1

gWidgets: Overview

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

The gWidgets package started as a port to RGtk2 of the iWidgets package of Simon Urbanek, initially implemented only for Swing through rJava (**?**). Along the way, gWidgets was extended and abstracted to work with different GUI toolkit backends available for R. A separate package provides the interface. As of writing there are interfaces for RGtk2, qtbase, and tcltk. The gWidgetsWWW provides a similar interface for web programming, but does not use gWidgets itself.

Figure 1.1 demonstrates the portability of gWidgets commands, as it shows realizations on different operating systems and with different graphical toolkits.

1.1 Constructors

We jump right in with an example leaving comments about installation to the end of the chapter. The following shows some sample gWidgets commands that set up a basic interface allowing a user to input some text. The first line loads the package, the others will be described later.

```
require(gWidgets)
options(guiToolkit="RGtk2")
w <- gwindow("Text input example", visible=FALSE)
g <- ggroup(container=w)
l <- glabel("Your name:", cont=g)
e <- gedit("", cont=g)
b <- gbutton("Click", cont=g, handler=function(h,...) {
  msg <- sprintf("Hello %s", svalue(e))
  cat(msg, "\n")
})
```

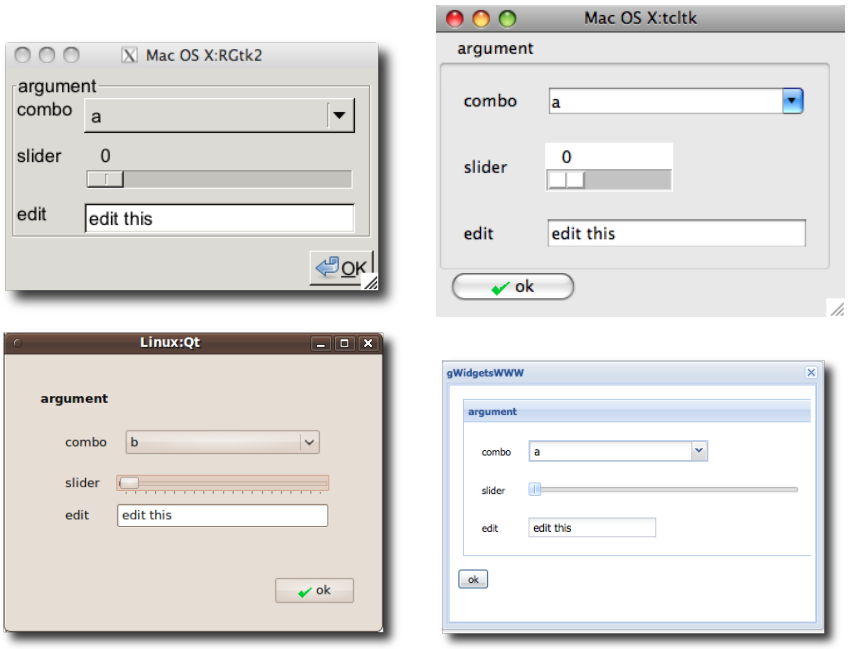

Figure 1.1: The `gWidgets` package works with different operating systems and different GUI toolkits. This shows, the same code using the `RGtk2`, `tcltk`, `qtbase` packages for a toolkit. Additionally, the `gWidgetsWWW` package is used in the lower right figure.

```
#
visible(w) <- TRUE
```

This example defines five different widgets, a window, a box container, a label, a single-line, text edit area and a button. These GUI objects are produced by constructors.

In the `gWidgets` API most constructors have the following form:

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

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

In the above, we see that the `gwindow` constructor, for a top-level window, has two arguments passed in, an unnamed one for a window title and a value for the `visible` property. Whereas the `ggroup` constructor takes all the default arguments except for the parent container.

A top-level window does not have a parent container, but other GUI components do. In `gWidgets`, for the sake of portability, the parent container is passed to the widget constructor through the `container` argument, as it done in all the other constructors. This argument name can always

be abbreviated `cont`. This defines the GUI layout, a topic taken up in Chapter 2.

The `toolkit` argument is usually not specified. It is there to allow the user to mix toolkits within the same R session, but in practice can cause problems due to competing event loops. The default for the `toolkit` argument is a call to `guiToolkit`. In our example we have the call

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

to explicitly set the toolkit. In that case, the call will return this choice. If there is more than one possibility, and one has not be selected, then the `guiToolkit` will query the user to choose one.

The constructors produce three general types of objects: containers, such as the top level window `w` and the box container `g` (Table 2.1); components, such as a label `l`, the edit area `e` or the button `b`. (Tables 3 and 4) show the basic and compound widgets); and dialogs.

1.2 Methods

In addition to creating a GUI object, except for modal dialog constructors a constructor also returns a useful R object. This is an S4 object of a certain class containing two components: `toolkit` and `widget`. (Modal dialogs do not return an object, as the dialog will be destroyed before the constructor returns. Instead, their constructors return values reflecting the user response to the dialog.)

GUI objects have a state determined by one or more of their properties. In `gWidgets` many properties are set at the time of construction. However, there are also several generic methods defined for `gWidgets` objects. [1]

Depending on the class of the object, the `gWidgets` package provides methods for the familiar S3 generics `[`, `[<-`, `dim`, `length`, `names`, `names<-`, `dimnames`, `dimnames<-` and `update`.

In our example, we see two cases of the use of generics defined by `gWidgets`. The call

```
svalue(e)
```

demonstrates the generic method *`svalue`* that can be used to get or set the main property of the widget. For the object `e`, the main property is the text, for the button and label widgets this property is the label. The `svalue<-` assignment method is used to adjust this property programatically. For the selection widgets, there is a natural mapping between vectors or data frames, and the data to be selected. In this case, the user may want

[1] We are a bit imprecise about the term "method" here. The `gWidgets` methods call further methods in the underlying toolkit interface which we think of a single method call. The actual S4 object has a slot for the toolkit and the widget created by the toolkit interface to dispatch on.

the value selected or the index of the selected value. The `index=TRUE` argument may be given to refer to the index, either when getting or setting the properties value. For these selection widgets the familiar `[` and `[<-` methods refer to the underlying data to be selected from.

The call,

```
visible(w) <- TRUE
```

sets the visibility property of the top-level window. In this example, the `gwindow` constructor is passed `visible=FALSE` to suppress an initial drawing, making this method call necessary to show the GUI.

Some other methods to adjust the widget's underlying properties are `font<-`, to adjust the font of an object; `size` and `size<-` to query and set the size of a widget; and `enabled<-`, to adjust if a widget is sensitive to user input.

The `gWidgets` API provides just a handful of generic functions for manipulating an object's properties 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.

1.3 Callbacks

In our example, the lines

```
b <- gbutton("Click", cont=g, handler=function(h,...) {
  msg <- sprintf("Hello %s", svalue(e))
  cat(msg, "\n")
})
```

create the button object. The constructor's argument `handler` is used to bind a callback to the click event of the button. Callbacks in `gWidgets` have a common signature `(h,...)` where `h` is a list with components `obj`, to pass in the object of the event (the button in this case), and `action` to pass along any value specified by the `action` argument (allowing one to parameterize the callback).

For example, a typical idiom within a callback is

```
prop <- svalue(h$obj)
```

which assigns the object's main property to `prop`. Some toolkits pass additional arguments through the callback's `...` argument, so for portability this part of the signature is not optional. For some classes, extra information is passed along. For instance, in the drop target callback the component `h$dropdata` holds the drag-and-drop value.

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

Method	Description
svalue, svalue<-	Get or set widget's main property
size<-	Set preferred size request of widget in pixels
show	Show widget if not visible
dispose	Destroy widget or its parent
enabled, enabled<-	Adjust sensitivity to user input
visible, visible<-	Show or hide object or part of object.
focus<-	Sets focus to widget
insert	Insert text into a multi-line text widget
font<-	Set a widget's font
update	Update widget values
isExtant	Does R object refer to GUI object that still exists
[, [<-	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
tag, tag<-	Sets an attribute for a widget that persists through copies
getToolkitWidget	Returns underlying toolkit widget for low-level use

While one can specify a callback to the constructor, it is often a better practice to keep separate the construction of an object and the definition of its callback. The package provides a number of methods to add callbacks for different events. The main method is addHandlerChanged, which is used to assign a callback for the typical event for the widget, such as the clicking of a button. (The handler argument, when specified, uses this method call.) In addition, there are many "addHandlerXXX" methods to assign callbacks to other events, where the XXX describes the event. These are useful in the case where more than one event is of interest. For example, for single line text widgets, like e in our example, the addHandlerChanged method sets a callback to respond when the user finishes editing, whereas a handler set by addHandlerKeystroke is called each time a key is pressed. Table 1.3 shows a list of the these other methods.

As an example, we could have specified the button as

```
b <- gbutton("Click", cont=g)
ID <- addHandlerClicked(b, handler=function(h, ...) {
```

```
  msg <- sprintf("Hello %s", svalue(h$action))
  cat(msg, "\n")
}, action=e)
```

We passed in the object `e` through the action argument as an illustration. This is useful, as one need not worry about the scope of the call to `svalue`.

The `addHandlerXXX` methods return 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.

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.

1.4 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 event loop, not allowing any other part of the GUI to be active for programmatic interaction. As such, in `gWidgets` constructors of modal dialogs do not return an object to manipulate through its methods, but rather return the user response to the dialog. For example, the `gfile` dialog, described later, is a modal dialog that pops up a means to select a file returning the selected file path or `NA`. It is used along the lines of:

```
if(!is.na(f <- gfile())) source(f)
```

Here we describe the dialogs that can be used to display a message or gather a simple amount of test. The `gfile` dialog is described in Section 3.2 and the `gbasicdialog`, which is implemented like a container, is described in Section 2.1.

