Teil 3: Ruby-Anwendungen

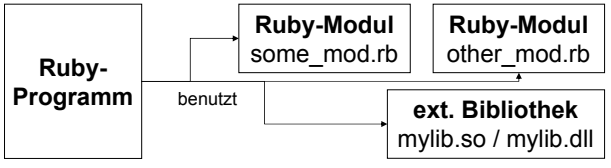
Notwendigerweise eine kleine Auswahl

Extensions

Ruby-Scripts von C aus starten
Ruby-Funktionen in C einbinden
C-Bibliotheken in Ruby nutzen: SWIG
Windows Extensions: Win32API, Win32OLE

Ruby erweitern - warum ?

- Benutzung vorhandener Schnittstellen
  - Existierende Bibliotheken einfach in Ruby mitbenutzen
  - Objektorientierte Kapselung älterer Komponenten
  - Hardwarenahe Programmierung
- Effizienz
  - Laufzeit- oder speicherplatzkritische Abschnitte in einer schlanken Compilersprache implementieren,
  - Integrationskomfort von Ruby beibehalten.



- Merke: Was Assembler für C, ist C für Ruby!

Ruby-Scripts von C aus starten

Ruby-Skripte von C aus verwenden

- Grundlage:
  - Ruby ist in C geschrieben, daher vorhanden:

`ruby.h, C-Bibliotheken von Ruby-Funktionen`

  - Embedded Ruby API:

```
void  ruby_init()
```
Immer als erstes aufzurufen

```
void  ruby_options(int argc, char **argv)
```
Kommandozeilenparameter an Ruby senden

```
void  ruby_script(char *name)
```
Name des Scripts setzen

```
void  rb_load_file(char *file)
```
Datei in den Interpreter laden

```
void ruby_run()
```
Script starten

Beispiel

```
#include "ruby.h"
main(int argc, char **argv) {
  /* ... unser Code ...*/
  /* Gelegentlich erforderlich:    */
#if defined(NT)
  NtInitialize(&argc, &argv);
#endif
  ruby_init();
  ruby_script("embedded");
  rb_load_file("start.rb");
  while (1) {
        if (need_to_do_ruby) {
              ruby_run();
        }
  /* Hier unser Anwendungscode ... */
  }
}
```


Ruby-Funktionen in C einbinden

Ruby-Funktionen in C einbinden

- Warum möglich?
  - Ruby selbst ist in C implementiert
  - Die Ruby-Bibliothek ist offen gelegt. Ihre Funktionen und Datenstrukturen können von beliebigen C-Programmen verwendet werden.
  - "ruby.h" und die Ruby-Laufzeitumgebung sind verfügbar
- Warum sinnvoll?
  - Effizienz: Stärken von Ruby in C nutzbar
  - Verständnis: Details des Übergangs C / Ruby
- Generelle Bemerkungen:
  - Die Verwendung von Ruby-Objekten in C erfordert ein gewisses Verständnis für den Aufbau von Ruby selbst.
  - Wichtig ist insbesondere das Umwandeln von Datentypen.
  - C-Extensions anderer Scriptsprachen sind komplizierter!

Ruby-Funktionen in C einbinden

- C-Implementierung einer Ruby-Klasse:

```
class Ministore
  def initialize
    @arr=Array.new
  end
  def add(anObj)
    @arr.push(anObj)
  end
  def retrieve
    @arr.pop
  end
end
```


Ruby-Funktionen in C einbinden

```
#include "ruby.h"
static VALUE t_init(VALUE self)
{  // Funktion zu "initialize"
  VALUE arr;
  arr = rb_ary_new();
  rb_iv_set(self, "@arr", arr);
  return self;
}

static VALUE t_add(VALUE self,
   VALUE anObj)
{  // Funktion zu "add"
  VALUE arr;
  arr = rb_iv_get(self, "@arr");
  rb_ary_push(arr, anObj);
  return arr;
}
//static VALUE t_retrieve(...){}
// gemeinsam an der Tafel
```

```
// Klasse ist globale Konstante:
VALUE cMinistore;

// Registrierung der Methoden:
void Init_Test() {
  cMinistore = rb_define_class(
               "Ministore",
               rb_cObject);
  rb_define_method(
               cMinistore,
               "initialize",
                t_init, 0);
  rb_define_method(
               cMinistore,
               "add",
               t_add, 1);
  // Fall "retrieve": gemeinsam!
}
```


Ruby-Funktionen in C einbinden

- Das Beispiel zeigte:
  - Erzeugen einer neuen Klasse
  - Erzeugen & Registrieren von Methoden, Verbindung mit C-Funktionen zur eigentlichen Arbeit
  - Erzeugen von Ruby-Variablen bzw. -Attributen, Verbindung von Attributen mit Klassen, Abrufen & Verändern von Werten.
- Zugriff auf Variablen (kleine Auswahl):

```
// Liefert "instance variable" zu name
VALUE rb_iv_get(VALUE obj, char *name)
// Setzt/ändert Wert der "instance variable" zu name
VALUE rb_iv_set(VALUE obj, char *name, VALUE value)
// Analog für globale Variablen bzw. Klassenattribute:
VALUE rb_gv_get/set, VALUE rb_cv_get/set
// Erzeugen von Objekten eingebauter Klassen:
VALUE rb_ary_new(), VALUE rb_ary_new2(long length), ...
VALUE rb_hash_new(), VALUE rb_str_new2(const char *src)
```


Ruby-Funktionen in C einbinden

- Der generische Datentyp VALUE:
  - Dahinter verbirgt sich eine Datenstruktur mit Verwaltungsinformation.
  - Bei der Umwandlung in C-Datentypen ist "*typecasting*" erforderlich.
  - Eine Reihe eingebauter Makros und Funktionen hilft dabei.
  - Low-level Beispiele:

```
VALUE str, arr;
RSTRING(str)->len    // Länge des Ruby-Strings
RSTRING(str)->ptr    // Zeiger zum Speicherbereich (!)
RARRAY(arr)->capa    // Kapazität des Ruby-Arrays
```

- Empfehlung:
  - Ausweichen auf *high-level* Makros (s.u.), Ruby-Interna vermeiden!
- Direkte Werte (*immediate values*):
  - Objekte der Klassen Fixnum und Symbol sowie die Objekte true, false und nil werden direkt in VALUE gespeichert; kein Zeiger auf Speicher

