1

gWidgets: Overview

The gWidgets package provides a toolkit-independent interface for the R user to program graphical user interfaces from within R. Although the package provides much less functionality than using a native toolkit interface, gWidgets can be used to create moderately complex GUIs quickly and easily using a programming interface that is simpler and more familiar to the R user.

The gWidgets package started as a port to RGtk2 of the iWidgets interface, initially implemented only for Swing through rJava (**?**). The gWidgets package enhances that original interface in terms of functionality and implements it for multiple toolkits.

1.1 Installation, toolkits

The gWidgets package is installed and loaded as other R packages that reside on CRAN. This can be done through the function install.packages or in an R graphical front-end through a dialog called from the menu bar. The gWidgets package only provides the application programming interface (API). To actually create a GUI, one needs to have:

1. An underlying toolkit library. This can be either the Tk libraries, the Qt libraries or the GTK+ libraries. The installation varies for each and depends on the underlying operating system.
2. An underlying R package that provides an interface to the libraries. The tcltk package is a recommended pacakge for R and comes with the R software itself, the RGtk2 and Qt packages may be installed through R's package management tools.
3. a gWidgetsXXX package to link gWidgets to the R package. As of this writing, there are basically three such packages gWidgetsRGtk2, gWidgetsQt and gWidgesttcltk. The gWidgetsWWW package is an independent implementation for web programming that is more or less faithful to the API, but not commented on further in this chapter.

Not all features of the API are available in each package. The help pages in the gWidgets package describe the API, with the help pages in the toolkit

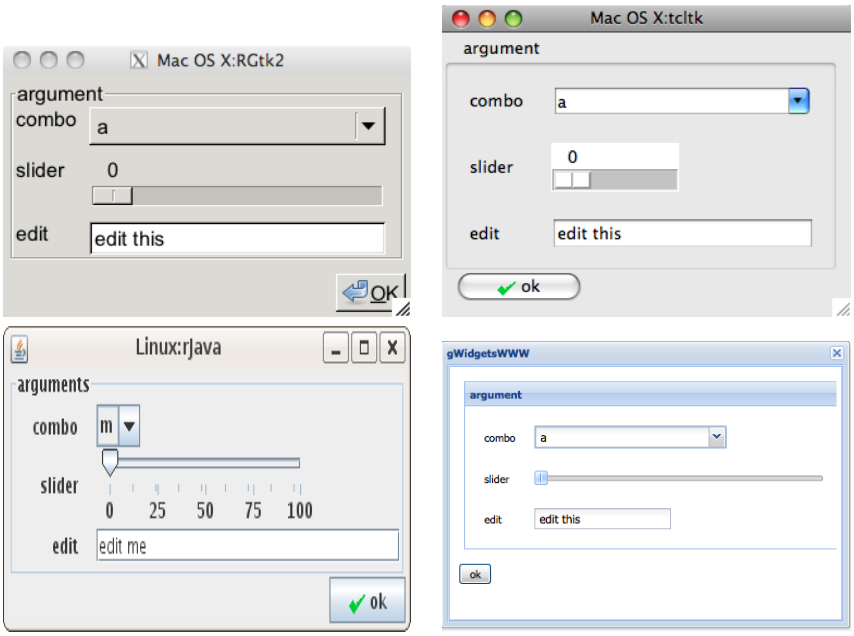

Figure 1.1: The `gWidgets` package works with different operating systems and different GUI toolkits. This shows, the same code using the `RGtk2`, `tcltk`, `qtbase` packages for a toolkit. Additionally, the `gWidgetsWWW` package is used in the lower right figure.

packages indicating differences or omissions from the API. For the most part, the omissions are gracefully handled by simply providing less functionality. We make note of major differences here, realizing that over time they may may be resolved. Consult the package documentation if in doubt.

Figure 1.1 shows how the same GUI code can be rendered differently depending on the OS and the toolkit.

XXX [insert Qt graphics here]

1.2 Startup

The `gWidgets` package is loaded as other R packages:

```
require(gWidgets)
```

A toolkit package is loaded when the first command is issued. If a user does not have a toolkit installed, a message instructs the user to install one.

If a user has exactly one toolkit package installed, then that will be used. But it is possible for more than one to be installed, in which case the user is prompted to choose one through an interactive menu. This choice can

be avoided by setting the option `guiToolkit` to the `XXX` in a `gWidgestXXX` package name, e.g.,

```
options("guiToolkit"="RGtk2")
```

Although in theory the different toolkits can be used together, in practice the different event loops created by each often lead to issues that can lockup the R process.

Example 1.1: A first GUI
As a first illustration of the use of `gWidgets`, a simple "hello world" type GUI can be produced through:

```
w <- gwindow("Hello world example")
b <- gbutton("Click me for a message", container=w)
addHandlerClicked(b, handler=function(h,...) {
  print("Hello world")
  dispose(w)
})
```

1.3 Constructors

GUI objects are produced by constructors. In Example 1.1 a top-level window and button constructor were called. In `gWidgets` most constructors have the following form:

```
gname(arguments, handler = NULL, action = NULL,
      container = NULL,... ,toolkit=guiToolkit())
```

where the `arguments` vary depending on the object being made.