The information dialogs are simple one-liners. For example, this command will cause a confirmation dialog to popup allowing the user to select a value which will be returned as `TRUE` or `FALSE`:

```
gconfirm("Yes or no? Click one.")
```

The information dialogs 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 is used to position the dialog near the `gWidgets` instance specified. Otherwise, placement will be controlled by the window manager.

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

Method	Description
`addHandlerChanged`	Primary handler call for when a widget's value is "changed." The interpretation of "change" depends on the widget.
`addHandlerClicked`	Sets handler for when widget is clicked with (left) mouse button. May return position of click through components `x` and `y` of the `h`-list.
`addHandlerDoubleclick`	Sets handler for when widget is double clicked
`addHandlerRightclick`	Sets handler for when widget is right clicked
`addHandlerKeystroke`	Sets handler for when key is pressed. The `key` component is set to this value if possible.
`addHandlerFocus`	Sets handler for when widget gets focus
`addHandlerBlur`	Sets handler for when widget loses focus
`addHandlerExpose`	Sets handler for when widget is first drawn
`addHandlerUnrealize`	Sets handler for when widget is undrawn on screen
`addHandlerDestroy`	Sets handler for when widget is destroyed
`addHandlerMouseMotion`	Sets handler for when widget has mouse go over it
`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`.
`addHandler`	(Not cross-toolkit) Allows one to specify an underlying signal from the graphical toolkit and handler
`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

Table 1.3: Table of constructors for basic dialogs in gWidgets

Constructor	Description
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
gfile	File and directory selection dialog

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 displays a message. The user has a cancel button to dismiss the dialog.

For example,

```
gmessage("Message goes here", title="example 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.

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

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

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.

This illustrates how to use the return value.

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

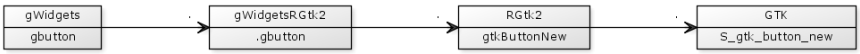

Figure 1.2: The construction of a button in `gWidgets`

Table 1.4: Installation notes for GUI toolkits.

	Gtk+	Qt	Tk
Windows	Download exe file	Install libraries, binary	In binary install of R
Linux	Standard	Standard	Standard
OS X	Download binary .pkg	Install from vendor	In binary install of R

1.5 Installation

The `gWidgets` package interfaces with an underlying R package through an intermediate package. For example, Figure 1.2 shows the sequence of calls to produce a button. First the `gWidgets` package dispatches to a toolkit package (`gWidgetsRGtk2`), which in turn calls functions in the underlying R package (`RGtk2`) which in turn calls into the graphical toolkit. As such, to use `gWidgets` with the GTK+ toolkit one must have installed on their computer the GTK libraries, the `RGtk2` package and the two `gWidgets` packages.

The difficulty for the end user is the installation of the graphic toolkit, as all other packages are installed through CRAN, or are recommended packages with an R installation (`tcltk`). Table **??** describes the installation process for different operating systems and toolkits.

Not all features of the API are supported by a particular toolkit. In particular, the easiest to use (`gWidgetstcltk`) might have the fewest features. The help pages in the `gWidgets` package describe the API, with the help pages in the toolkit packages indicating differences or omissions from the API (e.g. `?gWidgetsRGtk2-package`). For the most part, omissions are gracefully handled by simply providing less functionality.

2

gWidgets: Container Widgets

After identifying the underlying data to manipulate and how to represent it, GUI construction involves three basic steps: creation and configuration of the main components; the layout of these components; and linking the components to make a GUI interactive. This chapter discusses the layout process within `gWidgets`. Layout in `gWidgets` is done by placing child components within parent containers which in turn may be nested in other containers. (This is more like GTK+, and not Qtwhere layout managers control where the components are displayed.) The `gWidgets` package provides a just few types containers: top-level windows, box containers, a grid container, a paned container and a notebook container. Figure 2.1 shows most all of these employed to produce a GUI to select and then show contents of a file.

Except for the grid container, the primary method for containers is their `add` method. The basic call is of the form `add(parent, child, extra_arguments)`. However, this isn't typically used. In some toolkits, notably `tcltk`, the widget constructors require the specification of a parent window for the widget. To accomodate that, the `gWidgets` constructors – except for top-level windows – have the argument `container` to specify the immediate parent container. Within the constructor is the call `add(container, child,...)` where the constructor creates the child and `...` values are passed from the constructor down to the `add` method. That is, the widget construction and layout are coupled together. Although, this isn't necessary when utilizing `RGtk2` or `qtbase` – and the two aspects can be separated – for the sake of cross-toolkit portability we don't illustrate this here.

2.1 Top-level windows

The `gwindow` constructor creates top-level windows. The main window property is the title which is typically displayed at the top of the window. This can be set during construction via the `title` argument or accessed

later through the svalue<- method. A basic window then is constructed as follows:

```
w <- gwindow("Our title", visible=TRUE)
```

We can then use this as a parent container for a constructor. For example;

```
l <- glabel("A child label", container=w)
```

However, top-level windows only allow one child component. Typically, its child is a container allowing multiple children such as a box container.

The optional visible argument, used above with its default value TRUE [1], controls whether the window is initially drawn. If not drawn, the visible<- method, taking a logical value, can be used to draw the window later. 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. As well, this allows the underlying toolkit to compute the necessary size before it is displayed.

Size and placement In GUI programming, a window geometry is a specification of position and size, often abbreviated $w \times h + x + y$. The width and height can be specified at construction 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 initial placement of a window, $x + y$, will be decided by the window manager, unless the parent argument is specified. If this is done with a vector of x and y pixel values, the upper left corner will be placed there. The parent argument can also be another gwindow instance. In this case, the new window will be positioned over the specified window and be transient for the window. That is, it will be disposed when the parent window is. This is useful, say, when a main window opens a dialog window to gather values.

For example this call makes a child window of w with a square size of 200 pixels.

```
childw <- gwindow("A child window", parent=w, size=c(200,200))
```

Handlers Windows objects can be closed programmatically through their 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 cannot prevent that from

[1] If the option gWidgets:gwindow-default-visible-is-false is non NULL, then the default will be FALSE.

Table 2.1: Constructors for container objects

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

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. For example:

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

In most GUIs, the use of menubars, toolbars and statusbars is often reserved for the main window, while dialogs are not decorated so. In `gWidgets` it is suggested, although not strictly enforced unless done so by the underlying toolkit, that these be added only to a top-level window. We discuss these later in Section 3.5.

A modal window

The `gbasicdialog` constructor allows one to place an arbitrary widget within a modal window. It also adds OK and Cancel buttons. The argument `title` is used to specify the window title.

As with the `gconfirm` dialog, 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 which is called just before the dialog is disposed.

This dialog is used in a slightly different manner, requiring the use of a call to `visible` (not `visible<-`). There are three basic steps: an initial call to `gbasicdialog` to return a container to be 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`).

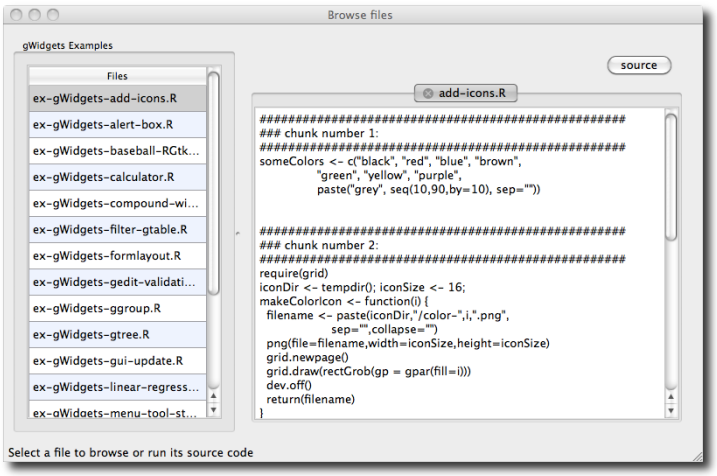

Figure 2.1: The example browser for gWidgets showing different layout components.

For a basic example where the handler simply echoes back the text stored in the label, we have:

```
w <- gbasicdialog("A modal dialog", handler=function(h,...) {
  print(svalue(l))
})
l <- glabel("A simple label", cont=w)
visible(w, set=TRUE)                    # not visible(w) <- TRUE
```

2.2 Box containers

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

The `ggroup` container

The basic box container is produced by `ggroup`. Its main argument is `horizontal` to specify whether the child widgets are packed in horizontally from left to right (the default) or vertically from top to bottom.

For example, to pack a `cancel` and `ok` button into a box container we might have:

```
w <- gwindow("Some buttons", visible=FALSE)
g <- ggroup(horizontal=TRUE, cont=w)
```

Table 2.2: Container methods

Method<	Description
`add`	Adds a child object to a parent container. Called when a parent container is specified to the `container` argument of the widget constructor, in which case, the `...` arguments are passed to this method.
`delete`	Remove a child object from a parent container
`dispose`	Destroy container and children
`enabled<-`	Set sensitivity of child components
`visible<-`	Hide or show child components

