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 the native toolkits through their R bindings, gWidgets can be used to create moderately large GUIs quickly and easily using a programming interface that is simpler and familiar for the R user to learn.

The gWidgets package started as a port to RGtk2 of the iWidgets interface for rJava (**?**). The gWidgets package enhances that original interface in terms of functionality and is now toolkit independent.

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 on some OSes through a dialog called from the menu bar. The gWidgets package provides the programming interface only (API). To actually create a GUI, one needs to have the underlying toolkit libraries, an underlying R package that provides an interface to the libraries, and a gWidgetsXXX package to link gWidgets to the R package. The installation of the libraries varies depending on toolkit and OS. Some details are presented by the function installing_gWidgets_toolkits.

As of this writing, there are basically complete gWidgets packages for the toolkit packages RGtk2 and tcltk; gWidgetsWWW ports the API to allow interactive web pages (cf. Chapter **??**). Also there is gWidgetsrJava, but that package is not being supported. Here, only the implementations for the RGtk2 and tcltk packages are covered.

Not all features of the API are available in each package. The help pages in gWidgets describe the API, with the help pages in the toolkit 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 these differences for gWidgetsRGtk2 and gWidgetstcltk here, realiz-

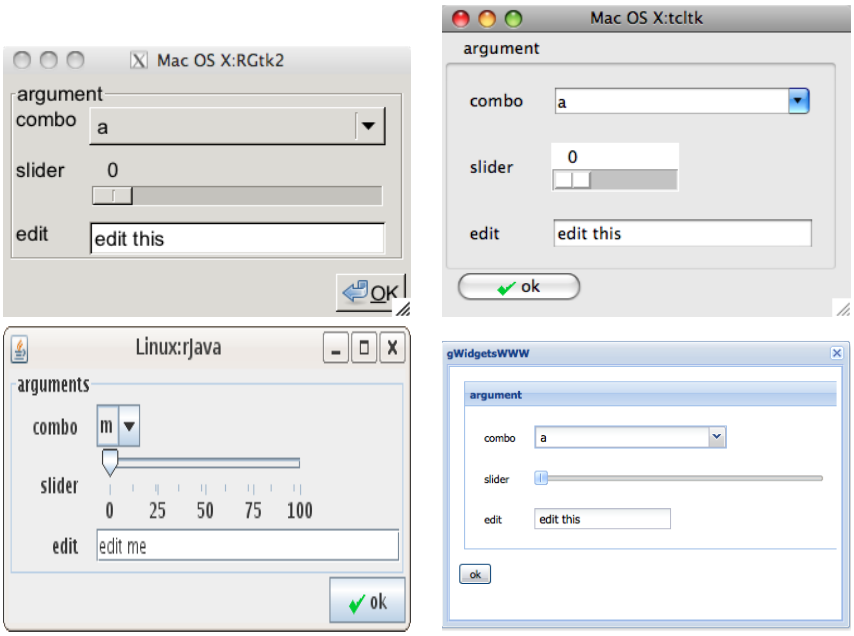

Figure 1.1: The `gWidgets` package works with different operating systems and different GUI toolkits. This shows, the same code for `RGtk2`, `tcltk`, `rJava` and `gWidgetsWWW`. Note, each toolkit has it's own sizing ideas for the controls.

ing 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.

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, then a message indicating the need to install an appropriate package is given.

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 underlying R package name, as in

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

The value is the name of one of the R packages that gWidgets can use. [1] Although in theory the different toolkits can be used together, in practice the different eventloops created by each often lead to issues that can lockup the R process.

1.3 Constructors

GUI objects are produced by constructors. 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.

Return value

Not only do constructors create visible GUI objects they also return 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 funtion returns the currently selected toolkit, or queries for one if none is selected point about dispatch 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 which calls a second generic implemented in the toolkits with a different name (svalue dispatches to .svalue). The tookit generic, then dispatches based on the class of widget component 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. The gWidgets package provides some familiar S3 methods, for the appropriate objects, for the familiar generics [, [<-, dim, length, names, names<-, dimnames, dimnames<-, update. In addition, it provides 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 methods to effect the display of the widget (visible <- , font <- , enabled <- , focus <-).

[1] As of writing, this is either RGtk2, tcltk, or rJava. The gWidgetsWWW package does not use gWidgets for dispatch, rather it is loaded directly.

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
id, id <-	A unique ID for a widget
getToolkitWidget	Returns underlying toolkit widget for low-level use

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 objects, as when they are visible the R sesssion is unresponsive. Consequently they have no methods defined for them. Instead, these constructors return values produced by the dialogs, which in turn can be used as arguments to functions.

The container argument

The constructors produce two 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 which in turn contain other containers or components. In a GUI, except for top-level windows and modal 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`. 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`) 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.

The `handler` and `action` arguments

For all the toolkits, when the user initiates some event with the mouse or keyboard, the underlying toolkit will emit some signal. The toolkits allow callbacks to be called when these signals are emitted allowing the GUI to be made interactive. In `gWidgets`, the callbacks are functions with signature `(h,...)` where `h` is a list that contains user data. Some toolkits pass information through the `...` argument. As this is not portable across toolkits, we do not use this here. In general, the `obj` component of `h` contains the widget that the callback is assigned to, and the component `action` contains user-specified data passed through the `action` argument of the contructor or the `addHandlerXXX` method (the `XXX` may be one of several values). 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.

A callback for an event can be specified through the `handler` argument of a constructor, or added at a later time through the "`addHandlerXXX`" methods. The generic `addHandlerChanged` can be used to add a callback for the most reasonably defined event. In many cases, more than one event is reasonable. 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. 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 a `addHandlerXXX` method is used, the return value is an ID. This ID 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 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 specified 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 the value returned by `svalue` method 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-argument list, `h`, holds the drop data in the call to the handler. The `addDropMotion` is used to call a handler for the event that a drag event passes over a widget.

Unfortunately, the drag and drop implememtions in the toolkits are not all the same. Some toolkits simply use the native drag and drop support, which can not be changed.

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

Method	Description
`addHandlerChanged`	Refers to the signal that is bound towhen the `handler` argument is used by theconstructor. Interpretation varies from widget to 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 ispressed. The `key` component is set to this valueif 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 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, unlesss 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 the document object model, 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 and required by some toolkits.

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 `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, 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.

Spacing and sizing For spacing between the child components, the argument `spacing` may be used to specify, in pixels, the amount of space between

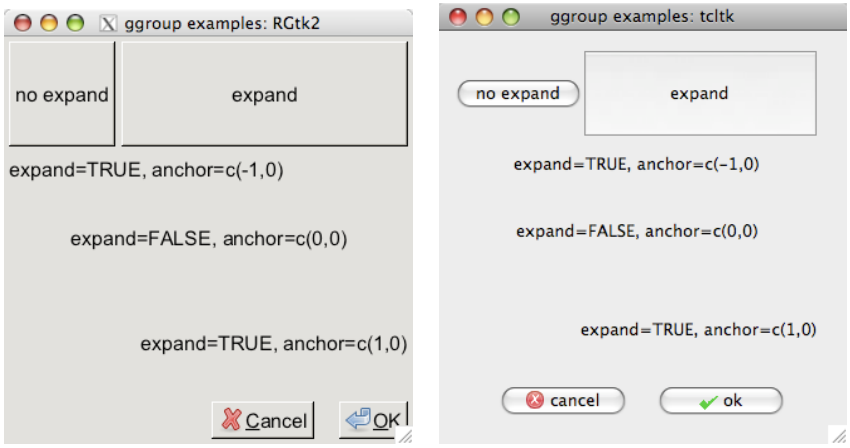

Figure 2.1: Use of `expand`, `anchor`, `addSpace` and `addSpring` with the `ggroup` constructor in `gWidgetsRGtk2` and `gWidgetstcltk`

the child widgets. 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 might use the `addSpace` and `addSpring` methods in a button bar.

```
g5 <- ggroup(cont=g, expand=FALSE)
addSpring(g5)
b  <- gbutton("cancel", cont=g5, handler=function(h,..) dispose(w))
addSpace(g5, 10)
b  <- 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() delete(alertBox, ig)
```

The state of the box can be toggled programattically via

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

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 expanded.

