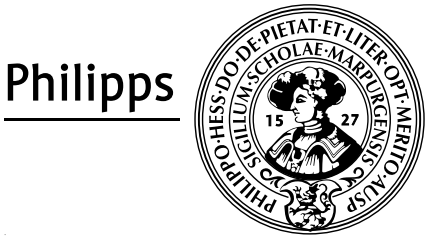


Fachbereich Mathematik & Informatik
AG Programmiersprachen und Parallelität
Prof. Dr. R. Loogen

Seminar "Fun of Haskell Programming"

Ausarbeitung zum Thema

GUI-Programmierung

von

Uwe Schäfer

Matr.-Nr. 2130998

Betreuerin: Prof. Dr. R. Loogen

Basierend auf:

- Gtk2Hs Team: *Gtk2Hs Documentation*, *http://www.haskell.org/gtk2hs/*, Januar 2010.
- Hans van Thiel, Alex Tarkovsky, Tony Gale, Ian Main, GTK+ Team: *Gtk2Hs Tutorial*, *http://www.muitovar.com/gtk2hs/*, Januar 2010.

1 Einleitung

Grafische Benutzeroberflächen Die Entwicklung benutzerfreundlicher Computersysteme, an die man intuitiv ohne umfangreiches Vorwissen über die Bedienung einer Konsole herangehen kann, ist seit jeher ein wichtiger Zweig der Softwareentwicklung. Bereits in den frühen 1970er Jahren gelang es Xerox einen Rechner mit grafischer Benutzeroberfläche (Graphical User Interface, GUI) auf den Markt zu bringen.

Die 70er Jahre sind längst vorbei und grafische Benutzeroberflächen sind bei den hauptsächlich genutzten Betriebssystemen Linux, Mac OS und Windows zu einer Selbstverständlichkeit geworden. Wer heute eine Software einer breiten Masse zugänglich machen möchte wird, genau wie in den Anfängen der grafischen Oberflächen, Wert darauf legen, dass sein Programm intuitiv bedienbar ist.

Es gibt inzwischen die verschiedensten Ansätze und Toolkits, um GUIs zu gestalten. Viele davon sind an bestimmte Betriebssysteme gebunden, aber aus dem Open-Source-Bereich heraus haben sich zudem auch Toolkits gebildet, die auf die verschiedensten Systeme portiert wurden. Die beliebtesten plattformübergreifenden Bibliotheken stellen Qt und das GIMP-Toolkit (GTK+) dar. Letzteres werden wir im weiteren Verlauf dieser Arbeit genauer kennenlernen.

Ursprünglich als Oberfläche für das GNU Image Manipulation Program entwickelt, ist das GTK heute in vielen Open-Source-Anwendungen zu finden. Als bekannteste systemübergreifende Beispiele seien hier sein Ursprung, GIMP, und der Instant Messenger Pidgin genannt. Linux-Nutzern wird sicher auch das GNOME Project etwas sagen.

Gtk2Hs Abgesehen von der Beliebtheit und Portabilität interessiert uns in dieser Arbeit aber vorrangig die Möglichkeit, ein GUI-Toolkit mit Haskell zu verwenden. Für die funktionale Sprache existieren einige Bibliotheken, die genau dies ermöglichen. Eine besonders weit entwickelte darunter ist Gtk2Hs. Sie unterstützt laut Projektseite die GTK+ API fast vollständig, bietet Unicode-Support für fremde Zeichensätze und ist ausführlich dokumentiert. Zudem erhalten wir die Möglichkeit, durch Support für alle gängigen Betriebssysteme plattformübergreifend zu programmieren.

Als wesentlicher Vorteil von Gtk2Hs – insbesondere gegenüber dem „Konkurrenten“ wxWidgets – erscheint zudem das automatische Speichermanagement. Gtk2Hs arbeitet mit Referenzzählung und kann daher entsprechend reagieren, wenn der Benutzer ein Fenster schließt. In anderen Bibliotheken, die nicht nach dieser Methode arbeiten, kann es passieren dass ein geschlossenes Fenster aus dem Speicher gelöscht wird, obwohl es noch vom Programm referenziert wird. Damit kann es in einem solchen Fall zu Abstürzen kommen, weil Pointer sich im Nirvana verlaufen.

Dass man mit Gtk2Hs auch wirklich alltagstaugliche Programme schreiben kann, ist deutlich am Beispiel des Haskell-IDEs Leksah zu erkennen. Leksah wurde in Haskell geschrieben und besitzt ein GUI, welches unterstützt durch GTK+ und Gtk2Hs entwickelt wurde.

Ziel Diese Arbeit soll einen Einblick in die GUI-Programmierung mit Haskell geben. Zum einen wird das Programmieren eines GUIs von Hand Thema sein, wobei zugleich die Struktur im Aufbau eines GUIs erkennbar werden soll. Zum anderen ist aber auch der Umgang mit der Software Glade, die XML-Beschreibungen für GTK-GUIs erstellt, Bestandteil dieser Arbeit. Des Weiteren steht natürlich auch die Verarbeitung von Nutzereingaben auf einem GUI im Mittelpunkt – wie reagiert die Oberfläche auf Benutzereingaben und wie kann sie sich ihnen anpassen?

2 Hallo Welt!

```
import Graphics.UI.Gtk
main :: IO ()
main = do
  initGUI
  window ← windowNew
  button ← buttonNew
  set window [ containerChild := button ]
  set button [ buttonLabel := "Hello␣world!" ]
  onClicked button (putStrLn "Hello␣world!")
  onDestroy window mainQuit
  widgetShowAll window
  mainGUI
```

Abbildung 1: Ein erstes Programm mit Gtk2Hs

Was wäre die Einführung neuer Möglichkeiten in der Programmierung ohne das obligatorische „Hello world“-Beispielprogramm? Um den grundlegenden Aufbau eines GUIs mit Gtk2Hs zu verdeutlichen, wählen wir das einfachste mögliche GUI für unser erstes Programm: Ein Fenster mit Button. Ein Klick auf den Button soll „Hello world!“ auf der Konsole ausgeben.

Da wir es bei einem GUI in der Regel mit Eingaben von Benutzern zu tun haben, befinden wir uns in der `IO()`-Monade. Betrachten wir nun das Programm zeilenweise, um den grundlegenden Aufbau eines Programms mit Gtk2Hs-Unterstützung zu verdeutlichen. Der Aufbau des GUIs erfolgt Schritt für Schritt – imperativ – wir bemötigen also einen `do`-Block.

