( "" )
(15)         self.set_border_width( 20 )
(16)         self.set_size_request( 150, 200 )
(17)         self.set_window_position( Position::MOUSE )
(18)         self.signal_connect( "destroy" ) { main_quit }
(19)
(20)         vbox1 = VBox.new( false, 10 )
(21)
(22)         #Frame
(23)         frame = Frame.new
(24)         frame.set_label( "Eigenschaften" )
(25)
(26)         vbox2 = VBox.new( true, 0 )
(27)
(28)         #CheckButtons
(29)         @chk_freundlich = CheckButton.new( "freundlich" )
(30)         vbox2.pack_start( @chk_freundlich )
(31)
(32)         @chk_hilfsbereit = CheckButton.new( "hilfsbereit" )
(33)         vbox2.pack_start( @chk_hilfsbereit )
(34)
(35)         @chk_hoeflich = CheckButton.new( "höflich" )
(36)         vbox2.pack_start( @chk_hoeflich )
(37)
(38)         frame.add( vbox2 )
(39)         vbox1.pack_start( frame )
(40)
(41)         btn = Button.new( Stock::OK )
(42)         btn.signal_connect( "clicked" ) { on_btn_clicked }
(43)         vbox1.pack_start( btn, false, false, 10 )
(44)
(45)         self.add( vbox1 )
(46)         self.show_all
(47)     end
(48)
(49)     def on_btn_clicked
(50)         eigenschaften = 0
(51)         msg = ""
(52)         if @chk_freundlich.active?
(53)             msg = msg + "Du bist freundlich!"
(54)             eigenschaften += 1
(55)         end
```



```

(56)     if @chk_hilfsbereit.active?
(57)         msg = msg + "\nDu bist hilfsbereit!"
(58)         eigenschaften += 1
(59)     end
(60)     if @chk_hoeflich.active?
(61)         msg = msg + "\nDu bist höflich!"
(62)         eigenschaften += 1
(63)     end
(64)
(65)     if eigenschaften == 0
(66)         msg = "Du solltest dich ein wenig anstrengen!"
(67)     end
(68)
(69)     dialog = MessageDialog.new( self, 1, 0, 1, msg )
(70)     dialog.run
(71)     dialog.destroy
(72) end
(73)end

```



Der Container „Frame“ dient der optischen Gruppierung von Elementen. In ein Frame wird die zu umrahmende Komponente eingefügt, im Beispiel ist dies „vbox2“.

DrawingArea, Thread

Willst du ein Programm schreiben, bei dem der Eindruck einer Animation besteht, kannst du dies in Ruby/GTK mit Hilfe eines Threads realisieren:

```

animation.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Animation < Window
(5)
(6)     #Membervariablen
(7)     @da
(8)     @rot; @grau #Farben
(9)     @vbox
(10)
(11)     def initialize
(12)         super
(13)
(14)         self.set_title( "" )
(15)         self.set_size_request( 80, 300 )
(16)         self.signal_connect( "destroy" ) { main_quit }
(17)
(18)         #Farben definieren
(19)         @rot = Gdk::Color.new( 65535, 0, 0 )
(20)         @grau = Gdk::Color.new( 30000, 30000, 30000 )
(21)
(22)         @da = []

```

```

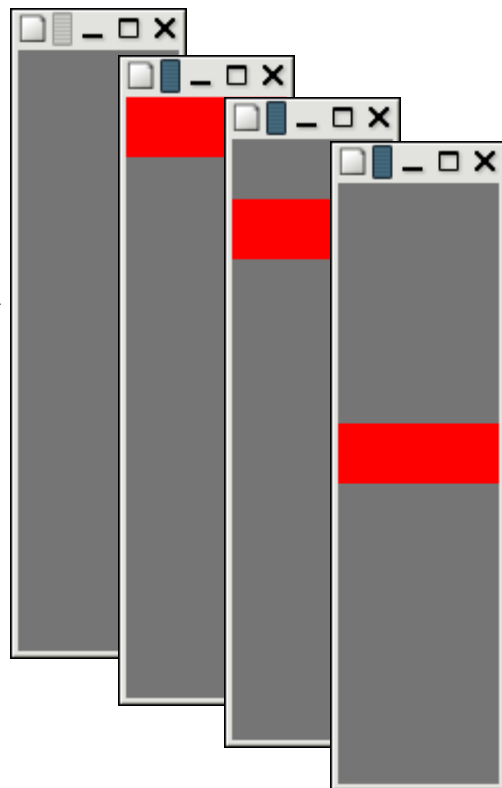
(23)     @vbox = VBox.new( true, 0 )
(24)     for i in 0..9
(25)         @da[ i ] = DrawingArea.new
(26)         @da[ i ].show
(27)         @vbox.pack_start( @da[ i ] )
(28)     end
(29)
(30)     self.add( @vbox )
(31)     self.show_all
(32)
(33)     starte_animation
(34) end
(35)
(36) #Animation
(37) def starte_animation
(38)     k = 0
(39)     Thread.start do
(40)         loop do #Endlosschleife
(41)             puts "sadf"
(42)             for i in 0..9
(43)                 if i == k
(44)                     @da[ k ].modify_bg( STATE_NORMAL, @rot )
(45)                 else
(46)                     @da[ i ].modify_bg( STATE_NORMAL, @grau )
(47)                 end
(48)             end
(49)             #k muss sich immer zwischen 0 und 10 bewegen
(50)             k = ( k + 1 ) % 11
(51)             self.queue_draw
(52)             sleep( 0.3 )
(53)         end
(54)     end
(55) end
(56) end

```

DrawingAreas sind im Wesentlichen „leere Widgets“, sie können somit gut angepasst werden. Wir haben DrawingAreas bereits im Programm grundfarben.rb kennen gelernt.

Ohne den Thread würde der Inhalt des Fensters nicht aktualisiert werden. Der Thread sorgt dafür, dass die System-Ressourcen ständig neu verteilt werden.

„self.queue_draw“ (Zeile 51) veranlasst, dass das Fenster neu gezeichnet wird.



Combo, Programme direkt aufrufen

Ein Combo stellt eine Pop-Down-Liste zur Verfügung. Ein Combo ist sehr nützlich, wenn aus einer vorgegebenen Liste von Möglichkeiten eine ausgewählt werden soll.

geburtstag.rb

```
(1) #!/usr/bin/env ruby
(2)
(3) require 'gtk2'
(4) include Gtk
(5)
(6) class Fenster < Window
(7)
(8)     #Membervariablen
(9)     @combo_tage
(10)    @combo_monate
(11)    @combo_jahre
(12)
(13)    def initialize
(14)        super
(15)
(16)        self.set_title( "" )
(17)        self.set_border_width( 20 )
(18)        self.signal_connect( "destroy" ) { main_quit }
(19)
(20)        hbox = HBox.new( false, 10 )
(21)
(22)        #Tage
(23)        tage = [] #Array
(24)        for i in 0..30
(25)            tage[ i ] = ( i + 1 ).to_s
(26)        end
(27)        @combo_tage = Combo.new
(28)        @combo_tage.set_size_request( 60, 30 )
(29)        @combo_tage.set_popdown_strings( tage )
(30)        hbox.pack_start( @combo_tage )
(31)
(32)        #Monate
(33)        monate = %w( 1 2 3 4 5 6 7 8 9 10 11 12 ) #Array
(34)        @combo_monate = Combo.new
(35)        @combo_monate.set_size_request( 60, 30 )
(36)        @combo_monate.set_popdown_strings( monate )
(37)        hbox.pack_start( @combo_monate )
(38)
(39)        #Jahre
(40)        jahre = [] #Array
(41)        for i in 0..100
(42)            jahre[ i ] = ( i + 1905 ).to_s
(43)        end
(44)        @combo_jahre = Combo.new
(45)        @combo_jahre.set_size_request( 100, 30 )
(46)        @combo_jahre.set_popdown_strings( jahre )
(47)        hbox.pack_start( @combo_jahre )
(48)
(49)        #VBox
(50)        vbox = VBox.new( false, 10 )
```

```

(51) vbox.pack_start( Label.new( "Geburtstag" ) )
(52) vbox.pack_start( hbox, false, false, 10 )
(53)
(54) #Button
(55) btn_start = Button.new( Stock::OK )
(56) btn_start.signal_connect( "clicked" ) { on_btn_start_clicked }
(57) vbox.pack_start( btn_start, false, false, 10 )
(58)
(59) self.add( vbox )
(60) self.show_all
(61) end
(62)
(63) def on_btn_start_clicked
(64)   tag = @combo_tage.entry.text
(65)   monat = @combo_monate.entry.text
(66)   jahr = @combo_jahre.entry.text
(67)   puts "Du bist am " + tag + "." + monat + "." + jahr + " geboren."
(68) end
(69)
(70)end

```

In den Zeilen 64 bis 66 werden die Werte der Combos ausgelesen.

Zeile 1 ist völlig neu: zum einen wird die Zeile durch ein Kommentarzeichen eingeleitet, zum anderen scheint es aber kein sinnvoller Kommentar zu sein.

In Wirklichkeit ist es auch kein Kommentar, die Zeile dient vielmehr dazu, das Programm ohne Umwege zu starten.



Mache das Programm unter Linux ausführbar:

```
chmod +x geburtstag.rb
```

Jetzt kannst du es wie jedes andere ausführbare Programm aufrufen:

```
./geburtstag.rb
```

Unter Windows wird Zeile 1 ignoriert, das Skript kann dort ohnehin direkt gestartet werden.

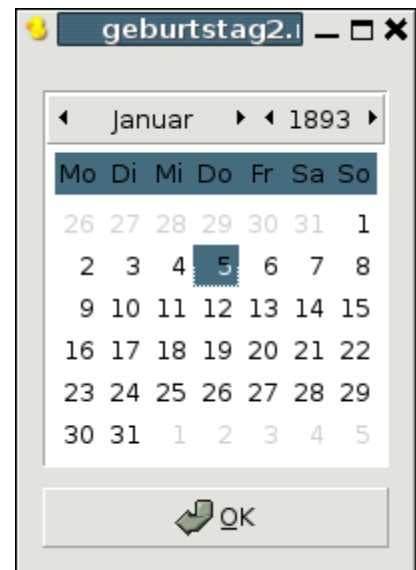
Wenn du verhindern möchtest, dass beim Programmstart die Konsole aufgerufen wird, kannst du die Endung von „rb“ in „rbw“ umbenennen. Dies gilt nicht für Linux!

Calendar, Programmfenster ein Icon zuweisen

Wenn du dich wie im vorigen Beispiel mit Tagen, Monaten und Jahren spielen möchtest, könnte die Klasse Calendar etwas für dich sein. Das Widget präsentiert einen Kalender, in dem du ein Datum wählen kannst:

geburtstag2.rb

```
(1) #!/usr/bin/env ruby
(2)
(3) require 'gtk2'
(4) include Gtk
(5)
(6) class Fenster < Window
(7)
(8)   #Membervariablen
(9)   @kalender
(10)
(11)   def initialize
(12)     super
(13)
(14)     self.set_border_width( 10 )
(15)     self.signal_connect( "destroy" ) { main_quit }
(16)
(17)     #Icon fuer das Fenster definieren
(18)     icon = Gdk::Pixbuf.new( "duck.png" )
(19)     self.set_icon( icon )
(20)
(21)     vbox = VBox.new( false, 0 )
(22)
(23)     @kalender = Calendar.new
(24)     @kalender.set_day( 1 )
(25)     @kalender.set_month( 0 )
(26)     @kalender.set_year( 1905 )
(27)     vbox.pack_start( @kalender, false, false, 10 )
(28)
(29)     button = Button.new( Stock::OK )
(30)     button.signal_connect( "clicked" ) { on_button_clicked }
(31)     vbox.pack_start( button, false, false, 0 )
(32)
(33)     self.add( vbox )
(34)     self.show_all
(35)   end
(36)
(37)   def on_button_clicked
(38)     tag = @kalender.day.to_s
(39)     monat = @kalender.month.to_s
(40)     jahr = @kalender.year.to_s
(41)     puts "Du bist am " + tag + "." + monat + "." + jahr + " geboren."
(42)   end
(43)end
```



Eine weitere Neuigkeit ist hier eingebaut: es gibt die Möglichkeit, dem Fenster ein Icon zuzuweisen (Zeilen 18 und 19). Dieses Icon erscheint im Fenster, im Taskmanager und in der Programmliste.

ScrolledWindow und TextView

Ein ScrolledWindow ist ein Container, welcher automatisch mit der Funktionalität zum Scrollen ausgestattet ist.

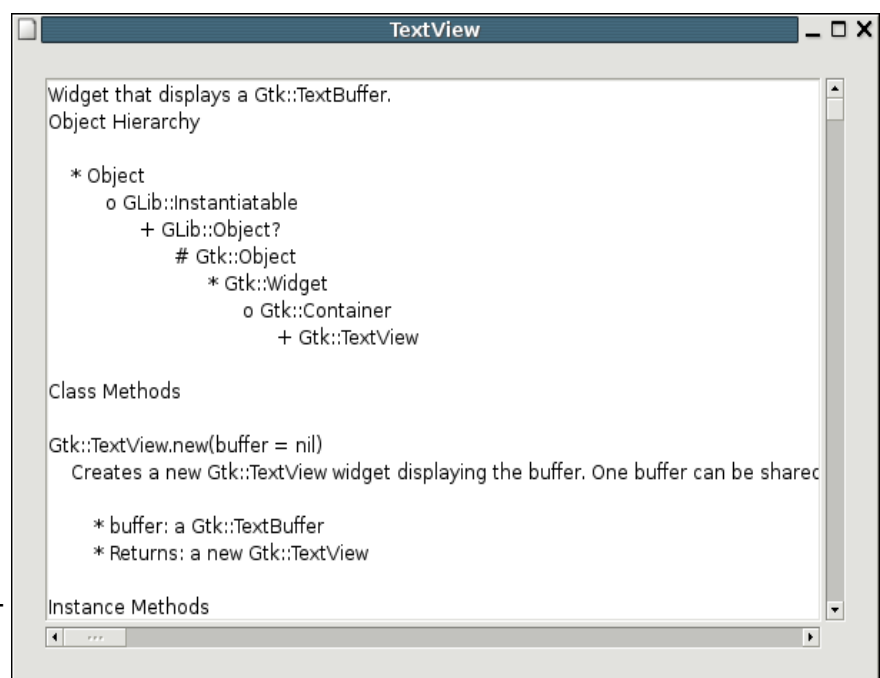
Das im folgenden Beispiel im ScrolledWindow enthaltene TextView ist ähnlich einem Entry, allerdings mit dem wichtigen Unterschied, dass es beliebig viele Zeilen enthalten kann.

editor.rb