In addition to creating a GUI object, a constructor also returns a useful R object. Except for modal dialog constructors, this is an S4 object of a certain class containing two components: `toolkit` and `widget`. The `toolkit` can be specified at time of construction allowing tookits, in theory, to be mixed. Otherwise, the `guiToolkit` function returns the currently selected toolkit, or queries for one if none is selected.

Constructors dispatch on the `toolkit` value to call the appropriate constructor in the toolkit implementation. The return value from the toolkit's constructor is kept in the `widget` component. Generic methods have a double dispatch when called. The first dispatch is based on the `toolkit` value and the method calls a second generic, implemented in the toolkit-specific package, with the same name as the first generic, except prefixed by a period (`svalue` calls `.svalue`). The tookit generic then dispatches based on the class of the `widget` argument and perhaps other arguments given to the generic. The actual class of the S4 object returned by the first constructor is (mostly) not considered, but when we refer to methods for an object, we gloss over

this double dispatch and think of it as a single dispatch. This design allows the toolkit packages the freedom to implement their own class structure.

As with most R objects, one calls generic functions to interact programatically with the object. Depending on the class, the `gWidgets` package provides methods, for the familiar S3 generics `[`, `[<-`, `dim`, `length`, `names`, `names<-`, `dimnames`, `dimnames<-`, `update`.

In addition, `gWidgets` provides the new generics listed in Table 1.3. These new generics provide a means to query and set the primary value of the widget (`svalue`, `svalue<-`), and various functions to effect the display of the widget (`visible<-`, `font<-`, `enabled<-`, `focus<-`). The methods `tag` and `tag<-` are implemented to bypass the pass-by-copy issues that can make GUI programming awkward at times.

The `gWidgets` API provides just a handful of generic functions for manipulating an object compared to the number of methods typically provided by a GUI toolkit for a similar object. Although this simplicity makes `gWidgets` easier to work with, one may wish to get access to the underlying toolkit object to work at that level. The `getToolkitWidget` will provide that object. We don't illustrate this, as we try to stay toolkit agnostic in our examples.

A few constructors create modal dialogs. These do not return the dialog object, because the dialog will be destroyed before the constructor returns. The R sesssion is unresponsive while waiting for user input. Consequently, modal dialogs have no methods defined for them. Instead, their constructors return values reflecting the user response to the dialog.

The `container` argument

The constructors produce two general types of objects: containers (Table 2.1) and components (the basic controls in Table 3.1 and the compound widgets in Table 3.9). A GUI consists of a heiarchical nesting of containers. Each container may contain contain controls or additional containers. In a GUI, except for top-level windows (including dialogs), every component and container is the child of some parent container. In `gWidgets` this parent is specified with the `container` argument when an object is constructed. This argument name can always be abbreviated `cont`. The package does not implement, layout managers, rather in the construction of a widget in `gWidgets`, the `add` method for the parent container is called with the new object as an argument and the values passed through the `...` argument as arguments. We remark that not all the toolkits (e.g., `RGtk2`, `qtbase`) require one to combine the construction of an object with the specification of the parent container. We don't illustrate this, as the resulting code is not cross-toolkit.

Table 1.1: Generic functions provided or used in gWidgets API.

Method	Description
svalue, svalue<-	Get or set value for widget
[, [<-	Refers to values in data store
length	length of data store
dim	dim of data store
names	names of data store
dimnames	dimnames of data store
update	Update widget values
size<-	Set size of widget in pixels
show	Show widget if not visible
dispose	Destroy widget or its parent
isExtant	Does R object refer to GUI object that still exists
enabled, enabled<-	Adjust sensitivity to user input
visible, visible<-	Adjust widget visibility.
focus<-	Sets focus to widget
defaultWidget<-	Set widget to have initial focus in a dialog
insert	Insert text into a multi-line text widget
font<-	Set a widget's font
tag, tag<-	Sets an attribute for a widget that persists through copies
getToolkitWidget	Returns underlying toolkit widget for low-level use

The handler and action arguments

The package provides a number of methods to add callbacks to different events. These main method is addHandlerChanged, which is used to assign a callback to a widget event. In addition, there are many "addHandlerXXX" methods to assign callbacks to other events, in the case where more than one event is of interest. For example, for single line text widgets, the addHandlerChanged responds when the user finishes editing, whereas addHandlerKeystroke is called each time the keyboard is used. Table 1.4 shows a list of the these other methods.

The arguments handler and action for a constructor assign the function specified to handler to be a callback for the addHandlerChanged event. In gWidgets, callbacks are functions with the signature (h,...) where h is a list containing the source of the event (the obj element), as well as user data that is specified when the callback is registered (the value passed through the action agument). Some toolkits pass additional arguments through the ... argument, so for portability this argument is not optional. For some classes, extra information is passed along, for instance for the drop target

generic, the component `dropdata` contains a string holding the drag-and-drop information.