Am Anfang eines jeden Programms mit GUI steht der Befehl `initGUI`. Dieser sorgt dafür, dass das GUI-Toolkit initialisiert wird. Erst danach stehen sämtliche GUI-Funktionen zur Verfügung.

```
initGUI
```

Die Funktionen `windowNew` und `buttonNew` erzeugen ein neues Fenster und einen neuen Button. Die Verweise auf die neuen Objekte binden wir an aussagekräftige Namen.

```
window ← windowNew
button ← buttonNew
```

Anschließend geben wir den Objekten `window` und `button` ein paar Eigenschaften mit. Hierfür benutzen wir die allgemeine `set`-Funktion, welche einen Verweis auf das betroffene Objekt und eine Liste von Wertzuweisungen an Attribute erhält. An dieser Stelle wird der Button dem Fenster als Inhalt zugewiesen und der Button bekommt eine Beschriftung.

```
set window [ containerChild := button ]
set button [ buttonLabel := "Hello␣world!" ]
```

Das für den Benutzer sichtbare GUI ist damit schon fertig konstruiert. Damit auch etwas passiert, wenn auf den Button geklickt oder das Programm komplett beendet wird (indem der Benutzer das Fenster schließt), müssen wir im Folgenden noch die `Signals`, die von den GUI-Elementen `button` und `window` bei Benutzereingaben ausgesendet werden, an Funktionen binden.

```
onClicked button (putStrLn "Hello␣world!")
onDestroy window mainQuit
```

Das GUI ist nun vollständig konstruiert und muss nur noch angezeigt werden. Dies übernimmt die Funktion `widgetShowAll`, die den obersten Container übergeben bekommt (in diesem Fall das Fenster `window`).

```
widgetShowAll window
```

Damit das Programm Benutzereingaben abwartet und nicht gleich wieder beendet wird, starten wir in der letzten Zeile des Programms die Eingabeschleife, in der das Programm auf eintretende Ereignisse reagiert. Dieser Befehl sollte genau wie `initGUI` in jedem Gtk2Hs-Programm zu finden sein.

```
mainGUI
```

2.1 Wesentliche Funktionen im Detail

Im Code des Hallo-Welt-Programms haben wir bereits einige Funktionen aus dem Gtk2Hs-Paket verwendet. Nun betrachten wir noch einmal die wesentlichen Funktionen eines Programms mit GUI.

```
initGUI :: IO [String]
```

Wie bereits eingangs erwähnt, muss diese Funktion vor allen anderen Funktionen aus der Gtk2Hs-Bibliothek aufgerufen werden. Sollte es Probleme beim Initialisieren des GUI-Toolkits geben, wird eine Exception geworfen.

```
mainGUI  :: IO ()
mainQuit :: IO ()
```

Diese beiden Funktionen rahmen die Eingabeschleife des Programms ein. Ist das GUI aufgebaut, sind alle gewünschten `Signals` gebunden und ist das GUI angezeigt, wird die Eingabeschleife gestartet, in der das Programm auf Eingaben reagiert. Der Befehl `mainQuit` beendet diese Eingabeschleife und somit im Normalfall auch das Programm.

```
get :: o → ReadWriteAttr o a b → IO a
set :: o → [AttrOp o] → IO ()
```

```
value ← get widget attribute
set widget [ attribute := value ]
```

Um die Eigenschaften eines Widgets zu modifizieren oder Eigenschaften auszulesen (der Inhalt eines Textfeldes ist z. B. auch eine solche Eigenschaft), gibt es eine Getter- und eine Setter-Funktion.

`get` bekommt als Parameter das zu betrachtende Objekt und den Namen des gesuchten Attributs und liefert den zugehörigen Wert, der in der Monade gebunden werden kann. `set` bekommt ebenfalls das zu modifizierende Objekt und dazu eine Liste von Operationen. Im Falle einer einfachen Zuweisung sieht eine Operation wie im Beispielaufruf oben aus. Es gibt jedoch noch eine Vielzahl weiterer Operatoren, die hier verwendet werden können. Ein nützlicher darunter ist `:~`, welcher eine Update-Funktion auf einen Attributwert anwendet:

```
set spinButton [ spinButtonValue :~ (+1) ]
```

Dieser Aufruf inkrementiert den Wert eines Spinbuttons (Eingabefeld für numerische Werte aus einem festgelegten Bereich mit Buttons zum De- und Inkrementieren).

3 Umfangreichere GUIs

3.1 Container und Packing

Container In unserem ersten Beispielprogramm bestand das GUI nur aus einem Fenster mit einem einzigen Button. Wir konnten den Button direkt als „Kind“ des Fensters einbinden. Möchten wir aber alltagstauglich arbeiten und mehr als ein einziges GUI-Element (Widget) in unserer Oberfläche unterbringen, so benötigen wir dafür unsichtbare Behälter (Container). An dieser Stelle seien zunächst die beiden einfachsten Container kurz vorgestellt: Eine `HBox` ordnet Elemente horizontal nebeneinander an und eine `VBox` tut dies in vertikaler Richtung.

Packing Das Packing bezeichnet das eigentliche Hinzufügen von Elementen in einen Container. Bei `HBox` und `VBox` verwenden wir dafür die Funktionen `boxPackStart` und `boxPackEnd`. `boxPackStart` füllt den Container von vorne nach hinten mit Widgets, während `boxPackEnd` ihn von hinten nach vorne füllt.

```
boxPackStart, boxPackEnd :: (BoxClass self, WidgetClass child)
  ⇒ self     -- Container, in den gepackt werden soll
  → child    -- Widget, das hineingepackt werden soll
  → Packing  -- Art, auf die es hineingepackt werden soll
  → Int      -- Abstand zu den Nachbarn (in Pixeln)
  → IO ()
```

Abbildung 2: Typ von `boxPackStart` und `boxPackEnd`

Beim Hinzufügen eines Elements wird durch einen Wert vom Typ `Packing` festgelegt, wie es gepackt werden soll. Erlaubte Werte sind hier:

`PackNatural`	das Element belegt soviel Platz, wie es benötigt, der Container füllt sich von der jeweiligen Seite
`PackRepel`	der Containerinhalt wird über den vollen zur Verfügung stehenden Platz verteilt, das Element selbst bleibt so groß wie es sein will, der Freiraum wird gleichmäßig um das Element aufgeteilt
`PackGrow`	der Containerinhalt wird über den vollen zur Verfügung stehenden Platz verteilt, das Element wird vergrößert, sodass es den kompletten zur Verfügung stehenden Platz belegt

Tabelle 1: Packing-Methoden für `BoxClass`-Container

Neben der Art des Packens sind für das Aussehen von `HBox` und `VBox` auch Parameter bei deren Erstellung von Bedeutung. Dazu werfen wir in Abbildung 3 einen Blick auf die Konstruktoren. Diese bekommen einen `Bool`-Wert übergeben,

```
hBoxNew :: Bool → Int → IO HBox
vBoxNew :: Bool → Int → IO VBox
```

Abbildung 3: Typen der Konstruktoren von `HBox` und `VBox`

der festlegt, ob der dem Container zur Verfügung stehende Platz gleichmäßig auf die enthaltenen Objekte aufgeteilt werden soll oder nicht (dann bekommt jedes Objekt Platz entsprechend seiner Größe). Der `Int`-Wert beschreibt den Abstand zwischen zwei Elementen im Container (dieser Abstand wird immer gegeben – beim Packing angegebener Abstand kommt noch hinzu).

Abschließend sei in Abbildung 4 noch einmal im Bild festgehalten, wie sich die verschiedenen Parameter und Packingfunktionen tatsächlich auswirken.