Ruby-Funktionen in C einbinden

- Umwandlung zwischen C-Typen und Ruby-Objekten...
  - ...mittels einer Reihe vordefinierter Makros und Funktionen. Beispiele:

```
INT2NUM(int) // Liefert ein Fixnum-bzw. Bignum-Objekt
INT2FIX(int)        // Fixnum (schneller als INT2NUM)
CHR2FIX(char)       // Fixnum
rb_str_new2(char *) // String
rb_float_new(double)// Float
```

```
int NUM2INT(Numeric)// incl. Typenüberprüfung
int FIX2INT(Fixnum) // schneller
unsigned int NUM2UINT(Numeric)          // analog
// usw.
char NUM2CHR(Numeric or String)
char * STR2CSTR(String)                 // ohne Länge
char * rb_str2cstr(String, int *length) // mit Länge
double NUM2DBL(Numeric)
```


Ruby-Funktionen in C einbinden

- Daten zwischen C und Ruby "teilen"?
  - Vorsicht - Ruby-Interna können sich ändern. Versuchen Sie möglichst nicht, die von Ruby genutzten internen Speicherbereiche von C aus (etwa per Pointer) zu verändern oder zu benutzen!
  - Besser: Gezielte Umwandlung incl. Kopieren
  - Bsp.: String in Ruby erlaubt NULL-Zeichen, char * in C nicht!
- Speicherallokation in C vs. Ruby's Garbage Collector
  - Hier stoßen zwei verschiedene Konzepte aufeinander.
  - Anpassung erfordert Handarbeit! Bsp: Kein free() auf Ruby-Objekte!
  - Vorsicht: Ruby's GC kann von C aus angelegte Ruby-Objekte jederzeit löschen, wenn sie nicht ordentlich in Ruby "registriert" sind.
  - Ruby's GC kann umgekehrt auch veranlasst werden, dynamisch erzeugte C-Strukturen bei Bedarf wieder freizugeben.
- Wir werden hier das Thema GC nicht weiter vertiefen.
  - Bei Bedarf: Pickaxe-Buch, Kap. 17 sowie Ruby Dev. Guide Kap. 10.

C-Bibliotheken in Ruby nutzen

Ruby-Funktionen in C einbinden

- Frage / Diskussion:
  - Was wäre alles notwendig, um C-Funktionen einer gegebenen Bibliothek (oder auch einer eigenen C-Objectdatei) von Ruby aus zu benutzen?
  - Beispiel:

```
double get_current_stock_price(
                        const char *ticker_symbol)
```

- Antworten sammeln (Tafel), z.B.:
  - Top-Level Methode eines Ruby-Moduls (etwa: "Stockprice")

```
Stockprice::get_current_stock_price(aString) --> aFloat
```

  - Umwandlungen

```
Ruby-String --> C-String   // Argument
double --> Float-Objekt    // Rückgabewert
```

  - Generierung eines Ruby-Moduls; evtl. Initialisierungen
  - "Wrapper", der die Ruby-Methode auf die Bibliotheksfunktion abbildet.

Ruby-Funktionen in C einbinden

- Die gute Nachricht:
  - **All dies lässt sich weitgehend automatisieren!**
- Bewährtes Hilfsmittel:
  - **SWIG** (*Simplified Wrapper and Interface Generator*) !
  - Ein *OpenSource-Tool*, seit V 1.3 auch mit Ruby-Modus

Ruby-Funktionen in C einbinden

- Vorgehen (einfacher Fall):
  1. SWIG mitteilen, was zu verbinden ist (in Datei "stockprice.i"):

```
%module Stockprice
%{
#include "stockprice.h"
%}
extern double get_current_stock_price(const char *);
```

  2. Mit SWIG eine Wrapper-Datei generieren:

```
$ swig -ruby stockprice.i  # stockprice_wrap.c erzeugt!
```

  3. Makefile generieren, mit Unterstützung von Ruby:

```
$ ruby -r mkmf -e"create_makefile('Stockprice')
# Erzeugt "Makefile" (mit Bezug auf "alle" C-Dateien)
```

  4. Shared Object-Datei erzeugen:

```
$ make        # Generiert shared object "Stockprice.so"
```

  5. Ruby-Anwendung starten

```
# Siehe nächste Seite
```


Ruby-Funktionen in C einbinden

- Neues Modul in Ruby-Anwendung nutzen (Datei stock.rb):

```
#!/usr/bin/env ruby

require "Stockprice"       # Ruby findet die *.so-Datei!

def get_prices( symbols )  # Beispiel für eine Nutzung
  prices = {}
  symbols.each do |symbol| # C-Funktion anwenden:
   prices[symbol] =
       Stockprice::get_current_stock_price(symbol)
  end
  prices
end
```

- Bemerkungen:
  - Ganz natürliche Verwendung, wie Ruby-Modul!
  - Vergleiche auch die Funktionen des Moduls "Math"

Ruby und Windows

Beispiele für Ruby-Extensions:

Win32API und Win32OLE

Win32API

- Extension-Modul speziell für MS-Windows
- ***Low-level*** Zugriff auf alle Funktionen aus dem Win32 API.
- Hinweise:
  - Viele dieser Funktionen benötigen oder liefern einen Zeiger auf einen Speicherbereich. Ruby verwaltet Speicherblöcke über Objekte der Klasse String.
  - Geeignetes Umcodieren von bzw. in brauchbare Darstellungen bleibt dem Anwender überlassen - z.B. mittels "pack" / "unpack".
- 1 Klassenmethode:

```
Win32API.new( dllname, procname, importArray, export )
```

dllname	z.B. "user32", "kernel32"
procname	Name der aufzurufenden Funktion
importArray	Array von Strings mit Argumenttypen
export	String mit Typ des Rückgabewerts

Win32API

- Typencodes:
  - (Klein- wie auch Großbuchstaben sind zulässig)

"n", "i"	Zahlen,
"p"	Zeiger auf (in Strings gespeicherte) Daten,
"v"	Void-Typ (nur Fall "export").

- 1 normale Methode:

```
call / Call ( [args]* ) ==> anObject
```

  - Die Anzahl und Art der Argumente wird von "`importArray`", die Objektklasse wird von "`export`" in `new()` bestimmt.