```
cancel <- gbutton("cancel", cont=g)
ok <- gbutton("ok", cont=g)
visible(w) <- TRUE
```

The add method When packing in child widgets, the `add` method is used. In our example above, this is called by the `gbutton` constructor when the `container` argument is specified. 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.

The `add` method for box containers has a few arguments to customize how the child widgets respond when their parent window is resized. These are passed through the `...` argument of the constructor.

These arguments are:

expand, fill When `expand=TRUE` is specified then the child widget will expand to fill the space allocated to it. The `fill` argument can be specified as `"x"`, `"y"`, or the default `"both"` to indicate which direction to fill in. (This varies among the toolkits.)

anchor If a widget does not expand or if it does but does not fill in both directions, it can be anchored into its available space in more than one position. The `anchor` argument can be specified to suggest where to anchor the child. It takes a numeric vector representing Cartesian coordinates (length two), with either value being `-1`, `0`, or `1`. For example, a value of `c(1,1)` would specify the northwest corner.

Figure 2.2 shows combinations of these arguments under `gWidgetsQt`.

delete The `delete` method can be used to remove a child component from a container. In some toolkits, this child may be added back at a later time, but this isn't part of the API. In the case where you wish to hide a child temporarily, the `visible<-` method may be used.

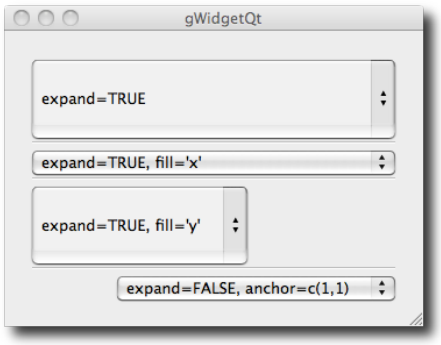

Figure 2.2: Different combinations of `expand`, `fill` and `anchor` for combo boxes in `gWidgetsQt`. The `fill` and `anchor` arguments may be overridden by the underlying toolkit for some widgets.

Spacing For spacing between the child components, the constructor's argument `spacing` may be used to specify, in pixels, the amount of space between the child widgets. For `ggroup` instances, this can later be set through the `svalue` method. The method `addSpace` can add a non-uniform amount of 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.

For example, we might modify our button layout example to include a "help" button on the far left and the others on the right with a fixed amount of space between them as follows (Figure 2.3):

```
w <- gwindow("Some buttons", visible=FALSE)
g <- ggroup(horizontal=TRUE, spacing=6, cont=w)
help <- gbutton("help", cont=g)
addSpring(g)
cancel <- gbutton("cancel", cont=g)
addSpace(g, 12)                          # 6 + 12 + 6 pixels
ok <- gbutton("ok", cont=g)
visible(w) <- TRUE
```

To illustrate how the right panel of Figure 2.1 was done, we used nested layouts as follows:

```
g <- ggroup(horizontal=FALSE, cont=w)
bg <- ggroup(cont=g)                     # nested group
addSpring(bg)
b <- gbutton("Source", cont=bg)
nb <- gnotebook(cont=g, expand=TRUE)     # fill space
```

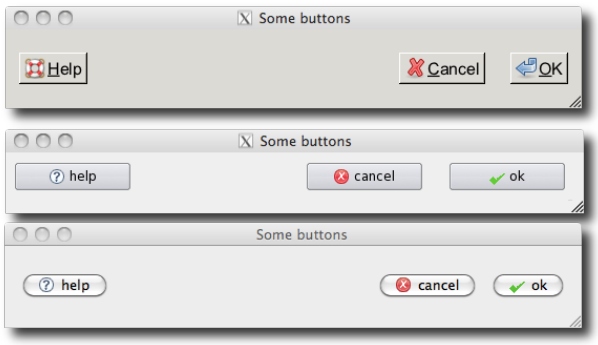

Figure 2.3: Button layout for `RGtk2` (top), `tcltk` (middle) and `qtbase` (bottom). Although the same code is used for each, the different styling yields varying sizes.

Sizing The overall size of a `ggroup` container is typically decided by how it is added to its parent. However, a requested size can be assigned through the `size<-` method.

For some toolkits the argument `use.scrollwindow`, when specified as `TRUE`, will add scrollbars to the box container so that a fixed size can be maintained. Setting a requested size in this case is a good idea. (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, this can be useful for displaying lists of data.)

The `gframe` and `gexpandgroup` containers

We discuss briefly two widgets that essentially subclass the `ggroup` class. Much of the previous discussion applies.

Framed containers are used to set off its child elements using a border and label. The `gframe` constructor produces them. In Figure 2.1 the table to select the file is from is nested in a frame to give the user some indication as to what to do.

For `gframe` the first argument, `text`, is used to specify the label. This can later be adjusted through the `names<-` method. The argument `pos` can be specified to adjust the label's positioning with 0 being the left and 1 the right.

The basic framed container is used along these lines:

```
w <- gwindow("gframe example")
f <- gframe("gWidgets Examples:", cont=w)
files <- list.files(system.file("Examples","ch-gWidgets",
                                package="ProgGUIInR"))
vars <- gtable(files, cont=f, expand=TRUE)
```

Expandable containers are useful when their child items need not be visible all the time. The typical design involves a trigger indicator with accompanying label indicating to the user that a click can disclose or hide some additional information. This class subclasses `gframe` where the `visible<-` method is overridden to initiate the hiding or showing of its child area, not the entire container.

In addition, a handler can be added that is called whenever the widget toggles its state.

Here we show how one might leave optional the display of a statistical summary of a model.

```
res <- lm(mpg ~ wt, mtcars)
out <- capture.output(summary(res))
w <- gwindow("gexpandgroup example", visible=FALSE)
eg <- gexpandgroup("Summary", cont=w)
l <- glabel(out, cont=eg)
visible(eg) <- TRUE                    # display summary
visible(w) <- TRUE
```

(How each toolkit resizes when the widget collapse varies.)

Separators Although not a container, the `gseparator` widget can be used to place a horizontal or vertical line (with the `horizontal=FALSE` argument) in a layout to separate off parts of the GUI.

2.3 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.

To see its use, we can layout a simple form for collecting information as follows:

```
w <- gwindow("glayout example", visible=FALSE)
tbl <- glayout(cont=w, spacing=5)
right <- c(1,0); left <- c(-1,0)
tbl[1,1, anchor=right] <- "name"
tbl[1,2, anchor=left ] <- gedit("", cont=tbl)
tbl[2,1, anchor=right] <- "rank"
tbl[2,2, anchor=left ] <- gedit("", cont=tbl)
tbl[3,1, anchor=right] <- "serial number"
tbl[3,2, anchor=left ] <- gedit("", cont=tbl)
visible(w) <- TRUE
```

The constructor has a few arguments to configure the appearance of the container. The spacing between each cell may be specified through the `spacing` argument, the default is 10 pixels. A value of 5 is used above

to tighten up the display. To impose a uniform cell size, the homogeneous argument can be specified with a value of TRUE. The default is FALSE.

As seen, children may be added to the grid at a specific row and column. To specify this, R's matrix notation, [<-, is used with the indices indicating the row and column. A child may span more than one row or column. The corresponding index should be a vector of indices indicating so.

The [method may be used to return the child occupying position i,j. To return the values of the widgets above can be done through:

```
vals <- sapply(seq_len(dim(tbl)[1]), function(i) {
  svalue(tbl[i,2])
  })
```

When adding a child the glayout container should be specified as the widget's container and also be on the left hand side of the [<- call. (This is necessary only for the toolkits where a container must be specified, where the right hand side is used to pass along the parent information and the left hand side is used for the layout.) 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 above illustrates how this can be used to achieve a center balance.

2.4 Paned containers: the gpanedgroup container

The gpanedgroup constructor produces a container which has two children separated by a visual gutter that can be adjusted by the user with their mouse to allocate the space among them. Figure 2.1 uses such a container to separate the file selection controls from the file display ones. For this container, the children are aligned side-by-side (by default) or top to bottom if the horizontal argument is given as FALSE.

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.)

The main property of this container is the sash position, a value in $[0,1]$. This may be configured programmatically through the svalue<- method, where a value from 0 to 1 specifies the proportion of space allocated to the leftmost (topmost) child. This specification only works after the containing window is drawn, as the percentage is based on the size of the window.

A simplified version of the layout in Figure 2.1 would be

```
d <- system.file("Examples/ch-gWidgets", package="ProgGUIinR")
files <- list.files(d)
#
```

```
w <- gwindow("gpanedgroup example", visible=FALSE)
pg <- gpanedgroup(cont = w)
tbl <- gtable(files, cont=pg)           # left side
t <- gtext("", cont=pg, expand=TRUE)    # right side
visible(w) <- TRUE
svalue(pg) <- 0.33                      # after drawing
```

2.5 Tabbed notebooks: the `gnotebook` container

The `gnotebook` constructor produces a tabbed notebook container. The GUI in Figure 2.1 uses a notebook to hold different text widgets, one for each file being displayed.

The `gWidgets` constructor has a few arguments. If permitted, the argument `tab.pos` is used to specify the location of the tabs using 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 (similar numbering as used in `par`). 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. (As of writing, this is not implemented in `gWidgetstcltk`.)

Methods Pages are added through the `add` method for the notebook container. The extra `label` argument is used to specify the tab label. (As `add` is called implicitly when a a widget is constructed, this argument is usually specified to the constructor.)

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 will want to start from the end as otherwise the tab positions change during removal.)

From some viewpoint, the notebook widget is viewed as a vector with a names attribute (the labels) and components being the child components. As such, 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. The `[` method is also implemented to return the child components by index.

Example 2.1: Tabbed notebook example
In the GUI of Figure 2.1 a notebook is used to hold differing pages. The following is the basic setup used.

```
w <- gwindow("gnotebook example")
nb <- gnotebook(cont=w)
```

New pages are added as follows:

```
fname <- "DESCRIPTION"                  # or something else
f <- system.file(fname, package="gWidgets")
gtext(readLines(f), cont = nb, label=fname)
```

To manipulate the displayed pages, say to set the page to the last one we have:

```
svalue(nb) <- length(nb)
```

To remove the current page

```
dispose(nb)
```

3

gWidgets: Control Widgets

This Chapter discusses the basic GUI controls provided by `gWidgets`. In the following one, we discuss some R-specific widgets.

Buttons

The button widget allows a user to initiate an action through clicking on it. Buttons have labels, conventionally verbs indicating action, and often icons. The `gbutton` constructor has an argument `text` to specify the text. For text that matches the stock icons of `gWidgets` (Section 3) an icon will appear. (The `ok` button below, but not the custom `par...` one.)

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 in Section **??**). The default handler is the click handler which can be specified at construction, or afterward through `addHandlerClicked`. The underlying toolkit's method of invoking a callback through keyboard navigation is used.

The following example shows how a button can be used to call a sub dialog to collect optional information. We imagine this as part of a dialog to generate a plot.