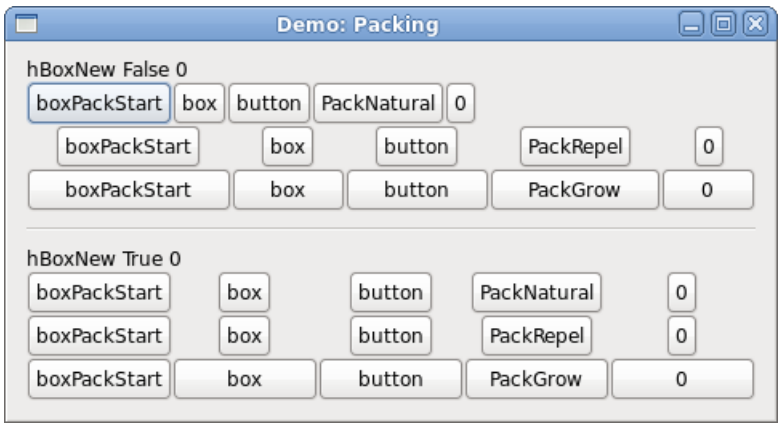


Abbildung 4: Verschiedene Packing-Methoden einer `HBox`

```
box     ← hBoxNew False 0
button1 ← buttonNewWithLabel "boxPackStart"
boxPackStart box button1 PackNatural 0
button2 ← buttonNewWithLabel "box"
boxPackStart box button2 PackNatural 0
button3 ← buttonNewWithLabel "button"
boxPackStart box button3 PackNatural 0
button4 ← buttonNewWithLabel "PackNatural"
boxPackStart box button4 PackNatural 0
button5 ← buttonNewWithLabel "0"
boxPackStart box button5 PackNatural 0
```

Abbildung 5: Code für die erste Buttonreihe im Fenster aus Abbildung 4

Während sich `boxPackStart` und `boxPackEnd` in allen anderen Modi nur auf die Reihenfolge der Buttons auswirken, führt die gemischte Anwendung ohne gleichmäßige Platzverteilung und Auffüllen zu dem in Abbildung 6 dargestellten Ergebnis.


```
box     ← hBoxNew False 0
button1 ← buttonNewWithLabel "Dies (1)"
boxPackStart box button1 PackNatural 0
button2 ← buttonNewWithLabel "ist (2)"
boxPackStart box button2 PackNatural 0
button3 ← buttonNewWithLabel "eine (3)"
boxPackEnd box button3 PackNatural 0
button4 ← buttonNewWithLabel "Packingdemo (4)"
boxPackEnd box button4 PackNatural 0
```

Abbildung 6: Veranschaulichung von `boxPackStart` und `boxPackEnd`

3.1.1 Tabellen

Eine Alternative zu `HBox` und `VBox` stellt der äußerst vielseitige Container `Table` dar, der an dieser Stelle jedoch nur kurz beschrieben werden soll. Dem Konstruktor werden Zeilen- und Spaltenzahl übergeben, gefolgt von einem `Bool`-Wert, der festlegt, ob sich alle Zellen in ihrer Größe dem größten eingefügten Element anpassen sollen.

```
tableNew :: Int → Int → Bool → IO Table
```

Abbildung 7: Typ des Konstruktors `tableNew`

Zeilen- und Spaltenzahl werden zwar zu Beginn festgelegt, können aber nachträglich mit der Funktion `tableResize` noch angepasst werden. Sie dürfen den Bereich [0 .. 65535] jedoch nicht verlassen.

Elemente werden in einer `Table` mit der Funktion `tableAttach` (erlaubt viele zusätzliche Einstellungen) oder `tableAttachDefaults` (verlässt sich auf die für die Tabelle geltenden Standardeinstellungen) platziert.

```
tableAttachDefaults :: (TableClass self, WidgetClass widget)
  ⇒ self     -- Tabelle
  → widget   -- das einzufuegende Widget
  → Int      -- Start-Spalte  (leftAttach)
  → Int      -- End-Spalte    (rightAttach)
  → Int      -- Start-Zeile   (topAttach)
  → Int      -- End-Zeile     (bottomAttach)
  → IO ()
```

Abbildung 8: Typ der Funktion `tableAttachDefaults`

Für das Platzieren in der Tabelle wird ein Zellenbereich angegeben, den das einzufügende Element belegen soll.

Weitere Container Sollte einmal mehr gefragt sein, als man mit Tabellen, Zeilen und Spalten ausdrücken kann, so stehen noch einige andere Container zur Auswahl bereit. Das `Notebook` bietet die Möglichkeit, GUI-Elemente auf Tabs zu sammeln, mit einem `Layout` erhält man eine scrollbare Ebene und ein `Expander` ist in der Lage, seinen Inhalt auf Klick zu zeigen und wieder zu verstecken.

Für Ansammlungen von Buttons stehen die Container `HButtonBox` und `VButtonBox` zur Verfügung, die in ihrer Handhabung den bereits ausführlich beschriebenen Boxen ähneln. Eine `ButtonBox` sorgt automatisch dafür, dass alle enthaltenen Buttons dieselbe Größe aufweisen. Ihren Konstruktoren `hButtonBoxNew` und `vButtonBoxNew` müssen keine Parameter übergeben werden.

Eine variable Teilung eines Containers erhält man mit `HPaned` und `VPaned`, Pixelzähler werden `Fixed` lieben (erlaubt das pixelgenaue Positionieren von Inhalten), `AspectFrame` kann seinen Inhalt skalieren (nützlich für Grafikanwendungen) und innerhalb eines `Alignment`s kann man ein Objekt ausgerichtet positionieren.

3.2 Widgets

Was nützt uns die ganze Theorie über das Packing, wenn wir außer einfachen Buttons nichts haben, was zu packen wäre? Daher folgt nun ein Überblick über die wichtigsten Widgets wie Schaltflächen, Textfelder und Anzeigefelder und die damit verbundenen Events.

3.2.1 Schaltflächen

In unseren bisherigen Beispielen hatten wir schon mehrfach mit Schaltflächen (Buttons) zu tun. In Abbildung 9 seien noch einmal die zugehörigen Konstruktoren im Überblick gezeigt.

```
buttonNew              :: IO Button
buttonNewWithLabel     :: String → IO Button
buttonNewWithMnemonic  :: String → IO Button
buttonNewFromStock     :: StockId → IO Button
```

Abbildung 9: Konstruktoren für `Button`

Während `buttonNew` einen einfachen Button ohne alles erstellt, dem man dann über einen Setter Eigenschaften wie eine Beschriftung mitgeben kann, gibt es mit `buttonNewWithLabel` auch die Möglichkeit, direkt dem Konstruktor eine Beschriftung zu übergeben. `buttonNewWithMnemonic` funktioniert wie `buttonNewWithLabel` – mit dem Unterschied, dass Tastaturkürzel festgelegt werden können. Dies geschieht durch einen Unterstrich vor dem abkürzenden Buchstaben – z. B. `_Ja`. *Mehr zu Tastaturkürzeln im Abschnitt 3.3.*

Interessant, um Dialoge einheitlich zu gestalten, sind fertige Buttons aus dem Lager (Stock). Hierbei handelt es sich um Buttons wie „Apply“ oder „Cancel“ mit Beschriftung und Symbol. Diese werden anhand einer `StockId` identifiziert.

Die im Stock gelagerten Bilder können auch nebst einer eigenen Button-Beschriftung genutzt werden. Dazu genügt es allerdings nicht, einen Button aus dem Lager zu nehmen und sein Label neu zu setzen (dabei verschwindet das Symbol wieder), sondern man muss Beschriftung und Bild einzeln setzen (siehe Abbildung 10).

```
button     ← buttonNew
applyImage ← imageNewFromStock stockApply IconSizeButton
set button [buttonImage := applyImage, buttonLabel := "Nehmen␣wir!"]
```

```
button     ← buttonNewWithLabel "Nehmen␣wir!"
applyImage ← imageNewFromStock stockApply IconSizeButton
buttonSetImage button applyImage
```

Abbildung 10: Möglichkeiten, einen Button mit angepasster Beschriftung mit dem Stock-Bild `stockApply` zu versehen

Neu ist an dieser Stelle noch der Befehl `imageNewFromStock`, der die `StockId` des gewünschten Symbols und die gewünschte `IconSize`, in der wir das Bild haben möchten, übergeben bekommt. Die `StockId` kann einer langen Liste in der Dokumentation entnommen werden. Auch für die `IconSize` gibt es eine Liste, die Größen für jede denkbare Anwendung innerhalb von GUI-Elementen vorsieht, aber auch eine eigene Angabe über `IconSizeUser Int` ermöglicht.

