



Programmieren mit Ruby

GUI Toolkits für Ruby

- [Einführung](#)
- [Verwendung Standard Ruby GUI: TK](#)
- [Verwendung GTK+ Toolkit](#)
- [Verwendung FOX Toolkit](#)
- [Verwendung SWin/VRuby Erweiterungen](#)
- [Andere GUI Toolkits](#)
- [Zusammenfassung](#)
- [Schneller Überblick über die Themen](#)
- [Häufig gestellte Fragen\(FAQ\)](#)

Einführung

Wenn auch Ruby ein ausgezeichneter Tool für die Programmierung von "low-level" Skripts für die Systemadministration ist, ist die Sprache genauso für die Programmierung von Anwendungsprogramme anzuwenden. Und weil Fenstergesteuerte Programme (graphical user interfaces) ein "Muss" für moderne Anwendungsprogramme sind, sollten Sie sich beibringen, wie man GUIs mit Ruby entwickelt. Ein von mehreren Vorteilen der Rubyprogrammierung ist es, Sie kriegen es zu spüren, dass es möglich ist, die Entwicklungszeit des gewünschten Programmes zu reduzieren (rapid application development, RAD). Im Gegensatz zum zeitaufwendigen "code compile test" ("codieren compilieren testen") Zyklus einer traditionellen Programmiersprache, können Sie schnell Änderungen vornehmen in Ihrem Rubyskript um neue Ideen auszuprobieren. Dieser Vorteil wird ganz mehr offensichtlicher wenn Sie anfangen, GUI applications mit Ruby zu entwickeln. Schrittweise grafische Oberfläche zu entwickeln, hinzufügen eines neuen Elements und Restarting des Programmes um ein sofortiges Resultat zu sehen, wie GUI sich verändert, kann man das alles schnell lernen.

Sie dürfen bereits wissen, dass Standard Ruby Distribution eine Schnittstelle zu TK beinhaltet, einer Schnittstelle, die schon längst al ein "cross-platform" GUI Toolkit gilt. Wenn Sie sich jedoch in Ruby Application Archive (RAA) nachsehen, würden Sie sich schnell entdecken, dass es mehrere GUI Toolkits für Ruby gibt. [RAA](#) Warum sollte man sich nicht nur mit TK beschäftigen obwohl der ein Standard ist? Ja nun, während Sie dieses Kapitel durcharbeiten, würden Sie sich es überlegen, dass man auch weitere Alternativen anwenden kann. Diese GUI Toolkits befinden sich in mehreren Stadien der Entwicklung. Einige sind neue und instabil, während andere sind alt und ganz robust, jedoch in meisten Fällen irgendwo dazwischen. Meistens sind GUI Toolkits für Ruby cross-plattformig, gemeint wird es damit, dass die Programme so aufgebaut sind, dass sie auf Multi-User-Plattform laufen, während andere auf Singl-User-Plattform laufen.

Jeder GUI Toolkit bringt mit sich eigene einzigartige Eigenschaften und Besonderheiten, jedoch es gibt

einige Merkmale, die in fast allen GUI Toolkits vorhanden sind, mit denen Sie arbeiten würden. GUI Applicationen sind ereignisgesteuert. Viele Programme werden Sie fortgehend von Start bis Ende schreiben, dabei wissen Sie, was Sie vom Programm wollen, das eine festgesetzte Menge der Information einnimmt (Input) und gleiche Menge von Output erzeugt mit wenigem oder gar keinem Eingreifen von Anwender (User). Zum Beispiel, betrachtet man ein Rubyskript das zum Verarbeiten grossen Anzahl von Textdateien in der Stapelverarbeitung geschrieben wurde, eventuell auch "updating" eines ausgewählten Abschnittes in diesen Dateien und das auch ohne Eingreifen vom User. Im Gegensatz zu solchen Programmen verbringen "event-driven" Applicationen ihre meiste Zeit im "Nichtstun", bis der Anwender was unternimmt bzw. ein Input erzeugt, was das Fortfahren des Programms veranlässt. Jeder Toolkit realisiert auf seine eigene Art die Kommunikation zwischen einem Ereignis, welches ein Anwender auslöst, und einem Programmecode, welcher mit einer bestimmter Funktion festgeknüpft ist und somit es ist eine gewisse Kommunikation miteinander gewährleistet. GUI Toolkits bestehen aus grosser Anzahl von elementaren Objekten, sogenannten "Widgets" wie "Schaltflächen", "Beschriftungen", und "Textfeldern", als auch aus noch mehr vielseitigen Widgets, wie "Tabellen", "Kalender" oder "Texteditoren". Der Aufbau von grafischen Oberflächen erfolgt auf dem Basis "Vater-Kind"-Beziehung. Es gibt einen Toplevel-Widget als "Main-Window" bezeichnet, der ein oder mehrere "Kind" Fenster beheimatet. Jedes "Kind" Fenster kann auch weitere "Kind" Fenster enthalten und so geht's weiter. Das Prinzip somit ist eine Verwendung solchen "Zusammengesetzten Muster" und die Handhabung ist anwendbar zu den "Vater-Fenster", die gewöhnlich "Kind-Fenster" beeinflussen. GUI Toolkit bieten mehrere Optionen für die Gestaltung "Kind-Fenster" innerhalb des "Behälterfensters" an.

Das Ziel dieses Kapitels ist es einige von meist populären GUI Toolkits für Ruby vorzustellen, wie man häufige Sprachelemente anwenden kann, ein Erörtern über dessen Ausführung und das Helfen bei der Entscheidung, welcher von GUI Toolkits am besten Ihren Vorstellungen entspricht. Um das zu machen, werden wir zuerst ein einfaches Programm beschreiben das eine Menge von Merkmalen und Funktionen hat, was Sie in anderen GUI Programmen anwenden würden. Danach werden wir einen Blick darauf werfen, wie Sie dieses Programm in jeder von diesen Toolkits entwickeln könnten. Weiter werden wir einige andere Themen besprechen wie man diese GUI Toolkits an Ihrem System herunterladen, installieren und kompilieren kann und auch zusätzliche Tools (solche wie GUI Builder), mit denen man die Entwicklung bequemer und schneller machen kann. Eine Quelle für weitere zusätzliche Information und Ressourcen

kann in einem Forum des jeweiligen GUI Toolkits gefunden werden.

Verwendung unseres Beispielprogrammes

Für jeden von diesen vier grossten Tools(Tk, GTK+, FOX und SWin/VRuby) werden wir ein kleines gleiches Programm entwickeln, so dass Sie schnell Ähnlichkeiten und Unterschiede zwischen Toolkits erkennen können, während Sie lernen, wie man sie benutzen kann. Das Programm ist ein einfacher XML Betrachter, das NOXML Modules von Jim Menard verwendet als seine Dokumentvorlage, also Sie müssen sich Erweiterungen verschaffen und sie installieren, bevor Sie eigentlich das Beispielprogramm auf Ihrem System laufen lassen können. Um Ihnen das benötigte Wissen zu übermitteln, damit Sie Ihre eigene Programme entwickeln, wird am Beispiel dieses XML-Betrachters folgende allgemeine Merkmale gezeigt:

- Anzeige Menüleiste, mit verschiedenen Pulldownmenüs z.B. zum Öffnen XML Dateien oder zum Verlassen des Programmes
- Verwendung des allgemeinen Dialogboxes, wie z.B. "Datein-Öffnen" Dialog
- Verwendung des Gestaltungsmanagers zur automatischen Anordnung von Widgets
- Verwendung verschiedenen Widgets zum Zeigen XML-Dokumentknotes und ihren Attributen

Verwendung Standard Ruby GUI:TK

Der Standard für grafische Oberfläche, der in Rubyprogrammen am häufigsten erscheint, ist TK. TK wurde speziell als GUI für die in der Mitte achtziger entwickelte Tcl-Skriptsprache von John Ousterhout bestimmt, seitdem aber ist Toolkit fast auf allen gängigen Plattformen integriert worden, genau so kann Tk mit anderen Skriptsprachen Perl und Python verwendet werden. Tk's Widgets sind mit mehr moderneren GUI's vergleichbar und es unterstützt sogar MAC OS.

Installation

Ein von wesentlichen Vorteilen der Verwendung Tk mit Ruby ist es, dass Tk Standard Toolkit für Entwicklung GUI's mit Ruby ist und es ist leicht und ohne grosse Aufwand das Programmieren anzufangen. Ruby/Tk benötigt Tk als Selbstpaket sowie als Ruby/Tk Erweiterungsmodul. Sie können sich Source code für Tk bei der Tcl/Tk Homepage www.tcltk.org besorgen und kompilieren Sie es dabei selbst, aber es gibt schon fertige Binäres für Tk und zwar für alle gängige OS(Linux, Windows). Um sich leichter zu machen, empfiehlt es sich Standard Ruby's Distribution für Windows bei der "[Pragmatic Programmers' site](#)". Sobald Sie es installieren, haben Sie bereits eine funktionfähige Tk's Umgebung. Viele Linux Distributionen bieten auch Tcl/Tk als Standard Option bei der Installation des Betriebesystemes. Die andere Möglichkeit Tk zu installieren, was allerdings ein wenig komplizierter ist, ist das Erweiterungsmodule für Ruby, die in Ruby's Source Code vorhanden sind. Wenn Sie sich selbst Ruby kompilieren würden, werden Ruby/Tk Erweiterungsmodule automatisch in Ihrem System installiert. Standard Ruby Installer für Windows also beinhaltet Ruby/Tk Erweiterungsmodule

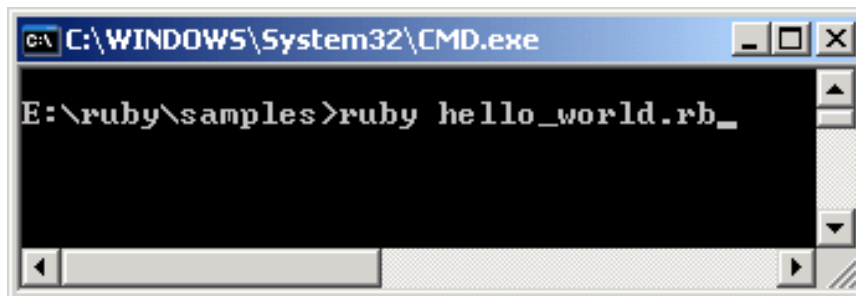
Ruby/Tk Basis

Ruby/Tk stellt eine Anzahl von Klassen zur Verfügung um verschiedene Tk Widgets zu präsentieren. Dabei sind Namen von Ruby/Tk Klassen für Widgets gleich Tk Basis Klassen. Zum Beispiel "Entry-Widget" von Tk

ist auch als *TkEntry* Klasse in Ruby/Tk vertreten. Ein typischer Aufbau von einem Ruby/Tk Programm ist eine Erzeugung eines "main" oder "root" Fensters(als Beispiel *TkRoot*-Klasse), dann werden andere Widgets zu dem "root" Fenster hinzugefügt und damit wird "user interface" aufgebaut, und am Schluss des Codes wird sogenannte "main event loop" mit *Tk.Mainloop* aufgerufen. Das traditionelle Programm "Hello Word" für Ruby/Tk wird so aussehen:

```
require 'tk'
root = Tkroot.new
button = TkButton.new(root) {
  text "Hello, World!"
  command proc {puts "Ich sage Hello!" }
}
button.pack
Tk.mainloop
```

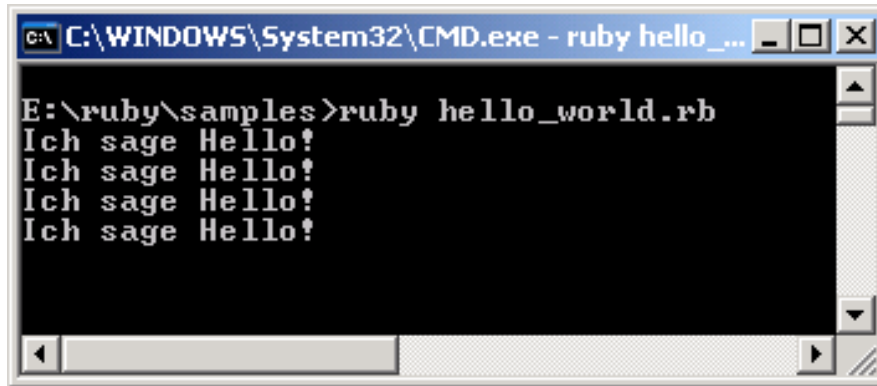
Nun wird and der Shell oder MS-DOS diese Datei, die wir z.B. *hello_world.rb* nennen würden aufgerufen. Und so könnte es aussehen:



Nachdem Sie das Skript aufgerufen haben, erscheint GUI:



Und wenn Sie nun die Schaltfläche drücken, sieht es so aus:



```
C:\WINDOWS\System32\CMD.exe - ruby hello_...
E:\ruby\samples>ruby hello_world.rb
Ich sage Hello!
Ich sage Hello!
Ich sage Hello!
Ich sage Hello!
```

Die erste Zeile lädt einfach Ruby/Tk Erweiterung in den Ruby Interpreter und zweite Zeile errichtet "top-level window" (Hauptfenster) für das Programm. Schließlich kommen wir zum interessantesten Teil:

```
button = TkButton.new(root) {
  text "Hello, World!"
  command proc { puts "Ich sage Hello!" }
}
```

Hier errichten wir ein Button, dessen Eltern's Widget das Hauptfenster ist. Genauso wie bei anderen GUI Toolkits, werden wir sehen, dass Ruby/Tk ein Modell verwendet, das sich auf "Zusammensetzung" basiert, wo die Eltern's Widgets ein oder mehrere Kinders Widgets enthalten, die auch als "Behälter für andere Widgets zur Verfügung stehen können. Dieser Codesteil zeigt nur einen Weg, wie man bei der Configuration von verschiedenen Optionen vorgehen kann. Innerhalb vom Codeblocks können Sie Methoden aufrufen, die das Aussehen des Erscheinens oder des Verhaltens von Widgets ändern. In unserem Beispiel die Methode `text` wurde verwendet um den Text in Button erscheinen zu lassen, während die `command` Methode mit einer Prozedur verbunden, des sogenannten "Callback" (mehr darüber wird später erklärt). Als abwechselnde aber gleichwertige Form zum Angeben von Widgets Optionen ist das, dass man sie als `key, value` (Hash-style Paare) verwendet werden als zweite und weitere Argumente für die `new` Funktion und das als Folgendes aussieht:

```
button = TkButton.new(root, text => "Hello, World!",  
command => proc { puts "Ich sage Hello!"})
```

Die zweite Zeile ist wichtig, weil sie dem Hauptfenster eine Anweisung gibt, den Button am richtigen Platz zu setzen. Für diesen Beispiel ist das natürlich nicht so relevant, da das Hauptfenster nur ein Kind-Widget hat. Wie zum Beispiel wir später sehen werden, haben wirkliche Programme viel mehr komplizierte Layouts, die noch mehrere Ebenen haben und auch ihrerseits andere Widgets beinhalten. Für diese Fälle werden wir auch zusätzliche Argumente zur *pack* Methode übergeben, um hinzuweisen, wo die Widgets in Eltern-Widgets platziert werden sollten, wie die Größe des Buttons geändert werden könnte, wenn die Größe des Eltern's Widget geändert wurde und auch andere mit Layout zusammenhängende Aspekte.

Dieses Beispielprogramm endet, wie die meisten Ruby/Tk Programme es tun, mit dem Aufrufen *Tk.mainloop*;. Aber die Methode greift eigentlich schon am Beginn des Programmes. An dieser Stelle befindet sich das Programm in einer sogenannten unbegrenzten Schleife, wartet auf Ereignisse, die vom Anwender losgelöst werden und dann werden sie an den passenden "Handler" abgeschickt. Viele Programme, die mit Ruby/Tk entwickelt wurden, haben schon eine gewisse Struktur an Ereignissen, und dann wird der Code dafür geschrieben um diese Ereignisse zu führen. Das ist das Thema des folgenden Teil.

Integrieren von Ereignissen und Aufrufen in Ruby/Tk GUI's

Tk's Ereignis Model ist auf zwei eng zusammenhängende Abläufe getrennt. Auf einer Seite das Betriebssystem erzeugt auf niedriger Stufe ein Ereignis wie z.Beiispiel: "der Mauscursor bewegt sich gerade in einem Fenster oder der Anwender drückt einfach eine Taste "S" auf der Tastatur". Auf anderer Seite ruft Tk in Ihrem Programm ein Ereignis um darauf hinzuweisen, dass etwas Bedeutsames in Widget geschehen ist.(z.B. Button wurde angeklickt). Für beide Fälle können Sie einen Codeblock oder einen Ruby *Proc* Objekt schreiben, das genau angibt, wie das Programm auf das Ereignis oder den Aufruf reagiert.

Zuerst lassen Sie uns einen Blick werfen, wie die Methode *bind* verwendet wird um

betriebsystemspezifische Ereignisse mit Ruby's Prozeduren zu verbinden, welche dementsprechend darauf reagieren. Die `bind` Methode wird in einfachster Form für die Einbindung des Ereignisses und des Codeblockes eingesetzt. Daraufhin führt Tk das Codeblock, was als eine Antwort auf das entsprechende Ereignis ist. Zum Beispiel um ein `ButtonRelease` Ereignis für den ersten Mausbutton in irgendeinem Fenster zu packen, würden so schreiben:

```
someWidget.bind('ButtonRelease-1') {  
  ...Codeblock...  
}
```

Für manche Ereignistypen ist es ausreichend, den elementaren Ereignisnamen zu verwenden, wie z.B. "Configure" oder "Destroy", aber für andere brauchen wir mehr spezifischen Namen. Dabei kann der Eventsname auch zusätzliche `modifiers` und `details` enthalten. Der "Modifier" ist ein "String" wie "Shift" "Control" oder "Alt", welches hindeutet, was dabei gedrückt wird. Im Detail wird eine von 1 bis 5 Nummer ausgewählt, welche auf die Nummer von Mausbutton hinweisen, oder kennzeichnen eine Taste auf der Tastatur. So z.B. um ein Event abzufangen, welches erzeugt wird, wenn der Anwender die **Ctrl** Taste festhält und dabei die Rechtenmaustaste betätigt (manchmal als Button 3 genannt wird) über das Fenster, würde man so geschrieben:

```
aWindow.bind('Control-ButtonPress-3', proc {  
  puts "Wauuu!!!" })
```

Die Namen von diesen Events sind von den Namen entsprechenden X11 Eventstypes abgeleitet worden, hauptsächlich aus historischen Gründen; Tcl/Tk war ursprünglich für Unix Betriebssystem entwickelt und sein X Window System. Danach wurde Tk auf Windows und Macintosh portiert und diese Portierungen verwenden gleiche Eventsnamen für die Darstellung ihrer "native" Ereignissen, die auf jeweiliges System beheimatet sind. Das Beispielprogramm, das wir später entwickeln werden, verwendet ein paar Eventstypen, aber wenn Sie sich für die komplette Liste von aktuellen Eventsnamen, `modifier` und Details interessieren, sollten Sie "manual pages" für Tk's `bind` command zur Rate ziehen. Eine gute Online-Quelle für diese Art von Referenzen ist die Tcl/Tk Dokumentation auf "Tcl Developer Xchange Webseite" ([siehe](#)

[hier](#))

Es ist natürlich nützlich, solche Art von "low-level" Events abfangen zu können, aber mehr häufiger werden Sie sich mit "high-level" Aktionen beschäftigen. Zum Beispiel Sie hätten gern einfach gewusst, wenn der Anwender auf den **Help** Button klickt, wobei Sie brauchen wirklich nicht zu wissen, wie es abläuft, wennn der Anwender mit linken Maustaste diesen Button anklickt und dann einige Millisekunden später den loslassen Viele Ruby/Tk Widgets können *callbacks* auslösen, wenn der Anwender sie aktiviert und Sie könnet *command callback* genau angeben, welche bestimmte Codeblocks oder Prozeduren aufgerufen werden müssen und was danach geschieht. Z.Beispiel mit einer beliebigen Option können Sie festlegen *command callback* Prozedure, wenn Sie ein Widget zaubern:

```
helpButton = TkButton.new(buttonFrame) {  
  text "Help"  
  command proc { showHelp }  
}
```

oder Sie können es später die Verwendung Widget's *command* Methode festlegen:

```
helpButton.command proc { showHelp }
```

Sie können aber auch disen Code so schreiben, seitdem *command* Methode es erlaubt:

```
helpButton = TkButton.new(buttonFrame) {  
  text "Help"  
  command { showHelp }  
}
```

Viele Widgets, wie *TkCanvas*, *TkListbox* und *TkText* können nicht verwendet werden, um z.B. ihren Inhalt im ihnen zugewiesenen Raum zu zeigen. Zum Beispiel, wenn Sie *TkText* Widget für die Darstellung eines langen Textes verwenden, werden höchstens nur ein Teil von Text sehen. Deswegen werden Sie auf

Hauptwidget für ein paar Seiten einen typischen *TkScrollbar* beifügen. Damit wird es dem Anwender eine Möglichkeit gegeben, durch den gesamten Inhalt zu scrollen. Um alles das richtig zu machen und zwar das Zusammenspielen von "Scrollbalken" und Widgets wie *TkText* und *TkListbox* kann man also *callbacks* erzeugen wenn der Inhalt von Widgets vertikal oder horizontal gescrollt werden muß. Um die *callbacks* mit Codeblocks zu verbinden können Sie Widget's *xscrollcommand* or *yscrollcommand* Methode verwenden. Wir werden es uns später bei unserem Beispielprogramm ansehen, wie das alles funktioniert.

Arbeitsweise mit Ruby/Tk Layout Manager

Wenn Sie Ihres Programm entwerfen, ist es wichtig zu verstehen, wie Ruby/Tk Events und Callback's funktionieren. Ebenso ein relevanter Aspekt beim Entwurf ist die Gestaltung von "User Interface". Viele Widgets dienen als interaktive Komponenten, when der Anwender darauf klickt oder tippt etwas um das Programm zu bringen, irgendwelche Aktion auszuführen. Andere Widgets, wie auch immer, sind mehr "passiv" und könnten als Behälter(Eltern) für schon anderer Widgets. Wir haben bereits gesehen, dass "top-level" "root" Widget ist einer von solchen Widgets und innerhalb des Hauptfensters können Sie TkFrame Widget verwenden. Damit werden die Kinder Widgets zusammengruppiert.

Der Layout Manager für den Behälter definiert im Grunde Leitfaden und wird zum Festlegen der Position und Größe im Hauptfenster vorhandenen Widgets Wie Sie sehen werden ,gleich nachdem Sie verstanden haben, wie Layout Manager funktioniert dass es Einiges an Experimentfreude gibt und Sie werden Möglichkeiten haben, um Ihre geistige Vorstellungen,was GUI's betrifft,in Code zu verwirklichen und sie richtig zu implementieren. Mit Ruby/Tk können Sie aus drei verschiedenen Layout Manager auswählen, obwohl Sie nicht brauchen denselben Layout zu verwenden. In Wirklichkeit für wichtigen GUI's ist es ziemlich wahrscheinlich, dass Sie mehrere Layouts Manager verwenden werden.

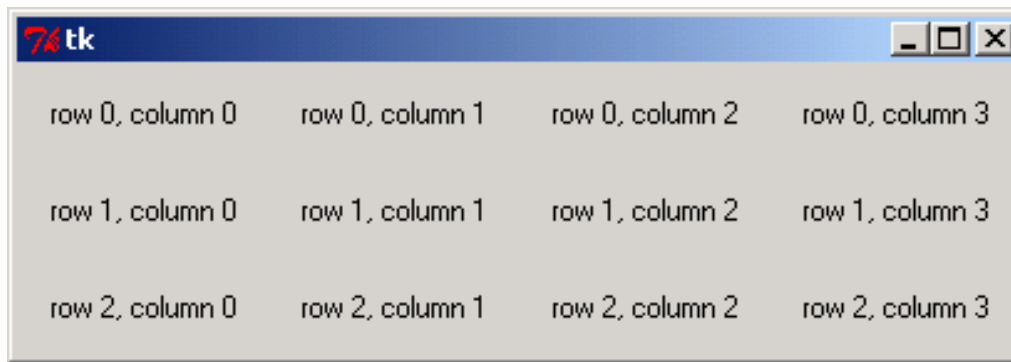
Der einfachste Layout Manager ist *placer*, mit dem Sie lediglich Kinder Elementen plazieren würden, bzw. die Position und Größe festlegen. Auf ersten Blick könnte es angemessen klingeln, doch viele "GUI Builder" Tools dass sie Ihnen erlaubt Widgets auf Toolpalette wegzuzerren, oder fallen lassen auf dem Arbeitsfenster, verwenden diese Methode in dem Code, während sie den erzeugen. Nachteile diesen unflexiblen festen Layout Managers werden offensichtlich, wenn man sobald versucht, das Programm

auf anderen Computer mit anderer Konfiguration und wahrscheinlich auch dass da ein anderes OS ist, zum Laufen zu bringen. Zum Beispiel, wenn Systemfonts verschieden sind, der Button erfordert auf Ihrem System für den Text nur 40 px Breite auf anderen System könnte es 60 px sein. Wenn Sie einen Textfeld fest zu diesem Button unmittelbar rechts verankert haben, werden jetzt die zwei Widgets sich gegenseitig überlappen. Da es so unflexibel ist, werden Sie höchstwahrscheinlich nicht diesen Layout Manager im Praxis benutzen.

Der nächste Layout Manager ist *grid*, welcher seine Kinder Widgets in einer tabellenähnlichen Ordnung platziert. Wenn man Kinder Widgets zu dem Elternfenster hinzufügt, legt man sozusagen Spalten und Reihen fest, die jetzt mit diesem Layout Manager für die Platzierung von Elementen verwendet werden können. Die Kinder Elemente werden so zugeordnet, dass alle Elementen in einer oder mehreren gleichen Spalten und Reihen mit gleichen Breite und Höhe zu finden sind. Um den schnellen Eindruck zu beschaffen, bietet sich hier ein Beispiel, mit *grid* Layout und mit 3 Reihen und 4 Spalten auf "Label" Widget:

```
require 'tk'
root = TkRoot.new
3.times { |r|
  4.times { |c|
    TLabel.new(root) {
      text "row #{r}, column #{c}"
    }.grid('row' => r, 'column' => c,
          'padx' => 10, 'pady' => 10)
  }
}
Tk.mainloop
```

Unten sieht man ein Ergebnis, wenn dieses kleines Programm laufen lässt:



Der *grid* Layout Manager ist aber mehr leistungsfähig, als man an diesem Beispiel erkennen lässt. Sie können sehen, dass man bei Grundeinstellungen so ist, dass *grid* jedes Kind Element in seiner Zelle zentriert, aber man kann zusätzliche Argumente verwenden und so kann *grid* Methode festsetzen, dass ein oder mehrere Seiten vom Kind-Element wie von einem "Kleber" ausgedehnt werden können. Außerdem *grid* Zellen können mehrere Reihen und Spalten umfassen. Um mehr darüber eine Information zu beschaffen, sollte man in Tk Referenz nachschauen.

Der letzte Layout Manager, den wir besprechen werden, ist *packer*. Ihn werden Sie häufiger verwenden, weil er sehr flexibel sogar leichter zu benutzen ist. Der *packer* verwendet als ein Model zur Zuteilung der Position und Größe sozusagen "Hohlraum". Man kann sich einen leeren Raum vorstellen, bevor man die Kinder Element hinzugefügt werden. Wenn man danach das erste Kind-Element hinzufügt legt man die Position der Seiten vom den Rand des rechtwinkligen "Hohlraums"(links, rechts, oben, unten) fest. Der *packer* weist dem Kind-Element die gesamte Seite zu und dann verringert die Größe der Seite um die Menge, die das erste Kind-Element beansprucht hat. Das ist wichtig zu merken, dass der *packer* die ganze Seite des Raums für den neuen dazugekommenen Kind-Element zuteilt, selbst wenn das Kind-Element nicht braucht. Die aufeinanderfolgende Kind-Elemente werden also gegen ausgewählte Seiten des übriggebliebenen Raums drangepackt, bis es mehr den Platzt gibt.

Also ist es ganz relevant, zu verstehen, dass der *packer* Layout Manager zwischen *packing space* und *display space* unterscheidet. Der "Display-Raum" ist ein Teil des Raums, das bestimmtes Kind-Element für sich genommen hätte, um sich selbst korrekt zu präsentieren. Zum Beispiel ein "label" oder "button" Element hat ein "display space", das ein wenig weiter als das benötigte Raum für die Darstellung von Text

auf "label" oder "button" Widgets ist. Das "packing space" ist dagegen das gesamte Raum das verfügbar für die Positionierung von Kind-Element im Hohlraum ist und das kann größer oder kleiner als "display space" für den Widget sein.

Wenn "packing space" seine Maße als dazu benötigte "display space" überschreitet, tritt das "default" Verhalten ein und zwar wird das Kind-Element gerade ins Zentrum von "packin space" gesetzt und das übrige Raum von "packin space" wird anderen Seiten verschoben. Wenn Sie stattdessen genau so für das Kind-Element das ganze zur Verfügung stehende Raum füllen möchten, dann kann man den Parameter *fill* setzen mit drei Werten (x, y, both), die jeweilige Richtung hindeuten, in welcher das Element gefüllt wird. Wir werden das sehen später in unserem Beispielprogramm

Ein noch damit verwandter Parameter ist *expand*, welcher macht nicht anderes als dem Kind-Element hinweisen, wie es sich verhalten soll, wenn das Hauptfenster sich in seiner Dimensionen verändert. Default Einstellung beim *expand* Parameter ist *false*, gemeint ist damit nämlich das, wenn Hauptfenster sich vergrößert oder "schrumpft", wird der Kind-Widget seine Größe und Position beibehalten. Wenn man anstatt *expand* als *true* setzt, werden Kinder-Elemente auch ihre Dimensionen entsprechend and die neue Größe des Hauptfensters anpassen. Üblich ist es, wenn man *fill* Parameter als "*both*" festgesetzt hat für einen bestimmten Kind-Widget, dann sollte man *expand* als "*true*" für diesen Kind-Widget festlegen.

