GUI-Programmierung

Besonderheiten der Programmierung grafischer Benutzeroberflächen (GUI)

GUI-Bibliotheken unter Ruby

Schwerpunkt: FXRuby

GUI-Toolkits: Ereignisse und ihre Quellen

- Keyboard
  - Elementar: Drücken einer bestimmten Taste (etwa: "Linke Strg-Taste"), Loslassen derselben
  - Zusammengesetzt: Eingabe eines Zeichens: "u", Strg-C, { (AltGr-7), Strg-Alt-A
- Maus
  - Elementar: Drücken der linken Maustaste, Loslassen derselben, Elementarbewegung
  - Zusammengesetzt: Klick, Doppelklick, „zieh“ -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:

    ```
    Menu, MenuEntry, 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
  - Anordnung 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.
    - Beispiel: Button „versinkt“ beim Anklicken
  - 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). Oft werden sog. Callback-Methoden zu registrieren sein, die vom Windows-System je nach Ereignis aufgerufen werden.
  - **`stdin, stdout, stderr`** verlieren an Bedeutung.
  - Aspekte der Parallelverarbeitung (insb. *threads*) treten hinzu.

- Vorsicht vor der Fülle
  - Die Vielzahl der 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 (*event queue*) 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 Skriptsprachen

- Auswahlkriterien
  - Plattformübergreifende Verfügbarkeit?
  - Lizenzfragen: Proprietär? *Open Source*? Auch kommerziell einsetzbar?
  - *Look & feel: consistent vs. native*
  - "*advanced widgets*"
  - Integration in die gewünschte Skriptsprache?
  - (Vergleichsweise) Einfach anwendbar?
  - Dokumentation? Stabilität? Verfügbarkeit? Unterstützung von OpenGL?
  - Entwicklungsstand des *bindings*?
  - Dokumentationsstand:
    - Toolkit selbst? Skriptsprachen-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* (*Lesser GPL*)
  - Modernes *look&feel*, viele leistungsfähige *widgets*
  - OpenGL-Unterstützung für 3D-Objekte
  - Plattform-übergreifend
  - Vereinfachung durch sinnvolle Defaults
  - Start 1997 - konnte aus Designschwächen anderer *toolkits* lernen

- 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
  - Inzwischen sehr stabil und brauchbar
  - Gute Portierung nach Windows
  - Dokumentation noch lückenhaft, aber auf gutem Weg
    - 2008: Buch „FXRuby“ von Lyle Johnson, dem Entwickler selbst

- 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
  - **http://www.fox-toolkit.com**

- Dokumentation zu FXRuby
  - User Guide:
    http://www.fxruby.org/doc/book.html,
    lokal z.B. unter:
    c:\Programme\ruby\doc\FXRuby\doc\book.html
  - API-Dokumentation:
    http://www.fxruby.org/doc/api/
  - Beispiel-Code:
    (im User Guide, und in der Windows-Installation)
    c:\Programme\ruby\samples\FXRuby*.rb?
    c:\Programme\ruby\lib\ruby\gems\1.8\gems\fxruby-1.6.12-mswin32\exsamples*.rb?
  - Buch-Beispiele (leider alle nicht mehr aktuell):
    FXRuby. Create Lean and Mean GUIs with Ruby

FOX und FXRuby

- Dokumentation zu FOX
  - **http://www.fox-toolkit.com**

- Dokumentation zu FXRuby
  - User Guide:

    http://www.fxruby.org/doc/book.html,

    lokal z.B. unter:

    c:\Programme\ruby\doc\FXRuby\doc\book.html
  - API-Dokumentation:

    http://www.fxruby.org/doc/api/
  - Beispiel-Code:

    (im User Guide, und in der Windows-Installation)

    c:\Programme\ruby\samples\FXRuby*.rb?

    c:\Programme\ruby\lib\ruby\gems\1.8\gems\fxruby-1.6.12-mswin32\exsamples*.rb?
  - Buch-Beispiele (leider alle nicht mehr aktuell):