If these few methods are insufficient, and toolkit-portability is not of interest, then the `addHandler` generic can be used to specify a toolkit-specific signal and a callback.

When an `addHandlerXXX` method is used, the return value is an ID or list of IDs. This can be used with the method `removeHandler` to remove the callback, or with the methods `blockHandler` and `unblockHandler` to temporarily block a handler from being called.

1.4 Drag and Drop

Drag and drop support is implemented through three methods: one to set a widget as a drag source, one to set a widget as a drop target, and one to call a handler when a drop event passes over a widget. The `addDropSource` method needs a widget and a handler to call when a drag and drop event is initiated. This handler should return the value that will be passed to the drop target. The default value is that returned by calling `svalue` on the object. The `addDropTarget` method is used to allow a widget to receive a dropped value and to specify a handler to call when a value is dropped. The `dropdata` component of the first callback argument, `h`, holds the drop data. The `addDropMotion` registers a handler for when a drag event passes over a widget.

Unfortunately, drag and drop is not well supported in `gWidgetstcltk`.

Table 1.2: Generic functions to add callbacks in `gWidgets` API.

Method	Description
`addHandlerChanged`	Primary handler call for when a widget's value is "changed." The interpretation of "change" depends on the widget.
`addHandlerClicked`	Sets handler for when widget is clicked with (left) mouse button. May return position of click through components `x` and `y` of the `h`-list.
`addHandlerDoubleclick`	Sets handler for when widget is double clicked
`addHandlerRightclick`	Sets handler for when widget is right clicked
`addHandlerKeystroke`	Sets handler for when key is pressed. The `key` component is set to this value if possible.
`addHandlerFocus`	Sets handler for when widget gets focus
`addHandlerBlur`	Sets handler for when widget loses focus
`addHandlerExpose`	Sets handler for when widget is first drawn
`addHandlerDestroy`	Sets handler for when widget is destroyed
`addHandlerUnrealize`	Sets handler for when widget is undrawn on screen
`addHandlerMouseMotion`	Sets handler for when widget has mouse go over it
`addHandler`	For non cross-toolkit use, allows one to specify an underlying signal from the graphical toolkit
`removeHandler`	Remove a handler from a widget
`blockHandler`	Temporarily block a handler from being called
`unblockHandler`	Restore handler that has been blocked
`addHandlerIdle`	Call a handler during idle time
`addPopupmenu`	Bind popup menu to widget
`add3rdMousePopupmenu`	Bind popup menu to right mouse click
`addDropSource`	Specify a widget as a drop source
`addDropMotion`	Sets handler to be called when drag event mouses over the widget
`addDropTarget`	Sets handler to be called on a drop event. Adds the component `dropdata`.

2

gWidgets: Containers

The gWidgets package provides a few useful containers: top-level windows, box containers, grid-like containers and notebook containers.

2.1 Top-level windows

The gwindow constructor creates top-level windows. The title of the window can be set during construction via the title argument or later through the svalue<- method. As well, the initial size can be set through the width and height arguments. This initial size is the default size, but may be adjusted later through the size method or through the window manager. The visible argument controls whether the window is initially drawn. If not drawn initially, the visible<- method, taking a logical value, can be used to draw the window later in a program. The default is to initially draw the window, but often it is good practice to suppress the initial drawing, especially for displaying GUIs with several controls as the incremental drawing of subsequent child components can make the GUI seem sluggish.

Windows can be closed programatically with the dispose method. Windows may also be closed through the window manager, by clicking a close icon in the title bar. The handler argument is called just before the window is destroyed, but will not prevent that from happening. The addHandlerUnrealize method can be used to call a handler between the initial click of the close icon and the subsequent destroy event of the window. This handler must return a logical value: if TRUE the window will not be destroyed, if FALSE the window will be, as illustrated in the example.

The initial placement of a window will be decided by the window manager, unless the parent argument is specified. If this is done with a vector of x and y pixel values, the upper left corner will be placed there. If it is specified as a gwindow instance, the new window will be positioned over the specified window and be disposed of when the parent widget is. This is useful, say, when a main window opens a dialog window to gather values.

Table 2.1: Constructors for container objects

Constructor	Description
`gwindow`	Creates a top-level window
`ggroup`	Creates a box-like container
`gframe`	Creates a container with a text label
`gexpandgroup`	Creates a container with a label and trigger to expand/collapse
`gpanedgroup`	Creates a container for two child widgets with a handle to assign allocation of space.
`glayout`	A grid container
`gnotebook`	A tabbed notebook container for holding a collection of child widgets

In most GUIs, the use of menubars, toolbars and statusbars is often reserved for the main window, while dialogs are not decorated so. In `gWidgets` it is suggested that these be added only to a top-level window.