```
(1) #!/usr/bin/env ruby
(2)
(3) require 'gtk2'
(4) include Gtk
(5)
(6) class Fenster < Window
(7)
(8)   def initialize
(9)     super
(10)
(11)     self.set_title( "TextView" )
(12)     self.set_border_width( 20 )
(13)     self.set_size_request( 600, 400 )
(14)     self.maximize
(15)     self.set_window_position( Position::MOUSE )
(16)     self.signal_connect( "destroy" ) { main_quit }
(17)
(18)     scw = ScrolledWindow.new
(19)     textview = TextView.new
(20)     scw.add_with_viewport( textview )
(21)
(22)     self.add( scw )
(23)     self.show_all
(24)   end
(25)
(26)end
```

Die Methode „maximize“ (Zeile 14) lässt das Fenster im Fullscreen-Modus starten.

Hinweis: Es gibt die von TextView abgeleitete Klasse SourceView, welche bereits mit vielen Fähigkeiten wie Syntax-Highlighting für beispielsweise auch Ruby-Code ausgestattet ist. In das Textview des



vorherigen Beispiels kann über die Tastatur beliebiger Text eingegeben werden. Es ist jedoch nicht möglich, Text aus dem Programm heraus einzugeben, hierzu ist der Umweg über einen TextBuffer nötig:

editor2.rb

```
(1) #!/usr/bin/env ruby
(2)
(3) require 'gtk2'
(4) include Gtk
(5)
(6) class Fenster < Window
(7)
(8)   def initialize
(9)     super
(10)
(11)     self.set_title( "TextView" )
(12)     self.set_border_width( 20 )
(13)     self.set_size_request( 600, 400 )
(14)     self.maximize
(15)     self.set_window_position( Position::MOUSE )
(16)     self.signal_connect( "destroy" ) { main_quit }
(17)
(18)     textbuffer = TextBuffer.new
(19)     textbuffer.set_text( "beliebiger Text..." )
(20)
(21)     textview = TextView.new
(22)     textview.set_buffer( textbuffer )
(23)     textview.set_editable( false )
(24)     textview.set_cursor_visible( false )
(25)
(26)     scw = ScrolledWindow.new
(27)     scw.add_with_viewport( textview )
(28)
(29)     self.add( scw )
(30)     self.show_all
(31)   end
(32)
(33)end
```

Jetzt fehlt unserem kleinen Editor noch ein Menü.

Menu

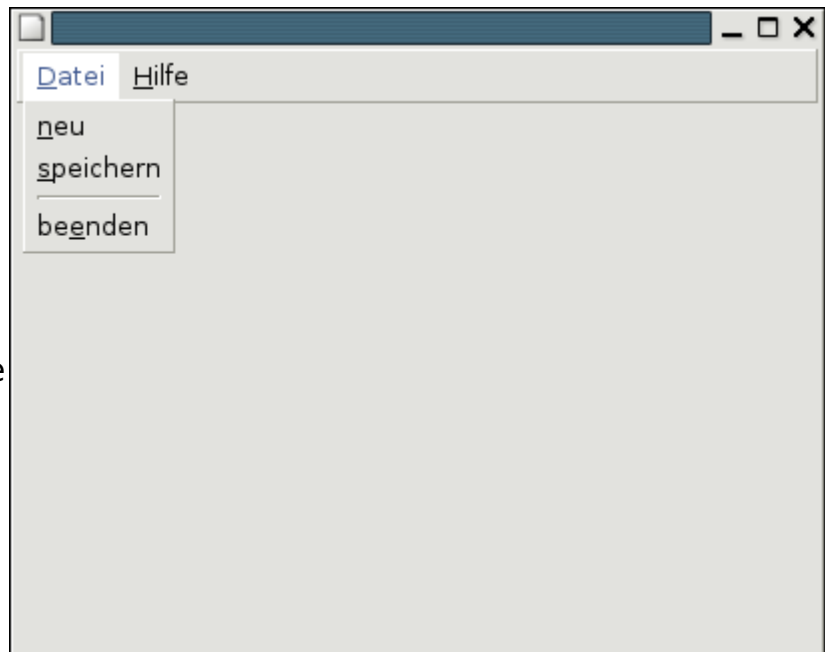
So genannte Pop-Down-Menüs sind in vielen Programmen zu finden.

```
menu.rb
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)   def initialize
(7)     super
(8)
(9)     self.set_size_request( 400, 300 )
(10)    self.set_title( "" )
(11)    self.signal_connect( "destroy" ) { main_quit }
(12)
(13)    #VBox
(14)    vbox = VBox.new( false, 0 )
(15)
(16)    menubar = MenuBar.new
(17)    vbox.pack_start( menubar, false, true, 0 )
(18)
(19)    #Datei-Menue
(20)    dateimenu_leiste = MenuItem.new( "_Datei", true )
(21)    #Menuepunkt zur Menueleiste hinzufuegen
(22)    menubar.append( dateimenu_leiste )
(23)
(24)    dateimenu = Menu.new
(25)
(26)    dateimenu_neu = MenuItem.new( "_neu", true )
(27)    dateimenu_neu.set_name( "dateimenu_neu" )
(28)    dateimenu_neu.signal_connect( "activate" ) do
(29)      | widget |
(30)      on_menu_activate( widget )
(31)    end
(32)    dateimenu.append( dateimenu_neu )
(33)
(34)    dateimenu_speichern = MenuItem.new( "_speichern", true )
(35)    dateimenu_speichern.set_name( "dateimenu_speichern" )
(36)    dateimenu_speichern.signal_connect( "activate" ) do
(37)      | widget |
(38)      on_menu_activate( widget )
(39)    end
(40)    dateimenu.append( dateimenu_speichern )
(41)
(42)    #Separator
(43)    dateimenu.append( SeparatorMenuItem.new )
(44)
(45)    dateimenu_ende = MenuItem.new( "be_enden", true )
(46)    dateimenu_ende.set_name( "dateimenu_ende" )
(47)    dateimenu_ende.signal_connect( "activate" ) do
(48)      | widget |
(49)      on_menu_activate( widget )
(50)    end
(51)    dateimenu.append( dateimenu_ende )
(52)    dateimenu_leiste.set_submenu( dateimenu )
(53)
```

```

(54)      #Hilfe-Menu
(55)      hilfemenu_leiste = MenuItem.new( "_Hilfe", true )
(56)      #Menuepunkt zur Menueleiste hinzufuegen
(57)      menubar.append( hilfemenu_leiste )
(58)
(59)      hilfemenu = Menu.new
(60)
(61)      hilfemenu_info = MenuItem.new( "_info", true )
(62)      hilfemenu_info.set_name( "hilfemenu_info" )
(63)      hilfemenu_info.signal_connect( "activate" ) do
(64)          | widget |
(65)              on_menu_activate( widget )
(66)      end
(67)      hilfemenu.append( hilfemenu_info )
(68)      hilfemenu_leiste.set_submenu( hilfemenu )
(69)
(70)      self.add( vbox )
(71)      self.show_all
(72)  end
(73)
(74)  def on_menu_activate( w )
(75)      puts w.name
(76)      if w.name == "dateimenu_ende"
(77)          main_quit
(78)      end
(79)  end
(80)
(81)end

```



Alle Einträge im gesamten Menü sind MenuItems, einige gehören zum MenuBar (menubar), andere zu einem Menu (dateimenu, hilfemenu).

Die MenuItems des Hauptmenüs (dateimenu_leiste, hilfemenu_leiste) werden mit dem MenuBar verbunden:

```

menubar.append( dateimenu_leiste )
menubar.append( hilfemenu_leiste )

```

Die eigentlichen Menus (dateimenu, hilfemenu) beinhalten weitere MenuItems (neu, speichern, beenden, info), welche direkt zugeordnet werden:

```

dateimenu.append( dateimenu_neu )

```

Diese Menus werden schließlich mit den MenuItems des MenuBar verbunden:

```

dateimenu_leiste.set_submenu( dateimenu )

```

Der boolsche Wert „true“ im Konstruktor der MenuItem's erlaubt das Verwenden einer Tastenkombination zur Navigation durch das Menü. Die Tastenkombination setzt sich aus „Alt“ und jenem Buchstaben zusammen, dem im Bezeichner ein Unterstrich „_“ vorausgeht.

Erscheint dir das Erstellen von Menüs nach dem ersten Durchlesen als etwas undurchsichtig, programmiere am besten selbst eine einfache Menüstruktur - dies hilft dir mit Sicherheit weiter, deshalb gleich ein kleiner Arbeitsauftrag.

Sehr viel flexibler (mit einer allerdings etwas steilen Lernkurve verbunden) geht das Gestalten von Menüs mit Hilfe der Klasse Gtk::UIManager vonstatten, dort kann dem Menü eine Beschreibung im XML-Format zugrunde gelegt werden.

Arbeitsauftrag: ergänze das Programm editor.rb um ein Menü mit den Funktionen zum Laden und Speichern von Dateien. Weiters könntest du ins Menü die Punkte und damit verbunden die entsprechenden Dialogs einbauen, um Schriftart, -grad und -größe zu manipulieren und Hintergrundfarbe des Editorfensters sowie die Schriftfarbe zu ändern.

Kontextmenü, Reagieren auf das Drücken verschiedener Maustasten

In vielen Anwendungen erscheint beim Drücken der rechten Maustaste ein so genanntes Kontext- oder Popup-Menü.

Damit dieses nur beim Drücken der rechten und nicht der mittleren oder linken Maustaste aufgerufen wird, gilt es, zwischen diesen Ereignissen zu unterscheiden:

mausklick.rb

```
(1) #!/usr/bin/env ruby
(2)
(3) require 'gtk2'
(4) include Gtk
(5)
(6) class Fenster < Window
(7)
(8)   #Membervariablen
(9)   @btn
(10)
(11)   def initialize
(12)     super
(13)
(14)     self.set_size_request( 100, 40 )
(15)     self.signal_connect( "destroy" ) { main_quit }
(16)
```



```

(17)     @btn = Button.new( "Start" )
(18)     @btn.signal_connect( "button_press_event" ) do
(19)         | widget, event |
(20)         on_btn_button_press_event( event )
(21)     end
(22)
(23)     self.add( @btn )
(24)     self.show_all
(25) end
(26)
(27) def on_btn_button_press_event( event )
(28)     if event.button == 1
(29)         puts "linke Maustaste"
(30)     elsif event.button == 2
(31)         puts "mittlere Maustaste"
(32)     elsif event.button == 3
(33)         puts "rechte Maustaste"
(34)     end
(35) end
(36)
(37)end

```

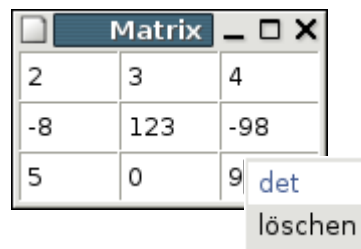
Im folgenden Programm kann mit Hilfe eines Popup-Menüs die Determinante einer Matrix berechnet werden:

kontextmenu.rb