```
w <- gwindow("Make a plot")
g <- ggroup(horizontal=FALSE, cont=w)
glabel("... Fill me in ...", cont=g)
bg <- ggroup(cont=g)
addSpring(bg)
parButton <- gbutton("par (mfrow) ...", cont=bg)
```

Our callback opens a subwindow to collect a few values for the `mfrow` option.

```
addHandlerClicked(parButton, handler=function(h,...) {
  w1 <- gwindow("Set par values for mfrow", parent=w)
  lyt <- glayout(cont=w1)
```

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

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

```
  lyt[1,1, align=c(-1,0)] <- "mfrow: c(nr,nc)"
  lyt[2,1] <- (nr <- gedit(1, cont=lyt))
  lyt[2,2] <- (nc <- gedit(1, cont=lyt))
  lyt[3,2] <- gbutton("ok", cont=lyt, handler=
                function(h,...) {
                  x <- as.numeric(c(svalue(nr), svalue(nc)))
                  par(mfrow=x)
                  dispose(w1)
                })
})
```

The button's label is its main property and can be queried or set with svalue or svalue<-. 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 grayed-out state. As with other components, the `enabled<-` method can set or disable whether a widget can accept input.

Labels

The `glabel` constructor produces a basic label widget. We've already seen its use in a number of examples. The main property, 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 label text as a single string, whereas the `svalue<-` method is available to set the text programmatically.

The `font<-` method can also be used to set the text markup (Table 3.1). For some of the underlying toolkits, setting the argument `markup TRUE` allows a native markup language to be used (GTK+ had PANGO, Qt has rich text).

To make a form's labels have some emphasis we could do:

```
w <- gwindow("label example")
f <- gframe("Summary statistics:", cont=w)
lyt <- glayout(cont=f)
lyt[1,1] <- glabel("xbar:", cont=lyt)
lyt[1,2] <- gedit("", cont=lyt)
lyt[2,1] <- glabel("s:", cont=lyt)
lyt[2,2] <- gedit("", cont=lyt)
sapply(1:2, function(i) {
  tmp <- lyt[i,1]
  font(tmp) <- c(weight="bold", color="blue")
})
```

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

HTML text

Not all toolkits have the native ability, but for those that do (Qt) the `ghtml` constructor allows HTML-formatted text to be displayed, in a manner similar to `glabel`. There is no API clicking of links, etc.

Statusbars

Statusbars are simply labels placed at the bottom of a top-level window to leave informative, but non-disruptive, messages for the user. The `gstatusbar` constructor provides this widget. The `container` argument should be a

top-level window instance. The only property is the label's text. This may be specified at construction with the argument `text`. Subsequent changes are made through the `svalue<-` method.

Displaying icons and images stored in files

The `gWidgets` package provides a few stock icons that can be added to various GUI components. A list of the defined stock icons is returned by the function `getStockIcons`. The names attribute defines the valid stock icon names. It was mentioned that if a button's label matches a stock icon name, that icon will appear adjacent to the label.

Other graphic files and the stock icons can be displayed by the `gimage` widget. [1] 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. Stock icons are specified by using their name for the `filename` argument and the character string `"stock"` for the `dirname` argument. [2]

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

The default handler is a button click handler.

To illustrate, a simple means to embed a graph within a GUI is as follows:

```
f <- tempfile()
png(f)                                  # not gWidgetstcltk!
hist(rnorm(100))
dev.off()
#
w <- gwindow("Example to show a graphic")
gimage(basename(f), dirname(f), cont=w)
```

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

Example 3.1: Adding and using stock icons
This example shows how to add to the available stock icons and use `gimage` to display them. It creates a table to select a color from, as an alternative to a more complicated color chooser dialog. [3]

[1] Not all file types may be displayed by each toolkit, in particular `gWidgetstcltk` can only display gif, ppm, and xbm files.

[2] 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"`.

[3] If `gWidgetstcltk` is used 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 <- sprintf("%s%scolor-%s.png", iconDir,
                      .Platform$file.sep, i)
  png(file=filename, width=iconSize, height=iconSize)
  grid.newpage()
  grid.draw(rectGrob(gp=gpar(fill=i)))
  dev.off()
  return(filename)
}
```

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

```
icons <- sapply(someColors, makeColorIcon)
iconNames <- sprintf("color-%s", someColors)
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,...) galert(h$action, parent=w)
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[ind], handler=f,
                       action=iconNames[ind], cont=tbl)
  }
}
```

gsvg Finally, we mention the `gsvg` constructor is similar to `gimage`, but allows one to display SVG files, as produced by the `svg` driver, say. It currently is not available for `gWidgetsRGtk2` and `gWidgetstcltk`.

3.1 Text editing controls

The gWidgets package, following the underlying toolkits, has two main widgets for editing text: gedit for a single line of editable text, and gtext for multi-line, editable text. Each provides much less flexibility than is possible.

Single-line, editable text

The gedit constructor produces a widget to display a single line of editable text. The main property is the initial text which can be set through the text argument. If this is not specified, and the argument initial.msg is, then this initial message is shown until the widget receives the focus to guide the user. 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.

A simple usage might be:

```
w <- gwindow("Simple gedit example", visible=FALSE)
g <- ggroup(cont=w)
e <- gedit("", initial.msg="Enter your name...", cont=g)
visible(w) <- TRUE
```

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.

Auto completion The underlying toolkits offer some form of auto completion where the entered text is matched against a list of values. These values anticipate what a user wishes to type and a simple means to complete a entry is offered. The [<- method allows these values to be specified through a character vector, as in obj[] <- values.

For example, the following can be used to collect one of the 50 state names in the U.S.:

```
w <- gwindow("gedit example", visible=FALSE)
g <- ggroup(cont=w)
glabel("State name:", cont=g)
```

```
e <- gedit("", cont=g)
e[] <- state.name
visible(w) <- TRUE
```

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

Example 3.2: Validation
GUIs for R may differ a bit from many GUIs users typically interacts with, as R users expect to be able to use variables and expressions where typically a GUI expects just characters or numbers. As such, it is helpful to indicate to the user if their value is a valid expression. This example shows how to implement a validation framework on a single-line edit widget so that the user has feedback when an expression will not evaluate properly. When the value is invalid we set the text color to red.

```
w <- gwindow("Validation example")
tbl <- glayout(cont=w)
tbl[1,1] <- "R expression:"
tbl[1,2] <- (e <- gedit("", cont = tbl))
```

We use the `evaluate` package to see if the expression is valid.

```
require(evaluate)
isValid <- function(e) {
  out <- try(evaluate:::evaluate(e), silent=TRUE)
  !(inherits(out, "try-error") ||
    is(out[[2]], "error"))
}
```

We validate our expression when the user commits the change, by pressing the return key while the widget has focus. Alternatively, we could have used `addHandlerKeystroke`, to validate after each key press, or `addHandlerBlur`, to validate when the widget loses focus.

```
addHandlerChanged(e, handler = function(h,...) {
  curVal <- svalue(h$obj)
  if(isValid(curVal)) {
    font(h$obj) <- c(color="black")
  } else {
    font(h$obj) <- c(color="red")
    focus(h$obj) <- TRUE
  }
})
```

Table 3.2: Possible specifications 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")`

Attribute	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

Multi-line, editable text

The `gtext` constructor produces a multi-line text editing widget with scrollbars to accommodate large amounts of text. The `text` argument is for specifying the initial text. The initial width and height can be set through similarly named arguments. For widgets with scrollbars, specifying an initial size is usually required as there otherwise is no indication as to how large the widget should be.

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 clear the buffer, the `dispose` method is used. The `insert` method adds text to a buffer. The signature is `insert(obj, text, where, font.attr)` where `text` is a character vector. New text is added to the end of the buffer, by default, but the `where` argument can specify `"beginning"` or `"at.cursor"`.

Fonts Fonts can be specified for the entire buffer or the selection using the specifications in Table 3.1. To specify fonts for the entire buffer use the `font.attr` argument of the constructor. The `font<-` method serves the same purpose, provided there is no selection when called. If there is a selection, the font change will only be applied to the selection. Finally, the `font.attr` argument for the `insert` method specifies the font attributes for the inserted text.

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

Example 3.3: A calculator
This example shows how one might use the widgets just discussed to make

a GUI that resembles a calculator. Such a GUI may offer familiarity to new R users, although certainly it 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", visible=FALSE)
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))
visible(w) <- TRUE
```

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(outputArea) <- ""              # 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))),
             silent=TRUE)
  if(inherits(out, "try-error")) {
    galert("There is an error")
  } else {
    svalue(outputArea) <- out
    svalue(eqnArea) <- ""            # restart
  }
})
```

3.2 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. Figure 3.1 shows a simple GUI for the `EBImage` package allowing a user to adjust a few of the image properties using various selection widget.

In `gWidgets` the abstract view is that the user is selecting from an set of items stored as a vector (or data frame). The familiar 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. The `svalue` method then refers to the user selected value. This selection may be a value or an index, and the `svalue` method has the argument `index` to specify which.

This section discusses several selection controls that serve a similar purpose but make different use of screen space.

