1

gWidgets: Overview

The gWidgets package provides a convenient means to rapidly create small to medium size GUIs within R. The package provides an abstract interface for the other graphical toolkits discussed in this text, allowing for similar access to each. Unlike the underlying toolkits, gWidgets has relatively few constructors and methods. Basically, the entire set is enumerated in Tables 3, 2.1, 1.2, and 2.2. This means gWidgets is relatively easy to learn, allowing for rapid prototyping. (It also means that as projects progress, one might need to move to a more powerful underlying toolkit.)

Typical uses of GUIs written in R involve teaching demos, sharing functionality with less technically proficient colleagues, etc. In many cases the end user may have a different operating system or different set of graphical libraries installed. The underlying toolkits supported by gWidgets are all cross platform, and gWidgets code is mostly cross toolkit, although differences do come up. (Compare for example, the same code realized on different operating system and toolkits in Figure 1.1.) This means, there is a good chance that code you write can be shared easily with someone else.

The gWidgets package started as a port to RGtk2 of the iWidgets package of Simon Urbanek written for Swing through rJava [2]. 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 gWidgetsWWW2 package provides a similar interface for web programming, but there are enough differences that we will not discuss it further.

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

[2] Simon Urbanek. iwidgets - basic gui widgets for r. http://www.rforge.net/iWidgets/index.html.

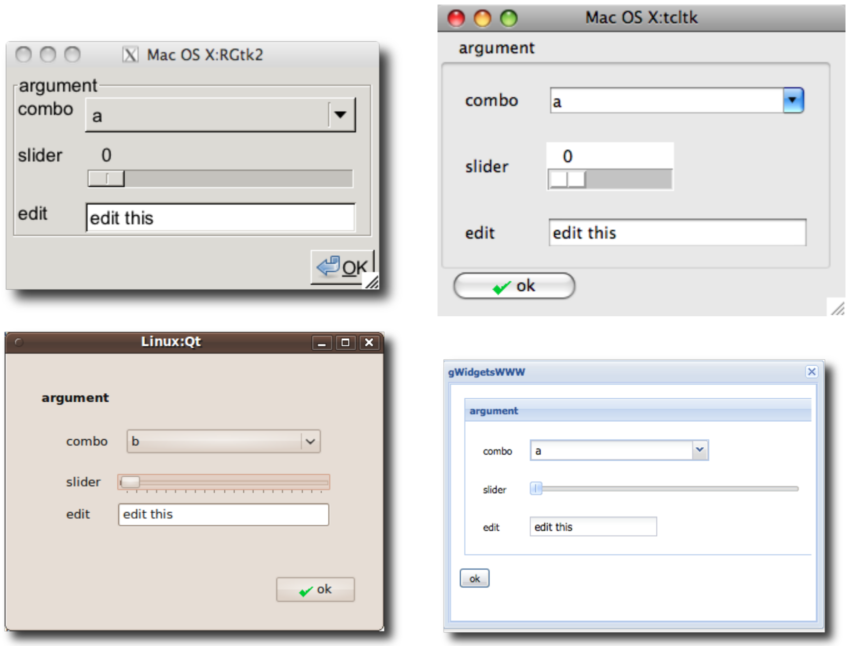

Figure 1.1: The `gWidgets` package works with different operating systems and different GUI toolkits. This shows, the same code using the `RGtk2`, `tcltk`, `qtbase` packages for a toolkit. Additionally, the `gWidgetsWWW` package is used in the lower right figure.

1.1 Constructors

We jump right in with an example[1] and leave 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 search their hard drive for files matching a user-specified pattern. The first line loads the package, the others will be described in the following.

```
require(gWidgets)
options(guiToolkit="RGtk2")
##
w <- gwindow("File search", visible=FALSE)
g <- gpanedgroup(cont=w)
## label and file selection widget
f <- ggroup(cont=g, horizontal=FALSE)
glabel("Search for (filename):", cont=f, anchor=c(-1,0))
txtPattern <- gedit("", initial.msg="Possibly wildcards",
```

[1]Many thanks to Richie Cotton for suggesting this example and its follow up in Example 3.5.

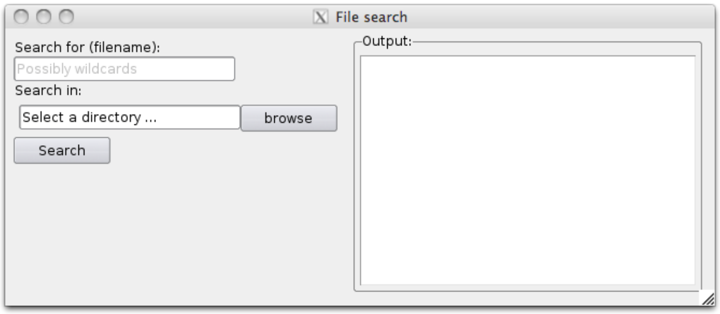

Figure 1.2: A simple GUI for search for files matching a pattern. This GUI uses a paned group to separate the controls for searching from the results.

```
                        cont=f)
##
glabel("Search in:", cont=f, anchor=c(-1,0))
startDir <- gfilebrowse(text="Select a directory ...",
                        quote=FALSE,
                        type="selectdir", cont=f)
## A button to initiate the search
searchBtn <- gbutton("Search", cont=f)
addSpring(f)
## Area for output
f1 <- gframe("Output:", cont=g, horizontal=FALSE)
searchResults <- gtext("", cont=f1, expand=TRUE)
size(searchResults) <- c(350, 200)
## add interactivity
addHandlerChanged(searchBtn, handler=function(h,...) {
  pattern <- glob2rx(svalue(txtPattern))
  fnames <- dir(svalue(startDir), pattern, recursive=TRUE)
  if(length(fnames))
    svalue(searchResults) <- fnames
  else
    galert("No matching files found", parent=w)
})
## display GUI
visible(w) <- TRUE
```

This example shows several different widgets being used to construct a GUI, as seen in Figure 1.2. For example, on the left is a text entry widget (`gedit`), a directory browsing widget (`gfilebrowse`) and a button (`gbutton`). On the right, is a multi-line text widget (`gtext`) in a framed container (`gframe`).

The widgets are all produced by calling the appropriate constructor. In the gWidgets API most of these constructors have the following basic form:

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

where some_arguments varies depending on the object being made. We discuss now the common arguments.

In the example above, we can 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 gpanedgroup constructor takes all the default arguments except for the parent container.

Containers A top-level window does not have a parent container, but the 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. The ... arguments are used to pass layout information to the parent container. This nesting defines the GUI layout, a topic taken up in Chapter 2.

The toolkit argument 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 this can cause problems due to competing event loops. In our example we have called

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

to explicitly set the toolkit. The default for the toolkit argument though is to call guiToolkit. This function will check if a toolkit has been specified, or only one is available. If neither case is so, then a menu will be provided for the user to choose one.

The handler and action arguments The handler and action arguments are used to pass in event handlers. We discuss those in Section 1.3.

Side effects The constructors produce one of three general types of widgets:

Containers such as the top level window w, the paned group g or the frame f1 (Table 2.1);

Components such as the unnamed labels, the edit area txtPattern, or the button searchBtn (Tables 3 and 4);

Dialogs such as galert and gfilebrowse (Table 1.4).

1.2 Methods

In addition to creating a GUI object, most gWidgets constructors also return 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 methods defined for gWidgets objects to adjust these properties.[2]

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 generic methods defined by gWidgets. The call

```
svalue(txtPattern)
```

demonstrates the most used new generic svalue, that is used to get the main property of the widget. For the object txtPattern, the main property is the text, for the button and label widgets this property is the label. The svalue<- assignment method is used to set this property programatically. We see the call

```
svalue(searchResults) <- fnames
```

to update the text for the multi-line text widget searchResults.

For the selection widgets (of which there are none in our example), there is a natural mapping between vectors or data frames, and the data to be selected. In this case, the user may want the value selected or the index of the selected value. The index=TRUE argument of svalue may be specified to refer to values by their index.

