

# WxHaskell und Graphische Benutzerschnittstellen (GUI)

---

- Haskell hat keine festgelegte GUI-Bibliothek
- Es gibt verschiedene Projekte / Implementierungen
- WxHaskell / HtK (keine Wartung mehr) / ...

- gute GUI-Bibliotheken sind komplex  
Viele Elemente / Funktionalitäten
- Pro Plattform gibt es eigene Implementierungen  
der plattformspezifischen Funktionen
- Es gibt bereits portable Implementierungen  
für anderer Programmiersprachen / Skriptsprachen

- Benutzt existierendes wxWindows  
ca. 500 C++ Klassen und ca. 4000 Methoden
- wxWindows hat eigene MSWindows /  
MacOS und Linux Bibliotheken
- mit plattformspezifischem Aussehen und Verhalten
- wxHaskell gibt es im Moment nur für ghc und Version 6.4  
2007: es wird eine neue Version vorbereitet
- Benutzt das foreign function interface (FFI) von Haskell (ghc)

- Haskell-Bibliothek [GRAPHICS.UI.WX](#)
- Diese bildet die WX-Funktionalität in Haskell ab und stellt ein Haskell-Interface bereit
- Muss imperativ programmiert werden mit Zuständen, und richtiger Reihenfolge der Aufrufe
- ⇒ Monadisches wxHaskell-Programmieren
- man benötigt ghc mit `-fglasgow-exts`

```
{- demonstrates the use of a simple menu, statusbar, and dialog -}
```

```
module Main where
```

```
import Graphics.UI.WX
```

```
import EnableGUI -- MAC OS X spezifisch
```

```
main :: IO ()
```

```
main = enableGUI >> -- MAC OS X spezifisch
```

```
start hello
```

```
hello :: IO ()
hello = do
  f <- frame [text := "Hello world!", clientSize := sz 300 200]
    -- create file menu:
  file <- menuPane [text := "&File"]
  quit <- menuQuit file [help := "Quit ...", on command := close f]
    -- create Help menu:
  hlp <- menuHelp []
  about <- menuAbout hlp [help := "About wxHaskell"]
    -- create statusbar field:
  status <- statusField [text := "Welcome to wxHaskell"]
    -- set the statusbar and menubar—
  set f [ statusBar := [status]
        , menuBar := [file,hlp]
        -- as an example, put the menu event handler
        -- for an about box on the frame.
        ,on (menu about) := infoDialog f "About wxHaskell"
                                     "This is a wxHaskell demo"
        ]
```

## Bildschirmaufbau; statisch

- Elemente sind hierarchisch gegliedert.
- `TopElement` ist immer ein `Frame`
- Ein `Frame` kann `Panels` enthalten;
- diese wiederum “`Widgets`” (zB. `Buttons`)

- Fenster-Elemente (widgets, panels, frame,...) haben **Attribute**
- Pro Element eine Liste von “Properties” :  
das sind (verallgemeinerte) (Attribut,Wert)-Paare
- Typ: eines Attributs: **Attr w a**:  
wobei Widget-Typ: w, Attributwert-Typ: a
- Attribute können sein:  
**statische Eigenschaften**: Texte, Farben, Positionen o.ä.  
**dynamische Eigenschaften**: verschiedene Ereignisbehandlungen  
Ausgaben, Änderungen, usw.

data Attr w a      Typ der Attribute

4 Konstruktoren für Properties: (Attribut,Wert)-Paare

```
data Prop w =  
  | forall a . (:=) (Attr w a) a  
  | forall a . (:~) (Attr w a) (a -> a)  
  | forall a . (::=) (Attr w a) (w -> a)  
  | forall a . (::~) (Attr w a) (w -> a -> a)
```

- (:=)      Zuweisung Wert an Attribut
- (:~)      Anwendung einer Funktion auf Attribut mit Wert Attribut
- ::=)     wie := mit widget als extra-Argument
- ::~)     wie :~ mit widget als extra-Argument