Example 2.1: An example of `gwindow`

To illustrate, the following will open a new window. The initial drawing is postponed until after a button is placed in the window.

```
w1 <- gwindow("parent window", visible=FALSE)
b <- gbutton("a button", cont=w1)
visible(w1) <- TRUE
```

This shows how one might use the `parent` argument to specify where a subwindow will be placed.

```
w2 <- gwindow("child window", width=100, height=100,
              parent=w1)               # center on w1
b <- gbutton("button on child", cont = w2)
dispose(w1)                           # closes w2 also
```

This shows how the `addHandlerUnrealize` method can be used to intercept the closing of the window through the "close" icon of the window manager. The modal `gconfirm` dialog returns `TRUE` or `FALSE` depending on the button clicked, as will be explained in 3.8.

```
w <- gwindow("Close through the window manager")
id <- addHandlerUnrealize(w, handler=function(h,...) {
  !gconfirm("Really close", parent=h$obj)
})
```

Table 2.2: Container methods

Method	Description
`add`	Adds a child object to a parent container. Called when a parent container is specified to the `container` argument of the widget constructor, in which case, the ... arguments are passed to this method.
`delete`	Remove a child object from a parent container

2.2 Box containers

The container produced by `gwindow` is intended to contain just a single child widget, not several. This section demonstrates the box containers produced by `ggroup` that can be used to hold multiple child components. Through nesting, fairly complicated layouts can be produced.

The `ggroup` container

The `ggroup` box container provides an argument `horizontal` to specify whether the child widgets are packed in horizontally left to right (the default) or vertically from top to bottom. Unlike with the underlying graphical toolkits, there is no means to specify other styles of packing such as from the ends, or in the middle by some index.

add When packing in child widgets, the `add` method is used. In our examples, this is called internally by the constructors when the `container` argument is specified. The appropriate ... values for a constructor are passed to the `add` method. For `ggroup` the important ones are `expand` and `anchor`. When more space is allocated to a child then is needed by that child, the `expand=TRUE` argument will cause the child to grow to fill the available space in both directions. (No means is available in `gWidgets` to restrict to just one direction.) If `expand=TRUE` is not specified, then the `anchor` argument will instruct how to anchor the child into the space allocated. The direction is specified by x-y coordinates with both values from either -1, 0 or 1, where 1 indicates top and right, whereas -1 is left and bottom. The example will demonstrate their use.

delete The `delete` method can be used to remove a child component from a box container. In some toolkits, this child may be added back at a later time, but this isn't part of the API.

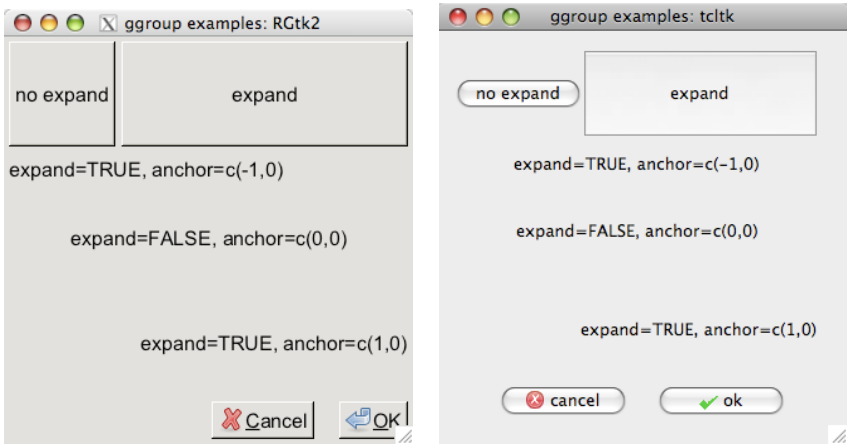

Figure 2.1: Use of `expand`, `anchor`, `addSpace` and `addSpring` with the `ggroup` constructor in `gWidgetsRGtk2` and `gWidgetstcltk`

Spacing and sizing For spacing between the child components, the argument `spacing` may be used to specify, in pixels, the amount of space between the child widgets. For box containers, this can later be set through the `svalue` method. The method `addSpace` can add space between two widgets packed next to each other, whereas the method `addSpring` will place an invisible spring between two widgets, forcing them apart. Both are useful for laying out buttons.

The overall size of `ggroup` container is controlled through it being a child of its parent container. However, a size can be assigned through the `size<-` method. This will be a preferred size, but need not be the actual size, as the container may need to be drawn larger to accomodate its children. The argument `use.scrollwindow` when specified as `TRUE` will add scrollbars to the box container so that a fixed size can be maintained. Although, it is generally considered a poor idea to use scrollbars when there is a chance the key controls for a dialog will be hidden.