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 our example, the gwindow constructor is passed visible=FALSE to suppress an initial drawing, making this call to visible<- necessary to show the GUI. The visible<- generic has different interpretations for the various widgets.

[2] 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.

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<-	Set focus to widget
insert	Insert text into a multi-line text widget
font<-	Set a widget's font
update	Update widget value
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
getToolkitWidget	Return underlying toolkit widget for low-level use

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 underlying toolkit widget 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 take advantage of a richer API. In most cases, the getToolkitWidget will provide that object. For convenience, the method $ is implemented to call a method on the underlying toolkit widget and the methods [[and [[<- are implemented to inspect and set properties of the underlying widget. We will not illustrate here though, as we try to stay toolkit agnostic in our examples.

1.3 Event handlers

In our example, the search button is created with:

```
searchBtn <- gbutton("Search", cont=f)
```

However, without doing more work, this button will not initiate an action. For that we need to add an event handler, or callback, to be called when an event occurs. For our example, our event is a button click and the action we want consists of several steps: turning our pattern into a regular expression; searching for the specified pattern; and presenting the results. In our example, this is done through:

```
addHandlerChanged(searchBtn, handler=function(h,...) {
  pattern <- glob2rx(svalue(txtPattern))
  fnames <- dir(svalue(startDir), pattern, recursive=TRUE)
  if(length(fnames))
    svalue(searchResults) <- fnames
  else
    galert("No matching files found", parent=w)
})
```

Callbacks in `gWidgets` have a common signature `(h,...)` where `h` is a list with components `obj`, to pass in the receiver 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 handler calls, extra information is passed along through the list `h`. For instance, in the drop target callback the component `h$dropdata` holds the drag-and-drop value.

Although it generally is best to keep separate the construction of the widgets and the definition of the handlers, it is possible to pass in a handler for the main event through the constructor's `handler` argument. This argument, along with the `action` argument, will be passed to the widget's `addHandlerChanged` method.

The package provides a number of generic methods (Table 1.3) to add callbacks for different events beyond `addHandlerChanged`, which is used to assign a callback for the typical event for the widget, such as the clicking of a button. We refer to these methods as "`addHandlerXXX`", where the `XXX` describes the event. These are useful in the case where more than one event on that widget is of interest. For example, for single-line text widgets, like `txtPattern` 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.

As an example of combining the handler and constructor, we could have specified the search button through:

```
searchBtn <- gbutton("Search", cont=f,
                handler=function(h,...) {
                  pattern <- glob2rx(svalue(h$action$txt))
                  fnames <- dir(svalue(h$action$dir),
                                pattern, recursive=TRUE)
                  if(length(fnames))
                    svalue(h$action$results) <- fnames
                  else
                    galert("No matching files found", parent=w)
                },
                action=list(txt=txtPattern, dir=startDir,
                  results=searchResults)
                )
```

By passing in the other widgets through the `action` argument one can avoid worrying about any potential issues with scope.

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 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)
```

In the example, we use two non-modal dialogs `gfilebrowse` to select a directory and `galert` to display a transient message if no files are found through our search. Here we describe the dialogs that can be used to display a message or gather a simple amount of text. The `gfile` dialog is

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`	Set 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`	Set handler for when widget is double clicked
`addHandlerRightclick`	Set handler for when widget is right clicked
`addHandlerKeystroke`	Set handler for when key is pressed. The `key` component is set to this value, if possible.
`addHandlerFocus`	Set handler for when widget gets focus
`addHandlerBlur`	Set handler for when widget loses focus
`addHandlerExpose`	Set handler for when widget is first drawn
`addHandlerUnrealize`	Set handler for when widget is undrawn on screen
`addHandlerDestroy`	Set handler for when widget is destroyed
`addHandlerMouseMotion`	Set handler for when widget has mouse go over it
`addDropSource`	Specify a widget as a drop source
`addDropMotion`	Set handler to be called when drag event mouses over the widget
`addDropTarget`	Set 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

described in Section 3.4 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.

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.

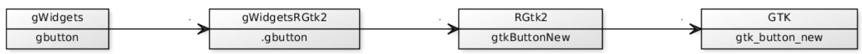

Figure 1.3: The construction of a button widget in gWidgets requires several steps

```
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))
  message("Hello", ret,"\n")
```

1.5 Installation

The `gWidgets` package interfaces with an underlying `R` package through an intermediate package. For example, Figure 1.3 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 to produce an object. This is then packaged into an S4 object to manipulate.[3]

As such, to use `gWidgets` with the GTK+ toolkit one must have installed on their computer the GTK libraries, the `RGtk2` package, the `gWidgetsRGtk2` package and the `gWidgets` package.

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 1.5 roughly describes the installation process for different operating systems and toolkits. For Windows users, some details are linked to in the `R` for Windows FAQ.

Not all features of the `gWidgets` API are implemented for a toolkit. In particular, the easiest to install toolkit package (`gWidgetstcltk`) might have the fewest features, as the `Tk` libraries themselvers are not as featureful.

[3]The S4 object consists of a `gWidgets` object and a toolkit reference. The `gWidgets` package simply provides generic functions that dispatch down to a toolkit counterpart using this S4 object. The actual class structure, methods and their inheritance is within the toolkit package. (This allows one to follow the class structure of the underlying graphical library.) As such, `gWidgets` simply provides an interface (in the sense of constructors and methods to implement) for the toolkit packages to implement. Any discussion to classes, methods and inheritance for `gWidgets` here then is for simplicity of exposition.

Table 1.4: Installation notes for GUI toolkits.

	Gtk+	Qt	Tk
Windows	Installed by `RGtk2`	Included with `qtbase`	In binary install of R
Linux	Standard	Standard	Standard
OS X	Download binary .pkg	Download from vendor	In binary install of R

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
- connecting the components through callbacks 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.[1] In our file search example from the previous chapter, we nested a framed box container inside a paned container inside a top level window.

The `gWidgets` package provides a just few types of containers: top-level windows (`gwindow`), box containers (`ggroup`, `gframe`, `gexpandgroup`), a grid container (`glayout`), a paned container (`gpanedgroup`) and a notebook container (`gnotebook`). Figure 2.1 shows most all of these employed to produce a GUI to select and then show the contents of a file.

In some toolkits, notably `tcltk`, the widget constructors require the specification of a parent container for the widget. To accomodate that, the `gWidgets` constructors – except for top-level windows and dialogs – 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 do not illustrate this style here.

[1]This is more like GTK+, and not Qt, where layout managers control where the components are displayed.

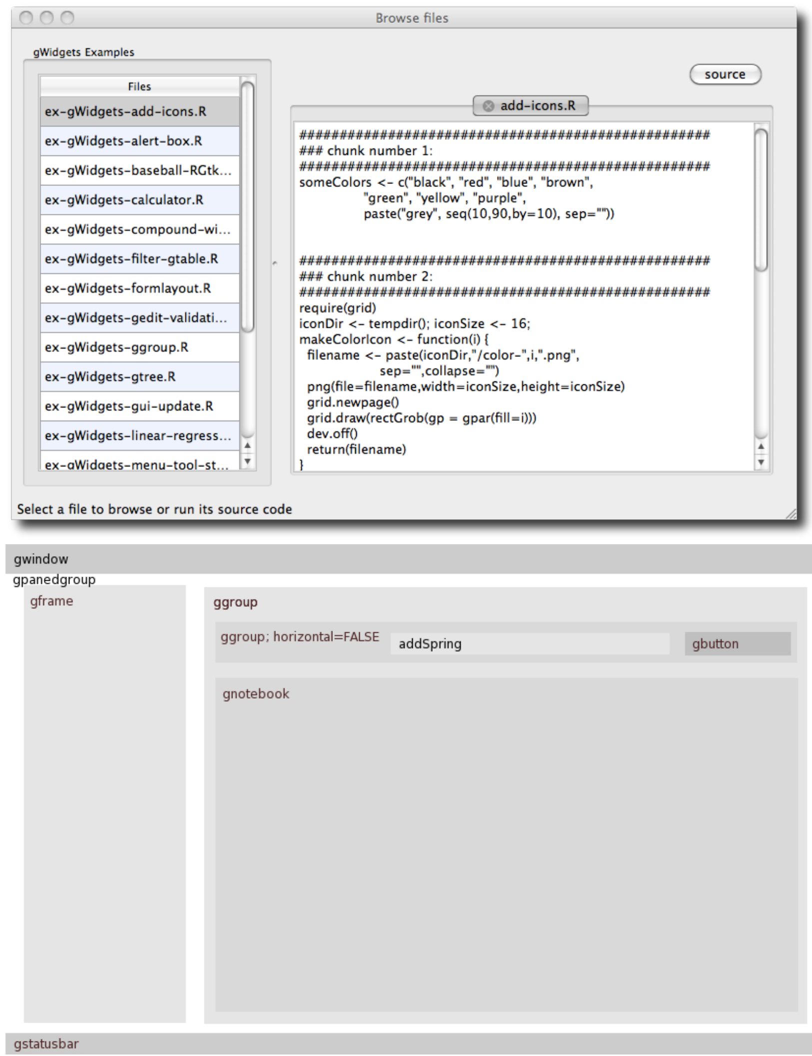

Figure 2.1: The example browser for gWidgets showing different layout components. The lower image shows the different containers used.

2.1 Top-level windows

The `gwindow` constructor creates top-level windows. The main window property is the title which is typically displayed in the window's title bar. This can be set during construction via the `title` argument or accessed later through the object's `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, this child is a container, such as a box container, allowing for multiple children.

The optional `visible` argument, used above with its default value `TRUE`[2], 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.[3]

For example, a typical usage follows this pattern:

```
w <- gwindow("Title", visible=FALSE)
## perform layout here ...
visible(w) <- TRUE
```

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 at this point. The `parent` argument can also be another `gwindow` instance. In this case, the new window will be positioned over the specified window

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

[3] For `gWidgetstcltk` the `update` method will initiate this recomputation. This may be necessary to get the window to size properly.

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,
                  width=200, height=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 default event is the close event. For example, the following will popup any error messages through a in a call to `galert` until the window is closed:

```
oldOptions <- options(error = function() {
  if(msg <- geterrmessage() != "")
    galert(msg, parent=win)
  invisible(msg)
})
#
win <- gwindow( "Popup errors", visible=FALSE,
                handler = function(h, ...) {
                  ## restore old options when gui is closed
                  options(oldOptions)
                })
```

To illustrate, we add a button to initiate an error:

```
btn <- gbutton("Click for error",  cont = win,
                handler = function(h, ...) {
                  stop("This is an error")
                })
```

Clicking the button will signal an error and the error handler will display an alert popup. (This last part fails under `tcltk` due to that packages handling of errors in callbacks.)

The `handler` argument is called just before the window is destroyed, but cannot prevent that from happening. The `addHandlerUnrealize` method can be used to call a handler between the initial click of the close icon and the subsequent destroy event of the window. This handler must return a logical value: if `TRUE` the window will not be destroyed, if `FALSE` the window will be. For example:

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

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

In most GUIs, the use of menubars, toolbars and status bars 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 widgets later in Section 3.7.

A modal window

The gbasicdialog constructor allows one to place an arbitrary widget within a modal window. It also adds OK and Cancel buttons, unless the argument do.buttons is specified as FALSE. 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 can be specified at construction. This 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 specified. The dialog is closed through clicking one of its buttons, through a window manager event, or programmatically through its dispose method.

In Example 3.6 we define a GUI to assist with the task of collapsing factor levels. This wrapper function is used:

```
collapseFactor <- function(f, parent=NULL) {
  out <- character()
  w <- gbasicdialog("Collapse factor levels", parent=parent,
                    handler=function(h,...) {
```

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

```
                      new_f <- relf$get_value()
                      out <<- factor(new_f)
                    })
  g <- ggroup(cont=w)
  relf <- CollapseFactor$new(f, cont=g)
  visible(w, set=TRUE)
  out
}
```

By wrapping the `gbasicdialog` call within a function, we can return the factor, not just a logical, so the above can be used as

```
mtcars$am <- collapseFactor(mtcars$am)
```

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)
```

```
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.[4] 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 where the child widgets are placed and how they respond when their parent window is resized. These are passed through the `...` argument of the constructor. Figure 2.2 shows some differences in how these argument are implemented.[5]

expand The underlying layout algorithms have a means to allocate space to child widgets when the parent container expands to provide more space then requested by the children. Those widgets which have `expand=TRUE` specified should get the excess space shared amongst them. (This isn't the case in `gWidgetsQt`, where a `fill` value needs to be specified as well.)

fill, anchor When a child widget is placed into its allocated space, the space is generally large enough to accommodate the child. If there is additional space, it can be desirable that that the widget grow to fill the available space. The `fill` argument, taking a value of `x`, `y` or `both` (also `TRUE`) indicates how the widget should fill any additional allocation (only when `expand=TRUE`).[6]

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.

[4]In this text, the `add` method is typically called from the constructor, but there are two cases where one calls it directly. The first is if one wishes to integrate a widget from the underlying graphical toolkit into a `gWidgets` GUI. An example where the `tkrplot` package is embedded in a GUI is given in Section 4.1. The second case, is when a widget is removed from a GUI through `delete`. In most cases it may be added back in with `add`.

[5]These arguments are not implemented consistently across toolkits, as the underlying toolkit may prevent it. For example, for `RGtk2` the child widgets always fill in the direction opposite of how they are added (horizontal widgets always fill top to bottom), where as for `tcltk` widgets will fill only if the `expand` argument is `TRUE`.

[6]For GTK+, filling always occurs orthogonally to the direction of packing. This is why the top and bottom buttons (when `expand=FALSE`) in Figure 2.2 for `gWidgetsRGtk2` stretch across the container. To avoid this filling, pack the button in a horizontal `ggroup` container.

Figure 2.2: The `expand`, `fill`, and `anchor` arguments are implemented slightly differently in the different packages. (`gWidgetsRGtk2` on left, `gWidgetstctlk` in middle and `gWidgetsQt` on right.). For GTK+ child components packed in a box container always fill in the direction opposite the packing, in this case the "x" direction. As such, the `anchor` directive has no effect. For `tcltk` a widget only fills if `expand=TRUE` is given. For `gWidgetsQt` expansion and fill are linked together.

Deleting components 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 (with `add`), but this isn't part of the API. In the case where you wish to hide a child temporarily, its `visible<-` method may usually be used, although some widgets give this method a different meaning.[7]

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. We used a spring before the "source" button for the GUI in Figure 2.1 to push it to the right.

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)
```

[7] In `gWidgetstcltk` the use of `visible<-` to hide a component is not supported.

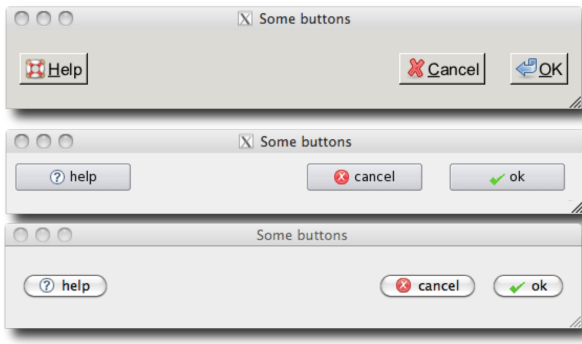

Figure 2.3: Button layout for `RGtk2` (top), `tcltk` (middle) and `qtbase` (bottom). Although the same code is used for each, the different styling yields varying sizes.

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

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 provide the same interface as `ggroup`. Much of the previous discussion applies.