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. The children are aligned side-by-side by default, or top to bottom if the horizontal argument is given as FALSE. These two children are separated by a visual gutter which can be adjusted using the mouse to allocate the space between the two children. This 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) <- .5
```

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. (Not available for all toolkits.) 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 this is called implicitly when a a widget is constructed, this argument is 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, 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 click handler can be specified at construction, or afterward through the `addHandlerClicked` which is also aliased to the generic `addHandlerChanged`.

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 button's label. The method `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.

Example 3.1: Hello world button
This example shows how a button is assigned a handler to respond to click events. [1] When working with handlers, one can use an object name that will

[1] Each toolkit has its idiosyncracies. If this example is run using `RGtk2` the button will stretch to fill the space. At times this is not desired. Placing the button within a `ggroup` container can

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
gradio	A radio button group
gcheckbox	A checkbox
gcheckboxgroup	A group of checkboxes
gcombobox	A drop-down list of values
gtable	A table (vector or data frame) of values for selection
gslider	A slider to select a value
gspinbutton	A spinbutton to select from a set 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
gfilebrowser	A widget to select a file or directory
gcalendar	A widget to select a date
gaction	A resusable definition of an action
gmenubar	Puts 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 tooltip to widget
gseparator	A widget to display a horizontal or vertical line

be found through R's scoping rules, or the components passed through the h argument, as below.

```
w <- gwindow("Button example")
b <- gbutton("Click me", cont=w)
id <- addHandlerChanged(b, action=w, handler=function(h,...) {
  btnText <- svalue(h$obj)                    # or svalue(b)
  svalue(h$obj) <- paste("don't", btnText, "again") # set text
  enabled(h$obj) <- FALSE
  svalue(h$action) <- "Button example is finished" # set title
})
```

prevent this. Whereas, under tcltk the parent window will shrink to fit the button. The size method can prevent this if it is not desired.

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 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, as may be done through the `tooltip<-` method for `gWidgets` objects.

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 allows one to easy 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`.

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

The document object model, when employed, often relies on the custom that the main window showing the document possibly has a menu bar, a toolbar and a status bar. Subwindows, such as those for dialogs, would not have these decorations to emphasize their role in the GUI.

In `gWidgets`, toolbars (and menubars) are specified ahead of time as as a named list. 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. (Alternatively, these items can be specified through lists as described in the manual page.)

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.

The main constructor `gmenu` has its first arugment to specify the list, and the `container` argument to specify the top-level window.

In Mac OS X, toolbars 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 3.2: 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 named list, as is illustrated next. The menubar list, has a nested list specifying 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 should be `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 3.3: 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, "\n")
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 GUIs, say the tab in a spreadsheet, it has not proven to be intuitive to most users, as typically labels are not expected to change. In the wxWidgets toolkit labels are constructed by a function named `staticText` to emphasize this permanence.

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 it losing focus or the return key being pressed. 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 3.4: 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. The use of a modal dialog is a bit extreme here, a more user-friendly correction is suggested.

```
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))
```

Table 3.2: Possible specfications for setting font properties. Font values of an object are changed with named vectors, as in `font(obj) <- c(weight="bold", size=12, color="red")`

Attr	Possible value
weight	light, normal, bold
style	normal, oblique, italic
family	normal, sans, serif, monospace
size	a point size, such as 12
color	a named color

```
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$obj)
  if(length(grep(validRegexpr, curVal)) == 0) {
    focus(h$obj) <- TRUE
    gmessage("not valid")
  }
})
```

Multi-line, editable text

The `gtext` constructor produces a multi-line text editing widget with scrollbars to accomodate large amounts of text. The `text` argument is for specifying the intial text. This text can have a font attribute specified through the `font.attr` argument. This argument takes the same values as the `font <-` method. 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. Text in multiple lines is returned as a single string with "`\n`" separating the lines.

The contents of the text buffer can be replaced with the `svalue <-` method. To add text to a buffer, the `insert` method is used. The signature is `insert(obj,text)` where `text` is a character vector. New text is added to the end of the buffer. The font for newly added text can be set with the `font.attr` argument. The font for the selected text can be set with the `font <-` method. To clear the text buffer, the `dispose` method is used.

As with, `gedit`, the `addHandlerKeystroke` method is used to set a handler to be called for each keystroke. This is the default handler.

Example 3.5: A calculator

The following example shows how one might use the widgets just discussed to make a GUI that resembles a calculator. Which may offer familiarity to new R users, although certainly is no replacement for a command line.

The `glayout` container is used to neatly arrange the widgets. This example illustrates how a child widget can span a block of multiple cells by using the appropriate indexing. Furthermore, the `spacing` argument is used to tighten up the appearance. The example also illustrates a useful strategy of storing the widgets using a list for subsequent manipulations.

The following sets up the layout of the display and buttons.

```
buttons <- rbind(c(7:9, "(", ")"),
                 c(4:6, "*", "/"),
                 c(1:3, "+", "-"))
bList <- list()
w <- gwindow("glayout for a calculator")
g <- ggroup(cont=w, expand=TRUE, horizontal=FALSE)
tbl <- glayout(cont=g, spacing=2)
tbl[1, 1:5, anchor=c(-1,0)] <-          # span 5 columns
  (eqnArea <- gedit("", cont=tbl))
tbl[2, 1:5, anchor=c(1,0)] <-
  (outputArea <- glabel("", cont=tbl))
for(i in 3:5) {
  for(j in 1:5) {
    val <- buttons[i-2, j]
    tbl[i,j] <- (bList[[val]] <- gbutton(val, cont=tbl))
  }
}
tbl[6,2] <- (bList[["0"]] <- gbutton("0", cont=tbl))
tbl[6,3] <- (bList[["."]] <- gbutton(".", cont=tbl))
tbl[6,4:5] <- (eqButton <- gbutton("=", cont=tbl))
outputArea <- gtext("", cont = g)
```

This code defines the handler for each button except the equals button and then assigns the handler to each button. This is done efficiently, using the generic `addHandlerChanged`. The handler simply pastes the text for each button into the equation area.

```
addButton <- function(h, ...) {
  curExpr <- svalue(eqnArea)
  newChar <- svalue(h$obj)              # the button's value
  svalue(eqnArea) <- paste(curExpr, newChar, sep="")
  svalue(outputLabel) <- ""             # clear label
}
out <- sapply(bList, function(i)
              addHandlerChanged(i, handler=addButton))
```

When the equals sign is clicked, the expression is evaluated and if there are no errors, the output is displayed in the label.