Win32API

- Beispiel:

```
require 'Win32API'

getCursorPos =
   Win32API.new("user32", "GetCursorPos", ['P'], 'V')

lpPoint = " " * 8   # Platz für zwei 'long'-Plätze
getCursorPos.Call( lpPoint )

x, y = lpPoint.unpack("LL")       # Decodieren
print "x: ", x, "\ty: ", y, "\n"  # Ausgeben
```

- Demo:
  - win32api01.rb (erweitert, mit Schleife)

Win32API

- Beispiel (Forts.):

```
ods = Win32API.new("kernel32", "OutputDebugString",
                   ['P'], 'V')
ods.Call( "Hello, World\n" )

GetDesktopWindow =
   Win32API.new("user32", "GetDesktopWindow", [], 'L')
GetDesktopWindow =
   Win32API.new("user32", "GetDesktopWindow", [], 'L')
GetActiveWindow =
   Win32API("user32", "GetActiveWindow", [], 'L')
SendMessage =
   WinAPI32("user32", "SendMessage", ['L'] * 4, 'L')

SendMessage.Call(GetDesktopWindow.Call, 274, 0xf140, 0)
```

  - Bem.: Wörtlich aus Quelle übernommen, aber nicht brauchbar!?

Win32API

- Eigenes Beispiel: Einfache Töne

```
PlayNote =
   Win32API.new("kernel32", "Beep", ['I', 'I'], 'V')
PlayNote( 440, 500 ) # Kammerton A, 500 ms lang
```

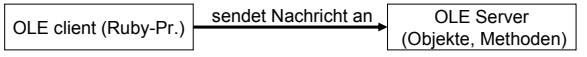
- Eigenes Beispiel: MessageBox

```
GetDesktopWindow =
   Win32API.new("user32", "GetDesktopWindow", [], 'L')
MBox = Win32API.new("user32", "MessageBox",
                    ['L', 'P', 'P', 'L'], 'L')
rc = MBox.call( GetDesktopWindow.call,
                    "Meine Nachricht", "Boxtitel", 1 )
```

- Hal's Beispiel: *Unbuffered character input*
  - Problem:
    Eingabe eines Passwords am Bildschirm verdecken,
    indirektes Echo von "*" statt der gedrückten Tasten.
  - Demo dazu:
    pw_prompt.rb

Win32OLE

- ***High-level*** Zugriff auf Win 32 OLE Automation Server
  - Ruby ersetzt hier z.B Visual Basic (VB) oder den Windows Scripting Host.
  - Ruby tritt an die Stelle eines OLE *clients,* d.h. OLE Server wie MS Excel, Word, PowerPoint etc. können oo. "ferngesteuert" werden:



- Mapping der OO-Modelle: (vgl. Code-Beispiele unten)
  - Methoden eines OLE Servers werden über gleich lautende Ruby-Methoden aktiviert,
  - Parameter der OLE-Objekte durch Abfragen & Setzen entsprechender Ruby-Attribute (genauer: Getter/Setter) kontrolliert.
  - Zugriff auf Eigenschaften erfolgt per Hash-Notation.

Win32OLE

- Hinweise:
  - OLE-Objekte und Eigenschaften verrät die On-line Hilfe von VBA.
  - Tipp:
    - VBA-Makro mit OLE-Server wie z.B. Excel aufzeichnen,
    - Einzelheiten ggf. in VB-Onlinedoku nachschlagen,
    - dann mit Ruby nachbauen / automatisieren.
  - Ruby-Methoden werden wie üblich klein geschrieben, auch wenn die entsprechenden Windows-Objekte gross geschrieben werden.

Win32OLE

- Einfaches Beispiel:
  - IE starten, eingestellte *homepage* anzeigen lassen

```
require 'win32ole'

ie = WIN32OLE.new( 'InternetExplorer.Application' )
ie.visible = true
ie.gohome
```

  - **Demo mit irb!**

- **Konvention** für benannte Argumente

VB-Def.:	`Song(artist, title, length):`
VB-Aufruf:	`Song title := 'Get it on';`
Ruby-Aufruf, simpel:	`Song( nil, 'Get it on', nil)`
Ruby-Aufruf, smart:	`Song( 'title' => 'Get it on' )`

  - Vorteile: Frei von spezieller Reihenfolge, selbst-dokumentierend.

Win32OLE

- **OLE Demo mit Excel**
  - Vgl. Demo-Datei "08/win32ole01.rb und das Pickaxe-Buch, Kap. "Ruby and MS Windows", S. 168f.
- **Optimierungshinweise**

```
# Vorsicht - ineffizient:
workbook.Worksheets(1).Range("A1").value = 1
workbook.Worksheets(1).Range("A2").value = 2
workbook.Worksheets(1).Range("A3").value = 4
workbook.Worksheets(1).Range("A4").value = 8
```

```
# Besser so:
worksheet = workbook.Worksheets(1)
worksheet.Range("A1").value = 1
# usw.
worksheet.Range("A5").value = 8
```


GUI-Programmierung

Besonderheiten der GUI-Programmierung
GUI-Bibliotheken unter Ruby
Schwerpunkt: FXRuby

GUI-Toolkits: Ereignisse und ihre Quellen

- Keyboard
  - Elementar: Drücken einer bestimmten Taste (etwa: "Linke Ctrl-Taste"), Loslassen derselben
  - Zusammengesetzt: Eingabe eines Zeichens: "u", Ctrl-C, { (AltGr-7), Ctrl-Alt-A
- Maus
  - Elementar: Drücken der linken Maustaste, Loslassen derselben, Elementarbewegung
  - Zusammengesetzt: Klick, Doppelklick, "drag" -Bewegung
- Timer
  - Signal nach Ablauf einer Frist
- Scheduler
  - Signal zu bestimmten Zeiten oder infolge anderer Ereignisse (Verkettung)
- System
  - Software- und Hardware-Interrupts, Signale zwischen Prozessen

GUI-Toolkits: Leitgedanken zur Programmierung

- Fensteraufbau
  - Widgets: Die Gestaltungselemente
  - Beispiele: Menü, M-Eintrag, Button, MessageBox, FileDialog, RadioButton, TextItem, TextInput, DirTree, ...
- Widget-Gestaltung
  - Festlegung des konkreten Aussehens
  - Beschriftung, Hintergrundfarbe, Icon, Länge/Breite, ...
  - I.d.R. über Attribute gesteuert
- Layout
  - Anordung der Widgets in Relation zum Fenster bzw. zueinander, explizit vs. dynamisch.
  - Ressourcen-Editor vs. Layout-Manager & "Hints"

GUI-Toolkits: Leitgedanken zur Programmierung

- Aktionen
  - Die ausgewählten und platzierten Widgets kümmern sich selbständig um ihr Aussehen, auch beim Eintreffen von Ereignissen.
  - Die gewünschten Aktionen aufgrund erwarteter Ereignisse werden i.d.R. von Methoden benutzerspezifischer Klassen ausgeführt.
  - Diese Methoden müssen mit den Ereignissen und ihren Quellen verbunden werden!