Framed containers are used to visually link the child elements using a border and label. The `gframe` constructor produces them. In Figure 2.1 the table to select the file 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.[8] This class overrides the `visible<-` method 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)
xgrp <- gexpandgroup("Summary", cont=w)
l <- glabel(out, cont=xgrp)
visible(xgrp) <- TRUE                    # display summary
visible(w) <- TRUE
```

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)
lyt <- glayout(cont=w, spacing=5)
right <- c(1,0); left <- c(-1,0)
lyt[1,1, anchor=right] <- "name"
lyt[1,2, anchor=left ] <- gedit("", cont=lyt)
#
lyt[2,1, anchor=right] <- "rank"
lyt[2,2, anchor=left ] <- gedit("", cont=lyt)
#
lyt[3,1, anchor=right] <- "serial number"
```

[8]How each toolkit resizes when the widget collapse varies, so using this container can cause layout issues if cross-toolkit portability is an issue.

```
lyt[3,2, anchor=left ] <- gedit("", cont=lyt)
visible(w) <- TRUE
```

When adding a child, in addition to being on the left hand side of the `[<-` call, the `glayout` container should be specified as the widget's parent `container`.[9] 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.

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 contiguous vector of indices indicating so.

The `[` method may be used to return the children. This method returns a single item, a list of items or a matrix of items. To return the main properties of the widgets in the above example can be done through:

```
sapply(lyt[,2], svalue)
```

```
[1] "" "" ""
```

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 between them. Figure 2.1 uses such a container to separate the file selection controls from those for file display. 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 passed as the parent during the construction of each of the two child widgets. These might 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 accepts the two children through the arguments `widget1` and `widget2`.)

[9]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.

The main property of this container is the sash position, a value in [0,1]. This may be configured programmatically through the `svalue<-` method. 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 code 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 constructor has a few arguments, not all supported by each toolkit. The argument `tab.pos` is used to specify the location of the tabs using a value of 1 through 4 with 1 being the bottom, 2 the left side, 3 the top and 4 the right side, 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.

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 widget is constructed, this argument is usually passed 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 of child widgets, named according to the tab labels. As such, the `[` method returns the child components (by index), the `names` method refers to the tab names, and the `length` method returns the number of pages held by the notebook.

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:

```
addAPage <- function(fname) {
  f <- system.file(fname, package="ProgGUIinR")
  gtext(readLines(f), cont = nb, label=fname)
}
addAPage("DESCRIPTION")
```

For pages holding more than one widget, a container is used:

```
hg <- glayout(cont=nb, horizontal=FALSE, label="Help")
hg[1,1] <- gimage("help", dir="stock", cont=hg)
hg[1,2] <- glabel(paste("To add a page:",
            "Click on a file in the left pane, and its contents",
            "are displayed in a notebook page.", sep="\n"),
            cont=hg)
```

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. We defer discussion of the R-specific widgets to the next chapter.

3.1 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 3.2) an icon will appear. (The ok button below, but not the parButton 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 3.7). The default handler is the click handler which can be specified at construction, or afterward through addHandlerClicked.

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)
  lyt[1,1, align=c(-1,0)] <- "mfrow: c(nr,nc)"
  lyt[2,1] <- (nr <- gedit(1, cont=lyt))
```

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, possibly 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
`gfilebrowse`	A widget to select a file or directory
`gcalendar`	A widget to select a date
`gaction`	A reusable definition of an action
`gmenubar`	Add a menubar on a top-level window
`gtoolbar`	Add a toolbar to a top-level window
`gstatusbar`	Add 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[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.

3.2 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 3.3).[1]

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(lyt[,1], function(i) {
  font(i) <- 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`. This widget is intended simply for displaying HTML-formatted pages. There are no methods to handle the clicking of links, etc.

Status bars

In `gWidgets`, status bars 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

[1] For some of the underlying toolkits, setting the argument `markup` to `TRUE` allows a native markup language to be used (GTK+ had PANGO, Qt has rich text).

label's text. This may be specified at construction with the argument `text`. Subsequent changes are made through the `svalue<-` method.

Icons and images

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 text 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.[2] 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.[3]

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 `addStockIcons`. 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 (Figure 3.1) to select a color from, as an alternative to a more complicated color chooser dialog.[4]

We begin by defining 16 arbitrary colors.

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

[3]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"`.

[4]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.

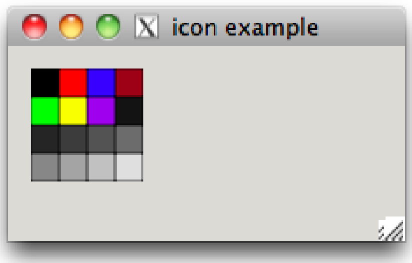

Figure 3.1: A table of stock icons created on the fly

```
someColors <- c("black", "red", "blue", "brown",
                "green", "yellow", "purple",
                paste("grey", seq.int(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 <- file.path(iconDir,
                        sprintf("color-%s.png", 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)
  }
}
```

SVG graphics 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.

3.3 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 is simple to use, but provides much less flexibility than is possible with the toolkit widgets.

Single-line, editable text

The gedit constructor produces a widget to display a single line of editable text. The main property is the text which can be set initially through the text argument. If 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.

The visible method is overridden to mask out the letters in the field, not hide the component. This allows one to use the widget to collect passwords.

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).[5]

Example 3.2: Validation
GUIs for R may differ a bit from many GUIs users typically interact 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.[6]

```
require(evaluate)
```

[5]There are differences in what keys are returned. Currently, only the letter keys are consistently given. In particular, no modifier keys or other keys are returned.

[6]The basic way to evaluate an R expression given as a string is to use the combination of `eval` and `parse`, as in `eval(parse(text=string))`. The resulting output can usually be captured with the `capture.output` function. However, there can be errors: parse errors or otherwise. A few packages provide functions to assist with this task, notably the `evaluate` function in the same-named `evaluate` package, and the `Parse` and `captureAll` functions in the `svMisc` package. We use both in this part of the text.

```
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.

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

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 may be 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 3.3. 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.

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

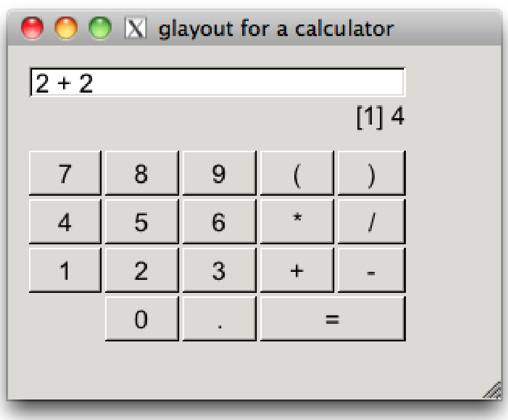

Figure 3.2: Dumbing down R with gWidgets to make a calculator interface

Example 3.3: A calculator

This example shows how one might use the widgets just discussed to make a GUI (Figure 3.2) 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, "+", "-"))
#
```

```
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))
#
bList <- list()
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
}
sapply(bList, addHandlerChanged, 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.

```
require(evaluate)
addHandlerClicked(eqButton, handler = function(h,...) {
  curExpr <- svalue(eqnArea)
  out <- try(evaluate:::evaluate(curExpr), silent=TRUE)
  if(inherits(out, "try-error")) {
    galert("Parse error", parent=eqButton)
  } else if(is(out[[2]], "error")) {
    msg <- sprintf("Error: %s", out[[2]]$message)
    galert(msg, parent=eqButton)
  } else {
    svalue(outputArea) <- out[[2]]
    svalue(eqnArea) <- ""            # restart
```

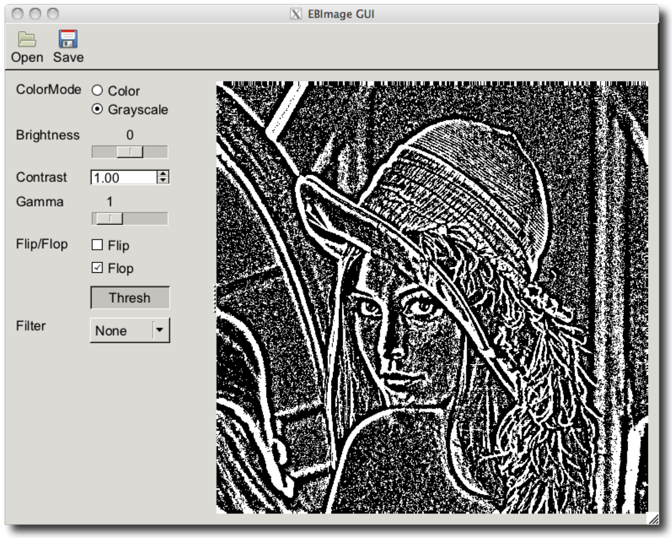

Figure 3.3: A simple GUI for the `EBImage` package illustrating many selection widgets

```
  }
})
```

3.4 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 3.3 shows a simple GUI for the `EBImage` package allowing a user to adjust a few of the image properties using various selection widget. Although it is unlikely one would use `R` for such a task, as opposed to Gimp say, we use this example, as the mapping between controls and actions should be familiar.