**Beachte:** a “fehlt” in Prop w

Zwei Erweiterungen:

- **GADT**: Die data-Syntax wird erweitert
- **Existentielle** Typen

Beispiel

Haskell-Syntax

---

```
data List a =  
  Nil  
  | Cons a (List a)
```

GADT-Syntax

---

```
data List a =  
  Nil :: List a  
  Cons :: a -> (List a) -> List a
```

```
data Prop w where
  (:=)    :: forall w a . (Attr w a) -> a -> Prop w
  (:~)    :: forall w a . (Attr w a) -> (a -> a) -> Prop w
  (::=)    :: forall w a . (Attr w a) -> (w -> a) -> Prop w
  (::~)    :: forall w a . (Attr w a) -> (w -> a -> a) -> Prop w
```

“existenzielle” Typen?: s.u.

Beachte das fehlende a im Ausgabetypp Prop w

```
data Prop w =  
  forall a . (:=) (Attr w a) a
```

**Beobachtung:**  $a$  ist kein Argument von Prop.  
Ist somit “gebunden”

**Bemerkung:** die forall-Notation ist verwirrend;  
wirkt nur als Bindungsoperator

## Beispiel

Heterogene Listen: Paar von Objekt und Methode

```
data InhPaar = forall a. IP a (a -> Bool)
              |   EmptyPair
```

Der Datentyp `InhPaar` hat die Konstruktoren

```
IP           :: forall a. a -> (a -> Bool) -> InhPaar
EmptyPair   :: InhPaar
```

Beispiel Folgender Ausdruck ist wohlgetypt:

```
[IP 3 even, IP 1 (< 2), EmptyPair, IP 'c' isUpper] :: [InhPaar]
```

der interne `a`-Parameter-Typ ist jeweils verschieden.

Welche Programmierung ist erlaubt?

```
f (IP wert fun) = ???
```

Korrekt ist:

```
f :: InhPaar -> Bool  
f (IP wert fun) = fun wert
```

Falsch sind:

```
f1 :: InhPaar -> a -> Bool  
f1 (IP wert fun) = fun
```

```
f2 :: InhPaar -> a  
f2 (IP wert fun) = wert
```

Erkennung durch den Typ-Checker:

Existentielle gebundene Typen dürfen **nicht frei** vorkommen

Beschränkungen der Benutzung:

`Pattern`: Beim `Pattern Match` wird für jede existenziell quantifizierte Variable eine neue Konstante eingeführt, Diese können nicht mit anderen Typen unifiziert werden, Sie dürfen auch Ihren Scopus nicht verlassen

`im let`: kein `Pattern Match` erlaubt für `EX-Konstrukturen`

`newtype` eingeschränkt

`deriving` eingeschränkt

Nochmal:

- `data Attr w a` Typ der Attribute
- Typ: eines Attributs: `Attr w a`:  
wobei Widget-Typ `w`, Attributwert-Typ `a`
- Bindung an Widget mittels Properties:
- wurde erzeugt mit den Konstruktoren (s.o.)  
z.B. `(:=) :: forall a . (Attr w a) -> a-> Prop w`

```
get :: w -> Attr w a -> IO a
--- Get the value of an attribute
  t <- get w text

--      text ist Attributname
--      text:: forall w. Textual w => Attr w String
--          somit a = String
--          und t: String
```

Die Attributnamen wie `text`, `value`, `help` sind **überladen**

z.B. `text`-Attribut

Allgemein: `text:: forall w. Textual w => Attr w String`

in Buttons: `text:: forall a . Attr (Button a) String`

Typklassen dazu:

```
class Textual w where
  text:: Attr w String
instance Textual (Window a) where
  text = .....
```

```
set :: w -> [Prop w] -> IO ()
--- Set a list of properties.
    set w [text := "Hi"]

--      (:=) :: forall a . (Attr w a) -> a -> Prop w
--      text:: forall w. Textual w => Attr w String

swap :: w -> Attr w a -> a -> IO a
-- Set the value of an attribute and return the old value.
    t <- swap w text "neuerEintrag"
```

```
ausrufeZeichendazu :: Button a -> IO a
ausrufeZeichendazu b =
  do s <- get b text
     set b [text := s ++ "!"]
```