GUI-Toolkits: Leitgedanken zur Programmierung

- Andere Sicht der Dinge
  - Ihr Programm "agiert" nicht mehr - es reagiert (auf Ereignisse).
  - stdin, stdout, stderr verlieren an Bedeutung, Aspekte der Parallelverarbeitung (insb. Threads) treten hinzu.
- Vorsicht vor der Fülle
  - Die Vielzahl an Gestaltungsoptionen lenkt leicht vom Wesentlichen ab.
  - GUI-Toolkits enthalten zahlreiche Widgets und Hilfskonstrukte. Diese stehen in enger Wechselwirkung.
  - Objekt-orientiertes Design ist hier besonders wichtig (und wird verbreitet eingesetzt), um noch den Überblick zu behalten. OOP sollte daher sicher beherrscht werden

GUI-Toolkits: Leitgedanken zur Programmierung

- Event Loop, Event Queue
  - Ihr Programm ist nur eines von mehreren, das auf Ereignisse wartet.
  - Die zentrale Verteilung der Ereignisse übernimmt der *window manager*. Ihr Programm muss sich dort an- und abmelden. Missachtung verursacht z.B. "Trümmer" auf dem Desktop.
  - Ereignisse werden in der Reihenfolge ihres Eintreffens abgearbeitet, und zwar vom *event loop*.
  - Sie werden ggf. serialisiert und per Warteschlange verwaltet, wenn ihre Bearbeitung lange dauert.
- Kooperatives Multitasking?
  - Auch *preemptive multitasking* Ihres Betriebssystems bewahrt Sie nicht vor Blockaden im *event queue*. Daher erfordern lang dauernde Aktionen besondere Techniken, etwa Abarbeitung in eigenen *threads*.

GUI-Toolkits für Ruby

- Auswahlkriterien
  - Plattformübergreifende Verfügbarkeit?
  - Lizenzfragen: Proprietär? OpenSource? Auch kommerziell einsetzbar?
  - Look & feel: consistent vs. native
  - "advanced widgets"
  - Integration in Ruby?
  - (Vergleichsweise) Einfach anwendbar
  - doc? stability? availability? OpenGL support?
  - Entwicklungsstand des Ruby-Bindings?
  - Dokumentationsstand: Toolkit selbst? Ruby-API?

GUI-Toolkits für Ruby

- Ruby/Tk
  - Der Noch-Standard! Sehr ähnlich zu Perl/Tk
  - Hoher Verbreitungsgrad, gute Dokumentation
  - Standard Widget-Set "mager", aber erweiterbar
- GTK+ Binding
  - Das Toolkit hinter Gnome! Schwerpunkt daher: Linux
  - Unter Windows ist die Verfügbarkeit & Stabilität noch problematisch
- Qt Binding
  - Das Toolkit hinter KDE! Schwerpunkt daher: Linux
  - Ruby-Binding offenbar noch in den Anfängen
- SWin / VRuby
  - Nur unter Windows, da auf Win32API aufbauend
- Andere:
  - FLTK, curses(!), native Xlib, wxWindows (Python), Apollo (Delphi), Ruby & .NET, JRuby und "swing"
- FOX: "Free Objects for X" / FXRuby: Siehe unten!

FOX und FXRuby

- Warum FOX?
  - Relativ einfach und effizient
  - Open Source (GPL)
  - Modernes look&feel, viele leistungsfähige Widgets
  - OpenGL-Unterstützung für 3D-Objekte
  - Plattform-übergreifend
  - Vereinfachung durch sinnvolle Defaults
  - Start 1997 - hat aus Designschwächen anderer gelernt
- FOX und FXRuby
  - Erschließung des FOX-API als Ruby-Modul via SWIG, dabei Weitergabe der FOX-Vorzüge an Ruby
  - Ruby-spezifische Ergänzungen zur besonders einfachen Integration im Ruby-Stil, etwa: "connect"

FXRuby: Plus und Minus

- Start von FXRuby in 2001
  - Sehr stabil und brauchbar für sein geringes Alter
  - Gute Portierung nach Windows
  - Dokumentation leider noch sehr lückenhaft!
- C++ vs. Ruby
  - Eine gewisse Sprachanpassung ist erforderlich. Diese hat seit 2001 deutliche Fortschritte gemacht, ist aber noch nicht abgeschlossen. Beispiel:
  - Bitoperation - in C/C++ typisch - sind auch in FXRuby erforderlich, hier aber kein üblicher Stil.
- Warum FXRuby?
  - Demos: groupbox, glviewer !

FOX und FXRuby

- Dokumentation zu FOX
  - www.fox-toolkit.org
- Dokumentation zu FXRuby
  - User Guide:
    `http://www.fxruby.org/doc/book.html`,
    `c:\Programme\ruby\doc\FXRuby\doc\book.html`
  - API Dokumentation (erst im Entstehen, nur on-line):
    `http://www.fxruby.org/doc/api`
  - Beispiel-Code:
    `(im User Guide, und in der Windows-Installation)`
    `c:\Programme\ruby\samples\FXRuby\*.rb?`
  - Buch-Beispiele (leider alle nicht mehr aktuell):
    `Pickaxe-Buch: Wenig; Fulton: mehr;`
    `"Dev. Guide": ähnlich.`

FXRuby: *Basics*

FXRuby: Basics

- Grundgerüst einer FXRuby-Anwendung


```
require "fox"
include Fox

app = FXApp.new( "Autor", "Firma / Quelle" )
app.init( ARGV )
main = FXMainWindow.new( app, "Titel" )

app.create
main.show( PLACEMENT_SCREEN )
app.run
```