Ereignisse Widgets reagieren auf Benutzereingaben, indem sie Signals senden. Möchte man, dass das Programm auf bestimmte Eingaben reagiert, so sollte man den Signals Aktionen zuordnen. Dies geschieht beim Button über folgende Funktionen:

`onClicked`	Button wird betätigt und Mauscursor befindet sich auf dem Button, wenn er wieder losgelassen wird. Alternativ: Button wird durch Tastatur betätigt (über Mnemonic oder Auswahl und Enter)
`onPressed`	Button wird gedrückt (das Loslassen spielt hier keine Rolle)
`onEnter`	Mauscursor betritt den Button
`onLeave`	Mauscursor verlässt den Button
`onButtonActivate`	Button wird via Tastatur betätigt (in der Regel wird direkt auf dieses Signal folgend `onClicked` abgearbeitet)

Tabelle 2: `Signal`-Handler für `Button`

Alle zum `Button` gehörenden Signal-Funktionen haben den Typen

```
:: ButtonClass b ⇒ b → IO () → IO (ConnectId b)
```

und werden beispielsweise wie folgt verwendet:

```
onClicked button (putStrLn "Hello␣world!")
```

```
onClicked button $ do
  buttonSetLabel button "Klick!"
  putStr "Klick␣erfolgt."
```

Zu den oben genannten `on`-Funktionen gibt es jeweils auch eine `after`-Funktion. Bindet man an beide eine Aktion, so werden sie direkt nacheinander ausgeführt.

Wie der Typ der Funktionen schon vermuten lässt, liefern sie einen Wert zurück, der zur späteren Verwendung gebunden werden kann. Die `ConnectId` dient dazu, die Signalbindung zur späteren Modifikation im Programm zu identifizieren. Mit der Funktion `signalDisconnect` kann sie beispielsweise wieder gelöst werden (siehe Abbildung 11).

```
signalDisconnect :: GObjectClass obj ⇒ ConnectId obj → IO ()
```

```
aktion ← onClicked button (putStr "Klick␣erfolgt.")
afterClicked button (signalDisconnect aktion)
```

Abbildung 11: Beispiel zu `signalDisconnect`: Beim ersten Klick auf den Button wird eine Ausgabe erzeugt. Anschließend verliert der Button seine Wirkung, indem die Signalbindung gelöst wird.

3.2.2 Eingabefelder

Einzeilige Textfelder werden in Gtk2Hs als `Entry` bezeichnet. So einfach wie die Funktionsbeschreibung fällt auch die Übersicht über die zugehörigen Konstruktoren aus – es gibt nur einen:

```
entryNew :: IO Entry
```

Abbildung 12: Konstruktor für `Entry`

Wichtig im Umgang mit `Entry`s ist natürlich das Lesen und Schreiben des enthaltenen Textes. Dies wird mit den quasi selbsterklärenden Funktionen `entrySetText` und `entryGetText` realisiert.

```
entrySetText :: EntryClass self ⇒ self → String → IO ()
entryGetText :: EntryClass self ⇒ self → IO String
```

Ein `Entry` hat neben seinem Inhalt aber noch andere nützliche Eigenschaften:

`entryEditable`	Der Inhalt des `Entry`s kann verändert werden *Bool, True*
`entryVisibility`	Die eingegebenen Zeichen sind sichtbar (wenn nicht: Passwort-Modus) *Bool, True*
`entryInvisibleChar`	Das Zeichen, was bei unsichtbarem Text als Platzhalter dient *Char, System-Standard*
`entryMaxLength`	Maximale Eingabelänge (max 66535) *Int, 0 (unbegrenzt)*
`entryWidthChars`	Breite des Textfelds auf eine ungefähre Zeichenzahl festlegen *Int, -1 (keine Festlegung)*

Tabelle 3: Auswahl von Attributen eines `Entry`s (Typ und Standardwert kursiv hinter der Beschreibung)

Ereignisse Bestätigt der Benutzer seine Eingabe durch die Eingabetaste im Textfeld, so sendet dieses genau wie eine Schaltfläche ein Signal aus. Gebunden wird es analog zu den Signals eines Buttons über die Funktionen `onEntryActivate` bzw. `afterEntryActivate`. Außerdem gibt es noch Signals für das Kopieren, Ausschneiden und Einfügen von Text und den Wechsel zwischen Überschreiben und Einfügen während der Eingabe (durch Drücken der Einfügen-Taste).

3.2.3 Anzeigefelder

Sogenannte `Label`s dienen dazu, Text innerhalb eines GUIs anzuzeigen. Der Konstruktor heißt, wer hätte es gedacht, `labelNew`. Ihm kann direkt eine erste Textzeile übergeben werden.

```
labelNew             :: Maybe String → IO Label
labelNewWithMnemonic :: String → IO Label
```

Abbildung 13: Konstruktoren für `Label` mit und ohne Mnemonic

Ein `Label` kann, genau wie Buttons, mit einem Tastaturkürzel versehen werden, das z. B. auf ein benachbartes Eingabefeld verweist. *Mehr zu diesem Thema im Abschnitt 3.3.*

Um den Text eines `Label`s zu verändern, gibt es unter anderem die üblichen Funktionen `labelSetText` und `labelGetText`. Da Labels die Verwendung von Pango (formatierter Text mit Unterstützung anderer Zeichensätze für Fremdsprachen) und Tastaturkürzel erlauben, gibt es aber zusätzlich noch die Funktionen `labelSetLabel` und `labelGetLabel`, welche die Eigenschaften des Labels bzgl. der Interpretation von Pango-Markup und Unterstreichungen erhalten.

```
labelGetText  :: LabelClass self ⇒ self → IO String
labelGetLabel :: LabelClass self ⇒ self → IO String
labelSetText  :: LabelClass self ⇒ self → String → IO ()
labelSetLabel :: LabelClass self ⇒ self → String → IO ()
```

Zunächst unterscheiden sich die Ergebnisse dieser beiden Blöcke nicht:

```
label ← labelNew Nothing
labelSetText label "_Pango␣ist␣<b>toll</b>"
```

```
label ← labelNew Nothing
labelSetLabel label "_Pango␣ist␣<b>toll</b>"
```

_Pango ist toll Beide Labels enthalten den uninterpretierten Markup-Code. Auch die Ergebnisse dieser Aufrufe unterscheiden sich nicht voneinander:

```
label ← labelNew Nothing
labelSetUseMarkup label True
labelSetUseUnderline label True
labelSetLabel label "_Pango␣ist␣<b>toll</b>"
```

```
label ← labelNew Nothing
labelSetMarkupWithMnemonic label "_Pango␣ist␣<b>toll</b>"
```

Pango ist **toll** Beide Labels enthalten eine Beschriftung mit interpretiertem Markup. Im zweiten Block übernimmt der Befehl `labelSetMarkupWithMnemonic` die beiden Settings, die im oberen Block von Hand gemacht wurden. Nutzt man jedoch nach dem Setzen eines Textes mit Markup wieder die Funktion `labelSetText`, so werden sämtliche Einstellungen ignoriert und der String wird so angezeigt, wie er eingegeben wurde.

```
label ← labelNew Nothing
labelSetUseMarkup label True
labelSetUseUnderline label True
labelSetText label "_Pango␣ist␣<b>toll</b>"
```

Interessanterweise behält das `Label` aber dennoch seine allgemeinen Anzeigeeigenschaften. Eine erneute Verwendung von `labelSetLabel` würde wieder die Verwendung von Markup erlauben.