In `gWidgets` the abstract view for selection widgets 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 such 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, uncommitted state as possible in some toolkits). By default the label will be drawn aside a box which the user can check. If the argument `use.togglebutton` is `TRUE`, a toggle button – which appears depressed when `TRUE` – is used instead.

In Figure 3.3 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 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 3.3 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 may be respecified with the `[<-` method.

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

A group of checkboxes

A 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 than one 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, as there is a scrollbar provided. (This provides a similar functionality as using the `gtable` widget with multiple selection.)

In Figure 3.3 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 selected indices instead. These are 0-length if no selection is made. 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`.[7] 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 3.3 could be coded with:

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

This example shows how to create a combo 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.

[7] Some make a distinction between drop down lists and combo boxes, the latter allowing editing. We don't here, although we note that the constructor `gdroplist` is an alias for `gcombobox`.

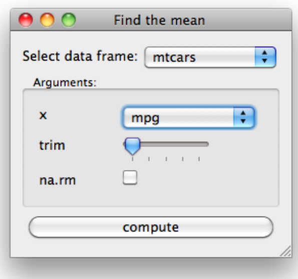

Figure 3.4: GUI used to collect arguments for a call to `mean.default`

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.

Example 3.4: Updating combo boxes
A common feature in many GUIs is to have one combo box update another once a selection is made. The following employs this to create a simple GUI for collecting the arguments for computing the mean of a numeric variable (Figure 3.4).

We make use of the functions from the `ProgGUIinR` package in the following to return character vectors of data frame names and numeric variables.

```
availDfs <- function() {
  c("", ".GlobalEnv", ProgGUIinR:::avail_dfs(.GlobalEnv))
}
```

```
getNumeric <- function(where) {
  val <- get(where, envir=.GlobalEnv)
  ProgGUIinR:::find_vars(val, is.numeric)
}
```

Our layout uses nested groups and a `glayout` container.

```
w <- gwindow("Find the mean", visible=FALSE)
g <- ggroup(cont=w, horizontal=FALSE)
g1 <- ggroup(cont=g)
glabel("Select data frame:", cont=g1)
dfC <- gcombobox(availDfs(), cont=g1)
##
f <- gframe("Arguments:", cont=g, horizontal=FALSE)
enabled(f) <- FALSE
```

```
lyt <- glayout(cont=f, expand=TRUE)
l <- list()                                    # store widgets
##
lyt[1,1] <- "x"
lyt[1,2] <- (l$x <- gcombobox("           ", cont=lyt))
##
lyt[2,1] <- "trim"
lyt[2,2] <-
  (l$trim <- gslider(from=0, to=0.5, by=0.01, cont=lyt))
##
lyt[3,1] <- "na.rm"
lyt[3,2] <-
  (l$na.rm <- gcheckbox("", checked=TRUE, cont=lyt))
g2 <- ggroup(cont=g)
compute <- gbutton("compute", cont=g2)
```

We stored the primary widgets in a list with names matching the arguments to our function, `mean.default`. As well, the initial argument to the `x` combo box pads out the width under some toolkits.

Here is how we update the `x` combo box, when the data frame combo box is changed. If there is a value, we enable our widgets and then populate the secondary combo box with the names of the numeric variables.

```
addHandlerChanged(dfC, handler=function(h,...) {
  val <- svalue(h$obj)
  enabled(f) <- val !=""
  enabled(compute) <- val != ""
  if(val != "")
    l$x[] <- getNumeric(val)
  svalue(l$x, index=TRUE) <- 0
})
```

As we stored the widgets in an appropriately named list, we can conveniently use `do.call` below to write the callback for the `compute` button in just a few lines. The only trick is to replace the variable name with its actual value.

```
addHandlerChanged(compute, handler=function(h,...) {
  out <- lapply(l, svalue)
  out$x <- get(out$x, get(svalue(dfC), envir=.GlobalEnv))
  print(do.call(mean.default, out))
})
```

A slider control

The `gslider` constructor creates a scale widget 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`. However, if `from` is a vector, then it is assumed it presents an orderable sequence of values to select 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 3.3 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.

In Figure 3.3 the `gWidgetsRGtk2` package is used. That toolkit shows a tooltip with the current value, for others the slider implementation does not show the value. One can add a label to show this (or combine the slider with a spin button). Adding a label follows this pattern:

```
w <- gwindow("Add a label to the slider", visible=FALSE)
g <- ggroup(cont=w, expand=TRUE)
sl <- gslider(from=0, to=100, by=1, cont=g, expand=TRUE)
l <- glabel(sprintf("%3d", svalue(sl)), cont=g)
font(l) <- c(family="monospace")
addHandlerChanged(sl, function(h,...) {
  svalue(h$action) <- sprintf("%3d", svalue(h$obj))
  }, action=l)
visible(w) <- TRUE
```

(Using `sprintf` and `monospace` ensures the label takes a fixed amount of space.)

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 3.3 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 3.3 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. The filter argument is toolkit dependent. 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.

Example 3.5: Selecting from a file system
We return to the File selection GUI used as an example in Chapter 1. Our goal here is to add in more features to have advanced searching. Imagine we have a function file_search which in addition to arguments for a

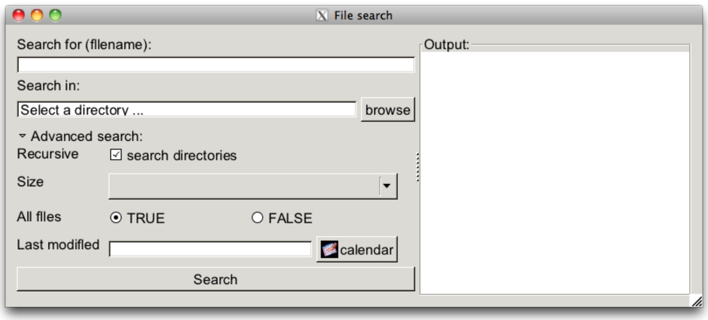

Figure 3.5: File search dialog showing advanced search features disclosed

pattern and directory has arguments `modified` to pass a date string, `size` to pass a descriptive `small`, `medium` or `large` and an argument `visible` to indicate if all files (including dot files) should be looked at.

We want to update our GUI to collect values for these. Since these are advanced options, we want the user to have access only on request. We use `gexpandgroup` to provide this. Here we define the additional code for the layout:

```
advSearch <- gexpandgroup("Advanced search:", cont=f)
visible(advSearch) <- FALSE
tbl <- glayout(cont=advSearch)
tbl[1,1] <- "Recursive"
tbl[1,2] <- (advRec <-
   gcheckbox("search directories", checked=TRUE, cont=tbl))
tbl[2,1] <- "Size"
tbl[2,2] <- (advSize <-
   gcombobox(c("", "small", "medium", "large"),  cont=tbl))
tbl[3,1] <- "All files"
tbl[3,2] <- (advVisible <-
   gradio(c(TRUE, FALSE), horizontal=TRUE, cont=tbl))
tbl[4,1] <- "Last modified"
tbl[4,2] <- (advModified <-
             gcalendar("", format="%Y-%m-%d", cont=tbl))
```

As can be seen (Figure 3.5), we use a grid layout and a mix of the controls offered by `gWidgets`.