Als eine Vorführung, wie der *pack* Layout Manager funktioniert könnte es hilfreich sein an einer konkreten Aufgabe hinzuschauen. Erinnern Sie sich, dass wir uns mit einem leeren rechtwinkligen "Hohlraum". Nun lassen wir uns anfangen und fügen oben an der Seite des Raums ein Kind-Element hinzu(Widget 1 s.h. Abb.1):



Nach diesem Schritt hat sich der erste Kind-Widget quasi die Seite oben beanspucht. Ungeachtet, wie wir weitere Kind-Widgets packen, das ist der einzige Widget, welcher am oberen Rand des Raums angegrenzt sein kann. Der unterste Rand des ersten Elementes wird quasi "oberster" Rand des übriggebliebenes Hohlräume. Als Nächstes fügen wir am linken Rand des Raums noch ein Kind-Widget 2(s.h.Abb.2)

Wieder einmal verringert sich übriggebliebenes Raum, diesmal um die Breite des zweiten Kind-Widgets. Der unterste Rand des ersten Widgets immer noch der oberste Rand des Raums, aber der rechter Rand des zweiten Kind-Widgets wird nun als neue linke Seite des verbleibenden Hohlraum. Jetzt fügen wir den dritten Kind-Widget 3(s.h.Abb.3) hinzu, diesmal am untersten Rand des Raums.

Nachdem wir diesen dritten Widget zugefügt haben, schrumpft das übriggebliebenes Raum um die Höhe des dritten Widgets und dessen oberste Rand wird nun der neue unterste Rand des übrig. Raums. Als Letztes fügen wir den letzten Widget 4(s.h.Abb.4) hinzu, diesmal am rechten Rand.

Ruby/Tk Beispielprogramm

Unten wird Quellcode gezeigt, welchen Sie sich auch herunterladen können [tk-xmlviewer](#)

```
#!/usr/bin/ruby -w
# Sample Application(Beispielprogramm) tk-xmlviewer.rb

require "tk"
require "nqxml/treeparser"

class XMLViewer < TkRoot
def createMenubar
  menubar = TkFrame.new(self)
  fileMenuButton = TkMenubutton.new(menubar,
    'text' => 'Datei',
    'underline' => 0)
  fileMenu = TkMenu.new(fileMenuButton, 'tearoff' => false)

  fileMenu.add('command',
    'label' => 'Öffnen',
    'command' => proc { openDocument },
    'underline' => 0,
    'accel' => 'Ctrl+O')
  self.bind('Control-o', proc { openDocument })

  fileMenu.add('command',
    'label' => 'Beenden',
    'command' => proc { exit },
    'underline' => 0,
    'accel' => 'Ctrl+Q')
  self.bind('Control-q', proc { exit })

  fileMenuButton.menu(fileMenu)
end
end
```

```
fileMenuButton.pack('side' => 'left')

helpMenuButton = TkMenubutton.new(menuBar,
    'text' => 'Hilfe',
    'underline' => 0)
helpMenu = TkMenu.new(helpMenubutton, 'tearoff' => false)

helpMenu.add('command',
    'label' => 'Info',
    'command' => proc { showAboutBox })

helpMenuButton.menu(helpMenu)
helpMenuButton.pack('side' => 'right')
menuBar.pack('side' => 'top', 'fill' => 'x')
end

def createContents
    # Lists
    listBox = TkListbox.new(self) {
        selectmode 'single'
        background 'white'
        font 'courier 10 normal'
    }
    scrollbar = TkScrollbar.new(self) {
        command proc { |*args|
            listBox.yview(*args)
        }
    }
    rightSide = TkFrame.new(self)
```

```
attributesForm = TkFrame.new(rightSide)
attributesForm.pack('side' => 'top', 'fill' => 'x')
TkFrame.new(rightSide).pack('side' => 'top', 'fill' => 'both',
    'expand' => true)
listBox.yscrollcommand(proc { |first, last|
    scrollbar.set(first, last)
})
listBox.bind('ButtonRelease-1') {
    itemIndex = listBox.curselection[0]
    if itemIndex
        # Remove currently displayed attributes
        TkGrid.slaves(attributesForm, nil).each { |slave|
            TkGrid.forget(attributesForm, slave)
        }

        #Add labels und entry widgets for this entity's attributes
        entity = @entities[itemIndex]
        if entity.kind_of?(NOXML::NamedAttributes)
            keys = entity.attrs.keys.sort
            keys.each_index { |row|
                TkLabel.new(attributesForm) {
                    text keys[row] + ":"
                    justify 'left'
                }.grid('row' => row, 'column' => 0, 'sticky' => 'nw')
                entry = TkEntry.new(attributesForm)
                entry.grid('row' => row, 'column' => 0, 'sticky' => 'e')
                entry.value = entity.attrs[keys[row]]
                TkGrid.rowconfigure(attributesForm, row, 'weight' => 1)
            }
            TkGrid.columnconfigure(attributesForm, 0, 'weight' => 1)
        end
    end
}
```

```
    TkGrid.columnconfigure(attributesForm, 1, 'weight' => 1)
  else
  end
end
}

listBox.pack('side' => 'left', 'fill' => 'both', 'expand' => true)
scrollBar.pack('side' => 'left', 'fill' => 'y')
rightSide.pack('side' => 'left', 'fill' => 'both', 'expand' => true)

@listBox = listBox
@attributesForm = attributesForm
end

def initialize(root)
  # Initialize base class
  super

  # Main Window Title
  title 'TkXMLViewer'
  geometry '600x400'

  #Menu bar
  createMenubar
  createContents
end

def populateList(docRootNode, indent)
  entity = docRootNode.entity
```

```
if entity.instance_of?(NOXML::Tag)
  @listBox.insert('end', "*indent + entity.to_s)
  @entities.push(entity)
  docRootNode.children.each do |node|
    populateList(node, indent + 2)
  end
elsif entity.instance_of?(NOXML::Text) &&
  entity.to_s.strip.length != 0
  @listBox.insert('end', "*indent + entity.to_s)
  @entities.push(entity)
end
end

def loadDocument(filename)
  @document = nil
  begin
    @document = NOXML::TreeParser.new(File.new(filename)).document
  rescue NOXML::ParserTree => ex
    Tk.messageBox('icon' => 'error', 'type' => 'ok',
      'title' => 'Fehler', 'parent' => self,
      'message' => "Das XML Dokument konnte nicht geparst werden")
  end
  if @document
    @listBox.delete(0, @listBox.size)
    @entities = []
    populateList(@document.rootNode, 0)
  end
end
```

```

def openDocument
  types = [["Alle Datei", "*"], ["XML Documents", "*.xml"]]
  filename = Tk.getOpenFile('filetypes' => types, 'parent' => self)
  if filename != ""
    loadDocument(filename)
  end
end

def showAboutBox
  Tk.messageBox('icon' => 'info', 'type' => 'ok',
    'title' => 'Über TkXMLViewer',
    'parent' => self,
    'message' => 'Ruby/Tk XML Viewer Application')
end

# Run Application( Ausführen des Programms)
root = XMLViewer.new
Tk.mainloop

```

Die ersten zwei Zeilen sind einfach zum Importieren erforderlichen Tk und NOXML Modulen. Danach wird Hauptprogramm-*XMLViewer* Klasse initialisiert, abgeleitet von *TkRoot*. Die Methode dieser Klasse *initialize* bekommt als ein Argument *root*

```

def initialize(root)
  # Initialize base class
  super

  # Main Window Title

```

```
title 'TkXMLViewer'  
geometry '600x400'  
  
#Menu bar  
createMenubar  
createContents  
end
```

Die erste Zeile dieser Methode ruft *super* auf, um die Base-Klasse zu initialisieren, das ist sehr wichtig, ohne das wird das Programm nicht zum Vorschein kommen. Die nächste zwei Zeilen rufen TkRoot's *title* und *geometry* Methode auf, beziehungsweise wird man der Titel des Hauptfensters und dessen anfängliche Breite und Höhe in pixel festgelegt. Diese zwei Methoden sind übrigens von Ruby/Tk *Wm* Module bereitgestellt, welche die Anzahl der Funktionen fürs Aufeinanderwirken mit "Window Manager" definieren.

Die letzte zwei Zeilen der *initialize* Methode rufen die andere *XMLViewer* Methode herbei. Damit werden Window's "Menüleiste" und *Contents* Raum dargestellt. Wir hätten wohl die Codes von diesen Methoden direkt in die *initialize* Methode einschließen können, aber das würde andere Teile der GUI Konstruktion sozusagen zerbrechen, also das ist viel besser und einfacher umfangreiche und schwierige Programme aufzubauen, wenn Codes in verschiedenen Methoden verteilt und so werden wir weiter dieser Logik folgen. Im Gegensatz zu einigen anderen Toolkits, wir werden es sehen, hat Ruby/Tk keine spezifische Klasse für die "Menüleiste"; stattdessen werden wir einfach einen *TkFrame-Widget* Behälter verwenden, ausgedehnt entlang der obersten Rande des Hauptfensters.

Die "pulldown" Menü von Ruby/Tk besteht aus dem *TkMenubutton* Objekt verbunden mit dem *TkMenu* Objekt. Der *TkMenubutton* Widget ist es, was man auf der Menü selbst sieht, der beinhaltet den Text, was der Name der Menü ist, wie z.B. *Datei*, *Bearbeiten* oder *Hilfe*. Wenn der Anwender auf den Button klickt, der damit verbundenen *TkMenu* Widget wird angezeigt. Sie können ohne weiteres eine odere mehrere Optionen zu *TkMenu* Widget hinzufügen, wobei wird die *add* Methode verwendet. Lassen wir uns an der Organisation unseres Beispielprogrammes ansehen, nämlich an der "Datei" Menü:

```
fileMenuButton = TkMenubutton.new(menubar,  
    'text' => 'Datei',  
    'underline' => 0)
```

Die Menü's Buttons werden als Kind-Widget der Menüleiste selbst erzeugt. Das *underline* Attribut von *TkMenubutton* Widget ist eine "integer"(ganze Zahl), welche auf eine Buchstabe hinweist, die im Titel des Menübuttons als unterstrichen bezeichnet wird. Die Hervorhebung einer Buchstabe in Menütitel ist ein allgemein-verwendeter optischer Hinweis in GUI Programms, um sozusagen eine "Beschleunigung-Lösung" festzulegen, die uns erlaubt die Menü zu aktivieren, z.Beispiel in meisten Windows Applicationen würde die Tastekombination **Alt+D** die "Datei" Menü aktivieren.

Als Nächstes erzeugen wir *TkMenu*Widget, verbunden mit dem *TkMenubutton* und fügen für diese Menü einen ersten Eintrag:

```
fileMenu = TkMenu.new(fileMenuButton, 'tearoff' => false)  
  
fileMenu.add('command',  
    'label' => 'Öffnen',  
    'command' => proc { openDocument },  
    'underline' => 0,  
    'accel' => 'Ctrl+O')  
self.bind('Control-o', proc { openDocument })
```

Beachten Sie dabei, dass *TkMenu* als Kind-Widget von Menübutton erzeugt wird. An dieser Stelle legen wir fest, dass die Menüstil nicht "abreißbar" ist. Der erste Eintrag, den wir zu "Datei"-Menü hinzufügen ist "command"-Eintrag für **Öffne** Befehl. Der erste Argument zu *add* Methode ist ein "string", der uns gerne verrät, um welche Type des Menüeintrages sich handelt; zusätzlich zum *command* Attribut gibt's noch einige Type, wie z.Beispiel für den "checkboxbutton" Eintrag ist das *check*, "radiobutton" Eintrag *radio*, "separator" *separator* und für "cascading sub-menu" *cascade*. Das *command* Attribut für diesen Menüeintrag ist ein Ruby *Proc* Objekt, welches eine andere XMLViewer "Instanz"-Methode aufruft, in

diesem Fall *openDocument*, welche wir uns noch ansehen werden. Das *accel* Attribut legt die Tastaturkombination, die uns eine Möglichkeit gibt, auf schnelle Weise eine Datei auszuwählen und sie zu öffnen. Der String dafür steht hinter dem Menüeintrag, was aber nicht automatisch für diese "Beschleunigungskombination" eingebunden wird, wir müssen dafür die *bind* Methode verwenden. Damit wird diese Tastaturkombination in unserem Hauptfenster bzw. Programm eingebunden.

Nun lassen wir uns, auf von uns definierte *createContents* Methode einen Blick werfen. Diese Methode baut einen Raum für einen Inhalt(Content) auf dem Hauptprogrammfenster auf, wobei er in zwei Teilen getrennt wird. Ein Teil auf der linken Seite ist für eine Aufstellung von XML Dokumentenknoten und auf der rechten Seite eine Aufstellung des Knotenattributes.

```
listBox = TkListbox.new(self) {  
    selectmode 'single'  
    background 'white'  
    font 'courier 10 normal'  
}  
scrollBar = TkScrollbar.new(self) {  
    command proc { |*args|  
        listBox.yview(*args)  
    }  
}  
listBox.yscrollcommand(proc { |first, last|  
    scrollBar.set(first, last)  
})
```

Tk's "list-box" Widget zeigt eine Liste von "strings", von denen wir uns etwas aussuchen können. Unsere *TkListbox* Instanz bestimmt das *selectmode* Attribut zu *single*, was darauf hinweist, dass nur einen Posten zur gleichen Zeit ausgewählt werden kann. Es gibt auch andere "selectionmode", die es mehrfach erlauben, zur gleichen Zeit Posten auszuwählen (unter Posten versteht man z. Beispiel einen Eintrag in XML-Datei). Wir setzen auch das Attribut *font* als "Courier mit der Größe 10" fest, anstatt eine Default

Schriftart zu verwenden, welche gewöhnlich die systemabhängige und proportional-raumige Schriftart ist. Da die Liste von Posten(Einträgen) sehr lang sein könnte(zu viel um auf einer Seite des Bildschirms zu passen), werden wir zu diesem "Listbox" einen *TkScrollbar* Widget hinzufügen. Der Codeblock oder eine Prozedure, woraus die "scrollbar's *command* Methode besteht, modifiziert die Ansicht des "listboxes", bzw. das Aufstellen von Posten, die in diesem Widget gezeigt werden. Die Anzahl von Parametern, die an "scrollbar" *command* Methode übergeben werden, variiert abhängig davon, wie oft der Anwender den Scrollbar betätigt hat. Zum Beispiel, wenn der Anwender die Position von "scrollbar" durch das Betätigen einer von zwei "Pfeiltasten" bestimmt, empfängt die *command* Methode drei Argumente(*scroll, 1, units*). Um mehr Information über verschiedene Argumente für den *TkScrollbar* Widget zu bekommen, sollten Sie die Tk Reference Dokumentation zur Rate ziehen. Im Allgemeinen braucht man überhaupt nicht, sich damit zu beschäftigen, wie man für die "command" Methode passende Argumente besorgt, man gibt sie einfach der *xview* oder *yview* Methode über. So ähnlich wird ein Codeblock oder eine Prozedur an die *yscrollcommand* Methode weitergeleitet, damit wird die Position und das Aufstellen von "scrollbar" , when sich "list contents" verändert, angepasst. Der nächste Abschnitt unseres Programmcodes legt die rechte Seite des Hauptfensters fest.

```
rightSide = TkFrame.new(self)  
attributesForm = TkFrame.new(rightSide)  
attributesForm.pack('side' => 'top', 'fill' => 'x')  
TkFrame.new(rightSide).pack('side' => 'top', 'fill' => 'both',  
    'expand' => true)
```

Im Moment ist es nicht besonders interresant, weil die "attributes" Formular noch leer ist. Aber der nächste Teil von Programmcode, welcher mit der Auswahl von Einträgen in XML-Datei handelt, macht diese Absicht ein wenig deutlicher:

```
listBox.bind('ButtonRelease-1') {  
    itemIndex = listBox.curselection[0]  
    if itemIndex
```

```

    # Remove currently displayed attributes
    TkGrid.slaves(attributesForm, nil).each { |slave|
      TkGrid.forget(attributesForm, slave)
    }

    #Add labels und entry widgets for this entity's attributes
    entity = @entities[itemIndex]
    if entity.kind_of?(NOXML::NamedAttributes)
      keys = entity.attrs.keys.sort
      keys.each_index { |row|
        TkLabel.new(attributesForm) {
          text keys[row] + ":"
          justify 'left'
        }.grid('row' => row, 'column' =>0, 'sticky'=> 'nw')
        entry = TkEntry.new(attributesForm)
        entry.grid('row' => row, 'column' => 0, 'sticky' => 'e')
        entry.value = entity.attrs[keys[row]]
        TkGrid.rowconfigure(attributesForm, row, 'weight' => 1)
      }
      TkGrid.columnconfigure(attributesForm, 0, 'weight' => 1)
      TkGrid.columnconfigure(attributesForm, 1, 'weight' => 1)
    else
      end
    end
  }
}

```

Dieser gesamte Codeblock ist mit [ButtonRelease](#) Ereignis festgebunden, wenn man mit linken Maustaste in Listbox auf eine von Einträgen klickt. Das ist das Ereignis, das erzeugt wird, wenn der Anwender einen von Listeeinträgen auswählt und darauf klickt und dann einfach den linken Mausbutton auf den ausgewählten Eintrag loslässt. Wir beginnen mit dem Aufruf der TkListbox's [curselection](#) Methode um in

einem Array Indexes von ausgewählten Einträgen zu speichern. Da es eine "singl-selection" Liste ist, erwarten wir nur einen ausgesuchten Eintrag und zwar als Element, das den Index "0" im Array hat. Als Nächstes löschen wir aktuellen *attributes* Formularinhalt, erreichen wir es, wenn man *TkGrid.slaves* aufruft, die in der Form eines Arrays die Kind-Widgets von Formularen erhält, und dann wenn man *TkGrid.forget* aufruft, wird man mit der Hilfe *each* Methode durchgeforstet und gelöscht. Während die meisten GUI Toolkits die "parent-child" Terminologie verwendet, um auf den hierarchischen Aufbau von GUI Containers zu verweisen, verwendet Tk Häufig Termine "master" und "slave", vor allem when man so einen Layout Manager als *TkGrid* einsetzt.

Der nächste Abschnitt dieses "Event-Handler" "schleift" über alle Attributes des aktuell ausgewählten Dokumentenknotens und fügt für jeden Knoten einen *TkLabel* und *TkEntry* Widgets hinzu. Merken Sie es, dass wir die Default-Einstellung für Ausrichtung von *TkLabel* und *TkEntry* Widgets nicht berücksichtigen. Gültige Werte für das *justify* Attribut sind *left*, *center* und *right*, aber die Grundeinstellung der "Label's" Ausrichtung ist *center*. Also weil wir gern sehen würden, dass "grid" Manager Spalten und Zeilen gleich behandeln wird, rufen wir *TkGrid.rowconfigure* und *TkGrid.columnconfigure* Module-Methode auf, um das *weight* Attribut für jede Spalte und Zeile auf "1" zu setzen.

Nun sehen wir uns noch weitere sogenannte "lower-level" Methoden für unsere *XMLViewer* Klasse. Für den Anfang haben wir *openDocument* Methode, welche ausgeführt wird, sobald der Anwender den **Offnen** Eintrag in der **Datei** Menü auswählt oder "Shortkey" **Ctrl+O** verwendet:

```
def openDocument
  types = [{"Alle Datei", "*"}, {"XML Documents", "*.xml"}]
  filename = Tk.getOpenFile('filetypes' => types, 'parent' => self)
  if filename != ""
    loadDocument(filename)
  end
end
```

Der wichtigste Teil dieser Funktion ist das Aufrufen von *Tk.getOpenFile* Widget und selbstdefinierter

`loadDocument` Methode. `Tk.getOpenFile` ist eine `Tk` Module-Methode, die man ermöglicht, das systemspezifische Datei-Dialogbox anzuzeigen, wo man sich existierende Dateinamen aussuchen kann. Ähnliche Funktion hat `Tk.getSaveFile` Modul-Methode, die man verwenden kann, wenn man eine von existierenden oder neuen Datei speichern will. Das `filetypes` Attribut setzt eine Liste von Dateitypen als eine Schilderung und ein Muster dafür fest. Das nächste Attribut `parent` legt den Besitzer `owner` des Fensters für den `Datei` Dialog fest. Angenommen, dass der Anwender nicht diesen Dialog beendet und versorgt sich mit einem legitimen Dateinamen, dann wird unsere selbstdefinierte `loadDocument` Methode aufgerufen um die aktuelle, ausgesuchte XML-Datei auszulesen:

```
def loadDocument(filename)
  @document = nil
  begin
    @document = NQXML::TreeParser.new(File.new(filename)).document
  rescue NQXML::ParserTree => ex
    Tk.messageBox('icon' => 'error', 'type' => 'ok',
      'title' => 'Fehler', 'parent' => self,
      'message' => "Das XML Dokument konnte nicht geparkt werden")
  end
  if @document
    @listBox.delete(0, @listBox.size)
    @entities = []
    populateList(@document.rootNode, 0)
  end
end<
```

Wenn der XML Parser eine Ausnahmesituation meldet während des Auslesens von XML-Datei, kann man sich ein einfaches Dialogbox anzeigen lassen, dargestellt mit der Hilfe von der `Tk.messageBox` Module-Methode. Das `icon` Attribut kann ein von vier "strings" aufnehmen und zwar `error`, `info`, `question`, `warning`, um einen optischen Effekt zu liefern. Das `type` Attribut weist darauf hin, welcher der "Ende-Button" verwendet wird. Für unseren einfachen Fall werden wir einfach den Button als "OK" anzeigen

lassen und beim Drücken auf den Button wird dieser kleine Dialog beendet. Es gibt natürlich auch andere Optionen fürs *type* Attribut. Das sind *abortretrycancel*, *okcancel*, *retrycancel*, *yesno* und *yesnocancel*.

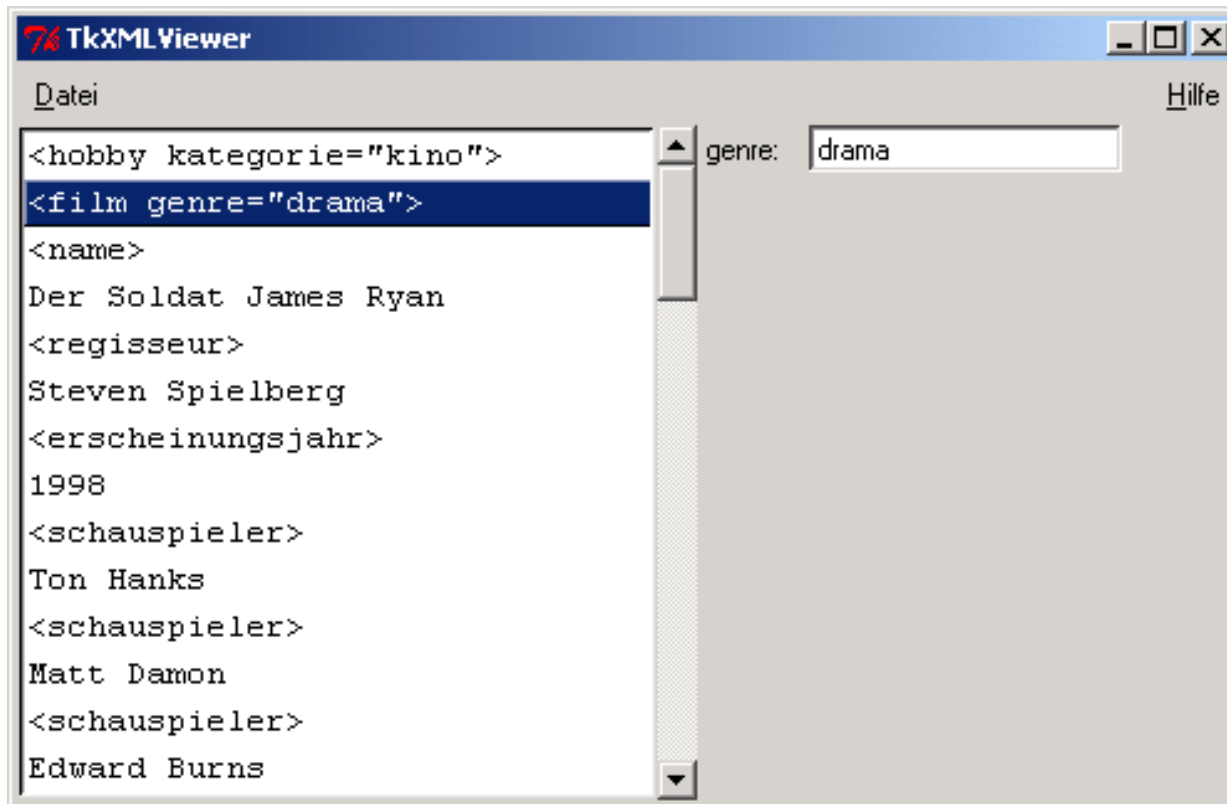
Angenommen, es gab keine Fehler beim Auslesen von XML-Document, die Instanz-Variable *@document* eine Referenz(einen Verweis) auf ein *NQXML::Document* Objekt besitzen. Wir rufen die *delete* Methode auf, um die aktuelle Liste zu löschen und dann wird die Methode *populateList* gestartet, welche die Liste mit dem neuen Documentinhalt ausfüllt:

```
def populateList(docRootNode, indent)
  entity = docRootNode.entity
  if entity.instance_of?(NQXML::Tag)
    @listBox.insert('end', "*indent + entity.to_s)
    @entities.push(entity)
    docRootNode.children.each do |node|
      populateList(node, indent + 2)
    end
  elsif entity.instance_of?(NQXML::Text) &&
    entity.to_s.strip.length != 0
    @listBox.insert('end', "*indent + entity.to_s)
    @entities.push(entity)
  end
end
```

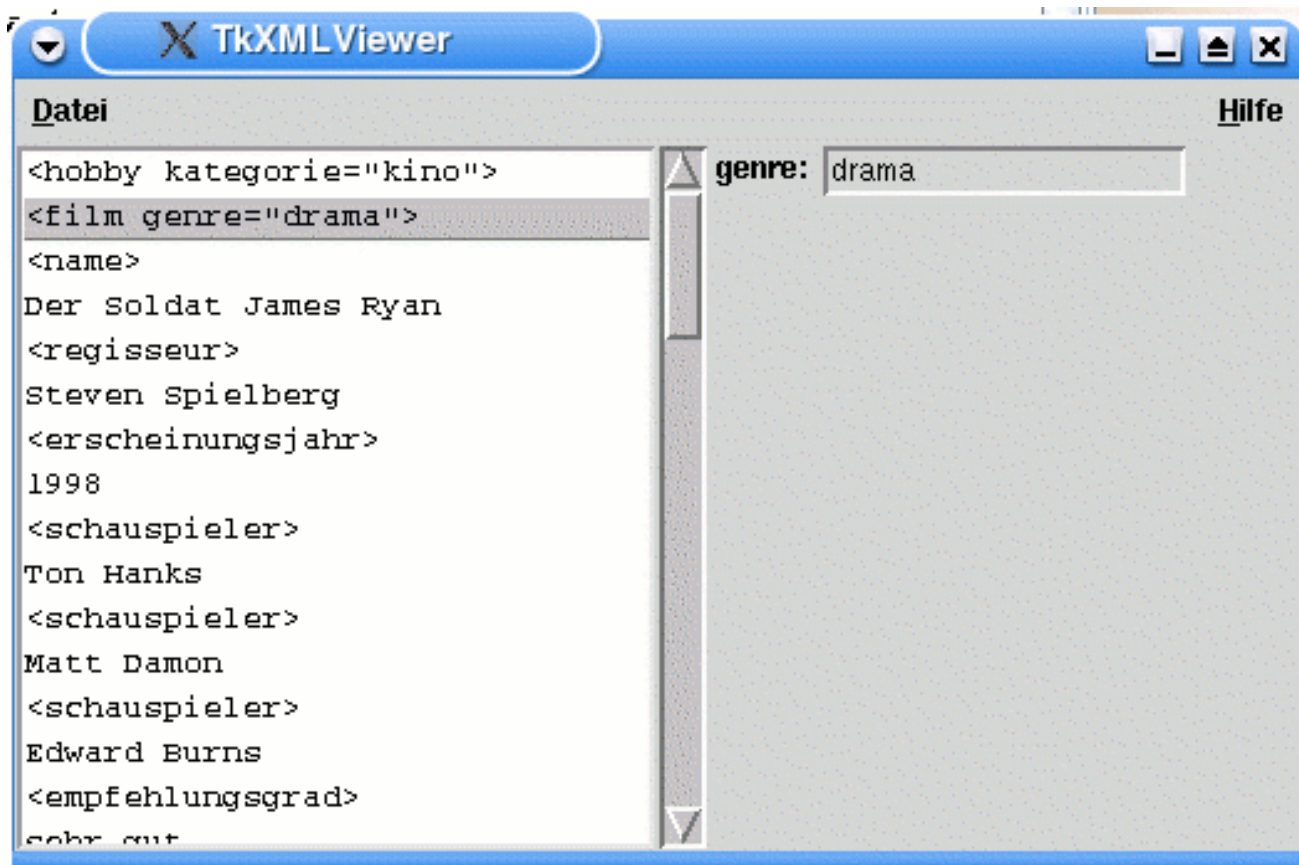
Andere GUI Toolkits, wir werden später es sehen, haben eine baumähnliche Struktur für die Darstellung von XML-Dateien. Tk hat aber solche Möglichkeit nicht, es gibt allerdings eine TK's Erweiterung - "Tix". Mehr Information über Tix erhalten Sie im Abschnitt "Tk's Erweiterung: Tix und BLT". Da Tix nicht immer verfügbar ist, werden wir die nur ungefähr behandeln, aber wenn Sie sich keine Gelegenheit "vermissen" möchten, wie man den baumstrukturigen Widget in einer regulären *TkListbox* mit der Einrücken von Listeeinträgen verwendet, sollten Sie sich diese Erweiterung näher ansehen. Die *populateList* Methode wird solange aufgerufen, bis das ganze Document dargestellt wird und anschließend sehen wir, wie die

Applikation die Auswahl der Liste von Einträgen behandelt.