3.2.4 Schaltflächen mit Statuswechsel

Zu den sog. ToggleButtons gehören alle Schaltflächen, die bei einem Klick ihren Status wechseln:

- `ToggleButton`: eine Schaltfläche, die durch einen Klick gedrückt wird und mit einem weiteren Klick gelöst werden kann

- `CheckButton`: ein Label mit einem benachbarten Kästchen, in das ein Häkchen gesetzt werden kann
- `RadioButton`: ähnlich wie der `CheckButton`, allerdings rund und in Gruppen zusammenschließbar, in denen üblicherweise einer immer aktiv ist (sinnvoll für Entscheidungsfragen)

Wesentliche Signalbindungen für diese Gruppe von Widgets sind `on/afterToggled`. Entscheidendes Attribut ist `toggleButtonActive`, abzufragen über den Standardgetter oder `toggleButtonGetActive`. Die Konstruktoren `toggleButtonNewWithMnemonic`, `checkButtonNewWithMnemonic` und `radioButtonNewWithMnemonic` bekommen eine Beschriftung als String übergeben, wie wir es bereits von den gewöhnlichen Schaltflächen kennen. Hier gibt es natürlich auch dieselben Konstruktorvarianten (mit oder ohne Label/Mnemonic) wie beim Button. Wichtig für `RadioButton`s ist zudem der Konstruktor `radioButtonNewWithMnemonicFromWidget`, der vor der Beschriftung noch den Namen eines weiteren RadioButtons übergeben bekommt. Mit diesem bildet der neue Button eine Gruppe.

3.2.5 Wertebereiche mit SpinButtons, Scrollbars und Scales verwenden

Alle im folgenden erwähnten Widgets können auf Basis eines `Adjustment`s konstruiert werden. Ein solches Adjustment besteht aus einem Startwert, einer unteren und einer oberen Intervallgrenze, einer Schritt- und einer Seitenweite und einer Seitengröße. Letztere gibt bei einer Scrollbar z. B. die Breite des „Griffs“ an – Gtk2Hs empfiehlt, diese auf 0 zu setzen, was auch sinnvoll ist, da andernfalls nicht jeder Bereich im Intervall erreicht werden kann: Bei einer Scrollbar liegt bei einer Seitengröße > 0 der Anfang des Griffs auf dem Startwert – dieser kann folglich auch nur bis zur oberen Intervallgrenze abzüglich der Seitengröße geschoben werden.

Beginnen wir aber zunächst mit dem `SpinButton`, welcher bereits am Anfang kurz erwähnt wurde: ein Eingabefeld für Zahlenwerte mit Buttons zum Inkrementieren und Dekrementieren. Der zugehörige Konstruktor `spinButtonNewWithRange` bekommt Minimum und Maximum des abzudeckenden Bereichs und eine Schrittweite übergeben. Die Genauigkeit des SpinButtons geht bis zu 20 Nachkommastellen – einstellbar durch `spinButtonSetDigits`. Alternativ kann er auch an ein Adjustment gebunden werden – mit `spinButtonNew`. Parameter: Adjustment, Schrittweite, Nachkommastellen. Die Schrittweite wird hier allerdings zugunsten der im Adjustment festgelegten Schrittweite vernachlässigt. Wesentliche Signalbindungen des SpinButtons sind `on/afterValueSpinned`, die jeweils abgearbeitet werden, wenn sich der Wert des SpinButtons ändert.

Benutzt man ein Widget, das an ein Adjustment gebunden ist, so kann ebenfalls – und in manchen Fällen (Scrollbar) auch nur – der aktuell eingestellte Wert direkt vom Adjustment bezogen werden. Auch sendet das Adjustment ein eigenes Signal bei einer Änderung des eingestellten Wertes:

```
onValueChanged adj $ do
  val ← adjustmentGetValue adj
```

Bekannter als der SpinButton sind vermutlich die eben schon genannten Scrollbars. Diese gibt es in Gtk2Hs nur auf Basis eines Adjustments. Entsprechend einfach

ist auch die Verwendung der Konstruktoren `hScrollbarNew` und `vScrollbarNew`: Einziger Parameter ist das Adjustment. Scrollbars verfügen nicht über eigene Attribute und senden auch keine Signale.

Eine weitere Möglichkeit, einen Wert aus einem Intervall zu wählen, ist ein Objekt namens Scale, bestehend aus einem verschiebbaren Regler. Der Wert kann neben dem „Griff“ angezeigt werden. Die Genauigkeit einer Scale geht bis zu 64 Dezimalstellen und kann wie beim SpinButton über `scaleSetDigits` eingestellt werden. Mögliche Konstruktoren sind `h/vScaleNew` mit einem Adjustment als Parameter oder `h/vScaleWithRange` mit Angabe der Intervallgrenzen und der Schrittweite.

3.2.6 Weitere Widgets

Alle verfügbaren Widgets aufzuzählen würde ohne Frage den Rahmen dieser Arbeit sprengen. Es sei aber noch erwähnt, dass natürlich die Möglichkeit besteht, Menüstrukturen und Toolbars zu implementieren, ein Statussymbol (wahlweise auch mit Popup-Menü) im System-Tray zu platzieren, eine Statusbar in einem Fenster einzurichten und einen Fortschrittsbalken anzuzeigen. Zudem gibt es vielseitige Möglichkeiten im Grafikbereich mit OpenGL und Cairo. Man werfe einfach einmal einen Blick in die Dokumentation, welche man auf der Projektseite finden kann.

3.3 Tastaturkürzel

Um grafische Oberflächen auch ohne Maus komfortabel benutzbar zu machen, kann man sogenannte Mnemonics (Tastaturkürzel) verwenden, über die Menüs, Buttons und Eingabefelder leicht erreicht werden können. Gekennzeichnet werden diese Abkürzungen durch Unterstreichungen in der Beschriftung eines Objekts. Aktiviert werden kann das zugehörige Objekt mit der Kombination Alt+[unterstrichener Buchstabe].

In Gtk2Hs ist es möglich, solche Mnemonics zu setzen. Allerdings gelang mir dies bisher nur unter Linux[1]. Unter Windows[2] war zwar eine Formatierung des Textes über Markup möglich, allerdings zeigten Mnemonics keinerlei Wirkung.

3.3.1 Schaltflächen

Tastaturkürzel auf Schaltflächen haben wir ja bereits in Abschnitt 3.2.1 in Kurzform kennengelernt. Wiederholend sei noch einmal der zugehörige Konstruktor genannt:

```
buttonNewWithMnemonic :: String → IO Button
```

Gekennzeichnet werden Mnemonics im Beschriftungsstring mit einem Unterstrich vor dem abkürzenden Buchstaben. Stock-Schaltflächen werden bereits voll ausgestattet mit Mnemonic geliefert. Wird ein Button über das Tastaturkürzel aktiviert, so werden die Signalbindungen `on/afterButtonActivate` und `on/afterButtonPressed` abgearbeitet.