Example 2.2: Example of `ggroup` usage

This example shows the nesting of vertical and horizontal box containers and the effect of the `expand` and `anchor` arguments. Figure 2.1 shows how it is implemented in two different toolkits.

```
w  <- gwindow("ggroup examples")
g  <- ggroup(cont=w, horizontal=FALSE, expand=TRUE)
g1 <- ggroup(cont=g, expand=TRUE)
b  <- gbutton("no expand", cont=g1)
b  <- gbutton("expand", cont=g1, expand=TRUE)
```

```
g2 <- ggroup(cont=g)
l  <- glabel("expand=TRUE, anchor=c(-1,0)", anchor=c(-1,0),
             expand=TRUE, cont=g2)
g3 <- ggroup(cont=g, expand=TRUE)
l  <- glabel("expand=FALSE, anchor=c(0,0)", anchor=c(0,0),
             expand=TRUE, cont=g3)
g4 <- ggroup(cont=g, expand=TRUE)
l  <- glabel("expand=TRUE, anchor=c(1,0)", anchor=c(1,0),
             expand=TRUE, cont=g4)
```

This demonstrates how one can use the `addSpace` and `addSpring` methods to right align buttons in a button bar.

```
g5 <- ggroup(cont=g, expand=FALSE)
addSpring(g5)
cancel  <- gbutton("cancel", cont=g5, handler=function(h,..) {
  dispose(w)
})
addSpace(g5, 12)
ok  <- gbutton("ok", cont=g5)
```

The next example shows an alternative to the expand group widget.

Example 2.3: The `delete` method of `ggroup`
This example shows nested `ggroup` containers and the use of the `delete` method to remove a child widget from a container. In this application, a box is set aside at the top of the window to hold a message that can be set via `openAlert` and closed with `closeAlert`. This example works better under `RGtk2`, as the space allocated to the alert is reclaimed when it is closed.

This code sets up the area for the alert box to appear from.

```
w <- gwindow("Alert box example")
g <- ggroup(horizontal=FALSE, cont = w)
alertBox <- ggroup(cont = g)
mainBox <- ggroup(cont = g, expand=TRUE)
l <- glabel("main box label", cont = mainBox, expand=TRUE)
ig <- NULL                                  # global
```

These two functions will open and close the alert box respectively. In this example we use the global value, `ig`, to store the inner group.

```
openAlert <- function(message="message goes here") {
  ig <<- ggroup(cont=alertBox)
  glabel(message, cont = ig)
}
closeAlert <- function() {
  if(!is.null(ig))
    delete(alertBox, ig)
  ig <<- NULL
}
```

The state of the box can be toggled programmatically via

```
QT <- openAlert("new message")                # open
QT <- closeAlert()                            # close
```

To elaborate on this, one might add a timer to close the alert after a specified time interval and a close icon so the user may dismiss the alert.

The `gframe` and `gexpandgroup` containers

Framed containers are used to set off elements and are provided by `gframe`. Expandable containers are used to preserve screen space unless requested and are provided by `gexpandgroup`. Both of these containers can be used in place of the `ggroup` container.

In addition to the `ggroup` arguments, the `gframe` constructor has the arguments `text` to specify the text marking the frame and `pos` to specify the positioning of the text, using 0 for left and 1 for right. If the toolkit supports markup, such as `RGtk2`, the `markup` argument takes a logical indicating if markup is being used in the specification of `text`. The `names` method can be used to get and set the label after construction of the widget.

The `gexpandgroup` constructor, like `gframe`, has the `text` argument, but no `pos` argument for positioning the text label. The widget has two states, which may be toggled either by clicking the trigger or through the `visible<-` method. A value of `TRUE` means the child is visible. The `addHandlerChanged` method is used to specify a callback for when the widget is toggled.

Example 2.4: The `gframe` and `gexpandgroup` containers
This example shows how the `gframe` container can be used.

```
w <- gwindow("gframe example")
f <- gframe(text="title", pos=1, cont=w)
l <- glabel("Some text goes here", cont=f)
names(f) <- "new title"
```

This is a similar example for `gexpandgroup`.

```
w <- gwindow("gexpandgroup example")
g <- gexpandgroup(text="title", cont=w)
l <- glabel("Some text goes here", expand=TRUE, cont=g)
visible(g) <- FALSE
visible(g) <- TRUE                        # toggle visibility
```

2.3 Paned containers: the `gpandedgroup` container

The `gpanedgroup` constructor produces a container which has two children which are separated by a visual gutter which can be adjusted using the mouse to allocate the space between the two children. The children are aligned

side-by-side (by default) or top to bottom if the horizontal argument is given as FALSE. The sash position can also be done programatically using the svalue<- method, where a value from 0 to 1 specifies the proportion of space allocated to the leftmost (topmost) child.

To add children, the container should be used as the parent container for two constructors. These can be other container constructors which is the typical usage for more complicated layouts. (For toolkits which support the separation of widget construction and layout, the gpanedgroup constructor can have two children specified to the arguments widget1 and widget2.)

Example 2.5: Paned groups
This example shows how one could use this container.

```
w <- gwindow("gpanedgroup example", visible=FALSE)
pg <- gpanedgroup(cont=w)
g <- ggroup(cont=pg)                    # left child
l <- glabel("left child", cont=g)
b <- gbutton("right child", cont=pg)
visible(w) <- TRUE
```

To adjust the sash position, one can do:

```
svalue(pg) <- 0.75
```

2.4 Tabbed notebooks: the gnotebook container

The gnotebook constructor produces a tabbed notebook container. The constructor has the argument tab.pos to specify the location of the tabs. A value of 1 through 4 with 1 being botton, 2 left side, 3 top and 4 right side being used, with the default being 3. The closebuttons argument takes a logical indicating whether the tabs should have close buttons on them. In this case, the argument dontCloseThese can be used to specify which tabs, by index, should not be closable. (Some toolkits do not implement these features though.)