Und so sieht unsere fertige Ruby/Tk Applikation aus(das Programm wurde unter Windows XP getestet):

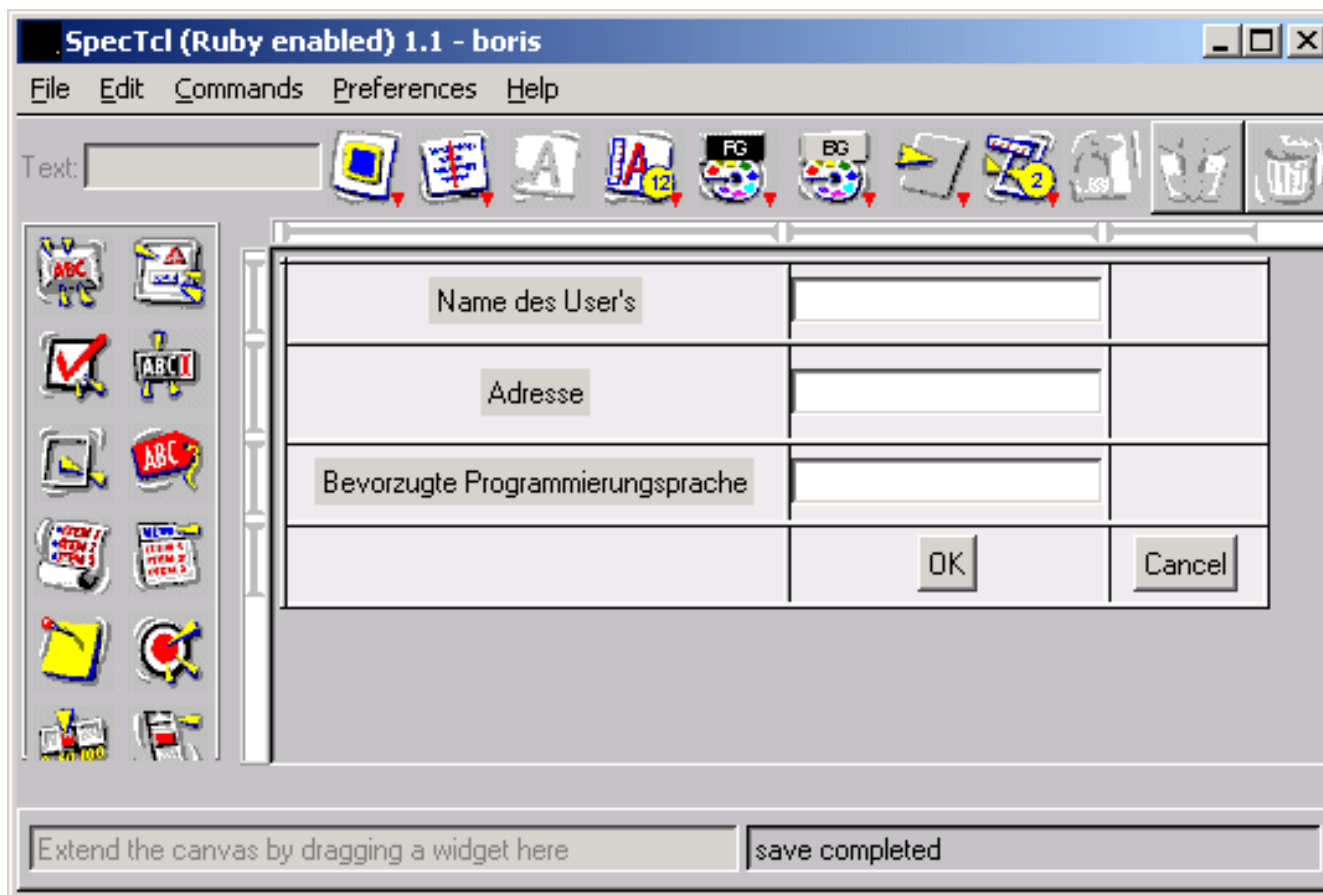


Für Linuxbegeisterten, zu denen ich mich auch zähle, würde es so aussehen:

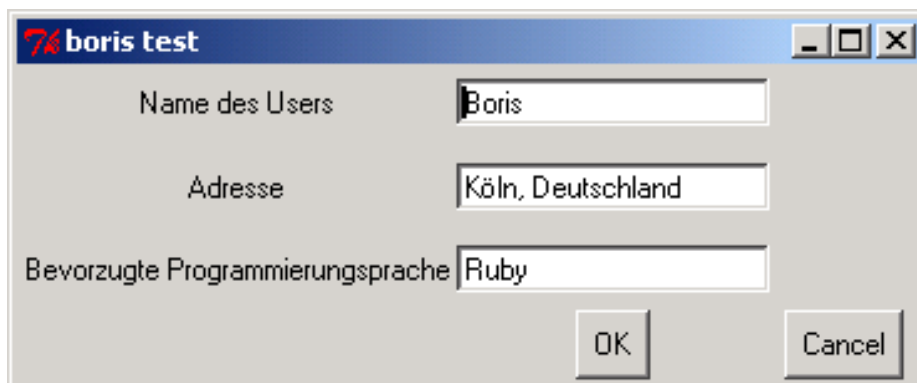


Anwenden SpecTcl GUI Builder

GUI-Building Tool für Tk [SpecTcl](#) war ursprünglich von "Sun Microsystems" entwickelt worden. Seit schon geraumiger Zeit wird dieses Projekt von einer Gruppe Freiwilligen unter Leitung von Morten Jensen weiterentwickelt und gewartet. Obwohl die ursprüngliche Absicht von "SpecTcl" war, die auf "user interface" basierende Tcl Sourcecodes zu erzeugen, haben Entwickler seitdem eine Generation kompatiblen Codes für andere Skriptsprachen entwickelt (Perl, Python, und Ruby), welche Tk als GUI verwenden. Eine experimentale Version von Ruby "backend" für "SpecTcl", genannt als [specRuby](#) wurde von Conrad Schneiker entwickelt und gewartet von Jonathan Conway. Eine aktuelle Version kann man von [RAA-specRuby](#) herunterladen. Folgende Abbildungen zeigen das Programm selbst mit einem einfachen Tk "user interface":



und als ein Ergebnis, wenn man das testet:



Zur Zeit gibt's keine Homepage von "specRuby", aber wie gesagt das Projekt kann man in "RAA" finden.

Um Ruby/Tk zu verwenden, muss selbstverständlich eine funktionierende Tcl/Tk Installation auf dem System vorhanden sein.

Verwendung der Tk Erweiterung: Tix und BLT

Zusätzlich zu Tk's sozusagen "Kern-Widgets", gibt's aus viele Widgets von "dritter Hand", die mit Tk kompatibel sind. Während die Beschaffung von dieser Tk Erweiterung jemandem zu schaffen macht, ist aber das Resultat dieser Anstrengung eher verblüffend, als man nur Tk allein einsetzt. Einer velleicht der bekanntsten Tk-compatiblen Erweiterungen ist [Tix](http://tix.sourceforge.net)- "Tk Interface Extension" genannt (<http://tix.sourceforge.net>) Tix bietet einen hierarchischen List-Widget (baumstrukturierte Liste), einen "notebook-widget" und auch "combo-box-widget" sowie andere widgets an. Sie können immer den Source-Code von Tix Homepage herunterladen, aber um diese Erweiterung zu verwenden, müssen Sie Tcl/Tk Source-Code herunterladen und selbst kompilieren.

Eine andere bekannte Erweiterung für Tk ist BLT(www.tcltk.com/blt) Als Tix bringt BLT eine Vielfalt von neuen Möglichkeiten, meistens für die Erstellung von Diagrammen und Grafiken. Sie können den Source-Code oder schon vorkompilierte Binaries für Windows von der BLT Homepage herunterladen, aber im Gegensatz zu Tix ist die Erstellungsprozedure ganz normal, wie man sie aus der Unix-Welt sicher kennt.: **configure**, **make** und **make install**. Um Tix und BLT mit Ruby/Tk zu verwenden müssen Sie ein TclTk-ext Package von Hidetoshi Nagai. Das können Sie von RAA downloaden, aber alle Dokumentation dazu wird in Japanisch geschrieben.

Verwendung GTK+ Toolkit

GTK+ ist ein cross-platform GUI Toolkit, der ursprünglich entwickelt worden war, um "GNU Image Manipulation Programm(GIMP) zu unterstützen. Wie schon der Name sagt, ist GIMP ein Bildbearbeitungsprogramm(www.gimp.org), das sich mit vielen kommerziellen Programmen messen lässt. Obwohl GTK+ hauptsächlich für X Window System entworfen wurde, ist es auch auf Microsoft Windows portiert worden. Als ein Teil des GNU Projektes wird GTK+ für viele populäre freie Software verwendet und er ist sozusagen ein Kernkomponent des GNU Projektes "GNU Network Object Model Environment(GNOME)-Desktop Umgebung"

Ruby/GTK ist eine Ruby's Erweiterung-Module, die in "C" geschrieben wurde. Sie bietet eine Schnittstelle von Ruby zu GTK+ an. Diese Erweiterung war ursprünglich von Yukihiro Matsumoto(der Author Ruby) und zur Zeit wird das Projekt von Hiroshi Igarashi gewartet.

Installieren Ruby/GTK

Die Homepage von Ruby/GTK ist www.ruby-lang.org/gtk. Wenn Sie schon ein Standard Ruby Installation für Windows von der "Pragmatic Programmers' site" haben, ist es wahrscheinlich, dass man auch vorkompilierte Binäres für Ruby/GTK hat. Für andere Plattformen(Unix und Linux) muss man Ruby/GTK von Sourcecoden selbst kompilieren

Um Ruby/GTK zu kompilieren muss man unbedingt schon eine GTK+ Installation haben. Viele Linux-Distributions beinhalten schon eine GTK+ Development Package als eine Installationsoption; Red Hat z.B. hat eine [gtk+-devel](#) Package. Wenn ihre Distribution schon keine GTK+ Installation hat, haben Sie eine Möglichkeit, sich auf GTK+ Homepage(www.gtk.org) reichlich mit der Information, was und wie heruntergeladen und installiert wird, zu bedienen. Dann müssen Sie sich auch Ruby/GTK Sourcecode besorgen und wenn man schon alles hat, kann man mit der Installation anfangen.

Erst wenn man funktionfähige GTK+ Installation hat, muss man mit Ruby/GTK Package beginnen. Zur Zeit ist die aktuelle Version von "ruby-gtk" 0.34. Auf der entsprechenden Seite gibt's auch die komplette "ruby-gnome" Source-Code. Ich rate Ihnen, die komplette Package herunterzuladen und zu installieren.

Nachdem Sie alle Vorbereitungen getroffen haben, entpacken Sie Package, die als tar.gz gepackt ist. Mit dem schon bekanntem Befehl `tar xzfv ruby-gtk-0.34.tar.gz` entpacken Sie die Package und danach wechseln Sie in das enpackte Verzeichnis und führen Sie den folgenden Befehle `ruby extconf.rb` aus. Damit wird automatisch Makefile erzeugt. Daraufhin geben Sie an der Konsole `make` ein, um der Code zu kompilieren. Anschließend führen Sie als root `make install` aus.

Ruby/GTK Basis

Obschon die GTK+ in "C" geschrieben wurde, ist der Aufbau der Bibliothek object-orientierend und deswegen spiegelt Ruby/GTK derer "Klass-Hierarchie" wieder. Wenn Sie bereits mit "GTK+ C" API (Application Programm Interface) vertraut sind und auch schon solchen Widgets Namen wie GtkLabel, GtkButton, etc., begegnet sind, dann ein Übergang zur Programmierung mit Ruby/GTK fällt Ihnen bestimmt leicht. GTK+ Widgets werden in der Form `GtkWidgetName` benannt und sind in Ruby/GTK als `Gtk::WidgetName` zu definieren. Das heisst, dass der Ruby Modulename `Gtk` ist, und der Name der Widget Klasse ist `WidgetName`. Und so ähnlicherweise sind Ruby/GTK Instanzmethoden den entsprechenden "C" Funktionnamen ähnlich, z.B. "C" Funktion `gtk_label_set_text()` wird in Ruby/GTK `Gtk::Label` mit der Instanzmethode `set_text`

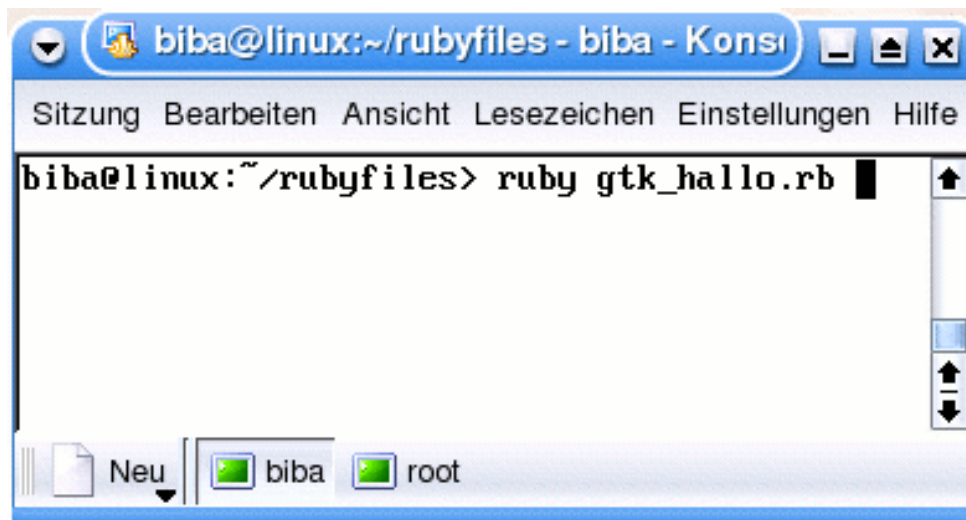
Wir werden nun ein kleines Ruby/GTK Programm erstellen, das aus dem Hauptfenster und einem oder mehreren Kind Widgets besteht, dann setzen wir sogenannte "signal handler" ein, und wenn das Programm starten, wird die Steuerung dem GTK+ "main event loop" übergeben. Ohne weitere Umstände präsentieren wir eine Ruby/GTK Version von "Hello World":

```
require 'gtk'

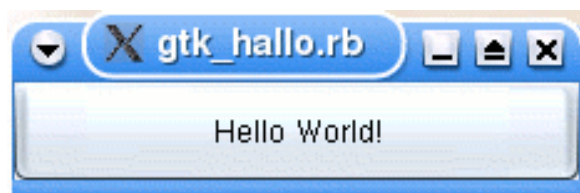
window = Gtk::Window::new
button = Gtk::Button::new("Hello World!")
```

```
button.signal_connect(Gtk::Button::SIGNAL_CLICKED) {  
  puts "Goodbye, World!"  
  exit  
}  
window.add(button)  
button.show  
window.show  
Gtk::main
```

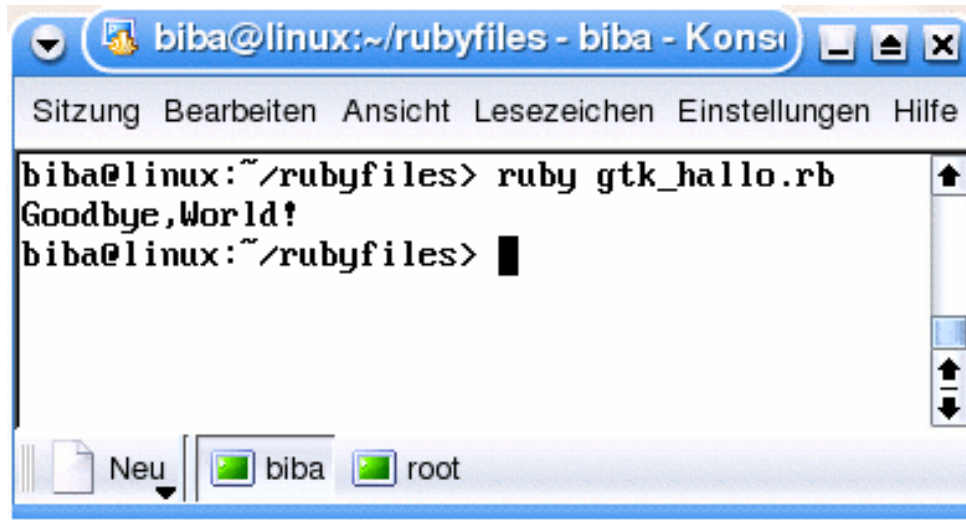
Nun kann man unseres Programm unter einem Namen speichern z.B. `gtk_hallo.rb` und danach laufen lassen, tippen Sie auf der Konsole: `ruby gtk_hallo.rb` ein:



Als Ergebnis bekommen wir unseres kleines Ruby/GTK Programm:



Und wenn wir anschließend den Button betätigen, würd es so aussehen:

A screenshot of a terminal window titled "biba@linux:~/rubyfiles - biba - Konsol". The window has a menu bar with "Sitzung", "Bearbeiten", "Ansicht", "Lesezeichen", "Einstellungen", and "Hilfe". The terminal content shows the command "ruby gtk_hallo.rb" being executed, which outputs "Goodbye, World!". The prompt "biba@linux:~/rubyfiles>" is visible again. At the bottom, there is a taskbar with a "Neu" button and two window icons labeled "biba" and "root".

```
biba@linux:~/rubyfiles> ruby gtk_hallo.rb
Goodbye, World!
biba@linux:~/rubyfiles> █
```

Das Programm beginnt mit der Einbindung der erforderlichen Ruby/GTK Bibliothek: [gtk](#). Als Nächstes erstellen wir zwei neue Widgets: GtkWindow Widget, welcher standardmäßig als "top-level" Hauptfenster gestartet wird und zweiter Widget ist GtkButton Widget mit der Beschriftung "Goodbye World!". Wie Sie vielleicht schon gemerkt haben, bis dahin gab es keine Verbindung zwischen diesen beiden Widgets. Mit Ruby/Gtk bilden Sie ihre ganze "User Inteface" mit wenigen Schritten am Ende, wenn man Kinder's Widgets zu Vater's Widget hinzufügt.

Die nächste Zeile zeigt uns eine Art von Ruby/GTK "event handler". Der Code innerhalb des Codeblockes wird nicht sofort ausgeführt, stattdessen verbindet Ruby/GTK dieser Codeblock mit dem Signal vom Button [Gtk::Button::SIGNAL_CLICKED](#), welches später generiert wird, sobald das Programm läft. Im nächsten Abschnitt werden wir ausführlicher darauf eingehen, aber jetzt lassen wir uns einfach auf den Button klicken, das Programm antwortet uns damit, dass auf der Konsole "Goodbye World!" erscheint, und gleichdanach beendet sich Programm selbst.

Die nächste zwei Zeilen(button.show und window.show) sind entscheidend, die sind etwas Einzigartiges in Ruby/GTK, was man in andreen Toolkit's nicht trifft. Standardmäßig sind neue erstellte Widgets nicht zu sehen, man muss danach explizit mit der Methode [show](#) darauf hinweisen, dass die Widgets sichtbar werden. Dieser Schritt wird häufig bei neuen Ruby/GTK Programmieren vergessen. Und die letzte Zeile

des Programmes ruft GTK+ "main event loop"(Programmschleife). Ab diesem Zeitpunkt wartet das GTK+ Programm auf ihren Input, passt sozusagen besonderes auf Signale(wie "Buttonclick") auf, für welche man schon "signal handler" Methode definiert wurde. Nun aber lassen wir uns mehr detailliert darauf einen Blick werfen und sehen wir uns, wie das in "wirklichen" Programm funktioniert.

Programmsignale und Signal Handlers

Ruby/GTK Ereignis Model basiert auf die Idee, dass man etwas geschieht(gemeint wird damit z.B. User's Handlungen), z.B. geben "user interface objects(Widgets) *signale* ab. Viele von diesen Signale sind "low-level" Ereignisse und werden von Betriebssystem generiert und zeigen generelle Information solche wie "die Maus wird bewegt" oder "die linke Maustaste wird geklickt". Andere Signale werden von GTK+ selbst erzeugt und bieten eher bestimmte Information an, so wie z.B. "ein Listeneintrag wird gewählt" Ein Widget kann eine beliebige Anzahl von Signalen aussenden und jedes Signal hat in Ruby/GTK einen Namen, welcher auf seine Bedeutung hinweist. Zum Beispiel *GtkButton* Widget sendet ein "clicked" Signal aus, when der Button geklickt wird. Da GTK+ ein objekt-orientierender Toolkit ist, kann ein bestimmter Widget nicht nur seine widget-spezifische Signale abgeben, sondern auch diejenige, die seinem "Vorfahr-Klasse" gehören.

Um ein Signal von einem Widget mit irgendeiner Aktion zu verbinden, kann man eine Widget's *signal_connect* Methode aufrufen. Diese Methode nimmt ein String-Argument ein, der auf den Signalname hinweist(wie in unserem Beispiel "clicked") und der Coderblock wird in Zusammenhang des Aufrufers ausgewertet. Zum Beispiel, wenn ihre Ruby/GTK-basierendes Tabellenkalkulationsprogramm ein Inhalt in der schon vorhandenen Tabellendatei speichern möchte, jedesmal wird der **Speichern** Button geklickt,dann kann man solche Zeilen im Programm einschließen:

```
saveButton.signal_connect('clicked') {  
  saveSpreadsheetContents if contentsModified?  
}
```

Jede Klasse definiert eine symbolische Konstante für den Namen des Signales, das die abgeben kann und

diese Konstante kann anstatt von "literal string"(in unserem Fall "clicked") verwendet werden. Zum Beispiel im Falle des Codes, den wir oben geschrieben haben, würde es so aussehen:

```
saveButton.signal_connect(Gtk::Button::SIGNAL_CLICKED) {  
  saveSpreadsheetContents if contentsModified?  
}
```

Der Vorteil der Verwendung der symbolischen Konstante ist es, wenn Sie einen Schreibfehler machen, werden Sie wahrscheinlich viel schneller den Fehler entdecken, sobald Sie das Programm laufen lassen. Wenn Sie versuchen eine Konstante zu referenzieren und haben Sie dabei nicht richtig den Namen der Konstante geschrieben, beschwert sich Ruby und das Programm wird mit der Ausnahmefehler *NameError* beendet. Wenn Sie den literalen String verwenden, hat Ruby keine Möglichkeit zu überprüfen, ob das gültige Signalname war, bevor man den die Methode *signal_connect* weitergegeben wird.

Als Programmier ist es entscheidend, welche Widget's Signal vom besonderen Interesse sind und wie ihres Programm würde reagieren, when solche Signale ausgesendet werden. Ebenfalls achten Sie darauf, wenn es einen Sinn für ihres Programm macht, können Sie das gleiche Signal(vom gleichen Widget) mit mehreren Codeblöcken. In diesem Fall die Signal Handler werden ausgeführt und zwar in der Reihenfolge, in der sie anfänglich verbunden waren. Wenn wir mit unserem Beispielprogramm anfangen, sehen wir ein paar Beispiele, wie die Signale mit Codeblöcken verbunden werden können. Für die vollständige Auflistung von Signalen, die von verschiedenen Ruby/GTK Widgets ausgesendet werden, schlagen Sie bitte die Online-Dokumentation auf der Ruby/GTK Homepage(www.ruby-lang.org/gtk) nach.

Arbeitsweise mit Ruby/GTK Layout Manager

Genauso wie Ruby/Tk bietet Ruby/GTK eine Auswahl an flexiblen Layout Managern an. Jeder von drei Layout Managern, wir werden das betrachten, ist ein Behälter(Container)-Widget, zu dem Sie ein oder mehrere Kinder-Widgets hinzufügen können. Der Container selbst ist für alle praktische Zwecke unsichtbar. Die erste zwei Layout Manager *horizontal packing box*(`Gtk::HBox`) und *vertical packing box* (`Gtk::VBox`) ordnen ihrer Kinder-Widgets in Zeilen bzw. Spalten an. Der dritte Layout Manager "Gtk::

Table" ordnet seine Kinder-Widgets in der tabellarischen Format an, genauso wie Ruby/Tk's "grid layout.

Der "horisontal packing box" Layout Manager(Gtk::*HBox*) ordnet seine Kinder-Widgets, wie der Name schon sagt, horisontal. Alle Kinder haben die gleiche Höhe, aber ihre Breite können sich entsprechend Parametren von "packing box" anpassen. Im Gegensatz zu Gtk::*HBox* ordnet Gtk::*VBox* seine Kinder-Widgets vertical an, und sie alle haben die gleiche Breite. Da diese zwei Layout Manager gleichartig sind, konzentrieren wir uns auf Gtk::*HBox*. Die *new* Methode für den Gtk::*HBox* nimmt zwei Argumenten ein:

```
hbox = Gtk::HBox.new(homogeneous=false, spacing=0)
```

Der erste Argument ist ein Booleanischer Wert, der darauf hinweist, ob die Kinder Widget's Größen *homogeneous*(gleichartig) oder nicht sind. When dieser Argument "true" ist, heisst es einfach, dass die "packing" Box ihre Breite zwischen ihrer Kinder-Widgets teilen wird; when der Wert "false"(nicht gleichartig) ist, heisst es, dass jeder Kind-Widget so viel Platz beansprucht, so viel er braucht, nicht mehr. Und der zweite Argument ist ein in pixel gemessener Raum, der zwischen einzelnen Kinder-Widgets ist.

Man kann ein Kind-Widget zu der "packing" Box hinzufügen, wenn man ein von beiden *pack_start* oder *pack_end* Instanz-Methoden verwendet. Sie können sich bestimmt erinnern, dass wir in Ruby/Tk den Eltern-Widget als erster Argument zu der Methode *new* übergeben haben, um einen neuen Kind-Widget zu erstellen. Ruby/Gtk ergreift dabei eine andere Methode: Kinder-Widget werden zuerst erzeugt und dann zum Behälter-Widget hinzugefügt oder gepackt. Um einen Kind-Widget zu Gtk::*HBox* oder Gtk::*VBox* zu packen, können *pack_start* Methode aufrufen.

```
hBox.pack_start(kind, expand=true, fill=true, padding=0)
```

Der erste Argument von der *pack_start* Methode ist einfach ein Verweis auf den Kind-Widget, den zur "packing" Box hinzufügen möchten, weitere drei Argumente erfordern aber eine ausführliche Erklärung. Der zweite Argument *expand* ist quasi eine Anweisung zur "packing" Box, dass der Kind-Widget ausdehnen kann, wenn das möglich wäre(z.B. User ändert das Fenster, um das zu größer zu machen).

Wir haben schon erläutert, worin sich "homogeneous"(gleichartige) und "non-homogeneous"(nicht-gleichartige) "packing" Boxes unterscheiden; "homogeneous" "packing" Box teilt sein Raum gleichmäßig zwischen den Kinder-Widgets. Eine andere Weise, das sich zu vorzustellen, ist es, dass jeder Kind-Widget sich so viel Platz nimmt, so viel sich der breiteste Widget genommen hat. Die Nebenwirkung dieser Einstellung ist es, dass die Kinder-Widgets sich genau in der Mitte des für sie festgelegten Raumes plazieren würden. Wenn der Argument *fill* den Wert "true" bekommt, heisst es, dass der Kind-Widget sich in seinem festgelegten Raum wachsen kann. Der letzter Argument der Methode *pack_start* weist einfach auf die "Füllung"(in pixel), sozusagen den Abstand rund um den Kind-Widget bis zu dem Rand des festgelegten Raum, der Argument ist ein Zusatz zu dem schon erwähnten Argument *spacing* von der Methode "new". Wennn der Kind-Widget entweder erster oder letzter Widget in der "packing" Box ist, ist der Wert von diesem Argument ein Abstand vom Rand bis zu diesem Kind-Widget.

Ruby/GTK bietet auch `Gtk::Table` Layout Manager an. Dieser Manager ordnet die Kinder-Widgets in Zeilen und Spalten an. Die *new* Methode von diesem Layout Manager bekommt drei Argumente:

```
table = Gtk::Table.new(numRows, numColumns, homogeneous=false)
```

Die erste zwei Argumente sind die Anzahl von Zeilen und Spalten in der Tabelle und der "homogeneous" haben wir schon vorhin erklärt. Um den Abstand zwischen Zeilen und Spalten festzulegen, verwendet man eine von der Instanz-Methoden:

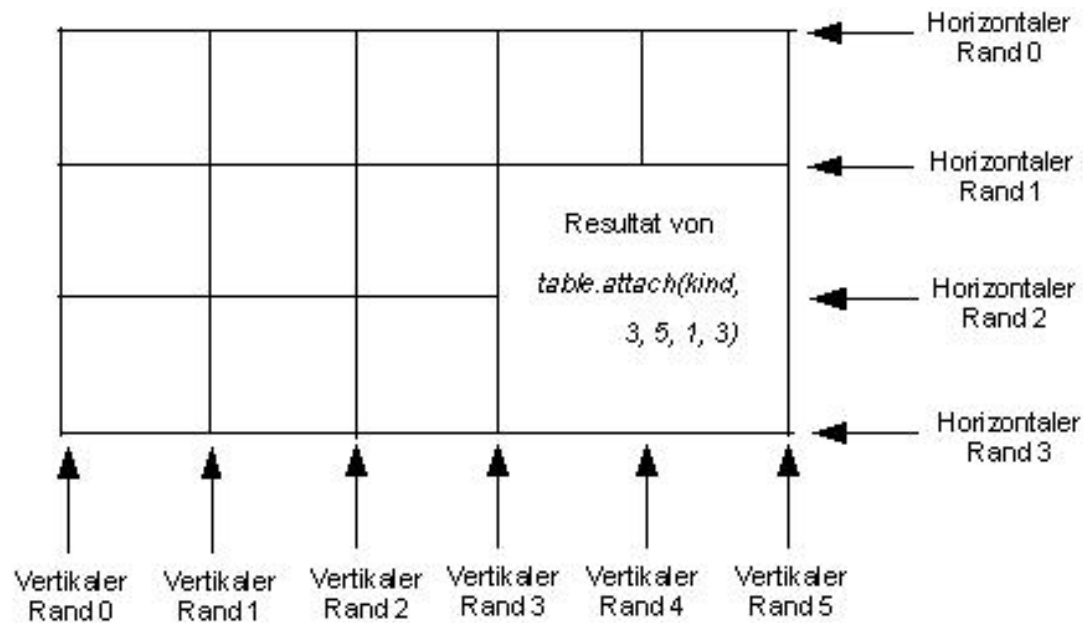
```
table.set_row_spacing(row, spacing)
table.set_row_spacings(spacing)
table.set_column_spacing(column, spacing)
table.set_column_spacings(spacing)
```

Die *set_row_spacings* und *set_column_spacings* Methoden setzen den globalen Wert für den Abstand(in pixel) fest, welcher für alle Spalten und Zeilen in der Tabelle gilt. Aber wenn Sie doch mehr expermentieren und selbst die Anordnung definieren möchten, können Sie *set_row_spacing* oder *set_column_spacing* verwenden. Damit kann man den Raum, der sich unter einer bestimmten Zeile oder

rechts von einer bestimmten Spalte befindet. Um diesen Wert festzusetzen, ruft man [set_row_spacing](#) und [set_column_spacing](#) Methoden auf, die den globalen Wert überschreiben. Also wenn man ein Kind-Widget zu `Gtk::Table` hinzufügen möchte, verwendet man eine [attach](#) Methode:

```
table.attach(kind, left, right, top, bottom,  
            xopt=GTK_EXPAND|GTK_FILL, yopts=GTK_EXPAND|GTK_FILL,  
            xpad=0, ypad=0)
```

Das erscheint viel komplexer, als die Argumentenliste von [pack_start](#) oder [pack_end](#) Methoden, man sollte sich merken, dass die letzte vier Argumenten Default-Werten haben. Der erste Argument referenziert den Kind-Widget, der hinzugefügt wird und die nächste vier Argumente sind "integer"(ganze Zahlen), die darauf hinweisen, wo der Kind-Widget in der Tabelle plaziert wird und wieviel Zeilen und Spalten der umfasst. Um besser zu verstehen wie diese Argumente verwendet werden, stellt sich ein Bündel von Linien vor, die eine Tabelle mit ihrer Zellen bilden, anstatt die Tabellenzellen selbst. Zum Beispiel betrachten wir eine Tabelle mit 5 Spalten und 3 Zeilen(s.h Abbildung unten). Um das zu zeichnen müssen Sie 6 vertikale Linien(eine links und jede rechte Seite von 5 Spalten) und auch 4 horizontale Linien(eine Linie oben und jede weitere von 3 Zeilen).