- Beobachtungen / Demo:
  - Standard-Fensterfunktionen vorhanden, incl. "Resize"

FXRuby: Basics

- Kommentare:

```
# Die FXApp-Klasse verwaltet viele Gemeinsamkeiten der
# Fenster und Widgets. Sie ist der "Ausgangspunkt":
app = FXApp.new( "Autor", "Firma / Quelle" )
# Nicht essentiell:
app.init( ARGV )
# Das übliche Top-Level Fenster, von app ableiten:
main = FXMainWindow.new( app, "Titel" )

# Nun wird die Anwendung "startklar" gemacht:
app.create
# Fenster sichtbar machen, mit Angabe wo:
main.show( PLACEMENT_SCREEN )
# Alternativ: PLACEMENT_CURSOR / _OWNER / _MAXIMIZED
# Schließlich: In event loop einschleusen...
app.run
```


FXRuby: Model-View-Controller Ansatz

```
require "fox"
include Fox

class MyMainWindow <
            FXMainWindow

  def initialize( *par )
    super( *par )
    # Hier neue Widgets!
  end

  def create
    super
    show( PLACEMENT_SCREEN)
  end
end
```

```
class MyController
  def initialize
    @app = FXApp.new( "Autor",
           "Firma / Quelle" )
    @app.init( ARGV )
    @main = MyMainWindow.new(
             @app, "Titel" )
    @app.create
  end

  def run
    @app.run
  end
end


MyController.new.run
```

FXRuby: Model-View-Controller Ansatz

- M-V-C Konzept: Trennung zwischen
  - Datenmodell (z.B. der Baum der Registry-Knoten)
  - seiner Darstellung (hier: Fenster in der GUI)
  - und Klassen zur Steuerung der Abläufe
- Im vorliegenden Beispiel:
  - "Model": Nicht vorhanden
  - "View": MyMainWindow-Objekt
  - "Controller": Controller-Objekt

FXRuby: Basics

- **Ein Widget hinzufügen (Button)**


```
# Weitere Widgets ...
FXButton.new( self, "Ein langer &Knopftext" )
```

- Beobachtungen (Demo):
  - Der Knopf wird automatisch in das Fenster übernommen.
  - Er reagiert (auch auf 'Alt-K'), allerdings noch ohne Wirkung
  - Layout-Management: automatisch!
    Knopf linksbündig, Breite vom Text bestimmt
    Fensterbreite vom Titelbalken bestimmt
- **Weitere Widgets hinzufügen**

```
FXButton.new( self, "Ein zweiter K&nopf,\nmit zweiter
   Zeile\tDieser Knopf zeigt Tooltip-Text" )
FXTooltip.new( self.getApp )
```

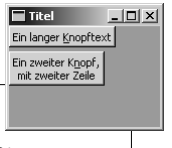
- Beobachtungen (Demo):
  - 2. Knopf wird linksbündig und unter dem ersten dargestellt, mit zweizeiliger Beschriftung. "Tooltip-Text" bei Mausberührung.

FXRuby: Basics

- Variante in der Fenstersteuerung:


```
# Nur Titel und "Close"-Button:
@main = MyMainWindow.new( @app, "Titel", nil, nil,
       DECOR_TITLE | DECOR_CLOSE
```

- Beobachtungen (Demo):
  - "Iconify" und "Maximize"-Kontrollflächen sind verschwunden.
  - Fenstergröße ist nicht mehr änderbar.
  - Systemmenü ist entsprechend "ausgegraut".
- Gestaltung des Basisfensters:
  - Größe festlegen, Hintergrundfarbe verändern durch Setzen von Attributen:




```
self.width = 300
self.height = 200
self.backColor = FXRGB(100, 200, 50)
```


FXRuby: Basics

- Ergänzung eines Menüs (Demo):

```
# Menu bar, along the top
  @menubar = FXMenubar.new( self,
                  LAYOUT_SIDE_TOP|LAYOUT_FILL_X )
# File menu
  @filemenu = FXMenuPane.new( self
  FXMenuTitle.new( @menubar, "&File",
                   nil, @filemenu )
  FXMenuCommand.new( @filemenu,
      "&Quit\tCtl-Q\tQuit the application.")
# Help menu, on the right
  helpmenu = FXMenuPane.new(self)
  FXMenuTitle.new(@menubar, "&Help", nil,
                  helpmenu, LAYOUT_RIGHT )
  aboutCmd = FXMenuCommand.new( helpmenu,
              "Über &Demo...\t\tBeispieltext." )
```

FXRuby: Ereignisse

- Das *message/target*-Konzept von FXRuby
  - Eine Nachricht besteht aus Nachrichten-Typ und -ID
  - Beispiele

    SEL_COMMAND

    Ein Nachrichten**typ**, der anzeigt, dass z.B. ein Knopf angeklickt wurde

    FXWindow::ID_SHOW, FXWindow::ID_HIDE

    Identifier, diejedes Fenster versteht, und die ihm mitteilen, sich (un)sichtbar zu machen.

    FXApp::ID_QUIT

    Identifier, den FXApp-Objekte verstehen und sich daraufhin beenden.
  - Selektor

    Aus historischen Gründen werden Typ und ID zu einem 32-bit-Wert gebündelt, dem *"selector"*

FXRuby: Ereignisse

- Ereignisse
  - Ereignisse haben einen Sender, einen Selektor und (optionale) Begleitdaten.
  - Entwickler legen fest, welches Objekt auf ein bestimmtes Ereignis reagieren soll.
  - Im Nachrichtenbild: Sie legen den Empfänger fest!
- Beispiele
  - Klicken auf einen Knopf
  - Auswahl eines Menüpunktes
- Traditionelles FOX-Schema
  - Zuordnungstabelle anlegen (etwas umständlich)
- Neues, Ruby-gemäßes Schema
  - Mit iterator-artiger Methode "**connect**"

FXRuby: Ereignisse

- Beispiel: Verbindung Knopfklick mit FXApp#exit
  - Zuordnen des ersten Knopfes, per Parameter:

```
FXButton.new( self, "Hier &klicken zum Beenden",
              nil, getApp, FXApp::ID_QUIT )
```

- Implizite Aussage:
  - Auf Knopfklick (bewirkt Typ SEL_COMMAND), soll dieser Sender eine Nachricht mit ID=FXApp::ID_QUIT an Empfänger (Ergebnis der Methode getApp()) senden.