[1] Fedora 12 i686, ghc 6.10.4, Gtk2Hs 0.10.1 mit Kompatibilitätspackage für neuen ghc
[2] Windows 7 32-bit, ghc 6.10.3, Gtk2Hs 0.10.1

3.3.2 Eingabefelder

Interessanter als die Mnemonics bei Schaltflächen sind die, mit denen auf Eingabefelder verwiesen werden kann. Da sich der Text eines Eingabefeldes sinnvollerweise ändern kann, ist es nicht möglich, eine Abkürzung direkt auf ein `Entry` zu legen. Man geht hierbei den Umweg über ein `Label`, dessen Mnemonic wir mit dem `Entry` verbinden[3].

Nehmen wir nun an, wir haben bereits ein Textfeld namens `entry` und möchten dieses via Tastaturkürzel erreichbar machen. Dazu erstellen wir ein neues Label mit Mnemonic und leiten das Kürzel mithilfe der Funktion `labelSetMnemonicWidget` auf unser Textfeld um.

```
label ← labelNewWithMnemonic "_Text"
labelSetMnemonicWidget label entry
```

Ebenso können wir natürlich das dem Label zugeordnete Kürzel auf jedes beliebige andere Widget umleiten.

3.3.3 Default Widgets

Eine weitere Tastaturabkürzung ist das Betätigen der Enter-Taste nach einer Eingabe im Textfeld. Um diesen Fall zu behandeln, können wir, wie in Abschnitt 3.2.2 gesehen, die Signalbindung `onEntryActivate` verwenden. Eine weitere Möglichkeit ist jedoch, das sog. „Default Widget“ zu aktivieren – mittels Attribut `entryActivatesDefault`.

Dies bietet sich an, wenn man mehrere Signals auf ein bestimmtes Widget umleiten möchte (zum Beispiel auf eine OK-Schaltfläche). Ebenfalls kann es gerade dann von Vorteil sein, wenn man dieses Widget variabel setzen möchte. Mit der Funktion `widgetGrabDefault` wird das möglich. Zu beachten ist allerdings, dass nur ein Widget mit dem Attribut `widgetCanDefault` auch ein Default Widget werden kann.

Gehen wir von einem Beispiel aus, in dem wir einem Button namens `button` die Default-Widget-Eigenschaft zuweisen möchten. Zunächst müssen wir ihm, sofern wir dies noch nicht direkt bei seiner Erstellung getan haben, die Eigenschaft `widgetCanDefault` mitgeben, ihn sich dann mittels `widgetGrabDefault` die Default-Eigenschaft holen lassen und anschließend noch unserem `Entry` namens `entry` sagen, dass es bei Aktivierung diese an das Default Widget weiterleiten soll.

```
set button [widgetCanDefault := True]
widgetGrabDefault button
entrySetActivatesDefault entry True
```

3.4 Dynamische Anpassung des GUIs

Man kann Widgets temporär ihre Funktionalität nehmen (z. B. eine Schaltfläche unklickbar oder den Inhalt eines Eingabefelds unerreichbar machen). Dies wird in Gtk2Hs mittels `widgetSetSensitivity` oder direkt über das Attribut `widgetSensitive` realisiert.

[3] Beim `Button` wird ein ähnlicher Weg gegangen – dort liegt das `Label` innerhalb der Schaltfläche. Es wird mit dem entsprechenden Konstruktor automatisch angelegt.

```
widgetSetSensitivity :: (WidgetClass self) ⇒ self → Bool → IO ()
```

Abbildung 14: Funktion, um die Benutzbarkeit eines Widgets festzulegen (`True`: Widget reagiert auf Benutzereingaben)

4 GUIs mit Glade

Wie die vorangehenden Kapitel bereits vermuten lassen, kann die Programmierung umfangreicherer GUIs sehr aufwendig werden. Daher erscheint es sinnvoll, sein Wunsch-GUI mithilfe eines grafischen Interface-Designers zusammenzustellen. Für GTK gibt es ein solches Programm – Glade.

Glade erstellt eine XML-Beschreibung des darin gebastelten GUIs, die dann im Programm verwendet werden kann. Problematisch und zugleich praktisch ist, dass die GUI-Beschreibung dem Programm (insbesondere der Binary) als externe .glade-Datei beiliegen muss. Auf diese Weise kann das GUI auch nach dem Kompilieren noch mittels Glade modifiziert werden. Solange die internen Bezeichnungen der Widgets nicht verändert werden, bleibt das Programm mit dem modifizierten GUI lauffähig.

4.1 Erstellen einer GUI-Beschreibung

Glade begrüßt den Benutzer in der aktuellen Version 3.6.7 mit einem Einstellungsdialog. Hier sollte man zunächst „Libglade“ als Format auswählen, damit Gtk2Hs die Beschreibung versteht. Anderenfalls bricht das Programm bereits beim Einlesen der Beschreibung ab.

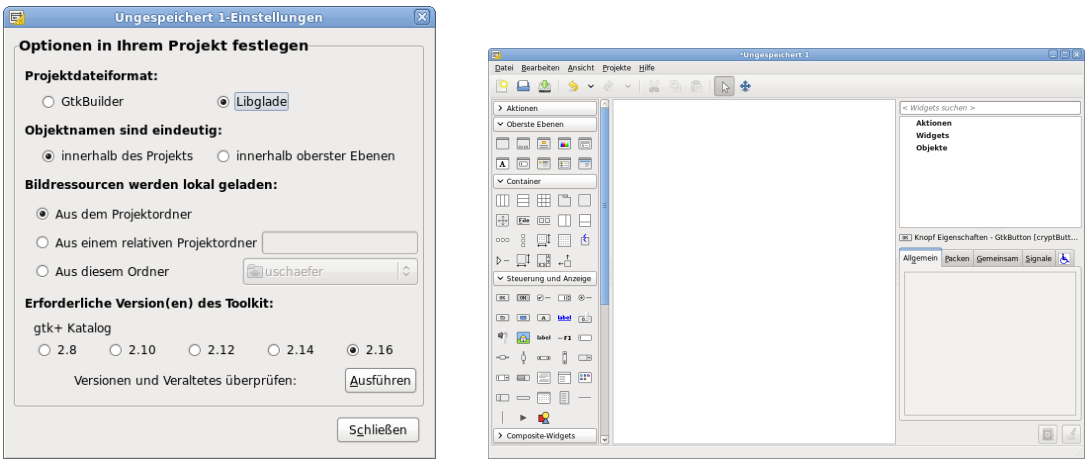


Abbildung 15: Glade-Einstellungsdialog und Hauptfenster

In Glade stellt man sich nun seine GUI-Beschreibung im WYSIWYG-Modus zusammen. Dies läuft analog zur Programmierung von Hand: Als erstes Element, „oberste Ebene“, wählt man eine Basis für das zu gestaltende Fenster. Anschließend

kommen bei Bedarf ein oder mehrere „Container“ hinzu und letztendlich die für den Benutzer interessanten benutzbaren Widgets – „Steuerung und Anzeige“.

4.2 Einlesen der GUI-Beschreibung

Zunächst benötigen wir in unserem Programm eine weitere Bibliothek, um Glade-Dateien einzulesen:

```
import Graphics.UI.Gtk.Glade
```

Damit stehen uns neue Funktionen zur Verfügung, mit denen wir Glade-XML verarbeiten können. `xmlNew` liest die aus Glade stammende XML-Beschreibung des Fensters ein und macht daraus ein Objekt vom Typ `GladeXML`, das im weiteren Programm verwendet wird. Da das Einlesen fehlschlagen kann, liefert `xmlNew` zunächst den Typ `Maybe GladeXML`. Mittels Fallunterscheidung (verbunden mit eventuellem Abbruch des Programms) erhalten wir den reinen Typ `GladeXML`, den wir im Weiteren verwenden können.

```
windowXmlM ← xmlNew "window.glade"
let windowXml = case windowXmlM of
                  (Just w) → w
                  Nothing  → error "GUI-Beschreibung fehlerhaft."
```

```
xmlNew :: FilePath → IO (Maybe GladeXML)
```

Abbildung 16: Einlesen der XML-Datei aus Glade, Typ von `xmlNew`

4.3 Verwendung der beschriebenen Widgets

An einem kleinen Beispielprogramm sei im Folgenden die Verwendung von Widgets aus einer Glade-Beschreibung gezeigt.