Mit dieser Abbildung wird gemeint, dass die Bedeutung von *left*, *right*, *top* und *bottom* Argumenten für `Gtk::Table.new` Folgendes sind:

- **Left** zeigt, welche vertikale Linie der Tabelle der linke Rand vom Kind-Widget ist
- **Right** zeigt, welche vertikale Linie der Tabelle der rechte Rand vom Kind-Widget ist
- **Top** zeigt, welche horizontale Linie der Tabelle der oberste Rand vom Kind-Widget ist
- **Bottom** zeigt, welche horizontale Linie der Tabelle der unterste Rand vom Kind-Widget ist

Für die Widgets, die lediglich eine Tabellenzelle belegen, ist der Wert für *right* immer um eins mehr als der Wert für *left* und der Wert für *bottom* um eins mehr als für *top*. Aber für Widgets, die vier oder fünf Spalten und drei oder vier Zeilen der Tabelle, dann könnten Sie wie jetzt vorgehen:

```
table.attach(kind, 3, 5, 1, 3)
```

Wenn die grafische Darstellung nicht korrekt aussieht, überprüfen Sie die Werte, die Sie an die Methode *attach* übergeben haben. Ruby/GTK wird sich beschweren, wenn die Argumente, die man beifügt um eine Zelle zu erzeugen, die Null Breite oder Höhe hat (wenn *left* ist kleiner oder gleich als *right* oder *top* ist

kleiner oder gleich als *bottom*). Aber meistens wird Ruby/GTK die Tatsache akzeptieren, dass die Werte von *left*, *right*, *top* oder *bottom* falsch sind, selbst wenn die Tabellenzellen sich überschneiden werden

Die *xopts* und *yopts* Argumente legen es fest, wie die Tabelle sich einen zusätzlichen Platz für ihre Kinder-Widgets schaffen würde. Gültige Werte für die beigen Argumente wären z.Beiispiel, GTK_EXPAND, GTK_FILL oder GTK_EXPAND|GTK_FILL(beiide expand und fill). Die Bedeutung von diesen zwei "flags" ist genau gleich, wie entsprechende Parameter für die *pack_start* und *pack_end* Methoden von der "packing" Box. Und als Letztes legen zwei Argumente *xpad* und *ypad* horizontalen und vertikalen Abstand(in pixel) fest, und das gilt als ein Füller, der um einen Widget herum entsteht. Diese Argumente sind als ein Zusatz zu schon früher bekannten Einstellungen, die wir für die Tabelle in `Gtk::Table.new` angewendet haben.

Ruby/GTK Beispielprogramm

Unten wird der Quellcode vorgeführt, welchen Sie sich auch natürlich herunterladen können [gtk-xmlviewer](#)

```
#!/usr/local/bin/ruby -w
# Sample Application for Ruby/GTK - gtk-xmlviewer.rb
# Beispielprogramm fuer Ruby/GTK -gtk-xmlviewer.rb

require 'gtk'
require 'nqxml/treeparser'

class XMLViewer < Gtk::Window
  def initialize
    super(Gtk::WINDOW_TOPLEVEL)
    set_title('Ruby/Gtk XML Viewer')
    set_usize(600, 400)

    menubar = createMenubar
```

```
@treeList = Gtk::Tree.new
```

```
@treeList.show
```

```
@columnList = Gtk::CList.new(['Attribut', 'Wert'])
```

```
@columnList.show
```

```
bottom = Gtk::HBox.new(false, 0)
```

```
bottom.pack_start(@treeList, true, true, 0)
```

```
bottom.pack_start(@columnList, true, true, 0)
```

```
bottom.show
```

```
contents = Gtk::VBox.new(false, 0)
```

```
contents.pack_start(menuBar, false, false, 0)
```

```
contents.pack_start(bottom, true, true, 0)
```

```
add(contents)
```

```
contents.show
```

```
signal_connect(Gtk::Widget::SIGNAL_DELETE_EVENT) { exit }
```

```
end
```

```
def createMenuBar
```

```
  menuBar = Gtk::MenuBar.new
```

```
  fileMenuItem = Gtk::MenuItem.new("Datei")
```

```
  fileMenu = Gtk::Menu.new
```

```
  openItem = Gtk::MenuItem.new("Öffnen...")
```

```
  openItem.signal_connect(Gtk::MenuItem::SIGNAL_ACTIVATE) {
```

```
    openDocument
  }
  openItem.show
  fileMenu.add(openItem)

  quitItem = Gtk::MenuItem.new("Beenden")
  quitItem.signal_connect(Gtk::MenuItem::SIGNAL_ACTIVATE) { exit }
  quitItem.show
  fileMenu.add(quitItem)

  fileMenuItem.set_submenu(fileMenu)
  fileMenuItem.show

  helpMenuItem = Gtk::MenuItem.new("Hilfe")
  helpMenu = Gtk::Menu.new

  aboutItem = Gtk::MenuItem.new("Über..")
  aboutItem.signal_connect(Gtk::MenuItem::SIGNAL_ACTIVATE) {
    showMessageBox('Über XMLViewer', 'Ruby/GTK Beispielprogramm')
  }
  aboutItem.show
  helpMenu.add(aboutItem)

  helpMenuItem.set_submenu(helpMenu)
  helpMenuItem.show

  menubar.append(fileMenuItem)
  menubar.append(helpMenuItem)
  menubar.show
```

```
    menubar
end

def selectItem(entity)
  @columnList.clear
  if entity.kind_of?(NOXML::NamedAttributes)
    keys = entity.attrs.keys.sort
    keys.each { |key|
      @columnList.append([key, entity.attrs[key]])
    }
  end
end

def populateTreeList(docRootNode, treeRoot)
  entity = docRootNode.entity
  if entity.instance_of?(NOXML::Tag)
    treeItem = Gtk::TreeItem.new(entity.to_s)
    treeRoot.append(treeItem)
    if docRootNode.children.length > 0
      subTree = Gtk::Tree.new
      treeItem.set_subtree(subTree)
      docRootNode.children.each do |node|
        populateTreeList(node, subTree)
      end
    end
  end
  treeItem.signal_connect(Gtk::Item::SIGNAL_SELECT) {
    selectItem(entity)
  }
  treeItem.show
end
```

```
elsif entity.instance_of?(NOXML::Text) &&
  entity.to_s.strip.length != 0
  treeItem = Gtk::TreeItem.new(entity.to_s)
  treeRoot.append(treeItem)
  treeItem.signal_connect(Gtk::Item::SIGNAL_SELECT) {
    selectItem(entity)
  }
  treeItem.show
end
end

def loadDocument(filename)
  @document = nil
  begin
    @document = NOXML::TreeParser.new(File.new(filename)).document
  rescue NOXML::ParserError => ex
    showMessageBox("Fehler", "XML Document kann nicht geparkt werden")
  end
  if @document
    @treeList.children.each { |child|
      @treeList.remove_child(child)
    }
    populateTreeList(@document.rootNode, @treeList)
  end
end

def openDocument
  dlg = Gtk::FileSelection.new('Öffnen Datei')
  dlg.ok_button.signal_connect(Gtk::Button::SIGNAL_CLICKED) {
```

```
    dlg.hide
    filename =dlg.get_filename
    loadDocument(filename) if filename
  }
  dlg.cancel_button.signal_connect(Gtk::Button::SIGNAL_CLICKED) {
    dlg.hide
  }
  dlg.show
end

def showMessageBox(title, msg)
  msgBox = Gtk::Dialog.new

  msgLabel = Gtk::Label.new(msg)
  msgLabel.show

  okButton = Gtk::Button.new('OK')
  okButton.show
  okButton.signal_connect(Gtk::Button::SIGNAL_CLICKED) {
    msgBox.hide
  }

  msgBox.set_usize(250, 100)
  msgBox.vbox.pack_start(msgLabel)
  msgBox.action_area.pack_start(okButton)
  msgBox.set_title(title)
  msgBox.show
end
end
```

```
if $0 == __FILE__
  mainWindow = XMLViewer.new
  mainWindow.show
  Gtk::main
end
```

Nachdem Sie den Code geschrieben oder heruntergeladen haben, nehmen wir uns die Zeit, um den Code genau zu analysieren. Wir beginnen mit der *initialize* Methode für die *XML Viewer* Klasse. XML Viewer ist eine Unterklasse von *Gtk::Window* und deswegen ist der erste Schritt, dass man die Basisklasse, durch den Aufruf der *super* Methode, initialisiert wird. Dabei hat *Gtk::Window.new* ein Singlargument, der auf den Fenstertyp hinweist; die Grundeinstellung ist *Gtk::WINDOW_TOPLEVEL*, aber es gibt auch andere gültige Werte *Gtk::WINDOW_POPUP* und *Gtk::DIALOG*. Die nächste zwei Zeilen legen den Fenstertitel fest und wird auch die Breite und Höhe initialisiert.

Die nächste Aufgabe ist das Erzeugen einer Menübalke und Pulldownmenüs. Wir haben es absichtlich in eine separate *createMenubar* hineingesteckt, um den Code transparenter zu machen. Eine Erzeugung Menüs in Ruby/Gtk erfordert eine Erstellung des *Gtk::MenuBar* Widgets und dann werden zu dem ein oder mehrere *Gtk::MenuItem* Objekte hinzugefügt. Ein Menüeintrag kann einen wirklichen Menübefehl darstellen, oder der kann verwendet werden, um ein Untermenü oder andere Menü's Einträge, die im *Gtk::Menu* Widget enthalten sind. Dieser Auszug aus der *createMenubar* Methode veranschaulicht uns sozusagen wichtige Schlüsselpunkte:

```
fileMenuItem = Gtk::MenuItem.new("Datei")
fileMenu = Gtk::Menu.new

openItem = Gtk::MenuItem.new("Öffnen...")
openItem.signal_connect(Gtk::MenuItem::SIGNAL_ACTIVATE) {
  openDocument
}
```

```
openItem.show
fileMenu.add(openItem)

fileMenuItem.set_submenu(fileMenu)
fileMenuItem.show
menubar.append(fileMenuItem)
```

Der *Datei* Menüeintrag(fileMenuItem) ist eine Instanz vom *Gtk::MenuItem* Widget, deren Absicht ist, das Untermenü(fileMenu) zu zeigen, das noch weitere Menüeinträge unterbringen kann. Wir rufen eine *set_submenu* Methode auf, um eine Verbindung zwischen *fileMenuItem* und *fileMenu* herzustellen. Im Gegensatz zu *Datei* Menüeintrag stellt der *Offen...* Untermenüeintrag(openItem) einen Befehl für das Programm dar, wir verbinden den Eintrag durch das *activate* Signal mit *openDocument* Methode, welche wir uns später ansehen.

Zurück zur *initialize* Methode; wir erzeugen und zeigen den "tree list"(Baumstrukturige Liste) Widget:

```
@treeList = Gtk::Tree.new
@treeList.show
```

Und außerdem ein gespalteter List-Widget:

```
@columnList = Gtk::CList.new(['Attribut', 'Wert'])
@columnList.show
```

Hier haben wir eine Konstruktion für *Gtk::CList* verwendet, welche ein Array mit Spaltennamen spezifiziert; eine alternative Konstruktion erlaubt uns eine einfache Festlegung der Anzahl von Spalten und dann setzt man ihre Titels später zusammen und dabei wird die *set_column_title* Methode verwendet. Der gesamte Layout des Hauptfenster's Widgets wird durch die Verwendung der horizontalen "packing" Box dargestellt, die sich ihrerseits in vertikalen "packing" Box befindet. Die horizontale "packing" Box(*bottom* genannt) enthält den *Gtk::Tree* Widget auf der linken Seite und den *Gtk::CList* auf

der rechten. Die vertikale "packing" Box(*contents* genannt) enthält eine Menübar, die am oberen Rand liegt und der Rest des Raumes gehört zur horizontalen "packing" Box. Dabei ist zu merken, dass die Arguments von der *pack_start* Methode für die Menübar direkt in der vertikalen "packing" Box untergebracht sind, um die Ausdehnung der Menübar zu verhindern, selbst wenn es einen zusätzlichen Platz gibt:

```
contents.pack_start(menubar, false, false, 0)
```

Die letzte Zeile der *initialize* Methode baut einen "Signalhandler" fürs Hauptfenster selbst ein. Wenn das Hauptfenster sozusagen "deleted"(gelöscht) wird(gewöhnlich beim Klicken des **x** Buttons in der oberen rechten Ecke des Fensters), GTK+ wird das abschließen, dabei wird eine symbolische Konstante für *delete_event* `Gtk::Widget::SIGNAL_DELETE_EVENT` verwendet. Es wird dieses Ereignis eingefangen und das Fenster sofort geschlossen.

Nun tauchen wir weiter in die nächste Ebene des Programms ein, wir sehen uns "Signalhandler's" für die Menübefehle an. Die werden festgesetzt, wenn wir die Menüeinträge in der *createMenubar* Methode erzeugen. Wir können uns schnell den **Beenden** Befehl ansehen, damit schließt man einfach das Programm:

```
quitItem = Gtk::MenuItem.new("Beenden")  
quitItem.signal_connect(Gtk::MenuItem::SIGNAL_ACTIVATE) { exit }
```

Der **Über** Menübefehl zeigt eine kleine Dialogbox, die ein Info über das Programm beinhaltet:

```
aboutItem = Gtk::MenuItem.new("Über...")  
aboutItem.signal_connect(Gtk::MenuItem::SIGNAL_ACTIVATE) {  
    showMessageBox("Über XMLViewer", "Ruby/GTK Beispielprogramm")  
}
```

Hier ist *showMessageBox* eine Hilfemethode für die *XMLViewer* Klasse, die eine Dialogbox mit einem

festgelegten Titel und einem Nachrichtstring zeigt und dazu auch ein **OK** Button, welcher die Dialogbox schließen lässt.

```
def showMessageBox(title, msg)
  msgBox = Gtk::Dialog.new

  msgLabel = Gtk::Label.new(msg)
  msgLabel.show

  okButton = Gtk::Button.new('OK')
  okButton.show
  okButton.signal_connect(Gtk::Button::SIGNAL_CLICKED) {
    msgBox.hide
  }

  msgBox.set_usize(250, 100)
  msgBox.vbox.pack_start(msgLabel)
  msgBox.action_area.pack_start(okButton)
  msgBox.set_title(title)
  msgBox.show
end
end
```

Es gibt viele zweckbedingte Methoden, die uns mehr Möglichkeiten geben könnten, die Größe der Box und den Layout zu kontrollieren, aber dieser einfacher Vorgang dient unseren Zwecken. Allerdings die GNOME Bibliothek (aufgebaut auf GTK+) bietet mehr leistungsfähige und leichtbediente Klasse für die Erstellung der Messageboxen und sogenannten **Über** Boxen in ihren Programmen an. Mehr Information über Ruby Bibliotheken für GNOME findet man auf Ruby/GTK Homepage.

Der Menübefehl, der sozusagen etwas zum Bewegen bringt, wie auch immer, ist der **Offnen** Befehl, welcher die XMLViewer's *openDocument* Methode aufruft:

```
def openDocument
  dlg =Gtk::FileSelection.new('Öffnen Datei')
  dlg.ok_button.signal_connect(Gtk::Button::SIGNAL_CLICKED) {
    dlg.hide
    filename =dlg.get_filename
    loadDocument(filename) if filename
  }
  dlg.cancel_button.signal_connect(Gtk::Button::SIGNAL_CLICKED) {
    dlg.hide
  }
  dlg.show
end
```

Die *new* Methode für *Gtk::FileSelection* hat ein String-Argument, der auf den Titelname von der Dialogbox hinweist. Diese Dialogbox bietet uns an, irgendwelche Datei auszuwählen. Dabei ist besonderes interessant, wie die "clicked" Signale für beide **OK** und **Abbrechen** in diesem Dialog abzufangen sind und wie dabei die *ok_button* und *cancel_button* Zugriffsmethode vom *Gtk::FileSelection* Widget zu verwenden sind. Insbesondere wür den wir gern eine Datei aufrufen, wir uns ausgesucht haben, (mit hilfe der *get_filename*) und schließlich die Datei zum Gesicht bekommen könnten. Um das Letzteres zu machen, brauchen wir die *loadDocument* Methode:

```
def loadDocument(filename)
  @document = nil
begin
```

```

    @document = NOXML::TreeParser.new(File.new(filename)).document
  rescue NOXML::ParserError => ex
    showMessageBox("Fehler", "XML Document kann nicht geparst werden")
  end
  if @document
    @treeList.children.each { |child|
      @treeList.remove_child(child)
    }
    populateTreeList(@document.rootNode, @treeList)
  end
end
end

```

Wenn der XML Parser während der Erzeugung des `NOXML::Document` Objektes eine Ausnahme auslöst, werden wir wieder unsere `showMessageBox` Hilfemethode benutzen. Dadurch wird der Anwender aufmerksam gemacht und es wird ein Hinweis auf den Fehler gegeben. Angenommen, die Datei wurde erfolgreich geladen und man wird danach den schon existierenden im Speicher baumstruktiregen Inhalt losgeworden, und schließlich durch die `populateTreeList` Methode wieder gefüllt. Um den Inhalt der baumstrukturigen Liste zu aufräumen, benutzen wir die `children` Methode (geerbt vom `Gtk::Container`), welche ein Array mit Baum's Elementen zurückgibt. Dann lassen wir jedes Element mit Hilfe von "Iterator" durchlaufen und dann löschen wir jeden Eintrag durch den Aufruf `remove_child` Methode.

Die `populateTreeList` Methode ruft sich selbst wiederholend auf, um neuen baumstrukturigen Inhalt aufzubauen. Der Prozess des Aufbaus vom `Gtk::Tree's` Widget ist ähnlich zum Prozess des Aufbaus von "Pull-Down" Menü, das wir in der `createMenubar` gemacht haben. Sie können `Gtk::TreeItem` Objekte zum `Gtk::Tree` Widget hinzufügen und dann zu diesen Einträgen Signal-Handler's anhängen, um eine Anzeige zu bekommen, when diese Elemente selektiert oder deselektiert, ausgebreitet oder zusammengeschloßen und etc. werden. Aber ebenso wie `Gtk::MenuItem` Objekte Submenüs haben können, die dem Menüeintrag verbunden sind (stufenähnliche Pulldown's Menüs), können `Gtk::TreeItem` Objekte "Sub-Trees" haben, das ist aber ganz Anderes, als `Gtk::Tree` Objekt. In folgenden Auszug aus der `populateTreeList` Methode verwenden wir diese Konstruktion, um XML Datei's Knoten zu modellieren:

```

treeItem = Gtk::TreeItem.new(entity.to_s)
treeRoot.append(treeItem)
if docRootNode.children.length > 0
  subTree = Gtk::Tree.new
  treeItem.set_subtree(subTree)
  docRootNode.children.each do |node|
    populateTreeList(node, subTree)
  end
end
end

```

Hier *treeItem* ist ein Kind des aktuellen *treeRoot* Objektes (welches selbst eine *Gtk::Tree* Instanz ist). Wenn wir sehen, dass diese XML Entity eine oder mehrere Entities hat, erstellen wir eine neue *Gtk::Tree* Instanz (genannt *subTree*) und danach rufen wir die *set_subtree* Methode, um einen Eintrag (*treeItem*) für diese Instanzen zu setzen.

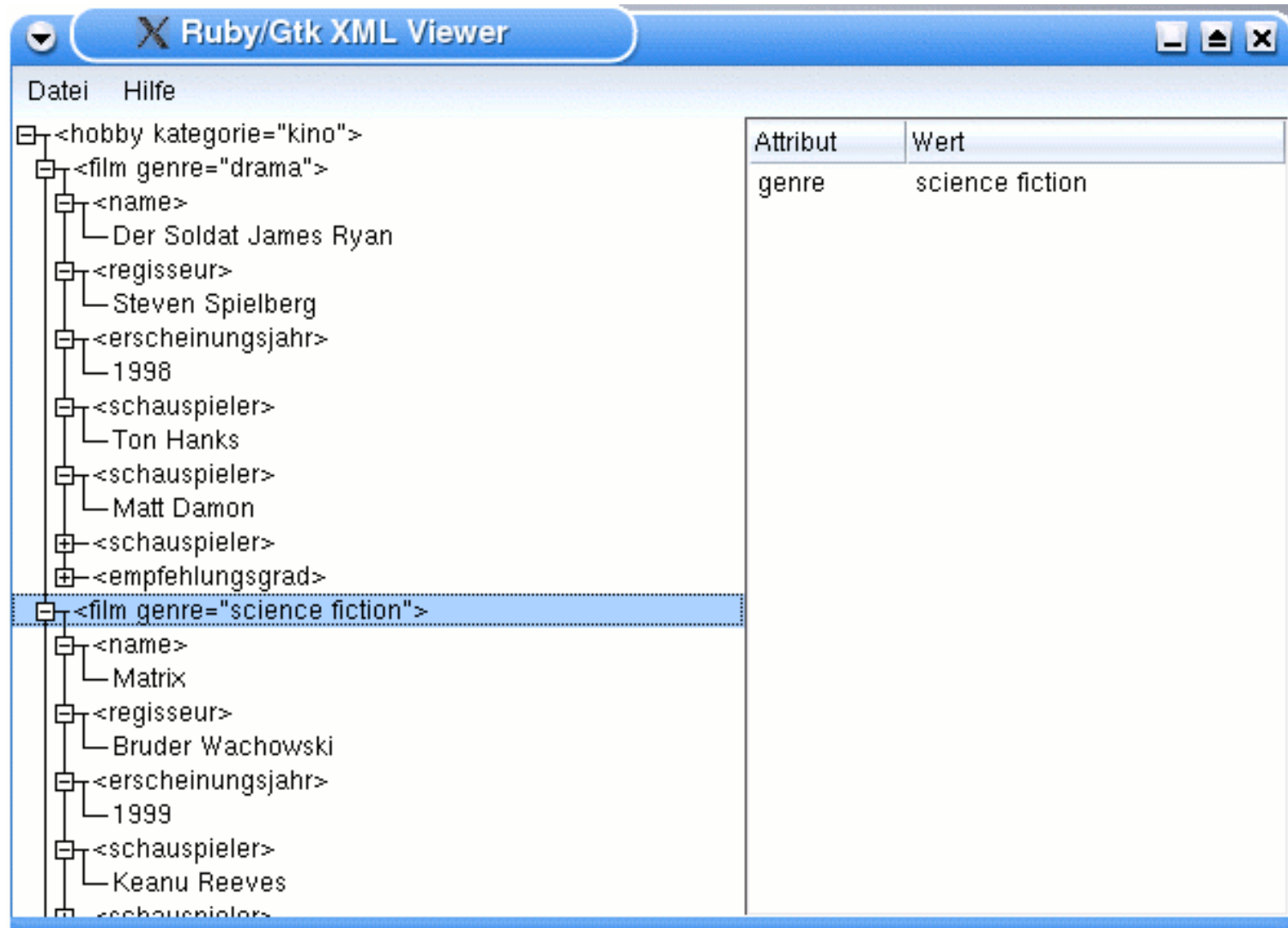
Jedesmal wenn ein Element im XML Dokument ausgewählt wird, möchten wir auch, dass die Attributliste (unseres *Gtk::CList* Objekt), das sich in der rechten Hälfte befindet, neu geladen wird. Um das zu machen, hängen wir ein Handler zu jedem Element an. Damit wird durch das Signal *Gtk::Item::SIGNAL_SELECT* unsere *selectItem* Methode aufgerufen:

```

def selectItem(entity)
  @columnList.clear
  if entity.kind_of?(NOXML::NamedAttributes)
    keys = entity.attrs.keys.sort
    keys.each { |key|
      @columnList.append([key, entity.attrs[key]])
    }
  end
end
end

```

Diese Methode beginnt mit dem Ausräumen der alten Inhaltsliste und dann wenn es irgendwelche Attributes gibt, die mit ausgewählten XML-Entity verbunden sind, werden sie rechts angezeigt. Und so sieht unsere Applikation unter Linux aus:



Verwendung Glade GUI Builder

Glade(<http://glade.gnome.org>) ist ein GUI Programmtool für GTK+ und GNOME. Seine Entwickler sind

Damon Chaplin und Martijn van Beers. Sie können sich den aktuellen Sourcecode für Glade von der Glade Homeseite herunterladen, aber üblich ist es, dass bei meisten Linuxdistributionen Glade schon vorhanden ist. Dieser Teil ist nicht dazu gedacht, einen Bericht über Glade zusammenzustellen, aber es gibt im Internet reichlich Information über diese Anwendung. Es gibt auf <http://glade.gnome.org/FAQ> Seite eine einfache Textversion von der Glade FAQ Liste, außerdem beinhaltet GNOME Version von Glade Clade FAQ-Liste, Manual und Quick-Start Guide.

Glade's Projektdatei ist(*.glade Datei) ist eine XML Datei, die alle Information über die grafische Oberfläche beinhaltet. James Henstridge hat zur Unterstützung eine Bibliothek entwickelt [libglade](#), welche uns erlaubt, Glade's Projektdateien auszulesen und dynamisch in Echt-Zeit(runtime) die GUI's zu erzeugen. Das ist wichtig, weil es uns erlaubt, Glade zu verwenden, um grafische Oberfläche für die Programmiersprachen, die Glade nicht direkt unterstützen, zu entwickeln. Die Homeseite vom libglade Projekt ist www.daa.com.au/~james/gnome, aber wie gesagt genauso wie Glade kommt diese Bibliothek bei allen Linuxdistributionen als Standardoption vor.

Ruby/Libglade ist eine Erweiterung, die von Avi Bryant entwickelt war und die bietet uns sozusagen "Hülle"(wrapper) für [libglade](#) an. Zur Zeit gibt's keine offizielle Homeseite dafür, aber man kann sich die letzte Version der Erweiterung von RAA herunterladen. Der Sourcecode enthält unter anderen Installations- und Verwendungsanweisungen, sowie auch ein Beispielprojekt für Testzwecken.

Ruby/LibGlade definiert eine Single-Klasse [GladeXML](#). Die [new](#) Methode vom [GladeXML](#) hat als Argument den Dateinamen des Glade-Projektes, und optional den Namen des Root-Widgets vom Teil der Benutzeroberfläche, die sie entwickeln möchten. Wenn Sie doch das ganze Projekt laden wollen, dann kann man einfach den zweiten Argument auslassen.

Letzendlich [GladeXML.new](#) erwartet auch eine in einem "Iterator-Style" Coderblock, welcher verwendet wird, um Signal-Handler's Namen mit Ruby Prozeduren oder Methoden zu verbinden. Während [libglade](#) mit dem Laden einer Information über ihre Benutzeroberfläche aus der Glade Projektdatei beginnt, ruft sie einen "Iterator-Codeblock" auf und zwar für jeden Handler's Namen, auf den der sozusagen stößt. Der Codeblock würde dabei einen von beiden Ruby's [Proc](#) oder [Method](#) zurückgeben, die uns den Code anbietet, um GTK+ Signal zu handzuhaben. Zum Beispiel eine Version, die [Proc](#) Objekte zurückgibt,

würde so aussehen:

```
GladeXML.new('myproject.glade') { |handler_name|
  case handler_name
    when "on_button1_clicked"
      proc { puts "Goodbye, World!"; exit }
    when "on_button2_clicked"
      proc { puts "button2 was clicked" }
  end
}
```

Wenn Sie ihren Code so erzeugt haben, dass die Ruby Methoden, die die Signale handhaben, die gleiche Namen haben, wie Handler's Namen, die Sie in Glade festgelegt haben, und noch sogar saubere Methode wäre es, Ruby's "Kernel#method" Methode zu verwenden, damit wird automatisch zu diesem Handler's Methoden Verweise definiert:

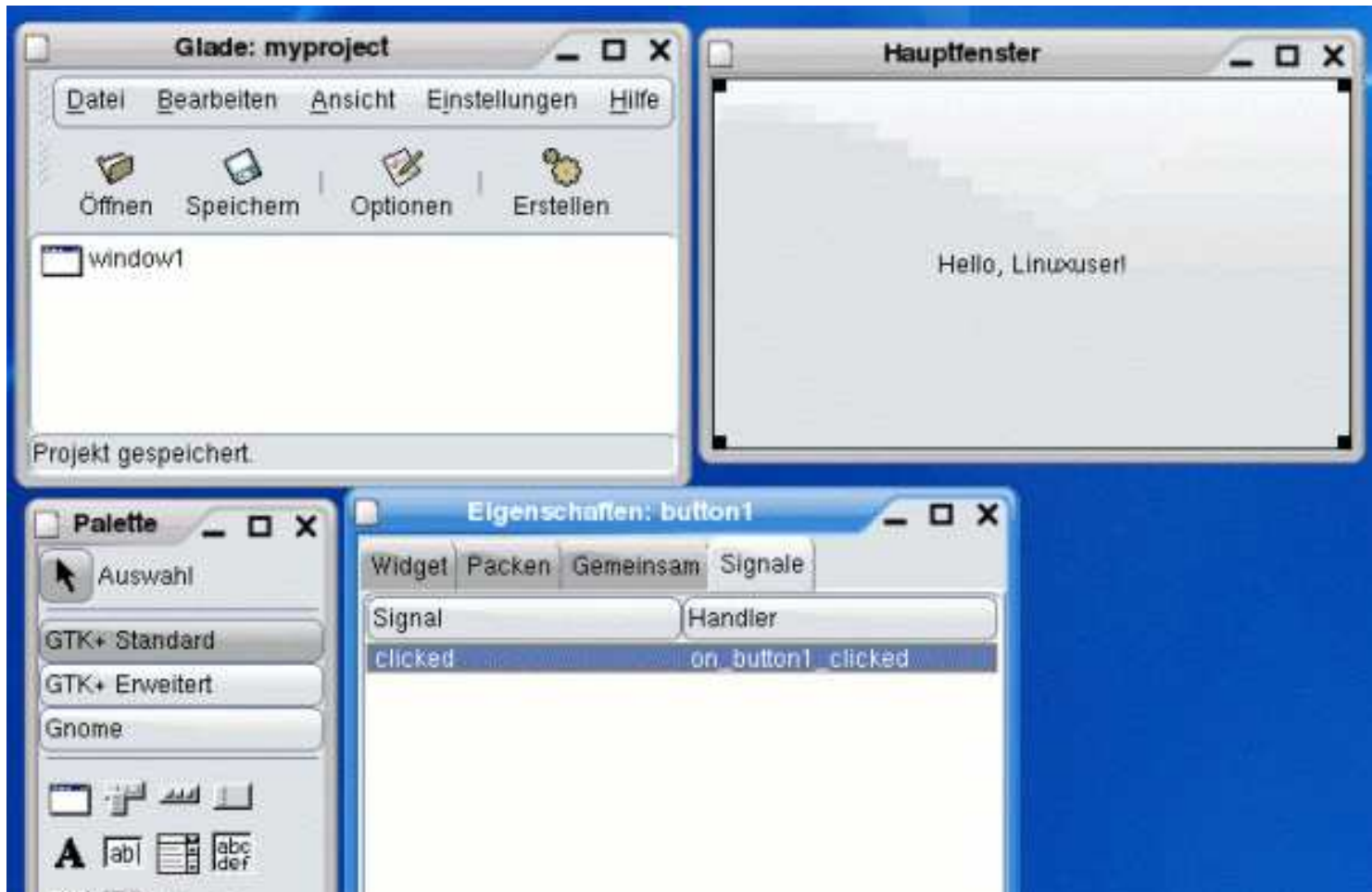
```
def on_button1_clicked
  puts "Goodbye, World!"
  exit
end

def on_button2_clicked
  puts "button2 was clicked"
end

GladeXML.new('myproject.glade') { |handler_name|
  method(handler_name)
}
```

Die *GladeXML* Klasse bietet zwei andere Instanz-Methoden *getWidget* und *getWidgetByLongName* an. Beide Methode geben zu spezifischen Widgets Referenzen zurück und beide haben als Input ein Sing-String Argument. Die *getWidget* Methode nimmt für den Widget einen kleinen Namen(z.B. "button1"), während *getWidgetByLongName* den Namen hat, welcher auf absoluten Pfad hinweist(z.B. "mainWindow.hbox.button1").