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` (there is no third state). By default the label will be drawn aside a checkbox, if the argument `use.togglebutton` is `TRUE`, a toggle button – which appears depressed when `TRUE` – is used.

In Figure 3.1 a toggle button is used for "Thresh" and could be constructed as

```
w <- gwindow("Checkbox example with toggle button")
cb <- gcheckbox("Thresh", checked=TRUE, use.togglebutton=TRUE,
                cont=w)
```

The `svalue` method returns a logical indicating if the widget is in the checked state. Use `svalue<-` to set the state. The label's value is returned by the `[` method, and can be adjusted through `[<-`. (We take the abstract

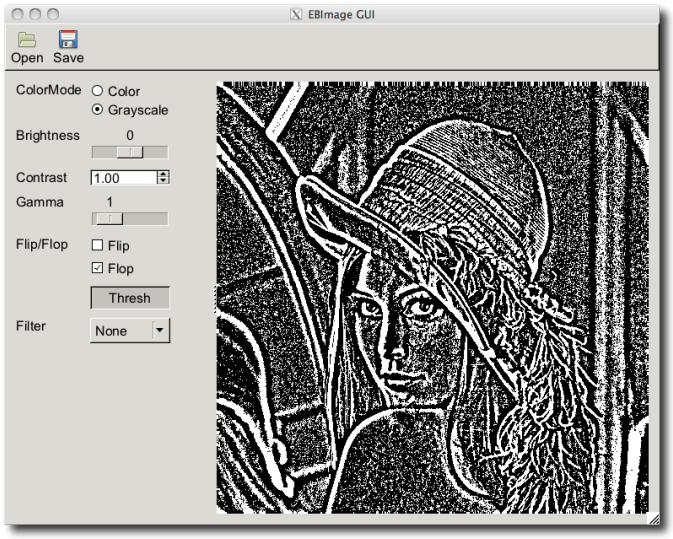

Figure 3.1: A simple GUI for the `EBImage` package illustrating many selection widgets

view that the user is selecting, or not, from the length-1 vector, so `[` is used to set the data to select from.)

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. For example

```
w <- gwindow("checkbox example")
cb <- gcheckbox("label", cont=w, handler=function(h,...) {
  if(svalue(h$obj))                    # it is checked
    print("define handler here")
})
```

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 (2 or more). 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, by index, with a default of the first.

In Figure 3.1 a radio button is used for "ColorMode" and could be constructed as

```
w <- gwindow("Radio button example")
rb <- gradio(c("Color", "Grayscale"), selected=2,
             horizontal=FALSE, cont=w)
```

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 and are referenced through the `[` method. They may be respecified with the `[<-` method. For convenience, the `length` method returns the number of labels.

The handler, if given to the constructor or set with `addHandlerChanged`, is called on a click event.

A group of checkboxes

The group of checkboxes is produced by the `gcheckboxgroup` constructor. This convenience widget is similar to a radio group, only it allows the selection of none, one or more of a set of items. The `items` argument 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`).

For some toolkits, the argument specification `use.table=TRUE` will render the widget in a table with checkboxes to select from. This allows much larger sets of items to comfortably be used. This provides a similar functionality as using the `gtable` widget with multiple selection.

In Figure 3.1 a group of check boxes is used to allow the user to "flip" or "flop" the image. It could be created with

```
w <- gwindow("Checkbox group example")
cbg <- gcheckboxgroup(c("Flip","Flop"), horizontal=FALSE,
                      checked=c(FALSE, TRUE), cont=w)
```

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 checkbox group is like both a checkbox and a radio button group, one can set the selected values three 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.

That is, each of these has the same effect:

```
svalue(cbg) <- c("Flop")
svalue(cbg) <- c(FALSE, TRUE)
svalue(cbg, index=TRUE) <- 2
```

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 combo box

Combo boxes are constructed by gcombobox. As with the other selection widgets, the choices are specified to the argument items. However, this may be a vector of values or a data frame whose first column defines the choices. For toolkits which support icons in the combo box widget, if the data is specified as a data frame, the second column signifies which stock icon is to be used. By design, a third column specifies a tooltip to appear when the mouse hovers over a possible selection, but this is only implemented for gWidgetsQt.

The combo box in Figure 3.1 could be done through:

```
w <- gwindow("gcombobox example")
cb <- gcombobox(c("None", "Low", "High"), cont=w)
```

This example shows how to create agcombo box to select from the available stock icons. For toolkits that support icons in a combo box, they appear next to the label.

```
nms <- getStockIcons()                    # gWidgets icons
d <- data.frame(names=names(nms), icons=names(nms),
                stringsAsFactors=FALSE)
w <- gwindow("Combo box with icons example")
cb <- gcombobox(d, cont=w)
```

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.

Methods 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 value can be set by its value through the svalue<- method, or by index if index is TRUE. The [method returns the items of the data store, and [<- is used to assign new values to the data store. The value may be a vector, or data frame if an icon or tooltip is being assigned. The length method returns the number of possible selections.

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 method sets a handler to respond to keystroke events.

A slider control

The gslider constructor creates a slider that allows the user to select a value from the specified sequence. The basic arguments mirror that of the seq function in R: from, to, and by. If from is a vector, then it is assumed it presents an orderable sequence of values to selecte from. 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.

In Figure 3.1 a slider is used to update the brightness. The call is similar to:

```
w <- gwindow("Slider example")
brightness <- gslider(from=-1, to=1, by=.05, value=0,
   handler=function(h,...) {
     cat("Update picture with brightness", svalue(h$obj), "\n")
   }, cont=w)
```

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 **??** 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 when used with numeric data, but presents the user a more precise way to select the value. The from, to and by arguments must be specified. The argument digits specifies how many digits are displayed.

In Figure 3.1 a spin button is used to adjust the contrast, a numeric value. The following will reproduce it

```
w <- gwindow("Spin button example")
sp <- gspinbutton(from=0, to=10, by=.05, value=1, cont=w)
```

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 initiates this selection.

The "Open" button in Figure 3.1 is bound to this action:

```
f <- gfile("Open an image file",
           type="open",
           filter=list("Image file"=list(
                         patterns=c("*.gif", "*.jpeg", "*.png")
```

```
                          ),
             "All files" = list(patterns = c("*"))
             ))
if(!is.na(f))
  readImage(f) ## ...
```

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, used above, will filter the possible selections. The dialog returns the path of the file, or `NA` if the dialog was canceled.

Although working with the return value is easy enough, if desired, one can 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.

Selecting a date

The `gcalendar` constructor returns a widget for selecting a date. If there is a native widget in the underlying toolkit, this will be a text area with a button to open a date selection widget. Otherwise it is just a text entry widget. The argument `text` argument specifies the initial text. The format of the date is specified by the `format` argument.

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.

3.3 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 `gWidgetsRGtk2` and `gWidgetsQt` is much faster and able to handle larger data stores than under `gWidgetstcltk`, as there is no native table widget in Tcl/Tk. All perform well on moderate-sized data sets (10 or so columns and fewer than 500 rows). [4]

The data is specified through the `items` argument. This value may be a data frame, matrix or vector. Vectors and matrices are coerced to data frames, with `stringsAsFactors=FALSE`. The data is presented in a tabular

[4] For some of the toolkits, other similar widgets are available. The `gbigtable` constructor of `gWidgetsQt` is faster with large tables and has selection by rectangle, the `gdfedit` constructor of `gWidgetsRGtk2` shows very large tables taking advantage of the underlying `RGtk2Extras` package.

form, with column headers derived from the `names` attribute of the data frame (but no row names).

To illustrate, a widget to select from the available data frames in the global environment can be generated with

```
availDfs <- function(envir=.GlobalEnv) {
  x <- ls(envir=envir)
  x[sapply(x, function(i) is.data.frame(get(i, envir=envir)))]
  data.frame(dfs=x, stringsAsFactors=FALSE)
}
#
w <- gwindow("gtable example")
dfs <- gtable(availDfs(), cont=w)
```

Often the table widget is added to a box container with the argument `expand=TRUE`. Otherwise, The size of the widget can be specified by `size<-`. This can be list with components `width` and `height` (pixel widths). As well, the component `columnWidths` can be used to specify the column widths. (Otherwise a heuristic is employed.)

The `icon.FUN` argument can be 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.

Selection Users can select a case (row) – not by observation (a cell) – from this widget. The value returned by a selection is controlled by the constructor's 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.

Methods 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 (below). The argument `drop` specifies if just the chosen column's value is returned (the default) or, if specified as `FALSE`, the entire row.

The underlying data store is referenced by the `[` method. Indices may be used to access a slice. 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.

To update the list of data frames in our `dfs` widget, one can define a function such as

```
updateDfs <- function() {
  dfs[] <- availDfs()
}
```

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

To add a handler to the data frame selection widget above, we could have

```
addHandlerDoubleclick(dfs, handler=function(h,...) {
  val <- svalue(h$obj)
  ## some action
  print(summary(get(val, envir=.GlobalEnv)))
})
```

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. The simplest case is if a column number is specified to the `filter.column` argument. In which case a combo box is added to the widget with values taken from the unique values in the specified column. Changing the value of the combo box restricts the display of the data to just those rows where the value in the filter column matches the combo box value. 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 combo box 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 3.5 for an application.

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 toolkit packages directly can involve a fair amount of coding as compared to `gtable`, which makes it very easy. However, `gWidgets` provides far less functionality. For example, there is no means to adjust the formatting of the displayed text, or to embed other widgets into the tabular display, such as check boxes.