```

(1) #!/usr/bin/env ruby
(2)
(3) require 'gtk2'
(4) include Gtk
(5)
(6) class Fenster < Window
(7)
(8)     #Membervariablen
(9)     @f #Feld
(10)    @tab
(11)
(12)    def initialize
(13)        super
(14)
(15)        self.set_title( "Matrix" )
(16)        self.signal_connect( "destroy" ) { main_quit }
(17)
(18)        @tab = Table.new( 3, 3, true )
(19)        @f = []
(20)        k = 0
(21)        for i in 0...3
(22)            for j in 0...3
(23)                @f[ k ] = Entry.new
(24)                @f[ k ].set_size_request( 50, 25 )
(25)                @f[ k ].signal_connect( "button_press_event" ) do
(26)                    | widget, event |
(27)                    on_feld_button_press_event( event )
(28)                end
(29)                @tab.attach( @f[ k ], j, j + 1, i, i + 1 )
(30)                k += 1

```



```

(31)         end
(32)     end
(33)
(34)     self.add( @tab )
(35)     self.show_all
(36) end
(37)
(38) def on_feld_button_press_event( event )
(39)     if event.button == 3 #wenn rechte Maustaste gedrückt wurde
(40)         menu = Menu.new
(41)
(42)         menu_det = MenuItem.new( "det" )
(43)         menu_det.set_name( "det" )
(44)         menu_det.signal_connect( "activate" ) do
(45)             | widget |
(46)             on_menu_activate( widget )
(47)         end
(48)         menu.append( menu_det )
(49)
(50)         menu_del = MenuItem.new( "löschen" )
(51)         menu_del.set_name( "del" )
(52)         menu_del.signal_connect( "activate" ) do
(53)             | widget |
(54)             on_menu_activate( widget )
(55)         end
(56)         menu.append( menu_del )
(57)
(58)         menu.show_all
(59)         menu.popup( nil, nil, event.button, event.time )
(60)     end
(61) end
(62)
(63) def on_menu_activate( w )
(64)     if w.name == "det"
(65)         det
(66)     elsif w.name == "del"
(67)         del
(68)     end
(69) end
(70)
(71) #Methode zur Berechnung der Determinante
(72) def det
(73)     d = @f[ 0 ].text.to_i * @f[ 4 ].text.to_i * @f[ 8 ].text.to_i + \
(74)         @f[ 1 ].text.to_i * @f[ 5 ].text.to_i * @f[ 6 ].text.to_i + \
(75)         @f[ 2 ].text.to_i * @f[ 3 ].text.to_i * @f[ 7 ].text.to_i - \
(76)         @f[ 6 ].text.to_i * @f[ 4 ].text.to_i * @f[ 2 ].text.to_i - \
(77)         @f[ 7 ].text.to_i * @f[ 5 ].text.to_i * @f[ 0 ].text.to_i - \
(78)         @f[ 8 ].text.to_i * @f[ 3 ].text.to_i * @f[ 1 ].text.to_i
(79)
(80)     msg = "Determinante: " + d.to_s
(81)     dialog = MessageDialog.new( self, 0, 0, 1, msg )
(82)     dialog.run
(83)     dialog.destroy
(84) end
(85)
(86) #Methode löscht Elemente der Matrix und setzt Cursor in erstes Feld
(87) def del
(88)     for i in 0...9

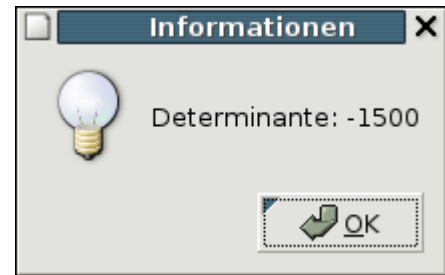
```



```

(89)         @f[ i ].set_text( "" )
(90)     end
(91)     @f[ 0 ].set_focus( true )
(92) end
(93)
(94)end

```



Das Signal „key_press_event“

Du bist bisher bereits auf viele Ereignisse (events) gestoßen: clicked und enter bei Buttons, destroy und delete_event bei der Klasse Window, button_press_event bei EventBoxen, wobei dieses Signal von Widget geerbt wird und damit allen Widgets zur Verfügung steht. Weiter triffst du auf value_changed bei ScrollBars, toggled bei RadioButtons und schließlich activate bei Menüitems.

Hier lernst du das Signal „key_press_event“ kennen, es wird ausgelöst, wenn du eine Taste auf der Tastatur drückst. Du kannst dabei zwischen dem Drücken verschiedener Tasten unterscheiden und unterschiedlich darauf reagieren.

Im Beispiel wird das Fenster beim Drücken der Pfeiltasten in seiner Größe verändert, dazu wird die Eigenschaft „keyval“ des Ereignisses „key_press_event“ ausgewertet:

```

keyval.rb
(1) #!/usr/bin/env ruby
(2)
(3) require 'gtk2'
(4) include Gtk
(5)
(6) class Fenster < Window
(7)
(8)   def initialize
(9)     super
(10)
(11)     self.signal_connect( "destroy" ) { main_quit }
(12)     self.signal_connect( "key_press_event" ) do
(13)       | widget, event |
(14)         on_key_pressed( event )
(15)     end
(16)
(17)     self.show_all
(18)   end
(19)
(20)   def on_key_pressed( e )
(21)     if e.keyval == 65362
(22)       self.resize( self.size[ 0 ], self.size[ 1 ] - 1 )
(23)     elsif e.keyval == 65364
(24)       self.resize( self.size[ 0 ], self.size[ 1 ] + 1 )
(25)     elsif e.keyval == 65361
(26)       self.resize( self.size[ 0 ] - 1, self.size[ 1 ] )

```

```

(27)         elsif e.keyval == 65363
(28)             self.resize( self.size[ 0 ] + 1, self.size[ 1 ] )
(29)         end
(30)     end
(31)
(32)end

```

Die Methode `self.size[0]` gibt die Breite, `self.size[1]` die Höhe des Fensters zurück.

TreeView

Die Komponente `TreeView` eignet sich hervorragend zum Anzeigen von Daten, beispielsweise zum Anzeigen der Ergebnisse einer Datenbankabfrage. Ein `TreeView` erlaubt die Ausgabe in tabellarischer und in verzweigter Form, je nach dem, ob du ein `Gtk::ListStore` oder `Gtk::TreeStore` verwendest.

telefonbuch.rb

```

(1) #!/usr/bin/env ruby
(2) require 'gtk2'
(3) include Gtk
(4)
(5) class Fenster < Window
(6)
(7)     def initialize
(8)         super
(9)
(10)         @list_store = ListStore.new( String, String )
(11)         view = TreeView.new( @list_store )
(12)
(13)         #Spalte 1
(14)         r1 = CellRendererText.new
(15)         col1 = TreeViewColumn.new( "Name", r1 , :text => 0 )
(16)         view.append_column( col1 )
(17)
(18)         #Spalte 2
(19)         r2 = CellRendererText.new
(20)         col2 = TreeViewColumn.new( "Telefon", r2, :text => 1 )
(21)         view.append_column( col2 )
(22)
(23)         #Eintraege ins ListStore (und damit ins TreeView)
(24)         eintrag = []
(25)
(26)         #Zeile 1
(27)         eintrag[ 0 ] = @list_store.append
(28)         eintrag[ 0 ][ 0 ] = "Susi"
(29)         eintrag[ 0 ][ 1 ] = "3401234567"
(30)
(31)         #Zeile 2
(32)         eintrag[ 1 ] = @list_store.append
(33)         eintrag[ 1 ][ 0 ] = "Herbert"
(34)         eintrag[ 1 ][ 1 ] = "3402345678"

```



```

(35)
(36)     self.add( view )
(37)     self.show_all
(38) end
(39)
(40)end

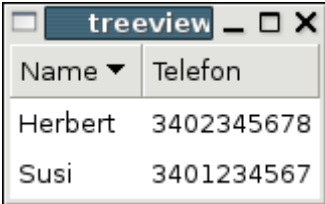
```

In telefonbuch2.rb ist es möglich, Einträge via Drag & Drop umzusortieren sowie durch Klicken auf den Spaltenkopf die Einträge automatisch sortieren zu lassen:

```

(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)     def initialize
(7)         super
(8)
(9)         @list_store = ListStore.new( String, String )
(10)        view = TreeView.new( @list_store )
(11)        view.reorderable = true
(12)
(13)        r1 = CellRendererText.new
(14)        col1 = TreeViewColumn.new( "Name", r1 , :text => 0 )
(15)        col1.sort_column_id = 0
(16)        col1.clickable = true
(17)        col1.resizable = true
(18)        col1.reorderable = true
(19)        view.append_column( col1 )
(20)
(21)        r2 = CellRendererText.new
(22)        col2 = TreeViewColumn.new( "Telefon", r2, :text => 1 )
(23)        col2.sort_column_id = 1
(24)        col2.clickable = true
(25)        col2.resizable = true
(26)        col2.reorderable = true
(27)        view.append_column( col2 )
(28)
(29)        #Eintraege ins TreeView
(30)        eintrag = []
(31)
(32)        eintrag[ 0 ] = @list_store.append
(33)        eintrag[ 0 ][ 0 ] = "Susi"
(34)        eintrag[ 0 ][ 1 ] = "3401234567"
(35)
(36)        eintrag[ 1 ] = @list_store.append
(37)        eintrag[ 1 ][ 0 ] = "Herbert"
(38)        eintrag[ 1 ][ 1 ] = "3402345678"
(39)
(40)        self.add( view )
(41)        self.show_all
(42)    end
(43)
(44)end

```



Name ▼	Telefon
Herbert	3402345678
Susi	3401234567

telefonbuch3.rb zeigt die Verwendung der Klasse `TreeView` inmitten einer Kontaktverwaltung:

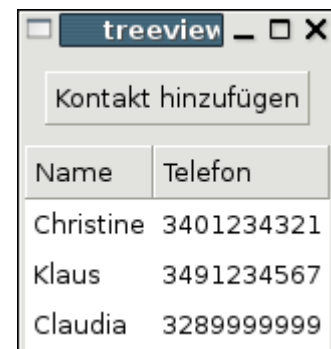
```
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Kontakt
(5)   attr_reader :name, :telefon
(6)
(7)   #Membervariablen
(8)   @name
(9)   @telefon
(10)
(11)   def initialize( name_, telefon_ )
(12)     @name = name_
(13)     @telefon = telefon_
(14)   end
(15) end
(16)
(17) class Fenster < Window
(18)
(19)   #Membervariablen
(20)   @kontakte
(21)   @item
(22)
(23)   def initialize
(24)     super
(25)     self.signal_connect( "destroy" ) { main_quit }
(26)
(27)     @kontakte = []
(28)     @item = []
(29)
(30)     vbox = VBox.new( false, 0 )
(31)
(32)     btn_neu = Button.new( "Kontakt hinzufügen" )
(33)     btn_neu.signal_connect( "clicked" ) { on_btn_neu_clicked }
(34)     btn_neu.set_border_width( 10 )
(35)     vbox.pack_start( btn_neu, false, false )
(36)
(37)     @list_store = ListStore.new( String, String )
(38)     view = TreeView.new( @list_store )
(39)     view.reorderable = true
(40)
(41)     #Spalte 1
(42)     r1 = CellRendererText.new
(43)     col1 = TreeViewColumn.new( "Name", r1, :text => 0 )
(44)     col1.sort_column_id = 0
(45)     col1.clickable = true
(46)     col1.resizable = true
(47)     col1.reorderable = true
(48)     view.append_column( col1 )
(49)
(50)     #Spalte 2
(51)     r2 = CellRendererText.new
(52)     col2 = TreeViewColumn.new( "Telefon", r2, :text => 1 )
(53)     col2.sort_column_id = 1
(54)     col2.clickable = true
(55)     col2.resizable = true
(56)     col2.reorderable = true
```



```

(57)         view.append_column( col2 )
(58)
(59)         vbox.pack_start( view, true, true )
(60)         self.add( vbox )
(61)         self.show_all
(62)     end
(63)
(64)     def on_btn_neu_clicked
(65)         text = "neuen Kontakt eingeben"
(66)         dialog = MessageDialog.new( self, 0, 2, 5, text )
(67)         tab = Table.new( 2, 2, true )
(68)         tab.attach( Label.new( "Name " ), 0, 1, 0, 1 )
(69)         txt_name = Entry.new
(70)         tab.attach( txt_name, 1, 2, 0, 1 )
(71)         tab.attach( Label.new( "Telefon " ), 0, 1, 1, 2 )
(72)         txt_telefon = Entry.new
(73)         tab.attach( txt_telefon, 1, 2, 1, 2 )
(74)         dialog.vbox.pack_start( tab )
(75)         dialog.show_all
(76)         dialog.run do
(77)             | response |
(78)             if response == -5
(79)                 i = @kontakte.length
(80)                 @kontakte << Kontakt.new( txt_name.text, txt_telefon.text )
(81)                 @item[ i ] = @list_store.append
(82)                 @item[ i ][ 0 ] = @kontakte[ i ].name
(83)                 @item[ i ][ 1 ] = @kontakte[ i ].telefon
(84)             end
(85)         end
(86)         dialog.destroy
(87)     end
(88) end
(89)
(90)end

```



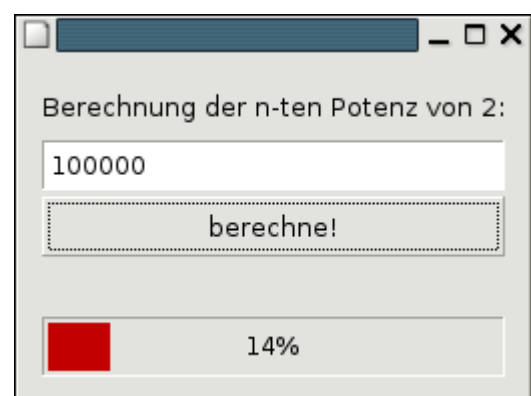
ProgressBar

Dieses Widget stellt einen Fortschrittsbalken dar. Wenn ein Arbeitsvorgang einen längeren Zeitraum beansprucht, ist es sinnvoll, den Benutzer über den Fortschritt zu informieren, vor allem, wenn das Programm selbst keine Rückmeldung liefert.

Es gibt zwei Möglichkeiten, einen ProgressBar anzuzeigen: einen prozentuellen Modus, wo die Restzeit abschätzbar ist und einen reinen Aktivitätsmodus. Im folgenden Beispiel wird die n -te Potenz von 2 berechnet. Da jede der Multiplikationen annähernd gleich lange dauert, bietet sich der prozentuelle Modus an:

zweierpotenz.rb