$(:\sim)$  :: Attr w a -> (a -> a) -> Prop w

Mit  $:\sim$  kürzer geschrieben:

```
ausrufeZeichendazu b = set b [text :~ (++) "!"]
```

Die Funktion  $(++)$  wird auf den Text-Attribut-Wert des Widgets  $b$  destruktiv angewendet

Aus dem Beispiel timeFlows:

```
set p [ on paint      := onPaint vmouseHistory
      , on idle       := onIdle vmouseHistory p
      , on drag       := onDrag vmouseHistory
      ] ....

-- mouse drag handler
onDrag vmouseHistory mousePos
= do time <- getTime
    -- prepend a new time/position pair
    varUpdate vmouseHistory ((time,mousePos):)
    return ()
```

# Attribute: Beispiele für Typen

```
on drag := onDrag vmouseHistory
```

Vordefiniert: `on`, `drag`, `:=`      benutzerdefiniert: `onDrag`, `vmouseHistory`

Typen:

```
on :: forall a w. Event w a -> Attr w a
drag :: forall w. (Reactive w) => Event w (Point -> IO ())
(:=) :: forall w a. Attr w a -> a -> Prop w
(onDrag) :: forall b. Var [(Time, b)] -> b -> IO ()
(onDrag vmouseHistory) :: (Point -> IO ())
```

Ergibt Typ der Property:

```
Reactive w => Attr w (Point -> IO ())
```

Dynamisches Verhalten:

(`onDrag`) erhält Maus-Pos als zweites Argument

```
data Attr w a = Attr (w -> IO a) (w -> a -> IO a)
```

Genauer für text:

```
class Textual w where
  text :: Attr w String
instance Textual (Window a) where
  text = Attr windowGetLabel windowSetLabel
```

Implementierung von get:

```
get :: w -> Attr w a -> IO a
get w (Attr getter setter) = getter w
```

```
set :: w -> [Prop w] -> IO ()
set w props =
  mapM_ setone props
where
  setone ((Attr getter setter) := x) = setter w x
  setone ((Attr getter setter) :~ f =
    do x <- getter w
       setter w (f x)

-- aus Prelude:
mapM_      :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as = sequence_ (map f as)
sequence_  = foldr (>>) (return ())
```

## Graphics.UI.WX.Controls

### interaktive Steuerelemente:

Button  
Text entry  
CheckBox  
Choice  
ComboBox  
ListBox  
RadioBox  
Spin Control

Slider  
Gauge  
Tree control  
List control  
Static text  
SplitterWindow  
ImageList

teilweise analog zum monadischen Stream-” Calculator”

Zustände und Zustandsübergänge wie im Calculator

Zustand als “globale Variable” in der do-Notation  
Alle Aktionen sind in der IO-Monade