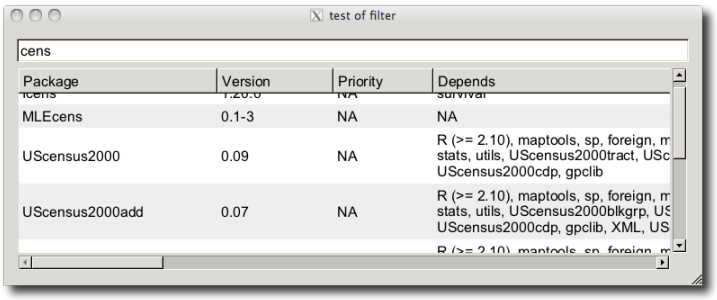

Figure 3.2: Example of using a filter to narrow the display of tabular data

Example 3.4: 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 below. 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(sprintf("You selected the %s %s", val[,1], val[,2]))
})
```

Example 3.5: 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 table for selection unless some means of searching or filtering the data is used. This example uses the many possible CRAN packages, to show how a `gedit` instance can be used as a search box to filter the display of data (Figure 3.2). 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 we skip adding 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` value of `"manual"` allows us to filter by specifying a logical vector.

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) {
  grepl(curVal, vals)
}
```

Finally, the `addHandlerKeystroke` method calls its handler every time 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, vals)
  visible(tbl) <- vis
})
```

Example 3.6: Using the "observer pattern" to write a workspace view
This example takes the long way to make a workspace browser. (The short way is `gvarbrowser`.) The goal is to produce a GUI that will allow the user to view the objects in their current workspace. We would like these views to by dynamic though – when the workspace changes we would like the views to update. Furthermore, we may want to have different views, such as one for functions and one for data sets. Such requirements lead us to the observer design pattern in programming, where each view is notified of changes to an "observable."

The observer design pattern uses observables, objects which record a list of observers and notify them when there is a change and observers. We will use an observable for a data model to store information about our current workspace and the views will be the observers. The basic observable pattern can be coded using reference classes as:

```
setRefClass("Observable",
            fields=list(..observers="list"),
```

```
            methods=list(
              add_observer=function(o) {
                "Add an observer."
                ..observers <<- c(..observers, o)
              },
              remove_observer=function(o) {
                "Remove observer"
                ind <- sapply(..observers,
                              function(i) identical(i, o))
                if(any(ind))
                  ..observers[[which(ind)]] <<- NULL

              },
              notify_observers=function(...) {
                "Notify observers there has been a change"
                sapply(..observers, function(o) {
                  o$update(...)
                })
              }))
```

This can get more involved (we can block observers, etc.), but we keep it simple for this example.

The basic observer pattern just creates a class for observers so that they have a `update` method. Again, a simple implementation follows:

```
setRefClass("Observer",
      fields=list(o = "function"),
      methods=list(
        update=function(...) {
          "Call self. Arguments passed by notify_observers"
          o(...)
        }))
```

Our workspace will be stored in a model instance. Below we define one that extends the observable class. This is meant to be subclassed. We define a `set` method to both set a properties values and to notify the observers.

```
setRefClass("Model",
      contains="Observable",
      methods=list(
        set=function(key, value, notify=TRUE) {
          "Set key field to value. Notify observers."
          assign(key, value, inherits=TRUE)
          if(notify)
            notify_observers(model=.self)
          invisible()
        }))
```

To illustrate how this works, we define a simple subclass of our `Model` call and an observer.

```
TestModel <- setRefClass("TestModel",
                         contains="Model",
                         fields=list(prop1="character"))
m <- TestModel$new()
f <- function(model) print(model$prop1)
o <- getRefClass("Observer")$new(o=f)
m$set("prop1", "Some value")
m$add_observer(o)
m$set("prop1", "A new value")
```

```
[1] "A new value"
```

For the task at hand, we subclass the `Model` class. Notifying the observers is potentially expensive and disruptive, so we check in the `update` method that objects have indeed changed. The `digest` function is used to create a summary record of the workspace each time `update` is run. If there are changes, then we notify the observers. Below we filter out the values with class `envRefClass`.

The `get_objects` method, which returns the names of the objects in the work space, adds some complexity, but allows us to filter by class.

```
require(digest)
setRefClass("WSModel",
            contains="Model",
            fields=list(
              ws_objects="character",
              ws_objects_digest="character"
              ),
            methods=list(
              update=function() {
                "update vector of ws_objects if applicable"
                x <- sort(ls(envir=.GlobalEnv))
                ## filter out envRefClass objects --
                isRef <- function(i)
                  is(get(i, envir=.GlobalEnv), "envRefClass")
                x <- x[!sapply(x, function(i) isRef(i))]
                ds <- sapply(x, function(i)
                             digest(get(i, envir=.GlobalEnv)))

                if((length(ds) != length(ws_objects_digest)) ||
                   length(ws_objects_digest) == 0 ||
                   any(ds != ws_objects_digest)) {
                  ws_objects_digest <<- ds
                  ws_objects <<- x
                  notify_observers(model=.self) # pass model
```

```
                }
                invisible()
              },
              get_objects=function(klass) {
                "Get objects. If klass given, restrict to those.
Klass may have ! in front, as in '!function'"
                if(missing(klass) || length(klass) == 0)
                  return(ws_objects)
                ind <- sapply(ws_objects, function(i) {
                  x <- get(i, envir=.GlobalEnv)
                  any(sapply(klass, function(j)  {
                    if(grepl("^!", j))
                      !is(x, substr(j, 2, nchar(j)))
                    else
                      is(x, j)
                  }))
                })
                if(length(ind))
                  ws_objects[ind]
                else
                  character(0)
              }))
```

The `update` function should be called periodically. This can be done through a timer or through a `taskCallback`, but neither is illustrated.

To use this model, we create a base view class and then a widget view class:

```
setRefClass("View",
            contains="Observer",
            fields=list(model = "WSModel"),
            methods=list(
              set_model=function(new_model) {
                ## set model and add view as observer
                if(exists("model", .self))
                  model$remove_observer(.self)
                model <<- new_model
                model$add_observer(.self)
              },
              update = function(...) {
                "Update view as model has updated"
              }))
```

The following `WidgetView` class uses the template method pattern leaving subclasses to construct the widgets through the call to `initialize`.

```
setRefClass("WidgetView",
            contains="View",
            fields=list(
```

```
              klass="character", # which classes to show
              widget = "ANY"
              ),
            methods=list(
              initialize=function(parent, model, ...) {
                if(!missing(model)) set_model(model)
                if(!missing(parent)) init_widget(parent, ...)
                initFields()
                .self
              },
              init_widget=function(parent, ...) {
                "Initialize widget"
              }))
```

The first subclass of the widget view class will be one where we show the values in the workspace using a table widget. To generate data on each object, we define some S3 classes. These are more convenient than a reference classes for this task. First we want a nice description of the size of the object:

```
sizeOf <- function(x, ...) UseMethod("sizeOf")
sizeOf.default <- function(x, ...) "NA"
sizeOf.character <- sizeOf.numeric <-
  function(x, ...) sprintf("%s elements", length(x))
sizeOf.matrix <- function(x, ...)
  sprintf("%s x %s", nrow(x), ncol(x))
```

Now we, desire a short description of the type of object we have.

```
shortDescription <- function(x, ...) UseMethod("shortDescription")
shortDescription.default <- function(x, ...) "R object"
shortDescription.numeric <- function(x, ...) "Numeric vector"
shortDescription.integer <- function(x, ...) "Integer"
```

The following function produces a data frame summarizing the objects passed in by name to `x`. It is a bit awkward, as the data comes row by row, not column by column and we want to have a default when `x` is empty.

```
makeDataFrame <- function(x, envir=.GlobalEnv) {
  d <- data.frame(variable=character(0),
                  size=character(0), description=character(0),
                  class=character(0),
                  stringsAsFactors=FALSE)
  if(length(x)) {
    l <- lapply(x, get)
    d <- data.frame(variable=x,
                    size=sapply(l, sizeOf),
                    description=sapply(l, shortDescription),
                    class = sapply(l, function(i) class(i)[1]),
                    stringsAsFactors=FALSE)
```

```
  }
  d
}
```

Finally, we write a `WidgetView` subclass to view the workspace objects using a `gtable` widget.

```
TableView <-
  setRefClass("TableView",
        contains="WidgetView",
        methods=list(
          init_widget=function(parent, ...) {
            widget <<- gtable(makeDataFrame(character(0)),
                              cont=parent, ...)
          },
          update=function(...) {
            widget[] <- makeDataFrame(model$get_objects(klass))
          }))
```

To illustrate the flexibility of this framework, we also define a subclass to show just the data frames in a combo box.

```
DfView <-
  setRefClass("DfView",
              contains="TableView",
              methods=list(
                initFields= function(...) klass <<- "data.frame",
                init_widget = function(parent, ...) {
                  d <- data.frame("Data frames"=character(0),
                                  stringsAsFactors=FALSE)
                  widget <<- gcombobox(d, cont=parent, ...)
                },
                update = function(...) {
                  widget[] <- model$get_objects(klass)
                }))
```

We can put these pieces together to make a simple GUI. The combo box is a bit contrived here, but having means to select a data frame is common to R GUIs.

```
w <- gwindow()
nb <- gnotebook(cont=w)
#
m <- getRefClass("WSModel")$new()
#
view <- TableView$new(parent=nb, model=m, label="data")
view$klass <- c("factor", "numeric", "character",
                "data.frame", "matrix", "list")
#
view1 <- TableView$new(parent=nb, model=m,
```

```
                        label="not a function")
view1$klass <- "!function"
#
view2 <- TableView$new(parent=nb, model=m, label="all")
view3 <- DfView$new(parent=nb, model=m, label = "data frames")
#
m$update()
svalue(nb) <- 1
```

3.4 Display of hierarchical data

The `gtree` constructor can be used to display hierarchical structures, such as a file system or the components of a list. Populating the `gtree` widget is done by dynamically computing the child components. A parameterizion of the data to be displayed in terms of the path of the node that is currently selected is used.

The `offspring` argument is assigned a function of two arguments, the path a particular node and the arbitrary object 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 identifies each of the offspring.

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.

As an illustration, this function produces an offspring function to explore the hierarchical structure of a list. It uses a closure to encapsulate the list, rather than using the `offspring.data` argument or a global variable.