Folgende Abbildung zeigt uns eine Glade's Session mit orinaller **Hello, Linuxuser!** Benutzeroberfläche an. Diese besteht aus einem Top-Level Fenster und einem Button, welcher als Kind-Widget fürs Hauptfenster ist. Wie man schon sieht, haben wir hier für den Button Signal-Handler *clicked* hinzugefügt und haben ihn als *on_button1_clicked* bennant. Die Namen für die Signal-Handler, die Sie in Glade festlegen, sind die Verbundung zwischen Benutzeroberflächen und Ruby's Code wichtig.





Nachdem Sie das Glade Projekt in der Datei(myproject.glade) gespeichert haben, schreiben wir ein kleines Rubyprogramm, das Ruby/LibGlade verwendet:

```
require 'gtk'
require 'lglade'

def on_button1_clicked
  puts "Goodbye Linuxuser!"
  exit
end

GladeXML.new('myproject.glade') { |handler_name|
  method(handler_name)
}
```

Gtk::main

Die ersten zwei Zeilen in diesem Programm importieren Ruby/GTK und Ruby/LibGlade Erweiterungen, wo "libglade" eine Module von Ruby/LibGlade ist. Der nächste Teil des Programms definiert die *on_button1_clicked* Methode, die wir fürs Button's *clicked* Signal verwenden. Als Nächstes erzeugen wir *GladeXML* Objekt fürs unsere myproject.glade Projekt und danach verbinden wir den Handler's Namen "on_button1_clicked" mit der passenden *on_button1_clicked* Handler's Methode. Schließlich wie bei Ruby/GTK's Programmen, beenden wir die Application mit GTK+ main event loop.

Verwendung FOX Toolkit

Free Objects for X(FOX) ist ein cross-platform GUI Toolkit, der vom Jeroen van der Zijp entwickelt wurde. In Vergleich zu Tk und GTK+ ist FOX ein ziemlich junger Werkzeug, aber mit der Zeit gewinnt er unter Softwareentwicklern mehr an Beliebtheit. FOX wurde ursprünglich für UNIX und Linux geschrieben und später auf andere Betriebssysteme(Windows und MacOS) übertragen. FXRuby ist ein Ruby Erweiterungsmodul, das eine Schnittstelle zwischen FOX und einem Ruby Programm anbietet. Das Modul wurde vom Lyle Johnson entwickelt und kann von Homeseite <http://fxruby.sourceforge.net> heruntergeladen werden.

Installation

Die Voraussetzung für die Programmierung mit FXRuby ist eine arbeitsfähige FOX-Umgebung. Wenn Sie Standard-Rubyinstallation fürs Windows verwenden(von der Pragmatic Programmer's Homeseite), ist die Wahrscheinlichkeit, dass das Modul schon mit Rubyinstallation mitinstalliert wurde, groß. Sie können aber eine kompatible schon vorbereitete Binary-Distribution des FXRuby Moduls von der FXRuby Homeseite herunterladen. Der Vorteil dieses Pakets ist, dass die "shared" Bibliothek in dieser Distribution schon ein FOX Bibliothek enthält, so dass man nach dem Downloaden und dem Installieren der Distribution in der Lage ist, mit dem Programmieren anzufangen und besonders in der erster Stelle sich das Ergebnis anzuzeigen. Man braucht also keine Extrainstallation von FOX.

Wenn Sie eine andere Ruby-Version benutzen(nicht für Windows), werden Sie sich höchstwahrscheinlich etwas mehr anstrengen. Im Gegensatz zu Tk und GTK+ beinhaltet keine Linux-Distributionen ein Standard-Installation Paket von FOX, was sich allerdings mit der Zeit ändern wird. Also Sie müssen sich FOX downloaden, kompilieren und anschließend installieren. Den Sourcecode von FOX kann man sich von der FOX Homeseite herunterladen(www.fox-toolkit.org/). Eine Installationsanweisung sollte dabei sein. Für Linux und Unix-Derivaten wird der Prozess der Installation bekannt vorkommen: **configure**, **make** und

make install. Sobald man eine funktionsfähige FOX-Installation hat, kann man mit dem FXRuby-Erweiterungsmodul beginnen. Der Installtionsprozess für FXRuby beginnt mit der Konfiguration:

```
ruby setup.rb config
```

Dann ruft man die Installtion selbst:

```
ruby setup.rb setup
```

Erst wenn der Buildprozess beendet ist, kann man FXRuby komplett installieren:

```
ruby setup.rb install
```

Um mehr die Information über den Installationsprozess zu bekommen, schlagen Sie in der FXRuby Dokumentation nach.

FXRuby Basis

FXRuby's API (Application Programming Interface)-Schnittstelle folgt genau der FOX's C++ API und meistens werden Sie die Standard FOX-Klasse Dokumentation verwenden, um FXRuby Klassehierarchie und Schinttstelle zu lernen. Alle FXRuby Klassen , Methoden un Konstanten sind in dem Single-Ruby Modul, benannt als *Fox* untergebracht und das meistens von FXRuby wurde im C++ Erweiterungsmodul implementiert, dessen Name *fox* ist. Ein kleines FXRuby Programm würde so aussehen:

```
require 'fox'  
  
include Fox  
  
application = FXApp.new("Hello", "FoxTest")  
application.init(ARGV)
```

```
main = FXMainWindow.new(application, "Hello", nil, nil, DECOR_ALL)
FXButton.new(main, "&Hello World!", nil, application, FXApp::ID_QUIT)
application.create()
main.show(PLACEMENT_SCREEN)
application.run()
```

Das Programm lädt das *Fox* Modul bei erforderlicher *fox* Feature. Trotz der Tatsache, dass alle FXRuby Klasse in dem entsprechenden Namensraum vom *Fox* Modul zu finden sind, beginnt der Klassenname mit *FX* Prefix, um sozusagen eine Diskrepanz mit anderen Klassennamen zu vermeiden. Und das ist aus dem Grund, dass viele FXRuby Programms können sicher den Inhalt des *Fox* Modules direkt in globalen Namensraum verwenden(*include Fox*)

Bevor wir uns mit der Analyse dieses kleines Programms beginnen, füge ein paar Worte dazu, die im Originalartikel nicht zu finden sind. Also dieses kleines Programm kann man statt normalen Dateinamenerweiterung *.rb* die Erweiterung *.rbw* verwenden. Das ist die übliche Erweiterung für FXRuby-Applikationen Welche Vorteile bringt das uns? Unter Windows kann man einfach auf die Datei doppelklicken, dabei wird keine Eingabeaufforderung geöffnet, die da für Ein- und Ausgabe bereitsteht, sondern wird das Programm wie üblich gestartet. Der "Nachteil" dieses Vorgehens ist es, dass man dabei alle Fehlermeldungen verloren gehen.

Die Applikation beginnt mit der Erzeugung eines *FXApp* Objektes. Bei seiner Erzeugung mit *new* werden zwei Zeichenketten(stings) als Argumente übergeben: ein interner Name für FOX-Anwendung(hier einfach "Hello") und ein so genannter *vendor key*, welcher die Art oder den Hersteller der Anwendung(hier "FoxTest") beschreibt. Von allen Toolkits, die wir unter der Lupe nehmen, ist FOX der Einzige, der eine explizite Erzeugung und Verweisung auf *application* Objekt braucht, welches als eine Art der Quelle für globale Programmenressourcen(wie Grundapplikation's Farben, Schriften und etc.) Das ist auch im Wesentlichen für die Verwaltung des Ereignisschleife(event loop) verantwortlich, wie wir am Ende sehen werden.

Der nächste Schritt ist das Initialisieren des Programmes. Wir rufen die *init* Methode auf, die zu unserem

Anwendung's Objekt gehört und übergeben in der Kommandozeile ein Argument und zwar Ruby's [ARGV](#) Array, welches für FOX sinnvolle Optionen übergibt. Zum Beispiel, um FOX's tracing(spur) Output zu aktivieren, kann man in der Kommandozeile die [tracelevel](#) Option verwenden:

```
ruby hello.rb -tracelevel 301
```

Um mehr Information darüber zu bekommen, schlagen Sie bitte in der FOX Dokumentaion.

An dieser Stelle sind wir endlich dazu gekommen, unseren ersten Widget zu erzeugen. Das Hauptfenster (bennant [main](#)) ist eine Instanz von der [Fox::FXMainWindow](#) Klasse, die [new](#) Methode erwartet eine Referenz zu Applikation's Objekt, das ist zuvor erzeugte FOX-Anwendung, den Fenstertitel und auch drei Optionen, die das Aussehen des Fensters beeinflussen.

Der nächste Widget ist ein Button(also eine Instanz von [Fox::FXButton](#)), und wurde als ein Kind des Hauptfensters erzeugt. Dieser Button zeigt uns eine Zeichenkette "Hello World!" und die erste Buchstabe ("H") wird untergestrichen, weil wir ein "Ampersand"-Zeichen("&") vor der Buchstabe plaziert haben. Das ist ein spezieller Signal, das zum [FXButton](#) Widget gesendet wird, um den sogenannter "Schortkey" in diesem Fall **Strg+H** zu verwenden, damit kann man den gleichen Effekt erreichen, wenn man einfach direkt auf den Button klickt. Der vierte und fünfte Argumente von der [new](#) Methode sind wichtig, und zwar in Bezug, wie FOX die Ereignisse verarbeitet. Also der vierte Argument ist ein Zielobjekt(eine Instanz von [FXApp](#) oder eine von seiner Subklasse), das eine Nachricht bekommt, und schließlich der fünfte Argument, der als "Nachrichtenbezeichner" (message identifier) da ist. In unserem Fall ist "application"(das Programm selbst) ein "Nachrichtenziel" für den Button, und wenn man darauf klickt, werden, bzw. wird, Ereignisse generiert und als ein Ergebnis an das Programm geschickt. Der Nachrichtenbezeichner dient zur Unterscheidung zwischen ähnlichen aber verschiedenen Nachrichten, welche das Zielobjekt empfangen könnte; irgendein Objekt kann als "Nachrichtenziel" für mehrere Widgets sein.

Die serverseitige(gemeint wird hier,dass FOX ursprünglich für X-Server-Client Applikationen geschafft worden war) Ressourcen, wie Fenster, Schriftarten und Cursors, werden während des Aufrufes der [create](#) Methode erzeugt. Dass der FOX zwischen der Darstellung von klientseitigen Objekten(so wie seine

Instanzvariablen) und serverseitigen Ressourcen, die mit ihren Objekten verbunden sind, unterscheidet, macht FOX einzigartig. Eins wenn Fenster's und andere Ressourcen erzeugt worden sind, wird das Hauptfenster zentriert und danach starten wir Hauptereignisschleife (main event loop) mit der Hilfer der *run* Methode.

Zielobjekte und Nachrichten

FOX's Ereignis Model basiert auf der Idee der Sendung *messages* von einem Applikationsobjekt zum anderen, das als sein Ziel da ist, when etwas geschieht. Also das ist Ihr Job als Programmier, sich zu entscheiden, wie das Zielobjekt auf eine Nachricht(message) antwortet. Da dieses "Ziel-Nachricht" System "innewohnend"(inherently) und "bidirectional" ist, bringt ein Teil des Antwortes mit sich häufig eine *back* Nachricht zum Originalsender mit.

Jeder Widget in ihrem Programm hat eine Fähigkeit zum Senden einer Nachricht, aber manche Nachrichten sind mehr "bedeutungsvoller" als andere; zum Beispiel Sie möchten sich wahrscheinlich ein Programm, das sich in irgendeiner Art und Weise reagiert, when der Anwender einen neuen Text in einem Textfeld eintippt, oder sich einen Eintrag in einer baumstrukturigen Liste auswählt. Sie würden ein Zielobjekt, das Nachrichten entgegenimmt, für diese Widgets festlegen, an denen Sie möchten die wichtige Nachrichten senden. Fast jeder von FOX's Widgets erlaubt uns in seiner Konstruktion, seine Zielobjekte festzulegen. Zum Beispiel um eine neue Instanz von der *FXList* Klasse und ihres *anObject* als Ziel festzulegen, würden so schreiben:

```
myList = FXList.new(parent, numItems, anObject, ...)
```

Sie können auch, wenn man so braucht, das Ziel, nachdem der Widget schon erzeugt wurde, ändern, dabei wird eine *setTarget* Methode aufgerufen:

```
myList.setTarget(anotherObject)
```

Jede Nachricht hat ein sogenannter Typ *type*, was auf seine Wichtigkeit hinweist. Manche Nachrichten

stellen low-level Ereignisse dar, die vom Windows System generiert wurden.; zum Beispiel die `SEL_LEFTBUTTONPRESS` Nachricht wird gesendet, wenn die linke Maustaste gedrückt wird, während die `SEL_LEFTBUTTONRELEASE` Nachricht gesendet wird, wenn der gleiche Button released wird. Andere Nachrichten werden von FOX generiert und weisen mehr auf die Ereignisse, die von Benutzern ausgelöst sind, hin, wie zum Beispiel das Löschen eines Textes aus dem Textwidget oder das Auswählen einer Zelle in einer Tabelle. Die Nachrichtentypen sind einfach ganze Zahlen (integer) (in Gegensatz zur Zeichenketten (Strings), die in Ruby/Tk Ereignissen oder Ruby/GTK Signalen verwendet werden), aber Sie müssen immer eine symbolische Konstante mit dem Namen in der Form wie `SEL_name` verwenden.

Ein Nachrichtentyp, den Sie häufig in FOX's Programmen treffen würden ist die `SEL_COMMAND` Nachricht. In Allgemeinen wird damit gemeint, dass der Widget nur seine Hauptaktion erledigt, es klingt ein wenig komplizierter, aber lassen wir uns das an einem Beispiel erläutern; ein `FXButton` Widget wird eine `SEL_COMMAND` zu seinem Zielobjekt aussenden, nachdem der User auf den Button klickt, während ein `FXTextField` Widget wird eine `SEL_COMMAND` aussenden, wenn der User die **Enter** Taste im Textfeld betätigt, oder einfach außer dem Textfeldes klickt.

Es ist schon eine allgemeine Gewohnheit in FOX Applikationen, ein Objekt als ein Ziel mehreren Widgets Nachrichten zu machen. Zum Beispiel, ihr Programm könnte mehrere Menübuttons enthalten, für solche Operationen wie "Öffnen Datei", "Speichern Datei", oder "Drücken Datei", und alle diese Buttons senden ihre `SEL_COMMAND` Nachrichten zu einem Zielobjekt. Aber Sie werden sich fragen, wie dieser Zielobjekt in Zustände ist, zwischen ähnlichen Nachrichten von verschiedenen Widgets zu unterscheiden. Sie denken, wenn das Zielobjekt eine `SEL_COMMAND` Nachricht empfängt, wie das Zielobjekt herausgefunden, welcher Button eine Nachricht geschickt hat? Die Antwort ist einfach, weil jede Nachricht ein Nachrichten-ID (Identifier) enthält (zusätzlich zu seinem Typ), damit wird zusätzliche Information über die Quelle und die Wichtigkeit der Nachricht erreicht.

Das Nachrichten-ID ist einfach eine ganze Zahl (integer), gewöhnlich als eine symbolische Konstante dargestellt, die beim "Empfänger" der Nachricht definiert wurde. Eine Klasse definiert verschiedene Nachrichten-ID und, da FOX ein objekt-orientierendes Toolkit ist, ein Objekt versteht auch alle Nachrichten-ID, die bei der Vorfahrklasse festgelegt wurden. Zum Beispiel, da `FXButton` eine Subklasse

von FXLabel ist, erbt er die Nachrichten-ID, die bei FXLabel und FXLabel's Basisklassen definiert wurden.

Um für ein Zielobjekt eine Nachricht zu empfangen und danach zu antworten, muss man Nachrichtenbehandlungsfunktionen für verschiedene Nachrichtentypen und Nachrichten-ID registrieren. Diese Zuordnung erfolgt in der *initialize* Methode unserer definierten Klasse durch die Verwendung FXMAPFUNC Methode erledigt, damit werden der Nachrichtentyp und die Nachrichten-ID mit dem Namen der Instanzmethode für diese Klasse verbunden. Zum Beispiel wenn wir sozusagen *SEL_COMMAND* Nachrichtentyp mit *ID_OPEN_FILE* Nachrichten-ID verwenden wollen, und benutzen *onOpenFile* Methode um diese Nachricht zuhandhaben, würde wir so in der *initialize* Methode schreiben:

```
FXMAPFUNC(SEL_COMMAND, ID_OPEN_FILE, "onOpenFile")
```

Die Funktionen, die die Nachrichten abwickeln, wie unsere *onOpenFile* Methode, haben immer drei Argumente:

```
def onOpenFile(sender, sel, ptr)
...irgendwelcher Code...
end
```

Das erste Argument(*sender*) ist das FOX-Objekt, das die Nachricht versendet(der Sender), Es ist oft praktisch zu wissen, wer eine Nachricht versendet hat, vor allem, wenn ein Teil der Antwort eine Nachricht *back* mit sich bringt, um zum ursprünglichen Sender etwas mitzuteilen. Das zweite Argument ist der sogenannte *Selektor* und ist eigentlich eine ganze Zahl, welcher(Selektor) die beide Attribute Nachrichtentyp und Nachrichten-ID verschlüsselt. Wenn es erforderlich ist, können Sie den Nachrichtentyp und die Nachrichten-ID aus dem Selektor entnehmen, dabei muss man die SELTYPE und SELID Funktionen verwenden:

```
def onOpenFile(sender, sel, ptr)
| hfkhfmessageType = SELTYPE(sel)
| hfkhfmessageId = SELID(sel)
| hfkhf...irgendwelcher Coder...
```

end

Das letzte Argument, das zur Nachrichtbehandlung weitergeleitet wird, ist *ptr*; es enthält zusätzliche Daten; der Typ der Daten hängt von Sender- und Nachrichtstypen ab. Zum Beispiel, wenn ein `FXTextField` Widget eine `SEL_COMMAND` Nachricht zu einem Ziel aussendet, die Daten, die mit der Nachricht mitgesendet werden, beinhalten eine Zeichenkette(string), die nicht alles Anderes ist, als ein Inhalt des Textfeldes. Wenn ein `FXColorWell` Widget zu seinem Ziel `SEL_COMMAND` Nachricht aussendet, wie auch immer, die Nachrichtendaten sind ein ganze Zahl, die auf die Farbe der aktuellen Farbenquelle hinweist. Unseres Beispielprogramm hat einige Arten der Nachrichten, die innerhalb des Programm verwendet werden, aber um komplette Information darüber zu erlangen, schlagen Sie auf der FOX Homepage nach.

Arbeitsweise mit FOX's Layout Manager

Die Auswahl von Fox's Layout Manager ist ähnlich zu der, die wir für Tk und GTK+ hatten, es gibt dabei ein wenig Unterschiede. Wir werden uns lediglich auf vier Arten des Layout Manager's konzentrieren und zwar: `FXPacker`, `FXHorizontalFrame`, `FXVerticalFrame` und `FXMatrix`. Wie ihre Rivalen in Tk und GTK+ sind diese Layout Manager, die am häufigsten beim GUI Programmieren verwendet werden. Für eine Information darüber, wie andere spezifische Layout Manager(wie `FXSwitcher`, `FXSplitter` und `FX4Splitter`) benutzt werden, müssen in der FOX Dokumentation nachschlagen.

Wie in GTK+ sind FOX Layout Manager selbst einfach unsichtbare Behälter für andere Widgets. Sie sind aber nicht so genau unsichtbar, weil Sie eine gewisse sozusagen Kontrolle darüber haben, wie zum Beispiel die Außenränder des Behälters gezogen werden(Frame Style), aber wir werden uns meistens mit der Zuordnung von Kinder-Widgets auf der Oberfläche beschäftigen. Wie in Tk werden FOX Widgets immer so erzeugt, dass sie dabei als erstes Argument ihre Vater-Widget bekommen. Sie können später einem Kind-Widget einen neuen Elternteil zuweisen (das ist, wenn Sie ihn von einem Vater-Widget entfernen und dem anderen hinzufügen), aber in Gegensatz zu Ruby/GTK kann Kind-Widget ohne den Vater nicht existieren. Auch so ungleich verhalten sich die Fox Kind-Widgets bei der Festlegung ihrer Layout's Präferenzen(oder layout hints genannt) und zwar als ein Teil ihrer Konstruktion. Sie können ihre Layout's Einstellungen ändern, auch wenn sie schon existieren. Man ruft dabei eine `setLayout`

Instanzmethode, aber das ist trotzdem ein anderes Modell, als das in Tk und GTK+ verwendet wird. Wie der FOX Layout Manager funktioniert, ist es einzigartige Layout's Strategie. Er braucht die Layout's Einstellungen von jedem seiner Kind-Widgets und danach verwendet sie, um die Größe und Position festzusetzen.

Wir werden uns jetzt FXPacker anschauen, weil dieser meistens von allen Layout Manager verwendet wird und der dient als Basisklasse für die anderen drei: FXHorizontalFrame, FXVerticalFrame und FXMatrix). FXPacker verwendet ungefähr die gleiche Layout' Strategie als Tk's Packer und Namen der Layout's Präferenzen spiegeln sozusagen ihre "Erbchaft" wieder. Die *new* Methode für FXPacker könnte es so aussehen:

```
aPacker = FXPacker.new(parent, opts=0,  
                        hfkhfkahkahjhkhkjx=0, y=0, h=0,  
                        hfkhfkahkahjhkhkjpl=DEFAULT_SPACING, pr=DEFAULT_SPACING,  
                        hfkhfkahkahjhkhkjpt=DEFAULT_SPACING, pb=DEFAULT_SPACING  
                        hfkhfkahkahjhkhkjhs=DEFAULT_SPACING, vs=DEFAULT_SPACING)
```

Es mag sein, dass Sie beim Ansehen dieses Codes den Eindruck gewinnen können, dass der eine sehr lange Argumentenliste hat. Bei näherer Prüfung sollten Sie erleichtert sein, weil alles außer den ersten Argument optional ist, das heisst, sie haben eigentlich "default" Werte. Genauso wenn Sie sich die *new* Methode für andere Widgets ansehen, werden Sie diesen Mustern wieder begegnen: eine lange Argumentenliste mit Grundwerten für die meisten Argumenten. Und eigentlich, meistens oder alle von diesen Argumenten können umgetauscht werden, nachdem der Widget schon erzeugt wurde, und dabei werden seine zusätzliche Methode verwendet, so dass man folgende Code schreiben kann:

```
aPacker = FXPacker.new(parent, LAYOUT_EXPLICIT, 0, 0, 150, 80)
```

und das gleichbedeutend mit diesen vier Zeilen des Codes:

```
aPacker = FXPacker.new(parent)hfkhfkahkhjh# annehmen der Grundwerten
```

```
aPacker.width = 150 # festsetzen der Breite(in pixel)
aPacker.height = 80 # festsetzen der Höhe(in pixel)
aPacker.layoutHints = LAYOUT_EXPLICIT # kontrolliert die Aktualität der Breite und Höhe
enforced!
```

Nun müssen wir mehr über die Bedeutung dieser Argumente sagen. Lassen wir uns kurz das zweite Argument(*opts*) stehen und betrachten wir restlichen Argumente. Die *x*, *y*, *w* und *h* sind ganze Zahlen, die auf die Position, wobei wird das "vaterliche" Koordinatensystem verwendet, und die Größe für den Widget hindeuten. Diese Argumente kann man ignorieren, wenn wir nicht auch die entsprechende Layoutshinweise festsetzen (*LAYOUT_FIX_X*, *LAYOUT_FIX_Y*, *LAYOUT_FIX_WIDTH* oder *LAYOUT_FIX_HEIGHT*). Diese Argumente zeigen sich in beinahe jeder *new* Methode eines Widgets, und sie sind übliche Argumente für *FXPacker.new* Widgets. Im Codebeispiel, den wir gerade besprochen haben, haben wir die "shortcut option" *LAYOUT_EXPLICIT* verwendet, die einfach die oben erwähnte Argumente vereinigt; das hat Sinn, weil man ein Layout mit festgelegten Positionen und Größe verwendet wird, alles, dass wir brauchen, diese Option zu setzen. Die nächsten vier Argumenten(*pl=DEFAULT...*) für *FXPacker.new* sind die innere Abstände links, rechts, oben und unten vom Rand in pixel. Wie schon erwähnt wurde, bezieht sich das auf den Extra-Raum, der um die innere Ränder des Behälters plziert wird, und wenn der Layout in ihrem Programm besonders aussehen sollte, kann man extra einen Raum an den Rändern verschaffen, um zu versuchen, die Werte vom Grundwert *DEFAULT_SPACING*(ein Konstantwert gleich 4 pixel) abweichen zu lassen. Die letzte zwei Argumente für den Widget deuten auf einen horizontalen und vertikallen Abstand in pixel hin, welcher zwischen den Kind-Widgets ist. Wie die oben erwähnte interne Abstände(padding), haben diese zwei Argumente Grundwerte jeweils vier pixel. Nun lassen wir uns zum zweiten Argument zurückkehren und zwar der Option *opts* Die meisten FOX's *new* Methoden verwenden diesen Wert. Damit werden verschiedene "bitartige-flags" ein-, oder ausgeschaltet. Diese "flags" beschreiben das Erscheinen und das Verhalten von Widgets. Wir haben schon bereits gesehen, dass manche von diesen "flags" Layouthinweise beinhalten, die Hinweise vom Kind-Widget zum seinen Vater-Widget, und auch darüber, wie es während der Layoutprozedur behandelt werden sollte. Zusätzlich zum *LAYOUT_FIX* Hinweis gibt's auch: *LAYOUT_SIDE_LEFT*, *LAYOUT_SIDE_RIGHT* und *LAYOUT_SIDE_TOP*, *LAYOUT_SIDE_BOTTOM* die deuten darauf hin, welcher(n) Seite(n) gegenüber die Kind-Widgets gepackt werden können;

[LAYOUT_FILL_X](#), [LAYOUT_CENTER_X](#) und [LAYOUT_FILL_Y](#), [LAYOUT_CENTER_Y](#) weisen darauf hin, wie der Kind-Widget sich verhalten würde, sich ausdehnen und damit den ganzen Raum füllen oder sich lediglich im Zentrum platzieren.

Es gibt noch zwei spezifischen Optionen oder sozusagen "packing styles": `PACK_UNIFORM_WIDTH` und `PACK_UNIFORM_HEIGHT`. Ähnlich wie "homogeneous" Eigenschaften des Ruby/GTK Layout Managers, erzwingen sozusagen diese zwei Optionen den Layout Manager für die Kind-Widgets die gleiche Breite oder, bzw. und Höhe. Diese zwei Optionen sind auf alle Kind-Widgets anwendbar und dabei überschreiben sie alle andere Präferenzen (einschließlich `LAYOUT_FIX_WIDTH` und `LAYOUT_FIX_HEIGHT`). Diese Optionen sind mehr für die andereene Layout Manager geeignet, die von `FXPacker` abgeleitet sind, aber man kann sie trotzdem generell verwenden, wenn man weiss, was man tut.

Nun betrachten wir die nächste Layout Manager und zwar [FXHorizontalFrame](#) und [FXVerticalFrame](#) zusammen, weil sie ähnlich sind. Wie Sie schon vermuten, ordnen diese beide ihrer Kind-Widgets in der horizontaler, bzw. vertikaler Richtung an. Die `new` Methode für [FXHorizontalFrame](#) sieht so aus:

```
aHorizFrame = FXHorizontalFrame.new(parent, opts=0,  
enjoyyourselfenjoyyourselfenjoyyourselfx=0, y=0, w=0, h=0,  
enjoyyourselfenjoyyourselfenjoyyourselfpl=DEFAULT_SPACING,  
enjoyyourselfenjoyyourselfenjoyyourselfpr=DEFAULT_SPACING,  
enjoyyourselfenjoyyourselfenjoyyourselfpt=DEFAULT_SPACING,  
enjoyyourselfenjoyyourselfenjoyyourselfpb=DEFAULT_SPACING,  
enjoyyourselfenjoyyourselfenjoyyourselfhs=DEFAULT_SPACING,  
enjoyyourselfenjoyyourselfenjoyyourselfvs=DEFAULT_SPACING)
```

Die Grundeinstellung für diesen Layout Manager sieht es vor, die Kind-Widgets horizontal von links nach rechts anzuordnen, in dieser Reihenfolge werden sie hinzugefügt. Wenn es erforderlich ist, dass ein bestimmter Kind-Widget an der rechten Seite des Raumes angeordnet werden soll, gibt man einfach den [LAYOUT_RIGHT](#) Layoutshinweis über. Vertikale Frames ordnen ihre Kind-Widgets von oben nach unten und zwar "defaultmäßig, mit dem [LAYOUT_BOTTOM](#) Hinweis kann man dieses Vorgehen verändern. Der

letzte Layout Manager, den wir besprechen, ist `FXMatrix`, welcher seine Kind-Widgets in Zeilen und Spalten anordnet. Die `new` Methode für `FXMatrix` ist:

```
aMatrix = FXMatrix.new(parent, size=1, opts=0,  
enjoyyourselfenjoyyourself x=0, y=0, w=0, h=0,  
enjoyyourselfenjoyyourself pl=DEFAULT_SPACING, pr=DEFAULT_SPACING,  
enjoyyourselfenjoyyourself pt=DEFAULT_SPACING, pb=DEFAULT_SPACING,  
enjoyyourselfenjoyyourself hs=DEFAULT_SPACING, vs=DEFAULT_SPACING)
```

Der Widget `FXMatrix` hat zwei wichtige Optionen `MATRIX_BY_ROWS` und `MATRIX_BY_COLUMNS`. Sie weisen darauf hin, wie das `size` Argument für den `FXMatrix.new` interpretiert werden kann. Für die `MATRIX_BY_ROWS`, das ist die Grundeinstellung, zeigt die `size` die Anzahl der Zeilen; die Anzahl der Spalten ist schließlich die gesamte Anzahl der Kind-Widgets für den Matrix. In unserem Beispiel bedeutet es, dass der erste Kind-Widget zur ersten Reihe hinzugefügt wird, der zweite zur zweiten Reihe und so weiter, wobei alle Kind-Widgets sich in einer Spalte befinden. In anderem Fall wird die Option `MATRIX_BY_COLUMNS` verwendet, wobei das Argument `size` die Anzahl der Spalten zeigt und die Anzahl der Reihen wird unterschiedlich.