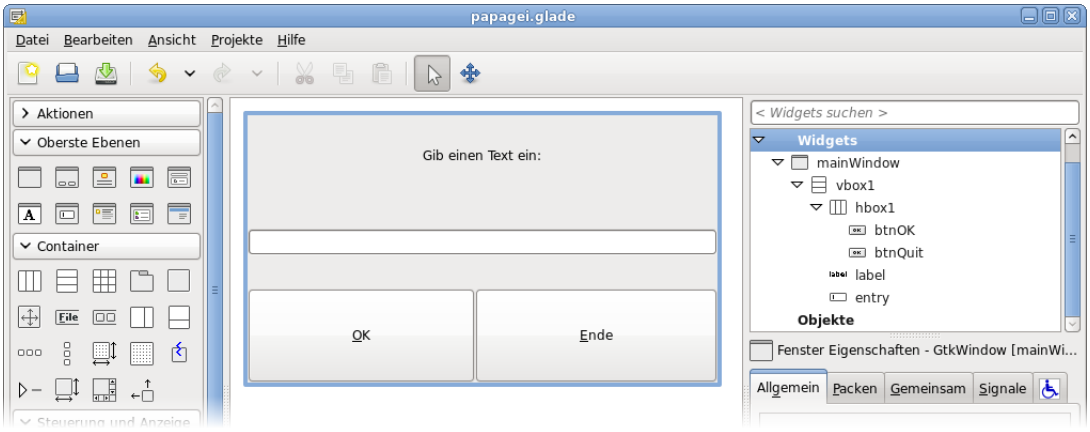


Abbildung 17: Beispiel-GUI in Glade mit Hierarchie

```
window  ← xmlGetWidget windowXml castToWindow "mainWindow"
entry   ← xmlGetWidget windowXml castToEntry  "entry"
label   ← xmlGetWidget windowXml castToLabel  "label"
btnQuit ← xmlGetWidget windowXml castToButton "btnQuit"
btnOK   ← xmlGetWidget windowXml castToButton "btnOK"
```

```
xmlGetWidget :: (WidgetClass widget)
  ⇒ GladeXML
  → (GObject → widget)   -- Cast-Funktion
  → String               -- Bezeichner des Widgets
  → IO widget
```

Abbildung 18: Bindung der Widgets aus dem GUI in Abbildung 17 an Variablen, Typ von `xmlGetWidget`

Die Funktion `xmlGetWidget` ermöglicht uns das Auslesen eines einzelnen Widgets aus der XML-Beschreibung. Identifiziert wird jedes Widget durch einen eindeutigen Namen, der in Glade in den zugehörigen Attributen festgelegt wird. Eine passende Cast-Funktion sorgt dafür, dass wir anstelle eines allgemeinen `GObject`s einen Widget-Typen bekommen, mit dem wir im weiteren Programm etwas anfangen können.

```
onClicked btnOK $ do
  text ← entryGetText entry
  labelSetText label ("Dein␣Text:␣" ++ text)
onEntryActivate entry (buttonClicked btnOK)
onClicked btnQuit mainQuit
```

Abbildung 19: Die entscheidenden `Signal`s werden an Funktionen gebunden

Die Verwendung des mittels Glade erstellten GUIs im weiteren Programmcode verläuft genau wie bei einem von Hand gecodeten GUI. Neu in diesem Beispiel ist, dass mit `buttonClicked` ein `Signal` per Code ausgelöst wird. Dies bietet sich an, da beim Drücken der Eingabetaste im `Entry` dasselbe passieren soll wie beim Klick auf den `Button`.

5 Standarddialoge

GTK stellt für verschiedene Anwendungen Standarddialoge zur Verfügung. Der wichtigste unter ihnen dürfte der einfache Ja-Nein-Dialog sein. In Gtk2Hs wird diese Dialogvariante als `MessageDialog` bezeichnet und kann – je nach Konstruktor-Parametern – in den verschiedensten Varianten generiert werden.

5.1 Textdialoge

```
messageDialogNew ::
     Maybe Window       -- Uebergeordnetes Fenster
  → [DialogFlags]       -- Einstellungen
  → MessageType         -- Art der Nachricht (legt Symbol fest)
  → ButtonsType         -- Button-Konstellation unter der Nachricht
  → String              -- Nachricht (erste Zeile)
  → IO MessageDialog
```

Abbildung 20: Konstruktor für einen `MessageDialog`

Analog zu `messageDialogNew` gibt es auch einen Konstruktor `messageDialogNewWithMarkup`, der anstelle eines normalen Strings einen Markup-String nimmt.

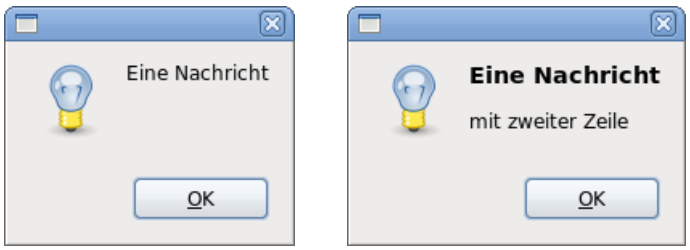
Erlaubte Werte für `MessageType` sind `MessageInfo`, `MessageWarning`, `MessageQuestion`, `MessageError` und `MessageOther` (letztgenannter sorgt dafür, dass der Dialog ohne Illustration generiert wird). Natürlich kann nachträglich auch ein eigenes Symbol in den Dialog eingebunden werden – hierbei hilft die Funktion `messageDialogSetImage`, welche den Dialog und das Bild übergeben bekommt.

Erlaubte Werte für `ButtonsType` sind `ButtonsNone`, `ButtonsOk`, `ButtonsClose`, `ButtonsCancel`, `ButtonsYesNo` und `ButtonsOkCancel`. Wer seine eigenen Buttons hinzufügen möchte, kann dies mit der Funktion `dialogAddButton` umsetzen. Sie generiert einen neuen Button mit angegebener Beschriftung und `ResponseId`.

```
dialogAddButton :: DialogClass self
  ⇒ self         -- betroffener Dialog
  → String       -- Beschriftung
  → ResponseId   -- Ergebnis des Dialogs, wenn angeklickt
  → IO Button
```

Abbildung 21: Typ von `dialogAddButton` (fügt einem Dialogfenster einen Button hinzu)

Jeder Button wird mit einer `ResponseId` ausgestattet, die später bei der Auswertung des Dialogs von Nutzen sein kann. Analog zu den Stock-Buttons gibt es die möglichen Werte `ResponseOk`, `ResponseCancel`, `ResponseClose`, `ResponseYes`, `ResponseNo`, `ResponseApply` und `ResponseHelp`. Hinzu kommen noch die Antwortkennungen `ResponseReject` und `ResponseAccept`, welche nur auf Wunsch des Programmierers geliefert werden. `ResponseNone` stammt von einem Widget ohne `ResponseId` oder resultiert aus dem Schließen des Fensters durch das Programm. Schließt der Benutzer hingegen das Fenster auf eine andere Weise als durch einen Button, so lautet die Antwort des Dialogs `ResponseDeleteEvent`. `ResponseUser Int` ist für Eigenkreationen des Programmierers reserviert. Durch den `Int`-Wert wird es möglich, mehrere benutzerdefinierte Buttons auseinanderzuhalten.