```
listOffspring <- function(lst) {
  offspring <- function(path=character(0),...) {
    if(length(path))
      obj <- lst[[path]]
    else
      obj <- lst
    nms <- names(obj)
    hasOffspring <- sapply(nms, function(i) {
      newobj <- obj[[i]]
      is.recursive(newobj) && !is.null(names(newobj))
    })
    data.frame(comps=nms, hasOffspring=hasOffspring,
```

```
                stringsAsFactors=FALSE)
  }
  return(offspring)
}
```

The above offspring function will produce a tree with just one column, as the data frame has just the `comps` column specifying values. By adding columns to the data frame above, say a column to record the class of the variable, more information can easily be presented

To see the above used, we define a list to explore.

```
l <- list(a="1", b= list(a="21", b="22", c=list(a="231")))
o <- listOffspring(l)
w <- gwindow("Tree test")
t <- gtree(o, cont=w)
```

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 selected node. This is what is passed to the `offspring` function. The `update` method updates the displayed tree by reconsidering the children of the root node. The method `addHandlerDoubleclick` specifies a function to call on a double click event.

Example 3.7: 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 hierarchical nature of the fitted object. As working directly with the return object is not for the faint of heart, such a GUI can be useful.

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 hierarchical nature through its nodes. This object has 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 (bypassing the `hasOffspring` function); and computes a 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) == 0)  # 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",
                      stringsAsFactors=FALSE))

  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 3.3)

```
w <- gwindow("Example of gtree")
g <- ggroup(cont=w, horizontal=FALSE)
l <- glabel("Click on the tree to investigate the partition",
```

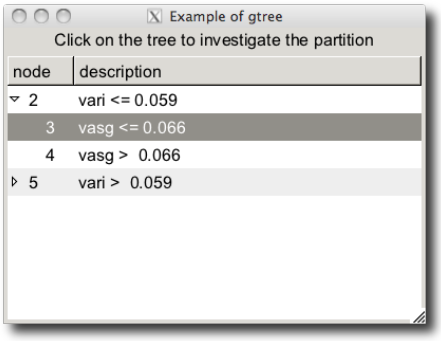

Figure 3.3: GUI to explore return value of a model fit by the `party` package.

```
              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 `party` 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, ])
  }})
```

3.5 Actions, menus and toolbars

Actions are invisible objects representing an application command that is executable through one or more widgets. See **??** for more details. Actions in `gWidgets` are created through the `gaction` constructor. The arguments are `label`, `tooltip`, `icon`, `key.accel`, `parent` and the standard `handler` and `action`. The label appears as the text on a button, the menu item or toolbar text, whereas the icon will decorate the same if possible. For some toolkits, the tooltip pops up when the mouse hovers. (See also the `tooltip<-` method.) The key accelerator will bind a keyboard shortcut, such as Control-s to an action. The `parent` argument is used to specify a widget whose toplevel container will process the shortcut.

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 proxies of the action are set through one call.

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.

```
w <- gwindow("gaction example")
a <- gaction("click me", tooltip="Click for a message",
             icon="ok",
             handler=function(h,...) {
               print("Hello")
               })
b <- gbutton(action=a, cont=w)
## .. to change
enabled(a) <- FALSE                    # can't click now
```

Action handlers generally do not have the sender object (`b` above) passed back to them.

Toolbars

Toolbars and menubars are implemented in `gWidgets` using `gaction` items. Toolbars (and menubars) are specified using a named list of menu components.

For a toolbar, the list has a simple structure. Each named component 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.

For example:

```
w <- gwindow("gtoolbar example")
l <- list(Open=gaction("Open", icon="open",
            handler=function(...) print("Open")),
          Close=gaction("Close", icon="cancel",
            handler=function(...) print("Close")),
          sep=gseparator(),
          Quit=gaction("Quit", icon="quit",
            handler=function(...) print("Quit"))
          )
tb <- gtoolbar(l, cont=w)
gtext("Placeholder", cont=w)
```

The `gtoolbar` constructor takes the list as its first argument. As toolbars belong to the window, the corresponding `gWidgets` objects use a `gwindow` object as the parent container. (Some of the toolkits relax this to allow other containers.) 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 submenus require a nested structure. This 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 submenu 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.

A example of such a list might be

```
f <- function(h,...) print(h$action)                # a stub
ml <- list(File=list(
             Source=gaction("Source file...", action="source",
               handler=f),
             Load=gaction("Load workspace...", action="load",
               handler=f),
             sep=gseparator(),
             New=list(
               Plot=gaction("Plot window", action="plot",
                 handler=f),
               Rfile=gaction("R file", action="file", handler=f)
               )
             ),
           Help=list(
             about=gaction("About", action="about", handler=f)
             )
           )
```

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

Menubar and Toolbar 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.

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 3.8: Popup menus
This example shows how to add a simple popup menu to a button.

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

4

gWidgets: R-specific widgets

The gWidgets package provides some R specific widgets for producing GUIs. Table 4 lists them.

4.1 A graphics device

Some toolkits support an embeddable graphics device (gWidgetsRGtk2 through cairoDevice, gWidgetsQt through qtutils). In which case, the ggraphics constructor produces a widget that can be added to a container. The arguments width, height, dpi, and ps are similar to other graphics devices.

When working with multiple devices, it becomes necessary to switch between devices. A ggraphics instance can be made to represent the current device if the user clicks in the window. Otherwise, the visible<- method can be used to set the object as the current device. The ggraphicsnotebook creates a notebook that allows the user to easily navigate multiple graphics devices.

Table 4.1: Table of constructors for R-specific widgets in gWidgets

Constructor	Description
ggraphics	An embeddable graphics device
ggraphicsnotebook	Notebook for multiple devices
gdf	A data frame editor
gdf	Notebook for multiple gdf instances
gvarbrowser	GUI for browsing variables in the workspace
gcommandline	Command line widget
gformlayout	Creates a GUI from a list specifying layout
ggenericwidget	Creates a GUI for a function based on its formal arguments or a defining list

The default handler for the widget is set by `addHandlerClicked`. The coordinates of the mouse click, in user coordinates, are passed to the handler in the components `x` and `y`. As well, the method `addHandlerChanged` is used to assign a handler to call when a region is selected by dragging the mouse. The components `x` and `y` describe the rectangle that was traced out, again in user coordinates.

This shows how the two can be used:

```
library(gWidgets); options(guiToolkit="RGtk2")
w <- gwindow("ggraphics example", visible=FALSE)
g <- ggraphics(cont=w)
x <- mtcars$wt; y <- mtcars$mpg
#
addHandlerClicked(g, handler=function(h,...) {
  cat(sprintf("You clicked %.2f x %.2f\n", h$x, h$y))
})
addHandlerChanged(g, handler=function(h,...) {
  rx <- h$x; ry <- h$y
  if(diff(rx) > diff(range(x))/100 &&
     diff(ry) > diff(range(y))/100) {
    ind <- rx[1] <= x & x <= rx[2] & ry[1] <=y & y <= ry[2]
    if(any(ind))
      print(cbind(x=x[ind], y=y[ind]))
  }
})
visible(w) <- TRUE
#
plot(x, y)
```

Example 4.1: A GUI for filtering and visualizing a data set

A common GUI application for data analysis consists of means to visualize, query, aggregate and filter a data set. This example shows how one can create such a GUI using `gWidgets` featuring an embedded graphics device. In addition a visual display of the filtered data, and a means to filter, or narrow, the data that is under consideration, is presented (Figure 4.1). Although, our example is not too feature rich, it illustrates a framework that can easily be extended.

This example is centered around filtering a data set, we choose a convenient one and give it a non-specific name.

```
data("Cars93", package="MASS")
x <- Cars93
```

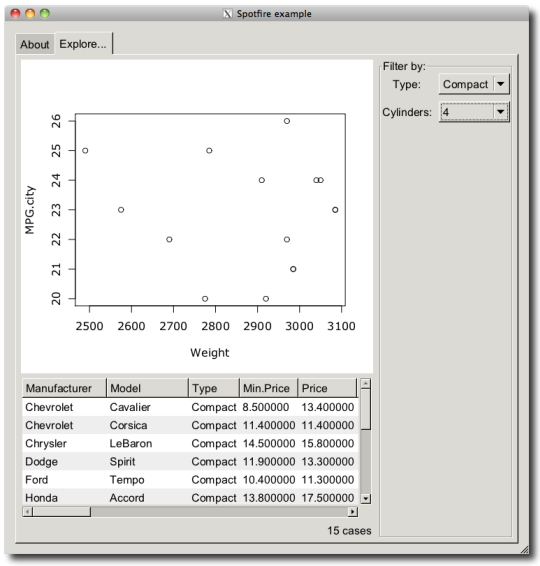

Figure 4.1: A GUI to filter a data frame and display an accompanying graphic.

We use a notebook to hold two tabs, one to give information and one for the main GUI. This basic design comes from the spotfire demos at `tibco.com`.

```
w <- gwindow("Spotfire example", visible=FALSE)
nb <- gnotebook(cont=w)
```

We use a simple label for information, although a more detailed description would be warranted in an actual application.

```
descr <- glabel(gettext("A basic GUI to explore a data set"),
                cont=nb, label=gettext("About"))
```

Now we specify the layout for the second tab. This is a nested layout made up of three box containers. The first, `g`, uses a horizontal layout in which we pack in box containers that will use a vertical layout.