The add method for the notebook container uses the label argument to specify the tab label. As add is called implicitly when a a widget is constructed, this argument is usually specified to the constructor.

Methods The svalue method returns the index of the currently raised tab, whereas svalue<- can be used to switch the page to the specified tab. The currently shown tab can be removed using the dispose method. To remove a different tab, use this method in combination with svalue<-. When removing many tabs, you may want to start from the end as otherwise the tab positions change, which can be confusing when using a loop. The names

method can be used to retrieve the tab names, and `names<-` to set the names. The `length` method returns the number of pages held by the notebook.

Example 2.6: Tabbed notebook example
A simple example follows. The `label` argument is passed along from the constructor to the `add` method for the notebook instance.

```
w <- gwindow("gnotebook example")
nb <- gnotebook(cont=w, tab.pos=3)
l <- glabel("first page", cont=nb, label="one")
b <- gbutton("second page", cont=nb, label="two")
```

To set the page to the first one:

```
svalue(nb) <- 1
```

To remove the first page (the current one)

```
dispose(nb)
```

2.5 Grid layout: the `glayout` container

The layout of dialogs and forms is usually seen with some form of alignment between the widgets. The `glayout` constructor provides a grid container to do so, using matrix notation to specify location of the children. The argument `homogeneous` can be used to specify that each cell take up the same size, the default is `FALSE`. Spacing between each cell may be specified through the `spacing` argument.

Children may be added to the grid at a specific row and column, and a child may span more than one row or column. To specify this, R's matrix notation, `[<-`, is used with the indices indicating the row and column. When a child is to span more than one row or column, the corresponding index should be a vector of indices indicating so. There is no `[` method defined to return the child components. To add a child, the `glayout` container should be specfied as the `container` and be on the left hand side of the `[<-` call. For convenience, if the right hand side is a string, a label will be generated. To align a widget within a cell, the `anchor` argument of the `[<-`glayout method is used. The example illustrates how this can be used to achieve a center balance.

Example 2.7: Layout with `glayout`
This example shows how a simple form can be given an attractive layout using a grid container. It uses the `gedit` constructor to provide a single-line text entry widget. As the matrix notation does not have a means to return the child widget (a `[` method say), we store the values of the `gedit` widgets into variables.

```
w <- gwindow("glayout example")
tbl <- glayout(cont=w)
right <- c(1,0); left <- c(-1,0)
tbl[1,1, anchor=right] <- "name"
tbl[1,2, anchor=left ] <- (name <- gedit("", cont=tbl))
tbl[2,1, anchor=right] <- "rank"
tbl[2,2, anchor=left ] <- (rank <- gedit("", cont=tbl))
tbl[3,1, anchor=right] <- "serial number"
tbl[3,2, anchor=left ] <- (snumber <- gedit("", cont=tbl))
```

3

gWidgets: Control Widgets

3.1 Basic controls

This section discusses the basic controls provided by `gWidgets`.

Buttons, Menubars, Toolbars

The button widget allows a user to initiate an action through clicking on it. Buttons have labels – usually verbs indicating action – and often icons. The `gbutton` constructor has an argument `text` to specify the text. For text that matches the stock icons of `gWidgets`, an icon will also be rendered. A list of stock icons is returned by `getStockIcons`. In common with the other controls, the argument `handler` is used to specify a callback and the `action` argument will be passed along to this callback (unless it is a `gaction` object, whose case is described below).

The default handler is the click handler which can be specified at construction, or afterward through the `addHandlerClicked`.

A new button may or may not have the focus when a GUI is constructed. If it does have the focus, then the return key will initiate the button click signal. To make a GUI start with its focus on a button, the `defaultWidget` method is available.

The `svalue` method will return a button's label, and `svalue<-` is used to set the label text. Most GUIs will make a button insensitive to user input if the button's action is not currently permissible. Toolkits draw such buttons in a greyed out state. The `enabled<-` method can set or disable whether a widget can accept input.

Ab basic example of a button with a handler was given in Example **??**.

Actions

In GUI programming an action is a reusable code object that can be shared among buttons, toolbars, and menubars. Common to these three controls are that the user expects some "action" to occur when a value is selected. For

Table 3.1: Table of constructors for control widgets in gWidgets. Most, but not all, are implemented for each toolkit.

Constructor	Description
glabel	A text label
gbutton	A button to initiate an action
gcheckbox	A checkbox
gcheckboxgroup	A group of checkboxes
gradio	A radio button group
gcombobox	A drop-down list of values, possible editable
gtable	A table (vector or data frame) of values for selection
gslider	A slider to select from a sequence value
gspinbutton	A spinbutton to select from a sequence of values
gedit	Single line of editable text
gtext	Multi-line text edit area
ghtml	Display text marked up with HTML
gdf	Data frame viewer and editor
gtree	A display for heirarchical data
gimage	A display for icons and images
ggraphics	A widget containing a graphics device
gsvg	A widget to display SVG files
gfilebrowser	A widget to select a file or directory
gcalendar	A widget to select a date
gaction	A resusable definition of an action
gmenubar	Adds a menubar on a top-level window
gtoolbar	Adds a toolbar to a top-level window
gstatusbar	Adds a status bar to a top-level window
gtooltip	Add a tooltip to widget
gseparator	A widget to display a horizontal or vertical line

example, some save dialog is summoned, or some page is printed. Actions contain enough information to be displayed in several manners. An action would contain some text, an icon, perhaps some keyboard accelerator, and some handler to call when the action is selected. When a particular action is not possible due to the state of the GUI, it should be disabled, so as not to be sensitive to user interaction.