```
addHandlerClicked(eqButton, handler = function(h,...) {
  curExpr <- svalue(eqnArea)
  out <- try(capture.output(eval(parse(text=curExpr))))
  if(inherits(out,"try-error")) {
    gmessage("There is an error")
    return()
  }
  svalue(outputArea) <- out
  svalue(eqnArea) <- ""            # restart
})
```

3.3 Selection controls

A common task for a GUI control is to select a value or values from a set of numbers or a table of numbers. Many toolkits implement these widgets using a model-view-controller paradigm whereby the control is just one of possibly many views of the data store (the model). This approach isn't taken with `gWidgets`. Rather, each widget has its own data store containing the data for selection, and familar R methods are used to manipulate this underlying data store. The controls in `gWidgets` that display such data have the methods `[`, `[ <- `, `length`, `dim`, `names` and `names <- `, as appropriate.

This section discusses several different controls that do basically the same thing, but which exist primarily because they use screen space differently.

Checkbox widget

The simplest selection control is the `checkbox` widget that allows the user to set a state as `TRUE` or `FALSE`. The constructor has an argument `text` to set a label and `checked` to indicate if the widget should initially be checked. The default is `TRUE`.

The `svalue` method returns a logical indicating if the widget is in the checked state. Use `svalue <- ` to set the state. The label can be adjusted, if the underlying toolkit allows it, with the `[ <- ` method and is returned by the `[` method.

The default handler would be called on a click event, when the state toggles. If it is desired that the handler be called only in the `TRUE` state, say, one needs to check within the handler for this.

Radio button widget

A radio button group allows the user to choose one of a few items. A radio button group object is returned by `gradio`. The items to choose from are specified as a vector of values to the `items` argument. These items may be displayed horizontally or vertically (the default) as specified by the `horizontal`

argument which expects a logical. The `selected` argument specifies the initially selected item, with a default of the first.

The currently selected item is returned by `svalue` as the label text or by the index if the argument `index` is `TRUE`. The item may be set with the `svalue <-` method. Again, the item may be specified by the label or by an index, the latter when the argument `index=TRUE` is specified. The data store is the set of labels so are referenced through the `[` method, and may be set (if the underlying toolkit allows it) with the `[ <-` method. If `gWidgetstcltk` one can not change the number of radio buttons. For convenience, the `length` method returns the number of labels.

The default handler would be called on a click event.

A group of checkboxes

The checkbox group widget, produced by the `gcheckboxgroup` constructor, allows the selection of one or more of a set of items. The `items` arugment is used to specify the values. The state of whether an item is selected can be set with a logical vector of the same size as the number of items to the `checked` argument, recycling is used. The item layout can be controlled by the `horizontal` argument. The default is a vertical layout (`horizontal=FALSE`).

The state is retrieved as a character vector through the `svalue` method. The `index=TRUE` argument instructs `svalue` to return the indices instead. As a checkboxgroup is like both a checkbox and a radio button group, one can set the selected values two different ways. As with a checkbox, the selected values can be set by specifying a logical vector through the `svalue <-` method. As with radio button groups, the selected values can also be set with a character vector indicating which labels should be selected, or if `index=TRUE` is given, using a numeric index vector.

The labels are returned through the `[` method and if the underlying toolkit allows it, set through the `[ <-` method. As with `gradio`, the `length` method returns the number of items.

A combobox

A combobox is used as an alternative to a radio button group when there are too many choices to comfortably fit on the screen. Comboboxes are constructed by `gcombobox`. [2] The possible choices are specified to the argument `items`. This may be a vector of values or a data frame whose first column defines the choices. For toolkits which support icons in the combobox, if the data is specified as a data frame, the second column can be used to signify which stock icon is to be used. By design, a third column can be used to specify a tooltip, but this is not implemented in `RGtk2` and `tcltk`.

[2] This was at one time called `gdroplist` in `gWidgets`, as comboboxes appear like drop-down lists.

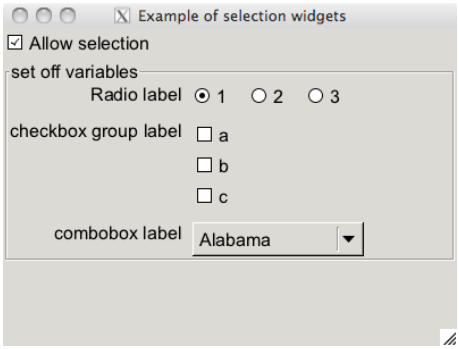

Figure 3.1: A template for a GUI using some of the widgets for selection.

The argument `editable` accepts a logical value indicating if the user can supply their own value by typing into a text entry area. The default is `FALSE`. When editing is possible, the constructor also has the `coerce.with` argument like `gedit`.

The currently selected value is returned through the `svalue` method. If `index` is `TRUE`, the index of the selected item is given if possible. The state can be set through the `svalue <-` method. This is specified by a character unless `index` is `TRUE`, in which case as a numeric index with respect to the underlying items. The `[` method returns the items of the data store, and `[ <-` is used to assign new values to the data store. The `length` method returns the number of items.

The default handler is called when the state of the widget is changed. This is also aliased to `addHandlerClicked`. When `editable` is `TRUE`, then the `addHandlerKeystroke` can be used to set a handler to response to keystroke events.

Example 3.6: Selection widgets

This example provides template for a possible GUI that would allow a specification of arguments for a function (Figure 3.1). A checkbox is used to toggle whether the other controls are enabled or not.

```
w <- gwindow("Example of selection widgets", visible=FALSE)
g <- ggroup(horizontal=FALSE, cont=w)
cb <- gcheckbox("Allow selection", cont=g, checked=FALSE,
                handler = function(h, ...) {
                  enabled(f) <- svalue(cb)
                })
f <- gframe("set off variables", cont=g)
tbl <- glayout(cont=f)
right <- c(1, 1); left <- c(-1, 1)
```

```
tbl[1,1, anchor=right] <- "Radio label"
tbl[1,2, anchor=left] <- (rb <- gradio(1:3, horizontal=TRUE,
          cont = tbl))
tbl[2,1, anchor=right] <- "checkbox group label"
tbl[2,2, achor=left] <- (chb <- gcheckboxgroup(letters[1:3],
          horizontal=FALSE, cont = tbl))
tbl[3,1, anchor=right] <- "combobox label"
tbl[3,2, achor=left] <- (combo <- gcombobox(state.name,
          cont = tbl))
enabled(f) <- FALSE
visible(w) <- TRUE
```

Display of tabular data

The `gtable` constructor produces a widget that displays data in a tabular form from which the user can select one (or more) rows. The widgets performance under `RGtk2` is much faster and able to handle larger data stores than under `tcltk`, as there is no native table widget in Tcl/Tk. Both perform well on moderate-sized data sets (10 or so columns and fewer than 500 rows),

The data is specified through the `items` argument. This may be a data frame, matrix or vector. Vectors are coerced to data frames. The data is presented in a tabular form, with column headers derived from the `names` attribute of the data frame. The `icon.FUN` argument is used to place a stock icon in a left-most column. This argument takes a function of a single argument – the data frame being shown – and should return a character vector of stock icon names, one for each row.

Filtering The arguments `filter.column` and `filter.FUN` allow one to specify whether the user can filter, or limit, the display of the values in the data store. If a column number is specified to `filter.column` then a combobox is added to the widget with values taken from the unique values in the specified column. Changing the value of the combobox, restricts the display of the data to just those rows which match that column's values. More advanced filtering can be specified using the `filter.FUN` argument. If this is a function, then it takes arguments `(data_frame, filter.by)` where the data frame is the data, and the `filter.by` value is the state of a combobox whose values are specified through the argument `filter.labels`. This function should return a logical vector with length matching the number of rows in the data frame. Only rows corresponding to `TRUE` values will be displayed. If `filter.FUN` is the character string "`manual`" then the `visible <-` method can be used to control the filtering, again by specifying a logical vector of the proper length. See Example 3.8 for an application.

Selection Users can select a row, not a cell from this widget. The value returned by a selection is controlled by the arguments `chosencol`, which specifies which column value will be returned, as the user can only specify the row; and `multiple` which controls whether the user may select more than one row. The `svalue` method will return the currently selected value. If the argument `index` is specified as `TRUE`, then the selected row index (or indices) will be returned. These refer to the data store, not the visible data when filtering is being used. The argument `drop` specifies if just the chosen column's value is returned (the default) or if specified as `FALSE` the entire row.

Methods The underlying data store is referenced by the `[` method. Indices may be used to access just a portion. Values may be set using the `[ <-` method, but be warned it is not as flexible as assigning to a data frame. The underlying toolkits may not like to change the type of data displayed in a column, so when updating a column do not assume some underlying coercion, as is done with R's data frames. To replace the data store, the `[ <-` can be used via `obj[] <- new_data_frame`. The methods `names` and `names <-` refer to the column headers, and `dim` and `length` the underlying dimensions of the data store.

Handlers Selection is done through a single click. The `addHandlerClick` method can be used to assign a handler to those events. The default handler is `addHandlerDoubleclick`, which assigns a handler for a double click event. Also of interest are the `addHandlerRightclick` and `add3rdMousePopupMenu` methods for assigning handlers to right-click events.

The `gtable` widget shows clearly the trade offs between using `gWidgets` and a native toolkit under R. As will be seen in later chapters, setting up a table to display a data frame using the `RGtk2` or `tcltk` packages can involve a fair amount of coding as compared to `gtable`, which makes it very easy. However, there is far less power possible from `gWidgets`. For example, there is no method to adjust the column sizes programatically (although they can be adjusted with the mouse),there is no means to adjust the formatting of the displayed text, or to embed other widgets into the tabular display.

Example 3.7: Simple filtering
We use the `Cars93` data set from the `MASS` package to show how to set up a display of the data which provides simple filtering based on the type of car, whose value is stored in column 3.