    FXRuby. Create Lean and Mean GUIs with Ruby

    Lyle Johnson, The Pragmatic Programmer, 2008

FXRuby: *Basics*

FXRuby: Basics

- Grundgerüst einer FXRuby-Anwendung

```
require "rubygems"
require "fox16"
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

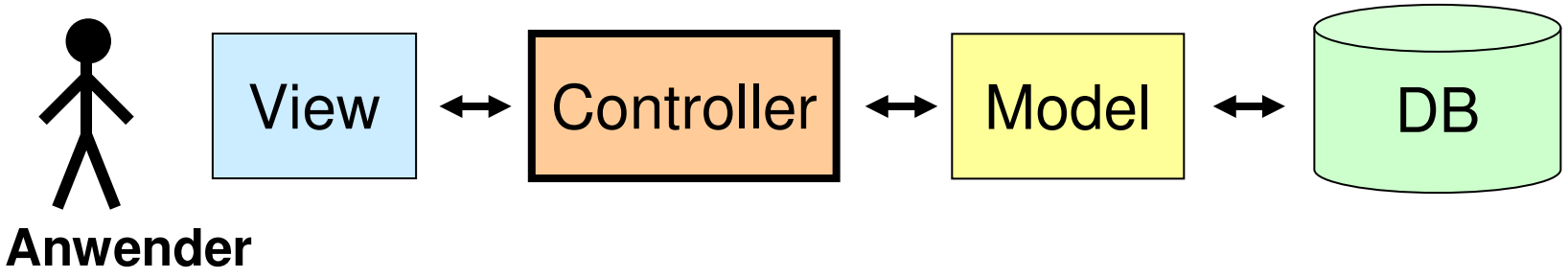
```
require "fox16"
include Fox

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

- MVC-Architekturmuster: Klare Trennung zwischen
  - Datenmodell (*model*, z.B. der Baum der Registry-Knoten)
  - seiner Darstellung (*view*, hier: Fenster in der GUI)
  - und Klassen zur Steuerung der Abläufe (*controller*)

- Im vorliegenden Beispiel:
  - "Model": Hier: Nicht vorhanden
  - "View": **`MyMainWindow`**-Objekt
  - "Controller": **`MyController`**-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, 150, 250)
```

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." )
```

- Bisher erreicht:
  - Menü-Elemente vorhanden, noch ohne Wirkung.

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, die jedes 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 besitzen 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()`, also das zugrunde liegende `FXApp`-Objekt) 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_COMMAND
bq.connect( SEL_COMMAND ) do |snd, sel, data|
    exit        # exit ist eine Methode von FXApp!
end
```

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

  - Lesbarer, mehr *code re-use* mit action-Methoden

```
bq = FXButton.new( self, "Hier &klicken zum Beenden" )
# Verbinde Empfänger bq mit Nachrichtentyp SEL_COMMAND
bq.connect( SEL_COMMAND, method(:do_exit) )
```

```
def do_exit
    exit        # Lohnend bei Mehrzeilern…
end
```

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 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) 2009 :-)" )
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(bottom,
                LAYOUT_SIDE_LEFT | PACK_UNIFORM_WIDTH)
lowerRight = FXVerticalFrame.new(bottom,
                LAYOUT_SIDE_RIGHT)
```

- LAYOUT_SIDE_TOP / _BOTTOM / _LEFT / _RIGHT:
  - Anwahl der jew. Seite der Unterteilung
- LAYOUT_FILL_X / _Y:
  - Ausdehnung in Richtung X bzw. 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