FOX Beispielprogramm

Der folgende Code zeigt den kompletten Sourcecode von unserem Beispielprogramm. Den können Sie auch herunterladen [fox-xmlviewer](#)

```
#!/bin/env ruby -w  
  
require "fox"  
require "fox/responder"  
  
require "nqxml/treeparser"  
include Fox
```

```

class XMLViewer < FXMainWindow

  include Responder

  # Define message identifiers for this class
  ID_ABOUT, ID_OPEN, ID_TREELIST =
  enum(FXMainWindow::ID_LAST, 3)

  def createMenubar
    menubar = FXMenubar.new(self, LAYOUT_SIDE_TOP|LAYOUT_FILL_X)
    filemenu = FXMenuPane.new(self)
    FXMenuTitle.new(menubar, "&Datei", nil, filemenu)
    FXMenuCommand.new(filemenu,
      you"&Öffnen...\tCtl-O\tOpen document file.", nil, self, ID_OPEN)
    FXMenuCommand.new(filemenu,
      you"&Beenden\tCtl-Q\tQuit the application.", nil,
      youcgetApp(), FXApp::ID_QUIT, 0)

    helpmenu = FXMenuPane.new(self)
    FXMenuTitle.new(menubar, "&Hilfe", nil, helpmenu, LAYOUT_RIGHT)
    FXMenuCommand.new(helpmenu,
      you"&Über FOX...\t\tDisplay FOX about panel.",
      youcnil, self, ID_ABOUT, 0)
  end

  def createTreeList
    listFrame = FXVerticalFrame.new(@splitter,
      youLAYOUT_FILL_X|LAYOUT_FILL_Y|FRAME_SUNKEN|FRAME_THICK)
    @treeList = FXTreeList.new(listFrame, 0, self, ID_TREELIST,
      you(LAYOUT_FILL_X|LAYOUT_FILL_Y|
      youTREELIST_SHOWS_LINES|TREELIST_SHOWS_BOXES|TREELIST_ROOT_BOXES))
  end
end

```

```
enjoyend

enjoydef createAttributesTable
enjoy    tableFrame = FXVerticalFrame.new(@splitter,
enjoy        LAYOUT_FILL_X|LAYOUT_FILL_Y|FRAME_SUNKEN|FRAME_THICK)
enjoy    @attributesTable = FXTable.new(tableFrame, 5, 2, nil, 0,
enjoy        (TABLE_COL_SIZABLE|TABLE_ROW_SIZABLE|
enjoy        FRAME_SUNKEN|FRAME_THICK|LAYOUT_FILL_X|LAYOUT_FILL_Y))
enjoyend

enjoydef initialize(app)
enjoy    # Initialize base class first
enjoy    super(app, "XML Editor", nil, nil, DECOR_ALL, 0, 0, 800, 600)

enjoy    # Set up the message map
enjoy    FXMAPFUNC(SEL_COMMAND, ID_ABOUT, "onCmdAbout")
enjoy    FXMAPFUNC(SEL_COMMAND, ID_OPEN, "onCmdOpen")
enjoy    FXMAPFUNC(SEL_COMMAND, ID_TREELIST, "onCmdTreeList")

enjoy    # Create the menu bar
enjoy    createMenubar

enjoy    @splitter = FXSplitter.new(self, LAYOUT_FILL_X|LAYOUT_FILL_Y)

enjoy    # Create the tree list on the left
enjoy    createTreeList

enjoy    # Attributes table on the right
enjoy    createAttributesTable

enjoy    # Make a tool tip
enjoy    FXTooltip.new(getApp(), 0)
enjoyend
```

```
en|# Create and show the main window
en|def create
en|enjoysuper
en|enjoyshow(PLACEMENT_SCREEN)
en|end

en|def loadDocument(filename)
en|enjoy@document = nil
en|enjoybegin
en|enjoyyou@document = NQXML::TreeParser.new(File.new(filename)).document
en|enjoyrescue NQXML::ParserError => ex
en|enjoyyouFXMessageBox.error(self, MBOX_OK, "Error",
en|enjoyyou"Couldn't parse XML document")
en|enjoyend
en|enjoyif @document
en|enjoyyou@treeList.clearItems()
en|enjoyyoupopulateTreeList(@document.rootNode, nil)
en|enjoyend
en|end

en|def populateTreeList(docRootNode, treeRootNode)
en|enjoyentity = docRootNode.entity
en|enjoyif entity.instance_of?(NQXML::Tag)
en|enjoyyoutreeItem = @treeList.addItemLast(treeRootNode, entity.to_s, nil, nil, entity)
en|enjoyyoudocRootNode.children.each do |node|
en|enjoyyoupopulateTreeList(node, treeItem)
en|enjoyyouend
en|enjoyelsif entity.instance_of?(NQXML::Text) && entity.to_s.strip.length != 0
en|enjoyyoutreeItem = @treeList.addItemLast(treeRootNode, entity.to_s, nil, nil, entity)
en|enjoyend
```

```
enjoyend

enjoydef onCmdOpen(sender, sel, ptr)
enjoydlg = FXFileDialog.new(self, "Öffnen")
enjoydlg.setPatternList([
enjoyyyo"Alle Dateien (*)",
enjoyyyo"XML Documents (*.xml)"])
enjoyif dlg.execute() != 0
enjoyyoloadDocument(dlg.getFilename())
enjoyend

enjoyreturn 1
enjoyend

enjoydef onCmdTreeList(sender, sel, treeItem)
enjoyif treeItem
enjoyyocentity = treeItem.getData()
enjoyyyocif entity.kind_of?(NOXML::NamedAttributes)
enjoyyoukeys = entity.attrs.keys.sort
enjoyyou@attributesTable.setTableSize(keys.length, 2)
enjoyyoukeys.each_index { |row|
enjoyyous@attributesTable.setItemText(row, 0, keys[row])
enjoyyous@attributesTable.setItemText(row, 1, entity.attrs[keys[row]])
enjoyyou}
enjoyyocend
enjoyend

enjoyreturn 1
enjoyend

enjoy# About box
enjoydef onCmdAbout(sender, sel, ptr)
enjoyFXMessageBox.information(self, MBOX_OK, "Über XMLViewer",
```

```
enjoyyo "FXRuby Sample Application/Beispielprogramm")
enjoyreturn 1
enjoyend
enjoyend

enjoyif $0 == __FILE__
enjoy# Make application
enjoyapplication = FXApp.new("XMLViewer", "FoxTest")

enjoy# Open the display
enjoyapplication.init(ARGV)

enjoy# Make window
enjoymainWindow = XMLViewer.new(application)

enjoy# Create the application windows
enjoyapplication.create

enjoy# Run the application
enjoyapplication.run
enjoyend
```

Die FXRuby Version unseres Programm beginnt mit einer Einbindung der Module *fox* und *fox/responder*, welche der FOX Hauptbibliothek entsprechen und außerdem wird dadurch erreicht, dass die Widgets ihre Nachricht-Handler Methoden sozusagen bei der Bibliothek registrieren können:

```
require "fox"
require "fox/responder"

require "nqxml/treeparser"
include Fox
```

```
class XMLViewer < FXMainWindow

  include Responder

  # Define message identifiers for this class
  ID_ABOUT, ID_OPEN, ID_TREELIST =
  enum(FXMainWindow::ID_LAST, 3)
```

Wie man schon sieht, brauchen wir eine Hauptfensterklasse(XMLViewer) um auf drei Nachricht-ID's zu reagieren: *ID_ABOUT*, welche mit dem **Über...** Menübefehl zusammenhängt; *ID_OPEN*, welche mit dem **Öffnen** Menübefehl in der Beziehung steht; und *ID_TREELIST*, welche verwendet wird, wenn man ein Eintrag in der Liste auswählt. Um Nachricht-Handler Funktionen für diese Klasse anzumelden, muss man auch mix-in *Responder* Module einbinden. Die *enum* Funktion ist sozusagen "Helfer", die von diesem Modul bereitgestellt wird und das baut einfach ein Array mit ganzen Zahlen auf. Der Array beginnt mit seinem ersten Input(FXMainWindow) und als zweites Argument wird die Anzahl der Nachricht-ID's übergeben. In unserem Fall stellen wir sicher, dass die Argumente für die *enum* Funktion mit *FXMainWindow::ID_LAST* beginnt, damit erreichen wir, dass es keine gegenseitige Störungen mit anderen FXMainWindow's Nachricht-ID's gibt. Und das ist eigentlich ein Standard-Vorgang für FXRuby Applikationen. Der erste Schritt in *initialize* Methode ist das Deklarieren der Basisklasse(FXMainWindow):

```
super(app, "XML Editor", nil, nil, DECOR_ALL, 0, 0, 800, 600)
```

Dabei ist es wichtig, diesen Schritt nicht zu unterlassen. Hier ist das zweite Argument zu *FXMainWindow* initialize Methode der Titel des Fensters("XML Editor"). Das fünfte Argument ist eine Reihe von Optionen, die Hinweise für Fenster's Manager beinhalten, welche Fenster's Dekorationen(z.B. ein Titelménü oder eine Handhabung der Fenstergröße) gezeigt werden könnten. In unserem Fall wünschen wir uns alle mögliche Dekorationen(*DECOR_ALL*). Die letzte zwei Argumente setzen die anfängliche Breite und Höhe für Hauptfenster fest. Der nächste Schritt ist die Zuordnung einer Nachrichten-ID zu einer Methode:

```
FXMAPFUNC(SEL_COMMAND, ID_ABOUT, enj "onCmdAbout")
```

```
◦FXMAPFUNC(SEL_COMMAND, ID_OPEN, enjoy "onCmdOpen")
◦FXMAPFUNC(SEL_COMMAND, ID_TREELIST, "onCmdTreeList")
```

Die *FXMAPFUNC* Methode ist ein "mixed-in" aus dem *Responder* Modul. Die Methode hat drei Argumente und zwar Nachrichten-Typ, Nachrichten-ID und den Methodennamen. Der erste Aufruf, zum Beispiel, gibt es bekannt, wenn *XMLViewer* Objekt vom Nachrichtentyp *SEL_COMMAND* gefragt wird, wobei die Nachrichten-ID *XMLViewer::ID_ABOUT* verwendet wird, dass die *onCmdAbout* Methode aufgerufen wird. Der Rest der *initialize* Methode legt den Inhalt und den Layout des Hauptfensters fest. Die erste sozusagen interessante Stelle in der Applikation kommt, wenn die Menübar des Programmes erzeugt wird. Dafür haben wir die *createMenubar* Methode verwendet:

```
enjoy createMenubar
enjoy menubar = FXMenubar.new(self, LAYOUT_SIDE_TOP|LAYOUT_FILL_X)
enjoy filemenu = FXMenuPane.new(self)
enjoy FXMenuTitle.new(menubar, "&Datei", nil, filemenu)
enjoy FXMenuCommand.new(filemenu,
enjoy you "&Öffnen...\tCtl-O\tOpen document file.", nil, self, ID_OPEN)
enjoy FXMenuCommand.new(filemenu,
enjoy you "&Beenden\tCtl-Q\tQuit the application.", nil,
enjoy you getApp(), FXApp::ID_QUIT, 0)

enjoy helpmenu = FXMenuPane.new(self)
enjoy FXMenuTitle.new(menubar, "&Hilfe", nil, helpmenu, LAYOUT_RIGHT)
enjoy FXMenuCommand.new(helpmenu,
enjoy you "&Über FOX...\t\tDisplay FOX about panel.",
enjoy you nil, self, ID_ABOUT, 0)
enjoy end
```

Ein "FXMenubar" wird als ein horizontalorientierender Behälter, so dass er ein oder mehrere "FXMenuTitle"- Widgets enthalten kann. "FXMenuTitle" hat als Argument ein Textstring, der als ein Name

für den Menütitel benutzt wird (wie "Datei") und außerdem wird beim Anklicken des Menütitels ein Pop-up-Fenster("FXMenuPane")eröffnet, welches ein oder mehrere "FXMenuCommand" widgets enthält. Der Textstring für den Menütitel kann vorne das "Ampersandzeichen("&"); enthalten, wenn das vorhanden ist, wird die erste Buchstabe unterstrichen, was den FOX veranlasst, den "Keyboard-Beschleuniger" einsetzen, um das Menü mit der Tastekombination zu aktivieren. Zum Beispiel hat "FXMenuTitle" für das **Datei** Menü:

```
FXMenuTitle.new(menuubar, "&Datei", nil, filemenu)
```

den Textstring "Datei" mit "D" unterstrichen, und wenn man die **Alt+F** Tastekombination verwendet, wird das Menü "Datei" benutzt. So ähnlich kann der Textstring für Menücommands spezielle Kontrollzeichen beinhalten:

```
FXMenuCommand.new(filemenu,  
enjoyyou"&Öffnen...\tCtl-O\tOpen document file.", nil, self, ID_OPEN)
```

Das "Ampersandzeichen" vor der Buchstabe "Ö" in "Öffnen" legt "hot key" für den Menübefehl; wenn das **Datei** Menü schon geöffnet ist, kann man **Ö** Taste drücken um den "Öffnen..." Befehl zu aktivieren. Die Tabulatorzeichen("\t") werden bei FOX als Feldseparatoren erkannt. Das erste Feld ist der Haupttext, welcher in "FXMenuCommand" Widget gezeigt wird. Das zweite Feld ist ein optionaler String, der auf die Tastekombination hindeutet, die verwendet kann, um direkt auf den Befehl zugreifen, nur aber wenn das Menü schon geöffnet wurde. Und das letzte Feld ist auch optional, welches ein Textstring enthält, der uns einfach erklärt, was dahinter steckt.

Obwohl wir in unserem Beispiel keine Separatoren verwenden, ist es oft hilfreich, ein oder mehrere horizontalen Separatoren zu "Pull-down-Menü hinzuzufügen. Damit kann man zusammenhängende Menübefehle gruppieren. Um ein Separator zu "FXMenuPane" hinzuzufügen, erzeugt man einfach eine Instanz von der *FXMenuSeparator* Klasse in der gewünschten Position:

```
FXMenuSeparator.new(filemenu)
```

Die linke Seite des Hauptfensters unterbringt eine baumstrukturierte Liste von XML Dokumentknoten; das wird durch die `createTreeList` erzeugt:

```
def createTreeList
  enjoylistFrame = FXVerticalFrame.new(@splitter,
  enjoyyou LAYOUT_FILL_X|LAYOUT_FILL_Y|FRAME_SUNKEN|FRAME_THICK)
  enjoy@treeList = FXTreeList.new(listFrame, 0, self, ID_TREELIST,
  enjoyyou (LAYOUT_FILL_X|LAYOUT_FILL_Y|
  enjoyyou TREELIST_SHOWS_LINES|TREELIST_SHOWS_BOXES|TREELIST_ROOT_BOXES))
  enjoyend
```

Da die `FXTreeList` Klasse nicht von `FXFrame` abgeleitet wird, hat man hier keine Framestyle's Optionen gesetzt. Um nun jetzt gut aussehende tiefliegende Kanten zu bekommen, müssen wir den `FXTreeList` Widget sozusagen umgeben, bzw. in ein von `FXFrame`-abgeleiteter Kind-Widget packen; hier werden wir `FXVerticalFrame` verwenden. Die dritte und vierte Argumente von `new` Methode für den `FXTreeList` Widget stellen das Hauptfenster(`self`) als Nachrichtenziel und auch deren Nachrichten-ID und zwar `XMLViewer::ID_TREELIST`. Wie Sie sich schon bestimmt erinnern, haben wir in der `initialize` Methode der Nachrichten-ID `ID_TREELIST` eine `onCmdTreeList` Methode zugeordnet, also wenn `SEL_COMMAND` Nachricht mit dieser Nachrichten-ID gesendet wird, veranlasst es, dass die `onCmdTreeList` aufgerufen wird:

```
enjoydef onCmdTreeList(sender, sel, treeItem)
  enjoyif treeItem
  enjoyyou entity = treeItem.getData()
  enjoyyou if entity.kind_of?(NOXML::NamedAttributes)
  enjoyyou keys = entity.attrs.keys.sort
  enjoyyou @attributesTable.setTableSize(keys.length, 2)
  enjoyyou keys.each_index { |row|
```

```

enjoyyous@attributesTable.setItemText(row, 0, keys[row])
enjoyyous@attributesTable.setItemText(row, 1, entity.attrs[keys[row]])
enjoyyou}
enjoyycend
enjoyend
enjoyreturn 1
enjoyend

```

Wenn der *FXTreeList* Widget den *SEL_COMMAND* Befehl zu seinem Nachrichtenziel sendet, ist die Nachricht (das dritte Argument, das an die "message handler methode" weitergeleitet wird) ein Verweis auf den ausgewählten Eintrag in der XML-Datei, wenn überhaupt. Angenommen, dass der ausgewählte Eintrag (als *treeItem* genannt) nicht *nil* ist, dann empfangen wir einen Verweis, der auf Angaben, die mit diesem Eintrag verbunden sind, hindeutet. Wie Sie später sehen werden, wenn wir die *populateTreeList* Methode besprechen werden, durch die die Einträge der XML-Datei dargestellt werden, also Angaben, die der User vornimmt, sind mit den Einträgen verbunden, die ihrerseits die XML-Entities repräsentieren. Wenn ein Eintrag mit ihm verbundene Attribute hat, modifizieren wir die Anzahl der Tabellenzeilen, erreichen wir das, wenn man die *setTableSize* Methode aufrufen und danach übergeben wir alle vorhandene Attribute, um den Inhalt in Tabellenzellen auf einen neuen Zustand zu bringen.

Die mit den Attributen gefüllte Tabelle auf der rechten Seite des Hauptfensters wird durch die *createAttributesTable* Methode erzeugt und besteht aus *FXTable* Widget, welcher ebenfalls in *FXVerticalFrame* Widget eingeschlossen wird:

```

enjoydef createAttributesTable
enjoytableFrame = FXVerticalFrame.new(@splitter,
enjoyyu LAYOUT_FILL_X|LAYOUT_FILL_Y|FRAME_SUNKEN|FRAME_THICK)
enjoy@attributesTable = FXTable.new(tableFrame, 5, 2, nil, 0,
enjoyyu (TABLE_COL_SIZABLE|TABLE_ROW_SIZABLE|
enjoyyu FRAME_SUNKEN|FRAME_THICK|LAYOUT_FILL_X|LAYOUT_FILL_Y))
enjoyend

```

Wie Sie sich schon erinnern können, haben einige von anderen GUI Toolkits den Namen *table* nur dafür, um an den Layout-Manager zu verweisen, der seine Kinder in Reihen und Spalten anordnet (was dafür FOX den *FXMatrix* Layout-Manager aufruft). Für FOX ist *FXTable* mehr als ein Tabellenkalkulationsähnlicher Widget. Wir haben anfangs den *FXTable* Widget mit 5 sichtbaren Zeilen und 2 sichtbaren Spalten erzeugt, obwohl, wir werden es später sehen, die Tabellengröße sich dynamisch ändern kann, während das Programm läuft. Die *onCmdOpen* Methode behandelt die *SEL_COMMAND* Nachricht, die erzeugt wird, sobald der User auf den **Öffnen...** Menübefehl aus dem **Datei** Menü klickt:

```
enjoydef onCmdOpen(sender, sel, ptr)
  enjoydlg = FXFileDialog.new(self, "Öffnen")
  enjoydlg.setPatternList([
    enjoyyo "Alle Dateien (*)",
    enjoyyo "XML Documents (*.xml)"])
  enjoyif dlg.execute() != 0
    enjoyyc loadDocument(dlg.getFilename())
  enjoyend
  enjoyreturn 1
enjoyend
```

In dieser Methode wird ein neues *FXFileDialog* Objekt aufgebaut, dann wird seine Musterliste initialisiert und anschließend wird durch den Aufruf der *execute* Methode ein Dialogfenster angezeigt. Die *execute* Methode gibt "nicht-null" zurück, wenn der User den **OK** Button drückt, und das der Fall ist, werden wir nach einem Dateinamen gefragt, den wir entweder auswählen oder eingeben müssen. Daraufhin wird die *loadDocument* Methode aufgerufen, die eine XML-Datei laden kann:

```
enjoydef loadDocument(filename)
  enjoy@document = nil
  enjoybegin
  enjoyyu @document = NOXML::TreeParser.new(File.new(filename)).document
```

```

enjoyrescue NOXML::ParserError => ex
enjoyyou FXMessageBox.error(self, MBOX_OK, "Error",
enjoyyou "Couldn't parse XML document")
enjoyend
enjoyif @document
enjoyyou @treeList.clearItems()
enjoyyou populateTreeList(@document.rootNode, nil)
enjoyend
enjoyend

```

Wenn der XML Parser eine Ausnahme auslöst, während der Versuchung ein *NOXML::Document* Objekt zu erzeugen, rufen wir eine *FXMessageBox.error* Singleton Methode auf, um ein einfaches Dialogbox zu zeigen, die dem User erklärt, was geschehen ist. Das erste Argument von *FXMessageBox.error* identifiziert sozusagen den Besitzer der Dialogbox, andersgesagt wem die Dialogbox gehört. Die Box schwebt sozusagen über dem Besitzer, bis die abgewiesen wird. Das zweite Argument ist ein "flag", der darauf hinweist, welche Art von "Ende-Buttons" in dieser Dialogbox erzeugt werden könnte; es gibt dafür auch andere Optionen und zwar *MBOX_OK_CANCEL*, *MBOX_YES_NO*, *MBOX_YES_NO_CANCEL*, *MBOX_QUIT_CANCEL* und *MBOX_QUIT_SAVE_CANCEL*. Die *FXMessageBox* Klasse unterstützt ein paar anderen praktischen singleton Methoden, durch die nützliche Nachrichten wie *information*, *question* und *warning* angezeigt werden könnten. Wenn es keine Fehlermeldungen gibt, löschen wir den alten Inhalt der Liste und danach bauen durch den wiederholenden Aufruf der *populateTreeList* Methode neuen Listeninhalt auf:

```

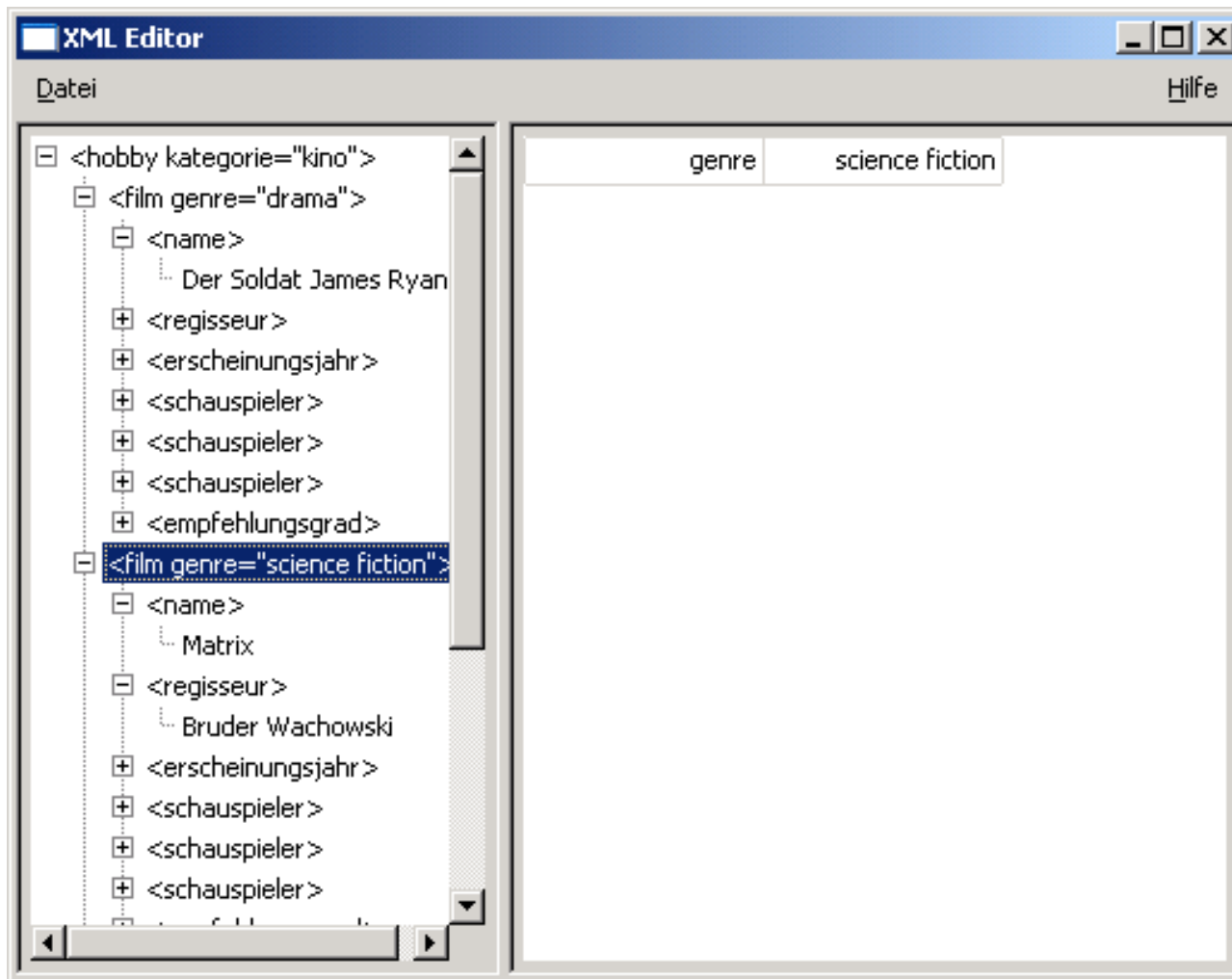
enjoydef populateTreeList(docRootNode, treeRootNode)
enjoyentity = docRootNode.entity
enjoyif entity.instance_of?(NOXML::Tag)
enjoyyou treeItem = @treeList.addItemLast(treeRootNode, entity.to_s, nil, nil, entity)
enjoyyou docRootNode.children.each do |node|
enjoyyou populateTreeList(node, treeItem)

```

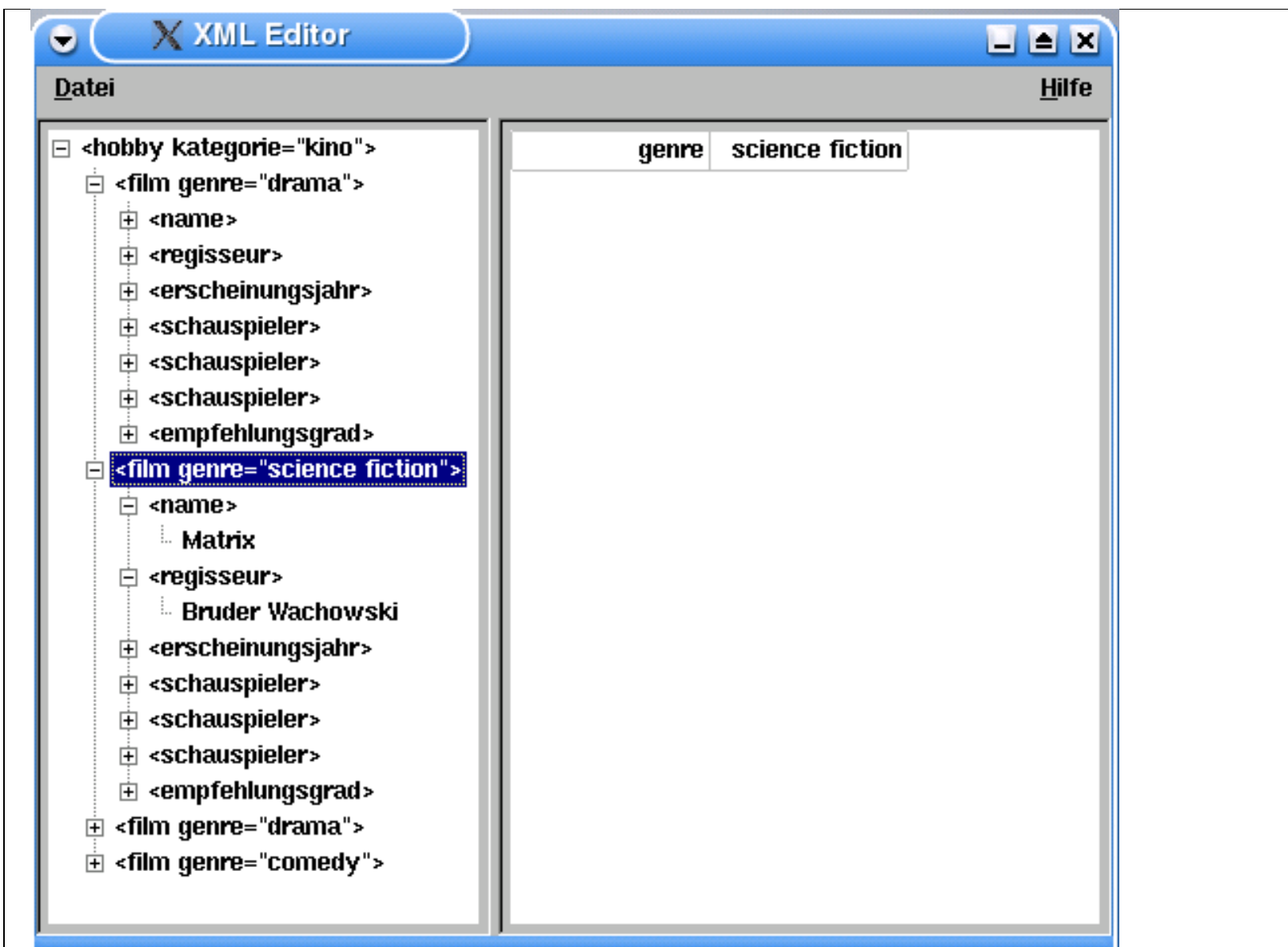
```
enjoyycend
enjoyelsif entity.instance_of?(NOXML::Text) && entity.to_s.strip.length != 0
enjoyyc treeItem = @treeList.addItemLast(treeRootNode, entity.to_s, nil, nil, entity)
enjoyend
enjoyend<
```

Hier verwenden wir eine `addItemLast` Methode, die zum `FXTreeList` Widget gehört, damit wird es erreicht, dass neue Listenenträge hinzugefügt werden. Das erste Argument von der `addItemLast` Methode ist ein Verweis auf einen Listeneintrag, es wird sozusagen ein "Vater" für diesen Eintrag erzeugt; die Listeneinträge in "top-level" erzeugt, man kann stattdessen `nil` für dieses Argument übergeben. Das zweite Argument für diese Methode ist ein Textstring, der angezeigt wird, wenn man auf den Eintrag klickt, das dritte und vierte Argumente sind optional und das sind "plus" Icon für "öffnen" und "minus" Icon für "schließen". Die werden vor dem Textstring des Listeneintrages angezeigt. Wenn es keine Icons angeboten werden, wird der `FXTreeList` Widget anstatt der "plus" und "minus" Icons eine standartmäßige viereckige Icon darstellen. Das letzte Argument der `addItemLast` Methode ist eine optionale User's Angabe; das ist ein Ruby Objekt, welches mit dem frisch erzeugten XML-Eintrag verbunden ist. In unserem Fall speichern wir hier einen Verweis auf den XML-Entity, welche durch diesen Eintrag repräsentiert wird.