```
require(MASS)
w <- gwindow("gtable example")
tbl <- gtable(Cars93, chosencol=1, filter.column=3, cont=w)
```

Adding a handler for the double click event is illustrated bleow. This handler prints both the manufacturer and the model of the currently selected row when called.

```
addHandlerChanged(tbl, handler=function(h,...) {
  val <- svalue(h$obj, drop=FALSE)
  cat(paste("You selected the", val[,1], val[,2], "\n", sep=" "))
})
```

Example 3.8: More complex filtering

Even with moderate-sized data sets, the number of rows can be quite large, in which case it is inconvenient to use a GUI for selection unless some means of searching or filtering the data is used. This example uses the possible CRAN sites, to show how a `gedit` instance can be used as a search box to filter the display of data. The `addHandlerKeystroke` method is used so that the search results are updated as the user types.

The `available.packages` function returns a data frame of all available packages. If a CRAN site is not set, the user will be queried to set one.

```
d <- available.packages()       # pick a cran site
```

This basic GUI is barebones, for example it has no text labels to guide the user.

```
w <- gwindow("test of filter")
g <- ggroup(cont=w, horizontal=FALSE)
ed <- gedit("", cont=g)
tbl <- gtable(d, cont=g, filter.FUN="manual", expand=TRUE)
```

The `filter.FUN` provides a means to have a combobox control the display of the table. For this example, we desire more flexibility, so we specify the value of `"manual"`.

Different search criteria may be desired, so it makes sense to separate out this code from the GUI code using a function. The one below uses `grep` to match, so that regular expressions can be used. Another reasonable choice would be to use the first letter of the package. (That filtering could also be specified easily through the `filter.FUN` argument.)

```
ourMatch <- function(curVal, vals) {
  ind <- grep(curVal, vals)             # indices
  vis <- rep(FALSE, length(vals))
  if(legnth(ind) > 0)
    vis[ind] <- TRUE
  return(vis)                           # logical
}
```

Finally, the `addHandlerKeystroke` method calls its handler everytime a key is released while the focus is in the edit widget. In this case, the handler

finds the matching indices using the ourMatch function, converts these into logical format, and then updates the display using the visible <- method for gtable.

```
id <- addHandlerKeystroke(ed, handler=function(h, ...) {
  vals <- tbl[, 1, drop=TRUE]
  curVal <- svalue(h$obj)
  vis <- ourMatch(curVal, as.character(vals))
  visible(tbl) <- vis
})
```

An editor for tabular data

The gdf constructor returns a widget for editing data frames. This is similar to the GUI provided by the data.entry function, but uses the underlying toolkit in use by gWidgets. Each cell can be edited. Users can click (or double click) in a cell to select it, or use the arrow and tab keys to navigate. For gWidgetstcltk, there is no native widget for editing tabular data, so the tktable, add-on widget is used (tktable.sourceforge.net). A warning will be issued if this is not installed. Again, the widget under RGtk2 is much faster than that under tcltk, but both can load a moderately sized data frame in a reasonable time.

The constructor has argument items to specify the data frame to edit and name to specify the data frame name, if desired. The column types are important, in particular factors and character types are treated differently, although they may render in a similar manner.

Methods Under both RGtk2 and tcltk there are bindings for right-mouse clicks that allow the user to modify the data frame displayed. In addition, there are several methods defined that follow those of a data frame. The [and [<- methods can be used to extract and set values from the data frame by index. As with gtable, these are not as flexible as for a data frame though. In particular, it may not be possible to changes the type of a column, or add new rows or columns through these methods. Using no indices, as in obj[,] will return the current data frame, which can be assigned to some value for saving. The current data frame can be completely replaced, when no indices are specified in the replacement call. Additionally, the data frame methods dimnames, dimnames <- , names, names <- , and length are defined.

The gdfnotebook constructor produces a notebook that can hold several data frames at once.

3.4 Selection from a sequence of numbers

The previous widgets allowed selection from a user-specified set of values. When these values are a sequence of numbers, the slider control and spin button control are also commonly used. Both of these widgets have arguments to specify the sequence that match those of the `seq` function in R: `from`, `to`, and `by`.

A slider control

The `gslider` constructor creates a slider that allows the user to select a value from the specified sequence. In `gWidgetstcltk` the sequence must have integer steps. If this is not the case, the spin button control is used instead. In addition to the arguments to specify the sequence, the argument `value` is used to set the initial value of the widget and `horizontal` controls how the slider is drawn, `TRUE` for horizontal, `FALSE` for vertical.

The `svalue` method returns the currently chosen value. The `[ <-` method can be used to update the sequence of values to choose from.

The default handler is called when the slider is changed. Example 3.9 shows how this can be used to update a graphic.

A spin button control

The spin button control constructed by `gspinbutton` is similar to `gslider`, but presents the user a different way to select the value. The argument `digits` specifies how many digits are displayed.

Example 3.9: Example of sliders and spin buttons
The use of sliders and spin buttons to dynamically adjust a graphic is common in R GUIs targeted towards teaching statistics. Here is an example, similar to the `tkdensity` example of `tcltk`, where the slider controls the bandwidth of a density estimation and the spin button the sample size of a random sample.

```
w <- gwindow("Slider and Spin Button example")
tbl <- glayout(cont=w)
tbl[1,1] <- "sample size"
tbl[1,2] <- (spinner <- gspinbutton(from=10, to=100, by=5,
                                    value=25, cont=tbl))
tbl[2,1] <- "adjusted bandwidth"
tbl[2,2, expand=TRUE] <- (slider <- gslider(from=0.1, to=1,
           by=0.01, value=1, cont=tbl))
plotGraph <- function(h,...) {
  x <- rexp(svalue(spinner))
  plot(density(x, adj=svalue(slider)))
}
```

```
QT <- sapply(list(spinner, slider), function(i)
  addHandlerChanged(i, handler=plotGraph))
```

3.5 Display of heirarchical data

The `gtree` constructor can be used to display heirarchical structures, such as a file system. This constructor parameterizes the data to be displayed in terms of the node of the tree that is currently selected. The `offspring` argument is assigned a function of two variables, the path in the tree that the node in question is on and any data passed through the optional `offspring.data` argument. This function should return a data frame with each row referring to an offspring for the node and whose first column is a key that characterizes the node of the offspring, unless the argument `chosencol` is used to specify otherwise.

To indicate if a node has offspring, a function can be passed through the `hasOffspring` argument. This function takes the data frame returned by the `offspring` function and should return a logical vector with each value indicating which rows have offspring. If it is more convenient to compute this within the `offspring` function, then when `hasOffspring` is left unspecified and the second column returned by `offspring` is a logical, then that column will be used.

A single click is used to select a row. Multiple selections are possible if the `multiple` argument is given a `TRUE` value.

For some toolkits the `icon.FUN` can be used to specify a stock icon to be displayed next to the first column. This function, like `hasOffspring` has as an argument the data frame returned by `offspring` and should return a character vector with each entry indicating which stock icon is to be shown.

For some toolkits, the column type must be determined prior to rendering. By default, a call to `offspring` with argument `c()` indicating the root node is made. The returned data frame is used to determine the column types. If that is not correct, the argument `col.types` can be used. It should be a data frame with column types matching those returned by `offspring`.

methods The `svalue` method returns the currently selected key, or node label. There is no assignment method. The `[` method returns the path for the currently node. This is what is passed to the `offspring` function. The method `addHandlerDoubleclick` can be used to specify a function to call on a double click event.

Example 3.10: Using `gtree` to explore a recursive partition

The `party` package implements a recursive partitioning algorithm for tree-based regression and classification models. The package provides an

excellent `plot` method for the object, but in this example we demonstrate how the `gtree` widget can be used to display the heiarchical nature of the fitted object. First, we fit a model from an example appearing in the package's vignette.