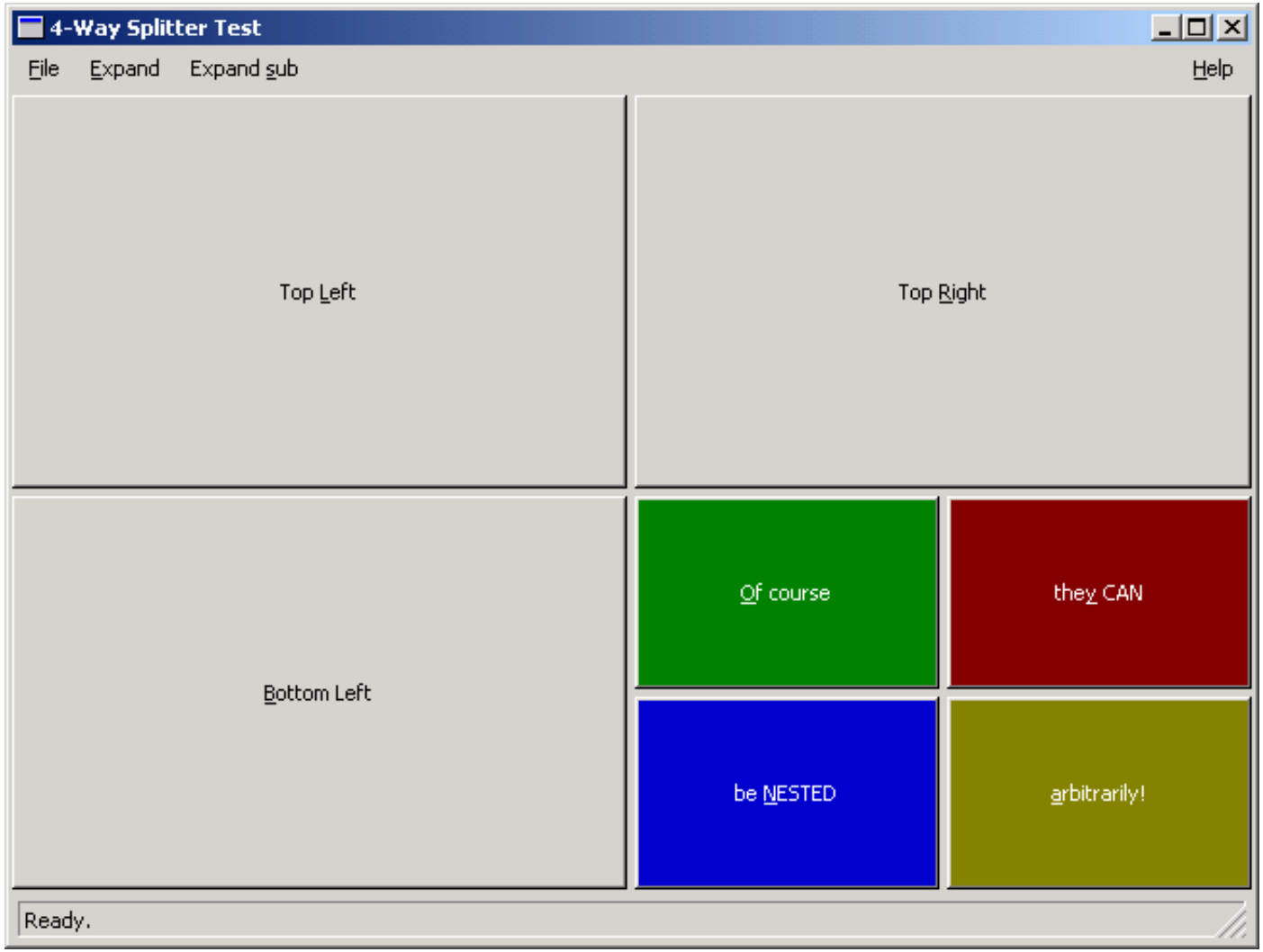
  2x2-Aufteilung, je ein Objekt pro Quadrant