We modify our button handler so that it uses these values, if specified.

```
addHandlerChanged(searchBtn, handler=function(h,...) {
  pattern <- glob2rx(svalue(txtPattern))
  start_dir <- svalue(startDir)
  modified <- NULL
  size <- NULL
```

```
  ## from advanced
  subfolders <- svalue(advRec)
  visible <- svalue(advVisible)
  if((tmp <- svalue(advSize)) != "") size <- tmp
  if(!is.na(tmp <- svalue(advModified))) modified <- tmp

  ## function call
  fnames <- file_search(pattern, start_dir, subfolders,
                        modified=modified,
                        size=size, visible=visible)
  dispose(searchResults)              # clear
  if(length(fnames))
    svalue(searchResults) <- fnames
  else
    galert("No matching files found", parent=w)
})
```

3.5 Display of tabular data

The `gtable` constructor[8] produces a widget that displays data in a tabular form from which the user can select one (or more) rows. The performance under `gWidgetsRGtk2` and `gWidgetsQt` is much faster and able to handle larger data stores than under `gWidgetstcltk`, as there is no enhanced data frame model in Tcl/Tk. At a minimum, all perform well on moderate-sized data sets (10 or so columns and fewer than 500 rows).[9]

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 form, with column headers derived from the `names` attribute of the data frame (but no row names). The `items` argument can be a 0-length data frame, but the column classes must match the eventual data to be used.

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

```
w <- gwindow("gtable example")
dfs <- gtable(ProgGUIinR:::avail_dfs(), cont=w)
```

Often the table widget is added to a box container with the argument `expand=TRUE`. Otherwise, the size of the widget should be specified through

[8] 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.

[9] For `gWidgetsRGtk2`, the `gdfedit` widget can show very large tables taking advantage of the underlying `RGtk2Extras` package.

size<-. This size 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.)

Icons 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 by case (row) – not by observation (column) – from this widget. The actual value returned by a selection is controlled by the constructor's argument chosencol, which specifies which column's value will be returned for the given index, as the user can only specify the row. The multiple argument can be specified to allow the user to 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 or reduce the number of columns displayed, so when updating a column do not assume some underlying coercion, as is done with R's data frames. (This is why the initial items, even if a 0-length data frame, need to be of the correct class.) To replace the data store, the [<- can be used, as with 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[] <- ProgGUIinR:::avail_dfs()
}
```

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

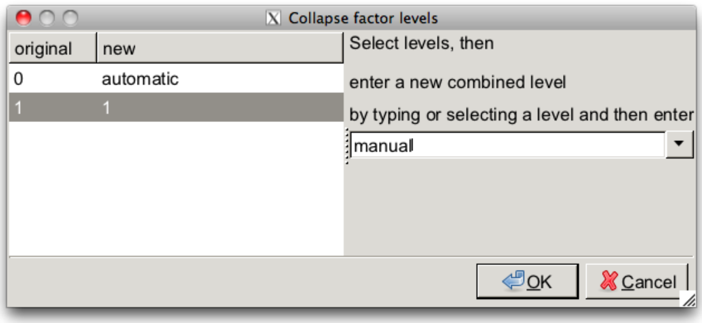

Figure 3.6: A GUI to facilitate the recoding or collapsing of a factor's levels. For this, one selects the desired levels to rename or collapse, then enters a new label on the right. Activating the combo box will update the "new" column on the left.

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)
  print(summary(get(val, envir=.GlobalEnv))) # some action

})
```

Example 3.6: Collapsing factors
A somewhat tedious task in R is the recoding or collapsing of factor levels. This example provides a GUI to facilitate this. In Section 2.1 we provided a function to wrap this GUI within a modal dialog. Here we just setup the GUI.

We will use a reference class, as it allows us to couple together the main method and the widgets without needing to worry about scoping issues. For formatting purposes, we define the methods individually, then piece together.

Our initialization call simple stores the values and then passes on the call to make the GUI.

```
initialize <- function(f, cont=gwindow()) {
  old <<- as.character(f)
  make_gui(cont)
  callSuper()
}
```

This make_gui function does the hard work. (Figure 3.6 shows a screenshot.) We have just two widgets, placed in a paned group. The left one is a table that displays two columns: the old values and the collapsed or recoded values. The widget on the right is a combo box for entering a new factor level or selecting an existing level. The handler on the combo box updates the second column of the table to reflect the new values. We block any handler calls to avoid a loop when we set the index back to 0.

```
make_gui <- function(cont) {
  g <- gpanedgroup(cont=cont)
  levs <- sort(unique(as.character(old)))
  d <- data.frame(original=levs,
                  new=levs, stringsAsFactors=FALSE)
  #
  widget <<- tbl <- gtable(d, cont=g,  multiple=TRUE)
  size(tbl) <- c(300, 200)
  #
  g1 <- ggroup(cont=g, horizontal=FALSE)
  instructions <- gettext("Select levels, then\n
enter a new combined level\n
by typing or selecting a level and then enter")
  #
  glabel(instructions, cont=g1)
  cb <- gcombobox(levs, selected=0, editable=TRUE, cont=g1)
  enabled(cb) <- FALSE
  #
  addHandlerClicked(widget, function(h,...) {
    ind <- svalue(widget, index=TRUE)
    enabled(cb) <- (length(ind) > 0)
  })

  addHandlerChanged(cb, handler=function(h,...) {
    ind <- svalue(tbl, index=TRUE)
    if(length(ind) == 0)
      return()
    #
    tbl[ind,2] <- svalue(cb)
    svalue(tbl, index=TRUE) <- 0
    blockHandler(cb)
    cb[] <- sort(unique(tbl[,2]))
    svalue(cb, index=TRUE) <- 0
    unblockHandler(cb)
  })
}
```

This method returns the newly recoded factor. The tediousness of the task is in the specification of the new levels, not necessarily this.

```
get_value <- function() {
  "Return factor with new levels"
  old_levels <- widget[,1]
  new_levels <- widget[,2]
  new <- old
  for(i in seq_along(old_levels)) # one pass
    new[new == old_levels[i]] <- new_levels[i]
  factor(new)
}
```

Finally, we stitch the above together into a reference class.

```
CollapseFactor <- setRefClass("CollapseFactor",
                              fields=list(
                                old="ANY",
                                widget="ANY"
                                ),
                             methods=list(
                               initialize=initialize,
                               make_gui = make_gui,
                               get_value=get_value
                             ))
```

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 3.8 for an application.

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

```
require(MASS)
```

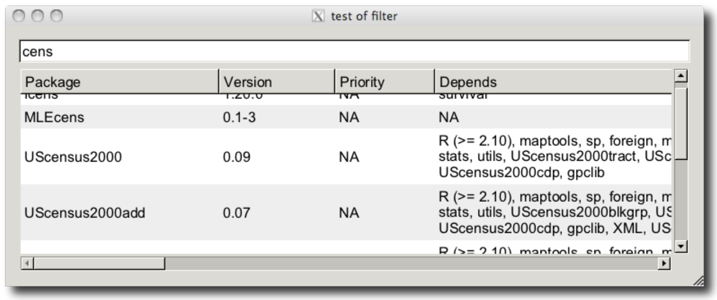

Figure 3.7: Example of using a filter to narrow the display of tabular data

```
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 3.8: More complex filtering

Even with moderate-sized data sets, the number of rows can be quite large, in which case it is inconvenient to use a table for selection unless some means of searching or filtering the data is used. This example displays 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 3.7). 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.