```
require(party)
data("GlaucomaM", package="ipred")      # load data
gt <- ctree(Class ~ ., data=GlaucomaM)  # fit model
```

The `party` object tracks the heirarchical nature through its nodes. This object if a complex structure using lists to store data about the nodes. We define an `offspring` function next that tracks the node by number, as is done in the `party` object; records whether a node has offspring through the `terminal` component (by passing the `hasOffspring` function); and computes condition on the variable that creates the node. For this example, the trees are all binary trees with 0 or 2 offspring so this data frame has only 0 or 2 rows.

```
offspring <- function(key, offspring.data) {
  if(missing(key) || length(key) < 1)  # which node?
    node <- 1
  else
    node <- as.numeric(key[length(key)]) # key is a vector

  if(nodes(gt, node)[[1]]$terminal)    # return if terminal
    return(data.frame(node=node, hasOffspring=FALSE,
                      description="terminal"))

  df <- data.frame(node=integer(2), hasOffspring=logical(2),
                   description=character(2),
                   stringsAsFactors=FALSE)
  ## party internals
  children <-  c("left","right")
  ineq <- c(" <= "," >  ")
  varName <- nodes(gt, node)[[1]]$psplit$variableName
  splitPoint <- nodes(gt, node)[[1]]$psplit$splitpoint

  for(i in 1:2) {
    df[i,1] <- nodes(gt, node)[[1]][[children[i]]][[1]]
    df[i,2] <- !nodes(gt, df[i,1])[[1]]$terminal
    df[i,3] <- paste(varName, splitPoint, sep=ineq[i])
  }
  df                                    # returns a data frame
}
```

We make a simple GUI to show the widget (Figure 3.2)

```
w <- gwindow("Example of gtree")
g <- ggroup(cont=w, horizontal=FALSE)
```

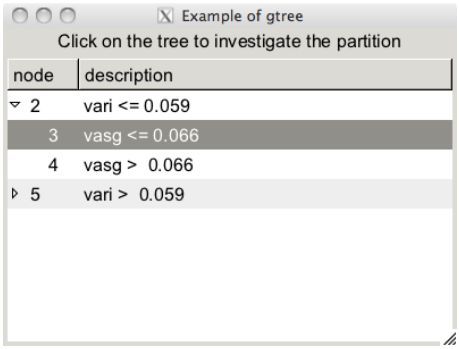

Figure 3.2: GUI to explore return value of a model fit by the party package.

```
l <- glabel("Click on the tree to investigate the partition",
            cont=g)
tr <- gtree(offspring, cont=g, expand=TRUE)
```

A single click is used to expand the tree, here we create a binding to a double click event to create a basic graphic. The vignette shows how to make more complicated – and meaningful – graphics for this model fit.

```
addHandlerDoubleclick(tr, handler=function(h,...) {
  node <- as.numeric(svalue(h$obj))
  if(nodes(gt, node)[[1]]$terminal) {   # if terminal plot
    weights <- as.logical(nodes(gt,node)[[1]]$weights)
    plot(response(gt)[weights, ])
  }})
```

3.6 Selecting from the file system

The gfile dialog allows one to select a file or directory from the file system. This is a modal dialog, which returns the name of the selected file or directory. The gfilebrowse constructor creates a widget that has a button that allows the user to initiate this selection.

The selection type is specified by the type argument with values of open, to select an existing file; save to select a file to write to; and selectdir to select a directory. For RGtk2, the filter argument can be used to narrow the listed files. The dialog returns the path of the file, or NA if the dialog was canceled. One can also specify a handler to the constructor to call on the file or directory name. The component file of the first argument to the handler contains the file name.

```
if(!is.na(tmp <- gfile()))
  source(tmp)
## or
```

```
gfile(handler=function(h,...) {
  if(!is.na(h$file))
    source(h$file)
})
```

Selecting a date

The `gcalendar` constructor returns a widget that can be used to select a date if the underlying toolkit supports such a widget or a text edit box to allow the user to enter a date. The argument `text` arugment can used to specify the initial text. The `format` is used to specify the format of the date.

The methods for the widget inherit from `gedit`. In particular, the `svalue` method returns the text in the text box as a character vector formatted by the value specified by the `format` argument. To return a value of a different class, pass a function, such as `as.Date` to the `coerce.with` argument.

3.7 Display of graphics

Displaying icons and images store in files

Graphics files can be displayed by the `gimage` widget. (Not all file types may be displayed by each toolkit, in particular `gWidgetstcltk` can only display gif, ppm, and xbm files.) The file to display is specified through the `filename` argument of the constructor. This value is combined with that of the `dirname` argument to specify the file path.

The `gWidgets` package provides a few stock icons, that simplify the above. Stock icons, can be specified by using their name for the `filename` argument and the character string `"stock"` for the `dirname` argument. A list of the defined stock icons is returned by the function `getStockIcons`. The names attribute defines the valid stock icon names. For `gWidgetsRGtk2`, the size of a stock icon can be adjusted through the `size` argument, with a value from `"menu"`, `"small_toolbar"`, `"large_toolbar"`, `"button"`, or `"dialog"`.

The `svalue <-` method can be used to change the graphics file. In this case, a full path name is specified, or the stock icon name.

More stock icon names may be added through the function `addStockIcons`. This function requires a vector of stock icon names and a vector of corresponding file paths, and is illustrated in the example.

The default handler is called on a click event.

Example 3.11: Adding and using stock icons
This example shows how to add to the available stock icons and use `gimage` to display them. It creats a table to select a color from, as an alternative to a more complicated color chooser dialog. Under `gWidgetstcltk` the image

files would need to be converted to gif format, as png format is not a natively supported image type.

We begin by defining 16 arbitrary colors.

```
someColors <- c("black", "red", "blue", "brown",
                "green", "yellow", "purple",
                paste("grey", seq(10,90,by=10), sep=""))
```

This is the function that is used to create an icon file. We use some low-level grid functions to draw the image to a png file.

```
require(grid)
iconDir <- tempdir(); iconSize <- 16;
makeColorIcon <- function(i) {
  filename <- paste(iconDir, "/color-", i, ".png",
                    sep="", collapse="")
  png(file=filename, width=iconSize, height=iconSize)
  grid.newpage()
  grid.draw(rectGrob(gp=gpar(fill=i)))
  dev.off()
  return(filename)
}
```

To add icons, we need to define the stock names and the file paths for addStockIcons.

```
icons <- sapply(someColors, makeColorIcon)
iconNames <- paste("color-", someColors, sep="")
QT <- addStockIcons(iconNames, icons)
```

We use a table layout to show the 16 colors. As an illustration of assigning a handler for a click event, we assign one that returns the corresponding stock icon name.