FXRuby: Ereignisse

- Beispiel: Verbindung Knopfklick mit FXApp#exit
  - Nachträgliche explizite Zuordnung per connect-Methode

```
bq = FXButton.new( self,
                   "Hier &klicken zum Beenden" )
```

```
# Verbinde Empfänger bq mit Nachrichtentyp SEL_...
bq.connect( SEL_COMMAND ) do |snd, sel, data|
   exit      # exit ist eine Methode von FXApp!
end
```

```
# oder kürzer:
bq.connect( SEL_COMMAND ) { exit }
```

FXRuby: Ereignisse

- Analog: Verbindung von File/Quit mit FXApp#exit

```
FXMenuCommand.new( @filemenu,
      "&Quit\tCtl-Q\tQuit the application.",
      nil, getApp, FXApp::ID_QUIT)
```

  - Nachträgliche explizite Zuordnung per connect-Methode

```
quitCmd = FXMenuCommand.new( @filemenu,
      "&Quit\tCtl-Q\tQuit the application." )
# Verbinde Empfänger bq mit Nachrichtentyp SEL_...
quitCmd.connect( SEL_COMMAND ) { exit }
```


FXRuby: Ereignisse

- Beispiel: Verbindung von Help/About mit Block

```
aboutCmd = FXMenuCommand.new( helpmenu,
                "Über &Demo...\t\tBeispieltext." )
# Verbinde Empfänger mit Nachrichtentyp SEL_...:
aboutCmd.connect( SEL_COMMAND ) do
   FXMessageBox.information( self, MBOX_OK,
     "Über Demo",
     "HWW: Kleine Beispiele\nCopyright (c) 2003 :-)" )
end
```

  - **Demo!**
  - Falls genug Zeit:
    Mehr "Action" mit der **Keyboard-Demo**

FXRuby: Layout Management

1. Das Hauptfenster wird mittels spezieller Layout-Managerobjekte unterteilt, z.B. so:

```
top = FXHorizontalFrame.new( self, LAYOUT_SIDE_TOP | )
      LAYOUT_FILL_X | LAYOUT_FILL_Y )
bottom = FXHorizontalFrame.new(self, LAYOUT_SIDE_BOTTOM)
lowerLeft = FXVerticalFrame.new(top,
   LAYOUT_SIDE_LEFT | PACK_UNIFORM_WIDTH)
lowerRight = FXVerticalFrame.new(top,
   LAYOUT_SIDE_RIGHT)
```

- LAYOUT_SIDE_TOP / BOTTOM / LEFT / RIGHT:
  - Anwahl der jew. Seite der Unterteilung
- LAYOUT_FILL_X / Y:
  - Ausdehnung in Richtung X/Y bei Fenstergrößenänderung
- PACK_UNIFORM_WIDTH:
  - Gleiche Breite für Widgets in diesem Rahmen

FXRuby: Layout Management

2. Widgets platziert man nun in die Teilbereiche, indem man die jeweiligen Layoutmanager-Objekte anstelle des Hauptfensters als Elternobjekte verwendet:

```
button = FXButton.new( lowerLeft, "&Beenden" )
```

3. Dekor

   Zur optischen Betonung der Unterteilungen gibt es Trennlinien-Objekte:

```
FXHorizontalSeparator, FXVerticalSeparator
```

4. Weitere Layout-Manager (Bsp.):
   - FXSplitter
     `Generische Aufteilung`
   - FX4Splitter
     `2x2-Aufteilung, je ein Objekt pro Quadrant`
   - FXMatrix
     `n x n-Aufteilung, wahlweise zeilen- oder spaltenweise Befüllung`

FXRuby: Layout Management

- Erläuterungen am Demo-Beispiel foursplit.rbw:

FXRuby: Layout Management

- Beobachtungen
  - Dynamische Anpassung der Widgets
  - Tooltip-Wirkung
  - Unterschiedliche Wirkungen beim Verschieben der Grenzen
  - Weitere Beispiele für Menüs
  - "Status bar": Zusatztext hinter \t\t aus den Menüs dort!
- **Fazit:**

  Kein Ressourcen-Editor erforderlich!
- Optionale Demo: splitter.rbw

FXRuby: Grundlegende Widgets

- Gruppierung von Objekten

```
group2 = FXGroupBox.new( refFrame, "Beschriftung",
    FRAME_RIDGE)     # Alternativ etwa: FRAME_GROOVE
# Nun Gruppenobjekte von "group2" ableiten...
```

- RadioButtons

```
rBut1 = FXRadioButton.new( group2, "HR &1" )
rBut2 = FXRadioButton.new( group2, "&Deuschlandfunk" )
```

```
# Auf Anwahl reagieren:
rBut1.connect(SEL_COMMAND) { ctrl.channel = 1 }
rBut2.connect(SEL_COMMAND) { ctrl.channel = 2 }
# etc.
```

- Auswahlboxen

```
sel = FXComboBox.new(group2, width, 3, nil, 0,
    COMBOBOX_INSERT_LAST|FRAME_SUNKEN|LAYOUT_SIDE_TOP)
["Wahl 1", "Wahl 2", "Wahl 3"].each do |i|
    sel.appendItem(i); end          # Befüllung
```

```
sel.currentItem = 1                 # Default setzen
```

```
sel.getItemData(sel.currentItem)  # Abruf der Auswahl!
```


FXRuby: Grundlegende Widgets

- Text-Ein/Ausgabe

```
txtCtrl = FXText.new( group3, nil, 0,
                      LAYOUT_FILL_X|LAYOUT_FILL_Y )
```

```
txtCtrl.text = ''                   # Textfeld löschen
# ...
```

```
txtCtrl.text = "Initial text\n2nd line" # Belegen
# ...
txtCtrl.text += "some more text"        # Anfügen
```

```
txtCtrl.connect( SEL_COMMAND, method(:onCmdText )
```