Actions in gWidgets are created through the gaction contstructor. The arguments are label, tooltip, icon, key.accel and the standard handler and action. The label appears as the text on a button, the menu item or toolbar text, whereas the icon will decorate the same if possible. For some toolkits, the tooltip pops up when the mouse hovers. (See also the tooltip<- method.)

methods The main methods for actions are `svalue<-` to set the label text and `enabled<-` to adjust whether the widget is sensitive to user input. All instances of the action are set through one call. In some toolkits, such as `RGtk2`, actions are bundled together into action groups. This grouping allows one to set the sensitivity of related actions at once. In R, one can store like actions in a list, and get similar functionality by using `sapply`, say.

buttons An action can be assigned to a button by setting it as the `action` argument of the `gbutton` constructor, in which case all other arguments for the constructor are ignored.

Otherwise, actions are used as list components which define the toolbar or menubar, as described in the following.

Toolbars

Toolbars and menubars are implemented in `gWidgets` using `gaction` items. Toolbars (and menubars) are specified using a named list of menu components. This is similar to how `RGtk2` can use an XML specification to define a user interface, but unlike how menubars and toolbars can be created one item at a time in the toolkits.

For a toolbar, the list has a simple structure. The list has named components each of which either describes a toolbar item or a separator. The toolbar items are specified by `gaction` instances and separators by `gseparator` instances with no container specified.

The `gtoolbar` constructor takes as its first argument the list. As toolbars belong to the window, the corresponding `gWigdets` objects use a `gwindow` object as the parent container. (Some of the toolkits relax this.) The argument `style` can be one of `"both"`, `"icons"`, `"text"`, or `"both-horiz"` to specify how the toolbar is rendered. Toolbars in `gWidgetstcltk` are not native widgets, so the implementation uses aligned buttons.

Menubars, popup menus

Menubars and popup menus are specified in a similar manner as toolbars with menu items being defined through `gaction` instances, and visual separators by `gseparator` instances. Menus differ from toolbars, as sub-menus give a nested structure. This structure is specified using a nested list as the component to describe the sub menu. The lists all have named components, in this case the corresponding name is used to label the sub menu item. For menu bars, it is typical that all the top-level components be lists, but for popup menus, this wouldn't necessarily be the case.

In Mac OS X with a native toolkit, menubars may be drawn along the top of the screen, as is the custom of that OS.

Methods The main methods for toolbar and menubar instances are the `svalue` method which will return the list. Whereas, the `svalue<-` method can be used to redefine the menubar or toolbar. Use the `add` method to append to an existing menubar or toolbar, again using a list to specify the new items.

Example 3.1: Menubar and toolbar example

The following commands create some standard looking actions. The handler `f` is just a stub to be replaced in a real application.

```
f <- function(...) print("stub")        # a stub
aOpen <- gaction("open", icon="open", handler = f)
aQuit <- gaction("quit", icon="quit", handler = f)
aUndo <- gaction("undo", icon="undo", handler = f)
```

A menubar and toolbar are specified through a list with named components, as is illustrated next. The menubar list uses a nested list with named components to specify a submenu.

```
tl <- list(open = aOpen, quit = aQuit)
ml <- list(File = list(
             open = aOpen,
             sep = gseparator(),
             quit = aQuit),
           Edit = list(
             undo = aUndo
             ))
```

Menubars and toolbars are added to top-level windows, so their parent containers are `gwindow` objects.

```
w <- gwindow("Example of menubars, toolbars")
mb <- gmenu(ml, cont=w)
tb <- gtoolbar(tl, cont=w)
l <- glabel("Test of DOM widgets", cont=w)
```

By disabling a `gaction` instance, we change the sensitivity of all its realizations. Here this will only affect the menu bar.

```
enabled(aUndo) <- FALSE
```

An "undo" menubar item, often changes its label when a new command is performed, or the previous command is undone. The `svalue<-` method can set the label text. This shows how a new command can be added and how the menu item can be made sensitive to mouse events.

```
svalue(aUndo) <- "undo: command"
enabled(aUndo) <- TRUE
```

Good GUI building principles suggest that one should not replace values in the a menu, rather one should simply disable those that are not being used. This allows the user to more easily become familiar with the possible menu items. However, it may be useful to add to a menu or toolbar. The `add` method can do so. For example, to add a help menu item to our example one could do:

```
hl <- list(help = list(
             help = gaction("manual", handler=f)
             ))
add(mb, hl)
```

Popup menus Popup menus can be created for a right click event through the `add3rdMousePopupmenu` constructor. (Or control-button-1 for Mac OS X.) This constructor has arguments `obj` to specify a widget, like a button, to initiate the popup, `menulist` to specify the menu and optionally an `action` argument.