```
w <- gwindow("Icon example")
f <- function(h,...) print(h$action)
tbl <- glayout(cont = w, spacing=0)
for(i in 1:4) {
  for(j in 1:4) {
    ind <- (i - 1) * 4 + j
    tbl[i,j] <- gimage(icons[(i-1)*4 + j], handler=f,
                       action=iconNames[ind], cont=tbl)
  }
}
```

A graphics device

Some toolkits support an embeddable graphics device (RGtk2 through cairoDevice). In which case, the ggraphics constructor produces a widget that can be

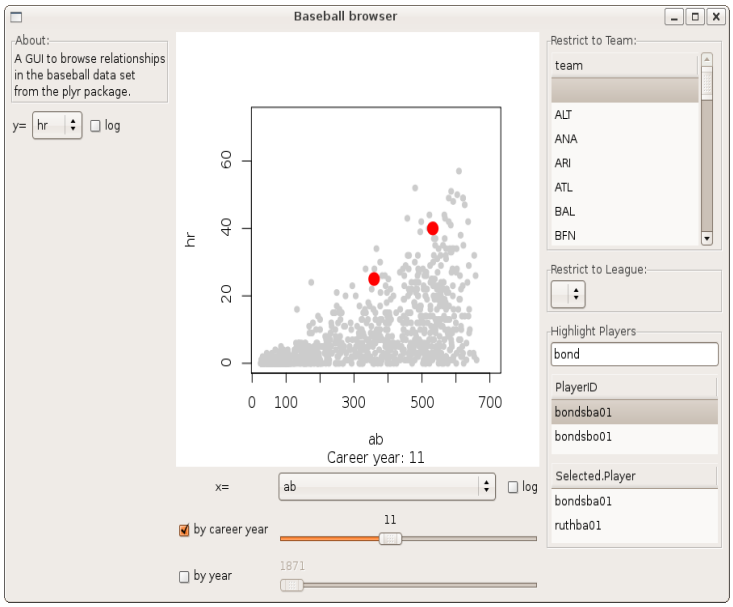

Figure 3.3: A `RGtk2` GUI for exploring the `baseball` data set of the `plyr` package. One can subset by year or career year through the slider widgets.

added to a container. The arguments `width`, `height`, `dpi`, `ps` are similar to other graphics devices.

The `ggraphicsnotebook` creates a notebook that allows the user to easily navigate multiple graphics devices.

Example 3.12: A GUI to explore a data set

```
The hash package provided by:
                                  _        _
  ___  _ __   ___ _ __   __| | __ _| |_ __ _
 / _ \| '_ \ / _ \ '_ \ / _` |/ _` | __/ _` |
| (_) | |_) |  __/ | | | (_| | (_| | || (_| |
 \___/| .__/ \___|_| |_|\__,_|\__,_|\__\__,_|
      |_|    http://www.opendatagroup.com
```

This example creates a GUI to explore the `baseball` data set of the `plyr` package. The baseball data set contains information by year for players who had 15-year or longer careers. Several interesting things can be seen by looking at specific players, such as Babe Ruth (coded `ruthba01`) or Barry Bonds (`bondsba01`). Before beginning, we follow an example from the `plyr` package to create a new variable to hold the career year of a player.

```
data(baseball, package="plyr")
```

```
calc <- function(df)
  transform(df,
            cyear = year - min(year),
            cpercent = (year - min(year))/(max(year) - min(year)))
b <- ddply(baseball, .(id), calc)
b <- subset(b, ab >= 25)
nVars <- names(b)[-c(1:5,23:24)]    # numeric variables
```

This example uses the `hash` package to store our data and an environment to store our widgets.

```
require(hash)
dat <- hash()
e <- new.env()
```

The following function transfers values from the GUI to our data store, `dat`, returning `TRUE` if all goes well. The widgets are all stored in an environmen, `e` below, using names which are again used as keys to the hash. We also define a function `plotIt` to produce a graphic based on the current state of the data store, but don't reproduce it here.

```
transferData <- function() {
  out <- try(sapply(e, svalue, drop=TRUE), silent=TRUE)
  if(inherits(out,"try-error"))
    return(FALSE)
  dat[[names(out)]] <- out              # hash keys
  dat$id <- e$id[]                      # not svalue
  return(TRUE)                          # works!
}
```

We now create a GUI so that the user can select which graphic to make. Our GUI will have a main plot window to show a scatter plot, and controls to adjust the variables that are plotted, and to filter that values plotted.

Our layout will use box containers to split the top-level window into three panes. The middle one holds the graphic, so we set it to `expand` when the window is resized.

```
w <- gwindow("Baseball browser", visible=FALSE)
g <-  ggroup(cont=w, horizontal=TRUE)
lp <- ggroup(cont=g, horizontal=FALSE)
cp <- ggroup(cont=g, horizontal=FALSE, expand=TRUE)
rp <- ggroup(cont=g, horizontal=FALSE, spacing=10)
```

The left panel holds a short description and a combobox to select the y-variable plotted.

```
f <- gframe("About:", cont=lp)
l <- glabel(paste("A GUI to browse relationships",
             "in the baseball data set",
             "from the plyr package.",
```

```
              sep="\n"),
       cont=f)
g1 <- ggroup(cont=lp)
l <- glabel("y=", cont=g1)
e$y <- gcombobox(nVars, selected=4, cont=g1)
e$ylog <- gcheckbox("log", checked=FALSE, cont=g1)
```

The center panel holds the `ggraphics` object, along wtih controls to select the x variable. As well, we add controls to filter out the display by either the year a player played and/or their career year. A `gtable` instance is used for layout.

```
gg <- ggraphics(cont=cp)
tbl <- glayout(cont=cp)
tbl[1,1] <- "x="
tbl[1,2, expand=TRUE] <- (e$x <- gcombobox(nVars, selected=2,
           cont=tbl))
tbl[1,3] <- (e$xlog <- gcheckbox("log", checked=FALSE,
                                 cont=tbl))
##
tbl[2,1] <- (e$doCareerYear <- gcheckbox("by career year",
                                         checked=TRUE, cont=tbl))
tbl[2,2:3, expand=TRUE] <- (e$cyear <-
             gslider(min(b$cyear), max(b$cyear), by=1, cont=tbl))
enabled(e$cyear) <- TRUE
##
tbl[3,1] <- (e$doYear <- gcheckbox("by year",
                                   checked=FALSE, cont=tbl))
tbl[3,2:3, expand=TRUE] <- (e$year <-
             gslider(min(b$year), max(b$year), by=1, cont=tbl))
enabled(e$year) <- FALSE
```

The right panel includes a few means to filter the display of values. We use a simple `gtable` widget to allow the user to restrict the display to one or more teams. A combobox allows the user to restrict to one of the historic leagues. To allow certain players to stand out, a compound widget is made using a `gedit` object to filter values, a `gtable` object to show all possible IDs, and a `gtable` object to show the selected IDs to highlight. Frame are used to visually combine these controls.

```
rpWidth <- 200
f <- gframe("Restrict to Team:", cont = rp)
teams <- data.frame(team=c("", sort(unique(b$team))),
                    stringsAsFactors=FALSE)
e$team <- gtable(teams, cont=f, multiple=TRUE, width=rpWidth)
size(e$team) <- c(200,200)
svalue(e$team, index=TRUE) <- 1
##
f <- gframe("Restrict to League:", cont=rp)
```

```
leagues <- names(table(b$lg))[-1]       # drop ""
e$lg <- gcombobox(c("", leagues), cont=f)
##
f <- gframe("Highlight Players", horizontal=FALSE, cont=rp)
searchPlayer <- gedit("", cont=f)
listPlayers <- gtable(data.frame("PlayerID"=unique(b$id),
                                 stringsAsFactors=FALSE),
                      filter.FUN="manual", cont=f)
e$id <- gtable(data.frame("Selected Player"=character(0),
                          stringsAsFactors=FALSE), cont=f)
```

We define several handlers to make the GUI responsive to user output. Rather than write an `updateUI` function to update the GUI at periodic intervals, we use an event-driven model. These first two handlers, simply toggle whether the user can control the display by year or career year.