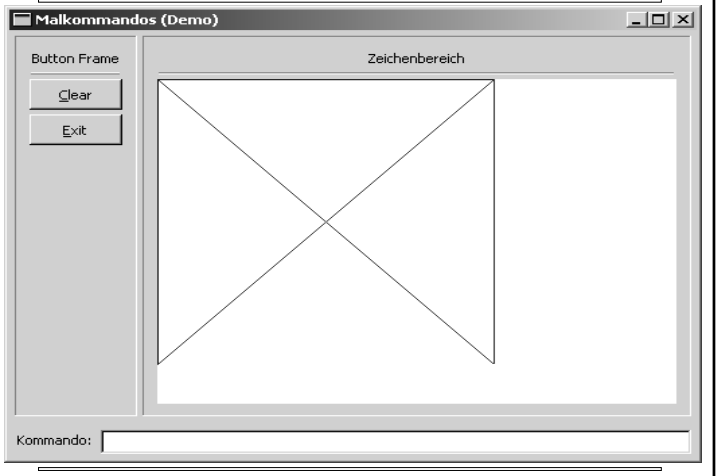
```
# Controller-Methode zur Texteingabe:
def onCmdText( sender, sel, data )
   # Text in data nutzen...
end
```

- Demo dazu: draw_cmd.rb
  - Darin enthalten: Canvas-Demo, hier nicht mehr behandelt
  - Quelldatei dazu im Verzeichnis zu Aufgabe 10 !

FXRuby: Grundlegende Widgets

- Fragen aus der Demo:
  - Umgang mit "Canvas"?
    `Repaint-Aktivitäten, Konsequenzen beim Abschalten`
  - Kommandozeilen-Interpreter:
    `Wie implementiert man ihn möglichst einfach?`
  - Dynamische Erweiterbarkeit
    `Hinzufügen der Kommandos "move" und "quit"`
    `Wie funktioniert's ?`
- Überleitung zum Abschnitt "*Reflection*"

FXRuby: Canvas / Textbox-Demo "draw_cmd"



FXRuby: Moderne Widgets

- TreeView, DirList, Table, ...
- Empfehlung:
  - **Studium der folgenden FXRuby-Beispiele:**
  - **bounce**
  - **browser**
  - **dctest**
  - **(glviewer)**
  - **(groupbox)**
  - **scribble**
  - **tabbook**
  - **table**

FXRuby: Tipps für die Praxis

- User Guide erarbeiten,
- "Sample"-Programme sichten, ähnliche Fälle heraussuchen.
- Diese analysieren, dabei die Doku (API, notfalls FOX) einsetzen.
- Sample-Fälle variieren, auf eigene Bedürfnisse anpassen.

"Dynamisches" Programmieren

Selbstauskünfte mittels ObjectSpace,
methods, respond_to?, id/class/kind_of?/...
superclass, ancestors
Ein Klassenbrowser
Methoden dynamisch aufrufen

Marshaling & DRb

Marshaling:
- Grundlagen
- Hinweis auf xDBM

Distributed Ruby:
- Verteilte Objekte - ganz einfach

Marshaling

- Die Aufgabe
  - Wir haben (komplizierte) Objekte aufgebaut.
  - Diese sollen nun auch außerhalb des Kontexts des laufenden Programms zur Verfügung stehen - ohne komplette Neuberechnung.
- Beispiele
  - Speichern von Objekten, zwecks späterer Weiterverarbeitung
  - Übertragung von Daten auf ein anderes DV-System in plattformunabhängiger Weise
  - Client/Server-Anwendungen mit verteilten Objekten
- Der Lösungsansatz:
  - "*Marshaling*" der Objekte
  - Auch genannt: "Objekt-**Persistenz**". Java-Begriff: "**Serialisierung**"

Marshaling

- Einfaches *Marshaling*: Das Modul "Marshal"

```
# Ein nicht-triviales Objekt aus weiteren Objekten:
geom_objects = [My::Rect.new(4, 5), My::Square.new(5),
                My::Circle(3), ... ]
# Das wollen wir speichern in Datei "my_obj_store.dat":
File.open("my_obj_store.dat","w") do |file|
   Marshal.dump(geom_object, file)
end
```

```
# Später / u.U. anderes Programm, andere Plattform:
# Objekt rekonstruieren:
File.open("my_obj_store.dat") do |file|
   my_obj_array = Marshal.load(file)
end
# Weitere Verwendung ...
```

Marshaling

- Die Methoden Marshal.dump und Marshal.load
  - Anstelle der IO-Objekte nehmen die Methoden auch Objekte an, die auf "**to_str**" ansprechen. Dazu zählen insbesondere String-Objekte.
  - Synonym zu Marshal.load: **Marshal.restore**
- Nicht speicherbare Objekte
  - Es gibt einige Objekte, deren Natur sich nicht zum *Marshaling* eignen. In Ruby sind das Objekte der Klassen:
    `IO, Proc, Binding`
  - Ebenso nicht serialisierbar: *Singleton*-Objekte

Marshaling

- Eigene Eingriffe: Die Methoden _dump und _load
  - Situation:
    Nicht alle Teile eines Objekts sollen gespeichert werden, z.B. solche, die sich leicht rekonstruieren lassen.
  - Lösung:
    Methoden _dump und _load in betroffene Klassen implementieren.

```
_dump( depth ) # Erzeugt String mit gewünschten Inform.
               # depth: Max. Verschachtelungstiefe.
_load(aString) # Klassenmethode; erzeugt Objekt aus
               # einem String, der von _dump stammte
```

- Beispiel:
  - Die Klasse **My::Rect** speichere nicht nur die **Kantenlängen**, sondern auch noch **Fläche** und **Umfang** des Rechtecks in Attributen.
  - Letztere lassen sich aber einfach rekonstruieren und müssen daher nicht mitgespeichert werden.