```
(1) #!/usr/bin/env ruby
(2)
(3) require 'gtk2'
(4) include Gtk
(5)
(6) class Fenster < Window
(7)
(8)   #Membervariablen
(9)   @pb
(10)  @entry
(11)  @label
(12)
(13)  def initialize
(14)    super
(15)
(16)    self.set_title( "" )
(17)    self.set_border_width( 10 )
(18)    self.signal_connect( "destroy" ) { main_quit }
(19)
(20)    vbox = VBox.new( true, 0 )
(21)
(22)    vbox.pack_start( Label.new( "Berechnung der n-ten Potenz von 2:" ) )
(23)
(24)    @entry = Entry.new
(25)    vbox.pack_start( @entry )
(26)
(27)    button = Button.new( "berechne" )
(28)    button.signal_connect( "clicked" ) { on_button_clicked }
(29)    vbox.pack_start( button )
(30)
(31)    @label = Label.new
(32)    vbox.pack_start( @label )
(33)
(34)    #ProgressBar
(35)    @pb = ProgressBar.new
(36)    #Farbe des ProgressBars aendern
(37)    @pb.modify_bg( STATE_SELECTED, Gdk::Color.new( 50000, 0, 0 ) )
(38)    vbox.pack_start( @pb )
(39)
(40)    self.add( vbox )
(41)    self.show_all
(42)  end
(43)
(44)  def on_button_clicked
(45)    @label.set_label( "" )
(46)    hz = @entry.text.to_i
(47)    ergebnis = 1
(48)    fortschritt = 1 / hz.to_f
(49)    Thread.start do
(50)      t1 = Time.now.to_f
(51)      for i in 1..hz
(52)        ergebnis = ergebnis * 2
(53)        @pb.fraction = fortschritt
(54)
(55)        #Fortschritt in Prozent in den ProgressBar einfuegen
(56)        @pb.set_text( ( fortschritt * 100 ).to_i.to_s + "%" )
(57)        fortschritt += 1 / hz.to_f
(58)      end
(59)    end
(60)  end
(61)end
```



```

(59)         t2 = Time.now.to_f
(60)         zeit = ( ( t2 - t1 ) * 100 ).to_i / 100.to_f ).to_s
(61)         @label.set_label( "benötigte Zeit: " + zeit + " sec" )
(62)     end
(63) end
(64)
(65)end

```

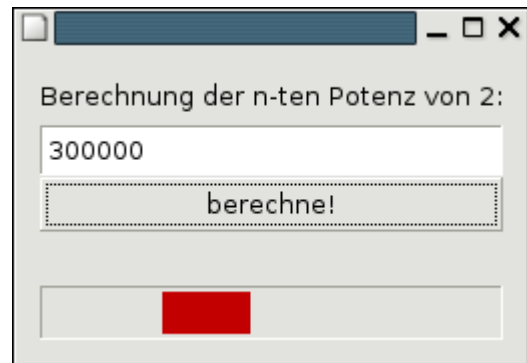
Dasselbe Beispiel noch mal, diesmal ist der ProgressBar im Aktivitätsmodus:

zweierpotenz2.rb

```

(1) #!/usr/bin/env ruby
(2)
(3) require 'gtk2'
(4) include Gtk
(5)
(6) class Fenster < Window
(7)
(8)     #Membervariablen
(9)     @pb
(10)    @entry
(11)    @label
(12)
(13)    def initialize
(14)        super
(15)
(16)        self.set_title( "" )
(17)        self.set_border_width( 10 )
(18)        self.signal_connect( "destroy" ) { main_quit }
(19)
(20)        vbox = VBox.new( true, 0 )
(21)
(22)        vbox.pack_start( Label.new( "Berechnung der n-ten Potenz von 2:" ) )
(23)
(24)        @entry = Entry.new
(25)        vbox.pack_start( @entry )
(26)
(27)        button = Button.new( "berechne!" )
(28)        button.signal_connect( "clicked" ) { on_button_clicked }
(29)        vbox.pack_start( button )
(30)
(31)        @label = Label.new
(32)        vbox.pack_start( @label )
(33)
(34)        #ProgressBar
(35)        @pb = ProgressBar.new
(36)        #Schrittweite festlegen
(37)        @pb.set_pulse_step( 0.01 )
(38)
(39)        #Farbe des ProgressBars aendern
(40)        @pb.modify_bg( STATE_SELECTED, Gdk::Color.new( 50000, 0, 0 ) )
(41)        vbox.pack_start( @pb )
(42)
(43)        self.add( vbox )
(44)        self.show_all
(45)    end
(46)
(47)    def on_button_clicked

```



```

(48)    @label.set_label( "" )
(49)    hz = @entry.text.to_i
(50)    ergebnis = 1
(51)    Thread.start do
(52)        for i in 1..hz
(53)            ergebnis = ergebnis * 2
(54)            #damit der ProgressBar nicht allzu oft
(55)            #seine Position aendert
(56)            if i % 500 == 0
(57)                #ProgressBar aktualisieren
(58)                @pb.pulse
(59)            end
(60)        end
(61)        @label.set_label( "Berechnung abgeschlossen" )
(62)    end
(63) end
(64)
(65)end

```

Signale blockieren und entfernen

Mit der Methode „signal_connect“ fügst du einer Komponente bekanntlich einen Signalhandler hinzu.

Du kannst Signalhandler auch blockieren (signal_handler_block), die Blockade wieder aufheben (signal_handler_unblock) oder Signalhandler entfernen (signal_handler_disconnect).

Du findest diese und weitere Methoden inklusive Beschreibung in der Dokumentation der Klasse GLib::Instantiatable.

Beim Hinzufügen eines Signalhandlers zu einer Komponente wird auch eine eindeutige Handler-ID vergeben. Diese wird beim Blockieren oder Entfernen des Handlers benötigt.

Im Beispiel kann das Signal eines Buttons blockiert und wieder freigegeben werden:

```

signal_blockieren.rb
(1) #!/usr/bin/env ruby
(2)
(3) require 'gtk2'
(4) include Gtk
(5)
(6) class Fenster < Window
(7)
(8)     #Membervariablen
(9)     @btn_start #Button Start
(10)    @btn_start_handler_id #Handler-ID
(11)    @signal_angeschlossen #boolsche Variable
(12)    @btn_signal

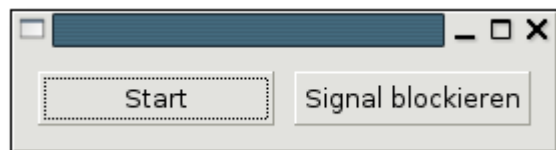
```



```

(13)
(14) def initialize
(15)     super
(16)
(17)     self.set_title( "" )
(18)     self.set_border_width( 10 )
(19)     #self.set_size_request( 200, 50 )
(20)     self.signal_connect( "destroy" ) { main_quit }
(21)
(22)     @signal_angeschlossen = true
(23)
(24)     hbox = HBox.new( true, 10 )
(25)
(26)     @btn_start = Button.new( "Start" )
(27)     @btn_start_handler_id = @btn_start.signal_connect( "clicked" ) do
(28)         on_btn_start_clicked
(29)     end
(30)     hbox.pack_start( @btn_start )
(31)
(32)     @btn_signal = Button.new( "Signal blockieren" )
(33)     @btn_signal.signal_connect( "clicked" ) { on_btn_signal_clicked }
(34)     hbox.pack_start( @btn_signal )
(35)
(36)     self.add( hbox )
(37)     self.show_all
(38) end
(39)
(40) def on_btn_start_clicked
(41)     puts "Start"
(42) end
(43)
(44) def on_btn_signal_clicked
(45)     if @signal_angeschlossen
(46)         @btn_start.signal_handler_block( @btn_start_handler_id )
(47)         @signal_angeschlossen = false
(48)         @btn_signal.set_label( "Signal freigeben" )
(49)     else
(50)         @btn_start.signal_handler_unblock( @btn_start_handler_id )
(51)         @signal_angeschlossen = true
(52)         @btn_signal.set_label( "Signal blockieren" )
(53)     end
(54) end
(55) end
(56)
(57)end

```



Die Instanzvariable „@btn_start_handler_id“ speichert die Handler-ID des Signals „clicked“ beim Hinzufügen (Zeile 27). Bei der Handler-ID handelt es sich schlicht um einen Integer.

Die Methode „signal_handler_block“ (Zeile 46) blockiert den Signalhandler „clicked“ des Buttons „btn_start“. In Zeile 50 wird die Blockade wieder aufgehoben.

Mauszeiger ändern

Du kannst in deinen Anwendungen den Mauszeiger verändern:

```
cursor.rb
(1)require 'gtk2'
(2)include Gtk
(3)
(4)class Fenster < Window
(5)
(6)  def initialize
(7)    super
(8)    self.signal_connect( "destroy" ) { main_quit }
(9)
(10)    cursor = Gdk::Cursor.new( Gdk::Cursor::X_CURSOR )
(11)
(12)    self.realize
(13)    self.window.set_cursor( cursor )
(14)
(15)    self.show_all
(16)  end
(17)
(18)end
```

In Zeile 10 wird die Art des Cursor definiert.

Die Methode „realize“ (Zeile 11) erstellt das zum Gtk::Window gehörende Gdk::Window (self.window, Zeile 12), dessen Cursor in Zeile 12 gesetzt wird.

Es steht eine Vielzahl von unterschiedlichen Mauszeigern zur Auswahl, mehr dazu findest du in der Dokumentation zur Klasse Gdk::Cursor bei deren Konstanten.



Zeichnen

Nun knöpfen wir uns die bereits bekannte DrawingArea unter einem neuen Aspekt nochmals vor: wir werden sie benutzen, um Linien und Figuren zu zeichnen.

Linie

linien.rb

```
(1) #!/usr/bin/env ruby
(2)
(3) require 'gtk2'
(4) include Gtk
(5)
(6) class Fenster < Window
(7)
(8)   #Membervariablen
(9)   @area
(10)  @txt_x1; @txt_y1; @txt_x2; @txt_y2 #Entrys
(11)  @x1; @y1; @x2; @y2 #Punkte
(12)
(13)  def initialize
(14)    super
(15)
(16)    @x1 = 0
(17)    @y1 = 0
(18)    @x2 = 0
(19)    @y2 = 0
(20)
(21)    self.set_title( "" )
(22)    self.maximize
(23)    self.set_border_width( 20 )
(24)    self.signal_connect( "destroy" ) { main_quit }
(25)
(26)    vbox = VBox.new( false, 10 )
(27)
(28)    #Eingabefelder
(29)    @txt_x1 = Entry.new
(30)    @txt_y1 = Entry.new
(31)    @txt_x2 = Entry.new
(32)    @txt_y2 = Entry.new
(33)
(34)    tab = Table.new( 2, 8, false )
(35)    tab.attach( Label.new( "x1" ), 0, 1, 0, 1 )
(36)    tab.attach( @txt_x1, 1, 2, 0, 1 )
(37)
(38)    tab.attach( Label.new( "y1" ), 0, 1, 1, 2 )
(39)    tab.attach( @txt_y1, 1, 2, 1, 2 )
(40)
(41)    tab.attach( Label.new( "x2" ), 2, 3, 0, 1 )
(42)    tab.attach( @txt_x2, 3, 4, 0, 1 )
(43)
(44)    tab.attach( Label.new( "y2" ), 2, 3, 1, 2 )
(45)    tab.attach( @txt_y2, 3, 4, 1, 2 )
(46)    #Eingabefelder Ende
```

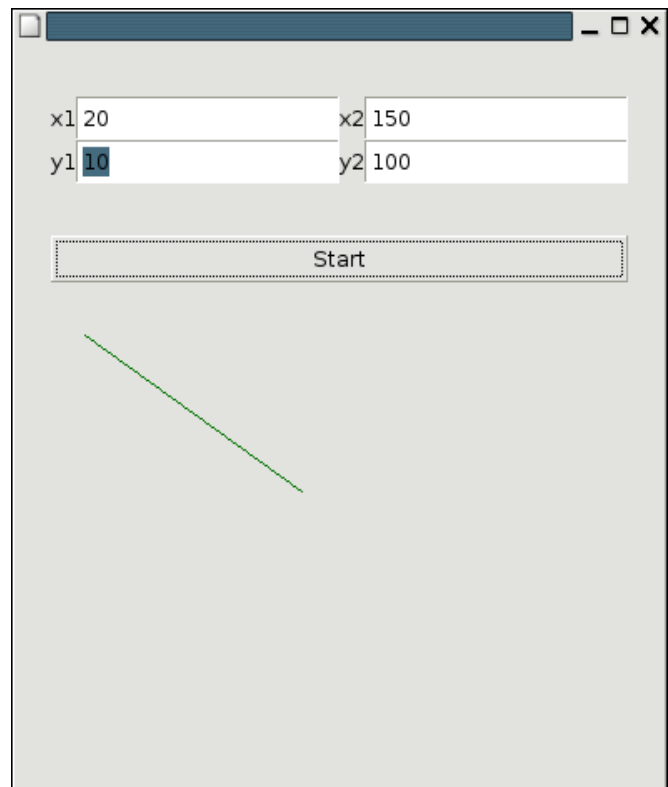
```