```
addHandlerChanged(e$doYear, handler = function(h,...) {
  val <- ifelse(svalue(e$doYear), TRUE, FALSE)
  enabled(e$year) <- val
})
addHandlerChanged(e$doCareerYear, handler = function(h,...) {
  val <- ifelse(svalue(e$doCareerYear), TRUE, FALSE)
  enabled(e$cyear) <- val
})
```

This next handler updates the graphic when any of several widgets is changed.

```
QT <- sapply(list(e$x, e$xlog, e$y, e$ylog, e$year, e$cyear,
                  e$doYear, e$doCareerYear, e$lg), function(i)
             addHandlerChanged(i, handler=function(h, ...)
                               transferData() && plotIt()))
```

For `gtable` objects, it is more natural here to bind to a single mouse click, rather than the default double click.

```
QT <- sapply(list(e$team, e$id), function(i)
       addHandlerClicked(i, handler=function(h, ...)
                         transferData() && plotIt()))
```

These handlers are used to select the IDs to highlight.

```
addHandlerKeystroke(searchPlayer, handler=function(h, ...) {
  cur <- svalue(h$obj)
  ind <- grep(cur, unique(b$id))
  tmp <- rep(FALSE, length(unique(b$id)))
  if(length(ind) > 0) {
    tmp[ind] <- TRUE
    visible(listPlayers) <- tmp
  } else if(length(grep("^\\s$", cur))) {
    visible(listPlayers) <- !tmp
```

```
  } else {
    visible(listPlayers) <- tmp
  }
})
addHandlerChanged(listPlayers,handler=function(h, ...) {
  val <- svalue(h$obj)
  e$id[] <- sort(c(val, e$id[]))
})
addHandlerChanged(e$id, handler=function(h, ...) {
  val <- svalue(h$obj)
  cur <- e$id[]
  e$id[] <- setdiff(cur, val)
})
```

Finally, we implement functionality similar to the `locator` function for the graphic. This handler labels the point nearest to a mouse click in the plot area.

```
distance <- function(x,y)  {
  ds <- apply(y, 1, function(i) sum((x-i)^2))
  ds[is.na(ds)] <- max(ds, na.rm=TRUE)
  ds
}
addHandlerClicked(gg, function(h,...) {
  x <- c(h$x, h$y)
  ds <- distance(x, curdf[,2:3])
  ind <- which(ds == min(ds))
  ids <- curdf[ind, 1]
  points(y[ind,1], y[ind,2], cex=2, pch=16, col="blue")
  text(y[ind,1], y[ind,2], label=ids, adj=c(-.25,0))
})
```

To end, we show the GUI and initialize the plot.

```
visible(w) <- TRUE
QT <- transferData() && plotIt()
```

3.8 Dialogs

The `gWidgets` package provides a few constructors to quickly make some basic dialogs for showing messages or gathering information. Mostly these are modal dialogs that take control of the eventloop, not allowing any other part of the GUI to be active for programmatic interaction. As such, the constructors do not return an object to manipulate through its methods, but rather the value of the dialog specified by the user. Hence, they are used differently than other constructors. For example, the `gfile` dialog, previously described, is a modal dialog that pops up a means to select a file returning the selected file path or `NA`.

Table 3.3: Table of constructors for basic dialogs in gWidgets

Constructor	Description
gfile	File and directory selection dialog
gmessage	Dialog to show a message
galert	Unobtrusive (non-modal) dialog to show a message
gconfirm	Confirmation dialog
ginput	Dialog allowing user input
gbasicdialog	Flexible modal dialog

The dialogs pop up a window with a common appearance. The constructors have arguments message for a message; title for the window title; and icon to specify an icon, whose value is one of "info", "warning", "error", or "question". Buttons will appear at the bottom of the dialog, and are determined by choice of the constructor. The parent argument will place the dialog near the gWidgets instance specified. Otherwise, placement will be controlled by the window manager.

The dialogs, except for galert, have the standard handler and action arguments, for calling a handler, but typically it is easier to use the return value when programming.

A message dialog The simplest dialog is produced by gmessage, which is used to display a message. The user has a cancel button to dismiss the dialog.

An alert dialog The galert dialog is similar to gmessage only it is meant to be less obtrusive, so it is non-modal. It does not take the focus and vanishes after a time delay.

A confirmation dialog The constructor gconfirm produces a dialog that allows the user to confirm the message. This dialog returns TRUE or FALSE depending on the user's selection.

An input dialog The ginput constructor produces a dialog which allows the user to input a single line of text. If the user confirms the dialog, the value of the string is returned, otherwise if the user cancels the dialog through the button a value of NA is returned.

A basic dialog The gbasicdialog constructor allows one to place an arbirary widget within a modal window with OK and Cancel. The handler, if specified, will be called if the user clicks the OK button. This allows users to create their own modal dialogs.

As with the others, the argument `title` is used to specify the window title, but there is no `icon` or `message` arguments, as there is no standard appearance. Rather, the `widget` argument specifies a widget to pack into the dialog. This can be a simple control, or a container containing other widgets.

As with `gconfirm`, this widget returns `TRUE` or `FALSE` depending on the user's selection. To do something more complicated than `gconfirm`, a handler should be specified at construction. If the user selects OK, the handler, if specified, is called before the value `TRUE` is returned.

This dialog is called a bit awkwardly, to allow it to work when controls need a parent container specified at construction time (e.g., `tcltk`). The construction is in three stages: an initial call to `gbasicdialog` to return a container which is used as the parent container for a child component; a construction of the dialog; then a call to the `visible` method on the dialog with `set=TRUE` value (not though `visible(obj) <- TRUE`).

Example 3.13: Modal dialogs
The basic input dialog requires just the first argument.

```
ginput("Message goes here", title="example dialog")
```

Here we use the question icon for a confirmation dialog, as the message is a question.

```
ret <- gconfirm("Really delete file?", icon="question")
```

This illustrates how to use the return value.

```
ret <- ginput("Enter your name", icon="info")
if(!is.na(ret))
  cat("Hello",ret,"\n")
```

The `gbasicdialog` constructor can be used to make modal dialogs. This example will force the user to select a color before proceeding with anything else.