Graphik und Ereignisbehandlung mittels wxHaskell

```
import Char
import Graphics.UI.WX
import EnableGUI -- MAC OS X spezifisch

main :: IO ()
main
  = enableGUI >> -- MAC OS X spezifisch
    start calculator

calculator :: IO ()
calculator
  = do
    f      <- frame      [text := "WXHaskell-Taschenrechner",  clientSize := sz 800 600]
    status <- variable [value := startState]
    zahlfeldwert <- variable [value := " "]
    zahlfeld <- entry f [text := "                "]
    but0 <- button f [text := "0",on command := digitDo status zahlfeld zahlfeldwert 0]
    but1 <- button f [text := "1",on command := digitDo status zahlfeld zahlfeldwert 1]
    but2 <- button f [text := "2",on command := digitDo status zahlfeld zahlfeldwert 2]
    but3 <- button f [text := "3",on command := digitDo status zahlfeld zahlfeldwert 3]
    but4 <- button f [text := "4",on command := digitDo status zahlfeld zahlfeldwert 4]
    but5 <- button f [text := "5",on command := digitDo status zahlfeld zahlfeldwert 5]
    but6 <- button f [text := "6",on command := digitDo status zahlfeld zahlfeldwert 6]
    but7 <- button f [text := "7",on command := digitDo status zahlfeld zahlfeldwert 7]
```

```
but8 <- button f [text := "8",on command := digitDo status zahlfeld zahlfeldwert 8]
but9 <- button f [text := "9",on command := digitDo status zahlfeld zahlfeldwert 9]
butk <- button f [text := ".",on command := kommaDo status zahlfeld zahlfeldwert]
butc <- button f [text := "c", on command := clearDo status zahlfeld zahlfeldwert]
buteq <- button f [text := "=", on command := totalDo status zahlfeld zahlfeldwert]
butpm <- button f [text := "+-", on command := operDo status zahlfeld zahlfeldwert 'n']
butdiv <- button f [text := "/", on command := operDo status zahlfeld zahlfeldwert '/']
butmal <- button f [text := "X", on command := operDo status zahlfeld zahlfeldwert '*']
butplus <- button f [text := "+", on command := operDo status zahlfeld zahlfeldwert '+']
butmin <- button f [text := "-", on command := operDo status zahlfeld zahlfeldwert '-']
set f [layout := column 10 [floatCenter (widget zahlfeld)
    ,floatCenter $
    row 10 [widget butc, widget butpm,widget butdiv, widget butmal]
    ,floatCenter $
    row 10 [widget but7, widget but8, widget but9, widget butmin]
    ,floatCenter $
    row 10 [widget but4, widget but5, widget but6, widget butplus]
    ,floatCenter $
    row 10 [widget but1, widget but2, widget but3, widget buteq]
    ,floatCenter $
    row 10 [widget but0 , widget butk]
]]

-- interner Status: ((letzte Aktion, Anzahl Stellen bei Komma),
--                   gespeicherte Operation,
--                   aktueller Wert)
```

```
startState = ((" ",0),id, 0.0)
statusvalue (_,_,x) = x
statusflag  ((x,_),_,_) = x

compOper x
| x == '+' = oper (+)
| x == '-' = oper (-)
| x == '*' = oper (*)
| x == '/' = oper (/)
| x == 'n' = oper (\x y -> (-x))

digitDo status zahlfeld zahlfeldwert i =
  do stwertalt <- varGet status
     varUpdate status (digit i)
     wert <- varGet zahlfeldwert
     varSet zahlfeldwert
       (if statusflag stwertalt == "t"
          then (show (floor i))
          else (wert++ (show (floor i))))
  wert' <- varGet zahlfeldwert
  set zahlfeld [text := wert' ]
```

```
clearDo status zahlfeld zahlfeldwert =
```

```
do varUpdate status clear
  varSet zahlfeldwert ""
  set zahlfeld [text := ""]
```

```
totalDo status zahlfeld zahlfeldwert =
  do varUpdate status total
    wert <- varGet status
    varSet zahlfeldwert (show (statusvalue wert))
    wert' <- varGet zahlfeldwert
    set zahlfeld [text := wert']
```

```
operDo status zahlfeld zahlfeldwert otext =
  do
    varUpdate status (compOper otext)
    varUpdate zahlfeldwert (\x -> x ++ [otext])
    wert' <- varGet zahlfeldwert
    set zahlfeld [text := wert' ]
```

```
kommaDo status zahlfeld zahlfeldwert =
  do
    varUpdate status komma
    varUpdate zahlfeldwert (\x -> x ++ ['.'])
    wert' <- varGet zahlfeldwert
    set zahlfeld [text := wert' ]
```

```
clear (str,g, 0.0) = (("c",0),id, 0.0)
```

```
clear (str,g, z) = (("c",0), g, 0.0)

total (_,g, z) = (("t",0),id, g z)

digit d ((str,dignum),g, z) =
  if str == "t" then (("d",0),g, d)
  else if str == "."
  then ((".", dignum+1),g, z + (0.1^dignum)*d)
  else (("d",0), g, z*10.0 + d)

komma ((str,_), g, z) =
  if str == "d" || str == "c" || str == " " || str == "o"
  then ((".",1),g, z)
  else if str == "t"
  then ((".",1),g, 0.0)
  else ((".",30),g, z)

oper o (_,g, z) = (("o",0), o (g z), 0.0)
```