(47)
(48)     vbox.pack_start( tab, false, false, 10 )
(49)
(50)     #Button
(51)     button = Button.new( "Start" )
(52)     button.signal_connect( "clicked" ) { on_button_clicked }
(53)     vbox.pack_start( button, false, false, 10 )
(54)
(55)     #DrawingArea
(56)     @area = DrawingArea.new
(57)     @area.set_size_request( 250, 250 )
(58)     @area.modify_fg( STATE_NORMAL, Gdk::Color.new( 0, 30000, 0 ) )
(59)     @area.signal_connect( "expose_event" ) { on_area_expose_event }
(60)     vbox.pack_start( @area )
(61)
(62)     self.add( vbox )
(63)     self.show_all
(64) end
(65)
(66) def on_button_clicked
(67)     #Werte aus den Textfeldern uebernehmen
(68)     @x1 = @txt_x1.text.to_i
(69)     @y1 = @txt_y1.text.to_i
(70)     @x2 = @txt_x2.text.to_i
(71)     @y2 = @txt_y2.text.to_i
(72)     #DrawingArea neu zeichnen
(73)     @area.queue_draw
(74) end
(75)
(76) def on_area_expose_event
(77)     gc = @area.style.fg_gc( @area.state )
(78)     @area.window.draw_line( gc, @x1, @y1, @x2, @y2 )
(79) end
(80)
(81)end

```

Das Ereignis „expose_event“ muss registriert werden (Zeile 59), damit gezeichnet wird.

Wenn auf den Button geklickt wird, werden die vier Werte aus den Entrys als Anfangs- (x1, y1) und Endpunkt (x2, y2) der Linie verwendet.



Ellipse

arc.rb

```
(1) require 'gtk2'
(2) include Gtk
(3)
(4) class Fenster < Window
(5)
(6)   #Membervariablen
(7)   @area
(8)   @entry
(9)   @ende
(10)
(11)   def initialize
(12)     super
(13)
(14)     self.set_title( "" )
(15)     self.set_border_width( 20 )
(16)     self.signal_connect( "destroy" ) { main_quit }
(17)
(18)     @ende = 0
(19)
(20)     vbox = VBox.new( false, 10 )
(21)
(22)     vbox.pack_start( Label.new( "Wert zwischen 0 und 23040 eingeben" ) )
(23)
(24)     @entry = Entry.new
(25)     vbox.pack_start( @entry, false, false, 10 )
(26)
(27)     button = Button.new( "Start" )
(28)     button.signal_connect( "clicked" ) { on_button_clicked }
(29)     vbox.pack_start( button, false, false, 10 )
(30)
(31)     @area = DrawingArea.new
(32)     @area.set_size_request( 250, 250 )
(33)     @area.modify_fg( STATE_NORMAL, Gdk::Color.new( 0, 30000, 0 ) )
(34)     @area.signal_connect( "expose_event" ) { on_area_expose_event }
(35)     vbox.pack_start( @area )
(36)
(37)     self.add( vbox )
(38)     self.show_all
(39)   end
(40)
(41)   def on_button_clicked
(42)     @ende = @entry.text.to_i
(43)     @area.queue_draw
(44)   end
(45)
(46)   def on_area_expose_event
(47)     gc = @area.style.fg_gc( @area.state )
(48)     voll = true
(49)     #Startpunkt der Zeichnung in der DrawingArea
(50)     x = 0
(51)     y = 0
(52)     #Breite und Hoehe der DrawingArea bestimmen
(53)     breite = @area.allocation.width
(54)     hoehe = @area.allocation.height
(55)     start = 0
```