```
g <- ggroup(cont=nb, label=gettext("Explore..."))
lg <- ggroup(cont=g, horizontal=FALSE)
rg <- ggroup(cont=g, horizontal=FALSE)
```

The left side will contain an embedded graphic device and a view of the filtered data. The `ggraphics` widget provides the graphic device.

```
ggraphics(cont = lg)
```

Our view of the data is provided by the `gtable` widget, which facilitates the display of a data frame. The last two arguments allow for multiple selection (for marking points on the graphic) and for filtering through the `visible<-` method. In addition to the table, we add a label to display the number of cases being shown. This label is packed into a box container, and forced to the right side through the `addSpring` method of the box container.

```
tbl <- gtable(x, cont = lg, multiple=TRUE, filter.FUN="manual")
size(tbl) <- c(500, 200)                # set size
labelg <- ggroup(cont = lg)
addSpring(labelg)
noCases <- glabel("", cont = labelg)
```

The right panel is used to provide the user a means to filter the display. We place the widgets used to do this within a frame to guide the user.

```
filterg <- gframe(gettext("Filter by:"), cont = rg, expand=TRUE)
```

The controls are layed out in a grid. We have two here to filter by: type and the number of cylinders. data set.

```
lyt <- glayout(cont=filterg)
l <- list() # store widgets
lyt[1,1] <- "Type:"
lyt[1,2] <- (l$Type <- gcombobox(c("", levels(x$Type)),
                                 cont=lyt))
lyt[2,1] <- "Cylinders:"
lyt[2,2] <- (l$Cylinders <-
              gcombobox(c("", levels(x$Cylinders)), cont=lyt))
```

Of course, we could use many more criteria to filter. The above filters are naturally represented by a combo box. However, one could have used many different styles, depending on the type of data. For instance, one could employ a checkbox to filter through Boolean data, a slider to pick out numeric data, or a text box to specify a filtering by a string. The type of data dictates this.

There are three main components in our GUI: the display, the table and the filters. We create handlers to connect these components. This first handler is used to update the data frame when the filter controls are changed. For this we need to compute a logical variable indicating which rows are to be considered. Within the definition of the function, we use the global variables `l`, `tbl` and `noCases`.

```
updateDataFrame <- function(...) {
  ind <- rep(TRUE, nrow(x))
  for(i in c("Type", "Cylinders"))  {
    if((val <- svalue(l[[i]])) != "")
      ind <- ind & (x[,i] == val)
```

```
  }

  visible(tbl) <- ind                    # update table

  nsprintf <- function(n, msg1, msg2,...)
    ngettext(n, sprintf(msg1, n), sprintf(msg2,n), ...)
  svalue(noCases) <- nsprintf(sum(ind), "%s case", "%s cases")
}
```

This next function is used to update the graphic. Of course, this graphic is not so interesting, but in a real application, it should be.

```
updateGraphic <- function(...) {
  ind <- visible(tbl)
  if(any(ind))
    plot(MPG.city ~ Weight, data=x[ind,])
  else
    plot.new()
}
```

We now add a handler to be called whenever one of our combo boxes is changed. This handler simply calls both our update functions.

```
f <- function(h, ...) {
  updateDataFrame()
  updateGraphic()
}
sapply(l, function(i) addHandlerChanged(i, handler=f))
```

For the data display, we wish to allow the user to view individual cases by clicking on a row of the table. The following will do so.

```
addHandlerClicked(tbl, handler=function(h,...) {
  updateGraphic()
  ind <- svalue(h$obj, index=TRUE)
  points(MPG.city ~ Weight, cex=2, col="red", pch=16,
         data=x[ind,])
})
```

We could also use the `addHandlerChanged` method to add a handler to call when the user drags our a region in the graphics device, but leave this for the interested reader.

Finally, we draw the GUI with an initial graphic (the `visible` method draws the GUI here, unlike its use with `gtable`).

```
visible(w) <- TRUE
updateGraphic()
```

4.2 A data frame editor

The gdf constructor returns a widget for editing data frames. The intent is for each toolkit to produce a widget at least as powerful as the data.entry function. The implementations differ between toolkits, with some offering much more. We describe what is in common below. [1]

The constructor has its main argument items to specify the data frame to edit. A basic usage might be:

```
w <- gwindow("gdf example")
df <- gdf(mtcars, cont=w)
## ... make some edits ...
newDataFrame <- df[,]                    # store changes
```

Some toolkits render columns differently for different data types, and some toolkits use character values for all the data, so values must be coerced back when transferring to R values. As such, column types are important. Even if one is starting with a 0-row data frame, the columns types should be defined as desired. Also, factors and character types may be treated differently, although they may render in a similar manner.

Methods The svalue method will return the selected values or selected indices if index=TRUE is given. The svalue<- method is used to specify the selection by index. This is a vector or row indices, or for some toolkits a list with components rows and columns indicating the selection to mark. 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. In particular, it may not be possible to change the type of a column, or add new rows or columns through these methods. Using no indices, as in the above example with df[,], will return the current data frame. The current data frame can be completely replaced, when no indices are specified in the replacement call.

There are also several methods defined that follow those of a data frame: dimnames, dimnames<-, names, names<-, and length.

The following methods can be used to assign handlers: addHandlerChanged (cell changed), addHandlerClicked, addHandlerDoubleclick. Some toolkits also have addHandlerColumnClicked, addHandlerColumnDoubleclick, and addHandlerColumnRightclick implemented.

[1] 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, as with gtable, the widget under gWidgetstcltk is slower, but can load a moderately sized data frame in a reasonable time.

For gWidgetsRGtk2 there is also the gdfedit widget which can handle very large data sets and has many improved usability features. The gWidgets function merely wraps the gtkDfEdit function from RGtk2Extras. This function is not exported by gWidgets, so the toolkit package must be loaded before use.

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

Workspace browser

A workspace browser is constructed by gvarbrowser, providing a means to browse and select the objects in the current global environment. This workspace browser uses a tree widget to display the items and their named components.

The svalue method returns name of the currently selected value using $ to refer to child elements. One can call svalue on this to get the object.

The default handler object calls do.call on the object for the function specified by name 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. One can pass in other handler functions if desired.

The update method will update the list of items being displayed. This can be can be time consuming. Some heuristics are employed to do this automatically, if the size of the workspace is modest enough. Otherwise it can be done by request.

Example 4.2: Using drag and drop with gWidgets
We use the drag and drop features to create a means to plot variables from the workspace browser. Our basic layout is fairly simple. We place the workspace browser on the left, and on the right have a graphic device and few labels to act as drop targets.

```
w <- gwindow("Drag and drop example")
g <- ggroup(cont=w)
vb <- gvarbrowser(cont=g)
g1 <- ggroup(horizontal=FALSE, cont=g, expand=TRUE)
ggraphics(cont=g1)
xlabel <- glabel("", cont=g1)
ylabel <- glabel("", cont=g1)
clear <- gbutton("clear", cont=g1)
```

We create a function to initialize the interface.

```
init_txt <- "<Drop %s variable here>"
initUI <- function(...) {
  svalue(xlabel) <- sprintf(init_txt, "x")
  svalue(ylabel) <- sprintf(init_txt, "y")
  enabled(ylabel) <- FALSE
}
initUI()                                  # initial call
```

Separating this out allows us to link it to the clear button.

```
addHandlerClicked(clear, handler=initUI)
```

Next, we write a function to update the user interface. As we didn't abstract out the data from the GUI, we need to figure out which state the GUI is currently in by consulting the text in each label.

```
updateUI <- function(...) {
  if(grepl(svalue(xlabel), sprintf(init_txt, "x"))) {
    ## none set
    enabled(ylabel) <- FALSE
  } else if(grepl(svalue(ylabel), sprintf(init_txt, "y"))) {
    ## x, not y
    enabled(ylabel) <- TRUE
    x <- eval(parse(text=svalue(xlabel)), envir=.GlobalEnv)
    plot(x, xlab=svalue(xlabel))
  } else {
    enabled(ylabel) <- TRUE
    x <- eval(parse(text=svalue(xlabel)), envir=.GlobalEnv)
    y <- eval(parse(text=svalue(ylabel)), envir=.GlobalEnv)
    plot(x, y, xlab=svalue(xlabel), ylab=svalue(ylabel))
  }
}
```

Now we add our drag and drop information. Drag and drop support in `gWidgets` is implemented through three methods: one to set a widget as a drag source (`addDropSource`), one to set a widget as a drop target (`addDropTarget`), and one to call a handler when a drop event passes over a widget (`addDropMotion`).

The `addDropSource` method needs a widget and a handler to call when a drag and drop event is initiated. This handler should return the value that will be passed to the drop target. The default value is that returned by calling `svalue` on the object. In this example we don't need to set this, as `gvarbrowser` already calls this with a drop data being the variable name using the dollar sign notation for child components.

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 of the callback, `h`, holds the drop data. In our example below we use this to update the receiver object, either the x or y label.

```
dropHandler <- function(h,...) {
  svalue(h$obj) <- h$dropdata
  updateUI()
}
addDropTarget(xlabel, handler=dropHandler)
addDropTarget(ylabel, handler=dropHandler)
```

The `addDropMotion` registers a handler for when a drag event passes over a widget. We don't need this for our GUI.

Help browser

The `ghelp` constructor produces a widget for showing help pages using a notebook container. Although R now has excellent ways to dynamically view help pages through a web browser (in particular the `helpr` package and the standard built-in help page server) this widget provides a light-weight alternative.

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 any of R's commandlines, but is provided for light-weight usage. A text box allows users 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 simplify the creation of GUIs. [2] 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, which can be modified by the user before the GUI is constructed. We leave the details to their manual pages.

[2] The `traitr` package provides another, but is not discussed here.