```
## create a parent container
dlg <- gbasicdialog("Pick a color", handler =
                    function(h,...) print(svalue(widget)))
## create the dialog using dlg as the parent container
widget <- gtable(colors(), cont = dlg)
## show the modal dialog (not visible(dlg) <- TRUE)
visible(dlg, set=TRUE)
```

3.9 `gWidgets`: Compound widgets

The `gWidgets` package provides some R specific widgets for producing GUIs. Table 3.9 lists them.

Table 3.4: Table of constructors for compound widgets in gWidgets

Constructor	Description
constructorgvarbrowser	GUI for browsing variables in the workspace
constructorghelp	GUI for a help page
constructorghelpbrowser	A help browser
gcommandline	Command line widget
constructorgformlayout	Uses list to specify layout of a GUI
constructorggenericwidget	Creates a GUI for a function based on its formal arguments or a defining list

Workspace browser

A workspace browser is constructed by gvarbrowser, providing a means to browse and select the objects in the current global environment. The quality of the implementation varies depending on the toolkit. The default handler object calls do.call on the object for the function specified through the action argument. The default is to print a summary of the object. This handler is called on a double click. A single click is used for selection. The name of the currently selected value is returned by the svalue method.

Help browser

The ghelp constructor produces a widget for showing help pages using a notebook container. This widget does not use the html help pages or the chm help pages, so it may not work for all operating systems. (For Windows, the help browser of the GUI is much better anyways.) To add a help page, the add method is used, where the value argument describes the desired page. This can be a character string containing the topic, a character string of the form package:::topic to specify the package, or a list with named components package and topic. The dispose method of notebooks can be used to remove the current tab.

The ghelpbrowser constructor produces a stand-alone GUI for displaying help pages, running examples from the help pages or opening vignettes provided by the package. This GUI provides its own top-level window and does not return a value for which methods are defined.

Command line widget

A simple command line widget is created by the gcommandline constructor. This is not meant as a replacement for R's typical commandlines, but is there for lightweight usage. A text box allows users to to type in R commands. The programmer may issue commands to be evaluated and displayed through the svalue <- method. The value assigned is a character string holding

the commands. If there is a names attribute, the results will be assigned to a variable in the global workspace with that name. The `svalue` and `[` methods return the command history.

Simplifying creation of dialogs

The `gWidgets` package has two means to simpify the creation of GUIs. The `gformlayout` constructor takes a list defining a layout and produces a GUI, the `ggenericwidget` constructor can take a function name and produce a GUI based on the formal arguments of the function. This too uses a list, that can be modified by the user before the GUI is constructed.

Laying out a form

The `gformlayout` constructor takes a list defining a layout and creates the specified widgets. The design borrows from the `extjs` javascript libraries for web programming, where a similar function can be used to specify the layout of web forms. Several toolkits have a means to specify a layout using XML (eg. glade), this implementation uses a list, assuming this is more familiar to the R user. By defining the layout ahead of time, pieces of the layout can be recycled for other layouts.

To define the layout, each component is specified using a list with named components. The component `type` specifies what component to be created, as a string. This can be a the name of a container constructor, a widget constructor or the special value `"fieldset"`. Field sets are used to group a set of common controls. If the component `name` is specified, then the component that is created will be stored in the list returned by the `[` method.

The `label` component can be specified to add a descriptive label to the layout. When specified, the component `label.pos` can be specified with value `"top"` to have the label on top of the widget, or `"side"` to place the label on the side (the default positioning). The `label.font` component can be used to specify the label's font properties using a label's `font <-` method.

If the type is a container or fieldset, then the `children` component is a list whose components specify the children as above. Except for fieldsets, these children can contain other containers or components. Fieldsets only allow components as children.

Whether a widget is enabled or not can be controlled by specifying values for `depends.on`, `depends.FUN`, and `depends.signal`. If the component `depends.on` specifies the name of a previous component, then the function `depends.FUN` will be consulted when the signal specified by `depends.signal` is emitted. This uses the `addHandlerXXX` names with a default value of `addHandlerChanged`. The `depends.FUN` function has a single argument consisting of the value returned by `svalue` when called on the widget specified

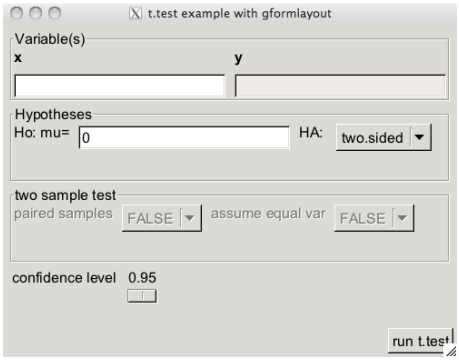

Figure 3.4: A dialog to collect arguments for a *t*-test made with `gformlayout`.

through `depends.on`. This function should return a logical indicating if the widget is enabled or not.

Methods The constructor returns an object with a few methods. The `[` method will return a list with components being the widgets that were named in the defining list. The `svalue` method simply applies the `svalue` method for each component of the list returned by the `[` method. The `names` method returns the names of the widgets in the list.

Example 3.14: The `gformlayout` constructor
This example uses `gformlayout` to make a GUI for a *t*-test (Figure 3.4). The first task is to define the list that will set up the GUI. We do this in pieces. This first piece will define the part of the GUI where the null and alternative hypotheses are specified. The null is specified as a numeric value with a default of 0. We use the `gedit` widget which by default will return a character value, so the `coerce.with` argument is specified. For the alternative, this requires a selection for just 3 possibilities, so a combo box is employed.

```
hypotheses <-
  list(type = "fieldset",
       label = "Hypotheses",
       columns = 2,
       children = list(
         list(type="gedit",
              name="mu", label="Ho: mu=",
              text="0", coerce.with=as.numeric),

         list(type="gcombobox",
              name="alternative", label="HA: ",
              items=c("two.sided","less","greater")
```

```
              )))
```

Basic usage of the `t.test` function allows for an `x`, or `x` and `y` variable to be specified. Here we disable the `y` variable until the `x` one has been entered. The `addHandlerChanged` method is called when the enter key is pressed after the `x` value is specified.

```
variables <-
  list(type="fieldset",
       columns = 2,
       label = "Variable(s)",
       label.pos = "top",
       label.font = c(weight="bold"),
       children = list(
         list(type = "gedit",
              name = "x", label = "x",
              text = ""),
         list(type = "gedit",
              name = "y", label = "y",
              text = "",
              depends.on = "x",
              depends.FUN = function(value) nchar(value) > 0,
              depends.signal = "addHandlerChanged"
              )))
```

If a `y` value is specified, then the two-sample options make sense. This enables them dependent on that happening.

```
two.sample <-
  list(type = "fieldset",
       label = "two sample test",
       columns = 2,
       depends.on = "y",
       depends.FUN = function(value) nchar(value) > 0,
       depends.signal = "addHandlerChanged",
       children = list(
         list(type = "gcombobox",
              name = "paired", label = "paired samples",
              items = c(FALSE, TRUE)
              ),
         list(type = "gcombobox",
              name = "var.equal", label = "assume equal var",
              items = c(FALSE, TRUE)
              )))
```

The confidence interval specification is specified using a slider for variety.

```
conf.level <-
  list(type = "fieldset",
       columns = 1,
```

```
        children = list(
          list(type = "gslider",
               name = "conf.level", label = "confidence level",
               from=0.5, to=1.0, by=.01, value=0.95
               )))
```

Finally, the constituent pieces are placed inside a box container.

```
tTest <- list(type = "ggroup",
              horizontal = FALSE,
              children = list(
                variables,
                hypotheses,
                two.sample,
                conf.level
                ))
```

The layout of the GUI is primarily done by the `gformlayout` call. The following just places the values in a top-level window and adds a button to initiate the call to `t.test`.

```
w <- gwindow("t.test example with gformlayout")
g <- ggroup(horizontal=FALSE, cont=w)
fl <- gformlayout(tTest, cont=g, expand=TRUE)
```

```
DEBUG$weight
weight
"bold"

[1] " bold"
DEBUG$weight
weight
"bold"

[1] " bold"
```

```
bg <- ggroup(cont=g)
addSpring(bg)
b <- gbutton("Run t.test", cont=bg)
```

The handler is very simple, as the names chosen match the argument names of `t.test`, so the list returned by the `svalue` method can be used with `do.call`. The only needed adjustment is for the one-sample case.

```
addHandlerChanged(b, function(h, ...) {
  out <- svalue(fl)
  out$x <- svalue(out$x) # turns text string into numbers
  if(out$y == "") {
    out$y <- out$paired <- NULL
  } else {
```

```
    out$y <- svalue(out$y)
   }
   print(do.call("t.test", out))
 })
```

Automatically creating a GUI

The `ggenericwidget` constructor can create a basic GUI for a function using the function's formal arguments as a guide for the proper widget to use to collect values for an argument of the function. The `fgui` package provides a similar function using just the `tcltk` package, only it improves `ggenericwidget` by parsing the function's help page.

The implementation actually has two stages, the first creates a list specifying the layout of the GUI and the second a call to layout the GUI. This list is different from that used by `gformlayout`. It does not provide as much flexibility and is described in the help page for `ggenericwidget`. This list can be edited if desired and then used directly.

The formal arguments of an S3 method may be different from those of its generic. For instance, those for the `t.test` generic are much different (and less useful for this purpose) than the `t.test.default` method for numeric values for `x`. Knowing this, a useful GUI can be quickly created for the `t.test` with the commands:

```
w <- gwindow("t.test through ggenericwidget")
f <- stats:::t.test.default;
widget <- ggenericwidget("f", cont=w)
```

```
DEBUG$style
[1] "bold"

[1] ""
DEBUG$style
[1] "bold"

$size
[1] 10

[1] " 10"
DEBUG$style
[1] "bold"

[1] ""
```