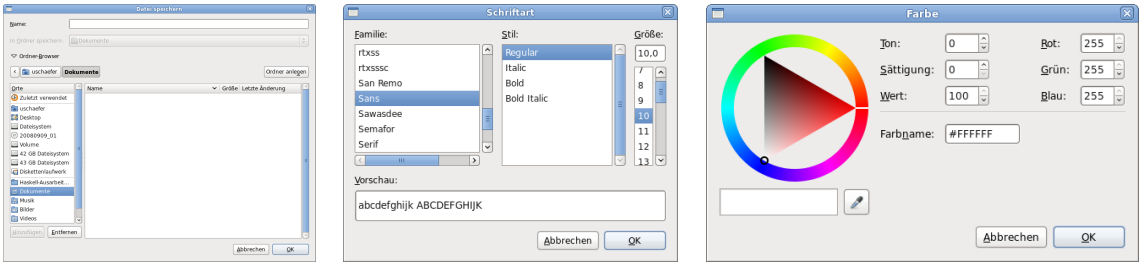

```
dialog ← messageDialogNew Nothing [] MessageInfo ButtonsOk
                                                    "Eine Nachricht"
messageDialogSetSecondaryText dialog "mit zweiter Zeile"

res ← dialogRun dialog
case res of
  ResponseOk → putStrLn "Dialog akzeptiert."
  _          → putStrLn "Das war so nicht gedacht..."
```

Abbildung 22: Veränderung der Textanzeige bei mehrzeiligen `MessageDialog`s und Beispielanwendung

Dialoge können, wie im Beispiel gezeigt, mit der Funktion `dialogRun` angezeigt werden. Die Abarbeitung eines imperativen Blocks stoppt an dieser Stelle und wird erst dann fortgesetzt, wenn ein Ergebnis vorliegt. Ebenso ist es aber auch möglich, den Dialog genau wie ein Fenster mit `widgetShowAll` anzuzeigen. Behandelt werden kann das Ergebnis dann ereignisgesteuert mit `onResponse` bzw. `afterResponse`.

5.2 Auswahldialoge



Die Gtk2Hs-Bibliothek unterstützt unter anderem GTK-Standarddialoge zum Öffnen/Speichern von Dateien und Ordnern (Buttons werden nach Bedarf hinzugefügt, der Dateifilter kann angepasst werden), zur Wahl einer Schriftart (der Vorschautext kann angepasst werden) und zur Farbwahl.

```
fcd ← fileChooserDialogNew (Just "Datei speichern") Nothing
                  FileChooserActionSave [("_Abbrechen", ResponseCancel),
                                         ("_OK", ResponseOk)]
fnd ← fontSelectionDialogNew "Schriftart"
cld ← colorSelectionDialogNew "Farbe"
```

Ist eine Wahl getroffen, so ist diese als Eigenschaft des Dialogs gespeichert und kann über die zugehörige Getter-Funktion bzw. das zugehörige Attribut ausgelesen werden.

5.3 Weitere Dialoge

Weiter ist es möglich mit `dialogNew` einen Dialog anzulegen, der vollkommen an die eigenen Bedürfnisse angepasst werden kann (in Glade können auch Fenster auf dieser Basis erstellt werden). Zudem gibt es die Möglichkeit, einen Standard-GTK-`AboutDialog` mit eigenen Inhalten zu füllen.

6 Fazit und Ausblick

Wir haben in dieser Arbeit einiges über die GUI-Programmierung mit Gtk2Hs erfahren. Wir haben gesehen, dass die Entwicklung benutzerfreundlicher und alltagstauglicher Programme in Haskell durchaus möglich ist.

Entwicklungsfortschritt Gtk2Hs ist bereits weit fortgeschritten und unterstützt einen großen Teil der Widgets, die das GTK liefert. Gleichzeitig habe ich aber beim Schreiben dieser Arbeit auch feststellen müssen, dass sich Gtk2Hs immer noch massiv in der Entwicklung befindet. Viele Demoprogramme und Tutorials, die man im Internet findet, verwenden Funktionen, die in der aktuellen Dokumentation schon als „deprecated“ markiert wurden und Compilerwarnungen verursachen. Auch macht sich die vorangehende Entwicklung darin bemerkbar, dass das Signal-Handling für einige Widgets vollkommen unterschiedlich verläuft. So findet man z. B. für ein `Entry` nur die Funktionen `after-` und `onEntryActivate`, um das Activate-Signal zu verarbeiten. Für ein `StatusIcon` hingegen sind aber die Funktion `after-` und `onActivate` bereits als veraltet markiert. Hier wird nun das allgemeine Signal `statusIconActivate` über die Funktionen `on` und `after` angesprochen.

Weiter macht sich ein leichtes Durcheinander darin bemerkbar, dass es für das Lesen und Schreiben von Attributen zumeist zwei Wege gibt – zum einen über spezielle Getter und Setter für einzelne Attribute, zum anderen über die allgemeinen Funktionen `set` und `get`. Bisher ist hier jedoch keiner der beiden Wege als veraltet gekennzeichnet.

GUI-Programmierung funktional? Bei all den praktischen Dingen, die uns die betrachtete Bibliothek ermöglicht, bleibt das Funktionale etwas auf der Strecke. Der Grundaufbau eines Programms erfolgt in imperativen `do`-Blöcken und auch die Reaktionen auf `Signal`s sind meist durch `do`-Blöcke festgelegt. Bleibt die Frage: Geht das Ganze auch etwas funktionaler? Lassen sich die Grundideen von Haskell auch auf die GUI-Programmierung übertragen?

Mit diesem Thema haben sich bereits viele Leute beschäftigt. Ein aktuelles Projekt zu funktionaler reaktiver Programmierung (FRP) läuft unter dem Namen „Grapefruit“ an der TU Cottbus seit 2007. Verfügbar ist zurzeit eine Release, die die Versionsnummer 0.0.0.0 trägt und erst einige wenige Widgets unterstützt.

Die Bibliothek ermöglicht das Programmieren von Widgets, die aufeinander reagieren. So gibt es ein Beispielprogramm „Codebreaker“, das eine Implementierung des bekannten Mastermind-Spiels ist (man muss eine vierstellige Kombination raten). Der Eingabe-Button wird in diesem Spiel erst dann aktiv, wenn eine gültige

Eingabe im Eingabefeld steht (vier Zahlen). Diese direkte Interaktion zwischen zwei Widgets gehört zu den Dingen, die allein mit Gtk2Hs derzeit nicht effizient lösbar sind. Wer sich weiter mit Grapefruit auseinandersetzen möchte, dem sei an dieser Stelle die Projekt-Homepage empfohlen: http://www.haskell.org/haskellwiki/Grapefruit

Schlusswort Abschließend bleibt festzustellen, dass die Arbeit mit Gtk2Hs trotz entwicklungsbedingter Verwirrungen Spaß macht und es auf diesem Wege für jeden Haskell-Programmierer eine schöne Möglichkeit gibt, seine Programme einer breiteren Öffentlichkeit zugänglich zu machen.