Bemerkungen zu Methoden:

- sequentieller Aufbau des GUI:  
Zuerst Frame, danach eingebettete Knöpfe (Buttons)
- Definition von Zustand und globalen Variablen
- Buttons enthalten in der Property-Liste die Reaktionen auf Events
- zustandsbehaftete Programmierung ist notwendig

- Starkes Typsystem verhindert illegale Operationen auf den Widgets
- Speicher-Management ist automatisch
- Null-Pointer-Fehler werden abgefangen

## Prinzipielle Fehlermöglichkeiten zur Laufzeit:

- Vergessen eines Widgets
- Verdopplung eines Widgets
- Verletzung der Hierarchie der Fensterkomponenten

.....

```
f <- frame [text := "Example"]  
tx <- staticText [text := "Hallo WX"]  
ok <- button [text := "OK"]  
can <- frame [text := "Cancel"]
```

Verdopplungsfehler:

```
set f [layout := row 5 [widget ok, widget ok]]
```

Hierarchiefehler:

```
set ok [layout := widget can]
```

Abbildung der Klassenhierarchie von wxWindows:  
um mittels Typcheck die Hierarchiefehler zu eliminieren:

Sogenannte **Phantom-Typen** (ohne Daten)

## Beispiel

```
type Object a = Ptr a
```

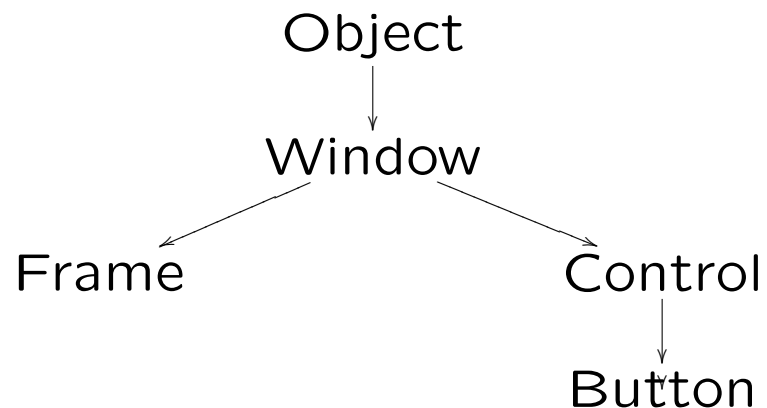
```
data CWindow a
```

```
data CFrame a
```

```
data CControl a
```

```
data CButton a
```

```
type Window a = Object (CWindow a)
type Frame a  = Window (CFrame a)
type Control a = Window (CControl a)
type Button a = Control (CButton a)
```



Typen einiger Widget-Erzeugungsfunktionen:

```
frame      :: [Prop (Frame ())] -> IO (Frame ())
button     :: Window a -> [Prop (Button ())] -> IO (Button ())
staticText :: Window a -> [Prop (Label ())] -> IO (Label ())
```

## Beispiel zur Wirkung

```
do f <- frame []
    b <- button f []
```

## Typen im do:

`f :: Frame ()`      Es gilt      `Frame () = Window (CFrame ())`

Als erstes Argument von `button`:

`Window a` mit `a = (CFrame ())`

## WX-GUI in Haskell: Vorteile und Nachteile

- ++ Typsicherheit
- + eingebettete pure funktionale Berechnungen sind möglich
- + abstrakte GUI-Programmierung
  
- Berechnung ist doch imperativ und nutzt Seiteneffekte
- Dokumentation ist wenig hilfreich
- Mischung von 2 Programmierstilen
- Zukunft: deklarativere GUI-Programmierung