```

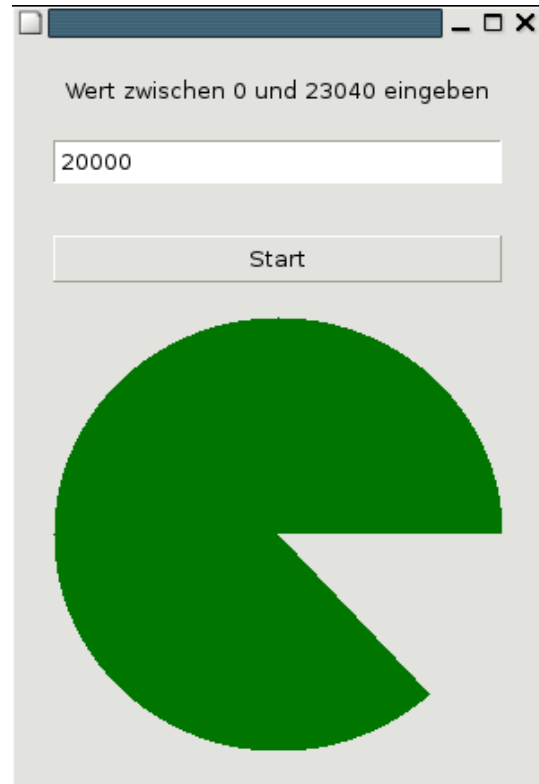
(56)         @area.window.draw_arc( gc, voll, x, y, breite, hoehe, start, @ende )
(57)     end
(58)
(59)end

```

Die Zeichnung beginnt bei 3 Uhr (gegen den Uhrzeigersinn), jeder Grad ist in 64 Abschnitte unterteilt. Um den gesamten Kreis zu zeichnen, muss der Endwert also auf $64 \text{ mal } 360 = 23040$ (oder mehr) gesetzt werden.

Die Methode `DrawingArea#window` (Zeile 56) gibt das `Gdk::Window` zurück, in welches gezeichnet wird.

Weitere Methoden zum Zeichnen von Figuren findest du in der Superklasse von `Gdk::Window`, `Gdk::Drawable`.



Das wars...

Hier endet vorerst unser gemeinsamer Weg. Aber wie immer irgendwann im Leben kommst du auch hier nicht daran vorbei zu zeigen, dass du jetzt auf den eigenen Füßen stehst (natürlich rein programmiertechnisch betrachtet :-)).

Zum Abschluss gebe ich dir noch einige fortgeschrittene Arbeitsaufträge mit auf den Weg:

- Programmiere deine ganz persönliche Variante des Spiels Memory!
- Das Spiel Mastermind ist ebenso eine tolle Herausforderung!
- Versuche dich am Spiel Minesweeper, das Gnome-Spiel Minen vermittelt dir eventuell einen ersten Eindruck, da es ebenfalls auf GTK+ basiert.
- Als Herausforderung schlechthin darf allerdings die Umsetzung des Klassikers Tetris gelten.

Anhang A: Oberflächen-Design mit Glade

Mit Glade (<http://glade.gnome.org/>) steht ein GUI-Builder zum visuellen Erstellen grafischer Oberflächen für GTK+-Anwendungen bereit. Dieses Tool kannst du auch zum Designen von Ruby/GTK-Programmen verwenden.

Unter Windows wird Glade mit dem GTK+-Developer-Paket installiert, das im Zuge der Installation von Ruby/GTK den Weg auf die Platte gefunden hat.

Für Gentoo-Linux benötigst du die beiden Pakete

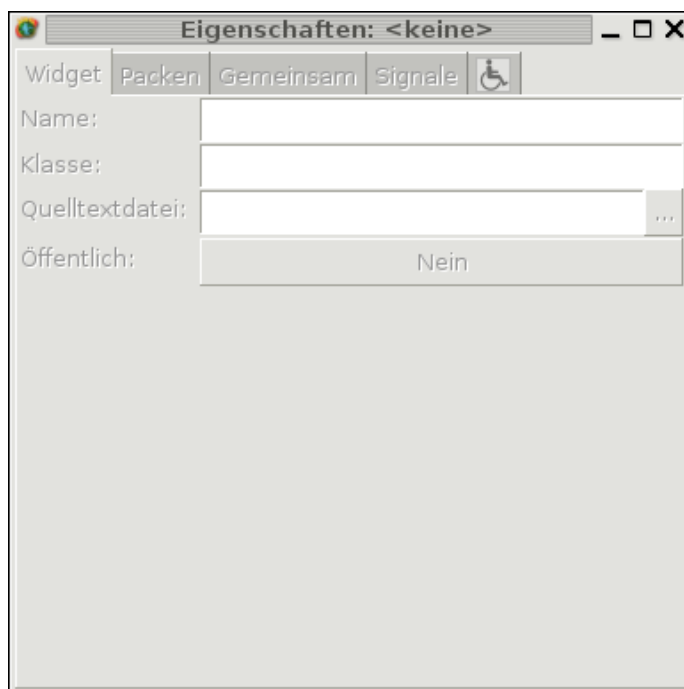
```
dev-util/glade
dev-ruby/ruby-libglade2
```

Ersteres stellt das Tool selbst dar, letzteres sind die Ruby-Bindings, um diese Schnittstelle nutzen zu können.

Ubuntu-Benutzer installieren das Paket „libglade2-ruby“.

Nach dem Start präsentiert Glade drei Fenster:

Hauptfenster

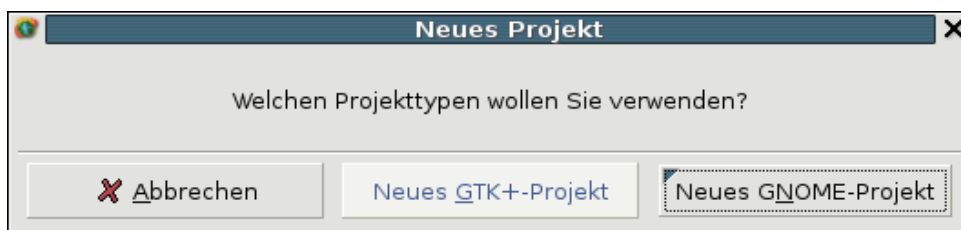


Eigenschaften für die Widgets werden hier eingestellt. Auch Signale können hier registriert werden.

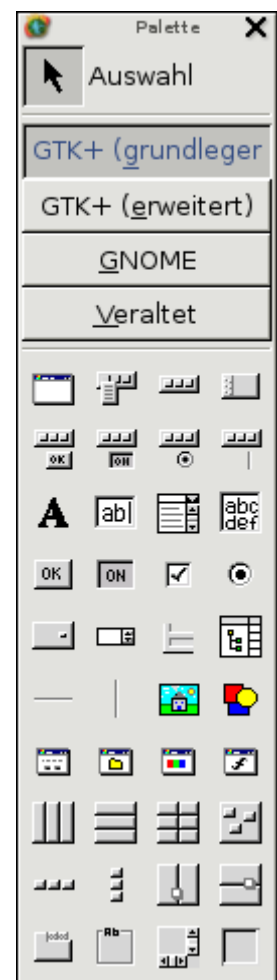
In der „Palette“ kannst du die Widgets auswählen, um sie in dein Fenster einzufügen.

Grafische Oberfläche

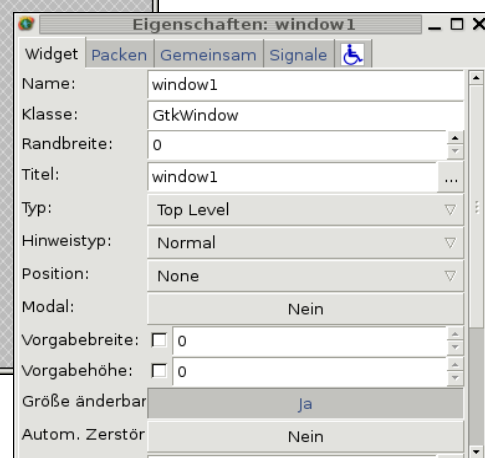
Öffne ein neues Projekt:



Wähle hier das GTK+-Projekt

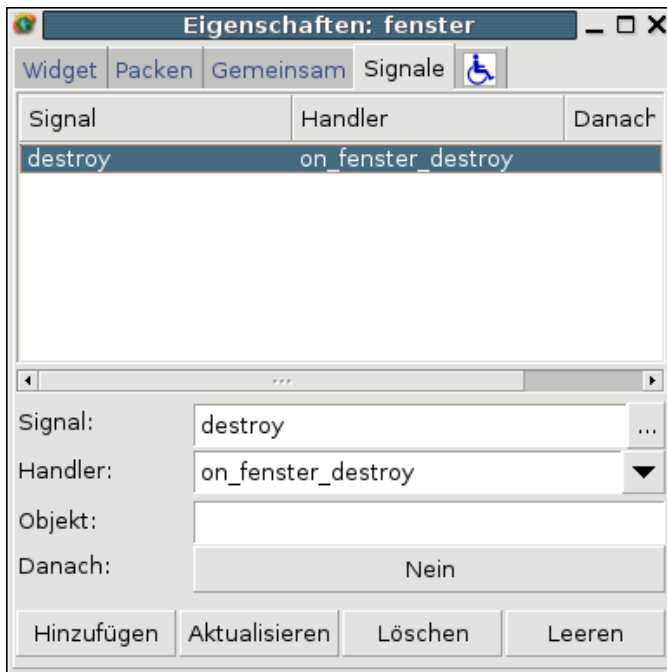


Wähle aus der Palette das Fenster



Es erscheint ein neues Fenster (window1). Ändere nun im Fenster „Eigenschaften“ den Namen auf „fenster“ und den Titel auf „Ein- und Ausgabe“.

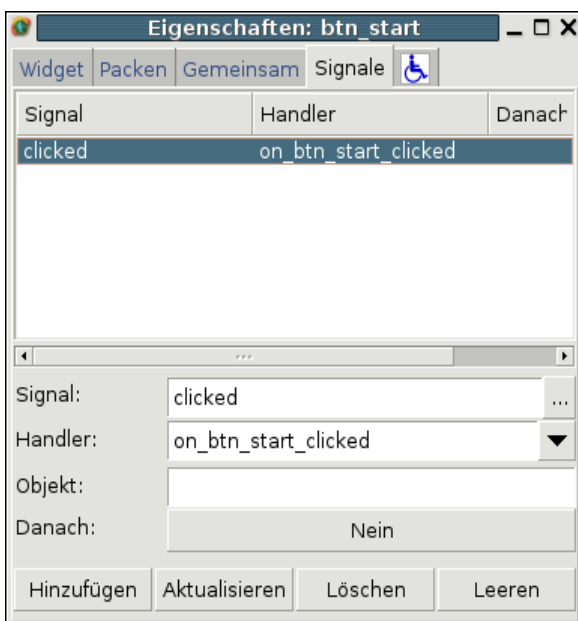
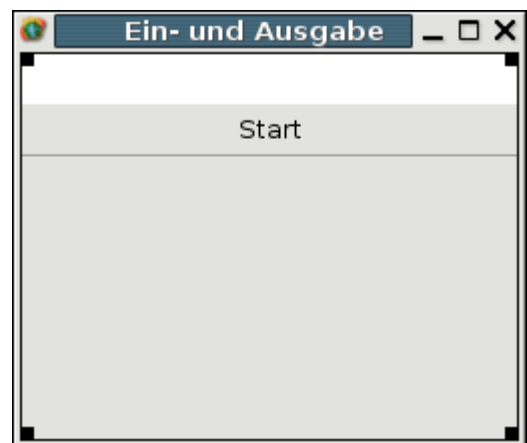
Füge im Reiter Signale das Signal „destroy“ hinzu.



Der Name des Handlers lautet „on_fenster_destroy“, dieser Name sollte später im Programm mit dem Namen der Methode übereinstimmen.

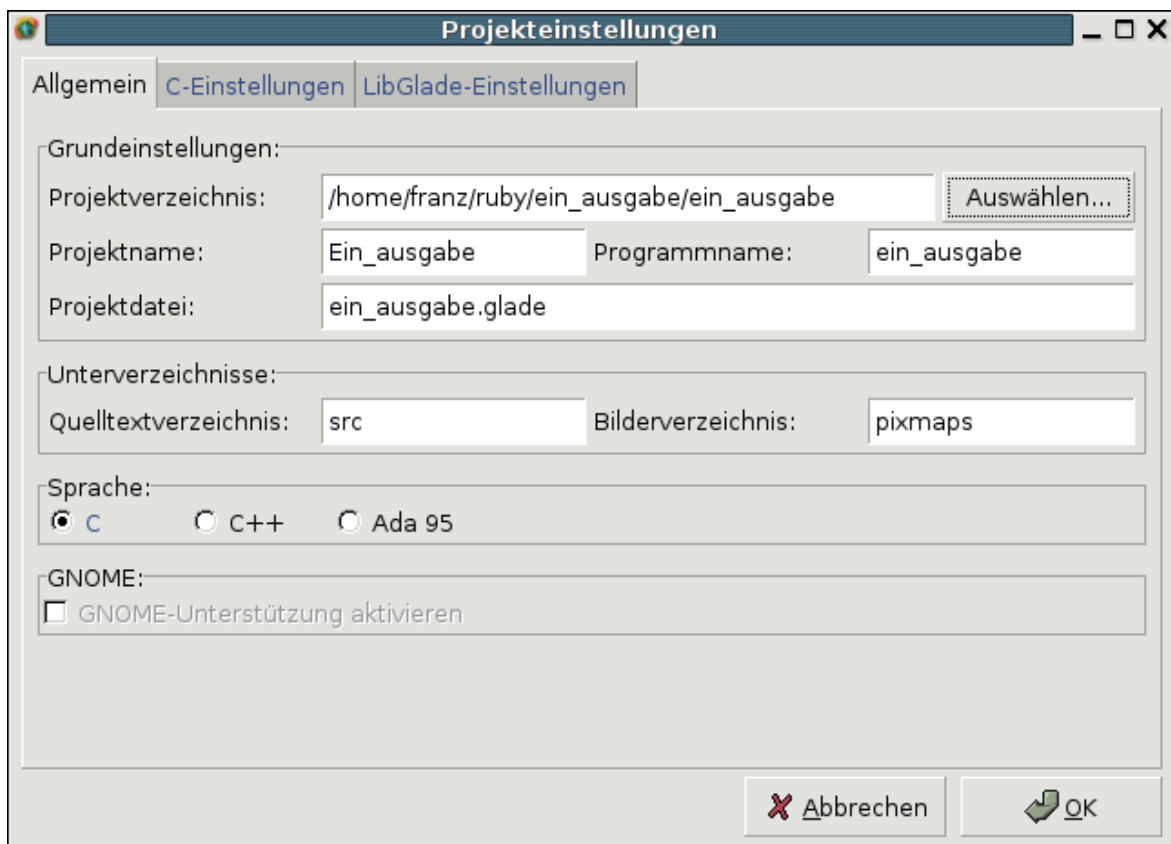
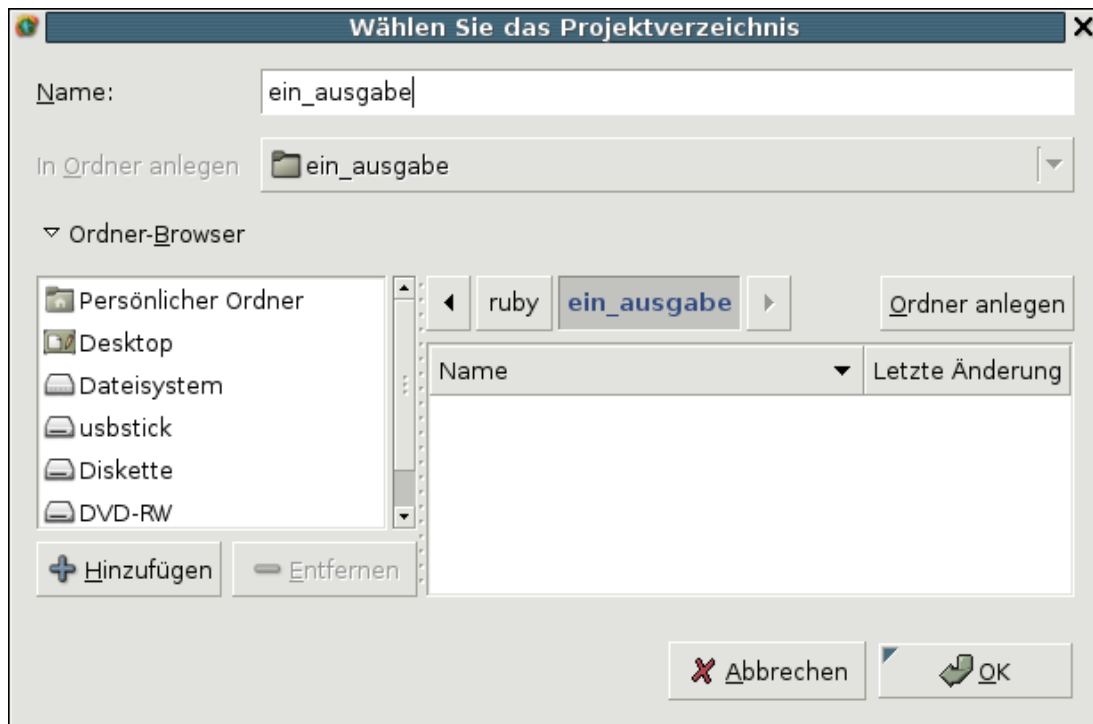
Wähle aus der Palette eine VBox mit drei Zeilen und füge sie ins Fenster ein.

Füge in die VBox ein Entry mit dem Namen „txt_eingabe“, einen Button mit Namen „btn_start“ und Aufschrift „Start“ und ein Label mit Namen „lbl_ausgabe“ ein.



Füge zum Button das Signal „clicked“ hinzu, der Name des Handlers (on_btn_start_clicked) sollte im Programm wieder dem Namen der Methode entsprechen.

Speichere das Projekt beispielsweise in
~/ruby/ein_ausgabe/ein_ausgabe.glade
(„~/“ steht für dein Home-Verzeichnis)

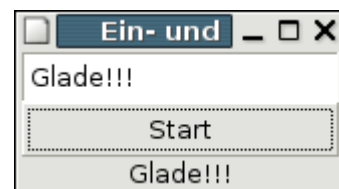


Im angegebenen Ordner wird nun die Datei „ein_ausgabe.glade“ erstellt, in welcher alle Informationen im XML-Format gespeichert sind.

Programmcode

Diese Datei kann in ein Ruby-Programm eingebunden werden, speicher dazu den Code in `~/ruby/ein_ausgabe/ein_ausgabe.rb`

```
ein_ausgabe.rb
(1) #!/usr/bin/env ruby
(2)
(3) require 'gtk2'
(4) require 'libglade2'
(5) include Gtk
(6)
(7) class Fenster
(8)
(9)     #Membervariablen
(10)     @txt_eingabe
(11)     @lbl_ausgabe
(12)
(13)     def initialize
(14)         glade = GladeXML.new( "ein_ausgabe/ein_ausgabe.glade" ) do
(15)             | handler |
(16)             method( handler )
(17)         end
(18)
(19)         @txt_eingabe = glade.get_widget( "txt_eingabe" )
(20)         @lbl_ausgabe = glade.get_widget( "lbl_ausgabe" )
(21)     end
(22)
(23)     def on_btn_start_clicked
(24)         @lbl_ausgabe.set_label( @txt_eingabe.text )
(25)     end
(26)
(27)     def on_fenster_destroy
(28)         main_quit
(29)     end
(30)
(31) end
```



In den Zeilen 14 bis 17 wird die Verbindung zur Glade-XML-Datei hergestellt, im Konstruktor von GladeXML steht schlicht der Name der Datei. Die Zeilen

```
| handler |
method( handler )
```

verknüpfen die Signalhandler von Glade mit jenen des Programms, die Namen müssen, wie bereits angeführt, übereinstimmen.

Die Verknüpfung der „Glade-Widgets“ mit jenen im Programm schließlich erfolgt in den Zeilen 19 und 20:

```
@txt_eingabe = glade.get_widget( "txt_eingabe" )
@lbl_ausgabe = glade.get_widget( "lbl_ausgabe" )
```

Anhang B: Ein Ruby-Skript in ein unter Windows und Linux direkt ausführbares Programm „verpacken“

Mehrere Dateien zu einem Paket zusammenfassen

Ein Ruby-Programm präsentiert sich standardmäßig als ein oder mehrere Skript(e) mit vielleicht noch der einen oder anderen Text- und/oder Bilddatei.

Wenn du mehrere zu einem Programm gehörende Dateien in ein einziges Skript packen möchtest, hilft dir `tar2rubyscript.rb` von Erik Veenstra (<http://www.erikveen.dds.nl>).

Wie das Programm funktioniert, zeige ich dir anhand des in diesem Skriptum enthaltenen Programms `drucken.rb` (Seite 27).

Das Programm besteht aus zwei Dateien, dem Programm selbst und einer Bilddatei. Die Ruby-Datei muss zwingend den Dateinamen „`init.rb`“ tragen:



Begib dich auf der Kommandozeile nun an jene Stelle des Verzeichnisbaums, dass der Ordner „`printer`“ relativ zu deiner Position einen Unterordner darstellt. Auf meinem Computer ist das beim aktuellen Beispiel `/home/franz/ruby/`.

Kopiere das Programm `tar2rubyscript.rb` (du findest es auf der Seite des Autors oder hier: <http://sourceforge.net/projects/tar2rubyscript/>) an die aktuelle Position und führe folgenden Befehl aus:

```
ruby tar2rubyscript.rb printer/
```

Das vollständige Programm nennt sich jetzt „`printer.rb`“, es befindet sich im

aktuellen Ordner. Du startest es wie gehabt mit dem Befehl

```
ruby printer.rb
```

Du kannst es dabei bewenden lassen, wie du jedoch ein nicht auf eine Ruby-Installation angewiesenes, direkt ausführbares Programm erstellen kannst, folgt bei Fuß:

Ausführbare Datei erstellen

Es geht noch einen Schritt weiter: du kannst das gerade gepackte Ruby-Programm oder dein ursprüngliches Ruby-Programm, wenn es aus nur einer Datei besteht, in ein unter Linux und Windows direkt ausführbares Programmpaket verpacken.

Diese ausführbare Datei kommt ohne Ruby-Installation aus, da alles nötige darin enthalten ist.

Lediglich für Programme mit grafischer Oberfläche musst du unter Linux bzw. Windows die entsprechende Bibliothek separat installieren. Für Ruby/GTK-Programme wird unter Windows die GTK+-Laufzeitumgebung (das Entwicklerpaket erfüllt seinen Zweck natürlich ebenso, ist jedoch nicht nötig) benötigt. Die gängigen Linux-Distributionen bringen die Bibliothek bereits mit.

Das Ruby-Programm, das diese Tat vollbringt, kommt vom selben Autor wie das eben behandelte Programm zum Packen und nennt sich `rubyscript2exe.rb`. Du kannst es von der Seite des Autors oder auch hier downloaden:
<http://sourceforge.net/projects/rubyscript2exe/>.

Kopiere es an die Stelle, wo sich dein Ruby-Skript bzw. -Paket befindet und starte es:

```
ruby rubyscript2exe.rb printer.rb
```

Wenn du ein Programm mit GUI umwandelst, wird es während des Vorgangs automatisch gestartet, du kannst es einfach schließen.

Unter Linux wird die ausführbare Datei „`printer.bin`“ erstellt, unter Windows nennt sie sich „`printer.exe`“.

Beide benötigen, wie bereits angesprochen, kein installiertes Ruby mehr. Du kannst die Datei, eventuell inklusive der GUI-Bibliothek, weitergeben.

Weitere Informationen findest du auf den Internetseiten des Autors.

Anhang C: Soundausgabe

SDL (Simple DirectMedia Layer, <http://www.libsdl.org/>) ist eine einheitliche, plattformunabhängige Schnittstelle zur Multimedia- und Spieleprogrammierung.

Mit Ruby/SDL (<http://www.kmc.gr.jp/~ohai/rubysdl.en.html>) existiert eine Anbindung von Ruby an SDL.

Wir sehen uns hier lediglich die Audioausgabe von Ruby/SDL an, abgespielt werden können Audiodateien in verbreiteten Formaten wie Ogg Vorbis und WAVE.

Installation in Linux

Ubuntu-Benutzer installieren Ruby/SDL mit dem Befehl

```
apt-get install libsdl-ruby1.8
```

In Gentoo-Linux befördern die folgenden Befehle alles Nötige auf die Platte:

```
emerge sdl-mixer  
emerge ruby-sdl
```

Für Linux-Distributionen, für die keine Binärpakete bereitstehen, können die Quellen von den Seiten des Projekts herunter geladen und installiert werden. Siehe dazu die Anleitung in der Datei README.en im Quelltextpaket.

Installation in Windows

Der Windowsbenutzer findet auf der Ruby/SDL-Seite das benötigte Paket zum Herunterladen (getestet wurde rubysdl-1.1.0-mswin32-1.8.4.zip). Anschließend kann das (entpackte) Paket mit dem Befehl „ruby install_rubysdl.rb“ installiert werden.

Hinweis: wenn du ein Rubyprogramm mit rubyscript2exe packst und weitergibst, das die Schnittstelle Ruby/SDL verwendet, muss die Datei SDL.dll auf jenen Rechnern in den Windows-System-Ordner kopiert werden, auf denen das Programm laufen soll. Damit sich der Benutzer dies (und evt. die separate Installation der GTK+-Laufzeitumgebung) spart, kannst du einen Installer verwenden, siehe dazu Anhang D.

Im folgenden Beispiel wird eine Vorbis-Datei abgespielt:

```
sound.rb
(1) require 'sdl'
(2)
(3) #Sound laden
(4) SDL::init( SDL::INIT_AUDIO )
(5) SDL::Mixer.open
(6) song = SDL::Mixer::Music.load( "sound.ogg" )
(7)
(8) #Sound abspielen
(9) SDL::Mixer.playMusic( song, 0 )
(10)
(11) #warten, bis der Sound fertig gespielt ist
(12) while SDL::Mixer::playMusic?
(13)     sleep( 1 )
(14) end
```

In Zeile 1 wird die Bibliothek SDL eingebunden, die Zeilen 4 bis 6 initiieren die Audiofunktionen und laden die Sounddatei, in Zeile 9 wird schließlich das Abspielen der Audiodatei veranlasst, das zweite Argument in der Methode „playMusic“ gibt an, wie oft das Abspielen der Datei wiederholt werden soll.

Die Dokumentation findest du unter der URL
http://www.kmc.gr.jp/~ohai/rubysdl_docs.en.html.

Beim folgenden Beispiel handelt es sich um das um Soundausgabe erweiterte Programm „bild2.rb“:

```
bild2_sound.rb
(1) require 'gtk2'
(2) require 'sdl'
(3)
(4) include Gtk
(5)
(6) class Fenster < Window
(7)
(8)     #Membervariablen
(9)     @eb1
(10)    @eb2
(11)    @image
(12)    @pos #aktuelle Position des Bildes
(13)    @sound_klick #Audio
(14)
(15)    def initialize
(16)        super
(17)        self.set_title( "" )
(18)        self.signal_connect( "destroy" ) { main_quit }
(19)
(20)        @image = Image.new( "duck.png" )
(21)        @pos = "eb1"
(22)
(23)        hbox = HBox.new( true, 0 )
(24)
```