```
ap <- 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(ap, cont=g, filter.FUN="manual", expand=TRUE)
```

The `filter.FUN` value of `"manual"` allows us to filter by specifying a logical vector.

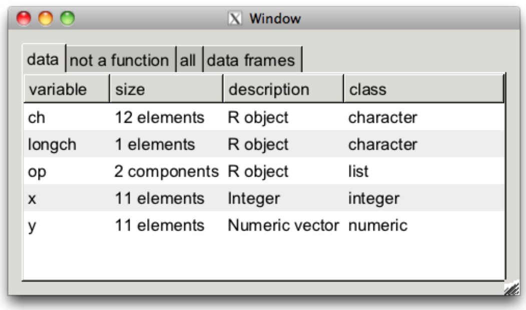

Figure 3.8: A notebook showing various views of the objects in the global workspace. The example uses the Observer pattern to keep the views synchronized.

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 3.9: Using the "observer pattern" to write a workspace view
This example takes the long way to make a workspace browser. (The short way is to use `gvarbrowser`.) The goal is to produce a GUI that will allow the user to view the objects in their current workspace. We would like this view to bedynamic – when the workspace changes we would like the view to update. Furthermore, we may want to have different views, such as one for functions and one for data sets. These should all be coordinated.

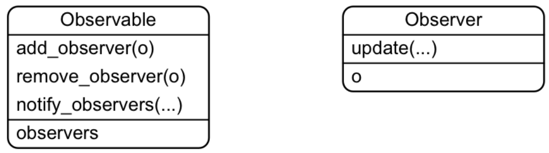

Figure 3.9: Observable and Observer classes and their basic methods. An observable object may have many observers which are notified through their update method when a change is made.

This pattern where a central, dynamic source of data is to used and shared amongst many different pieces of a GUI is a common one. To address the complexity that arises as the components of a GUI get more intertwined, standard design patterns have been employed. For this task, the *Observer Pattern* is often used. This pattern is defined in [1] to describe a one-to-many relationship between a set of objects where when the state of one object changes, all of its dependents are notified.

Figure 3.9 shows a class diagram of the two different types of objects involved:

Observables The objects which notify observers when a change is made. The basic methods are to add and remove an observer; and to notify all observers when a change is made. In our example, we will create a workspace model which will notify the various observers (views) when `R`'s global workspace has changes.

Observers The objects which listen for changes to the observable object. Observers are registered with the observable and are notified of changes by a call to the observer's `update` method. In our example, the different views of the workspace are observers.

The package `objectSignals` provides a comprehensive implementation of this pattern which we use below. We use its `signalingField` function to create an "observable" and its `connect` method to add an observer.

The data in our workspace model keeps track of the objects in the workspace by name and records a digest of each variable. The digest allows us to compare if objects have been updated, not just renamed. As notifying views can be potentially expensive, we will only notify on a change.

```
require(objectSignals)
WSModel <- setRefClass("WSModel",
```

[1] Eric T Freeman; Elisabeth Robson; Bert Bates; Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, Inc, October 25, 2004.

```
               fields=c(
                 signalingField("ws_objects","character"),
                 ws_objects_digests="character"
                 ))
```

For the task at hand, we don't really have a `set` method, but rather we define a `refresh` method to synchronize the workspace with our model object. When the property `ws_objects` is set, the `objectSignals` package takes care of notifying any registered observers. This model needs to track changes in the underlying workspace. This can be done calling the `refresh` method at periodic intervals, through a *taskCallback*, or by user request. In the definitions below, we call a helper function to list the objects in the global environment and produce a digest of each.

```
WSModel$methods(
       .get_objects_digests = function() {
         "Helper function to return list with names, digests"
         items <- ls(envir=.GlobalEnv)
         objects <- mget(items, .GlobalEnv)
         trim <- !sapply(objects, is, class2="refClass")
         list(items[trim],
              sapply(objects[trim], digest))
       },
       initialize=function() {
         objs <- .get_objects_digests() # call helper
         initFields(ws_objects=objs[[1]],
                    ws_objects_digests=objs[[2]])
         .self
       },
       refresh=function() {
         objs <- .get_objects_digests()
         cur_objects <- objs[[1]]
         cur_digests <- objs[[2]]
         ## changes?
         if(length(cur_digests) != ws_objects_digests ||
            length(ws_objects_digests) == 0 ||
            any(cur_digests != ws_objects_digests)) {
           ws_objects <<- cur_objects # signal
           ws_objects_digests <<- cur_digests
         }
       })
```

To simplify the work for our views, our model provides a `get` method that filters its return value to specified classes. This class is specified with a character string, and may include a not operator.

```
WSModel$methods(
        get=function(klass) {
          "klass a string, such as 'numeric' or '!function'"
```

```
          if(missing(klass) || length(klass) == 0)
            return(ws_objects)
          ## if we have klass, more work
          ind <- sapply(mget(ws_objects, .GlobalEnv), function(x) {
            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)
        })
```

Finally, our model defines a convenience method to add an observer using the naming convention of objectSignals.

```
WSModel$methods(
                add_observer=function(FUN, ...) {
                  .self$ws_objectsChanged$connect(FUN, ...)
                })
```

To use this model, we create a base view class adding a new method to set the model. A view has atleast two methods, an update method to refresh the view and one to set the model, so that it can play the part of an observer.

```
WSView <- setRefClass("WSView",
                      methods=list(
                        update=function(model) {
                          "Subclass this"
                        },
                        set_model=function(model) {
                          FUN <- function() .self$update(model)
                          model$add_observer(FUN)
                        }
                        ))
```

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

```
WidgetView <-
  setRefClass("WidgetView",
              contains="WSView",
              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()
                   update(model)
                   .self
                 },
                 init_widget=function(parent, ...) {
                   "Initialize widget"
                 }))
```

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(model, ...) {
            widget[] <<- makeDataFrame(model$get(klass))
          }))
```

This subclass of the widget view class shows the values in the workspace using a table widget. The makeDataFrame function generates the details. We now turn to the task of defining that function.

To generate data on each object, we define some S3 classes. These are more convenient than 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 <- mget(x, envir)
    d <- data.frame(variable=x,
                    size=sapply(l, sizeOf),
                    description=sapply(l, shortDescription),
                    class = sapply(l, function(i) class(i)[1]),
                    stringsAsFactors=FALSE)
  }
  d
}
```

To illustrate the flexibility of this framework, we also define a subclass of `WidgetView` to show just the data frames in a combo box. Selecting a data frame is a common task in R GUIs, and this allows keeps the selection synchronized with the workspace.

```
DfView <-
  setRefClass("DfView",
        contains="WidgetView",
        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(model, ...) {
            widget[] <<- model$get(klass)
          }
          ))
```

We can put these pieces together to make a simple GUI.

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

```
#
view1 <- TableView$new(parent=nb, model=model,
                       label="not a function")
view1$klass <- "!function"
#
view2 <- TableView$new(parent=nb, model=model, label="all")
## a bit contrived here
view3 <- DfView$new(parent=nb, model=model, label="data frames")
#
model$refresh()                              # notifies views
svalue(nb) <- 1
```

3.6 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. To use gtree one describes the tree to be shown dynamically through a function that computes the child components in terms of the path of the parent node. Although a bit more complex, this approach allows trees with many ancestors to be shown, without needing to compute the entire tree at the time of construction.

The offspring argument is assigned a function of two arguments, the path of 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 vector, then that column will be used.

As an illustration, this function produces an offspring function to explore the hierarchical structure of a list. It has the list passed in through the offspring.data argument of the constructor.

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

```
                stringsAsFactors=FALSE)
}
```

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="2", b="3", c=list(a="4")))
w <- gwindow("Tree test")
t <- gtree(offspring, offspring.data=l, 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 (just as is needed for `gtable`). 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 3.10: Using `gtree` to explore a recursive partition

The `party` package implements a recursive partitioning algorithm for tree-based regression and classification models. The package provides an excellent `plot` method for the object, but in this example we demonstrate how the `gtree` widget can be used to display the 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 party 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 3.10)

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

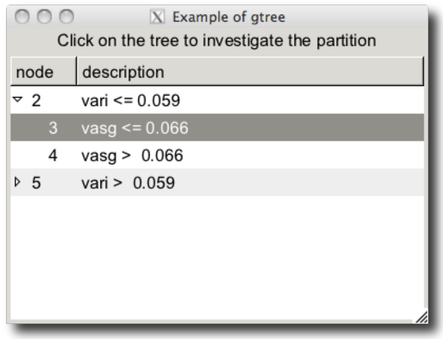

Figure 3.10: GUI to explore return value of a model fit by the `party` package.

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, ])
  }})
```

3.7 Actions, menus and toolbars

Actions are non-graphical objects representing an application command that is executable through one or more widgets. Actions in `gWidgets` are created through the `gaction` constructor. The arguments are `label`, `tooltip`, `icon`, `key.accel`,[10] `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. 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. There is no method to invoke the action.