Nun ist die Zeit zum Ausprobieren. Die unten ausgeführte Abbildung lief auf Windows OS:



Und das ist die Abbildung des Programms, das auf Linux OS ausprobiert war:



Anschließend noch Mal zur Erinnerung, dass man den aktuellen Stand FXRuby Projektes auf der jeweiligen Homeseite erfahren kann.

Verwendung SWin/VRuby Erweiterungen

SWin und VRuby sind zwei Grundbestandteilen des VisualuRuby's Projektes. SWin ist ein Ruby-Erweiterungsmodul, geschrieben in C, welches vielen von Win32 API zu den Ruby-Interpreter bereitstellt. VRuby ist eine Bibliothek, die aus reinen Rubymodulen besteht, bildet sich auf SWin auf, so dass es eine Schnittstelle auf höchster Ebene für Win32-Programmierung zur Verfügung stellt. Wie es der Fall mit anderen Ruby-Erweiterungen ist, gibt es meistens die Dokumentaion für SWin und VRuby nur in Japanisch. Auf der offiziellen Seite des VisualuRuby Proejktes kann man sich Beispielprogramme besorgen, die man als Musterprogramme für seine Ideen ausprobieren und was eigentlich mit SWin und VRuby machen kann, und dadurch können Sie sich schon mit allgemeinen Windowsprogrammierungsvorstellungen vertraut machen. Dieser Abschnitt liefert genug Information der VRuby(API) Schnittstelle um unseres Beispielprogramm zu entwickeln, aber für die tieferen Kenntnissen in SWin/VRuby müssen sich gut genug mit den Hintergründen der Windowsprogrammierung auskennen. Es gibt einige hervorragende Bücher in diesem Gebiet, und wenn auch sie aus dem Blinkwinkel der C Programmierung Win32 API besprechen, könnten sie durchaus gut einige Lücken des nötigen Wissens mit VRuby Programmierung auffüllen. Außerdem gibt es auch auf anderer Seite eine sehr nützbare Information der Win32 API Referenz auf der Microsoft Developer Network Seite auch als CD's erhältlich.

Installation

SWin und VRuby Module sind im Ruby-Installer von der "Pragmatic Programmers" Webseite enthalten. Also heisst es, dass wenn man sich Ruby installiert hat(es handelt sich natürlich nur um Windows OS), kann man sich loslegen. Man kann sich auch Sourcecode für SWin von der VisualuRuby Projekt Seite herunterladen, und kann auch so gut schon precompilierte Binäres finden.

VRuby Bibliothek Basis

Da es so wenig Dokumentaion in Englisch für SWin und VRuby gibt, werden wir an dieser Stelle zusätzliche Information liefern, die Ihnen einen Überblick von der VRuby Bibliothek und ihren Klassen und Modulen verschafft. Um mehr Information darüber herauszubekommen, wäre es hilfreich, Kommentare von der Sourcedateien zu studieren, welche selbst allerdings ziemlich skizzenhaft sind, aber das könnte Ihnen helfen, anzufangen, wenn Sie beabsichtigen, mit VRuby zu arbeiten. Die VRuby Bibliothek ist als eine Reihe der Ruby Sourcedateien aufgebaut, und wird ins [vr](#) Package-Verzeichnis installiert. Die unten dargestellte Tabelle 1 listet einige der meistens wichtigen Dateien, die Sie in ihren Programmen verwenden werden, und entsprechend [require](#) Statement. Dabei ist es, anzumerken, dass die meiste oder alle der nicht sozusagen "Kern-Dateien" von der Kern-Klassen und Modulen aus [vr/vruby.rb](#) abhängig sind, so dass es wahrscheinlich nicht notwendig ist, diese Dateien zusätzlich zu importieren.

Tabelle 1 Inhalt der VRuby Bibliothek

Beschreibung	Wie wird es importiert
Kern-Klasse und Module	require 'vr/vruby'
Layout Manager	require 'vr/vrlayout'
Standard Steuerung	require 'vr/vrcontrol'
Allgemeine Steuerung	require 'vr/vrcomctl'
Multipane Windows	require 'vr/vrtwopane'

Die Kern-Klasse und Module für VRuby(Tabelle 2) besteht aus allein "High-Level" Basisklassen und "mixin"-Modulen, die Sie aber nicht direkt in meisten Programmen verwenden werden. Es gibt aber zwei bemerkenswerte Ausnahmen, wie auch immer, [VRForm](#) und [VRScreen](#). [VRForm](#) ist die Basisklasse, die für die "top-level" Fenster verwendet wird, solche, wie in ihren Programmen Hauptfenster sind. Wie Sie in unserem Beispielprogramm sehen werden, muss man nur lediglich unseres Hauptfenster von der [VRForm](#) Klasse ableiten, eventuell auch eine andere Verhalten des Fensters bringen, und werden Sie es kennenlernen, wie in dieser Klasse durch die [construct](#) Methode nach ihren Bedürfnissen das Verhalten und das Aussehen der Applikation handhaben wird.

Tabelle 2 Kern-Klasse und Module für VRuby

Name von Klasse/Modul	Beschreibung
VRMessageHandler	Dieses Modul könnte in irgendwelchen Klassen verwendet werden, die Windows Nachrichten empfangen müssen; das stellt acceptEvents und addHandler zur Verfügung
VRParent	Dieses Modul liefert die Modulmethode, die von dem Vater-Fenster verwendet wird(das Fenster kann als ein Behälter für die anderen Kind-Fenster sein), passt gut unter anderem mit VRForm und VRPanel Klassen zusammen, weil sie die meist verwendete "Behälterklasse", die man am häufigsten verwendet wird
VRWinComponent	Das ist die Basisklasse von allen Fenster
VRControl	Basisklasse für alle Controlwidgets; eine Subklasse von VRWinComponent
VRCommonDialog	Dieses Modul bietet mehrere passende Funktionen an, die für allgemeinen Fensterdialogen verwendet werden; liefert Modulmethoden wie openFilenameDialog , saveFilenameDialog , chooseColorDialog und chooseFontDialog
VRForm	Das ist Basisklasse für alle Hauptfenster, wie das Hauptfenster von unseren Beispielprogramm, und kann auch mit VRMessageHandler , VRParent und VRCommonDialog Modulen verwendet werden
VRPanel	Das ist die Basisklasse für alle Kind-Fenster(wie ControlWidgets); eine Subklasse von VRControl
VRDrawable	Dieses Modul kann mit einigen Klassen verwendet werden, die man braucht, um "Bild-Nachrichten" handzuhaben(ein Verweis, dass die Fensterinhalte rekonstruiert werden); diese Klasse kann die self_resize Methode außer Kraft setzen

VRResizeable	Dieses Modul kann in irgendwelchen Klassen benutzt werden, um Ereignisse abzufangen, die die Änderung der Fenstergröße handhaben; diese Klasse kann die <i>self_resize</i> Methode außer Kraft setzen
VRUserMessageUseable	Dieses Modul kann in irgendeiner Klasse verwendet werden, um Benutzerdefinierte Nachrichten mit dem Betriebssystem zu passen
VRScreen	Diese Klasse stellt die Fensteroberfläche dar, und bietet Methoden an, um neue Hauptfenster-Widgets zu erzeugen und zu zeigen, und dazu kann dies die Hauptereignisschleife zu steuern. VRuby legt genau eine globale Instanz dieser Klasse, VRLocalScreen

Einige der Window's Steuerungen existierten seit der Zeiten, als noch Windows 95 noch nicht zur Welt geblickt hat, was aber, wie man schon weiß, nicht verhindern konnte, dass diese Steuerungen sich als *standard* Steuerungen etabliert haben. Die Standardsteuerungen(wie Buttons, Labels, Textfelder) sind zugänglich, wenn man **vr/vrcontrol.rb** in Anspruch nimmt. Die nachstehende Tabelle 3 listet die Klassen, die dieses Modul präsentieren.

Tabelle 3 Standard Steuerelemente für VRuby

Klassen/Modul Name	Beschreibung
VRStdControl	Diese Klasse(eine Subklasse von VRControl ist einfach die Basisklasse von allen Standard Steuerungselementen
VRButton	Das ist ein Standard Drück-Button; Man kann auch den Button's Text zuweisen, dabei wird eine <i>caption</i> Instanz-Methode verwendet
VRGroupBox	Die Gruppenbox
VRCheckbox	Das ist eine Checkbox Steuerungseinheit. Man kann ein sogenanntes Häkchen setzen oder das entfernen, gemacht wird das durch die <i>check</i> Instanz-Methode, um den aktuellen Zustand zu prüfen, verwendet man die <i>checked?</i> Instanz-Methode

VRRadioButton	Das ist ein Radio-Button, gewöhnlich wird es als eine Gruppe von Radio-Buttons angezeigt, wird verwendet, wenn nur eine Auswahl getroffen werden muß. Da es besonderer Fall(eine Subklasse) der VRCheckbox ist, wird auch hier die <i>check</i> und <i>checked?</i> Instanz-Methoden unterstützt
VRStatic	Das ist ein statischer Text-Steuerungselement, welches einfach eine nicht editierbare Textbeschriftung anzeigt. Der Beschriftungstext kann durch die <i>caption</i> Instanz-Methode zugewiesen werden
VREdit	Das ist einzeilige editierbare Text-Steuerungseinheit. Die liefert eine Zahl von Instanz-Methoden zusammenhängend mit dem Editieren und Eingeben vom Text, und sowie die Bearbeitung des ausgewählten Textes und auch die Übertragung von einem Text zwischen der VREdit und Windows Clipboard(Zwischenspeicher)
VRText	Das ist vielzeilige editierbare Text-Steuerungseinheit. Sie ist ähnlich wie einzeilige Darstellung (VREdit), aber bietet eine Zahl von zusätzlichen Instanz-Methoden an, die mit dem Navigieren durch den Text zusammenhängen
VRListbox	Das ist eine vertikalorientierte Liste von Strings, von welchen der User sich einen oder mehrere Einträge aussuchen kann
VRCombobox	Das ist eine "Drop-Down" Liste von Strings, ähnlich wie VRListbox , nur aber mit dem Unterschied, dass man sich lediglich einen Eintrag aussuchen kann
VRMenu	Das ist eine Menübar, die mit pulldown Menüs ausgestattet ist, und gewöhnlich erscheint entlang des oberen Randes von Hauptprogrammfenster
VRMenuItem	Das ist ein einzelnen Menüeintrag, welcher in VRMenu Widget erscheint(z.B. Speichern als... Befehl in der Datei Menü eines Programmes). Sie werden im Allgemeinen es implizit erzeugen, wenn Sie eine Menü's set Befehl aufrufen

VRMenuUseable	Dieses Modul könnte mit anderen verschiedenen Klassen verwendet werden, welche Menüs benutzen müssen; das liefert die <i>newMenu</i> und <i>setMenu</i> Modulmethode
VRBitmapPanel	Das ist eine besondere Art von Panel, die verwendet wird, um statische Bitmaps zu zeigen
VRCanvasPanel	Das ist eine spezielle Art von Panel, die benutzt wird, um zeichengeeignete Bitmap Images zu zeigen; Sie können die Größe und Höhe von Panel und kann man auch GDI Funktionen, um darauf zu zeichnen

Windows 95 hat sehr viele neue Benutzerfreundlichen Steuereinheiten eingeführt, die als *common* (allgemeine) Steuerelemente genannt werden. Diese beinhalten solche Steuerelemente, die als *list views*, *tree views* bekannt sind und werden umfassend in Windows Explorer verwendet. Die VRuby Klassen stellen allgemeine Steuerelemente, die in der Tabelle 4 auflistet sind.

Tabelle 4 Allgemeine Steuerelemente für VRuby

Klassenname	Beschreibung
VRNotifyControl	Diese Klasse(eine Subklasse von VRControl) ist einfach die Basisklasse für alle allgemeine Steuerelemente
VRListView	Das ist eine steigende "Listensteuerung", welche konfiguriert werden kann, um Daten in verschiedenen Formaten zeigen zu lassen(z.B. als Icons mit einer einfachen Beschreibung, oder in der mehr detaillierten Listenform). Meistens wird diese Steuerung, wie in Windows Explorer bekannter Form verwendet.
VRTreeview	Diese Steuerung, auch gewöhnlich als Tree-Liste bekannt ist, wird verwendet, um die hierarchisch-strukturierte Daten zeigen zu lassen. Das ist eine beliebte Auswahl für meiste Windowsprogrammen, wird zum Beispiel in Register Editor Utility verwendet

VRProgressbar	Diese Steuerung wird während eines sozusagen "zeitaufgebrauchten" Vorgehens verwendet, damit wird den User eine Rückmeldung angeboten, über die er erfahren kann, wie weit irgendwelche Operation vorangekommen ist und wieviel Zeit übriggeblieben ist, bis die Operation beendet wäre
VRTrackbar	Dieses Steuerungselement besteht aus einem sozusagen Schieber oder einer Führung und optionalen Markierungszeichen. Das kann benutzt werden, wenn der User möchte, einen diskreten Wert aus einer Reihe von Werten auszuwählen
VRUpdown	Diese Steuerung besteht aus ein Paar Pfeil-Buttons(gewöhnlich ein nach oben und andere nach unten gerichtet),die der User benutzen kann, um eine Zunahme oder eine Abnahme einigen Werten im Programm zu zeigen. Die bildet fast immer ein Paar mit "buddy" Fenster, wie mit VREdit Steuerung, um ihre aktuelle Einstellungen zu zeigen
VRStatusbar	Das ist ein horizontales Streifen, welches am unteren Rand des Hauptfensters erscheinen kann, um die Statusinformation anzeigen zu lassen
VRStatusbarDockable	Dieses Modul könnte mit der VRForm Klasse verwendet werden. Damit wird die Statusbar-Steuerung eingeschlossen. Benutzt wird dabei eine addStatusbar Methode, um die Statusbar in den Widget hinzuzufügen
VRTabControl	Diese Steuerung zeigt ein oder mehrere Tabs, welche man verwenden kann, um als eine Basis für dei Eigenschaftentabelle oder Tabspanel zu benutzen
VRTabbedPanel	Diese Steuerung kombiniert VRTabControl mit einer Reihe von Panels, um "tabbed" notebook Steuerungen zu erzeugen

Layout Manager

Im Gegensatz zu allen anderen GUI Tookits, die wir bis jetzt gesehen haben, bietet VRuby nicht so viel,

was Layout Manager betrifft. Das ist hauptsächlich auf den Fakt zurückzuführen, dass Win32 API, auf der SWin und VRuby basieren, überhaupt keine irgendwelche Layout Manager enthalten. Obwohl die Layout Manager jetzt mit VRuby mitgeliefert werden, sind sie etwas begrenzt, eins würde in dieser Situation erwarten, als VRuby ausgereifter auszubauen. Es gibt für VRuby drei Layout Manager und zwar *VRHorizLayoutManager*, *VRVertLayoutManager* und *VRGridLayoutManager*. Diese Layout Manager sind einfach Module, die man mit einer Behälterklasse, wie *VRPanel*, verwenden kann, und die dienen nicht als Standalone-Behälterklasse. Ähnlich wie ihre Gegenstücke in Ruby/GTK und FXRuby ordnen die erste zwei Layout Manager ihre Kind-Widgets in der horizontalen oder vertikalen Richtung. Um diese Layout Manager zu benutzen, muss man erstens das gewünschte Layout Manager-Modul mit seiner Behälterklasse zusammenmischen:

```
require 'vr/vrlayout'  
  
class MyPanelClass < VRPanel  
  include VRHorizLayoutManager  
end
```

Danach werden einen oder mehrere Kind-Widgets (oder in der "Windows-Sprache" als *controls* (Steuerelemente) genannt) zum Behälterfenster durch die *addControl* Methode hinzugefügt:

```
aPanel = MyPanelClass.new ...  
  
aPanel.addControl(VRButton, "button1", "Überschrift für den ersten Button")  
aPanel.addControl(VRButton, "button2", "Überschrift für den zweiten Button")
```

Die volle Argumentenliste für unsere Layout Manager *VRHorizLayoutManager#addControl* (oder *VRVertLayoutManager#addControl*) ist:

```
addControl(type, name, caption, style=0)
```

wo *type* eine VRuby Klasse fürs Steuerungselement ist, *name* sein Name ist, und *caption* ein Textstring

ist, der auf dem Steuerungselement erscheint (natürlich für diejenige, die das unterstützen). Das letzte Argument `style` kann verwendet werden, um zusätzlich "windows style flags" für dieses Steuerelement zu übergeben. Die Grundeinstellung dafür ist es, dass VRuby eine Steuerung erzeugt, die nur "basis style flags" enthält; zum Beispiel, VRButton-Steuerungselemente werden durch die `WStyle::WS_VISIBLE` oder `WStyle::BS_PUSHBUTTON` flags erzeugt. Im Gegensatz zu anderen GUI Toolkits brauchen Sie tatsächlich keine Steuerungen über die Parameter von Kinder-Widgets, um deren Größe zu beeinflussen. Zum Beispiel für `VRHorizLayoutManager` ist es so, dass die Höhe von Kinder-Widgets ebenso gleich wie die Höhe des Vater-Widgets ist und die Breite des Vater-Fensters wird durch die Anzahl der Kinder-Steuerungen dividiert. Ferner jeder Kind-Widget breitet sich automatisch (horizontal oder vertikal) aus, um seinen zugewiesenen Raum zu füllen.

Der `VRGridLayoutManager` entspricht ungefähr dem Tk's `grid` Layout, GTK's `Gtk::Table` und FOX's `FXMatrix`, allerdings mit einer Art der Beschränkung für die Größe der Kind-Steuerungen, die wir schon bei den `VRHorizLayoutManager` und `VRVertLayoutManager` gesehen haben. Bevor die Kind-Steuerungen zum Behälter hinzugefügt werden, dabei wird `VRGridLayoutManager` verwendet, müssen wir zunächst eine Anzahl von Spalten und Zeilen mit Hilfe der `setDimension` Methode festlegen:

```
require 'vr/vrlayout'  
  
class MyPanel < VRPanel  
  include VRGridLayoutManager  
end  
  
aPanel = MyPanel.new ...  
aPanel.setDimension(5, 3)
```

wo die zwei Argumente auf die Anzahl der Zeilen beziehungsweise Spalten hindeuten. Gleich danach kann man die Kind-Steuerungen durch die `VRGridLayoutManager#addControl` Methode hinzufügen.

```
addControl(type, name, caption, x, y, w, h, style=0)
```

Hier sind die erste drei Argumenten gleich den Argumenten, die wir vorhin für die [VRHorizLayoutManager](#) und [VRVertLayoutManager](#) erläutert haben. Die nächste zwei Argumente(x und y) weisen auf eine Zelle hin, die sich oben links in der Reihe von Zellen befindet, diese Zelle wird die entsprechende Steuerung belegt, und schließlich die "w" und "h" Argumente weisen darauf hin, wieviel die Breite und die Höhe der insgesamt in der Reihe vorhandenen Tabellenzellen beträgt. Für die Steuerung, die nur eine Tabellenzelle einnimmt, werden Sie lediglich die Breite und die Höhe einer Zelle verwenden.

Handhabung von Ereignissen

VRuby verwendet Callback-basierende Methode, um die Ereignisse zu behandeln, die ganz anderes als die Methoden von anderen GUI's ist, die wir schon gesehen haben. Sie können erstaunt sein, wie man den Namen für jede Steuerung, die Sie zur einer Form hinzufügen möchten, festlegt. Zum Beispiel, wenn man ein Button(das ist natürlich ein Steuerungselement) hinzufügen möchte, kann man so vorgehen: /p>

```
aForm.addControl(VRButton, "button1", "Beschriftung für Button", ...)
```

Das dritte Argument ist ein Beschriftungsstring, der auf dem Button angezeigt wird. Das allerdings nicht heisst, dass dieser Namenstring überall erscheinen kann, aber eigentlich, jedesmal, wenn Sie neue Steuerung zu den [VRForm](#) oder [VRPanel](#) hinzufügen(die sind übrigens im [VRParent](#) Modul vorhanden), legt das Behälter-Objekt auch eine neue Instanzvariable und Zugriffsmethoden fest, und zwar mit den Namen, den Sie durch den Aufruf die Methode [addControl](#) vorher erzeugt haben. Diese neue Instanz-Variable ist einfach ein Verweis auf das Kind-Steuerungselement. Also nachdem der obengennante Zeilencode ausgeführt wird, könnten Sie später die Beschriftung von Button mit dem folgenden Code ändern:

```
aForm.button1.caption = "Neue Beschriftung für den Button"
```

Die Namen von Steuerungselementen sind dafür da, um die Callbacks zu erzeugen und diese Callbacksmethoden müssen solche Namen tragen, wie wir das an folgenden Coden erkennen können:

```
def controlname_eventname
```

```

en...Behandle dieses Ereignis
end

```

So zum Beispiel, wenn Sie möchten, ein *clicked* Ereignis für die **button1** Steuerung abzufangen, müssen Sie der Methode als Name *button1_clicked* vergeben:

```

def button1_clicked
  evputs "Button1 wurde angeklickt!"
end

```

Diese Callbacksmethoden müssen als Instanzmethoden fürs Behälterfenster festgelegt werden, und das heisst, wenn wir *button1* Steuerung als ein Kind-Widget für den *aForm* erzeugt haben, muss die *button1_clicked* Methode als eine Instanzmethode vom *aForm* Vater-Widget sein. Wir werden später es sehen, wie das im unseren Beispielprogramm funktioniert. Die folgende Tabellen 5 und 6 bieten eine Liste von Ereignisnamen für alle VRuby's Steuerungselementen, die Sie verwenden können.

Tabelle 5 Ereignisnamen für Standard-Steuerungen

Steuerungsklassen	Ereignisname(n)
VRButton	clicked, dblclicked
VREdit	Changed
VRListbox	Selchange
VRCombobox	Selchange

Tabelle 6 Ereignisnamen für allgemeine Steuerungselementen

Steuerungsklassen	Ereignisname(n)
Alle allgemeine Steuerungen	clicked, dblclicked, gotfocus, hitreturn, lostfocus, rclicked, rdblclicked

VRListView	itemchanged, itemchanging, columnclick, beginndrag, beginngrad
VRTreeview	selchanged, itemexpanded, deleteitem, beginndrag, beginndrag
VRUpdown	Changed
VRTabControl	Selchanged

VRuby Beispielprogramm

Der untenstehende Code zeigt uns eine VRuby's Version unseres Beispielprogrammes im seinen vollen Umfang. Den können Sie selbstverständlich auch herunterladen [vruby-xmlviewer](#). Der Code beginnt mit dem Importieren von verschiedenen VRuby Bibliotheken, die wir benötigen werden:

```
require 'vr/vruby'  
require 'vr/vrcontrol'  
require 'vr/vrcomctl'  
require 'vr/vrtwopane'
```

Als Nächstes definieren wir globale Konstanten, die benötigt werden, um später im Programm Nachrichtenboxen zu zeigen

```
# The values of these constants were lifted from  
MB_OKenjoyenjoyenjoy = 0x00000000  
MB_ICONEXCLAMATION = 0x00000030  
MB_ICONINFORMATION = 0x00000040
```

Die Win32 Schnittstelle verwendet eine lange Zahl für die Namen von Konstanten(wie MB_OK, MB_ICONEXCLAMATION und MB_ICONINFORMATION), um die "style flags" für die Steuerungen festzulegen. In unserem Fall legt das letzte Argument der VRuby's `messageBox` Funktion Optionen von der Nachrichtenbox fest, solche wie gezeigte Buttons und Icons. Da diese Konstanten noch nicht gut bei SWin/VRuby repräsentiert werden, müssen Sie sozusagen in allen Windows Header Dateien wählen, um

die aktuelle numerische Werte für diese Konstanten festzusetzen. Wie Sie schon bestimmt geahnt haben, ist das ein Zeitpunkt, wo die Windowsprogrammierung ihre Stelle einnehmen würde. Der nächste bedeutungsvolle Block des Codes legt die [XMLViewerForm](#) Klasse fest, und damit auch ihre Instanzmethoden; das ist sozusagen ein Brennpunkt der unseren Applikation. Bevor wir uns damit beschäftigen, lassen wir uns für eine kurze Zeit einen Teil des Codes überfliegen und werfen wir einen Blick auf letzte wenige Zeilen des Programmes:

```
mainWindow = VRLocalScreen.newform(nil, nil, XMLViewerForm)
mainWindow.create
mainWindow.show

# Start der Nachrichtenschleife
VRLocalScreen.messageloop
```

Nachdem wir die [XMLViewerForm](#) Klasse definiert haben (die ist einfach eine Subklasse von [VRForm](#), rufen wir [newform](#) Methode auf, die zur [VRLocalScreen](#) Klasse gehört, damit erzeugen wir eine Instanz von der [XMLViewerForm](#) Klasse. Zur Erinnerung, [VRLocalScreen](#) ist eine spezielle Variable in VRuby; das ist einzige globale Instanz von [VRScreen](#) Klasse und sie entspricht locker dem Windows Desktop. Die erste zwei Argumente für [newform](#) Methode sind Vater-Fenster und "style flags" für dieses neuen Fenster. Im unseren Fall ist das das Hauptfenster unseres Programmes und so damit übergeben wir **nil** als Argument an die [newform](#) Methode. Letzendlich haben wir erreicht, dass sozusagen "Stil" (style) "default" wird. Danach rufen wir [mainWindow.create](#) auf, die eigentlich nichts Anderes macht, als das wirkliche Fenster als Ruby Objekt erzeugt. Von diesem Zeitpunkt an ist das Behälter-Fenster, der Aufruf dieser Methode (create) löst den Aufruf der anderen Methode aus, und zwar [construct](#) Methode (wir werden das später sehen), um Steuerungselemente und Menüs zu erzeugen. Und schließlich wird die [show](#) Methode aufgerufen, damit wird das Hauptfenster sichtbar, bevor die Hauptereignisschleife (main even loop) eintritt.

Nun lassen wir uns zurückgehen, und schauen wir uns den Code für unsere "Formklasse" und zwar [XMLViewerForm](#). Wir beginnen damit, dass wir zwei Module einbinden, die sehr nützliche Funktionen von VRForm zur Verfügung stellen:

```
include VRMenuUseable
include VRHorizTwoPane
```

Das *VRMenuUseable* Modul liefert uns die *newMenu* und *setMenu*, die dazu dienen, dass man damit Pulldown's Menüs aufbauen kann. Das *VRHorizTwoPane* Modul fügt die *addPanelControl* Methode hinzu, und ermöglicht einen horizontal-gesplitterten Layout für den Inhalt des Hauptfensters. Als nächstes definieren wir zur Form gehörige *construct* Methode, die nichts anderes macht, als die Menüs erzeugt und Steuerungselemente zum Hauptfenster hinzufügt:

```
erdef construct
enj# Set caption for application main window
enjself.caption = "XML Viewer"

enj# Create the menu bar
enj@menu = newMenu()
enj@menu.set([ ["&File", [ ["&Open...", "open"], ["Quit", "quit"] ] ],
enjoyenjoyenj["&Help", [ ["About...", "about"] ] ] ])
enjsetMenu(@menu)

enj# Tree view appears on the left
enjaddPanedControl(VRTreeview, "treeview", "")

enj# List view appears on the right
enjaddPanedControl(VRListview, "listview", "")
enj@listview.addColumn("Attribute Name", 150)
enj@listview.addColumn("Attribute Value", 150)
erend
```

Der Aufruf der *newMenu* Methode erzeugt einfach eine leere Menü und weist die danach zur unseren *@menu* Instanzvariable zu; und gleich danach der Aufruf der *set* Methode legt einfach den Inhalt des

Menüs fest. Das sieht ein wenig kompliziert aus, muss aber nicht sein, weil `set` Methode ein Argument hat, welches ein Array von Array's ist, ein pro Pulldown Menü. Das für jede Pulldown Menü Subarray ist selbst ein zweielementiges Array. Betrachten wir das erste Array, welches der File Menü entspricht:

```
[ ["&File", [ ["&Open...", "open"], ["Quit", "quit"] ] ]
```

Das erste Arrayelement ist der Name des Pulldown Menü. Durch die Platzierung "ampersand"("&") vor der Buchstabe "F" in "File" wird es erreicht, dass die **Alt+F** Shortkey-Kombination verwendet kann, um das "File" Menü aufzuklappen. Das zweite Element in diesem Array ist auch ein anderes Array, diesmal ein Array, das einen Eintrag im "File" Menü darstellt. Jedes Element solches Array's ist entweder ein zweielementiges Array, das für den Menüeintrag eine Beschriftung und einen Namen anbietet, oder soeben die Beschriftung und noch ein anderes Array eines Menüeintrages um ein weiteres "cascading" sozusagen Submenü zu präsentieren. Solche Submenüs machen unseren Author(M.Neumann) ein bisschen "verrückt", und so bleibt der Author weiter hin mit Single-Level Menü zu beschäftigen. Einen Kapitel zuvor haben wir das Thema "Ereignisbehandlung" unterrichtet und haben gesehen, dass Namen von Steuerungselementen sehr wichtig sind, weil sie in den Namen für VRuby's Callback's Methoden verwendet werden. Auf gleicherweise sind auch von Ihnen zugewiesenen Namen von Menüeinträgen wichtig; wenn der Anwender auf einen solchen Menüeintrag klickt, wird VRuby nach der Callbackmethode suchen, die den Namen `name_clicked` trägt. Bald werden wir sehen, dass die `XMLViewerForm` Klasse die Callback's Methoden `open_clicked`, `quit_clicked`, `about_clicked` definiert, um diese drei Menübefehle festzusetzen.