- FXMatrix

  n x n-Aufteilung, Befüllung wahlweise zeilen- oder spaltenweise

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,
    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 ) )
```

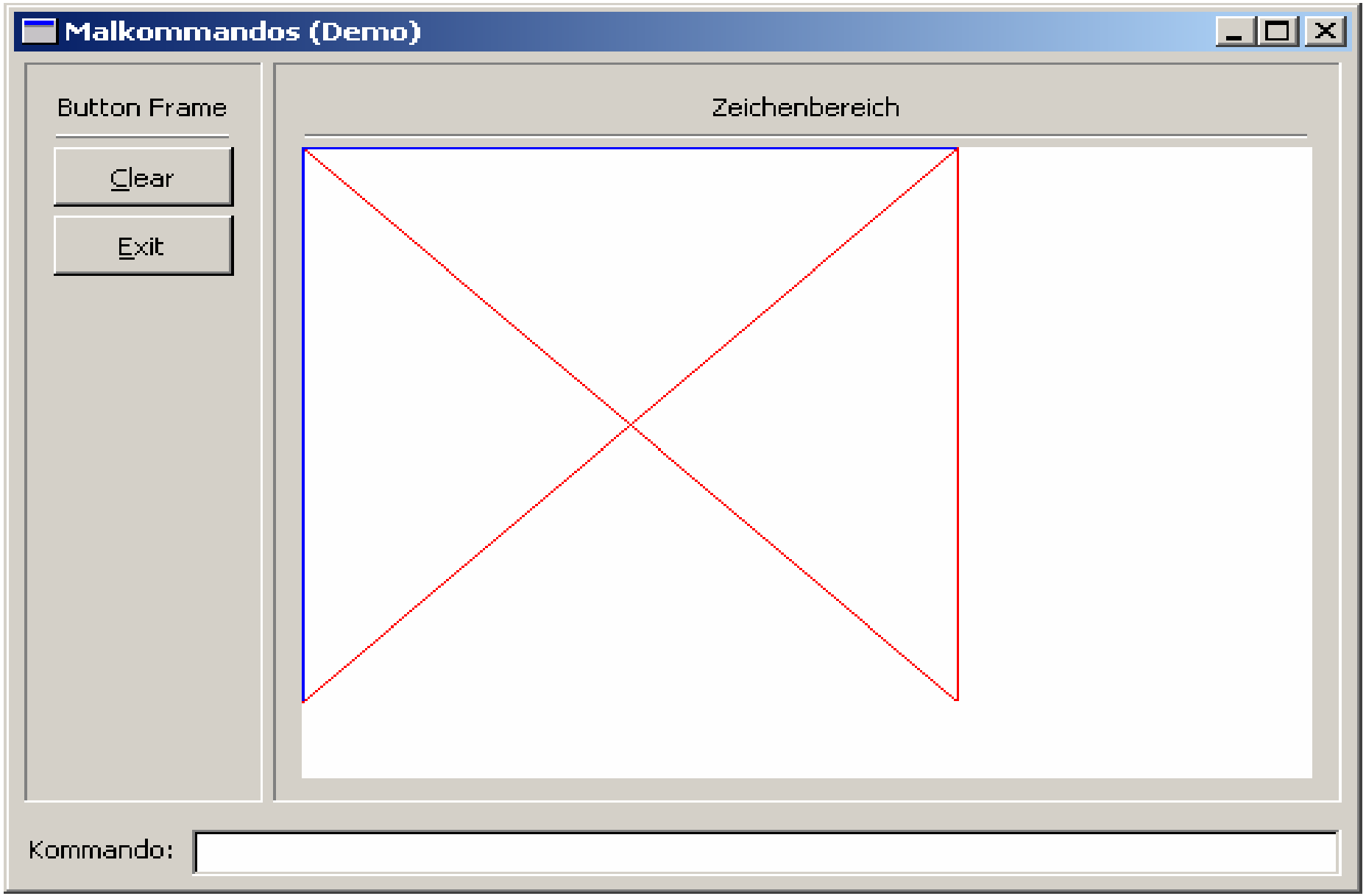
```
# Controller-Methode zur Texteingabe:
def onCmdText( sender, sel, data )
    # Text in data nutzen...
end
```

- 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.