[10]The key accelerator implementation varies depending on the underlying toolkit.

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")
             },
             parent=w)
b <- gbutton(action=a, cont=w)
## .. to change
enabled(a) <- FALSE                    # can't click now
```

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

Toolbars

Toolbars and menubars are implemented in `gWidgets` using `gaction` items. Both are specified using a named list of action components.

For a toolbar, this list has a simple structure. Each named component either describes a toolbar item or a separator, where the toolbar items are specified by `gaction` instances and separators by `gseparator` instances with no container specified.

For example. Here we first define some actions:

```
stub <- function(h,...) gmessage("called handler", parent=w)
actlist = list(
  new = gaction(label="new", icon="new",
    handler = stub, parent = w),
  open = gaction(label="open", icon="open",
    handler = stub, parent = w),
  save = gaction(label="save", icon="save",
    handler = stub, parent = w),
  save.as = gaction(label="save as...", icon="save as...",
    handler = stub, parent = w),
  quit = gaction(label="quit", icon="quit",
    handler = function(...) dispose(w), parent = w),
  cut = gaction(label="cut", icon="cut",
    handler = stub, parent = w)
  )
```

Then a toolbar list might look like:

```
w <- gwindow("gtoolbar example")
tl <- c(actlist[c("new","save")],
        sep=gseparator(),
```

```
        actlist["quit"])
tb <- gtoolbar(tl, cont=w)
gtext("Lorem ipsum ...", 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.

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 menubars, 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

```
ml <- list(file = list(
             new = actlist$new,
             open = actlist$open,
             save = actlist$save,
             "save as..." = actlist$save.as,
             sep=gseparator(),
             quit = actlist$quit
             ),
           edit = list(
             cut = actlist$cut
             )
           )
```

Figure 3.11 shows this simple GUI using `gWidgetsRGtk2`. Under 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.

Here we show how to disable groups of actions. Suppose, we want to disable the saving and cut actions if there are no characters in the text buffer, then we could use this handler:

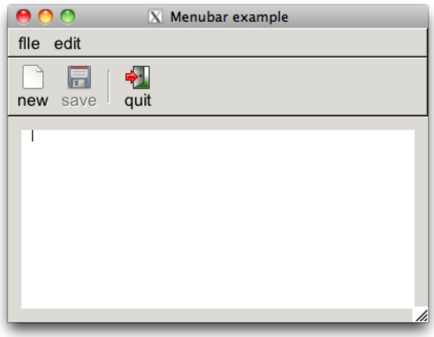

Figure 3.11: Menubar and toolbar decorating a basic text editing widget. The "Save" icon is disabled, as there is no text typed in the buffer.

```
noChanges <- c("save","save.as","cut")
keyhandler <- function(...) {
  for(i in noChanges)
    enabled(actlist[[i]]) <- (nchar(svalue(txt)) > 0)
}
addHandlerKeystroke(txt, handler=keyhandler)
keyhandler()
```

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 3.11: 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 mouse click in a ggraphics instance will make that device the current one. 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.

The default handler for the widget is set by addHandlerClicked. The coordinates of the mouse click, in user coordinates, are passed to the han-

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

Constructor	Description
ggraphics	Embeddable graphics device
ggraphicsnotebook	Notebook for multiple devices
gdf	Data frame editor
gdfnotebook	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

dler 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)
```

The underlying toolkits may pass in more information about the event, such as whether a modifier key was being pressed, but this isn't toolkit independent.

Using tkrplot The tkrplot provides a means to embed graphics in Tk GUIs, but is not a graphics device. As such, there is no ggraphics implementation in gWidgetstcltk. You can embed tkrplot though. The following is a simple modification of the example from the help page for tkrplot:

```
options(guiToolkit="tcltk"); require(tkrplot)
w <- gwindow("How to embed tkrplot", visible=FALSE)
g <- ggroup(cont=w, horizontal=FALSE)
bb<-1
img <-tkrplot(getToolkitWidget(g),
              fun=function() plot(1:20,(1:20)^bb))
add(g, img)
f<-function(...) {
  b <- svalue(sl)
  print(b)
```

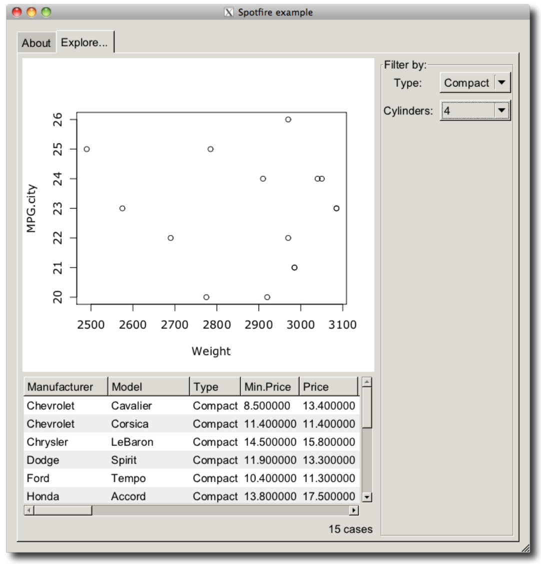

Figure 4.1: A GUI to filter a data frame and display an accompanying graphic.

```
  if (b != bb) {
    bb <<- b
    tkrreplot(img)
  }
}
sl <- gslider(from=0.05, to=2, by=0.05, cont=g,
              handler=f, expand=TRUE)
visible(w) <- TRUE
```

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
```

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.

```
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 by. 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 checkbox group to allow multiple selection, a slider to pick out numeric data, or a text box to specify filtering by a string. The type of data dictates this. In this example it isn't needed, but since the layout is done, we might have code to initialize the controls in the filter. Adding such a call, makes it easy to save the state of the GUI.

We now move on to the task of making the three main components – the display, the table and the filters – interact with each other. We keep this example simple, but note that if we were to extend the example we would likely write using the observer pattern introduced in Example 3.9 as that makes it easy to decouple the components of an interface. As it is we define function calls to a) update the data frame when the filters change and b) update the graphic.

For the first, we need to compute a logical variable indicating which rows are to be displayed. Within the definition of the following function, we use the global variables `l`, `tbl` and `noCases`.

```
updateDataFrame <- function(...) {
  vals <- lapply(l, svalue)
  vals <- vals[vals != ""]
  out <- sapply(names(vals), function(i) x[[i]] == vals[[i]])
  ind <- apply(out, 1, function(x) Filter("&&", x))
  ## update table
  visible(tbl) <- ind
  ## update label
  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. A real application would provide a more compelling plot.

```
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, addHandlerChanged, 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

```
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
```

[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.

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.

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 the name of the currently selected value using the $-notation to refer to child elements. One can call svalue on this string to get the corresponding R 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 time consuming. Some heuristics are employed to do this automatically, if the size of the workspace is modest enough. Otherwise it can be done programmatically.

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 that can be embedded in a GUI.

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 command lines, 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. There are similar facilities in `RGtk2Extras` for `RGtk2` and the `fgui` package can do such a thing for `tcltk`.

Concept index

Class and method index