Nachdem wir mit der Hilfe der `construct` Methode die Applikationsmenü's definiert haben, füügen wir genau zwei Steuerungselemente zum unseren horizontalen "Schauplatz"(pane):

```
o# Tree view appears on the left
oaddPanedControl(VRTreeview, "treeview", "")

o# List view appears on the right
oaddPanedControl(VRListview, "listview", "")
```

Die Namen von Kinder-Steuerungselementen, die man zu *VRParent* hinzufügte, werden als Namen Instanzvariablen. Wir werden daraus einen Vorteil nutzen, um weiter den Listview Widget mit dem Namen *listview* zu konfigurieren:

```
@listview.addColumn("Attribute Name", 150)
@listview.addColumn("Attribute Value", 150)
```

Als nächstes lassen wir uns auf drei Callback's Methoden konzentrieren. Die behandeln **Open...**, **Quit** und **About** Menübefehle. Da der Name fürs Menüeintrag **Open...** *open* war, wurde seine Callback-Methode als *open_clicked* genannt:

```
endef open_clicked
  enfilters = [{"All Files (*.*)", "*. *"},
  enjoyenjoy["XML Documents (*.xml)", "*.xml"]]
  enfilename = openFileDialog(filters)
  enloadDocument(filename) if filename
end
```

Die *openFileDialog* Methode ist eine bequeme Funktion um einen üblichen Windows Dateialog zu zeigen. Das ist eine Modulmethode, die im *VRCommonDialog* Modul integriert ist. Die kann genauso gut auch in der *VRForm* Klasse verwendet werden, so dass man diese Methode für irgendwelche Form benutzen. Als Argument erwartet diese Methode ein Array von Dateinamen's Mustern(oder wie in unserem Fall *filters*), welches im Dateialog angezeigt wird und danach den Namen der ausgesuchten Datei zurückliefert(oder **nil**, wenn der Anwender den Dialog abbricht). Der Callback für den **About...** Menübefehl wird durch die Methode gehandelt:

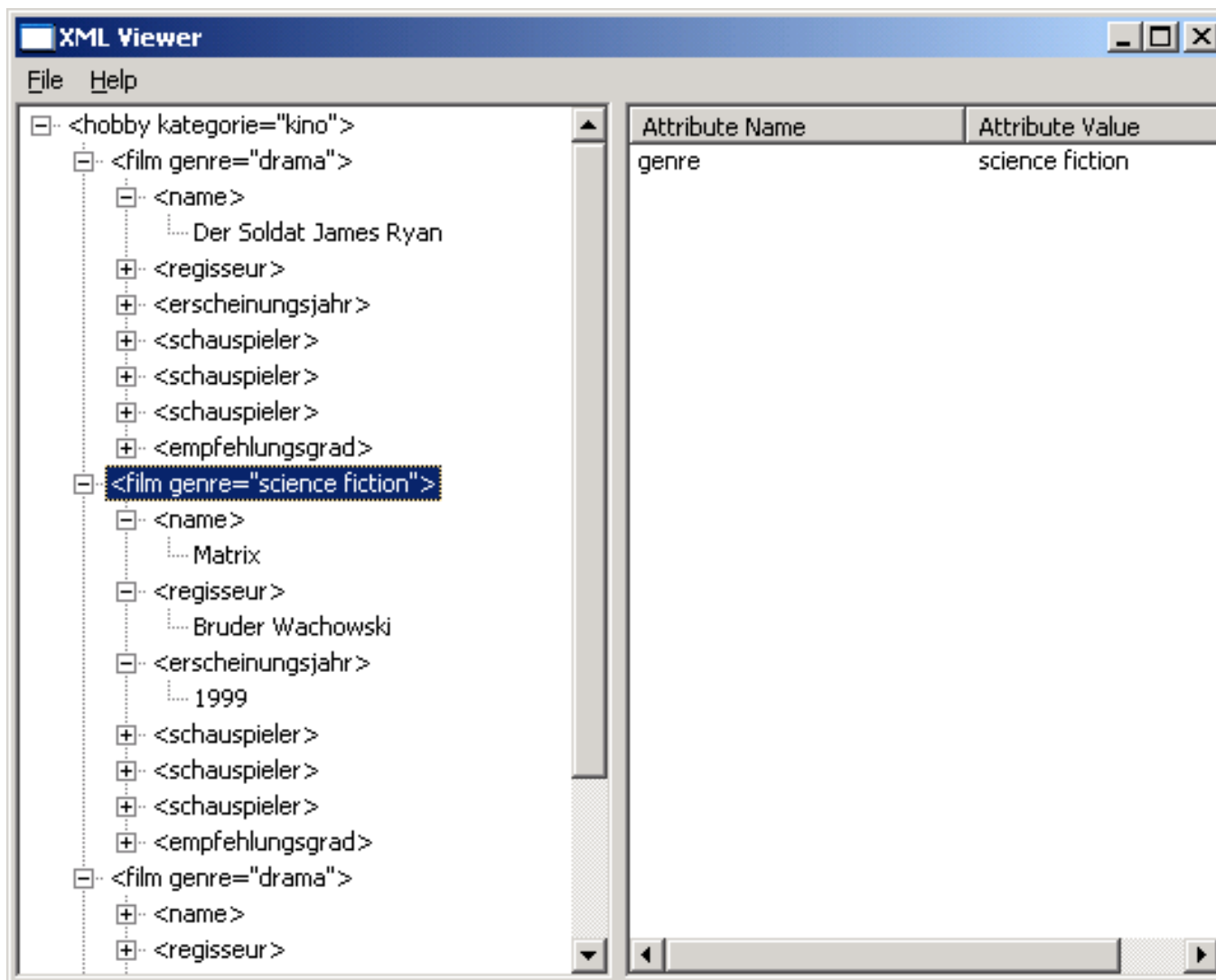
```
endef about_clicked
  enmessageBox("VRuby XML Viewer Example", "About XMLView",
  enjoyMB_OK|MB_ICONINFORMATION)
```

```
end
```

Die `messageBox` Methode ist eine andere Art der "bequem"-Funktion, und sie ist einfach eine Instanzmethode `SWin::Window` Klasse (eine weit entfernte Vorfahrklasse von unserer Form). Wie Sie schon wahrscheinlich vermutet haben, sind die drei Argumente die Nachricht selbst, die auf der Oberfläche des Fensters erscheint, der Titel des Fensters und "style flags", das darauf hinweist, welche Icons und Buttons (in der Regel zum Beenden) angezeigt werden. Der Callback für den **Quit** Menübefehl ist am leichtesten zu verstehen, weil der nur einfach Ruby's `exit` Methode aufruft:

```
endef quit_clicked  
  exit  
end
```

Die folgende Abbildung zeigt eine VRuby Version unseres Beispielprogrammes, das unter Window XP ausprobiert wurde:



Und hier ist der vollständige Code unseres Beispielprogrammes:

```
require 'vr/vruby'  
require 'vr/vrcontrol'  
require 'vr/vrcomctl'  
require 'vr/vrtwopane'  
  
require 'nqxml/treeparser'
```

```
e# The values of these constants were lifted from
eMB_OKenjoyenjoyenjoy = 0x00000000
eMB_ICONEXCLAMATION = 0x00000030
eMB_ICONINFORMATION = 0x00000040

eclass XMLViewerForm < VRForm

eninclude VRMenuUseable
eninclude VRHorizTwoPane

endef construct
enj# Set caption for application main window
enjself.caption = "XML Viewer"

enj# Create the menu bar
enj@menu = newMenu()
enj@menu.set([ ["&File", [ ["&Open...", "open"], ["Quit", "quit"] ] ],
enjoyenjoyenj["&Help", [ ["About...", "about"] ] ] ])
enjsetMenu(@menu)

enj# Tree view appears on the left
enjaddPanedControl(VRTreeview, "treeview", "")

enj# List view appears on the right
enjaddPanedControl(VRListview, "listview", "")
enj@listview.addColumn("Attribute Name", 150)
enj@listview.addColumn("Attribute Value", 150)
enend

erdef populateTreeList(docRootNode, treeRootItem)
enjentity = docRootNode.entity
enjif entity.instance_of?(NOXML::Tag)
```

```
enjoytreeltem = @treeview.addItem(treeRootItem, entity.to_s)
enjoy@entities[treeltem] = entity
enjoydocRootNode.children.each do |node|
enjoyenjpopulateTreeList(node, treeltem)
enjoyend
enjelsif entity.instance_of?(NOXML::Text) &&
enjoyenjentity.to_s.strip.length != 0
enjoytreeltem = @treeview.addItem(treeRootItem, entity.to_s)
enjoy@entities[treeltem] = entity
enjoyend
enend

enendef loadDocument(filename)
enj@document = nil
enjbegin
enjoy@document = NOXML::TreeParser.new(File.new(filename)).document
enjrescue NOXML::ParserError => ex
enjoymessageBox("Couldn't parse XML document", "Error",
enjoyenjMB_OK|MB_ICONEXCLAMATION)
enjoyend
enjif @document
enjoy@treeview.clearItems()
enjoy@entities = {}
enjoypopulateTreeList(@document.rootNode, @treeview.root)
enjoyend
enend

enendef open_clicked
enjfilters = [{"All Files (*.*)", "*. *"},
enjoyenjoy["XML Documents (*.xml)", "*.xml"]]
enjfilename = openFileDialog(filters)
```

```
enloadDocument(filename) if filename
enend

en def quit_clicked
enexit
enend

en def about_clicked
enmessageBox("VRuby XML Viewer Example", "About XMLView",
enjoyMB_OK|MB_ICONINFORMATION)
enend

en def treeview_selchanged(hItem, IParam)
enentity = @entities[hItem]
enif entity and entity.kind_of?(NOXML::NamedAttributes)
enjoykeys = entity.attrs.keys.sort
enjoy@listview.clearItems
enjoykeys.each_index { |row|
enjoyenjoy@listview.addItem([ keys[row], entity.attrs[keys[row]] ])
enjoy}
enend
enend
enend

enmainWindow = VRLocalScreen.newform(nil, nil, XMLViewerForm)
enmainWindow.create
enmainWindow.show

en# Start the message loop
enVRLocalScreen.messageLoop
```

```
require 'vr/vruby'
```

```
require 'vr/vrcontrol'
```

```
require 'vr/vrcomctl'
```

```
require 'vr/vrtwopane'
```

```
require 'nqxml/treeparser'
```

```
# The values of these constants were lifted from <winuser.h>
```

```
MB_OK          = 0x00000000
```

```
MB_ICONEXCLAMATION = 0x00000030
```

```
MB_ICONINFORMATION = 0x00000040
```

```
class XMLViewerForm < VRForm
```

```
  include VRMenuUseable
```

```
  include VRHorizTwoPane
```

```
  def construct
```

```
    # Set caption for application main window
```

```
    self.caption = "XML Viewer"
```

```
    # Create the menu bar
```

```
    @menu = newMenu()
```

```
@menu.set([ ["&File", [ ["&Open...", "open"], ["Quit", "quit"] ] ],  
           ["&Help", [ ["About...", "about"] ] ]  
         ]  
       )
```

```
setMenu(@menu)
```

```
# Tree view appears on the left
```

```
addPanedControl(VRTreeview, "treeview", "")
```

```
# List view appears on the right
```

```
addPanedControl(VRListview, "listview", "")
```

```
@listview.addColumn("Attribute Name", 150)
```

```
@listview.addColumn("Attribute Value", 150)
```

```
end
```

```
def populateTreeList(docRootNode, treeRootItem)
```

```
  entity = docRootNode.entity
```

```
  if entity.instance_of?(NQXML::Tag)
```

```
    treeItem = @treeview.addItem(treeRootItem, entity.to_s)
```

```
    @entities[treeItem] = entity
```

```
    docRootNode.children.each do |node|
```

```
      populateTreeList(node, treeItem)
```

```
    end
```

```
  elsif entity.instance_of?(NQXML::Text) &&
```

```

    entity.to_s.strip.length != 0

    treeItem = @treeview.addItem(treeRootItem, entity.to_s)

    @entities[treeItem] = entity

end

end

def loadDocument(filename)

    @document = nil

    begin

        @document = NQXML::TreeParser.new(File.new(filename)).document

    rescue NQXML::ParserError => ex

        messageBox("Couldn't parse XML document", "Error",

            MB_OK|MB_ICONEXCLAMATION)

    end

    if @document

        @treeview.clearItems()

        @entities = {}

        populateTreeList(@document.rootNode, @treeview.root)

    end

end

def open_clicked

    filters = [{"All Files (*.*)", "*..*"},

```

```

        ["XML Documents (*.xml)", "*.xml"]]

filename = openFileDialog(filters)

loadDocument(filename) if filename

end

def quit_clicked

    exit

end

def about_clicked

    messageBox("VRuby XML Viewer Example", "About XMLView",

        MB_OK|MB_ICONINFORMATION)

end

def treeview_selchanged(hItem, lParam)

    entity = @entities[hItem]

    if entity and entity.kind_of?(NQXML::NamedAttributes)

        keys = entity.attrs.keys.sort

        @listview.clearItems

        keys.each_index { |row|

            @listview.addItem([ keys[row], entity.attrs[keys[row]] ])

        }

    end
end

```

end

end

mainWindow = VRLocalScreen.newform(nil, nil, XMLViewerForm)

mainWindow.create

mainWindow.show

Start the message loop

VRLocalScreen.messageloop

Andere GUI Toolkits

Wie Sie schon geahnt haben, gibt's auch andere GUI Toolkit's für Rubyprogrammierung, und wie auch immer, könnten Sie danach in RAA nachforschen. Der Fast Light Toolkit (FLTK) www.fltk.org ist ein netter cross-plattformiger GUI, der zum Teil von Bill Spitzak entwickelt wurde. FLTK ist in Sachen der Geschwindigkeit und Arbeitsspeicherverwaltung sehr effizient und außerdem bietet hervorragende Unterstützung für die Entwicklung von OpenGL-basierenden Programmen an. Zur Zeit ist der für beide Windows und X(Unix) Plattformen verfügbar. Die Ruby's Schnittstelle zu FLTK wird von Takaaki Tateishi und Kevin Smith entwickelt und es gibt natürlich dazu eine Webseite <http://ruby-fltk.sourceforge.net>

Qt www.trolltech.com ist ein hervorragender plattformunabhängiger GUI Toolkit, der für Unix-Derivate, Windows und neulich auch für Mac OS X zur Verfügung steht. Der ist der Basis des populären KDE Desktops für Linux. Die Ruby's Module für Qt wurden von Nobuyuki Horie entwickelt und stehen auf diese Seite [Ruby-Qt](#)

Apollo www.morig.com/apollo/index-en.html entwickelt von Yoshida Kazuhiro, ist ein Projekt, dessen Ziel ist, nach den Worten des Authors, ein "Dream Duet" Delphi und Ruby anzubieten. Delphi ist eine kommerzielle Entwicklungsumgebung (RAID) von der Firma Borland/Inprise. Das Besondere an diesem Projekt ist es, dass es eine Ruby's Erweiterung anbietet, die es erlaubt, auf die Delphi's Visual Component Library (VCL) zu zugreifen. Wie Sie schon wissen, ist Delphi reines Windows-Phänomen, aber es gibt schon die Unix-Portierung von Delphi und zwar Kylix.

Es kann sein, dass es sehr schwierig wird, so viel Möglichkeiten zu haben, um sich für einen GUI Toolkit für Ruby zu entscheiden. Schließlich gibt es keine magische Formel, für Sie eine Entscheidung zu treffen, aber hier sind ein paar sozusagen Erwägungen, die man sich gut merken sollte:

- Auf welchen Plattformen soll das Programm laufen? Wenn das Programm für Macintosh entwickelt

wird, ist der Einsatz von Tk am sinnvollsten. So ähnlich, wenn man das Programm auf verschiedenen Plattformen laufen soll, es ist wahrscheinlich nicht angebracht, die Applikation mit SWin/VRuby zu entwickeln.

- Wenn Sie doch beabsichtigen, Ihres Programm auf verschiedenen Plattformen laufen zu lassen, und bevorzugen ein einheitliches "look-and-feel" für GUI, oder hätten Sie besser ein native "look-and-feel" für jede Zielplattform? Tk bietet ein native "look-and-feel" auf jeder Plattformen an, aber "theme-able" Toolkits wie GTK kann äußerst benutzerfreundliche "customizable" Schnittstelle anbieten(vielleicht anders als irgendwelche "native" GUI's). FOX bietet ein gleich bleibendes "look-and-feel" für beide Unix und Windows Plattformen an(allerdings dieses "look-and-feel" ist eher Windowsmässig).
- Das Problem der Softwarelizenz kann von großer Bedeutung sein, insbesondere wenn Sie kommerzielle Software entwickeln möchten. Meisten von GUI Toolkits für Ruby verwenden irgendeine Art von Open-Source Software Lizenz, aber Sie sollten diese Lizenz sorgfältig studieren, um alle Bedingungen zu verstehen

Abgesehen von allen anderen Problemen ist der größter Faktor, der nicht gerade leicht gelöst werden kann, wie komfortabel die Entwicklung von Programmen mit diesen gegebenen GUI Toolkit's sein kann. Von dem Sicht des Programmieres an, hat jeder GUI Toolkit seine individuelle sozusagen Stellung und einmaliges Aussehen und Sie können es nicht einfach befühlen, was es Ihnen am bestimmten Toolkit gefällt. Wenn Sie ausreichend Zeit haben, nutzen Sie die Gelegenheit, mehr über alle diese Toolkit's, die wir in diesem Kapitel vorgestellt haben, zu lernen, bevor man sich für einen oder anderen entscheidet.

Zusammenfassung

In diesem Kapitel möchten wir zusammenfassen, was einige von am populärsten GUI Toolkit's für Ruby anbieten. Es ist natürlich gut, eine Menge der Möglichkeiten zu haben, die Ihnen zur Verfügung stehen, aber um ein effektiver Entwickler zu sein, ist es in Ihren Interessen mit verschiedenen GUI Toolkit's zu experimentieren und danach den einen auszuwählen, der Ihnen am besten zu passen erscheint, damit Sie Ihre Pläne umsetzen können. Die Erfahrung mit GUI-Programmierung kann nicht in den Büchern gelehrt werden; die erfordert einige Ausprobierungen und Experimentierungen in diesem Feld.

Wir beginnen damit, dass wir uns kurz Ruby/Tk anschauen. Dieser Toolkit ist zweifellos ein Standard für Ruby GUI-Programmierung und natürlich ein Favorit für viele Programmierer. Tk war ein der ersten plattformunabhängigen GUI's. Leichte Programmierung von Applikationen, die durch Tcl/Tk durchaus möglich ist, öffnete vielen Entwicklern, die früher mit C-basierenden GUI-Bibliotheken wie Motif und Windows Win32 API gekämpft haben, eine schöne Welt der GUI-Programmierung. Von allen GUI Toolkit's, die wir uns angeschaut haben, und das ist auch wahr, dass Ruby/Tk die kleinste Menge von Code erfordert, um ein GUI zu programmieren und danach laufen zu lassen. Und das alles, wie auch immer, auf die Kosten mehrerer neuer GUI-Innovationen, wie zum Beispiel "drag and drop" (ziehen und fallen lassen), oder fortschrittliche Widgets wie "spreadsheets" (Tabellenkalkulation) und "tree lists" (Baumstrukturierte Listen).

Der nächste GUI Toolkit, den wir betrachtet haben, war Ruby/GTK. Für Entwickler, die hauptsächlich auf Linux OS arbeiten und sind sie schon mit GTK+ und GNOME-basierenden Applikationen vertraut, ist das sicherlich eine beste Wahl. Die Ruby/GTK Bindungen sind ganz vollständig und es gibt eine umfassende Online-Dokumentation, um Ihnen den Einstieg zu verschaffen, die auch Übungen und Tutoriale beinhalten. Der einzige Nachteil scheint für Windows-Entwickler zu sein, weil es ist, ab und zu schwierig GTK+ und Ruby/GTK zu verstehen und zum Laufen zu bringen.

FXRuby ist ein mächtiger plattformübergreifende GUI Toolkit für Ruby, und er läuft genau gleich auf Unix und Windows Plattformen. Zusätzlich zum vollen Einsatz von modernen Widgets, bieten FOX und FXRuby eine Menge von Möglichkeiten, um Merkmale wie OpenGL-basierende 3-D Grafiken, "drag and drop" and "persistent" Registereinstellungen. In seiner verhältnismäßig kurzen Zeit auf Ruby GUI Szene ist FXRuby ein der beliebten GUI Toolkits für Ruby geworden. Seine Nachteile können aber nicht ignoriert werden, wie auch immer: seine feste Anbindung an C++ Bibliothek, die FXRuby's Ereignisbehandlungsmethode ist unpraktisch im Vergleich zu anderen Ruby GUI Toolkits. Der Mangel an der umfassenden Userdokumentation und Referenzen für FOX und FXRuby ist auch ein Makel, den für viele Entwickler als Argument erscheint.

Apropos schlechte Dokumentation, SWin/VRuby ist auch mühsam für einen Anfänger als einen erfahrenen Windowsprogrammierer sozusagen zu vermarkten. Andererseits, wenn Ihre Programmierungserfahrung so ist, dass Sie bereits in der Kunst der Win32-Programmierung erfahren sind, und würden Sie gern Ihres Wissen auf die Rubyprogrammierung unter Windows übertragen, dann könnte SWin und VRuby die richtige Wahl für Sie sein.

Schließlich, obwohl wir uns nur vier von meist bekanntesten GUI Toolkits angeschaut haben, ist das lediglich die Spitze vom Eisberg. Es gibt sicherlich eine Menge anderen GUI Toolkits, und während Sie diesen Artikel lesen, kann schon noch ein Toolkit das Licht der Welt erblicken. Nehmen Sie sich einfach genügend Zeit und forschen Sie in RAA nach, und sowie Newsgroups und Mailinglisten, um das Neuste in der Ruby's Entwicklungswelt zu erfahren.

Schneller Überblick über die behandelte Themen

Verwendung Standard Ruby GUI : Tk

- Tk ist noch immer der Standard GUI für Ruby, und das allein ist ein guter Grund dafür, Tk für die Entwicklung ihren Programmen in Betracht zu ziehen. Er bietet eine Möglichkeit an, Ihre Applikationen auf fast allen gängigen Plattformen laufen zu lassen, weil Sie fast gesichert sind, dass Anwender, die ihre Programme verwenden werden, eine lauffähige Ruby/Tk Installation am Platz haben.
- Das größte Problem bei der Verwendung von Tk ist sein Mangel an modernen Widgets wie "combo-boxen", "tree-listen" und so weiter. Zwar gibt's spezielle Erweiterungen wie BLT und Tix, durch die man Tk erweitert werden kann, und auch wenigstens existiert ein Ruby Modul, um diese Tk's Erweiterungen einzubinden und zu verwenden, allerdings verläuft die Installation von diesen Erweiterungen nicht reibunslos und erfordert dabei viele Anstrengungen und Mühe.

Verwendung GTK+ Toolkit

- Da GTK+ als ein der Kern-Komponenten des populären GNOME Desktops für Linux dient, kann der Einsatz dieses Toolkits in der absehbarer Zeit stark zunehmen. Die Ruby/GTK Erweiterung ist ebenso andauernd in der Entwicklung und bietet bereits die meiste oder fast alle der GTK+ Funktionalitäten an.
- Eine potenzielle Quelle des Problems für GTK+ und damit auch für Ruby/GTK ist seine Schwäche bei der Portierung auf Windows OS, was natürlich auf X Window System zurückzuführen ist. Jedoch ist wahrscheinlich, dass diese Probleme ab einem Zeitpunkt gelöst werden, spätestens mit dem neuen Entwurf GTK+ 2.0.

Verwendung FOX Toolkit

- FOX bietet eine hervorragende plattformübergreifende GUI-Lösung an, und im Gegensatz zu GTK+ läuft dieser Toolkit genau so gut auf beiden Plattformen Linux und Windows. Zusätzlich zur seinen umfassenden Sammlung von modernen Widgets, bietet Fox "built-in" Unterstützung für "drag and drop", OpenGL, und große Vielfalt von Imagedatei-Formaten.
- Ein Nachteil des FOX Toolkits ist der Mangel an der gedruckten Dokumentation. Meist der größten Buchhandlungsketten(oder Online Buchhändler) haben eine große Auswahl der Bücher und Referenzen für beide Tk und GTK+ Toolkits, aber Sie werden nicht leider fündig, wenn man irgendwelche Bücher für FOX-Programmierung sucht.

Verwendung SWin/VRuby Toolkit

- SWin und VRuby bieten eine schnelle und einheitliche Lösung für die Entwicklung von grafischen Benutzeroberflächen auf Windows OS an. Wenn Sie ihre Applikationen auf keine OS als Windows laufen müssen, oder irgendeine andere alternative GUI-Lösung für nicht-Windows Betriebssysteme haben, könnte es sein, dass man richtigen Toolkit ausgewählt hat.
- Eine fehlende Dokumentation ist ein Problem, wenn man mit diesem Toolkit zu programmieren anfängt, besonders wenn man keine Erfahrung in der Windowsprogrammierung hat. Es wird wahrscheinlich dabei sehr hilfreich, sich zuerst über die Basis der Windowsprogrammierung beibringen lassen. Es gibt viele sehr ausgezeichnete Referenzbücher in diesem Gebiet.

Andere GUI Toolkits

- Wir haben uns für die meist populärsten GUI Toolkits für Ruby in diesem Kapitel entschieden aber lassen Sie sich bitte nicht von anderen GUI Toolkits abschrecken, die sehr interresant für Sie sein könnten. Sie können sich auch in Ruby Newgroups und Mailingslisten fündig machen, und außerdem ist es ratsam, regulär nach der Information in der RAA nachforschen, weil sonst man niemals weiss, wo und wann die Neuerungen bekommen kann.

Auswählen eines GUI Toolkits

- Obwohl Ruby eine mächtige Programmierungssprache für irgendeine einzelne Plattform ist sind manche Programmierer nicht dazu geneigt, Ruby als Entwicklungssprache für ihr GUI-Applikationen zu verwenden und das liegt an seiner(Ruby) plattformübergreifenden Natur. Wenn Sie ihre GUI Applikationen in Ruby schreiben möchten müssen Sie es berücksichtigen, auf welcher Plattform das Programm laufen wird und dementsprechend der GUI Toolkit auswählen.
- Das Grundgedanke dieses Kapitels ist es, dass es keine für alle "befriedigende" Lösung gibt, wenn man einen GUI Toolkit sucht. Wenn Sie von einem Toolkit zu anderen tendieren, nehmen Sie sich Zeit und probieren Sie zwei oder drei Toolkits aus, die nach Ihrer Meinung vielversprechend sind und entscheiden Sie sich schließlich für einen.

Häufig gestellte Fragen(FAQ)

Frage: Sind irgendwelche GUI Toolkits für Ruby "thread-safe"?

Antwort: Die Antwort auf diese Frage hängt sehr von der Definition des Begriffes "thread-safe" ab. Im allgemeinen Sinne, nicht aber im Sinne GUI-Toolkits, haben wir "thread-safe" betrachtet; das heisst, dass die Instanzmethode von GUI Objekten keine geeignete Unterstützung dafür haben. Eine gute Verfahrensweise für die multithread-GUI Applikationen ist es, die GUI in "Haupt-Thread" laufen zu lassen, und nicht GUI "worker"-Threads für Hintergrundtasks wann auch immer möglicherweise zu reservieren.

Frage: Ruby/GTK und FXRuby kommen mit einigen guten Programmen herbei, aber Ich kann nicht bei weitem für Ruby/Tk finden. Gibt's eine gute Quelle, wo man sich zusätzlich Ruby/Tk Beispielprogramme besorgen kann?

Antwort: Schauen Sie in "Ruby Applikation Archives" nach den letzten Versionen von "Ruby/Tk Widgets Demos" Ausgaben nach, die werden von Jonathan Conway gewartet. Das ist sozusagen Ruby/Tk Portierung von ursprünglichen Tcl/Tk Widget Demos und das würde Ihnen anfangs eine gute Hilfestellung geben, um ihre Ruby/Tk Applikationen zu entwickeln. Sie könne ebenso in RAA nach Ruby/Tk Programmen suchen, und dann sie als den Lernstoff benutzen.

Frage: Ich habe mir zuerst FOX und gleich danach FXRuby installiert, und allem Anschein nach ist das Kompilieren von Sourcecode ohne Fehler verlaufen. When ich aber versuche, irgendeine FXRuby Beispielprogramm unter Linux laufen zu lassen, reagiert Ruby darauf mit einer Fehlernachricht, die damit beginnt: "LoadError(Fehler beim Laden): libFOX.so: cannot open shared object file(kann nicht shared Objekt-Datei öffnen)". Ich habe im FOX Installationsverzeichnis nachgeschaut und habe ich mir festgestellt, dass libFOX.so eigentlich vorhanden ist, also warum bringt Ruby diesen Fehler heraus?

Antwort: Das Problem hat damit zu tun, wie das Betriebssystem gemeinsam benutzte Bibliotheken(shared libraries) festlegt, von denen Ruby Erweiterungen wie FXRuby abhängen. Ruby hat die FXRuby in der Tat gefunden, aber er kann nicht FOX shared Bibliotheken finden, die von FXRuby benötigt werden, weil

diese Bibliotheken nicht im Standard-Verzeichnis für DDL(dynamically-loaded shared libraries) vorhanden sind. Um dieses Problem zu beheben, müssen Sie einfach das Verzeichnis, wo sich shared Bibliotheken befinden(üblich /usr/local/lib) zu LD_LIBRARY_PATH Umgebungsvariable hinzufügen. Sehen Sie dabei in der FXRuby Installationsanweisung für mehr Information über dieses Problem nach.