Example 3.2: Popup menus

```
w <- gwindow("Popup example")
b <- gbutton("click me or right click me", cont=w,
             handler=function(h, ...) {
               cat("You clicked me\n")
             })
f <- function(h,...) cat("you right clicked on", h$action)
mbList <- list(one = gaction("one", action="one", handler=f),
               two = gaction("two", action="two", handler=f)
               )
add3rdMousePopupmenu(b, mbList)
```

3.2 Text widgets

A number of widgets are geared toward the display or entry of text. The `gWidgets` API defines `glabel` for displaying a single-or multiple-line string of static text, `gstatusbar` to place message labels at the foot of a window, `gedit` for a single line of editable text, and `gtext` for multi-line, editable text. For some toolkits, a `ghtml` widget is also defined, but neither `RGtk2` or `tcltk` have this implemented.

Labels

The `glabel` constructor produces a basic label widget. The label's text is specified through the `text` argument. This is a character vector of length 1

or is coerced into one by collapsing the vector with newlines. The `svalue` method will return the text as a single string, and the `svalue<-` method can be used to set the text programatically. The `font<-` method can also be used to set the text markup (Table 3.2). For some toolkits, the argument `markup` for the constructor takes a logical value indicating if the text is in the native markup language (PANGO for `RGtk2`).

The widget constructor also has the argument `editable`, which when specified as `TRUE` will add a handler to the event that the label is clicked that allows the text to be edited. Although this is popular in some familiar interfaces, say the tab in a spreadsheet, it has not proven to be intuitive to most users, as typically labels are not expected to change.

Statusbars

Statusbars are simply labels placed at the bottom of a window to leave informative, but non-disruptive, messages for the user. The `gstatusbar` constructor provides this widget. The argument `text` can be given to set the intial text away from its default of no message. Subsequent changes are made through the `svalue<-` method. As with toolbars and menubars, a top-level window should be specified for the `container` argument.

Single-line, editable text

The `gedit` constructor produces a widget to display a single line of editable text. The initial text can be set through the `text` argument. If it is desirable to set the width of the widget, the `width` argument allows the specification in terms of number of characters allowed to display without horizontal scrolling. The width of the widget may also be specified in pixel size through the `size` method.

Methods The text is returned by the `svalue` method and may be set through the `svalue<-` method. The `svalue` method will return a character vector by default. However, it may be desirable to use this widget to collect numeric values or perhaps some other type of variable. One could write code to coerce the character to the desired type, but it is sometimes convenient to have the return value be a certain non-character type. In this case, the `coerce.with` argument can be used to specify a function of a single argument to call before the value is returned by `svalue`.

Some toolkits allow type-ahead values to be set. These values anticipate what a user wishes to type and offers a means to complete a word. The `[<-` method allows these values to be specifed through a character vector, as in `obj[] <- values`.

Handlers The default handler for the `gedit` widget is called when the text area is "activated" through the return key being pressed. Use `addHandlerBlur` to add a callback for the event of losing focus. The `addHandlerKeystroke` method can assign a handler to be called when a key is released. For the toolkits that support it, the specific key is given in the `key` component of the list `h` (the first component).

Example 3.3: Validation
In web programming it is common to have textarea entries be validated prior to their values being submitted. By validating ahead of time, the programmer can avoid the lag created by communicating with the server when the input is not acceptable. However, despite this lag not being the case for the GUIs considered now, it may still be a useful practice to validate the values of a text area when the underlying handlers are expecting a specific type of value.

The `coerce.with` argument can be used to specify a function to coerce values after an action is initiatied, but in this example we show how to validate the text widget when it loses focus. If the value is invalid, we set the text color to red.

```
w <- gwindow("Validation example")
validRegexpr <- "[[:digit:]]{3}-[[:digit:]]{4}"
tbl <- glayout(cont=w)
tbl[1,1] <- "Phone number (XXX-XXXX)"
tbl[1,2] <- (e <- gedit("", cont = tbl))
tbl[2,2] <- (b <- gbutton("submit", cont = tbl,
                          handler=function(h,...) print("hi")))
## Blur is focus out event
addHandlerBlur(e, handler = function(h,...) {
  curVal <- svalue(h$obj)
  if(grepl(validRegexpr, curVal)) {
    font(h$obj) <- c(color="black")
  } else {
    font(h$obj) <- c(color="red")
    focus(h$obj) <- TRUE
  }
})
```

Multi-line, editable text

The `gtext` constructor produces a multi-line text editing widget with scrollbars to accommodate large amounts of text. The `text` argument is for specifying the intial text. The initial width and height can be set through similarly named arguments, which is useful under `tcltk`.

The `svalue` method retrieves the text stored in the buffer. If the argument `drop=TRUE` is specified, then only the currently selected text will be returned.