```

(25)     @eb1 = EventBox.new
(26)     @eb1.set_name( "eb1" )
(27)     @eb1.signal_connect( "button_press_event" ) do
(28)         | widget, event |
(29)             eb_pressed( widget )
(30)     end
(31)
(32)     hbox.pack_start( @eb1 )
(33)
(34)     @eb2 = EventBox.new
(35)     @eb2.set_name( "eb2" )
(36)     @eb2.signal_connect( "button_press_event" ) do
(37)         | widget, event |
(38)             eb_pressed( widget )
(39)     end
(40)     hbox.pack_start( @eb2 )
(41)
(42)     @eb1.add( @image )
(43)
(44)     self.add( hbox )
(45)     self.show_all
(46)
(47)     #Sound laden
(48)     SDL::init( SDL::INIT_AUDIO )
(49)     SDL::Mixer.open
(50)     @sound_klick = SDL::Mixer::Music.load( "klick.wav" )
(51)
(52) end
(53)
(54) def eb_pressed( w )
(55)     if @pos == w.name
(56)         #Sound abspielen
(57)         SDL::Mixer.playMusic( @sound_klick, 0 )
(58)         sleep( 0.1 )
(59)         if w.name == "eb1"
(60)             @eb1.remove( @image )
(61)             @eb2.add( @image )
(62)             @pos = "eb2"
(63)         else
(64)             @eb2.remove( @image )
(65)             @eb1.add( @image )
(66)             @pos = "eb1"
(67)         end
(68)     end
(69) end
(70)
(71)end

```


Anhang D: Installationsroutine für MS Windows erstellen

Mit dem bereits vorgestellten Skript „rubyscript2exe.rb“ (Anhang B) werden der Ruby-Interpreter und alle vom Programm benötigten Ruby-Bibliotheken inklusive des Programms selbst in eine ausführbare Datei verpackt.

Zwei Mankos bleiben: wenn weitere Bibliotheken wie GTK+ oder SDL benötigt werden, müssen diese eigens mitgeliefert und installiert werden. Weiters muss der Anwender das Programm „zu Fuß“ starten, da es keinen Eintrag im Startmenü gibt.

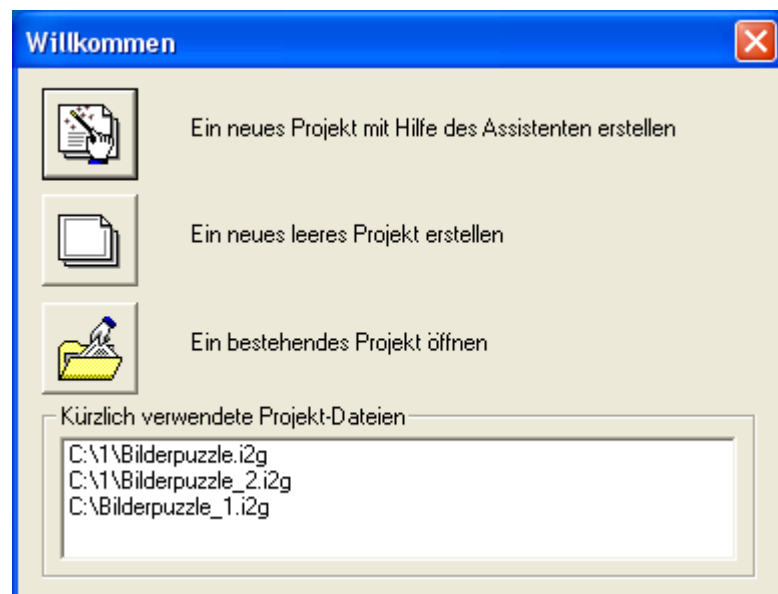
Diesen Punkten begegnen Installer wie „Installer2GO“ (<http://dev4pc.com/>). Eine mit diesem Programm erstellte Setup-Routine kann neben der integrierten Installation beliebiger Bibliotheken auch automatisch Einträge im Startmenü erstellen.

Das eben angesprochene Programm „Installer2GO“ demonstriere ich dir nun mit seinen grundlegenden Funktionen anhand eines Beispiels.

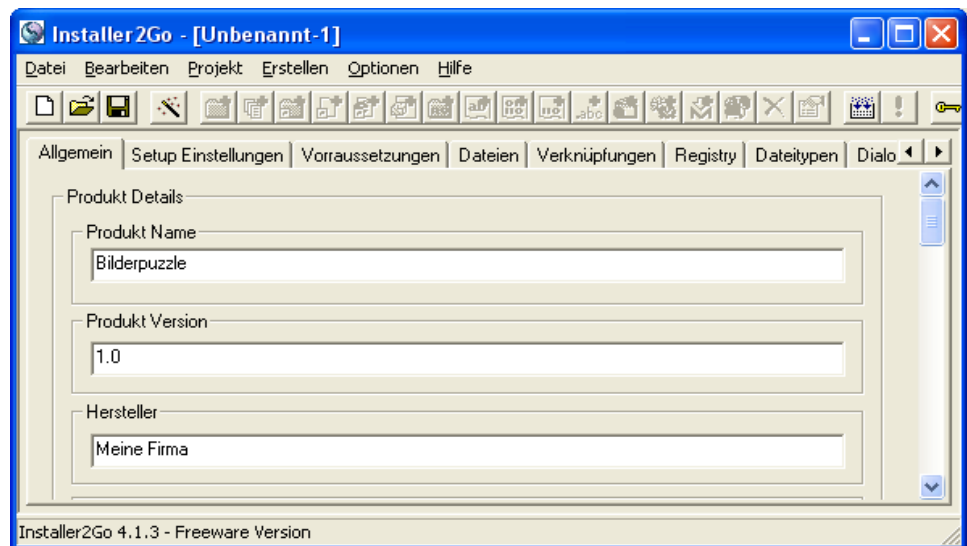
Wir erstellen eine Installationsroutine für ein Bilderpuzzle, das mit Ruby/GTK programmiert wurde und damit der GTK+-Laufzeitumgebung bedarf. Weiters ist zur Soundausgabe die dynamische Bibliothek „SDL.dll“ nötig (soll im Zuge der Installation automatisch Windows-System-Ordner abgelegt werden).

Als Vorarbeit wurde das Programm mit tar2rubyscript und rubyscript2exe gepackt und ausführbar gemacht (siehe Anhang B).

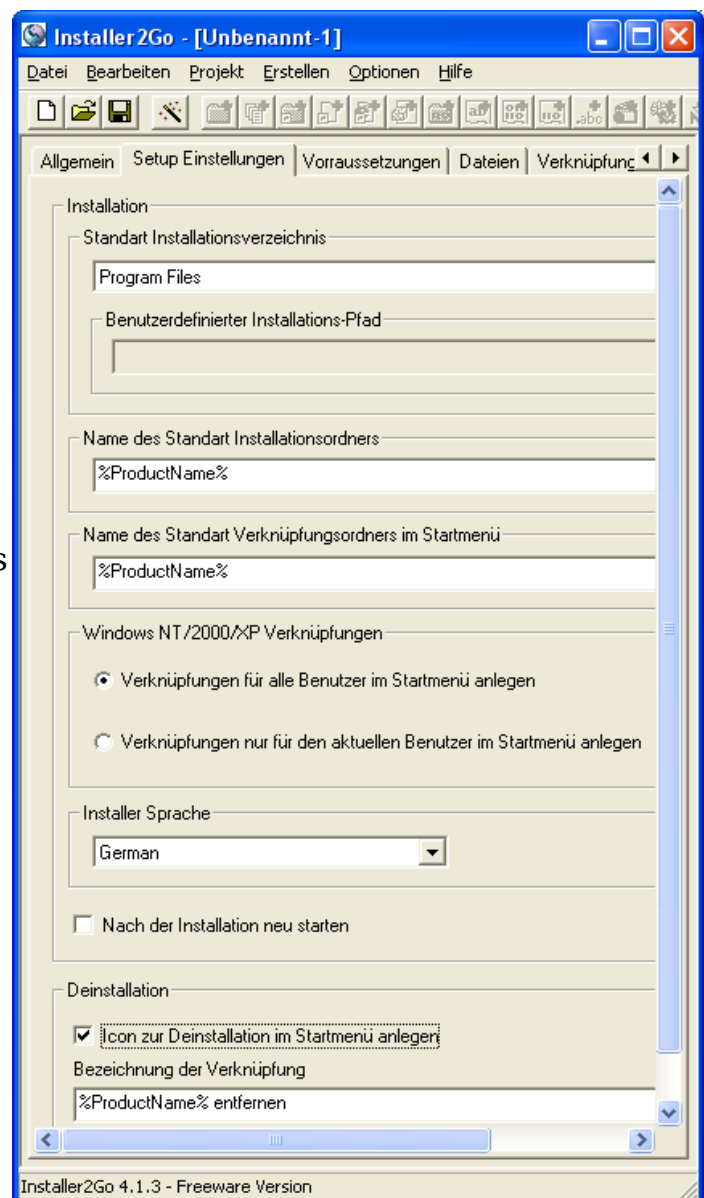
Wähle nach dem Starten des Programms den Punkt „Ein neues leeres Projekt erstellen“:



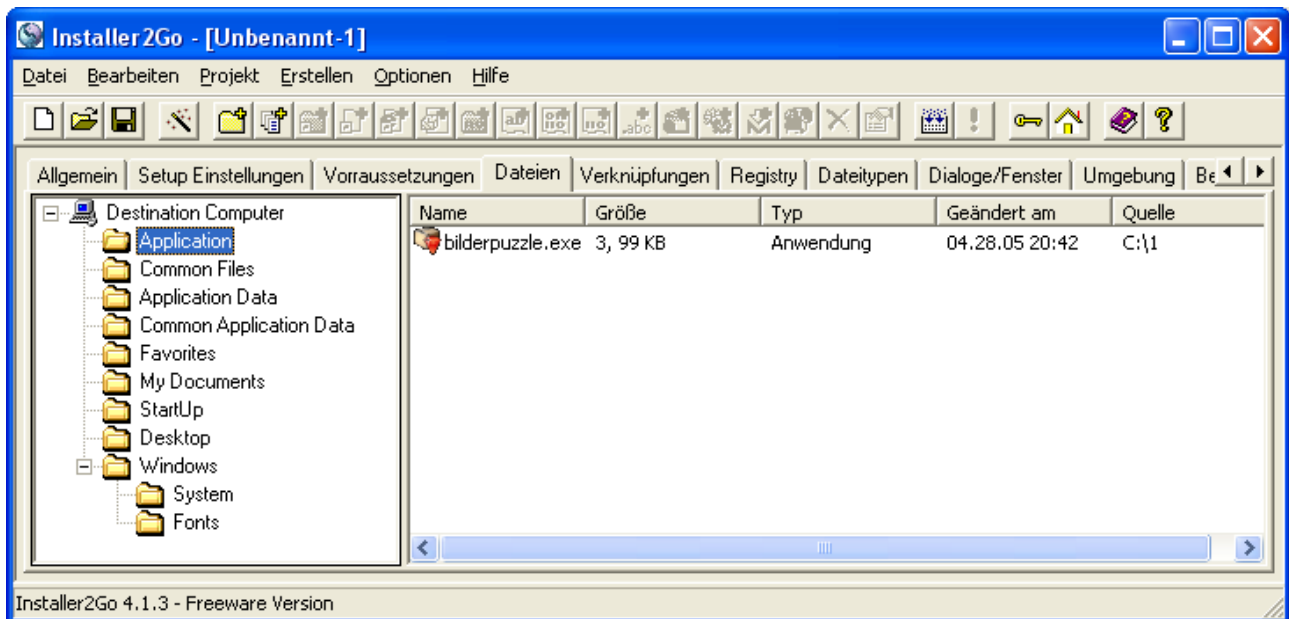
Im Hauptfenster gibst du im Reiter „Allgemein“ den Programmnamen vor.



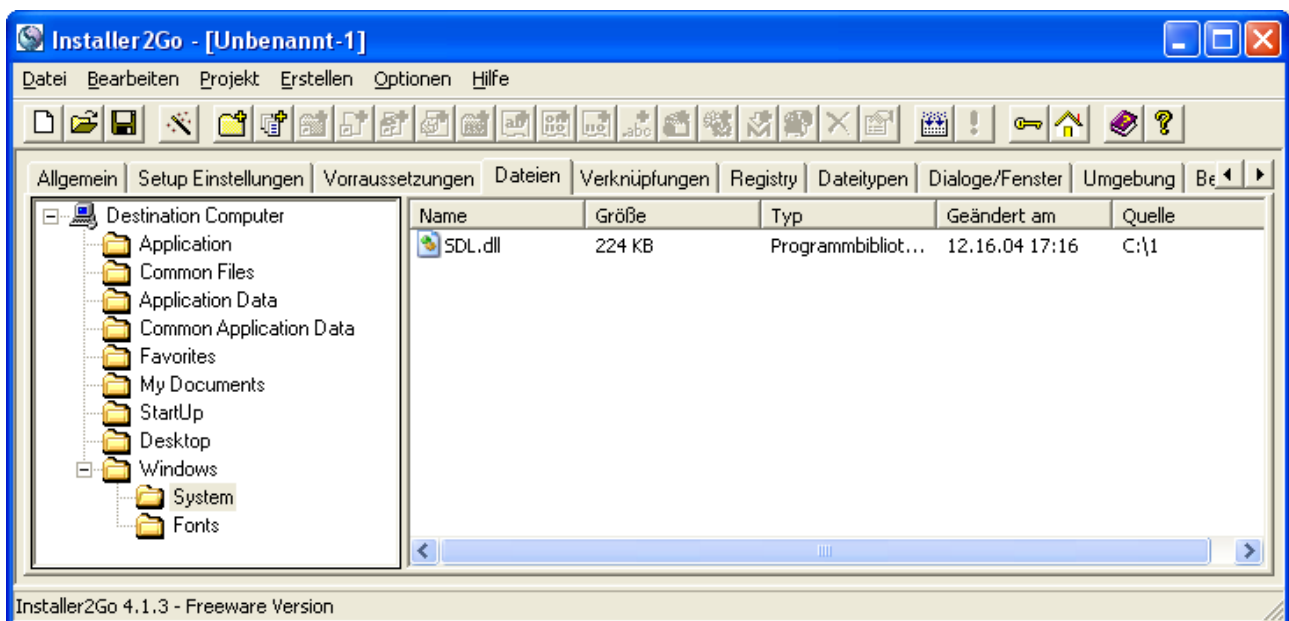
Im Reiter „Setup Einstellungen“ kannst du die Sprache des Installers wählen, weiters bietet sich dir die Möglichkeit, eine Verknüpfung zum Deinstallieren des Programms vorzusehen.



Im Reiter „Dateien“ fügst du bei „Application“ das ausführbare Rubyprogramm hinzu.



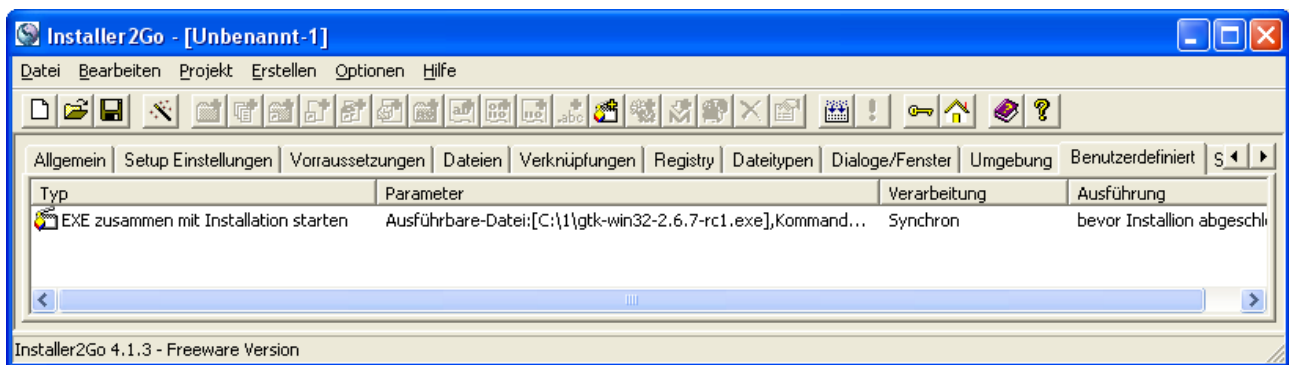
Weiter kannst du dafür sorgen, dass die Datei „SDL.dll“ während des Installationsvorgangs automatisch im Windows-System-Ordner abgelegt wird.



Im Reiter „Verknüpfungen“ definierst du den Eintrag ins Startmenü.



Im Reiter „Benutzerdefiniert“ kannst du mittels „EXE ausführen hinzufügen“ - „Hinzugefügt zu Installation“ die in die Installationsroutine integrierte Installation der GTK+-Laufzeitumgebung veranlassen.



Im Reiter „Setup erstellen“ schließlich kannst du das Erstellen der Setup-Routine starten.

Die erstellte exe- oder msi-Datei kannst du nun weitergeben, wird sie ausgeführt, werden automatisch alle benötigten Dateien installiert – und, nicht zuletzt, es gibt anschließend einen Eintrag im Startmenü.

Inhaltsverzeichnis

Grafische Oberflächen mit Ruby.....	2
Installation der Ruby/GTK-Bindings.....	2
Installation in Linux.....	2
Installation in Microsoft Windows.....	3
Installation in Mac OS X.....	4
Ein einfaches Fenster.....	4
Eine Komponente ins Fenster einfügen.....	7
Signal-Handler.....	7
Dokumentation.....	10
Dokumentation Online.....	10
Ruby Browser.....	10
Installieren.....	10
Aktualisieren.....	10
Starten.....	11
Beispielprogramme.....	11
Mehrere Komponenten in dasselbe Fenster einfügen.....	12
VBox.....	12
HBox.....	13
Table.....	14
Layout.....	16
Eine Callback-Methode für mehrere Signal-Handler	17
Fenster schließen und Schließen verhindern.....	19
Etwas Farbe bitte.....	20
Label.....	22
Image.....	23
EventBox.....	24
RadioButton.....	26
Widgets mit individuellem Icon versehen.....	27
Tooltip.....	28
Scrollbar.....	29
Dialoge.....	32
MessageDialog.....	32
Informationen anzeigen.....	33
Auswahl präsentieren.....	34
Daten eingeben.....	35
Dialog.....	36
AboutDialog.....	38
FileSelectionDialog.....	39
ColorSelectionDialog.....	40
FontSelectionDialog.....	41
Schrift direkt ändern.....	42
CheckButton und Frame.....	44
DrawingArea, Thread.....	45
Combo, Programme direkt aufrufen.....	47
Calendar, Programmfenster ein Icon zuweisen.....	49
ScrolledWindow und TextView.....	50

Menu.....	52
Kontextmenü, Reagieren auf das Drücken verschiedener Maustasten.....	54
Das Signal „key_press_event“.....	57
TreeView.....	58
ProgressBar.....	61
Signale blockieren und entfernen.....	64
Mauszeiger ändern.....	66
Zeichnen.....	67
Linie.....	67
Ellipse.....	69
Anhang A: Oberflächen-Design mit Glade.....	71
Grafische Oberfläche.....	72
Programmcode.....	75
Anhang B: ausführbares Programm erstellen.....	76
Mehrere Dateien zu einem Paket zusammenfassen.....	76
Ausführbare Datei erstellen.....	77
Anhang C: Soundausgabe.....	78
Installation in Linux.....	78
Installation in Windows.....	78
Anhang D: Installationsroutine für MS Windows erstellen.....	81