Marshaling

- _dump und _load am Beispiel My::Rect

```
class Rect          # In Modul "My"...
  attr_reader :a, :b, :area, :circumference
  def initialize( a, b )   # a, b: Hier nur "Integer"
    @a, @b = a, b
    @area = a * b
    @circumference = 2*(a+b)
  end
# weitere Methoden...
  def _dump
    @a.to_s+':'+@b.to_s
  end
  def _load( str )
    a, b = str.split(':').collect{|x| x.to_i}
    Rect.new( a, b )
  end
end
```


Marshaling

- *Marshaling* mit der Klasse **PStore**
  - Mehrere Objektsammlungen simultan in einer Datei
  - Transaktionsschutz, mit Methoden **abort** und **commit**.
  - Zugriff auf die Objektsammlungen erfolgt Hash-artig.
    Da die meisten Sammlungen Objekthierarchien sind, spricht man aber von "root" anstelle von "key".
- Beispiel (aus dem Pickaxe-Buch):
  - Serialisierung eines String-Arrays und eines Binärbaum

```
require "pstore"

class T       # Grundlage des Binärbaums im Beispiel
  def initialize( val, left=nil, right=nil )
      @val, @left, @right = val, left, right
  end
  def to_a; [ @val, @left.to_a, @right.to_a ]; end
end
```

Marshaling

```
store = PStore.new("/tmp/store") # R/W-Zugriff
store.transaction do
  store['cities']=['London', 'New York', 'Tokyo']
  store['tree'] =
     T.new( 'top',
      T.new('A', T.new('B'),
      T.new('C', T.new('D', nil, T.new('E'))) )
end   # 'commit' implizit bei normalem Ende!
```

```
# Einlesen:
store.transaction do
  puts "Roots: #{store.roots.join(', ')}"
  puts store['cities'].join(', ')
  puts store['tree'].to_a.inspect
end
```


Marshaling

```
# Ergebnis:
Roots: cities, tree
London, New York, Tokyo
["top", ["A", ["B", [], []], []], ["C", ["D", [],
   ["E", [], []]], []]
```

- Übersicht zu den PStore-Methoden
  - Klassenmethoden:
    `new`
  - Normale Methoden:
    `[], []=, roots, root?,`
    `path,`
    `abort, commit,`
    `transaction`

Marshaling

- *Marshaling* mit DBM (und "Verwandten")
  - Auf Unix-Systemen gibt es seit langem eine einfache Datenbank-Vorstufe unter dem Namen "dbm".
  - Mittels "dbm" lassen sich prinzipiell beliebige Datenstrukturen einem Suchschlüssel zuordnen und über diesen Schlüssel persistent speichern sowie effizient wiederherstellen.
  - Dies entspricht dem Verhalten einer persistenten Hash-Tabelle! Perl hatte daher "dbm" mit einem einfachen Hash-artigen, transparenten Zugriffsmechanismus versehen.
  - Ruby folgte mit Klasse "DBM" dieser Tradition!

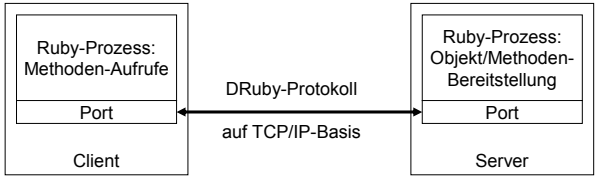
Marshaling

- *Marshaling* mit DBM (und "Verwandten")
  - Objekte der Klasse DBM verhalten sich ähnlich wie Hashes, sind aber keine!
  - Bei Bedarf ist eine Umwandlung möglich mit **to_hash.**
  - Einschränkung: *key*- wie *value*-Objekte müssen Strings sein!
  - Konsequenz: ggf. kombinieren mit "Marshal" !
- Beispiel:

```
require "DBM"
d = DBM.new("my_data")      # "my_data.dbm" neu angelegt
d["cities"] = Marshal.dump( my_array_of_city_names )
d["tree"] = Marshal.dump( my_T_obj )
d.close
# Später:
d = DBM.open("my_data")     # Darf nur einmal offen sein!
print Marshal.load( d["cities"] ).join(', ')
d.close       # puts ergibt: "London, New York, Tokyo"
```

Distributed Ruby

- Objekte auf verteilten Systemen sind realisierbar mittels
  - Mechanismen zur Objekt-Persistenz (Marshaling)
  - Netzwerk-Protokollen, insb. TCP/IP
- Ruby beherrscht beides, daher war der Weg zu DRuby nicht mehr fern:



- Ergebnis:
  - Leistungen ähnlich wie elementare CORBA- oder Web Services-Funktionen, aber mit sehr geringem Aufwand!

Distributed Ruby

- Beispiel für ein Server-Programm

```
require "drb"

class OurDRbServer
  def meth1
    "Hello from " + `hostname`.strip
  end
end

aServerObj = OurDRbServer.new
# Dieses Objekt soll nun den Clients dienen:
DRb.start_service('druby://localhost:12349',aServerObj)
# Normales Prozessende verhindern:
DRb.thread.join
```


Distributed Ruby

- Beispiel für ein dazu passendes Client-Programm

```
require "drb"

DRb.start_service
obj = DRbObject.new( nil,
   'druby://servername.domain.tld:12349',
   aServerObj )
# Methoden des entfernten Objekts nun lokal verwendbar:
print obj.meth1     # "Hello from lx3-beam"
# etc. ...
```

- Kriterien zur Verwendbarkeit
  - Ports verfügbar? Keine Kollisionen? Router/Firewall-Aspekte??
  - Nur Methoden eines Objekts pro Port: Ausreichend?
  - Performance: 50 remote calls / sec @ 233 MHz-CPU ok?

System hooks & Co

Erweiterungen durch alias-Verwendung
Eingebaute *hooks*
Tracing
Init- und Exit-Behandlung

System hooks, tracing

- Stichwortsammlung
  - Alias-Technik:
    Wrapper um existierende Methode schreiben, indem die Methode umbenannt wird in einen privaten Namen und der Wrapper ihre Nachfolge antritt. Problem Namenskollision...
  - Anwendbar auch auf Systemklassen! Beispiel:
    Überladen von Class.new ermöglicht Verfolgen des Anlegens neuer Klassen
  - Eingebaute Callbacks:

```
Module#method_added
Kernel.singleton_method_added
Class#inherited
Module#extend_object
```

  - Prinzip: o.g. Methoden implementieren, um beim entsprechenden Ereignis "aufgerufen" zu werden.
  - set_trace_func (vieles), trace_var (für Änderungen in globalen Var.)

System hooks, tracing

- Stichwortsammlung
  - BEGIN { ...}, END { ... }
  - Mehrere solche Blöcke können def. werden. Abarbeitung in "natürlicher" Reihenfolge - BEGIN: FIFO, END: LIFO
- Aktivitäten bei "exit"
  - Exception "SystemExit" - kann abgefangen werden!
  - Kernel-Funktionen at_exit()
  - *Object finalizers*: Proc, wird vor Löschung eines Objekts (durch GC) aufgerufen.
  - Schließlich: END {...}
- Umgang mit Signalen; *signal handlers*